Язык C++ программирования

специальное издание

Бьерн Страуструп
создатель C++

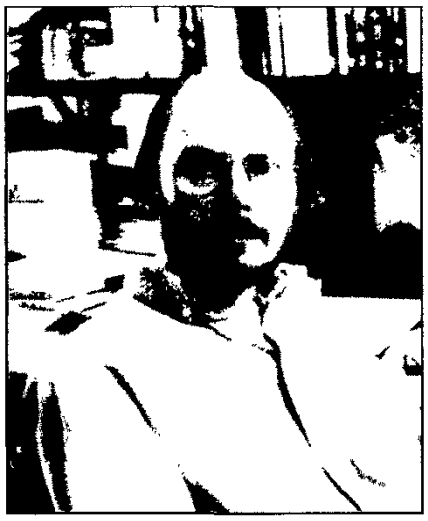
БОЛЕЕ 750 ТЫСЯЧ ПРОГРАММИСТОВ ВО ВСЕМ МИРЕ ПРИОБРЕЛИ ПРЕДЫДУЩИЕ ИЗДАНИЯ ЭТОЙ КНИГИ

Перед вами – специальное издание самой читаемой и содержащей наиболее достоверные сведения книги по C++. В него были добавлены два новых приложения: *"Локализация"* и *"Безопасность исключений и стандартная библиотека"* (оба доступны на сервере **http://www.research.att.com/-bs**). **В результате монография содержит полное описание** языка C++, его стандартной библиотеки (STL) и ключевых методов разработки программ. Основанная на стандарте ANSI/ISO, книга является источником самого последнего и **полного описания** всех возможностей языка C++, **включая компоненты** стандартной **библиотеки**, в том числе:

- абстрактные классы в качестве интерфейсов;
- иерархию классов при объектно-ориентированном программировании;
- шаблоны как основу безопасного относительно типов обобщенного программирования;
- обработку исключений, возникающих в результате типичных ошибок;
- использование пространств имен (namespaces) для достижения модульности больших проектов;
- определение типа на этапе исполнения (RTTI) для слабо связанных систем;
- подмножество C языка C++ для совместимости с C и работы на системном уровне;
- стандартные контейнеры и алгоритмы;
- стандартные строки, потоки ввода/вывода и числовые данные;
- совместимость с C, локализацию (интернационализацию) и безопасность при обработке исключениий.

В данном издании Бьерн Страуструп излагает C++ в форме, еще более доступной для начинающих. В то же время в него включены такие сведения и методики, которые могут оказаться полезными даже для экспертов по C++.

Web-поддержка книги осуществляется по адресу: http://www.aw.com/cseng/



Бьерн Страуструп (***Bjarne Stroustrup***) непосредственно разработал и первым осуществил реализацию языка программирования C++. Он – автор книг: ***"Язык программирования C++"*** (***The C++ Programming Language***, первое издание – 1985, второе – 1991, третье – 1997), ***"Справочное руководство по языку C++ с комментариями"*** (***The Annotated C++ Reference Manual***) и ***"Проектирование и эволюция C++"*** (***The Design and Evolution of C++***). Закончил университет в Аархусе (Дания) и Кэмбриджский университет (Англия). В настоящее время д-р Страуструп возглавляет Отдел исследований в области крупномасштабного программирования компании AT&T Labs и является членом совета AT&T, AT&T Bell Laboratories и ACM. Область его научных интересов – распределенные системы, операционные системы, моделирование, проектирование и программирование. Он является редактором серии **C++ In-Depth**, выпускаемой издательством Addison-Wesley.

Язык Программирования C++

специальное издание

THE C++ PROGRAMMING LANGUAGE

SPECIAL EDITION

Bjarne Stroustrup
AT&T Labs
Florham Park, New Jersey

Addison-Wesley
An Imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Краткое оглавление

Оглавление

Предисловие автора к третьему русскому изданию

В августе 1998 года был ратифицирован стандарт языка C++ (ISO/IEC 14882 «Standard for the C++ Programming Language», результат голосования национальных комитетов по стандартизации: 22–0). Это событие знаменует новую эру стабильности и процветания C++, а также связанных с языком инструментальных средств и приемов программирования.

Лично для меня главным является то, что стандартный C++ лучше чем любая его предыдущая версия соответствует моему замыслу C++. Стандартный C++ и его стандартная библиотека позволяют мне писать лучшие, более элегантные и более эффективные программы, чем те, что я мог писать в прошлом.

Чтобы добиться этого, я и сотни других людей долго и интенсивно работали в комитетах по стандартизации ANSI и ISO. Целью этих усилий было описать язык и библиотеку, которые будут исправно служить всем пользователям языка, не предоставляя каких-либо преимуществ одной группе пользователей, компании или стране перед остальными. Процесс стандартизации был открыт, справедлив, нацелен на качество и согласие.

Усилия по стандартизации были инициированы Дмитрием Ленковым (Dmitry Lenkov) из Hewlett-Packard, который работал в качестве первого председателя комитета. Во время завершения работ над стандартом председателем был Стив Кламаг (Steve Clamage) из Sun. Джонатан Шапиро (Jonathan Shopiro) и Эндрю Кениг (Andrew Koenig) (оба из AT&T) были редакторами, которые — с помощью многих членов комитета — подготовили текст стандарта, основанный на моем исходном справочном руководстве по C++.

Открытый и демократичный процесс стандартизации потенциально таит в себе одну опасность: результат может оказаться «комитетской разработкой». В случае с C++ этого по большей части удалось избежать. Во-первых, я состоял председателем рабочей группы по расширению языка. В этом качестве я оценивал все основные предложения и составлял окончательные варианты тех из них, которые я, рабочая группа и комитет считали стоящими и в тоже время реализуемыми. Таким образом, комитет в основном обсуждал направляемые ему относительно завершенные проекты, а не занимался разработкой таковых. Во-вторых, главная из новых составляющих стандартной библиотеки — стандартная библиотека шаблонов STL, предоставляющая общую, эффективную, типобезопасную и расширяемую среду контейнеров, итераторов и алгоритмов — была, в основном, результатом работы одного человека, Александра Степанова (Alexander Stepanov) из Hewlett-Packard.

Важно, что стандарт C++ — это не просто бумажка. Он уже встроен в наиболее свежие реализации C++. Большинство основных реализаций поддерживают стандарт (с очень немногочисленными исключениями), причем все обещают полное соответствие в течение ближайших месяцев. Чтобы стимулировать честность производителей, две компании предлагают пакеты, проверяющие «стандартность» реализации C++.

Итак, код, который я пишу сейчас, использует, если это необходимо, большинство возможностей, предлагаемых стандартным C++ и описанных в данном третьем издании «Языка программирования C++».

Улучшения в языке C++ и расширение стандартной библиотеки заметно изменили то, как я пишу код. Мои теперешние программы короче, яснее и эффективнее прежних, что является прямым результатом лучшей, более ясной и систематичной поддержки абстракций в стандартном C++. Улучшенная поддержка таких средств, как шаблоны и исключения, сокращают потребность в более низкоуровневых и запутанных возможностях языка. Поэтому я считаю, что стандартный C++ проще в использовании, чем его предшествующие варианты, где мне приходилось имитировать средства, которые теперь поддерживаются непосредственно.

Основываясь на приведенных наблюдениях, я решил, что необходимо полностью переписать второе издание «Языка программирования C++». Никак иначе нельзя было должным образом отразить новые возможности, предлагаемые языком, и поддерживаемые ими приемы программирования и проектирования. В результате 80% материала третьего издания добавлено по сравнению со вторым. Изменилась и организация книги – теперь больший упор делается на технику и стиль программирования. Рассматриваемые изолированно, особенности языка скучны — они оживают лишь в контексте приемов программирования, ради поддержки которых и задумывались.

Соответственно, C++ может теперь преподаваться как язык более высокого уровня. То есть основное внимание можно уделять алгоритмам и контейнерам, а не жонглированию битами, объединениями, строками в стиле C, массивами и проч. Более низкоуровневые понятия (такие как массивы, нетривиальное использование указателей и приведения типов), естественно, также придется изучить. Но их представление теперь удается отложить до тех пор, пока новичок в программировании на C++, читатель или студент не обретут зрелость, необходимую, чтобы воспринимать эти средства в контексте более высокоуровневых концепций, к применению которых обучаемые уже привыкли.

В частности, невозможно переусердствовать в подчеркивании важности преимущества статически типобезопасных строк и контейнеров перед стилями программирования, предполагающими использование множества макросов и приведений типов. В третьем издании «Языка программирования C++» мне удалось избавиться от почти всех макросов и свести использование приведений типов к немногочисленным случаям, когда они действительно существенны. Я считаю серьезным недостатком принятую в C/C++ форму макросов, которую теперь можно считать устаревшей благодаря наличию более подходящих средств языка, таких как шаблоны, пространства имен, встроенные функции и константы. Точно также, широкое использование приведений типов в любом языке сигнализирует о плохом проектировании. Как макросы, так и приведения являются частыми источниками ошибок. Тот факт, что без них можно обойтись, делает программирование на C++ куда более безопасным и элегантным.

Стандарт C++ предназначен для того, чтобы изменить наш способ программировать на C++, проектировать программы и обучать программированию на C++. Подобные изменения не происходят за один вечер. Я призываю вас не торопясь, внимательно присмотреться к стандарту C++, к приемам проектирования и программирования, используемым в третьем издании «Языка программирования C++», — и к вашему собственному стилю программирования. Предполагаю, что вполне возможны значительные улучшения. Но пусть ваша голова остается холодной! Чудес не бывает — при написании кода опасно использовать языковые возможности и приемы, которые вы понимаете лишь частично. Настало время изучать и экспериментировать — только поняв новые концепции и приемы, можно воспользоваться по-настоящему существенными преимуществами стандартного C++.

Добро пожаловать! *Бьерн Страуструп*

От редакции третьего русского издания

Книга, которую Вы держите в руках, не нуждается ни в представлении, ни в каком-либо особом вступлении со стороны редакции русского издания. Все что нужно, сказано автором. Существует, однако, несколько моментов, требующих пояснений, которые мы (с согласия автора) сочли возможным поместить до основного текста (в противном случае соответствующие сведения пришлось бы организовывать в сноски, что не всегда удобно читателю, тем более что не все читают книгу от корки до корки). Речь идет об исправлениях, внесенных в текст оригинального издания, о переводе некоторых терминов и о состоянии дел с прохождением стандарта C++. В сноски нами вынесено лишь несколько незначительных комментариев; эти немногочисленные сноски снабжены пометкой «Примеч. ред.», чтобы отличить их от авторских.

Исправления

К моменту завершения работы над черновым текстом русского перевода оригинальное англоязычное издание выдержало множество тиражей. Настоящий перевод отражает все исправления и улучшения, внесенные в оригинальное издание и собранные в файлах с 3rd_printing2.html по 3rd_printing18.html, которые доступны по адресу

http://www.research.att.com/~bs/3rd.html

Мы рекомендуем периодически заглядывать на Web-страничку с исправлениями в оригинальном издании, так как там могут появиться очередные файлы, работу с которыми мы вынуждены оставить заинтересованному читателю.

Web-страничку книги также можно отыскать по адресу:

http://www.aw.com/cseng

Перевод терминов

Мы очень старались переводить текст и особенно специальные термины таким образом, чтобы перевод был совершенно ясен и не нуждался в каких-либо комментариях. И все-таки в нескольких случаях принятые переводчиками и редакторами решения могут показаться произвольными, и мы хотели бы прокомментировать соответствующие термины.

Слово «statement» принято переводить на русский язык как «оператор». Мы привыкли к тому, что *if*, *while*, *case* и т. д. — это операторы. Увы, в контексте C++ такой перевод неприемлем. Дело в том, что в C++ слово «operator» (которое и переведено как «оператор») имеет совсем другое значение. Оно применяется для обозначения сложения (оператор +), разыменования (оператор *), выяснения размера (оператор *sizeof*()) и в других подобных случаях. С учетом того, что большинство операторов C++ допускает перегрузку, синтаксически и семантически напоминающую (пере)оп-

ределение функций, можно сказать, что операторы в C++ сродни функциям. Кстати, автор употребляет также и термин «operation» (так и переведенный: «операция»), который в C++ не имеет какого-то специального смысла и в большинстве случаев может рассматриваться или как синоним слова «оператор», или как обозначение какого-то действия. Что же касается термина «statement», то из предлагаемых различными словарями вариантов «утверждение», «предложение» и «инструкция» мы избрали последний, так как он, по-видимому, лучше всего соответствует сущности обозначаемых словом «statement» конструкций C++ и, кроме того, периодически встречается в книгах и документации в нужном значении. Итак, ***if***, ***while***, ***case***, ... — это инструкции, а +, −, ***sizeof*** (), ... — операторы. В частности, = — это оператор присваивания, семантику которого можно изменить путем перегрузки, а вот запись ***a*** = ***b***; является инструкцией присваивания, семантика которой неизменна, фиксирована языком и состоит в вызове (возможно перегруженного) оператора присваивания с аргументами ***a*** и ***b***.

Говоря о преобразованиях типов, мы используем два термина. Здесь мы просто следуем оригиналу: термин «conversion» переведен как «преобразование», а «cast» — как «приведение».

Оператор :: в переводе назван «оператором разрешения области видимости», что в точности соответствует оригинальному «scope resolution operator». Такой перевод представляется нам более удачным, чем иные варианты, предлагаемые словарями (например, «оператор определения контекста»). Разумеется, слово «разрешение» в данном случае употреблено в значении «уточнение, выяснение, разрешение неоднозначности», а не в смысле «позволение, допущение».

В C++, в отличие от некоторых других языков программирования, существенно различие между объявлением (declaration) и определением (definition) переменной, класса, функции и т. д. Не всякое объявление является определением, а в некоторых случаях объявление просто необходимо поместить до соответствующего определения. Таким образом, употребление двух различных терминов («объявление» и «определение») вместо единственного привычного многим слова «описание» является намеренным и неизбежным.

В авторском тексте встречаются как термин «header» («заголовок»), так и словосочетание «header file» («заголовочный файл»). Никаких смысловых различий между этими вариантами не проводится, поэтому в переводе мы чаще использовали более привычное «заголовочный файл», но иногда (хотя бы для разнообразия) оставляли и «заголовок».

Стандарт

В разгар работ по подготовке русского издания стандарт C++, на котором основана данная книга, находился в состоянии «ANSI/ISO final draft», то есть еще не был формально утвержден, но был «почти окончательно» согласован. На этой стадии стандарт уже не мог претерпеть сколь-нибудь существенных изменений, однако некоторые уточнения были еще возможны. По мере появления соответствующей информации необходимые исправления вносились как в оригинальное издание, так и в русский перевод (см. выше). В августе 1998 года стандарт ISO/IEC 14882 «Standard for the C++ Programming Language» был единогласно принят.

В заключении хотелось бы поблагодарить д-ра Страуструпа за блестящую книгу, работа с которой доставила нам истинное удовольствие, и за то, что он нашел время для подготовки предисловия к ее русскому изданию.

Предисловие

Программировать — значит понимать.
— Кристин Нюгард

Я нахожу C++ очень удобным языком, причем в последние годы мое мнение только укрепилось. За минувшие несколько лет средства поддержки проектирования и программирования на C++ улучшились кардинальным образом, и, кроме того, было предложено множество новых методов их использования. Однако программирование на C++ — не только доставляет удовольствие. Обыкновенные программисты-практики достигли значительного повышения производительности, уровня сопровождения, гибкости и качества в проектах самого разного масштаба. К настоящему времени C++ оправдал большинство надежд, которые я связывал с ним вначале, и даже преуспел в задачах, о которых я и не мечтал.

Эта книга представляет стандарт C++[1] и основные методы программирования и проектирования, поддерживаемые C++. Предлагаемый стандарт C++ гораздо мощнее и яснее, чем тот, который описан в первом издании этой книги. Новые свойства языка, такие как пространства имен, обработка исключений, шаблоны и определение типа во время выполнения, позволяют применять многие методы непосредственно — что было невозможно ранее, а стандартная библиотека предоставляет программисту возможность начать со значительно более высокого уровня, чем просто «голый» язык.

Всего лишь около трети информации из первого издания перешло во второе. Это, третье издание, является еще более полной переработкой предыдущего текста. В нем излагается информация, полезная даже самым опытным программистам на C++. С другой стороны, эту книгу новичку читать проще, чем предшествующие. Столь конструктивная переработка книги стала возможной благодаря широкому применению C++ и большому накопленному опыту.

Понятие стандартной библиотеки позволяет по-новому взглянуть на концепции C++. Как и ранее, в этой книге C++ описывается независимо от конкретной реализации, и учебные главы вводят конструкции и понятия «снизу вверх»; таким образом, они используются только после того, как описаны. Однако, значительно проще применять хорошо спроектированную библиотеку, чем понять детали ее реализации. Поэтому стандартную библиотеку можно использовать для создания реалистичных и интересных примеров задолго до того, как читатель познакомится с ее внутренним устройством. Стандартная библиотека сама по себе является богатейшим источником примеров программирования и проектирования.

В этой книге описываются все основные средства языка C++ и стандартной библиотеки. Изложение материала базируется на возможностях языка и библиотеки, рассматриваемых в контексте их использования. Таким образом, внимание, скорее сосредоточено на языке, как на средстве проектирования и программирования, чем на языке как таковом. Книга демонстрирует ключевые методы, которые делают C++ эффективным, и учит фундаментальным концепциям, необходимым для достижения мастерства. За исключением мест, иллюстрирующих технические детали, примеры

[1] ISO/IEC 14882, Standard for the C++ Programming Language (Стандарт языка программирования C++).

взяты из области системного программирования. В качестве дополнения к этой книге можно рассматривать справочник *The Annotated C++ Language Standard* («Стандарт языка C++ с комментариями»), в котором дано полное описание языка с исчерпывающими комментариями.

Основная цель данной книги — помочь читателю понять, как средства C++ поддерживают ключевые методы программирования. Задача состоит в том, чтобы поднять читателя над тем уровнем, на котором он в основном копирует примеры кода и эмулирует стили программирования, присущие другим языкам. Только хорошее понимание идей, стоящих за свойствами языка, ведет к мастерству. Изложенная информация, дополненная документацией по конкретной реализации, является достаточной для написания даже больших реальных проектов. Я надеюсь, что эта книга поможет читателю по-новому взглянуть на вещи и научиться лучше программировать и проектировать.

Благодарности

Помимо лиц, упомянутых в предисловиях к первому и второму изданиям, я хотел бы поблагодарить Мэта Остерна (Matt Austern), Ханса Боэма (Hans Boehm), Дона Кэлдвелла (Don Caldwell), Лоуренса Крала (Lawrence Crowl), Алана Феура (Alan Feuer), Эндрю Форреста (Andrew Forrest), Дэвида Гэя (David Gay), Тима Гриффина (Tim Griffin), Питера Джула (Peter Juhl), Брайана Кернигана (Brian Kenighan), Эндрю Коэнига (Andrwe Koeing), Майка Моубрэя (Mike Mowbray), Роба Мюррэя (Rob Murray), Ли Нэкмэна (Lee Nakman), Джозефа Ньюкамера (Joeseph Newcomer), Алекса Степанова (Alex Stepanov), Дэвида Вандевурда (David Vandevoorde), Питера Вейнбергера (Peter Weinberger) и Криса Ван Вика (Chris Van Wyk) за комментарии при подготовке глав третьего издания. Без их помощи и предложений эту книгу было бы сложнее понимать, она содержала бы больше ошибок, была бы несколько менее полной и, вероятно, оказалась бы немного тоньше.

Я также хотел бы поблагодарить добровольцев из комитета по стандартизации C++, которые проделали огромную конструктивную работу, чтобы C++ стал тем, что он есть. Несколько несправедливо выделять отдельных людей, но еще менее справедливым было бы не сказать ни о ком, поэтому я хотел бы упомянуть Майка Балла (Mike Ball), Дага Брюка (Dag Brück), Сина Корфилда (Sean Corfield), Тэда Голдстейна (Ted Goldstein), Кима Кнутила (Kim Knutilla), Эндрю Кенига (Andrew Koenig), Жози Лажойе (Josee Lajoie), Дмитрия Ленкова (Dmitry Lenkov), Натана Мейерса (Nathan Meyers), Мартина О'Риордана (Martin O'Riordan), Тома Плама (Tom Plum), Джонатана Шопиро (Jonathan Shopiro), Джона Спайсера (John Spicer), Джерри Шварца (Jerry Schwarz), Алекса Степанова (Alex Stepanov) и Майка Вилота (Mike Vilot) — тех, кто непосредственно работали со мной над некоторыми частями языка C++ и его стандартной библиотеки.

После выхода первого тиража этой книги[1] десятки людей присылали мне исправления и предложения по ее улучшению. Многие из этих предложений я смог учесть при подготовке очередных тиражей, что помогло значительно их усовершенствовать. Многое помогли разъяснить переводчики этой книги на различные языки. В ответ на просьбы читателей я добавил приложения Г и Д. Особенно я хотел бы поблагодарить за помощь: Дэйва Абрахамса (Dave Abrahams), Мэта Остерна (Matt Austern), Яна Бьелавски (Jan Bielawski), Янину Минсер Дашкевич (Janina Mincer Daszkiewicz), Эндрю Кёнинга (Andrew Koenig), Дитмара Куля (Dietmar Kühl), Николая Джозуттиса (Nicolai Josuttis), Натана Майерса (Nathan Myers), Пола Е. Севинка (Paul E. Sevinç), Энди Тенни-Сенса (Andy Tenne-Sens), Шоичи Учиду (Shoichi Uchida), Пин-Фея (Майка) Янга (Ping-Fai (Mike) Yang) и Дениса Йелли (Dennis Yelle).

Мюррей-Хилл, Нью-Джерси *Бьерн Страуструп*

[1] Имеется в виду 3-е английское издание. — *Примеч. ред.*

Предисловие ко второму изданию

Бежит дорога все вперед.
— Бильбо Бэггинс

Как и было предсказано в первом издании этой книги, C++ развивался в соответствии с запросами его пользователей. Эволюция языка направлялась опытом большого числа пользователей различного уровня, работающих в самых разнообразных прикладных областях. За шесть лет, минувших с выхода первого издания, сообщество пользователей C++ выросло в сотни раз. Множество новых методов было открыто и/или проверено на опыте. Часть этого опыта отражена в данной книге.

Главная цель развития языка за последние шесть лет состояла в улучшении абстракции данных, развитии средств объектно-ориентированного программирования в целом и превращение языка в удобный инструмент для написания высококачественных библиотек, в частности, библиотек типов, определяемых пользователем. «Высококачественная библиотека» — это та, которая предоставляется пользователю в виде одного или нескольких удобных, безопасных и эффективных классов. В этом контексте слово *безопасный* означает, что класс обеспечивает специфический, безопасный с точки зрения типов интерфейс между пользовательскими программами и своим содержимым. Слово *эффективный* означает, что использование класса не подразумевает значительных дополнительных затрат времени или памяти на этапе выполнения по сравнению с кодом, вручную написанным на языке C.

Данная книга дает полное описание языка C++. Главы с 1 по 10 содержат вводный обучающий материал. В главах с 11 по 13 обсуждаются вопросы, связанные с проектированием и разработкой программ. Далее следует полное справочное руководство по C++. Естественно, в этой книге представлены новые свойства и решения, принятые с момента выхода первого издания. Они включают: уточнение при разрешении перегрузки, средства управления памятью, механизмы управления доступом, безопасную с точки зрения типов компоновку, ***const*** и ***static*** функции-члены, абстрактные классы, множественное наследование, шаблоны и обработку исключений.

C++ является языком общего назначения. Его «родная» сфера применения — системные программы в самом широком смысле. К тому же, C++ успешно используется во многих прикладных областях, не попадающих в категорию системных. Реализации C++ существуют как на самых скромных микрокомпьютерах, так и на мощнейших суперкомпьютерах. Они разработаны для большинства операционных систем. Как следствие, в этой книге описывается собственно язык C++ без учета особенностей реализации, программного окружения или библиотек.

В книге имеется множество примеров классов, которые хотя и полезны, но должны быть охарактеризованы как «игрушечные». Такой стиль позволяет изложить общие

принципы и полезные методы более ясно, чем при написании реалистично сложной программы, где они были бы похоронены в деталях. Большинство приведенных здесь полезных классов, таких как связные списки, массивы, строки символов, матрицы, графические классы, ассоциативные массивы и т. д. можно найти в доведенном до полнейшего совершенства виде у целого ряда коммерческих и некоммерческих поставщиков. Многие из этих «промышленных» классов и библиотек в действительности являются прямыми или косвенными потомками игрушечных версий, изложенных здесь.

В этом издании делается больший, чем в предыдущем, акцент на обучении. Однако форма изложения материала по прежнему нацелена на опытных программистов и разработчиков, дабы не оскорблять их интеллект и опыт. Для удовлетворения потребности в информации, не относящейся непосредственно к свойствам языка и его использованию, значительно расширен круг вопросов, связанных с проектированием. Увеличено число технических деталей и повышена точность изложения. В частности, справочное руководство по языку отражает результат многолетней работы в этом направлении. Я хотел написать книгу с достаточно глубоким изложением материала, чтобы ее стоило прочесть не один раз. Другими словами, эта книга представляет язык C++, его фундаментальные принципы и ключевые методы, необходимые для применения языка. Добро пожаловать!

Благодарности

Помимо лиц, упомянутых в предисловии к первому изданию, я хотел бы поблагодарить Ала Ахо (Al Aho), Стива Бурова (Steve Buroff), Джима Коплина (Jim Coplien), Теда Голдстейна (Ted Goldstein), Тони Хансена (Tony Hansen), Лорэйна Джула (Lorraine Juhl), Питера Джула (Peter Juhl), Брайана Кернигана (Brian Kernighan), Эндрю Кенига (Andrew Koenig), Билла Легета (Bill Leggett), Уоррена Монтгомери (Warren Montgomery), Майка Моубрэя (Mike Mowbray), Роба Мюррея (Rob Murray), Джонатана Шопиро (Jonathan Shopiro), Майка Вилота (Mike Vilot) и Питера Венбергера (Peter Weinberger) за комментарии при подготовке глав второго издания. Многие люди оказали влияние на развитие C++ в период с 1985 по 1991 год. Я могу упомянуть только некоторых из них: Эндрю Кениг, Брайан Керниган, Дуг МакИлрой (Doug McIlroy) и Джонатан Шопиро. Я также благодарю всех участников «просмотра со стороны» черновика справочного руководства и людей, настрадавшихся в первый год работы комитета X3J16.

Мюррей-Хилл, Нью-Джерси *Бьерн Страуструп*

Предисловие к первому изданию

Язык формирует способ нашего мышления и предопределяет, о чем мы можем думать.
— Б. Л. Ворф

C++ — это язык программирования общего назначения, цель которого — сделать работу серьезных программистов более приятным занятием. За исключением несущественных деталей, C++ является надмножеством языка C. Помимо возможностей, предоставляемых C, C++ обеспечивает гибкие и эффективные средства определения новых типов. Программист может разделить приложение на несколько фрагментов, определив новые типы, отражающие базовые концепции предметной области. Такой способ разработки часто называют *абстракцией данных*. Объекты типов, определяемых пользователем, содержат необходимую информацию, свою для каждого типа. Такие объекты можно удобно и безопасно использовать даже в контексте, где их тип нельзя определить во время компиляции. Программы, использующие объекты таких типов, часто называют *объектными*. При надлежащем использовании подобные методы дают более короткие, понятные и простые в сопровождении программы.

Ключевое понятие в C++ — класс. Класс — это тип, определяемый пользователем. Классы обеспечивают сокрытие информации, гарантированную инициализацию данных, неявное преобразование определяемых пользователем типов, динамическое определение типа, контроль пользователя над управлением памятью и механизм перегрузки операторов. C++ предоставляет гораздо лучшие, чем C, средства для проверки типов и поддержки модульного программирования. Кроме того, язык содержит усовершенствования, непосредственно не связанные с классами, такие как: символические константы, встраивание функций в место вызова, аргументы функций по умолчанию, перегруженные имена функций, операторы управления свободной памятью и ссылки. C++ сохраняет способность языка C эффективно работать с аппаратной частью на уровне битов, байтов, слов, адресов и т. д. Это позволяет реализовывать типы, определяемые пользователем, с достаточной степенью эффективности.

C++ и стандартные библиотеки языка рассчитаны на переносимость. Текущая реализация будет работать на большинстве систем, поддерживающих язык C. Библиотеки языка C можно использовать в программе, написанной на C++, и большинство инструментальных средств C можно применять для C++.

Назначение данной книги, в первую очередь, состоит в том, чтобы помочь серьезным программистам освоить язык C++ и использовать его в нетривиальных проектах. Книга содержит полное описание C++, множество завершенных примеров и еще большее количество фрагментов программ.

Благодарности

C++ никогда не достиг бы зрелости без постоянного использования, предложений и конструктивной критики со стороны многих моих друзей и коллег. В частности, Том Каргил (Tom Cargil), Джим Коплен (Jim Coplien), Сту Фельдман (Stu Feldman), Сэнди Фрейзер (Sandy Fraser), Стив Джонсон (Steve Johnson), Брайан Керниган (Brian Kernighan), Барт Локанси (Bart Locanthi), Дуг МакИлрой (Doug McIlroy), Дэнис Ричи (Dennis Ritchie), Ларри Рослер (Larry Rosler), Джерри Шварц (Jerry Schwarz) и Джонатан Шопиро (Jon Shopiro) подарили мне несколько важных для развития языка идей. Дэйв Пресото (Dave Presotot) написал текущую реализацию библиотеки потокового ввода/вывода.

Кроме того, сотни людей внесли вклад в развитие языка C++ и его компилятора, присылая мне предложения по улучшению, описание встретившихся проблем и ошибок компилятора. Я могу упомянуть только нескольких: Гари Бишоп (Gary Bishop), Эндрю Хьюм (Andrew Hume), Том Карцес (Tom Karzes), Виктор Миленкович (Victor Milenkovic), Роб Мюррэй (Rob Murray), Леони Роуз (Leonie Rose), Брайан Шмульт (Brian Schmult) и Гари Уолкер (Gary Walker).

Многие люди помогли этой книге увидеть свет. Среди них: Джонатан Бентли (Jon Bentley), Лаура Ивз (Laura Eaves), Брайан Керниган (Brian Kernighan), Тэд Ковальский (Ted Kowalski), Стив Махани (Steve Mahaney), Джонатан Шопиро (Jon Shopiro) и участники семинара по C++ в Bell Labs, Колумбус, штат Огайо, 26–27 июня 1985 г.

Мюррей-Хилл, Нью-Джерси *Бьерн Страуструп*

Введение

Во введении представлен обзор основных концепций и свойств языка программирования C++ и его стандартной библиотеки. Объясняется общая структура книги и излагается подход, принятый при описании средств языка и методов их использования. Кроме того, вводные главы дают некоторую базовую информацию о языке C++, его структуре и примерах применения.

«... И ты, Маркус, ты дал мне многое; теперь я дам тебе хороший совет. Будь сразу многими. Брось играть в бытие одним лишь Маркусом Кокоза. Ты так сильно беспокоился о Маркусе Кокоза, что стал его рабом и пленником. Ты ничего не делал, не подумав прежде, как это повлияет на довольство и престиж Маркуса Кокоза. Ты всегда так опасался, что Маркус совершит глупость или заскучает. Ну так что с того? Во всем мире люди совершают глупости... Мне хочется, чтобы к тебе и твоему сердцу вновь вернулась легкость. С этого дня ты должен быть не одним, но многими людьми, столь многими, сколько ты можешь задумать...»

— Карен Бликсен (Karen Blixen)

(«Сновидцы», из книги «Семь готических историй»;
написано под псевдонимом Исак Динесен;
Random House, Inc.

1

Обращение к читателю

И молвил Морж:
«Пришла пора поговорить о многом».
— Льюис Кэрролл

Структура этой книги — как изучать C++ — структура C++ — эффективность и структура — философские замечания — исторические замечания — для чего используется C++ — C и C++ — рекомендации для программистов на C — рекомендации для программистов на C++ — размышления о программировании на C++ — советы — ссылки.

1.1. Структура этой книги

Книга состоит из шести частей:

Введение: в главах 1–3 приводится обзор языка C++, основных стилей программирования, которые он поддерживает, и стандартной библиотеки C++.

Часть I: главы 4–9 содержат учебное описание встроенных типов C++ и базовых средств построения программ.

Часть II: главы 10–15 содержат учебный материал по объектно-ориентированному и обобщенному программированию на C++.

Часть III: в главах 16–22 описана стандартная библиотека.

Часть IV: в главах 23–25 обсуждаются проблемы, связанные с проектированием и разработкой программ.

Приложения: Приложения А–Д содержат технические подробности языка.

В главе 1 приводятся обзор этой книги, некоторые рекомендации по ее чтению и базовая информация о C++ и методах применения языка. Вы можете бегло пролистать главу, прочитав только то, что покажется интересным, и вернуться к ней снова, после прочтения других частей книги.

В главах 2 и 3 описываются основные концепции и свойства языка программирования C++ и его стандартной библиотеки. Их цель — склонить вас уделить внимание фундаментальным концепциям и основным элементам языка путем демонстрации того, что может быть выражено на полном C++. Как минимум, эти главы должны убедить вас, что C++ — это не просто C, и что C++ прошел большой путь со времени первого и второго изданий этой книги. Глава 2 начинает знакомство с C++. Обсуждение сосредоточено на элементах языка, поддерживающих абстракцию данных, объектно-ориентированное и обобщенное программирование. Глава 3 знакомит вас с

базовыми принципами и основными возможностями стандартной библиотеки. Это позволит мне использовать средства стандартной библиотеки в последующих главах, а вам — задействовать их в упражнениях, вместо того, чтобы полагаться только на встроенные средства низкого уровня.

Ознакомительные главы дают примеры типичных методов программирования, которые применяются во всей книге. Чтобы обсуждение некоторых методов или свойств далее выглядело более конкретно и реалистично, я вначале коротко излагаю концепцию, а углубленное обсуждение оставляю на потом. Такой подход позволяет мне изложить конкретные примеры до того, как проблема будет обсуждена основательно. Таким образом, организация книги отражает мысль, что мы обычно учимся быстрее, продвигаясь от конкретного к абстрактному — даже там, где абстрактная идея в ретроспективе кажется простой и очевидной.

В части I описывается подмножество C++, которое поддерживает стили программирования, традиционные для C и Pascal. Она охватывает фундаментальные типы, выражения и управляющие структуры программ на языке C++. Здесь же обсуждается модульность — пространства имен, исходные файлы и обработка исключений. Я полагаю, что вы знакомы с основными принципами программирования, используемыми в части I. Например, я объясняю средства C++ для выражения рекурсии и итераций, но не трачу много времени на объяснение того, насколько эти понятия полезны сами по себе.

В части II описаны средства C++ для определения и использования новых типов. Здесь представлены (главы 12, 15) конкретные и абстрактные классы (интерфейсы), перегрузка операторов (глава 11), полиморфизм и иерархии классов (главы 12, 15). В главе 13 описаны шаблоны, то есть средства C++ для определения семейств типов и функций. Демонстрируются основные методы создания контейнеров, таких, как списки, и поддержки обобщенного программирования. В главе 14 описываются обработка исключений, методы обработки ошибок и стратегии, позволяющие создавать устойчивые к ошибкам программы. Я полагаю, что вы либо недостаточно знакомы с объектно-ориентированным и обобщенным программированием, либо можете получить пользу от изложения того, как абстракция данных поддерживается в C++. Таким образом, я не только описываю средства языка, которые поддерживают технику абстракции данных, но также объясняю и саму эту технику. Часть IV продвигается дальше в этом направлении.

В части III описывается стандартная библиотека C++. Цель части состоит в том, чтобы объяснить, как пользоваться библиотекой, продемонстрировать основные методы проектирования и программирования, а также показать, каким образом можно дополнять библиотеку. Библиотека предоставляет контейнеры (такие как ***list***, ***vector*** и ***map*** — главы 16, 17), стандартные алгоритмы (такие как ***sort***, ***find*** и ***merge*** — главы 18, 19), строки (глава 20), операции ввода/вывода (глава 21) и поддержку вычислений (глава 22).

В части IV обсуждаются вопросы, возникающие при проектировании и реализации больших программных систем. В главе 23 внимание сосредоточено на вопросах проектирования и управления. В главе 24 обсуждается связь между языком программирования C++ и проектированием. В главе 25 представлены некоторые способы использования классов при проектировании.

В приложении A описана грамматика C++ с комментариями. В приложении Б обсуждается связь между C и C++ и между Standard C++ (называемом также ISO C++ и ANSI C++) и предыдущими версиями C++. В приложении В представлены

некоторые примеры, связанные с техническими аспектами языка. Приложение Г посвящено средствам стандартной библиотеки, обеспечивающим локализацию программного обеспечения. В приложении Д обсуждаются гарантии безопасности исключений и соответствующие требования стандартной библиотеки.

1.1.1. Примеры и ссылки

В этой книге акцент сделан скорее на организации программ, а не на создании алгоритмов. Соответственно, я избегал заумных и трудных для понимания алгоритмов. Тривиальный алгоритм обычно лучше подходит для иллюстрации нюансов определения языка или структуры программы. Например, в этой книге я использую сортировку Шелла, тогда как в реальной жизни быстрая сортировка подходит лучше. Часто реализация на базе более подходящего алгоритма предлагается в качестве упражнения. В реальном коде обычно более желателен вызов библиотечной функции, нежели код, используемый здесь для иллюстрации средств языка.

Учебные примеры неизбежно дают искаженное представление о разработке программ. За счет упрощения примеров, сложности, возникающие из-за больших размеров программ, исчезают. Я не вижу другого пути, кроме написания программ реального размера, для получения реального представления о программировании и конкретном языке программирования. Основное внимание в этой книге уделяется элементам языка, базовым методам, на основе которых строятся все программы, а также правилам этого построения.

Выбор примеров обусловлен моим прошлым опытом в моделировании, написании компиляторов и библиотек. Примеры являются упрощенными вариантами того, что содержится в реальном коде. Упрощение необходимо, чтобы язык программирования и вопросы, связанные с проектированием, не затерялись среди деталей. В книге нет претенциозных примеров, не имеющих аналогов в реальном коде. Там где это возможно, я переносил в приложение В примеры, связанные с техническими деталями языка. В них частенько используются переменные с именами x и y, типы называются A и B, а функции — $f()$ и $g()$.

В примерах программ используется пропорциональный шрифт. Например:

```
#include<iostream>

int main()
{
    std::cout << "Здравствуй, мир!\n";[1]
}
```

На первый взгляд такое написание кажется неестественным для программистов, привыкших видеть шрифт с буквами одинаковой ширины. Однако пропорциональный шрифт легче и лучше воспринимается, чем моноширинный. Использование пропорционального шрифта также позволяет избежать многих неестественных разры-

[1] Здесь и далее многие строки будут записываться по-русски. В большинстве случаев это не должно вызвать проблем. Однако следует учитывать, что если вы пишете консольное приложение для Windows (например, в Microsoft Visual C++ 5.0), то выводимый русский текст может оказаться нечитаемым: в интегрированной среде он по умолчанию набирается в кодировке Windows, а консоль правильно отображает русские буквы только в кодировке DOS. — *Примеч. ред.*

вов строк в коде. Более того, мои эксперименты показали, что большинство людей через некоторое время считают новый стиль более читабельным.

Там где это возможно, свойства языка C++ и библиотеки описаны скорее в контексте их использования, чем в сухой манере руководства или справочника. Представленные элементы языка и степень детализации их описания отражают мой взгляд на то, что именно необходимо для эффективного использования C++. Дополнением к настоящему изданию сможет послужить книга *The Annotated C++ Language Standard* (*Стандарт языка C++ с комментариями*), которую мы с Эндрю Кенигом (Andrew Koenig) сейчас пишем. Это издание будет содержать полное определение языка с комментариями, призванными сделать его более доступным для понимания. Логично предположить, что должно быть и другое дополнение, *The Annotated C++ Standard Library* (*Стандартная библиотека C++ с комментариями*). Однако ввиду того, что время и мои возможности как автора ограничены, я не могу обещать, что подготовлю такую книгу.

Ссылки на разделы книги даны в форме § 2.3.4 (глава 2, раздел 3, подраздел 4), § Б.5.6 (приложение Б, подраздел 5.6) или § 6.6[10] (глава 6, упражнение 10). Курсив используется для логического выделения (например, «строковая переменная *не* допустима»), при первом упоминании важной концепции (например, *полиморфизм*) и в комментариях в коде. Полужирным курсивом пишутся идентификаторы, ключевые слова и числовые величины в примерах (например, ***class***, ***counter***, ***1712***).

1.1.2. Упражнения

Упражнения находятся в конце глав. Большинство упражнений принадлежат к типу «напиши программу». Всегда пишите код, достаточный для того, чтобы его можно было откомпилировать и запустить с несколькими (по крайней мере) тестовыми вариантами данных. Упражнения значительно различаются по сложности, поэтому они помечены цифрами, оценивающими их сложность. Масштаб экспоненциальный, так что упражнение (*1) займет десять минут, (*2) может занять час, а (*3) — целый день. Время, необходимое для написания программы, больше зависит от вашего опыта, чем от самого упражнения. Упражнение со сложностью (*1) может занять у вас целый день, если вы впервые знакомитесь с новой компьютерной системой и разбираетесь, как ее запустить. С другой стороны, упражнение (*5) может быть сделано кем-нибудь за час, если у него под рукой находится подходящий набор утилит.

Любая книга по программированию на C может быть использована в качестве дополнительных упражнений к части I. Любая книга по структурам данных и алгоритмам может быть использована в качестве источника упражнений для частей II и III.

1.1.3. Замечания по реализации

В книге используется «чистый C++», описанный в стандарте C++ [C++, 1997], [C++, 2003]. Поэтому примеры должны выполняться в любой реализации C++. Большинство фрагментов программ в этой книге были проверены с использованием нескольких реализаций C++. Примеры, использующие свойства, только недавно реализованные в C++, компилировались не всеми реализациями. Однако я не вижу смысла в упоминании реализаций, на которых они не компилировались. Эта информация скоро устареет, потому что поставщики компиляторов ин-

тенсивно работают и быстро внедряют все новые свойства C++. В приложении Б вы найдете предложения по устранению проблем, связанных со старыми компиляторами C++ и кодом, написанным для компиляторов C.

1.2. Изучение C++

Самая важное при изучении C++ — сконцентрировать внимание на концепциях и не потеряться в технических деталях языка. Целью изучения языка программирования является повышение профессионального уровня, то есть вы должны научиться более эффективно проектировать и реализовывать новые системы, равно как и поддерживать существующие. Для этой цели осознание методов проектирования и программирования является более важным, нежели понимание деталей — последнее придет позднее вместе с опытом.

C++ поддерживает множество стилей программирования. Все они основаны на строгой статической проверке типов, и целью большей их части является достижение высокого уровня абстракции данных и непосредственного отображения идей программиста. Каждый стиль может эффективно достичь этих целей, в то же время обеспечивая эффективность выполнения и использования ресурсов. Программисты, приходящие из различных языковых сред (например, C, Fortran, Lisp, ML, Ada, Eiffel, Pascal или Modula-2), должны понять: для того чтобы воспользоваться преимуществами C++, необходимо потратить время на изучение и применение стилей программирования, приемлемых для C++. Тоже самое касается программистов, пользовавшихся более ранней и менее выразительной версией C++.

Бездумное применение в новом языке методов, эффективных в другом языке, обычно ведет к неуклюжему, медленному и сложному в сопровождении коду. Такой код к тому же еще и неприятно писать, потому что каждая строка кода и каждое сообщение компилятора об ошибке напоминают программисту, что используемый язык отличается от «старого языка». Вы можете писать в стиле Fortran, C, Smalltalk и т. п. на любом языке, но это и неприятно и неэкономично в языке с другой философией. Любой язык может послужить щедрым источником идей о том, как писать программы на C++. Однако для того, чтобы быть эффективным в новом контексте, эти идеи нужно трансформировать в нечто другое, более соответсвующее общей структуре и системе типов C++. Над базовой системой типов возможны только пирровы победы.

C++ поддерживает постепенный подход к обучению. Ваш путь к изучению нового языка программирования зависит от того, что вы уже знаете и что хотите узнать. Нет единого для всех подхода. Я полагаю, что вы изучаете C++, чтобы повысить свое профессиональное мастерство программиста и проектировщика. То есть я полагаю, что вашей целью изучения C++ является не просто освоение нового синтаксиса для применения в том же духе, как вы это делали раньше, а изучение новых и лучших способов построения систем. Обучение должно происходить постепенно, потому что освоение существенно новых навыков требует времени и практики. Подумайте о том, сколько времени заняло бы качественное изучение нового для вас естественного языка или обучение игре на новом музыкальном инструменте. Стать более профессиональным разработчиком легче и быстрее, но не настолько легко и быстро, как хотелось бы большинству людей.

Из этого следует, что вы будете использовать C++ (зачастую для построения реальных систем) до того, как поймете все свойства и методы языка. Поддерживая

несколько парадигм программирования (глава 2), C++ позволяет продуктивно писать на нескольких уровнях мастерства. Каждый новый стиль программирования добавляет инструмент в вашу коллекцию, но каждый действенен и сам по себе, повышая эффективность вашей работы как программиста. C++ организован таким образом, что вы можете изучать его концепции строго последовательно и получать практические результаты уже в процессе изучения. Кроме того, результаты, как правило, оказываются прямо пропорциональны вложенным усилиям.

Несмотря на продолжительность дебатов на тему «стоит ли изучать C до C++», я твердо убежден, что лучше всего начинать непосредственно с C++. C++ безопаснее, выразительнее, он уменьшает необходимость концентрации внимания на низкоуровневой технике. Вам будет легче выучить хитроумные штучки на C, которые требуются для компенсации отсутствия в нем высокоуровневых возможностей, после того, как вы познакомитесь с общим подмножеством C и C++ и с некоторыми методами высокого уровня, поддерживаемыми в C++ непосредственно. Приложение Б является руководством для программистов, переходящих с C++ на C, например с целью использования уже созданного кода.

Существует несколько независимо разработанных и распространяемых реализаций C++. Вы можете выбирать среди большого количества инструментальных средств, библиотек и сред разработки программ. Существует масса учебников, руководств, журналов, газет, электронных досок объявлений, списков рассылки, конференций и курсов, которые могут познакомить вас с последними достижениями в области C++, его использования, инструментов, библиотек, реализаций и т. п. Если вы собираетесь серьезно использовать C++, я настоятельно рекомендую получить доступ к таким источникам. Каждый имеет свои приоритеты и склонности, поэтому воспользуйтесь по крайней мере двумя. Например, см. [Barton, 1994], [Booch, 1994], [Hencricson, 1997], [Koenig, 1997], [Martin, 1995].

1.3. Структура C++

Важным критерием при проектировании языка была простота. Там где существовал выбор между упрощением определения языка и упрощением компилятора, делался выбор в пользу первого. Однако, большое внимание было уделено совместимости с C [Koenig, 1989], [Stroustrup, 1994], (приложение Б), что выразилось в сохранении синтаксиса C.

В C++ нет встроенных высокоуровневых типов данных и операций. Например, в C++ нет типа «матрица с операцией взятия обратной» или «строка с оператором конкатенации». Если пользователю нужен такой тип, он может быть определен средствами языка. В действительности, определение новых типов общего назначения или специфичных для данного приложения является основной работой при программировании на C++. Хорошо спроектированный тип, определяемый пользователем, отличается от встроенного только тем, как он определен, а не тем, как он используется. Стандартная библиотека C++, описанная в части III, предоставляет множество примеров таких типов. С точки зрения пользователя, разница между встроенным типом и типом, предоставляемым стандартной библиотекой, невелика.

При проектировании C++ избегались средства, которые потребовали бы дополнительных затрат времени или ресурсов на этапе выполнения даже в тех случаях, когда они не используются. Например, конструкции, которые потребовали бы хранения «внутренней информации» для каждого объекта не были реализованы; поэтому,

если пользователь объявляет структуру, состоящую из двух 16-битных элементов, эта структура поместится в один 32-битный регистр.

C++ разрабатывался для использования в традиционной среде «компиляция–выполнение», то есть типичной среде программирования на C в системе UNIX. К счастью, C++ никогда не ограничивался только UNIX, он просто использовал UNIX и C в качестве модели для связи языка с библиотеками, компиляторами, компоновщиками, средой выполнения и т. п. Эта минимальная модель помогла C++ успешно работать практически на любой платформе. Тем не менее, существуют веские причины использовать C++ в окружении, обеспечивающем гораздо большую поддержку. Такие средства, как динамическая загрузка, инкрементная компиляция и база данных определений типов могут с успехом использоваться, не оказывая влияния на сам язык.

Проверка типов и сокрытие данных в C++ базируются на анализе программы во время компиляции, что предохраняет от случайной порчи данных. Эти меры не обеспечивают секретности или защиты от того, кто намеренно нарушает правила. Однако, они могут свободно использоваться, не влияя на время выполнения или требуемые ресурсы. Идея состоит в том, что свойство языка должно быть не только элегантным, но и приемлемым в контексте реальной программы.

Систематическое и детальное описание структуры C++ см. в [Stroustrup, 1994].

1.3.1. Эффективность и структура

C++ создан на основе языка программирования C, и, за небольшим исключением, C остается подмножеством C++. Основа языка, C-подмножество C++, разработана таким образом, что существует очень тесная связь между типами, операторами, инструкциями и объектами, с которыми непосредственно работает компьютер: числами, символами, адресами. За исключением операторов ***new***, ***delete***, ***typeid***, ***dynamic_cast***, ***throw*** и блока ***try***, отдельные выражения и инструкции C++ не требуют поддержки во время выполнения.

C++ может использовать те же механизмы вызова функций и возвращения результатов, что и C, или даже более эффективные. В тех случаях, когда даже такие эффективные механизмы слишком дороги, функция в C++ может быть встроена в место вызова. Таким образом, мы получаем преимущества функций, не неся при этом дополнительных накладных расходов.

Одной из целей языка C было устранение необходимости кодировать на ассемблере наиболее критичные системные программы. При развитии C++ достижение компромисса в этой области не было главной задачей. Разница между C и C++ состоит, в первую очередь, в степени акцента на типах и структуре. C — выразителен и гибок. C++ — еще более выразителен. Однако, для достижения этой выразительности вы должны уделять большее внимание типам объектов. Зная типы объектов, компилятор может правильно обращаться с выражениями, тогда как в противном случае вы должны были бы детально определять операции. Знание типов объектов также позволяет компилятору обнаруживать ошибки, которые в противном случае не были бы выявлены до этапа тестирования или даже во время тестирования. Имейте в виду, что использование системы типов для проверки аргументов функций, защиты данных от случайного изменения, создания новых типов, введения новых операторов и т. д. не увеличивает времени выполнения и не требует дополнительных системных ресурсов.

Акцент на структуры в C++ отражает увеличение размеров программ по сравнению с временами C. Вы можете написать небольшую программу (скажем 1000 строк), используя грубую силу и нарушая все правила хорошего стиля. Для программ большего размера вы не сможете этого сделать. Если структура программы, состоящей из 100 000 строк, плоха, вы обнаружите, что новые ошибки появляются с той же скоростью, с которой исправляются старые. C++ разрабатывался таким образом, чтобы предоставить возможность рационально структурировать большие программы, и чтобы один человек мог работать с большим объемом кода. Кроме того, хотелось, чтобы средняя строка кода C++ имела большую выразительность, нежели средняя строка на C или Pascal. К настоящему времени C++ перевыполнил эти задачи.

Не каждый фрагмент кода может быть хорошо структурирован, независим от аппаратуры, легко читаем и т. д. В C++ вы можете работать с аппаратными средствами непосредственно и эффективно, не принося в жертву безопасность и легкость понимания. Язык также имеет средства сокрытия такого кода за элегантными и безопасными интерфейсами.

Естественно, использование C++ для написания больших программ ведет к применению C++ группами программистов. При этом акцент C++ на модульность, интерфейсы со строгим определением типов и гибкость окупается сторицей. C++ имеет столь же удовлетворительный набор средств для написания больших программ, как и любой другой язык. Но по мере увеличения программы сложности, связанные с ее разработкой и поддержкой, смещаются от чисто языковых в сторону более глобальных проблем инструментальных средств и управления. В части IV обсуждаются такого рода вопросы.

Книга акцентирует внимание на методах создания программ общего назначения, полезных типов, библиотек и т. д. Эти методы будут полезны как программистам, пишущим небольшие программы, так и создателям больших приложений. Более того, так как все нетривиальные программы состоят из множества почти независимых частей, методы написания таких частей будут интересны всем программистам.

Вы можете заподозрить, что спецификация программы посредством использования более детализированной структуры типов ведет к увеличению размеров исходных текстов. В C++ это не так. Программа на C++, в которой объявлены типы аргументов функций, используются классы и т. д., обычно слегка меньше, чем эквивалентная программа на C, не применяющая эти средства. Если же используются библиотеки, программа на C++ окажется намного короче эквивалента на C — разумеется в предположении, что работающий эквивалент на C вообще может быть создан.

1.3.2. Философские замечания

Язык программирования служит двум взаимосвязанным целям: он предоставляет программисту инструмент для описания подлежащих выполнению действий и набор концепций, которыми оперирует программист, обдумывая, что можно сделать. Первая цель в идеале требует языка, близкого к компьютеру, чтобы все важные элементы компьютера управлялись просто и эффективно способом, достаточно очевидным для программиста. Язык C создавался, отталкиваясь именно от этой идеи. Вторая цель в идеале требует языка, близкого к решаемой задаче, чтобы концепции решения могли быть выражены понятно и непосредственно. Эта идея привела к пополнению C свойствами, превратившими его в C++.

Связь между языком, на котором мы думаем (пишем программы), и задачами (решениями), которые мы можем себе представить, очень тесная. По этой причине огра-

ничение возможностей языка с целью предотвращения программистских ошибок в лучшем случае опасно. Также как и в случае с естественными языками, огромную пользу приносит знание по крайней мере двух языков. Язык предоставляет программисту набор концептуальных средств. Если эти средства не адекватны поставленной задаче, они просто игнорируются. Качественное проектирование и отсутствие ошибок не могут быть гарантированы просто присутствием или отсутствием специфических возможностей в языке.

Использование типов особенно полезно в нетривиальных задачах. Понятие класса в C++ на практике оказалось мощным концептуальным инструментом.

1.4. Исторические замечания

Я придумал C++, записал его первоначальное определение и выполнил первую реализацию. Я выбрал и сформулировал критерии проектирования C++, разработал его основные возможности и отвечал за судьбу предложений по расширению языка в комитете по стандартизации C++.

Очевидно, что C++ многим обязан C [Kernighan, 1978]. Язык C остается подмножеством C++ (но в C++ устранены несколько серьезных брешей системы типов C; подробнее см. в приложении Б). Я также сохранил средства C, которые являются достаточно низкоуровневыми, чтобы справляться с самыми критичными системными задачами. Язык C, в свою очередь, многим обязан своему предшественнику, BCPL [Richards, 1980]; кстати, стиль комментариев // был взят в C++ из BCPL. Другим основным источником вдохновения был язык Simula67 [Dahl, 1970], [Dahl, 1972]. Концепция классов (с производными классами и виртуальными функциями) была позаимствована из него. Средства перегрузки операторов и возможность помещения объявлений в любом месте, где может быть записана инструкция, напоминает Algol68 [Woodward, 1974].

Со времени первого издания этой книги язык интенсивно пересматривался и обновлялся. Вот области, наиболее подвергшиеся изменениям: разрешение перегрузки, компоновка и средства управления памятью. Кроме того, для улучшения совместимости с C внесено несколько менее значительных изменений. Добавлено несколько обобщений и важных расширений: множественное наследование, статические функции-члены, константные функции-члены, защищенные члены класса, шаблоны, обработка исключений, определение типа во время выполнения, пространства имен. Общей целью этих изменений и расширений было улучшение средств языка для написания и использования библиотек. Развитие C++ описано в [Stroustrup, 1994].

Шаблоны изначально разрабатывались для реализации контейнеров со статически определенными типами (такими как списки, вектора и ассоциативные массивы) и поддержки элегантного и эффективного использования таких контейнеров (обобщенное программирование). Ключевой целью было сокращение использования макросов и явных преобразований типов. Идея шаблонов была частично вдохновлена механизмом наследования в языке Ada (как его достоинствами, так и недостатками) и параметризованными модулями в Clu. Аналогично, механизм обработки исключений возник на основе Ada [Ichbian, 1979], Clu [Liskov, 1979] и ML [Wikström, 1987]. Другие нововведения периода с 1985 по 1995 год, такие как множественное наследование, чисто виртуальные функции и пространства имен, являются скорее обобщениями, сделанными на основе опыта использования C++, чем идеями, взятыми из других языков.

Более ранние версии языка, широко известные под названием «С с классами» [Stroustrup, 1994], появлялись с 1980 года. Изначально язык был изобретен, потому что я хотел писать управляемые событиями моделирующие программы, для которых Simula67 была бы идеальным средством, если бы не требования эффективности. «С с классами» тщательно тестировался на проектах, требовавших минимального объема памяти и времени выполнения. В нем отсутствовали перегрузка операторов, ссылки, виртуальные функции, шаблоны, исключения и многое другое. Впервые C++ был использован вне исследовательской организации в июле 1983 года.

Название «C++» (произносится «си плюс-плюс»[1]) было придумано Риком Маскитти (Rick Mascitti) летом 1983 года. Имя отражает природу изменений в языке по сравнению с С. «++» — это оператор инкремента в С. Чуть более короткое имя «C+» является синтаксической ошибкой и, кроме того, оно уже использовалось в качестве названия другого языка. Знатоки семантики С считают, что C++ предпочтительней ++C. Язык не назван D («Ди»), потому что является расширением С и не пытается устранять проблемы путем удаления элементов С. Другие интерпретации названия C++ можно найти в приложении к [Orwell, 1949].

Первоначальная цель создания C++ состояла в том, чтобы избавить автора и его друзей от программирования на ассемблере, С или других современных языках высокого уровня. Главной целью было облегчение программирования и превращение его в более приятное занятие. C++ никогда не проектировался на бумаге: проектирование, документирование и реализация шли параллельно. Не существовало «C++ проекта» или «комитета по C++». C++ развивался с целью разрешения проблем, с которыми сталкивались пользователи, а также в результате моих разговоров с друзьями и коллегами.

Позднее интенсивный рост C++ вызвал некоторые изменения. Около 1987 года стало очевидно, что неизбежно создание формального стандарта C++, и что нам необходимо начать готовить почву для этих усилий [Stroustrup, 1994]. Результатом явилось сознательное расширение контактов между создателями компиляторов C++ и основными пользователями посредством переписки, электронной почты, личных встреч на конференциях, посвященных C++, и в других местах.

AT&T Bell Laboratories внесла значительный вклад в этот процесс, позволив мне делиться предварительными версиями руководства по C++ с разработчиками компиляторов и пользователями. Значение этого вклада не следует недооценивать, потому что многие из этих людей работали в фирмах, являвшихся конкурентами AT&T. Менее просвещенная компания могла бы создать серьезные проблемы, которые бы привели к фрагментации языка. Сотни отдельных лиц из десятков организаций прочли и высказали свое мнение по поводу того, что стало руководством по C++ и базовым документом для стандарта ANSI C++. Их имена можно найти в *The Annotated C++ Reference Manual* (*Справочное руководстве по языку C++ с комментариями*) [Ellis, 1989]. В конце концов, в декабре 1989 года по инициативе Hewlett-Packard был создан комитет X3J16 при ANSI. В июне 1991 года усилия ANSI (Американский Национальный Институт по Стандартизации) по стандартизации C++ стали частью деятельности ISO (Международная Организация по Стандартизации). С 1990 года эти два объединенных комитета по стандарту C++ стали основным форумом, содействовавшим развитию C++. Я работал в этих комитетах с самого начала. В качестве главы

[1] Можно произносить также «си плас плас» (на английский манер). — *Примеч. ред.*

рабочей группы по расширениям языка я непосредственно отвечал за обработку предложений по изменениям в C++ и добавлению новых средств. Первоначальный проект стандарта был выпущен для публичного обсуждения в апреле 1995 года. Международный стандарт ISO (ISO/IEC 14882) по C++ был принят в 1998 году. Техническое соглашение, устраняющее незначительные ошибки и противоречия, было введено в 2003 г. [C++, 2003].

C++ развивался рука об руку с некоторыми из классов, описанными в этой книге. Например, я разработал классы комплексных чисел, векторов и стека и ввел перегрузку операторов. Те же усилия привели Джонатана Шопиро (Jonathan Shopiro) и меня к созданию классов строк и списков. Классы Джонатана для строк и списков были первыми активно использовавшимися библиотечными классами. Класс строк в стандартной библиотеке C++ происходит от этих более ранних классов. Библиотека задач, описанная в [Stroustrup, 1987] и в § 12.7[11], была частью первой в мире программы, написанной на «C с классами». Я написал ее и соответствующие классы для поддержки стиля программирования, присущего Simula. Библиотека задач была пересмотрена и переписана в значительной степени Джонатаном Шопиро и до сих пор активно используется. Библиотека потоков, как она описана в первом издании этой книги, была разработана и реализована мной. Джерри Шварц (Jerry Schwarz) преобразовал ее в библиотеку потокового ввода/вывода (глава 21), использовав технику манипуляторов Эндрю Кенига и другие идеи. В дальнейшем библиотека потокового ввода/вывода обновлялась в процессе стандартизации. Большая часть работы была проделана Джерри Шварцем, Натаном Майерсом (Nathan Myers) и Норихиро Кумагайи (Norihiro Kumagai). На развитие шаблонов оказали влияние реализации шаблонов ***vector***, ***map***, ***list*** и ***sort***, выполненные Эндрю Кенигом, Алексом Степановым (Alex Stepanov), мной и другими. В свою очередь, работа Алекса Степанова о механизме обобщенного программирования с использованием шаблонов привела к контейнерам и некоторым алгоритмам стандартной библиотеки C++ (§ 16.3, главы 17, 18, § 19.2). Библиотека ***valarray*** для численных расчетов (глава 22) изначально была результатом работы Кента Баджа (Kent Budge).

1.5. Использование C++

C++ используется сотнями тысяч программистов практически во всех прикладных областях. Язык поддерживается десятком независимых реализаций, сотнями библиотек, руководств, несколькими техническими журналами, многочисленными конференциями и бесчисленными консультантами. Широко доступно обучение на различном уровне.

Ранние приложения имели ярко выраженный системный оттенок. Например, несколько основных операционных систем и ключевые части многих других были написаны на C++ [Campbell, 1987], [Rozier, 1988], [Hamilton, 1993], [Berg, 1995], [Parrington, 1995]. Я не шел на компромиссы, которые могли бы привести к снижению эффективности низкоуровневых средств C++. Эффективность позволяет нам использовать C++ для написания драйверов и других программ, предназначенных для непосредственного управления аппаратурой в реальном времени. В таком коде предсказуемость выполнения играет по меньшей мере такую же роль, как и скорость. Часто то же самое можно сказать и о размере программы. Каждое свойство C++ вводилось с учетом ограничений по времени выполнения и требуемой памяти [Stroustrup, 1994, § 4.5].

В большинстве приложений есть участки кода, критичные ко времени выполнения. Однако большая часть кода расположена не в них. Как правило, ключевыми аспектами являются удобство сопровождения, расширяемость и простота тестирования. Эти свойства C++ привели к его широкому использованию в областях, где совершенно необходима надежность, а также там, где требования к программам значительно меняются со временем. В качестве примеров можно привести банковское дело, торговлю, страхование, телекоммуникации и военные приложения. В течение многих лет централизованное управление системой междугородних переговоров в США осуществлялось программой на C++, и каждый звонок по номеру, начинающемуся с 800 (то есть звонок, оплачиваемый вызываемой стороной), маршрутизировался программой, написанной на C++ [Kamath, 1993]. Многие из этих приложений являются большими программами и живут очень долго. Соответственно, стабильность, совместимость и масштабируемость всегда оставались в центре внимания при развитии C++. Программы на C++, содержащие миллион строк, не являются большой редкостью.

Создание C++ (как и C) не имело целью поддержку численных расчетов. Тем не менее, на C++ решается множество численных, научных и инженерных задач. Главная причина этого в том, что традиционные численные задачи часто должны совмещаться с графикой и вычислениями, связанными со структурами данных, которые не укладываются в традиционный Fortran [Budge, 1992], [Barton, 1994]. Графика и пользовательские интерфейсы — области интенсивного использования C++. Любой, кто работал на Apple Macintosh или PC под управлением Windows, косвенно пользовался C++, потому что базовые пользовательские интерфейсы этих систем написаны на C++. Таким образом, C++ является типичным языком для написания приложений со сложным пользовательским интерфейсом.

Вышесказанное определяет самую сильную сторону C++: возможность эффективного использования в программах, предназначенных для широкого диапазона прикладных областей. В этом смысле приложение, включающее в себя доступ к локальной и глобальной сетям, численные расчеты, графику, интерактивное взаимодействие с пользователем и обращение к базе данных, можно считать типичным. Традиционно эти области считались раздельными и они часто обслуживались различными техническими группами с использованием разных языков программирования. C++ широко применяется во всех этих областях. Более того, он способен сосуществовать с фрагментами кода и программами, написанными на других языках.

C++ широко используется для обучения и исследований. Это удивляет многих, кто (справедливо) утверждает, что C++ не является ни самым маленьким, ни самым понятным языком. Тем не менее, он:

- достаточно ясен для обучения основным концепциям;
- достаточно реалистичен, эффективен и гибок для критичных проектов;
- достаточно доступен для организаций и объединений с различными средами разработки и выполнения;
- достаточно содержателен, чтобы служить инструментом обучения более сложным концепциям и методам;
- достаточно «рыночен», чтобы приобретенные знания оказались полезными вне академической среды.

C++ — язык, вместе с которым вы можете расти.

1.6. C и C++

В качестве базового языка для C++ был выбран C, потому что он:

[1] является многоцелевым, лаконичным и относительно низкоуровневым языком;
[2] подходит для решения большинства системных задач;
[3] исполняется везде и на всем;
[4] стыкуется со средой программирования UNIX.

В C есть свои проблемы, но их имел бы и разработанный с нуля язык, а проблемы C нам известны. Важно и то, что, благодаря C, язык «C с классами» стал мощным (разве что несколько неуклюжим) инструментом уже в первые несколько месяцев с момента зарождения идеи добавить Simula-подобные классы в C.

По мере того, как C++ набирал популярность и его возможности все более превосходили средства C, вновь и вновь поднимался вопрос о целесообразности сохранения совместимости. Очевидно, что можно было бы избежать некоторых проблем, доставшихся по наследству от C (см., например, [Sethi, 1981]). Совместимость была оставлена, потому что:

[1] существуют миллионы строк кода на C, которые могут выиграть от использования C++, если не потребуется их полного переписывания с C на C++;
[2] на C написаны миллионы строк библиотечных функций и утилит; они могут быть использованы программами на C++, благодаря синтаксическому сходству и совместимости по компоновке с C;
[3] сотни тысяч программистов знают C; им требуется изучить только новые свойства C++, не переучивая основы;
[4] C++ и C будут использоваться на одних и тех же системах одними и теми же людьми в течение многих лет, так что разница должна быть либо очень незначительной, либо очень большой, чтобы свести ошибки к минимуму и избежать путаницы.

Определение C++ было пересмотрено с тем, чтобы любая конструкция, допустимая и в C, и в C++, имела бы одинаковый смысл в обоих языках (с несколькими незначительными исключениями, см. § Б.2).

Сам язык C частично развивался под влиянием C++ [Rosler, 1984]. Стандарт ANSI C [C, 1990] содержит синтаксис объявления функций, заимствованный из «C с классами». Заимствование происходит в обоих направлениях. Например, тип указателя ***void**** придуман для ANSI C, а реализован впервые в C++. Как и было обещано в первом издании этой книги, определение C++ было пересмотрено с целью устранения непринципиальных несовместимостей. В настоящее время, C++ более совместим с C, чем вначале. Идеал C++ — быть близким к ANSI C настолько, насколько это возможно, но не ближе [Koenig, 1989]. Стопроцентная совместимость никогда не являлась целью. Иначе пришлось бы пожертвовать безопасностью с точки зрения типов и гармоничным сочетанием встроенных и определяемых пользователем типов.

Знание C не является обязательным для изучения C++. Программирование на C поощряет многие технические трюки, которые становятся ненужными благодаря C++. Например, явное преобразование типов в C++ используется реже, чем в C (§ 1.6.1). Однако *хорошие* программы на C имеют тенденцию походить на программы на C++. Например, любая программа из книги *The C Programming Language (2nd Edition)*, Kernighan and Ritchie (*Язык программирования C, 2-е издание*, Керниган и Ритчи) [Kernighan, 1988] является программой на C++. При изучении C++ поможет опыт использования любого языка со статическим определением типов.

1.6.1. Рекомендации для программистов на C

Чем лучше вы знаете C, тем труднее вам будет избежать программирования на C++ в стиле C, теряя при этом потенциальные преимущества C++. Просмотрите, пожалуйста, приложение Б, в котором описываются различия C и C++. Вот некоторые вещи, с которыми C++ справляется лучше, чем C:

[1] Макросы почти никогда не требуются в C++. Пользуйтесь ***const*** (§ 5.4) или ***enum*** (§ 4.8) для определения констант, ***inline*** (§ 7.1.1) во избежание накладных расходов на вызов функций, ***template*** (глава 13) для определения семейства функций или типов и ***namespace*** (§ 8.2) для предотвращения конфликтов имен.

[2] Не объявляйте переменную, пока она вам не потребуется, чтобы вы тут же могли инициализировать ее. Объявление можно поместить в любом месте, где допустима инструкция (§ 6.3.1), в разделе инициализации ***for-инструкции*** (§ 6.3.3) и в условиях (§ 6.3.2.1).

[3] Не используйте ***malloc***(). Оператор ***new*** (§ 6.2.6) делает то же самое лучше. Вместо ***realloc***() пользуйтесь ***vector*** (§ 3.8).

[4] Пытайтесь избегать ***void****, арифметических операций над указателями, объединений и приведений типов, за исключением разве что глубоко спрятанных в реализацию функций или классов. В большинстве случаев приведение типа означает ошибку на этапе проектирования. Если вам необходимо приведение, попробуйте воспользоваться одним из «новых преобразований типа» (§ 6.2.7), которое позволит выразить более точно то, что вы пытаетесь сделать.

[5] Сведите к минимуму использование массивов символов и строк C. Стандартные библиотеки классов C++ ***string*** (§ 3.5) и ***vector*** (§ 3.7.1) часто могут упростить программирование по сравнению с традиционным стилем C. В общем, не пытайтесь создавать то, что уже сделано в стандартной библиотеке.

Для обеспечения соглашений C о компоновке функция C++ должна быть объявлена соответствующим образом (§ 9.2.4).

Самое важное: пытайтесь представить программу в виде набора взаимодействующих понятий, реализованных в виде классов и объектов, а не в виде набора структур данных с функциями, перемалывающими их биты.

1.6.2. Рекомендации для программистов на C++

Многие люди пользовались C++ в течение длительного времени. Большинство из них используют C++ в одной среде и привыкли к ограничениям, полагавшимися ранними компиляторами и библиотеками первого поколения. Тем, чего не замечал опытный программист на C++ в течение этих лет, было не появление новых средств как таковых, а скорее изменения в отношениях между средствами языка, которые сделали возможными новые методы программирования. Другими словами, то, о чем вы не думали, когда изучали C++, или то, что вы сочли непрактичным, сегодня может оказаться исключительно полезным. Вы сможете познакомиться с этим только путем повторного просмотра основ.

Читайте главы по порядку. Если вы знакомы с содержанием главы, вам удастся пролистать ее за несколько минут. Если вы не знакомы с материалом, вам встретится что-нибудь неожиданное. Я кое-что узнал, пока писал эту книгу и подозреваю, что вряд ли существует программист на C++, который знает все представленные здесь

свойства и технологии. Более того, чтобы правильно пользоваться языком, вам необходима некая перспектива, которая привнесла бы порядок в набор средств и методов. Эта книга с ее организацией и примерами призвана выстроить такую перспективу.

1.7. Размышления о программировании на C++

В идеале вы разрабатываете программу в три этапа. Сначала вы добиваетесь ясного понимания задачи (анализ), затем определяете ключевые концепции, в терминах которых выражается решение задачи (проектирование), и, наконец, воплощаете решение в программе (программирование). Однако детали проблемы и некоторые аспекты решения часто становятся понятными только в процессе написания программы и при попытках заставить ее работать. Тогда-то и становится важным выбор языка.

В большинстве приложений существуют понятия, которые не просто представить в виде предопределенного типа или функции без связанных с ней данных. Если такое понятие есть, объявите класс, представляющий это понятие в программе. Класс в C++ — это тип. Он описывает поведение объектов этого класса: как они создаются, управляются и уничтожаются. Класс также может определять представление объектов, хотя на ранних стадиях проектирования это не должно быть главной заботой. Ключом к проектированию хороших программ является выделение классов, каждый из которых ясно описывает отдельное понятие. Часто это означает, что вы должны сосредоточиться на вопросах типа: «Как создаются объекты класса? Можно ли объекты этого класса копировать и/или уничтожать? Какие операции можно применять к объектам класса?». Если на эти вопросы нет хороших ответов, вероятно нет и ясного понимания самих понятий. Возможно, стоит еще раз подумать о задаче и предлагаемых решениях, а не кидаться программировать.

Легче всего работать с традиционными математическими понятиями: всевозможными числами, множествами, геометрическими фигурами и т. п. Текстовый ввод/вывод, строки, основные контейнеры, базовые алгоритмы, применяемые для обработки таких контейнеров, и некоторые математические классы являются частью стандартной библиотеки C++ (глава 3, § 16.1.2). Кроме того, существует огромное количество библиотек поддержки общих и узкоспециальных понятий.

Понятия существуют не в вакууме, они всегда объединяются в группы. Часто сложнее организовать отношения между классами в программе (то есть точно определить отношения между различными понятиями, используемыми в решении), чем отдельные классы. Лучше избегать путаницы, когда каждый класс (понятие) связан со всеми другими. Представьте два класса A и B. Отношения типа «A вызывает функцию из B», «A создает объекты B» и «объект типа B является членом A» редко вызывают серьезные проблемы, в то время как отношений типа «A использует данные B» обычно следует избегать.

Одним из мощнейших интеллектуальных инструментов преодоления сложности является иерархическое упорядочивание, то есть организация связанных понятий в древообразную структуру с наиболее общим понятием в корне. В C++ производные классы представляют именно такую структуру. Программа часто может быть организована в виде набора деревьев или ориентированных графов классов, не содержащих циклы. То есть программист определяет ряд базовых классов, каждый из которых имеет набор производных классов. Для определения операций с наиболее общими

понятиями (операций базовых классов) часто можно воспользоваться виртуальными функциями (§ 2.5.5, § 12.2.6). Затем в конкретных случаях интерпретация этих операций может быть переопределена в производных классах.

Иногда даже ориентированного графа без циклов недостаточно для организации понятий в программе. Некоторые понятия изначально взаимно зависимы. В таких ситуациях мы пытаемся локализовать циклические зависимости так, чтобы они не влияли на структуру программы в целом. Если вам не удается избавиться от таких зависимостей или локализовать их, то, похоже, вы находитесь в тупике, выбраться из которого не поможет ни один язык программирования. Если вам не удается добиться относительно просто формулируемых отношений между основными понятиями, программа имеет все шансы стать неуправляемой.

Одним из лучших способов развязывания зависимостей является отчетливое разграничение интерфейса и реализации. Основным инструментом для этого в C++ является абстрактный класс (§ 2.5.4, § 12.3).

Другая форма общности может быть выражена при помощи шаблонов (§ 2.7, глава 13). Шаблон класса определяет семейство классов. Например, шаблон списка определяется как «список объектов типа T», где «T» может быть любым типом. Таким образом, шаблон — это механизм создания типа, которому другой тип передается в качестве аргумента. Наиболее распространенными шаблонами являются классы контейнеров (такие как списки, массивы, ассоциативные массивы) и основные алгоритмы, использующие эти контейнеры. Выражение зависимости класса и связанных с ним функций от типа с использованием механизма наследования обычно является ошибкой. Лучше использовать шаблоны.

Помните, что большую часть работы можно просто и понятно выполнить с использованием только примитивных типов, структур данных, обычных функций и нескольких библиотечных классов. Не стоит использовать полный аппарат определения новых типов, если в этом нет действительной необходимости.

Вопрос «Как писать хорошие программы на C++?» напоминает вопрос «Как писать хорошую английскую прозу?». Есть два совета: «Знай, что хочешь сказать» и «Тренируйся. Подражай хорошему стилю». Оба совета годятся как для C++, так и для английской прозы, и им обоим одинаково сложно следовать.

1.8. Советы

Вот набор «правил», которые могут пригодиться при изучении C++. По мере продвижения вы сможете развить их в некую более осмысленную концепцию, соответствующую вашим задачам и вашему стилю программирования. Они намеренно упрощены и поэтому детали опущены. Не воспринимайте их буквально. Для написания хороших программ требуется ум, вкус и терпение. Не ожидайте, что у вас хорошо получится с первого раза. Экспериментируйте!

[1] В процессе программирования вы воплощаете свое решение некоторой задачи в конкретный код. Постарайтесь, чтобы структура программы отражала ваши идеи как можно более непосредственно:

- [a] Если вы думаете об «этом» как об отдельном понятии, оформите «это» в виде класса.
- [b] Если вы думаете об «этом» как об отдельной сущности, сделайте «это» объектом какого-нибудь класса.

[c] Если два класса имеют общий интерфейс, оформите этот интерфейс в виде абстрактного класса.
[d] Если реализации двух классов имеют нечто существенно общее, реализуйте это общее в виде базового класса.
[e] Если класс является контейнером объектов, сделайте из него шаблон.
[f] Если функция реализует алгоритм для контейнера, оформите ее в виде шаблона функции, выполняющего алгоритм для семейства контейнеров.
[g] Если классы, шаблоны и т. п. логически связаны между собой, поместите их в одно пространство имен.

[2] Какой бы класс вы не определяли (если только он не реализует математические сущности вроде матриц, комплексных чисел или низкоуровневые типы наподобие связанного списка):
[a] Не используйте глобальные данные (пользуйтесь членами).
[b] Не используйте глобальные функции.
[c] Не используйте открытые члены класса.
[d] Не используйте функций-друзей , разве что во избежание [a] или [c].
[e] Не создавайте в классе «поля типа» — пользуйтесь виртуальными функциями.
[f] Не применяйте встроенные функции, разве что для значительной оптимизации.

Более специфические или подробные правила вы найдете в советах в конце каждой главы. Помните, что это всего лишь рекомендации, а не непреложные законы. Советами нужно пользоваться там, где они применимы. Не существует замены уму, опыту, здравому смыслу и хорошему вкусу.

Я считаю, что правила типа «никогда не делай это» бесполезны. Поэтому большинство советов построено в форме предложений типа «что делать», а негативные утверждения выражены без оттенка абсолютного запрета. Я не знаю ни одного существенного средства C++, примеры качественного использования которого мне никогда бы не встречались. Раздел «Советы» не содержит объяснений. Вместо них за каждым советом следует ссылка на соответствующий раздел книги. Если совет выражен в негативной форме, в указанном разделе приведены позитивные альтернативы.

1.8.1. Литература

В тексте мало прямых ссылок; ниже приводится список книг и статей, упомянутых прямо или косвенно.

[Barton, 1994] John J. Barton and Lee R. Nackman: *Scientific and Engineering C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 1-201-53393-6.

[Berg, 1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38 No. 10. October 1995.

[Booch, 1994] Grady Booch: *Object-Oriented Analysis and Design*. Benjamin/Cummings. Menlo Park, Calif. 1994. ISBN 0-8053-5340-2.

[Budge, 1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C, 1990] X3 Secretariat: *Standard — The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association. Washington, DC, USA.

[C++, 1998] X3 Secretariat: *International Standard — The C++ Language.* X3J16-14882. Information Technology Council (NSITC). Washington, DC, USA.

[C++, 2003] X3 Secretariat: *International Standard — The C++ Language* (including the 2003 Technical Corrigendum). 14882:2003(E). Information Technology Council (NSITC). Washington, DC, USA.

[Campbell, 1987] Roy Campbell, et al.: *The Design of a Multiprocessor Operating System.* Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Coplien, 1995] James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design.* Addison-Wesley. Reading, Mass. 1995. ISBN 1-201-60734-4.

[Dahl, 1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language.* Norwegian Computing Center S-22. Oslo, Norway. 1970

[Dahl, 1972] O-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction in Structured Programming.* Academic Press, New York. 1972.

[Ellis, 1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Gamma, 1995] Erich Gamma, et al.: *Design Patterns.* Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-63361-2.

[Goldberg, 1983] A. Goldberg and D. Robson: *SMALLTALK-80 — The Language and Its Implementation.* Addison-Wesley. Reading, Mass. 1983.

[Griswold, 1970] R. E. Griswold, et al.: *The Snobol4 Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1970.

[Hamilton, 1993] G. Hamilton and P. Kougiouris: *The Spring Nucleus: A Microkernel for Object.* Proc. 1993 Summer USENIX Conference. USENIX.

[Griswold, 1983] R. E. Griswold and M. T. Griswold: *The ICON Programming Language.* Prentice-Hall. Engewood Cliffs, New Jersey. 1983.

[Henricson, 1997] Mats Henricson and Erik Nyquist: *Industrial Strength C++: Rules and Recommendations.* Prentice-Hall. Engewood Cliffs, New Jersey. 1997. ISBN 0-13-120965-5.

[Ichbian, 1979] Jean D. Ichbiah, et al.: *Rationale for the Design of the ADA Programming Language.* SIGPLAN Notices. Vol. 14 No. 6. June 1979.

[Kamath, 1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology.* AT&T Technocal Journal. Vol. 72 No. 5. September/October 1993.

[Kernighan, 1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan, 1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language (Second Edition).* Prentice-Hall. Englewood Cliffs, New Jersey. 1978. ISBN 0-13-110362-8.

[Koenig, 1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible — but no closer.* The C++ Report. Vol. 1 No. 7. July 1989.

[Koenig, 1997] Andrew Koenig and Barbara Moo: *Ruminations on C++*. Addison Wesley Longman. Reading, Mass. 1997. ISBN 1-201-42339-1.

[Knuth, 1968] Donald Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Mass.

[Liskov, 1979] Barbara Liskov et al.: *Clu Reference Manual*. MIT/LCS/TR-225. MIT Cambridge. Mass. 1979.

[Martin, 1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall. Englewood Cliffs, New Jersey. 1995. ISBN 0-13-203837-4.

[Orwell, 1949] George Orwell: *1984*. Secker and Warburg. London. 1949.

[Parrington, 1995] Graham Parrington et al.: *The Design and Implementation of Arjuna*. Computer Systems. Vol. 8 No. 3. Summer 1995.

[Richards, 1980] Martin Richards and Colin Whitby-Strevens: *BCPL — The Language and Its Compiler*. Cambridge University Press, Cambridge. England. 1980. ISBN 0-521-21965-5.

[Rosler, 1984] L. Rosler: *The Evolution of C — Past and Future*. AT&T Bell Laboratories Technical Journal. Vol. 63 No. 8. Part 2. October 1984.

[Rozier, 1988] M. Rozier, et al.: *CHORUS Distributed Operating Systems*. Computing Systems. Vol. 1 No. 4. Fall 1988.

[Sethi, 1981] Ravi Sethi: *Uniform Syntax for Type Expressions and Declarations*. Software Practice & Experience. Vol. 11. 1981.

[Stepanov, 1994] Alexander Stepanov and Meng Lee: The Standard Template Library. HP Labs Technical Report HPL-94-34 (R. 1). August, 1994.

[Stroustrup, 1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Mass. 1986. ISBN 0-201-12078-X.

[Stroustrup, 1987] Bjarne Stroustrup and Jonathan Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup, 1991] Bjarne Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Mass. 1991. ISBN 0-201-53992-6.

[Stroustrup, 1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-2-1-54330-3.

[Tarjan, 1983] Robert E. Tarjan: *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics. Philadelphia, Penn. 1983. ISBN 0-898-71187-8.

[Unicode, 1996] The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley Developers Press. Reading, Mass. 1996. ISBN 0-201-48345-9.

[UNIX, 1985] UNIX Time-Sharing System: *Programmer's Manual. Research Version, Tenth Edition*. AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

[Wilson, 1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge. Mass. 1996. ISBN 0-262-73118-5.

[Wikström, 1987] Eke Wikström: *Functional Programming Using ML*. Prentice-Hall. Englewood Cliffs, New Jersey. 1987.

[Woodward, 1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office. London. England. 1974.

Ссылки на литературу, посвященную проектированию и разработке больших программных систем, можно найти в конце главы 23.

Русские переводы:[1]

[Booch, 1994] Г. Буч. *Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е издание*. СПб. «Невский Диалект». 1998.

[Ellis, 1989] Б. Страуструп, М. А. Эллис. Справочное руководство по языку C++ с комментариями: проект стандарта ANSI. М. «Мир». 1992.

[Kernighan, 1988] Б. Керниган, Д. Ричи. Язык программирования Си. *3-е издание*. СПб. «Невский Диалект». 2001.

[Knuth, 1968] Д. Кнут. Искусство программирования для ЭВМ. т. 1, 2, 3. Москва. «Мир». 1976.

[Orwell, 1949] Дж. Оруэлл. 1984 (см. в сборнике: Дж. Оруэлл. Проза отчаяния и надежды. СПб. Лениздат. 1990).

[Stroustrup, 1986] Б. Страуструп. Язык программирования Си++. Москва. «Радио и связь». 1991.

[1] Далее при ссылках на литературу указаны номера страниц оригинальных изданий. — *Примеч. ред.*

2

Обзор C++

Первым делом давайте избавимся
от всех защитников языка.
— Генрих VI, часть II

Что такое C++? — парадигмы программирования — процедурное программирование — модульность — раздельная компиляция — обработка исключений — абстракция данных — типы, определяемые пользователем — конкретные типы — абстрактные типы — виртуальные функции — объектно-ориентированное программирование — обобщенное программирование — контейнеры — алгоритмы — язык и программирование — советы.

2.1. Что такое C++?

C++ — язык программирования общего назначения с уклоном в сторону системного программирования, который:

- лучше, чем C,
- поддерживает абстракцию данных,
- поддерживает объектно-ориентированное программирование,
- поддерживает обобщенное программирование.

В данной главе объясняется, что все это значит, без анализа тонких деталей определения языка. Цель главы — дать общее представление о C++ и его ключевых методах, а *не* детальную информацию, необходимую, чтобы начать программировать на C++.

Если некоторые части этой главы покажутся вам трудными для понимания — просто пропустите их и пробивайтесь дальше. Все будет объяснено подробно в последующих главах. Однако, если вы пропустили часть этой главы, будьте добры — возвратитесь к ней позднее.

Детальное понимание средств языка — даже *всех* средств языка — не может компенсировать отсутствия общего представления о языке и основных методах его использования.

2.2. Парадигмы программирования

Объектно-ориентированное программирование является техникой программирования — парадигмой для написания «хороших» программ, решающих различные задачи. Если термин «объектно-ориентированный язык программирования» вообще что-либо означает, он должен означать язык программирования, который предоставляет удобные механизмы поддержки объектно-ориентированного стиля программирования.

Отметим следующий существенный момент. Можно сказать, что язык *поддерживает* данный стиль, если он предоставляет средства, которые делают использование стиля удобным (достаточно простым, надежным и эффективным). Язык не поддерживает технику программирования, если для написания соответствующей программы требуются чрезмерные усилия либо мастерство. Такой язык просто *предоставляет возможности* для использования данной техники. Например, можно написать структурную программу на языке Fortran77 или объектно-ориентированную программу на C, но это неоправданно сложно, потому что упомянутые языки не поддерживают соответствующие техники непосредственно.

Поддержка парадигмы проявляется не только в наличии средств языка, позволяющих непосредственно использовать парадигму, но и (более тонко) в виде проверок в момент компиляции и/или выполнения на неумышленное отклонение от парадигмы. Наиболее очевидной иллюстрацией является проверка соответствия типов. Выявление неоднозначности и проверка во время выполнения также используются для расширения поддержки парадигмы. Дополнительные внеязыковые средства, такие как стандартные библиотеки и среды программирования, также могут существенно улучшить поддержку парадигмы.

Один язык, имеющий некоторое средство, не обязательно лучше другого языка, не имеющего его. Тому есть множество подтверждений. Важным является не то, какие в языке есть средства, а то, что средства, которыми он располагает, достаточны для поддержки соответствующего стиля программирования в требуемых прикладных областях:

[1] Все средства должны быть встроены в язык понятным и элегантным образом.

[2] Должна существовать возможность комбинирования средств для решения задач, которые в противном случае потребовали бы дополнительных, отдельных средств.

[3] Должно быть как можно меньше неестественных средств «специального назначения».

[4] Реализация средства не должна приводить к значительным накладным расходам в не использующих его программах.

[5] Пользователь не обязан знать ничего, кроме того подмножества языка, которое он явно применяет при написании программы.

Первый принцип апеллирует к эстетике и логике, два следующих отражают концепцию минимализма, а два последних могут быть сформулированы так: «то что вам не нужно, не должно мешать».

C++ создавался с целью добавления поддержки абстракции данных, объектно-ориентированного и обобщенного программирования к традиционному языку C с учетом указанных ограничений. *Не* подразумевалось принуждение всех пользователей к какому-либо конкретному стилю программирования.

В следующих разделах рассматриваются некоторые стили программирования и ключевые механизмы языка, необходимые для их поддержки. Описаны несколько методов, начиная с процедурного программирования и заканчивая иерархиями классов и объектно-ориентированным и обобщенным программированием с использованием шаблонов. Каждая парадигма строится на основе своих предшественниц, вносит что-нибудь свое в набор инструментов программиста и отражает соответствующий стиль проектирования.

Описание языковых средств не претендует на полноту. Акцент делается на подходах к проектированию и оптимизации программ, а не на деталях языка. На данной стадии гораздо важнее получить представление о том, *что* можно сделать при помощи C++, чем понять, *как* этого конкретно можно достигнуть.

2.3. Процедурное программирование

Вот исходная парадигма программирования:

> *Реши, какие требуются процедуры;*
> *используй наилучшие доступные алгоритмы.*

Акцент здесь делается на обработке — алгоритме, необходимом для выполнения требуемых вычислений. Языки поддерживают эту парадигму, предоставляя средства для передачи аргументов функциям и возврата значений из функций. Литература, имеющая отношение к такому образу мыслей, пестрит обсуждениями способов передачи аргументов, различий между разными видами аргументов, описаниями разновидностей функций (процедуры, подпрограммы, макросы...) и т. п.

Типичным примером «хорошего стиля» является функция извлечения квадратного корня. Ей передается в качестве аргумента число с плавающей точкой двойной точности — она возвращает результат. При этом выполняются всем понятные математические вычисления:

```
double sqrt(double arg)
{
    // код, вычисляющий квадратный корень
}

void f()
{
    double root2 = sqrt(2);
    //...
}
```

В C++ фигурные скобки { } выражают группировку. В примере они обозначают начало и конец тела функции. Двойная наклонная черта // означает начало комментария, который продолжается до конца строки. Ключевое слово ***void*** означает, что функция не возвращает значения.

С точки зрения организации программы, функции используются для наведения порядка в хаосе алгоритмов. Алгоритмы сами по себе записываются с использованием вызовов функций и других средств языка. Следующие подразделы содержат описания основных средств C++, предоставляемых для организации вычислений.

2.3.1. Переменные и арифметические операции

Каждое имя и каждое выражение имеет тип, определяющий набор допустимых операций, которые можно выполнить с выражением. Например, объявление

```
int inch;
```

указывает, что *inch* имеет тип *int*; то есть *inch* — переменная целого типа.

Объявление — это инструкция (statement), которая вводит имя в программе. Объявление указывает тип имени. *Тип* определяет использование имени или выражения.

C++ предлагает множество встроенных типов, которые непосредственно связаны с архитектурой компьютера. Например:

```
bool    // логическая величина,допустимые значения — истина или ложь
char    // символ, например 'f', 'я' или '9'
int     // целое число, например 1, 42 или 1216
double  // вещественное число с плавающей точкой
        // двойной точности, например 3.14 или 299793.0
```

Переменная типа ***char*** имеет размер, необходимый на данном компьютере для хранения одного символа (как правило, один байт). Переменная типа ***int*** имеет размер, необходимый для выполнения целочисленных арифметических операций (обычно слово).

Арифметические операторы можно использовать с любой комбинацией этих типов:

```
+   // плюс, как унарный, так и бинарный
-   // минус, как унарный, так и бинарный
*   // умножение
/   // деление
%   // остаток от деления
```

Это же относится к операторам сравнения:

```
==  // равно
!=  // не равно
<   // меньше
>   // больше
<=  // меньше или равно
>=  // больше или равно
```

При присваивании и в арифметических операциях C++ производит осмысленные преобразования типов для обеспечения их совместного использования:

```
void some_function ()       // функция, не возвращающая значение
{
    double d = 2.2;         // присвоить начальное значение
                            // переменной с плавающей точкой
    int i = 7;              // присвоить начальное значение целой переменной
    d = d + i;              // присвоить сумму переменной d
    i = d * i;              // присвоить произведение переменной i
}
```

Также, как и в языке C, = означает оператор присваивания, а == — проверку равенства.

2.3.2. Условия и циклы

C++ предоставляет обычный набор инструкций для реализации ветвления и циклов. Вот пример функции, которая выводит приглашение к вводу и возвращает логическое значение, зависящее от ответа пользователя:

```
bool accept ()
{
    cout << "Будете продолжать - y (да) или n (нет)?\n";      // вывести вопрос
```

```
    char answer = 0;
    cin >> answer;                          // считать ответ

    if (answer == 'y') return true;
    return false;
}
```

Оператор << (вывести) используется в качестве оператора вывода; ***cout*** — поток стандартного вывода. Оператор >> (прочесть) используется в качестве оператора ввода; ***cin*** — поток стандартного ввода. Тип операнда справа от >> определяет, какой ожидается ввод, а сам операнд принимает ввод. Символ \n в конце означает переход на новую строку.

Пример можно немного улучшить так, чтобы пользователь не обязан был нажимать только клавишу *y* или *n*:

```
bool accept2 ()
{
    cout << "Будете продолжать – y (да) или n (нет)?\n";   // вывести вопрос

    char answer = 0;
    cin >> answer;                                         // считать ответ

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "Ответ считается отрицательным.\n";
        return false;
    }
}
```

Инструкция ***switch*** сравнивает значение с набором констант. Константы после ключевых слов ***case*** должны отличаться друг от друга. Если проверяемое значение не равно ни одной из них, выбирается ***default***. Присутствие ***default*** не является обязательным.

Редкие программы пишутся без использования циклов. В нашем примере мы могли бы предоставить пользователю несколько попыток:

```
bool accept3 ()
{
    int tries = 1;                                  // номер попытки
    while (tries < 4) {
        cout << "Будете продолжать – y или n?\n";   // вывести вопрос

        char answer = 0;
        cin >> answer;                              // считать ответ

        switch (answer) {
        case 'y':
            return true;
        case 'n':
            return false;
        default:
```

```
            cout << "Извините, я Вас не понял.\n";
            tries = tries + 1;
        }
    }
    cout << "Ответ считается отрицательным.\n";
    return false;
}
```

Инструкция ***while*** выполняется до тех пор, пока условие в скобках не примет значение «ложь».

2.3.3. Указатели и массивы

Массив можно объявить следующим образом:

```
char v[10];      // массив из 10 символов
```

Аналогично, указатель можно объявить следующим образом:

```
char* p;         // указатель на символ
```

В объявлениях [] означает «массив», а * — «указатель на». У всех массивов нижняя граница — ***0***. Таким образом, в ***v*** содержится десять элементов, ***v[0]...v[9]***. Указатель может содержать адрес объекта соответствующего типа:

```
p = &v[3];       // p указывает на четвертый элемент
```

Унарный оператор & означает получение адреса операнда.

Рассмотрим копирование десяти элементов из одного массива в другой:

```
void another_function ()
{
    int v1[10];
    int v2[10];
    // ...
    for (int i=0; i<10; ++i) v1[i]=v2[i];
}
```

Эту ***for-инструкцию*** можно прочитать как «присвой ***i*** ноль; пока ***i*** меньше ***10***, скопируй ***i***-й элемент и увеличь ***i***». Если оператор инкремента ++ применяется к целой переменной, он просто увеличивает ее на единицу.

2.4. Модульное программирование

С течением времени акцент при разработке программ сместился от проектирования процедур в сторону организации данных. Помимо прочего, это явилось отражением факта увеличения размеров программ. Набор связанных процедур вместе с данными, которые они обрабатывают, часто называют *модулем*. Парадигмой программирования становится:

> *Реши, какие требуются модули;*
> *разбей программу так, чтобы скрыть данные в модулях.*

Эта парадигма также известна как «принцип сокрытия данных». Там, где не требуется группировка процедур вместе с данными, достаточно процедурного стиля программирования. Техника проектирования «хороших процедур» теперь применяется для каждой процедуры в модуле. Весьма типичным примером модуля является определение стека. Здесь нужно решить следующие главные задачи:

[1] Предоставить пользовательский интерфейс для стека (например, функции ***push*** () и ***pop*** (), помещающие данные в стек и извлекающие их оттуда).

[2] Гарантировать, что представление стека (например, в виде массива элементов) доступно только через этот пользовательский интерфейс.

[3] Обеспечить инициализацию стека до первого использования.

C++ предоставляет механизм группировки связанных данных, функций и т. д. в пространства имен (namespace). Например, пользовательский интерфейс модуля ***Stack*** может быть объявлен и использован так:

```
namespace Stack {                    // интерфейс
        void push (char);
        char pop ();
}

void f()
{
        Stack::push ('c');
        if (Stack::pop () != 'c') error ("Такое невозможно!");
}
```

Квалификатор ***Stack***:: означает, что ***push*** () и ***pop*** () берутся из пространства имен ***Stack***. Использование таких же имен в другом месте не приведет к путанице.

Определение стека может быть выполнено в раздельно компилируемой части программы:

```
namespace Stack {                    // реализация
        const int max_size = 200;
        char v[max_size];
        int top = 0;

        void push (char c)      { /* проверить на переполнение и поместить c в стек*/ }
        char pop ()             { /* проверить, не пуст ли стек, и извлечь символ из стека */ }
}
```

Ключевым в этом модуле ***Stack*** является то, что пользовательский код отделен от способа представления данных в модуле ***Stack*** и способа реализации ***Stack::push*** () и ***Stack::pop*** (). Пользователю нет необходимости знать, что ***Stack*** реализован при помощи массива. Таким образом, реализация ***Stack*** может быть изменена, причем так, что это никак не повлияет на программу пользователя.

Так как данные — это только часть того, что хочется «спрятать», понятие «сокрытие данных» тривиальным образом расширяется до понятия «сокрытие информации». А именно, имена переменных, констант, функций и типов также могут быть сделаны локальными в модуле. Соответственно, C++ позволяет поместить любое объявление в пространство имен (§ 8.2).

Модуль ***Stack*** демонстрирует один из способов представления стека. В последующих разделах используется множество стеков для иллюстрации различных стилей программирования.

2.4.1. Раздельная компиляция

C++ поддерживает соглашения C о раздельной компиляции. Это можно использовать для организации программы в виде почти независимых частей. Как правило, мы помещаем объявления, которые описывают интерфейс модуля, в файл с характерным именем, отражающим его использование. Так, интерфейс

```
namespace Stack {          // интерфейс
    void push (char);
    char pop ();
}
```

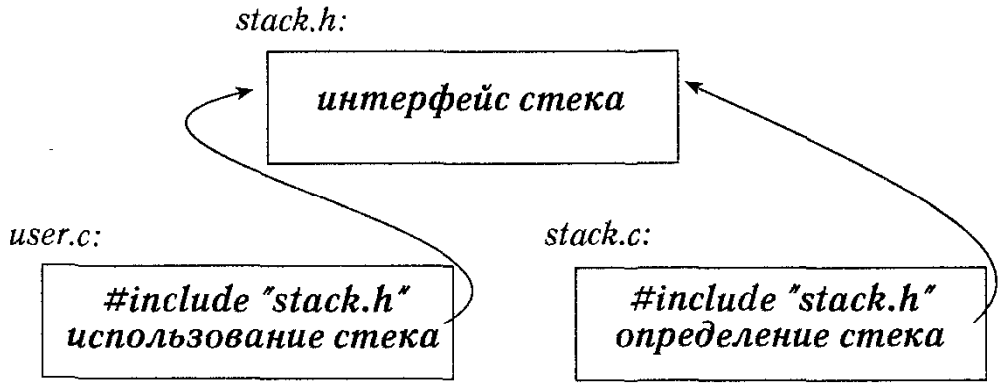
будет помещен в файл ***stack.h***. Пользователи смогут *включить* (***#include***) этот файл, называемый *заголовочным файлом*, следующим образом:

```
#include "stack.h"          // включить интерфейс

void f()
{
    Stack::push ('c');
    if (Stack::pop () != 'c') error ("Такое невозможно!");
}
```

Чтобы помочь компилятору, файл, содержащий реализацию модуля ***Stack***, также включит интерфейс (этот файл может называться, например, ***stack.c***):

```
#include "stack.h"          // включить интерфейс

namespace Stack {           // представление
    const int max_size = 200;
    char v[max_size];
    int top = 0;
}

void Stack::push (char c)   { /* проверить на переполнение и поместить с в стек*/ }

char Stack::pop ()          { /* проверить, не пуст ли стек, и извлечь символ из стека*/ }
```

Код пользователя находится в третьем файле, скажем, ***user.c***. Тексты ***user.c*** и ***stack.c*** совместно используют информацию об интерфейсе, содержащуюся в ***stack.h***; во всем другом эти два файла независимы и могут быть раздельно откомпилированы. Графически, упомянутые фрагменты программы можно представить следующим образом:



Раздельная компиляция имеет значение во всех реальных программах. Это не просто способ представления средств, таких как ***Stack***, в виде модулей. Строго говоря, использование раздельной компиляции не является элементом языка, а относится к

вопросам конкретной его реализации. Однако, это вопрос большого практического значения. Наилучшим подходом является максимальное использование модульности, выражение этой модульности средствами языка и затем ее физическое представление в файлах для эффективной раздельной компиляции (главы 8, 9).

2.4.2. Обработка исключений

Когда программа разработана в виде набора модулей, обработка ошибок должна рассматриваться в свете этих модулей. Какой модуль несет ответственность за обработку тех или иных ошибок? Часто модуль, обнаруживший ошибку, не знает, какие действия нужно предпринять. Действия по восстановлению скорее зависят от модуля, вызвавшего операцию, чем от модуля, который обнаружил ошибку, пытаясь выполнить эту операцию. По мере роста программ и, особенно, когда интенсивно используются библиотеки, стандарты по обработке ошибок (или, в более широком смысле, «исключений») становятся очень важными.

Рассмотрим снова пример ***Stack***. Какие действия нужно предпринять, когда мы пытаемся поместить в стек (push) слишком много символов (то есть в случае переполнения)? Автор модуля ***Stack*** не знает, что предпочел бы пользователь в этой ситуации, а пользователь не может сам обнаружить проблему (если бы мог, то не возникло бы никакого переполнения). Решением для автора ***Stack*** является обнаружение переполнения стека и сообщение о нем (неизвестному) пользователю. Тогда пользователь может предпринять необходимые действия. Например:

```
namespace Stack {          // интерфейс
        void push (char);
        char pop ();

        class Overflow { };    // тип, представляющий исключение, связанное с переполнением
}
```

При обнаружении переполнения ***Stack::push*** () может вызвать код обработки исключений, то есть «генерирует (throw) исключение ***Overflow***»:

```
void Stack::push (char c)
{
        if (top == max_size) throw Overflow ();
        // поместить с в стек
}
```

throw передает управление обработчику исключений типа ***Stack::Overflow*** в некоторой функции, которая прямо или косвенно вызвала ***Stack::push*** (). Для воплощения этого механизма реализация «раскрутит» стек вызовов функций для восстановления контекста вызывающей процедуры. Таким образом, ***throw*** работает как многоуровневая инструкция ***return***. Например:

```
void f()
{
        //...
        try {  // с возникающими здесь исключениями разбирается
               // обработчик, определенный ниже
               while (true) Stack::push ('c');
        }
```

```
        catch (Stack::Overflow) {
            // ошибка: переполнение стека; предпримем надлежащие действия
        }
        //...
    }
```

Цикл ***while*** в ***try***-блоке пытается выполняться «вечно». Соответственно, после очередного вызова, функция ***Stack::push*** () сгенерирует(***throw***) исключение ***Overflow***, и управление будет передано ***catch***-блоку, предоставляющему обработчик ***Stack::Overflow***.

Использование механизма обработки исключений делает реакцию на ошибки более регулярной и понятной. Подробное обсуждение см. в § 8.3 и главе 14.

2.5. Абстракция данных

Модульность — фундаментальный аспект всех успешно работающих крупных систем. Она остается в центре внимания при обсуждении вопросов проектирования на протяжении всей книги. Однако, модули в той форме, как они описаны выше, недостаточны для ясного представления сложных систем. Ниже я сначала излагаю способ использования модулей для представления некой формы типов данных, определяемых пользователем, а затем показываю, как обойти ряд проблем этого подхода, непосредственно определяя пользовательские типы.

2.5.1. Модули, определяющие типы

Модульное программирование приводит к централизации управления данными определенного типа в отдельном модуле. Например, если мы хотим иметь много стеков, а не один, обеспечиваемый модулем ***Stack*** в примерах выше, мы могли бы описать менеджер стеков с интерфейсом следующего вида:

```
namespace Stack {
    struct Rep;                     // определение стека находится где-то в другом месте
    typedef Rep& stack;

    stack create ();                // создает новый стек
    void destroy (stack s);         // удаляет стек s

    void push (stack s, char c);    // помещает c в s
    char pop (stack s);             // извлекает символ из s
}
```

Объявление

```
struct Rep;
```

говорит, что ***Rep*** (представление) является именем типа, но определение типа оставляет на потом (§ 5.7). Объявление

```
typedef Rep& stack;
```

дает имя ***stack*** «ссылке на ***Rep***» (подробности см. в § 5.5). Идея заключается в том, что тип стека может задаваться с помощью ***Stack::stack***, а детали реализации скрыты от пользователей.

Stack::stack ведет себя почти как переменная встроенного типа:

```
struct Bad_pop { };                    // неудачное извлечение данных из стека
void f()
{
        Stack::stack s1 = Stack::create ();   // создает новый стек
        Stack::stack s2 = Stack::create ();   // создает еще один новый стек

        Stack::push (s1, 'c');
        Stack::push (s2, 'k');

        if (Stack::pop (s1) != 'c') throw Bad_pop ();
        if (Stack::pop (s2) != 'k') throw Bad_pop ();

        Stack::destroy (s1);
        Stack::destroy (s2);
}
```

Мы могли реализовать этот ***Stack*** несколькими способами. Важно, что пользователю нет необходимости знать, как мы это сделали. Пользователя не волнуют наши решения касательно изменения реализации ***Stack***, пока не затронут его интерфейс.

Реализация стека может заранее создать несколько стеков и по ***Stack::create*** () возвращать ссылку на неиспользованный еще стек. ***Stack::destroy*** () может помечать конкретный стек как неиспользуемый, чтобы последующие ***Stack::create*** () снова могли воспользоваться им:

```
namespace Stack {                      // представление
        const int max_size = 200;      // максимальный размер стека

        struct Rep {
                char v[max_size];
                int top;               // вершина стека
        };

        const int max = 16;            // максимальное количество стеков

        Rep stacks[max];               // представления стеков создаются заранее
        bool used[max];                // used[i] == true, если stacks[i] используется

        typedef Rep& stack;
}

void Stack::push (stack s, char c)     { /* проверить стек s на переполнение
                                            и записать символ c */ }

char Stack::pop (stack s)   { /* проверить, не пуст ли стек s, и вернуть символ */ }

Stack::stack Stack::create ()
{
        // выбрать неиспользуемый Rep, пометить его как используемый,
        // инициализировать и вернуть ссылку на него
}

void Stack::destroy (stack s) { /* пометить стек s как неиспользуемый */ }
```

Что мы сделали? Мы заключили тип представления в оболочку из интерфейсных функций. Как ведет себя получившийся «стековый тип» зависит частично от того, как мы определили эти интерфейсные функции, частично от того, как мы предоставляем тип представления пользователям ***Stack***, и, наконец, от проектирования самого типа представления.

Очень часто такое решение далеко от идеала. Существенной проблемой является то, что предоставление таких «псевдотипов» пользователям может очень сильно зависеть от деталей типа представления, и пользователи вынуждены их знать. Например, если бы мы выбрали для определения стека более сложную структуру данных, значительно изменились бы правила присваивания и инициализации для ***Stack::stacks***. Иногда это может быть и к лучшему. Однако, это демонстрирует то, что мы просто перенесли проблему создания хороших стеков из модуля ***Stack*** в тип представления ***Stack::stack***.

Еще важнее то, что определяемые пользователем типы, реализованные посредством модуля и предоставляющие доступ к реализации, ведут себя не так, как встроенные типы, и имеют по сравнению с ними иную (меньшую) поддержку. Например, время, в течение которого можно использовать ***Stack::Rep***, контролируется ***Stack::create*** () и ***Stack::destroy*** (), а не обычными правилами языка.

2.5.2. Типы, определяемые пользователем

C++ стремится решить задачу, позволяя пользователю непосредственно определять типы, которые ведут себя (почти) также, как и встроенные. Такой тип часто называют *абстрактным типом данных*. Я предпочитаю термин *тип, определяемый пользователем (пользовательский тип)*. Достаточно точное определение абстрактного типа данных потребовало бы «абстрактной» математической формулировки. При ее наличии то, что здесь называется *типами*, служило бы конкретными примерами таких истинно абстрактных сущностей. Парадигма программирования становится такой:

> *Реши, какие требуются типы;*
> *обеспечь полный набор операций для каждого типа.*

Везде, где не нужно более одного объекта определенного типа, достаточно стиля программирования с сокрытием данных при помощи модулей.

Арифметические типы, такие как дробные и комплексные числа, являются наиболее распространенными примерами типов, определяемых пользователем. Пример:

```
class complex {
        double re, im;
public:
        // создать комплексное число из двух вещественных
        complex (double r, double i) { re = r; im = i; }
        // создать комплексное число из одного вещественного
        complex (double r) { re = r; im = 0; }
        // создать комплексное число по умолчанию (0, 0)
        complex () { re = im = 0; }

        friend complex operator+ (complex, complex);
        friend complex operator- (complex, complex);        // бинарная операция
        friend complex operator- (complex);                 // унарная операция
        friend complex operator* (complex, complex);
        friend complex operator/ (complex, complex);

        friend bool operator== (complex, complex);          // проверка на равенство
        friend bool operator!= (complex, complex);          // проверка на неравенство
        //...
};
```

Объявление класса (то есть типа, определяемого пользователем) ***complex*** описывает представление комплексного числа и набор операций над комплексными числами. Представление является *закрытым* (private); то есть доступ к ***re*** и ***im*** имеют только функции, указанные внутри объявления класса ***complex***. Сами функции можно определить примерно так:

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re + a2.re, a1.im + a2.im);
}
```

Функция-член с тем же самым именем, что и класс, называется *конструктором*. Конструктор определяет способ инициализации объекта класса. Класс ***complex*** предоставляет три конструктора. Один преобразует вещественное число в комплексное. Второй — создает комплексное число из пары вещественных, а третий — создает комплексное число со значением по умолчанию.

Классом ***complex*** можно пользоваться следующим образом:

```
void f(complex z)
{
    complex a = 2.3;
    complex b = 1/a;
    complex c = a+b*complex(1, 2.3);
    // ...
    if (c != b) c = -(b/a)+2*b;
}
```

Компилятор преобразует операторы, работающие с комплексными числами, в соответствующие вызовы функций. Например, ***c!=b*** означает ***operator!=(c,b)***, а ***1/a*** означает ***operator/(complex(1), a)***.

Большинство модулей (но не все) лучше представлять в виде типов, определяемых пользователем.

2.5.3. Конкретные типы

Типы, определяемые пользователем, можно разрабатывать для реализации широкого набора задач. Рассмотрим определяемый пользователем тип ***Stack*** по аналогии с типом ***complex***. Чтобы сделать пример более реалистичным, этот тип ***Stack*** получает в качестве аргумента количество элементов в стеке:

```
class Stack {
    char* v;
    int top;
    int max_size;
public:
    class Underflow { };     // исключение (стек пуст)
    class Overflow { };      // исключение (переполнение)
    class Bad_size { };      // исключение (неправильный размер)

    Stack(int s);            // конструктор
    ~Stack();                // деструктор

    void push(char c);
    char pop();
};
```

Конструктор ***Stack*** (***int***) будет вызываться каждый раз при создании нового объекта класса. Он берет на себя заботу об инициализации. Если требуется какая-либо очистка памяти, когда объект класса выходит за пределы области видимости, объявляется логическое дополнение конструктора — *деструктор:*

```
Stack::Stack(int s)          // конструктор
{
    top = 0;
    if (10000<s) throw Bad_size();
    max_size = s;
    v = new char[s];         // разместить элементы в свободной памяти
                             // (куче, динамической памяти)
}

Stack::~Stack()              // деструктор
{
    delete[] v;              // освободить элементы для возможного
                             // последующего использования их памяти (§ 6.2.6)
}
```

Конструктор инициализирует новую переменную типа ***Stack***. Для этого он выделяет место в свободной памяти (называемой также *кучей* или *динамической памятью*) с помощью оператора ***new***. Деструктор освобождает эту память. Все происходит без вмешательства пользователей ***Stack***. Пользователи просто создают и используют ***Stack*** практически также, как переменные встроенных типов. Например:

```
Stack s_var1(10);                // глобальный стек из десяти элементов

void f(Stack& s_ref, int i)      // ссылка на Stack
{
    Stack s_var2[i];             // локальный стек из i элементов
    Stack* s_ptr = new Stack(20); // указатель на Stack,
                                 // размещенный в свободной памяти

    s_var1.push('a');
    s_var2.push('b');
    s_ref.push('c');
    s_ptr->push('d');
    //...
}
```

Этот тип ***Stack*** подчиняется тем же правилам именования, области видимости, создания, времени жизни, копирования и т. д., что и встроенные типы, такие как ***int*** и ***char***.

Естественно, функции-члены ***push***() и ***pop***() тоже должны быть где-то определены:

```
void Stack::push(char c)
{
    if(top == max_size) throw Overflow();
    v[top] = c;
    top = top + 1;
}

char Stack::pop()
```

```
{
        if (top == 0) throw Underflow ();
        top = top - 1;
        return v[top];
}
```

Типы, подобные ***complex*** и ***Stack***, называются *конкретными типами*, в противовес *абстрактным типам*, у которых интерфейс в еще большей степени изолирует пользователя от деталей реализации.

2.5.4. Абстрактные типы

При превращении ***Stack*** из «псевдотипа», реализованного в модуле (§ 2.5.1), в настоящий тип (§ 2.5.3), одно свойство было потеряно. Представление не отделено от пользовательского интерфейса; скорее оно является частью того, что будет включено во фрагмент программы, использующей стеки. Представление закрыто, и поэтому доступ к нему возможен только через функции-члены, но оно все же присутствует. Если представление сколь-нибудь значительно изменится, пользователь должен будет перекомпилировать программу. Это является платой за то, что конкретные типы ведут себя в точности как встроенные. В частности, у нас не может быть подлинных локальных переменных, если мы не знаем размера их представления.

Для типов, которые редко изменяются, и в тех случаях, когда локальные переменные обеспечивают достаточную ясность и эффективность, такой способ приемлем и часто является идеальным решением. Однако, если мы хотим полностью изолировать пользователей стека от изменений его реализации, предыдущее определение ***Stack*** не годится. Решением является отделение интерфейса от представления и отказ от подлинных локальных переменных.

Во-первых, определим интерфейс:

```
class Stack {
public:
        class Underflow { };    // тип исключения
        class Overflow { };     // тип исключения

        virtual void push (char c) =0;
        virtual char pop () =0;
};
```

Слово ***virtual*** (виртуальный) в языках Simula и C++ означает «может быть замещено позднее в классе, производном от этого». Класс, производный от ***Stack***, обеспечивает реализацию интерфейса ***Stack***. Любопытный синтаксис ***=0*** означает, что некоторый производный от ***Stack*** класс *должен* определить эту функцию. Таким образом, этот ***Stack*** может служить в качестве интерфейса любого класса, который реализует его функции ***push*** () и ***pop*** ().

Приведенное определение ***Stack*** можно использовать следующим образом:

```
void f(Stack& s_ref)
{
        s_ref.push ('c');
        if (s_ref.pop () != 'c') throw Bad_pop ();
}
```

Обратите внимание на то, как *f*() использует интерфейс ***Stack***, находясь в полном неведении относительно деталей реализации. Класс, обеспечивающий интерфейс для множества других классов, называется *полиморфным типом*.

Неудивительно, что реализация может состоять из всего того, что есть в конкретном классе ***Stack***, но оставлено за бортом интерфейса ***Stack***:

```
class Array_stack : public Stack { // Array_stack реализует Stack
        char* p;
        int max_size;
        int top;
public:
        Array_stack(int s);
        ~Array_stack();

        void push(char c);
        char pop();
};
```

: ***public*** в верхней строке можно читать как «производный от», «реализует» и «является подтипом».

Для того чтобы функция, подобная *f*(), могла использовать ***Stack***, не зная о деталях реализации, некоторая другая функция должна создать объект, с которым *f*() может работать. Например:

```
void g()
{
        Array_stack as(200);
        f(as);
}
```

Так как *f*() не знает об ***Array_stack***, а знает только интерфейс ***Stack***, она будет работать одинаково хорошо с различными реализациями ***Stack***. Например:

```
class List_stack : public Stack {     // List_stack реализует Stack
        list<char> lc;                // список символов
                                      // из стандартной библиотеки (§ 3.7.3)
public:
        List_stack() { }

        void push(char c) { lc.push_front(c); }
        char pop();
};

char List_stack::pop()
{
        char x = lc.front();          // получить первый элемент
        lc.pop_front();               // удалить первый элемент
        return x;
}
```

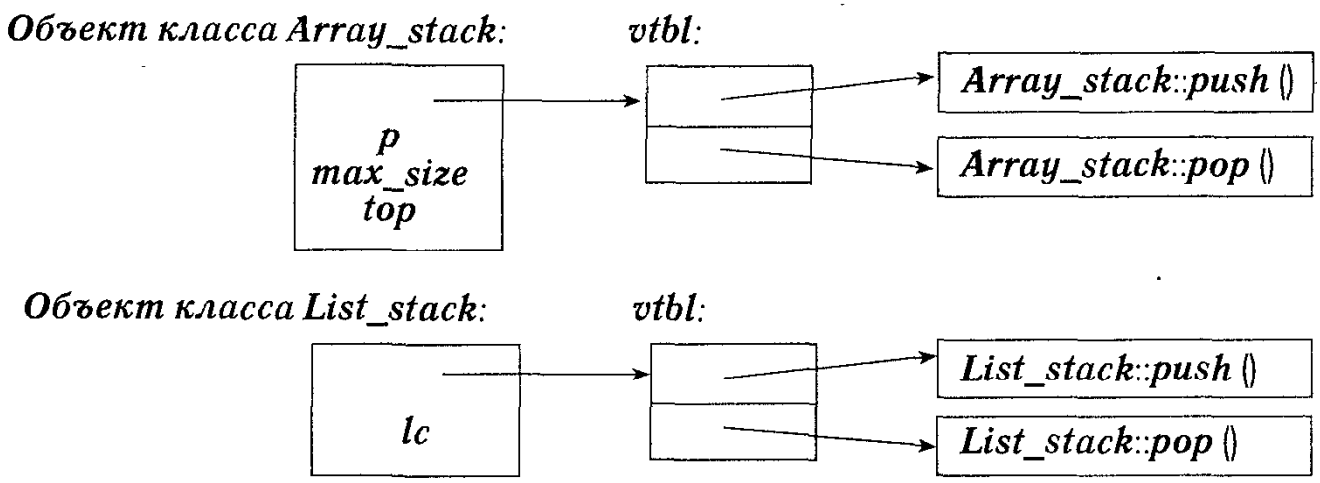
Здесь стек реализован с помощью списка символов. Функция-член ***lc.push_front***(*c*) добавляет *c* в качестве первого элемента ***lc***; вызов ***lc.pop_front***() удаляет первый элемент, а ***lc.front***() извлекает первый элемент ***lc***.

Теперь любая функция может создать ***List_stack*** и вызвать *f*() для работы с ним:

```
void h()
{
        List_stack ls;
        f(ls);
}
```

2.5.5. Виртуальные функции

Как вызов *s_ref.pop* () в *f* () связывается с подходящим определением функции? Ведь когда *f* () вызывается из *h* (), должна быть вызвана *List_stack::pop* (). Когда же *f* () вызывается из *g* (), должна быть вызвана *Array_stack::pop* (). Для правильного разрешения имен объект *Stack* должен содержать информацию, которая указывает, какая именно функция должна быть вызвана во время исполнения. Обычно реализации компиляторов пользуются технологией преобразования имени виртуальной функции в индекс в таблице, содержащей указатели на функции. Такая таблица часто называется «таблицей виртуальных функций (virtual function table)» или просто *vtbl*. Каждый класс с виртуальными функциями имеет свою собственную *vtbl*, идентифицирующую его виртуальные функции. Это можно изобразить графически следующим образом:



Функции в *vtbl* позволяют корректно использовать объект даже в тех случаях, когда ни размер объекта, ни расположение его данных не известны в месте вызова. Единственное, что нужно знать вызывающей стороне — это положение *vtbl* в *Stack* и индекс каждой виртуальной функции. Этот механизм виртуальных вызовов может быть реализован практически так же эффективно, как и «нормальный вызов функции». Требования по памяти составляют один указатель на каждый объект класса с виртуальными функциями, плюс одна *vtbl* для каждого такого класса.

2.6. Объектно-ориентированное программирование

Абстракция данных является фундаментальным аспектом качественного проектирования и будет оставаться в центре внимания при обсуждении вопросов проектирования на протяжении всей книги. Однако типы, определяемые пользователем, сами по себе недостаточно гибки, чтобы удовлетворить наши потребности. В данном разделе сначала демонстрируется проблема, связанная с простым типом, определяемым пользователем, а затем показывается, как ее решить с помощью иерархии классов.

2.6.1. Проблемы, связанные с конкретными типами

Конкретный тип, например «псевдотип», определенный с помощью модуля, описывает нечто вроде «черного ящика». После того, как черный ящик определен, он практически не взаимодействует с остальной частью программы. Нет иного способа приспособить конкретный тип к нуждам других пользователей, кроме как изменить его определение. Эта ситуация может выглядеть идеальной, но она также может привести к черезвычайно негибкой системе. Рассмотрим определение типа ***Shape*** (фигура) для использования в графической системе. Предположим, у нас есть два класса:

```
class Point { /* ... */ }; // Точка
class Color { /* ... */ }; // Цвет
```

Символы /* и */ означают начало и конец комментариев. Такую нотацию можно использовать для многострочных комментариев и комментариев, которые оканчиваются в середине строки, то есть справа от них может идти код.

Мы можем определить ***Shape*** следующим образом:

```
enum Kind { circle, triangle, square };    // перечисление (§ 4.8)
                                           // круг, треугольник, квадрат
class Shape {
        Kind k;                    // поле типа (какая фигура?)
        Point center;              // центр фигуры
        Color col;                 // цвет фигуры
        // ...
public:
        void draw ();              // нарисовать
        void rotate (int);         // повернуть
        // ...
};
```

«Поле типа» ***k*** необходимо, чтобы такие операции, как ***draw*** () и ***rotate*** (), могли определить, с каким видом фигуры они имеют дело (в Pascal-подобном языке можно использовать вариантную запись с тэгом ***k***). Функцию ***draw*** () можно определить следующим образом:

```
void Shape::draw ()
{
        switch (k) {
        case circle:
                // нарисовать окружность
                break;
        case triangle:
                // нарисовать треугольник
                break;
        case square:
                // нарисовать квадрат
                break;
        }
}
```

Как видите, все смешалось: такие функции, как ***draw*** (), должны «знать» обо всех возможных видах фигур. Поэтому код любой такой функции растет с добавлением новой

фигуры в систему. Если мы определили новую фигуру, каждую операцию над фигурой нужно просмотреть и (вероятно) модифицировать. У нас есть возможность добавить новую фигуру к системе, только если мы имеем исходные тексты каждой функции. Так как добавление новой фигуры связано с внесением изменений в код каждой важной операции над фигурами, оно требует большого мастерства и потенциально влечет появление ошибок в коде, управляющем другими (старыми) фигурами. Кроме того, выбор представления той или иной фигуры может быть сильно ограничен, так как требуется, чтобы (по крайней мере некоторые) представления вписывались в фиксированные границы, заданные определением общего типа ***Shape***.

2.6.2. Иерархия классов

Проблема в том, что не видно разницы между общими свойствами всех фигур (все они имеют цвет, их можно нарисовать и т. д.) и особыми свойствами фигур определенного типа (у окружности есть радиус, она вычерчивается специфическим способом и т. д.). Выражение этой разницы и использование вытекающих отсюда преимуществ составляет основу объектно-ориентированного программирования. Языки, располагающие конструкциями, которые позволяют выразить и использовать указанную разницу, поддерживают объектно-ориентированное программирование. Другие языки — нет.

Механизм наследования (позаимствованный в C++ из Simula) предоставляет следующее решение. Сначала мы описываем класс, который определяет общие свойства всех фигур:

```
class Shape {
        Point center;
        Color col;
        // ...
public:
        Point where () { return center; }
        void move (Point to) { center = to; /* ... */ draw (); }

        virtual void draw () =0;
        virtual void rotate (int angle) =0;
        // ...
};
```

Как и в абстрактном типе ***Stack*** из § 2.5.4, функции, интерфейс вызова которых может быть определен, а реализация — нет, объявлены виртуальными. В частности, реализации функций ***draw*** () и ***rotate*** () могут быть определены только для конкретных фигур, поэтому эти функции объявлены виртуальными.

При наличии такого определения мы можем написать общие функции для работы с векторами указателей на фигуры:

```
void rotate_all (vector<Shape*>& v, int angle) // поворачивает элементы v
                                               // на angle градусов
{
        for (int i = 0; i < v.size (); ++i ) v[i]->rotate (angle);
}
```

Для определения конкретной фигуры мы должны сказать, что нечто является фигурой, и указать особые свойства (в том числе определить виртуальные функции):

```
class Circle : public Shape {
        int radius;
public:
        void draw () { /* ... */ }
        void rotate (int) { }       // да, функция ничего не делает
};
```

В терминах C++ мы скажем, что класс ***Circle*** является производным от ***Shape***, а класс ***Shape*** является базовым для класса ***Circle***. Альтернативная терминология называет ***Circle*** и ***Shape*** подклассом и суперклассом (или надклассом) соответственно. Говорят, что производный класс наследует члены от базового класса, поэтому применение и базового и производного класса в совокупности обычно называют *наследованием*. Парадигма программирования теперь звучит так:

> *Реши, какие требуются классы;*
> *обеспечь полный набор операций для каждого класса;*
> *явно вырази общность через наследование.*

В тех случаях, когда такой общности нет, достаточно абстракции данных. Степень общности между типами, которую можно выделить, использовав наследование и виртуальные функции, является лакмусовой бумажкой для определения, применим ли объектно-ориентированный подход к данной задаче. В некоторых областях, таких как интерактивная графика, очевидно, что объектно-ориентированное программирование весьма полезно. В других задачах, таких как классические арифметические типы и вычисления, основанные на них, похоже, трудно найти применение чему-то большему, чем абстракция данных, а средства, необходимые для поддержки объектно-ориентированного программирования, выглядят бесполезными.

Выявление общности между типами в системе является нетривиальным процессом. Степень общности, которою можно выделить, зависит от способа организации системы. Когда система проектируется, и даже когда только пишутся технические требования, нужно активно искать общность. Классы можно спроектировать так, чтобы использовать их как строительные блоки для других типов. Следует также проверять, не проявляют ли существующие классы сходства, которые можно выделить в базовый класс.

Объяснение того, что такое объектно-ориентированное программирование, без подробного рассмотрения конкретных конструкций языков программирования, см. в работах [Kerr, 1987] и [Booch, 1994] (подробные ссылки приведены в § 23.6).

Иерархии классов и абстрактные классы (§ 2.5.4) дополняют, а не исключают друг друга (§ 12.5). Вообще, парадигмы, приведенные здесь, имеют тенденцию взаимно дополнять и поддерживать друг друга. Например, и классы, и модули содержат функции, тогда как модули содержат и классы, и функции. Опытный разработчик применяет различные парадигмы по мере необходимости.

2.7. Обобщенное программирование

Тому, кто захочет использовать стек, вряд ли всегда будет нужен именно стек символов. Стек — более общее понятие, которое не зависит от определения символа. Следовательно, он должен быть представлен независимо.

Вообще, если алгоритм можно выразить независимо от деталей представления и если это можно сделать приемлемым (с точки зрения накладных расходов) способом и без логических искажений, то так и нужно поступить.

Парадигма программирования для этого стиля звучит так:

> *Реши, какие требуются алгоритмы;*
> *параметризуй их так, чтобы они могли работать*
> *со множеством подходящих типов и структур данных.*

2.7.1. Контейнеры

Мы можем обобщить стек символов до стека элементов любого типа, объявив его как *шаблон* (template) и заменив конкретный тип ***char*** на параметр шаблона. Например:

```
template<class T> class Stack {
        T* v;
        int max_size;
        int top;
public:
        class Underflow { };
        class Overflow { };

        Stack (int s);      // конструктор
        ~Stack ();          // деструктор

        void push (T);
        T pop ();
};
```

Префикс *template<class T>* делает *T* параметром объявления, которому этот префикс предшествует.

Функции-члены можно определить аналогичным образом:

```
template<class T> void Stack<T>::push (T c)
{
        if (top == max_size) throw Overflow ();
        v[top] = c;
        top = top + 1;
}

template<class T> T Stack<T>::pop ()
{
        if (top == 0) throw Underflow ();
        top = top - 1;
        return v[top];
}
```

При наличии этих определений, стек можно использовать следующим образом:

```
Stack<char> sc (200);          // стек для 200 символов
Stack<complex> scplx (30);     // стек для 30 комплексных чисел
Stack<list<int> > sli (45);    // стек для 45 списков целых чисел
```

```
void f()
{
        sc.push('c');
        if (sc.pop() != 'c') throw Bad_pop();

        scplx.push(complex(1, 2));
        if (scplx.pop() != complex(1, 2)) throw Bad_pop();
}
```

Точно также в качестве шаблонов можно определить списки, вектора, ассоциативные массивы и т. д. Класс, содержащий набор элементов некоторого типа, обычно называют *классом-контейнером* или просто *контейнером*.

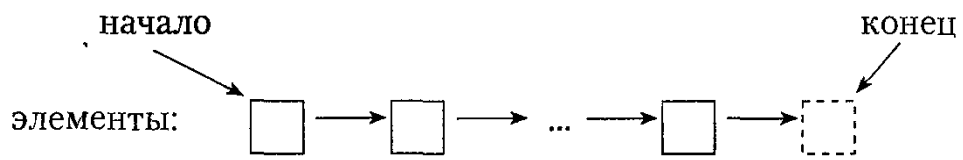
Шаблоны являются механизмом времени компиляции, поэтому их использование не влечет дополнительных накладных расходов во время исполнения по сравнению с «программированием вручную».

2.7.2. Обобщенные алгоритмы

Стандартная библиотека C++ предоставляет множество контейнеров, кроме того пользователи могут написать свои собственные (главы 3, 17 и 18). Оказывается, что мы можем применить парадигму обобщенного программирования еще и для параметризации алгоритмов контейнерами. Например, мы хотим сортировать, копировать, осуществлять поиск в векторах, списках и массивах и при этом не желаем писать функции *sort*(), *copy*() и *search*() отдельно для каждого контейнера. Мы также не хотим заниматься преобразованием к какой-нибудь одной конкретной структуре данных, допустимой для единственной функции сортировки. Поэтому мы должны найти обобщенный способ определения контейнеров, который позволит нам обрабатывать контейнер, не зная точно, что это за контейнер.

Один из подходов, а именно подход, принятый для контейнеров и нечисловых алгоритмов в стандартной библиотеке C++ (§ 3.8, глава 18), состоит во введении понятия последовательности и манипулировании последовательностями посредством итераторов.

Вот графическое представление понятия последовательности:



Последовательность имеет начало и конец. Итератор ссылается на элемент и предоставляет операцию, заставляющую его ссылаться на следующий элемент последовательности. Концом последовательности является итератор, который ссылается на элемент, следующий за последним. Физическим представлением конца может служить «элемент-страж», но это не обязательно. В действительности, такое определение последовательности охватывает множество различных представлений, включая списки и массивы.

Нам требуются стандартные обозначения таких операций, как «получить доступ к элементу через итератор» и «заставить итератор ссылаться на следующий элемент». Очевидным выбором (после того, как вы осмыслили идею) является исполь-

зование оператора разыменования * (dereference) для доступа к элементу через итератор и оператора инкремента ++ для того, чтобы заставить итератор ссылаться на следующий элемент.

Имея в виду вышесказанное, мы можем написать следующий код:

```
template<class In, class Out> void copy(In from, In too_far, Out to)
{
    while(from != too_far) {
        *to = *from;        // копируем элемент, на который указывает итератор
        ++to;               // следующий выходной элемент
        ++from;             // следующий входной элемент
    }
}
```

Приведенный пример копирует любой контейнер, для которого мы можем определить итераторы с подходящим синтаксисом и семантикой.

Встроенные в C++ типы низкого уровня, такие как указатели и массивы, имеют соответствующие операции, поэтому мы можем записать:

```
char vc1[200];    // массив из 200 символов
char vc2[500];    // массив из 500 символов

void f()
{
    copy(&vc1[0], &vc1[200], &vc2[0]);
}
```

Процедура скопирует с первого по последний элементы массива ***vc1*** в ***vc2***, заполняя ***vc2*** с первого элемента.

Все контейнеры стандартной библиотеки (§ 16.3, глава 17) поддерживают технику итераторов и последовательностей.

Два параметра шаблона (а не один) ***In*** и ***Out*** используются для указания типов источника и приемника. Так сделано потому, что часто возникает нужда в копировании из контейнера одного типа в контейнер другого типа. Например:

```
complex ac[200];

void g(vector<complex>& vc, list<complex>& lc)
{
    copy(&ac[0], &ac[200], lc.begin());
    copy(lc.begin(), lc.end(), vc.begin());
}
```

Эта процедура копирует массив в список и список в вектор. В стандартном контейнере ***begin()*** — это итератор, указывающий на его первый элемент.

2.8. Заключение

Не существует идеальных языков программирования. К счастью, язык программирования и не обязан быть идеальным, чтобы быть хорошим инструментом для написания даже огромных систем. В действительности, язык общего применения не может быть идеальным для всех задач. То, что является совершенством для одной задачи, очень часто оказывается недостатком для другой, потому что достижение совершен-

ства в конкретной области подразумевает специализацию. C++ создавался в качестве удобного инструмента для решения широкого круга задач и для более непосредственного выражения разнообразных идей.

Не все можно выразить непосредственно, пользуясь встроенными возможностями языка. На практике к этому и не нужно стремиться. Средства языка существуют для поддержки разнообразных стилей и методов программирования. Следовательно, при изучении языка нужно делать упор на освоении стиля, который является родным и естественным для этого языка, а не на детальном понимании всех его возможностей.

При практическом программирования мало толку от знания самых «тайных» средств языка или от использования максимально возможного количества средств. То или иное средство языка само по себе не представляет большого интереса. Только в контексте общей технологии и других средств оно приобретает смысл и значение. Поэтому при чтении следующих глав пожалуйста помните, что подлинной целью глубокого проникновения в C++ должно быть стремление научиться использовать средства языка в комплексе, опираясь на хороший стиль программирования и выверенные методы проектирования.

2.9. Советы

[1] Не паникуйте! Все станет понятным в свое время; § 2.1.
[2] Для написания хороших программ не нужно знать о C++ все; § 1.7.
[3] Сосредоточьте внимание на технологиях программирования, а не на элементах языка; § 2.1.

3

Обзор стандартной библиотеки

Зачем тратить время на обучение,
когда невежество дается даром?
— Хоббс

Стандартные библиотеки — вывод — строки — ввод — вектора — проверка диапазона — списки — ассоциативные массивы — обзор контейнеров — алгоритмы — итераторы — итераторы ввода/вывода — предикаты — алгоритмы, использующие функции-члены — обзор алгоритмов — комплексные числа — векторная арифметика — обзор стандартной библиотеки — советы.

3.1. Введение

Ни одна программа приличных размеров не пишется с использованием только «голых» конструкций языка. Сначала разрабатываются библиотеки поддержки. Они и создают основу для дальнейшей работы.

В продолжение главы 2 здесь дается краткий обзор основных возможностей библиотеки, чтобы вы поняли, что можно сделать при помощи C++ и стандартной библиотеки. Полезные библиотечные типы, такие как ***string***, ***vector***, ***list*** и ***map***, описаны вместе с наиболее общими способами их использования. Благодаря этому, я смогу в последующих главах приводить более осмысленные примеры и упражнения. Как и в главе 2, настоятельно рекомендуется не отвлекаться и не огорчаться из-за неполного понимания деталей. Цель этой главы — дать вам почувствовать, что вас ожидает, и помочь понять простейшие способы использования наиболее полезных библиотечных средств. Более подробное введение в стандартную библиотеку приведено в § 16.1.2.

Средства стандартной библиотеки, описанные в этой книге, являются частью любой полной реализации C++. В добавление к стандартной библиотеке C++, большинство реализаций предоставляют «графический интерфейс пользователя», часто именуемый «GUI» (Graphical User Interface) или «оконной системой», для взаимодействия между пользователем и программой. Аналогично, большинство сред разработки приложений предоставляют «библиотеки основных компонент» для поддержки корпоративной или отраслевой «стандартной» среды разработки и/или выполнения. Я не описываю такие системы и библиотеки. Целью является замкну-

тое изложение C++, как он определен стандартом. За исключением особо оговоренных случаев, я привожу полностью переносимые примеры. Естественно, программисту следует ознакомиться со средствами, доступными в разных системах, но это оставлено в качестве упражнений.

3.2. Здравствуй, мир!

Минимальной программой на C++ является следующая:

```
int main () { }
```

Она определяет функцию с именем ***main***, которая не принимает аргументов и ничего не делает.

Каждая программа на C++ должна содержать функцию с именем ***main*** (). Программа начинает выполняться с этой функции. Целое значение, возвращаемое ***main*** (), если оно вообще возвращается, предназначается «для системы». Если не возвращается никакого значения, система получит значение, означающее успешное завершение. Ненулевое значение, возвращенное ***main*** (), означает аварийное завершение.

Обычно программы что-то выводят. Вот пример программы, которая выводит фразу «*Здравствуй, мир!*»:

```
#include <iostream>

int main ()
{
    std::cout << "Здравствуй, мир!\n";
}
```

Строка ***#include <iostream>*** дает указание компилятору *включить* (include) объявления средств ввода/вывода стандартной библиотеки, находящиеся в ***iostream***. Без этих объявлений выражение

```
std::cout << "Здравствуй, мир!\n"
```

не имело бы смысла. Оператор << («записать в») записывает свой второй аргумент в первый. В данном случае, строковый литерал ***"Здравствуй, мир!\n"*** записывается в стандартный поток вывода ***std::cout***. Строковый литерал является последовательностью символов, заключенной в двойные кавычки. В строковом литерале обратная наклонная черта \, за которой следует символ, означает один специальный символ. В нашем случае, ***\n*** означает символ перевода строки, поэтому выводится фраза ***Здравствуй, мир!***, за которой следует новая строка.

3.3. Пространство имен стандартной библиотеки

Стандартная библиотека определена в пространстве имен (§ 2.4, § 8.2) ***std***. Вот почему я написал ***std::cout***, а не просто ***cout***. Я явно обратился к стандартному ***cout***, а не к какому-либо другому ***cout***.

Каждое средство стандартной библиотеки становится доступным через какой-либо стандартный заголовочный файл наподобие ***<iostream>***. Например

```
#include <string>
#include <list>
```

делает доступными стандартные объявления *string* и *list*. Для обращения к ним можно воспользоваться префиксом *std*::, например:

```
std::string s = "Четыре ноги хорошо, две пло-о-охо!";
std::list<std::string> slogans;
```

Для простоты я редко буду явно использовать префикс *std*:: в примерах. Я также не всегда буду явно включать (*#include*) заголовочные файлы. Для того чтобы откомпилировать и запустить фрагменты программ, приведенные здесь, вы должны будете включить соответствующие заголовочные файлы (перечисленные в § 3.7.5, § 3.8.6 и главе 16). Кроме того, вы должны либо использовать префикс *std*::, либо объявить все имена из *std* глобальными (§ 8.2.3). Например:

```
#include <string>                    // сделаем доступными стандартные
                                     // средства работы со строками
using namespace std;                 // обеспечили доступ к именам из std
                                     // без префикса std::

string s = "Невежество — блаженство!"; // правильно: string означает std::string
```

Как правило, объявление всех имен из данного пространства глобальными является признаком плохого стиля. Однако ради краткости изложения фрагментов программ, иллюстрирующих средства языка и библиотеки, я опущу повторяющиеся *#include* и квалификаторы *std*::. В этой книге я почти всегда использую только стандартную библиотеку, так что если задействуется имя из стандартной библиотеки, то это либо использование, предлагаемое стандартом, либо часть объяснения того, как может быть определено стандартное средство.

3.4. Вывод

Библиотека *iostream* определяет вывод для каждого встроенного типа. Легко можно определить вывод для типа, определяемого пользователем. По умолчанию, значения, выводимые в *cout*, преобразуются в последовательность символов. Например,

```
void f()
{
    cout << 10;
}
```

поместит символ *1*, а затем *0* в стандартный поток вывода. То же самое произойдет и в следующем примере.

```
void g()
{
    int i = 10;
    cout << i;
}
```

Вывод различных типов можно объединить очевидным способом:

```
void h(int i)
{
    cout << "значение i равно ";
```

```
        cout << i;
        cout << '\n';
}
```

Если *i* равно *10*, то на выходе мы получим:

значение i равно 10

Символьной константой называется символ, заключенный в одиночные кавычки. Обратите внимание, что символьная константа выводится как символ, а не как числовое значение. Например, функция

```
void k()
{
        cout << 'a';
        cout << 'b';
        cout << 'c';
}
```

выведет *abc*.

Довольно быстро вам надоест повторять имя потока вывода при выводе нескольких связанных элементов. К счастью, результат операции вывода может в свою очередь быть использован для дальнейшего вывода:

```
void h2 (int i)
{
        cout << "значение i равно " << i << '\n';
}
```

Это эквивалентно *h* (). Потоки излагаются более подробно в главе 21.

3.5. Строки

В стандартной библиотеке имеется тип ***string***, который дополняет строковые литералы, использовавшиеся ранее. Тип ***string*** обеспечивает множество полезных операций над строками, таких, например, как конкатенация (объединение):

```
string s1 = "Здравствуй";
string s2 = "мир";

void m1 ()
{
        string s3 = s1 + ", " + s2 + "!\n";

        cout << s3;
}
```

В этом примере *s3* инициализируется последовательностью символов

Здравствуй, мир!

за которой следует символ перевода строки. Сложение строк означает конкатенацию. Вы можете складывать строку со строкой, строковым литералом и символом.

Во многих приложениях наиболее часто встречающаяся форма конкатенации — добавление чего-либо в конец строки. Это непосредственно поддерживается операцией +=. Например:

```
void m2 (string& s1, string& s2)
{
        s1 = s1 + '\n';          // добавить символ перевода строки
        s2 += '\n';              // добавить символ перевода строки
}
```

Оба способа добавления к концу строки семантически эквивалентны, но я предпочитаю второй, потому что он более точно отражает смысл операции и, скорее всего, реализован более эффективно.

Естественно, строки могут сравниваться друг с другом и со строковыми литералами. Например:

```
string incantation;

void respond (const string& answer)
{
        if (answer == incantation) {
                // сотворим чудеса (incantation — заклинание)
        }
        else if (answer == "да") {
                // ...
        }
// ...
}
```

Класс ***string*** из стандартной библиотеки описывается в главе 20. Помимо других полезных вещей, он предоставляет операции с подстроками (фрагментами строк). Например:

```
string name = "Niels Stroustrup";

void m3 ()
{
        string s = name.substr (6, 10);          // s = "Stroustrup"
        name.replace (0, 5, "Nicholas");         // name = "Nicholas Stroustrup"
}
```

Операция ***substr*** () возвращает строку, которая является копией части исходной строки. Первый аргумент является индексом в строке (указателем позиции), а второй указывает размер фрагмента строки, который требуется получить. Так как индексация начинается с ***0***, ***s*** получает значение ***Stroustrup***.

Операция ***replace*** () замещает фрагмент строки другим значением. В нашем случае, подстрока, начинающаяся с ***0***-й позиции, длины ***5*** (***Niels***), заменяется на ***Nicholas***. Значит ***name*** примет значение ***Nicholas Stroustrup***. Обратите внимание, что замещающая строка не обязана иметь размер, равный размеру замещаемой подстроки.

3.5.1. C-строки

C-строки — это массивы символов, ограниченные нулем (§ 5.2.2). Мы можем легко преобразовать C-строку в ***string***. Для вызова функции, аргументом которой является C-строка, нам нужно уметь представлять значение ***string*** в виде C-строки. Это выполняет функция ***c_str*** () (§ 20.3.7). Например, мы можем вывести ***name***, воспользовавшись функцией вывода языка C ***printf*** () (§ 21.8), следующим образом:

```
void f()
{
    printf("name: %s\n", name.c_str());
}
```

3.6. Ввод

Стандартная библиотека предлагает потоки ввода (*istream*). Как и потоки вывода, потоки ввода работают с символьным представлением встроенных типов и их можно легко расширить для работы с типами, определяемыми пользователем.

В качестве оператора ввода, используется >> («прочесть из»). Стандартным потоком ввода является *cin*. Тип операнда в правой части оператора >> определяет способ интерпретации вводимых символов и то, куда будут записываться значения из входного потока. Например, функция

```
void f()
{
    int i;
    cin >> i;      // считать целое в i

    double d;
    cin >> d;      // считать число с плавающей точкой двойной точности в d
}
```

считывает число (скажем, *1234*) из стандартного ввода и помещает его в целую переменную *i*, а число с плавающей точкой (например, *12.34e5*) помещается в переменную с плавающей точкой двойной точности *d*.

Приведем пример преобразования дюймов в сантиметры и сантиметров в дюймы. Вы вводите число, за которым следует символ, означающий единицу измерения — сантиметры или дюймы. Программа выведет соответствующее значение в других единицах:

```
int main()
{
    const float factor = 2.54;      // 1 дюйм равен 2.54 см
    float x, in, cm;                // in — это дюймы (inches)
    char ch = 0;

    cout << "введите длину: ";

    cin >> x;           // считать число с плавающей точкой
    cin >> ch;          // считать суффикс

    switch (ch) {
    case 'i':           // дюймы
        in = x;
        cm = x*factor;
        break;
    case 'c':           // см
        in = x/factor;
        cm = x;
        break;
    default:
```

```
            in = cm = 0;
            break;
        }
        cout << in << " дюймов = " << cm << " см\n";
    }
```

switch-инструкция сравнивает значение с набором констант. ***break*-инструкции** используются для выхода из инструкции ***switch***. Все константы в инструкции ***case*** должны отличаться друг от друга. Если сравниваемое значение не равно ни одной из констант, выбирается ***default***.

Часто нам нужно прочитать последовательность символов. Удобным способом решения этой задачи является чтение в ***string***. Например:

```
int main ()
{
    string str;

    cout << "Пожалуйста, введите ваше имя\n";
    cin >> str;
    cout << "Здравствуй, " << str << "!\n";
}
```

Если вы введете

Эрик

вывод будет

Здравствуй, Эрик!

По умолчанию, символ-разделитель (например, пробел) (§ 5.2.2) означает конец ввода, поэтому, если вы введете

Эрик Кровавый Топор

притворяясь несчастным королем Йорка, ответом все равно будет

Здравствуй, Эрик!

Считать всю строку можно с помощью функции ***getline*** (). Например:

```
int main ()
{
    string str;

    cout << "Пожалуйста, введите ваше имя\n";
    getline (cin, str);
    cout << "Здравствуй, " << str << "!\n";
}
```

Теперь при вводе

Эрик Кровавый Топор

на выходе мы получим ожидаемое:

Здравствуй, Эрик Кровавый Топор!

Стандартные строки имеют замечательное свойство расширяться для хранения всей информации, которую вы хотите в них поместить, поэтому если вы введете пару мегабайт двоеточий, программа выведет вам в ответ всю эту кучу двоеточий — если только ваш компьютер или операционная система не исчерпают раньше какой-либо критичный ресурс.

3.7. Контейнеры

Многие вычисления подразумевают создание наборов объектов в различных формах и обработку таких наборов. Простыми примерами являются чтение символов в строку и вывод строки. Класс, главной целью которого является хранение объектов, называется *контейнером*. Реализация контейнеров, подходящих для данной задачи, и поддержка их основными полезными операциями — важнейшие шаги при написании любой программы.

Для иллюстрации наиболее полезных контейнеров стандартной библиотеки рассмотрим простую программу, хранящую имена и номера телефонов. Здесь для людей с разным опытом «простыми и очевидными» окажутся совершенно различные подходы к реализации таких программ.

3.7.1. Вектор

Для многих программистов на C подходящей точкой отсчета будет стандартный массив пар (имя, номер):

```
struct Entry {                          // запись телефонной книги
        string name;
        int number;
};

Entry phone_book[1000];                 // телефонная книга из 1000 записей

void print_entry (int i)                // использование (печатаем запись)
{
        cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}
```

Но встроенные массивы имеют фиксированный размер. Если мы выберем слишком большой размер, то впустую израсходуем память. Если же выбранный размер слишком мал, массив может переполниться. В любом случае нам придется написать код низкого уровня для управления памятью. Стандартная библиотека предоставляет вектора (*vector*) (§ 16.3), которые сами позаботятся об этом:

```
vector<Entry> phone_book(1000);

void print_entry (int i)                // используем как раньше
{
        cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}

void add_entries (int n)                // увеличивает размер книги на n записей
{
        phone_book.resize(phone_book.size() + n);
}
```

Функция *size* (), член класса *vector*, возвращает количество элементов.

Обратите внимание на использование скобок при определении *phone_book*. Мы создали единственный объект типа *vector<Entry>* и в качестве инициализатора передали его начальный размер (в круглых скобках). Это существенно отличается от объявления встроенного массива:

```
vector<Entry> book(1000);        // вектор из 1000 элементов
vector<Entry> books[1000];       // 1000 пустых векторов
```

Если при объявлении переменной типа *vector* вы сделаете ошибку, записав [] вместо (), компилятор почти наверняка «выловит» ее и выдаст сообщение об ошибке при попытке использования переменной *books*.

Вектор является единым объектом, который можно присвоить. Например:

```
void f(vector<Entry>& v)
{
    vector<Entry> v2 = phone_book;
    v = v2;
    // ...
}
```

Присваивание переменной типа *vector* означает копирование элементов из присваиваемой переменной. Поэтому после инициализации и присваивания в *f*(), *v* и *v2* содержат отдельные копии всех записей в телефонной книге. Когда вектор содержит много элементов, такие невинные на первый взгляд присваивания и инициализации могут оказаться недопустимо ресурсоемкими. Там, где нет необходимости копировать, следует пользоваться ссылками или указателями.

3.7.2. Проверка допустимости диапазона

Класс *vector* из стандартной библиотеки по умолчанию не обеспечивает проверку диапазона (§ 16.3.3). Например:

```
void f()
{
    int i = phone_book[1001].number; // 1001 — вне пределов диапазона
    // ...
}
```

Это присваивание, скорее всего, поместит в *i* какое-нибудь случайное значение, а не выдаст сообщение об ошибке. Это нежелательно, поэтому я воспользуюсь простой, проверяющей диапазон адаптацией класса *vector* по имени *Vec*. *Vec* во всем похож на *vector*, но он будет генерировать исключение типа *out_of_range*, если индекс окажется вне пределов диапазона.

Технологии реализации типов, подобных *Vec*, и эффективного использования исключений обсуждаются в § 11.12, § 8.3 и главе 14. Однако, приведенного здесь определения вполне достаточно для примеров в этой книге:

```
template< class T> class Vec : public vector<T> {
public:
    Vec() : vector<T>() { }
    Vec(int s) : vector<T>(s) { }
```

```
        T& operator[](int i) { return at(i); }              // с проверкой диапазона
        const T& operator[](int i) const { return at(i); } // с проверкой диапазона
};
```

Функция-член *at()* — это операция индексирования вектора (то есть операция доступа к элементу вектора по индексу), которая генерирует исключение типа *out_of_range* (из <*stdexcept*>), если ее аргумент находится вне пределов диапазона вектора (§ 16.3.3).

Возвращаясь к задаче о хранении имен и телефонных номеров, мы теперь можем воспользоваться *Vec*, будучи уверенными, что обращения вне диапазона вектора будут перехвачены. Например:

```
Vec<Entry> phone_book(1000);
void print_entry(int i) // используем как раньше
{       cout << phone_book[i].name << ' ' << phone_book[i].number << '\n'; }
```

Обращение вне границ диапазона сгенерирует исключение, которое пользователь сможет перехватить. Например:

```
void f()
{
        try { for (i = 0; i < 10000; i++) print_entry(i); }
        catch(out_of_range) {
                cout << "выход за пределы диапазона\n";
        }
}
```

Когда будет предпринята попытка доступа к *phone_book[i]* с *i==1000*, будет сгенерировано, а затем и перехвачено исключение. Если пользователь не перехватывает этот тип исключений, программа будет завершена хорошо определенным образом, а не продолжит выполняться непредсказуемо и не повиснет. Одним из способов минимизации сюрпризов от исключений является использование *main()* с блоком *try* в качестве тела функции:

```
int main()
try {
        // код программы
}
catch(out_of_range) {
        cerr << "выход за пределы диапазона\n";
}
catch(...) {
        cerr << "сгенерировано неизвестное исключение\n";
}
```

Это обеспечит обработчик исключений по умолчанию. Таким образом, если мы не перехватим какое-нибудь исключение, соответствующее сообщение запишется в стандартный поток диагностики ошибок *cerr* (§ 21.2.1).

3.7.3. Список

Обычными операциями для телефонного справочника являются включение и удаление номеров. Поэтому список (list) — более подходящая (чем вектор) структура для представления простого телефонного справочника. Например:

```
list<Entry> phone_book;
```

При использовании списка мы не применяем индексирование для доступа к элементу, как мы обычно поступаем с векторами. Вместо этого осуществляется поиск элемента списка с заданным значением. Мы пользуемся тем, что список является последовательностью, как объяснено в § 3.8:

```
void print_entry (const string& s)
{
    typedef list<Entry>::const_iterator LI;
    for (LI i = phone_book.begin (); i != phone_book.end (); ++i) {
        const Entry& e = *i;    // для краткости используется ссылка
        if (s == e.name) {
            cout << e.name << ' ' << e.number << '\n';
            return;
        }
    }
}
```

Поиск вхождения *s* стартует с начала списка и продолжается до тех пор, пока либо не будет найден *s*, либо не закончится весь список. Каждый контейнер стандартной библиотеки снабжен функциями ***begin*** () и ***end*** (), которые возвращают итераторы на первый элемент и «элемент, следующий за последним» соответственно (§ 16.3.2). Если имеется итератор *i*, следующим элементом будет *++i*. Если есть итератор *i*, элемент, на который он ссылается, это **i*.

Пользователю нет необходимости точно знать тип итератора стандартного контейнера. Тип итератора является частью определения контейнера и на него можно сослаться по имени. Когда нет необходимости изменять элементы контейнера, нужен итератор типа ***const_iterator***. В противном случае пользуются обычным типом ***iterator*** (§ 16.3.1).

Добавлять элементы к списку или удалять их из списка очень просто:

```
void add_entry (Entry& e, list<Entry>::iterator i, list<Entry>::iterator p)
{
    phone_book.push_front (e);  // добавить к началу
    phone_book.push_back (e);   // добавить в конец
    phone_book.insert (i, e);   // вставить перед элементом *i

    phone_book.erase (p);       // удалить элемент *p
}
```

Более полное описание ***insert*** () и ***erase*** () см. в § 16.3.6.

3.7.4. Ассоциативные массивы

Писать код для поиска имени в списке, состоящем из пар (имя, номер) — весьма утомительное занятие. Кроме того, последовательный поиск достаточно неэффективен за исключением очень коротких списков. Вставку, удаление и поиск значения непосредственно поддерживают другие структуры данных. В частности, в стандартной библиотеке имеется тип ***map*** (§ 17.4.1). Тип ***map*** является контейнером для хранения пар величин. Например:

```
map<string, int> phone_book;
```

В других контекстах ***map*** называют «ассоциативным массивом» или «словарем».

Будучи проиндексирован по значению первого типа (называемому *ключом*), ***map*** возвращает соответствующее значение второго типа (называемое *значением* или *отображенным типом*). Например:

```
void print_entry (const string& s)
{
    if (int i = phone_book[s]) cout << s << ' ' << i << '\n';
}
```

Если для ключа ***s*** вхождение не найдено, возвращается значение по умолчанию. Значением по умолчанию для целого типа в ***map*** является ***0***. В нашем случае я предполагаю, что ***0*** не является допустимым телефонным номером.

3.7.5. Стандартные контейнеры

И ***map***, и ***list***, и ***vector*** могут использоваться для представления телефонной книги. Каждый из этих типов имеет свои сильные и слабые стороны. Например, доступ по индексу к элементу вектора прост и эффективен. С другой стороны, вставка нового элемента между двумя соседними является для вектора довольно дорогим занятием. Со списком все обстоит ровно наоборот. Тип ***map*** реализует список пар (ключ, значение), но кроме этого он оптимизирован для поиска значения по ключу.

Стандартная библиотека предоставляет самые общие и полезные типы контейнеров, что позволяет программисту выбрать контейнер, наиболее полно удовлетворяющий потребностям приложения:

Стандартные контейнеры

vector<T>	вектор переменного размера (§ 16.3)
list<T>	двусвязный список (§ 17.2.2)
queue<T>	очередь (§ 17.3.2)
stack<T>	стек (§ 17.3.1)
deque<T>	очередь с двумя концами (§ 17.2.3)
priority_queue<T>	очередь, отсортированная по значению (§ 17.3.3)
set<T>	множество (§ 17.4.3)
multiset<T>	множество, в котором одно значение может встречаться несколько раз (§ 17.4.4)
map<key, val>	ассоциативный массив (§ 17.4.1)
multimap<key, val>	ассоциативный массив, в котором ключ (key) может встречаться несколько раз (§ 17.4.2)

Стандартные контейнеры описаны в § 16.2, § 16.3 и главе 17. Контейнеры определены в пространстве имен ***std*** и представлены в заголовочных файлах <***vector***>, <***list***>, <***map***> и т. д., см. § 16.2.

Стандартные контейнеры и их основные операции разрабатывались так, чтобы обеспечить единство записи и обозначений. Более того, смысл одноименных операций одинаков для различных контейнеров. Основные операции применимы к любому типу контейнера. Например, ***push_back()*** можно использовать (с разумной эффек-

тивностью) для добавления элемента в конец контейнеров и типа *vector*, и типа *list*. Каждый контейнер имеет функцию-член *size* (), которая возвращает количество элементов в нем.

Это единство обозначений и смысла позволяет программистам создавать новые типы контейнеров, которые можно использовать практически так же, как и стандартные. Примером может служить тип *Vec* с проверкой диапазона (§ 3.7.2). В главе 17 демонстрируется, как добавить к библиотеке *hash_map* (хешируемый ассоциативный массив). Единство интерфейсов контейнеров также позволяет разрабатывать алгоритмы независимо от конкретных типов контейнеров.

3.8. Алгоритмы

Такие структуры данных, как список или вектор, сами по себе не представляют большого интереса. Для их использования нужны базовые операции доступа, такие как добавление и удаление элементов. Объекты редко просто хранятся в контейнере. Они сортируются, печатаются, удаляются; извлекаются подмножества элементов контейнера, ищутся вхождения элементов и т. д. Поэтому в стандартной библиотеке, наряду с самыми общеупотребительными контейнерами, имеются также и общие алгоритмы их обработки. Например, следующий пример сортирует вектор и помещает все уникальные элементы вектора в список:

```
void f(vector< Entry>& ve, list< Entry>& le)
{
        sort (ve.begin (), ve.end ());
        unique_copy (ve.begin (), ve.end (), le.begin ());
}
```

Стандартные алгоритмы описываются в главе 18. Они выражаются в терминах последовательностей элементов (§ 2.7.2). Последовательность представлена парой итераторов, указывающих на первый элемент и «элемент, следующий за последним». В нашем примере *sort* () сортирует последовательность, начиная с *ve.begin* () по *ve.end* (), то есть все элементы вектора. Для записи вам необходимо задать только первый элемент, который должен быть записан. Если записывается более одного элемента, элементы, следующие за начальным, будут перезаписаны.

Если бы мы захотели добавить новые элементы в конец контейнера, можно было бы написать следующее:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
        sort (ve.begin (), ve.end ());
        unique_copy (ve.begin (), ve.end (), back_inserter (le));        // добавить к le
}
```

back_inserter () добавляет элементы в конец контейнера, увеличивая его до необходимых размеров (§ 19.2.4). Таким образом, стандартные контейнеры посредством *back_inserter* () устраняют необходимость использования подверженного ошибкам явного управления памятью в стиле C с помощью *realloc* () (§ 16.3.5). Если вы забудете воспользоваться *back_inserter* () при добавлении, то это может привести к ошибкам. Например:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    copy(ve.begin(), ve.end(), le);          // ошибка: le — не итератор
    copy(ve.begin(), ve.end(), le.end());    // плохо: пишет за конец
    copy(ve.begin(), ve.end(), le.begin());  // перезаписывает элементы
}
```

3.8.1. Использование итераторов

Впервые появившись в программе, контейнер предоставляет лишь несколько итераторов, ссылающихся на полезные элементы. Лучшими примерами служат ***begin*** () и ***end*** (). Кроме того, итераторы возвращаются многими алгоритмами. Например, стандартный алгоритм ***find*** ищет значение в последовательности и возвращает итератор, указывающий на найденный элемент. С помощью ***find*** мы можем подсчитать количество вхождений символа в строку:

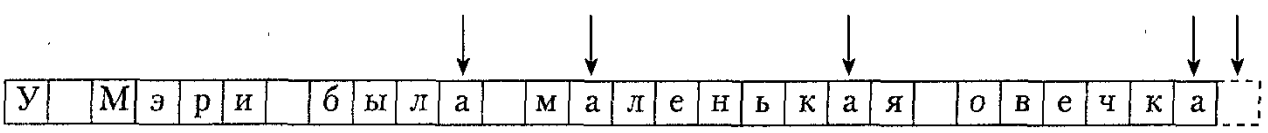
```
int count(const string& s, char c)
{
    int n = 0;
    string::const_iterator i = find(s.begin(), s.end(), c);
    while (i != s.end()) {
        ++n;
        i = find(i+1, s.end(), c);
    }
    return n;
}
```

Функция ***find*** возвращает либо итератор, указывающий на первое вхождение значения в последовательность, либо итератор, указывающий на элемент, следующий за последним. Рассмотрим, что произойдет при простом вызове ***count***:

```
void f()
{
    string m = "У Мэри была маленькая овечка";
    int a_count = count(m, 'a');
}
```

Первый вызов ***find*** () найдет *'a'* в слове ***была***. Таким образом, итератор укажет на этот символ, а не на ***s.end*** (), поэтому мы войдем в цикл ***while***. В цикле мы начинаем поиск с ***i+1***, то есть со следующего за *'a'* символа. После того, как все *'a'* будут найдены, ***find*** () достигнет конца и вернет ***s.end*** (). Условие ***i != s.end*** () не выполнится и мы выйдем из цикла.

Вызов ***count*** () можно графически представить следующим образом:



Стрелки указывают на первое, промежуточные и конечное значения итератора ***i***.

Естественно, ***find*** будет работать аналогичным образом с любым стандартным контейнером. Следовательно, мы могли бы написать функцию ***count*** () в более общем виде:

```
template<class C, class T> int count(const C& v, T val)
{
    typename C::const_iterator i = find(v.begin(), v.end(), val);    // "typename" см. § C.13.5
    int n = 0;
    while (i != v.end()) {
        ++n;
        ++i;        // перейти на элемент, следующий за только что найденным
        i = find(i, v.end(), val);
    }
    return n;
}
```

Это работает и поэтому можно написать:

```
void f(list<complex>& lc, vector<string>& vs, string s)
{
    int i1 = count(lc, complex(1, 3));
    int i2 = count(vs, "Хризипп");
    int i3 = count(s, 'x');
}
```

На самом деле нам не нужно определять шаблон ***count***. Подсчет числа вхождений элемента настолько часто встречается, что этот алгоритм реализован в стандартной библиотеке. Для достижения большей общности аргументом функции ***count*** из стандартной библиотеки является последовательность, а не контейнер, поэтому мы можем писать так:

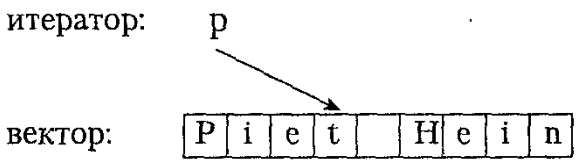
```
void f(List<complex>& lc, vector<string>& vs, string s)
{
    int i1 = count(lc.begin(), lc.end(), complex(1, 3));
    int i2 = count(vs.begin(), vs.end(), "Диоген");
    int i3 = count(s.begin(), s.end(), 'x');
}
```

Использование последовательностей дает возможность вызывать ***count*** для встроенных массивов, а также производить подсчет числа вхождений в части контейнера. Например:

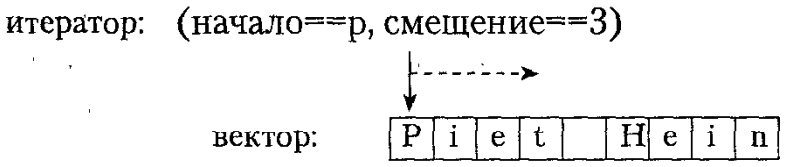
```
void g(char cs[], int sz)
{
    int i1 = count(&cs[0], &cs[sz], 'z');      // число символов 'z' в массиве
    int i2 = count(&cs[0], &cs[sz/2], 'z');    // число символов 'z' в первой половине массива
}
```

3.8.2. Типы итераторов

Что же такое итераторы на самом деле? Каждый конкретный итератор является объектом некоторого типа. Однако существует множество разновидностей итераторов, потому что итератор должен содержать информацию, необходимую для работы с контейнером определенного типа. Типы итераторов могут отличаться настолько же, насколько отличаются типы контейнеров и цели, для которых используются итераторы. Например, итератором для вектора (***vector***), скорее всего, будет обыкновенный указатель, потому что указатель является достаточно разумным способом ссылки на элемент вектора:

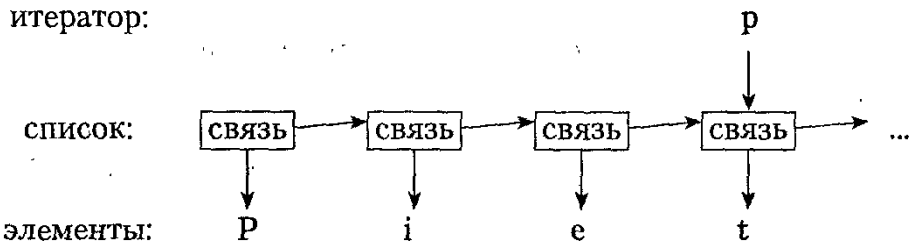


Или можно реализовать итератор вектора в виде пары (указатель на вектор, индекс):



Использование такого итератора позволяет осуществить проверку диапазона (§ 19.3).

Итератор для списка должен быть немного сложнее, чем простой указатель на элемент, потому что элемент в списке, в общем случае, не знает, где находится следующий элемент списка. Таким образом, итератор списка может быть указателем на связь (link):



Что является общим для всех итераторов, так это их смысл и имена операций. Например, применение ++ к любому итератору приведет к тому, что итератор будет указывать на следующий элемент. Аналогично, * означает элемент, на который ссылается итератор. В действительности, любой объект, подчиняющийся нескольким простым правилам, наподобие перечисленных, является итератором (§ 19.2.1). Более того, пользователям редко требуется знать тип конкретного итератора. Каждый контейнер «знает» тип своего итератора и обеспечивает доступ к нему через стандартные имена ***iterator*** и ***const_iterator***. Например, ***list<Entry>::iterator*** является общим типом итератора для ***list<Entry>***. Мне очень редко приходилось беспокоиться о деталях определения этого типа.

3.8.3. Итераторы и ввод/вывод

Итераторы являются общей и полезной концепцией обработки последовательностей элементов в контейнерах. Однако контейнеры — не единственный случай, когда нам встречаются последовательности элементов. Например, поток ввода является последовательностью значений; мы записываем последовательность значений в поток вывода. Следовательно, понятие итераторов можно применить ко вводу/выводу.

Чтобы создать итератор ***ostream_iterator***, нам необходимо указать, какой поток будет использоваться, и тип объектов, записываемых в него. Например, мы можем определить итератор, который ссылается на поток стандартного вывода ***cout***:

```
ostream_iterator<string> oo (cout);
```

Эффект от присваивания ***oo** состоит в записи присваиваемого значения в стандартный поток вывода ***cout***. Например:

```
int main ()
{
        *oo = "Здравствуй, ";          // означает cout << "Здравствуй, "
        ++oo;
        *oo = "мир!\n";                // означает cout << "мир!\n"
}
```

Это еще один метод записи «канонического» текста в стандартный вывод. Выражение ++*oo* имитирует запись в массив через указатель. Я вряд ли стал бы этим пользоваться для такой простой задачи, но идея интерпретации вывода в качестве контейнера с атрибутом доступа «только для записи» скоро станет, если уже не стала, очевидной.

Аналогично, итератор ***istream_iterator*** является тем, что позволит нам интерпретировать поток ввода, как контейнер «только для чтения». Мы снова должны указать используемый поток и типы ожидаемых величин:

```
istream_iterator <string> ii (cin);
```

Так как итераторы ввода неизменно появляются парами, определяющими некую последовательность, мы должны предоставить ***istream_iterator***, указывающий на конец ввода. Вот ***istream_iterator*** по умолчанию:

```
istream_iterator<string> eos;
```

Теперь мы можем прочитать "***Здравствуй, мир***!" из потока ввода, а затем и вывести эту строку следующим образом:

```
int main ()
{
        string s1 = *ii;
        ++ii;
        string s2 = *ii;

        cout << s1 << ' ' << s2 << '\n';
}
```

На самом деле итераторы ***istream_iterator*** и ***ostream_iterator*** не предназначены для непосредственного применения. Они, как правило, используются в качестве параметров некоторых алгоритмов. Например, мы можем написать простую программу, которая читает файл, сортирует прочитанные слова, удаляет дубликаты и записывает результат в другой файл:

```
int main ()
{
        string from, to;
        cin >> from >> to;                  // прочитать имена исходного
                                            // и целевого файлов

        ifstream is (from.c_str ());        // поток ввода (c_str(), см. § 3.5)
        istream_iterator<string> ii (is);   // итератор ввода для потока
        istream_iterator<string> eos;       // страж ввода

        vector<string> b (ii, eos);         // b — вектор, инициализируемый вводом
        sort (b.begin (), b.end ());        // сортируем буфер

        ofstream os (to.c_str ());          // поток вывода
        ostream_iterator<string> oo (os, '\n');  // итератор вывода для потока
```

```
    unique_copy (b.begin (), b.end (), oo);      // копировать буфер в поток,
                                                 // удалив повторяющиеся значения

    return !is.eof() || !os;                     // возвратить состояние
                                                 // ошибки (§ 3.2, § 21.3.4)
}
```

Тип ***ifstream*** — это поток ввода ***istream***, который может быть связан с файлом, а ***ofstream*** — это поток вывода ***ostream***, который может быть связан с файлом. Второй аргумент ***ostream_iterator*** используется для того, чтобы ограничить выходные значения.

3.8.4. Проход с выполнением и предикаты

Итераторы позволяют писать циклы для перебора элементов последовательностей. Однако, написание циклов может оказаться утомительным занятием, поэтому стандартная библиотека предоставляет способы вызова функций для каждого элемента последовательности.

Рассмотрим программу чтения слов из потока ввода и записи частот их вхождений. Очевидным способом представления строк с указанием частоты вхождения является ассоциативный массив ***map***:

```
map<string, int> histogram;        // гистограмма частот
```

Очевидным действием, применяемым к каждой строке для подсчета частот, является:

```
void record (const string& s)
{
    histogram[s]++;                // частота вхождений s
}
```

После считывания ввода нам необходимо вывести собранные данные. Ассоциативный массив ***map*** состоит из последовательности пар (***string***, ***int***). Следовательно, нам нужно вызвать

```
void print (const pair<const string, int>& r)
{
    cout << r.first << ' ' << r.second << '\n';
}
```

для каждого элемента в массиве (***first*** — это первый элемент в паре, ***second*** — второй). Итак, главная программа принимает вид:

```
int main ()
{
    istream_iterator<string> ii (cin);
    istream_iterator<string> eos;

    for_each (ii, eos, record);
    for_each (histogram.begin (), histogram.end (), print);
}
```

Обратите внимание, что нам не приходится сортировать массив, чтобы вывод проводился в правильном порядке. Ассоциативный массив хранит свои элементы по

порядку, поэтому итерация по массиву проходит с учетом порядка (в порядке возрастания).

Многие задачи требуют поиска в контейнере, а не просто совершения какой-либо операции над каждым элементом. Например, алгоритм ***find*** (§ 18.15.2) предоставляет удобный способ поиска конкретного значения. Более общим вариантом этой задачи является поиск значения, удовлетворяющего некоторому условию. Например, нам может потребоваться найти в ассоциативном массиве первый элемент, больший ***42***. Ассоциативный массив — это последовательность пар (ключ, значение), поэтому мы будем осуществлять поиск в последовательности такой пары ***pair<string, int>***, где ***int*** больше ***42***.

```
bool gt_42 (const pair<const string, int>& r)    // больше 42?
{
    return r.second > 42;
}

void f(map<string, int>& m)
{
    typedef map<string, int>::const_iterator MI;
    MI i = find_if(m.begin (), m.end (), gt_42);
    // ...
}
```

Для разнообразия можно подсчитать количество слов с частотой вхождения больше ***42***:

```
void g (const map<string, int>& m)
{
    int c42 = count_if (m.begin (), m.end (), gt_42);
    // ...
}
```

Функции типа ***gt_42*** (), используемые для управления алгоритмом, называются *предикатами*. Предикат вызывается для каждого элемента и возвращает логическое значение, используемое алгоритмом для принятия решения о выполнении необходимого действия. Например, функция ***find_if*** () осуществляет поиск до тех пор, пока ее предикат не возвратит значение ***true***, означающее, что нужный элемент найден. Аналогично, функция ***count_if*** () производит подсчет столько раз, сколько ее предикат равен ***true***.

В стандартной библиотеке имеется несколько полезных предикатов и несколько шаблонов для создания новых предикатов (§ 18.4.2).

3.8.5. Алгоритмы, использующие функции-члены

Многие алгоритмы применяют функцию к каждому элементу последовательности. Например, в § 3.8.4

```
for_each (ii, eos, record);
```

вызывает функцию ***record*** () для каждой строки, прочтенной из потока ввода.

Часто мы работаем с контейнером указателей и нам хочется вызывать функции-члены объектов, на которые они ссылаются, а не глобальные функции над указателем. Например, нам может понадобиться вызвать функцию-член ***Shape::draw*** () для каждого элемента списка ***list<Shape*>***. Для решения данной конкретной задачи мы

можем просто написать функцию, не являющуюся членом, которая будет вызывать функцию-член. Например:

```
void draw (Shape* p)
{
    p->draw ();
}
void f (list<Shape*>& sh)
{
    for_each (sh.begin (), sh.end (), draw);
}
```

Обобщая эту технику, приходим к:

```
void g (list<Shape*>& sh)
{
    for_each (sh.begin (), sh.end (), mem_fun (&Shape::draw));
}
```

Стандартный библиотечный шаблон ***mem_fun*** () (§ 18.4.4.2) принимает в качестве аргумента указатель на функцию-член (§ 15.5) и возвращает нечто, что можно вызвать для указателя на класс члена. Результат ***mem_fun*** (&***Shape***::***draw***) принимает аргумент ***Shape**** и возвращает тоже, что и ***Shape***::***draw*** ().

Механизм ***mem_fun*** () важен, так как позволяет использовать стандартные алгоритмы для контейнеров с полиморфными объектами.

3.8.6. Алгоритмы стандартной библиотеки

Что такое алгоритм? Вот общее определение алгоритма: «конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: Конечность... Определенность .. Ввод... Вывод... Эффективность» [Knuth, 1968, § 1.1]. В контексте стандартной библиотеки C++ алгоритм — это набор шаблонов, работающих с последовательностями элементов.

Стандартная библиотека содержит десятки алгоритмов. Алгоритмы определены в пространстве имен ***std*** и представлены в заголовочном файле <***algorithm***>. Вот список некоторых алгоритмов, которые я счел особенно полезными:

Избранные стандартные алгоритмы	
for_each ()	Вызвать функцию для каждого элемента (§ 18.5.1)
find ()	Найти первое вхождение аргументов (§ 18.5.2)
find_if ()	Найти первое соответствие предикату (§ 18.5.2)
count ()	Сосчитать число вхождений элемента (§ 18.5.3)
count_if ()	Сосчитать число соответствий предикату (§ 18.5.3)
replace ()	Заменить элемент новым значением (§ 18.6.4)
replace_if ()	Заменить элемент, соответствующий предикату, новым значением (§ 18.6.4)
copy ()	Скопировать элементы (§ 18.6.1)
unique_copy ()	Скопировать только различные элементы (§ 18.6.1)

sort ()	Отсортировать элементы (§ 18.7.1)
equal_range ()	Найти диапазон всех элементов с одинаковыми значениями (§ 18.7.2)
merge ()	Слияние отсортированных последовательностей (§ 18.7.3)

Эти и многие другие алгоритмы (см. главу 18) можно применять к контейнерам, строкам ***string*** и встроенным массивам.

3.9. Математические вычисления

Как и C, C++ не разрабатывался специально для решения вычислительных задач. Однако, с помощью C++ производится достаточно много численных расчетов и стандартная библиотека отражает этот факт.

3.9.1. Комплексные числа

Стандартная библиотека поддерживает семейство типов комплексных чисел, в том числе класс ***complex***, описанный в § 2.5.2. Чтобы обеспечить поддержку комплексных чисел, где скалярными значениями являются числа одинарной точности с плавающей точкой (***float***), двойной точности (***double***) и т. д., класс ***complex*** стандартной библиотеки сделан шаблоном:

```
template<class scalar> class complex {
public:
        complex (scalar re, scalar im);
        //...
};
```

Для комплексных чисел поддерживаются обычные арифметические операции и наиболее общие математические функции. Например:

```
// степень — стандартная функция от комплексного числа
template<class C> complex<C> pow (const complex<C>&, int);

void f(complex<float> fl, complex<double> db)
{
        complex<long double> ld = fl + sqrt (db);
        db += fl * 3;
        fl = pow (1/fl, 2);
        //...
}
```

Подробности см. в § 22.5.

3.9.2. Векторная арифметика

Тип ***vector***, описанный в § 3.7.1, разработан с целью предоставления общего механизма хранения значений, гибкого и согласованного с архитектурой контейнеров, итераторов и алгоритмов. Однако, он не поддерживает математические векторные операции. Добавить такие операции к типу ***vector*** достаточно просто, но его общность и гибкость препятствуют оптимизации, которой часто придается решающее значение

в серьезных вычислительных задачах. Поэтому в стандартной библиотеке имеется векторный тип, который называется ***valarray***. Он не настолько общий, но более удобен для оптимизации при вычислениях:

```
template<class T> class valarray {
        //...
        T& operator[] (size_t);
        //...
};
```

Тип *size_t* означает целый тип без знака, используемый реализацией для индексирования массива.

Для ***valarray*** реализованы обычные математические операции и большинство распространенных математических функций. Например:

```
// стандартная функция из <valarray>, которая вычисляет модуль
template<class T> valarray<T> abs (const valarray<T>&);

void f(valarray<double>& a1, valarray<double>& a2)
{
        valarray<double> a = a1*3.14 + a2/a1;
        a2 += a1*3.14;
        a = abs (a);
        double d = a2[7];
        //...
}
```

Подробности см. в § 22.4.

3.9.3. Поддержка базовых численных операций

Обычно стандартная библиотека содержит наиболее распространенные математические функции — вроде ***log*** (), ***pow*** () или ***cos*** () — для типов с плавающей точкой (см. § 22.3). Кроме того, для встроенных типов предусмотрены классы, описывающие их свойства, такие как максимальное значение экспоненты для ***float*** (см. § 22.2).

3.10. Средства стандартной библиотеки

Можно классифицировать средства стандартной библиотеки следующим образом:

[1] Базовая поддержка средств языка времени выполнения (например, выделение памяти и определение типа во время выполнения); см. § 16.1.3.

[2] Стандартная библиотека C (с очень незначительными изменениями для минимизации нарушений системы типов); см. § 16.1.2.

[3] Строки и потоки ввода/вывода (с поддержкой национальных алфавитов и локализации); см. главы 20 и 21.

[4] Контейнеры (такие как ***vector***, ***list*** и ***map***) и алгоритмы, их использующие (такие как проход с выполнением, сортировка и слияние); см. главы 16, 17, 18 и 19.

[5] Поддержка численных расчетов (комплексные числа, вектора с арифметическими операциями, BLAS-подобные и обобщенные срезы); см. главу 22.

Главным критерием для включения класса в библиотеку было то, что он должен: использоваться в какой-то мере практически каждым программистом на C++ (и нович-

ками, и экспертами); быть выражен в достаточно общем виде, но при этом не терять в производительности по сравнению с более простой формой; быть таковым, чтобы его простому использованию можно было легко научиться. Важно то, что стандартная библиотека C++ содержит наиболее общие фундаментальные структуры данных вместе с фундаментальными алгоритмами их обработки.

Каждый алгоритм работает с каждым контейнером без каких-либо преобразований. Такие среды разработки, обычно называемые STL (Standard Template Library, стандартная библиотека шаблонов) [Stepanov, 1994], можно расширить в том смысле, что пользователи имеют возможность легко создавать контейнеры и алгоритмы, дополняющие стандартные и работающие почти также, как они.

3.11. Советы

[1] Не изобретайте колесо — пользуйтесь библиотеками.
[2] Не верьте в чудеса; разберитесь, что делают ваши библиотеки, как они это делают и какова цена их использования.
[3] Когда у вас есть выбор, отдавайте предпочтение стандартной библиотеке.
[4] Не думайте, что стандартная библиотека идеально подходит для всех задач.
[5] Не забывайте использовать директиву *#include* для указания файлов с нужными вам средствами; § 3.3.
[6] Помните, что средства стандартной библиотеки определены в пространстве имен *std*; § 3.3.
[7] Пользуйтесь *string* вместо *char**; § 3.5, § 3.6.
[8] Если сомневаетесь — пользуйтесь вектором с проверкой диапазона (таким как *Vec*); § 3.7.2.
[9] Используйте *vector<T>*, *list<T>* и *map<T>*, а не *T*[];§ 3.7.1, § 3.7.3, § 3.7.4.
[10] При добавлении элементов в контейнер пользуйтесь *push_back* () или *back_inserter* (); § 3.7.3, § 3.8.
[11] Используйте *push_back* () с *vector*, а не *realloc* () со встроенным массивом; § 3.8.
[12] Перехватывайте типичные исключения в функции *main* (); § 3.7.2.

ЧАСТЬ ПЕРВАЯ

Основные средства

В этой части описываются встроенные типы языка C++ и основные средства построения на их основе программ. Характеризуется подмножество C++, совместимое с языком C. Анализируются дополнительные средства C++, поддерживающие традиционные стили программирования. Обсуждаются базовые возможности составления программ из логических и физических частей.

4

Типы и объявления

Не соглашайся ни на что,
кроме совершенства!
— Анонимный автор

Совершенство достигается
только к моменту полного краха.
— К. Н. Паркинсон

Типы — фундаментальные типы — логические значения — символы — символьные литералы — целые — целые литералы — типы с плавающей точкой — литералы с плавающей точкой — размеры — ***void*** — перечисления — объявления — имена — область видимости — инициализация — объекты — ***typedef*** — советы — упражнения.

4.1. Типы

Рассмотрим выражение:

```
x = y + f(2);
```

Чтобы это выражение имело смысл в программе на C++, имена ***x***, ***y*** и ***f*** должны быть подходящим образом объявлены. То есть программист должен указать, что существуют некие сущности с именами ***x***, ***y*** и ***f***, и для их типов имеют смысл операции = (присваивание), + (сложение) и () (вызов функции).

Каждое имя (идентификатор) в программе на C++ имеет связанный с ним тип. Он определяет, какие операции применимы к имени (то есть к сущности, связанной с этим именем) и как эти операции интерпретируются. Например, объявления

```
float x;        // x — переменная с плавающей точкой
int y = 7;      // y — целая переменная с начальным значением 7
float f(int);   // f — функция с аргументом целого типа,
                // возвращающая число с плавающей точкой
```

сделают предыдущий пример осмысленным. Так как переменная ***y*** объявлена как целая, ее можно присваивать, использовать в арифметических выражениях и т. д. С другой стороны, ***f*** объявлена как функция с аргументом целого типа, поэтому ее можно вызывать с подходящим аргументом.

В этой главе описываются фундаментальные типы (§ 4.1.1) и объявления (§ 4.9). Примеры, приведенные в ней, просто демонстрируют свойства языка; не подразуме-

вается, что они делают что-нибудь полезное. Более подробные и реалистичные примеры оставлены до следующих глав, когда большая часть языка будет уже описана. Здесь же просто изложены базовые элементы, из которых строится программа на C++. Для написания законченного реального проекта на C++ и, особенно, для того, чтобы понимать код, написанный другими, вам необходимо знать эти элементы, освоить терминологию и овладеть синтаксисом. Тем не менее, полное понимание всех деталей, упомянутых в этой главе, не является обязательным требованием для чтения последующих глав. Следовательно, при желании вы можете бегло просмотреть данную главу, обратив внимание на основные концепции, и вернуться к ней позднее при необходимости.

4.1.1. Фундаментальные типы

В C++ имеется набор фундаментальных типов, отражающих характерные особенности организации памяти большинства компьютеров и наиболее распространенные способы хранения данных:

§ 4.2 Логический тип (***bool***)
§ 4.3 Символьные типы (например, ***char***)
§ 4.4 Целые типы (например, ***int***)
§ 4.5 Типы с плавающей точкой (например, ***double***)

Кроме того, пользователь может определить

§ 4.8 Перечислимые типы для представления значений из конкретного множества (***enum***)

Также имеется

§ 4.7 Тип ***void***, используемый для указания на отсутствие информации.

Кроме этих типов, мы можем сконструировать и другие:

§ 5.1 Указатели (например, ***int****)
§ 5.2 Массивы (например, ***char***[])
§ 5.5 Ссылки (например, ***double***&)
§ 5.7 Структуры данных и классы (глава 10)

Логические, символьные и целые типы вместе называются *интегральными типами*. Интегральные типы, вместе с типами с плавающей точкой, называются *арифметическими типами*. Перечисления и классы (глава 10) называются *типами, определяемыми пользователем* (или пользовательскими типами), потому что в отличие от фундаментальных типов, которые можно использовать без предварительного объявления, они должны быть определены пользователем. Другие типы называются *встроенными типами*.

Интегральные типы и типы с плавающей точкой могут иметь различные размеры, предоставляя программисту возможность выбора количества используемой памяти, точности и допустимого диапазона значений (§ 4.6). Предполагается, что в компьютере имеются байты для хранения символов, слова для хранения целых чисел и выполнения арифметических операций с ними, нечто, подходящее для операций с плавающей точкой, а также адреса для обращения к этим сущностям. Фундаментальные типы C++ совместно с указателями и массивами предоставляют программисту понятия машинного уровня, но в форме, достаточно независимой от реализации.

В большинстве приложений можно обойтись ***bool*** для логических значений, ***int*** — для целых, ***char*** — для символов и ***double*** — для чисел с плавающей точкой. Остальные

фундаментальные типы являются вариациями, предназначенными для оптимизации и решения других специальных задач. Ими лучше не пользоваться, пока не возникла острая необходимость. Однако их нужно знать, чтобы читать код на C и C++.

4.2. Логические типы

Логические переменные (***bool***) могут принимать одно из двух значений: *истина* (***true***) или *ложь* (***false***). Логические переменные используются для выражения результатов логических операций. Например:

```
void f(int a, int b)
{
        bool b1 = a==b;  // = означает присваивание, == — проверка на равенство
// ...
}
```

Если ***a*** и ***b*** имеют одинаковые значения, ***b1*** будет равно ***true***, в противном случае — ***false***.

Распространенным примером использования логического типа является анализ выполнения некоторого условия (предиката), реализуемый в виде вызова функции. Например:

```
bool is_open (File*);

bool greater (int a, int b) { return a>b; }
```

По определению, ***true*** имеет значение ***1*** при преобразовании к целому типу, а ***false*** — ***0***. И наоборот, целые можно неявно преобразовать в логические значения; при этом ненулевые целые преобразуются в ***true***, а ноль — в ***false***. Например:

```
bool b = 7;       // bool(7) означает true; b принимает значение true
int i = true;     // int(true) равно 1; i примет значение 1
```

В арифметических и логических выражениях логические значения преобразуются в целые (***int***); арифметические и битовые логические операции выполняются над преобразованными величинами. Если результат приводится обратно к логическому типу, ***0*** преобразуется в ***false***, а ненулевое значение — в ***true***.

```
void g()
{
        bool a = true;
        bool b = true;

        bool x = a+b;     // a+b равно 2, поэтому x становится true
        bool y = a|b;     // a|b равно 1, поэтому y становится true
}
```

Указатель можно неявно преобразовать в ***bool*** (§ B.6.2.5). Ненулевой указатель принимает значение ***true***, нулевой — ***false***.

4.3. Символьные типы

В переменной типа ***char*** может храниться один из символов, имеющихся в наборе символов реализации. Например:

```
char ch = 'a';
```

Практически всегда на объект типа *char* отводится 8 бит, так что существуют 256 различных значений этого типа. Как правило, набор символов основывается на ISO-646. К числу таких наборов относится ASCII, включающий в себя символы вашей клавиатуры. Множество проблем возникает из-за того, что этот набор символов стандартизован лишь частично (§ B.3).

Между наборами символов, поддерживающими различные национальные алфавиты, и даже между наборами символов, поддерживающими один и тот же национальный алфавит, имеются большие отличия. Нас интересует только то, каким образом это влияет на правила C++. Более масштабные и интересные проблемы, связанные с программированием в средах, поддерживающих несколько языков или несколько наборов символов, выходят за рамки этой книги, хотя и упоминаются в некоторых местах (§ 20.2, § 21.7, § B.3.3).

Можно с достаточной степенью уверенности предположить, что набор символов конкретной реализации включает в себя десятичные цифры, 26 букв английского алфавита и некоторые основные знаки пунктуации. Небезопасно полагать, что 8-битный набор символов содержит только 127 символов (некоторые наборы содержат 256 символов), что нет символов, кроме английских букв (в большинстве европейских языков их больше), что символы алфавита непрерывны (в стандарте EBCDIC между *i* и *j* имеется разрыв) или что будут реализованы все символы, используемые в синтаксисе C++ (в некоторых национальных кодировках отсутствуют символы {, }, [,], |, \ — см. § B.3.1). Там, где это только возможно, следует избегать предположений о форме представления объектов. Это общее правило применимо даже к символам.

Каждая символьная константа имеет числовое значение. Например, значением *'b'* в наборе символов ASCII является *98*. Приведем пример программы, которая будет выводить целое значение введенного символа:

```
#include <iostream>

int main ()
{
        char c;
        std::cin >> c;
        std::cout << "значением '" << c << "' является " << int (c) << '\n';
}
```

Выражение *int (c)* возвращает целое значение, соответствующее символу *'c'*. Вследствие возможности преобразования *char* в целое возникает следующая проблема: как интерпретировать *char* — со знаком или без? 256 значений, представляемых восемью битами, можно интерпретировать как значения от 0 до 255, либо как значения от −128 до 127. К сожалению, выбор зависит от конкретной реализации (§ B.1, § B.3.4). В C++ имеется два способа явного указания диапазона: *signed char* означает диапазон от −128 до 127, а *unsigned char* — от 0 до 255. К счастью, разница касается только значений вне диапазона 0−127, а в последний попадают все наиболее употребительные символы.

Значения вне разрешенного диапазона, хранимые в *char*, могут привести к тонким проблемам переносимости. Обратитесь к разделу § B.3.4, если вам необходимо использовать более одного типа *char*, или если вы храните целые числа в переменных типа *char*.

Для хранения символов из больших наборов, таких как Unicode, имеется тип *wchar_t*. Это специальный тип. Размер *wchar_t* зависит от реализации; он достаточно велик для представления всего набора символов, поддерживаемого данной реа-

лизацией (см. § 21.7, § B.3.3). Странное имя (*wchar_t*) досталось по наследству от C. В C *wchar_t* реализован как *typedef* (§ 4.9.7), а не как встроенный тип. Суффикс *_t* использовался, чтобы указать на определение через *typedef*.

Обратите внимание, что символьные типы являются интегральными (§ 4.1.1), так что к ним можно применять арифметические и логические операции (§ 6.2).

4.3.1. Символьные литералы

Символьным литералом (символьной константой) называется символ, заключенный в одиночные кавычки. Например, *'a'* или *'O'*. Типом символьного литерала является *char*. Такие символьные литералы в действительности являются символическими константами, обозначающим целые значения символов из набора символов на компьютере, на котором будет выполняться программа. Например, если вы запускаете программу на машине, использующей набор ASCII, значением *'O'* будет 48. Использование символьных литералов вместо десятичных обозначений делает программы более переносимыми. Несколько символов имеют специальное назначение. Они записываются с помощью символа обратной косой черты \. Например, *'\n'* является символом перевода строки, а *'\t'* — горизонтальной табуляцией. Подробности см. в § B.3.2.

Символьные литералы из расширенного набора записываются в виде *L'ab'*, при этом количество символов между одиночными кавычками и их значение зависят от реализации и соответствуют типу *wchar_t*. Литералы из расширенного символьного набора имеют тип *wchar_t*.

4.4. Целые типы

Также как и *char*, каждый целый тип может быть представлен в одном из трех видов: «просто» *int*, *signed int* и *unsigned int*. Кроме того, целые могут быть трех размеров: *short int*, «просто» *int* и *long int*. Вместо *long int* можно писать просто *long*. Аналогично, *short* является синонимом для *short int*, *unsigned* — для *unsigned int* и *signed* — для *signed int*.

Типы *unsigned* (без знака) идеально подходят для задач, в которых память интерпретируется как массив битов. Использование *unsigned* вместо *int* с целью заработать лишний бит для представления положительных целых почти всегда оказывается неудачным решением. Использование же объявления *unsigned* для гарантии того, что целое будет неотрицательным, почти никогда не сработает из-за правил неявного преобразования типов (§ B.6.1, § B.6.2.1).

В отличие от «просто» *char*, «просто» *int* всегда знаковые. Тип *signed int* (целое со знаком) является более точным синонимом «просто» *int*.

4.4.1. Целые литералы

Целые литералы бывают: десятичные, восьмеричные, шестнадцатеричные и символьные (§ A.3). Наиболее часто используются десятичные литералы и выглядят они так, как вы и ожидаете:

```
0 1234 976 12345678901234567890
```

Компилятор должен выдавать предупреждающее сообщение о литералах, слишком длинных для внутреннего представления.

Литерал, который начинается с нуля и символа «*x*» (*0x*), является шестнадцатеричным числом. Литерал, который начинается с нуля, за которым следуют цифры, является восьмеричным числом. Например:

десятичные:		*2*	*63*	*83*
восьмеричные:	*00*	*02*	*077*	*0123*
шестнадцатеричные:	*0x0*	*0x2*	*0x3f*	*0x53*

Буквы *a*, *b*, *c*, *d*, *e* и *f*, либо их эквиваленты в верхнем регистре, используются для обозначения *10*, *11*, *12*, *13*, *14* и *15* соответственно. Восьмеричная и шестнадцатеричная формы наиболее полезны для записи цепочек битов. Использование этих форм для записи обычных чисел может привести к неприятным сюрпризам. Например, на машине, где *int* реализован комплиментарным (дополнительным) 16-битным словом, *0xffff* означает десятичное *–1*. Если же под целое отводится больше бит, *0xffff* будет равно *65535*. Для явной записи литералов без знака (*unsigned*), можно использовать суффикс *U*. Аналогично, суффикс *L* можно использовать для явной записи *long*-литералов. Например, *3* является константой типа *int*, *3U* — *unsigned int* и *3L* — *long int*. Если суффикс отсутствует, компилятор присваивает целому литералу подходящий тип, основываясь на его значении и размере целых в данной реализации (§ B.4).

Неплохо ограничить использование неочевидных констант несколькими хорошо откомментированными *const* (§ 5.4) или инициализаторами перечислений (§ 4.8).

4.5. Типы с плавающей точкой

Типы с плавающей точкой представляют числа с плавающей точкой. Как и целые, типы с плавающей точкой представлены тремя размерами: *float* (одинарной точности), *double* (двойной точности) и *long double* (расширенной точности).

Точный смысл каждого типа зависит от реализации. Выбор нужной точности в реальных задачах требует хорошего понимания природы машинных вычислений с плавающей точкой. Если у вас его нет, либо проконсультируйтесь с кем-нибудь, либо изучите проблему сами, либо используйте *double* и надейтесь на лучшее.

4.5.1. Литералы с плавающей точкой

По умолчанию литералы с плавающей точкой являются константами типа *double*. Как обычно, компилятор должен вывести предупреждение, если литерал слишком длинен для его правильного представления. Вот несколько примеров литералов с плавающей точкой:

```
1 23   .23   0.23   1.   1.0   1.2e10   1.23e-15
```

Обратите внимание, что в записи литералов с плавающей точкой не должно быть пробелов. Например, *65.43 e-21* не является литералом, и будет воспринято, скорее всего, как набор четырех различных лексических единиц (что вызовет синтаксическую ошибку):

```
65.43   e   -   21
```

Если требуется литерал типа *float*, можно явно определить его с помощью суффикса *f* (или *F*):

```
3.14159265f   2.0f   2.997925F   2.9e-3f
```

Если требуется литерал типа *long double*, можно явно определить его с помощью суффикса *l* (или *L*):

```
3.14159265l   2.0l   2.997925L   2.9e-3l
```

4.6. Размеры

Некоторые свойства фундаментальных типов C++, например, размер переменной типа ***int***, зависят от реализации (§ B.2). Я всегда указываю на эти зависимости и часто рекомендую избегать их или по крайней мере предпринимать шаги для сведения к минимуму их воздействия. Почему следует обращать на это внимание? Люди, пишущие программы в нескольких системах или пользующиеся несколькими компиляторами, просто вынуждены уделять этому огромное внимание, поскольку в противном случае им придется тратить время на поиск и исправление нетривиальных ошибок. Люди, которые заявляют, что их не волнует проблема переносимости, обычно работают в одной системе и думают примерно так: «C++ это то, что понимает мой компилятор на моем компьютере». Я считаю эту точку зрения узкой и недальновидной. Если вы написали удачную программу, вполне возможно ее захотят использовать в другой среде и кому-то придется искать и устранять проблемы, связанные с нюансами реализации. Кроме того, часто возникает потребность скомпилировать программу другими компиляторами в той же самой среде и даже последующие версии вашего любимого компилятора могут интерпретировать некоторые вещи по-другому. Намного проще понять и ограничить влияние вещей, зависящих от реализации, непосредственно во время написания программы, чем пытаться разобраться с возникающими проблемами «потом».

Ограничить воздействие реализации относительно просто. Гораздо сложнее устранить специфику библиотечных средств, зависящих от системы. Одним из способов решения этой проблемы является использование стандартных библиотек, когда это возможно.

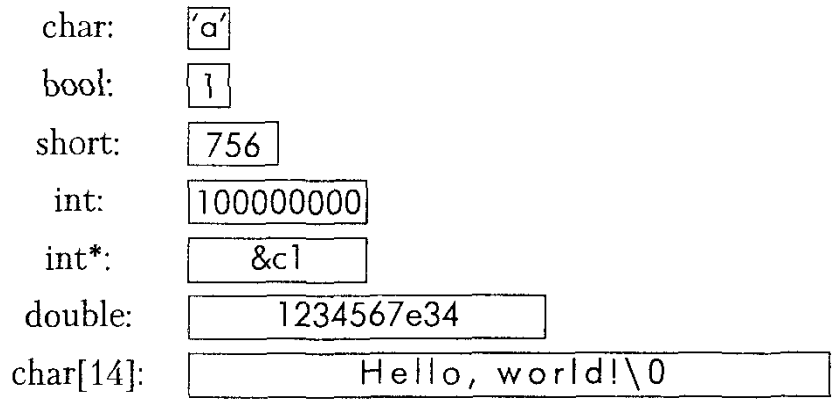
Целью существования более одного целого типа, нескольких беззнаковых типов и нескольких вариантов чисел с плавающей точкой является предоставление программисту возможности эффективно использовать аппаратные средства. На многих машинах объем памяти, время доступа и скорость вычисления существенно зависит от выбора типа. Если вы знаете архитектуру машины, как правило, несложно выбрать подходящий целый тип для конкретной переменной. Гораздо сложнее написать реально переносимый код.

Размеры объектов в C++ выражаются в единицах размера ***char***. Таким образом, размер ***char*** по определению равен ***1***. Размер объекта или типа можно получить с помощью оператора ***sizeof*** (§ 6.2). По поводу размеров фундаментальных типов гарантируется следующее:

$$1 \equiv sizeof(char) \leqslant sizeof(short) \leqslant sizeof(int) \leqslant sizeof(long)$$
$$1 \leqslant sizeof(bool) \leqslant sizeof(long)$$
$$sizeof(char) \leqslant sizeof(wchar\_t) \leqslant sizeof(long)$$
$$sizeof(float) \leqslant sizeof(double) \leqslant sizeof(long\ double)$$
$$sizeof(N) \equiv sizeof(signed\ N) \equiv sizeof(unsigned\ N)$$

где ***N*** может быть ***char***, ***short int***, ***int*** или ***long int***. Кроме того гарантируется, что для представления ***char*** используется по меньшей мере 8 бит, для представления ***short*** — по меньшей мере 16 бит и для ***long*** — по меньшей мере 32 бита. Переменная типа ***char*** может хранить любой символ из машинного набора символов.

Приведем графическое представление некоторого набора фундаментальных типов и строки:



При данном масштабе (0.2 дюйма на байт), мегабайт памяти растянется на три мили (пять километров).

Предполагается, что в конкретной реализации размер типа ***char*** будет выбран наиболее подходящим для хранения и манипулирования символами на данном компьютере; обычно байта (8 бит) достаточно. Подобным же образом размер типа ***int*** будет выбран наиболее подходящим для хранения целых чисел и выполнения операций над ними на данном компьютере; обычно это 4-байтное слово (32 бита). Неразумно предполагать что-либо кроме этого. Например, есть машины с 32-битным типом ***char***.

Если возникнет необходимость, значения, зависящие от конкретной реализации, можно получить из *<limits>* (§ 22.2). Например:

```
#include <limits>
#include <iostream>

int main ()
{
        std..cout << "наибольшее число с плавающей точкой == "
        << std::numeric_limits  max () << ", mun char знаковый == "
        << std.::numeric_limits.::is_signed << '\n';
}
```

Фундаментальные типы можно свободно смешивать в операциях присваивания и в выражениях. Там где это возможно, значения преобразовываются без потери информации (§ B.6).

Если значение *v* может быть точно представлено в переменной типа ***T***, преобразование *v* в ***T*** не ведет к потере информации, и проблем не возникает. Лучше избегать преобразований типов, которые приводят к потере информации (§ B.6.2.6).

Для того чтобы довести до конца большой проект и в особенности для того, чтобы понимать реальный код, написанный другими, необходимо знать правила неявных преобразований. Однако для чтения следующих глав этого не требуется.

4.7. Тип void

Синтаксически тип ***void*** является фундаментальным типом. Однако, его можно использовать только как часть более сложного типа, так как объектов типа ***void*** не существует. Этот тип используется либо для указания на то, что функция не возвращает значения, либо в качестве базового типа для указателей на объекты неизвестного типа. Например:

```
void x,      // ошибка: не существует объектов типа void
void f(),    // функция f не возвращает значение (§ 7.3)
void* pv,    // указатель на объект неизвестного типа (§ 5.6)
```

При объявлении функции вы должны указать тип возвращаемого значения. Логично предположить, что для указания того, что функция не возвращает значение, нужно просто опустить тип. Это, однако, сделало бы синтаксис менее единообразным (см. приложение А) и противоречило бы правилам языка С. Как следствие, в качестве «псевдотипа возвращаемого результата» используется тип ***void***, указывающий на отсутствие возвращаемого значения.

4.8. Перечисления

Тип ***enum*** (перечисление) задает набор значений, определяемый пользователем. После своего определения перечисление используется почти так же, как и целые типы.

В качестве элементов перечисления можно определить именованные целые константы. Например:

```
enum { ASM, AUTO, BREAK };
```

определяет три целые константы, называемые элементами перечисления, и присваивает им значения. По умолчанию, элементам перечисления присваиваются значения начиная с нуля; значение для каждого следующего элемента увеличивается на единицу. Таким образом, ***ASM==0***, ***AUTO==1*** и ***BREAK==2***. Перечислению можно присвоить имя. Например:

```
enum keyword { ASM, AUTO, BREAK },
```

Каждое перечисление является отдельным типом. Типом элемента перечисления является само перечисление. Например, ***AUTO*** имеет тип ***keyword***.

Объявление переменной типа ***keyword***, а не простого ***int***, подсказывает пользователю и компилятору, как предполагается употреблять эту переменную. Например, в случае

```
void f(keyword key)
{
    switch (key){
    case ASM:
        // некоторые действия
        break;
    case BREAK:
        // некоторые действия
        break;
    }
}
```

компилятор может выдать предупреждающее сообщение о том, что проверены только два значения ***keyword*** из трех возможных.

Перечисление можно инициализировать константным[1] выражением (§ В.5) интегрального типа (§ 4.1.1). Диапазон значений перечисления определяется следующим образом. Пусть n — максимальное возможное значение элемента перечисления. Пусть m такое минимальное целое, что 2^m-1 больше или равно n. Тогда верхняя граница диапазона равна 2^m-1. Если наименьший элемент имеет неотрицательное значение, нижняя граница диапазона равняется нулю. Если наименьшее значение элемента отрицательно,

[1] Мы используем прилагательное «константный», чтобы подчеркнуть связь с модификатором ***const***. — *Примеч. ред.*

нижней границей диапазона является наименьшая ближайшая отрицательная степень двойки плюс *1*. Таким образом, диапазон определяется минимальным количеством бит, требуемым для представления значений всех элементов перечисления. Например:

```
enum e1 { dark, light };                    // диапазон 0:1
enum e2 { a = 3, b = 9 };                   // диапазон 0:15
enum e3 { min = -10, max = 1000000 };       // диапазон -1048576:1048575
```

Значение интегрального типа можно явно привести к типу перечисления. Если значение находится вне пределов диапазона, результат преобразования не определен. Например:

```
enum flag { x = 1, y = 2, z = 4, e = 8 }          // диапазон 0:15

flag f1 = 5,                // ошибка типа: 5 не принадлежит типу flag
flag f2 = flag(5);          // правильно — flag (5) имеет тип flag
                            // и находится в пределах диапазона flag
flag f3 = flag(z|e);        // правильно — flag (12) имеет тип flag
                            // и находится в пределах диапазона flag
flag f4 = flag(99);         // не определено; 99 находится
                            // вне пределов диапазона flag
```

Последнее присваивание демонстрирует причину отсутствия неявного преобразования целого в перечисление. Большинство целых значений не имеет представления в произвольно взятом конкретном перечислении.

Понятие диапазона значений перечисления в C++ отличается от понятия перечисления в языках типа Pascal. Несмотря на это, примеры манипулирования битами, требующие, чтобы значения вне набора элементов перечисления были корректно определены, имеют долгую историю в C и C++.

Размер (*sizeof*) перечисления является размером некоторого интегрального типа, который в состоянии содержать весь диапазон значений перечисления. Результат не больше, чем *sizeof(int)* при условии, что элементы перечисления представимы в виде *int* или *unsigned int*. Например, *sizeof(e1)* может равняться *1* или *4*, но не *8* на машине, где *sizeof(int)==4*.

По умолчанию при выполнении арифметических операций перечисления преобразуются в целые (§ 6.2). Перечисления являются типами, определяемыми пользователем, так что для них можно вводить свои собственные операции, например ++ или << (§ 11.2.3).

4.9. Объявления

Прежде чем имя (идентификатор) может быть использовано в программе на C++, оно должно быть объявлено. То есть должен быть указан тип имени, чтобы компилятор знал, на сущность какого вида ссылается имя. Приведем несколько примеров, иллюстрирующих различные объявления:

```
char ch,
string s,
int count = 1,
const double pi = 3.1415926535897932385,
extern int error_number,
```

```
const char* name = "Njal";
const char* season[] = { "весна", "лето", "осень", "зима" };

struct Date { int d, m, y; };
int day (Date* p) { return p->d; }
double sqrt (double);
template<class T> T abs (T a) { return a<0 ? -a : a; }

typedef complex<short> Point;
struct User;
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

Как видно из этих примеров, объявления могут делать нечто большее, чем просто связывание типа с именем. Большинство *объявлений* являются еще и *определениями*, то есть они определяют некую сущность, которая соответствует имени. Для переменной ***ch*** этой сущностью является подходящее количество памяти; память будет выделена. Для ***day*** — это функция. Для константы ***pi*** — значение ***3.14159265358979323***. Сущностью ***Date*** является новый тип. Для ***Point*** — тип ***complex<short>***, так что ***Point*** становится синонимом ***complex<short>***. Из всех приведенных выше объявлений только

```
double sqrt (double);
extern int error_number;
struct User;
```

не являются определениями. То есть сущности, на которые они ссылаются, должны быть определены где-то в другом месте. Код (тело) функции ***sqrt*** должен быть задан каким-то другим объявлением, память для целой переменной ***error_number*** выделяется некоторым другим объявлением ***error_number***, и какое-то другое объявление типа ***User*** должно определить как выглядит этот тип. Например:

```
double sqrt (double d) { /* ... */ }
int error_number = 1;
struct User { /* ... */ };
```

В программе на C++ для каждого имени должно быть ровно одно определение (о действии директивы ***#include*** см. в § 9.2.3). Объявлений же может быть несколько. Все объявления некой сущности должны согласовываться по типу этой сущности. Поэтому следующий фрагмент содержит две ошибки:

```
int count;
int count;                    // ошибка: повторное определение

extern int error_number;
extern short error_number;  // ошибка: несоответствие типов
```

Следующий фрагмент кода не содержит ошибок (об использовании ***extern*** см. в § 9.2):

```
extern int error_number;
extern int error_number;
```

Некоторые определения указывают «значение» определяемой сущности. Например:

```
struct Date { int d, m, y; };
typedef complex<short> Point;
```

```
int day (Date* p) { return p->d, }
const double pi = 3 1415926535897932385,
```

Для типов, шаблонов, функций и констант «значение» является постоянным. Для неконстантных типов данных начальное значение может быть позднее изменено. Например:

```
void f()
{
        int count = 1,
        const char* name = "Bjarne",      // name является переменной,
                                          // указывающей на константу (§ 5 4.1)
        // ...
        count = 2,
        name = "Marian",
}
```

В приведенных в самом начале раздела определениях только в двух случаях

```
char ch,
string s,
```

не указаны значения. Объяснения по поводу того, как и когда переменной присваивается значение по умолчанию, см. в § 4.9.5 и § 10.4.2. Любое объявление, в котором задается значение, является определением.

4.9.1. Структура объявления

Объявление состоит из четырех частей: необязательного спецификатора, базового типа, объявляющей части и, возможно, инициализатора. За исключением определений функций и пространств имен объявление заканчивается точкой с запятой. Например:

```
char* kings[] = { "Антигон", "Селевк", "Птолемей" },
```

В этом примере базовым типом является ***char***, объявляющей частью — ****kings***[], а инициализатором — = {...}.

В качестве спецификаторов могут выступать ключевые слова, такие как ***virtual*** (§ 2.5.5, § 12.2.6) и ***extern*** (§ 9.2). Они приводятся в начале объявления и описывают характеристики, не связанные с типом.

Объявляющая часть состоит из имени и, возможно, операторов объявления. Наиболее часто встречаются следующие операторы объявления (§ A.7.1):

*	*указатель*	*префикс*
*const	*константный указатель*	*префикс*
&	*ссылка*	*префикс*
[]	*массив*	*суффикс*
()	*функция*	*суффикс*

Их использование значительно упростилось бы, будь они все либо префиксами, либо суффиксами. Однако, *, [] и () разрабатывались так, чтобы отражать их смысл в выражениях (§ 6.2). Таким образом, * является префиксом, а [] и () — суффиксами. Суффиксные операторы объявления «крепче связаны» с именем, чем префиксные. Следовательно, ****kings***[] означает массив указателей на какие-либо объекты, а для определения

типов наподобие «указатель на функцию», необходимо использовать скобки (см. примеры в § 5.1). Детали см. в приложении А, где описана грамматика.

Обратите внимание: тип должен присутствовать в объявлении. Например:

```
const c = 7;                          // ошибка: не указан тип
gt(int a, int b) { return (a>b) ? a : b; }   // ошибка: не указан тип
                                      // возвращаемого значения

unsigned ui;                          // правильно — unsigned является
                                      // сокращением unsigned int
long li;                              // правильно — long является типом long int
```

Стандарт C++ отличается от ранних версий C и C++, в которых первые два объявления допустимы. Если тип не указывался, по умолчанию подразумевался ***int*** (§ Б.2). Это правило «неявного ***int***» было источником нетривиальных ошибок и недоразумений.

4.9.2. Объявление нескольких имен

Разрешается объявлять несколько имен в одном операторе объявления. Объявление просто содержит список объявителей, разделенных запятыми. Например, мы можем объявить две целые переменные следующим образом:

```
int x, y;          // int x; int y;
```

Обратите внимание, что операторы объявления применяются только к ближайшему имени и не относятся к соседним именам в объявлении. Например:

```
int* p, y;         // int* p; int y; (НЕ int* y;)
int x, *q;         // int x; int* q;
int v[10], *pv;    // int v[10], int* pv;
```

Такие конструкции делают программу менее читабельной, и их следует избегать.

4.9.3. Имена

Имя (идентификатор) состоит из последовательности букв и цифр. Первым символом должна быть буква. Символ подчеркивания _ считается буквой. В C++ нет ограничений на количество символов в имени. Однако некоторые части реализации (в частности, компоновщик) недоступны автору компилятора и они, к сожалению, иногда накладывают такие ограничения. Некоторые среды исполнения иногда требуют расширить или ограничить набор символов, допустимых в идентификаторе. Расширения (например, использование символа $ в имени) часто приводят к непереносимым программам. Ключевые слова C++ (приложение А), такие как ***new*** или ***int***, нельзя использовать в качестве имени сущности, определяемой пользователем. Примеры имен:

```
hello          this_is_a_most_unusually_long_name
DEFINED        foO        bAr        u_name        HorseSense
var0           var1       CLASS      _class        ___
```

Примеры последовательностей символов, которые нельзя использовать в качестве идентификаторов:

```
012            a fool     $sys       class 3var
pay.due        foo~bar    .name      if
```

Имена, начинающиеся с символа подчеркивания, зарезервированы для специфических нужд реализации и среды исполнения, поэтому такие имена не следует использовать в прикладных программах.

При считывании программы компилятор всегда ищет наибольшую строку символов, которая может составить имя. Поэтому ***var10*** является одним именем, а не именем ***var***, за которым следует число ***10***. Также и ***elseif*** является одним именем, а не ключевым словом ***else***, за которым следует ключевое слово ***if***.

Символы в верхнем и нижнем регистре различаются, поэтому ***Count*** и ***count*** — разные имена. Не слишком разумно выбирать имена, различающиеся только регистром. В целом лучше избегать имен, которые внешне отличаются незначительно. Например, заглавную о (***O***) и ноль (***0***) трудно отличить друг от друга. То же самое можно сказать о прописной букве L (***l***) и цифре ***1*** (единица). Следовательно, ***l0***, ***lO***, ***l1*** и ***ll*** — неудачные имена.

Имена с большой областью видимости должны быть относительно подробными и осмысленными. Например, такими: ***vector*** (вектор), ***Window_with_border*** (окно с рамкой) или ***Department_number*** (номер отдела). Напротив, код выглядит чище, если переменные с небольшой областью видимости имеют короткие привычные имена, такие как ***x***, ***i*** и ***p***. Для того чтобы ограничить область видимости, можно пользоваться классами (глава 10) и пространствами имен (§ 8.2). Полезно делать часто используемые имена короткими, а длинные оставлять для редко используемых имен. Выбирайте имена, исходя из смысла объекта, а не его представления. Например, ***phone_book*** (телефонная книга) предпочтительнее, чем ***number_list*** (список номеров), хотя телефонные номера вероятно будут храниться в списке (***list***) (§ 3.7). Выбор хороших имен является искусством.

Пытайтесь последовательно придерживаться определенного стиля при выборе имен. Например, начинайте имена типов, определяемых пользователем, из нестандартных библиотек с заглавной буквы, а имена не типов — с прописной (например, ***Shape*** и ***current_token***). Макросы записывайте заглавными буквами (если непременно нужен макрос; пишите, например, ***HACK***) и пользуйтесь знаком подчеркивания для отделения слов в идентификаторе. И все-таки сложно последовательно придерживаться какого-то одного стиля, так как обычно программы состоят из нескольких фрагментов, взятых из разных источников; каждый из них может использовать свой собственный разумный стиль. Будьте последовательны при выборе сокращений и составных имен.

4.9.4. Область видимости

Объявление вводит имя в область видимости. Это значит, что имя может использоваться только в определенной части текста программы. Для имени, объявленного в теле функции (такое имя часто называют *локальным*), область видимости начинается с места объявления имени и заканчивается в конце блока, в котором это имя объявлено. *Блоком* называется фрагмент текста, заключенный в фигурные скобки { }.

Имя называется *глобальным*, если оно объявлено вне любой функции, класса (глава 10) или пространства имен (§ 8.2). Область видимости глобальных имен простирается от места их объявления до конца файла, содержащего объявление. Объявление имени в блоке может скрыть объявление этого имени в охватывающем блоке или глобальное имя. То есть имя может быть замещено внутри блока и будет ссылаться там на другую сущность. После выхода из блока имя восстанавливает свой прежний смысл. Например:

```
int x;                  // глобальная переменная x

void f()
{
    int x;              // локальная переменная x скрывает
                        // глобальную переменную x
    x = 1;              // присваивание локальной переменной x
    {
        int x,          // «скрывает» первую локальную переменную x
        x = 2,          // присваивание второй локальной переменной x
    }
    x = 3,              // присваивание первой локальной переменной x
}

int* p = &x;            // взять адрес глобальной переменной x
```

Сокрытие имен неизбежно при написании больших программ. Однако читающий программу может не заметить того, что какое-то имя «спряталось». Ввиду того, что такие ошибки встречаются сравнительно редко, их очень трудно обнаружить. Поэтому следует свести к минимуму сокрытие имен. Хотите иметь проблемы — используйте имена типа *i* или *x* в качестве глобальных переменных или в качестве локальных переменных в больших функциях.

К скрытому глобальному имени можно обратиться с помощью оператора разрешения области видимости ::. Например:

```
int x;

void f2 ()
{
    int x = 1,    // глобальная переменная x скрыта
     x = 2;       // присваивание глобальной переменной x
    x = 2;        // присваивание локальной переменной x
     // ...
```

Не существует способа обращения к скрытой локальной переменной.

Область видимости имени начинается в точке объявления, точнее, сразу после объявителя, но перед инициализатором. Поэтому допускается использование имени в качестве инициализирующего значения для себя самого. Например:

```
void f3 ()
{
    int x = x;    // странно: присваиваем переменной x ее собственное
                  // (неинициализированное) значение
}
```

Это допустимо, но неразумно. Хороший компилятор выдаст предупреждение в случае использования переменной до того, как она будет инициализирована (§ 5.9[9]).

Можно использовать одно имя для ссылки на два различных объекта в блоке без использования оператора ::. Например:

```
int x = 11;

void f4 ()
```

```
{
    int y = x;    // глобальная x: y = 11
    int x = 22;
    y = x;        // локальная x: y = 22
}
```

Считается, что имена аргументов функции объявлены в самом внешнем блоке функции, поэтому

```
void f5 (int x)
{
    int x;        // ошибка: повторное определение
}
```

является ошибкой, так как переменная *x* дважды определяется в пределах одной и той же области видимости. Отнесение подобных ситуаций к числу запрещенных позволяет (компилятору) отловить нетривиальные, тонкие ошибки.

4.9.5. Инициализация

Если для объекта указан инициализатор, он определяет начальное значение объекта. Если инициализатор не задан, то глобальным объектам (§ 4.9.4), объектам из пространства имен (§ 8.2) и локальным статическим объектам (§ 7.1.2, § 10.2.4) (все вместе они называются *статическими объектами*) присваивается нулевое значение соответствующего типа. Например:

```
int a;      // означает int a = 0;
double d,   // означает double d = 0.0;
```

Локальные переменные (иногда называемые *автоматическими объектами*) и объекты, создаваемые в области свободной памяти (иногда называемые *динамическими объектами* или *объектами из кучи*), не инициализируются по умолчанию. Например:

```
void f()
{
    int x;        // x не имеет надежно определенного значения
    // ...
}
```

Элементы массивов и члены структур инициализируются или нет в зависимости от того, являются ли они статическими. Для типов, определяемых пользователем, может определяться инициализация по умолчанию (§ 10.4.2).

Сложные объекты требуют более одного инициализатора. Для массивов и структур используется синтаксис списков инициализаторов, принятый в C. Список значений заключается в фигурные скобки { } (§ 5.2.1, § 5.7). Для инициализации типов, определяемых пользователем, применяются конструкторы с синтаксисом вызова функции (§ 2.5.2, § 10.2.3).

Обратите внимание на то, что «пустая» пара круглых скобок () в объявлении всегда означает функцию (§ 7.1).

Примеры:

```
int a[] = { 1, 2 };        // инициализатор массива
```

```
Point z(1,2);          // инициализатор в стиле функции (вызов конструктора)
int f();               // объявление функции
```

4.9.6. Объекты и lvalue

Допускается выделять память и «использовать переменные», у которых нет имен, а также присваивать значения странно выглядящим выражениям (например, *p[a+10]=10). Соответственно, возникает потребность в названии для «чего-то в памяти». Это и есть самое простое и фундаментальное понятие объекта. А именно — *объект* есть непрерывная область памяти. *Lvalue* (именующее выражение) — это выражение, ссылающееся на объект. Исходным смыслом lvalue (от английского «left value» — значение в левой части) было «нечто, что может быть использовано в левой части оператора присваивания». Однако не каждое lvalue может использоваться в левой части оператора присваивания. Lvalue может ссылаться на константу (§ 5.5). Lvalue, которое не было объявлено константой, часто называют *модифицируемым lvalue*. Приведенное простое низкоуровневое понятие объекта не следует путать с понятиями объекта класса или объекта полиморфного типа (§ 15.4.3).

Если программист не указал явно иное (§ 7.1.2, § 10.4.8), объект, объявленный в функции, создается, когда встречается его определение, а уничтожается в момент выхода его имени из области видимости (§ 10.4.4). Такие объекты называются автоматическими. Объекты, объявленные глобально или в пространстве имен, а также *статические* объекты, объявленные в функциях или классах, создаются и инициализируются лишь однажды и «живут», пока не завершится программа (§ 10.4.9). Такие объекты называются статическими. Время жизни элементов массивов и нестатических членов структур или классов определяется объектами, частью которых они являются .

Создавая объекты операторами ***new*** и ***delete*** (§ 6.2.6), вы непосредственно управляете временем их жизни.

4.9.7. typedef

Объявление, начинающееся с ключевого слова ***typedef***, вводит новое имя для типа, а не для переменной данного типа. Например:

```
typedef char* Pchar;
Pchar p1,p2;           // p1 и p2 типа char*
char* p3 = p1;
```

Целью такого объявления часто является назначение короткого синонима для часто используемого типа. Например, при частом применении ***unsigned char*** можно ввести синоним ***uchar***:

```
typedef unsigned char uchar;
```

Другое использование ***typedef*** — свести в одно место все непосредственные ссылки на какой-то тип. Например:

```
typedef int int32,
typedef short int16;
```

Таким образом, используя *int32* везде, где могут потребоваться большие числа, мы можем перенести наше приложение на машину с *sizeof(int) == 2*, просто заместив единственную строчку кода с *int*:

```
typedef long int32;
```

Хорошо это или плохо, но имена, вводимые *typedef*, являются синонимами, а не новыми типами. Следовательно, старые типы можно использовать совместно с их синонимами. Если вам нужны различные типы с одинаковой семантикой или с одинаковым представлением, обратитесь к перечислениям (§ 4.8) или классам (глава 10).

4.10. Советы

[1] Ограничивайте область видимости имен; § 4.9.4.
[2] Не используйте одно и то же имя и в текущей, и в объемлющей областях видимости; § 4.9.4.
[3] В каждом объявлении объявляйте только одно имя; § 4.9.2.
[4] Локальным и часто используемым переменным присваивайте короткие имена; глобальным и редко используемым — длинные; § 4.9.3.
[5] Избегайте внешне похожих имен; § 4.9.3.
[6] Старайтесь придерживаться единого стиля именования; § 4.9.3.
[7] Выбирайте имена, отражающие смысл, а не форму представления; § 4.9.3.
[8] Используйте *typedef* для введения осмысленного нового имени встроенного типа, если этот встроенный тип, представляющий некоторое значение, может подлежать замене; § 4.9.7.
[9] Применяйте *typedef* для задания синонимов типов; используйте перечисления и классы для определения новых типов; § 4.9.7.
[10] Помните, что в каждом объявлении должен быть указан тип (больше не существует правила «неявного *int*»); § 4.9.1.
[11] Избегайте избыточных предположений о численных значениях символов; § 4.3.1, § B.6.2.1.
[12] Избегайте избыточных предположений о размере целых; § 4.6.
[13] Избегайте избыточных предположений о диапазоне значений чисел с плавающей точкой; § 4.6.
[14] Предпочитайте *int* по сравнению с *short int* или *long int*; § 4.6.
[15] Предпочитайте *double* по сравнению с *float* или *long double*; § 4.5.
[16] Предпочитайте *char* по сравнению с *signed char* или *unsigned char*; § B.3.4.
[17] Избегайте избыточных предположений о размерах объектов; § 4.6.
[18] Избегайте беззнаковой арифметики; § 4.4.
[19] С подозрением относитесь к преобразованиям из *signed* в *unsigned* и из *unsigned* в *signed*; § B.6.2.6.
[20] С подозрением относитесь к преобразованиям чисел с плавающей точкой в целые; § B.6.2.6.
[21] С подозрением относитесь к преобразованиям в «укороченные» типы, например, *int* в *char*; § B.6.2.6.

4.11. Упражнения

1. (*2) Запустите программу «Здравствуй, мир!» (§ 3.2). Если не получится, обратитесь к § Б.3.1.
2. (*1) Проделайте следующее для каждого объявления в § 4.9: если объявление не является определением, допишите определение. Если объявление является определением, напишите соответствующее объявление, которое не является определением.
3. (*1.5) Напишите программу, которая печатает размеры фундаментальных типов, нескольких типов указателей и нескольких перечислений по вашему выбору. Воспользуйтесь оператором *sizeof*.
4. (*1.5) Напишите программу, которая печатает символы от *'a'* до *'z'* и цифры от *'0'* до *'9'* и их целые значения. Проделайте то же самое для других печатаемых символов. Проделайте то же самое, но воспользуйтесь шестнадцатеричным представлением чисел.
5. (*2) Каковы в вашей системе максимальные и минимальные значения переменных следующих типов: *char*, *short*, *int*, *long*, *float*, *double*, *long double* и *unsigned*?
6. (*1) Какой максимальной длины может быть локальное имя в программе на C++ в вашей системе? Какой максимальной длины может быть внешнее имя в программе на C++ в вашем системе? Существуют ли какие-либо ограничения на использование символов в именах?
7. (*2) Нарисуйте граф целых и фундаментальных типов, где от одного типа идет стрелка к другому, если все значения первого типа могут быть представлены значениями второго в любой реализации, соответствующей стандарту. Нарисуйте такой граф для типов вашей любимой реализации.

5

Указатели, массивы и структуры

Великое и смешное
часто так тесно переплетены друг с другом,
что их бывает трудно отличить.
— Том Пэйн

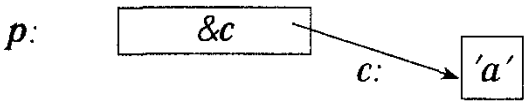
Указатели — ноль — массивы — строковые литералы — указатели на массивы — константы — указатели и константы — ссылки — ***void**** — структуры данных — советы — упражнения.

5.1. Указатели

Для данного типа ***T***, тип ***T** ** является «указателем на ***T***». То есть переменная типа ***T** ** содержит адрес объекта типа ***T***. Например:

```
char c = 'a';
char* p = &c;          // p содержит адрес c
```

или графически:



К сожалению, запись указателей на массивы и функции сложнее:

```
int* pi;               // указатель на int
char** ppc;            // указатель на указатель на char
int* ap[15];           // массив из 15 указателей на int
int (*fp)(char*);      // указатель на функцию с аргументом char*, возвращающую int
int* f(char*);         // функция с аргументом char*,
                       // возвращающая указатель на int
```

Объяснение синтаксиса объявлений см. в § 4.9.1, а полное изложение грамматики см. в приложении А.

Основной операцией над указателями является *разыменование*, то есть получение объекта, на который указывает указатель. Эта операция называется также *косвенным обращением*. Оператором разыменования является (префиксный) унарный *. Например:

```
char c = 'a',
char* p = &c,        // содержит адрес переменной c
char c2 = *p,        // c2=='a'
```

Переменной, на которую указывает ***p***, является ***c***, а значением, хранимым в ***c***, является *'a'*, поэтому значение ****p*** (присваиваемое ***c2***) равно *'a'*.

Над указателями на элементы массивы можно выполнять некоторые арифметические операции (§ 5.3). Указатели на функции могут быть исключительно полезны; они обсуждаются в § 7.7.

Указатели задуманы с целью непосредственного отражения механизмов адресации компьютеров, на которых исполняются программы. Большинство компьютеров может адресовать байт. Те, которые не могут, все равно так или иначе умеют извлекать байты из слов. С другой стороны, очень немногие машины могут непосредственно обращаться к битам. Следовательно, наименьшим объектом, который можно независимо разместить и на который можно независимо указать с помощью встроенных типов, является ***char***. Обратите внимание, что тип ***bool*** занимает по меньшей мере столько же памяти, что и ***char*** (§ 4.6). Для более компактного хранения малых (по объему) величин можно воспользоваться логическими операциями (§ 6.2.4) или битовыми полями в структурах (§ В.8.1).

5.1.1. Ноль

Ноль (***0***) имеет тип ***int***. Благодаря стандартным преобразованиям (§ В.6.2.3), ***0*** можно использовать в качестве константы любого интегрального типа (§ 4.1.1), типа с плавающей точкой, указателя или указателя на член класса. Тип нуля определяется по контексту. Ноль, как правило (но не всегда), будет физически представлен в виде последовательности нулевых битов соответствующей длины.

Гарантируется, что нет объектов с нулевым адресом. Следовательно, указатель, равный нулю, можно интерпретировать как указатель, который ни на что не ссылается.

В языке С было очень популярно определять макрос ***NULL*** для представления такого нулевого указателя. Так как в С++ типы проверяются более жестко, использование банального нуля вместо ***NULL*** приведет к меньшим проблемам. Если вы чувствуете, что просто обязаны определить ***NULL***, воспользуйтесь

```
const int NULL = 0;
```

Модификатор ***const*** (§ 5.4) предотвращает ненамеренное замещение ***NULL*** и гарантирует, что ***NULL*** можно использовать везде, где требуется константа.

5.2. Массивы

Для данного типа ***T***, ***T*** [***size***] есть тип «массив из ***size*** элементов типа ***T***». Элементы нумеруются (индексируются) от ***0*** до ***size–1***. Например:

```
float v[3],          // массив из трех элементов с плавающей точкой: v[0], v[1], v[2]
char* a[32],         // массив из 32 указателей на char: a[0], a[31]
```

Количество элементов массива (размер массива) должно быть константным выражением (§ В.5). Если вам потребуется менять размер массива, воспользуйтесь типом ***vector*** (§ 3.7.1, § 16.3). Например:

```
void f(int i)
{
        int v1[i];              // ошибка: размер массива не является
                                // константным выражением
        vector<int> v2(i),      // правильно
}
```

Многомерные массивы описываются как массивы массивов. Например:

```
int d2[10][20];     // d2 является массивом из 10 массивов по 20 целых
```

Запись индексов через запятую, используемая в других языках, приведет к ошибке на этапе компиляции, потому что запятая является разделителем в последовательности (§ 6.2.2) и не допускается в константных выражениях (§ B.5). Можете попробовать, например:

```
int bad[5, 2];      // ошибка: запятая запрещена в константных выражениях
```

Многомерные массивы описаны в § B.7. В коде высокого уровня их лучше вообще избегать.

5.2.1. Инициализаторы массивов

Начальное значение массиву можно присвоить, указав список значений. Например:

```
int v1[] = { 1, 2, 3, 4 },
char v2[] = { 'a', 'b', 'c', 0 },
```

Когда массив объявлен без указания размера, но при этом инициализирован списком, его размер вычисляется путем подсчета числа элементов этого списка. Следовательно, **v1** и **v2** являются массивами **int[4]** и **char[4]** соответственно. Если размер явно указан, задание большего числа элементов в списке инициализации будет ошибкой. Например:

```
char v3[2] = { 'a', 'b', 0 };       // ошибка: слишком много элементов
char v4[3] = { 'a', 'b', 0 };       // правильно
```

Если в списке элементов инициализации недостает элементов, всем остальным элементам массива присваивается значение **0**. Например:

```
int v5[8] = { 1, 2, 3, 4 };
```

равнозначно

```
int v5[] = { 1, 2, 3, 4, 0, 0, 0, 0 };
```

Обратите внимание, что не существует присваивания массиву, соответствующего описанному выше способу инициализации:

```
void f()
{
        v4 = { 'c', 'd', 0 };  // ошибка: такое присваивание массиву не допустимо
}
```

Если вам нужно подобное присваивание, воспользуйтесь типом **vector** (§ 16.3) или **valarray** (§ 22.4).

Массив символов удобно инициализировать строковым литералом (§ 5.2.2).

5.2.2. Строковые литералы

Строковым литералом называется последовательность символов, заключенная в двойные кавычки:

```
"Это строка"
```

В строковом литерале на один символ больше, чем используется при записи; он всегда заканчивается нулевым символом '\0', значение которого равно 0. Например:

```
sizeof("Бор") == 4
```

Тип строкового литерала есть «массив с надлежащим количеством константных символов»; таким образом "Бор" принадлежит типу const char[4].

Строковый литерал можно присвоить переменной типа char*. Это разрешается, потому что в предыдущих определениях C и C++ типом строкового литерала был char*. Благодаря такому разрешению миллионы строк кода на C и C++ остаются синтаксически корректными. Однако изменение строкового литерала через такой указатель является ошибкой:

```
void f()
{
    char* p = "Платон",
    p[4] = 'e';              // ошибка: присваивание константе,
                             // результат не определен
}
```

Такого рода ошибки, как правило, не могут быть выявлены до выполнения программы (см. также § В.2.3). Кроме того, различные реализации по-разному относятся к нарушению этого правила. То, что строковые литералы являются константами, не только является очевидным, но и позволяет при реализации произвести значительную оптимизацию методов хранения и доступа к строковым литералам.

Если нам нужна строка, которую мы гарантировано сможем модифицировать, следует скопировать символы в массив:

```
void f()
{
    char p[] = "Зенон";      // p — массив из шести символов
    p[0] = 'R',              // правильно
}
```

Память под строковые литералы выделяется статически, поэтому их свободно можно возвращать в качестве значения функции. Например:

```
const char* error_message (int i)
{
    // ...
    return "выход за пределы диапазона";
}
```

Память, содержащая строку "выход за пределы диапазона", не будет освобождена после вызова функции error_message ().

Будут ли одинаковые строковые литералы записываться в одно место памяти или нет, зависит от реализации (§ В.1). Например:

```
const char* p = "Гераклит";
const char* q = "Гераклит";

void g()
{
        if (p==q) cout << "в одном месте!\n";    // результат зависит от
                                                 // конкретной реализации
}
```

Обратите внимание, что *p==q* сравнивает адреса (значения указателей), а не значения, на которые они указывают.

Пустая строка записывается в виде пары соседних двойных кавычек "" (и имеет тип *const char*[*1*]).

Форма записи специальных символов с обратной косой чертой (\) (§ В.3.2) может использоваться и в строках. Благодаря этому в строках можно хранить символ двойной кавычки и обратной косой черты. Самым распространенным символом такого типа является символ перехода на новую строку ('\n'). Например:

```
cout << "звуковой сигнал в конце сообщения\a\n";
```

Специальный символ '\a' из набора ASCII означает BEL (звонок); при его выводе раздается звуковой сигнал.

Нельзя получить перевод строки, «по-настоящему» переведя строку:

```
"это не строка,
а синтаксическая ошибка"
```

Длинные строки можно разделять символами-разделителями, например, чтобы сделать код более читабельным. Строки

```
char alpha[] =   "abcdefghijklmnopqrstuvwxyz"
                 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

компилятор объединит в одну строку, поэтому *alpha* можно было инициализировать и следующей строкой:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Можно включать в строку нулевой символ, но большинство программ не догадаются, что после него есть еще что-то. Например, строка *"Йенс\000Мунк"* будет интерпретироваться как *"Йенс"* такими функциями стандартной библиотеки, как *strcpy* () или *strlen* () (см. § 20.4.1).

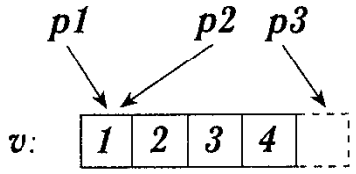
Строка с префиксом *L*, например *L"angst"*, является строкой символов из расширенного набора (§ 4.3, § В.3.3). Ее тип *const wchar_t*[].

5.3. Указатели на массивы

В C++ указатели и массивы тесно связаны. Имя массива можно использовать в качестве указателя на его первый элемент. Например:

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v;            // указатель на первый элемент
                        // (неявное преобразование)
int* p2 = &v[0];        // указатель на первый элемент
int* p3 = &v[4];        // указатель на элемент, следующий за последним
```

или графически



Гарантируется осмысленность значения указателя на элемент, следующий за последним элементом массива. Это важно для многих алгоритмов (§ 2.7.2, § 18.3). Впрочем, ввиду того, что такой указатель на самом деле не указывает ни на какой элемент массива, его нельзя использовать ни для чтения, ни для записи. Результат получения адреса элемента массива, предшествующего первому, не определен, и такой операции следует избегать. На некоторых машинах память под массивы выделяется с самого начала адресного пространства, поэтому «адреса элемента, предшествующего первому» возможно просто не существует.

Неявное преобразование имени массива в указатель на его первый элемент широко используется в вызовах функций С. Например:

```
extern "C" int strlen (const char*);        // находится в <string.h>

void f()
{
        char v[] = "Анна-Мария";
        char* p = v;                        // неявное преобразование char[] в char*
        strlen (p);
        strlen (v);                         // неявное преобразование char[] в char*
        v = p;                              // ошибка: нельзя присваивать массиву
}
```

При обоих вызовах стандартной библиотечной функции ***strlen*** () передается одно и то же значение. Загвоздка в том, что невозможно избежать неявного преобразования типов. Другими словами, невозможно объявить какую-нибудь функцию таким образом, чтобы при ее вызове массив ***v*** копировался. К счастью, не существует ни явного, ни неявного преобразования указателя в массив.

Неявное преобразование массива в указатель при вызове функции приводит к потере информации о размере массива. Но вызываемая функция должна каким-либо образом определить этот размер, чтобы выполнять осмысленные действия. Как и другие функции С из стандартной библиотеки, принимающие указатель на строку символов, ***strlen*** () считает, что ее аргумент, строка завершается нулем; ***strlen*** () возвращает количество символов в строке, вплоть до завершающего нуля, но не считая его. Все это — вопросы достаточно низкого уровня. Типы ***vector*** (§ 16.3) и ***string*** (глава 20) стандартной библиотеки не страдают подобными недостатками.

5.3.1. Доступ к элементам массива

Эффективный и элегантный доступ к массивам (и подобным структурам данных) является ключевым моментом для многих алгоритмов (см. § 3.8 и главу 18). Доступ к элементам массива может осуществляться либо при помощи указателя на массив и индекса, либо через указатель на элемент массива. Рассмотрим пример прохода по строке с использованием индекса:

```
void fi (char v[])
{
    for (int i=0; v[i]!=0; i++) use (v[i]);
}
```

Это эквивалентно следующему примеру, где используется указатель:

```
void fp (char v[])
{
    for (char* p=v; *p!=0; p++) use (*p);
}
```

Префиксный оператор * означает разыменование, поэтому ***p*** есть символ, на который указывает ***p***. Оператор ++ увеличивает значение указателя таким образом, что он указывает на следующий элемент массива.

Не существует внутренних причин, по которым один вариант был бы быстрее другого. Современный компилятор должен сгенерировать тождественный код для обоих примеров (см. § 5.9[8]). Программисты могут выбирать между этими вариантами, исходя из логических или эстетических соображений.

Результат применения арифметического оператора +, −, ++ или −− к указателю зависит от типа объекта, на который ссылается указатель. Если к указателю ***p*** типа ***T**** применяется арифметическая операция, предполагается, что ***p*** указывает на элемент массива типа ***T***; ***p+1*** указывает на следующий элемент массива, а ***p−1*** — на предыдущий. То есть целое значение ***p+1*** будет на ***sizeof (T)*** больше, чем целое значение ***p***. Например, результатом выполнения

```
#include <iostream>

int main ()
{
    int vi[10];
    short vs[10];

    std::cout << &vi[0] << ' ' << &vi[1] << '\n';
    std::cout << &vs[0] << ' ' << &vs[1] << '\n';
}
```

будет

```
0x7fffaef0      0x7fffaef4
0x7fffaedc      0x7fffaede
```

при использовании принятой по умолчанию шестнадцатеричной формы записи указателей. Пример демонстрирует, что в моей реализации ***sizeof (short)*** равен ***2***, а ***sizeof (int)*** равен ***4***.

Вычитание указателей друг из друга определено только в том случае, если оба указателя указывают на элементы одного и того же массива (хотя язык не позволяет быстро проверить, так ли это). Результатом вычитания одного указателя из другого будет количество элементов массива (целое число) между этими указателями. К указателю можно прибавлять целое и вычитать из него целое; в обоих случаях, результатом будет указатель. Если полученный таким образом указатель не указывает на элемент того же массива (или на элемент, следующий за последним), что и исходный, то результат его использования не определен. Например:

```
void f()
{
    int v1[10];
    int v2[10];

    int i1 = &v1[5] - &v1[3];      // i1=2
    int i2 = &v1[5] - &v2[3];      // результат не определен

    int* p1 = v2 + 2;              // p1 = &v2[2]
    int* p2 = v2 - 2;              // *p2 не определен
}
```

Обычно нет необходимости в использовании относительно сложной арифметики указателей, лучше ее вообще избегать. Сложение указателей смысла не имеет и поэтому запрещено.

Массивы не самодостаточны в том смысле, что не гарантируется хранение информации о количестве элементов вместе с самим массивом. Предполагается, что для просмотра элементов массива в случаях, когда в массиве нет ограничивающего символа (как для строк), необходимо каким-либо образом указать размер явно. Например:

```
void fp(char v[], unsigned int size)
{
    for (int i=0; i<size; i++) use(v[i]);

    const int N = 7;
    char v2[N];
    for (int i=0; i<N; i++) use(v2[i]);
}
```

Обратите внимание, что в большинстве реализаций C++ отсутствует проверка диапазона для массивов. Таков традиционный низкоуровневый подход к массивам. Более совершенное понятие массива можно реализовать при помощи классов (см. § 3.7.1).

5.4. Константы

В C++ введена концепция определяемых пользователем констант (***const***) для указания на то, что значение нельзя изменить непосредственно. Это может быть полезно в нескольких отношениях. Например, многие объекты не меняются после инициализации; использование символических констант приводит к более удобному в сопровождении коду, чем применение литералов непосредственно в тексте программы; указатели часто используются только для чтения, но не для записи; большинство аргументов функций читаются, но не перезаписываются.

Чтобы объявить объект константой, в объявление нужно добавить ключевое слово ***const***. Так как константе нельзя присваивать значения, она должна быть инициализирована. Например:

```
const int model = 90;          // model является константой
const int v[] = { 1, 2, 3, 4 }; // все v[i] являются константами
const int x;                   // ошибка: нет инициализатора
```

Объявление чего-либо в качестве ***const*** гарантирует, что в текущей области видимости его значение не изменится:

```
void f()
{
        model = 200,        // ошибка
        v[2]++,             // ошибка
}
```

Отметьте, что ***const*** модифицирует тип, то есть ***const*** ограничивает возможное использование объекта, но не указывает способ размещения константного объекта. Например.

```
void g(const X* p)
{
        // здесь нельзя изменить *p
}

void h()
{
        X val,          // val можно менять
        g(&val),
        // ...
}
```

В зависимости от степени изощренности, компилятор может несколькими способами воспользоваться константностью объекта. Например, инициализатор константы часто (но не всегда) является константным выражением (§ В.5); если это так, его можно вычислить в момент компиляции. Более того, если компилятору известны все случаи использования константы, он может не выделять под нее память. Например:

```
const int c1 = 1,
const int c2 = 2,
const int c3 = my_f(3),     // значение c3 не известно во время компиляции
extern const int c4,        // значение c4 не известно во время компиляции
const int* p = &c2,         // необходимо выделить память под c2
```

При наличии таких объявлений компилятор знает значения ***c1*** и ***c2***, поэтому их можно использовать в константных выражениях. Так как значения ***c3*** и ***c4*** неизвестны во время компиляции (при использовании только информации, доступной в данной единице компиляции; см. § 9.1), для них должна быть выделена память. Так как берется адрес ***c2*** (и предположительно где-то используется), должна быть выделена память под ***c2***. Простым и типичным использованием константы является тот случай, когда значение константы известно во время компиляции и под нее не требуется выделение памяти. Примером является ***c1***. Ключевое слово ***extern*** означает, что ***c4*** определена в другом месте (§ 9.2).

Как правило, для массива констант требуется выделение памяти, так как, в общем случае, компилятор не в состоянии определить, к какому элементу массива происходит доступ в выражениях. Впрочем, на многих машинах и здесь достигается повышение эффективности за счет помещения массивов констант в специальную область памяти только для чтения.

Типичным является использование констант в качестве размера массивов и меток в инструкции ***case***. Например:

```
const int a = 42,
const int b = 99,
```

```
const int max = 128;
int v[max];
void f(int i)
{
        switch (i) {
        case a
                // .
        case b
                // .
        }
}
```

В таких ситуациях альтернативой константам часто выступают перечисления (§ 4.8).

То, каким образом ***const*** можно использовать с функциями, являющимися членами классов, обсуждается в § 10.2.6 и § 10.2.7.

Для избежания «магических чисел» в коде следует систематически пользоваться символическими константами. Если некое число, такое как размер массива, регулярно повторяется в коде, модифицировать код будет сложно, потому что придется найти и изменить каждое вхождение данного числа. Использование символической константы локализует информацию. Обычно численные константы означают некоторые предположения о программе. Например, ***4*** может означать количество байт, требуемых для представления целого, ***128*** — объем буфера ввода, а ***6.24*** — курс датской кроны по отношению к доллару США. Тому, кто сопровождает программу, трудно будет понять смысл этих чисел, если они будут записаны в виде числовых констант. Часто на такие числа не обращают внимания при переносе в другую среду или при внесении изменений, что приводит к ошибкам. Представляя такие значения в виде ясно комментированных символических констант, вы сводите к минимуму упомянутые проблемы сопровождения.

5.4.1. Указатели и константы

В операциях с указателями участвуют два объекта: сам указатель и объект, на который он ссылается. Помещение ключевого слова ***const*** перед объявлением указателя делает константой объект, а не указатель. Для объявления самого указателя в качестве константы, используется оператор объявления ****const***, а не просто *. Например:

```
void f1 (char* p)
{
        char s[] = "Gorm";

        const char* pc = s,          // указатель на константу
        pc[3] = 'g',                 // ошибка: pc указывает на константу
        pc = p,                      // правильно

        char *const cp = s,          // константный указатель
        cp[3] = 'a',                 // правильно
        cp = p,                      // ошибка · cp является константой

        const char *const cpc = s,   // константный указатель на константу
        cpc[3] = 'a',                // ошибка: cpc указывает на константу
        cpc = p,                     // ошибка: cpc является константой
}
```

Оператор объявления, который делает указатель константой, — это ***const***. Нет оператора объявления ***const****, поэтому ключевое слово ***const*** слева от * интерпретируется как часть базового типа. Например:

```
char *const cp;          // константный указатель на char
char const* pc;          // указатель на константу типа char
const char* pc2;         // указатель на константу типа char
```

Некоторые люди читают такие объявления справа налево. Например, «***cp*** является константным указателем на ***char***», а «***pc2*** является указателем на константу типа ***char***».

Некоторый объект при обращении к нему через один указатель может быть константой, а при обращении через другой — переменной. Это свойство особенно полезно для аргументов функций. Объявив аргумент-указатель константой, функция не сможет изменить объект, на который он ссылается. Например:

```
char* strcpy (char* p, const char* q),     // нельзя изменить *q
```

Вы можете присвоить адрес переменной указателю на константу, потому что это безвредная операция. Нельзя, однако, присвоить адрес константы произвольному указателю, потому что в этом случае можно будет изменить значение объекта. Например:

```
void f4 ()
{
    int a = 1;
    const int c = 2;
    const int* p1 = &c;      // правильно
    const int* p2 = &a;      // правильно
    int* p3 = &c             // ошибка: инициализация int* значением типа const int*
    *p3 = 7;                 // попытка изменить значение константы c
}
```

При помощи явного преобразования типа можно устранить ограничение, состоящее в том, что указатель указывает на константу (§ 10.2.7.1 и § 15.4.2.1).

5.5. Ссылки

Ссылка (reference) является альтернативным именем объекта. Ссылки чаще всего используются для указания аргументов функций и возвращаемых значений вообще, и при перегрузке операторов в частности (глава 11). Запись ***X***& означает *ссылку на* ***X***. Например:

```
void f()
{
    int i = 1;
    int& r = i;          // r и i ссылаются на одно и то же целое
    int x = r;           // x = 1
    r = 2;               // i = 2
}
```

Чтобы быть уверенными, что ссылка является именем чего-либо (то есть, привязана к объекту), мы должны инициализировать ее. Например:

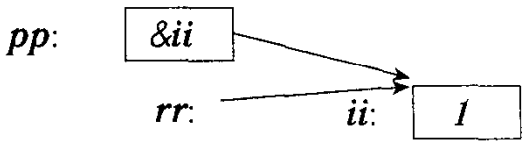
```
int i = 1;
int& r1 = i;          // правильно — r1 инициализирована
int& r2;              // ошибка: отсутствует инициализатор
extern int& r3;       // правильно — r3 инициализирована в другом месте
```

Инициализация ссылки кардинально отличается от присваивания ей значения. Несмотря на форму записи, ни один оператор не выполняет действий над ссылкой. Например:

```
void g()
{
    int ii = 0;
    int& rr = ii;
    rr++;             // ii увеличена на 1
    int* pp = &rr;    // pp указывает на ii
}
```

Выражение *rr*++ допустимо, но не увеличивает ссылку *rr*; ++ применяется к целому значению *ii*. Как следствие, значение ссылки нельзя изменить после инициализации. Она всегда ссылается на объект, которым инициализирована. Чтобы получить указатель на объект, именем которого является ссылка *rr*, мы можем написать *&rr*.

Очевидной реализацией ссылки является (константный) указатель, при каждом использовании которого происходит разыменование. Большого вреда в такой интерпретации ссылки нет, но при этом надо помнить, что ссылка, в отличие от указателя, не является объектом, над которым можно выполнять операции:



В некоторых случаях компилятор может оптимизировать ссылку таким образом, что во время исполнения вообще не будет существует объекта, представляющего ссылку.

Инициализация ссылки тривиальна, когда инициализитором является lvalue (объект, адрес которого можно получить; см. § 4.9.6). Инициализатором для «просто» *T*& должно быть lvalue типа *T*.

Инициализатор для *const T*& не обязан быть lvalue и даже иметь тип *T*. В таких случаях:

[1] если необходимо, осуществляется неявное преобразование к типу *T* (см. § B.6);
[2] результирующее значение помещается во временную переменную типа *T*;
[3] временная переменная используется как значение инициализатора.

Рассмотрим пример:

```
double& dr = 1;          // ошибка: требуется lvalue
const double& cdr = 1;   // правильно
```

Можно проинтерпретировать последнюю инициализацию следующим образом:

```
double temp = double(1),     // создать временную переменную
                             // с соответствующим значением
const double& cdr = temp;    // затем использовать временную
                             // переменную как инициализатор cdr
```

Временная переменная, созданная для хранения инициализатора, существует до конца области видимости инициализируемой ею ссылки.

Ссылки на переменные и ссылки на константы различаются: создание временной переменной в случае ссылки на переменную интенсивно провоцирует ошибки. Присваивание переменной сводится к присваиванию (скоро исчезающей) временной переменной. Подобных проблем не существует со ссылками на константы. Ссылки на константы часто находят применение в качестве аргументов функций (§ 11.6).

Ссылка может использоваться в качестве аргумента функции, которая изменяет значение передаваемого ей объекта. Например:

```
void increment (int& aa) { aa++, }

void f()
{
        int x = 1,
        increment (x),          // x = 2
}
```

Семантика передачи аргументов тождественна семантике инициализации, поэтому после вызова ***increment*** его аргумент ***aa*** становится другим именем для ***x***. Чтобы программа была читаемой, как правило стоит избегать функций, изменяющих свои аргументы. Предпочтительнее явно возвратить значение функции или запросить указатель в качестве аргумента:

```
int next (int p) { return p+1, }

void incr (int* p) { (*p)++, }

void g()
{
        int x = 1,
        increment (x),          // x = 2
        x = next (x),           // x = 3
        incr (&x),              // x = 4
}
```

По форме записи ***increment*** (***x***) нельзя догадаться, что значение ***x*** может быть изменено, в отличие от ***x=next*** (***x***) и ***incr*** (&***x***). Соответственно, «простой» ссылочный аргумент следует использовать только тогда, когда имя функции прозрачно намекает на изменение ею ссылочного аргумента.

Ссылки также применяются для написания функций, которые можно использовать и в левой, и в правой частях присваивания. Многие из самых любопытных случаев такого применения обнаруживаются в нетривиальных типах, определяемых пользователем. В качестве примера определим простой ассоциативный массив. Сначала определим структуру для пар (имя, значение) ***Pair*** следующим образом:

```
struct Pair {
        string name,
        double val,
},
```

Идея состоит в том, что у строковой переменной есть связанное с ней значение с плавающей точкой. Несложно определить функцию ***value*** (), которая поддерживает структуру данных, состоящую из одной пары (***Pair***) для каждой строки, переданной функции. Для краткости можно написать очень простую (и неэффективную) реализацию:

```
vector<Pair> pairs;

double& value (const string& s)
/*
        поддерживает набор пар Pair:
        ищет строку s; если найдена, возвращает соответствующее значение;
        иначе создает новую пару и возвращает значение по умолчанию 0
*/
{
        for (int i=0; i<pairs.size (); i++)
                if (s == pairs[i].name) return pairs[i].val;

        Pair p = { s, 0 };
        pairs.push_back (p);            // добавить пару в конец (§ 3.7.3)
        return pairs[pairs.size () - 1].val;
}
```

Эту функцию можно интерпретировать как массив чисел, проиндексированный строкой символов. Для заданного строкового аргумента ***value*** () находит соответствующий числовой объект (а *не* значение соответствующего числового объекта), а затем возвращает ссылку на него. Например:

```
int main ()         // сосчитать количество вхождений каждого слова во входном потоке
{
        string buf;
        while (cin>>buf) value (buf)++;
        for (vector<Pair> :const_iterator p = pairs.begin (); p!=pairs.end (); ++p)
                cout << p->name << ": " << p->val << '\n';
}
```

В цикле ***while*** при чтении каждого слова из стандартного потока ввода ***cin*** в строку ***buf*** (§ 3.6) модифицируется соответствующий счетчик. В конце печатается результирующая таблица слов с числом их вхождений. Например, если на входе было:

```
aa bb bb aa aa bb aa aa
```

то программа выведет:

```
aa: 5
bb: 3
```

Несложно преобразовать нашу реализацию в подходящий тип ассоциативного массива с помощью шаблона класса с перегруженным оператором [] (§ 11.8). Еще проще воспользоваться типом ***map*** из стандартной библиотеки (§ 17.4.1).

5.6. Указатель на void

Указатель на объект любого типа можно присвоить переменной типа ***void****, один ***void**** можно присвоить другому ***void****, пару ***void**** можно сравнивать на равенство и неравенство, и, наконец, ***void**** можно явно преобразовать в указатель на другой тип. Прочие операции могут оказаться опасными, потому что компилятор не знает, на какого сорта объект ссылается указатель на самом деле. Поэтому другие операции вызывают сообщение об ошибке на этапе компиляции. Чтобы воспользоваться ***void****, мы должны явно преобразовать его в указатель определенного типа. Например:

```
void f(int* pi)
{
    void* pv = pi;                  // правильно — неявное преобразование типа
                                    // из int* в void*
    *pv;                            // ошибка: нельзя разыменовать void*
    pv++;                           // ошибка: нельзя произвести инкремент void*
                                    // (размер «указываемого» объекта не известен)

    int* pi2 = static_cast<int*>(pv);       // явное преобразование в int*

    double* pd1 = pv;               // ошибка
    double* pd2 = pi;               // ошибка
    double* pd3 = static_cast<double*>(pv); // небезопасно
}
```

Как правило не безопасно использовать указатель, преобразованный (или приведенный с помощью функций ...*_cast*) к типу, отличному от типа объекта, на который он указывает. Например, компилятор полагает, что для ***double*** отводится 8 байт. Если это так, то можно получить странные результаты, когда ***pi*** ссылается на ***int***, память под который выделяется по другому. Подобная форма явного преобразования типа традиционно небезопасна и некрасива. Соответственно, использованный в примере тип преобразования ***static_cast*** был спроектирован при разработке языка таким образом, чтобы явно напоминать об этом.

Основными применениями ***void**** являются передача указателей функциям, которым не позволено делать предположения о типе объектов, а равно возврат объектов «неуточненного типа» из функций. Чтобы воспользоваться таким объектом, необходимо явно преобразовать тип указателя.

Функции, использующие указатели ***void****, обычно существуют на самых нижних уровнях системы, где происходит работа с аппаратными ресурсами. Например:

```
void* my_alloc(size_t n);     // выделяет n байт из моей специальной кучи
```

Наличие ***void**** на более высоких уровнях подозрительно и, скорее всего, является индикатором ошибки на этапе проектирования. Если ***void**** используется для оптимизации, его можно скрыть за безопасным интерфейсом (§ 13.5, § 24.4.2).

Указатели на функции (§ 7.7) и указатели на члены (§ 15.5) не могут быть присвоены переменным типа ***void****.

5.7. Структуры

Массивом называется набор элементов одинакового типа. Структура является набором элементов (почти) произвольных типов. Например:

```
struct address {
    char* name;             // "Jim Dandy"
    long int number;        // 61
    char* street;           // "South St"
    char* town;             // "New Providence"
    char state[2];          // 'N' 'J'
    long zip;               // 7974
};
```

Эта запись определяет новый тип ***address*** (адрес), состоящий из элементов, которые необходимо знать для отправки письма. Обратите внимание на точку с запятой в конце. Это одно из немногих мест в C++, где после закрывающей фигурной скобки нужно ставить точку с запятой и поэтому программисты имеют склонность забывать о ней.

Переменные типа ***address*** можно объявлять точно так же, как и другие переменные, а к конкретным *членам* обращаться с использованием оператора . (точка). Например:

```
void f()
{
        address jd;
        jd.name = "Jon Dandy";
        jd number = 61,
}
```

Формой записи, используемой при инициализации массивов, можно воспользоваться и для инициализации переменных структур. Например:

```
address jd = {
        "Jim Dandy",
        61, "South St",
        "New Providence", {'N', 'J'}, 7974
};
```

Впрочем, обычно лучше пользоваться конструктором (§ 10.2.3). Обратите внимание, что *jd.state* нельзя инициализировать строкой *"NJ"*. Строки ограничены символом *'\0'*. Следовательно, *"NJ"* имеет три символа — на один больше, чем можно разместить в *jd.state*.

К объектам типа структуры часто обращаются через указатели, используя оператор –> (разыменование указателя на структуру). Например:

```
void print_addr (address* p)
{
        cout << p->name << '\n'
                << p->number << ' ' << p->street << '\n'
                << p->town << '\n'
                << p->state[0] << p->state[1] << ' ' << p->zip << '\n';
}
```

Если *p* — указатель, то *p->m* эквивалентно *(*p).m*. Объекты типа структуры можно присваивать, передавать в качестве аргументов и возвращать в качестве значений функций. Например:

```
address current;

address set_current (address next)
{
        address prev = current;
        current = next;
        return prev;
}
```

Другие операторы, например сравнения (== и !=), не определены. Пользователь может определить их сам (глава 11).

Размер объекта типа структуры не обязательно равен сумме размеров его членов. Такое может произойти потому, что на многих машинах объекты определенных типов должны выравниваться по аппаратно определяемым границам или, по крайней мере, при выполнении этих условий они обрабатываются значительно быстрее. Например, целые числа должны начинаться на границе аппаратного слова. Говорят, что на таких машинах к объектам применено «выравнивание». Это приводит к «дырам» в структурах. Так на многих машинах ***sizeof*** (***address***) равен ***24***, а не ***22***, как можно было ожидать. Вы можете минимизировать неиспользуемое пространство отсортировав члены по размеру (начиная с наибольших). Однако, часто лучше упорядочивать члены из соображений читабельности, по размеру же сортировать только в случае жестких требований к оптимизации.

Имя типа можно использовать немедленно после его появления, а не обязательно после завершения всего объявления. Например:

```
struct Link {                   // связь
        Link* previous;         // предыдущий
        Link* successor;        // следующий
};
```

До полного завершения объявления структуры запрещается использовать ее имя для объявления других объектов. Например:

```
struct No_good {
        No_good member;         // ошибка: рекурсивное определение
};
```

Подобная запись является ошибкой, потому что компилятор не может определить размер ***No_good***. Для того чтобы два (или более) объекта типа структуры ссылались друг на друга, можно сначала объявить только имя типа. Например:

```
struct List,                    // список будет определен позднее

struct Link {
        Link* pre,
        Link* suc;
        List* member_of,
},

struct List {
        Link* head;
},
```

Использование ***List*** в объявлении ***Link*** при отсутствии первого объявления ***List*** приведет к синтаксической ошибке.

Именем типа структуры можно пользоваться до определения типа, пока не требуется знания имен членов или размера структуры. Например:

```
class S;                // S является именем некоторого типа

extern S a,
S f(),
void g(S),
S* h(S*),
```

Но многими подобными объявлениями нельзя пользоваться до тех пор, пока тип **S** не определен:

```
void k (S* p)
{
        S a,            // ошибка: S не определен, а для выделения памяти
                        // требуется знать его размер
        f();            // ошибка: S не определен, а для возврата значения
                        // требуется знать его размер

        g(a);           // ошибка: S не определен, а для передачи аргумента
                        // требуется знать его размер
        p->m = 7;       // ошибка: S не определен и имя члена неизвестно

        S* q = h(p),    // правильно — указатель можно передать и разместить
        q->m = 7;       // ошибка: S не определен и имя члена неизвестно
}
```

Структура является упрощенной формой класса (глава 10).

В силу причин, уходящих корнями глубоко в предысторию C, разрешается объявлять структуру и не структуру с одинаковыми именами в одной и той же области видимости. Например:

```
struct stat { /* ... */ };
int stat (char* name, struct stat* buf);
```

В этом случае простое имя (*stat*) есть имя не структуры, а доступ к структуре должен осуществляться с использованием ключевого слова ***struct***. Сходным образом можно использовать в качестве префиксов для разрешения неоднозначности ключевые слова ***class***, ***union*** (§ B.8.2) и ***enum*** (§ 4.8). Однако лучше избегать подобной перегрузки имен.

5.7.1. Эквивалентность типов

Две структуры являются разными типами, даже если у них одинаковые члены. Например,

```
struct S1 { int a; };
struct S2 { int a; };
```

являются двумя различными типами, поэтому

```
S1 x;
S2 y = x;    // ошибка: несоответствие типа
```

Структуры также отличаются от фундаментальных типов, так что

```
S1 x;
int i = x,   // ошибка: несоответствие типа
```

Каждая структура должна иметь единственное определение в программе (§ 9.2.3).

5.8 Советы

[1] Избегайте нетривиальной арифметики указателей; § 5.3.

[2] Принимайте специальные меры, чтобы не обратиться за границы массива; § 5.3.1.

[3] Пользуйтесь нулем (*0*) вместо *NULL*; § 5.1.11.
[4] Пользуйтесь *vector* и *valarray* вместо встроенных массивов; § 5.3.1.
[5] Пользуйтесь *string* вместо массивов символов, ограниченных нулем; § 5.3.
[6] Старайтесь как можно реже пользоваться простыми ссылками в качестве аргументов функций; § 5.5.
[7] Пользуйтесь *void** только в коде низкого уровня; § 5.6.
[8] Избегайте нетривиальных литералов («магических чисел») в коде. Вместо этого определите и используйте символические константы; § 4.8, § 5.4.

5.9. Упражнения

1. (*1) Напишите следующие объявления: указатель на символ, массив из 10 целых, ссылка на массив из десяти целых, указатель на массив символьных строк, указатель на указатель на символ, целая константа, указатель на целую константу, константный указатель на целое. Проинициализируйте все объекты.
2. (*1.5) Какие ограничения на типы указателей *char**, *int** и *void** существуют в вашей системе? Например, может ли *int** иметь нечетное значение (подсказка: подумайте о выравнивании)?
3. (*1) При помощи *typedef* определите типы *unsigned char*, *const unsigned char*, указатель на целое, указатель на указатель на *char*, указатель на массив *char*, массив из 7 указателей на целые числа, указатель на массив из 7 указателей на целые числа и массив из 8 массивов по 7 указателей на целые.
4. (*1) Напишите функцию, которая обменивает значения двух целых чисел. В качестве типа аргумента воспользуйтесь *int**. Напишите аналогичную функцию с аргументом типа *int*&.
5. (*1.5) Каков размер массива *str* в следующем примере:

   char str[] = *"короткая строка"*,

   Какова длина строки *"короткая строка"*?
6. (*1) Определите функции *f*(*char*), *g*(*char*&) и *h*(*const char*&). Вызовите их с аргументами *'a'*, *49*, *3300*, *c*, *uc* и *sc*, где *c* — *char*, *uc* — *unsigned char* и *sc* — *signed char*. Какие вызовы допустимы? При каких вызовах компилятор создаст временные переменные?
7. (*1.5) Определите таблицу, содержащую название месяцев и количество дней в каждом из них. Выведите содержимое таблицы. Проделайте это дважды: первый раз воспользуйтесь массивом символов для имен и массивом целых для дней, а второй раз воспользуйтесь массивом структур, каждая из которых хранит название месяца и количество дней в нем.
8. (*2) Выполните какие-нибудь тесты, чтобы убедиться, что ваш компилятор создает эквивалентные коды для итераций по индексу и с указателем (§ 5.3.1). Если у компилятора есть несколько уровней оптимизации, посмотрите, как это влияет на качество сгенерированного кода.
9. (*1.5) Найдите пример, когда имеет смысл воспользоваться именем в его собственном инициализаторе.
10. (*1) Определите массив строк, в котором строки содержат названия месяцев. Распечатайте эти строки. Передайте массив в функцию, которая печатает строки.

11. (*2) Прочитайте последовательность слов из потока ввода. Пусть слово ***Quit*** будет означать конец ввода. Распечатайте слова в порядке их ввода. Исключите из печати одинаковые слова. Отсортируйте слова перед печатью.
12. (*2) Напишите функцию, подсчитывающую количество вхождений пар букв в строку ***string***. Напишите функцию, которая делает то же самое в массиве символов, ограниченном нулем (в С-строке). Например, пара "ab" дважды встречается в "xabaacbaxabb".
13. (*1.5) Определите структуру ***Date*** для хранения дат. Напишите функции, которые читают даты из потока ввода, выводят даты и инициализируют переменные типа ***Date***.

6

Выражения и инструкции

Преждевременная оптимизация —
корень всех зол.
— Д. Кнут

С другой стороны, мы не можем игнорировать
эффективность.
— Джонатан Бентли

Пример с калькулятором — ввод — параметры командной строки — обзор операторов — логические и реляционные операторы — инкремент и декремент — свободная память — явное преобразование типов — сводка инструкций — объявления — инструкции выбора — объявления в условиях выбора — итерации — пресловутый ***goto*** — комментарии и отступы — советы — упражнения.

6.1. Калькулятор

Мы рассмотрим инструкции и выражения на примере программы-калькулятора, которая поддерживает четыре стандартных арифметических действия в качестве инфиксных операторов над числами с плавающей точкой. Кроме того, пользователь может определять переменные. Например, если имеется ввод:

```
r = 2.5
area = pi * r * r
```

где ***pi*** — предопределенная переменная, программа калькулятора выведет:

```
2.5
19.635
```

где ***2.5*** является результатом присваивания в первой строке ввода, а ***19.635*** — результатом второй строки.

Калькулятор состоит из четырех основных частей: синтаксического анализатора (parser) или обработчика, функции ввода, таблицы символов и драйвера (управляющей программы). В действительности, эта программа является миниатюрным компилятором, в котором обработчик осуществляет синтаксический анализ, функция ввода управляет вводом и осуществляет лексический анализ, в таблице символов содержится постоянная информация, а драйвер управляет инициализацией, выводом и обработ-

кой ошибок. Мы могли бы добавить много других средств к этому калькулятору, чтобы он стал более полезным (§ 6.6[20]), но, во-первых, код и так будет достаточно велик, а во-вторых, большинство средств, полезных для практического калькулятора, просто увеличат размер кода, не добавляя при этом ничего к иллюстрации возможностей C++.

6.1.1. Синтаксический анализатор

Грамматика языка, воспринимаемого калькулятором, описывается следующим образом:

```
program
        END                          // END — это конец ввода
        expr_list END

expr_list                            // список_выражений
        expression PRINT             // PRINT — это точка с запятой
        expression PRINT expr_list

expression                           // выражение
        expression + term
        expression - term
        term

term                                 // терм
        term / primary
        term * primary
        primary

primary                              // первичное выражение
        NUMBER                       // число
        NAME                         // имя
        NAME = expression
        - primary
        (expression)
```

Иными словами, программа есть последовательность выражений, разделенных точкой с запятой. Базовые элементы выражений — это числа, имена, операторы *, /, +, – (унарный и бинарный) и =. Имена не обязательно объявлять до их использования.

Применяемый нами стиль синтаксического анализа обычно называют *рекурсивным спуском* — это довольно популярная и «прозрачная» техника анализа «сверху вниз». В таких языках, как C++, где вызов функций обходится довольно дешево, эта техника еще и эффективна. Для каждого грамматического предложения имеется функция, вызывающая другие функции. Терминальные символы (например, ***END***, ***NUMBER***, + и –) распознаются лексическим анализатором, ***get_token*** (); нетерминальные символы распознаются функциями синтаксического анализатора ***expr*** (), ***term*** () и ***prim*** (). Как только оба операнда (под)выражения становятся известны, вычисляется значение выражения. В настоящем компиляторе в этот момент может генерироваться код.

Для ввода синтаксический анализатор пользуется функцией ***get_token*** () (получить лексему). Результат последнего вызова функции ***get_token*** () хранится в глобальной переменной ***curr_tok*** (текущая лексема). Переменная ***curr_tok*** имеет тип перечисления ***Token_value*** (значение лексемы):

```
enum Token_value {
        NAME,     NUMBER,        END,
        PLUS = '+', MINUS = '-',    MUL = '*',  DIV = '/',
        PRINT = ';', ASSIGN = '=',  LP = '(',   RP = ')'
};

Token_value curr_tok = PRINT;
```

Представление каждой лексемы в виде целого значения символа, который ее определяет, удобно и эффективно и может оказаться полезным при использовании отладчика. Это работает при условии, что символы, используемые при вводе, не имеют значений из перечисления ***enum Token_value***; ни в одном известном мне используемом наборе символов нет печатных символов, целые значения которых являлись бы однозначными числами. Начальным значением ***curr_tok*** я выбрал ***PRINT***, потому что это именно то значение, которое имеет ***curr_tok*** после вычисления значения выражения и вывода результата. Таким образом я «запускаю систему» в нормальном состоянии, минимизируя вероятность ошибки и потребность в специальном инициализирующем коде.

Каждая функция синтаксического анализатора принимает логический (§ 4.2) аргумент, который указывает, должна ли эта функция вызывать ***get_token*** () для выделения следующей лексемы из входного потока. Каждая функция синтаксического анализатора вычисляет значение «своего» выражения и возвращает его. Функция ***expr*** () обрабатывает сложение и вычитание. Она состоит из простого цикла, осуществляющего поиск термов для сложения и вычитания:

```
double expr(bool get)              // сложение и вычитание
{
    double left = term(get);

    for(;;)                        // «вечно»
        switch(curr_tok) {
        case PLUS
            left += term(true);
            break;
        case MINUS
            left -= term(true);
            break;
        default
            return left;
        }
}
```

Эта функция, в действительности, сама по себе делает немного. Как это и принято в высокоуровневых функциях в больших рограммах, для выполнения реальных действий она вызывает другие функции

switch-инструкция сравнивает указанное значение (в круглых скобках) с набором констант. ***break-инструкция*** используется для выхода из инструкции ***switch***. Константы, следующие за метками ***case***, должны отличаться друг от друга. Если проверяемое значение не равно ни одной из констант, выполняется инструкция, указанная после ***default***. В общем случае программист не обязан задавать ***default***.

Обратите внимание, что выражения типа *2−3+4* в соответствии с определением грамматики вычисляются как *(2−3)+4*.

Странная запись ***for*(;;)** является стандартным способом задания бесконечного цикла Вы можете читать это как «бесконечное количество раз». Такая запись является вырожденной формой ***for-инструкции*** (§ 6 3.3); другой альтернативой является выражение ***while (true)***. Инструкция ***switch*** выполняется до тех пор, пока не найдено что-нибудь отличное от + и −, после чего выполняется ***return-инструкция,*** следующая за ***default***

Операторы += и −= использованы для выполнения сложения и вычитания. Можно было записать ***left=left+term (true)*** и ***left=left−term (true)***, что не привело бы к изменению работы программы. Однако запись ***left+=term (true)*** и ***left−=term (true)*** не только короче, но и выражает необходимые действия в более непосредственной форме. Каждый оператор присваивания является отдельной лексемой, поэтому *a + = 1* является синтаксической ошибкой из-за пробела между + и =.

Операторы присваивания определены для бинарных операторов

```
+   −   *   /   %   &   |   ^   <<   >>
```

поэтому допустимы следующие операторы присваивания

```
=   +=   −=   *=   /=   %=   &=   |=   ^=   <<=   >>=
```

% — оператор деления по модулю (то есть остаток). &, | и ^ — битовые логические операторы ***И***, ***ИЛИ*** и исключающее ***ИЛИ***. << и >> — операторы сдвига влево и вправо В § 6.2 представлена сводная информация об операторах. Выражение *x@=y* означает *x=x@y*, если операнды имеют встроенные типы, за той лишь разницей, что *x* вычисляется только один раз.

В главах 8 и 9 обсуждается организация программы в виде набора модулей. За одним исключением все объявления в нашем примере можно упорядочить таким образом, что все будет объявлено один раз и до использования. Исключение составляет ***expr*** (), которая вызывает ***term*** (), которая вызывает ***prim*** (), которая в свою очередь вызывает ***expr*** (). Этот замкнутый круг надо каким-нибудь образом разорвать. Объявление

```
double expr(bool);
```

до объявления ***prim*** () прекрасно решает эту проблему.

Функция ***term*** () обрабатывает умножение и деление аналогично тому, как ***expr*** () делает это со сложением и вычитанием:

```
double term(bool get)          // умножение и деление
{
    double left = prim(get);
    for(;;)                    // «вечно»
        switch(curr_tok) {
        case MUL:
            left *= prim(true);
            break;
        case DIV:
            if(double d = prim(true)) {
                left /= d;
                break;
            }
```

```
                return error("деление на 0");
            default:
                return left;
            }
    }
```

Результат деления на ноль не определен и обычно приводит к катастрофическим последствиям. Поэтому мы проверяем делитель на ноль до операции деления, и если он равен нулю, вызываем функцию ***error*** (). Она описывается в § 6.1.4.

Переменная ***d*** вводится в программу в том месте, где она понадобилась, и сразу же инициализируется. Областью видимости имени, объявленного в условной инструкции, является тело условной инструкции, а результат присваивания одновременно является и проверяемым условием (§ 6.3.2.1). Следовательно, деление и присваивание ***left/=d*** выполняются тогда и только тогда, когда ***d*** не равно нулю.

Функция ***prim*** (), обрабатывающая первичные выражения, похожа на ***expr*** () и ***term*** (), за тем исключением, что по мере спуска по иерархии вызовов в ней уже выполняются некоторые полезные действия и не требуется цикл:

```
double number_value;            // численное значение
string string_value;            // строковое значение

double prim (bool get)          // первичные выражения
{
    if (get) get_token ();
    switch (curr_tok) {
    case NUMBER:                // константа с плавающей точкой
    {   double v = number_value;
        get_token ();
        return v;
    }
    case NAME:
    {   double& v = table[string_value];
        if (get_token () == ASSIGN) v = expr (true);
        return v;
    }
    case MINUS:                 // унарный минус
        return -prim (true);
    case LP:
    {   double e = expr (true);
        if (curr_tok != RP) return error ("ожидалась )");
        get_token ();           // пропустить скобки ')'
        return e;
    }
    default:
        return error ("ожидалось первичное выражение");
    }
}
```

Когда встречается ***NUMBER*** (то есть целый литерал или литерал с плавающей точкой), возвращается его значение. Процедура ввода ***get_token*** () помещает значение в глобаль-

ную переменную *number_value*. Использование глобальной переменной в программе часто является признаком не очень четкой структуры или проведения некоторой оптимизации. Так произошло и в нашем случае. В идеале лексема состоит из двух частей: величины, определяющей вид лексемы (в нашей программе это *Token_value*), и (при необходимости) значение лексемы. В нашем случае имеется только одна простая переменная *curr_tok*, поэтому потребовалась глобальная переменная *number_value* для хранения значения последнего прочитанного числа *NUMBER*. Исключение этой глобальной переменной оставляется в качестве упражнения (§ 6.6[21]). В действительности, нет необходимости сохранять *number_value* в локальной переменной *v* перед вызовом *get_token* (). При каждом корректном вводе калькулятор всегда использует одно число в вычислениях перед тем, как ввести другое. Но в случае ошибки сохранение значения и его вывод поможет пользователю.

Подобно тому, как значение последнего *NUMBER* хранится в *number_value*, строковое представление последнего *NAME* хранится в *string_value*. Прежде чем как-нибудь обработать имя, калькулятор должен заглянуть вперед и посмотреть, присваивается ли имени что-нибудь или оно просто используется. В обоих случаях происходит обращение к таблице символов. Таблица символов имеет тип *map* (§ 3.7.4, § 17.4.1):

```
map<string, double> table,
```

Это значит, что таблица проиндексирована по *string* (строке), и возвращаемым значением является *double*, соответствующее строке. Например, если пользователь ввел:

```
radius = 6378.388,
```

калькулятор выполнит следующее:

```
double& v = table["radius"],
// ... expr( ) вычисляет значение, которое будет присвоено ...
v = 6378.388,
```

Ссылка *v* используется для хранения значения *double*, связанного с *radius*, пока *expr* () вычисляет значение *6378.388* из введенной последовательности символов.

6.1.2. Функция ввода

Считывание из потока ввода — часто самая запутанная часть программы. Это происходит потому, что программа должна общаться с человеком — учитывать его капризы, привычки и совершенно необъяснимые на первый взгляд ошибки. Попытка заставить человека поступать так, как это удобно компьютеру, обычно считается (и справедливо) оскорбительной. Задача процедуры ввода низкого уровня состоит в считывании символов и составлении из них лексем более высокого уровня. Затем лексемы становятся элементами ввода для процедур более высокого уровня. В нашем примере низкоуровневый ввод осуществляется функцией *get_token* (). К счастью, написание процедур ввода низкого уровня не является нашей ежедневной задачей. Многие системы имеют стандартные функции такого рода.

Я построю *get_token* () в два этапа. Сначала я напишу обманчиво простую версию, которая всю сложность ввода переложит на пользователя. Затем я модифицирую ее в менее элегантную, но более простую в использовании процедуру.

Задача состоит в считывании символа, определении по этому символу вида лексемы и возвращении соответствующего ***Token_value***.

Начальные операторы считывают первый символ, не являющийся символом-разделителем, в ***ch*** и проверяют успешность ввода:

```
Token_value get_token ()
{
    char ch = 0;
    cin>>ch;
    switch (ch) {
    case 0:
        return curr_tok = END;        // присваивание и возврат
```

По умолчанию оператор >> пропускает символы-разделители (пробел, табуляция, перевод строки и т. д.) и оставляет значение ***ch*** прежним, если операция ввода завершилась неуспешно. Следовательно, ***ch==0*** означает конец ввода.

Присваивание является оператором; результатом присваивания является значение переменной слева от него. Это позволяет мне присвоить значение ***END*** переменной ***curr_tok*** и возвратить это значение в одной единственной инструкции. Использование одной инструкции вместо двух полезно при сопровождении. Если присваивание находится в одной строке, а возврат значения в другой, программист может модифицировать одну строку и забыть про вторую.

Давайте рассмотрим несколько случаев отдельно до того, как строить законченную функцию. Если встречается символ завершения ввода ';', скобки или оператор, то просто возвращается его значение:

```
    case ';':
    case '*':
    case '/':
    case '+':
    case '-':
    case '(':
    case ')':
    case '=':
        return curr_tok = Token_value (ch);
```

Числа обрабатываются следующим образом:

```
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    case '.':
        cin.putback (ch);
        cin >> number_value;
        return curr_tok = NUMBER;
```

Расположение меток ***case*** горизонтально, а не вертикально — это обычно не слишком хорошо, потому что ухудшается восприятие. Однако утомительно писать однообразные строки одну за другой. Так как оператор >> умеет читать константы с плавающей точкой в переменную типа ***double***, код становится тривиальным. Сначала первый символ (цифра или точка) помещается обратно в ***cin***. Затем константа читается в ***number_value***.

Имя обрабатывается аналогичным образом:

```
        default:            // NAME, NAME =, или ошибка
            if (isalpha(ch)) {
                cin.putback(ch);
                cin >> string_value;
                return curr_tok=NAME;
            }
            error("неправильная лексема");
            return curr_tok=PRINT;
```

Функция стандартной библиотеки ***isalpha*** () (§ 20.40.2) используется, чтобы избежать длинного списка ***case*** для каждого символа. Оператор >> считывает строку (в ***string_value***) до тех пор, пока во входном потоке не окажется символ-разделитель. Следовательно, пользователь должен после имени ввести символ-разделитель перед оператором, использующим это имя в качестве операнда. Такой подход далек от идеала, поэтому мы вернемся к этой проблеме в § 6.1.3.

И, наконец, полная функция ввода:

```
Token_value get_token()
{
    char ch = 0;
    cin>>ch;

    switch (ch) {
    case 0:
        return curr_tok = END;

    case ';':
    case '*':
    case '/':
    case '+':
    case '-':
    case '(':
    case ')':
    case '=':
        return curr_tok = Token_value(ch);
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    case '.':
        cin.putback(ch);
        cin >> number_value;
        return curr_tok = NUMBER;
    default:                    // NAME, NAME = или ошибка
        if (isalpha(ch)) {
            cin.putback(ch);
            cin >> string_value;
            return curr_tok = NAME;
        }
        error("неправильная лексема");
        return curr_tok =PRINT;
    }
}
```

Преобразование оператора в значение его лексемы тривиально, потому что ***Token_value*** оператора определено в качестве целого значения оператора (§ 4.8).

6.1.3. Низкоуровневый ввод

Применение калькулятора в таком виде вскрывает несколько неудобных моментов. Приходится помнить, что надо ввести точку с запятой в конце выражения, чтобы калькулятор вывел значение, а обязательное следование символа-разделителя за именем может представлять серьезную проблему. Например, *x=7* является единым идентификатором, а не идентификатором *x* за которым следуют оператор = и число *7*. Обе проблемы решаются заменой операций ввода в ***get_token*** (), ориентированных на типы, на код, который осуществляет посимвольный ввод.

Во-первых, пусть перевод строки, так же как и точка с запятой, означает конец выражения:

```
Token_value get_token ()
{
    char ch;

    do {                    // пропустить все символы-разделители кроме '\n'
        if (!cin.get (ch)) return curr_tok = END;
    } while (ch != '\n' && isspace (ch));

    switch (ch) {
    case ';':
    case '\n':
        return curr_tok = PRINT;
```

Мы воспользовались ***do-инструкцией***. Она эквивалентна ***while-инструкции*** за тем исключением, что тело цикла выполняется по крайней мере один раз. Вызов ***cin.get (ch)*** считывает один символ из стандартного потока ввода в ***ch***. По умолчанию ***get*** () не пропускает символы-разделители, как это делает оператор >>. Результатом проверки ***if*** (!***cin.get*** ()) будет ***false***, если не считан ни один символ из ***cin***. В этом случае возвращается ***END*** и работа калькулятора завершается. Используется оператор ! (логическое отрицание), потому что ***get*** () в случае успешного завершения возвращает ***true***.

Функция стандартной библиотеки ***isspace*** () проверяет, является ли символ разделителем (§ 20.4.2); ***isspace*** (*c*) возвращает ненулевое значение, если *c* является символом-разделителем и ноль — в противном случае. Проверка реализована в виде поиска по таблице, поэтому ***isspace*** () работает значительно быстрее, чем сравнивание с конкретными символами-разделителями (пробел, табуляция, перевод строки и т. д.). Аналогичные функции проверяют, является ли символ цифрой — ***isdigit*** (), буквой — ***isalpha*** (), буквой или цифрой — ***isalnum*** ().

После того как пропущены символы-разделители, следующий символ используется для определения типа лексемы.

Проблема, вызванная тем, что >> считывает в строку до тех пор, пока не встретится символ-разделитель, решается посимвольным чтением до тех пор, пока не встретится символ, отличный от буквы или цифры:

```
    default:            // NAME, NAME= или ошибка
        if (isalpha (ch)) {
            string_value = ch;
            while (cin.get (ch) && isalnum (ch)) string_value.push_back (ch);
            cin.putback (ch);
```

```
            return curr_tok = NAME;
        }
        error ("неправильная лексема");
        return curr_tok = PRINT;
```

К счастью, оба этих улучшения достигаются модификацией одного локального фрагмента кода. Конструирование программ таким образом, что улучшения можно реализовать модификацией только локальных фрагментов, является важной целью проектирования.

6.1.4. Обработка ошибок

Ввиду того, что программа достаточно проста, обработка ошибок не является сложной задачей. Функция обработки ошибок просто подсчитывает количество ошибок и выводит сообщение об ошибке:

```
int no_of_errors;

int error (const string& s)
{
        no_of_errors++;
        cerr << "ошибка: " << s << '\n';
        return 1;
}
```

Поток *cerr* — это небуферизованный поток вывода, который обычно используется для выдачи сообщений об ошибках (§ 21.2.1).

Возвращать значение из функции имеет смысл потому, что обычно ошибки возникают в процессе вычисления выражения, и надо либо полностью прервать процесс вычисления, либо возвратить значение, которое вряд ли вызовет последующие ошибки. Для нашего простого калькулятора годится второй подход. Если бы *get_token* () сохраняла информацию о номерах строк, *error* () могла бы сообщить пользователю, где примерно произошла ошибка. Это особенно полезно при использовании калькулятора в неинтерактивном режиме (§ 6.6[19]).

Часто необходимо завершить выполнение программы после возникновения ошибки, потому что нет разумного способа ее продолжить. Это можно сделать, вызвав функцию *exit* (), которая сначала освобождает ресурсы типа потоков вывода, а затем завершает программу, возвращая значение, переданное ей в качестве аргумента (§ 9.4.1.1).

Можно применить более регулярный метод обработки ошибок, обратившись к механизму обработки исключений (см. § 8.3, глава 14), но того, что мы имеем в нашем примере, вполне достаточно для программы калькулятора из 150 строк.

6.1.5. Драйвер

Теперь, когда все части программы в наличии, нам нужна только управляющая программа. Вот пример простой функции *main* ():

```
int main ()
{
        table["pi"] = 3.1415926535897932385;   // инициализация предопределенных имен
        table["e"] = 2.7182818284590452354;
```

```
        while (cin) {
            get_token();
            if (curr_tok==END) break;
            if (curr_tok==PRINT) continue;
            cout << expr(false) << '\n';
        }

        return no_of_errors;
    }
```

Как правило, ***main*** () возвращает ноль, если программа завершилась успешно и значение, отличное от нуля — в противном случае (§ 3.2). Возврат количества ошибок прекрасно реализует эту идею. В качестве инициализации потребовалось только записать предопределенные имена в таблицу символов.

Главной задачей основного цикла является считывание выражений и вывод ответа. Это выполняется строкой:

```
cout << expr(false) << '\n';
```

Аргумент ***false*** сообщает ***expr*** (), что она не должна вызывать ***get_token*** () для выделения лексемы.

Проверка ***cin*** при каждом проходе тела цикла гарантирует, что программа завершит выполнение, если что-нибудь произойдет с потоком ввода, а сравнение с ***END*** гарантирует, что цикл корректно завершится, когда ***get_token*** () обнаружит конец файла. Инструкция ***break*** реализует выход из текущей инструкции ***switch*** или цикла (***for***, ***while***, ***do***). Сравнение с ***PRINT*** (то есть, с '\n' или ';') освобождает ***expr*** () от ответственности за обработку пустых выражений. Инструкция ***continue*** эквивалентна переходу в самый конец цикла, поэтому в данном случае

```
while (cin) {
    // ...
    if (curr_tok==PRINT) continue;
    cout << expr(false) << '\n';
}
```

эквивалентно

```
while (cin) {
    // ...
    if (curr_tok != PRINT)
        cout << expr(false) << '\n';
}
```

6.1.6. Заголовочные файлы

Калькулятор использует средства стандартной библиотеки. Поэтому в завершенную программу должны быть включены заголовочные файлы:

```
#include<iostream>    // ввод/вывод
#include<string>      // строки
#include<map>         // ассоциативные массивы
#include<cctype>      // isalpha() и т. д.
```

Все эти заголовочные файлы предоставляют средства из области имен ***std***, поэтому мы должны либо пользоваться явным квалификатором ***std***::, либо перевести все имена в глобальное пространство имен:

using namespace std,

Для того чтобы не смешивать обсуждение выражений и обсуждение концепции модульности, я выбрал второй путь. В главах 8 и 9 обсуждаются способы разбиения нашего калькулятора на модули с помощью пространств имен и разбиение его на файлы исходных текстов. Во многих системах наряду со стандартными заголовочными файлами имеются их эквиваленты с расширением ***.h***, в которых классы, функции и т. д. объявляются и помещаются в глобальное пространство имен (§ 9.2 1, § 9.2.4, § Б.3.1).

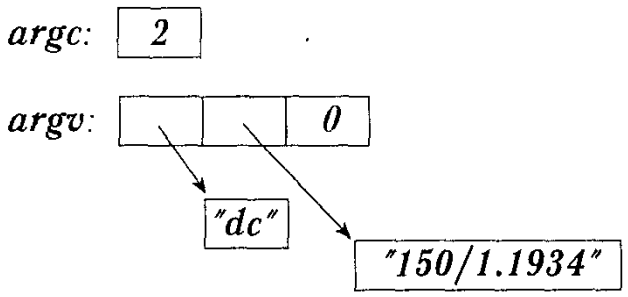
6.1.7. Параметры командной строки

После того как я написал и оттестировал программу, я обнаружил, что не очень удобно сначала запускать программу, затем вводить выражение и, наконец, выходить из нее. Наиболее часто я использовал калькулятор для вычисления единственного выражения. Если можно было бы ввести это выражение в качестве параметра в командной строке, удалось бы избежать нескольких нажатий клавиш.

Любая программа начинается с вызова функции ***main*** () (§ 3 2, § 9.4). При этом ***main*** () получает два аргумента. Первый определяет количество параметров (обычно называется ***argc***), а второй — массив параметров (обычно называемый ***argv***). Аргументы передаются в ***main*** () оболочкой, вызывающей программу; если оболочка не может передать аргументы, то ***argc*** устанавливается в ***0***. Так как соблюдается соглашение С о вызове ***main*** (), то при передаче аргументов используются строки в стиле С (***char****[***argc+1***]). Имя программы (так, как оно записано в командной строке) передается в ***argv***[***0***]. Список аргументов ограничен нулем, то есть ***argv***[***argc***]==***0***. Например, если командная строка выглядит:

dc 150/1 1934

то аргументы будут иметь следующие значения:



Несложно получить доступ к параметрам командной строки. Вопрос в том, как их использовать с минимальными изменениями в программе. Можно считывать из строки точно таким же образом, как из потока ввода. Не удивительно, что поток для чтения из строки называется ***istringstream***. К сожалению, нет элегантного способа заставить ***cin*** ссылаться на ***istringstream***. Поэтому нам нужно заставить функцию ввода калькулятора обращаться к ***istringstream***. Более того, нам нужно придумать, как заставить функцию ввода обращаться либо к ***istringstream***, либо к ***cin*** в зависимости от того, как выглядит командная строка.

Простым решением будет объявление глобального указателя ***input***, который указывает на используемый поток; все функции ввода будут на него ориентироваться:

```
istream* input;                          // указатель на поток ввода

int main (int argc, char* argv[])
{
        switch (argc) {
        case 1:                          // стандартный поток ввода
                input = &cin;
                break;
        case 2:                          // строка аргументов
                input = new istringstream (argv[1]);
                break;
        default:
                error ("слишком много параметров");
                return 1;
        }

        table["pi"] = 3.1415926535897932385;
        table["e"] = 2.7182818284590452354;

        while (input) {
                get_token ();
                if (curr_tok==END) break;
                if (curr_tok==PRINT) continue;
                cout << expr (false) << '\n';
        }

        if (input!=&cin) delete input;

        return no_of_errors;
}
```

Поток ***istringstream*** является потоком ввода, который читает из строки, переданной ему в качестве аргумента (§ 21.5.3). При достижении конца строки ***istringstream*** возвращает ***false***, точно так же, как другие потоки при достижении конца ввода (§ 3.6, § 21.3.3). Для того чтобы воспользоваться ***istringstream***, нужно включить заголовочный файл <***sstream***>.

Несложно модифицировать ***main*** () таким образом, чтобы она воспринимала несколько параметров командной строки, но похоже в этом нет необходимости, поскольку несколько выражений можно передать одним параметром:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

Я использую двойные кавычки, потому что точка с запятой является разделителем команд в моей версии UNIX. В других операционных системах могут быть другие соглашения по поводу передачи параметров при запуске.

Модификация всех процедур ввода таким образом, чтобы они использовали ****input*** вместо ***cin*** дабы иметь возможность читать из различных источников, является не самым элегантным решением. Этих изменений можно было бы избежать, если бы я предвидел такую ситуацию и реализовал бы что-нибудь типа ***input*** с самого начала. Более общим и полезным решением было бы задание источника ввода в качестве параметра для модуля калькулятора. Таким образом, главная проблема в нашем приме-

ре с калькулятором заключается в следующем: то, что я называю «калькулятором» — это только набор функций и данных. Нет ни модуля (§ 2.4), ни объекта (§ 2.5.2), который явно представлял бы собой калькулятор. Если бы я разрабатывал модуль калькулятора или тип калькулятора, я бы, естественно, задумался над тем, какие должны быть параметры (§ 8.5[3], § 10.6[16]).

6.1.8. Замечание о стиле

Программист, не знакомый с ассоциативными массивами, счел бы жульничеством использование стандартного библиотечного класса ***map*** для реализации таблицы символов. Это не так. И стандартная, и другие библиотеки создаются для того, чтобы ими пользовались. Очень часто библиотеки разрабатываются и реализуются с гораздо большей тщательностью, чем программист может позволить себе при написании кода, используемого только в одной программе.

Взглянув на код калькулятора, особенно в первой версии, мы видим, что в нем очень мало от традиционного низкоуровневого кода, присущего С. Многие традиционные трюки заменены библиотечными классами, такими как ***ostream***, ***string*** и ***map*** (§ 3.4, § 3.5, § 3.7.4, глава 17).

Обратите внимание, как редко встречаются арифметические операции, циклы и даже присваивания. Так и должно быть в коде, который не управляет непосредственно аппаратурой и не работает с низкоуровневым представлением данных.

6.2. Обзор операторов

В этом разделе проводится обзор выражений и ряд примеров. Каждый оператор сопровождается именем (или несколькими именами), обычно используемым(и) для его обозначения, и примером использования. В таблицах *class-name* означает имя класса, *namespace-name* — имя пространства имен, *qualified-name* — квалифицированное имя, *name* — какое-то имя, *member* — имя члена, *object* — выражение, дающее объект класса, *pointer* — выражение, дающее указатель (*pointer-to-member* — указатель на член), *expr* — некое выражение (*expr-list* — список выражений) и *lvalue* — выражение, обозначающее неконстантный объект. *type* может быть полностью произвольным именем типа (возможно с *, () и т. д.) только тогда, когда он записан в скобках; в других случаях имеются ограничения (§ A.5).

Синтаксис выражений не зависит от типа операндов. Смысл операторов таков, как описан ниже, только для встроенных типов (§ 4.1.1). Вы можете сами определить смысл операторов, применяемых к операндам типов, определяемых пользователем (§ 2.5.2, глава 11).

Операторы	
разрешение области видимости	*class-name* :: *member*
разрешение области видимости	*namespace-name* :: *member*
глобально	:: *name*
глобально	:: *qualified-name*
выбор члена	*object* . *member*
выбор члена	*pointer* -> *member*

Операторы (продолжение)	
доступ по индексу	*pointer* [*expr*]
вызов функции	*expr* (*expr-list*)
конструирование значения	*type* (*expr-list*)
постфиксный инкремент	*lvalue* ++
постфиксный декремент	*lvalue* --
идентификация типа	***typeid*** (*type*)
идентификация типа во время выполнения	***typeid*** (*expr*)
преобразование с проверкой во время выполнения	***dynamic_cast*** < *type* > (*expr*)
преобразование с проверкой во время компиляции	***static_cast*** < *type* > (*expr*)
преобразование без проверки	***reinterpret_cast*** < *type* > (*expr*)
константное преобразование	***const_cast*** < *type* > (*expr*)
размер объекта	***sizeof*** *expr*
размер типа	***sizeof*** (*type*)
префиксный инкремент	++ *lvalue*
префиксный декремент	-- *lvalue*
дополнение	~ *lvalue*
отрицание	! *expr*
унарный минус	- *expr*
унарный плюс	+ *expr*
адрес	& *lvalue*
разыменование	* *expr*
создать (выделить память)	***new*** *type*
создать (выделить память и инициализировать)	***new*** *type* (*expr-list*)
создать (разместить)	***new*** (*expr-list*) *type*
создать (разместить и инициализировать)	***new*** (*expr-list*) *type* (*expr-list*)
уничтожить (освободить память)	***delete*** *pointer*
уничтожить массив	***delete***[] *pointer*
приведение (преобразование типа)	(*type*) *expr*
выбор члена	*object* .* *pointer-to-member*
выбор члена	*pointer* ->* *pointer-to-member*
умножение	*expr* * *expr*
деление	*expr* / *expr*
остаток от деления (деление по модулю)	*expr* % *expr*
сложение (плюс)	*expr* + *expr*
вычитание (минус)	*expr* - *expr*
сдвиг влево	*expr* << *expr*
сдвиг вправо	*expr* >> *expr*
меньше	*expr* < *expr*
меньше или равно	*expr* <= *expr*
больше	*expr* > *expr*
больше или равно	*expr* >= *expr*
равно	*expr* == *expr*
не равно	*expr* != *expr*

Операторы (продолжение)	
побитовое И (AND)	*expr* & *expr*
побитовое исключающее ИЛИ (OR)	*expr* ^ *expr*
побитовое ИЛИ (OR)	*expr* | *expr*
логическое И (AND)	*expr* && *expr*
логическое ИЛИ (OR)	*expr* || *expr*
условное выражение	*expr* ? *expr* : *expr*
простое присваивание	*lvalue* = *expr*
умножение и присваивание	*lvalue* *= *expr*
деление и присваивание	*lvalue* /= *expr*
остаток и присваивание	*lvalue* %= *expr*
сложение и присваивание	*lvalue* += *expr*
вычитание и присваивание	*lvalue* −= *expr*
сдвиг влево и присваивание	*lvalue* <<= *expr*
сдвиг вправо и присваивание	*lvalue* >>= *expr*
И и присваивание	*lvalue* &= *expr*
ИЛИ и присваивание	*lvalue* |= *expr*
исключающее ИЛИ и присваивание	*lvalue* ^= *expr*
генерация исключения	***throw*** *expr*
запятая (последовательность)	*expr* , *expr*

В каждом блоке расположены операторы с одинаковым приоритетом. Операторы в блоке, расположенном выше, имеют более высокий приоритет. Например: ***a+b*c*** означает ***a+(b*c)***, а не ***(a+b)*c***, потому что * имеет более высокий приоритет, чем +.

Унарные операторы и операторы присваивания правоассоциативны, а все остальные левоассоциативны. Например, ***a=b=c*** означает ***a=(b=c)***, ***a+b+c*** означает ***(a+b)+c***.

Несколько грамматических правил нельзя выразить в терминах приоритетов (называемых также силой связывания) и ассоциативности. Например, ***a=b<c ? d=e : f=g*** означает ***a=((b<c) ? (d=e) : (f=g))***, но для того чтобы об этом догадаться, нужно обратиться к правилам грамматики (§ A.5).

Перед использованием грамматических правил из символов составляются лексические обозначения (лексемы, § A.3). Для получения каждой лексемы выбирается самая длинная из возможных последовательностей символов.

6.2.1. Результаты

Тип результата арифметических операций определяется набором правил, известных как «стандартные арифметические преобразования» (§ B.6.3). Общая цель, которая при этом преследуется, — получить результат «наибольшего» типа операнда. Например, если бинарный оператор имеет операнд с плавающей точкой, вычисления производятся с плавающей точкой и результатом является число с плавающей точкой. Если он имеет операнд ***long***, вычисления производятся с использованием арифметики длинных целых чисел и результат имеет тип ***long***. Операнды, которые меньше ***int*** (такие как ***bool*** и ***char***) преобразуются в ***int*** до применения оператора.

Результатом применения операторов сравнения ==, <= и т. д. является логическое значение. Смысл и тип результата для операторов, вводимых пользователем, определяется исходя из их объявления (§ 11.2).

Там где это приемлемо с точки зрения логики, результат применения оператора к операнду lvalue является lvalue, обозначающим этот операнд. Например:

```
void f(int x, int y)
{
        int j=x=y;              // значением x=y является значение x после присваивания
        int* p = &++x;          // p указывает на x
        int* q=&(x++);          // ошибка: x++ не является lvalue
                                // (это не значение, хранимое в x)
        int* pp=&(x>y ? x : y); // адрес целого с большим значением
}
```

Если и второй, и третий операнды оператора ?: являются lvalue и имеют одинаковый тип, результат имеет такой же тип и является lvalue. Такие правила работы с lvalue предоставляют большую гибкость при использовании операторов. Особенно это полезно при написании кода, который должен работать одинаково и эффективно как со встроенными, так и с определяемыми пользователем типами данных (например при написании шаблонов или программ, которые генерируют код C++).

Типом результата оператора ***sizeof*** является интегральный тип без знака, называемый ***size_t*** и определенный в ***<cstddef>***. Типом разности указателей будет интегральный тип со знаком, называемый ***ptrdiff_t*** и определенный в ***<cstddef>***.

Реализации не обязаны проверять на арифметическое переполнение и вряд ли какая-нибудь из них это делает. Например:

```
void f()
{
        int i = 1
        while (0<i) i++;
        cout << "переменная i стала отрицательной!" << i << '\n';
}
```

Эта функция со временем попытается прибавить единицу к наибольшему возможному целому. Что при этом произойдет — не определено, но, как правило, значение «перескакивает на другую сторону» и становится отрицательным (на моей машине ***-2147483648***). Аналогично, результат деления на ноль не определен, но при этом обычно происходит аварийное завершение программы. В частности, при переполнении (когда число становится либо слишком маленьким, либо слишком большим) и делении на ноль не генерируются стандартные исключения (§ 14.10).

6.2.2. Последовательность вычислений

Порядок вычислений подвыражений внутри выражений не определен. В частности, не стот предполагать, что выражения вычисляются слева направо. Например:

```
int x = f (2) + g (3);      // неизвестно, что вызовется первым — f () или g()
```

При отсутствии ограничений на порядок вычислений можно сгенерировать более качественный код. Однако отсутствие ограничений на порядок вычислений может привести к неопределенным результатам. Например,

```
int i = 1,
v[i] = i++,          // результат не определен
```

может вычисляться либо как *v[1] = 1* либо как *v[2] = 1*, либо привести к еще более странным результатам. Компиляторы могут выдавать предупреждающие сообщения при возникновении такой неоднозначности. К сожалению, большинство из них этого не делает.

Операторы , (запятая), && (логическое И) и || (логическое ИЛИ) гарантируют, что операнд слева будет вычислен до операнда справа. Например, *b=(a=2, a+1)* присвоит *3* переменной *b*. Примеры использования || и && можно найти в § 6.2.3. Для встроенных типов второй операнд оператора && вычисляется, только если его первый операнд равен ***true***. Для оператора || второй аргумент вычисляется только в том случае, если первый аргумент равен ***false***. Иногда такой подход называют *быстрым вычислением*. Обратите внимание, что , (запятая) в качестве указателя последовательности логически отличается от запятой, используемой в качестве разделителя аргументов при вызове функций. Например:

```
f1 (v[i], i++),      // два аргумента
f2 ((v[i], i++)),    // один аргумент
```

Вызов *f1* осуществляется с двумя аргументами, *v[i]* и *i++*, и порядок вычисления аргументов не определен. Расчет на определенный порядок вычисления аргументов является исключительно плохим стилем и приводит к непредсказуемому поведению программы. Вызов *f2* имеет один аргумент — последовательность выражений, разделенных с запятой, что эквивалентно *i++*.

Для явного задания группирования можно пользоваться скобками. Например, *a*b/c* означает *(a*b)/c*, поэтому нужно пользоваться скобками, чтобы вычислить *a*(b/c)*; *a*(b/c)* может быть вычислено как *(a*b)/c*, только в том случае, если пользователь не заметит разницы. В частности, для многих вычислений с плавающей точкой результаты *a*(b/c)* и *(a*b)/c* могут значительно отличаться, поэтому компилятор будет обрабатывать выражения точно так, как они записаны.

6.2.3. Приоритет операторов

Уровни приоритетов и правила ассоциативности выбраны из соображений наиболее типичного использования. Например,

```
if ( i<=0 || max<i ) //...
```

означает «если *i* меньше или равно *0* или если ***max*** меньше *i*». То есть, это эквивалентно следующему:

```
if ( (i<=0) || (max<i) ) //...
```

а не

```
if ( i<= (0 || max) <i ) //...
```

что хотя и допустимо, но вряд ли имеет смысл.

Тем не менее следует применять скобки каждый раз, когда программист сомневается в правилах. Чем более сложными становятся выражения, тем чаще используются скобки. Следует, однако, помнить, что сложные выражения являются источниками ошибок. Поэтому, если вы ощущаете непреодолимую потребность в скобках,

подумайте о том, чтобы завести дополнительную переменную для разбиения одного сложного выражения на несколько более простых.

Есть случаи, когда последовательность выполнения операторов неочевидна. Например:

```
if (i&mask==0)          // не то! выражение == будет операндом для &
```

В примере не происходит наложение маски (***mask***) на ***i*** и затем проверки результата на равенство нулю. Так как оператор == имеет более высокий приоритет, чем &, выражение интерпретируется как ***i*** & (***mask==0***). К счастью, в таких случаях компилятор может легко обнаружить ошибку и выдать предупреждение. В нашем примере скобки очень важны:

```
if ((i&mask) == 0) // ...
```

Следующее выражение означает совсем не то, что ожидал бы математик:

```
if (0<=x<=99) // ...
```

Это выражение допустимо, но интерпретируется как (***0*** <= ***x***) <= ***99***, где результат первого сравнения равен либо ***true*** либо ***false***. Затем это логическое значение преобразуется в ***1*** или ***0*** и сравнивается с ***99***, что всегда выдаст ***true***. Чтобы проверить, находится ли значение ***x*** в диапазоне ***0..99***, мы можем написать:

```
if (0<=x && x<=99) //..
```

Новички обычно делают следующую ошибку — используют знак = (присваивание) вместо == (сравнение на равенство):

```
if (a=7)                // не то! в условии присваивается константа
```

Эта ошибка естественна, потому что во многих языках = означает «равно». Такие вещи компилятор также может легко обнаружить, и многие из них это делают.

6.2.4. Побитовые логические операторы

Побитовые логические операторы &, |, ^, ~, >> и << применяются к интегральным типам и перечислениям — то есть к ***bool***, ***char***, ***short***, ***int***, ***long*** (возможно, с модификатором ***unsigned***). Для определения типа результата выполняются обычные арифметические преобразования (В.6.3).

Типичным применением побитовых логических операторов является реализация небольшого множества (вектора бит). В этом случае каждый бит целого без знака представляет элемент множества, а количество бит ограничивает количество его элементов. Бинарный оператор & интерпретируется как пересечение, | — как объединение, ^ — как симметричная разность и ~ — как дополнение. Для обозначения членов такого множества можно воспользоваться перечислением. Приведем небольшой пример, заимствованный из реализации ***ostream***:

```
enum ios_base iostate {
        goodbit=0, eofbit=01, failbit=010, badbit=0100
};
```

Можно устанавливать и проверять состояние следующим образом:

```
state = goodbit,
// ...
if (state & (badbit|failbit))          // проблемы с потоком
```

Дополнительные скобки необходимы, так как & имеет более высокий приоритет, чем |.

Функция, которая встретила конец файла, может сообщить об этом следующим образом:

```
state |= eofbit;
```

Оператор | используется для добавления к состоянию ***eofbit***. Простое присваивание ***state=eofbit*** очистило бы все остальные биты.

Эти флаги состояния потока доступны и вне реализации потока. Например, мы можем посмотреть, как отличаются состояния двух потоков следующим образом:

```
int diff = cin.rdstate() ^ cout.rdstate();      // rdstate() возвращает состояние
```

Определение различия состояний потоков не является типичной задачей. Для других подобных типов определение различий более существенно. Например, сравнение битового вектора, который представляет набор обрабатываемых прерываний, с другим вектором, представляющим прерывания, ожидающие обработки, является вполне типичным.

Обратите внимание, что вся эта суета с битами производится в реализации ***iostream***, а не в пользовательском интерфейсе. Удобства манипулирования битами может быть очень важным моментом, но из соображений надежности, сопровождаемости, переносимости и т. д. работу с битами лучше вести на нижних уровнях системы. За более подробным описанием понятия множества обратитесь к классам стандартной библиотеки ***set*** (§ 17.4.3), ***bitset*** (§ 17.5.3) и ***vector<bool>*** (§ 16.3.11).

Использование полей (§ B.8.1) является удобной формой осуществления сдвигов и маскирования для извлечения битовых полей из слова. Это, конечно, можно сделать и с помощью побитовых логических операций. Например, можно извлечь средние 16 бит из 32-разрядного ***long*** следующим образом:

```
unsigned short middle(long a) { return (a>>8) &0xffff; }
```

Не путайте побитовые логические операции с логическими операторами &&, || и !. Последние возвращают ***true*** или ***false*** и в основном используются для сравнения в операторах ***if***, ***while*** или ***for*** (§ 6.3.2, § 6.3.3). Например, ***!0*** (не ноль) имеет значение ***true***, в то время как ***~0*** (дополнение нуля) означает цепочку битов, состоящую из единиц, что в двоичном дополнительном представлении означает ***–1***.

6.2.5. Инкремент и декремент

Оператор ++ используется для явного указания операции увеличения вместо менее явной записи, состоящей из комбинации сложения и присваивания. По определению, *++lvalue* означает *lvalue* += *1*, что в свою очередь означает *lvalue* = *lvalue* + *1*, при условии, что *lvalue* не имеет побочных эффектов. Выражение, означающее объект, с которым производится инкремент, вычисляется только один раз. Подобным же образом записывается оператор декремента --. Операторы ++ и -- могут быть как постфиксными, так и префиксными. Значением *++x* будет новое (увеличенное) значение *x*. Например, *y=++x* эквивалентно *y=(x+=1)*. Напротив, значением *x++* является старое значение *x*. Например, *y=x++* эквивалентно *y=(t=x, x+=1, t)*, где тип переменной *t* такой же, как у *x*.

Также как при сложении и вычитании указателей, операторы ++ и -- над указателями работают в единицах размеров элементов, на которые ссылаются указатели; *p++* приказывает *p* указать на следующий элемент массива (§ 5.3.1).

Операторы инкремента и декремента особенно полезны при модификации переменных в циклах. Например, копирование строки, ограниченной нулем, можно записать следующим образом:

```
void cpy (char* p, const char* q)
{
        while (*p++ = *q++);
}
```

Также как и C, C++ и любят и ненавидят за возможность написания подобного, ориентированного на выражения, кода. Так как выражение

```
while (*p++ = *q++);
```

выглядит более чем странно для программистов, пишущих не на C, а такой код довольно часто встречается в C и C++, стоит рассмотреть этот фрагмент более подробно.

Сначала рассмотрим более привычный способ копирования массивов символов:

```
int length = strlen (q);
for (int i=0; i<=length; i++) p[i] = q[i];
```

Это довольно расточительно. Длина строки, ограниченной нулем, определяется путем просмотра этой строки и поиском завершающего нуля. Поэтому мы просматриваем строку дважды: первый раз при определении размера, и второй — при копировании. Перепишем пример следующим образом:

```
int i;
for (i=0; q[i]!=0; i++) p[i] = q[i];
p[i] = 0;                    // завершающий ноль
```

Можно убрать переменную *i*, используемую в качестве индекса, так как *p* и *q* являются указателями:

```
while (*q != 0) {
        *p = *q;
        p++;                 // указывает на следующий символ
        q++;                 // указывает на следующий символ
}
*p = 0;                      // завершающий ноль
```

Постинкрементная операция сначала использует значение, а потом его увеличивает, потому мы можем переписать цикл следующим образом:

```
while (*q != 0) {
        *p++ = *q++;
}
*p = 0,                      // завершающий ноль
```

Значением выражения **p++ = *q++* является **q*. Поэтому, мы можем переписать пример так:

```
while ((*p++ = *q++) != 0) { }
```

В этом примере, равенство **q* нулю обнаруживается только после того, как символ скопирован в **p* и выполнен инкремент *p*. Таким образом мы избавились от строки, в которой производилось присваивание завершающего нуля. И, наконец, мы можем сокра-

тить код, убрав пустой блок операторов. Кроме того, сравнение «! = *0*» является лишним, так как указатель и так сравнивается с нулем. Таким образом, мы получаем выражение, с которого и начали:

```
while (*p++ = *q++);
```

Является ли эта версия менее читабельной? Для опытного программиста на C или C++ — нет. Является ли эта версия более эффективной с точки зрения времени выполнения или размера кода, чем предыдущая? За исключением варианта с вызовом ***strlen*** (), далеко не обязательно. Какая версия эффективней, зависит от архитектуры машины и компилятора.

Самым эффективным способом копирования строки, ограниченной нулем, на вашей конкретной машине должно быть использование стандартной функции копирования строк:

```
char* strcpy (char*, const char*);        // находится в <string.h>
```

Можно воспользоваться более обобщенным алгоритмом копирования ***copy*** (§ 2.7.2, § 18.6.1). Там, где это возможно, пользуйтесь средствами стандартной библиотеки для работы с указателями и байтами. Стандартные библиотечные функции могут быть встроены в место вызова (§ 7.1.1) либо реализованы с использованием специализированных машинных инструкций. Поэтому проведите тщательные измерения, для того чтобы убедиться в том, что некоторый фрагмент вручную написанного кода работает быстрее библиотечной функции.

6.2.6. Свободная память

Время жизни именованного объекта определяется его областью видимости (§ 4.9.4). С другой стороны, часто возникает необходимость в создании объектов, которые существуют вне зависимости от области видимости, в которой они были созданы. Типичным примером является создание объектов, которые используются после возвращения из функции, в которой они были созданы. Такие объекты создаются при помощи оператора ***new*** и уничтожаются при помощи оператора ***delete***. Говорят, что место под объекты, созданные при помощи оператора ***new***, выделяется из «свободной памяти» (такие объекты называют еще «объектами из кучи» или «объектами, размещаемыми в динамической памяти»).

Давайте посмотрим, как мы могли бы написать компилятор в стиле настольного калькулятора (§ 6.1). Функции синтаксического анализа могли бы следующим образом строить дерево выражений для его последующего использования генератором кода:

```
struct Enode {
        Token_value oper;
        Enode* left;
        Enode* right;
        // ...
}

Enode* expr (bool get)
{
        Enode* left = term (get);

        for (;;)
```

```
            switch (curr_tok) {
            case PLUS:
            case MINUS:
            {       Enode* n = new Enode;       // создать Enode в свободной памяти
                    n->oper = curr_tok;
                    n->left = left;
                    n->right= term (true);
                    left = n;
                    break;
            }
            default:
                    return left;        // вернуть узел
            }
}
```

Генератор кода может использовать эти узлы и затем их удалить:

```
void generate (Enode* n)
{
        switch (n->oper) {
        case PLUS:
                // ...
                delete n;               // удалить Enode из свободной памяти
        }
}
```

Объект, созданный при помощи оператора ***new***, существует до тех пор, пока он не удален при помощи оператора ***delete***. После этого память, занимаемая объектом, может быть снова использована следующим оператором ***new***. Реализация C++ не гарантирует наличие «сборщика мусора», осуществляющего поиск объектов, на которые отсутствуют ссылки, и делающего эту память доступной для повторного использования оператором ***new***. Соответственно, я полагаю, что объекты, созданные при помощи оператора ***new***, удаляются вручную при помощи оператора ***delete***. При наличии сборщика мусора в большинстве случаев можно было бы обойтись без использования оператора ***delete*** (§ B.9.1).

Оператор ***delete*** можно применять либо к указателю, возвращенному оператором ***new***, либо к нулю. Применение ***delete*** к нулю не вызывает никаких действий.

Можно определить и более специализированную версию оператора ***new*** (§ 15.6).

6.2.6.1. Массивы

При помощи оператора ***new*** можно создавать массивы объектов. Например:

```
char* save_string (const char* p)
{
        char* s = new char[strlen (p)+1];
        strcpy (s, p);                  // скопировать p в s
        return s;
}

int main (int argc, char* argv[])
{
        if (argc < 2) exit (1);
```

```
        char* p = save_string (argv[1]);
        // ...
        delete[] p;
}
```

Оператор ***delete*** используется для удаления отдельных объектов, ***delete***[] — для удаления массивов.

Для того чтобы освободить память, выделенную оператором ***new***, операторы ***delete*** и ***delete***[] должны иметь возможность определить размер объекта. Из этого следует, что стандартная реализация ***new*** выделяет памяти немного больше, чем потребовалось бы для статического объекта. Как правило, используется одно дополнительное слово для хранения размера объекта.

Обратите внимание, что ***vector*** (§ 3.7.1, § 16.3) является корректным объектом, и поэтому память под него можно выделить и освободить при помощи простых ***new*** и ***delete***. Например:

```
void f(int n)
{
        vector<int>* p = new vector<int>(n);      // объект
        int* q = new int[n];                      // массив объектов
        // ...
        delete p;
        delete[] q;
}
```

Оператор ***delete***[] может применяться только к указателю на массив, полученному посредством ***new***, или к нулю. Применение ***delete***[] к нулю не дает какого-либо эффекта.

6.2.6.2. Отсутствие памяти

Операторы выделения и освобождения памяти ***new***, ***delete***, ***new***[] и ***delete***[] реализованы при помощи функций, представленных в заголовочном файле <***new***> (§ 19.4.5):

```
void* operator new (size_t);        // выделяет память для индивидуального объекта
void operator delete (void* p);     // если p!=0, то освобождает память,
                                    // выделенную new( )

void* operator new[](size_t);       // выделяет память для массива
void operator delete[](void* p);    // если p!=0, то освобождает память,
                                    // выделенную new[ ]( )
```

Для выделения памяти под объект ***new*** вызывает ***operator new*** (), выделяющий подходящее количества байт. Аналогично, для выделения памяти под массив ***new*** вызывает ***operator new***[]().

Стандартная реализация ***operator new*** () и ***operator new***[]() не инициализирует выделяемую память.

Что происходит, когда ***new*** не может найти достаточного объема памяти? По умолчанию генерируется исключение ***bad_alloc***. Например:

```
void f()
{
        try {
                for (;;) new char[10000];
        }
```

```
        catch (bad_alloc) {
            cerr << "Нет свободной памяти!\n";
        }
}
```

Когда вся память будет использована, вызовется обработчик исключения ***bad_alloc***.

Мы можем вручную указать, какие действия должны выполниться, когда оператор ***new*** не может выделить память. При неудачной попытке выделения памяти ***new*** сначала вызывает функцию, указанную аргументом при вызове ***set_new_handler*** (), объявленной в <***new***> (если такой вызов вообще имел место). Например:

```
void out_of_store ()
{
    cerr << "неуспешное завершение оператора new:"
            "нет свободной памяти\n";
    throw bad_alloc;
}

int main ()
{
    set_new_handler (out_of_store);          // задаем новый обработчик
                                             // на случай нехватки памяти
    for (;;) new char [10000];
    cout << "сделано\n";
}
```

Слово «сделано» никогда не будет выведено. Вместо этого, программа напишет:

```
неуспешное завершение оператора new: нет свободной памяти
```

В § 14.4.5 приведена приемлемая реализация ***operator new*** (), которая проверяет наличие обработчика и при его отсутствии генерирует исключение ***bad_alloc***. Новый обработчик может выполнять и более осмысленные действия, чем просто завершение программы. Если вы знаете, как работают ***new*** и ***delete***, например, если вы реализовали свои собственные ***operator new*** () и ***operator delete*** (), обработчик может попытаться найти память для ***new***. Другими словами, пользователь может реализовать сборщик мусора, делая таким образом явное использование ***delete*** необязательным. Правда, такая задача определенно не для новичка. Большинству программистов, нуждающихся в автоматическом сборщике мусора, можно посоветовать воспользоваться уже написанным и оттестированным (§ B.9.1).

Реализуя обработчик, мы осуществляем проверку наличия свободной памяти при каждом использовании ***new***. Существует два различных способа управления выделением памяти. Мы можем либо реализовать нестандартные функции выделения и освобождения памяти (§ 15.6) для стандартного использования ***new***, либо полагаться на дополнительную информацию о выделении памяти, предоставляемую пользователем (§ 10.4.11, § 19.4.5).

6.2.7. Явное преобразование типов

Иногда нам приходится иметь дело с «сырой» памятью, то есть памятью, в которой хранятся или будут храниться объекты неизвестного компилятору типа. Например, функция выделения памяти может возвратить ***void****, указывающий на только что вы-

деленную память, или мы хотим работать с целым значением, как с адресом устройства ввода/вывода:

```
void* malloc(size_t);

void f()
{
    int* p = static_cast<int*>(malloc(100));        // память выделена под
                                                    // целые
    IO_device* dl =
        reinterpret_cast<IO_device*>(0Xff00);       // устройство по
                                                    // адресу 0Xff00
    // ...
}
```

Компилятор не знает тип объекта, на который указывает ***void****. Он также не знает, является ли ***0Xff00*** корректным адресом. Следовательно, правильность преобразований полностью на совести программиста. Явное преобразование типа, часто называемое *приведением типа*, иногда очень важно. Однако, традиционно оно используется неоправданно часто и является основным источником ошибок.

Оператор ***static_cast*** осуществляет преобразование родственных типов, например указателя на один тип к указателю на другой тип из той же иерархии классов, целый тип в перечисление или тип с плавающей точкой в интегральный. Оператор ***reinterpret_cast*** управляет преобразованиями между несвязанными типами, например целых в указатели или указателей в другие (несвязанные) указатели. Такое различие позволяет компилятору осуществлять минимальную проверку типов при ***static_cast***, а программисту — легче обнаружить опасные преобразования, представляемые ***reinterpret_cast***. Некоторые преобразования ***static_cast*** являются переносимыми, но ***reintepret_cast*** практически никогда не переносимо. Хотя никаких гарантий относительно ***reinterpret_cast*** дать нельзя, в результате обычно получается значение нового типа, состоящее из той же цепочки битов, что и аргумент преобразования. Если результат содержит не меньше бит, чем исходное значение, мы сможем преобразовать (с помощью ***reinterpret_cast***) результат обратно в исходный тип и использовать получившееся значение. То, что результатом преобразования ***reinterpret_cast*** можно пользоваться, гарантировано только если тип его результата есть в точности тип, использованный для определения задействованного в преобразовании значения.

Если у вас возникло искушение воспользоваться явным преобразованием, задумайтесь над тем, так ли это необходимо. В большинстве случаев в C++ нет нужды в явном преобразовании типов даже тогда, когда это необходимо в C (§ 1.6) или в более ранних версиях C++ (§ 1.6.2, § Б.2.3). Во многих программах можно избежать явного преобразования типов. В остальных — локализовать их в нескольких процедурах. В этой книге явное преобразование типов используется в реалистичных ситуациях только в § 6.2.7, § 7.7, § 13.5, § 13.6, § 15.4 § 17.6.2.3, § 25.4.1 и § Е3.1.

Также имеются преобразование ***dynamic_cast***, выполняемое и проверяемое на этапе выполнения (§ 15.4.1), и ***const_cast*** (§ 15.4.2.1), которое аннулирует действие модификатора ***const*** и ***volatile***.

C++ унаследовал от C форму записи ***(T)e***, означающую любое преобразование, которое может быть выражено комбинацией ***static_cast***, ***reinterpret_cast*** и ***const_cast***, для получения значения типа ***T*** из выражения ***e*** (§ Б.2.3). Такой стиль намного опас-

нее, чем именованные операторы преобразования, потому что приведенную форму записи сложнее отслеживать в большой программе и вид преобразования, который имел в виду программист, не очевиден. Ведь (*T*)*e* может осуществлять и переносимое преобразование между близкими типами, и непереносимое преобразование между несвязанными типами, и, наконец, аннулировать действие модификатора ***const*** для указателя. Не зная конкретно типы *T* и *e*, невозможно сказать, что именно делает (*T*)*e*.

6.2.8. Конструкторы

Создание значения типа *T* из значения *e* можно выразить при помощи функционального обозначения *T* (*e*). Например:

```
void f(double d)
{
        int i = int (d),                // отбросить дробную часть d
        complex z = complex (d),        // создать комплексное число из d
        //...
}
```

Конструкцию *T* (*e*) иногда называют *функциональным стилем приведения (типов)*. К сожалению, для встроенного типа *T* запись *T* (*e*) эквивалентна (*T*) *e* (§ 6.2.7). Соответственно, для многих встроенных типов использование *T* (*e*) не безопасно. Для арифметических типов значения могут оказаться урезанными и даже явное преобразование более длинных целых в более короткие типы (например, ***long*** в ***char***) может привести к непереносимому, зависящему от реализации поведению. Я пытаюсь использовать такую форму записи исключительно в случаях, когда конструирование значения хорошо определено, например для «сужающих» арифметических преобразований (§ B.6), преобразований из целых в перечисления (§ 4.8) и для создания объектов типов, определяемых пользователем (§ 2.5.2, § 10.2.3).

Преобразование указателей нельзя задавать прямо при помощи нотации *T* (*e*). Например, ***char****(2) является синтаксической ошибкой. К сожалению, соответствующую защиту против столь опасных преобразований можно обойти при помощи ***typedef*** (§ 4.9.7) для типов указателей.

Форма записи *T* () используется для создания значения по умолчанию типа *T*. Например:

```
void f (double d)
{
        int j = int (),                 // значение int по умолчанию
        complex z = complex (),         // комплексное число по умолчанию
        //...
}
```

Результатом явного применения конструктора для встроенных типов является ***0***, преобразованный в соответствующий тип (§ 4.9.5). Таким образом, ***int*** () является формой записи ***0***. Для типа *T*, определяемого пользователем, значение *T* () определяется конструктором по умолчанию (если он имеется) (§ 16.3.4, § 17.4.1.2).

Использование конструкторов для встроенных типов важно, в частности, при написании шаблонов, когда программист не знает, будет ли в качестве параметра шаблона указан встроенный тип или тип, определяемый пользователем.

6.3. Инструкции

Ниже приведена сводка инструкций C++ и представлены некоторые примеры:

Синтаксис инструкций[1]

инструкция:
объявление
{ *последовательность_инструкций*$_{opt}$ }
try { *последовательность_инструкций*$_{opt}$ } *список_обработчиков*
выражение$_{opt}$;

if (*условие*) *инструкция*
if (*условие*) *инструкция* ***else*** *инструкция*
switch (*условие*) *инструкция*

while (*условие*) *инструкция*
do *инструкция* ***while*** (*выражение*);
for (*инициализирующая_инструкция условие*$_{opt}$; *выражение*$_{opt}$) *инструкция*

case *константное_выражение* : *инструкция*
default : *инструкция*
break ;
continue ;

return *выражение*$_{opt}$;

goto *идентификатор* ;
идентификатор : *инструкция*

последовательность_инструкций :
инструкция последовательность_инструкций$_{opt}$

условие :
выражение
спецификатор_типа объявитель = *выражение*

список_обработчиков :
catch (*объявление_исключения*) { *последовательность_инструкций*$_{opt}$ }
список_обработчиков список_обработчиков$_{opt}$

Здесь $_{opt}$ означает «optional», то есть «необязательно».

Обратите внимание, что объявление является инструкцией, а оператор присваивания и вызов функции инструкциями не являются; присваивание и вызов функции — это выражения. Инструкции для обработки исключений — блок ***try*** — описаны в § 8.3.1.

[1] Здесь возникает следующее затруднение, очень неприятное как для редактора (переводчиков) книги, так, вероятно, и для читателей. Используемый в оригинале термин «statement» часто переводится на русский язык как «оператор». Мы все привыкли, что ***for***, ***if***, ***case*** и т. д. — это операторы, что неверно (по крайней мере в контексте C++). Термином «оператор» (operator) обозначается присваивание (=), сложение (+) и т. д. Операторы могут быть замещены в пользовательском классе. Большинство операторов имеет возвращаемое значение. В связи с тем, что понятия «operator» и «statement» в оригинале четко разделены, мы вынуждены сделать это и в переводе. Поэтому для перевода «statement» мы выбрали отдельный термин — «инструкция», который также достаточно распространен в русскоязычной литературе. Мы надеемся, что это не вызовет у читателей недоумения. Таким образом, ***for***, ***if***, ***case*** и т. д. — это инструкции, а не операторы. — *Примеч. ред*

6.3.1. Объявления в качестве инструкций

Объявление является инструкцией. Если только переменная не объявлена как *static*, инициализатор выполняется каждый раз, когда поток управления достигает объявления (см. также § 10.4.8). Объявления разрешатся помещать в любом месте, где допустима инструкция (и еще в некоторых местах; § 6.3.2.1, § 6.3.3.1). Это сделано для того, чтобы программист мог свести к минимуму количество ошибок, вызываемых неинициализированными переменными, и для лучшей локализации кода. Редко возникают ситуации, когда переменную нужно объявить до того, как она примет некоторое значение. Например:

```
void f(vector<string>& v, int i, const char* p)
{
        if (p==0) return;
        if (i<0 || v.size()<=i) error("неправильный индекс");
        string s = v[i];
        if (s == p) {
                //...
        }
        //...
}
```

Возможность помещения объявлений после исполняемого кода может иметь существенное значение для многих констант и для стилей программирования с однократным присваиванием, когда значение объекта не меняется после инициализации. В случае с типами, определяемыми пользователями, отказ от инициализации до тех пор, пока не станет доступен приемлемый инициализатор, может привести к лучшей производительности. Например,

```
string s; /* ... */ s = "Лучшее – враг хорошего.";
```

вполне может оказаться медленнее, чем

```
string s = "Вольтер";
```

Наиболее частой причиной объявления переменной без инициализации является случай, когда для инициализации требуется инструкция. Примерами могут служить переменные, в которые осуществляется ввод, и массивы.

6.3.2. Инструкции выбора

Значение может проверяться в инструкциях *if* и *switch*:

```
if (условие) инструкция
if (условие) инструкция else инструкция
switch (условие) инструкция
```

Результатом применения операторов сравнения

```
==   !=   <   <=   >   >=
```

является *true*, если сравнение истинно, и *false* в противном случае.

Если выражение в инструкции *if* не равно нулю, выполняется первая (и возможно единственная) инструкция, в противном случае — вторая (если она указана). Отсюда

следует, что в качестве условия могут использоваться любые арифметические выражения или выражения с указателями. Например, если ***x*** целое, то

```
if (x) //...
```

означает

```
if (x != 0) //...
```

Для указателя ***p***

```
if (p) //...
```

является явной проверкой того, что «***p*** указывает на допустимый объект» (т. е. надлежащим образом инициализированный), в то время как

```
if (p != 0) //...
```

осуществляет ту же самую проверку менее явно, сравнивая указатель со значением, о котором известно, что оно не может указывать на объект. Обратите внимание, что «нулевой» указатель не обязательно представляется нулевой последовательностью битов на всех машинах (§ 5.1.1). Все компиляторы, которые я проверял, генерируют одинаковый код для обеих форм сравнения.

Логические операторы

```
&&  ||  !
```

наиболее часто используются в сравнениях. Операторы && и || вычисляют второй аргумент, только если в этом есть необходимость. Например,

```
if (p && 1<p->count)    //...
```

сначала сравнивает ***p*** с нулем. Проверка ***1<p->count*** осуществляется только в том случае, если ***p*** не равно нулю.

Некоторые условные *if-инструкции* можно записывать в виде *условных выражений*. Например,

```
if (a <= b)
        max = b;
else
        max = a,
```

лучше выглядит в следующем виде:

```
max = (a <= b) ? b . a;
```

Заключать сравнение в скобки необязательно, но, по-моему, так легче читать код.

switch-инструкцию всегда можно заменить набором условных инструкций. Например,

```
switch (val) {
case 1.
        f (),
        break;
case 2·
        g ();
        break;
```

```
default:
        h ();
        break;
}
```

может быть записано как

```
if (val == 1)
        f ();
else if (val == 2)
        g ();
else
        h ();
```

Смысл одинаковый, но первая форма (*switch*) предпочтительней, потому что лучше отражает идею сравнения (сопоставление значения с набором констант). Поэтому *switch* легче воспринимается при чтении нетривиальных примеров и сгенерированный код может оказаться лучше.

Отдавайте себе отчет в том, что каждая ветвь *case* должна как-нибудь завершаться, если вы не хотите, чтобы продолжали выполняться и другие инструкции *case*. Например,

```
switch (val) {
case 1:
        cout << "case 1\n";
case 2
        cout << "case 2\n";
default:
        cout << "default: ни один случай не соответствует";
}
```

при *val==1* к великому изумлению непосвященных выдаст

```
case 1
case 2
default: ни один случай не соответствует
```

Редкие случаи, когда такой подход осуществляется намеренно, лучше комментировать. Обычным способом завершения *case* является инструкция *break*, но часто полезна и *return* (§ 6.1.1).

6.3.2.1. Объявления в условиях

Во избежание случайного неправильного использования переменных их лучше вводить в наименьшей возможной области видимости. В частности, локальную переменную лучше объявлять в тот момент, когда ей надо присвоить значение. В этом случае исключаются попытки использования переменной до момента ее инициализации.

Одним из самых элегантных применений этих идей является объявление переменной в условии. Рассмотрим пример:

```
if (double d = prim (true)) {
        left /= d;
        break;
}
```

В примере происходит объявление переменной ***d***, и после инициализации она проверяется в условии. Область видимости переменной ***d*** простирается от ее точки объявления до конца инструкции, контролируемой условием. Если бы в инструкции ***if*** была ветвь ***else***, областью видимости ***d*** были бы обе ветви.

Очевидной и традиционной альтернативой является объявление переменной ***d*** до условия, но в этом случае область видимости началась бы до места использования переменной и продолжалась бы после завершения ее «сознательной» жизни:

```
double d,
// ...
d2 = d,              // Внимание!
// ...
if (d = prim (true)) {
      left /= d,
      break,
}
// ...
d = 2 0,             // два несвязанных использования d
```

Объявление переменных в условиях, кроме логических преимуществ, приводит к более компактному исходному коду.

Объявление в условии должно объявлять и инициализировать единственную переменную или константу.

6.3.3. Инструкции циклов

Циклы можно выразить инструкциями трех видов — ***for***, ***while*** и ***do***:

```
while (условие) инструкция
do инструкция while (выражение);
for (инициализирующая_инструкция условие_opt; выражение_opt) инструкция
```

Каждая из этих инструкций приводит к многократному выполнению *инструкций* (называемых также *управляемыми* инструкциями или *телом цикла*) до тех пор, пока условие не примет значение ***false***, либо программист не прервет цикл каким-либо другим способом.

for-инструкция предназначена для реализации регулярных циклов. Переменную цикла, условие завершения и выражение, которое модифицирует переменную цикла, можно записать в единственной строке. Это, как правило, увеличивает читабельность кода и, следовательно, ведет к уменьшению ошибок. Если инициализация не требуется, соответствующая часть может быть опущена. Если отсутствует *условие*, ***for-инструкция*** будет либо выполняться вечно, либо будет явно прервана при помощи ***break***, ***return***, ***goto***, ***throw*** или менее явным способом, вроде вызова функции ***exit*** () (§ 9.4.1.1). Если *выражение* пропущено, мы должны обновлять некоторую переменную цикла в теле цикла. Если цикл не принадлежит типу «введи переменную цикла, проверь условие, модифицируй переменную цикла», его лучше записать в виде инструкции ***while***. Полезным примером использования цикла ***for*** является случай, когда явно не указывается условие завершения:

```
for(;;) {      // «вечно»
        //...
}
```

while-инструкция просто исполняет тело цикла до тех пор, пока условие не станет *false*. Я предпочитаю пользоваться циклом *while* вместо *for*, когда нет очевидной переменной цикла, либо когда естественным местом для ее модификации является середина тела цикла. Цикл ввода является примером цикла, где нет очевидной переменной цикла:

```
while(cin>>ch) //...
```

По моему мнению, *do-инструкция* является источником ошибок и путаницы. Причина заключается в том, что ее тело всегда один раз выполняется до проверки условия. Но для того чтобы тело выполнялось корректно, что-то типа условия должно выполняться даже при первом проходе. Чаще, чем я мог бы предположить, я сталкивался со случаями, когда условие не выполнялось (как от него ожидали) либо когда программа была первый раз написана и отлажена, либо позднее, после того, как предшествующий код был изменен. Также я предпочитаю явные условия, которые я мог бы легко проверить. Соответственно, я не склонен использовать *do-инструкции*.

6.3.3.1. Объявления в for-инструкции

Переменную можно объявить в части инициализации инструкции *for*. В этом случае область видимости переменной (или переменных) простирается до конца *for-инструкции*. Например:

```
void f(int v[], int max)
{
        for (int i = 0; i<max; i++) v[i] = i*i;
}
```

Если требуется узнать значение индекса после выхода из цикла *for*, переменную надо объявить вне его (пример § 6.3.4).

6.3.4. goto

В C++ имеется пресловутая инструкция *goto*:

```
goto идентификатор;
идентификатор: инструкция
```

Существует очень мало примеров использования *goto* при программировании на высоком уровне, но эта инструкция может быть крайне полезна, когда код C++ генерируется программой, а не человеком. Например, *goto* могут быть использованы в анализаторе, сгенерированном по описанию грамматики. Инструкция *goto* может также оказаться полезной в редких случаях, когда требуется максимальная производительность, например во внутреннем цикле приложения, работающего в реальном времени.

Одним из немногих разумных способов использования *goto* в обычном коде является выход из вложенного цикла или *switch-инструкции* (*break* прекращает выполнение только самого внутреннего цикла или *switch-инструкции*). Например:

```
void f()
{
        int i,
        int j,
        for (i=0, i<n, i++)
                for (j=0, j<m, j++) if (nm[i][j] == a) goto found,
        // не найдено
        // ..
found
        // nm[i][j] == a
}
```

Имеется также инструкция ***continue***, которая передает управление в конец цикла (см. § 6.1.5).

6.4. Комментарии и отступы

Разумное использование комментариев и согласованное употребление отступов может сделать чтение и понимание программы более приятным занятием. Существует несколько стилей отступов. Я не вижу основательных причин предпочитать один из них другому (хотя, как и у большинства программистов, у меня есть свои привычки, продемонстрированные в этой книге). То же самое можно сказать и о стиле комментариев

При неправильном использовании комментариев может серьезно пострадать читабельность программы. Компилятор не понимает смысл комментариев, поэтому не существует способа проверить, что комментарий

[1] содержателен;
[2] имеет какое-то отношение к программе;
[3] не устарел.

В большинстве программ можно найти труднопонимаемые, противоречивые и просто неверные комментарии. Плохой комментарий хуже его отсутствия.

Если что-то может быть выражено *непосредственно конструкциями языка*, так оно и должно быть сделано, простого упоминания в комментарии недостаточно. Предыдущее замечание имеет в виду комментарии типа:

```
// переменную "v" надо проинициализировать
// переменная "v" должна использоваться только функцией "f()"
// перед вызовом любой другой функции из этого файла, вызовите функцию "init()"
// вызовите функцию "cleanup()" в конце вашей программы
// не пользуйтесь функцией "weird()"
// у функции "f()" два аргумента
```

При правильном использовании C++ такие комментарии часто становятся ненужными. Например, чтобы избавиться от предыдущих комментариев, можно полагаться на правила компоновки (§ 9.2) и области видимости, инициализации и очистки для классов (см. § 10.4.1).

Если что-либо ясно выражено в языке, не надо это повторять второй раз в комментарии. Например:

```
a = b + c,          // a принимает значение, равное b+c
count++,            // увеличили счетчик
```

И дело не только в том, что такие комментарии просто излишни. Они увеличивают количество текста, который надо прочесть, часто затеняют структуру программы и не редко просто ошибочны. Впрочем, обратите внимание: подобные комментарии часто встречаются в учебных пособиях типа того, что вы сейчас читаете. Это одна из многих черт, которыми реальные программы отличаются от программ из учебника.

Я предпочитаю использовать комментарии в следующих случаях:

[1] Комментарий в начале каждого файла исходного кода, где поясняются основные объявления, делаются ссылки на литературу и приводятся наиболее важные соображения по поводу сопровождения и т. п.

[2] Комментарий для каждого класса, шаблона и пространства имен.

[3] Комментарий для каждой нетривиальной функции, в котором указано ее назначение, использованный алгоритм (если он не очевиден) и, может быть, предположения, которые она делает об окружении.

[4] Комментарий для каждой глобальной переменной, переменной из пространства имен и константы.

[5] Небольшие комментарии в тех местах, где код неочевиден и/или непереносим.

[6] Очень редко в других случаях.

Например:

```
// tbl.c: реализация таблицы символов

/*
Исключение методом Гаусса.
См. Ralston: «A first course ...», стр. 411.
*/

// swap() полагает, что стек устроен как в SGI R6000.

/**********************************

Copyright (c) 1997 AT&T, Inc.
All rights reserved

**********************************/
```

Удачно подобранный и написанный набор комментариев является существенной частью хорошей программы. Написание «правильных» комментариев может оказаться не менее сложной задачей, чем написание самой программы. Стоит развивать это искусство.

Обратите внимание, что если в функции используется только //-стиль комментариев, то любой фрагмент функции может быть «закомментирован» (временно исключен) при помощи /* */ и наоборот.

6.5. Советы

[1] Отдавайте предпочтение стандартной библиотеке по отношению к другим библиотекам и коду, написанному вручную; § 6.1.8.

[2] Избегайте слишком сложных выражений; § 6.2.3.

[3] Если вы сомневаетесь в порядке выполнения операторов, пользуйтесь скобками; § 6.2.3.

[4] Избегайте явного преобразования типов (приведения); § 6.2.7.

[5] Если явное преобразование типов необходимо, отдавайте предпочтение явным операторам приведения по отношению к С-подобным преобразованиям; § 6.2.7.
[6] Пользуйтесь формой ***T*** (*e*) только тогда, когда четко известны правила конструирования значения; § 6.2.8.
[7] Избегайте выражений с неопределенной последовательностью вычислений; § 6.2 2.
[8] Старайтесь не использовать ***goto***; § 6.3.4.
[9] Старайтесь не использовать ***do-инструкцию***; § 6.3.3.
[10] Не объявляйте переменную до тех пор, пока у вас нет значения для ее инициализации; § 6.3.1, § 6.3.2.1, § 6.3.3.1.
[11] Пишите ясные и лаконичные комментарии; § 6.4.
[12] Придерживайтесь согласованного стиля в отступах; § 6.4.
[13] Лучше определить ***operator new*** () в качестве члена класса (§ 15.6), чем заменять глобальный ***operator new*** (); § 6.2.6.2.
[14] Всегда имейте в виду возможность «неправильного» ввода; § 6.1.3

6.6. Упражнения

1 (*1) Перепишите следующий пример с инструкцией ***for*** в виде эквивалентного кода с использованием инструкции ***while***:

```
for (i=0, i<max_length, i++)
    if (input_line[i] == '?') quest_count++,
```

Перепишите так, чтобы переменной цикла был указатель, то есть, чтобы проверка выглядела как-нибудь так: *p=='?'.

2. (*1) Расставьте скобки в следующих выражениях:

```
a = b + c * d << 2 & 8
a & 077 != 3
a == b || a == c && c < 5
c = x! = 0
0 <= i < 7
f (1, 2) + 3
a = -1 + + b - -5
a = b == c++
a = b = c = 0
a[4] [2] *= *b ? c * d * 2
a - b, c = d
```

3. (*2) Введите последовательность (возможно с символами-разделителями) пар (имя, значение), где имя является словом, отделяемым символами-разделителями, а значение — целым или числом с плавающей точкой. Вычислите и выведите сумму и среднее как для каждого отдельного имени, так и для всех имен (см. § 6.1.8).
4 (*1) Выведите таблицу результатов битовых логических операций (§ 6.2.4) для всех возможных комбинаций операндов 0 и 1.
5. (*1.5) Приведите 5 различных конструкций С++, смысл которых не определен (§ В.2). (*1.5) Приведите 5 различный конструкций С++, смысл которых зависит от реализации (§ В.2).

6. (*1) Приведите 10 различных примеров непереносимого кода на C++.
7. (*2) Напишите 5 выражений, в которых не определен порядок вычислений. Выполните их и посмотрите, что делает ваша реализация (по возможности попробуйте примеры на нескольких реализациях).
8. (*1.5) Что происходит при делении на 0 в вашей системе? Что происходит при переполнении сверху и снизу?
9. (*1) Расставьте скобки в следующих выражениях:

```
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

10. (*2) Напишите следующие функции: ***strlen*** (), которая возвращает длину С-строки; ***strcpy*** (), которая копирует одну строку в другую; ***strcmp*** (), которая сравнивает две С-строки. Определите, какими должны быть аргументы и возвращаемые типы. Затем сравните ваши функции со стандартными библиотечными функциями, объявленными в <*cstring*> (<*string.h*>) и описанными в § 20.4.1.
11. (*1) Посмотрите, как реагирует ваш компилятор на следующие ошибки:

```
void f(int a, int b)
{
        if (a = 3) //...
        if (a&077 == 0) //...
        a := b+1;
}
```

Придумайте несколько других простых ошибок и посмотрите, как на них реагирует компилятор.

12. (*2) Модифицируйте пример из § 6.6[3] таким образом, чтобы он вычислял и медиану.
13. (*2) Напишите функцию ***cat*** (), принимающую в качестве аргументов две С-строки, которая возвращает строку, являющуюся их конкатенацией. Воспользуйтесь оператором ***new*** для выделения памяти под результат.
14. (*2) Напишите функцию ***rev*** (), берущую в качестве аргумента символьную С-строку и переставляющую в ней символы в обратном порядке. То есть, после вызова ***rev*** (***p***) последний символ ***p*** становится первым и т. д.
15. (1.5) Что делается в следующем примере?

```
void send(int* to, int* from, int count)
// Черный ящик. Полезные комментарии умышленно удалены.
{
        int n = (count+7) / 8;
        switch (count%8) {
        case 0:     do {  *to++ = *from++;
        case 7:           *to++ = *from++;
        case 6:           *to++ = *from++;
        case 5:           *to++ = *from++;
        case 4:           *to++ = *from++;
```

```
        case 3:         *to++ = *from++;
        case 2:         *to++ = *from++;
        case 1          *to++ = *from++;
            } while (--n>0);
        }
    }
```

Зачем кому-нибудь может понадобиться подобный код?

16. (*2) Напишите функцию ***atoi*** (***const char****), которая получает С-строку, состоящую из цифр, и возвращает соответствующее целое значение. Например, результатом вызова ***atoi*** (***"123"***) должно быть ***123***. Модифицируйте ***atoi*** () таким образом, чтобы она могла работать с восьмеричными и шестнадцатеричными представлениями в дополнение к обычным десятичным числам. Модифицируйте ***atoi*** () таким образом, чтобы она могла работать с символьными константами C++.
17. (*2) Напишите функцию ***itoa*** (***int i, char b***[]), которая создает строковое представление ***i*** в ***b*** и возвращает ***b***.
18. (*2) Введите программу калькулятора и заставьте ее работать. Не пользуйтесь командами редактора «скопировать и вставить» при вводе повторяющихся фрагментов кода. Вы научитесь большему, отыскивая и исправляя разные «мелкие глупые ошибки».
19. (*2) Модифицируйте калькулятор таким образом, чтобы он выводил номер строки с ошибкой.
20. (*3) Предоставьте пользователю возможность определять функции в калькуляторе. Подсказка: определите функцию в виде последовательности операций, в точности такой, какую ввел пользователь. Последовательность можно хранить либо в виде строки, либо как список лексем. При вызове функции прочитайте и выполните эти операции. Если вы захотите, чтобы пользовательские функции принимали аргументы, вам придется придумать для этого соответствующую форму записи.
21. (*1.5) Модифицируйте настольный калькулятор таким образом, чтобы в нем использовалась структура ***symbol*** вместо статических переменных ***number_value*** и ***string_value***.
22. (*1.5) Напишите программу, которая удаляет комментарии из программы на C++. Пусть она читает из ***cin***, удаляет комментарии обоих видов (// и /* */) и записывает результат в ***cout***. Не заботьтесь о внешнем виде вывода (это было бы другим и гораздо более сложным упражнением). Программа должна работать одинаково и для корректных, и для некорректных программ на ее входе. Не забудьте о возможности наличия символов // , /* и */ в комментариях, строках и символьных константах.
23. (*2) Просмотрите несколько программ, обращая внимание на различные стили отступов, задания имен и оформления комментариев.

7

Функции

Итерация — от человека,
рекурсия — от Бога.
— Л. Питер Дойч

Объявления и определения функций — передача аргументов — возвращаемые значения — перегрузка функций — разрешение неоднозначности — аргументы по умолчанию — ***stdargs*** — указатели на функции — макросы — советы — упражнения.

7.1. Объявления функций

Чтобы выполнить что-нибудь полезное в C++, как правило, вызывают функцию. Определить функцию — значит задать способ выполнения операции. К функции нельзя обратиться, если она не была предварительно объявлена.

Объявление функции задает имя функции, тип возвращаемого значения (если оно есть) и количество и типы аргументов, которые должны присутствовать при вызове функции. Например:

```
Elem* next_elem ();
char* strcpy (char* to, const char* from);
void exit (int);
```

Семантика передачи аргументов аналогична инициализации. Производится проверка типов аргументов и, если необходимо, осуществляется неявное преобразование типов. Например:

```
double sqrt (double);

double sr2 = sqrt (2);      // вызов sqrt() с аргументом double(2)
double sr3 = sqrt ("mpu");  // ошибка: sqrt() требует аргумент типа double
```

Не следует недооценивать значение проверки соответствия и преобразования типов.

Объявление функции может содержать имена аргументов. Это может оказаться полезным для читающего программу, но компилятор их просто игнорирует. Как упоминалось в § 4.7, указание ***void*** в качестве типа возвращаемого значения означает, что функция не возвращает значения.

7.1.1. Определения функций

Каждая функция, вызываемая в программе, должна быть где-нибудь определена (и только один раз). Определение функции является объявлением функции, в котором присутствует тело функции. Например:

```
extern void swap(int*, int*);        // объявление
void swap(int* p, int* q)            // определение
{
        int t = *p;
        *p = *q;
        *q = t;
}
```

Типы в определении и объявлениях функции должны совпадать. Однако, имена аргументов не являются частью типа и не обязаны совпадать.

Бывают функции, имеющие неиспользуемый аргумент:

```
void search(table* t, const char* key, const char*)
{
        // третий аргумент не используется
}
```

Как показано в примере, тот факт, что аргумент не используется, можно отразить, не указывая для него имени. Обычно, неименованные аргументы появляются либо в результате упрощения кода, либо когда планируется расширение возможностей функции. В обоих случаях наличие аргумента, пусть и не используемого, обеспечивает независимость вызывающих программ от этих изменений.

Функцию можно определить со спецификатором ***inline***. Такие функции называются *встроенными*. Например:

```
inline int fac(int n) { return (n < 2) ? 1  n*fac(n-1); }
```

Спецификатор ***inline*** указывает компилятору, что он должен пытаться каждый раз генерировать в месте вызова код, соответствующий функции ***fac***() (факториал), а не создавать отдельно код функции (единожды) и затем вызывать ее посредством обычного механизма вызова. Хороший компилятор может сгенерировать константу ***720*** в месте вызова ***fac***(***6***). Ввиду допустимости рекурсивных и взаимно рекурсивных встроенных функций, невозможно гарантировать, что в месте каждого вызова такой функции будет действительно осуществлено встраивание кода функции. В зависимости от качества, один компилятор может сгенерировать ***720***, другой — ***6*fac***(***5***), а третий — обычный вызов ***fac***(***6***).

Для того чтобы сделать возможным встраивание при отсутствии «сверхумного» компилятора и изощренных средств компоновки, определение (а не только объявление) встроенной функции должно находится в данной области видимости (§ 9.2). Спецификатор ***inline*** не оказывает влияния на смысл вызова функции. В частности, встроенная функция все равно имеет уникальный адрес, так же как ее статические переменные (§ 7.1.2).

7.1.2. Статические переменные

Локальная переменная инициализируется в момент выполнения строки, содержащей ее определение. По умолчанию это происходит при каждом вызове функции и каждый

раз создается новая переменная. Если локальная переменная объявлена как ***static***, при всех вызовах функции для хранения ее значения будет использоваться единственный, статически размещенный в памяти объект (§ В.9). Она будет проинициализирована только при первом выполнении строки, содержащей ее определение. Например:

```
void f(int a)
{
    while (a--) {
        static int n = 0;          // инициализируется один раз
        int x = 0;                 // инициализируется 'a' раз при каждом вызове f()
        cout << "n== " << n++ << ", x==" << x++ '\n';
    }
}
int main ()
{
    f(3),
}
```

В результате будет напечатано:

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

Статическая переменная позволяет функции «помнить о прошлом», не создавая при этом глобальной переменной, к которой могли бы обратиться (и испортить ее) другие функции (см. также § 10.2.4).

7.2. Передача аргументов

При вызове функции выделяется память под ее формальные аргументы, и каждому формальному аргументу присваивается значение соответствующего фактического аргумента. Семантика передачи аргументов идентична семантике инициализации. В частности, проверяется соответствие типов формальных и фактических аргументов и при необходимости выполняются либо стандартные, либо определенные пользователем преобразования типов. Существуют специальные правила для передачи массивов в качестве аргумента (§ 7.2.1), средства для передачи аргументов, соответствие типов для которых не проверяется (§ 7.6), и средства для задания аргументов по умолчанию (§ 7.5). Рассмотрим пример:

```
void f(int val, int& ref)
{
    val++;
    ref++;
}
```

При вызове *f*() ***val***++ увеличивает значение локальной копии первого фактического аргумента, в то время как ***ref***++ увеличивает значение второго фактического аргумента. Например,

```
void g()
{
```

```
    int i = 1;
    int j = 1;
    f(i, j);
}
```

увеличит *j*, но не *i*. Первый аргумент, *i*, передан *по значению*, а второй, *j* — *по ссылке*. В § 5.5 упоминалось, что функции, которые модифицируют аргументы, переданные по ссылке, делают программу трудно читаемой и в большинстве случаев их следует избегать (но см. § 21.3.2). Однако, большие объекты значительно эффективней передавать по ссылке, чем по значению. В этом случае аргумент можно объявить с модификатором ***const***, чтобы указать, что ссылка используется только из соображений эффективности, а не для того, чтобы функция изменила значение объекта:

```
void f(const Large& arg)
{
    // значение arg нельзя изменить без использования
    // явного преобразования типа
}
```

Отсутствие ***const*** в объявлении аргумента, передаваемого по ссылке, рассматривается как указание на то, что переменная модифицируется в функции:

```
void g(Large& arg);    // предполагаем, что g() модифицирует arg
```

Аналогично, объявление аргумента, являющегося указателем, с модификатором ***const***, говорит читателю, что значение объекта, на который ссылается указатель, не изменится в функции. Например:

```
int strlen(const char*);                    // количество символов в строке
char* strcpy(char* to, const char* from);   // копирование строк
int strcmp(const char*, const char*);       // сравнение строк
```

Использование константных аргументов становится все более важным по мере роста размера программы.

Обратите внимание, что семантика передачи аргументов отличается от семантики присваивания. Это важно для константных аргументов, аргументов, передаваемых по ссылке, и аргументов некоторых типов, определяемых пользователем (§ 10.4.4.1).

Литералы, константы и аргументы, требующие преобразования типа, можно передавать как ***const***&-аргументы и нельзя — в качестве не ***const***& аргументов. Разрешение преобразования для аргумента типа ***const T***& обеспечивают гарантию того, что аргументу будет присвоен тот же набор значений, что и аргументу типа ***T***; при необходимости значение будет передано через временную переменную. Например:

```
float fsqrt(const float&);    // sqrt в стиле Fortran с передачей аргумента по ссылке

void g(double d)
{
    float r = fsqrt(2.0f);    // передает ссылку на временную переменную,
                              // содержащую 2.0f
    r = fsqrt(r);             // передает ссылку на r
    r = fsqrt(d);             // передает ссылку на временную переменную,
                              // содержащую float(d)
}
```

Запрещение преобразования неконстантных аргументов, передаваемых по ссылке (§ 5.5), позволяет избежать глупых ошибок, возникающих из-за создания временных переменных. Например:

```
void update (float& i);

void g (double d, float r)
{
        update (2.0f);      // ошибка: константный аргумент
        update (r);         // передает ссылку на r
        update (d);         // ошибка: требуется преобразование типов
}
```

Если бы такие вызовы были разрешены, ***update*** () спокойно изменила бы временные переменные, которые тут же были бы удалены. Скорее всего, это привело бы к неприятным сюрпризам для программиста.

7.2.1. Массивы в качестве аргументов

Если в качестве аргумента функции используется массив, передается указатель на его первый элемент. Например:

```
int strlen (const char*);

void f ()
{
        char v[] = "массив";
        int i = strlen (v);
        int j = strlen ("Nicholas");
}
```

То есть при вызове функции аргумент типа ***T***[] будет преобразован в ***T****. Из этого следует, что присваивание элементу массива, являющегося аргументом, изменяет значение элемента самого массива. Другими словами, массивы отличаются от других типов тем, что их нельзя передать по значению. Размер массива неизвестен в вызываемой функции. Это может составлять неудобство, но существует несколько способов решения данной проблемы. С-строки ограничены нулем, и поэтому их размер легко вычислить. Для других массивов можно передавать второй аргумент, означающий размер массива. Например:

```
void compute1 (int* vec_ptr, int vec_size);      // один способ

struct Vec {
        int* ptr;
        int size;
};

void compute2 (const Vec& v);                    // другой способ
```

В качестве альтернативы вместо массивов можно воспользоваться такими типами, как ***vector*** (§ 3.7.1, § 16.3).

С многомерными массивами проблем больше (см. § B.7), но часто вместо них можно использовать массивы указателей; в этом случае не требуется никаких специальных трюков. Например:

```
char* day[] = {
        "понедельник", "вторник", "среда", "четверг",
        "пятница", "суббота", "воскресенье"
};
```

Подчеркну еще раз, что ***vector*** и подобные типы являются альтернативой встроенным массивам низкого уровня и указателям.

7.3. Возвращаемое значение

Функция должна возвращать значение, если она не объявлена как ***void*** (при том, что функция ***main*** (), является исключением, см. § 3.2). И наоборот — значение не может быть возвращено из функции, если она объявлена как ***void***. Например:

```
int f1 () { }           // ошибка: не возвращается значение
void f2 () { }          // правильно

int f3 () { return 1; } // правильно
void f4 () { return 1; } // ошибка: возвращаемое значение в функции объявленной, как void

int f5 () { return; }   // ошибка: не указано возвращаемое значение
void f6 () { return; }  // правильно
```

Возвращаемое значение задается инструкцией ***return***. Например:

```
int fac (int n) { return (n>1) ? n*fac (n-1) : 1; }
```

Функции, которые вызывают сами себя, называются *рекурсивными*.

В функции может быть более одной инструкции ***return***:

```
int fac2 (int n)
{
        if (n > 1) return n*fac2 (n-1);
        return 1;
}
```

Как и передача аргументов, семантика возврата значения из функции идентична семантике инициализации. Инструкцию ***return*** можно рассматривать как инициализацию неименованной переменной возвращаемого типа. Осуществляется сравнение типа выражения в инструкции ***return*** с типом возвращаемого значения и при необходимости выполняются стандартные, либо определяемые пользователем преобразования. Например:

```
double f ()
{
        // ...
        return 1;       // 1 неявно преобразуется в double(1)
}
```

При каждом вызове функции создаются новые копии аргументов и локальных (автоматических) переменных. Память может повторно использоваться после возврата из функции, поэтому, указатель на локальную переменную возвращать не следует. Значение, на которое он указывает, будет меняться непредсказуемо:

```
int* fp ()
{
```

```
    int local = 1;
    // ...
    return &local;    // плохо
}
```

Такая ошибка встречается реже, чем эквивалентная ошибка с использованием ссылок:

```
int& fr()
{
    int local = 1;
    // ..
    return local,     // плохо
}
```

К счастью, компилятор может предупредить о том, что возвращается ссылка на локальную переменную.

Функция ***void*** не может возвращать значение. Однако вызов функции ***void*** не дает значения, так что функция ***void*** может использовать вызов функции ***void*** как выражение в инструкции ***return***. Например:

```
void g(int* p);

void h(int* p) { /* ..*/ return g(p); }  // правильно: возвращает «никакое значение»
```

Такая форма инструкции ***return*** важна при написании шаблонов функций, когда тип возвращаемого значения является параметром шаблона (§ 18.4.4.2).

7.4. Перегруженные имена функций

Как правило, разным функциям дают различные имена, но когда функции выполняют концептуально аналогичные задачи для объектов различных типов, может оказаться удобным присвоить им одно и то же имя. Использование одного имени для операции, выполняемой с различными типами, называется *перегрузкой*. Такая техника уже используется для базовых операций C++. Например, существует только одно имя для сложения +, но его можно использовать для сложения целых, чисел с плавающей точкой и для инкремента указателей. Эту идею легко распространить на функции, определяемые пользователем. Например:

```
void print(int);              // печать целого
void print(const char*),      // печать символьной С-строки
```

С точки зрения компилятора, единственное, что функции имеют общего между собой — это имена. Предположительно, такие функции в некотором смысле похожи друг на друга, но сам язык не накладывает в связи с этим никаких дополнительных ограничений и никак не помогает программисту. Таким образом, перегруженные функции предназначены в первую очередь для удобства записи. Это имеет особое значение для распространенных имен, таких как ***sqrt***, ***print*** и ***open***. Если само имя значимо с точки зрения семантики, преимущества перегрузки возрастают. Так происходит, например, для операторов +, * и <<, конструкторов (§ 11.7) и при обобщенном программировании (§ 2.7.2, глава 18). Когда вызывается функция *f*, компилятор должен определить, какую из функций с именем *f* использовать. Идея состоит в том,

чтобы использовать функцию с наиболее подходящими аргументами и выдать сообщение об ошибке, если таковой не найдено. Например:

```
void print(double);
void print(long);

void f()
{
        print(1L);      // print(long)
        print(1 0)      // print(double)
        print(1)        // ошибка: двусмысленно — print(long(1))
                        // или print(double(1))?
}
```

Процесс поиска подходящей функции из множества перегруженных заключается в нахождении наилучшего соответствия типов формальных и фактических аргументов. Вышесказанное осуществляется путем проверки набора критериев в следующем порядке:

[1] Точное соответствие типов; то есть полное соответствие или соответствие, достигаемое тривиальными преобразованиями типов (например, имя массива и указатель, имя функции и указатель на функцию, тип ***T*** и ***const T***).

[2] Соответствие, достигаемое «продвижением» («повышением в чине») интегральных типов (например, ***bool*** в ***int***, ***char*** в ***int***, ***short*** в ***int*** и ***unsigned*** аналоги; § B.6.1) и ***float*** в ***double***.

[3] Соответствие, достигаемое путем стандартных преобразований (например, ***int*** в ***double***, ***double*** в ***int*** и ***double*** в ***long double***, указателей на производные типы в указатели на базовые (§ 12.2), указателей на произвольные типы в ***void**** (§ 5.6), ***int*** в ***unsigned int***; § B.6).

[4] Соответствие, достигаемое при помощи преобразований, определяемых пользователем (§ 11.4).

[5] Соответствие за счет многоточий (...) в объявлении функции (§ 7.6).

Если соответствие может быть получено двумя способами на одном и том же уровне критериев, вызов считается неоднозначным и отвергается. Такой подход при разрешении[1] перегрузки отражает правила C и C++ для встроенных числовых типов (§ B.6). Например:

```
void print(int);
void print(const char*);
void print(double),
void print(long),
void print(char);

void h(char c, int i, short s, float f)
{
        print(c),       // точное соответствие; вызывается print(char)
        print(i),       // точное соответствие; вызывается print(int)
        print(s);       // интегральное «продвижение»; вызывается print(int)
        print(f),       // «продвижение» float в double; вызывается print(double)
```

[1] Здесь и далее, когда речь идет о вызовах перегруженных функций, слово «разрешение» (resolution) означает не «допущение» или «позволение», а «анализ с целью выяснения, какую функцию выбрать». — *Примеч. ред.*

```
    print('a');        // точное соответствие; вызывается print(char)
    print(49);         // точное соответствие; вызывается print(int)
    print(0);          // точное соответствие; вызывается print(int)
    print("a");        // точное соответствие; вызывается print(const char*)
}
```

Вызов *print(0)* обращается к *print(int)*, потому что *0* имеет тип *int*. Функция *print('a')* вызывает *print(char)*, потому что *'a'* имеет тип *char* (§ 4.3.1). Смысл разделения на преобразования и «продвижения» состоит в том, чтобы отличить безопасные «продвижения», такие как *char* в *int*, и потенциально опасные преобразования, такие как *int* в *char*.

Результат разрешения перегрузки не зависит от порядка объявления функций.

Механизм перегрузки основывается на относительно сложном наборе правил и в некоторых случаях программисту будет неочевидно, какая функция вызовется. Возникает вопрос: «Зачем все это?». Рассмотрим альтернативу перегрузке. Нам часто требуются сходные операции над объектами разных типов. Без перегрузки мы должны определить несколько функций с различными именами:

```
void print_int(int);
void print_char(char);
void print_string(const char*);

void g(int i, char c, const char* p, double d)
{
    print_int(i);          // правильно
    print_char(c);         // правильно
    print_string(p);       // правильно

    print_int(c);          // правильно? Вызывает print_int(int(c))
    print_char(i);         // правильно? Вызывает print_char(char(i))
    print_string(i);       // ошибка
    print_int(d);          // правильно? Вызывает print_int(int(d))
}
```

По сравнению с перегруженной *print()*, нам приходится помнить несколько имен и то, как их правильно использовать. Это довольно неприятно, препятствует использованию метода обобщенного программирования (§ 2.7.2) и, в общем случае, стимулирует программиста обращать неоправданно большое внимание на проблемы относительно низкого уровня. При отсутствии перегрузки к аргументам функций применяются все стандартные преобразования. Это также может привести к дополнительным ошибкам. В предыдущем примере только в одном из четырех примеров с «неправильными» аргументами ошибка обнаружена компилятором. Механизм перегрузки может увеличить шансы, что неправильный аргумент будет обнаружен компилятором.

7.4.1. Перегрузка и возвращаемые типы

Возвращаемые типы не участвуют в разрешении перегрузки. Цель состоит в том, чтобы обеспечить контекстную независимость разрешения перегрузки для операторов (§ 11.2.1, § 11.2.4) и вызовов функций. Рассмотрим пример:

```
float sqrt(float);
double sqrt(double);
```

```
void f(double da, float fla)
{
    float fl = sqrt(da);      // вызов sqrt(double)
    double d = sqrt(da);      // вызов sqrt(double)
    fl = sqrt(fla);           // вызов sqrt(float)
    d = sqrt(fla);            // вызов sqrt(float)
}
```

Если бы при разрешении использовался возвращаемый тип, было бы невозможно только по вызову ***sqrt*** () определить, которую функцию в действительности вызывать.

7.4.2. Перегрузка и область видимости

Функции, объявленные в различных областях видимости (не пространствах имен), не являются перегруженными. Например:

```
void f(int);

void g()
{
    void f(double);
    f(1);          // вызывается f(double)
}
```

Ясно, что *f*(***int***) была бы идеальным соответствием для *f*(*1*), но в данной области видимости находится только *f*(***double***). В подобных случаях можно вводить и удалять локальные объявления для получения желаемого поведения. Как и всегда, намеренное сокрытие может быть полезным, но случайное сокрытие является источником неожиданностей. Если нужно, чтобы область действия перегрузки пересекала области видимости классов (§ 15.2.2) или пространств имен (§ 8.2.9.2), можно воспользоваться директивой ***using*** (§ 8.2.2). См. также § 8.2.6.

7.4.3. Явное разрешение неоднозначности

Объявление небольшого (или слишком большого) количества перегруженных вариантов функции может привести к неоднозначности. Например:

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i);
{
    f1(i);         // неоднозначность: f1(char) или f1(long)
    f2(0);         // неоднозначность: f2(char*) или f2(int)
}
```

Там где это возможно, в подобных случаях следует рассматривать набор перегруженных функций как единое целое и проверять, имеет ли аргумент смысл с точки зрения семантики функции. Часто проблему можно решить добавлением функции, которая разрешает неоднозначность. Например, добавление

```
inline void f1 (int n) { f1 (long (n)); }
```

разрешило бы все двусмысленности, подобные *f1 (i)*, в пользу большего типа ***long int***.

Также можно воспользоваться явным преобразованием типа в конкретном вызове. Например:

```
f2 (static_cast<int> (0));
```

Однако, часто это не более, чем не слишком красивая временная мера. Как только встретится следующий подобный вызов, снова нужно будет решать ту же самую проблему.

Некоторых новичков в C++ раздражают сообщения о неоднозначности, выдаваемые компилятором. Более опытные программисты ценят эти сообщения, рассматривая их как полезные указания на ошибки проектирования.

7.4.4. Разрешение в случае нескольких аргументов

При имеющихся правилах разрешения перегрузки можно быть уверенным, что будут использованы самые простые алгоритмы (функции), когда эффективность или точность в значительной степени зависят от типов аргументов. Например:

```
int pow (int, int);
double pow (double, double);

complex pow (double, complex);
complex pow (complex, int);
complex pow (complex, double);
complex pow (complex, complex);

void k (complex z)
{
        int i = pow (2, 2);             // вызов pow (int, int)
        double d = pow (2.0, 2.0);      // вызов pow (double, double)
        complex z2 = pow (2, z);        // вызов pow (double, complex)
        complex z3 = pow (z, 2);        // вызов pow (complex, int)
        complex z4 = pow (z, z);        // вызов pow (complex, complex)
}
```

В процессе выбора среди перегруженных функций, имеющих несколько аргументов, наилучшее соответствие находится для каждого аргумента в соответствии с правилами из § 7.4. Вызывается функция, у которой наилучшим образом соответствует один аргумент и лучшим либо таким же образом соответствуют остальные аргументы. Если такой функции не найдено, вызов считается неоднозначным. Например:

```
void g ()
{
        double d = pow (2.0, 2);        // ошибка: pow (int (2.0), 2)
                                        // или pow (2.0, double (2))?
}
```

Вызов неоднозначен, потому что ***2.0*** наилучшим образом подходит для первого аргумента ***pow (double, double)***, а ***2*** наилучшим образом подходит для второго аргумента ***pow (int, int)***.

7.5. Аргументы по умолчанию

У функций общего назначения часто больше аргументов, чем требуется в простых случаях. В частности, функции, создающие объекты (§ 10.2.3), для увеличения гибкости часто предоставляют несколько возможностей. Рассмотрим функцию, которая печатает целое. Кажется разумным предоставить пользователю возможность выбора основания, по которому его печатать, но в большинстве программ целые будут печататься в виде десятичных чисел. Например:

```
void print(int value, int base = 10);        // основание по умолчанию равно 10

void f()
{
        print(31);
        print(31, 10);
        print(31, 16),
        print(31, 2);
}
```

В результате на выходе получится:

```
31   31   1f   11111
```

Того же самого результата можно было добиться при помощи перегруженных функций:

```
void print(int value, int base);
inline void print(int value) { print(value, 10); }
```

Однако, перегрузка делает для программиста менее очевидным тот факт, что цель была — получить одну функцию плюс некоторую короткую форму ее записи.

Тип аргумента по умолчанию проверяется в месте объявления функции и вычисляется в момент вызова. Аргументы по умолчанию можно задавать только в конце списка аргументов. Например:

```
int f(int, int=0, char* =0);        // правильно
int g(int=0, int=0, char*);         // ошибка
int h(int=0, int, char* =0);        // ошибка
```

Обратите внимание, что пробел между * и = имеет значение (*= — оператор присваивания; § 6.2):

```
int nasty(char*=0);          // синтаксическая ошибка
```

Аргумент по умолчанию не может быть повторен или изменен в последующих объявлениях в той же области видимости. Например:

```
void f(int x = 7);
void f(int = 7);             // ошибка: нельзя повторять аргумент по умолчанию
void f(int = 8);             // ошибка: другое значение аргумента по умолчанию

void g()
{
        void f(int x = 9);   // правильно: это объявление скрывает предыдущее
        //...
}
```

Объявление имени во вложенной области видимости , скрывающее объявление того же имени во внешней области, скорее всего приведет к ошибке.

7.6. Неуказанное количество аргументов

Для некоторых функций невозможно указать количество и типы всех аргументов. Список аргументов в объявлениях таких функций заканчивается многоточием (...), что означает «и может быть еще несколько аргументов». Например:

```
int printf(const char* ...);
```

Это означает, что вызов функции стандартной библиотеки C ***printf***() (§ 21.8) должен содержать по крайней мере один аргумент типа ***char****** и может либо содержать, либо не содержать еще несколько аргументов. Например:

```
printf("Здравствуй, мир!\n");
printf("Меня зовут %s %s\n", first_name, second_name);
printf("%d + %d = %d\n", 2, 3, 5);
```

Во время интерпретации списка аргументов такая функция пользуется информацией, недоступной компилятору. В случае с ***printf***(), первым аргументом является строка, содержащая специальную последовательность символов, которая позволяет ***printf***() правильно обрабатывать последующие аргументы: ***%s*** означает «ожидаю аргумент типа ***char****», а ***%d*** означает «ожидаю аргумент типа ***int***». Однако, в общем случае компилятор не может все это проверить, поэтому он не в состоянии гарантировать, что ожидаемые аргументы действительно присутствуют или что они имеют правильные типы. Например, программа

```
#include <stdio.h>

int main()
{
        printf("Меня зовут %s %s\n", 2);
}
```

будет успешно откомпилирована и, в лучшем случае, распечатает нечто странное (посмотрите, что получится!).

Ясно, что если аргумент не был объявлен, компилятор не имеет информации, необходимой для выполнения стандартной проверки и преобразований типа. В таких случаях ***char*** и ***short*** передаются как ***int***, а ***float*** — как ***double***. Совсем необязательно, что программист ожидает именно этого.

В качественно разработанной программе требуется максимум всего лишь несколько функций, у которых указаны типы не всех аргументов . В большинстве случаев вместо функции с неуказанным количеством аргументов можно воспользоваться перегруженными функциями или функциями с аргументами по умолчанию для того, чтобы обеспечить проверку соответствия типов. Только в тех случаях, когда и количество и типы аргументов неизвестны, следует пользоваться многоточием. Типичным применением многоточий является создание интерфейса для библиотечных функций C, которые были определены до того, как в C++ появились их альтернативы:

```
int fprintf(FILE*, const char* ...);      // из <cstdio>
int execl(const char* ...);               // из заголовочного файла UNIX
```

В ***<cstdarg>*** можно найти стандартный набор макросов для доступа к неспецифированным аргументам в таких функциях. Рассмотрим пример функции, у которой один целый аргумент, означающий серьезность ошибки, за которым следует произвольное

количество строк. Идея состоит в том, что сообщение об ошибке будет составляться из отдельных слов, переданных в качестве строковых аргументов. Список строковых аргументов должен заканчиваться нулевым указателем на ***char***:

```
extern void error (int ...);
extern char* itoa (int, char[]);          // см. § 6.6.17

const char* Null_cp = 0;

int main (int argc, char* argv[])
{
        switch (argc) {
        case 1:
                error (0, argv[0], Null_cp);
                break;
        case 2:
                error (0, argv[0], argv[1], Null_cp);
                break;
        default:
                char buffer[8];
                error (1, argv[0], "с", itoa (argc-1, buffer), "аргументами", Null_cp);
        }
        //...
}
```

Функция ***itoa*** () возвращает символьную строку, соответствующую целому аргументу.

Обратите внимание, что использование ***0*** в качестве ограничителя было бы не переносимо. В некоторых реализациях целый ноль и нулевой указатель имеют различные представления. Это является иллюстрацией нюансов и дополнительных сложностей, с которыми приходится иметь дело программисту, когда проверка типов отсутствует из-за использования многоточия.

Функцию вывода сообщения об ошибке можно определить следующим образом:

```
void error (int severity ...)          // «серьезность» ошибки, за которой
                                       // следует ограниченная нулевым
                                       // указателем последовательность char*
{
        va_list ap;
        va_start (ap, severity);       // подготовка аргументов

        for (;;) {
                char* p = va_arg (ap, char*);
                if (p == 0) break;
                cerr << p << ' ';
        }

        va_end (ap);                   // очистка аргументов

        cerr << '\n';
        if (severity) exit (severity);
}
```

Сначала определен и инициализирован ***va_list*** при помощи вызова ***va_start*** (). Макрос ***va_start*** использует в качестве аргументов имя переменной типа ***va_list*** и имя

последнего формального аргумента. Макрос ***va_arg*** () используется для последовательного извлечения неименованных аргументов. При каждом вызове программист должен указать тип; ***va_arg*** () полагает, что был передан правильный аргумент данного типа, но, как правило, не существует способа проверить это. До выхода из функции, где была использована ***va_start*** (), необходимо осуществить вызов ***va_end*** (). Причина состоит в том, что ***va_start*** () может модифицировать стек таким образом, что станет невозможен нормальный выход из функции. Все эти модификации устраняются вызовом ***va_end*** ().

7.7. Указатель на функцию

С функцией можно проделать только две вещи: вызвать ее и получить ее адрес. Указатель, полученный взятием адреса функции, можно затем использовать для вызова функции. Например:

```
void error (string s) { /* ... */ }

void (*efct) (string);          // указатель на функцию

void f ()
{
    efct = &error;              // efct указывает на функцию error
    efct ("ошибка");            // вызов error через efct
}
```

Компилятор распознает, что ***efct*** является указателем и вызовет функцию, на которую он указывает. То есть, разыменование указателя на функцию при помощи оператора * является необязательным. Аналогично, необязательно пользоваться & для получения адреса функции:

```
void (*f1) (string) = &error;   // правильно
void (*f2) (string) = error;    // тоже правильно — означает то же самое,
                                // что и &error

void g ()
{
    f1 ("Vasa"),                // правильно
    (*f1) ("Mary Rose");        // тоже правильно
}
```

Аргументы указателей на функцию объявляются точно так же, как и аргументы самих функций. При присваивании типы функций должны в точности совпадать. Например:

```
void (*pf) (string)             // указатель на void(string)
void f1 (string);               // void(string)
int f2 (string);                // int(string)
void f3 (int*),                 // void(int*)

void f ()
{
    pf = &f1;                   // правильно
    pf = &f2;                   // ошибка: не тот возвращаемый тип
    pf = &f3;                   // ошибка: не тот тип аргумента
```

```
    pf("Гера");          // правильно
    pf(1);               // ошибка: не тот тип аргумента
    int i = pf("Зевс");  // ошибка: присваивание void переменной int
}
```

Правила передачи аргументов при вызове функций через указателей те же самые, что и при непосредственном вызове функции.

Часто принято определять имена для типов указателей на функции, чтобы избежать постоянного употребления неочевидного синтаксиса их объявлений. Вот примеры из заголовочного файла на некой системе UNIX:

```
typedef void (*SIG_TYP)(int);          // из <signal.h>
typedef void (*SIG_ARG_TYP)(int);
SIG_TYP signal(int, SIG_ARG_TYP);
```

Часто бывает полезен массив указателей на функции. Например, система меню в моем редакторе, управляемом мышью, реализована через массивы указателей на функции, которые выполняют соответствующие операции. Здесь невозможно описать всю систему в деталях. Охарактеризуем общую идею:

```
typedef void (*PF)();

PF edit_ops[] = {          // команды редактирования
    &cut, &paste, &copy, &search
};

PF file_ops[] = {          // файловые операции
    &open, &append, &close, &write
};
```

Теперь мы можем определить и проинициализировать указатели для действий, выбираемых из меню и связанных с кнопками мыши:

```
PF* button2 = edit_ops;
PF* button3 = file_ops;
```

В законченном приложении для определения каждого пункта меню требуется больше информации. Например, где-то надо хранить строку с текстом, который будет выводиться. В процессе использования системы назначение кнопок мыши часто меняется в зависимости от контекста. Эти изменения (частично) реализуются путем корректировки указателей, связанных с кнопками. Когда пользователь выбирает пункт меню, например пункт 3, с нажатой кнопкой 2, выполняется соответствующая операция:

```
button2[2]();      // вызов третьей функции button2
```

Одним из способов оценки выразительной мощи указателей на функции является попытка написать подобный код без них — и без использования их близких (более цивилизованных) родственников — виртуальных функций (§ 12.2.6). Меню можно модифицировать во время выполнения, добавляя новые функции в таблицу операторов. Также легко создавать новые меню во время исполнения.

Можно использовать указатели на функции для реализации простой формы полиморфных процедур, то есть процедур, которые можно вызывать с объектами нескольких различных типов:

```
typedef int (*CFT) (const void*, const void*);

void ssort (void* base, size_t n, size_t sz, CFT cmp)
/*
Сортировка n элементов вектора base в порядке возрастания с использованием функции сравнения, на которую указывает cmp.
Размер элементов равен sz.

Shell-сортировка (Knuth, том 3, стр. 84)
*/
{
    for (int gap=n/2; 0<gap; gap /= 2)
        for (int i=gap; i<n, i++)
            for (int j=i-gap, 0<=j, j-= gap) {
                // обязательное приведение
                char* b = static_cast<char*> (base);
                char* pj = b + j * sz;          // &base[j]
                char* pjf = b + (j + gap) * sz; // &base[j+gap]

                if (cmp(pjg, pj) < 0) {         // поменять местами base[j]
                                                // и base[j+gap]:
                    for (int k=0; k<sz; k++) {
                        char temp = pj[k];
                        pj[k] = pjg[k];
                        pjg[k] = temp;
                    }
                }
            }
}
```

Процедура *ssort* () не знает типы объектов, которые она сортирует; ей известны только количество элементов (размер вектора), размер каждого элемента и функция, которая производит сравнение. Тип *ssort* () выбран таким же, как тип стандартной функции библиотеки С *qsort* (). В реальных программах используется *qsort* (), стандартный алгоритм библиотеки С++ *sort* (§ 18.7.1) или специализированные процедуры сортировки. Такой стиль кода является обычным при программировании на С, но не может быть назван самым элегантным способом изложения алгоритма на С++ (см. § 13.3, § 13.5.2).

Нашу функцию можно использовать для сортировки таблицы следующим образом:

```
struct User {
    char* name,
    char* id;
    int dept;
}

User heads[] = {
    "Ritchie D. M.",    "dmr",    11271,
    "Sethi R.",         "ravi",   11272,
    "Szymanski T. G.",  "tgs",    11273,
    "Schryer N. L.",    "nls",    11274,
    "Schryer N. L.",    "nls",    11275,
    "Kernighan B. W.",  "bwk",    11276
};
```

```
void print_id (User* v, int n)
{
    for (int i=0; i<n; i++)
        cout << v[i].name << '\t' << v[i].id << '\t' << v[i].dept << '\n';
```

Для того чтобы осуществить сортировку, мы сначала должны определить функцию сравнения. Функция сравнения должна возвращать отрицательное значение, если ее первый аргумент меньше второго, ноль, если аргументы равны, и положительное значение в противном случае:

```
int cmp1 (const void* p, const void* q)          // сравнение строковых имен
{
    return strcmp (static_cast<const User*> (p)->name, static_cast<const User*> (q)->name);
}

int cmp2 (const void* p, const void* q)          // сравнение номеров (отдела)
{
    return static_cast<const User*> (p)->dept - static_cast<const User*> (q)->dept;
}
```

Следующая программа сортирует и выводит:

```
int main ()
{
    cout << "Начальники отделов в алфавитном порядке:\n";
    ssort (heads, 6, sizeof (User), cmp1);
    print_id (heads, 6);
    cout << '\n';

    cout << "Начальники отделов по номерам отделов:\n";
    ssort (heads, 6, sizeof (User), cmp2);
    print_id (heads, 6);
}
```

Вы можете получить адрес перегруженной функции путем присваивания указателю на функцию или инициализации указателя на функцию. В этом случае тип указателя, которому было произведено присваивание, используется при выборе перегруженной функции. Например:

```
void f (int);
int f (char);

void (*pf1) (int) = &f;          // void f(int)
int (*pf2) (char) = &f;          // int f(char)
void (*pf3) (char) = &f;         // ошибка: нет функции void f(char)
```

Функция должна вызываться через указатель на функцию с (в точности) правильными типами аргументов и возвращаемого значения. Не производится неявного преобразования типов аргументов или возвращаемого значения при присвоении указателям или их инициализации. Это означает, что

```
int cmp3 (const mytype*, const mytype*);
```

является недопустимым аргументом для ***ssort*** (). Причина в том, что разрешение использования ***cmp3*** в качестве аргумента ***ssort*** () нарушило бы гарантию того, что ***ssort*** () вызовется с аргументами ***mytype**** (см. также § 9.2.5).

7.8. Макросы

Макросы имеют большое значение в C, но в C++ они используются значительно реже. Первое правило о макросах: не используйте их, если вы не обязаны этого делать. Практически каждый макрос свидетельствует о недостатке в языке программирования, программе или программисте. Так как макросы изменяют текст программы до обработки его компилятором, они создают проблемы для многих инструментов разработки. Поэтому если вы пользуетесь макросами, можете ожидать худшей работы отладчиков, генераторов списков перекрестных ссылок, профайлеров[1] и т. д. Если вам необходимо использовать макросы, внимательно прочитайте руководство по вашей реализации препроцессора C++ и не пытайтесь искать слишком хитроумных решений. Кроме того, необходимо предостеречь читателей: следуйте соглашению об использовании в именах макросов преимущественно заглавных букв. Синтаксис макросов описан в § A.11.

Простой макрос определяется следующим образом:

```
#define NAME rest of line
```

Когда встречается лексема ***NAME***, она замещается на ***rest of line***. Например:

```
named = NAME
```

будет преобразовано в

```
named = rest of line
```

Макрос можно определить с аргументами. Например:

```
#define MAC (x, y) argument1: x argument2: y
```

При использовании ***MAC*** должны присутствовать два строковых аргумента. Они заменят ***x*** и ***y*** при макроподстановке. Например, результатом макроподстановки

```
expanded = MAC (foo bar, yuk yuk)
```

будет

```
expanded = argument1: foo bar argument2: yuk yuk
```

Имена макросов нельзя перегружать. Кроме того, рекурсивные вызовы создают проблемы, которые препроцессор не может решить:

```
#define PRINT (a, b) cout << (a) << (b)
#define PRINT (a, b, c) cout << (a) << (b) << (c)    // проблема? замещение, а не перегрузка

#define FAC (n) (n>1) ? n*FAC(n-1) : 1               // проблема: рекурсивный макрос
```

Макросы манипулируют строками символов и мало что знают о синтаксисе C++ и ничего — о типах C++ и областях видимости имен. Компилятор видит текст только после макроподстановки, поэтому и ошибка в макросе будет замечена после подстановки, а не в месте его определения. Это может привести к очень странным сообщениям об ошибках.

Вот несколько примеров приемлемых макросов:

```
#define CASE break;case
#define FOREVER for(;;)
```

[1] Профайлер — инструментальное средство, предназначенное для измерения производительности, расхода памяти и других параметров работы программы. — *Примеч. ред.*

Вот несколько примеров совершенно ненужных макросов:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

Несколько «опасных» макросов:

```
#define SQUARE (a) a*a
#define INCR_xx (xx)++
```

Чтобы убедиться, что они опасны, попытайтесь осуществить макроподстановку в следующих примерах:

```
int xx = 0;                    // глобальный счетчик

void f()
{
    int xx = 0;                // локальная переменная
    int y = SQUARE (xx+2);     // y=xx+2*xx+2; то есть y=xx+(2*xx)+2
    INCR_xx;                   // инкремент локальной xx
}
```

Если вам необходимо использовать макросы, пользуйтесь оператором разрешения области видимости :: при обращении к глобальным переменным (§ 4.9.4) и везде, где это только возможно, заключайте имя аргумента макроса в скобки. Например:

```
#define MIN (a, b) (((a) < (b)) ? (a) : (b))
```

Если вы написали сложный макрос, который требует комментариев, лучше пользоваться комментариями /* */, потому что препроцессор С, который не знает о //-стиле комментариев, иногда используется как часть среды C++. Например:

```
#define M2 (a) something (a)        /* правильный комментарий */
```

При помощи макросов вы можете создать свой собственный язык. Даже если вы предпочитаете такой «улучшенный» язык обыкновенному C++, для других программистов на C++ он будет непонятен. Более того, препроцессор С является очень простым макро-процессором. Когда вы попытаетесь сделать что-нибудь нетривиальное, скорее всего окажется, что это либо невозможно, либо слишком трудоемко. Механизмы ***const***, ***inline*** (встраивание тела функции в место ее вызова), ***template*** (шаблоны), ***enum*** и ***namespace*** (пространства имен) создавались в качестве альтернативы традиционному использованию препроцессора. Например:

```
const int answer = 42;
template<class T> inline T min (T a, T b) { return (a<b) ? a : b; }
```

При написании макроса иногда требуется новое имя для чего-нибудь. Строку можно создать при помощи конкатенации двух строк макро-оператором ##. Например:

```
#define NAME2 (a, b) a##b

int NAME2 (hack, cah) ();
```

Результатом будет

```
int hackcah ();
```

Директива

```
#undef X
```

гарантирует, что более не существует макроса по имени ***X*** независимо от того, существовал ли макрос с таким именем до этой директивы. Это обеспечивает некоторый механизм защиты от нежелательных макросов. Однако, не всегда легко узнать, какое действие ***X*** должен был бы оказывать макрос ***X*** на фрагмент кода.

7.8.1. Условная компиляция

Почти невозможно избежать применения макросов в одном случае. Директива *#ifdef идентификатор* заставляет компилятор (при выполнении условия) игнорировать последующий текст, пока не встретится директива *#endif*. Например:

```
int f(int a
#ifdef arg_two
, int b
#endif
);
```

выдает компилятору

```
int f(int a
);
```

если макрос по имени ***arg_two*** не был определен (***#define***). Подобный пример приводит в замешательство инструменты разработки, которые ожидают разумного поведения от программиста.

В большинстве случаев использование *#ifdef* менее загадочно и, при наличии определенных ограничений, приносит мало вреда. См. также § 9.3.3.

Имена макросов, используемые в *#ifdef*, следует тщательно выбирать, чтобы они не совпадали с обычными идентификаторами. Например:

```
struct Call_info {
        Node* arg_one,
        Node* arg_two,
        // ...
};
```

Этот невинный текст вызовет неприятные последствия, если кто-нибудь напишет:

```
#define arg_two x
```

К сожалению, в неизбежных заголовочных файлах общего пользования содержится множество опасных и ненужных макросов.

7.9. Советы

[1] Относитесь с подозрением к неконстантным аргументам; если вы хотите, чтобы функция модифицировала свои аргументы, пользуйтесь вместо этого указателями и возвращаемым значением; § 5.5.

[2] Если необходимо минимизировать затраты на копирование аргументов, пользуйтесь константными ссылками в качестве аргументов; § 5.5.

[3] Используйте модификатор ***const*** регулярно и последовательно; § 7.2.

[4] Избегайте применения макросов; § 7.8.
[5] Избегайте функций с неуказанным количеством аргументов; § 7.6.
[6] Не возвращайте указатели или ссылки на локальные переменные ; § 7.3.
[7] Используйте перегрузку, когда функции выполняют концептуально схожую работу над объектами различных типов; § 7.4.
[8] При перегрузке функций с целыми аргументами реализуйте достаточное количество функций для устранения типичных случаев неоднозначности; § 7.4.3.
[9] Если вы собираетесь воспользоваться указателем на функцию, подумайте, не будет ли виртуальная функция (§ 2.5.5) или шаблон (§ 2.7.2) лучшей альтернативой; § 7.7.
[10] Если вам приходится использовать макросы, дайте им некрасивые, бросающиеся в глаза имена с большим количеством заглавных букв; § 7.8.

7.10. Упражнения

1. (*1) Напишите следующие объявления: функция с аргументами «указатель на символ» и «ссылка на целое», которая не возвращает значение; указатель на такую функцию; функцию с таким указателем в качестве аргумента; функцию, возвращающая такой указатель. Напишите определение функции, которая принимает такой указатель в качестве аргумента и возвращает его. Подсказка: воспользуйтесь *typedef*.
2. (*2) Что означает следующее? Для чего это может пригодиться?

```
Typedef int (&rifii) (int, int);
```

3. (*1.5) Напишите программу типа «Здравствуй мир!», которая получает имя человека в качестве параметра командной строки и выводит «Здравствуй, *имя*!». Измените программу таким образом, чтобы она получала произвольное количество имен-аргументов и здоровалась с каждым из обладателей имен.
4. (*1.5) Напишите программу, которая читает произвольное количество файлов, чьи имена задаются в командной строке, и выводит их один за другим в ***cout***. Так как эта программа осуществляет конкатенацию своих аргументов для создания вывода, вы можете назвать ее ***cat***.
5. (*2) Перепишите небольшую С-программу на C++. Модифицируйте заголовочные файлы таким образом, чтобы в них были объявлены все вызываемые функции и типы всех аргументов. Где возможно замените ***#define*** на ***enum***, ***const*** или ***inline***. Удалите объявления ***extern*** из файлов *.c* и, если необходимо, приведите определения функций к синтаксису C++. Замените вызовы ***malloc*** () и ***free*** () на ***new*** и ***delete***. Удалите ненужные приведения типов.
6. (*2) Реализуйте ***ssort*** () (§ 7.7) с использованием более эффективного алгоритма. Подсказка: ***qsort*** ().
7. (*2.5) Имеется:

```
struct Tnode {
        string word;
        int count,
        Tnode* left;
        Tnode* right;
};
```

Напишите функцию добавления новых слов в дерево с узлами ***Tnode***. Напишите функцию вывода такого дерева. Напишите функцию вывода дерева с ***Tnode*** в алфавитном порядке. Измените ***Tnode*** таким образом, чтобы эта структура содержала (только) указатель на произвольно длинное слово, хранящееся (воспользуйтесь оператором ***new***) в виде массива символов в свободной памяти. Измените функции для работы с новым определением ***Tnode***.

8. (*2.5) Напишите функцию, которая транспонирует двумерный массив, то есть строки превращает в столбцы, а столбцы — в строки. Подсказка: § B.7.
9. (*2) Напишите программу шифрования, которая читает из ***cin*** и записывает закодированные символы в ***cout***. Вы можете воспользоваться следующей простой схемой кодировки: кодом, соответствующему символу ***c***, является ***c^key[i]***, где ***key*** — строка, переданная в качестве параметра командной строки. Программа использует циклически символы из ***key*** до тех пор, пока не будет считан весь ввод. В результате повторной шифровки закодированного текста с тем же ключом (***key***) получится исходная строка. Если передана нулевая строка, кодирование не осуществляется.
10. (*3.5) Напишите программу дешифровки сообщений, закодированных методом, описанным в § 7.10[9], не зная ключа. Подсказка: см. David Kahn: *The Codebreakers*, Macmillan, 1967, New York, pp. 207–213.
11. (*3) Напишите функцию ***error***, получающую ***printf***-подобную строку форматирования, содержащую директивы %***s***, %***c*** и %***d***, и произвольное количество других аргументов. Не пользуйтесь функцией ***printf*** (). Если вы не знаете смысл этих директив, обратитесь к § 21.8. Используйте <***cstdarg***>.
12. (*1) Как бы вы выбирали имена для типов указателей на функции, определяемых с помощью ***typedef***?
13. (*2) Просмотрите несколько программ, обращая внимание на разницу стилей при выборе имен. Как используются заглавные буквы? Как используется символ подчеркивания? Когда используются короткие имена, такие как ***i*** и ***x***?
14. (*1) Что плохо в следующих макро-определениях?

```
#define PI = 3.141593;
#define MAX (a, b) a>b?a:b
#define fac (a) (a)*fac ((a)-1)
```

15. (*3) Напишите простой макро-процессор (наподобие препроцессора C). Считывайте из ***cin*** и выводите в ***cout***. В начале не обрабатывайте макросы с аргументами. Подсказка: В калькуляторе (§ 6.1) имеется таблица символов и лексический анализатор, которые вы можете модифицировать.
16. (*2) Реализуйте ***print*** () из § 7.5.
17. (*2) Добавьте функции типа ***sqrt*** (), ***log*** () и ***sin*** () к калькулятору из § 6.1. Подсказка: предопределите эти имена и вызывайте функции при помощи массива указателей на функции. Не забывайте проверять аргументы перед вызовом функции.
18. (*1) Напишите функцию вычисления факториала, не использующую рекурсию. См. также § 11.14[6].
19. (*2) Напишите функции, которые добавляют один день, один месяц и один год к ***Date***, определенной в § 5.9[13]. Напишите функцию, которая возвращает день недели для заданной даты. Напишите функцию, которая возвращает дату первого понедельника, следующего за заданной датой.

8

Пространства имен и исключения

Год 787! От Рождества Христова?
— Монти Пайтон

Не бывает правил без исключений.
— Роберт Бартон

Модульность, интерфейсы и исключения — пространства имен — ***using*** — ***using namespace*** — разрешение конфликтов имен — поиск имен — композиция пространств имен — псевдонимы пространств имен — пространства имен и код на С — исключения — ***throw*** и ***catch*** — исключения и структура программы — советы — упражнения.

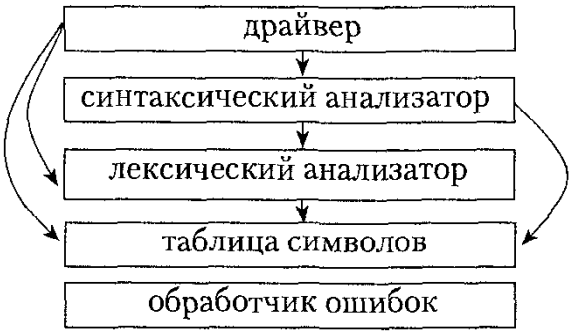
8.1. Разбиение на модули и интерфейсы

Любая реальная программа состоит из некоторого количества отдельных частей. Например, даже простая программа, такая как ***Здравствуй, мир***!, состоит из по крайней мере двух частей: пользовательский код требует вывести ***Здравствуй, мир***!, а система ввода/вывода непосредственно осуществляет вывод.

Рассмотрим пример калькулятора из § 6.1. Можно считать, что он состоит из пяти частей:

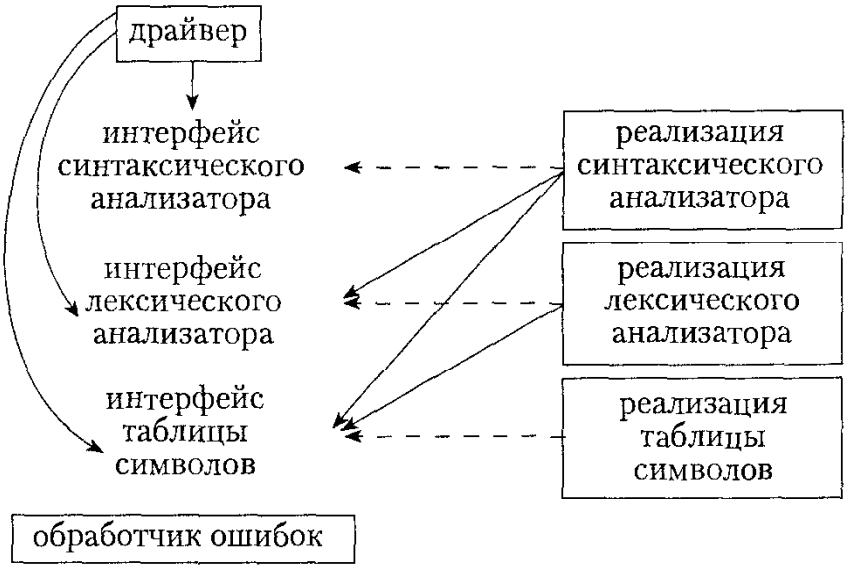
[1] Обработчика, выполняющего синтаксический анализ.
[2] Лексического анализатора, выделяющего лексемы из последовательности символов.
[3] Таблицы символов, в которой хранятся пары (строка, значение).
[4] Управляющей программы (драйвера) ***main*** ().
[5] Обработчика ошибок.

Это можно изобразить графически:



Стрелки означают «использует». С целью упрощения схемы, я не отобразил тот факт, что каждая часть пользуется обработчиком ошибок. В действительности, калькулятор был задуман в виде трех частей, а драйвер и обработчик ошибок были добавлены впоследствии для полноты картины.

Когда один модуль пользуется другим, ему не надо знать все подробности об используемом модуле. В идеале большая часть деталей реализации модуля должна быть неизвестна его пользователям. Следовательно, мы проводим различие между модулем и его интерфейсом. Например, синтаксический анализатор обращается только к интерфейсу лексического анализатора, а не к его реализации. Лексический анализатор просто реализует функции, объявленные в его интерфейсе. Это можно изобразить графически следующим образом:



Пунктирные линии означают «реализует». Я полагаю, что приведенная схема реально отражает структуру нашей программы, и наша работа в качестве программистов состоит в том, чтобы добросовестно отразить это в коде. Тогда код будет простым, эффективным, понятным, легким в сопровождении и т. д., потому что он будет непосредственно отражать имеющийся проект.

В последующих разделах показано, как можно сделать более простой и понятной логическую структуру калькулятора, а в § 9.3 описывается, как физически организовать исходные тексты программ в удобном виде. Калькулятор — крошечная программа и в «реальной жизни» я бы не стал пользоваться механизмом пространств имен и раздельной компиляцией (§ 2.4.1, § 9.1) в той мере, в какой я это делаю здесь. Речь идет о демонстрации методов, полезных при написании больших программ. В реальных программах каждый «модуль», представленный в виде отдельного пространства имен, содержал бы сотни функций, классов, шаблонов и т. д.

Для демонстрации различных методов и средств языка разбиение на модули проводится по этапам. В «реальной жизни» вряд ли были бы пройдены все эти этапы. Опытный программист с самого начала построил бы систему «почти правильно». Однако по мере развития программы ее структура нередко кардинально меняется.

Обработка ошибок используется во всех местах программы. При разбиении программы на модули, или наоборот, при сборке программы из модулей, мы должны заботиться о минимизации зависимостей между модулями, возникающих из-за обработки

ошибок. В C++ имеется механизм обработки исключений, который позволяют отделить процесс обнаружения ошибок и сообщения об ошибках от непосредственной их обработки. Поэтому обсуждение представления модулей как пространств имен (§ 8.2) сопровождается демонстрацией того, как можно воспользоваться исключениями для дальнейшего улучшения модульности (§ 8.3) программы.

Существует гораздо больше понятий модульности, чем обсуждается в этой и следующей главах. Например, для демонстрации важных аспектов модульности можно было бы воспользоваться параллельно работающими процессами, которые обмениваются между собой сообщениями. Аналогично, важными вопросами, не обсуждаемыми здесь, являются использование раздельных адресных пространств и обмен информацией между ними. Я полагаю, что эти вопросы, связанные с модульностью, довольно независимы и находятся в разных плоскостях. Любопытно, что в любом случае разбиение системы на модули реализуется довольно просто. Серьезная проблема — обеспечить безопасное, удобное и эффективное взаимодействие модулей.

8.2. Пространства имен

Пространство имен является механизмом отражения логического группирования. То есть если некоторые объявления можно объединить по какому-либо критерию, их можно поместить в одно пространство имен для отражения этого факта. Например, объявления синтаксического анализатора для калькулятора (§ 6.1.1) можно поместить в пространство имен ***Parser***:

```
namespace Parser {
        double expr (bool);
        double prim (bool get) { /* ... */ }
        double term (bool get) { /* ... */ }
        double expr (bool get) { /* ... */ }
}
```

Функция ***expr*** () должна быть сначала объявлена и только затем определена для того, чтобы разорвать замкнутый круг зависимостей, описанный в § 6.1.1.

Часть настольного калькулятора, отвечающую за ввод, тоже можно поместить в собственное пространство имен.

```
namespace Lexer {

        enum Token_value {
                NAME,           NUMBER,         END,
                PLUS='+',       MINUS='-',      MUL='*',    DIV='/',
                PRINT=';',      ASSIGN='=',     LP='(',     RP=')'
        };

        Token_value curr_tok;
        double number_value;
        std::string string_value;

        Token_value get_token () { /* ... */ }
}
```

Подобное использование пространства имен делает более очевидным что, собственно, синтаксический и лексический анализаторы предоставляют своим пользователям.

Однако если бы я добавил сюда исходные тексты функций, эта структура не выглядела бы такой стройной. Там, где тела функций включены в объявления пространства имен реального размера, вам пришлось бы пробиваться сквозь страницы кода для того, чтобы определить, какие функции предлагаются, или, другими словами, для того, чтобы понять интерфейс.

Альтернативой отдельно описанным интерфейсам может служить некоторый инструмент, который извлекает интерфейс из модуля, содержащего детали реализации. Я не считаю это хорошим решением. Определение интерфейсов является важнейшей задачей этапа проектирования (см. § 23.4.3.4.). Модуль может иметь различные интерфейсы для различных пользователей и довольно часто интерфейс разрабатывается задолго до прояснения деталей реализации.

Приведем пример ***Parser*** с интерфейсом, отделенным от реализации:

```
namespace Parser {
        double prim (bool get);
        double term (bool get);
        double expr (bool get);
}

double Parser::prim (bool get) { /* ... */ }
double Parser::term (bool get) { /* ... */ }
double Parser::expr (bool get) { /* ... */ }
```

Обратите внимание, что в результате разделения реализации и интерфейса, каждая функция имеет ровно одно объявление и одно определение. Пользователи увидят только интерфейс, содержащий объявления. Реализация (в нашем случае — тела функций) будет находиться «где-то в другом месте», куда пользователю нет необходимости заглядывать.

Как показано в примере, член пространства имен можно объявить внутри определения пространства имен, а определить позднее, при помощи следующей записи:

имя_пространства_имен::*имя_члена*.

Члены пространства имен объявляются следующим образом:

```
namespace имя_пространства_имен {
        // объявления и определения
}
```

Нельзя объявить новый член пространства имен вне его определения, используя явный квалификатор. Например:

```
void Parser::logical (bool);        // ошибка: в Parser нет logical()
```

Идея состоит в том, чтобы достаточно легко можно было бы найти все имена в объявлении пространства имен и быстро выявить такие ошибки, как опечатки и несоответствие типов. Например:

```
double Parser::trem (bool);         // ошибка: в Parser нет trem()
double Parser::prim (int);          // ошибка: у Parser::prim() аргумент bool
```

Пространство имен является областью видимости. Поэтому «пространство имен» является фундаментальной и относительно простой концепцией. Чем больше программа, тем полезнее становятся пространства имен для адекватного отражения логического разделения ее частей. Обычные локальные и глобальные области видимости и классы являются пространствами имен (§ B.10.3).

В идеале, любая часть программы принадлежит некоторому логическому фрагменту («модулю»). Поэтому любое объявление в нетривиальной программе в идеале должно находиться в некотором пространстве, имя которого указывает на его роль в программе. Исключение составляет ***main*** (), которое должно быть глобальным, чтобы среда времени выполнения распознавала его как специальное имя (§ 8.3.3).

8.2.1. Имена с квалификаторами

Пространство имен является областью видимости. Обычные правила областей видимости применимы и к пространствам. Поэтому, если имя предварительно объявлено в пространстве имен или в охватывающей области, дальше его можно использовать без проблем. Имя из другого пространства имен можно использовать при помощи явного указания этого пространства в качестве квалификатора. Например:

```
double Parser::term (bool get)          // Parser:: — это квалификатор
{
    double left = prim (get);           // квалификатор не требуется

    for (;;)
        switch (Lexer::curr_tok) {      // Lexer:: — это квалификатор
        case Lexer::MUL;                // Lexer:: — это квалификатор
            left *= prim (true);        // квалификатор не требуется
        //...
        }
    //...
}
```

Квалификатор ***Parser***:: необходим, чтобы указать, что ***term*** () объявлена в ***Parser***, а не является некоторой глобальной функцией. Так как ***term*** () является членом ***Parser***, при обращении из нее к ***prim*** () нет необходимости пользоваться квалификатором. Однако, при отсутствии квалификатора ***Lexer*** переменная ***curr_tok*** считалась бы необъявленной, потому что члены пространства имен ***Lexer*** не находятся в области видимости в пространстве ***Parser***.

8.2.2. Объявления using

Если имя часто используется вне пределов своего пространства имен, довольно утомительно писать его каждый раз с квалификатором. Рассмотрим пример:

```
double Parser::prim (bool get)          // обработка первичных выражений
{
    if (get) Lexer::get_token ()

    switch (Lexer::curr_tok) {
    case Lexer::NUMBER:                 // константа с плавающей точкой
        Lexer::get_token ();
        return Lexer::number_value;

    case Lexer::NAME:
    {   double& v = table[Lexer::string_value];
        if (Lexer::get_token () == Lexer::ASSIGN) v = expr (true);
```

```
            return v;
        }
        case Lexer::MINUS                   // унарный минус
            return -prim(true);
        case Lexer::LP
        {   double e = expr(true);
            if (Lexer::curr_tok != Lexer::RP) return error("ожидалась )");
            Lexer::get_token();             // пропустить скобки ')'
            return e;
        }
        case Lexer::END:
            return 1;
        default:
            return error("ожидалось первичное выражение");
        }
    }
```

Непрерывное повторение квалификатора ***Lexer*** к тому же отвлекает внимание. Эту многословность можно устранить ***using-объявлением***, которое позволяет указать в одном месте, что ***get_token*** () находится в пространстве ***Lexer***. Например:

```
double prim(bool get)                   // обработка первичных выражений
{
    using Lexer::get_token;             // использовать get_token из Lexer
    using Lexer::curr_tok;              // использовать curr_tok из Lexer
    using Error::error;                 // использовать error из Error

    if (get) get_token();

    switch (curr_tok) {
    case Lexer::NUMBER                  // константа с плавающей точкой
        get_token();
        return Lexer::number_value;
    case Lexer::NAME
    {   double& v = table[Lexer::string_value];
        if (get_token() == Lexer::ASSIGN) v = expr(true);
        return v;
    }
    case Lexer::MINUS                   // унарный минус
        return -prim(true);
    case Lexer::LP
    {   double e = expr(true);
        if (curr_tok != Lexer::RP) return error("ожидалась )");
        get_token();                    // пропустить скобки ')'
        return e;
    }
    case Lexer::END:
        return 1;
    default:
        return error("ожидалось первичное выражение");
    }
}
```

using-объявление вводит локальный синоним.

Такие синонимы следует делать как можно более локальными во избежание конфликтов имен. Однако, все функции синтаксического анализатора используют аналогичные наборы имен из других модулей. Поэтому мы можем поместить *using-объявление* в определение пространства имен ***Parser***:

```
namespace Parser {
        double prim (bool);
        double term (bool);
        double expr (bool);

        using Lexer::get_token;          // использовать get_token из Lexer
        using Lexer::curr_tok,           // использовать curr_tok из Lexer
        using Error::error;              // использовать error из Error
}
```

Это позволит нам упростить функции из ***Parser*** практически до их первоначального вида (§ 6.1.1):

```
double Parser::term (bool get)           // умножение и деление
{
        double left = prim (get);

        for (;;)
                switch (curr_tok) {
                case Lexer::MUL:
                        left *= prim (true);
                        break;

                case Lexer::DIV:
                        if (double d == prim (true)) {
                                left /= d;
                                break;
                        }
                        return error ("деление на 0");

                default:
                        return left;
                }
}
```

Я мог бы включить имена лексем в пространство ***Parser***. Однако, я умышленно оставил их с квалификаторами в качестве напоминания о зависимости ***Parser*** от ***Lexer***.

8.2.3. Директивы using

Что если бы мы задались целью упростить функции ***Parser*** *в точности* до их первоначального вида? Это было бы разумной целью для большой программы, которая преобразуется с целью использования пространств имен, исходя из программы с менее явно выраженной модульностью.

using-директива делает доступными имена из пространства имен почти точно так же, как если бы они были объявлены вне своих пространств (§ 8.2.8). Например:

```
namespace Parser {
        double prim (bool);
```

```
        double term (bool);
        double expr (bool);

        using namespace Lexer;          // делает доступными все имена из Lexer
        using namespace Error;          // делает доступными все имена из Error
}
```

Это позволит нам написать функции ***Parser*** в точности так, как мы это делали с самого начала (§ 6.1.1):

```
double Parser::term (bool get)          // умножение и деление
{
        double left = prim (get);

        for (;;)
                switch (curr_tok) {
                case MUL:               // умножение
                        left *= prim (true);
                        break;

                case DIV:               // деление
                        if (double d == prim (true)) {
                                left /= d;
                                break;
                        }
                        return error ("деление на 0");
                default:
                        return left;
                }
}
```

Глобальные ***using***-*директивы* хороши для переписывания старого кода в новом стиле (§ 8.2.9). В других ситуациях их лучше избегать. Можно сказать, что ***using***-*директива* является средством композиции пространств имен (§ 8.2.8). В функциях (и только в них) ***using***-*директива* используется для удобства формы записи (§ 8.3.3.1).

8.2.4. Множественные интерфейсы

Должно быть ясно, что созданное нами определение пространства ***Parser*** не является интерфейсом, который ***Parser*** предоставляет пользователям: наш ***Parser*** состоит из набора объявлений, которые позволяют удобным образом реализовать функции синтаксического анализатора. Интерфейс же пользователей ***Parser*** должен быть намного проще:

```
namespace Parser {
        double expr (bool);
}
```

К счастью, одновременно могут существовать два *определения пространства имен* ***Parser*** и каждое из них может быть использовано там, где это удобнее. Итак, пространство ***Parser*** можно рассматривать как:

[1] общую среду для функций, реализующих синтаксический анализатор.

[2] внешний интерфейс для пользователей синтаксического анализатора.

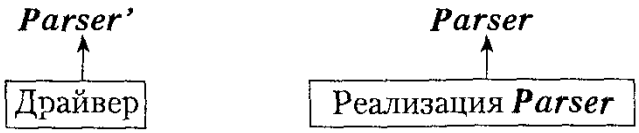
Итак, код драйвера ***main*** () должен видеть только

```
namespace Parser {                    // интерфейс для пользователей
        double expr (bool);
}
```

Функции же, реализующие синтаксический анализатор, должны видеть интерфейс, который мы сочли наилучшим для выражения общей среды. А именно:

```
namespace Parser {                    // интерфейс для разработчиков
        double prim (bool);
        double term (bool);
        double expr (bool);

        using Lexer::get_token,       // использовать get_token из Lexer
        using Lexer::curr_tok         // использовать curr_tok из Lexer
        using Error::error;           // использовать error из Error
}
```

или в графическом виде:



Стрелки отражают отношения «использует предоставляемый интерфейс».

Parser' — это «малый» интерфейс, предоставляемый пользователям. Имя ***Parser'*** (читается «Parser штрих») не является идентификатором C++. Оно намеренно было выбрано, чтобы подчеркнуть, что этот интерфейс не имеет отдельного имени в программе. Отсутствие отдельного имени не должно вызывать дополнительных затруднений, потому что программисты свободно придумывают различные имена для различных интерфейсов и потому что программа физически хранится (см. § 9.3.2) в виде нескольких файлов с различными именами.

Интерфейс для разработчиков синтаксического анализатора шире, чем интерфейс, предназначенный для его пользователей. Подобный интерфейс в реальном модуле реальной системы менялся бы гораздо чаще, чем интерфейс пользователя. Очень важно, что пользователи модуля (в нашем случае — ***main*** (), использующая ***Parser***) изолированы от этих изменений.

Нам не требуется два отдельных пространства имен для выражения двух различных интерфейсов, но мы могли бы обеспечить и это, если бы возникла соответствующая потребность. Разработка интерфейсов является одной из самых важных задач на этапе проектирования — решая ее, можно получить как много пользы, так и массу неприятностей. Следовательно, стоит подробно рассмотреть, что мы пытаемся получить и обсудить несколько альтернатив.

Помните, что предложенное выше решение является самым простым из рассматриваемых, и часто наилучшим. Его самым слабым местом является то, что оба интерфейса имеют одинаковые имена и что компилятор может не иметь достаточной информации для проверки согласованности двух определений пространства имен. Однако, обычно он в состоянии осуществить проверку. Более того, компоновщик отлавливает большинство ошибок, пропущенных компилятором.

Предложенное здесь решение я использую для обсуждения физического разбиения на модули (§ 9.3) и рекомендую его при отсутствии дополнительных логических ограничений (см. также § 8.2.7).

8.2.4.1. Альтернативы при проектировании интерфейса

Целью использования интерфейсов является сведение к минимуму зависимостей между различными частями программы. Минимальные интерфейсы приводят к легким в понимании, имеющим лучшие свойства в плане сокрытия данных, легче модифицируемым и быстрее компилируемым программам.

Когда речь идет о зависимостях, важно помнить, что компиляторы и программисты склонны применять примитивный подход типа «Если определение видимо в точке X, то все, написанное в точке X, зависит от всего определения». Пользуясь приведенными выше определениями, рассмотрим пример:

```
namespace Parser {                    // интерфейс для разработчиков
        //...
        double expr (bool),
        //...
}

int main ()
{
        //...
        Parser  expr (false),
        // ..
}
```

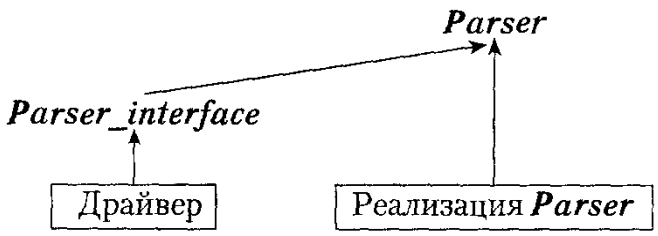
Функция ***main*** () зависит только от ***Parser::expr*** (), но требуется время, интеллект, опыт и т. д. для того, чтобы это определить. Как следствие, люди и компиляторы в программах реального размера не рискуют и полагают, что там, где может существовать зависимость, она имеет место быть. Как правило, такой подход прекрасно себя оправдывает.

Таким образом, нашей целью является написание программы, в которой набор потенциальных зависимостей сведен к набору действительных зависимостей.

Сначала мы пытаемся сделать очевидное: определить пользовательский интерфейс синтаксического анализатора в терминах интерфейса разработчика, который мы уже имеем:

```
namespace Parser {                    // интерфейс для разработчиков
        //...
        double expr (bool),
        //...
}

namespace Parser_interface {          // интерфейс для пользователей
        using Parser  expr,
}
```

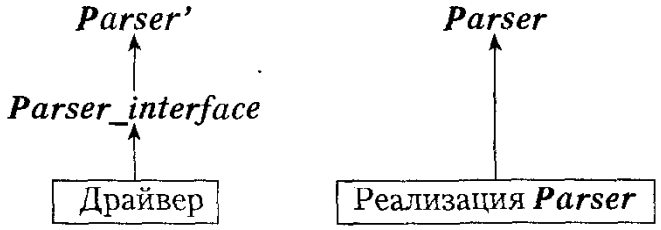
Ясно, что пользователи ***Parser_interface*** зависят только от ***Parser::expr*** () (причем косвенно). Поверхностный взгляд на схему зависимостей дает нам следующую картину:



Теперь драйвер сильно зависит от любых изменений в интерфейсе ***Parser***, от которых мы хотели его изолировать. Даже такой вид зависимости является нежелательным, поэтому мы явно ограничиваем зависимость ***Parser_interface*** от ***Parser***, оставив видимой только существенную часть интерфейса для разработчиков (ранее мы назвали ее ***Parser'***) при определении ***Parser_interface***:

```
namespace Parser {                  // интерфейс для пользователей
        double expr (bool),
}
namespace Parser_interface {        // отдельный именованный интерфейс
                                    // для пользователей
        using Parser::expr;
}
```

или графически:



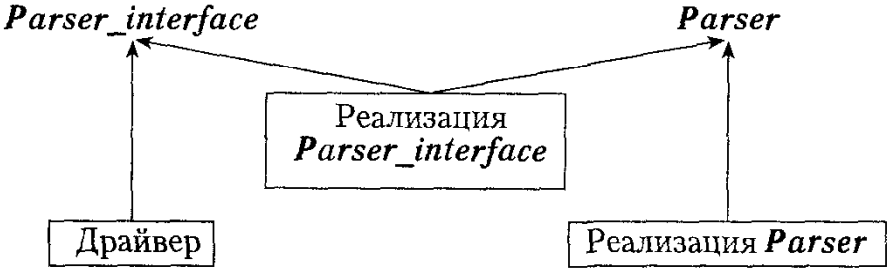
И снова для проверки согласованности ***Parser*** и ***Parser'*** мы полагаемся на всю систему компиляции, а не на компиляцию отдельного модуля. Это решение отличается от предложенного в § 8.2.4 только дополнительным пространством имен ***Parser_interface***. Если бы мы захотели, мы могли бы включить в ***Parser_interface*** конкретное представление, введя его собственную функцию ***expr*** ():

```
namespace Parser_interface {
        double expr (bool);
}
```

Теперь ***Parser*** не обязан находиться в области видимости для того, чтобы мы могли определить ***Parser_interface***. Он должен быть видимым только там, где определяется ***Parser_interface::expr*** ():

```
double Parser_interface::expr (bool get)
{
        return Parser::expr (get),
}
```

Последний вариант можно представить графически следующим образом:



Теперь все зависимости сведены к минимуму. Все конкретно и именовано соответствующим образом. Однако, для большинства проблем, с которыми я сталкиваюсь, такое решение является неоправданно сложным.

8.2.5. Разрешение конфликтов имен

Пространства имен предназначены для отражения логической структуры. Простейшим примером такой структуры является отделение кода, написанного одним человеком, от кода, написанного другим. Это простейшее разделение может иметь огромное практическое значение.

При использовании единственной глобальной области видимости неоправданно сложно формировать программу из отдельных частей. Проблема состоит в том, что в каждой отдельной (предположительно) части могут определяться одинаковые имена. Когда части объединяются в одну программу, происходит конфликт имен. Рассмотрим пример:

```
// my.h:
    char f(char);
    int f(int);
    class String { /* ... */ };

// your.h:
    char f(char);
    double f(double);
    class String { /* ... */ };
```

При наличии таких определений непросто использовать ***my.h*** и ***your.h*** одновременно. Очевидным решением является помещение каждого набора объявлений в свое собственное пространство имен:

```
namespace My {
    char f(char);
    int f(int);
    class String { /* ... */ };
}

namespace Your {
    char f(char);
    double f(double);
    class String { /* ... */ };
}
```

Теперь мы можем пользоваться объявлениями из ***My*** и ***Your*** при помощи явных квалификаторов (§ 8.2.1), *using-объявлений* (§ 8.2.2) или (§ 8.2.3) *using-директив*.

8.2.5.1. Неименованные пространства имен

Иногда полезно помещать объявления в пространство имен просто ради того, чтобы исключить возможность конфликта имен. В этом случае целью является сохранение локальности кода, а не предоставление интерфейса пользователям. Например:

```
#include "header.h"
namespace Mine {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

Так как мы не хотим, чтобы имя ***Mine*** было доступно вне локального контекста, введение имени вызывает определенное неудобство, которое к тому же может привести к случайному конфликту с другим именем. В этом случае мы можем ввести пространство имен без имени:

```
#include "header.h"
namespace {
        int a;
        void f() { /* ... */ }
        int g() { /* ... */ }
}
```

Ясно, что должен существовать какой-то способ доступа к членам неименованного пространства имен извне. Следовательно, неименованное пространство имен подразумевает ***using-директиву***. Предыдущее объявление эквивалентно

```
namespace $$$ {
        int a;
        void f() { /* ... */ }
        int g() { /* ... */ }
}
using namespace $$$;
```

где $$$ является некоторым именем, уникальным для области видимости, в которой определяется пространство имен. В частности, неименованные пространства имен в разных единицах трансляции различны. Как и предполагалось, нельзя осуществить доступ из одной единицы трансляции к члену неименованного пространства имен из другой единицы.

8.2.6. Поиск имен

Является скорее правилом, чем исключением, что функция с аргументом типа ***T*** определяется в том же пространстве имен, что и ***T***. Как следствие, если функция не найдена в контексте ее использования, осуществляется поиск в пространствах имен ее аргументов. Например:

```
namespace Chrono {
        class Date { /* ... */ };

        bool operator == (const Date&, const std::string&);
        std::string format (const Date&);          // символьное представление
        // ...
}

void f(Chrono::Date d, int i)
{
        std::string s = format (d);                // Chrono::format()
        std::string t = format (i);                // ошибка: в области видимости
                                                   // нет такой функции format()
}
```

По сравнению с использованием явных квалификаторов это правило поиска имен экономит время при вводе программы и не приводит к «загрязнению» пространства

имен, как это может делать *using-директива*. Это правило особенно полезно в случае применения к операндам операторов (§ 11.2.4) и аргументам шаблонов (§ B.13.8.4), когда явная квалификация может быть очень сложной.

Обратите внимание, что само пространство имен должно быть в области видимости и функция должна быть объявлена до ее использования.

Естественно, аргументы функций могут быть из различных пространств имен. Например:

```
void f(Chrono::Date d, std::string s)
{
    if (d==s) {
        //...
    }
    else if (d == "4 августа, 1914г.") {
        //...
    }
}
```

В таких случаях осуществляется сначала поиск функции в области видимости вызова (как обычно), а затем — в пространствах имен каждого аргумента (включая класс каждого аргумента и его базовые классы). После этого ко всем найденным функциям применяются обычные правила разрешения перегрузки имен (§ 7.4). В частности, для вызова *d==s* осуществляется поиск *operator* == в области видимости, окружающей *f*(), в пространстве имен *std* (там определен == для *string*) и в пространстве имен *Chrono*. Существует оператор *std::operator* == (), но у него нет аргумента *Date*, поэтому используется *Chrono::operator* == (), который имеет такой аргумент (см. также § 11.2.4).

Когда член класса вызывает функцию с некоторым именем, другие члены того же класса и его родительских классов имеют приоритет над функциями, найденными на основании информации о типах аргументов. Ситуация с операторами иная (см. § 11.2.1, § 11.2.4).

8.2.7. Псевдонимы пространств имен

Короткие названия пространств имен могут войти в конфликт друг с другом:

```
namespace A {                    // короткое имя рано или поздно приведет к конфликту
    //...
}

A::String s1 = "Григ";
A::String s2 = "Нильсен";
```

Однако, длинные названия пространств имен непрактичны при написании реального кода:

```
namespace American_Telephone_and_Telegraph {          // слишком длинное
    //...
}

American_Telephone_and_Telegraph::String s3 = "Григ";
American_Telephone_and_Telegraph::String s4 = "Нильсен";
```

Эту дилемму можно решить путем создания короткого псевдонима длинного названия пространства имен:

```
// воспользуемся псевдонимами для сокращения имен:

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "Григ";
ATT::String s4 = "Нильсен";
```

Псевдонимы, кроме того, позволяют пользователю ссылаться на «библиотеку» и в одном единственном объявлении определять реально используемую библиотеку. Например:

```
namespace Lib = Foundation_library_v2r11;

// ...

Lib::set s;
Lib::String s5 = "Сибелиус";
```

Это может значительно облегчить проблему смены версии библиотеки. Пользуясь ***Lib*** вместо ***Foundation_library_v2r11***, можно перейти к версии «v3r02», изменив инициализацию псевдонима ***Lib*** и перекомпилировав программу. Во время компиляции будут обнаружены несоответствия на уровне исходного текста. С другой стороны, злоупотребление псевдонимами может привести к некоторой путанице.

8.2.8. Объединение пространств имен

Порой возникает потребность в создании интерфейса из набора существующих интерфейсов. Например:

```
namespace His_string {
        class String { /* ... */ };
        String operator+ (const String&, const String&);
        String operator+ (const String&, const char*);
        void fill (char);
        // ...
}

namespace Her_vector {
        template<class T> class Vector { /* ... */ };
}

namespace My_lib {
        using namespace His_string;
        using namespace Her_vector;
        void my_fct (String&);
}
```

После этого мы можем писать программу в терминах ***My_lib***:

```
void f()
{
        My_lib::String s = "Байрон";          // My_lib::His_string::String
        // ...
}

using namespace My_lib;
```

```
void g(Vector<String>& vs)
{
        //...
        my_fct(vs[6]);
        //...
}
```

Если имя с явным квалификатором (типа ***My_lib::String***) не объявлено в указанном пространстве имен, компилятор осуществляет поиск в пространствах имен, упомянутых в директивах ***using*** (например, ***His_string***).

Только если нам требуется определить что-нибудь, мы должны знать имя реального пространства имен:

```
void My_lib::fill(char c)                    // ошибка: в My_lib не объявлена fill()
{
        //...
}
void His_string::fill(char c)                // правильно: fill() объявлена в His_string
{
        //...
}
void My_lib::my_fct(String & v)              // правильно
//String рассматривается как My_lib::String, то есть His_string::String
{
        //...
}
```

В идеале, пространство имен должно:

[1] выражать логически связанный набор средств;

[2] препятствовать доступу пользователей к ненужным им средствам;

[3] не требовать значительных дополнительных усилий при использовании.

Методы объединения, изложенные здесь и в следующих разделах, совместно с механизмом ***#include*** (§ 9.2.1), обеспечивают необходимую поддержку для реализации этих требований.

8.2.8.1. Отбор

Иногда нам требуется доступ только к нескольким именам из пространства. Мы можем реализовать это, написав определение пространства имен, содержащее только требуемые имена. Например, мы могли бы написать объявление версии ***His_String***, содержащее только ***String*** из ***His_string*** и оператор конкатенации:

```
namespace His_string {                       // только часть His_string
        class String { /* ... */ };
        String operator+(const String&, const String&);
        String operator+(const String&, const char*);
}
```

Но если я не являюсь разработчиком или сопровождающим ***His_string***, такой подход легко может привести к путанице. Изменения в «настоящем» пространстве ***His_string*** не будут отражены в данном объявлении. Отбор средств из пространства имен более явно производится при помощи ***using-объявления***:

```
namespace My_string {
        using His_string::String;
        using His_string::operator+;          // имеется в виду любой + из His_string
}
```

using-объявление помещает любое объявление с указанным именем в область видимости. В частности, единственное *using-объявление* делает видимыми все версии перегруженной функции.

В нашем случае если разработчик ***His_string*** добавит к ***String*** функцию-член или введет перегруженную версию оператора конкатенации, эти изменения автоматически станут доступными пользователям ***My_string***. С другой стороны, если средство удалено из ***His_string*** или изменился интерфейс, использование ***My_string***, ставшее некорректным, будет обнаружено компилятором (см. также § 15.2.2).

8.2.8.2. Объединение и отбор

Комбинация объединения (при помощи *using-директив*) и отбора (при помощи *using-объявлений*) обеспечивает гибкость, требуемую для большинства реальных задач. Используя эти механизмы, мы можем обеспечить доступ ко множеству средств таким образом, чтобы разрешать конфликты имен и неоднозначности, возникающие из-за объединения. Например:

```
namespace His_lib {
        class String { /* ... */ };
        template<class T> class Vector { /* ... */ };
        // ...
}

namespace Her_lib {
        template<class T> class Vector { /* ... */ };
        class String { /* ... */ };
        // ...
}

namespace My_lib {
        using namespace His_lib;              // все из His_lib
        using namespace Her_lib;              // все из Her_lib

        using His_lib::String;                // разрешение возможных
                                              // конфликтов в пользу His_lib
        using Her_lib::Vector;                // разрешение возможных
                                              // конфликтов в пользу Her_lib

        template<class T> class List { /* ... */ };  // и т. д.
        // ...
}
```

Имена, явно объявленные в пространстве имен (включая имена, объявленные с помощью *using-объявлений*), имеют приоритет по отношению к именам, сделанным доступными при помощи *using-директив* (см. также § B.10.1). Следовательно, пользователь ***My_lib*** увидит, что конфликты имен ***String*** и ***Vector*** будут решены в пользу ***His_lib***::***String*** и ***Her_lib***::***Vector***. Кроме того, ***My_lib***::***List*** будет использоваться по умолчанию, независимо от того, имеется ли ***List в His_lib*** или ***Her_lib***.

Как правило, я предпочитаю не менять имя при включении его в новое пространство имен. В этом случае мне не надо помнить два различных имени для одного объекта. Однако иногда требуется новое имя (или просто так удобнее). Например:

```
namespace Lib2 {
using namespace His_lib;                    // все из His_lib
using namespace Her_lib;                    // все из Her_lib

using His_lib::String;                      // разрешение возможных
                                            // конфликтов в пользу His_lib
using Her_lib::Vector;                      // разрешение возможных
                                            // конфликтов в пользуHer_lib

typedef Her_lib::String Her_string;         // переименование

template<class T> class His_vec             // «переименование»
    : public His_lib::Vector<T> { /* ... */ };

template<class T> class List { /* ... */ }; // и т. д.
// ...
}
```

В языке нет специальных средств для переименования. Вместо этого используется обычный механизм определения нового объекта.

8.2.9. Пространства имен и старый код

Миллионы строк кода на C и C++ используют глобальные имена и существующие библиотеки. Как мы можем воспользоваться пространствами имен для уменьшения проблем в таком коде? Не всегда возможно переписать код. К счастью, библиотеками C можно пользоваться так, как будто они были определены в пространстве имен. Однако, этого нельзя сделать с библиотеками, написанными на C++ (§ 9.2.4). С другой стороны, пространства имен реализованы таким образом, что их легко добавить с минимальными затратами в старые программы на C++.

8.2.9.1. Пространства имен и C

Рассмотрим первую каноническую программу на C:

```
#include <stdio.h>

int main()
{
    printf("Здравствуй, мир!\n");
}
```

Далеко не лучшей идеей было бы изменение программы, так же как и создание специальных версий стандартных библиотек. Поэтому правила языка для пространств имен разработаны таким образом, чтобы можно было программу, написанную без использования пространств имен, сравнительно легко переделать в более явно структурированную версию с применением пространств имен. В действительности, пример программы калькулятора (§ 6.1) является воплощением такого подхода.

Ключом к решению является ***using-директива***. Например, объявления стандартных средств ввода/вывода C в заголовочном файле ***stdio.h*** помещены в пространство имен следующим образом:

```
// stdio.h:

    namespace std {
        int printf(const char* ...);
        // ...
    }
    using namespace std;
```

Это обеспечивает совместимость сверху вниз. Для тех, кто не желает присутствия неявно доступных имен, определен новый заголовочный файл ***cstdio***:

```
// cstdio:

namespace std {
    int printf(const char* ...);
    // ...
}
```

Создатели стандартной библиотеки C++, которые хотят избежать повторных объявлений, конечно же, определят ***stdio.h*** путем включения ***cstdio***:

```
// stdio.h:

    #include<cstdio>
    using std::printf;
    // ...
```

Я рассматриваю глобальные ***using-директивы*** в основном как средство переноса старого кода. Большая часть кода со ссылками на имена из других пространств имен может быть выражена более ясно при помощи явных квалификаторов и ***using-объявлений***.

Связь между пространствами имен и компоновкой описана в § 9.2.4.

8.2.9.2. Пространства имен и перегрузка

Механизм перегрузки (§ 7.4) работает сквозь границы пространств имен. Это является важным моментом для осуществления перехода от существующих библиотек к использованию пространств имен с минимальными изменениями в исходном коде. Например:

```
// старый A.h:

    void f(int);
    // ...

// старый B.h:

    void f(char);
    // ...

// старый user.c:

    #include "A.h"
    #include "B.h"

    void g()
```

```
{
    f('a');        // вызывается f() из B.h
}
```

Эта программа может быть модифицирована с использованием пространств имен без изменения содержательной части кода:

```
// новый A.h:

namespace A {
    void f(int);
    // ...
}

// новый B.h:

namespace B {
    void f(char);
    // ...
}

// новый user.c:

#include "A.h"
#include "B.h"

using namespace A;
using namespace B;

void g()
{
    f('a');        // вызывается f() из B.h
}
```

Если бы мы хотели оставить ***user.c*** совершенно без изменений, мы могли бы поместить ***using-директивы*** в заголовочные файлы.

8.2.9.3. Пространства имен открыты

Пространства имен открыты; это означает, что вы можете добавлять к ним новые имена в нескольких объявлениях. Например:

```
namespace A {
    int f();       // теперь f() является членом A
}

namespace A {
    int g();       // теперь и f(), и g() являются членами A
}
```

Мы можем таким образом поддерживать большие фрагменты программ при помощи единственного пространства имен, подобно тому, как старые библиотеки и приложения «жили» в одном глобальном пространстве. Для реализации этого мы должны «распределить» определение пространства имен по нескольким заголовочным файлам и нескольким файлам исходного кода. Как показано на примере калькулятора (§ 8.2.4), открытость пространств имен позволяет нам реализовывать различные интерфейсы для различных категорий пользователей, открывая для них толь-

ко необходимые фрагменты пространства имен. Открытость также полезна для перехода от старого кода. Например, фрагмент

```
// мой заголовочный файл:
    void f(),           // моя функция
    //...
    #include<stdio h>
    int g (),           // моя функция
    //..
```

можно переписать, не меняя порядок объявлений:

```
// мой заголовочный файл:
    namespace Mine {
        void f(),       // моя функция
        //  ..
    }
    #include<stdio h>
    namespace Mine {
        int g (),       // моя функция
        //...
    }
```

При написании нового кода я предпочитаю использовать несколько относительно небольших пространств имен (§ 8.2.8), а не помещать большие фрагменты кода в единственное пространство имен. Однако, при модификации больших фрагментов с целью использования пространств имен это часто не слишком удобно.

При определении предварительно объявленного члена пространства имен безопасней пользоваться синтаксисом ***Mine.***: по сравнению с повторным открытием пространства ***Mine***. Например:

```
void Mine ff ()         // ошибка: ff() не объявлена в Mine
{
    //..
}
```

Компилятор обнаружит эту ошибку. Однако, ввиду того, что новые функции могут быть объявлены в пространстве имен, компилятор не сможет обнаружить аналогичную ошибку при повторном открытии пространства:

```
namespace Mine {    // повторное открытие Mine с целью определения функций
    void ff ()      // проблема: в Mine нет объявления ff(),
                    // этим определением в Mine добавляется ff()
    {
        //...
    }
    //...
}
```

Компилятор не может знать, что вы не хотели завести новую функцию *ff* ().

Псевдонимы пространства имен могут использоваться для уточнения имени при определении. Тем не менее, они не могут применяться для повторного открытия пространства имен.

8.3. Исключения

Когда программа конструируется из раздельных модулей, и, особенно, когда эти модули находятся в независимо разработанных библиотеках, обработка ошибок должна быть разделена на две части:

[1] генерация информации о возникновении ошибочной ситуации, которая не может быть разрешена локально;

[2] обработка ошибок, обнаруженных в других местах.

Автор библиотеки может обнаружить ошибки во время выполнения, но, как правило, не имеет представления о том, что делать в этом случае. Пользователь библиотеки может знать, как поступить в случае возникновения ошибок, но не в состоянии их обнаружить (если бы он мог — они бы обрабатывались в коде пользователя, а их поиск не перепоручался бы библиотеке).

В примере с калькулятором мы обошли эту проблему за счет того, что создали всю программу как единое целое. Благодаря этому мы смогли реализовать обработку ошибок, как часть общей системы. Однако, после того, как мы разделили логические части калькулятора на различные пространства имен, мы видим, что каждое пространство имен зависит от пространства ***Error*** (§ 8.2.2) и что обработка ошибок в ***Error*** подразумевает, что каждый модуль ведет себя определенным образом после возникновения ошибки. Предположим, мы не можем позволить себе разрабатывать калькулятор как единое целое и не хотим жесткой связи между ***Error*** и другими модулями. Наоборот, пусть при написании синтаксического анализатора (в частности) не было информации о том, как драйвер будет обрабатывать ошибки.

Несмотря на простоту, в ***error***() реализована определенная стратегия обработки ошибок:

```
namespace Error {
        int no_of_errors;

        double error (const char* s)
        {
                std::cerr << "ошибка: " << s << '\n';
                no_of_errors++;
                return 1;
        }
}
```

Функция ***error***() выводит сообщение об ошибке, возвращает некоторое значение, которое позволяет вызывающему модулю продолжить вычисления, и сохраняет информацию о состоянии. Существенным моментом является то, что каждая часть программы знает о существовании ***error***(), о том как ее вызывать, и что от нее ожидать. Это слишком большие требования к программе, создаваемой из независимо разработанных библиотек.

Исключения являются средствами C++ для отделения генерации информации о возникновении ошибки от ее обработки. В этом разделе кратко описаны исключения в контексте их использования в примере с калькулятором. В главе 14 приводится более детальное обсуждение исключений и их использования.

8.3.1. throw и catch

Понятие *исключения* (exception) введено для генерации в системе сообщения об ошибке. Например:

```
struct Range_error {
        int i;
        Range_error(int ii) { i = ii; }          // конструктор (§ 2.5.2, § 10.2.3)
};

char to_char(int i)
{
        if (i< numeric_limits< char>::min() ||
                numeric_limits< char>::max()< i)    // см. § 22.2
                throw Range_error(i);
        return i;
}
```

Функция **to_char()** либо возвращает **char** по числовому значению **i**, либо генерирует исключение **Range_error**. Основная идея состоит в том, что функция, обнаружившая проблему, которую она не знает как решать, генерирует (**throw**) исключение в надежде, что вызывающий (прямо или косвенно) модуль знает, что делать в этой ситуации. Функция, которая собирается обрабатывать ошибку, может объявить, что она будет *перехватывать* (**catch**) исключения данного типа. Например, для вызова **to_char()** и перехвата исключений, которые она может вызвать, мы могли бы написать:

```
void g(int i)
{
        try {
                char c = to_char(i);
                //...
        }
        catch (Range_error) {
                cerr << "проблема\n";
        }
}
```

Конструкция

```
        catch (/*...*/) {
                //...
        }
```

называется *обработчиком исключений*. Она может использоваться только сразу после блока, начинающегося с ключевого слова **try**, или сразу после другого обработчика; **catch** также является ключевым словом. В скобках находится объявление, которое используется аналогично объявлению аргументов функции. То есть, оно указывает тип объектов, которые могут быть перехвачены этим обработчиком, и (необязательно) присваивает имена перехватываемым объектам. Например, если бы мы хотели узнать значение **Range_error**, мы могли бы задать имя аргумента **catch** точно также, как мы указываем имена аргументам функций. Например:

```
void h (int i)
{
        try {
                char c = to_char (i);
                //...
        }
        catch (Range_error x) {
                cerr << "проблема: to_char (" << x.i << ")\n";
        }
}
```

Если код в ***try-блоке***, или код, вызываемый из него, генерирует исключение, будут проверяться обработчики этого блока ***try***. Если сгенерированное исключение имеет тип, указанный в одном из обработчиков, будет выполнен этот обработчик. В противном случае обработчики игнорируются и ***try-блок*** ведет себя как обыкновенный блок. Если исключение сгенерированно и ни один из ***try-блоков*** не перехватил его, выполнение программы прекращается (§ 14.7).

Обработка исключений в C++ в основном является методом передачи управления специальному коду в вызывающем модуле. Программисты на C могут рассматривать механизм обработки исключений как улучшенную замену ***setjmp/longjmp*** (§ 16.1.2). Важные связи между обработкой исключений и классами описываются в главе 14.

8.3.2. Выбор исключений

Как правило, в каждой программе существует несколько возможных типов ошибок на этапе выполнения. Такие ошибки можно распределить между исключениями с различными именами. Я предпочитаю определять типы исключений только ради обработки исключений. Это сводит к минимуму путаницу, связанную с их назначением. В частности, я никогда не пользуюсь встроенными типами, такими как ***int***, для описания исключения. В большой программе нет эффективного способа разделения исключений, обрабатывающих ***int***. Я не могу быть уверенным, что другие обработчики не смешаются с моими.

Наш калькулятор (§ 6.1) должен обрабатывать две ошибки времени выполнения: синтаксические ошибки и попытку деления на ноль. Нет необходимости передавать какое-либо значение обработчику из кода, обнаружившего попытку деления на ноль, поэтому деление на ноль может быть представлено простым пустым типом:

```
struct Zero_divide { };
```

С другой стороны, обработчик скорее всего предпочел бы получать информацию о том, какого вида встретилась синтаксическая ошибка. В этом случае мы передаем строку:

```
struct Syntax_error {
        const char* p;
        Syntax_error (const char* q) { p = q; }
};
```

Для удобства я добавил в структуру конструктор (§ 2.5.2, § 10.2.3).

Пользователь синтаксического анализатора может разделить обработку этих двух исключений, добавив обработчики обоих исключений после блока ***try***. При генерации

исключения выполнится соответствующий обработчик. По завершении обработки исключения управление передается за конец списка обработчиков:

```
try {
        //...
        expr(false);
        // мы попадем сюда в том и только в том случае,
        // если expr() не возбудит исключения
        //...
}
catch (Syntax_error) {
        // обработка синтаксической ошибки
}
catch (Zero_divide) {
        // обработка деления на ноль
}
// мы попадем сюда, если expr не сгенерировал исключения, либо если были
// сгенерированы исключения Syntax_error или Zero_divide
// (и их обработчики не сгенерировали исключения
// или некоторым другим способом не изменили потока управления).
```

Список обработчиков напоминает инструкцию ***switch***, с той лишь разницей, что не требуется инструкции ***break***. Синтаксис списка обработчиков отличается от списков ***case*** отчасти этим, а отчасти тем, что каждый обработчик является отдельной областью видимости (§ 4.9.4).

Функции не требуется перехватывать все возможные исключения. Например, предыдущий блок ***try*** не пытается перехватить исключения, которые потенциально могут возникнуть при выполнении операций ввода. Эти исключения просто «передаются наверх» в поисках вызывающей функции с соответствующим обработчиком.

С точки зрения языка считается, что исключение обработано сразу после входа в его обработчик. Это сделано для того, чтобы любые исключения, сгенерированные во время выполнения обработчика, обрабатывались функцией с *try-блоком*, вызвавшим исключение. Следующий пример не приведет к бесконечному циклу:

```
class Input_overflow { /* ... */ };

void f()
{
        try {
                //...
        }
        catch (Input_overflow) {
                //...
                throw Input_overflow();
        }
}
```

Обработчики исключений могут быть вложенными. Например:

```
class XXII { /* ... */ };

void f()
```

```
{
        //...
        try {
                //...
        }
        catch (XXII) {
                try {
                        // сложный код
                }
                catch (XXII) {
                        // ошибка в сложном коде
                }
        }
        //...
}
```

Однако такая вложенность редко встречается в коде, написанном человеком, и, как правило, является признаком плохого стиля.

8.3.3. Исключения в программе калькулятора

Имея базовый механизм обработки исключений, мы можем переделать пример с калькулятором из § 6.1, выделив обработку ошибок, возникающих на этапе выполнения из основной логики калькулятора. Это приведет к организации программы, которая более напоминает ту, что встречается в приложениях, сконструированных из отдельных, слабо зависимых частей.

Во-первых, можно исключить *error* (). Теперь функция синтаксического анализатора будет знать только типы, используемые для сообщения об ошибках:

```
namespace Error {
        int no_of_errors;

                struct Zero_divide {
                        Zero_divide() { no_of_errors++; }
        };

        struct Syntax_error {
                const char* p;
                Syntax_error(const char* q) { p = q; no_of_errors++; }
        };
}
```

Синтаксический анализатор обнаруживает три синтаксические ошибки:

```
Lexer::Token_value Lexer::get_token ()
{
        using namespace std;                    // для доступа к input, isalpha() и т. д. (§ 6.1.7)
                //...
                default:                        // NAME, NAME= или ошибка
                        if (isalpha (ch)) {
                                string_value = ch;
                                while (input->get (ch) && isalnum (ch)) string_value.push_back (ch);
                                input -> putback (ch);
```

```
                return curr_tok=NAME;
            }
            throw Error::Syntax_error("неверная лексема");
        }
}
double Parser::prim(bool get)
{
    //...
    case Lexer::LP:
    {   double e = expr(true);
        if (curr_tok != Lexer::RP) throw
            Error::Syntax_error("ожидалась ')'");
        get_token();                                    // пропустить скобки ')'
        return e;
    }
    case Lexer::END:
        return 1;
    default:
        throw Error::Syntax_error("ожидалось первичное выражение");
    }
}
```

После обнаружения синтаксической ошибки используется ***throw*** для передачи управления обработчику, указанному в вызывающей (прямо или косвенно) функции. Кроме того, ***throw*** передает обработчику значение. Например,

```
throw Error::Syntax_error("ожидалось первичное выражение");
```

передает обработчику объект ***Syntax_error***, содержащий указатель на строку ***"ожидалось первичное выражение"***.

Сообщение о делении на ноль не требует передачи данных:

```
double Parser::term(bool get)           // умножение и деление
{
    //...
    case Lexer::DIV;
        if (double d = prim(true)) {
            Left /= d;
            break;
        }
        throw Error::Zero_divide();
    //...
}
```

Теперь можно определить драйвер, который обрабатывает исключения ***Zero_divide*** и ***Syntax_error***. Например:

```
int main(int argc, char* argv[])
{
    //...
    while (*input) {
        try {
```

```
            Lexer::get_token ();
            if (Lexer::curr_tok == Lexer::END) break;
            if (Lexer::curr_tok == Lexer::PRINT) continue;
            cout << Parser::expr (false) << '\n';
        }
        catch (Error::Zero_divide) {
            cerr << "попытка деления на ноль\n";
            if ( Lexer::curr_tok != Lexer::PRINT) skip ();
        }
        catch (Error::Syntax_error e) {
            cerr << "синтаксическая ошибка: " << e.p << "\n";
            if (Lexer::curr_tok != Lexer::PRINT) skip ();
        }
    }

    if (input!=&cin) delete input;

    return no_of_errors;
}
```

Во всех случаях, кроме ошибки, произошедшей в конце выражения, ограниченного лексемой ***PRINT*** (т. е. концом строки или точкой с запятой) ***main*** () вызывает функцию восстановления ***skip*** (). Функция ***skip*** () пытается привести синтаксический анализатор в нормальное состояние после ошибки, пропуская лексемы до тех пор, пока не встретит конец строки или точку с запятой. Функции ***skip*** () и ***input*** являются очевидными кандидатами на включение в пространство имен ***Driver***:

```
namespace Driver {
    std::istream* input;
    void skip ();
}
void Driver::skip ()
{
    while (*input) {
        char ch;
        input->get (ch);

        switch (ch) {
        case '\n':
        case ';':
            return;
        }
    }
}
```

Код для функции ***skip*** () намеренно написан на более низком уровне абстракции, чем синтаксический анализатор. Это сделано для того, чтобы во время обработки исключений синтаксического анализатора не быть прерванным новыми исключениями от синтаксического анализатора.

Я сохранил подсчет числа ошибок и возврат этого числа из программы. Не бесполезно знать, встречались ли ошибки во время выполнения программы, даже если она смогла восстановиться после них.

Я не поместил ***main*** () в пространство ***Driver***. Глобальное имя ***main*** () является «точкой отсчета» программы (§ 3.2); ***main*** () в отдельном пространстве имен не имеет особого смысла. В программе реального размера большая часть кода из ***main*** () переместится в отдельную функцию в пространстве имен ***Driver***.

8.3.3.1. Альтернативные стратегии обработки ошибок

Исходный код обработки ошибок был короче и элегантней, чем версия с использованием исключений. Однако, эта элегантность достигалась тесной зависимостью между частями программы. Такой подход плохо масштабируется при переходе к программам, составляемым из независимо разработанных библиотек.

Мы могли бы рассмотреть вариант отказа от отдельной функции обработки ошибок ***skip*** () путем введения переменной состояния в ***main*** (). Например:

```
int main (int argc, char* argv[])    // пример плохого стиля
{
    //...
    bool in_error = false;

    while (*Driver::input) {
        try {
            Lexer::get_token ();
            if (Lexer::curr_tok == Lexer::END) break;
            if (Lexer::curr_tok == Lexer::PRINT) {
                in_error = false;
                continue;
            }
            if (in_error == false) cout << Parser::expr (false) << '\n';
        }
        catch (Error::Zero_divide) {
            cerr << "попытка деления на ноль\n";
            in_error = true;  ++no_of_errors;
        }
        catch (Error::Syntax_error e) {
            cerr << "синтаксическая ошибка: " << e.p << "\n";
            in_error = true;  ++no_of_errors;
        }
    }

    if (Driver::input!=&Std::cin) delete Driver::input;

    return Driver::no_of_errors;

}
```

Я считаю это плохой идеей по нескольким причинам:

[1] Переменные состояния являются распространенным источником ошибок и путаницы, особенно если они оказывают влияние на значительные фрагменты программы. В частности, я считаю, что вариант ***main*** () с использованием ***in_error*** менее читабелен, чем версия с функцией ***skip*** ().

[2] Как правило, разделение кода обработки ошибок и «нормального» кода является хорошей стратегий.

[3] Опасно писать обработку ошибок на том же уровне абстракции, что и код, который вызвал ошибку; код обработчика может повторить ту же самую ошибку, которая вызвала передачу ему управления. Выяснение того, как это может произойти в варианте ***main*** () с ***in_error***, я оставляю в качестве упражнения (§ 8.5[7]).

[4] Требуется больше модификаций при добавлении кода обработки ошибок к «нормальному» коду, чем при добавлении отдельных процедур обработки ошибок.

Целью обработки исключений является решение нелокальных по своей природе проблем. Если проблема может быть решена локально, почти всегда так ее и следует решать. Например, нет необходимости в использовании исключений для обработки ситуации, когда передано слишком много аргументов:

```
int main (int argc, char* argv[])
{
        using namespace std;
        using namespace Driver;

        switch (argc) {
        case 1:                 // чтение из стандартного ввода
                input = &cin;
                break;
        case 2:                 // чтение строки аргументов
                input = new istringstream (arg[1]);
                break;
        default:
                cerr << "слишком много аргументов\n";
                return 1;
        }

        // то же, что и раньше
}
```

Обсуждение исключений продолжается в главе 14.

8.4. Советы

[1] Пользуйтесь пространствами имен для выражения логической структуры; § 8.2.

[2] Помещайте все нелокальные имена, за исключением ***main*** (), в пространства имен; § 8.2.

[3] Проектируйте пространства имен таким образом, чтобы ими было удобно пользоваться, и исключайте возможность случайного доступа к пространствам, не имеющим отношения к вашей задаче; § 8.2.4.

[4] Избегайте слишком коротких названий пространств имен; § 8.2.7.

[5] Если требуется, пользуйтесь псевдонимами для сокращения имен пространств; § 8.2.7.

[6] Ваши пространства имен не должны усложнять жизнь их пользователей (в частности, из-за сложностей с обозначениями); § 8.2.2, § 8.2.3.

[7] Пользуйтесь формой ***Namespace::member*** при определении членов пространств имен; § 8.2.8.

[8] Пользуйтесь формой ***using namespace*** только для перенесения старого кода или в локальной области видимости; § 8.2.9.

[9] Пользуйтесь исключениями для отделения обработки ошибок от кода, выполняющего обычные действия; § 8.3.3.

[10] Для исключений применяйте типы, определяемые пользователем, а не встроенные типы; § 8.3.2.

[11] Не пользуйтесь исключениями, когда достаточно локальных управляющих структур; § 8.3.3.1.

8.5. Упражнения

1. (*2.5) Напишите модуль, реализующий двусвязный список строк ***string*** в стиле модуля ***Stack*** из § 2.4. Поэкспериментируйте с ним, создав список названий языков программирования. Реализуйте функцию ***sort*** () для этого списка и функцию, которая меняет в нем порядок строк на противоположный.
2. (*2) Возьмите какую-нибудь небольшую программу, в которой используется по крайней мере одна библиотека, но не применяются пространства имен. Модифицируйте ее так, чтобы в ней использовалось пространство имен для этой библиотеки. Подсказка: § 8.2.9.
3. (*2) Реализуйте программу калькулятора в виде модуля в стиле § 2.4 с использованием пространств имен. Не пользуйтесь глобальными ***using***-*директивами*. Запомните допущенные вами ошибки. Предложите способы устранения таких ошибок в будущем.
4. (*1) Напишите программу, которая генерирует исключение в одной функции, и перехватывает его в другой.
5. (*2) Напишите программу, состоящую из функций, вызывающих друг друга до глубины вложенности вызовов, равной 10. Введите аргумент для каждой функции, который определял бы, на какой глубине вложенности сгенерировано исключение. Пусть ***main*** () перехватывает эти исключения и печатает информацию о перехваченном исключении. Не забудьте случай, когда исключение перехватывается в функции, которая его сгенерировала .
6. (*2) Измените программу из § 8.5[5] таким образом, чтобы можно было измерить, отличаются ли затраты на перехват исключений в зависимости от того, где в стеке функций было сгенерировано исключение. Добавьте строковый объект в каждую функцию и произведите измерения снова.
7. (*1) Найдите ошибку в первой версии ***main*** () в § 8.3.3.1.
8. (*2) Напишите функцию, которая либо возвращает значение, либо генерирует исключение, в зависимости от значения аргумента. Измерьте разницу во времени выполнения для этих случаев.
9. (*2) Перепишите версию калькулятора из § 8.5[3] с использованием исключений. Запомните допущенные вами ошибки. Предложите способы устранения таких ошибок в будущем.
10. (*2.5) Напишите функции ***plus*** (), ***minus*** (), ***multiply*** () и ***divide*** (), проверяющие возможные переполнения (в обе стороны) и генерирующие исключения при возникновении таких ошибок.
11. (*2) Перепишите программу калькулятора с использованием функций из § 8.5[10].

9

Исходные файлы и программы

Форма должна следовать за функцией.
— Ле Корбузьер

Раздельная компиляция — компоновка — заголовочные файлы — заголовочные файлы стандартной библиотеки — правило одного определения (ODR) — компоновка кода, написанного не на C++ — компоновка и указатели на функции — использование заголовочных файлов для выражения модульности — единственный заголовочный файл — несколько заголовочных файлов — стражи включения — программы — советы — упражнения

9.1. Раздельная компиляция

Файл является традиционной единицей хранения информации в файловой системе и не менее традиционной единицей компиляции. Существуют системы, которые не хранят, не компилируют и не представляют программисту программы на C++ в виде набора файлов. Тем не менее, обсуждение в этой главе сосредоточено на системах, использующих традиционные файлы.

Как правило, невозможно хранить законченную программу в одном файле. В частности, код стандартных библиотек и операционной системы не предоставляется в виде исходных текстов в качестве части пользовательской программы. Даже хранение всего пользовательского кода в единственном файле для приложения реального размера и непрактично, и неудобно. Разбиение программы на файлы помогает подчеркнуть ее логическую структуру, облегчает ее понимание другими и позволяет компилятору обеспечить эту логическую структуру. Когда единицей компиляции является файл, весь он должен быть заново перекомпилирован при внесении изменений (не зависимо от того, насколько они малы) в него или в другой файл, от которого он зависит. Даже в случае программы скромных размеров количество времени, потраченное на перекомпиляцию, может быть значительно снижено за счет разбиения программы на файлы подходящего размера.

Пользователь предоставляет компилятору *исходный файл*. Сначала производится обработка файла препроцессором; то есть делаются макроподстановки (§ 7.8) и выполняются директивы ***#include***, вставляющие код из заголовочных файлов (§ 2.4.1, § 9.2.1). Результат обработки препроцессором исходного файла называется *единицей трансляции*. Она и является тем, над чем работает компилятор, и что, собственно,

описывают правила языка C++. В этой книге я провожу различие между исходным файлом и единицей трансляции только там, где это необходимо, чтобы отличать то, что видит программист, от того, что получает компилятор.

Для того чтобы сделать возможной раздельную компиляцию, программист должен предоставить объявления, дающие информацию о типах, необходимую для анализа единицы трансляции отдельно от остальной части программы. Объявления в программе, состоящей из нескольких раздельно компилируемых частей, должны быть согласованы абсолютно так же, как и в программе, состоящей из единственного исходного файла. В вашей системе должны быть средства, помогающие убедиться в этом. В частности, компоновщик может обнаружить много несогласованностей различных типов. *Компоновщик* (linker) является программой, которая связывает вместе раздельно откомпилированные части. Компоновщик иногда называют (и неправильно) *загрузчиком*. Компоновка[1] может быть полностью завершена до того, как программа запускается на выполнение. С другой стороны, новый код может быть добавлен к программе («динамически скомпонован») уже после загрузки.

Организацию программы в виде набора исходных файлов обычно называют *физической структурой* программы. Физическое разбиение программы на различные файлы должно определяться, исходя из логической структуры программы. Те же самые соображения о зависимости, которые помогают при разделении программы на пространства имен, относятся и к разбиению на исходные файлы. Однако, логическая и физическая структуры программы не обязаны быть идентичными. Например, может оказаться полезным использовать несколько файлов для хранения функций из одного пространства имен, хранить набор определений пространств имен в одном файле или поместить определение пространства в нескольких файлах (§ 8.2.4).

Сначала рассмотрим некоторые технические детали, имеющие отношение к компоновке, и затем обсудим два способа разбиения программы калькулятора (§ 6.1, § 8.2) на файлы.

9.2. Компоновка

Имена функций, классов, шаблонов, переменных, пространств имен и перечислений должны быть согласованы во всех единицах компиляции, если только эти имена явно не определены как локальные.

Задачей программиста является обеспечение того, чтобы каждое пространство имен, класс, функция и т. д. были правильно объявлены в каждой единице трансляции, в которой они используются, и чтобы все объявления одного и того же объекта были согласованы. Например, рассмотрим два файла:

```
//file1.c:
    int x = 1;
    int f() { /* некоторые действия /* }

//file2.c:
    extern int x;
    int f();
    void g() { x = f(); }
```

[1] Компоновку (linking или linkage) по-русски часто также называют «связыванием» или «редактированием связей».— *Примеч. ред.*

Переменная ***x*** и функция ***f***(), используемые в ***g***() из ***file2.c***, определены в ***file1.c***. Ключевое слово ***extern*** означает, что объявление ***x*** в ***file2.c*** является (только) объявлением, а не определением (§ 4.9). Если бы ***x*** была инициализирована, ***extern*** было бы проигнорировано, потому что объявление с инициализацией всегда является определением. Любой объект должен быть определен в программе только один раз. Он может быть объявлен много раз, но типы должны совпадать. Например:

```
//file1.c:
    int x = 1;
    int b = 1;
    extern int c;

//file2.c:
    int x,              // означает int x = 0;
    extern double b;
    extern int c;
```

В примере имеется три ошибки: ***x*** определена дважды, ***b*** объявлена дважды с различными типами и ***c*** дважды объявлена, но не определена. Ошибки такого типа (ошибки компоновки) не могут быть обнаружены компилятором, который в каждый момент времени рассматривает только один файл. Однако, большинство таких ошибок выявляются компоновщиком. Обратите внимание, что переменная, определенная без инициализации в глобальной области видимости или в пространстве имен, инициализируется по умолчанию. Этого не происходит с локальными переменными (§ 4.9.5, § 10.4.2) или с объектами, создаваемыми в свободной памяти (§ 6.2.6). Например, следующий фрагмент программы содержит две ошибки:

```
//file1.c:
    int x;
    int f() { return x; }

//file2.c:
    int x;
    int g() { return f(); }
```

Вызов ***f***() в ***file2.c*** является ошибкой, потому что ***f***() не была объявлена в ***file2.c***. Программа не будет скомпонована также и потому, что переменная ***x*** определена дважды. Отметим, что эти ошибки не являются таковыми в C (§ B.2.2).

Если имеется имя, которое может быть использовано в единице трансляции, отличной от той, в которой оно было определено, то говорят, что имеет место *внешняя компоновка* (external linkage). Все имена в предыдущих примерах компоновались внешним образом. Про имя, на которое можно ссылаться лишь в той единице трансляции, в которой оно определено, говорят, что оно *компонуется внутренним образом*.

Встроенная (inline) функция (§ 7.1.1, § 10.2.9) должна быть определена — идентичными определениями (§ 9.2.3) — в каждой единице трансляции, в которой она используется. Следовательно, следующий пример является не просто образцом плохого вкуса — он вообще недопустим:

```
//file1.c:
    inline int f(int i) { return i; }
```

```
//file2.c:
    inline int f(int i) { return i+1; }
```

К сожалению, эту ошибку трудно обнаружить любой реализации компилятора, и следующая комбинация внешней компоновки и встроенной функции, хотя и выглядит совершенно логично, запрещена, чтобы облегчить жизнь авторам компиляторов:

```
//file1.c:
    extern inline int g(int i);
    int h(int i) { return g(i); }      // ошибка: g() не определена
                                       // в этой единице трансляции

//file2.c:
    extern inline int g(int i) { return i+1; }
```

По умолчанию ***const*** (§ 5.4) и ***typedef*** (§ 4.9.7) подразумевают внутреннюю компоновку. Следовательно, следующий пример допустим (хотя потенциально может привести к ошибкам):

```
//file1.c:
    typedef int T;
    const int x = 7;

//file2.c:
    typedef void T;
    const int x = 8;
```

Глобальные переменные, которые являются локальными для одной единицы компиляции, являются типичным источником ошибок, и их лучше избегать. Для обеспечения согласованности лучше помещать глобальные ***const*** и ***inline*** только в заголовочные файлы (§ 9.2.1).

Можно заставить константу компоноваться внешним образом путем ее явного объявления:

```
//file1.c:
    extern const int a = 77;

//file2.c:
    extern const int a;

    void g()
    {
        cout << a << '\n';
    }
```

В этом примере *g()* выведет *77*.

Для того чтобы имена в данной единице компиляции были локальными, можно воспользоваться неименованными пространствами имен (§ 8.2.5). Эффект от применения неименованного пространства напоминает внутреннюю компоновку. Например:

```
//file1.c:
    namespace {
        class X { /* ... */ };
```

```
        void f();
        int i;
        // ...
    }
//file2.c:
    class X { /* ... */ };
    void f();
    int i;
    // ...
```

Функция *f()* в *file1.c* не та же самая, что и *f()* из *file2.c*. Иметь имя, локальное в рамках единицы компиляции, и одновременно использовать то же имя для сущности с внешней компоновкой где-нибудь еще, означает искать неприятности.

В программах на C и старых программах на C++ ключевое слово ***static*** используют (что приводит к путанице) для указания «использовать внутреннюю компоновку» (§ Б.2.3). Не пользуйтесь ***static*** кроме как внутри функций (§ 7.1.2) и классов (§ 10.2.4).

9.2.1. Заголовочные файлы

Во всех объявлениях типы одних и тех же объектов, функций, классов и т. д. должны быть согласованы. Следовательно, исходный код, обрабатываемый компилятором и затем компоновщиком, должен быть согласован. Одним, далеким от совершенства, но простым методом достижения согласованности объявлений в различных единицах трансляции является включение (***#include***) заголовочных файлов, содержащих информацию об интерфейсе в исходные файлы, в которых содержится исполняемый код и/или определения данных.

Механизм ***#include*** является средством манипулирования текстом, позволяющим собрать фрагменты исходного кода программы в одну единицу (файл) компиляции. Директива

```
#include "включаемая_компонента"
```

заменяет строку, содержащую ***#include***, на содержимое файла ***включаемая_компонента***. Содержимое этого файла должно быть исходным текстом на C++, потому что его будет обрабатывать компилятор.

Для включения библиотечных заголовочных файлов, пользуйтесь угловыми скобками < и > вместо кавычек. Например:

```
#include <iostream>        // из стандартного каталога включаемых файлов
#include "myheader.h"      // из текущего каталога
```

К сожалению, пробелы внутри < > или " " имеют значение в директиве включения:

```
#include< iostream >        // не найдет <iostream>
```

Может показаться экстравагантной перекомпиляция файла каждый раз при его включении куда-нибудь, но включаемые файлы, как правило, содержат только объявления, а не код, требующий серьезного анализа со стороны компилятора. Более того, большинство современных реализаций C++ обеспечивают некоторую форму предкомпиляции заголовочных файлов с целью минимизации затрат, требуемых для повторной компиляции одного и того же заголовочного файла.

Негласное практическое правило гласит, что заголовочный файл может содержать:

Именованные пространства имен	*namespace N { /* ... */ }*
Определения типов	*struct Point { int x, y; },*
Объявления шаблонов	*template<class T> class Z;*
Определения шаблонов	*template<class T> class V { /* ... */ };*
Объявления функций	*extern int strlen (const char*);*
Определения встроенных функций	*inline char get (char* p) { return *p++;}*
Объявления данных	*extern int a;*
Определения констант	*const float pi = 3.141593;*
Перечисления	*enum Light { red, yellow, green };*
Объявления имен	*class Matrix,*
Директивы включения	*#include <algorithm>*
Макроопределения	*#define VERSION 12*
Директивы условной компиляции	*#ifdef __cplusplus*
Комментарии	*/* проверка на конец файла */*

Это практическое правило не является требованием языка. Оно просто отражает разумный способ использования механизма *#include* для выражения физической структуры программы. С другой стороны, заголовочный файл никогда не должен содержать:

Определения обычных функций	*char get (char* p) { return *p++; }*
Определения данных	*int a;*
Определения агрегатов	*short tbl[] = { 1, 2, 3 },*
Неименованные пространства имен	*namespace { /* ... */ }*
Экспортируемые определения шаблонов	*export template<class T> f (T t) { /* ... */ }*

Заголовочные файлы обычно имеют расширение *.h*, а файлы, содержащие функции или определения данных, имеют расширение *.c*. Поэтому на них часто ссылаются как на «*.h*-файлы» и « *c*-файлы» соответственно. Также общепринятыми расширениями являются *.C*, *.cxx*, *.cpp* и *.cc*. В руководстве по вашему компилятору об этом имеется конкретная информация.

Причина рекомендации помещать определения простых констант, но не определения агрегатов в заголовочные файлы заключается в том, что компилятору трудно избежать репликации агрегатов из нескольких единиц трансляции. Кроме того, более простые случаи чаще встречаются и поэтому важнее для генерации хорошего кода.

Мудрым шагом будет не слишком усердствовать при использовании *#include*. Мои рекомендации состоят в том, чтобы включать только полные объявления и определения и делать это только в глобальной области видимости, блоках спецификации компоновки и в определениях пространств имен при конвертировании старого кода (§ 9.2.2). Как всегда, стоит избегать фокусов с макроподстановкой. Одним из моих самых нелюбимых занятий является отслеживание ошибок, вызванных макроподстановкой в нечто совершенно отличное от ожидаемого из-за наличия косвенно включенного заголовочного файла, о котором я в жизни ничего не слышал.

9.2.2. Заголовочные файлы стандартной библиотеки

Средства стандартной библиотеки представлены через набор стандартных заголовочных файлов (§ 16.12). Для указания заголовочных файлов стандартной библиотеки не требуется расширение. Они распознаются как заголовочные файлы, потому что вместо синтаксиса *#include"..."* используется форма записи *#include<...>*. Отсутствие расширения *.h* не подразумевает какого-либо конкретного способа хранения заголовочных файлов. Заголовочный файл *<map>* может храниться в виде текстового файла *map.h* в стандартном каталоге. С другой стороны, не требуется, чтобы стандартные заголовочные файлы хранились обычным способом. Конкретная реализация компилятора может воспользоваться знаниями о стандартной библиотеке для оптимизации реализации стандартной библиотеки и обработки стандартных заголовочных файлов. Например, реализация может знать о стандартной математической библиотеке (§ 22.3), сделать ее встроенной и интерпретировать *#include<cmath>* в качестве переключателя, который делает доступными стандартные математические функции, не читая при этом никакого файла.

Для каждого заголовочного файла стандартной библиотеки языка C *<X.h>*, имеется соответствующий стандартный заголовочный файл C++ *<cX>*. Например, *#include<cstdio>* обеспечивает то же, что и *#include<stdio.h>*. Типичный *stdio.h* будет выглядеть следующим образом:

```
#ifdef __cplusplus      // только для компиляторов C++ (§ 9.3.3)
namespace std {         // стандартная библиотека определена
                        // в пространстве имен std (§ 8.2.9)
extern "C" {            // функции stdio имеют компоновку, принятую в C (§ 9.2.4)
#endif

    /*  .*/
    int printf(const char* .. );
    /*  */

#ifdef __cplusplus
}
}

// ...
using std::printf;      // делает printf доступным
                        // в глобальном пространстве имен

// ...
#endif
```

Таким образом, реальные объявления (наиболее вероятно) используются совместно, но для того чтобы обеспечить совместное использование заголовочного файла C и C++ приходится задействовать средства компоновки и пространств имен.

9.2.3. Правило одного определения

Каждый конкретный класс, перечисление, шаблон и т. д. должны быть определены в программе ровно один раз.

С практической точки зрения это означает, что должно существовать ровно одно определение, скажем, класса, находящееся где-то в одном единственном файле. К сожале-

нию, правило языка не может быть таким простым. Например, определение класса может быть составлено при помощи макрорасширения (ох уж!), и в то же время определение класса может быть включено в два исходных файла при помощи директивы *#include* (§ 9.2.1). Хуже того, концепция «файла» не является частью определения языка C или C++; существуют реализации, которые не хранят программы в исходных файлах.

Следовательно, правило стандарта, говорящее о том, что должно существовать уникальное определение класса, шаблона и т. д. должно быть изложено в более сложной манере. Это правило называют «правилом одного определения» (One-Definition Rule, ODR). А именно, два определения класса, шаблона или встроенной функции приемлемы в качестве определения одной и той же сущности тогда и только тогда, когда

[1] они находятся в различных единицах трансляции;
[2] они идентичны лексема за лексемой;
[3] значение лексем одинаково в обеих единицах трансляции.

Например:

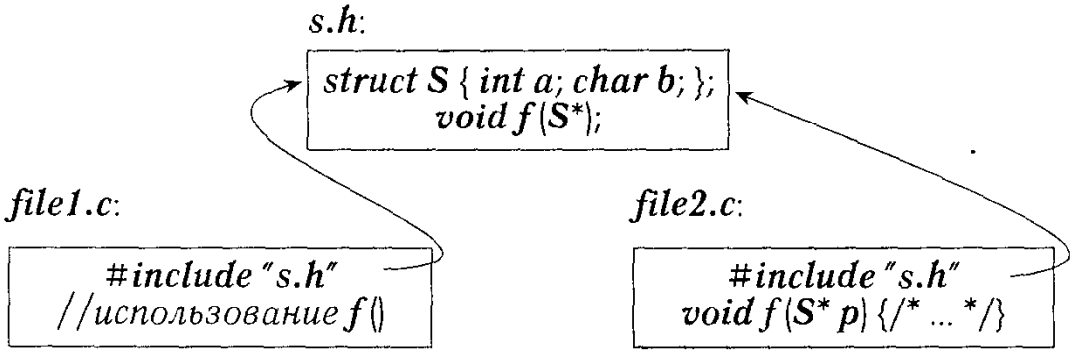
```
//file1.c:
    struct S { int a; char b; };
    void f(S*);

//file2.c:
    struct S { int a; char b; };
    void f(S* p) { /* ... */ }
```

Правило ODR говорит, что этот пример допустим, и что в обоих исходных файлах под ***S*** подразумевается одна и та же структура. Однако, неразумно записывать определение дважды подобным образом. Кто-нибудь из сопровождающих *file2.c* естественным образом предположит, что определение ***S*** в *file2.c* является единственным и будет считать себя вправе изменять его. Это может привести к трудно обнаруживаемым ошибкам.

Целью правила ODR является реализация возможности включения определения класса в различные единицы трансляции из одного общего исходного файла. Например:

```
//file s.h:
    struct S { int a; char b; };
    void f(S*);

//file1.c:
    #include "s.h"
    //использование f()

//file2.c
    #include "s.h"
    void f(S* p) { /* ... */ }
```

или в графическом виде:



Приведем примеры трех способов нарушения правила ODR:

```
//file1.c:
    struct S1 { int a; char b; };

    struct S1 { int a; char b; };     // ошибка: повторное определение
```

Это является ошибкой, потому что структуру нельзя дважды определить в одной единице трансляции.

```
//file1.c:
    struct S2 { int a; char b; };

//file2.c:
    struct S2 { int a; char bb; };    // ошибка
```

Это является ошибкой, потому что ***S2*** используется в качестве имени структуры, у которой отличаются имена членов.

```
//file1.c:
    typedef int X;
    struct S3 { X a; char b; };

//file2.c:
    typedef char X;
    struct S3 { X a; char b; };       // ошибка
```

Здесь имеется два определения ***S3***, которые идентичны лексема за лексемой, но пример является ошибочным, потому что смысл ***X*** различается в двух файлах.

Проверка согласованности определений классов в различных единицах трансляции, как правило, находится вне пределов возможностей большинства реализаций C++. Как следствие, нарушение правила ODR может являться источником очень тонких ошибок. К сожалению, техника помещения совместно используемых определений в заголовочные файлы с последующим их включением в исходный файл не предохраняет от последней формы нарушения правила. Локальные ***typedef*** и макросы могут изменить смысл включаемых объявлений:

```
//file s.h:
    struct S { Point a; char b; };

//file1.c:
    #define Point int
    #include "s.h"
    // ...

//file2.c:
    class Point { /* ... */ };
    #include "s.h"
    // ...
```

Наилучшей защитой от подобных проблем является создание как можно более самодостаточных заголовочных файлов. Например, если бы класс ***Point*** был объявлен в заголовочном файле ***s.h***, ошибка была бы обнаружена.

Определение шаблона можно включать в несколько единиц трансляции, пока выполняется правило ODR. Кроме того, экспортируемый шаблон можно использовать при наличии только объявления:

```
//file1.c:
    export template<class T> T twice (T t) { return t+t; }

//file2.c:
    template<class T> T twice (T t);          // объявление
    int g (int i) { return twice (i); }
```

Ключевое слово ***export*** означает «доступно из другой единицы трансляции» (§ 13.7).

9.2.4. Компоновка кода, написанного не на C++

Нередко программы на C++ содержат фрагменты, написанные на других языках. Аналогично, довольно часто фрагменты кода C++ используются как часть программ, написанных в основном на другом языке. Могут встретиться определенные препятствия на пути подобного «сотрудничества» между фрагментами программ, написанных на различных языках, и даже между кодом, написанным на одном языке, но скомпилированном разными компиляторами. Например, различные языки и различные реализации компиляторов одного и того же языка могут отличаться использованием машинных регистров для хранения аргументов, расположением аргументов в стеке, размещением встроенных типов, таких как строки и целые, формой имен, передаваемых компилятором компоновщику, и полнотой проверки соответствия типов, требуемой от компоновщика. Чтобы помочь компоновщику, в объявлениях ***extern*** указывают *соглашения по компоновке*. Например, в следующем примере объявляется функция ***strcpy*** () стандартной библиотеки C и C++ и указывается, что компоновка должна производиться в соответствии с соглашениями, принятыми в C:

```
extern "C" char* strcpy (char*, const char*);
```

Эффект от такого объявления отличается от «обычного» объявления

```
extern char* strcpy (char*, const char*);
```

только соглашением о вызове ***strcpy*** ().

Директива ***extern "C"*** особенно полезна ввиду тесной взаимосвязи C и C++. Обратите внимание, что ***"C"*** в ***extern "C"*** определяет соглашение о компоновке, а не язык. Часто ***extern "C"*** используется для компоновки процедур, написанных на Fortran или ассемблере, которые удовлетворяют соглашениям реализации C.

Директива ***extern "C"*** указывает (только) на соглашение о компоновке и не влияет на смысл вызова функции. В частности, функция, объявленная ***extern "C"***, подчиняется проверке типов и правилам преобразования C++, а не менее слабым правилам C. Например:

```
extern "C" int f ();

int g ()
{
    return f (1);        // ошибка: функция не ожидает аргумента
}
```

Добавление ***extern "C"*** к большому количеству объявлений может причинять неудобства. Имеется механизм указания соглашения о компоновке для группы объявлений. Например:

```
extern "C" {
        char* strcpy (char*, const char*);
        int strcmp (const char*, const char*);
        int strlen (const char*);
        //...
}
```

Этой конструкцией, обычно называемой *блоком спецификации компоновки* (linkage block), можно воспользоваться для включения целого заголовочного файла C таким образом, чтобы обеспечить использование указанного файла в C++. Например:

```
extern "C" {
#include <string.h>
}
```

Такая техника обычно используется для создания заголовочного файла C++ из заголовочного файла C. С другой стороны, можно воспользоваться директивами условной компиляции (§ 7.8.1) для создания заголовочного файла, общего для C и C++:

```
#ifdef __cplusplus
extern "C" {
#endif

        char* strcpy (char*, const char*);
        int strcmp (const char*, const char*);
        int strlen (const char*);

#ifdef __cplusplus
}
#endif
```

Предопределенное имя **__cplusplus** используется для того, чтобы исключить конструкции C++, когда файл используется в качестве заголовочного в C.

В блоке спецификации компоновки могут быть любые объявления:

```
extern "C" {            // любые объявления, например:
        int g1;         // определение
        extern int g2;  // объявление (не определение)
}
```

В частности, область видимости и класс памяти переменных остаются прежними, поэтому **g1** остается глобальной переменной — она определена, а не просто объявлена. Для того чтобы объявить переменную без ее определения, вы должны указать ключевое слово **extern** непосредственно в объявлении. Например:

```
extern "C" int g3;      // объявление без определения
```

На первый взгляд это выглядит странно. Однако, это простое следствие сохранения смысла неизменным при добавлении **"C"** к внешнему объявлению и сохранения назначения файла при заключении его в блок спецификации компоновки.

Имя, которое должно быть скомпоновано в стиле C, можно объявить в пространстве имен. Пространство имен окажет воздействие на способ доступа к имени в программе на C++, но не на то, как его видит компоновщик. Типичным примером является **printf**() из **std**:

```
#include<cstdio>

void f()
{
    std::printf("Здравствуй,");      // правильно
    printf("мир!\n");                // ошибка: нет глобальной printf()
}
```

Даже при наличии явного квалификатора *std::printf* вызовется все та же старая *printf()* из C (§ 21.8).

Обратите внимание, что это позволяет нам включать библиотеки с компоновкой в стиле C в выбранные нами, а не глобальные пространства имен. К сожалению, подобная гибкость недоступна для заголовочных файлов, определяющих функции с компоновкой в стиле C++ в глобальном пространстве имен. Причина в том, что компоновка сущностей C++ должна принимать во внимание пространства имен таким образом, чтобы сгенерированные объектные файлы отражали факт наличия или отсутствия использования пространств имен.

9.2.5. Компоновка и указатели на функции

При совместном использовании фрагментов кода на C и C++ в одной программе иногда возникает необходимость в передаче указателя на функцию, определенную в одном языке, функциям, определенным в другом. Если обе реализации двух языков разделяют соглашения о компоновке и механизмы вызова функций, подобная передача указателей на функции является тривиальной задачей. Однако, в общем случае нельзя ожидать подобного сходства, поэтому нужно предпринять специальные действия для обеспечения того, чтобы функция вызывалась ожидаемым образом.

При указании соглашения о компоновке в объявлении оно применяются ко всем типам функций, именам функций и именам переменных, помещенным в объявление (объявления). Это делает возможными все варианты странных на вид, но иногда имеющих большое значение комбинаций компоновки. Например:

```
// FT компонуется в стиле C++
typedef int (*FT)(const void*, const void*);

extern "C" {
    // CFT компонуется в стиле C
    typedef int (*CFT)(const void*, const void*);
    // cmp компонуется в стиле C
    void qsort(void* p, size_t n, size_t sz, CFT cmp);
}

// cmp компонуется в стиле C++
void isort(void* p, size_t n, size_t sz, FT cmp);
// cmp компонуется в стиле C
void xsort(void* p, size_t n, size_t sz, CFT cmp);
// cmp компонуется в стиле C++
extern "C" void ysort(void* p, size_t n, size_t sz, FT cmp);

// compare() компонуется в стиле C++
int compare(const void*, const void*);
// ccmp() компонуется в стиле C
extern "C" int ccmp(const void*, const void*),
```

```
void f(char* v, int sz)
{
    qsort(v, sz, 1, &compare);      // ошибка
    qsort(v, sz, 1, &ccmp);         // правильно

    isort(v, sz, 1, &compare);      // правильно
    isort(v, sz, 1, &ccmp);         // ошибка
}
```

Реализации, в которых C и C++ используют одинаковые соглашения о вызовах, могут принять случаи, помеченные как ошибочные, в качестве расширений языка.

9.3. Использование заголовочных файлов

Для иллюстрации использования заголовочных файлов я представлю несколько различных способов выражения физической структуры программы калькулятора (§ 6.1, § 8.2).

9.3.1. Единственный заголовочный файл

Простейшим решением проблемы разбиения программы на несколько файлов является помещение определений в подходящее число *.c* файлов, а объявлений типов, необходимых им для взаимодействия, в единственный *.h* файл, который будет включен в каждый *.c* файл. Для программы калькулятора мы можем воспользоваться пятью *.c* файлами — ***lexer.c***, ***parser.c***, ***table.c***, ***error.c*** и ***main.c*** — для хранения функций и определений данных, и заголовочным файлом ***dc.h*** для хранения объявлений каждого имени, используемого более чем в одном *.c* файле.

Заголовочный файл ***dc.h*** мог бы выглядеть следующим образом:

```
// dc.h:

namespace Error {
    struct Zero_divide { };

    struct Syntax_error {
        const char* p;
        Syntax_error(const char* q) { p = q; }
    };
}

#include <string>

namespace Lexer {
    enum Token_value {
        NAME,     NUMBER,     END,
        PLUS='+', MINUS='-',  MUL='*',  DIV='/',
        PRINT=';', ASSIGN='=', LP='(',  RP=')'
    };

    extern Token_value curr_tok;
    extern double number_value;
    extern std::string string_value;
    Token_value get_token();
}
```

```
namespace Parser {
        double prim (bool get);         // обработка первичных выражений
        double term (bool get);         // умножение и деление
        double expr (bool get);         // сложение и вычитание

        using Lexer::get_token;
        using Lexer::curr_tok;
}

#include <map>

extern std::map<std::string, double> table;
namespace Driver {
        extern int no_of_errors;
        extern std::istream* input;
        void skip ();
}
```

Ключевое слово ***extern*** используется в каждом объявлении переменных для гарантии того, что не произойдет множественных определений после того, как мы включим ***dc.h*** в различные ***.c*** файлы. Определения находятся в соответствующих ***.c*** файлах.

За вычетом реального кода ***lexer.c*** будет выглядеть следующим образом:

```
// lexer.c

#include "dc.h"
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token () { /* ... */ }
```

Использование заголовочных файлов подобным образом обеспечивает гарантию того, что каждое объявление в заголовочном файле будет помещено во включающий файл. Например, при компиляции ***lexer.c***, компилятору будет выдано:

```
namespace Lexer {               // из dc.h
        // ..
        Token_value get_token ();
}

// ...

Lexer::Token_value Lexer::get_token () { /* ... */ }
```

Это гарантирует, что компилятор обнаружит любые несоответствия типов, указываемых для имени. Например, если бы ***get_token*** () был объявлен с типом возвращаемого значения ***Token_value***, а определен с ***int***, компиляция ***lexer.c*** завершилась бы неудачей с сообщением об ошибке несоответствия типов. Если определение отсутствует, компоновщик обнаружит эту ошибку. Если отсутствует объявление, в одном из ***.c***-файлов ошибку обнаружит компилятор.

Файл ***parser.c*** будет выглядеть следующим образом:

```
// parser.c:

#include "dc.h"

double Parser::prim (bool get) { /* ... */ }
double Parser::term (bool get) { /* ... */ }
double Parser::expr (bool get) { /* ... */ }
```

Файл ***table.c*** будет выглядеть следующим образом:

```
// table.c:
#include "dc.h"

std::map<std::string, double> table;
```

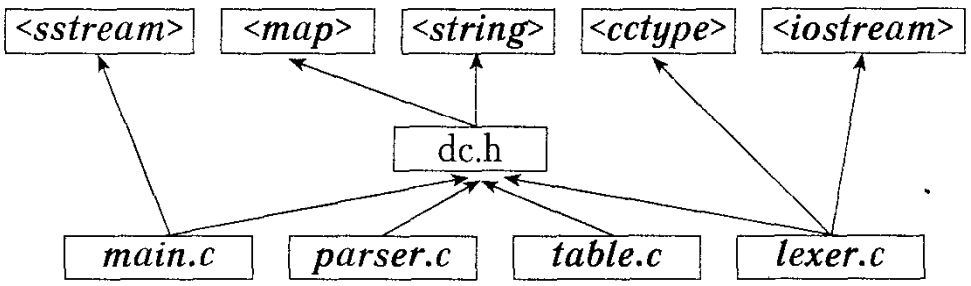
Таблица символов является просто переменной стандартного библиотечного типа ***map***. Это определяет ***table*** в качестве глобальной переменной. В программе реального размера подобные глобальные объявления рано или поздно вызовут проблемы. Я оставил эту «неуклюжесть» с единственной целью — предупредить о ней.

Наконец, ***main.c*** будет выглядеть следующим образом:

```
// main.c:

#include "dc.h"
#include <sstream>
#include <iostream>

int Driver::no_of_errors = 0;
std::istream* Driver::input = 0;

void Driver::skip () { /* ... */ }

int main (int argc, char* argv[]) { /* ... */ }
```

Для того чтобы функция ***main*** () распознавалась в качестве главной, она должна быть глобальной, поэтому здесь не используется пространство имен.

Физическая структура системы может быть представлена следующим образом:



Обратите внимание, что все заголовочные файлы в верхнем ряду являются средствами стандартной библиотеки. Во многих случаях при анализе эти библиотеки можно проигнорировать, потому что они хорошо известны и стабильны. В крошечной программе структура может быть упрощена путем помещения всех директив ***#include*** в общий заголовочный файл.

Этот стиль физического разбиения с единственным заголовочным файлом наиболее полезен, когда программа имеет небольшой размер и не подразумевается отдельного использования ее частей. Обратите внимание, что когда применяются пространства имен, логическая структура программы все еще отражается в ***dc.h***. Если пространства

имен не используются, то структура становится менее понятной. В этом случае могут быть полезны комментарии.

Для больших программ подход с единственным заголовочным файлом неработоспособен в обычной среде разработки, ориентированной на файлы. Изменения в общем заголовочном файле вызывают перекомпиляцию всей программы, а модификация одного заголовочного файла несколькими программистами провоцирует ошибки. Если не делается акцент на стиле программирования с активным использованием пространств имен и классов, логическая структура ухудшается с ростом программы.

9.3.2. Несколько заголовочных файлов

Альтернативной физической организацией является создание для каждого логического модуля своего заголовочного файла, определяющего реализуемые в нем средства. То есть у каждого *.c* файла имеется соответствующий *.h* файл, определяющий его интерфейс. Каждый *.c* файл включает свой *.h* файл и еще несколько других *.h* файлов, определяющих, что ему требуется от других модулей для реализации средств, объявленных в его интерфейсе. Физическая организация соответствует логической организации модуля. Интерфейс для пользователей помещен в *.h* файл, интерфейс для разработчиков выносится в файл, оканчивающийся на *_impl.h*, а определения функций модуля, его переменных и т. д. записываются в *.c* файлы. В этом случае синтаксический анализатор будет представлен тремя файлами:

```
// parser.h:

namespace Parser {                    // интерфейс для пользователей
        double expr (bool get);
}
```

Совместно используемое окружение для функций, реализующих анализатор, представлено файлом ***parser_impl.h***:

```
// parser_impl.h:

#include "parser.h"
#include "error.h"
#include "lexer.h"

namespace Parser {                    // интерфейс для разработчиков
        double prim (bool get);
        double term (bool get);
        double expr (bool get);

        using Lexer::get_token;
        using Lexer::curr_tok;
}
```

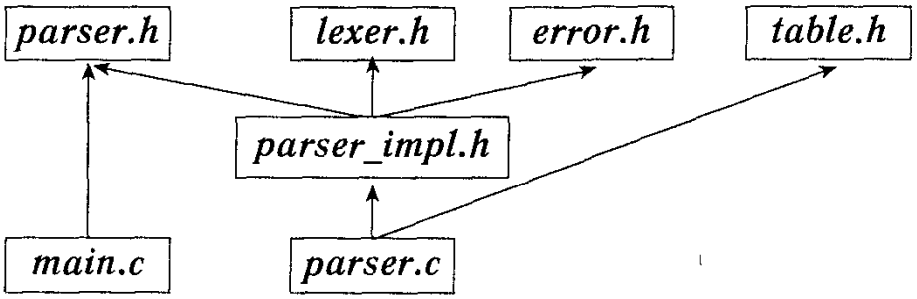
Пользовательский заголовочный файл ***parser.h*** включен для того, чтобы предоставить возможность компилятору проверить согласованность (§ 9.3.1).

Функции, реализующие анализатор, хранятся в ***parser.c*** вместе с директивами ***#include*** для заголовочных файлов, требуемых функциями ***Parser***:

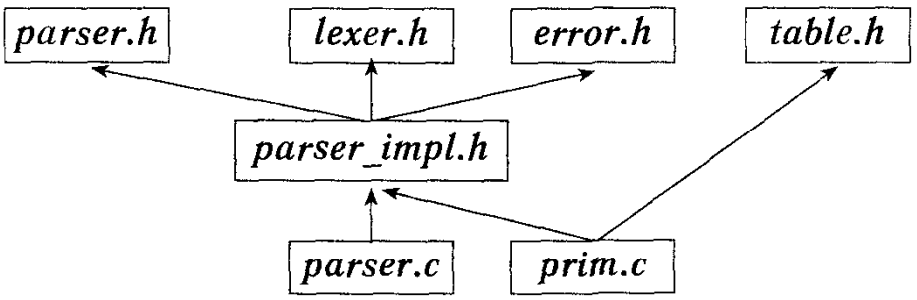
```
// parser.c:

#include "parser_impl.h"
```

```
#include "table.h"

double Parser::prim (bool get) { /* ... */ }
double Parser::term (bool get) { /* ... */ }
double Parser::expr (bool get) { /* ... */ }
```

В графическом виде, анализатор и его использование драйвером выглядят следующим образом:



Как мы того и хотели, существует тесная связь с логической структурой, описанной в § 8.3.3. Для упрощения этой структуры мы могли бы поместить включение *table.h* в *parser_impl.h*, вместо *parser.c*. Однако *table.h* не обязан выражать совместное использование функций синтаксического анализатора; этот заголовочный файл требуется только для реализации функций анализатора. В действительности, он используется только одной функцией, *prim* (), поэтому, если бы мы задались целью свести к минимуму зависимости, мы поместили бы *prim* () в собственный *.c* файл и включили *table.h* только туда:



Подобная тщательность годится только в случае больших модулей. В модулях реального размера обычной практикой является включение при необходимости дополнительных файлов для отдельных функций. Более того, не редко создается более одного файла *_impl.h*, потому что различным подмножествам функций модуля требуется различный совместно используемый контекст.

Обратите внимание, что форма записи *_impl.h* не является стандартом и более того, нельзя сказать, что она широко распространена; просто мне нравится такой стиль задания имен.

Зачем усложнять себе жизнь схемой с несколькими заголовочными файлами? Очевидно, что требуется гораздо меньше интеллектуальных усилий для помещения всех объявлений в единственный файл, как это было сделано с *dc.h*.

Декомпозиция на несколько заголовочных файлов начинает играть значительную роль в модулях, которые в несколько раз больше нашего игрушечного синтаксического анализатора, и программах, в несколько раз больших нашего калькулятора. Фундаментальная идея, которая лежит в декомпозиции такого типа, состоит в том, что она обеспечивает лучшую локализацию. Во время анализа и модификации большой програм-

мы программисту важно сосредоточить внимание на относительно небольших фрагментах кода. Декомпозиция на несколько заголовочных файлов позволяет легче определить, от чего зависит код анализатора и проигнорировать остальную часть программы. Вариант с единственным заголовочным файлом вынуждает нас просматривать все объявления, используемые каждым модулем, и решать, какие из них существенны. Простая истина состоит в том, что сопровождение кода всегда происходит при наличии неполной информации и локальных знаний. Декомпозиция на несколько файлов позволяет нам успешно применять метод «изнутри наружу», имея только локальное представление о программе. Метод с единым заголовочным файлом — как и любой другой метод, основанный на глобальном хранилище информации, — требует подхода «сверху вниз» и заставляет нас бесконечно выяснять, что от чего зависит.

Локализация ведет к сокращению информации, требуемой для компиляции модуля и, как следствие, к меньшему времени компиляции. Разница может быть огромной. Я наблюдал, как время компиляции уменьшалось в десять раз в результате простого анализа зависимостей, приведшего к лучшему использованию заголовочных файлов.

9.3.2.1. Остальные модули калькулятора

Оставшиеся модули калькулятора можно организовать аналогично модулям синтаксического анализатора. Однако, они настолько малы, что не требуют собственных файлов ***_impl.h***. Такие файлы требуются только тогда, когда логический модуль состоит из большого числа функций, которым требуется совместно используемый контекст.

Обработчик ошибок сократился до набора типов исключений, поэтому ***error.c*** не требуется:

```
// error.h:

namespace Error {
        struct Zero_divide { };

        struct Syntax_error {
                const char* p;
                Syntax_error(const char* q) { p = q; }
        };
}
```

Лексический анализатор предоставляет больший по размеру и более запутанный интерфейс:

```
// lexer.h:

#include <string>

namespace Lexer {

        enum Token_value {
                NAME,       NUMBER,     END,
                PLUS='+',   MINUS='-',  MUL='*',   DIV='/',
                PRINT=';',  ASSIGN='=', LP='(',    RP=')'
        };

        extern Token_value curr_tok;
        extern double number_value;
```

```
    extern std::string string_value;

    Token_value get_token ();
}
```

Реализация лексического анализатора, кроме *lexer.h*, зависит от *error.h*, <*iostream*> и функций, определяющих тип символов, объявленных в <*cctype*>:

```
// lexer.c

#include "lexer.h"
#include "error.h"
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token () { /* ... */ }
```

Мы могли бы выделить директиву *#include"error.h"* в файл *_impl.h* лексического анализатора. Однако я решил, что это уже слишком для такой крошечной программы.

Как обычно, мы включаем интерфейс модуля — в нашем случае *lexer.h* — в реализацию модуля для того, чтобы компилятор мог проверить согласованность.

Таблица символов вполне самодостаточна, хотя заголовочный файл стандартной библиотеки <*map*> предоставляет множество вещей, интересных с точки зрения реализации эффективного шаблона *map*:

```
// table.h:

#include <map>
#include <string>

extern std::map<std::string, double> table;
```

Так как мы предполагаем, что каждый заголовочный файл может быть включен в несколько *.c* файлов, мы должны отделить объявление *table* от реализации несмотря на то, что разница между *table.c* и *table.h* состоит в единственном ключевом слове *extern*:

```
// table.c:

#include "table h"

std::map<std: string, double> table;
```

Драйвер зависит от всех модулей:

```
#include "parser.h"
#include "lexer.h"
#include "error.h"
#include "table.h"

namespace Driver {
    int no_of_errors;
    std::istream* input;
    void skip ();
}
```

```
#include <sstream>

int main (int argc, char* argv[]) { /* ... */ }
```

Так как пространство имен ***Driver*** используется исключительно функцией ***main*** (), я поместил его в ***main.c***. В качестве альтернативы я мог поместить его в ***driver.h*** и произвести включение.

В большой системе обычно имеет смысл организовать модули так, чтобы драйвер имел меньше непосредственных зависимостей. Кроме того, часто имеет смысл свести к минимуму набор операций, выполняемых в функции ***main*** (), поместив в нее вызов драйвера, находящегося в отдельном исходном файле. Это особенно важно, если код планируется использовать в качестве библиотеки. В этом случае мы не можем использовать код в ***main*** () и должны быть готовы к вызову из различных функций (§ 9.6[8]).

9.3.2.2. Использование заголовочных файлов

Сколько заголовочных файлов нужно использовать в программе, зависит от множества факторов. Многие из них имеют большее отношение к тому, как обрабатываются файлы в вашей системе, чем к C++. Например, если в вашем редакторе нет возможности одновременного просмотра нескольких файлов, использование большого количества заголовочных файлов становится менее привлекательно. Аналогично, если открытие и чтение 20 файлов по 50 строк каждый занимает гораздо больше времени, чем чтение одного файла из 1000 строк, вы должны дважды подумать, прежде чем придерживаться стиля с несколькими заголовочными файлами в небольшом проекте.

Предупреждение: с десятком заголовочных файлов и стандартными файлами среды выполнения (которые часто исчисляются сотнями) обычно можно справиться. Однако, если вы разбиваете объявления большой программы на заголовочные файлы минимального размера (помещая каждое объявление структуры в отдельный файл), в результате вы легко можете получить мешанину из сотен файлов даже в небольшом проекте. Я считаю это излишним.

В больших проектах наличие нескольких заголовочных файлов неизбежно. В таких проектах появление нескольких сотен файлов (не считая стандартных заголовочных файлов) является нормой. Настоящая путаница наступает, когда их количество переваливает за тысячу. При таком масштабе обсуждаемые здесь базовые методы применимы, но управление файлами становится занятием, достойным Геракла. Помните, что для программ реального размера не годится стиль с единственным заголовочным файлом. Такие программы имеют множество заголовочных файлов. Выбор между двумя стилями происходит для каждой из составных частей программы.

Два стиля, в действительности, не являются взаимно исключающими. Они являются дополняющими друг друга методами, которые надо рассматривать при проектировании каждого значительного модуля и пересматривать по мере развития системы. Исключительно важно помнить, что один и тот же интерфейс не может быть идеальным во всех случаях. Как правило, следует различать интерфейсы для пользователей и для разработчиков. Кроме того, при создании большой системы полезно структурировать ее таким образом, чтобы предоставлялся простой интерфейс для большинства пользователей и более подробный интерфейс для экспертов. Пользовательский интерфейс для экспертов («полный интерфейс») будет включать в себя го-

раздо больше средств, чем обычный пользователь вообще хотел бы знать. В действительности, обычный пользовательский интерфейс часто можно получить исключением конструкций, требующих включения заголовочных файлов, которые определяют неизвестные рядовому пользователю средства. Термин «рядовой пользователь» не является унизительным. В тех областях, в которых я не *обязан* быть экспертом, я предпочитаю быть рядовым пользователем.

9.3.3. Стражи включения

Идея подхода с несколькими заголовочными файлами состоит в том, чтобы представить каждый логический модуль в виде согласованного, самодостаточного фрагмента. С точки зрения программы в целом многие объявления, необходимые для обеспечения завершенности каждого логического модуля, избыточны. В больших программах подобная избыточность может привести к ошибкам, так как заголовочный файл, содержащий определение класса или встроенных функций, может быть дважды включен в одной и той же единице компиляции (§ 9.2.3).

У нас есть два выбора:

[1] Реорганизовать программу таким образом, чтобы исключить избыточность.

[2] Найти метод, допускающий повторное включение заголовочных файлов.

Первый подход — он привел к заключительной версии калькулятора — довольно утомителен и непрактичен при создании программ реального размера. Кроме того, нам нужна избыточность для того, чтобы отдельные части программ были осмысленными сами по себе.

Выигрыш от устранения избыточных включений и вытекающего отсюда упрощения программы может быть весьма значительным и с логической точки зрения, и в плане сокращения времени компиляции. Однако, анализ такой избыточности очень редко бывает полным, поэтому нам требуется метод избыточного включения. Желательно, чтобы его можно было использовать систематически, потому что мы не знаем, насколько тщательно пользователь будет проводить анализ.

Традиционным решением является вставка *стражей включения* (include guards). Например:

```
// error.h

#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace Error {
        //...
}

#endif      // CALC_ERROR_H
```

Содержимое файла между ***#ifndef*** и ***#endif*** игнорируется компилятором, если ***CALC_ERROR_H*** определено. В этом случае при первом просмотре ***error.h*** во время компиляции его содержимое читается и ***CALC_ERROR_H*** присваивается значение. Если компилятор встретит ***error.h*** снова во время компиляции, его содержимое игнорируется. Это является частью трюкачества с макросами, но такой метод работает и широко распространен в мире C и C++. Все стандартные заголовочные файлы содержат стражей включения.

Заголовочные файлы включаются в произвольных местах и нет защиты против конфликтов имен макросов посредством механизма пространств имен. Поэтому я выбрал довольно длинное и неуклюжее имя в качестве «стража включения».

Привыкнув к заголовочным файлам и «стражам включения», программисты демонстрируют тенденцию включать множество заголовочных файлов прямо или косвенно. Даже в реализациях C++, которые оптимизируют обработку заголовочных файлов, это может быть нежелательно, потому что может привести к чрезмерно большому времени компиляции и привнести множество объявлений и макросов в область видимости. Последнее может оказать нежелательное воздействие на смысл программы, причем непредсказуемым и опасным образом. Заголовочные файлы нужно включать только тогда, когда в них есть реальная необходимость.

9.4. Программы

Программа является набором отдельных единиц компиляции, объединяемых компоновщиком. Каждая функция, объект, тип и т. д., используемые в этом наборе, должны иметь единственное определение (§ 4.9, § 9.2.3). Программа должна содержать ровно одну функцию с именем ***main*** () (§ 3.2). Выполнение программы начинается с вызова ***main*** () и заканчивается после выхода из ***main*** (). Целое значение, возвращенное ***main*** (), передается вызвавшей системе в качестве результата работы программы.

Эти простые соображения должны учитываться в программах, в которых имеются глобальные переменные (§ 10.4.9), или в которых генерируются неперехватываемые исключения (§ 14.7).

9.4.1. Инициализация нелокальных переменных

В принципе, переменные, определенные вне любой функции (то есть глобальные, глобальные в пространстве имен и статические переменные классов), инициализируются до вызова ***main*** (). Такие нелокальные переменные инициализируются в единице трансляции в порядке их объявления (§ 10.4.9). Если переменная не имеет явного инициализирующего значения, ей присваивается значение по умолчанию для ее типа (§ 10.4.2). Значение по умолчанию для встроенных типов и перечислений равно нулю. Например:

```
double x = 2,          // нелокальные переменные
double y,
double sqx = sqrt (x+y),
```

В примере ***x*** и ***y*** инициализируются прежде, чем ***sqx***, поэтому ***sqrt*** вызывается с аргументом 2.

Нет гарантированного порядка инициализации глобальных переменных из различных единиц трансляции. Следовательно, не стоит полагаться на какой-либо порядок такой инициализации. Кроме того, невозможно перехватить исключение, возбужденное инициализатором глобальной переменной (§ 14.7). Как правило, лучше свести к минимуму использование глобальных переменных, в особенности тех, которые требуют сложной инициализации.

Существует несколько методов задания порядка инициализации глобальных переменных из отдельных единиц трансляции. Однако ни одна из них не является ни переносимой, ни эффективной. В частности, динамически компонуемые библиотеки

не могут «мирно сосуществовать» с глобальными переменными со сложными зависимостями.

Часто функция, возвращающая ссылку, является хорошей альтернативой глобальной переменной. Например:

```
int& use_count ()
{
        static int uc = 0;
        return uc;
}
```

В этом случае вызов *use_count* () действует как глобальная переменная, за исключением того, что ее инициализация происходит при первом использовании (§ 5.5).

Например:

```
void f()
{
        cout << ++use_count (),        // увеличить значение и вывести его
        //..
}
```

Инициализация нелокальных статических переменных зависит от механизма запуска программы на C++, поддерживаемого конкретной реализацией. Гарантируется, что этот механизм правильно работает, только если выполняется *main* (). Поэтому следует избегать нелокальных переменных, требующих инициализации во время выполнения, в коде на C++, если этот код является фрагментом программы, написанной не на C++.

Обратите внимание, что переменные, инициализированные константными выражениями (§ B.5), не могут зависеть от значений объектов из других единиц трансляции и не требуют инициализации во время выполнения. Поэтому такие переменные безопасно использовать во всех случаях.

9.4.1.1. Завершение выполнения программы

Программа может завершить свое выполнение несколькими способами:

- выйдя из *main* ();
- вызвав *exit*();
- вызвав *abort* ();
- сгенерировав неперехватываемое исключение.

Кроме того, существует несколько плохих и зависящих от реализации методов доведения до краха.

Если завершение программы осуществляется вызовом стандартной библиотечной функции *exit* (), то при этом вызываются деструкторы для всех созданных статических объектов (§ 10.4.9, § 10.2.4). Если же завершение программы осуществляется вызовом стандартной библиотечной функции *abort* (), деструкторы не вызываются. Обратите внимание, что вызов *exit* () не завершает выполнение немедленно. Вызов *exit* () из деструктора может привести к бесконечной рекурсии. Функция *exit* () объявлена следующим образом:

```
void exit (int);
```

Так же как возвращаемое значение *main* () (§ 3.2), аргумент *exit* () возвращается в «систему» в качестве результата программы. Ноль означает успешное завершение.

Вызов *exit* () означает, что не будут вызваны деструкторы локальных переменных в вызывающей функции и в функциях, вызвавших ее. Генерация исключения с последующим его перехватом гарантирует, что все локальные объекты будут корректно уничтожены (§ 14.4.7). Кроме того, вызов *exit* () завершает выполнение программы, не позволяя вызвавшей функции выполнить какие-либо действия. Поэтому, как правило, лучше прекратить выполнение текущей функции, сгенерировав исключение, что позволит обработчику решить, что делать в этой ситуации.

Функция стандартной библиотеки C и C++ *atexit* () предоставляет возможность выполниться некоторому коду при завершении программы. Например:

```
void my_cleanup ();

void somewhere ()
{
        if (atexit (&my_cleanup) == 0) {
                // my_cleanup будет вызвана при нормальном завершении
        }
        else {
                // проблема: слишком много функций atexit
        }
}
```

Это напоминает автоматический вызов деструкторов для глобальных переменных при завершении программы (§ 10.4.9, § 10.2.4). Обратите внимание, что у функции-аргумента *atexit* () не может быть ни аргументов, ни возвращаемого значения. Кроме того, в зависимости от реализации существует предельное количество функций *atexit* (); при достижении предела *atexit* () возвращает ненулевое значение. Эти ограничения делают *atexit* () менее полезной, чем может показаться с первого взгляда.

Деструктор объекта со статическим выделением памяти (глобально: § 10.4.9, функции типа *static*: § 7.1.2 или класса типа *static*: § 10.2.4) и созданного до вызова *atexit* (*f*), будет вызван после вызова *f*. Деструктор аналогичного объекта, созданного после вызова *atexit* (*f*), будет вызван до вызова *f*.

Функции *exit* (), *abort* () и *atexit* () объявлены в <*cstdlib*>.

9.5. Советы

[1] Пользуйтесь заголовочными файлами для представления интерфейсов и явного выражения логической структуры; § 9.1, § 9.3.2.

[2] Включайте заголовочные файлы в исходные файлы, реализующие указанные функции; § 9.3.1.

[3] Не определяйте глобальные сущности с одинаковыми именами и со схожим, но различным смыслом в различных единицах трансляции; § 9.2.

[4] Избегайте определений невстроенных функций в заголовочных файлах; § 9.2.1.

[5] Используйте *#include* только в глобальной области видимости и в пространствах имен; § 9.2.1.

[6] Включайте только полные объявления; § 9.2.1.

[7] Пользуйтесь стражами включения; § 9.3.3.

[8] Включайте заголовочные файлы C в пространства имен во избежание появления глобальных имен; § 8.2.9.1, § 9.2.2.

[9] Создавайте самодостаточные заголовочные файлы; § 9.2.3.
[10] Проводите различие между интерфейсами для пользователей и для разработчиков; § 9.3.2.
[11] Проводите различие между интерфейсами для обычных пользователей и для экспертов; § 9.3.2.
[12] Избегайте наличия нелокальных объектов, требующих инициализации на этапе выполнения, если соответствующий код планируется использовать как часть программы, написанной не на C++; § 9.4.1.

9.6. Упражнения

1. (*2) Найдите, где находятся заголовочные файлы стандартной библиотеки в вашей системе. Просмотрите список их имен. Хранятся ли какие-нибудь нестандартные заголовочные файлы вместе со стандартными? Можно ли включить какие-либо нестандартные заголовочные файлы при помощи формы записи с <>?
2. (*2) Где находятся заголовочные файлы нестандартной «фундаментальной» («foundation») библиотеки?
3. (*2.5) Напишите программу, которая читает исходные файлы и выводит имена включенных (*#include*) файлов. Сделайте отступы, чтобы было видно, какие файлы включают другие файлы. Опробуйте эту программу на каких-нибудь исходных файлах (чтобы получить представление об объеме включаемой информации).
4. (*3) Модифицируйте программу из предыдущего упражнения таким образом, чтобы она выводила количество строк-комментариев, количество строк, не содержащих комментарии, и количество слов в каждом включаемом файле.
5. (*2.5) Внешним стражем включения называют конструкцию, которая проверяется за пределами охраняемого ей файла и включает файл только один раз при компиляции. Определите такую конструкцию, разработайте способ ее тестирования и обсудите ее преимущества и недостатки по сравнению со стражем включения, описанным в § 9.3.3. Есть ли в вашей системе значительные преимущества на этапе выполнения от внешнего стража?
6. (*3) За счет чего достигается динамическая компоновка в вашей системе? Какие ограничения накладываются на динамически компонуемый код? Какие требования предъявляются к коду, с которым будет производиться динамическая компоновка?
7. (*3) Откройте и прочитайте 100 файлов, каждый из которых содержит по 1500 символов. Откройте и прочитайте один файл, содержащий 150 000 символов. Подсказка: см. пример в § 21.5.1. Есть разница в производительности? Какое максимальное количество файлов можно одновременно открыть в вашей системе? Рассмотрите эту проблему в плане использования включаемых файлов.
8. (*2) Модифицируйте калькулятор таким образом, чтобы его можно было вызвать из *main* () или из другой функции путем простого вызова.
9. (*2) Нарисуйте диаграмму зависимости модулей (§ 9.3.2) для версии калькулятора, которая использовала *error* (), а не исключения (§ 8.2.2).

ЧАСТЬ ВТОРАЯ

Механизмы абстракции

В этой части описываются средства C++ для определения и использования новых типов. Обсуждаются так называемое объектно-ориентированное и обобщенное программирование.

«...нет дела, коего устройство было бы труднее, ведение опаснее, а успех сомнительнее, нежели замена старых порядков новыми. Кто бы ни выступал с подобным начинанием, его ожидает враждебность тех, кому выгодны старые порядки, и холодность тех, кому выгодны новые...»

— Николо Макиавелли («Государь» §vi)

10

Классы

Эти типы совсем не «абстрактные»,
они настолько же реальны, как int *и* float.
— Дуг МакИлрой

Концепции и классы — члены класса — управление доступом — конструкторы — статические члены — копирование по умолчанию — константные функции-члены — ***this*** — структуры — определение функции в классе — конкретные классы — функции-члены и функции-помощники — перегруженные операторы — использование конкретных классов — деструкторы — конструкторы по умолчанию — локальные переменные — копирование, определяемое пользователем — ***new*** и ***delete*** — объекты-члены — массивы — статическая память — временные переменные — объединения — советы — упражнения.

10.1. Введение

Целью введения концепции классов в C++ является предоставление программисту средств создания новых типов, которые настолько же удобны в использовании, как и встроенные. Кроме того, производные классы (глава 12) и шаблоны (глава 13) представляют способы организации классов, имеющих между собой нечто общее.

Тип является конкретным представлением некоторой концепции. Например, встроенный тип C++ ***float*** вместе с операциями +, −, * и т. д. представляет конкретное воплощение математической концепции вещественного числа. Класс — это определяемый пользователем тип. Мы создаем новые типы для определения концепции, не выражаемой непосредственно встроенными типами. Например, мы могли бы ввести тип ***Trunk_line*** (междугородняя линия) в программе, имеющей отношение к телефонии, тип ***Explosion*** (взрыв) в видеоигре или тип ***list<Paragraph>*** (список абзацев) в программе обработки текста. Программы, типы в которых близко соответствуют концепциям приложения, обычно легче понимать и модифицировать. Тщательно подобранный набор типов, определяемых пользователем, делает программу более краткой и выразительной. Кроме того, такие типы дают возможность проведение разнообразного анализа кода. В частности, они позволяют компилятору обнаружить случаи недопустимого использования объектов, которые в противном случае не были бы выявлены вплоть до этапа тестирования.

Основной смысл введения новых типов состоит в разделении малозначащих деталей реализации (например, расположение в памяти составных частей объектов данного типа) от свойств, имеющих определяющее значение для правильного использования

сущности (например, полный набор функций доступа к данным). Подобное разделение лучше всего выражается в терминах ограничения доступа к данным извне и использования для этой цели специальных процедур в рамках четко определенного интерфейса.

В этой главе особое внимание уделяется относительно простым «конкретным» типам, определяемым пользователем, которые с логической точки зрения незначительно отличаются от встроенных типов. В идеале, такие типы должны отличаться от встроенных не по методам их использования, а только способом создания.

10.2. Классы

Класс — это определяемый пользователем тип. В данном разделе изложены основные средства определения класса, создания объектов класса и манипулирования такими объектами.

10.2.1. Функции-члены

Рассмотрим реализацию концепции даты с использованием структуры ***Date***. Эта структура представляет дату и содержит набор функций, осуществляющих манипуляции с переменными-датами:

```
struct Date {                                   // представление
        int d, m, y,
};

void init_date (Date& d, int, int, int),        // инициализация d
void add_year (Date& d, int n),                 // прибавить n лет к d
void add_month (Date& d, int n),                // прибавить n месяцев к d
void add_day (Date& d, int n),                  // прибавить n дней к d
```

Здесь нет явной связи между типом данных и функциями. Такую связь можно установить, объявив функции в качестве членов структуры:

```
struct Date {
        int d, m, y,

        void init_date (int dd, int mm, int yy ),   // инициализация
        void add_year (int n),                      // прибавить n лет
        void add_month (int n),                     // прибавить n месяцев
        void add_day (int n),                       // прибавить n дней
};
```

Функции, объявленные внутри определения класса (структура является одним из видов класса; § 10.2.8) называются *функциями-членами* и их можно вызывать только для переменной соответствующего типа, используя стандартный синтаксис доступа к членам структуры. Например:

```
Date my_birthday,

void f()
{
        Date today,

        today init (16, 10, 1996),
        my_birthday init (30, 12, 1950),
```

```
        Date tomorrow = today;
        tomorrow.add_day (1);
        //...
}
```

Так как различные структуры могут иметь функции-члены с одинаковыми именами, при определении функции-члена мы должны указать имя структуры:

```
void Date::init (int dd, int mm, int yy)
{
        d = dd;
        m = mm;
        y = yy;
}
```

В теле функции-члена имена членов (этого же класса) можно использовать без явного указания объекта. В данном случае имя относится к члену того объекта, для которого вызвана функция. Например, если ***Date::init*** () вызывается для ***today***, ***m=mm*** означает ***today.m=mm***, если ***Date::init*** () вызывается для ***my_birthday***, ***m=mm*** означает ***my_birthday.m=mm***. Функция-член класса всегда «знает» для какого объекта она вызвана.

Конструкция

```
class X { ... },
```

называется *определением класса*, потому что она определяет новый тип. По историческим причинам определение класса часто называют *объявлением класса*. Также как и объявления, не являющиеся определениями, определение класса может быть повторено в другом исходном файле при помощи ***#include***, не нарушая при этом правила одного определения ODR (§ 9.2.3).

10.2.2. Управление доступом

Объявление ***Date*** из предыдущего раздела предоставляет набор функций для работы с ***Date***. Однако оно не указывает, что только эти функции непосредственно зависят от представления ***Date*** и только они могут непосредственно осуществлять доступ к объектам класса ***Date***. Эти ограничения можно отразить, воспользовавшись ключевым словом ***class*** вместо ***struct***:

```
class Date {
        int d, m, y;
public
        void init_date (int dd, int mm, int yy);    // инициализация

        void add_year (int n),                       // прибавить n лет
        void add_month (int n),                      // прибавить n месяцев
        void add_day (int n),                        // прибавить n дней
},
```

Метка ***public*** разделяет тело класса на две части. Имена в первой (***private*** — закрытой) части могут использоваться только функциями-членами. Вторая (***public*** — открытая) часть образует открытый интерфейс объектов класса. Структура, на самом деле, является классом, члены которого открыты по умолчанию (§ 10.2.8); функции-члены можно определить точно также, как и выше. Например:

```
inline void Date::add_year (int n)
{
        y += n;
}
```

Однако функции, не являющиеся членами класса, не могут осуществлять непосредственный доступ к закрытым членам. Например:

```
void timewarp (Date& d)
{
        d.y = 200;          // ошибка: переменная Date::y является закрытой
}
```

Ограничение доступа к структуре данных явно объявленным списком функций имеет несколько преимуществ. Например, ошибка, в результате которой ***Date*** приняла неверное значение (скажем, 36 декабря 1985 года), может быть вызвана только неверным кодом соответствующей функции-члена. Из этого следует, что первая стадия отладки — локализация ошибки — может быть завершена еще до запуска программы. Это является частным случаем общего подхода, состоящего в том, что любое изменение поведения типа ***Date*** может и должно влиять на его члены. В частности, если мы изменим представление класса, нам потребуется только изменить функции-члены, чтобы воспользоваться новым представлением. Код пользователя непосредственно зависит только от открытого интерфейса, и код не потребуется переписывать (хотя может понадобиться его перекомпиляция). Другое преимущество состоит в том, что для того чтобы научиться пользоваться классом, его потенциальному пользователю потребуется ознакомиться только с определениями функций-членов.

Защита закрытых данных базируется на ограничении использования имен членов класса. Эту защиту можно обойти манипулированием с адресами и путем явного преобразования типа. Но это, конечно, уже жульничество. C++ защищает от случайного, а не умышленного нарушения правил. Защиту против злонамеренного доступа к закрытым данным в языке высокого уровня можно осуществить только на аппаратном уровне, и даже это является довольно сложной задачей в реальной системе.

Функция *init* () добавлена отчасти потому, что обычно полезно иметь функцию, присваивающую объекту значение, и отчасти потому, что ввиду закрытости данных, мы вынуждены ввести такую функцию.

10.2.3. Конструкторы

Использование функций типа *init* () для инициализации объектов класса неэлегантно и подвержено ошибкам. Так как нигде не сказано, что объект должен быть проинициализирован, программист может забыть об этом, или сделать это дважды (часто с одинаково разрушительными последствиями). Лучшим подходом будет предоставление программисту возможности объявить функцию, имеющую явное назначение — инициализация объектов. Ввиду того, что такая функция создает (конструирует) значения данного типа, она называется *конструктором*. Конструктор распознается по имени, которое совпадает с именем самого класса. Например:

```
Class Date {
        //...
        Date (int, int, int);          // конструктор
};
```

Если класс имеет конструктор, все объекты этого класса будут проинициализированы. Если конструктору требуются аргументы, они должны быть предоставлены:

```
Date today = Date (23, 6, 1983);
Date xmas (25, 12, 1990),          // сокращенная форма
Date my_birthday,                  // ошибка: отсутствует инициализация
Date release1_0 (10, 12),          // ошибка: отсутствует третий аргумент
```

Довольно часто удобно иметь несколько способов инициализации объекта класса. Этого можно добиться, введя несколько конструкторов. Например:

```
class Date {
        int d, m, y;
public:
        //...
        Date (int, int, int);          // день, месяц, год
        Date (int, int);               // день, месяц, текущий год
        Date (int);                    // день, текущие месяц и год
        Date ();                       // дата по умолчанию — сегодня
        Date (const char*);            // дата в строковом представлении
};
```

Конструкторы подчиняются тем же правилам разрешения перегрузки, что и остальные функции (§ 7.4). Пока конструкторы значительно различаются типами своих аргументов, компилятор может выбрать нужный в каждом конкретном случае:

```
Date today (4);
Date july4 ("July 4, 1983");
Date guy ("5 Nov");
Date now,                          // инициализация по умолчанию текущей датой
```

Размножение конструкторов в примере ***Date*** является типичным. Во время проектирования класса у программиста постоянно возникает искушение добавить средства просто потому, что они кому-нибудь могут понадобиться. Требуется основательно подумать, какие средства нужны на самом деле, и включать только их. Эти дополнительные интеллектуальные усилия приводят к программам, которые более понятны и имеют меньший размер. Одним из способов уменьшения количества похожих друг на друга функций является использование аргументов по умолчанию (§ 7.5). В ***Date*** каждому аргументу может быть присвоено значение по умолчанию, которое можно определить словами: «используй соответствующее значение из текущей даты».

```
Class Date {
        int d, m, y;
public:
        Date (int dd=0, int mm=0, int yy=0);
        //...
};
```

```
Date::Date (int dd, int mm, int yy)
{
        d = dd ? dd : today.d;
        m = mm ? mm : today.m;
        y = yy ? yy : today.y;

        // проверка того, что Date имеет допустимое значение
}
```

Когда значение аргумента используется для указания «вставить значение по умолчанию», оно должно находиться вне пределов допустимых «обычных» значений этого аргумента. Для ***day*** и ***month*** это требование очевидно выполняется (они не могут быть равны нулю), но для ***year*** возникают сомнения. К счастью, в европейском календаре нет нулевого года; первый год от Рождества Христова (***year*** == *1*) идет сразу же за первым годом до Рождества Христова (***year*** == *-1*).

10.2.4. Статические члены

Удобство использования в ***Date*** значения по умолчанию получено за счет серьезной скрытой проблемы. Наш класс ***Date*** теперь зависит от глобальной переменной ***today***. Класс ***Date*** можно теперь применять только в том контексте, в котором ***today*** определена и корректно используется во всем коде. Этот вид ограничений делает класс бесполезным вне контекста, в котором он был изначально написан. Пользователи получают слишком много неприятных сюрпризов, пытаясь воспользоваться подобными контекстно-зависимыми классами, и сопровождение становится чрезмерно сложным. Возможно, «всего лишь одна маленькая глобальная переменная» не станет слишком неуправляемой, но подобный стиль приводит к коду, который бесполезен для всех, кроме программиста, написавшего его. Этого следует избегать.

К счастью, мы можем сохранить удобство значения по умолчанию и избежать неприятностей из-за появления глобальной переменной. Переменная, которая является частью класса, но не является частью объекта этого класса, называется *статическим* членом. Существует ровно одна копия статического члена, в отличии от обычных членов, когда каждый объект класса имеет свои независимые члены. Аналогично, функция, которой требуется доступ к членам класса, но не требуется, чтобы она вызывалась для конкретного объекта класса, называется *статической* функцией-членом.

Приведем пример, сохраняющий смысл конструктора значений по умолчанию ***Date***, в котором отсутствует проблема глобальной переменной:

```
class Date {
        int d, m, y;
        static Date default_date;
public:
        Date (int dd=0, int mm=0, int yy=0);
        // ...
        static void set_default (int, int, int);
};
```

Теперь мы можем определить конструктор ***Date*** следующим образом:

```
Date::Date (int dd, int mm, int yy)
{
```

```
        d = dd ? dd : defalut_date.d;
        m = mm ? mm : defalt_date.m;
        y = yy ? yy : default_date.y;

        // проверка на допустимость значения Date
}
```

Мы можем изменить дату по умолчанию в любой подходящий момент. К статическим членам можно обращаться так же, как и к любым другим членам. Кроме того, к статическому члену можно обращаться без указания имени объекта. Вместо этого в качестве квалификатора его имени используется имя самого класса. Например:

```
void f()
{
        Date::set_default(4, 5, 1945);
}
```

Статические члены — и функции и данные — должны быть где-то определены. Например:

```
Date Date::default_date(16, 12, 1770);

void Date::set_default(int d, int m, int y)
{
        Date::default_date = Date(d, m, y);
}
```

Теперь значением по умолчанию является дата рождения Бетховена (до тех пор, пока кто-нибудь не решит по-другому).

Обратите внимание, что ***Date***() служит другим обозначением для значения ***Date::default_date***. Например:

```
Date copy_of_default_date = Date();
```

Следовательно, нам не нужна отдельная функция для чтения значения даты по умолчанию.

10.2.5. Копирование объектов класса

По умолчанию, объекты класса можно копировать. В частности, объект некоторого класса можно проинициализировать при помощи копирования объекта того же класса. Это можно сделать даже там, где объявлен конструктор. Например:

```
Date d = today;                // инициализация копированием
```

По умолчанию, копия объекта класса содержит копию каждого члена. Если это не совсем то, что вам требуется для класса ***X***, можно реализовать более подходящее поведение, определив копирующий конструктор ***X::X(constX&)***. Это обсуждается в § 10.4.4.1. Аналогично, объекты класса могут по умолчанию копироваться при помощи операции присваивания. Например:

```
void f(Date& d)
{
        d = today;
}
```

И снова семантикой по умолчанию здесь является копирование каждого члена. Если это не годится для класса ***X***, пользователь может определить подходящий оператор присваивания (§ 10.4.4.1).

10.2.6. Константные функции-члены

Определенный нами класс ***Date*** предоставляет функции-члены, которые присваивают и изменяют значение объекта типа ***Date***. К сожалению, мы не обеспечили способа проверки значения объекта ***Date***. Эту проблему можно легко решить, добавив функции, возвращающие значения дня, месяца и года:

```
class Date {
        int d, m, y;
public:
        int day () const { return d; }
        int month () const { return m; }
        int year () const;
        // ...
}
```

Обратите внимание на ***const*** после (пустого) списка аргументов в объявлениях функций. Это означает, что эти функции не изменяют состояние ***Date***.

Естественно, компилятор обнаружит случайные попытки нарушить это обещание. Например:

```
inline int Date::year () const
{
        return y++;                    // ошибка: попытка изменить
                                       // значение члена в константной функции
}
```

Когда константная функция-член определяется вне класса, требуется суффикс ***const***:

```
inline int Date::year () const         // правильно
{
        return y;
}
```

Другими словами, суффикс ***const*** является частью типа функций ***Date***::***day*** () и ***Date***::***year*** ().

Константную функцию-член можно вызвать как для константного, так и для неконстантного объекта, в то время как неконстантную функцию-член можно вызвать только для объекта, не являющнгося константой. Например:

```
void f (Date& d, const Date& cd)
{
        int i = d.year ();             // правильно
        d.add_year (1);                // правильно

        int j = cd.year ()             // правильно
        cd.add_year (1);               // ошибка: нельзя изменить значение константы cd
}
```

10.2.7. Ссылка на себя

Функции-модификаторы состояния ***add_year*** (), ***add_month*** () и ***add_day*** () были определены, как не возвращающие значения. При использовании подобных связанных функций иногда возникает желание выстроить операции в цепочку. Для этого требуется, чтобы функции возвращали ссылку на измененный объект. Например, мы могли бы написать

```
void f(Date& d)
{
        //...
        d.add_day(1).add_month(1).add_year(1);
        //...
}
```

чтобы добавить один день, один месяц и один год к ***d***. Для этого нужно, чтобы функция возвращала ссылку на ***Date***:

```
class Date {
        //...

        Date& add_year(int n);          // прибавить n лет
        Date& add_month(int n);         // прибавить n месяцев
        Date& add_day(int n);           // прибавить n дней
};
```

Каждая (нестатическая) функция-член знает, для какого объекта она вызвана, и может явно на него ссылаться. Например:

```
Date& Date::add_year(int n)
{
        if (d==29 && m==2 && !leapyear(y+n))   // не забудьте о 29 февраля
                                               // leapyear — високосный год
                d = 1;
                m = 3;
        }
        y += n;
        return *this;
}
```

Выражение ****this*** означает объект, для которого вызвана функция-член. Это эквивалентно ***THIS*** в Simula и ***self*** в Smalltalk.

В нестатической функции-члене ключевое слово ***this*** является указателем на объект, для которого вызвана функция. В нестатической функции-члене класса ***X this*** имеет тип ***X****. Однако, это не обычная переменная; невозможно получить ее адрес или присвоить ей что-нибудь. В константной функции-члене класса ***X this*** имеет тип ***const X****для предотвращения модификации самого объекта (см. также § 5.4.1).

В большинстве случаев использование ***this*** является неявным. В частности, каждое обращение к нестатическому члену внутри класса неявно использует ***this*** для доступа к члену соответствующего объекта. Например, функцию ***add_year*** можно определить эквивалентным, хотя и более пространным, способом:

```
Date& Date::add_year(int n)
{
```

```
        // не забудьте о 29 февраля
        if (this->d==29 && this->m=2 && !leapyear(this->y+n)) {
                this->d = 1,
                this->m = 3,
        }
        this->y + = n,
        return *this,
}
```

Примером широко распространенного явного использования **this** являются операции со связным списком (см., например, § 24.3.7.4).

10.2.7.1. Физическое и логическое постоянство

Иногда, функция-член с логической точки зрения является константной, но тем не менее ей требуется модифицировать некоторые члены. С точки зрения пользователя функция не изменяет состояние объекта. Однако незаметно для пользователя могут меняться некоторые части объекта. Это явление часто называют *логическим постоянством*. Например, класс **Date** может иметь функцию, возвращающую строковое представление, которое могло бы понадобиться пользователю для вывода. Создание такого представления, в общем случае, может оказаться достаточно дорогой операцией. Поэтому, имело бы смысл хранить последнюю копию, чтобы при повторных запросах просто возвращать ее, если значение **Date** не изменилось. Кэширование значений подобным образом чаще встречается для более сложных структур данных, но давайте посмотрим, как это можно реализовать в **Date**:

```
class Date {
        bool cache_valid,
        string cache,
        void compute_cache_value() const,
        // ...
public
        // ...
        string string_rep() const,                    // строковое представление
},
```

С точки зрения пользователя, **string_rep** не меняет состояния **Date**, поэтому она, очевидно, должна быть константной функцией-членом. С другой стороны, требуется заполнить кэш перед его использованием. Этого можно добиться, используя грубую силу:

```
string Date string_rep() const
{
        if (cache_valid == false) {
                Date* th = const_cast<Date*>(this),    // снимаем const приведением типа
                th->compute_cache_value(),
                th->cache_valid = true,
        }
        return cache,
}
```

Оператор **const_cast** (§ 15.4.2.1) используется для получения указателя типа **Date*** на **this**. Это вряд ли является элегантным решением и, кроме того, нет гарантии, что оно будет работать с объектом, который был объявлен константой. Например:

```
Date d1;
const Date d2;
string s1 = d1.string_rep ();
string s2 = d2.string_rep ();          // неопределенное поведение
```

В случае с *d1* *string_rep* () просто осуществляет обратное приведение к исходному типу *d1* и вызов будет работать. Однако *d2* объявлена константой и реализация может использовать некоторую форму защиты памяти для гарантии того, что такой объект не будет разрушен. Следовательно, не гарантируется, что *d2.string_rep* () выдаст одинаковый и предсказуемый результат во всех реализациях.

10.2.7.2. Объявление mutable

Можно избежать явного преобразования типа «снятие *const* путем приведения» и последующей зависимости от реализации, объявив данные, участвующие в управлении кэш-памятью, как *mutable*:

```
class Date {
        mutable bool cache_valid;
        mutable string cache;
        void compute_cache_value ();
        // ...
public:
        // ...
        string string_rep () const;      // строковое представление
};
```

Квалификатор хранения *mutable* указывает, что член должен храниться таким способом, чтобы допускалась его модификация, даже если он является членом константного объекта. Другими словами, *mutable* означает: «ни при каких условиях не является *const*». Этим можно воспользоваться для упрощения определения *string_rep* ():

```
string Date::string_rep () const
{
        if (!cache_valid) {
                compute_cache_value ();
                cache_valid = true;
        }
        return cache;
}
```

В результате, использование *string_rep* () становится корректным. Например:

```
Date d3;
const Date d4;
string s3 = d3.string_rep ();
string s4 = d4.string_rep ();          // правильно!
```

Объявление имен с квалификатором *mutable* наиболее приемлемо, когда (только лишь) часть представления может быть модифицирована. Если большая часть объекта подвержена изменениям, в то время как с логической точки зрения объект остается константой, часто предпочтительнее поместить изменяющиеся данные в отдельный объект

и осуществлять доступ к нему косвенно. С использованием такой техники пример кэширования строки принимает следующий вид:

```
struct cache {
        bool valid;
        string rep;
};

class Date {

        cache* c;                       // инициализируется в конструкторе (§ 10.4.6)
        void compute_cashe_value () const;
        // ...
public:
        // ...
        string string_rep () const;     // строковое представление
};

string Date::string_rep () const
{
        if (!c->valid) {
                compute_cache_value ();
                c->valid = true;
        }
        return c->rep;
}
```

Техника программирования с использованием кэш-памяти обобщается до различных форм минимизации вычислений.

10.2.8. Структуры и классы

По определению структура есть класс, все члены которого по умолчанию являются открытыми, то есть

```
struct s { ...
```

является просто более короткой формой записи

```
class s { public: ...
```

Модификатор доступа ***private***: можно использовать для указания того, что последующие члены являются закрытыми, аналогично тому, как ***public***: указывает на то, что последующие члены открыты. За исключением различий в именах следующие объявления эквивалентны:

```
class Date1 {
        int d, m, y,
public:
        Date1 (int dd, int mm, int yy);

        void add_year (int n);          // прибавить n лет
};

struct Date2 {
private:
```

```
        int d, m, y;
public
        Date2 (int dd, int mm, int yy);
        void add_year (int n),          // прибавить n лет
};
```

Какого стиля придерживаться, зависит от обстоятельств и вкуса. Я обычно предпочитаю использовать структуры для классов, у которых все данные открыты. Я думаю о таких классах, как о «не совсем типах, являющихся просто структурами данных». Конструкторы и функции доступа могут быть весьма полезны даже для таких структур, но скорее для удобства, а не как гаранты свойств типа (инвариантов, см. § 24.3.7.1). Объявление данных в начале класса не является требованием. В действительности, имеет смысл помещать данные в конце, чтобы сделать акцент на функциях открытого интерфейса. Например:

```
class Date3 {
public:
        Date3 (int dd, int mm, int yy);
        void add_year (int n);          // прибавить n лет
private:
        int d, m, y;
};
```

В реальном коде, где и открытый интерфейс, и детали реализации обычно более объемны, чем в учебных примерах, я обычно предпочитаю стиль, использованный в ***Date3***.

Модификаторы доступа можно использовать несколько раз в одном и том же объявлении класса. Например:

```
class Date4 {
public
        Date4 (int dd, int mm, int yy),
private:
        int d, m, y;
public
        void add_year (int n);          // прибавить n лет
},
```

Введение нескольких открытых разделов (как в ***Date4***) часто запутывает программиста. То же самое можно сказать и о нескольких закрытых разделах. Однако разрешение множества модификаторов доступа в классе полезно при автоматической генерации кода.

10.2.9. Определение функции в классе

Функция, определенная в пределах определения класса (а не просто там объявленная), является встроенной функцией-членом. То есть, определение функций-членов в классе предназначено для небольших по размеру и часто используемых функций. Так же, как и определение класса, частью которого она является, определение в теле класса функции-члена может повторяться в нескольких единицах трансляции (в результате использования ***#include***). Как и в случае с классом, ее смысл должен быть одинаков везде, где она используется (§ 9.2.3).

Помещение определений членов данных в конец класса может привести к небольшим проблемам, связанным с открытыми встроенными функциями, которые ссылаются на представление. Рассмотрим пример:

```
class Date {
public
        int day () const { return d, }          // возвращает Date::d
        // ...
private:
        int d, m, y;
},
```

Здесь приведен совершенно правильный код на C++, потому что функция, объявленная в классе, может обращаться к любому члену класса так, как будто весь класс был полностью определен до рассмотрения тела функции. Однако это может быть не совсем очевидно читающему программу.

Поэтому, я обычно либо помещаю данные в начало, либо определяю встраиваемые функции после класса. Например:

```
class Date {
public.
        int day () const;
        // ...
private·
        int d, m, y;
};

inline int Date::day () const { return d, }
```

10.3. Эффективные типы, определяемые пользователем

В предыдущем разделе обсуждалось проектирование класса ***Date*** в контексте описания базовых средств языка для определения классов. В этом разделе я смещаю акцент и описываю проектирование простого и эффективного класса ***Date***, а также показываю, как средства языка способствуют такому проектированию.

Небольшие интенсивно используемые абстракции являются типичными для многих приложений. Примерами могут служить латинские буквы, китайские иероглифы, целые, числа с плавающей точкой, комплексные числа, точки, указатели, координаты, преобразования, пары *(указатель, смещение)*, даты, времена, диапазоны, связи, ассоциации, узлы, пары *(значение, единица измерения)*, расположение на диске, местонахождение исходного кода, символы набора BCD, валюты, строки, прямоугольники, масштабируемые числа с фиксированной точкой, дробные числа, строки символов, вектора и массивы. Каждое приложение использует что-нибудь из этого перечня. Довольно часто многие из этих простых конкретных типов используются весьма интенсивно. Типичное приложение применяет некоторые из них непосредственно и гораздо большее количество — косвенно, через библиотеки.

C++, как и другие языки программирования, непосредственно поддерживает некоторые из этих абстракций. Однако большинство подобных абстракций не поддерживается, и не может поддерживаться непосредственно, потому что их слишком много. Более того, разработчик языка высокого уровня не может знать о всех запросах

каждого приложения. Следовательно, пользователю должен быть предоставлен механизм определения небольших конкретных типов. Такие типы называются конкретными типами или конкретными классами для того, чтобы отличить их от абстрактных классов (§ 13.3) и иерархий классов (§ 12.2.4, § 12.4).

Одной из главных целей при проектировании C++ было добиться качественной поддержки определения и эффективного использования типов, определяемых пользователем. Они являются фундаментом элегантного программирования. Как обычно, простое и обыденное статистически является гораздо более значимым, чем сложное и утонченное.

В свете вышесказанного давайте создадим более качественный класс для представления дат:

```
class Date {
public                                  // открытый интерфейс
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec },

    class Bad_date { },                 // класс исключений

// 0 означает «используй значение по умолчанию»
    Date (int dd=0, Month mm=Month (0), int yy=0),

// функции доступа к дате
    int day () const,
    Month month () const,
    int year () const,

    string string_rep () const,         // строковое представление
    void char_rep (char s[]) const,     // строковое представление в стиле C

    static void set_default (int, Month, int),

// функции-модификаторы даты
    Date& add_year (int n),             // прибавить n лет
    Date& add_month (int n),            // прибавить n месяцев
    Date& add_day (int n),              // прибавить n дней

private
    int d, m, y,                        // представление
    static Date default_date,
},
```

Этот набор операций весьма характерен для типа, определяемого пользователем:

[1] Конструктор, определяющий как должны быть проинициализированы объекты/переменные данного типа.

[2] Набор функций доступа (функций-селекторов). Эти функции имеют модификатор ***const***, который указывает, что они не должны изменять состояние объектов/переменных, для которых они вызваны.

[3] Набор функций-модификаторов. При их использовании не возникает необходимости разбираться в деталях представления или долго думать о смысле того или иного члена данных.

[4] Набор неявно определенных операций, позволяющих свободно копировать объекты.

[5] Класс ***Bad_date***, используемый для сообщений об ошибках путем возбуждения исключений.

Я определил тип ***Month***, чтобы не путаться в том, пишется ли, например, 7 июня, как ***Date*** (***6***, ***7***) (стиль, принятый в Америке) или ***Date*** (***7***, ***6***) (европейский стиль). Я p_170.pstакже использовал механизм аргументов по умолчанию.

Я подумывал о том, чтобы ввести отдельные типы ***Day*** и ***Year*** во избежание возможной путаницы типа ***Date*** (***1995***, ***jul***, ***27***) и ***Date*** (***27***, ***jul***, ***1995***). Однако эти типы не настолько полезны, как ***Month***. В любом случае, почти все ошибки такого рода быстро обнаруживаются во время выполнения — дата 26 июля 27 года далеко не часто встречается в моей повседневной работе. Что делать с датами до 1800 года (или где-то около того) — вопрос довольно тонкий, и мы оставим его экспертам-историкам. Более того, число нельзя до конца проверить отдельно от значений месяца и года. В § 11.7.1 приводится способ определения удобного в использовании типа ***Year***.

Где-то должна быть определена корректная дата по умолчанию. Например:

```
Date Date::default_date(22,jan,1901);
```

Я убрал кэш, использовавшийся в § 10.2.7.1, потому что для такого примитивного типа он просто не нужен. При необходимости его можно добавить как деталь реализации — на пользовательский интерфейс это не повлияет.

Приведем небольшой пример использования ***Date***:

```
void f(Date& d)
{
    Date lvb_day = Date(16,Date::dec,d.year());
    if (d.day()==29 && d.month()==Date::feb) {
        //...
    }
    if (midnight()) d.add_day(1);
    cout << "следующий день: " << d+1 << '\n';
}
```

В примере предполагается, что операторы вывода << и сложения + объявлены в ***Date***. Я делаю это в § 10.3.3.

Обратите внимание на форму записи ***Date::feb***. Функция *f*() не является членом ***Date***, поэтому в ней требуется явно указывать, что имеется в виду ***feb*** из ***Date***, а не какой-либо другой объект.

Зачем нужно определять специальный тип для такого простого понятия, как дата? В конце концов, мы могли бы определить структуру:

```
struct Date {
    int day, month, year;
};
```

и позволить программистам решать, что с ней делать. Однако если бы мы поступили таким образом, каждый пользователь должен был бы манипулировать компонентами ***Date*** непосредственно, либо реализовать для этого отдельные функции. В результате, понятие даты было бы «размазано» по всей системе; его было бы сложно понимать, документировать и модифицировать. Реализация концепции в виде простой структуры неизбежно привела бы к дополнительным затратам со стороны каждого пользователя этой структуры.

Хотя тип ***Date*** и выглядит простым, придется затратить некоторые усилия на его реализацию. Например, при увеличении даты надо помнить о високосных годах, надо не забывать, что в месяцах содержится разное количество дней и т. д. (§ 10.6[1]). Кроме того, представление в виде «день/месяц/год» является недостаточным для многих приложений. Однако если бы мы захотели его изменить, нам потребовалось бы модифицировать только набор соответствующих функций. Например, для того чтобы представить ***Date*** в виде количества дней до или после 1 января 1970 года, нам пришлось бы изменить только функции-члены ***Date*** (§ 10.6[2]).

10.3.1. Функции-члены

Естественно, где-то должна находиться реализация каждой функции-члена. Приведем пример определения конструктора ***Date***:

```
Date::Date (int dd, Month mm, int yy)
{
        if (yy == 0) yy = default_date.year ();
        if (mm == 0) mm = default_date.month ();
        if (dd == 0) dd = default_date.day ();

        int max;

        switch (mm) {
        case feb:
                max = 28+leapyear (yy);
                break;
        case apr: case jun: case sep: case nov:
                max = 30;
                break;
        case jan: case mar: case may: case jul: case aug: case oct: case dec:
                max = 31;
                break;
        default:
                throw Bad_date ();      // кто-то что-то напутал
        }

        if (dd<1 || max<dd) throw Bad_date ();

        y = yy;
        m = mm;
        d = dd;
}
```

Конструктор проверяет, имеет ли дата допустимое значение. Если нет, например в случае ***Date*** (***30, Date::feb, 1994***), он возбудит исключение (§ 8.3, глава 14), что означает: случилось нечто, что нельзя проигнорировать. Если данные имеют допустимые значения, производится очевидная инициализация. Инициализация является относительно сложной операцией, потому что включает в себя проверку корректности данных. Это весьма типично. С другой стороны, после того как объект типа ***Date*** создан, им можно пользоваться и копировать без дополнительных проверок. Другими словами, конструктор устанавливает инвариант класса (в данном случае, это означает допустимость даты). Другие функции-члены могут полагаться на этот инвариант

и должны сохранять его. Такая техника проектирования может значительно упростить код (см. § 24.3.7.1).

Я использую значение ***Month*** (*0*) — что не соответствует ни одному месяцу — в качестве указания «воспользоваться датой по умолчанию». Я мог бы определить элемент перечисления ***Month***, чтобы выразить это. Но я решил, что лучше воспользоваться очевидно недопустимым значением для указания на дату по умолчанию, вместо того, чтобы создавать видимость существования 13 месяцев в году. Обратите внимание, что нулем в этой ситуации можно пользоваться, потому что он гарантированно находится внутри диапазона значений перечисления ***Month*** (§ 4.8).

Я подумывал о выделении проверки данных в отдельную функцию ***is_date*** (). Однако я счел результирующий код более сложным и менее надежным, чем код, использующий исключение. Например, предположим, что в ***Date*** определена операция ввода >>:

```
void fill (vector<Date>& aa)
{
    while (cin) {
        Date d;
        try {
            cin >> d;
        }
        catch (Date::Bad_date) {
            // моя обработка ошибки
            continue;
        }
        aa.push_back (d);        // см. § 3.7.3
    }
}
```

Как это обычно и происходит в случаях с такими простыми конкретными типами, сложность определения функций-членов колеблется от «тривиально» до «не слишком сложно». Например:

```
inline int Date..day () const
{
    return d;
}

Date& Date::add_month (int n)
{
    if (n==0) return *this;

    if (n>0) {
        int delta_y = n/12;
        int mm = m+n%12;
        if (12 < mm) {          // обратите внимание: int(dec) == 12
            delta_y++;
            mm -= 12;
        }

    // обработка ситуации, когда в Month(mm) не существует дня d

    y += delta_y;
    m = Month (mm);
```

```
        return *this;
        }
    // обработка отрицательного значения n
    return *this
}
```

10.3.2. Функции-помощники

Как правило, у класса есть набор функций, связанных с ним, но не требующих определения в классе, потому что они не нуждаются в непосредственном доступе к представлению. Например:

```
int diff(Date a, Date b);          // число дней в диапазоне [a, b) или [b, a)
bool leapyear(int y);              // год високосный?
Date next_weekday(Date d);
Date next_saturday(Date d);
```

Определение таких функций в самом классе усложнило бы интерфейс класса и потенциально увеличило бы количество функций, которые пришлось бы просматривать при изменении представления.

Как эти функции «связаны» с классом ***Date***? Традиционно, объявления такого рода функций просто помещались в тот же файл, что и объявление класса ***Date***, и пользователи, которым потребовался ***Date***, делали их доступными путем включения этого файла, определяющего интерфейс (§ 9.2.1). Например:

```
#include "Date.h"
```

Вместо использования ***Date.h*** — или в качестве альтернативы — мы можем установить связь явно, поместив класс, вместе с его функциями-помощниками, в пространство имен (§ 8.2):

```
namespace Chrono {                 // средства работы со временем
        class Date { /* ... */ }

        int diff(Date a, Date b);
        bool leapyear(int y);
        Date next_weekday(Date d);
        Date next_saturday(Date d);
        // ...
}
```

Пространство имен ***Chrono***, естественно, будет содержать в себе связанные классы, такие как ***Time*** и ***Stopwatch*** и их функции-помощники. Использование пространства имен только для одного класса обычно является избыточным и вызывает определенные неудобства.

10.3.3. Перегрузка операторов

Часто бывает полезно иметь функции, обеспечивающие привычную форму записи. Например, функция ***operator==*** определяет оператор проверки на равенство == для работы с ***Date***:

```
inline bool operator== (Date a, Date b)        // проверка на равенство
{
    return a.day ()==b.day () && a.month ()==b.month () && a.year ()==b.year ();
}
```

Другими очевидными кандидатами являются:

```
bool operator!= (Date, Date);                  // не равно
bool operator< (Date, Date);                   // меньше
bool operator> (Date, Date);                   // больше
// ...

Date operator++ (Date& d);                     // инкремент
Date operator-- (Date& d);                     // декремент

Date operator+= (Date& d, int n);              // прибавить n дней
Date operator-= (Date& d, int n);              // вычесть n дней

Date operator+ (Date d, int n);                // прибавить n дней
Date operator- (Date d, int n);                // вычесть n дней

ostream& operator<< (ostream&, Date d);        // вывести d
istream& operator>> (istream&, Date& d);       // считать в d
```

Для класса **Date** эти операторы можно рассматривать просто как удобство. Однако для многих типов — таких как комплексные числа (§ 11.3), вектора (§ 3.7.1) и объекты-функции (§ 18.4) — использование стандартных операторов настолько привычно, что их определение становится просто обязательным. Перегрузка операторов обсуждается в главе 11.

10.3.4. Роль конкретных классов

Я называю простые типы, определяемые пользователем, такие как **Date**, *конкретными типами*, чтобы отличать их от абстрактных классов (§ 2.5.4) и иерархии классов (§ 12.3), а также, чтобы подчеркнуть их сходство со встроенными типами, такими как **int** и **char**. Конкретные типы называют еще *типами значений*, а их использование — *программированием, ориентированным на значения*. Модель использования и «философия» проектирования конкретных типов сильно отличается от того, что часто называют объектно-ориентированным программированием (§ 2.6.2).

Конкретные типы служат для быстрого и эффективного решения одной небольшой задачи. Как правило, пользователю не предоставляется средств для изменения поведения конкретного типа. В частности, конкретные типы не демонстрируют полиморфное поведение (см. § 2.5.5, § 12.2.6).

Если вам не нравятся некоторые детали конкретного типа, вы создаете новый с желаемым поведением. Если вы хотите «повторно использовать» конкретный тип, вы используете его при реализации вашего нового типа точно так же, как вы могли бы применить **int**. Например:

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
```

```
        Date_and_time (Date d, Time t);
        Date_and_time (int d, Date::Month m, int y, Time t);
        // ...
};
```

Чтобы определить новые типы на основе конкретного типа путем описания соответствующих различий, можно воспользоваться механизмом производных классов, обсуждаемым в главе 12. Примером является определение ***Vec*** из ***vector*** (§ 3.7.2).

При наличии достаточно хорошего компилятора конкретные классы типа ***Date*** не вызывают скрытых дополнительных затрат памяти или увеличения времени выполнения. Размер конкретного типа известен во время компиляции, поэтому объекты могут быть помещены в стек (то есть не используется выделение свободной памяти). Способ хранения каждого объекта известен во время компиляции, поэтому встраивание операций достигается тривиальным образом. Аналогично, совместимость с другими языками, типа C и Fortran, достигается без специальных усилий.

Хороший набор конкретных типов может создать фундамент приложения. Отсутствие подходящих «маленьких эффективных типов» в приложении может привести к увеличению времени выполнения и к неэффективному расходованию памяти в результате использования чрезмерно обобщенных и больших классов. С другой стороны, отсутствие конкретных типов может привести к запутанным программам и потерям времени, когда каждый программист пишет код для непосредственного манипулирования «простыми и часто используемыми» структурами данных.

10.4. Объекты

Объекты могут создаваться несколькими способами. Некоторые объекты являются локальными переменными, другие — глобальными, третьи — членами классов и т. д. В данном разделе обсуждаются эти альтернативы, правила работы с объектами, конструкторы, используемые для инициализации объектов, и деструкторы, применяемые для «очистки» объектов перед тем, как они станут недоступными.

10.4.1. Деструкторы

Конструкторы инициализируют объект. Другими словами, они создают среду, в которой работают функции-члены. Иногда создание такой среды подразумевает захват каких-то ресурсов — таких как файл, блокировка или память, которые должны быть освобождены после их использования (§ 14.4.7). В результате, некоторым классам требуется функция, которая будет гарантированно вызвана при уничтожении объекта, аналогично конструктору, который гарантированно вызывается при создании объекта. Как вы догадываетесь, такие функции называются *деструкторами*. Они, как правило, очищают память и освобождают ресурсы. Деструкторы вызываются неявно, когда автоматическая переменная выходит из области видимости, удаляется объект, хранящийся в свободной памяти и т. д. Только в очень необычных ситуациях пользователю требуется явно вызывать деструктор (§ 10.4.11).

Наиболее часто деструктор используется для освобождении памяти, выделенной конструктором. Рассмотрим простую таблицу элементов типа ***Name***. Конструктор класса ***Table*** должен выделить память для хранения элементов. После того как таблица уничтожена каким-либо способом, мы должны быть уверены, что эта память будет

освобождена для дальнейшего использована. Мы можем добиться этого, реализовав специальную функцию, дополняющую конструктор:

```
class Name {
        const char* s;
        //...
};

class Table {
        Name* p;
        size_t sz;
public:
        Table (size_t s = 15) { p = new Name[sz = s]; }    // конструктор

        ~Table () { delete[] p; }                          // деструктор

        Name* lookup (const char*);
        bool insert (Name*);
};
```

В записи деструктора ***~Table ()*** используется символ дополнения ~ (тильда) в качестве напоминания о его связи с конструктором ***Table ()***.

Комплиментарные пары конструктор/деструктор являются типичным механизмом в C++ для выражения понятия объекта переменного размера. Контейнеры стандартной библиотеки, такие как ***map***, используют различные варианты этой техники для выделения памяти под свои элементы, так что последующее обсуждение иллюстрирует технику, с которой вы имеете дело каждый раз, когда используете стандартный контейнер (включая стандартный класс ***string***). Обсуждение применимо и к типам без деструкторов. Такие типы можно рассматривать как типы, деструкторы которых ничего не делают.

10.4.2. Конструкторы по умолчанию

Аналогичным образом можно полагать, что многие типы имеют конструкторы по умолчанию. Конструктор по умолчанию является конструктором, вызываемым без аргумента. В примере, приведенном выше, ***15*** является аргументом по умолчанию, поэтому ***Table::Table (size_t)*** является конструктором по умолчанию. Если пользователь объявил конструктор по умолчанию, он и будет задействован. В противном случае (и если пользователь не объявил другие конструкторы) компилятор попытается при необходимости сгенерировать конструктор по умолчанию. Конструктор по умолчанию, сгенерированный компилятором, неявно вызывает конструкторы по умолчанию для членов класса и конструкторы базовых классов (§ 12.2.2). Например:

```
struct Tables {
        int i;
        int vi[10];
        Table t1;
        Table vt[10];
};

Tables tt;
```

В этом примере переменная *tt* будет проинициализирована сгенерированным конструктором по умолчанию, который вызовет *Table(15)* для *tt.t1* и каждого элемента *tt.vt*. С другой стороны, *tt.i* и элементы *tt.vi* не проинициализированы, потому что их тип не является классом. Причина различной обработки классов и встроенных типов заключается в требовании совместимости с C и в боязни вызвать дополнительные затраты времени на этапе выполнения.

Так как константы и ссылки должны быть проинициализированы (§ 5.5, § 5.4), класс, содержащий члены, являющиеся константами или ссылками, не может быть сконструирован по умолчанию, если только программист не предоставил явно конструктор (§ 10.4.6.1). Например:

```
struct X {
        const int a;
        const int& r;
};

X x;        // ошибка: нет конструктора по умолчанию для X
```

Конструкторы по умолчанию можно вызывать явно (§ 10.4.10). Встроенные типы также имеют конструкторы по умолчанию (§ 6.2.8).

10.4.3. Конструирование и уничтожение

Рассмотрим различные способы создания и последующего уничтожения объектов. Объект может быть создан в качестве:

- § 10.4.4 Именованного автоматического объекта, создаваемого каждый раз, когда встречается его объявление во время выполнения программы и уничтожаемого при каждом выходе из блока, в котором он объявлен.
- § 10.4.5 Объекта в свободной памяти, создаваемого при помощи оператора *new* и уничтожаемого оператором *delete*.
- § 10.4.6 Нестатического члена-объекта, который создается и уничтожается тогда, когда создается и уничтожается содержащий его объект.
- § 10.4.7 Элемента массива, который создается и уничтожается тогда, когда создается и уничтожается массив, элементом которого он является.
- § 10.4.8 Локального статического объекта, который создается, когда его объявление встречается первый раз при выполнении программы и уничтожается один раз, при завершении программы.
- § 10.4.9 Глобального объекта, объекта в пространстве имен или статического объекта класса, которые создаются один раз «во время запуска программы» и уничтожаются один раз, при ее завершении.
- § 10.4.10 Временного объекта, который создается как часть вычисления выражения и уничтожается по завершении вычисления всего выражения.
- § 10.4.11 Объекта, помещенного в память, выделенную функцией пользователя, учитывающей аргументы, которые передаются ей операцией выделения памяти.
- § 10.4.12 Члена объединения *union*, который не может иметь ни конструктора, ни деструктора.

Список отсортирован (приблизительно) в порядке важности. В следующих подразделах объясняются эти способы создания объектов и их использования.

10.4.4. Локальные переменные

Конструктор локальной переменной вызывается каждый раз, когда управление передается инструкции, содержащей объявление этой локальной переменной. Деструктор локальной переменной выполняется каждый раз, когда происходит выход из блока, содержащего объявление локальной переменной. Деструкторы локальных объектов выполняются в порядке, противоположном выполнению их конструкторов. Например:

```
void f(int i)
{
        Table aa;
        Table bb;
        if (i>0) {
                Table cc;
                //...
        }
        Table dd;
        //...
}
```

В этом примере при каждом вызове *f*() переменные *aa*, *bb* и *dd* создаются именно в таком порядке и уничтожаются каждый раз при выходе из *f*() в порядке *dd*, *bb*, *aa*. Если во время вызова выполнится условие *i>0*, переменная *cc* будет создана после *bb* и уничтожена до создания *dd*.

10.4.4.1. Копирование объектов

Если *t1* и *t2* являются объектами класса *Table*, выражение *t2=t1* по умолчанию означает почленное копирование *t1* в *t2* (§ 10.2.5). При наличии у объекта членов, являющихся указателями, такая интерпретация присваивания может вызвать неожиданный (и обычно нежелательный) эффект. Почленное копирование обычно является неправильным при копировании объектов, имеющих ресурсы, управляемые парой конструктор/деструктор. Например:

```
void h()
{
        Table t1;
        Table t2 = t1,          // копирующая инициализация —.проблема
        Table t3;

        t3 = t2;                // копирующее присваивание — проблема
}
```

В этом примере конструктор *Table* по умолчанию будет вызван дважды — по одному разу для *t1* и *t3*. Он не вызовется для *t2*, потому что эта переменная проинициализирована при помощи копирования. Однако деструктор *Table* вызывался три раза: для *t1*, *t2* и *t3*! По умолчанию копирование интерпретируется как почленное копирование, поэтому *t1*, *t2* и *t3* к концу *h*() будут содержать указатели на массив имен, выделенный в свободной памяти при создании *t1*. Не осталось указателя на массив имен, выделенный при создании *t3*, потому что он перезаписан присваиванием *t3=t2*. В результате при отсутствии автоматической сборки мусора (§ 10.4.5), эта память будет навсегда потеряна для программы. С другой стороны, массив, созданный для *t1*, будет и в *t1*, и в *t2*, и в *t3*, поэтому он будет трижды удаляться. Результат непредсказуем и, вероятно, приведет к катастрофе.

Можно избежать подобных аномалий, определив, что понимать под копированием *Table*:

```
class Table {
        //...
        Table (const Table&);                   // копирующий конструктор
        Table& operator= (const Table&)         // копирующее присваивание
};
```

Программист может определить любой подходящий смысл этих операций копирования, но традиционным решением является поэлементное копирование хранимых элементов (или по крайней мере, создание у пользователя иллюзии, что такое копирование произведено; см. § 11.12). Например:

```
Table::Table (const Table& t)                   // копирующий конструктор

{
        p = new Name[sz=t.sz];
        for (int i=0, i<sz; i++) p[i] = t.p[i];
}
Table& Table::operator= (const Table& t)        // присваивание
{
        if (this != &t) {                       // чтобы уберечься от присваивания самому себе: t=t
                delete[] p;
                p = new Name[sz=t.sz];
                for (int i=0; i<sz; i++) p[i] = t.p[i];
        }
        return *this;
}
```

Как это и бывает в большинстве случаев, копирующий конструктор и копирующее присваивание значительно отличаются. Основная причина этого состоит в том, что копирующий конструктор инициализирует «чистую» (неинициализированную) память, в то время как копирующий оператор присваивания должен правильно работать с уже созданным объектом.

В некоторых случаях присваивание может быть оптимизировано, но основная стратегия при реализации оператора присваивания проста: защита от присваивания самому себе, удаление старых элементов, инициализация и копирование новых элементов. Как правило, все нестатические члены должны быть скопированы (§ 10.4.6.3). Для создания отчета об ошибке копирования могут быть использованы исключения (§ 14.4.6.2). Методику создания операций копирования, безопасных при исключениях, см. в § Д.3.3.

10.4.5. Свободная память

Для объекта, создаваемого в свободной памяти, вызывается конструктор класса, указанного в операторе ***new***. Такой объект существует до тех пор, пока к указателю на него не будет применен оператор ***delete***. Рассмотрим пример:

```
int main ()
{
        Table* p = new Table;
```

```
    Table* q = new Table;

    delete p;
    delete p;          // вероятно вызовет ошибку во время выполнения
}
```

Конструктор ***Table*::*Table*** () вызывается дважды, так же как и деструктор ***Table*::~*Table*** (). К сожалению, операторы ***new*** и ***delete*** в примере не соответствуют друг другу, поэтому объект, на который указывает ***p***, удаляется дважды, а объект, на который указывает ***q***, не удаляется вовсе. Неудаление объекта обычно не является ошибкой с точки зрения языка — это просто потеря памяти. Однако если предполагается, что программа будет выполняться в течение длительного времени, такие потери памяти являются серьезными и трудно отлавливаемыми ошибками. Существуют средства для обнаружения подобных утечек памяти. Удаление ***p*** дважды является серьезной ошибкой — последствия не определены и, скорее всего, катастрофичны.

Некоторые реализации C++ автоматически освобождают память, занимаемую недоступными из программы объектами (то есть реализуют сборщики мусора), но их поведение не стандартизовано. Даже при наличии сборщика мусора, ***delete*** вызовет деструктор, если таковой определен, поэтому удаление объекта дважды остается серьезной ошибкой. Во многих случаях это может быть и просто мелким неудобством. В частности, когда известно, что сборщик мусора существует, деструкторы, осуществляющие лишь управление памятью, могут быть опущены. Такое упрощение достигается за счет потери переносимости, а для некоторых программ и за счет возможного увеличения времени выполнения и потери предсказуемости поведения (§ B.9.1).

После применения оператора ***delete*** к объекту любая попытка обращения к этому объекту является ошибкой. К сожалению, реализации не могут надежно обнаруживать такие ошибки.

Пользователь может сам определить, каким образом ***new*** осуществляет выделение памяти и каким образом ***delete*** ее освобождает (см. § 6.2.6.2 и § 15.6). Кроме того, можно указать способы взаимодействия выделения памяти, инициализации (конструирования) и исключений (см. § 14.4.5 и § 19.4.5). Массивы в свободной памяти обсуждаются в § 10.4.7.

10.4.6. Объекты в качестве членов

Рассмотрим класс, который можно использовать для хранения информации о небольшой организации:

```
class Club {
    string name;
    Table members;
    Table officers;
    Date founded;
    // ...
    Club (const string& n, Date fd);
};
```

Конструктор ***Club*** в качестве аргументов получает имя клуба и дату основания. Аргументы конструкторов-членов указываются в списке инициализации членов в определении конструктора объемлющего класса. Например:

```
Club::Club (const string& n, Date fd)
        : name (n), members (), officers (), founded (fd)
{
        // ...
}
```

В начале списка инициализации членов стоит двоеточие, инициализаторы отдельных членов разделены запятыми.

Конструкторы членов вызываются до вызова конструктора самого класса. Конструкторы вызываются в том порядке, в котором они объявлены в классе, а не в том, в котором они записаны в списке инициализации. Во избежание путаницы, лучше записывать список в порядке, соответствующем объявлению. Деструкторы членов вызываются в порядке, обратном вызовам конструкторов после того, как будет выполнено тело деструктора класса.

Если конструктор члена не нуждается в аргументах, член можно не указывать в списке инициализации:

```
Club::Club (const string& n, Date fd)
        : name (n), founded (fd)
{
        // ...
}
```

Это эквивалентно предыдущей версии. В каждом из этих случаев член ***Club::officers*** создается конструктором ***Table::Table*** с аргументом по умолчанию, равным ***15***.

Когда уничтожается объект класса, содержащий объекты классов, сначала вызывается деструктор (если таковой имеется) класса, а потом деструкторы объектов-членов в порядке, обратном порядку объявления. Конструктор создает окружение исполнения для функций-членов снизу вверх (сначала члены). Деструктор уничтожает его сверху вниз (члены последними).

10.4.6.1. Необходимая инициализация членов

Инициализаторы членов имеют большое значение для типов, у которых инициализация отличается от присваивания — то есть для объектов-членов классов без конструкторов по умолчанию, для константных членов и для членов, являющихся ссылками. Например:

```
class X {
        const int i;
        Club c;
        Club& pc;
        // ...
        X (int ii, const string& n, Date d, Club& c) : i (ii), c (n, d), pc (c) { }
};
```

Не существует никакого другого способа инициализации таких членов, и отсутствие инициализации объектов таких типов является ошибкой. Однако для большинства типов программист имеет выбор между использованием инициализатора и присваиванием. В этом случае я предпочитаю воспользоваться синтаксисом инициализатора члена, делая таким образом явным тот факт, что инициализация произведена. Часто инициализация является и более эффективной. Например:

```
class Person {
        string name;
        string address;
        // ...
        Person (const Person&);
        Person (const string& n, const string& a);
};

Person::Person (const string& n, const string& a)
        : name (n)
{
        address = a;
}
```

В этом примере ***name*** инициализируется копией ***n***. С другой стороны, ***address*** сначала инициализируется пустой строкой, а затем ей присваивается копия ***a***.

10.4.6.2. Члены-константы

Можно проинициализировать член, являющийся статической константой интегрального типа, добавив к объявлению члена *константное выражение* в качестве инициализирующего значения. Например:

```
class Curious {
        static const int c1 = 7;          // правильно, но помните об определении
        static int c2 = 11;               // ошибка: не константа
        const int c3 = 13;                // ошибка: не статическая константа
        static const int c4 = f(17);      // ошибка: инициализатор не константа
        static const float c5 = 7.0;      // ошибка: не интегральный тип
        // ...
};
```

Если (и только если) вы используете инициализированный член таким образом, что требуется хранить его в памяти, как объект, член должен быть где-нибудь (один раз) определен. Инициализатор не может повторяться:

```
const int Curious::c1;          // обязательно, но не повторяйте здесь инициализатор
const int* p = &Curious::c1;    // правильно: Curious::c1 был определен
```

С другой стороны, вы можете использовать элементы перечисления (§ 4.8, § 14.4.6, § 15.3) в качестве символической константы внутри определения класса. Например:

```
class X{
        enum { c1 = 7, c2 = 11, c3 = 13, c4 = 17 };
        // ...
};
```

В этом случае у вас не возникает искушения объявить переменные, числа с плавающей точкой и т. д. внутри класса.

10.4.6.3. Копирование членов

Копирующие конструкторы по умолчанию и копирующие операторы присваивания по умолчанию (§ 10.4.4.1) копируют все элементы класса. Если такое копирование не может быть выполнено, попытка копирования объекта класса является ошибкой. Например:

```
class Unique_handle {
private:    // операции копирования закрыты с целью
            // предотвращения копирования (§ 11.2.2)
        Unique_handle (const Unique_handle&),
        Unique_handle& operator= (const Unique_handle&);
public:
        // ...
};

struct Y {
        // ...

        Unique_handle a;        // требует явной инициализации
};

Y y1,
Y y2 = y1;                      // ошибка: невозможно скопировать Y::a
```

Кроме того, присваивание по умолчанию не может бы сгенерировано, если нестатический член является ссылкой, константой или типом, определяемым пользователем, не имеющим копирующего оператора присваивания.

Обратите внимание, что в результате работы копирующего конструктора по умолчанию, член, являющийся ссылкой, в обеих копиях ссылается на один и тот же объект. Это может привести к проблеме, если объект, на который он ссылается, предполагается удалить.

При написании копирующего конструктора мы должны аккуратно скопировать все элементы, которые действительно надлежит скопировать. Элементы инициализируются по умолчанию, но очень часто это не то, что требуется в копирующем конструкторе. Например:

```
Person::Person (const Person& a) : name (a.name) { }        // осторожно!
```

В этом примере я забыл скопировать **address**, поэтому **address** инициализируется по умолчанию пустой строкой. При добавлении нового члена класса всегда проверяйте, не надо ли модифицировать определяемые пользователем конструкторы с учетом инициализации и копирования нового члена.

10.4.7. Массивы

Если объект класса может быть создан без явного задания инициализирующего значения, то можно определить массив элементов этого класса. Например:

```
Table tbl[10];
```

В результате будет создан массив из десяти объектов типа **Table**, и каждый элемент будет проинициализирован при помощи вызова конструктора **Table**::**Table** () с аргументом по умолчанию равным **15**.

Не существует способа явного указания аргументов конструктора (за исключением использования списка инициализации (§ 5.2.1, § 18.6.7)) при объявлении массива. Если вам совершенно необходимо проинициализировать члены массива различными значениями, вы можете написать конструктор по умолчанию, который непосредственно или косвенно считывает и записывает нелокальные данные. Например:

```
class Ibuffer {
        string buf;
public:
        Ibuffer() { cin >> buf; }
        // ...
};

void f()
{
        Ibuffer words[100];     // каждое слово инициализируется из cin
        // ...
}
```

Как правило, лучше избегать подобных сложностей.

Деструктор для каждого созданного элемента массива вызывается при уничтожении массива. Это происходит неявно для массивов, память под которые не выделялась оператором ***new***. Как и в C, в C++ не различаются указатель на отдельный объект и указатель на первый элемент массива (§ 5.3). Поэтому программист должен указать, удаляется ли массив или отдельный объект. Например:

```
void f(int sz)
{
        Table* t1 = new Table;
        Table* t2 = new Table[sz];
        Table* t3 = new Table;
        Table* t4 = new Table[sz];

        delete t1;          // правильно
        delete[] t2;        // правильно
        delete[] t3;        // неправильно
        delete t4;          // неправильно
}
```

То, как на самом деле выделяется память под массивы и отдельные объекты, зависит от реализации. Поэтому различные реализации будут реагировать по-разному на некорректное использование операторов ***delete*** и ***delete***[]. В простых и неинтересных случаях наподобие предыдущих, компилятор может сам обнаружить проблему, но как правило, неприятности происходят во время выполнения.

Специальный оператор ***delete***[] для массивов не является логически необходимым. Предположим однако, что от реализации управления свободной памятью потребовали хранения информации для каждого объекта о том, является ли он массивом или отдельным объектом. С пользователя было бы снято дополнительное бремя, но за счет значительных дополнительных затрат времени и памяти в некоторых реализациях C++.

Если массивы в стиле C покажутся вам слишком неудобными, воспользуйтесь вместо них классом, подобным ***vector*** (§ 3.7.1, § 16.3). Например:

```
void g()
{
        vector<Table> v(10);                        // нет необходимости в удалении
        vector<Table>* p = new vector<Table>(10);   //предпочтительнее использовать
                                                    //скалярный "delete", а не "delete[]"

        delete p;
}
```

Использование контейнера (вроде ***vector***) проще, чем запись пар ***new/delete***. Более того, ***vector*** обеспечивает безопасность исключений (приложение Д).

10.4.8. Локальная статическая память

Конструктор локального статического объекта (§ 7.1.2) вызывается один раз при первом выполнении инструкции, содержащей определение объекта. Рассмотрим пример:

```
void f(int i)
{
        static Table tbl;
        //...
        if (i) {
                static Table tbl2;
                //...
        }
}

int main ()
{
        f(0);
        f(1);
        f(2);
        //...
}
```

В этом примере конструктор ***tbl*** вызывается только при первом вызове *f*(). Так как переменная ***tbl*** объявлена статической, она не уничтожается по возвращению из *f*() и не создается повторно при следующем вызове *f*(). Ввиду того, что блок, содержащий объявление ***tbl2***, не выполняется при вызове *f*(*0*), ***tbl2*** не создается до вызова *f*(*1*). И эта переменная не конструируется снова при втором входе в блок.

Деструкторы локальных статических объектов вызываются в порядке, обратном созданию, при завершении программы (§ 9.4.1.1). В какой момент точно — не определено.

10.4.9. Нелокальная память

Переменная, определенная вне любой функции (то есть глобальная, объявленная в пространстве имен или статически в классе, § В.9) инициализируется (конструируется) до вызова ***main*** (). После выхода из ***main*** () будет вызван деструктор для каждой такой переменной. Динамическая компоновка слегка усложняет эту картину, откладывая инициализацию до того момента, когда код будет скомпонован в исполнимую программу.

Конструкторы нелокальных объектов в единице трансляции выполняются в порядке их определений. Рассмотрим пример:

```
class X {
        //...
        static Table memtbl;
};

Table tbl;

Table X::memtbl;
```

```
namespace Z {
        Table tbl2;
}
```

Порядок создания следующий: ***tbl***, затем ***X::memtbl*** и затем ***Z::tbl2***. Обратите внимание, что объявление (в отличие от определения), наподобие объявления ***memtbl*** в ***X***, не влияет на порядок создания. Деструкторы вызываются в порядке, обратном конструкторам: ***Z::tbl2***, ***X::memtbl***, ***tbl***.

Не дается никаких гарантий по поводу порядка конструирования нелокальных объектов из различных единиц компиляции. Например:

```
//file1.c:
        Table tbl1;

//file2.c
        Table tbl2;
```

Что будет создано раньше — ***tbl1*** или ***tbl2*** — зависит от реализации. Не гарантируется даже, что в одной и той же реализации порядок будет постоянным. Динамическая компоновка или даже небольшое изменение в процессе компиляции может изменить последовательность создания. Аналогично, порядок уничтожения также зависит от реализации.

Иногда при создании библиотеки необходимо или просто удобно изобрести тип с конструктором и деструктором, единственной целью которого является инициализация и последующая очистка. Такой тип будет использован только один раз: для выделения памяти под статический объект, так чтобы при этом вызвались конструктор и деструктор. Например:

```
class Zlib_init {
        Zlib_init ();       // подготавливает Zlib к использованию
        ~Zlib_init (),      // производит очистку после использования Zlib
};

class Zlib {
        static Zlib_init x;
        // ...
};
```

К сожалению, не гарантируется, что такой объект будет проинициализирован до его первого использования и уничтожен после последнего использования в программе, состоящей из раздельно компилируемых единиц. Конкретная реализация C++ может предоставить такую гарантию, но большинство из них этого не делает. Программист может обеспечить правильную инициализацию, придерживаясь обычной стратегии для локальных статических объектов: использовать индикатор первого вызова. Например:

```
class Zlib {
        static bool initialized;
        static void initialize () { /* инициализация */ initialized = true; }
public:
        // конструктор отсутствует

        void f ()
        {
```

```
            if (initialized==false) initialize ();
            //...
        }
        //...
},
```

Если во многих функциях требуется проверка индикатора первого вызова, этот метод может стать довольно утомительным, но тем не менее он работает. Эта техника использует тот факт, что статически размещаемые объекты без конструкторов инициализируются нулем. Действительно сложным является случай, когда первая операция критична по времени, и поэтому дополнительные затраты на проверку и возможную инициализацию могут оказаться серьезной проблемой. В подобных ситуациях потребуются дополнительные трюки (§ 21.5.2).

Альтернативным подходом в случае простых объектов является представление их в виде функций (§ 9.4.1):

```
int& obj () { static int x = 0, return x; }      // инициализация при первом вызове
```

Индикаторы первого вызова не решают всех возможных проблем. Например, можно создать объекты, которые ссылаются друг на друга во время конструирования. Таких случаев лучше избегать, но если такие объекты необходимы, их нужно конструировать аккуратно и поэтапно. Отметим также, что не существует простой аналогичной конструкции индикатора последнего вызова (см. § 9.4.1.1 и § 21.5.2).

10.4.10. Временные объекты

Временные объекты чаще всего возникают при вычислении арифметических выражений. Например, в некоторый момент вычисления ***x*y+z***, надо где-то сохранить промежуточный результат ***x*y***. За исключением случаев, когда скорость выполнения крайне важна (§ 11.6), программист редко обращает внимание на временные объекты. Но иногда приходится это делать (§ 11.6, § 22.4.7).

Если временный объект не связан со ссылкой или не используется для инициализации именованного объекта, он уничтожается по достижении конца полного выражения, в котором был создан. *Полное выражение* — это выражение, которое не является частью другого выражения.

Стандартный класс ***string*** имеет функцию-член ***c_str*** (), которая возвращает указатель на массив символов в стиле C с завершающим нулем (§ 3.5.1, § 20.4.1). Кроме того, определен оператор +, означающий конкатенацию строк. Эти операции являются очень важными при работе со строками. Однако при совместном их использовании, они могут привести к странным проблемам. Например:

```
void f(string& s1, string& s2, string& s3)
{
    const char* cs = (s1+s2).c_str ();
    cout << cs;
    if (strlen (cs= (s2+s3).c_str ())<8 && cs[0]=='a') {
        // использование cs
    }
}
```

Возможно, вашей первой реакцией будет «просто не пиши так, да и все», и я соглашусь. Однако такой код встречается, поэтому стоит знать, как он интерпретируется.

Создается временный объект класса ***string*** для хранения ***s1+s2***. Затем из объекта извлекается указатель на С-строку. Затем — в конце выражения — временный объект уничтожается. Где была размещена возвращаемая функцией ***c_str*** () С-строка? Вероятно, в части временного объекта, хранившего ***s1+s2***, но эта память не обязана существовать после уничтожения временной переменной. Следовательно, ***cs*** указывает на освобожденную память. Возможно, ***cout*** << ***cs*** и сработает так, как и ожидалось, но это будет просто удачей. Компилятор может обнаружить и предупредить о большинстве подобных проблем.

Пример с *if-инструкцией* гораздо тоньше. Условная инструкция отработает так, как и ожидалось, потому что полным выражением, в котором создается временная переменная для хранения ***s2+s3***, является само условие. Однако эта временная переменная будет уничтожена до выполнения следующей (контролируемой) инструкции, поэтому не гарантируется правильность последующего использования ***cs***.

Обратите внимание, что в этом случае, как и во многих подобных, проблема с временными переменными возникла из-за низкоуровнего использования типов данных высокого уровня. Более «чистый» стиль программирования приводит не только к более понятным фрагментам кода, но также позволяет полностью избежать проблем с временными переменными. Например:

```
void f(string& s1, string& s2, string& s3)
{
        cout << s1+s2;
        string s = s2+s3;
        if (s.length()<8 && s[0]=='a') {
                // использование s
        }
}
```

Временная переменная может использоваться в качестве инициализатора константной ссылки или именованного объекта. Например:

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
        const string& s = s1+s2;
        string ss = s1+s2;

        g(s, ss);       // мы можем использовать здесь s и ss
}
```

Все работает отлично. Временная переменная уничтожается, когда «ее» ссылка или именованный объект выходит из области видимости. Помните, что возврат из функции ссылки на локальную переменную является ошибкой (§ 7.3), и что неконстантная ссылка не может ссылаться на временную переменную (§ 5.5).

Временный объект также может быть создан при помощи явного вызова конструктора. Например:

```
void f(Shape& s, int x, int y)
{
```

```
        //...
        s.move(Point(x,y)),     // создается Point для передачи Shape::move()
        //...
}
```

Такие временные переменные уничтожаются точно таким же способом, как и неявно генерируемые.

10.4.11. Размещение объектов

По умолчанию оператор *new* создает объекты в свободной памяти. Что если бы мы захотели поместить такие объекты где-то в другом месте? Рассмотрим простой класс:

```
class X{
public:
        X(int),
        //...
};
```

Мы можем поместить объекты куда угодно, написав функцию выделения памяти, имеющую дополнительные аргументы, и затем задавая их при использовании оператора *new*:

```
// оператор явного размещения
void* operator new(size_t, void* p){ return p; }

// некоторый конкретный адрес
void* buf = reinterpret_cast<void*>(0xF00F);
// создание X в buf; вызывается operator new(sizeof(X), buf)
X* p2 = new(buf) X;
```

Из-за своего использования синтаксис *new(buf) X*, позволяющий задать дополнительные аргументы для *operator new()*, называется *синтаксисом размещения*. Обратите внимание, что каждый *operator new* получает в качестве первого аргумента размер, и что размер размещаемого объекта задается неявно (§ 15.6). Выбор функции *operator new()* для вызова оператора *new* подчиняется обычным правилам соответствия аргументов (§ 7.4); каждый *operator new()* имеет в качестве первого аргумента *size_t*[1].

Приведенный «оператор размещения» *operator new()* является простейшим таким распределителем (allocator). Он определен в стандартном заголовочном файле <*new*>.

Преобразование *reinterpret_cast* является самым подозрительным и потенциально наиболее опасным (§ 6.2.7). В большинстве случаев в результате его применения получается значение нужного типа с той же последовательностью битов, что и его аргумент. Этим преобразованием приходится пользоваться для традиционно опасных, зависящих от реализации, но иногда совершенно необходимых преобразований целых значений в указатели и наоборот.

Конструкция размещения *new* может использоваться для выделения памяти в конкретной области:

```
class Arena{
public:
```

[1] Тип *size_t* является типом значения, возвращаемого функцией *sizeof()*, см. § 16.1.2. — *Примеч. ред.*

```
        virtual void* alloc (size_t) = 0;
        virtual void free (void*) = 0,
        // ...
};

void* operator new (size_t sz, Arena* a)
{
        return a->alloc (sz),
}
```

Теперь объекты произвольных типов можно размещать в различных областях памяти («аренах»). Например:

```
extern Arena* Persistent;
extern Arena* Shared,

void g (int i)
{
        X* p = new (Persistent) X (i),          // X в Persistent
        X* q = new (Shared) X (i),              // X в Shared
        // ...
}
```

Помещение объекта в область, которая (непосредственно) не управляется стандартным распределителем свободной памяти подразумевает определенные действия при уничтожении объекта. Базовым механизмом для этого является явный вызов деструктора:

```
void destroy (X* p, Arena* a)
{
        p->~X ();           // вызов деструктора
        a->free (),         // освобождение памяти
}
```

Обратите внимание на то, что следует избегать явных вызовов деструкторов, равно как и использования *глобальных* распределителей памяти специального назначения, там где это только возможно. Но иногда они требуются. Например, было бы сложно реализовать обобщенный контейнер ***vector*** в духе стандартной библиотеки (§ 3.7.1, § 16.3.8) без использования явного вызова деструктора. Новичку же стоит дважды (лучше трижды) подумать, прежде чем вызывать деструктор явно — лучше проконсультироваться у более опытного коллеги.

См. в § 14.4.7 объяснения по поводу того, как размещение оператором ***new*** () взаимодействует с обработкой исключений.

Нет специального синтаксиса для размещения массивов. В нем нет особой необходимости, поскольку с помощью ***new*** можно размещать произвольные типы. Однако для массивов можно определить специальный оператор ***delete*** () (§ 19.4.5).

10.4.12. Объединения

Именованным объединением называется структура, в которой каждый член имеет один и тот же адрес (см. § B.8.2). Объединение может иметь функции-члены, но не статические члены.

Как правило, компилятор не может знать, какой член объединения используется; то есть тип объекта, хранящегося в объединении, неизвестен. Следовательно, объединение не может иметь члены с конструкторами и деструкторами. В противном случае было бы невозможно предохранить объект от разрушения или гарантировать, что будет вызван нужный деструктор, когда объединение выйдет за пределы области видимости.

Объединения лучше использовать в коде низкого уровня, или как часть реализации класса, который сохраняет информацию о том, что хранится в объединении (см. § 10.6[20]).

10.5. Советы

[1] Представляйте концепции в виде классов; § 10.1.

[2] Пользуйтесь открытыми данными (структурами) только тогда, когда вам нужны действительно только данные и нет никаких инвариантов для членов данных; § 10.2.8.

[3] Конкретные типы являются простейшими представителями классов. Там где это возможно, отдавайте предпочтение конкретным типам по сравнению как с более сложными классами, так и с обычными структурами данных; § 10.3

[4] Реализуйте функцию в качестве члена, только если ей требуется непосредственный доступ к представлению класса; § 10.3.2.

[5] Пользуйтесь пространством имен для явного выражения связи между классом и функциями-помощниками; § 10.3.2.

[6] Объявляйте функцию-член как константную (***const***), если она не должна модифицировать значение объекта; § 10.2.6.

[7] Объявляйте функцию-член как статическую (***static***), если ей требуется доступ к представлению класса, но ее не требуется вызывать для конкретного объекта класса; § 10.2.4.

[8] Пользуйтесь конструктором для установления инварианта класса; § 10.3.1.

[9] Если конструктору требуется какой-нибудь ресурс, в классе необходимо ввести деструктор для освобождения этого ресурса; § 10.4.1.

[10] Если в классе имеется член, являющийся указателем, ему требуются копирующие операции (копирующий конструктор и копирующее присваивание); § 10.4.4.1.

[11] Если в классе имеется член, являющийся ссылкой, ему вероятно потребуются копирующие операции (копирующий конструктор и копирующее присваивание); § 10.4.6.3.

[12] Если классу требуется копирующая операция или деструктор, ему вероятно потребуются конструктор, деструктор, копирующий конструктор и копирующее присваивание; § 10.4.4.1.

[13] Проверяйте, нет ли присваивания самому себе при копирующем присваивании; § 10.4.4.1.

[14] При написания копирующего конструктора, проследите за тем, чтобы были скопированы все требуемые элементы (не забывайте об инициализаторах по умолчанию); § 10.4.4.1.

[15] При добавлении в класс нового члена всегда проверяйте, не надо ли модифицировать для инициализации этого члена определяемые пользователем конструкторы; § 10.4.6.3.

[16] Пользуйтесь перечислениями для определения целых констант в объявлениях класса; § 10.4.6.2.
[17] Избегайте зависимости от порядка создания глобальных объектов и объектов в пространствах имен; § 10.4.9.
[18] Пользуйтесь индикаторами первого использования для минимизации зависимости от порядка; § 10.4.9.
[19] Помните, что временные объекты уничтожаются в конце обработки полного выражения, в котором они созданы; § 10.4.10.

10.6. Упражнения

1. (*1) Найдите ошибку в ***Date::add_year*** () в § 10.2.2. Найдите еще две ошибки в версии в § 10.2.7.
2. (*2.5) Завершите и протестируйте ***Date***. Реализуйте этот класс в представлении «количество дней после 1/1/1970».
3. (*2) Найдите коммерческий класс ***Date***. Покритикуйте средства, используемые в нем. Если возможно, обсудите его с реальным пользователем.
4. (*1) Как вы обратитесь к ***set_default*** из класса ***Date***, находящимся в пространстве имен ***Chrono*** (§ 10.3.2)? Предложите по крайней мере три различных способа.
5. (*2) Определите класс ***Histogram***, который хранит значения для некоторых интервалов, указанных в качестве аргументов конструктора. Напишите функции вывода гистограммы. Учтите случаи выхода значений за допустимый диапазон.
6. (*2) Определите несколько классов, генерирующих случайные числа, имеющие определенные распределения (например, равномерное и экспоненциальное). Пусть у каждого класса имеется конструктор, задающий тип распределения, и функция ***draw***, возвращающая случайное число.
7. (*2.5) Дополните класс ***Table*** для хранения пар (имя, значение). Затем модифицируйте программу калькулятора из § 6.1 таким образом, чтобы в ней использовался ***Table*** вместо ***map***. Сравните две версии.
8. (*2) Перепишите ***Tnode*** из § 7.10[7] в виде класса с конструкторами, деструкторами и т. д. Определите дерево с узлами ***Tnode*** в виде класса с конструкторами, деструкторами и т. д.
9. (*3) Определите, реализуйте и оттестируйте класс ***Intset*** для множества целых чисел. Реализуйте операции объединения, пересечения и симметричной разности.
10. (*1.5) Сделайте из класса ***Intset*** множество узлов ***Node***, где ***Node*** является определенной вами структурой.
11. (*3) Определите класс для анализа, хранения, вычисления и печати простых арифметических выражений, состоящих из целых констант и операторов +, –, * и /. Открытый интерфейс должен выглядеть следующим образом:

```
class Expr {
        //...
public:
        Expr(char*);
        int eval();
        void print();
};
```

Строковый аргумент конструктора ***Expr::Expr*** () является выражением. Функция ***Expr::eval*** () возвращает значение выражения, и ***Expr::print*** () выводит представление выражения в ***cout***. Программа может выглядеть следующим образом:

```
Expr x("123/4+123*4-3");
cout << "x = " << x.eval() << "\n";
x.print();
```

Определите класс ***Expr*** дважды: один раз с использованием в качестве представления связного списка узлов, и другой — с использованием строк. Поэкспериментируйте с различными способами печати выражения: с расставленными скобками, в постфиксной нотации и т. д.

12. (*2) Определите класс ***Char_queue*** (очередь символов) таким образом, чтобы открытый интерфейс не зависел от представления. Реализуйте ***Char_queue***:
    а) в виде связного списка;
    б) в виде вектора.
    Не думайте о многозадачности.
13. (*3) Разработайте класс таблицы символов и класс элементов таблицы символов для какого-нибудь языка. Посмотрите компилятор для этого языка. Как выглядит в действительности таблица символов?
14. (*2) Измените класс выражений из § 10.6[11] таким образом, чтобы он мог обрабатывать переменные и оператор присваивания =. Воспользуйтесь классом таблицы символов из § 10.6[13].
15. (*1) Имеется программа:

```
#include <iostream>

int main()
{
        std::cout << "Здравствуй, мир!\n";
}
```

Модифицируйте ее таким образом, чтобы она выводила:

```
Инициализация
Здравствуй, мир!
Очистка
```

Не делайте при этом никаких изменений в ***main*** ().

16. (*2) Определите класс ***Calculator***, большая часть которого реализуется при помощи функций из § 6.1. Создайте калькуляторы и вызовите их с аргументом из ***cin***, из командной строки и строки в программе. Реализуйте возможность вывода в различные места назначения, аналогично вводу.
17. (*2) Определите два класса, каждый из которых имеет статический член, таким образом, чтобы конструктор каждого статического члена содержал в себе ссылку на другой. Где в реальном коде можно встретить такие конструкции? Как можно исключить зависимость от порядка в таких конструкторах?
18. (*2.5) Сравните класс ***Date*** (§ 10.3) с вашим решением упражнений § 5.9[13] и § 7.10[19]. Обсудите найденные ошибки и возможные различия при сопровождении каждого из решений.

19. (*3) Напишите функцию, получающую в качестве аргументов ***istream*** и ***vector<string>*** и возвращающую ***map<string, vector<int> >***. Последний ассоциативный массив должен содержать каждую строку и количество раз, которое она встречалась. Запустите программу с текстовым файлом размером не менее 1000 строк, ища при этом не менее 10 слов.
20. (*2) Возьмите класс ***Entry*** из § B.8.2 и модифицируйте его таким образом, чтобы каждый член объединения всегда использовался в соответствии с его типом.

11

Перегрузка операторов

*Когда **я** использую слово,*
*оно означает то, что **я** хочу,*
не больше и не меньше.
— Шалтай-Болтай

Обозначения — операторные функции — бинарные и унарные операторы — предопределенный смысл операторов — смысл операторов, определяемый пользователем — операторы и пространства имен — комплексный тип — операторы-члены и не-члены — смешанная арифметика — инициализация — копирование — преобразования — литералы — функции-помощники — операторы преобразования — разрешение неоднозначности — друзья — члены и друзья — большие объекты — присваивание и инициализация — индексация — вызов функции — разыменование — инкремент и декремент — класс строк — советы — упражнения.

11.1. Введение

В каждой технической области — и в большинстве не технических — имеются свои стандартные обозначения, облегчающие представление и обсуждение часто встречающихся концепций. Например, благодаря постоянному использованию, выражение

```
x+y*z
```

яснее для нас, чем фраза

умножить y на z и прибавить результат к x

Трудно переоценить значение краткой и выразительной формы записи типичных операций.

Как и большинство других языков C++ поддерживает набор операторов для встроенных типов. Однако большинство концепций, для которых обычно используются операторы, не являются встроенными типами C++, поэтому они должны быть представлены в виде типов, определяемых пользователем. Например, если вам требуется комплексная арифметика, матричная алгебра, сигнальная логика или символьные строки в C++, вы пользуетесь механизмом классов для представления этих понятий. Определение операторов для таких классов иногда позволяет программисту реализовать более привычную и удобную форму записи для манипулирования объектами, чем та, которая доступна с использованием только базовой функциональной формы записи. Например,

```
class complex {                    // очень упрощенный класс complex
        double re, im;
public:
        complex (double r, double i) :re (r), in (i) { }
        complex operator+ (complex);
        complex operator* (complex);
};
```

определяет простую реализацию концепции комплексного числа. Класс ***complex*** представлен парой чисел с плавающей точкой двойной точности и двумя операторами + и *. Программист вводит операторы ***complex::operator+*** () и ***complex::operator**** (), которые определяют смысл операций сложения и умножения. Например, если и ***b*** и ***c*** имеют тип ***complex***, ***b+c*** означает ***b.operator+(c)***. Теперь мы можем записать комплексные выражения в форме, близкой к общепринятой:

```
void f()
{
        complex a = complex (1, 3.1);
        complex b = complex (1.2, 2);
        complex c = b;

        a = b+c;
        b = b+c*a;
        c = a*b+complex (1, 2);
```

Выполняются обычные правила приоритета операций, поэтому вторая инструкция означает *b=b+(c*a)*, а не *b=(b+c)*a*.

Наиболее очевидное использование перегруженных операторов — применение их к конкретным типам (§ 10.3). Однако польза от операторов, определяемых пользователем, не ограничивается только конкретными типами. Например, проектирование обобщенных и абстрактных интерфейсов часто приводит к использованию операторов, таких как ->, [] и ().

11.2. Операторные функции

Можно объявить функции, определяющие смысл следующих операторов (§ 6.2):

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	*new*	*new*[]	*delete*	*delete*[]

Следующие операторы не могут быть определены пользователем:

:: (разрешение области видимости; § 4.9.4, § 10.2.4);

. (выбор члена; § 5.7);

.* (выбор члена через указатель на член; § 15.5).

Правым операндом у них является не значение, а имя, и они предоставляют основные средства доступа к членам. Разрешение их перегрузки привело бы к очень тонким ошибкам [Stroustrup, 1994]. Тернарный условный оператор ?: также не может быть перегружен. Аналогично не могут быть прегружены операторы ***sizeof*** (§ 14.6) и ***typeid*** (§ 15.4.4).

Невозможно определить новую лексему оператора, но вы можете воспользоваться функцией, если вас не устраивает существующий набор операторов. Например, вы можете написать ***pow*** () вместо **. Это ограничение может показаться очень строгим, но более гибкие правила легко могут привести к неоднозначности. Например, определение оператора ** для обозначения степени может на первый взгляд показаться очевидной и легкой задачей, но подумайте еще раз. Должен ли ** быть левоассоциативным (как в Fortran) или правоассоциативным (как в Algol)? Выражение ***a**p*** должно интерпретироваться как ***a**** (****p***) или как (***a***)** (***p***)?

Имя операторной функции начинается с ключевого слова ***operator***, за которым следует сам оператор; например ***operator<<***. Операторная функция объявляется и может быть вызвана, как любая другая функция. Использование операторной функции как оператора является просто сокращенной формой ее явного вызова. Например:

```
void f(complex a, complex b)
{
        complex c = a + b,                  // сокращенная форма
        complex d = a operator+ (b);        // явный вызов
}
```

При наличии предыдущего определения ***complex*** обе инструкции являются синонимами.

11.2.1. Бинарные и унарные операторы

Бинарный оператор можно определить либо в виде нестатической функции-члена с одним аргументов, либо в виде функции-не-члена с двумя аргументами. Для любого бинарного оператора @ выражение ***aa@bb*** интерпретируется либо как ***aa.operator@(bb)*** либо ***operator@(aa, bb)***. Если определены обе функции, для выяснения того, какую (возможно никакую) из них использовать, применяется механизм разрешения перегрузки (§ 7.4). Например:

```
class X{
public.
        void operator+ (int);
        X(int),
},

void operator+ (X, X),
void operator+ (X, double);

void f(X a)
{
        a+1;            // a.operator+(1)
        1+a,            // ::operator+(X(1), a)
        a+1.0;          // ::operator+(a, 1.0)
}
```

Унарный оператор, префиксный или постфиксный, можно определить либо в виде нестатической функции-члена без аргументов, либо в виде функции-не-члена, с одним аргументом. Для любого унарного оператора @ выражение ***@aa*** интерпретируется либо как ***aa.operator@*** (), либо как ***operator@*** (***aa***). Если определены обе функции, для определения того, какую (возможно, никакую) из них использовать,

применяется механизм разрешения перегрузки (§ 7.4). Для любого постфиксного оператора @ выражение *aa*@ интерпретируется либо как *aa.operator*@ (*int*) либо как *operator*@ (*aa*, *int*). Более подробно это объясняется в § 11.11. Если определены обе функции, для определения того, какую (возможно, никакую) из них использовать, применяется механизм разрешения перегрузки (§ 7.4). Оператор может быть объявлен только с синтаксисом, существующем для него в грамматике (§ A.5). Например, пользователь не может определить унарный % или + с тремя операндами. Рассмотрим пример:

```
class X {
        // члены (с неявным указателем this)

        X* operator& ();            // префиксный унарный оператор & (чей-то адрес)
        X operator& (X);            // бинарный оператор & (и)
        X operator++ (int),         // постфиксный инкремент (см. § 11.11)
        X operator& (X, X);         // ошибка: три операнда
        X operator/ ();             // ошибка: унарный оператор /
};

//функции-не-члены

X operator- (X),                    // префиксный унарный минус
X operator- (X, X)                  // бинарный минус
X operator-- (X&, int);             // постфиксный декремент
X operator- ()                      // ошибка: отсутствует операнд
X operator- (X, X, X),              // ошибка: три операнда
X operator% (X),                    // ошибка: унарный оператор %
```

Оператор [] описывается в § 11.8, оператор () — в § 11.9, оператор -> — в § 11.10, операторы ++ и -- в § 11.11, а операторы выделения и освобождения памяти в § 6.2.6.2, § 10.4.11 и § 15.6.

11.2.2. Предопределенный смысл операторов

По поводу определяемых пользователем операторов делается всего несколько предположений. В частности, *operator*=, *operator*[] и *operator*-> должны быть нестатическими функциями-членами; это гарантирует, что их первый операнд будет lvalue (§ 4.9.6).

Некоторые встроенные операторы определены таким образом, что они эквивалентны некоторой комбинации других операторов с теми же операндами. Например, если *a* — целое, ++*a* означает *a*+=*1*, что в свою очередь означает *a*=*a*+*1*. Такая связь не сохраняется в операторах, определяемых пользователем, если только пользователь не позаботится об этом сам. Например, компилятор не сгенерирует определение *Z*::*operator*+= () из определения *Z*::*operator*+ () и *Z*::*operator*= ().

Из-за исторической случайности операторы = (присваивание), & (адрес) и , (задание последовательности; § 6.2.2) имеют предопределенный смысл, когда применяются к объектам класса. Этот предопределенный смысл может стать недоступным пользователям, если сделать операторы закрытыми:

```
class X {
private:
```

```
        void operator= (const X&);
        void operator& (),
        void operator, (const X&),
        // ...
};

void f(X a, X b)
{
        a = b;          // ошибка: оператор = является закрытым
        &a;             // ошибка: оператор & является закрытым
        a, b;           // ошибка: оператор , является закрытым
}
```

С другой стороны, операторам можно придать новый смысл, задав соответствующие определения.

11.2.3. Операторы и типы, определяемые пользователем

Операторная функция может быть либо членом, либо иметь по крайней мере один аргумент типа, определяемого пользователем (за исключением функций, замещающих операторы ***new*** и ***delete***). Это правило гарантирует, что пользователь не может изменить смысл выражения в случаях, если только оно не содержит объектов типов, определяемых пользователем. В частности, невозможно определить операторную функцию, которая работала бы исключительно с указателями. Таким образом, C++ расширяем (extensible), но не изменчив (mutable) (исключения составляют операторы =, & и , для объектов класса).

Операторная функция, у которой первый операнд принадлежит к встроенному типу (см. § 4.1.1), не может являться членом. Например, рассмотрим сложение комплексной переменной ***aa*** и целого числа ***2***, ***aa+2***. При наличии соответствующим образом объявленных функций-членов это выражение может быть проинтерпретировано как ***aa.operator+ (2)***, в то время, как ***2+aa*** — не может, поскольку не существует класса ***int***, для которого определен оператор + в смысле ***2.operator+ (aa)***. Даже если бы и был такой класс, операции ***2+aa*** и ***aa+2*** выполняли бы две различные функции. Так как компилятор не понимает смысл определяемого пользователем оператора +, он не может делать предположений о его коммутативности и интерпретировать ***2+aa***, в качестве ***aa+2***. Эту проблему легко решить при помощи функций-не-членов (§ 11.3.2, § 11.5).

Перечисления являются типами, определяемыми пользователем, поэтому мы можем ввести для них операторы. Например:

```
enum Day { sun, mon, tue, wed, thu, fri, sat };

Day& operator++ (Day& d)
{
        return d = (sat==d) ? sun : Day (d+1);
}
```

Каждое выражение проверяется на отсутствие неоднозначности. Всегда, когда определяемый пользователем оператор предоставляет возможность той или иной интерпретации, выражение проверяется в соответствии с правилами из § 7.4.

11.2.4. Операторы в пространствах имен

Оператор является либо членом класса, либо определяется в каком-либо пространстве имен (возможно в глобальном). Рассмотрим упрощенную версию ввода/вывода строк из стандартной библиотеки:

```
namespace std {          // упрощенное пространство std
    class ostream {
        // ...
        ostream& operator<< (const char*);
    }
    extern ostream cout;
    class string {
        // ...
    };
    ostream& operator<< (ostream&, const string&);
}
int main ()
{
    char* p = "Здравствуй";
    std::string s = "мир";
    std::cout << p << ", " << s << "!\n";
}
```

В результате, естественно, будет выведено "***Здравствуй, мир***!". Но почему? Обратите внимание, я не стал делать доступными все средства из *std* путем

```
using namespace std;
```

Вместо этого я воспользовался префиксом *std*:: для *string* и *cout*. Другими словами, я «вел себя хорошо» и не стал засорять глобальное пространство имен или каким-либо другим образом вводить необязательные зависимости.

Оператор вывода С-строк (*char**) является членом *std*::*ostream*, поэтому по определению

```
std::cout << p
```

означает

```
std::cout.operator<< (p)
```

Однако *std*::*ostream* не имеет функции-члена для вывода *std*::*string*, поэтому

```
std::cout << s
```

означает

```
operator<< (std::cout, s)
```

Поиск операторов, определяемых в пространствах имен, также как и поиск функций (§ 8.2.6), осуществляется на основе типов их операндов. В частности, *cout* находится в пространстве имен *std*, поэтому *std* включается в рассмотрение при поиске подходящего определения <<. В результате, компилятор находит и использует:

```
std::operator<< (std::ostream&, const std::string&)
```

Рассмотрим бинарный оператор @. Если *x* имеет тип ***X***, а *y* имеет тип ***Y***, правила разрешения выражения *x@y* применяются следующим образом:

- если ***X*** является классом, выяснить, определяется ли ***operator@*** в качестве члена класса ***X***, либо базового класса ***X***;
- просмотреть объявления ***operator@*** в контексте, окружающем *x@y*;
- если ***X*** определен в пространстве имен ***N***, поискать объявления ***operator@*** в ***N***;
- если ***Y*** определен в пространстве имен ***M***, поискать объявления ***operator@*** в ***M***.

Может быть найдено несколько объявлений ***operator@*** и могут быть применены правила разрешения перегрузки (§ 7.4) для нахождения наилучшего соответствия, если оно существует. Этот механизм поиска начинает работать, только если у оператора имеется, по крайней мере, один операнд типа, определяемого пользователем. Поэтому будут приняты во внимание преобразования, определяемые пользователем (§ 11.3.2, § 11.4). Обратите внимание, что ***typedef*** просто создает синоним имени и не является типом, определяемым пользователем (§ 4.9.7).

Унарные операторы разрешаются аналогично.

Обратите внимание, что при просмотре операторов нет преимуществ для членов перед не членами. Это отличается от просмотра именованных функций (§ 8.2.6). Результатом недостаточной сокрытости операторов является то, что встроенные операторы всегда доступны и что пользователи могут придавать новый смысл операторам без модификации объявления существующего класса. Например, стандартная библиотека потоков ввода/вывода определяет функции-члены << для вывода встроенных членов. Пользователь может определить << для вывода определенных им типов без изменения класса ***ostream*** (§ 21.2.1).

11.3. Тип комплексных чисел

Реализация комплексных чисел, представленная во введении, слишком ограничена, чтобы удовлетворить кого-либо. Например, зная математику, мы ожидаем, что следующее будет работать:

```
void f()
{
        complex a = complex(1, 2);
        complex b = 3;
        complex c = a+2 3;
        complex d = 2+b;
        complex e = -b-c;
        b = c*2*c;
}
```

Кроме того, мы хотели бы иметь и несколько дополнительных операторов, таких как == для сравнения, << для вывода и подходящий набор математических функций типа *sin* () и *sqrt* ().

Класс ***complex*** является конкретным типом, поэтому при его проектировании мы будем следовать указаниям § 10.3. Кроме того, использование комплексной арифметики настолько зависит от операторов, что при определении класса ***complex*** приходится обращаться к большинству основных правил перегрузки операторов.

11.3.1. Операторы-члены и не-члены

Я предпочитаю сводить к минимуму количество функций, непосредственно манипулирующих представлением объекта. Этого можно добиться определением в теле самого класса только тех операторов, которые должны модифицировать значение первого аргумента, таких как +=. Операторы, которые просто выдают новое значение на основе своих аргументов, такие как +, определяются вне класса; они пользуются операторами, имеющими доступ к представлению:

```
class complex {
        double re, im;
public:
        complex& operator+= (complex a); // требует доступа к представлению
        // ...
};

complex operator+ (complex a, complex b)
{
        complex r = a;
        return r += b;                          // доступ к представлению при помощи +=
}
```

При наличии этих объявлений мы можем записать:

```
void f(complex x, complex y, complex z)
{
        complex r1 = x+y+z,             // r1 = operator+(x,operator+(y,z))
        complex r2 = x;                 // r2 = x
        r2 += y,                        // r2.operator+=(y)
        r2 += z;                        // r2.operator+=(z)
}
```

За исключением разницы во времени вычисления ***r1*** и ***r2*** эквивалентны.

Составные операторы присваивания, такие как += и *=, обычно легче определить, чем их «простые» части + и *. Это сначала многих удивляет, но данный вывод следует из того факта, что в операции + участвуют три объекта (два операнда и результат), в то время, как в операции += — только два. В последнем случае (в результате избавления от временных переменных) улучшается производительность во время выполнения. Например, функция

```
inline complex& complex::operator+= (complex a)
{
        re += a.re;
        im += a.im;
        return *this;
}
```

не требует временной переменной для хранения результата сложения и достаточно проста для встраивания.

Хороший оптимизатор сгенерирует код, близкий к оптимальному и для простого оператора +. Однако у нас не всегда есть хороший оптимизатор, и не все типы настолько просты, как ***complex***, поэтому в § 11.5 обсуждаются способы определения операторов с непосредственным доступом к представлению класса.

11.3.2. Смешанная арифметика

Чтобы разобраться с выражением

```
complex d = 2 + b;
```

нам нужно определить оператор +, который бы работал с операндами разного типа. В терминологии Fortran, нам требуется *смешанная арифметика*. Мы можем этого добиться простым добавлением соответствующих версий операторов:

```
class complex {
        double re, im,
public.
        complex& operator+= (complex a) {
                re += a.re,
                im += a.im;
                return *this;
        }
        complex& operator+= (double a) {
                re += a;
                return *this;
        }
        // ...
};
complex operator+ (complex a, complex b)
{
        complex r = a,
        return r+= b;           // вызывает complex::operator+=(complex)
}
complex operator+ (complex a, double b)
{
        complex r = a,
        return r+= b,           // вызывает complex::operator+=(double)
}
complex operator+ (double a, complex b)
{
        complex r = b;
        return r+= a;           // вызывает complex::operator+=(double)
}
```

Операция сложения ***double*** с комплексным числом проще, чем сложение комплексных чисел. Этот факт отражен в приведенных определениях. Операции с операндом ***double*** не затрагивают мнимую часть комплексного числа и поэтому более эффективны.

При наличии этих объявлений, мы можем записать:

```
void f(complex x, complex y)
{
        complex r1 = x+y;       // вызывает operator+(complex, complex)
        complex r2 = x+2;       // вызывает operator+(complex, double)
        complex r3 = 2+x;       // вызывает operator+(double, complex)
}
```

11.3.3. Инициализация

Для инициализации и присваивания комплексным переменным скалярных значений, нам нужно преобразование скалярной величины (целого или числа с плавающей точкой) в ***complex***. Например:

```
complex b = 3;          // должно значить b.re=3, b.im=0
```

Конструктор с одним аргументом задает преобразование типа своего аргумента в тип конструктора. Например:

```
class complex {
    double re, im;
public:
    complex (double r) : re (r), im (0) { }
    // ...
};
```

Этот конструктор реализует традиционное представление вещественной оси в комплексной плоскости.

Конструктор является предписанием для создания значения данного типа. Конструктор используется, когда ожидается значение данного типа, и когда такое значение может быть создано конструктором из значения, заданного как инициализатор, или присваиваемого значения. Поэтому, конструктор, имеющий один аргумент, не нужно вызывать явно. Например,

```
complex b = 3;
```

означает

```
complex b = complex (3);
```

Преобразование, определяемое пользователем, неявно применяется только в том случае, если оно уникально (§ 7.4). См. в § 11.7.1 способ задания конструктора, который можно вызвать только явно.

Естественно, нам по-прежнему нужен конструктор комплексного числа из двух величин ***double*** и конструктор по умолчанию, создающий (***0***, ***0***):

```
class complex {
    double re, in;
public:
    complex () : re (0), im (0) { }
    complex (double r) : re (r), im (0) { }
    complex (double r, double i) : re (r), im (i) { }
    // ...
}
```

Используя аргументы по умолчанию, мы можем сократить это до:

```
class complex {
    double re, in;
public:
    complex (double r=0, double i=0) : re (r), im (i) { }
    // ...
};
```

Когда для типа явно объявлен конструктор, нельзя пользоваться списком инициализации (§ 5.7, § 4.9.5) в качестве инициализатора. Например:

```
complex z1 =3 { 3 };          // ошибка: у complex есть конструктор
complex z2 = { 3, 4 };        // ошибка: у complex есть конструктор
```

11.3.4. Копирование

Кроме явно объявленных конструкторов, ***complex*** по умолчанию имеет копирующий конструктор (§ 10.2.5). Копирующий конструктор по умолчанию просто копирует все члены. Чтобы быть совсем точными, мы могли бы написать:

```
class complex {
        double re, im;
public:
        complex (const complex& c) : re (c.re), im (c.im) { }
        // ...
};
```

Однако в тех случаях, когда копирующий конструктор по умолчанию имеет правильный смысл, я предпочитаю полагаться на это умолчание. Это короче, чем я могу написать, а читающие код должны понимать умолчания. Кроме того, компилятор знает об этом умолчании и о возможностях его оптимизации. Написание почленного копирования классов с большим количеством членов данных вручную — занятие утомительное, и при этом можно наделать массу ошибок (§ 10.4.6.3).

Я использую для копирующего конструктора аргумент, являющийся ссылкой, потому что я просто обязан это сделать. Копирующий конструктор определяет, что имеется в виду под копированием — включая и то, что понимается под копированием аргумента — поэтому запись

```
complex::complex (complex c) : re (c.re), im (c.im) { }    // ошибка
```

является ошибкой, так как любой вызов такого конструктора приведет к бесконечной рекурсии.

Для других функций, имеющих комплексные аргументы, я использую передачу аргументов по значению, а не по ссылке. Здесь разработчик имеет выбор. С точки зрения пользователя разница между функцией с аргументом ***complex*** и функцией с аргументом ***const complex***& невелика. Эта тема далее обсуждается в § 11.6.

В принципе, копирующие конструкторы используются при простых инициализациях типа

```
complex x = 2;                // создает complex(2), затем инициализирует им x
complex y = complex (2, 0);   // создает complex(2, 0), затем инициализирует им y
```

Однако от вызовов копирующих конструкторов можно легко избавиться. Мы могли бы с тем же успехом записать:

```
complex x (2);                // проинициализировать x значением 2
complex y (2, 0);             // проинициализировать y значением (2, 0)
```

Для арифметических типов, таких как ***complex***, я предпочитаю вариант с использованием =. Можно уменьшить набор значений, допустимых при стиле инициализации с = по сравнению со стилем с (), сделав копирующий конструктор закрытым (§ 11.2.2) или объявив конструктор с ключевым словом ***explicit*** (§ 11.7.1).

Аналогично инициализации, присваивание одного объекта класса другому по умолчанию определено как почленное присваивание (§ 10.2.5). Мы могли бы явно заставить ***complex*::*operator*=** осуществлять почленное присваивание. Однако в случае такого простого типа как ***complex***, в этом нет никакой необходимости. Умолчание работает правильно.

Копирующий конструктор — и определяемый пользователем, и генерируемый компилятором — используется не только при инициализации переменных, но также и при передаче аргументов, возврате значения из функции и при обработке исключений (см. § 11.7). Семантика этих операций по определению совпадает с семантикой инициализации (§ 7.1, § 7.3, § 14.2.1).

11.3.5. Конструкторы и преобразования

Мы определили три варианта каждой из четырех стандартных арифметических операций:

```
complex operator+ (complex, complex);
complex operator+ (complex, double);
complex operator+ (double, complex);
// ...
```

Это дублирование может стать довольно утомительным занятием, а то, что утомительно, часто приводит к ошибкам. Что если бы мы имели три альтернативы для типа каждого аргумента каждой функции? Нам бы потребовались три версии каждой функции с одним аргументом, девять версий каждой функции с двумя аргументами, двадцать семь версий каждой функции с тремя аргументами и т. д. Часто эти варианты очень похожи друг на друга. В действительности, почти все варианты включают в себя элементарное преобразование типов аргументов в простые типы и затем выполнение стандартного алгоритма.

Альтернативным способом реализации различных версий функции для каждой комбинации аргументов является использование преобразований типов. Например, наш класс ***complex*** имеет конструктор, который осуществляет преобразование ***double*** в ***complex***. Следовательно, мы могли объявить только одну версию оператора проверки на равенство для ***complex***:

```
bool operator== (complex, complex);

void f(complex x, complex y)
{
    x==y;          // означает operator==(x,y)
    x==3;          // означает operator==(x,complex(3))
    3==y;          // означает operator==(complex(3),y)
}
```

Могут существовать причины, которые вынудят нас отдать предпочтение варианту с несколькими отдельными функциями. Например, в некоторых случаях преобразование может вызвать дополнительные накладные расходы, а в других случаях для конкретного типа аргумента можно реализовать более простой алгоритм. Если подобные соображения не существенны, использование преобразований типов и реализация только наиболее общих вариантов функций — плюс возможно несколько критичных по ресурсам вариантов — предотвращает комбинаторный взрыв вариантов, который часто происходит в смешанной арифметике.

Если существует несколько вариантов функции или оператора, компилятор должен выбрать «правильный» вариант, основываясь на типах аргументов и доступных преобразованиях (стандартных и определяемых пользователем). Если наилучшее соответствие не находится, выражение неоднозначно и является ошибочным (см. § 7.4).

Объект, созданный с явным или неявным использованием конструктора в выражении, является автоматическим и будет уничтожен при первой же возможности (см. § 10.4.10).

К левой части оператора . (или ->) не применяется никаких неявных преобразований типов, определяемых пользователем. Это происходит даже в тех случаях, когда . задается неявно. Например:

```
void g(complex z)
{
        3+z;                    // правильно: complex(3)+z
        3.operator+=(z);        // ошибка: 3 не является объектом класса
        3+=z;                   // ошибка: 3 не является объектом класса
}
```

Следовательно, вы можете отразить тот факт, что оператор требует в качестве левого операнда *lvalue*, сделав оператор членом.

11.3.6. Литералы

Невозможно определить литералы типа, являющегося классом, в том смысле, как *1.2* и *12e3* являются литералами типа *double*. Однако в качестве замены можно пользоваться литералами встроенного типа, используя функции-члены для их интерпретации. Конструктор с единственным аргументом является стандартным механизмом реализации этой идеи. Если конструктор прост и реализован в виде встроенной функции, вполне разумно рассматривать вызов конструктора с литеральным аргументом в качестве литерала. Например, я рассматриваю *complex(3)* как литерал типа *complex*, хотя технически это не так.

11.3.7. Дополнительные функции-члены

Итак, мы реализовали класс *complex*, для которого имеются только конструкторы и арифметические операции. Этого недостаточно для реального использования. Например, нам часто потребуется извлечение вещественной и мнимой частей:

```
class complex {
        double re, im;
public:
        double real() const { return re; }
        double imag() const { return im; }
        //...
};
```

В отличие от других членов *complex* функции-члены *real()* и *imag()* не модифицируют значение комплексного числа, поэтому их можно объявить константными.

Имея *real()* и *imag()*, мы можем определить все необходимые операции, не позволяя им осуществлять непосредственный доступ к представлению класса *complex*. Например:

```
inline bool operator== (complex a, complex b)
{
    return a.real ()==b.real () && a.imag ()==b.imag ();
}
```

Обратите внимание, что нам понадобилось только чтение вещественной и мнимой частей — потребность в их записи возникает гораздо реже. Если требуется «частичная модификация», мы можем сделать это следующим образом:

```
void f (complex& z, double d)
{
    // ...
    z = complex (z.real (), d);                    // присвоить d члену z.im
}
```

Хороший оптимизатор сгенерирует для этой инструкции одну операцию присваивания.

11.3.8. Функции-помощники

Если мы соберем все фрагменты вместе, класс ***complex*** будет выглядеть следующим образом:

```
class complex {
    double re, im;
public:
    complex (double r=0, double i=0) : re (r), im (i) { }

    double real () const { return re; }
    double imag () const { return im; }

    complex& operator+= (complex);
    complex& operator+= (double);
    // -=, *=, /=
};
```

Кроме того, нам нужен набор функций-помощников:

```
complex operator+ (complex, complex);
complex operator+ (complex, double);
complex operator+ (double, complex);
// -, *, /

complex operator- (complex);                       // унарный минус
complex operator+ (complex);                       // унарный плюс

bool operator== (complex, complex);
bool operator!= (complex, complex);

istream& operator>> (istream&, complex&);          // ввод
ostream& operator<< (ostream&, complex);           // вывод
```

Обратите внимание, что функции-члены ***real*** () и ***imag*** () играют значительную роль в определении сравнений. Следующие функции-помощники также используют ***real*** () и ***imag*** ().

Мы могли бы реализовать функции, позволяющие пользователю работать в полярных координатах:

```
complex polar (double rho, double theta),
complex conj (complex),

double abs (complex),
double arg (complex),
double norm (complex),

double real (complex),      // для удобства записи
double imag (complex),      // для удобства записи
```

И, наконец, мы должны реализовать подходящий набор стандартных математических функций:

```
complex acos (complex),
complex asin (complex),
complex atan (complex),
// .
```

С точки зрения пользователя представленный здесь комплексный тип почти идентичен *complex<double>* из *<complex>* стандартной библиотеки (§ 22 5).

11.4. Операторы преобразования

Использование конструктора для преобразования типа удобно, но может иметь неприятные побочные эффекты. Конструктор не может осуществить:

[1] неявное преобразование из типа, определяемого пользователем, во встроенный тип (потому что встроенные типы не являются классами);

[2] преобразование из нового класса в ранее определенный класс (не модифицируя объявление старого класса).

Эти проблемы можно решить путем определения *оператора преобразования* для исходного типа. Функция-член ***X*::*operator T*()**, где ***T*** — имя типа, определяет преобразование ***X*** в ***T***. Например, можно определить 6-битовые неотрицательные целые, ***Tiny***, которые можно свободно использовать вместе с целыми в арифметических операциях.

```
class Tiny {
        char v;
        void assign (int i) { if (i&~077) throw Bad_range (); v=i; }
public
        class Bad_range { };

        Tiny (int i) { assign (i); }
        Tiny& operator= (int i) { assign (i); return *this; }

        operator int () const { return v; }                    // функция преобразования в int
};
```

Диапазон проверяется каждый раз при инициализации ***Tiny*** целым, и когда ***Tiny*** присваивается целое. Проверка диапазона не требуется при копировании ***Tiny***, поэтому копирующий конструктор и присваивание по умолчанию работают правильно.

Для реализации обычных целочисленных операций с переменными типа ***Tiny*** мы определили неявное преобразование из ***Tiny*** в ***int***: ***Tiny*::*operator int*()**. Обратите внимание, что преобразуемый тип является частью имени оператора, и его нельзя повторить в качестве типа возвращаемого значения функции преобразования:

```
Tiny::operator int() const { return v; }        // правильно
int Tiny::operator int() const { return v; }    // ошибка
```

В этом отношении оператор преобразования имеет сходство с конструктором.

При каждом появлении ***Tiny*** в тех местах, где требуется ***int***, используется соответствующий ***int***. Например:

```
int main()
{
    Tiny c1 = 2;
    Tiny c2 = 62;
    Tiny c3 = c2-c1;        // c3=60
    Tiny c4 = c3;           // не осуществляется проверка диапазона (не требуется)
    int i = c1+c2;          // i=64

    c1 = c1+c2;             // выход за пределы диапазона, c1 не может равняться 64
    i = c3-64;              // i=-4
    c2 = c3-64;             // выход за пределы диапазона, c2 не может равняться -4
    c3 = c4;                // не осуществляется проверка диапазона (не требуется)
}
```

Функции преобразования особенно полезны для работы со структурами данных, когда чтение (реализованное оператором преобразования) тривиально, в то время как присваивание и инициализация заметно менее тривиальны.

Типы ***istream*** и ***ostream*** используют функции преобразования для того, чтобы сделать осмысленными выражения типа

```
while (cin>>x) cout<<x;
```

Операция ввода ***cin>>x*** возвращает ***istream&***. Это значение неявно преобразуется в значение, отражающее состояние ***cin***. Полученное значение теперь может проверяться в ***while*** (см. § 21.3.3). Однако идея определения неявного преобразования одного типа в другой таким образом, что при преобразовании может произойти потеря информации, *не* слишком хороша.

Как правило, лучше не переусердствовать при создании операторов преобразования. При чрезмерном использовании они приводят к неоднозначности. Подобная неоднозначность обнаруживается компилятором, но, тем не менее, от нее бывает трудно избавиться. Вероятно, лучше сначала осуществлять преобразование при помощи именованных функций типа ***X::make_int()***. Если подобная функция становится слишком «популярной», так что ее использование становится неэлегантным, ее можно заменить оператором преобразования ***X::operator int()***, чтобы воспользоваться более красивым неявным преобразованием.

Если существуют и преобразование, определяемое пользователем, и операторы, определяемые пользователем, может возникнуть конфликт между определяемыми пользователем и встроенными операторами. Например:

```
int operator+(Tiny, Tiny);

void f(Tiny t, int i)
{
    t+i;        // неоднозначность: operator+(t,Tiny(i)) или int(t)+i?
}
```

Поэтому чаще имеет смысл пользоваться преобразованиями, определяемыми пользователем, или операторами, определяемыми пользователем, но не обоими сразу.

11.4.1. Разрешение неоднозначности

Присваивание значения типа ***V*** объекту класса ***X*** допустимо в том случае, если имеется оператор присваивания ***X*::*operator*=(*Z*)** такой, что ***V*** является ***Z*** или существует единственное преобразование ***V*** в ***Z***. Инициализация рассматривается аналогично.

В некоторых случаях значение требуемого типа может быть создано при помощи повторного использования конструкторов или операторов преобразования. Это должно осуществляться при помощи явных преобразований — допустим только один уровень неявных преобразований, определяемых пользователем. В некоторых случаях значение требуемого типа может быть создано более, чем одним способом — такое недопустимо. Например:

```
Class X { /* ... */ X(int), X(char*); };
Class Y { /* ... */ Y(int), };
Class Z { /* ... */ Z(X), };

Xf(X);
Yf(Y),

Zg(Z);

void k1()
{
        f(1);           // неоднозначность: f(X(1)) или f(Y(1))?
        f(X(1));        // правильно
        f(Y(1));        // правильно

        g("Mack");      // ошибка: требуются применение двух преобразований,
                        // определенных пользователем; g(Z(X("Mack"))) не пробуется
        g(X("Doc"));    // правильно: g(Z(X("Doc")))
        g(Z("Suzy"));   // правильно: g(Z(X("Suzy")))
}
```

Преобразования, определяемые пользователем, рассматриваются только если они необходимы для разрешения вызова. Например:

```
class XX { /* ... */ XX(int); };

void h(double);
void h(XX),

void k2()
{
        h(1);           // h(double(1)) или h(XX(1))? h(double(1))!
}
```

Вызов ***h*(*1*)** означает ***h*(*double*(*1*))**, потому что в этом варианте используются только стандартные преобразования (§ 7.4).

Правила преобразования не являются ни самыми простыми для реализации из всех возможных, ни самыми легкими для документирования, ни настолько общими, как можно себе представить. Однако они довольно безопасны, и их применение не приводит к неприятным сюрпризам. Гораздо легче вручную разрешить неоднозначности, чем найти ошибку, вызванную преобразованием, о котором и не подозревали.

Требование строгого анализа снизу-вверх подразумевает, что возвращаемый тип не используется при разрешении перегрузки. Например:

```
class Quad {
public:
        Quad (double);
        // ...
};

Quad operator+ (Quad, Quad);

void f (double a1, double a2)
{
        Quad r1 = a1 + a2;          // сложение с двойной точностью
        Quad r2 = Quad (a1)+a2;     // явное указание воспользоваться арифметикой
                                    // с «квадратичной» точностью
}
```

Причина такого подхода при проектировании состояла отчасти в том, что анализ снизу-вверх более понятен и частично в том, что не дело компилятора решать, с какой точностью программист хотел выполнить сложение.

После того, как при инициализации или присваивании типы обеих сторон выражения определены, оба типа используются для разрешения. Например:

```
class Real {
public:
        operator double (),
        operator int (),
        // ...
};

void g (Real a)
{
        double d = a;           // d = a.double();
        int i = a,              // i = a.int();

        d = a;                  // d = a.double();
        i = a,                  // i = a.int();
}
```

В этих случаях анализ типа по-прежнему производится снизу-вверх: учитывается только вид оператора и типы его аргументов.

11.5. Друзья класса

Обычное объявление функции-члена гарантирует три логически разные вещи:
[1] функция имеет право доступа к закрытой части объявления класса;
[2] функция находится в области видимости класса;
[3] функцию должна вызываться для объекта класса (имеется указатель ***this***).
Объявив функцию-член как ***static*** (§ 10.2.4), мы придаем ей только первые два свойства. Объявив функцию как ***friend***, мы наделяем ее только первым свойством.

Например, мы могли бы определить оператор, который умножает ***Matrix*** (матрицу) на ***Vector*** (вектор). Естественно, и ***Matrix*** и ***Vector*** скрывают свое представление и обеспечивают полный набор операций для манипулирования объектами их типов. Однако наша процедура умножения не может быть членом обоих классов. С другой стороны, мы не хотим предоставить функции низкоуровневого доступа, что позво-

лило бы пользователю и читать и записывать в полное представление ***Matrix*** и ***Vector***. Во избежание этого мы объявим ***operator**** *другом* (friend) обоих классов:

```
class Matrix;

class Vector {
        float v[4];
        // ...
        friend Vector operator* (const Matrix&, const Vector&);
};

class Matrix {
        Vector v[4];
        // ...
        friend Vector operator* (const Matrix&, const Vector&);
};

Vector operator* (const Matrix& m, const Vector& v)
{
        Vector r;
        for (int i=0; i<4; i++) {          // r[i] = m[i] * v
                r.v[i] = 0;
                for (int j=0; j<4; j++) r.v[i] += m.v[i].v[j] * v.v[j];
        }
        return r;
}
```

Объявление функций-друзей ***friend*** можно поместить и в закрытой и в открытой частях объявления класса — не имеет значения, где именно. Также как и функции-члены, функции-друзья явно указываются в объявлении класса, друзьями которого они являются. Поэтому они в той же мере являются частью интерфейса класса, в какой ею являются функции-члены.

Функция-член одного класса может быть другом иного класса. Например:

```
class List_iterator {
        // ...
        int* next ();
};

class List {
        friend int* List_iterator::next ();
        // ...
};
```

Нередко встречаются случаи, когда все функции одного класса являются друзьями другого. Для таких случаев существует более короткая форма записи:

```
class List {
        friend class List_iterator;
        // ...
},
```

Это объявление ***friend*** делает все функции-члены класса ***List_iterator*** функциями-друзьями класса ***List***.

Ясно, что классы-друзья должны использоваться только для отражения тесно связанных концепций. Часто существует выбор между реализацией класса в качестве члена (вложенного класса) или в качестве друга.

11.5.1. Поиск друзей

Также как и объявление члена, объявление ***friend*** не добавляет новое имя в охватывающую область видимости. Например:

```
class Matrix {
        friend class Xform;
        friend Matrix invert (const Matrix&);
        // ...
};
// ошибка: в текущей области видимости нет имени Xform
Xform x;
// ошибка: в текущей области видимости нет имени invert()
Matrix (*p) (const Matrix&) = &invert;
```

Для больших программ и классов просто замечательно, что классы не добавляют «потихоньку» имена в охватывающую область видимости. Это особенно важно в случае классов-шаблонов, которые могут инстанцироваться во многих различных контекстах (глава 13).

Класс-друг должен быть предварительно объявлен в охватывающей области видимости или определен в области видимости, непосредственно охватывающей класс, объявивший его другом. При этом не принимаются во внимание области видимости вне области видимости самого внутреннего охватывающего пространства имен. Например:

```
class AE { /* ... */ };                    // не друг класса Y

namespace N {
        class X { /* ... */ };             // друг класса Y
        class Y {
                friend class X;
                friend class Z;
                friend class AE;
        };
        class Z { /* ... */ };             // друг класса Y
}
```

Аналогично функцию-друга необходимо явно объявить в охватывающей области видимости, либо она должна иметь аргументы этого класса или класса, наследованного от него. Например:

```
void f(Matrix& m)
{
        invert (m);                        // функция invert() — друг класса Matrix
}
```

Из этого следует, что функция-друг класса должна быть либо явно объявлена в охватывающей области видимости, либо иметь аргументы этого класса. В противном случае функцию-друга вызвать нельзя. Например:

```
//имени f() в этой области видимости нет

class X {
        friend void f();                   // бесполезно
        friend void h (const X&);          // можно найти по типу аргумента
};
```

```
void g (const X&x);
{
    f (),                    // нет имени f() в области видимости
    h (x);                   // функция h(x) — друг класса X
}
```

11.5.2. Друзья и члены

Когда мы должны использовать функции-друзья, а когда функции-члены являются лучшим способом введения операции? Во-первых, мы должны свести к минимуму количество функций, имеющих доступ к представлению класса, чтобы оставался только самый необходимый набор. Поэтому, первый вопрос состоит не в том «Должна ли эта функция быть членом, статическим членом или другом?», а скорее «Действительно ли этой функции нужен доступ к представлению?». Как правило, набор функций, которым требуется доступ, меньше, чем нам кажется сначала.

Некоторые операции должны быть членами — например, конструкторы, деструкторы и виртуальные функции (§ 12.2.6), но, как правило, выбор существует. Так как имена членов являются локальными в классе, функция должна быть членом, если нет специфических причин для того, чтобы она им не была.

Рассмотрим класс **X**, реализующий альтернативные способы представления операций:

```
class X {
    // ...
    X (int);

    int m1 (),
    int m2 () const;

    friend int f1 (X&);
    friend int f2 (const X&);
    friend int f3 (X);
},
```

Функции-члены можно вызывать только с объектами их класса; не применяются никакие преобразования, определяемые пользователем. Например:

```
void g ()
{
    99 m1 ()          // ошибка: X(99).m1() не применяется
    99 m2 ();         // ошибка: X(99).m2() не применяется
}
```

Здесь преобразование **X (int)** не применяется для приведения **99** к типу **X**.

Глобальная функция **f1 ()** имеет сходное свойство, потому что неявные преобразования не применяются к неконстантным аргументам, которые являются ссылками (§ 5.5, § 11.3.5). Однако преобразования применяются к аргументам функций **f2 ()** и **f3 ()**:

```
void h ()
{
    f1 (99);          // ошибка: f1 (X(99)) не применяется
    f2 (99);          // правильно: f2(X(99))
    f3 (99);          // правильно: f3(X(99))
}
```

Поэтому оператор, модифицирующий состояние объекта класса, должен быть членом или глобальной функцией, имеющей в качестве аргумента неконстантную ссылку (или неконстантный указатель). Операторы, которые требуют наличия операндов lvalue для фундаментальных типов, (=, *=, ++ и т. д.) естественней всего определить в качестве членов типов, определяемых пользователем.

С другой стороны, если желательно неявное преобразование типов всех операндов в операции, функция должна быть не-членом и иметь аргументом константную ссылку или не ссылку. Это часто происходит в случаях с функциями, реализующими операторы, которые не требуют наличия операндов lvalue при применении к фундаментальным типам (+, −, || и т. д.). Такие операторы, как правило, нуждаются в доступе к представлению класса операнда. Следовательно, бинарные операторы наиболее часто являются функциями-друзьями.

Если не определены никакие преобразования типов, то нет никаких существенных причин для отдания предпочтения членам по отношению к функциям-друзьям, имеющим в качестве аргумента ссылку, или наоборот. В некоторых случаях, программист может отдавать предпочтение одному синтаксису вызова по отношению к другому. В частности, возникает впечатление, что большинство людей предпочитают форму записи *inv* (*m*) для транспонирования матрицы по сравнению с *m.inv* (). На самом деле, если функция *inv* () в действительности транспонирует саму *m*, а не возвращает новую матрицу, являющуюся транспонированной к *m*, то она должна быть членом.

При прочих равных условиях отдавайте предпочтение реализации в виде члена. Невозможно знать о том, определит ли кто-то где-то оператор преобразования. Также невозможно предвидеть, что последующие изменения потребуют изменения состояния используемых объектов. Синтаксис вызова функции-члена делает очевидным тот факт, что объект может быть изменен; аргумент, являющийся ссылкой, значительно менее очевиден. Более того, выражения в теле члена могут быть заметно короче, чем эквивалентные выражения в глобальной функции; функция, не являющаяся членом, должна использовать явный аргумент, в то время как функция-член может неявно использовать *this*. Кроме того, так как имена членов локальны в пределах класса, они могут быть короче имен функций-не-членов.

11.6. Большие объекты

Мы определили операторы *complex* таким образом, что их операнды имеют тип *complex*. Из этого следует, что при каждом использовании оператора класса *complex*, каждый операнд копируется. Дополнительные затраты на копирование двух *double* могут оказаться заметными, но часто меньшими, чем для пары указателей (доступ через указатель может оказаться излишне накла́дным). К сожалению, не все классы имеют удобное и компактное представление. Чтобы избавиться от интенсивного копирования, можно объявить функции, имеющие в качестве аргументов ссылки. Например:

```
class Matrix {
        double m[4][4];
public:
        Matrix ();
        friend Matrix operator+ (const Matrix&, const Matrix&);
        friend Matrix operator* (const Matrix&, const Matrix&);
};
```

Ссылки позволяют в случае больших объектов пользоваться выражениями, включающими в себя обычные арифметические операторы, без интенсивного копирования. Указателями нельзя воспользоваться потому, что невозможно заместить смысл оператора, применяемого к указателю. Сложение можно определить следующим образом:

```
Matrix operator+ (const Matrix& arg1, const Matrix& arg2)
{
        Matrix sum;
        for (int i=0; i<4; i++)
                for (int j=0; j<4; j++)
                        sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
        return sum;
}
```

Этот ***operator+*** () осуществляет доступ к операндам по ссылке, но возвращает значение объекта. Возвращение ссылки может показаться более эффективным решением:

```
class Matrix {
        // ...
        friend Matrix& operator+ (const Matrix&, const Matrix&);
        friend Matrix& operator* (const Matrix&, const Matrix&);
};
```

Такое допустимо, но вызывает проблемы при выделении памяти. Так как ссылка на результат будет выдана вызываемой функцией как ссылка на возвращаемое значение, возвращаемое из функции значение не может быть автоматической переменной (§ 7.3). Так как оператор может использоваться более одного раза в выражении, результат не может быть статической локальной переменной. Результат, как правило, будет размещаться в свободной памяти. Копирование возвращаемого значения часто обходится дешевле (в смысле времени выполнения, размера кода и данных), чем выделение и (последующее) освобождение памяти под объект. Кроме того, это легче запрограммировать.

Существуют методы, позволяющие избежать копирования результата. Простейшим из них является использование буфера статических объектов. Например:

```
const int max_matrix_temp = 7;

Matrix& get_matrix_temp ()
{
        static int nbuf = 0;
        static Matrix buf[max_matrix_temp];

        if (nbuf== max_matrix_temp) nbuf = 0;
        return buf[nbuf++];
}

Matrix& operator+ (const Matrix& arg1, const Matrix& arg2)
{
        Matrix& res = get_matrix_temp ();
        // ...
        return res;
}
```

Теперь ***Matrix*** копируется только при присваивании результата выражения. Да помогут вам небеса, если вы напишете выражение, в котором потребуется более, чем ***max_matrix_temp*** временных переменных!

Менее подверженная ошибкам техника подразумевает определение типа матрицы в качестве дескриптора (§ 25.7) для класса-представления, который в действительности хранит данные. В этом случае матричные дескрипторы могут управлять объектами представления таким образом, чтобы свести к минимуму выделение памяти и копирование (см. § 11.12 и § 11.14[18]). Однако при таком способе операторы возвращают объекты, а не ссылки или указатели. Другая техника состоит в определении тернарных операций (операций с тремя аргументами) и автоматическом их вызове для выражений типа *a*=*b*+*c* и *a*+*b***i* (§ 21.4.6.3, § 22.4.7).

11.7. Важные операторы

Как правило, для данного типа ***X*** копирующий конструктор ***X*** (***const X***&) отвечает за инициализацию объекта того же типа ***X***. Не лишним будет повторить, что *присваивание и инициализация являются различными операциями* (§ 10.4.4.1). Это имеет особое значение, когда объявлен деструктор. Если класс ***X*** имеет деструктор, который выполняет нетривиальную работу, такую как освобождение памяти, скорее всего классу потребуется полный набор функций, осуществляющих управление процессами конструирования, уничтожения и копирования:

```
class X {
        // . .
        X (Sometype),                   // конструктор: создает объекты
        X (const X&),                   // копирующий конструктор
        X& operator= (const X&),        // копирующее присваивание, очистка и копирование
        ~X (),                          // деструктор: очистка
},
```

Существует еще три случая, когда объект копируется: аргумент функции, возвращаемое значение функции и исключение. При передаче аргумента инициализируется формальный параметр. Соответствующая семантика идентична любой другой инициализации. То же самое происходит для возвращаемых из функции значений и исключений, хотя и в менее очевидной форме. В этих случаях применяется копирующий конструктор. Например:

```
string g (string arg)           // строка передается по значению
{                               // (используя копирующий конструктор)
        return arg,             // строка возвращается
}                               // (используя копирующий конструктор)

int main ()
{
        string s = "Ньютон",    // строка инициализируется
        s = g (s),              // (используя конструктор)
}
```

Совершенно ясно, что значение *s* после вызова ***g*** () должно быть ***"Ньютон"***. Несложно получить копию значения *s* для аргумента ***arg***; это осуществляется вызовом копирующего конструктора для ***string***. Получение копии этого значения из ***g*** () требует еще одного вызова ***string*** (***const string***&); в этот момент инициализируется временная переменная (§ 10.4.10), которая затем присваивается переменной *s*. Довольно часто одна (но не обе) из этих операций может быть устранена при оптимизации.

Для класса, в котором копирующий оператор присваивания и копирующий конструктор явно не объявлены программистом, отсутствующая операция или операции будут сгенерированы компилятором (§ 10.2.5). Это означает, что копирующие операции не наследуются (§ 12.2.3).

11.7.1. Явные конструкторы

По умолчанию конструктор с единственным аргументом также определяет неявное преобразование. Для некоторых типов это является идеальным решением. Например, ***complex*** может быть инициализирована с помощью ***int***:

```
complex z = 2;                  // проинициализировать z значением complex(2)
```

В других случаях неявное преобразование нежелательно и может привести к ошибкам. Например, если бы мы хотели инициализировать ***string*** размера ***int***, то могли бы написать:

```
string s = 'a';                 // создать строку s с int('a') элементами
```

Вряд ли это то, чего ожидал автор определения *s*.

Неявное преобразование можно подавить, объявив конструктор с модификатором ***explicit*** (явный). ***explicit***-конструктор будет вызываться только явно. В частности, там где копирующий конструктор в принципе необходим (§ 11.3.4), ***explicit***-конструктор не будет вызываться неявно. Например:

```
class String {
        // ...
        explicit String(int n);         // выделить n байт
        String(const char* p);          // инициализирующим значением является
                                        // С-строка p
};
String s1 = 'a';                        // ошибка: нет явного преобразования char в String
String s2(10);                          // правильно: строка для хранения 10 символов
String s3 = String(10);                 // правильно: строка для хранения 10 символов
String s4 = "Brian";                    // правильно: s4 = String("Brian")
String s5("Faulty");

void f(String);

String g()
{
        f(10);                          // ошибка: нет явного преобразования int в String
        f(String(10));
        f("Arthur");                    // правильно: String("Arthur")
        f(s1);

        String* p1 = new String("Eric");
        String* p2 = new String(10);

        return 10;                      // ошибка: нет явного преобразования int в String
}
```

Разница между

```
String s1 = 'a';                        // ошибка: нет явного преобразования char в String
String s2(10);                          // правильно: строка для хранения 10 символов
```

может показаться очень тонкой, но она более заметна в реальном коде, чем в надуманных примерах.

В **Date** мы пользовались простым **int** для представления года (§ 10.3). Если бы в нашей разработке абстракция **Date** играла большую роль, мы могли бы создать тип **Year** для реализации более серьезной проверки во время компиляции. Например:

```
class Year {
        int y;
public:
        explicit Year (int i) : y (i) { }          // создание Year из int
        operator int () const { return y; }        // преобразование Year в int
};

class Date {
public:
        Date (int d, Month m, Year y);
        // ...
};

Date d3 (1978, feb, 21);                           // ошибка: 21 это не Year
Date d4 (21, feb, Year (1978));                    // правильно
```

Класс **Year** является простой «оболочкой» для типа **int**. Благодаря оператору **int () Year** неявно преобразуется в **int** каждый раз, когда это необходимо. Объявив конструктор **explicit**, мы гарантируем, что преобразование **int** в **Year** произойдет только тогда, когда мы об этом попросим, и что «случайные» присваивания обнаружатся во время компиляции. Так как функции-члены **Year** легко реализуются в виде встроенных функций, это не приводит к дополнительным затратам памяти и увеличению времени выполнения.

Подобную технику можно использовать для типов диапазонов (§ 25.6.1).

11.8. Индексация

Чтобы придать смысл индексам для объектов класса можно воспользоваться функцией **operator**[]. Второй аргумент (индекс) функции **operator**[] может быть любого типа. Это делает возможным определение векторов, ассоциативных массивов и т. д.

В качестве иллюстрации давайте перепишем пример из § 5.5, в котором в небольшой программе, подсчитывающей количество вхождений слов в файл, использовался ассоциативный массив. Тогда мы применили функцию. Здесь мы определим тип ассоциативного массива:

```
class Assoc {
        struct Pair {
                string name;
                double val,
                Pair (string n="", double v=0) : name (n), val (v) { }
        };
        vector<Pair> vec;

        Assoc (const Assoc&);                      // закрытый, чтобы
                                                   // предотвратить копирование
        Assoc& operator= (const Assoc&);           // закрытый, чтобы
                                                   // предотвратить копирование
public:
```

```
        Assoc () { }
        const double& operator[] (const string&);
        double& operator[] (string&);
        void print_all () const;
};
```

Assoc (ассоциативный массив) хранит вектор **Pairs** (пар). Реализация использует тот же самый тривиальный и неэффективный метод поиска, что и в § 5.5:

```
// поиск s, возврат ее значения, если найдена; в противном случае создание
// нового объекта Pair и возврат значения по умолчанию (нулевого)
double& Assoc::operator[] (string& s)
{
        for (vector<Pair>::iterator p = vec.begin (); p != vec.end (); ++p)
                if (s==p->name) return p->val;

        vec.push_back (Pair (s, 0));    // начальное значение: 0

        return vec.back ().val;         // возврат последнего элемента (§ 16.3.3)
}
```

Так как представление **Assoc** скрыто, требуется некоторый способ его вывода на печать:

```
void Assoc::print_all () const
{
        for (vector<Pair>::const_iterator p = vec.begin (); p! = vec.end (); ++p)
                cout << p->name << ": " << p->val << '\n';
}
```

И, наконец, мы можем написать тривиальную главную программу:

```
// подсчет количества вхождений каждого слова во входном потоке
int main ()
{
        string buf;
        Assoc vec;
        while (cin >> buf) vec[buf]++;
        vec.print_all ();
}
```

Дальнейшее развитие идеи ассоциативного массива описывается в § 17.4.1.

Функция **operator**[] () должна быть членом.

11.9. Вызов функции

Вызов функции, то есть обозначение *выражение(список выражений)*, можно интерпретировать как бинарную операцию, где *выражение* — левый операнд, а *список выражений* — правый. Оператор вызова () может быть перегружен точно так же, как и другие операторы. Аргументы из списка оператора **operator** () () вычисляются и проверяются в соответствии с обычными правилами передачи аргументов. Перегрузка вызова функции скорее всего полезна в основном для определения типов с одной единственной операцией или типов с одной главной операцией. Оператор вызова также известен как прикладной оператор (application operator).

Наиболее очевидным и, вероятно, наиболее важным использованием оператора () является предоставление синтаксиса стандартного вызова функций для объектов, ко-

торые в некотором смысле ведут себя как функции. Объект, который ведет себя как функция, часто называют *функции-подобным объектом* или просто *объектом-функцией* (§ 18.4). Объекты-функции играют важную роль, поскольку они позволяют нам писать код с использованием нетривиальных операций в качестве параметров. Например, в стандартной библиотеке имеется множество алгоритмов, которые вызывают функцию для каждого элемента контейнера. Рассмотрим пример:

```
void negate (complex& c) { c = −c; }

void f (vector<complex>& aa, list<complex>& ll)
{
        for_each (aa.begin (), aa.end (), negate);   // поменять знак у всех элементов вектора
        for_each (ll.begin (), ll.end (), negate);   // поменять знак у всех элементов списка
}
```

Эта функция меняет знак у всех элементов вектора и списка.

Что, если бы мы захотели прибавить **complex** (2, 3) ко всем элементам? Это можно легко сделать следующим образом:

```
void add23 (complex& c)
{
        c += complex (2, 3);
}

void g (vector<complex>& aa, list<complex>& ll)
{
        for_each (aa.begin (), aa.end (), add23);
        for_each (ll.begin (), ll.end (), add23);
}
```

Каким образом мы написали бы функцию для регулярного добавления произвольного комплексного числа? Нам требуется нечто, чему мы можем передать произвольное значение и что затем может использовать это значение каждый раз, когда вызывается. Этого не происходит естественным образом при использовании функций. Обычно «передача» произвольного значения заканчивается оставлением его в окружающем функцию контексте. Это приводит к излишней путанице. Однако мы можем написать класс, который ведет себя желаемым образом:

```
class Add {
        complex val;
public:
        Add (complex c) { val = c; }                          // сохранить значение
        Add (double r, double i) { val = complex (r, i); }

        void operator () (complex& c) const { c += val; }     // прибавить значение
                                                              // к аргументу
};
```

Объект класса **Add** инициализируется комплексным числом, и при вызове с использованием () прибавляет это число к аргументу. Например:

```
void h (vector<complex>& aa, list<complex>& ll, complex z)
{
        for_each (aa.begin (), aa.end (), Add (2, 3));
        for_each (ll.begin (), ll.end (), Add (z));
}
```

В примере ***complex*** (*2, 3*) будет прибавлено к каждому элементу вектора и *z* — к каждому элементу списка. Обратите внимание, что ***Add*** (*z*) создает объект, который затем используется повторно функцией ***for_each*** (). Этот объект — не просто функция, которая вызвана один раз или даже повторно. Функция, которая вызывается повторно, — это ***operator*** () () объекта ***Add*** (*z*).

Изложенное выше работает, потому что ***for_each*** является шаблоном, который применяет () к третьему аргументу, не заботясь о том, чем он в действительности является:

```
template<class Iter, class Fct> Fct for_each (Iter b, Iter e, Fct f)
{
        while (b != e) f(*b++);
        return f;
}
```

На первый взгляд может показаться, что такая техника рассчитана на посвященных, но на самом деле она проста, эффективна и исключительно полезна (см. § 3.8.5, § 18.4).

Другим популярным способом использования ***operator*** () () является применение его в качестве оператора подстроки и оператора индексации в многомерных массивах (§ 22.4.5).

Функция ***operator*** () () должна быть членом.

11.10. Разыменование

Оператор разыменования -> можно определить в виде унарного постфиксного оператора. Например:

```
class Ptr {
        //...
        X* operator->();
};
```

Теперь можно пользоваться объектами класса ***Ptr*** для доступа к членам класса ***X*** совершенно аналогично указателям. Например:

```
void f(Ptr p)
{
        p->m = 7;                    // (p.operator->())->m = 7
}
```

Преобразование объекта *p* в указатель *p.operator->* () не зависит от члена *m*, на который он указывает. Как раз в этом смысле, ***operator->*** () является унарным постфиксным оператором. Однако никакой новый синтаксис не вводился, поэтому все равно требуется имя члена после ->. Например:

```
void g(Ptr p)
{
        X* q1 = p->;                 // синтаксическая ошибка
        X* q2 = p operator->();      // правильно
}
```

Перегрузка -> в основном полезна для создания «умных указателей», то есть объектов, которые ведут себя как указатели и, кроме того, выполняют некоторые действия, когда через них осуществляется доступ к объекту. Например, можно определить

класс ***Rec_ptr*** для доступа к объектам класса ***Rec***, хранящимся на диске. Конструктор ***Rec_ptr*** получает в качестве аргумента имя, которое используется для нахождения объекта на диске, ***Rec_ptr::operator–>*** () извлекает объект в память при доступе через его ***Rec_ptr***, а деструктор ***Rec_ptr*** записывает изменения объекта на диск:

```
class Rec_ptr {
        const char* identifier;
        Rec* in_core_address,
        // ...
public
        Rec_ptr (const char* p) . in_core_address (0) , identifier (p) { }
        ~Rec_ptr () { write_to_disk (in_core_address, identifier); }
        Rec* operator–> ();
};
Rec* Rec_ptr  operator–> ()
{
        if (in_core_address == 0) in_core_address = read_from_disk (identifier);
        return in_core_address;
}
```

Rec_ptr можно использовать следующим образом:

```
struct Rec {                        // Rec, на который указывает Rec_ptr
        string name;
        // ...
};
void update (const char* s)
{
        Rec_ptr p (s),              // получить Rec_ptr для s
        p–>name = "Roscoe";         // изменить s (при необходимости
                                    // считав его сначала с диска)
        // ...
}
```

Естественно, реальный ***Rec_ptr*** был бы сделан шаблоном, чтобы тип ***Rec*** был параметром. Кроме того, реальная программа содержала бы в себе код обработки ошибок и использовала менее наивный способ взаимодействия с диском.

Для обыкновенных указателей, использование оператора –> является синонимом некоторой комбинации унарных операторов * и []. Если имеется ***Y***, для которого –>, * и [] имеют значение по умолчанию, и ***Y**** вызывает ***p***, то

```
p–>m == (*p).m          // истина
(*p) m == p[0] m        // истина
p–>m == p[0] m          // истина
```

Как обычно, никаких гарантий не существует для операторов, определяемых пользователем. Если требуется подобная эквивалентность, ее можно добиться:

```
class Ptr_to_Y {
        Y* p,
public
        Y* operator–> () { return p, }
```

```
        Y& operator* () { return *p; }
        Y& operator[] (int i) { return p[i]; }
};
```

Если вы пишете несколько таких операторов, разумно обеспечить эквивалентность, точно также, как желательно убедиться, что ++*x* и *x*+=*1* имеют тот же эффект, что и *x*=*x*+*1* для простой переменной *x* некоторого класса, для которого реализованы операторы ++, +=, = и +.

Перегрузка -> имеет значение для целого круга интересных задач, так что это не просто занятная безделушка. Причина здесь в том, что *косвенный доступ* является ключевой концепцией, а -> предоставляет ясный, прямой и эффективный способ его реализации в программе. Важными примерами являются итераторы (глава 19). С другой стороны, можно рассматривать оператор -> в качестве способа реализации в C++ ограниченной, но полезной формы *делегирования* (§ 24.3.6).

Оператор -> должен быть функцией-членом. Типом возвращаемого результата должен быть либо указатель, либо объект класса, к которому вы можете применять ->. При объявлении для класса-шаблона ***operator->*** () часто не используется, поэтому имеет смысл отложить проверку ограничений на возвращаемый тип до реального использования.

11.11. Инкремент и декремент

Раз уж изобретены «умные указатели», часто возникает желание реализовать и операторы инкремента ++ и декремента --, аналогичные по смыслу тем, что используются со встроенными типами. Это особенно очевидно и необходимо там, где целью является замещение обыкновенного типа указателя типом «умного указателя» с той же семантикой, но с добавлением проверок во время выполнения. Например, рассмотрим традиционную программу с возможными проблемами:

```
void f1 (T a)           // традиционное использование
{
        T v[200];
        T* p = &v[0];
        p--;
        *p = a;         // проблема: p вне допустимых пределов
                        // и это не перехватывается
        ++p;
        *p = a;         // правильно
}
```

Мы можем заменить указатель *p* на объект класса ***Ptr_to_T***, к которому можно применять операцию разыменования только в том случае, если он действительно указывает на объект. Мы также хотели бы, чтобы к *p* можно было применять операции инкремента и декремента только в том случае, если он указывает на элемент массива, и в результате этих операций он по прежнему будет указывать на некий элемент массива. То есть, нам требуется что-то вроде следующего:

```
class Ptr_to_T {
        // ...
};
```

```
void f2 (T a)              // с проверкой
{
        T v[200];
        Ptr_to_T p (&v[0], v, 200);
        p--;
        *p = a;            // ошибка во время выполнения: p вне допустимого диапазона
        ++p;
        *p = a;            // правильно
}
```

Операторы инкремента и декремента уникальны среди операторов C++ тем, что их можно использовать как в качестве постфиксных, так и префиксных. Следовательно, мы должны определить как префиксный, так и постфиксный инкремент и декремент для ***Ptr_to_T***. Например:

```
class Ptr_to_T {
        T* p;
        T* array;
        int size;
public:
        Ptr_to_T (T* p, T* v, int s);      // связать с массивом v размера s
                                           // и начальным значением p
        Ptr_to_T (T* p);                   // связать с единственным объектом,
                                           // начальное значение p

        Ptr_to_T& operator++ ();           // префиксный
        Ptr_to_T operator++ (int);         // постфиксный

        Ptr_to_T& operator-- ();           // префиксный
        Ptr_to_T operator-- (int);         // постфиксный

        T& operator* ();                   // префиксный
};
```

Аргумент ***int*** используется для указания на постфиксную форму. Этот ***int*** никогда не будет задействован; аргумент является фиктивным и служит для распознавания префиксной и постфиксной форм. Чтобы запомнить, какая версия ***operator++*** является префиксной, обратите внимание, что она без фиктивного аргумента, точно также, как и все остальные унарные арифметические и логические операторы. Фиктивный аргумент используется только для «необычных» постфиксных ++ и --.

С использованием ***Ptr_to_T*** предыдущий пример эквивалентен следующему:

```
void f3 (T a)                      // с проверкой
{
        T v[200];
        Ptr_to_T p (&v[0], v, 200);
        p.operator-- (0);
        p.operator* () = a;        // ошибка во время выполнения:
                                   // p вне допустимого диапазона
        p.operator++ ();
        p.operator* () = a;        // правильно
}
```

Завершение класса ***Ptr_to_T*** оставлено в качестве упражнения (§ 11.14[19]). Его превращение в шаблон с использованием исключений для сообщений об ошибках во

время выполнения является еще одним упражнением (§ 14.12[2]). Примеры операторов ++ и — для итераций можно найти в § 19.3. Шаблон указателя с корректным поведением относительно наследования представлен в § 13.6.3.

11.12. Класс String

В этом разделе изложена более реалистичная версия класса ***String***. Этот класс является минимальной реализацией строк, которая бы меня устроила. В нем воплощена семантика значений, операции посимвольного чтения и записи, проверяемый и непроверяемый доступ, потоковый ввод/вывод, символьные строки в качестве литералов, операторы равенства и конкатенации. Класс хранит строку в виде массива символов с ограничивающим нулем в стиле C и использует механизм подсчета ссылок для минимизации операций копирования. Написание лучшего и/или реализующего больше возможностей класса является хорошим упражнением (§ 11.14[7–12]). После того, как это сделано, мы можем выбросить наши упражнения и пользоваться строками стандартной библиотеки (глава 20).

Мой «почти настоящий» класс ***String*** пользуется тремя вспомогательными классами: ***Srep***, который позволяет разделять действительное представление между несколькими объектами типа ***String*** с одинаковыми значениями; ***Range***, для генерации исключений в случае выхода за пределы диапазона, и ***Cref*** — для реализации оператора индексирования, который различает операции чтения и записи:

```
class String {
        struct Srep;            // представление
        Srep *rep;

        class Cref;             // ссылка на char

public:
        class Range { };        // для исключений

        // ...
};
```

Также как и другие члены, *класс членов* (часто называемый *вложенным классом*) может быть объявлен в самом классе, а определен позднее:

```
struct String::Srep {
        char* s;                        // указатель на элементы
        int sz;                         // количество символов
        int n;                          // количество обращений

        Srep(int nsz, const char* p)
        {
                n = 1;
                sz = nsz;
                s = new char[sz+1];     // зарезервировать место для завершающего нуля
                strcpy(s, p);
        }

        ~Srep() { delete[] s; }

        Srep* get_own_copy()            // сделать копию, если необходимо
        {
```

```
            if (n==1) return this;
            n--;
            return new Srep (sz, s);
        }
        void assign (int nsz, const char* p)
        {
            if (sz != nsz) {
                delete[] s;
                sz = nsz;
                s = new char[sz+1];
            }
            strcpy (s, p);
        }
    private:                                    // предохраняет от копирования
        Srep (const Srep&);
        Srep& operator= (const Srep&);
};
```

Класс ***String*** обеспечивает обычный набор конструкторов, деструкторов и операторов присваивания:

```
class String {
    // ...
    String ();                                  // x = ""
    String (const char*);                       // x = "abc"
    String (const String&);                     // x = other_string
    String& operator= (const char*);
    String& operator= (const String&);
    ~String ();
    // ...
};
```

Наш класс ***String*** имеет семантику значений. То есть, после присваивания ***s1=s2***, две строки ***s1*** и ***s2*** полностью отличны и последующие изменения в одной не оказывают никакого воздействия на другую. Альтернативой могло служить задание для ***String*** семантики указателей. При этом изменения в ***s2*** после ***s1=s2*** привели бы к изменению в ***s1***. Для типов с обычными арифметическими операциями, таких как комплексные числа, вектора, матрицы и строки, я предпочитаю семантику значений. Однако, для того, чтобы позволить себе роскошь семантики значений, класс ***String*** реализован в виде дескриптора, управляющего своим представлением, которое копируется только при необходимости:

```
String::String ()                       // значением по умолчанию является пустая строка
{
    rep = new Srep (0, "");
}
String::String (const String& x)        // копирующий конструктор
{
    x.rep->n++;
    rep = x.rep;                        // разделяемое представление
}
```

```
String::~String ()
{
    if (--rep->n == 0) delete rep;
}

String& String::operator= (const String& x)    // копирующее присваивание
{
    x.rep->n++;                                 // защита от st=st
    if (--rep->n == 0) delete rep;
    rep = x.rep;                                // разделяемое представление
    return *this;
}
```

Операции псевдокопирования с аргументами ***const char**** введены для того, чтобы разрешить строковые литералы:

```
String::String (const char* s)
{
    rep = new Srep (strlen (s), s);
}

String& String::operator= (const char* s)
{
    if (rep->n == 1)                            // обновить Srep
        rep->assign (strlen (s), s);
    else {                                      // использовать новый Srep
        rep->n--;
        rep = new Srep (strlen (s), s);
    }
    return *this;
}
```

Проектирование операторов доступа к строке является сложной проблемой, потому что (в идеале) доступ должен: осуществляться при помощи привычной формы записи (то есть с использованием []), быть максимально эффективен и осуществлять проверку диапазона. К сожалению, невозможно иметь все эти свойства одновременно. Я выбрал эффективные операции без проверки диапазона со слегка неудобной формой записи плюс чуть менее эффективные проверяемые операторы с привычной нотацией:

```
class String {
    // ...

    void check (int i) const { if (i<0 || rep->sz<=i) throw Range (); }

    char read  (int i) const { return rep->s[i]; }
    void write (int i, char c) { rep=rep->get_own_copy (); rep->s[i]=c; }

    Cref operator[] (int i) { check (i); return Cref (*this, i); }
    char operator[] (int i) const { check (i); return rep->s[i]; }

    int size () const { return rep->sz; }

    // ...
};
```

Идея состоит в том, чтобы при использовании обозначения [] производилась проверка, но пользователю предоставлена возможность оптимизации путем проверки диапазона один раз для нескольких обращений. Например:

```
int hash(const String& s)
{
    int h = s.read(0);
    const int max = s.size();
    // непроверяемое обращение к s
    for (int i=1; i<max; i++) h ^= s.read(i) >> 1;
    return h;
}
```

Сложно определить оператор типа [], который используется и для чтения и для записи, в тех случаях, когда неприемлемо просто возвратить ссылку и предоставить пользователю решать, что с ней делать. В нашем случае, подобное решение не подходит, потому что я определил ***String*** таким образом, что представление совместно используется различными объектами типа ***String***, которые присваивались, передавались в качестве аргументов и т. д. до тех пор, пока кто-то не записал в ***String***. Тогда и только тогда, представление копируется. Этот метод обычно называется *копированием при записи* (copy-on-write) или *отложенным копированием*. Фактическое копирование осуществляется вызовом ***String***::***Srep***::***get_own_copy*** ().

Для встраивания этих функций доступа определение ***Srep*** должно находиться в их области видимости. Из этого следует, что либо ***Srep*** должен быть определен в ***String***, либо функции доступа должны быть определены как ***inline*** вне ***String*** и после ***String***::***Srep*** (§ 11.14[2]).

Для того чтобы различать чтение и запись, ***String***::***operator***[] () возвращает ***Cref***, когда вызывается с неконстантным объектом. ***Cref*** ведет себя наподобие ***char***&, за исключением того, что он вызывает ***String***::***Srep***::***get_own_copy*** () при записи в него:

```
class String::Cref {                                    // ссылка на s[i]
friend class String;
    String& s;
    int i;
    Cref(String& ss, int ii) : s(ss), i(ii) { }
public:
    operator char() const  { return s.read(i); }        // выдать значение
    void operator=(char c) { s.write(i, c); }           // изменить значение
};
```

Например:

```
void f(String s, const String& r);
{
    int c1 = s[1];          // c1 = s.operator[](1).operator char()
    s[1] = 'c';             // s.operator[](1).operator=('c')

    int c2 = r[1];          // c2 = r.operator[](1)
    r[1] = 'd';             // ошибка: присваивание переменной типа char,
                            // r.operator[](1)='d'
}
```

Обратите внимание, что для неконстантного объекта, ***s.operator***[] (***1***). означает ***Cref*** (***s***, ***1***).

Для завершения класса ***String*** я добавил несколько полезных функций:

```
class String {
        //...

        String& operator+= (const String&);
        String& operator+= (const char*);

        friend ostream& operator<< (ostream&, const String&);
        friend istream& operator>> (istream&, String&);

        friend bool operator== (const String& x, const char* s)
                { return strcmp (x.rep->s, s) == 0; }

        friend bool operator== (const String& x, const String& y)
                { return strcmp (x.rep->s, y.rep->s) == 0; }

        friend bool operator!= (const String& x, const char* s)
                { return strcmp (x.rep->s, s) != 0; }

        friend bool operator!= (const String& x, const String& y)
                { return strcmp (x.rep->s, y.rep->s) != 0; }
};

String operator+ (const String&, const String&);
String operator+ (const String&, const char*);
```

С целью экономии места, я оставляю операции ввода/вывода и конкатенации в качестве упражнений.

Главная программа просто пользуется операторами ***String***:

```
String f (String a, String b)
{
        a[2] = 'x';
        char c = b[3]
        cout << "внутри f: " << a << ' ' << b << ' ' << c << '\n';
        return b;
}

int main ()
{
        String x, y;
        cout << "Пожалуйста, введите две строки\n";
        cin >> x >> y;
        cout << "введено: " << x << ' ' << y << '\n';
        String z = x;
        y = f (x, y);
        if (x != z) cout << "x испорчена!\n";
        x[0] = '!';
        if (x == z) cout << "ошибка записи!\n";
        cout << "выход: " << x << ' ' << y << ' ' << z << '\n';
}
```

В нашем классе ***String*** отсутствует много возможностей, которые вы, быть может, считаете важными или даже жизненно необходимыми. Например, в нем нет операции извлечения представления в виде С-строки (§ 11.14[10], глава 20).

11.13. Советы

[1] Определяйте операторы в основном для имитации привычной формы записи; § 11.1.
[2] Для больших по размерам операндов используйте аргументы с типом константных ссылок; § 11.6.
[3] При больших по размерам результатах рассмотрите возможность оптимизации возвращаемого значения; § 11.6.
[4] Отдавайте предпочтение операциям копирования по умолчанию, если они подходят для вашего класса; § 11.3.4.
[5] Замещайте или запрещайте копирование, если умолчание не подходит для данного типа; § 11.2.2.
[6] Отдавайте предпочтение функциям-членам над функциями-не-членами для операций, которым требуется непосредственный доступ к представлению; § 11.5.2.
[7] Отдавайте предпочтение функциям-не-членам по отношению к функциям-членам для операций, которым не требуется доступ к представлению; § 11.5.2.
[8] Пользуйтесь пространствами имен для указания связи функций-помощников с «их» классом; § 11.2.4.
[9] Пользуйтесь функциями-не-членами для симметричных операторов; § 11.3.2.
[10] Пользуйтесь оператором () для индексов в многомерных массивах; § 11.9.
[11] Объявляйте конструкторы с единственным «аргументом, задающим размер», как ***explicit***; § 11.7.1.
[12] В общих случаях отдавайте предпочтение стандартному классу ***string*** (глава 20), а не результатам ваших собственных упражнений; § 11.12.
[13] Будьте осторожны при введении неявных преобразований; § 11.4.
[14] Пользуйтесь функциями-членами для выражения операторов, которые требуют lvalue в качестве левого операнда; § 11.3.5.

11.14. Упражнения

1. (*2) Какие преобразования выполняются в каждом выражении в следующей программе?

```
Struct X {
        int i;
        X (int);
        X operator+ (int);
};

struct Y {
        int i;
        Y (X);
        Y operator+ (X);
        operator int ();
};

extern X operator* (X, Y);
extern int f (X);

        X x = 1;
```

```
Y y = x;
int i = 2,

int main ()
{
        i+10,       y+10;       y+10*y;
        x+y+i,      x*x+i;      f(7);
        f(y);       y+y;        106+y;
}
```

Измените программу таким образом, чтобы она при запуске выводила значения всех допустимых выражений.

2. (*2) Завершите и оттестируйте класс ***String*** из § 11.12.
3. (*2) Определите класс ***INT***, который ведет себя точно так же как ***int***. Подсказка: определите ***INT*****::operator int** ().
4. (*1) Определите класс ***RINT***, который ведет себя как ***int***, за исключением того, что допустимы только операторы + (унарный и бинарный), — (унарный и бинарный), *, / и %. Подсказка: не определяйте ***RINT*****::operator int** ().
5. (*3) Определите класс ***LINT***, который ведет себя как ***RINT***, но для представления чисел использует по крайней мере 64 бита.
6. (*4) Определите класс, реализующий арифметику с произвольной точностью. Оттестируйте его, вычислив факториал 1000. Подсказка: вам потребуется управление памятью наподобие того, как это сделано в классе ***String***.
7. (*2) Определите внешний итератор для класса ***String:***

```
class String_iter {
        // ссылка на строку и элемент строки
public:
        String_iter (String& s),          // итератор для s
        char& next ();                    // ссылка на следующий элемент
        // операции по вашему выбору
};
```

Сравните это по удобству, стилю программирования и эффективности с внутренним итератором для ***String*** (который бы вводил понятие текущего элемента для ***String*** и операции, относящиеся к этому элементу).

8. (*1.5) Реализуйте оператор взятия подстроки для класса строк, используя перегрузку (). Какие еще операции вы хотели бы выполнять со строками?
9. (*3) Разработайте класс ***String*** таким образом, чтобы оператор подстроки можно было использовать в левой части оператора присваивания. Сначала напишите версию, в которой строка может быть присвоена подстроке того же размера. Затем напишите версию, в которой размеры могут отличаться.
10. (*2) Определите оператор для ***String***, который возвращает представление строки в виде С-строки. Обсудите все за и против реализации такой операции в виде оператора преобразования. Обсудите различные варианты выделения памяти под представление в виде С-строки.
11. (*2.5) Определите и реализуйте простое средство для поиска подстрок в строках ***String***.
12. (*1.5) Модифицируйте предыдущий пример из § 11.14[11] таким образом, чтобы он работал со строками ***string*** из стандартной библиотеки. Вы не можете изменить определение ***string***.

13. (*2) Напишите программу, которая будет совершенно нечитабельна из-за использования перегрузки операторов и макросов. Идея: определите +, имеющий смысл оператора – (и наоборот) для ***INT***, затем напишите макрос, который заставлял бы *int* значить ***INT***. Замените часто употребляемые функции с использованием ссылок в качестве аргументов. Несколько неверных комментариев также создадут изрядную путаницу.
14. (*3) Обменяйтесь результатами упражнения § 11.14[13] с кем-нибудь из своих друзей. Не запуская, попытайтесь определить, что делает программа вашего приятеля. Теперь вы знаете, чего не следует делать.
15. (*2) Определите тип ***Vec4*** как вектор из четырех ***float***. Определите ***operator***[] для ***Vec4***. Определите операторы +, –, *, /, =, +=, –=, *= и /= для комбинаций векторов и чисел с плавающей точкой.
16. (*3) Определите класс ***Mat4*** как вектор из четырех ***Vec4***. Определите ***operator***[], возвращающий ***Vec4*** из ***Mat4***. Определите обычные матричные операции для этого типа. Определите функцию, реализующую метод исключения Гаусса.
17. (*2) Определите класс ***Vector*** по аналогии с ***Vec4***, но с размером, являющимся аргументом конструктора ***Vector***::***Vector*** (*int*).
18. (*3) Определите класс ***Matrix*** по аналогии с ***Matrix4***, но с размерами, задаваемыми в качестве аргументов конструктора ***Matrix***::***Matrix*** (*int*, *int*).
19. (*2) Завершите класс ***Ptr_to_T*** из § 11.11 и протестируйте его. Чтобы быть полным, ***Ptr_to_T*** должен, по крайней мере, иметь операторы *, ->, =, ++ и --. Сделайте так, чтобы ошибка на этапе выполнения не возникала до того, как реально производится разыменование «висячей» ссылки.
20. (*1) Используя две структуры:

```
struct S { int x, y; };
struct T { char* p; char* q; };
```

напишите класс ***C***, который допускает использование ***x*** и ***p*** из некоторых ***S*** и ***T***, примерно так же, как если бы ***x*** и ***p*** были членами ***C***.
21. (*1.5) Определите класс ***Index*** для хранения степени показательной функции ***mypow*** (***double***, ***Index***). Найдите способ, при котором ***2**I*** вызывает ***mypow*** (***2***, ***I***).
22. (*2) Определите класс ***Imaginary*** для представления мнимых чисел. Определите на его основе класс ***Complex***. Реализуйте основные арифметические операции.

12

Производные классы

Не множьте объекты без необходимости.
— В. Оккам

Концепции и классы — производные классы — функции-члены — конструирование и уничтожение — иерархии классов — поля типа — виртуальные функции — абстрактные классы — традиционные иерархии классов — абстрактные классы как интерфейсы — локализация создания объекта — абстрактные классы и иерархии классов — советы — упражнения.

12.1. Введение

C++ позаимствовал из Simula концепции класса, как определяемого пользователем типа, и иерархий классов. Кроме того, в системное проектирование была привнесена идея, что классы должны использоваться для моделирования концепций реального и программного мира. C++ предоставляет конструкции языка, непосредственно поддерживающие эту концепцию проектирования. С другой стороны, тесная связь с концепциями проектирования делает C++ очень эффективным. Использование конструкций языка просто в качестве подпорок для традиционного программирования означает потерю основных достоинств C++.

Ни одна концепция не существует в изоляции. Она сосуществует с родственными концепциями и именно этой связи она обязана своей мощью. Попытайтесь, например, объяснить, что такое автомобиль. Вскоре вы введете понятия колес, двигателей, водителей, пешеходов, грузовиков, скорой помощи, дорог, масла, штрафов за превышение скорости, мотелей и т. д. Так как мы пользуемся классами для представления концепций, возникает вопрос: как представить отношения между концепциями? Мы не можем выразить произвольные отношения непосредственно на языке программирования. Если бы и могли, мы не захотели бы этого делать. Наши классы должны быть уже реальных понятий и более точно определены. Понятие производного класса и связанные с ним механизмы языка предназначены для выражения иерархических отношений, то есть для отражения общности классов. Например, концепции круга и треугольника связаны — они являются фигурами; то есть концепция фигуры является общей для них. Поэтому мы должны явно определить, что классы ***Circle*** (круг) и ***Triangle*** (треугольник) имеют общий класс ***Shape*** (фигура). Представление круга и треугольника в программе без введения понятия «фигура» означало бы потерю чего-то существенного. В этой главе тщательно выясняются следствия этой простой идеи, которая является основой так называемого объектно-ориентированного программирования.

Мы представляем средства языка и соответствующие техники программирования в порядке от простого и конкретного к более изощренному и абстрактному. Для многих программистов это также будет означать движение от хорошо знакомого к менее знакомому. Это не просто путешествие от «старых плохих методов» к «единственно правильному способу». Когда я указываю на ограничения одного метода, чтобы оправдать переход к другому, я делаю это в контексте конкретных проблем; для других задач первый метод может оказаться наилучшим. С помощью всех упоминающихся здесь методов было написано вполне работоспособные и полезные программы. Наша цель состоит в том, чтобы помочь вам понять суть этих методов, дабы сделать разумный и сбалансированный выбор для реальных проблем.

В этой главе я сначала опишу базовые средства языка для поддержки объектно-ориентированного программирования. Затем использование этих средств для проектирования хорошо структурированных программ будет проанализировано на «большом» примере. Другие средства поддержки объектно-ориентированного программирования, такие как множественное наследование и определение типа во время выполнения, обсуждаются в главе 15.

12.2. Производные классы

Допустим мы создаем программу, обрабатывающую информацию о сотрудниках (employee) фирмы. Для такой программы может понадобиться следующая структура:

```
struct Employee {
        string first_name, family_name,
        char middle_initial;
        Date hiring_date;
        short department,
        // ...
};
```

Затем мы можем попытаться определить менеджера:

```
struct Manager {
        Employee emp,              // сведения о менеджере как о сотруднике
        set<Employee*> group,      // подчиненные
        short level,
        // ...
};
```

Менеджер также является сотрудником; соответствующие данные хранятся в члене ***emp*** объекта ***Manager***. Внимательно читающему человеку это может быть и очевидно, но здесь ничто не говорит компилятору и другим инструментам, что ***Manager*** является в то же время и ***Employee***. ***Manager**** не является ***Employee****, поэтому нельзя просто использовать один объект там, где требуется другой. В частности, нельзя поместить ***Manager*** в список объектов ***Employee***, не написав специальный код. Мы можем либо воспользоваться явным преобразованием типа для указателя ***Manager****, либо поместить адрес члена ***emp*** в список сотрудников. Однако оба решения не элегантны и могут только запутать дело. Правильный подход — явно указать, что менеджер является сотрудником, а кроме того характеризуется некоторой дополнительной информацией.

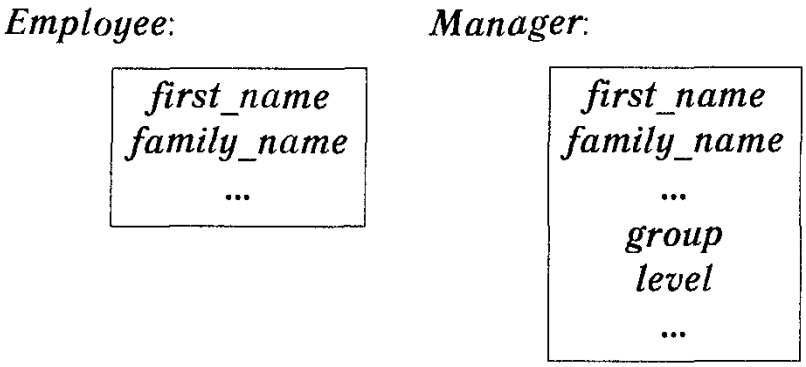
```
Struct Manager : public Employee {
        set<Employee*> group;
        short level;
        // ...
};
```

Класс ***Manager*** является *производным* от ***Employee***, а ***Employee*** является *базовым классом* для ***Manager***. Класс ***Manager*** кроме своих собственных членов (***group***, ***level*** и т. д.) содержит члены класса ***Employee*** (***first_name***, ***department*** и т. д.).

Такое отношение между классами часто представляют в графическом виде при помощи стрелки, указывающей из производного класса в базовый, тем самым производный класс ссылается на базовый (а не наоборот):

Часто говорят, что производный класс наследует свойства базового, поэтому их отношения нередко называют *наследованием*. Базовый класс иногда называют *суперклассом*, а производный — *подклассом*. Однако подобная терминология вводит в заблуждение людей, которые замечают, что данные в объекте производного класса являются надмножеством данных базового класса. Производный класс больше базового в том смысле, что он содержит больше данных и функций.

Популярной и эффективной реализацией понятия производных классов является представление объекта производного класса в виде объекта базового класса и информации, относящейся только к производному классу, добавленной в конец. Например:



Вывод ***Manager*** из ***Employee*** таким образом делает ***Manager*** подтипом ***Employee***. Следовательно объектом типа ***Manager*** можно пользоваться везде, где допустим ***Employee***. Например, мы можем создать список сотрудников ***Employee***, некоторые из элементов которого будут иметь тип ***Manager***:

```
void f(Manager m1, Employee e1)
{
        list<Employee*> elist;

        elist.push_front(&m1);
        edit.push_front(&e1);
        // ...
}
```

Manager является ***Employee***, поэтому ***Manager**** можно использовать как ***Employee****. Однако ***Employee*** не обязательно ***Manager***, поэтому ***Employee**** нельзя использовать

в качестве ***Manager****. В общем, если производный класс ***Derived*** имеет в качестве базового открытого класса (§ 15.3) ***Base***, то ***Derived**** может быть присвоен переменной типа ***Base**** без явного преобразования типа. Обратное преобразование из ***Base**** в ***Derived**** должно быть явным. Например:

```
void g (Manager mm, Employee ee)
{
    Employee* pe = &mm;                    // правильно: каждый Manager
                                           // является Employee
    Manager* pm = &ee;                     // ошибка: не каждый Employee
                                           // является Manager

    pm->level = 2,                         // катастрофа: у переменной pm нет level

    pm = static_cast<Manager*> (pe);       // грубая сила: pe указывает
                                           // на переменную mm класса Manager

    pm->level = 2;                         // прекрасно: pm указывает
                                           // на переменную mm класса Manager,
                                           // у которой есть level
}
```

Другими словами, с объектом производного класса можно обращаться как с объектом базового класса при обращении к нему при помощи указателей и ссылок. Обратное неверно. Использование ***static_cast*** и ***dynamic_cast*** обсуждается в § 15.4.2.

Использование класса в качестве базового эквивалентно объявлению (неименованного) объекта этого класса. Следовательно, для того чтобы класс можно было использовать в качестве базового, он должен быть определен (§ 5.7):

```
class Employee;                            // только объявление (без определения)

class Manager : public Employee {          // ошибка: Employee не определен
    //...
};
```

12.2.1. Функции-члены

Простые структуры данных, такие как ***Employee*** и ***Manager***, как правило не представляют большого интереса и не особенно полезны. То что нам требуется — это представление информации в виде подходящего типа с соответствующим набором операций, отражающих концепцию. Мы хотим это сделать, не вникая в детали конкретной реализации. Например:

```
class Employee {
    string first_name, family_name;
    char middle_initial;
    //...
public:
    void print () const;
    string full_name () const
        { return first_name + ' ' + middle_initial + ' ' + family_name; }
    //...
};
```

```
class Manager : public Employee {
        // ...
public:
        void print () const;
        // ...
};
```

Член производного класса может пользоваться открытыми (и защищенными, см. § 15.3) членами базового класса так, как будто они объявлены в самом производном классе. Например:

```
void Manager::print () const
{
        cout << "Имя: " << full_name () << '\n';
        // ...
}
```

Однако производный класс не может использовать закрытые имена базового класса:

```
void Manager::print () const
{
        cout << "Имя: " << family_name << '\n';  // ошибка!
        // ...
}
```

Вторая версия ***Manager::print*** () не будет откомпилирована. Члены производного класса не имеют специального разрешения на доступ к закрытым членам их базовых классов, поэтому к ***family_name*** нельзя обращаться из ***Manager::print*** ().

Для некоторых это является сюрпризом, но давайте рассмотрим альтернативный подход: функция-член производного класса может осуществлять доступ к закрытым членам базового. Концепция закрытых членов стала бы бессмысленной, если бы мы позволили программисту получить доступ к закрытой части класса, просто создав на его основе новый класс. Более того, невозможно будет найти все случаи использования закрытого имени путем просмотра функций-членов и функций-друзей класса. Пришлось бы проверить все файлы исходных текстов программы в поисках производных классов, затем проверить все функции в этих классах, затем найти все производные от этих классов и т. д. Это, в лучшем случае, утомительно, а часто и просто нереально. Там, где это приемлемо, можно воспользоваться защищенными членами вместо закрытых. Защищенный член ведет себя как открытый по отношению к члену производного класса и как закрытый по отношению к другим функциям (см. § 15.3).

Как правило, самым понятным решением является использование в производном классе только открытых членов его базового класса. Например:

```
void Manager::print () const
{
        Employee::print ();      // печать информации о сотруднике
        cout << level;           // печать информации, относящейся только к менеджеру
        // ...
}
```

Обратите внимание, что должен использоваться оператор ::, потому что ***print*** () была замещена в ***Manager***. Такое повторное использование имен является типичным. Опрометчивый человек мог бы написать:

```
void Manager::print() const
{
        print();            // не то!!!

        // печать информации, относящейся только к менеджеру
}
```

что неожиданно повлекло бы последовательность рекурсивных вызовов.

12.2.2. Конструкторы и деструкторы

Некоторые производные классы нуждаются в конструкторах. Если базовый класс имеет конструктор, он должен быть вызван. Конструкторы по умолчанию могут быть вызваны неявно. Однако если все конструкторы базового класса требуют указания аргументов, то конструктор этого базового класса должен быть вызван явным образом. Рассмотрим:

```
class Employee {
        string first_name, family_name;
        short department;
        // ...
public:
        Employee(const string& n, int d);
        // ...
};

class Manager : public Employee {
        set<Employee*> group;       // подчиненные
        short level;
        // ...
public:
        Manager(const string& n, int d, int lvl);
        // ...
};
```

Аргументы конструктора базового класса указываются в определении конструктора производного класса. В этом отношении базовый класс ведет себя как член производного класса (§ 10.4.6). Например:

```
Employee::Employee(const string& n, int d):
        family_name(n), department(d)          // инициализация членов
{
        // ...
}

Manager::Manager(const string& n, int d, int lvl):
        Employee(n, d),          // инициализация базового класса
        level(lvl)               // инициализация членов
{
        // ...
}
```

Конструктор производного класса может указать инициализаторы для своих собственных членов и членов базового класса; он не может непосредственно инициализировать члены базового класса. Например:

```
Manager::Manager(const string& n, int d, int lvl):
        family_name(n),        // ошибка: family_name не объявлен в Manager
        department(d),         // ошибка: department не объявлен в Manager
            level(lvl)
{  //...  }
```

Это определение содержит три ошибки: не вызывается конструктор ***Employee*** и дважды осуществляется попытка инициализации членов ***Employee*** непосредственно.

Объекты класса создаются снизу-вверх: сначала базовый класс, потом члены и затем сам производный класс. Они уничтожаются в противоположном порядке: сначала производный класс, члены, а затем базовый класс. Члены и подобъекты базовых классов конструируются в порядке их объявления в классе и уничтожаются в обратном порядке. См. также § 10.4.6 и § 15.2.4.1.

12.2.3. Копирование

Копирование объектов класса определяется копирующими конструктором и операторами присваивания (§ 10.4.4.1). Рассмотрим пример:

```
class Employee {
        //...
        Employee& operator+ (const Employee&);
        Employee(const Employee&);
};

void f(const Manager& m)
{
        Employee e = m;        // создание e из Employee-части m
        e = m;                 // присвоить Employee-часть m переменной e
}
```

Так как копирующая функция ***Employee*** ничего не знает о ***Manager***, копируется только ***Employee***-часть ***Manager***. Этот эффект часто называют *срезкой*; он нередко приводит к ошибкам и чреват сюрпризами. Одной из причин передачи указателей и ссылок на объекты в иерархии является желание избежать срезки. Другими причинами являются обеспечение полиморфного поведения (§ 2.5.4, § 12.2.6) и эффективности.

Заметьте, что если вы не определите оператор копирующего присваивания, он будет сгенерирован компилятором (§ 11.7). Отсюда следует, что операторы присваивания не наследуются. Конструкторы никогда не наследуются.

12.2.4. Иерархия классов

Производный класс, в свою очередь, сам может быть базовым классом. Например:

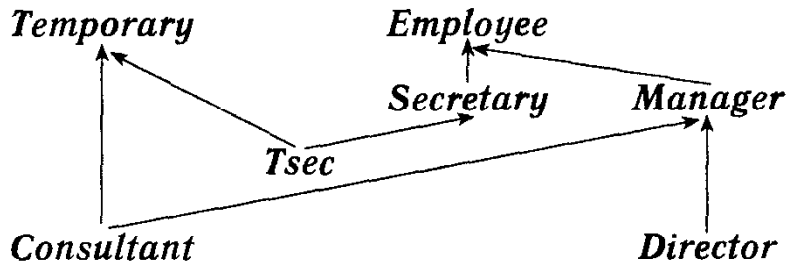
```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```

Такой набор связанных классов традиционно называется *иерархией классов*. Иерархия, в большинстве случаев, является деревом, но она может иметь и более общую структуру графа. Например:

```
class Temporary { /* ... */ };          // временный сотрудник
```

```
class Secretary: public Employee { /* ... */ };   // секретарь
class Tsec : public Temporary,
        public Secretary { /* ... */ };           // временный секретарь
class Consultant : public Temporary,              // временный консультант
        public Manager { /* ... */ };
```

Или в графическом виде:



Таким образом на C++ можно выразить направленный ациклический (то есть не содержащий циклов) граф классов. Подробнее см. в § 15.2.

12.2.5. Поля типа

Для того чтобы использовать производные классы в качестве чего-то большего, чем просто удобного сокращения объявления, мы должны решить следующую проблему: есть указатель типа ***base****; какому производному типу на самом деле принадлежит объект, на который он указывает? Существует четыре фундаментальных решения этой проблемы:

[1] Гарантировать, что ссылки осуществляются только на объекты одного типа (§ 2.7 и глава 13).

[2] Поместить поле с информацией о типе (поле типа) в базовый класс, чтобы заинтересованные функции могли его проверять.

[3] Использовать динамическое преобразование типа (*dynamic_cast*) (§ 15.4.2, § 15.4.5).

[4] Использовать виртуальные функции.

Указатели на базовые классы обычно используются при проектировании *классов-контейнеров*, таких как множества, вектора и списки. В подобных случаях решение [1] приведет к однородному списку, то есть списку объектов одного типа. Решениями [2], [3] и [4] можно пользоваться для построения разнородных списков, то есть списков, хранящих (указатели на) объекты нескольких различных типов. Решение [3] является вариантом решения [2], возможным при непосредственной поддержке со стороны языка. Решение [4] является специальным вариантом решения [2], безопасным с точки зрения типов. Комбинации решений [1] и [4] весьма мощны и очень интересны; почти во всех ситуациях они приводят к более ясному коду, чем [2] и [3].

Давайте сначала рассмотрим простое решение [2] с полем типа для того, чтобы понять, почему в большинстве случаев его стоит избегать. Пример менеджер/сотрудник можно заместить следующим образом:

```
struct Employee {
        enum Empl_type { M, E };          // тип сотрудника: менеджер или нет
        Empl_type type;

        Employee () : type (E) { }        // по умолчанию — не менеджер
```

```
        string first_name, family_name;
        char middle_initial;

        Date hiring_date;
        short department;
        //...
};
struct Manager : public Employee {
        Manager () { type = M; }

        set<Employee*> group;        // подчиненные
        short level;
        //...
};
```

Теперь мы можем написать функцию, которая выводит информацию о любом сотруднике:

```
void print_employee (const Employee* e)
{
        switch (e->type) {
        case Employee::E:
                cout << e->family_name << '\t' << e->department << '\n';
                //...
                break;
        case Employee::M:
        {       cout << e->family_name << '\t' << e->department << '\n';
                //...
                const Manager* p = static_cast<const Manager*> (e);
                cout << " level " << p->level << '\n';
                //...
                break;
        }
        }
}
```

Это работает прекрасно, особенно в небольшой программе, сопровождаемой одним человеком. Однако такое решение имеет врожденный недостаток: характер зависимости программы от типов, с которыми манипулирует программист, таков, что компилятор не имеет возможности осуществить проверку. Обычно ситуация еще ухудшается, потому что функции типа ***print_employee*** () организованы так, чтобы воспользоваться общностью классов. Например:

```
void print_employee (const Employee* e)
{
        cout << e->family_name << '\t' << e->department << '\n';
        //...
        if (e->type == employee::M) {
                const Manager* p = static_cast<const Manager*> (e);
                cout << " level " << p->level << '\n';
                //...
        }
}
```

Поиск всех таких проверок поля типа, «похороненных» в большой функции, которая работает со многими производными классами, может оказаться сложной задачей. Даже если все они найдены, бывает сложно понять, что на самом деле происходит. Более того, добавление нового производного от ***Employee*** класса подразумевает внесение изменений во все ключевые функции системы, которые содержат проверку поля типа. Программист после внесения каких-либо изменений должен проанализировать каждую функцию, которая (хотя бы потенциально) может зависеть от проверок поля типа. Это подразумевает доступ к основным частям исходного кода и дополнительные усилия на тестирование измененного кода. Сам факт использования явного преобразования типа служит недвусмысленным указанием на то, что существует лучшее решение.

Другими словами, использование поля типа подвержено ошибкам и ведет к проблемам сопровождения. Сложности увеличиваются по мере роста программы, потому что поле типа является надругательством над идеалами модульности и сокрытия данных. Каждая функция, использующая поле типа, должна знать о представлении и других деталях реализации каждого класса, производного от класса с полем типа.

Кроме того, складывается впечатление, что наличие любых общих данных (вроде поле типа), доступных из любого производного класса, вызывает у программистов искушение добавлять такие данные еще и еще. Общий базовый класс становится как бы хранилищем всякого рода «полезной информации». Это, в свою очередь, создает нежелательные дополнительные зависимости между реализациями базового и производных классов. Ради ясности проекта и упрощения сопровождения мы хотим разделять отдельные понятия и избегать взаимозависимостей.

12.2.6. Виртуальные функции

Виртуальные функции решают проблему, связанную с полем типа, предоставляя возможность программисту объявить в базовом классе функции, которые можно заместить в каждом производном классе. Компилятор и загрузчик гарантируют правильное соответствие между объектами и функциями, применяемыми к ним. Например:

```
class Employee {
        string first_name, family_name;
        short department;
        // ...
public:
        Employee (const string& name, int dept);
        virtual void print () const;
        // ...
};
```

Ключевое слово ***virtual*** означает, что ***print*** () может действовать в качестве интерфейса как к функции ***print*** (), определенной в данном классе, так и к функции ***print*** (), определенной в производном классе. Если такие функции ***print*** () определены в производных классах, компилятор обеспечивает вызов соответствующей функции ***print*** () для любого объекта класса, производного от ***Employee***.

Для того чтобы объявление виртуальной функции работало в качестве интерфейса к функциям, определенным в производных классах, типы аргументов функции в производном классе не должны отличаться от типов аргументов, объявленных в ба-

зовом классе, и только очень небольшие изменения допускаются для типа возвращаемого значения (§ 15.6.2). Виртуальную функцию-член иногда называют *методом*.

Виртуальная функция *должна* быть определена для класса, в котором она впервые объявлена (если только она не объявлена в виде чисто виртуальной функции; см. § 12.3). Например:

```
void Employee::print() const
{
    cout << family_name << '\t' << department << '\n';
    //...
}
```

Виртуальную функцию можно использовать, даже если у ее класса нет производных классов. Производный класс, который не нуждается в собственной версии виртуальной функции, не обязан ее реализовывать. При создании производного класса вы можете реализовать требуемую функцию, только если в этом есть необходимость. Например:

```
class Manager : public employee {
    set<Employee*> group;
    short level;
    //...
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    //...
};

void Manager::print() const
{
    Employee::print();
    cout << "\tlevel " << level << '\n';
    //...
}
```

Говорят, что функция из производного класса с тем же именем и с тем же набором типов аргументов, что и виртуальная функция в базовом классе, *замещает* (override) виртуальную функцию из базового класса. За исключением случаев, когда мы явно указываем, какая версия виртуальной функции должна вызываться (как в случае ***Employee::print()*** — вызов функции из базового класса), активизируется наиболее подходящий для вызывающего объекта вариант замещенной функции.

Теперь глобальная функция ***print_employee()*** (§ 12.2.5) не нужна, потому что ее место заняли функции-члены ***print()***. Список сотрудников можно вывести следующим образом:

```
void print_list(const list<Employee*>& s)
{
    for (list<Employee*>::const_iterator p = s.begin(); p!=s.end(); ++p)    // см. § 2.7.2
        (*p)->print();
}
```

или даже

```
void print_list (const list<Employee*>& s)
{
        for_each (s.begin (), s.end (), mem_fun (&Employee::print));      // см. § 3.8.5
}
```

Каждый сотрудник будет «выведен» в соответствии с его типом. Например:

```
int main ()
{
        Employee e ("Brown", 1234);
        Manager m ("Smith", 1234, 2);
        list<Employee*> empl;
        empl.push_front (&e),          // см. § 2.5.4
        empl push_front (&m);
        print_list (empl);
}
```

выведет:

```
Smith 1234
        level 2
Brown 1234
```

Обратите внимание, что все будет работать даже если функция ***print_list*** () была написана и откомпилирована до того, как класс ***Manager*** был вообще задуман! Это является очень важным моментом работы с классами. При правильном использовании данный факт служит краеугольным камнем объектно-ориентированных проектов и придает стабильность развивающийся программе.

Когда функции ***Employee*** ведут себя «правильно» независимо от того, какой конкретно производный от ***Employee*** класс используется, это называется *полиморфизмом*. Тип, имеющий виртуальные функции, называется *полиморфным типом*. Для достижения полиморфного поведения в C++ вызываемые функции-члены должны быть виртуальными, и доступ к объектам должен осуществляться через ссылки или указатели. При непосредственных манипуляциях с объектом (без помощи указателя или ссылки) его точный тип известен компилятору, и поэтому полиморфизм времени выполнения не требуется.

Понятно, что для реализации полиморфизма компилятор должен хранить некоторую информацию о типе в каждом объекте класса ***Employee*** и пользоваться ею для вызова нужной версии виртуальной функции ***print*** (). В типичной реализации используемое для этих целей количество памяти равно размеру указателя (§ 2.5.5). Это пространство используется в объектах классов с виртуальными функциями, а не в каждом объекте вообще и даже не в каждом объекте производного класса. Вы расплачиваетесь памятью только за классы, в которых объявляете виртуальные функции. Если бы вы выбрали другое решение — например поле типа — для хранения типа потребовалось бы сравнимое количество памяти.

Вызов функции с использованием оператора разрешения области видимости ::, как это сделано в ***Manager::print*** (), гарантирует, что виртуальный механизм не задействуется. В противном случае вызов ***Manager::print*** привел бы к бесконечной рекурсии. Использование имен с квалификатором дает еще один положительный эффект. Если виртуальная функция объявлена встраиваемой (inline), что встречается довольно часто, и задействован оператор ::, то может использоваться встраивание фун-

кции в место вызова. Это предоставляет программисту эффективное средство для важных специальных случаев, когда одна виртуальная функция вызывает другую с тем же объектом. Примером является функция ***Manager::print*** (). Так как тип объекта определен при вызове ***Manager::print*** (), нет необходимости динамически определять его снова при последнем вызове ***Employee::print*** ().

Стоит помнить, что традиционной и очевидной реализацией вызова виртуальной функции является просто косвенный вызов функции (§ 2.5.5), поэтому соображения эффективности не должны никого отпугивать от применения виртуальных функций там, где считается приемлемым вызов обычной функции.

12.3. Абстрактные классы

Многие классы схожи с классом ***Employee*** в том, что они полезны как сами по себе, так и в качестве базы для производных классов. Для таких классов методы, описанные в предыдущем разделе, являются вполне достаточными. Однако не все классы соответствуют такому образцу. Некоторые классы, такие как ***Shape*** (фигура), представляют абстрактную концепцию, для которой не могут существовать объекты. Класс ***Shape*** имеет смысл только в качестве базы для производных классов. Это можно усмотреть хотя бы из того факта, что невозможно разумно определить виртуальные функции в самом ***Shape***:

```
class Shape {
public
        virtual void rotate (int) { error ("Shape::rotate"); }      // как вращать?
        virtual void draw () { error ("Shape::draw"); }             // как рисовать?
        // ...
};
```

Попытка создания фигуры такого рода допустима, но неразумна:

```
Shape s;                                    // глупо: «фигура без формы»
```

Такое объявление неразумно, потому что каждая операция с ***s*** приводит к выводу сообщения об ошибке.

Лучшей альтернативой является объявление виртуальных функций класса ***Shape*** в виде *чисто виртуальных функций*. Виртуальная функция делается «чистой» при помощи инициализатора = ***0***:

```
class Shape {                               // абстрактный класс
public:
        virtual void rotate (int) = 0;      // чисто виртуальная функция
        virtual void draw () = 0;           // чисто виртуальная функция
        virtual void is_closed () = 0;      // чисто виртуальная функция
        // ...
};
```

Класс с одной или несколькими чисто виртуальными функциями называется *абстрактным классом*. Невозможно создать объект абстрактного класса:

```
Shape s;                                    // ошибка: переменная абстрактного класса Shape
```

Абстрактный класс можно использовать только как интерфейс и в качестве базы для других классов. Например:

```
class Point { /* ... */ };

class Circle : public Shape {
        Point center;
public:
        void rotate (int) { }                   // замещение Shape::rotate
        void draw ();                           // замещение Shape::draw
        bool is_closed () { return true; }      // замещение Shape::is_closed

        Circle (Point p, int r);
private:
        Point center,
        int radius;
};
```

Чисто виртуальная функция, которая не определена в производном классе, остается чисто виртуальной, поэтому такой производный класс, как и базовый, является абстрактным. Это позволяет нам строить реализацию поэтапно:

```
class Polygon : public Shape {                  // абстрактный класс многоугольников
public:
        bool is_closed () { return true; }      // замещение Shape::is_closed
        // ... draw и rotate не замещены ...
};

Polygon b;              // ошибка: объявление объекта абстрактного класса Polygon

class Irregular_polygon : public Polygon {
        list<Point> lp;
public:
        void draw ();                           // замещение Shape::draw
        void rotate (int);                      // замещение Shape::rotate
        // ...
};

Irregular_polygon poly (some_points);           // правильно (в предположении
                                                // наличия конструктора)
```

Важным примером использования абстрактных классов является предоставление интерфейса с полным отсутствием деталей реализации. Например, операционная система может скрыть детали реализации драйверов устройств за интерфейсом абстрактного класса:

```
class Character_device {
public:
        virtual int open (int opt) = 0;
        virtual int close (int opt) = 0;
        virtual int read (char* p, int n) = 0;
        virtual int write (const char* p, int n) = 0;
        virtual int ioctl (int ...) = 0;
        virtual ~Character_device () { }        // виртуальный деструктор
};
```

Мы можем определить драйверы в виде классов, производных от ***Character_device***, и управлять драйверами через этот интерфейс. Важность виртуального деструктора объясняется в § 12.4.2.

Теперь, после знакомства с абстрактными классами, мы имеем основные средства написания законченных программ в стиле модульного программирования с использованием классов в качестве строительных блоков.

12.4. Проектирование иерархий классов

Рассмотрим простую проектную задачу: ввод в программу целого значения через пользовательский интерфейс. Это можно сделать огромным количеством способов. Для выделения нашей программы из этого множества, а также чтобы иметь возможность исследовать различные варианты проекта, давайте начнем с определения программной модели столь простой операции ввода. Мы оставим на потом детали реализации с использованием реальной системы пользовательского интерфейса.

Идея состоит в создании класса ***Ival_box***, который знает диапазон воспринимаемых значений. Программа может затребовать у ***Ival_box*** его значение, а также попросить этот класс выдать пользователю приглашение на ввод (когда необходимо). Кроме того, программа может спросить ***Ival_box***, менял ли пользователь значение с момента последнего запроса.

Так как существует множество способов реализации этой базовой идеи, мы должны предположить, что может существовать много различных видов ***Ival_box***, таких как ползунки (sliders), простые диалоговые окна, в которые пользователь вводит числа, кнопки с числами и ввод с голоса.

Общий подход состоит в построении «виртуальной системы пользовательского интерфейса» для использования в приложениях. Эта система предоставляет некоторые услуги, реализованные в существующих системах пользовательского интерфейса. Она должна быть реализована в широком наборе систем, поэтому следует учитывать переносимость кода. Естественно, существуют другие способы изоляции приложения от системы пользовательского интерфейса. Я выбрал этот подход, потому что он позволяет мне продемонстрировать множество технологий и методов проектирования. Эти методы использовались при построении «реальных» систем пользовательского интерфейса и, что наиболее важно, они применимы для решения проблем, далеко выходящих за рамки интерфейсных систем.

12.4.1. Традиционная иерархия классов

Наше первое решение основывается на традиционной иерархии классов, существующей в Simula, Smalltalk и ранних программах на C++.

Класс ***Ival_box*** определяет базовый интерфейс ко всем классам ***Ival_box*** и вводит реализацию по умолчанию, которую более специальные варианты ***Ival_box*** могут заместить. Кроме того, мы объявим данные, необходимые для реализации основного понятия:

```
class Ival_box {
protected:
        int val;                // значение
        int low, high;          // нижняя и верхняя границы
        bool changed;           // индикатор изменений
public:
        Ival_box (int ll, int hh) { changed = false, val = low = ll; high = hh; }
```

```
        virtual int get_value () { changed = false; return val; }
        virtual void set_value (int i) { changed = true; val = i; }      // для пользователя
        virtual void reset_value (int i) { changed = false; val = i; }   // для приложения

        virtual void prompt () { }
        virtual bool was_changed () const { return changed; }
};
```

Реализация функций по умолчанию довольно небрежна и приведена здесь в первую очередь для иллюстрации их предполагаемого смысла. В реалистичном классе, по крайней мере, производилась бы проверка диапазона значений.

Программист мог бы использовать подобные «классы ***Ival***» следующим образом:

```
void interact (Ival_box* pb)
{
        pb->prompt (),                  // побеспокоить пользователя
        // ...
        int i = pb->get_value ();
        if (pb->was_changed ()) {
                // новое значение; делаем что-нибудь
        }
        else {
                // старое значение устраивало; делаем что-нибудь другое
        }
        // ...
}

void some_fct ()
{
        // Ival_slider является производным от Ival_box
        Ival_box* p1 = new Ival_slider (0, 5);
        interact (p1),

        Ival_box* p2 = new Ival_dial (1, 12);
        interact (p2);
}
```

Большая часть кода приложения написана в терминах обычных ***Ival_box*** (указателей на них), как например ***interact*** (). При этом приложение не должно знать о потенциально большом количестве разнообразных ***Ival_box***. Знание о таких специализированных классах сосредоточено в относительно немногих функциях, создающих соответствующие объекты. Это изолирует пользователя от изменений в реализациях производных классов. Большая часть кода может не обращать внимания на факт наличия различных видов ***Ival_box***.

С целью упрощения обсуждения я не рассматриваю то, каким образом программа ожидает ввод. Возможно она действительно ждет ввода пользователя в ***get_value*** (), а может быть программа ассоциирует ***Ival_box*** с событием и готовится отреагировать на обратный вызов (callback) или запускает фоновый поток для ***Ival_box*** и затем опрашивает его состояние. Эти решения имеют первостепенное значение при проектировании систем пользовательского интерфейса. Однако хоть сколько-нибудь детальное их обсуждение отвлекло бы нас от демонстрации основных методов программирования и средств языка. Описанные здесь методы проектирования, вместе с поддерживающими

их средствами языка, не являются специфичными для пользовательских интерфейсов. Они применимы к гораздо более широкому диапазону задач.

Различные виды ***Ival_box*** определены как классы, производные от ***Ival_box***. Например:

```
class Ival_slider : public Ival_box {
        // графический внешний вид и т. д.
public:
        Ival_slider (int, int);

        int get_value ();
        void prompt ();
};
```

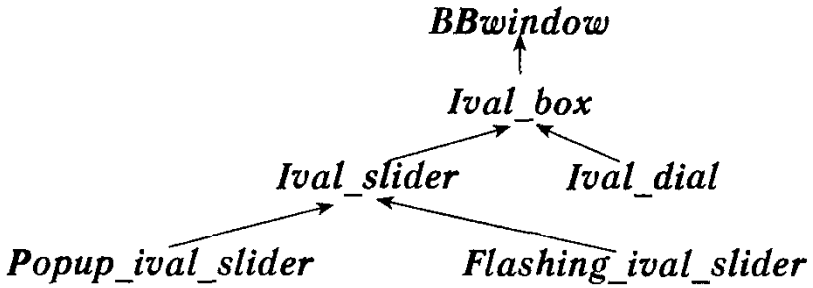
Члены данных ***Ival_box*** были объявлены защищенными для обеспечения доступа к ним из производных классов. Таким образом ***Ival_slider::get_value*** () может поместить значение в ***Ival_box::val***. Защищенные члены доступны для членов самого класса и для членов производных классов, но не для других функций (см. § 15.3).

Кроме ***Ival_slider*** мы можем определить другие варианты концепции ***Ival_box***. Это могут быть ***Ival_dial***, который позволяет выбирать значение нажатием кнопок, ***Flashing_ival_slider***, заставляющий курсор ввода мигать при вызове ***prompt*** (), и ***Popup_ival_slider***, который в ответ на ***prompt*** () «всплывает» на экране где-нибудь в заметном месте, не позволяя пользователю его проигнорировать.

Откуда мы возьмем графику? Большинство систем пользовательского интерфейса имеют класс, определяющий базовые свойства сущности на экране. Поэтому если мы пользуемся системой от компании «Крутые Баксы» («Big Bucks Inc.»), наши классы ***Ival_slider***, ***Ival_dial*** и т. д. должны стать разновидностью ***BBwindow***. Этого проще всего достичь, сделав наш ***Ival_box*** производным от ***BBWindow***. Тогда все наши классы будут наследовать свойства ***BBWindow***. Например, каждый ***Ival_box*** можно отобразить на экране (в соответствии с принятым графическим стилем), изменить его размер, переместить и т. д. в соответствии со стандартами, принятыми в ***BBwindow***. Наша иерархия классов будет выглядеть следующим образом:

```
class Ival_box : public BBwindow { /* ... */ };        // переписан для использования
                                                       // с BBWindow
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

или в графическом виде:



12.4.1.1. Критика

Изложенный подход к проектированию хорошо работает во многих случаях и соответствующая иерархия является хорошим решением для многих проблем. Однако имеются и некоторые неудобства, которые заставляют нас искать альтернативы.

Мы сделали ***BBwindow*** базовым классом ***Ival_box***. Это не совсем правильно. Использование ***BBwindow*** не является частью нашего фундаментального представления о ***Ival_box*** — это деталь реализации. Объявление ***Ival_box*** производным от ***BBwindow*** привело к выходу деталей реализации на первый план в процессе проектирования. Бывает, что такой подход приемлем. Например, использование среды от «Крутых Баксов» может быть неотъемлемой частью бизнес-процесса нашей организации. Но что если мы захотим реализовать ***Ival_box*** для систем от «Имперских Бананов» («Imperial Bananas»), «Освобожденного программного обеспечения» («Liberated Software») или «Шустрого Компилятора» («Compiler Whizzes»)? Нам бы пришлось поддерживать четыре разные версии нашей программы:

```
class Ival_box : public BBwindow { /* ... */ };
class Ival_box : public CWwindow { /* ... */ };
class Ival_box : public IBwindow { /* ... */ };
class Ival_box : public LSwindow { /* ... */ };
```

Наличие нескольких версий может превратиться в настоящий кошмар.

Другая проблема состоит в том, что все производные классы совместно используют базовые данные, объявленные в ***Ival_box***. Эти данные тоже, конечно же, являются деталью реализации, вкравшейся в интерфейс ***Ival_box***. С практической точки зрения эти данные во многих случаях вредны. Например, ***Ival_slider*** не нуждается в отдельно хранимом значении. Его легко можно вычислить по положению ползунка при выполнении ***get_value*** (). Как правило, хранение двух взаимосвязанных, но различных наборов данных, провоцирует ошибки. Рано или поздно будет нарушена синхронизация. Кроме того, как показывает практика, новички склонны к объявлению в качестве защищенных избыточного количества данных, что приводит к проблемам при сопровождении. Данные лучше иметь закрытыми, чтобы при написании производных классов не возникала путаница. А еще лучше, когда данные находятся в производных классах, где они могут быть определены наилучшим образом в соответствии с требованиями и не могут усложнить жизнь другим производным классам, не связанным с ними. Почти во всех случаях защищенный интерфейс должен содержать только функции, типы и константы.

Выигрыш от использования ***BBwindow*** в качестве базового класса состоит в том, что средства, реализованные в нем, становятся доступными пользователям ***Ival_box***. К сожалению, это также означает, что изменения, сделанные в классе ***BBwindow***, могут вынудить пользователей перекомпилировать или даже переписать код для восстановления работоспособности после этих изменений. В частности, в большинстве реализаций C++ изменение размера базового класса требует перекомпиляции всех производных классов.

Наконец, нашей программе, возможно, придется работать в смешанном окружении, где сосуществуют различные системы пользовательского интерфейса. Это может произойти в результате того, что две системы каким-то образом совместно используют экран или потому, что наша программа должна обмениваться информацией с пользователями других систем. Жесткое встраивание какой-то системы пользовательского интерфейса как одного и единственного базового класса для нашего одного и единственного интерфейса ***Ival_box*** не достаточно гибко для решения таких задач.

12.4.2. Абстрактные классы

Давайте начнем сначала и построим новую иерархию классов, которая решит выявленные проблемы, присущие традиционному подходу:

[1] Система пользовательского интерфейса должна быть деталью реализации, скрытой от пользователей, которые не хотят о ней знать.

[2] Класс ***Ival_box*** не должен содержать данных.

[3] После изменений в системе пользовательского интерфейса не должно происходить перекомпиляции классов семейства ***Ival_box***.

[4] Несколько ***Ival_box*** для различных систем пользовательского интерфейса должны иметь возможность сосуществовать в нашей программе.

Для достижения этой цели существует несколько альтернативных подходов. Здесь я представлю один из них, который непосредственно «отображается» на C++.

Во-первых, я определю класс ***Ival_box*** как чистый интерфейс:

```
class Ival_box {
public:
        virtual int get_value () = 0;
        virtual void set_value (int i) = 0;
        virtual void reset_value (int i) = 0;
        virtual void prompt () = 0;
        virtual bool was_changed () const = 0;
        virtual ~Ival_box () { }
};
```

Это намного понятнее, чем исходное объявление ***Ival_box***. Данные исчезли вместе с упрощенными реализациями функций-членов. Также исчез конструктор, потому что нет данных, которые нужно инициализировать. Вместо этого я добавил виртуальный деструктор, чтобы гарантировать правильную очистку данных, которая будет определена в производных классах.

Определение ***Ival_slider*** может выглядеть следующим образом:

```
class Ival_slider : public Ival_box, protected BBwindow {
public:
        Ival_slider (int, int);
        ~Ival_slider (),

        int get_value ();
        void set_value (int i);
        // ...

protected:
        // функции, замедляющие виртуальные
        // функции BBwindow, например,
        // BBwindow::draw(), BBwindow::mouse1hit()
private:
        // данные, необходимые Ival_slider
};
```

Производный класс ***Ival_slider*** является наследником абстрактного класса (***Ival_box***), который требует, чтобы ***Ival_slider*** реализовал чисто виртуальные функ-

ции базового класса. ***Ival_slider*** также наследует от ***BBwindow***, который предоставляет необходимые средства для реализации этих функций. Так как ***Ival_box*** предоставляет интерфейс для производного класса, наследование сделано открытым. ***BBwindow*** является только средством реализации, поэтому его наследование объявлено защищенным (§ 15.3.2). Их этого следует, что программист, пользующийся ***Ival_slider***, не может непосредственно использовать средства, определенные в ***BBwindow***. Интерфейс ***Ival_slider*** состоит из интерфейса, унаследованного от ***Ival_box***, плюс то, что явно объявлено в ***Ival_slider***. Я использовал защищенное наследование вместо более ограничительного (и обычно более безопасного) закрытого, для того чтобы ***BBwindow*** был доступен для классов, производных от ***Ival_slider***.

Непосредственное наследование более чем от одного класса обычно называют *множественным наследованием* (§ 15.2). Обратите внимание, что в ***Ival_slider*** должны быть замещены функции и из ***Ival_box***, и из ***BBwindow***. Поэтому, прямо или косвенно, он должен быть производным от обоих классов. Как показано в § 12.4.1.1, возможно косвенное наследование ***Ival_slider*** от ***BBwindow*** путем объявления ***BBwindow*** базовым классом для ***Ival_box***, но это привело бы к нежелательным побочным эффектам. Аналогично, объявление «класса реализации» ***BBwindow*** членом ***Ival_box*** не является решением, потому что класс не может заместить виртуальные функции своих членов (§ 24.3.4). Представление окна (window) в виде члена ***BBwindow**** в ***Ival_box*** ведет к совершенно другому стилю проектированию (§ 12.7[14], § 25.7).

Интересно, что приведенное объявление ***Ival_slider*** позволяет написать код приложения абсолютно также, как и раньше. Все что мы сделали — по-новому, более логичным образом, структурировали детали реализации.

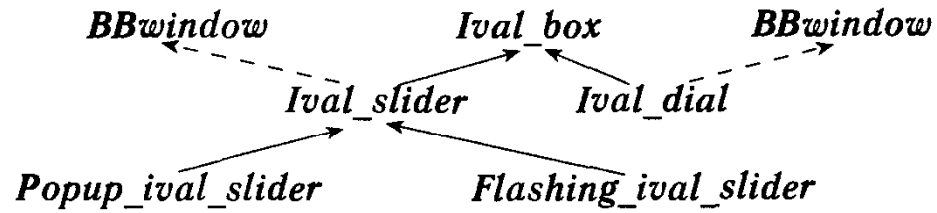
Многие классы требуют некоторой формы очистки до уничтожения объекта. Так как абстрактный класс ***Ival_box*** не может знать, требуется ли в производном классе такая очистка, он должен предположить, что требуется. Мы гарантируем необходимую очистку, объявив виртуальный деструктор ***Ival*::*box*::~*Ival_box*** () в базовом классе и замещая его в производных. Например:

```
void f(Ival_box* p)
{
        // ...
        delete p;
}
```

Оператор ***delete*** явным образом уничтожает объект, на который указывает ***p***. Нет способа точно определить, к какому классу принадлежит объект, на который указывает ***p***, но благодаря виртуальному деструктору ***Ival_box*** будет вызван нужный деструктор. Иерархию ***Ival_box*** можно теперь определить следующим образом:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class Ival_dial : public Ival_box, protected BBwindow { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

или графически (используя очевидные сокращения):



Я пользовался пунктирной линией для отображения защищенного наследования. Если речь идет об обычном пользователе, это становится просто деталью реализации.

12.4.3. Альтернативные реализации

Получившийся результат выигрывает в ясности, легкости сопровождения и не проигрывает в эффективности по сравнению с традиционным подходом. Однако этот вариант по-прежнему не решает проблему управления версиями:

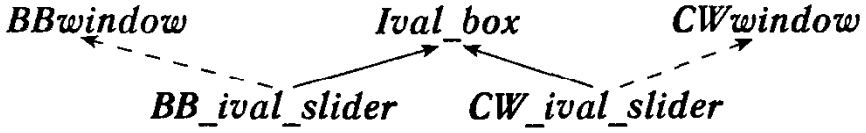
```
class Ival_box { /* ... */ };                                    // общий
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ };   // для BB
class Ival_slider  public Ival_box, protected CWwindow { /* ... */ };    // для CW
// ...
```

Кроме того, ***Ival_slider*** для ***BBwindow*** и ***CWwindow*** не могут сосуществовать, даже если обе системы пользовательского интерфейса в принципе совместимы.

Очевидным решением является определение нескольких классов ***Ival_slider*** с различными именами:

```
class Ival_box { /* ... */ };
class BB_Ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class CW_Ival_slider : public Ival_box, protected CWwindow { /* ... */ };
// ...
```

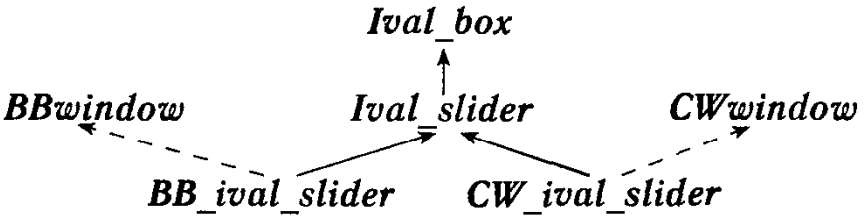
или в графическом виде:



Чтобы еще больше изолировать наши классы ***Ival_box***, ориентированные на приложения, от деталей реализации, мы можем произвести абстрактный класс ***Ival_slider*** от ***Ival_box***, а затем для каждой системы породить из него свой вариант ***Ival_slider***:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class BB_Ival_slider : public Ival_slider, protected BBwindow { /* ... */ };
class CW_Ival_slider : public Ival_slider, protected CWwindow { /* ... */ };
// ...
```

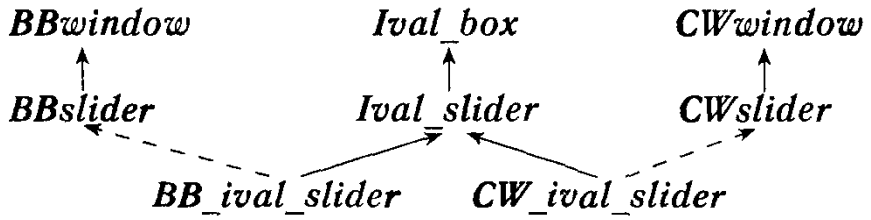
или в графическом виде:



Как правило, мы поступим еще лучше, если используем более специфические классы в иерархии реализации. Например, если бы в системе от «Крутых Баксов» существовал ползунок, мы определили бы наш ***Ival_slider*** как производный непосредственно от ***BBslider***:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
```

или в графическом виде:



Это улучшение становится особенно значительным, когда наша абстракция класса не слишком отличается от его представления в системе, что встречается довольно часто. В таком случае программирование сводится к отображению одной концепции в другую; вывод производных классов из общих базовых, таких как ***BBwindow***, происходит редко.

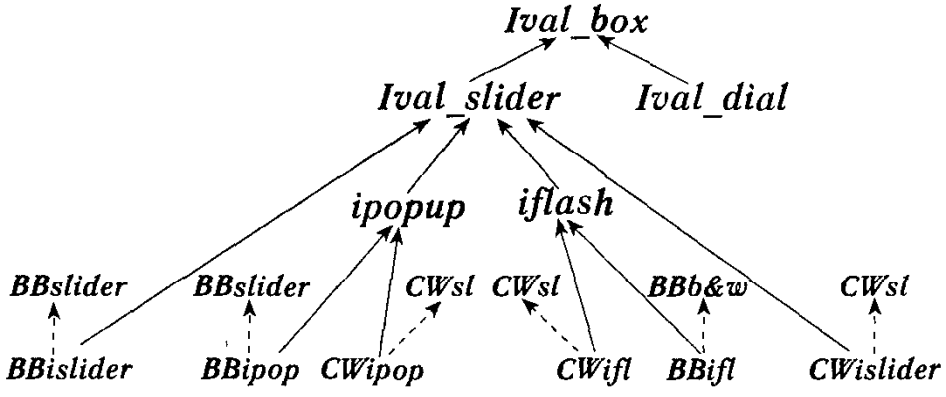
Полная иерархия будет состоять из нашей исходной, ориентированной на приложения, иерархии интерфейсов, выраженных в форме производных классов:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial . public Ival_box { /* ... */ };
class Flashing_ival_slider · public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ },
```

за которой следуют реализации этой иерархии для различных графических систем пользовательского интерфейса, выраженные в форме производных классов:

```
class BB_ival_slider · public Ival_slider, protected BBslider { /* ... */ },
class BB_flashing_ival_slider : public Flashing_ival_slider,
      protected BBwindow_with_bells_and_whistles { /* ... */ };
class BB_popup_ival_slider : public Popup_ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ },
// ...
```

Используя очевидные сокращения, эту иерархию можно графически представить в следующем виде:



Исходная иерархия классов ***Ival_box*** не изменилась — просто она теперь окружена классами реализации.

12.4.3.1. Критика

Проектирование на основе абстрактных классов почти настолько же просто, как эквивалентный метод с использованием общего базового класса, определяющего систему пользовательского интерфейса. В последнем случае класс окон является корнем дерева. При первом подходе исходная иерархия прикладных классов остается неизменной и выступает в качестве корневой для классов реализации. С точки зрения приложения оба подхода к проектированию эквивалентны в том смысле, что в обоих случаях почти весь код работает одинаково. В обоих случаях вы практически всегда можете рассматривать семейство классов ***Ival_box*** независимо от деталей реализации, относящихся к оконному интерфейсу. Например, нам не придется переписывать ***interact*** () из § 12.4.1 при переходе от одной иерархии классов к другой.

При любом подходе реализация каждого класса из семейства ***Ival_box*** должна быть переписана, когда открытый интерфейс системы пользовательского интерфейса изменяется. Однако при проектировании на основе абстрактных классов почти весь пользовательский код защищен от изменений в реализации иерархии и не нуждается в перекомпиляции после таких изменений. Это особенно важно, когда поставщик иерархии реализации выпускает новый «почти совместимый» релиз. Кроме того, пользователи иерархии абстрактных классов меньше рискуют попасть в зависимость от способа реализации внешней системы, чем пользователи классической иерархии. Пользователи абстрактного класса ***Ival_box*** не могут случайно воспользоваться механизмами реализации, потому что им доступны только средства, явно указанные в иерархии ***Ival_box***, и ничто не наследуется неявно из зависящего от реализации базового класса.

12.4.4. Локализация создания объектов

Большую часть приложения можно написать с использованием интерфейса ***Ival_box***. Если производные интерфейсы реализуют больше средств, чем просто ***Ival_box***, то большую часть приложения можно написать с использованием интерфейсов ***Ival_box***, ***Ival_slider*** и т. д. Однако создание объектов должно осуществляться с использованием специфических для реализации имен, таких как ***CW_ival_dial*** и ***BB_flashing_ival_slider***. Мы хотели бы минимизировать количество мест, где встречаются такие имена. Создание объектов трудно локализовать, если это не делается систематически.

Как всегда решение основывается на использовании косвенного обращения. Это можно сделать несколькими способами. Одним из простых решений является создание абстрактного класса с набором операций конструирования:

```
class Ival_maker {
public
        virtual Ival_dial* dial (int, int) =0,          // создать объект с кнопочным вводом
        virtual Popup_ival_slider*
                popup_slider (int, int) =0;             // создать всплывающий ползунок
        // ...
};
```

Для каждого интерфейса из семейства классов ***Ival_box***, о котором пользователь должен знать, ***Ival_maker*** предоставляет функцию создания объекта. Такой класс иногда называют *фабрикой*, а его функции иногда называют *виртуальными конструкторами* (что может несколько запутать) (§ 15.6.2).

Теперь мы представим каждую систему пользовательского интерфейса классом, производным от ***Ival_maker***:

```
class BB_maker : public Ival_maker {          // создать BB-версию
public:
        Ival_dial* dial (int, int);
        Popup_ival_slider* popup_slider (int, int);
        // ...
};
class LS_maker : public Ival_maker {          // создать LS-версию
public:
        Ival_dial* dial (int, int);
        Popup_ival_slider* popup_slider (int, int);
        // ...
};
```

Каждая функция создает объект с требуемым интерфейсом и типом реализации. Например:

```
Ival_dial* BB_maker::dial (int a, int b)
{
        return new BB_ival_dial (a, b);
}
Ival_dial* LS_maker::dial (int a, int b)
{
        return new LS_ival_dial (a, b);
}
```

Имея указатель на ***Ival_maker***, пользователь теперь может создавать объекты, не зная, какая система пользовательского интерфейса в действительности задействована. Например:

```
void user (Ival_maker* pim)
{
        Ival_box* pb = pim->dial (0, 99);     // создать подходящий объект
                                              // с кнопочным вводом
}
BB_maker BB_impl;          // для пользователей BB
LS_maker LS_impl;          // для пользователей LS

void driver ()
{
        user (&BB_impl);   // использует BB
        user (&LS_impl);   // использует LS
}
```

12.5. Иерархии классов и абстрактные классы

Абстрактный класс является интерфейсом. Иерархия классов — средством последовательного построения классов. Естественно, каждый класс предоставляет своим пользователям интерфейс, и некоторые абстрактные классы обеспечивают полезную функциональность, но тем не менее «интерфейс» и «строительные блоки» — вот главные роли абстрактных классов и иерархии классов.

Классическая иерархия — это иерархия, в которой отдельные классы предоставляют пользователям полезные функции и одновременно являются строительными блоками для реализации более мощных или специализированных классов. Такие иерархии идеальны для поддержки программирования методом последовательных усовершенствований. Они предоставляют максимум поддержки для реализации новых классов, в то время как новые классы в значительной степени используют существующую иерархию.

Классические иерархии имеют тенденцию смешивать вопросы реализации с интерфейсами для пользователей. В этих случаях на помощь приходят абстрактные классы. Иерархии абстрактных классов предоставляют ясный и мощный способ выражения концепций, не загроможденных вопросами реализации, и при этом не приводят к значительным накладным расходам. И последнее: вызов виртуальной функции довольно дешев и не зависит от того, барьер абстракции какого типа он пересекает. Вызов члена абстрактного класса стоит не дороже вызова любой другой виртуальной функции.

Логическим завершением этих размышлений является система, предоставляемая пользователю в виде иерархии абстрактных классов и реализуемая при помощи классической иерархии.

12.6. Советы

[1] Не используйте поля типа; § 12.2.5.
[2] Пользуйтесь указателями и ссылками во избежание «срезки» при копировании; § 12.3.
[3] Пользуйтесь абстрактными классами, чтобы сфокусировать проект на предоставлении ясных интерфейсов; § 12.3.
[4] Пользуйтесь абстрактными классами для минимизации интерфейсов; § 12.4.2.
[5] Пользуйтесь абстрактными классами для отделения деталей реализации от интерфейсов; § 12.4.2.
[6] Пользуйтесь виртуальными функциями, чтобы новые реализации могли добавляться без изменения пользовательского кода; § 12.4.1.
[7] Пользуйтесь абстрактными классами для сведения к минимуму перекомпиляции пользовательского кода; § 12.4.2.
[8] Пользуйтесь абстрактными классами, чтобы добиться сосуществования альтернативных реализаций; § 12.4.3.
[9] В классе с виртуальной функцией стоит иметь виртуальный деструктор; § 12.4.2.
[10] Абстрактный класс обычно не нуждается в конструкторе; § 12.4.2.
[11] Разделяйте представления различных концепций; § 12.4.1.1.

12.7. Упражнения

1. (*1) Определено

```
class base {
public:
        virtual void iam () { cout << "base\n"; }
};
```

Создайте из ***base*** два производных класса, и для каждого определите функцию ***iam*** (), выводящую имя класса. Создайте объекты этих классов и вызовите с ними ***iam*** (). Присвойте указатели на объекты производных классов указателям на ***base**** и вызовите ***iam*** () через эти указатели.

2. (*3.5) Реализуйте простую графическую систему с использованием любых графических средств, доступных в вашей системе (если в вашей системе нет хорошей графики или у вас нет никакого опыта работы с ней, вы можете создать «ASCII-реализацию», где пиксель — это знакоместо, и вы рисуете, помещая подходящий символ, например *, в определенную позицию). ***Window*** (*n*, *m*) (окно) создает на экране область размером *n* на *m*. Обращение к точкам на экране происходит в декартовых координатах (x, y). Окно ***Window w*** имеет текущую позицию ***w.current*** (). Вначале ***current*** равна ***Point*** (*0*, *0*). Текущую позицию можно задать с помощью ***w.current*** (*p*), где *p* имеет тип ***Point*** (позиция). Объект типа ***Point*** задается парой координат ***Point*** (*x*, *y*). ***Line*** задается парой объектов ***Point***: ***Line*** (***w.current*** (), *p2*); класс ***Shape*** (фигура) является общим интерфейсом для ***Dot*** (точка), ***Line*** (линия), ***Rectangle*** (прямоугольник), ***Circle*** (круг) и т. д. ***Point*** не является ***Shape***. Для представления ***Point p*** на экране можно пользоваться ***Dot***, ***Dot*** (*p*). ***Shape*** невидим до вызова функции ***draw*** () (нарисовать). Например: ***w.draw*** (***Circle*** (***w.current*** (), *10*)). Каждый ***Shape*** имеет девять специальных точек: ***e*** (восток), ***w*** (запад), ***n*** (север), ***s*** (юг), ***ne*** (северо-восток), ***nw*** (северо-запад), ***se*** (юго-восток), ***sw*** (юго-запад) и ***c*** (центр). Например, ***Line*** (***x.c*** (), ***y.nw*** ()) создает линию от центра ***x*** до верхнего левого угла ***y***. После вызова ***draw*** () текущая позиция ***Shape*** равна ***se***. ***Rectangle*** задается координатами левого нижнего и правого верхнего углов: ***Rectangle*** (***w.current*** (), ***Point*** (*10*, *10*)). В качестве теста выведите простой детский рисунок домика с крышей, двумя окнами и дверью.
3. (*2) Рисование ***Shape*** на экране производится сегментами линий. Реализуйте операции, которые меняют внешний вид этих сегментов: ***s.thickness*** (*n*) устанавливает толщину линии в *0*, *1*, *2* или *3*. Здесь *2* — толщина линий по умолчанию, а *0* означает невидимую линию. Кроме того, пусть сегменты могут быть: ***solid*** (сплошными), ***dashed*** (пунктирными) или ***dotted*** (из точек). Тип сегмента задается функцией ***Shape***::***outline*** ().
4. (*2.5) Реализуйте функцию ***Line***::***arrowhead*** (), которая добавляет стрелки в концы линии. У линии два конца и стрелки могут указывать в два направления относительно линии, поэтому аргументы ***arrowhead*** () должны уметь задавать по крайней мере четыре альтернативы.
5. (*3.5) Гарантируйте, что точки и сегменты линий, оказывающиеся за пределами окна, не появятся на экране. Это часто называют «отсечкой». Для этого (и только этого) упражнения не пользуйтесь средствами графической системы.
6. (*2.5) Добавьте к графической системе тип ***Text***. ***Text*** является прямоугольным ***Shape***, в котором выводятся символы. По умолчанию каждый символ занимает одну позицию вдоль каждой координатной оси.
7. (*2) Определите функцию, которая рисует линию, соединяющую две фигуры (***Shape***), вычисляя две «ближайшие точки» и соединяя их.
8. (*3) Добавьте к вашей простой графической системе цвет. Цветными могут быть: фон, внутренность замкнутой фигуры и контур фигуры.

9. (*2) Рассмотрим пример:

```
class Char_vec {
        int sz;
        char element[1];
public:
        static Char_vec* new_char_vec (int s);
        char& operator[] (int i) { return element[i]; }
        // ...
};
```

Определите ***new_char_vec*** (), выделяющую непрерывную память для объекта ***Char_vec*** таким образом, чтобы доступ к элементам мог осуществляться по индексу через ***element***, как показано выше. При каких условиях это вызовет серьезные проблемы?

10. (*2.5) При наличии классов ***Circle*** (круг), ***Square*** (квадрат) и ***Triangle*** (треугольник), производных от ***Shape*** (фигура), определите функцию ***intersect*** () (пересечение), которая принимает две фигуры ***Shape**** в качестве аргументов и вызывает подходящие функции для определения того, пересекаются ли эти фигуры. Для решения этой задачи вам придется добавить подходящие (виртуальные) функции к классам. Не пишите код, проверяющий пересечение; просто убедитесь, что вызываются правильные функции. Такой подход часто называют *двойной диспетчеризацией* или *мульти-методом*.
11. (*5) Спроектируйте и реализуйте библиотеку для написания моделей, управляемых событиями. Подсказка: <***task.h***>. Однако это старая программа, и вы можете сделать лучше. Должен быть объявлен класс ***task*** (задача). Объект класса ***task*** должен иметь возможность сохранять свое состояние и восстанавливать его (вы можете определить функции ***task***::***save*** () и ***task***::***restore*** ()), так что он сможет работать как сопрограмма. Специфические задачи могут быть определены в виде объектов классов, производных от ***task***. Программа, которая должна быть выполнена задачей, может быть объявлена в виде виртуальной функции. Должна быть реализована возможность передачи параметров новой задаче в качестве аргументов ее конструктора (или конструкторов). Должен иметься планировщик, реализующий концепцию виртуального времени. Введите функцию ***task***::***delay*** (***long***), которая «потребляет» виртуальное время. Будет ли планировщик частью класса ***task*** или реализован отдельно, станет одним из главных проектных решений. Задачам нужно обмениваться сообщениями. Разработайте для этого класс ***queue*** (очередь). Придумайте способ, который бы позволил задаче ожидать ввод из нескольких очередей. Научитесь обрабатывать ошибки времени выполнения некоторым стандартным образом. Как бы вы отлаживали программу, использующую такую библиотеку?
12. (*2) Определите интерфейсы для классов ***Warrior*** (воин), ***Monster*** (монстр) и ***Object*** (предмет, который вы можете поднять, бросить, использовать и т. п.) для ролевой игры.
13. (*1.5) Почему в § 12.7[2] есть и класс ***Dot***, и класс ***Point***? При каких условиях будет уместно дополнить классы ***Shape*** конкретными версиями ключевых классов типа ***Line***?

14. (*3) Определите в общих чертах различные стратегии реализации примера с ***Ival_box*** (§ 12.4), основанной на идее, что каждый класс, который виден из приложения, является интерфейсом, содержащим единственный указатель на реализацию. Таким образом, каждый «интерфейсный класс» будет служить в качестве дескриптора «класса реализации», и будут существовать иерархии интерфейсов и реализации. Напишите фрагменты кода, достаточно подробные для иллюстрации возможных проблем с преобразованиями типов. Принимайте во внимание простоту использования, легкость программирования, возможность повторного использования реализаций и интерфейсов после добавления новой концепции в иерархию, простоту внесения изменений в интерфейсы и реализации, а также потребность в перекомпиляции после изменений в реализации.

13

Шаблоны

Здесь — Ваша цитата.
— Б. Страуструп

Шаблоны — шаблон строк — инстанцирование — параметры шаблоны — проверка типа — шаблоны функций — выведение типов аргументов шаблона — задание аргументов шаблона — перегрузка шаблонов функций — выбор алгоритма через аргументы шаблона — аргументы шаблона по умолчанию — специализация — наследование и шаблоны — членов-шаблоны — преобразования — организация исходного кода — советы — упражнения.

13.1. Введение

Независимые концепции должны быть представлены независимо и объединяться только при необходимости. При нарушении этого принципа вы либо объединяете несвязанные концепции, либо создаете ненужные зависимости. В любом случае вы получаете менее гибкий набор компонент для построения системы. Шаблоны обеспечивают простой способ введения разного рода общих концепций и простые методы их совместного использования. Получающиеся в результате классы и функции сопоставимы по времени выполнения и требованиям к памяти с написанным вручную специализированным кодом.

Шаблоны обеспечивают непосредственную поддержку обобщенного программирования (§ 2.7), то есть программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает использование типа в качестве параметра при определении класса или функции. Шаблон зависит только от тех свойств параметра-типа, которые он явно использует, и не требует, чтобы различные типы, используемые в качестве аргументов, были связаны каким-либо другим образом. В частности, типы аргументов шаблона не должны принадлежать к одной иерархии наследования.

В этой главе основной акцент при описании шаблонов сделан на методах, необходимых при проектировании, реализации и использовании стандартной библиотеки. Стандартная библиотека должна обеспечивать большую степень общности, гибкости и эффективности, чем большинство других программ. Следовательно, методы, которые можно использовать при проектировании и реализации стандартной библиотеки, являются действенными и эффективными при разработке решений широкого спектра проблем. Эти методы позволяют программисту спрятать сложную реализацию за простыми интерфейсами и открывать ее пользователю только при возникновении у него такой потребности. Например, ***sort*** (*v*) может служить интерфейсом для

множества разных алгоритмов сортировки элементов различных типов, хранящихся в самых разнообразных контейнерах. Функция сортировки, наиболее подходящая для конкретного *v*, будет выбрана автоматически.

Каждая существенная абстракция в стандартной библиотеке представлена в виде шаблона (например, ***string***, ***ostream***, ***complex***, ***list*** и ***map***), также как и все ключевые операции (например, сравнение строк, оператор вывода <<, комплексное сложение, доступ к следующему элементу списка и сортировка). Главы этой книги, посвященные описанию библиотеки (часть III), являются богатым источником примеров шаблонов и методов программирования, связанных с их использованием. Поэтому в этой главе уделяется большее внимание небольшим примерам, иллюстрирующим технические аспекты шаблонов, и фундаментальным методам их использования:

§ 13.2: Базовые механизмы определения и использования шаблонов классов.
§ 13.3: Шаблоны функций, перегрузка функций и выведение типов.
§ 13.4: Параметры шаблона, используемые для управления поведением обобщенных алгоритмов.
§ 13.5: Множественные определения, обеспечивающие альтернативные реализации шаблона.
§ 13.6: Наследование и шаблоны (полиморфизм времени выполнения и времени компиляции).
§ 13.7: Организация исходного кода.

Знакомство с шаблонами состоялось в § 2.7.1 и § 3.8. Подробные правила разрешения имен шаблонов, синтаксиса шаблонов и т. д. можно найти в § B.13.

13.2. Простой шаблон строк

Рассмотрим строку символов. Строка является классом, который хранит символы и предоставляет операции, такие как доступ по индексу, конкатенацию и сравнение, которые мы обычно ассоциируем с понятием «строка». Мы хотели бы реализовать такое поведение для многих наборов символов. В различных контекстах полезны строки символов со знаком, без знака, китайских иероглифов, греческих букв и т. д. Поэтому, мы хотим реализовать понятие «строка» с минимальной зависимостью от конкретного набора символов. Определение строки полагается на тот факт, что символы можно копировать, и это почти все. Таким образом, мы можем создать более общий строковый тип, если воспользуемся строкой символов из § 11.12, объявив тип символа параметром:

```
template<class C> class String {
        struct Srep;
        Srep *rep;
public:
        String ();
        String (const C*);
        String (const String&);

        C read (int i) const;
        // ...
};
```

Префикс ***template<class C>*** указывает, что объявлен шаблон (***template***) и что аргумент ***C*** будет использован в объявлении как тип. После объявления ***C*** используется точно также, как имена других типов. Область видимости ***C*** простирается до конца

объявления, начинающегося с *template<class C>*. Обратите внимание, что *template<class C>* означает, что *C* — это имя *типа*, а не обязательно имя *класса*.

Имя шаблона класса, за которым следует тип, помещенный в угловые скобки < >, является именем класса (определяемого шаблоном) и его можно использовать точно также, как имена других классов. Например:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar {
        // японские символы
};

String<Jchar> js;
```

За исключением специального синтаксиса своего имени *String<char>* работает точно также, как если бы он был определен как *String* из § 11.12. То, что мы сделали *String* шаблоном, позволит нам реализовать средства, которые мы ввели для строк *char*, для строк с любыми символами. Так, если мы пользуемся шаблонами стандартной библиотеки *map* и *String*, пример с подсчетом слов из § 11.8 примет следующий вид:

```
int main ()          // подсчет количества вхождений каждого слова
{
        String<char> buf;
        map<String<char>, int> m;
        while (cin>>buf) m[buf]++;
        // вывод результата
}
```

Вариант для наших японских символов *Jchar* будет выглядеть так:

```
int main ()          // подсчет количества вхождений каждого слова
{
        String<Jchar> buf;
        map<String<Jchar>, int> m;
        while (cin>>buf) m[buf]++;
        // вывод результата
}
```

Стандартная библиотека предоставляет шаблон класса *basic_string*, который подобен *String* (§ 11.12, § 20.3). В стандартной библиотеке *string* определен как синоним *basic_string<char>*:

```
typedef basic_string<char> string;
```

что позволит нам написать программу подсчета слов следующим образом:

```
int main ()          // подсчет количества вхождений каждого слова
{
        string buf;
        map<string, int> m;
        while (cin>>buf) m[buf]++;
        // вывод результата
}
```

Как правило, *typedef* является полезным инструментом сокращения длинных имен классов, сгенерированных из шаблонов. Довольно часто мы предпочитаем не знать о том, как определен тип, и *typedef* позволяет нам скрыть тот факт, что тип генерируется из шаблона.

13.2.1. Определение шаблона

Класс, генерируемый из шаблона класса, является совершенно нормальным классом. Поэтому использование шаблона не подразумевает каких-либо дополнительных механизмов времени выполнения сверх тех, что использовались бы для эквивалентного написанного «вручную» класса. С другой стороны, использование шаблонов не подразумевает обязательного уменьшения объема сгенерированного кода.

Как правило, неплохой идеей является протестировать конкретный класс, например ***String***, до преобразования его в шаблон (***String<C>***). Поступая таким образом, мы решаем многие проблемы проектирования и обнаруживаем большую часть ошибок кода в контексте конкретного примера. Такой способ отладки знаком всем программистам, и большинству людей проще иметь дело с конкретными примерами, чем с абстрактными концепциями. Позднее, мы сможем решить многие проблемы, возникающие из-за обобщения, не отвлекаясь на обычные ошибки. Аналогично, при попытках разобраться в шаблоне, бывает полезно представить его поведение для конкретного типа аргумента, например ***char***, до того как пытаться осознать шаблон во всей его общности.

Члены шаблона класса объявляются и определяются точно также, как и для обычного класса. Член шаблона не обязательно определять внутри самого шаблона. В этом случае где-то должно существовать его определение, точно также, как и в случае с классом, не являющимся шаблоном (§ В.13.7). Члены шаблона класса в свою очередь являются шаблонами, параметризованными при помощи аргументов шаблона. Например:

```
template<class C> struct String<C>::Srep {
        C* s;       // указатель на элементы
        int sz;     // количество элементов
        int n;      // подсчет количества обращений
        //...
};

template<class C> C String<C>::read (int i) const { return rep->s[i]; }

template <class C> String<C>::String ()
{
        rep = new Srep (0, C ());
}
```

Параметры шаблонов, такие как ***C***, являются скорее параметрами, чем именами типов, определенными внешним образом по отношению к шаблону. Однако это никоим образом не влияет на то, как мы пишем код шаблона с их использованием. Внутри области видимости ***String<C>*** квалификация с ***<C>*** избыточна для имени самого шаблона, поэтому ***String<C>::String*** — имя конструктора. Если хотите, можете писать явно:

```
template <class C> String<C>::String<C> ()
```

```
{
    rep = new Srep(0, C());
}
```

Аналогично тому, как в программе может существовать только одна функция, определяющая функцию-член класса, может существовать только один шаблон функции, определяющий функцию-член шаблона класса. Однако перегрузка возможна только для функций (§ 13.3.2), в то время как специализация (§ 13.5) позволяет нам обеспечить альтернативные реализации шаблона.

Перегрузка имени шаблона класса невозможна, поэтому никакая другая сущность не может быть объявлена в той же области видимости и с тем же именем, что и шаблон класса (см. также § 13.5). Например:

```
template<class T> class String { /* ... */ };
class String { /* ... */ };          // ошибка: двойное определение
```

Тип, используемый в качестве аргумента шаблона, должен обеспечивать интерфейс, ожидаемый шаблоном. Например, тип, используемый в качестве аргумента шаблона **String**, должен обеспечивать обычные операции копирования (§ 10.4.4.1, § 20.2.1). Обратите внимание: не требуется, чтобы различные возможные аргументы для одного и того же параметра шаблона были связаны наследованием.

13.2.2. Инстанцирование

Процесс генерации объявления класса по шаблону класса и аргументу шаблона часто называется *инстанцированием шаблона* (template instantiation) (§ B.13.7). Аналогично, функция генерируется («инстанцируется») из шаблона функции и аргумента шаблона. Версия шаблона для конкретного аргумента шаблона называется *специализацией*.

Генерация версий функций шаблона для набора аргументов шаблона (§ B.13.7) является задачей компилятора, а *не* программиста. Например:

```
String<char> cs;

void f()
{
    String<Jchar> js;

    cs = "Какой код сгенерировать, решает компилятор";
}
```

В этом случае компилятор генерирует объявления для **String<char>** и **String<Jchar>**, для их соответствующих типов **Srep**, для их деструкторов и конструкторов по умолчанию и для присваивания **String<char>::operator=(char*)**. Другие функции-члены не используются и не должны генерироваться. Сгенерированные классы являются совершенно обычными классами, которые подчиняются всем стандартным правилам для классов. Аналогично, сгенерированные функции являются обычными функциями, которые подчиняются всем стандартным правилам для функций.

Очевидно, что шаблоны обеспечивают эффективный способ генерации кода из сравнительно коротких определений. Поэтому требуется некоторая осмотрительность во избежание заполнения памяти почти идентичными определениями функций (§ 13.5).

13.2.3. Параметры шаблонов

Параметрами шаблонов могут быть: параметры-типы, параметры обычных типов, такие как ***int***, и параметры-шаблоны (§ В.13.3). Естественно, у шаблона может быть несколько параметров. Например:

```
template<class T, T def_val> class Cont { /* ... */ };
```

Как видно, параметр шаблона может быть использован для определения последующих параметров шаблона.

Целые аргументы используются обычно для задания размеров и границ. Например:

```
template<class T, int max> class Buffer {
        T v[max],
public:
        Buffer () { }
        // ...
};

Buffer<char, 128> cbuf;
Buffer<int, 5000> ibuf;
Buffer<Record, 8> rbuf;
```

Простые контейнеры с ограниченными возможностями, такие как ***Buffer***, могут иметь большое значение, когда компактность кода и эффективность во время выполнения имеют первостепенное значение (что препятствует использованию более общих классов ***string*** или ***vector***). Передача размера в качестве аргумента шаблона ***Buffer*** позволяет его разработчику избежать использования лишней памяти. Другим примером может служить тип ***Range*** из § 25.6.1.

Аргумент шаблона может быть константным выражением (§ В.5), адресом объекта или функции с внешней компоновкой (§ 9.2) или неперегруженным указателем на член (§ 15.5). Указатель, используемый в качестве аргумента шаблона, должен иметь форму ***&of***, где ***of*** является именем объекта или функции, либо в форме ***f***, где ***f*** является именем функции. Указатель на член должен быть в форме ***&X::of***, где ***of*** является именем члена. В частности, строковый литерал *не* допустим в качестве аргумента шаблона.

Целый аргумент шаблона должен быть константой:

```
void f(int i)
{
        Buffer<int, i> bx;          // ошибка: требуется константное выражение
}
```

И наоборот, параметр шаблона, не являющийся типом, является константой в теле шаблона, поэтому попытка изменения значения этого параметра является ошибкой.

13.2.4. Эквивалентность типов

При наличии шаблона, мы можем генерировать типы путем задания аргументов шаблона. Например:

```
String<char> s1,
String<unsigned char> s2;
String<int> s3;
```

```
typedef unsigned char Uchar;
String<Uchar> s4;
String<char> s5;

Buffer<String<char>, 10> b1;
Buffer<char, 10> b2;
Buffer<char, 20-10> b3;
```

При использовании одного и того же набора аргументов шаблона мы всегда получаем один и тот же сгенерированный тип. Однако что означает «один и тот же» в этом контексте? Как всегда, ***typedef*** не создает новые типы, поэтому ***String<Uchar>*** является тем же типом, что и ***String<unsigned char>***. И наоборот, ***char*** и ***unsigned char*** являются различными типами (§ 4.3), поэтому ***String<char>*** и ***String<unsigned char>*** являются различными типами. Компилятор умеет вычислять константные выражения (§ В.5), поэтому тип ***Buffer<char, 20-10>*** тот же, что и ***Buffer<char, 10>***.

13.2.5. Проверка типов

Шаблон определяется, а затем используется в сочетании с набором его аргументов. При определении шаблона осуществляется проверка отсутствия синтаксических и, возможно, других ошибок, которые можно обнаружить вне зависимости от конкретного набора аргументов шаблона. Например:

```
template<class T> class List {
        struct Link {
                Link* pre;
                Link* suc,
                T val;
                Link (Link* p, Link* s, const T& v) : pre (p), suc (s), val (v) { }
                        // синтаксическая ошибка: отсутствует точка с запятой
        }
        Link* head;
public.
        List () head (7) { } // ошибка: инициализация указателя целым
        List (const T& t) : head (new Link (0, o, t)) { }      // ошибка: неопределенный
                                                               // идентификатор o
        //...
        void print_all () {
                for (Link* p = head; p; p = p->suc) cout << p->val << '\n'; }
};
```

Компилятор может обнаружить простые семантические ошибки в точке определения или позднее в точке использования. Пользователи обычно предпочитают более раннее обнаружение, но не все «простые» ошибки легко можно обнаружить. В примере выше я сделал три «ошибки». Независимо от того, какой у шаблона параметр, указатель ***Link**** не может быть инициализирован целым значением ***7***. Аналогично, идентификатор ***o*** (конечно, это опечатка — должен быть ноль) не может быть аргументом конструктора ***List<T>::Link***, потому что в области видимости нет такого имени.

Имя, использующееся в определении шаблона, должно либо находиться в области видимости, либо некоторым, достаточно очевидным образом, зависеть от параметра шаблона (§ В.13.8.1). Наиболее часто встречаемой и очевидной зависимостью от па-

раметра ***T*** является объявление члена типа ***T*** или функции-члена с аргументом типа ***T***. В качестве более тонкого примера можно привести ***List<T>::print_all*** (), ***cout<<p->val*** (см. выше).

Ошибки, имеющие отношение к использованию параметров шаблона, нельзя обнаружить до момента инстанцирования шаблона. Например:

```
class Rec { /* ... */ };

void f(List<int>& li, List<Rec>& lr)
{
        li.print_all();
        lr.print_all();
}
```

Проверка ***li.print_all*** () осуществляется успешно, но ***lr.print_all*** приводит к сообщению об ошибке использования типов, потому что оператор вывода << не определен для ***Rec***. Самое раннее, когда могут быть обнаружены ошибки, имеющие отношение к использованию параметров шаблона, — это точка первого задания конкретных аргументов шаблона. Это место обычно называют *первой точкой инстанцирования* или просто *точкой инстанцирования* (см. § В.13.7). Конкретные реализации могут откладывать проверки такого рода до этапа компоновки. Если бы в текущей единице компиляции находилось только объявление функции ***print_all*** (), а не ее определение, проверка соответствия типов могла бы быть отложена на более поздние этапы (см. § 13.7). Независимо от того, когда осуществляется проверка, используются одни и те же правила. Отметим еще раз, что пользователи предпочитают раннее обнаружение ошибок. Можно наложить ограничения на аргументы шаблона в терминах функций-членов (см. § 13.9[16]).

13.3. Шаблоны функций

Для большинства людей наиболее очевидным применением шаблонов является определение и использование классов-контейнеров, таких как ***basic_string*** (§ 20.3), ***vector*** (§ 16.3) и ***map*** (§ 17.4.1). Сразу же возникает потребность в шаблонах функций. В качестве простого примера можно привести сортировку массива:

```
template<class T> void sort(vector<T>&);          // объявление

void f(vector<int>& vi, vector<string>& vs)
{
        sort(vi);    // sort (vector<int>&);
        sort(vs);    // sort (vector<string>&);
}
```

При вызове функции-шаблона типы аргументов функции определяют, какая версия шаблона используется, то есть аргументы шаблона выводятся по аргументам функции (§ 13.3.1).

Естественно, функция-шаблон должна быть где-то определена (§ В.13.7):

```
template<class T> void sort(vector<T>& v)     // определение
        //сортировка Шелла (Knuth, том 3, стр. 84).
{
        const size_t n = v.size();
```

```
    for (int gap=n/2; 0<gap; gap /= 2)
        for (int i=gap, i<n; i++)
            for (int j=i-gap; 0<=j; j -= gap)
                if (v[j+gap] < v[j]) {      // поменять v[j] и v[j+gap]
                    T temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                }
}
```

Сравните это определение с ***sort*** () из (§ 7.7). Эта версия понятней и короче, потому что она может полагаться на более подробную информацию о типе сортируемых элементов. Скорее всего, она еще и быстрее, потому что в ней не используется указатель на функцию, которая непосредственно осуществляет сравнение. Это означает, что не требуется косвенного вызова функции, а простую операцию < (меньше) легко сделать встроенной.

Для дальнейшего упрощения можно воспользоваться шаблоном ***swap*** () (§ 18.6.8) из стандартной библиотеки для приведения обмена местами к его естественному виду:

```
if (v[j+gap] < v[j]) swap (v[j], v[j+gap]);
```

Это не приведет к дополнительным накладным расходам на этапе выполнения.

В примере для сравнения используется оператор <. Однако не у каждого типа есть этот оператор, что ограничивает возможности использования данной версии ***sort*** (). Но это ограничение легко преодолеть (см. § 13.4).

13.3.1. Аргументы шаблонов функций

Шаблоны функций имеют большое значение при написании обобщенных алгоритмов, применимых к широкому спектру типов контейнеров (§ 2.7.2, § 3.8 и глава 18). Существенным моментом здесь является возможность *выведения* (deduction) типа аргументов шаблона функции по типам аргументов при ее вызове.

Компилятор может вывести аргументы, как являющиеся типами, так и обычные, при условии, что список аргументов функции однозначно идентифицирует набор аргументов шаблона (§ В.13.4). Например:

```
template<class T, int i> T& lookup (Buffer<T, i>&b, const char* p);

class Record {
    const char v[12];
    // ...
};

Record& f (Buffer<Record, 128>& buf, const char* p)
{
    return lookup (buf, p);        // вызвать lookup (), где T — это Record, i — 128
}
```

Для этого примера компилятор вывел, что ***T — Record***, а ***i — 128***.

Обратите внимание, что параметры шаблона класса (в отличие от шаблона функции) никогда не выводятся. Причина заключается в том, что гибкость, обеспечиваемая наличием нескольких конструкторов класса, во многих случаях является непре-

одолимым препятствием на пути такого выведения, и еще в большем числе случаев выведение неоднозначно. Специализация предоставляет механизм неявного выбора из различных реализаций класса (§ 13.5). Если нам требуется создать объект выводимого типа, мы часто можем просто вызвать функцию; см. ***make_pair*** () в § 17.4.1.2.

Если аргумент шаблона не может быть выведен по списку аргументов шаблона функции (§ В.13.4), мы должны указать его явно. Это делается точно так же, как и при явном задании аргументов для шаблона класса. Например:

```
template<class T> class vector { /* ... */ };
template<class T> T* create ();      // создать T и вернуть указатель на него

void f()
{
    vector<int> v,                   // класс; аргумент шаблона int
    int* p = create<int> (),         // функция; аргумент шаблона int
}
```

Часто явная специализация используется для указания возвращаемого типа для шаблона функции:

```
template<class T, class U> T implicit_cast (U u) { return u; }

void g (int i)
{
    implicit_cast (i);               // ошибка: невозможно вывести T
    implicit_cast<double> (i);       // T — double, U — int
    implicit_cast<char, double> (i); // T — char, U — double
    implicit_cast<char*, int> (i);   // T — char*, U — int;
                                     // ошибка: невозможно преобразовать
                                     // int в char*
}
```

Как и в случае с аргументами функций (§ 7.5) по умолчанию, только последние аргументы в списке можно исключить из списка явных аргументов шаблона.

Явная специализация аргументов шаблона позволяет определить семейства функций преобразования и функций создания объектов (§ 13.3.2, § В.13.1, § В.13.5). Часто используется явный вариант неявных преобразований (§ В.6), такой как ***implicit_cast*** (). Синтаксис ***dynamic_cast***, ***static_cast*** и т. д. (§ 6.2.7, § 15.4.1) соответствует синтаксису явного задания шаблона функции. Однако операторы преобразования встроенных типов предоставляют операции, которые нельзя выразить другими средствами языка.

13.3.2. Перегрузка шаблонов функций

Можно объявить несколько шаблонов функции с одним и тем же именем и даже объявить комбинацию шаблонов и обычных функций с одинаковым именем. Когда вызывается перегруженная функция, требуется механизм разрешения перегрузки для нахождения требуемой функции или шаблона функции. Например:

```
template<class T> T sqrt (T),
template<class T> complex<T> sqrt (complex<T>);
double sqrt (double),

void f (complex<double> z)
```

```
{
    sqrt(2);        // sqrt<int>(int)
    sqrt(2.0);      // sqrt(double)
    sqrt(z);        // sqrt<double>(complex<double>)
}
```

Аналогично тому, как шаблон функции является обобщением понятия функции, правила разрешения при наличии шаблонов функций являются обобщением правил разрешения перегрузки функций. Для каждого шаблона осуществляется поиск специализации, наилучшим образом соответствующей списку аргументов. Затем применяются обычные правила разрешения перегрузки функций применительно к этим специализациям и обычным функциям:

[1] Ищется набор специализаций шаблонов функций (§ 13.2.2), которые примут участие в разрешении перегрузки. Это осуществляется с учетом всех шаблонов функции. Принимается решение, какие аргументы шаблона были бы использованы (если вообще были бы), если бы в текущей области видимости не было других шаблонов функций и обычных функций с тем же именем. В примере с вызовом *sqrt(z)* к рассмотрению будут приняты *sqrt<double>(complex<double>)* и *sqrt<complex<double>>(complex<double>)*.

[2] Если могут быть вызваны два шаблона функции, и один из них более специализирован (§ 13.5.1) чем другой, на следующих этапах только он и рассматривается. В примере с вызовом *sqrt(z)*, это означает, что предпочтение отдается *sqrt<double>(complex<double>)* по отношению к *sqrt<complex<double>>(complex<double>)*: любой вызов, который соответствует *sqrt<T>(complex<T>)*, также соответствует и *sqrt<T>(T)*.

[3] Разрешается перегрузка для этого набора функций, а также для любых обычных функций (для них перегрузка разрешается как для обычных функций (§ 7.4)). Если аргументы функции шаблона были определены путем выведения по фактическим аргументам шаблона (§ 13.3.1), к ним нельзя применять «продвижение», стандартные и определяемые пользователем преобразования. Для *sqrt(2)* *sqrt<int>(int)* является точным соответствием и поэтому ей отдается предпочтение по отношению к *sqrt(double)*.

[4] Если и обычная функция и специализация подходят одинаково хорошо, отдается предпочтение обычной функции. Следовательно, для *sqrt(2.0)* выбирается *sqrt(double)*, а не *sqrt<double>(double)*.

[5] Если ни одного соответствия не найдено, вызов считается ошибочным. Если процесс заканчивается с двумя или более одинаково хорошо подходящими вариантами, вызов является неоднозначным и это тоже ошибка.

Например:

```
template<class T> T max(T, T);

const int s = 7;

void k()
{
    max(1, 2);      // max<int>(1, 2)
    max('a', 'b');  // max<char>('a', 'b')
    max(2.7, 4.9);  // max<double>(2.7, 4.9)
```

```
        max(s, 7);          // max<int> (int (s), 7) — используется
                            // тривиальное преобразование
        max('a', 1);        // ошибка: неоднозначность
                            // (стандартные преобразования не применяются)
        max(2 7, 4);        // ошибка: неоднозначность
                            // (стандартные преобразования не применяются)
}
```

Мы могли бы разрешить две неоднозначности либо при помощи явного квалификатора:

```
void f()
{
        max<int>('a', 1),       // max (int ('a'), 1)
        max<double>(2.7, 4)     // max (2.7, double (4))
}
```

либо добавив подходящие объявления:

```
inline int max(int i, int j) { return max<int>(i, j); }
inline double max(int i, double d) { return max< double>(i, d); }
inline double max(double d, int i) { return max< double>(d, i), }
inline double max(double d1, double d2) { return max<double>(d1, d2); }

void g()
{
        max('a', 1);        // max (int ('a'), 1)
        max(2.7, 4)         // max (2.7, 4)
}
```

Для обыкновенных функций применяются обычные правила перегрузки (§ 7.4), а использование ***inline*** гарантирует, что не будет дополнительных затрат на вызов.

Определение функции ***max***() тривиально, поэтому мы могли бы написать его явно. Однако использование специализации шаблона является простым и общим способом определения таких разрешающих функций.

Правила разрешения перегрузки гарантируют, что функции-шаблоны корректно взаимодействуют с наследованием:

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

template<class T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
        f(pb);          //f<int> (pb)
        f(pd);          //f<int> (static_cast<B<int>*> (pd)); используется
                        // стандартное преобразование D<int>* в B<int>*
}
```

В этом примере функция-шаблон ***f***() принимает в качестве аргумента ***B<T>**** для любого типа ***T***. Аргумент имеет тип ***D<int>****, поэтому компилятор легко определяет, что ***T*** является ***int***. Разрешение вызова однозначно приводит к ***f(B<int>****).

Аргумент функции, не использующийся при выведении параметра шаблона, рассматривается точно также, как аргумент функции, не являющейся шаблоном.

В частности, к нему применяются обычные правила преобразования. Рассмотрим пример:

```
template<class C>
    int get_nth (C& p, int n);          // получение n-го элемента
```

Предположительно эта функция возвращает значение *n*-го элемента контейнера типа ***C***. Так как ***C*** должен быть выведен по фактическому аргументу в вызове ***get_nth*** (), преобразования типов для первого аргумента не разрешаются. Однако второй аргумент является совершенно обычным, поэтому рассматривается полный диапазон возможных преобразований. Например:

```
class Index {
public:
    operator int ();
    // ...
};

void f(vector<int>& v, short s, Index i)
{
    int i1 = get_nth<int> (v, 2);       // точное соответствие
    int i2 = get_nth<int> (v, s);       // стандартное преобразование short в int
    int i3 = get_nth<int> (v, i);       // преобразование Index в int,
                                        // определяемое пользователем
}
```

13.4. Использование аргументов шаблона для выбора алгоритма

Рассмотрим сортировку строк. В эту процедуру вовлечены три концепции: строка, тип элементов и критерий, используемый алгоритмом сортировки для сравнения элементов строк.

Мы не можем жестко задать критерий сортировки как часть контейнера, потому что контейнер (в общем случае) не может ожидать его наличия в своих элементах. Мы не можем задать критерий сортировки как часть типа элемента, потому что существует много различных способов сортировки элементов.

Следовательно, критерий сортировки не встроен ни в контейнер, ни в тип элементов. Вместо этого критерий указывается при выполнении конкретной операции. Например, если у меня есть строки символов, содержащие шведские имена, каким критерием я воспользуюсь для сравнения? Для сравнения шведских имен обычно используются две различных сортирующих последовательности (способы нумерации символов). Естественно, ни общий строковый тип, ни общий алгоритм сортировки не должны знать о соглашениях по сортировке шведских имен. Поэтому любое общее решение нуждается в том, чтобы алгоритм сортировки был выражен в общих терминах, которые могут быть определены не только для какого-то конкретного типа. В качестве примера давайте обобщим функцию стандартной библиотеки C ***strcmp*** () для строк любого типа ***T*** (§ 13.2):

```
template<class T, class C>
int compare (const String<T>& str1, const String<T>& str2)
```

```
{
    for (int i=0; i<str1.length () && i<str2.length (); i++)
        if (!C::eq (str1[i], str2[i])) return C::lt (str1[i], str2[i]) ? -1 : 1;
    return str1.length () - str2.length ();
}
```

Если кто-нибудь захочет, чтобы **compare** () игнорировала регистр букв, использовала национальные символы и т. п., можно определить подходящую **C::eq** () (равно) и **C::lt** () (меньше чем). Это позволит выразить любые алгоритмы (сравнение, сортировку и т. п.) в терминах «С-операций» и контейнеров. Например:

```
template<class T> class Cmp {       // обычное сравнение по умолчанию
public:
    static int eq (T a, T b) { return a==b; }
    static int lt (T a, T b) { return a<b; }
};

class Literate {                    // сравнение шведских имен по литературным правилам
public:
    static int eq (char a, char b) { return a==b; }
    static int lt (char, char);     // поиск в таблице на основе
                                    // «значения» символа (§ 13.9[14])
};
```

Теперь мы можем выбирать правила сравнения путем явного задания аргументов шаблона:

```
void f (String<char> swede1, String<char> swede2)
{
    compare< char, Cmp<char> > (swede1, swede2);
    compare< char, Literate> (swede1, swede2);
}
```

Передача операций сравнения в качестве параметра шаблона имеет два значительных преимущества при сопоставлении с альтернативными подходами, например, передачей указателей на функции. Можно передать несколько операций в качестве единственного аргумента без дополнительных затрат во время выполнения. Кроме того, операторы сравнения *eq* () и *lt* () легко реализовать в виде встроенных функций, в то время как встраивание вызова функции через указатель требует исключительного внимания со стороны компилятора.

Естественно, операции сравнения можно реализовать для типов, определяемых пользователем, также как и для встроенных типов. Это является существенным моментом для реализации обобщенных алгоритмов, работающих с типами с нетривиальными критериями сравнения (см. § 18.4).

Каждый генерируемый по шаблону класс получает копию каждого статического члена шаблона класса (см. § В.13.1).

13.4.1. Параметры шаблонов по умолчанию

Явное задание критерия сравнения при каждом вызове является довольно утомительным занятием. К счастью, можно легко задать умолчание таким образом, что критерий будет явно указываться только для необычных сравнений. Это можно реализовать при помощи механизма перегрузки:

```
template<class T, class C>
// сравнение с использованием класса C
int compare (const String<T>& str1, const String<T>& str2),

template<class T>
// сравнение с использованием Cmp<T>
int compare (const String<T>& str1, const String<T>& str2);
```

В качестве альтернативы мы можем указать «обычный» вариант в качестве аргумента шаблона по умолчанию:

```
template<class T, class C = Cmp<T> >
int compare (const String<T>& str1, const String<T>& str2)
{
    for (int i=0; i<str1.length () && i<str2.length (); i++)
        if (!C::eq (str1[i], str2[i])) return C::lt (str1[i], str2[i]) ? -1 : 1;
    return str1.length () - str2.length ();
}
```

Теперь мы можем записать:

```
void f (String<char> swede1, String<char> swede2){
    compare (swede1, swede2);                    // используется Cmp<char>
    compare<char, Literate> (swede1, swede2);    // используется Literate
}
```

Менее экзотический пример (для не шведов) — это сравнение обычным способом и без учета регистра:

```
class No_case { /* ... */ };

void f (String<char> s1, String<char> s2)
{
    compare (s1, s2);                    // учитывает регистр
    compare<char, No_case> (s1, s2);     // не учитывает регистр
}
```

Выбор алгоритма через аргумент шаблона и задание значения этого аргумента по умолчанию являются наиболее общим подходом и широко используются в стандартной библиотеке (см., например § 18.4). Довольно странно, но эти методы не используются для сравнений в ***basic_string*** (§ 13.2, глава 20). Параметры шаблонов, используемые для выбора того или иного алгоритма, часто называют «свойствами» (traits). Например, строковый класс из стандартной библиотеки использует ***char_traits*** (§ 20.2.1), алгоритмы стандартной библиотеки полагаются на свойства итераторов (§ 19.2.2), а контейнеры стандартной библиотеки — на свойства распределителей памяти (***allocators***) (§ 19.4).

Проверка семантики аргумента по умолчанию для параметра шаблона осуществляется тогда (и только тогда), когда аргумент по умолчанию действительно используется. В частности, пока мы воздерживаемся от использования аргумента шаблона по умолчанию ***Cmp<T>***, мы можем сравнивать (***compare*** ()) строки типа ***X***, для которых ***Cmp<X>*** не компилируется (например, потому что оператор < не определен для ***X***). Это является важнейшим моментом при проектировании стандартных контейнеров, которые используют аргументы шаблона для задания значений по умолчанию (§ 16.3.4).

13.5. Специализация

По умолчанию, шаблон предоставляет единственное определение, которое должно использоваться для всех аргументов шаблона (или комбинаций аргументов шаблона). Это не всегда имеет смысл для создателя шаблона. Я мог бы захотеть сказать «если аргументом шаблона является указатель, используй эту реализацию, если нет — используй ту» или «выдай сообщение об ошибке, если аргументом шаблона не является указатель на объект класса, производного от ***My_base***». Многие подобные проблемы можно решить, обеспечив альтернативные определения шаблона и предоставив компилятору делать выбор нужного варианта на основе аргументов шаблона, указанных при его использовании. Такие альтернативные определения шаблона называются *специализациями, определяемыми пользователем*, или просто *пользовательскими специализациями*.

Рассмотрим возможные использования шаблона ***Vector***:

```
template<class T> class Vector {          // общий тип vector
        T* v;
        int sz;
public:
        Vector ();
        explicit Vector (int);

        T& elem (int i) { return v[i]; }
        T& operator[] (int i);

        void swap (Vector&);
// ...
};

Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

Большинство векторов будут векторами указателей некоторого типа. На то существует несколько причин, главная из которых состоит в сохранении полиморфного поведения на этапе выполнения (§ 2.5.4, § 12.2.6). То есть любой, кто придерживается объектно-ориентированного программирования и использует безопасные с точки зрения типов контейнеры (такие, как контейнеры стандартной библиотеки), придет к множеству контейнеров с указателями.

В большинстве реализаций C++ код функций шаблона дублируется. Это положительно сказывается на производительности во время выполнения, но при отсутствии определенных мер предосторожности это ведет к разбуханию кода в критических случаях, таких как в примере с ***Vector***.

К счастью, имеется очевидное решение. Контейнеры указателей могут совместно пользоваться единственной реализацией. Это можно выразить при помощи специализации. Для начала определим версию (специализацию) ***Vector*** для указателей на ***void***:

```
template<> class Vector<void*> {
        void** p;
        // ...
```

```
    void*& operator[] (int i);
};
```

Затем эту специализацию можно использовать в качестве общей реализации для всех векторов указателей. Префикс ***template<>*** говорит о том, что речь идет о специализации, которая может быть указана без параметра шаблона. Аргументы шаблона, для которых должна использоваться специализация, задаются в угловых скобках <> после имени. То есть, ***<void*>*** означает, что данное определение должно использоваться как реализация для всех ***Vector***, у которых ***T*** является ***void****.

Vector<void*> является полной специализацией. То есть здесь нет параметра шаблона, который бы следовало задавать или который бы выводился при инстанцировании, когда мы используем специализацию. ***Vector<void*>*** используется с векторами, объявленными следующим образом:

```
Vector<void*> vpv,
```

Для определения специализации, которая используется для любого вектора указателей и только для векторов указателей, нам требуется *частичная специализация*:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
    typedef Vector<void*> Base;

    Vector () : Base () { }
    explicit Vector (int i) : Base (i) { }

    T*& elem (int i) { return reinterpret_cast<T*&> (Base::elem (i)); }
    T*& operator[] (int i) { return reinterpret_cast<T*&> (Base::operator[] (i)); }

// ...
};
```

Образец специализации ***<T*>*** после имени означает, что эта специализация должна использоваться для каждого типа указателя, то есть, это определение используется для каждого ***Vector***, аргумент шаблона которого можно выразить в виде ***T****. Например:

```
Vector<Shape*> vps;      // <T*> — это <Shape*>, поэтому T — Shape
Vector<int**> vppi;      // <T*> — это <int**>, поэтому T — int*
```

Обратите внимание, что когда используется частичная специализация, параметр шаблона выводится по образцу специализации; параметр шаблона не является фактическим аргументом шаблона. В частности, для ***Vector<Shape*>***, ***T*** — это ***Shape***, а не ***Shape****.

При наличии такой частичной специализации ***Vector***, мы имеем совместно используемую реализацию для всех векторов указателей. Класс ***Vector<T*>*** является просто интерфейсом к ***Vector<void*>***, выполненным исключительно с использованием механизмов наследования и встраивания.

Важным моментом является то, что это улучшение реализации ***Vector*** было достигнуто, не затрагивая интерфейс пользователя. Специализация является способом задания альтернативных реализаций для специфического использования общего интерфейса. Естественно, мы могли бы задать разные имена для общего вектора и вектора с указателями. Однако когда я попытался это сделать, многие люди стали забывать использовать классы указателей и обнаружили, что их код стал значительно больше, чем они ожидали. В подобных случаях гораздо лучше скрывать существенные детали реализации за общим интерфейсом.

Этот метод оказался полезным для предотвращения разбухания кода на практике. Те, кто не использует подобный метод (в C++ или других языках с аналогичными средствами параметризации типов) обнаруживают, что дублируемый код может вылиться в мегабайты даже в программах умеренного размера. За счет отсутствия необходимости компиляции этих дополнительных версий операций с векторами, такой подход может значительно сократить время, требуемое для компиляции и компоновки. Использование единственной специализации для реализации всех списков указателей является примером общей техники минимизации разбухания кода за счет увеличения совместно используемого кода.

Общий шаблон должен быть объявлен прежде любой специализации. Например:

```
template<class T> class List<T*> { /* ... */ },
template<class T> class List { /* ... */ };        // ошибка: общий шаблон
                                                   // после специализации
```

Критически важной информацией, предоставляемой шаблоном, является набор параметров шаблона, который должен обеспечить пользователь для применения самого шаблона или его специализаций. Поэтому объявления общего случая достаточно для объявления или определения специализации:

```
template<class T> class List;
template<class T> class List<T*> { /* ... */ };
```

Если общий шаблон используется, он должен быть где-то определен (§ 13.7).

Если пользователь специализирует где-нибудь шаблон, эта специализация должна быть в области видимости при каждом использовании шаблона с типом, для которого он был специализирован. Например:

```
template<class T> class List { /* ... */ };
List<int*> li;
template<class T> class List<T*> { /* ... */ },   // ошибка
```

В этом примере специализация ***List*** для ***int**** была сделана после использования ***List<int*>***.

Все специализации шаблона должны быть объявлены в том же самом пространстве имен, что и сам шаблон. Если используется явно объявленная специализация (в отличие от сгенерированной из более общего шаблона), она должна быть где-то явно определена (§ 13.7). Другими словами, явная специализация шаблона означает, что для этой специализации не генерируется определение.

13.5.1. Порядок специализаций

Одна специализация считается *более специализированной*, чем другая, если каждый список аргументов, соответствующий образцу первой специализации, также соответствует и второй специализации, но не наоборот. Например:

```
template<class T> class Vector;          // общий шаблон
template<class T> class Vector<T*>;      // специализация для любого указателя
template<> class Vector<void*>;          // специализация для void*
```

Любой тип можно использовать в качестве аргумента шаблона для наиболее общего класса ***Vector***, но только указатели можно использовать с ***Vector<T*>*** и только указатели ***void**** разрешается применять с ***Vector<void*>***.

Более специализированной версии будет отдано предпочтение в объявлениях объектов, указателей и т. д. (§ 13.5), а также при разрешении перегрузки (§ 13.3.2).

Образец специализации может быть указан в терминах типов, использующих конструкции, допустимые при выведении параметра шаблона (§ 13.3.1).

13.5.2. Специализация шаблонов функций

Естественно, специализация также полезна для шаблонов функций. Рассмотрим сортировку Шелла из § 7.7 и § 13.3. В ней производится сравнение элементов при помощи оператора < и затем перестановка элементов. Лучше определить этот алгоритм следующим образом:

```
template<class T> bool less (T a, T b) { return a < b; }

template<class T> void sort (vector<T>& v)          // определение
{
        const size_t n = v.size ();

        for (int gap = n/2; 0 < gap; gap /= 2)
                for (int i = gap; i < n; i++)
                        for (int j = i – gap; 0 <= j; j –= gap)
                                if (less (v[j+gap], v[j]))
                                        swap (v[j], v[j+gap]);
                                else
                                        break;
}
```

Это не является улучшением самого алгоритма, но позволяет улучшить его реализацию. Так, как он записан, алгоритм ***sort*** () не будет правильно сортировать ***Vector<char*>***, потому что < будет сравнивать два ***char****, то есть адреса первых ***char*** в каждой строке. Вместо этого мы бы хотели, чтобы сравнивались символы. Простая специализация ***less*** () для ***const char**** позаботится об этом:

```
template<> bool less<const char*> (const char* a, const char* b)
{
        return strcmp (a, b) < 0;
}
```

Как и в случае с классами (§ 13.5), префикс ***template<>*** означает, что речь идет о специализации, которую можно указывать без параметра шаблона. Выражение ***<const char*>*** после имени шаблона функции означает, что эта специализация будет использоваться в тех случаях, когда аргументом шаблона является ***const char****. Так как аргумент шаблона может быть выведен по списку аргументов функции, нет необходимости его явно указывать. Поэтому мы можем упростить определение специализации:

```
template<> bool less<> (const char* a, const char* b)
{
        return strcmp (a, b) < 0;
}
```

При наличии префикса *template*<> вторые пустые скобки <> избыточны, поэтому мы, как правило, можем написать просто:

```
template<> bool less(const char* a, const char* b)
{
    return strcmp(a, b) < 0;
}
```

Я предпочитаю эту более короткую форму объявления.

Рассмотрим очевидное определение *swap*():

```
template<class T> void swap(T& x, T& y)
{
    T t = x,      // копирование x во временную переменную
    x = y;        // копирование y в x
    y = t;        // копирование временной переменной в y
}
```

Это довольно неэффективная реализация при вызове для вектора векторов, так как происходит копирование всех элементов. Данную проблему также можно решить подходящей специализацией. Сам объект *Vector* будет содержать только необходимые данные для обеспечения косвенного доступа к элементам (наподобие *string*; § 11.12, § 13.2). Таким образом, перестановка будет осуществляться путем обмена представлениями. Для работы с этим представлением я реализовал в *Vector* функцию-член *swap*() (§ 13.5):

```
template<class T> void Vector<T>::swap(Vector& a) // обмен представлениями
{
    swap(v, a.v);
    swap(sz, a.sz);
}
```

Теперь эту функцию-член *swap*() можно использовать для определения специализации общей *swap*():

```
template<class T> void swap(Vector<T>& a, Vector<T>& b)
{
    a.swap(b);
}
```

Эти специализации *less*() и *swap*() используются в стандартной библиотеке (§ 16.3.9, § 20.3.16). Кроме этого, они являются примерами широко применяемых методов. Специализация полезна, когда существует более эффективная альтернатива общему алгоритму для конкретных аргументов шаблона (в нашем случае, *swap*()). Кроме того, специализация отлично работает, когда нерегулярность типа аргумента приводит к тому, что общий алгоритм выдает неправильный результат (см. пример с *less*() выше). Этими «нерегулярными типами» часто являются встроенные указатели и массивы.

13.6. Наследование и шаблоны

Шаблоны и наследование являются механизмами построения новых типов из уже существующих и написания полезного кода, использующего различные формы общности. Как продемонстрировано в § 3.7.1, § 3.8.5 и § 13.5, комбинация этих двух механизмов является основой для многих полезных методов программирования.

Создание производного шаблона класса от класса, не являющегося шаблоном, является способом обеспечения общей реализации для набора шаблонов. Хорошим примером является список из § 13.5:

```
template<class T> class List<T*> : private List<void*> { /* ... */ };
```

Можно рассматривать подобные примеры в качестве иллюстрации того, как шаблон используется для обеспечения элегантного и безопасного с точки зрения типов интерфейса к средству, которое в противном случае было бы неудобно использовать.

Естественно, часто бывает полезно создавать шаблоны, производные от другого шаблона. Базовый класс в этом случае является строительным блоком для реализации дальнейших классов. Если члены базового класса зависят от параметров шаблона производного класса, сам базовый класс должен быть параметризирован. Хорошим примером может служить ***Vec*** из § 3.7.2:

```
template<class T> class vector { /* ... */ };
template<class T> class Vec : public vector<T> { /* ... */ };
```

Правила разрешения перегрузки шаблонов функций гарантируют, что функции работают «правильно» для таких производных типов (§ 13.3.2).

Наличие одного и того же параметра шаблона для базового и производного классов является самым распространенным случаем, но это не обязательное требование. В интересных, хотя и реже встречаемых случаях, используется передача самого производного типа в базовый класс. Например:

```
template <class C> class Basic_ops {                    // базовые операции с контейнерами
public
        bool operator==(const C&) const;                // сравнивает все элементы
        bool operator!=(const C&) const;
        // ...
        // обеспечивает доступ к операциям класса C
        const  C& derived() const {return static_cast<const C&>(*this);}
};

template<class T> class Math_container : public Basic_ops<Math_container<T> > {
public:
        size_t size() const;
        T& operator[](size_t);
        const  T& operator[](size_t) const;
        // ...
};
```

Это позволяет определить базовые операции с контейнерами один единственный раз и отделить их от определения самих контейнеров. Однако определение операций, таких как == и !=, должно быть выражено в терминах и контейнера и его элементов, поэтому базовый класс должен быть передан в шаблон контейнера.

Предположив, что ***Math_container*** подобен традиционному вектору, определения членов ***Basic_ops*** могли бы выглядеть следующим образом:

```
template <class C> bool Basic_ops<C>::operator==(const C& a) const
{
        if(derived().size != a.size()) return false;
```

```
        for (int i = 0; i < derived().size(); ++i)
            if( derived()[i] != a[i]) return false;
        return true;
}
```

Альтернативной техникой разделения контейнеров и операций является их комбинирование через аргументы шаблона (а не посредством наследования):

```
template<class T, class C> class Mcontainer {
    C elements;
public:
    T& operator[](size_t i) { return elements[i]; }

    // сравнение элементов
    friend bool operator==<>(const Mcontainer&, const Mcontainer&);
    friend bool operator!=<>(const Mcontainer&, const Mcontainer&);
    //...
};

template<class T> class My_array { /* ... */ };

Mcontainer<double, My_array<double> > mc;
```

Класс, генерируемый из шаблона класса, является обычным классом. Следовательно, у него могут быть функции-друзья (§ В.13.2). В нашем случае я воспользовался функциями-друзьями для достижения привычного симметричного стиля аргументов операторов == и != (§ 11.3.2). Можно также рассмотреть вариант с передачей шаблона вместо контейнера в качестве аргумента *C* (§ В.13.3).

13.6.1. Параметризация и наследование

Шаблон является механизмом параметризации определения класса или функции произвольным типом. Код, реализующий шаблон, идентичен для всех типов параметров, также как и большая часть кода, использующая шаблон. Абстрактный класс определяет интерфейс. Большая часть кода различных реализаций абстрактного класса может совместно использоваться в иерархии классов, и большинство фрагментов, использующих абстрактный класс, не зависит от его реализации. С точки зрения проектирования оба подхода настолько близки, что заслуживают общего названия. Так как оба метода позволяют выразить алгоритм один раз и использовать его со множеством типов, их вместе часто называют *полиморфными*. Для того чтобы все-таки их различать, то, что обеспечивает виртуальные функции, называют *полиморфизмом времени выполнения*, а то, что предоставляют шаблоны — *полиморфизмом времени компиляции* или *параметрическим полиморфизмом*.

Итак, когда мы выбираем шаблон, а когда полагаемся на абстрактные классы? В любом случае мы работаем с объектами, которые совместно используют общий набор операций. Если между этими объектами не требуется иерархической зависимости, лучше использовать их в качестве аргументов шаблонов. Если фактические типы этих объектов не известны во время компиляции, их наилучшим представлением являются классы, производные от общего абстрактного класса. Если целью является эффективность на этапе выполнения, то есть большое значение имеет встраивание операций, следует использовать шаблоны. Эти вопросы подробнее обсуждаются в § 24.4.1.

13.6.2. Члены-шаблоны

Класс или шаблон класса может иметь члены, которые сами являются шаблонами. Например:

```
template<class Scalar> class complex {
        Scalar re, im,
public:
        template<class T> complex (const complex<T>& c) :
                re (c.real ()), im (c.imag ()) { }
        // ...
};

complex<float> cf (0, 0),
complex<double> cd = cf,      // правильно: используется преобразование float в double

class Quad {
        // нет преобразования в int
};

complex<Quad> cq;
complex<int> ci = cq;         // ошибка: нет преобразования Quad в int
```

Другими словами, вы можете создать ***complex<T1>*** из ***complex<T2>*** в том и только в том случае, если вы инициализируете ***T1*** при помощи ***T2***. Это выглядит разумно.

К сожалению, C++ допускает некоторые неразумные преобразования встроенных типов, такие как ***double*** в ***int***. Проблемы потери значения можно обнаружить во время выполнения при помощи проверяемого преобразования в стиле ***implicit_cast*** (§ 13.3.1) и функции ***checked*** () (§ B.6.2.6):

```
template<class Scalar> class complex {
        Scalar re, im,
public.
        complex () : re (0), im (0) { }
        complex (const complex<Scalar>& c) : re (c.real ()), im (c.imag ()) { }

        template<class T2> complex (const complex<T2>& c) :
                re (checked_cast<Scalar> (c.real ())),
                im (checked_cast<Scalar> (c.imag ())) { }
        // ...
};
```

Для полноты картины я добавил конструктор по умолчанию и копирующий конструктор. Любопытно, что конструктор шаблона никогда не используется для генерации копирующего конструктора (так, чтобы при отсутствии явно объявленного копирующего конструктора, генерировался бы копирующий конструктор по умолчанию). В этом случае сгенерированный копирующий конструктор был бы идентичен тому, что я явно указал. Аналогично, копирующее присваивание (§ 10.4.4.1, § 11.7) должно быть определено как нешаблонный оператор.

Член-шаблон не может быть виртуальным. Например:

```
class Shape {
        //..
        // ошибка: виртуальный шаблон
```

```
        template<class T> virtual bool intersect (const T&) const =0;
};
```

Это недопустимо. Если бы такое было дозволено, нельзя было бы пользоваться традиционной техникой с таблицей виртуальных функций (§ 2.5.5) для реализации виртуальных функций. Компоновщику пришлось бы добавлять новую точку входа в виртуальной таблице для класса ***Shape*** каждый раз, когда кто-нибудь вызывает ***intersect*** () с новым типом аргумента.

13.6.3. Отношения наследования

Шаблон класса обычно воспринимается как спецификация того, как должны создаваться конкретные типы. Другими словами, реализация шаблона является механизмом генерации необходимых типов на основе спецификации пользователя. Поэтому шаблон класса иногда называют *генератором типов.*

С точки зрения правил C++ два класса, сгенерированные из одного шаблона, не связаны друг с другом никакими отношениями. Например:

```
class Shape { /* ... */ };
class Circle : public Shape { /* ... */ };
```

При наличии этих объявлений, иногда пытаются обращаться с ***set<Circle*>*** как с ***set<Shape*>***. Это серьезная логическая ошибка, основанная на неверном рассуждении: «***Circle*** является ***Shape***, поэтому множество объектов ***Circle*** также является множеством объектов ***Shape***, следовательно я могу использовать множество объектов ***Circle*** как множество объектов ***Shape***». В этом рассуждении «следовательно» не верно. Причина: множество объектов типа ***Circle*** гарантирует, что членами множества являются объекты типа ***Circle***, множество объектов типа ***Shape*** не дает такой гарантии. Например:

```
class Triangle : public Shape { /* ... */ };

void f(set<Shape*>& s)
{
        //...
        s.insert (new Triangle ());
        //...
}

void g (set<Circle*>& s)
{
        f(s);  // ошибка несоответствия типов: s — это set<Circle*>, а не set<Shape*>
}
```

Подобный код не может быть скомпилирован, потому что нет встроенного преобразования ***set<Circle*>***& в ***set<Shape*>***&. И не должно быть. Гарантия того, что члены множества ***set<Circle*>*** являются объектами типа ***Circle*** позволяет нам безопасно и эффективно применять к этим членам специфические для окружностей операции, такие как определение радиуса. Если бы мы допустили, чтобы с ***set<Circle*>*** можно было бы обращаться как с ***set<Shape*>***, мы больше не смогли бы обеспечивать эту гарантию. Например, ***f*** () вставляет ***Triangle**** в свой аргумент ***set<Shape*>***. Если бы ***set<Shape*>*** мог быть заменен на ***set<Circle*>***, была бы нарушена фундаментальная гарантия того, что ***set<Circle*>*** содержит только объекты типа ***Circle****.

13.6.3.1. Преобразования шаблонов

В примере из предыдущего раздела демонстрируется, что не может существовать отношений *по умолчанию* между классами, сгенерированными из одного шаблона. Однако для некоторых шаблонов мы хотели бы выразить такие отношения. Например, когда мы определяем шаблон по указателю, мы хотели бы отразить отношения наследования между объектами, на которые указатели указывают. При желании члены-шаблоны (§ 13.6.2) позволяют нам выразить множество таких взаимоотношений. Рассмотрим пример:

```
template<class T> class Ptr {                    // указатель на T
        T* p;
public:
        Ptr (T*);
        Ptr (const Ptr&);                        // копирующий конструктор
        template<clas T2> operator Ptr<T2> ();   // преобразование Ptr<T> в Ptr<T2>
        // ...
};
```

Мы хотим определить операторы преобразования для этих определяемых пользователем указателей ***Ptr***, чтобы выразить отношения наследования, к которым мы привыкли для встроенных указателей. Например:

```
void f(Ptr<Circle> pc)
{
        Ptr<Shape> ps = pc;          // должно работать
        Ptr<Circle> pc2 = ps;        // не должно работать
}
```

Мы хотим разрешить первую инициализацию в том и только в том случае, когда ***Shape*** является прямым или косвенным открытым базовым классом для ***Circle***. В общем, нам нужно определить оператор преобразования таким образом, чтобы преобразование ***Ptr<T>*** в ***Ptr<T2>*** допускалось тогда и только тогда, когда ***T**** можно присвоить ***T2****. Это можно сделать следующим образом:

```
template<class T>
        template<class T2>
                Ptr<T>::operator Ptr<T2> () { return Ptr<T2> (p); }
```

Инструкция ***return*** будет скомпилирована в том и только в том случае, когда указатель ***p*** (типа ***T****) может быть аргументом конструктора ***Ptr<T2>*** (***T2****). Поэтому, если ***T**** можно неявно преобразовать в ***T2****, преобразование ***Ptr<T>*** в ***Ptr<T2>*** будет работать. Например:

```
void f(Ptr<Circle> pc)
{
        Ptr<Shape> ps = pc;          // правильно: можно преобразовать Circle* в Shape*
        Ptr<Circle> pc2 = ps;        // ошибка: нельзя преобразовать Shape* в Circle*
}
```

Будьте внимательны и определяйте только логически осмысленные преобразования.

Обратите внимание, что списки параметров шаблона и его члены-шаблоны нельзя объединить. Например:

```
template<class T, class T2>          // ошибка
        Ptr<T>::operator Ptr<T2> () { return Ptr<T2> (p); }
```

13.7. Организация исходного кода

Существует два очевидных способа организации кода с использованием шаблонов:

[1] Включать (*#include*) определения шаблонов до их использования в единице трансляции.

[2] Включать (только) объявления шаблонов до их использования в единице трансляции, и компилировать их определения отдельно.

Кроме того, иногда шаблоны функций сначала объявляются, затем используются, а потом определяются в одной и той же единице трансляции.

Для иллюстрации различий между двумя этими подходами, рассмотрим простой шаблон:

```
#include <iostream>

template<class T> void out(const T& t) { std::cerr << t; }
```

Мы могли бы поместить это в ***out.c*** и включать в том месте, где требуется ***out***(). Например:

```
// user1.c:
    #include "out.c"
    // использование out()

// user2.c:
    #include "out.c"
    // использование out()
```

Таким образом, определение функции ***out***() и все объявления, от которых она зависит, включаются в несколько единиц трансляции. Генерировать код (только) когда он требуется и оптимизировать процесс чтения избыточных определений является задачей компилятора. Согласно этой стратегии подход к шаблонам функций такой же, как к встроенным функциям.

Одна очевидная проблема состоит в том, что все, от чего зависит определение ***out***(), добавляется в каждый файл, использующий ***out***(), увеличивая таким образом количество информации, которое должен обработать компилятор. Другая проблема состоит в том, что пользователи могут впасть в зависимость от объявлений, включенных только для определения ***out***(). Эту опасность можно свести к минимуму путем использования пространств имен, отсутствием макросов и, в общем, уменьшением включаемой информации.

Стратегия раздельной компиляции является логическим завершением следующей мысли: если определение шаблона не включено в код пользователя, ни одна из его зависимостей не должна влиять на этот код. Таким образом, мы разбиваем исходный ***out.c*** на два файла:

```
// out.h:
    template<class T> void out(const T& t);

// out.c:
    #include<iostream>
    #include "out.h"

    export template<class T> void out(const T& t) { std::cerr << t; }
```

Теперь файл ***out.c*** содержит всю информацию, необходимую для определения ***out***(), а ***out.h*** содержит только то, что необходимо для ее вызова. Пользователь включает только объявление (интерфейс):

```
// user1.c:
    #include "out.h"
    // использование out ()

// user2.c:
    #include "out.h"
    // использование out ()
```

Обратите внимание: для того чтобы к нему можно было обращаться из различных единиц компиляции, определение шаблона должно быть явно объявлено с ***export*** (§ 9.2.3). Это можно сделать, добавив ***export*** к определению или предварив им объявление. В противном случае определение должно находиться в области видимости в момент использования шаблона.

То, какая из стратегий или комбинаций стратегий является наилучшей, зависит от: используемых компиляторов и компоновщиков; приложения, которое вы создаете; внешних ограничений на построение систем. Обычно, встроенные функции и другие небольшие функции, которые вызывают другие шаблоны функций, являются кандидатами для включения в каждую единицу компиляции, в которой они используются. В реализации с умеренной поддержкой со стороны компоновщика при инстанцировании шаблонов такой подход может ускорить компиляцию и повысить информативность сообщений об ошибках.

Включение определения делает его уязвимым в смысле возможного изменения его значения макросами и объявлениями в контексте, в котором оно включается. Следовательно, большие шаблоны функций и шаблоны функций с нетривиальными зависимостями от контекста лучше компилировать отдельно. Кроме того, если определение шаблона требует большого числа объявлений, эти объявления могут иметь нежелательные побочные эффекты при включении в контекст, в котором используется шаблон.

Я считаю, что подход с раздельно компилируемыми определениями шаблонов и включением в пользовательский код только объявлений является идеальным. Однако применение идеалов должно считаться с практическими ограничениями, а раздельная компиляция шаблонов является в некоторых реализациях дорогим удовольствием.

Независимо от выбора стратегии невстроенные статические члены (§ B.13.1) должны иметь уникальное определение в единице компиляции. Из этого следует, что такие члены лучше не использовать в шаблонах, которые предполагается включать во многие единицы трансляции.

Идеальным является код, который может одинаково работать независимо от того, компилируется он в виде единственного модуля, или разделяется на несколько раздельно компилируемых единиц. К этому идеалу следует стремиться, сокращая зависимости определения шаблона от окружения, а не пытаясь «втаскивать» как можно большую часть контекста определения в процесс инстанцирования.

13.8. Советы

[1] Используйте шаблоны для представления алгоритмов, применимых ко многим типам аргументов; § 13.3.

[2] Используйте шаблоны для представления контейнеров; § 13.2.

[3] Создавайте специализации для контейнеров указателей с целью минимизации размера кода; § 13.5.

[4] Всегда объявляйте общую форму шаблона до специализаций; § 13.5.
[5] Объявляйте специализацию до ее использования; § 13.5.
[6] Сводите к минимуму зависимости определения шаблона от контекста инстанцирования; § 13.2.5, § B.13.8.
[7] Определяйте каждую объявленную специализацию; § 13.5.
[8] Проанализируйте, требуется ли специализация вашего шаблона для массивов и С-строк ; § 13.5.2.
[9] Используйте аргументы шаблона для выбора алгоритма; § 13.4.
[10] Используйте специализацию и перегрузку для предоставления единого интерфейса для реализаций одной и той же концепции применительно к различным типам; § 13.5.
[11] Предоставьте простой интерфейс для простых случаев и используйте перегрузку и аргументы по умолчанию для менее общих случаев; § 13.5, § 13.4.
[12] Отлаживайте конкретные примеры до их обобщения в шаблоны; § 13.2.1.
[13] Не забывайте экспортировать (***export***) определения шаблонов, доступ к которым необходим из других единиц трансляции; § 13.7.
[14] Компилируйте раздельно большие шаблоны и шаблоны с нетривиальными зависимостями от контекста; § 13.7.
[15] Используйте шаблоны для определения преобразований, но определяйте эти преобразования очень внимательно; § 13.6.3.1.
[16] При необходимости ограничивайте аргументы шаблона при помощи функции-члена ***constraint***(); § 13.9[16].
[17] Используйте явную форму инстанцирования для минимизации времени компиляции и компоновки; § B.13.10.
[18] Отдавайте предпочтение шаблонам по отношению к производным классам, когда эффективность времени выполнения имеет исключительное значение; § 13.6.1.
[19] Отдавайте предпочтение производным классам по отношению к шаблонам, если большое значение имеет добавление нового варианта без перекомпиляции; § 13.6.1.
[20] Отдавайте предпочтение шаблонам по отношению к производным классам, когда нельзя определить общий базовый класс; § 13.6.1.
[21] Отдавайте предпочтение шаблонам по отношению к производным классам, если вы должны использовать встроенные типы и структуры из-за соображений совместимости; § 13.6.1.

13.9. Упражнения

1. (*2) Устраните ошибки в определении ***List*** в § 13.2.5 и напишите на C++ код, эквивалентный тому, что должен сгенерировать компилятор для определения ***List*** и функции *f*(). Выполните небольшой тест с вашим кодом, «сгенерированным» вручную и с кодом, сгенерированным компилятором из версии с шаблоном. Если вы знаете, как это сделать в вашей системе, сравните сгенерированные машинные коды.
2. (*3) Напишите шаблон класса односвязного списка, который принимает элементы любого типа, производного от класса ***Link***, имеющего всю информацию, необходимую для связывания элементов. Такой список называется *интрузивным*. При помощи этого списка напишите односвязный список, который принимает элементы любого типа (неинтрузивный список). Сравните производительность обоих классов списков и обсудите разницу.

3. (*2.5) Напишите интрузивные и неинтрузивные двусвязные списки. Какие дополнительные операции нужно реализовать по сравнению с теми, которые вы сочли нужным ввести для односвязных списков?
4. (*2) Завершите шаблон ***String***, из § 13.2, основанный на классе ***String*** из § 11.12.
5. (*2) Определите шаблон функции ***sort*** (), который получает критерий сравнения в виде аргумента шаблона. Определите класс ***Record*** с двумя членами ***count*** и ***price***. Отсортируйте ***vector<Record>*** по каждому члену.
6. (*2) Реализуйте шаблон ***qsort*** ().
7. (*2) Напишите программу, которая читает пары (***key***, ***value***) (ключ, значение) и выводит сумму всех ***value***, соответствующих каждому ***key***. Укажите, какие требования должны предъявляться к типам ***key*** и ***value***.
8. (*2.5) Реализуйте простой класс ***Map*** (ассоциативный массив), основанный на классе ***Assoc*** из § 11.8. Убедитесь, что ***Map*** правильно работает и с С-строками, и со строками ***string*** в качестве ключей. Убедитесь, что ***Map*** правильно работает с типами, как имеющими, так и не имеющими конструкторы по умолчанию. Реализуйте механизм итерации по элементам ***Map***.
9. (*3) Сравните производительность программы подсчета слов из § 11.8 с программой, не использующей ассоциативного массива. В обоих случаях, используйте один и тот же стиль ввода/вывода.
10. (*3) Реализуйте заново ***Map*** из § 13.9[8] с использованием более подходящих структур (например, «красно-черное дерево» или Splay-дерево).
11. (*2.5) Воспользуйтесь ***Map*** для реализации функции топологической сортировки. Топологическая сортировка описана в [Knuth, 1968], том 1 (второе издание), стр. 262.
12. (*1.5) Перепишите программу суммирования из § 13.9[7] так, чтобы она правильно работала с именами, содержащими пробелы, например «член данных».
13. (*2) Напишите шаблоны ***readline*** () для различных форматов строк (lines). Например (наименование, количество, цена).
14. (*2) Используйте технику, кратко описанную для ***Literate*** в § 13.4, для сортировки строк в обратном лексикографическом порядке. Убедитесь, что этот метод работает как для реализаций C++, где ***char*** является ***signed***, так и для тех, где ***char*** — ***unsigned***. Воспользуйтесь вариацией этой техники для реализации сортировки независимо от регистра.
15. (*1.5) Напишите пример, который демонстрирует по крайней мере три различия между шаблоном функции и макросом (не считая разницы в синтаксисе определения).
16. (*2) Разработайте схему, которая гарантирует, что компилятор проверяет общие ограничения на аргументы шаблона для каждого шаблона, для которого создается объект. Недостаточно просто проверить ограничения типа «аргумент ***T*** должен быть класса, производного от ***My_base***».

14

Обработка исключений

Не перебивайте меня,
когда я вас перебиваю.
— Уинстон В. Черчилль

Обработка ошибок — группировка исключений — перехват исключений — перехват всех исключений — повторная генерация — управление ресурсами — ***auto_ptr*** — исключения и ***new*** — исчерпание ресурсов — исключения в конструкторах — исключения в деструкторах — исключения, не являющиеся ошибками — спецификации исключений — неожидаемые исключения — неперехваченные исключения — исключения и эффективность — альтернативные методы обработки ошибок - стандартные исключения — советы — упражнения.

14.1. Обработка ошибок

Как отмечалось в § 8.3, автор библиотеки может обнаружить ошибки времени выполнения, но, в общем случае, не имеет ни малейшего представления, что с ними делать. Пользователь библиотеки может знать, как бороться с такими ошибками, но не может их обнаружить — в противном случае, они бы обрабатывались в коде пользователя, и их обнаружение не было бы возложено на библиотеку. Для помощи в решении подобных проблем введено понятие *исключения*. Фундаментальная идея состоит в том, что функция, обнаружившая проблему, но не знающая как ее решить, *генерирует* (throw) исключение в надежде, что вызвавшая ее (непосредственно или косвенно) функция сможет решить возникшую проблему. Функция, которая хочет решать проблемы данного типа, может указать, что она *перехватывает* (catch) такие исключения (§ 2.4.2, § 8.3).

Такой стиль обработки ошибок предпочтительней многих традиционных техник. Рассмотрим альтернативы. При обнаружении проблемы, которая не может быть решена локально, функция может:

[1] прекратить выполнение,
[2] возвратить значение, означающее «ошибка»,
[3] возвратить допустимое значение и оставить программу в ненормальном состоянии;
[4] вызвать функцию, предназначенную для обработки «ошибочных» ситуаций.

Вариант [1] — «прекратить выполнение» — это то, что происходит по умолчанию, когда не перехватывается исключение. Для большинства ошибок мы можем и долж-

ны придумать кое-что получше. В частности, библиотека, которая не знает о цели и общей стратегии программы, ее использующей, не может просто выйти (функцией ***exit*** ()) из программы или прервать ее выполнение (функцией ***abort*** ()). Библиотека, безусловно завершающая выполнение, не может использоваться в программе, первое требование к которой — надежность. Одним из способов представления исключений является их рассмотрение в качестве средства передачи управления вызывающей функции, когда локально невозможно выполнить никаких разумных действий.

Вариант [2] — «возвратить значение, сигнализирующее об ошибке» — не всегда выполним, потому что часто нет приемлемого соответствующего значения. Например, если функция возвращает целое, любое целое может быть приемлемым результатом. Даже в тех случаях, когда такой подход применим, он часто неудобен, потому что результат каждого вызова должен проверяться на ошибочное значение. Это легко может удвоить размер программы (§ 14.8). Поэтому такой подход редко используется систематически для обнаружения всех ошибок.

Вариант [3] — «возвратить допустимое значение и оставить программу в ненормальном состоянии» — имеет тот недостаток, что вызывающая функция может не заметить, что программа находится в ненормальном состоянии. Например, многие стандартные функции библиотеки C устанавливают значение глобальной переменной ***errno*** для индикации ошибки (§ 20.4.1, § 22.3). Однако программы в большинстве случаев не проверяют ***errno*** достаточно систематически, чтобы избежать последующих ошибок, вызванных значениями, возвращенными некорректно завершившимися функциями. Более того, использование глобальных переменных для записи информации об ошибках работает неудовлетворительно при наличии параллельных процессов.

Обработка исключений не предназначена для решения проблем, для которых подходит вариант [4] «вызвать функцию обработки ошибок». Однако в отсутствие исключений у функции обработки ошибок имеется в точности те же самые три указанные альтернативы, *как* обрабатывать ошибку. Дальнейшее обсуждение функций обработки ошибок см. в § 14.4.5.

Механизм обработки исключений предоставляет альтернативу традиционным методам в тех случаях, когда они не достаточны, не элегантны и подвержены ошибкам. Он предоставляет способ явного отделения кода обработки ошибок от «обычного» кода, делая таким образом программу более читабельной и лучше подходящей для различных инструментальных средств. Механизм обработки исключений предоставляет более регулярный способ обработки ошибок, упрощая в результате взаимодействие между отдельно написанными фрагментами кода.

Одним из аспектов обработки исключений, который будет новым для программистов на C и Pascal, является то, что по умолчанию реакцией на ошибку (особенно на ошибку в библиотеке) будет завершение выполнения программы. Традиционной реакцией является «как-нибудь довести дела до конца и надеяться на лучшее». Таким образом, обработка исключений делает программы более «хрупкими» в том смысле, что требуется больше внимания и усилий для их приемлемого выполнения . Однако такой подход выглядит предпочтительней, чем обнаружение неверного результата позднее в процессе разработки — или после того, как процесс разработки считается завершенным и программа отдана ничего не подозревающим конечным пользователям. В тех случаях, когда завершение программы является неприемлемым, мы мо-

жем перехватывать все исключения (§ 14.3.2) или перехватывать все исключения определенного вида (§ 14.6.2). Таким образом, исключение завершает выполнение программы только в том случае, если программист позволяет это сделать. Это лучше, чем безусловное завершение, которое происходит, когда традиционное неполное восстановление приводит к катастрофическим ошибкам.

В некоторых случаях программисты пытались смягчить непривлекательные аспекты подхода «как-нибудь довести дела до конца» путем вывода сообщений об ошибках, отображения диалоговых окон, обращением к пользователю за подсказкой и т. д. Подобные подходы прежде всего полезны при отладке в ситуации, когда пользователем является программист, знакомый со структурой программы. В руках же не разработчика, библиотека, которая запрашивает помощь у (возможно отсутствующего) пользователя/оператора, неприемлема. Кроме того, во многих случаях сообщения об ошибке некуда выводить (скажем, если программа выполняется в среде, где поток ***cerr*** не подключен к чему-нибудь, что видит пользователь); в любом случае они будут непонятны конечному пользователю. Как минимум, сообщение об ошибке может быть не на том языке (скажем, на финском для английского пользователя). Хуже того, сообщение об ошибке будет, как правило, иметь отношение к концепциям библиотеки, совершенно незнакомым пользователю (скажем, «недопустимый аргумент ***atan2***», вызванное неправильным вводом в графической системе). Хорошая библиотека не должна быть настолько беспомощной. Исключения предоставляют возможность коду, обнаружившему проблему, от последствий которой он не может восстановиться, передать эту проблему некоторой части системы, которая в состоянии ее решить. Только часть системы, имеющая представление о контексте, в котором выполняется программа, имеет шанс выдать осмысленное сообщение об ошибке.

Можно рассматривать механизм обработки исключений как аналог времени выполнения для таких механизмов времени компиляции, как проверка соответствия типов и контроль неоднозначности. Обработка исключений делает процесс проектирования более важным и может увеличить объем работ, который требуется выполнить, чтобы заставить работать исходную, содержащую ошибки, версию программы. Однако результатом является код, у которого намного больше шансов: выполняться так, как ожидалось; работать в качестве части большей программы; быть понятным для других программистов; успешно обрабатываться различными инструментальными средствами. Аналогично другим средствам C++, обработка исключений предоставляет конкретные средства языка для поддержки «хорошего стиля», который можно практиковать лишь неформально и не в полном объеме в таких языках, как C и Pascal.

Надо понимать, что обработка ошибок остается сложной задачей и что механизм обработки исключений — несмотря на большую формализацию, чем альтернативные методы — относительно менее структурирован по сравнению со средствами языка, обеспечивающими локальное управление выполнением. Механизм обработки исключений C++ предоставляет программисту средство обработки ошибок в том месте, где их наиболее естественно обрабатывать при данной структуре системы. Исключения делают сложность обработки ошибок более явной. Однако исключения не являются причиной этой сложности. Не обвиняйте того, кто принес плохую новость.

Возможно, наступил подходящий момент для повторного прочтения § 8.3, где изложены базовый синтаксис, семантика и стили использования исключений.

14.1.1. Альтернативный взгляд на исключения

«Исключение» является словом, которое означает разное для разных людей. Механизм исключений C++ разработан для обработки ошибок и других исключительных состояний (отсюда название). В частности, подразумевалась поддержка обработки ошибок в программах, составленных из независимо разработанных компонент.

Механизм исключений предназначен для обработки только синхронных исключений, таких как выход за пределы диапазона в массиве и ошибки ввода/вывода. Асинхронные события, такие как прерывания от клавиатуры и некоторые арифметические ошибки, не обязательно являются исключительными и не обрабатываются непосредственно обсуждаемым механизмом. Асинхронные события требуют механизма, фундаментально отличного от исключений (как они здесь определены), для обеспечения их обработки понятным и эффективным способом. Многие системы имеют механизмы, например сигналы, для работы с асинхронностью, но ввиду того, что они склонны приводить к зависимости от системы, здесь не приводится описание подобных средств.

Механизм обработки исключений является нелокальной структурой, основанной на «раскручивании» стека (§ 14.4), и его можно рассматривать в качестве альтернативы механизму возврата из функции. Поэтому вполне законно использовать исключения, которые не имеют никакого отношения к ошибкам (§ 14.5). Однако изначальной целью механизма обработки исключений и предметом обсуждения этой главы является обработка ошибок и обеспечение устойчивости при возникновении ошибок.

В стандарте C++ нет понятия потока или процесса. Поэтому исключения, имеющие отношение к параллельности, здесь не обсуждаются. Средства параллельного выполнения, имеющиеся в вашей системе, описаны в ее документации. Здесь я просто отмечу, что механизм обработки исключений C++ разработан так, чтобы быть эффективным и в параллельной среде при соблюдении программистом (или системой) базовых правил параллельного выполнения, таких как правильное блокирование совместно используемых структур данных во время их использования.

Механизмы обработки исключений C++ предназначены для генерации сообщения об ошибке и обработки ошибок и исключительных событий. Задача программиста — решить, что является исключительным в данной программе. Это не всегда просто (§ 14.5). Может ли событие, которое происходит в большинстве случаев при выполнении программы, считаться исключительным? Может ли запланированное и обрабатываемое событие считаться ошибкой? Ответ на оба вопроса — да. «Исключительный» не означает «почти никогда не происходящий» или «разрушительный». Лучше рассматривать исключения в смысле «некоторая часть системы не смогла сделать то, что от нее требовалось». После этого мы обычно можем попытаться сделать что-нибудь другое. Генерация исключений должна происходить реже по сравнению с вызовами функций, в противном случае структура системы будет недостаточно стройна. Однако мы вправе ожидать, что большинство крупных программ генерируют и перехватывают по крайней мере несколько исключений в процессе нормального и успешного выполнения.

14.2. Группировка исключений

Исключение является объектом некоторого класса, являющегося представлением исключительного случая. Код, обнаруживший ошибку (часто библиотека), генерирует объект (§ 8.3) инструкцией ***throw***. Фрагмент кода выражает свое жела-

ние обрабатывать исключение при помощи инструкции ***catch***. Результатом ***throw*** является раскручивание стека до тех пор, пока не будет обнаружен подходящий ***catch*** (в функции, которая непосредственно или косвенно вызывала функцию, сгенерировавшую исключение).

Часто исключения естественным образом разбиваются на семейства. Из этого следует, что наследование может оказаться полезным для структурирования исключений и помочь при их обработке. Например, исключения для математической библиотеки можно организовать следующим образом:

```
class Matherr { };
class Overflow: public Matherr { };          // переполнение сверху
class Underflow: public Matherr { };         // переполнение снизу
class Zerodivide: public Matherr { };        // деление на ноль
// ...
```

Это позволяет нам обрабатывать любой ***Matherr***, не заботясь о том, какое в точности исключение возникло. Например:

```
void f()
{
    try {
        // ...
    }
    catch (Overflow) {
        // обработка Overflow или всех производных от Overflow
    }
    catch (Matherr) {
        // обработка любой Matherr, не являющейся Overflow
    }
}
```

В этом примере ***Overflow*** обрабатывается специальным образом. Все остальные исключения ***Matherr*** будут обрабатываться вместе.

Организация исключений в виде иерархий может иметь большое значения для надежности кода. В качестве примера рассмотрим, как бы вы могли обрабатывать все исключения в библиотеке математических функций без такого механизма группировки. Это можно сделать путем утомительного перечисления исключений:

```
void g()
{
    try {
        // ...
    }
    catch (Overflow) { /* ... */ };
    catch (Underflow) { /* ... */ };
    catch (Zerodivide) { /* ... */ },
}
```

Это не только утомительно. Программист легко может забыть указать исключение в этом списке. Рассмотрим, что бы нам потребовалось, если бы мы не сгруппировали математические исключения. При введении нового исключения в математи-

ческой библиотеке каждый фрагмент кода, который пытается обрабатывать все математические исключения, подлежал бы модификации. Как правило, такие глобальные модификации неприемлемы после выхода начального релиза библиотеки. Часто не существует способа найти каждый затрагиваемый изменением фрагмент кода. Даже если такой способ и существует, мы, в общем случае, не можем предполагать, что все фрагменты исходного кода нам доступны или что мы захотели бы внести необходимые изменения, будь они доступны. Многочисленные перекомпиляции и проблемы сопровождения заставят отказаться от добавления новых исключений в библиотеку после выхода первого релиза. Это *неприемлемо* почти для всех библиотек. Перечисленные выше причины приводят к определению исключений в виде иерархий классов на уровне библиотеки или на уровне подсистемы (§ 14.6.2).

Обратите внимание на то, что ни встроенные математические операции, ни базовая математическая библиотека (совместно используемая с C) не сообщают об арифметических ошибках в форме исключений. Одной из причин является то, что обнаружение некоторых арифметических ошибок, таких как деление на ноль, является асинхронным на многих машинах с конвейерной архитектурой. Приведенная здесь иерархия ***Matherr*** является просто иллюстрацией. Исключения стандартной библиотеки описаны в § 14.10.

14.2.1. Производные исключения

Использование иерархий классов для обработки исключений естественным образом приводит к обработчикам, интересующимся лишь подмножеством информации, которую несут с собой исключения. Другими словами, исключение обычно перехватывается обработчиком его базового класса, а не обработчиком его собственного класса. Семантика перехвата и задания имен исключений идентична семантике функции с аргументом. То есть формальный аргумент инициализируется значением аргумента (§ 7.2). Из этого следует, что сгенерированное исключение «срезается» до перехваченного (§ 12.2.3). Например:

```
class Matherr {
        // ...
        virtual void debug_print() const
                cerr << "Математическая ошибка"; }
};

class Int_overflow : public Matherr {
        const char* op;
        int a1, a2;
public:
        Int_overflow(const char* p, int a, int b) { op = p; a1 = a; a2 = b; }
        virtual void debug_print() const
                { cerr << op << '(' << a1 << ',' << a2 << ')'; }
        // ...
};

void f()
{
```

```
        try {
                g ();
        }
        catch (Matherr m) {
                // ...
        }
}
```

Когда вызывается обработчик ***Matherr***, ***m*** является объектом ***Matherr*** — даже если вызов *g* () привел к генерации ***Int_overflow***. Это означает, что дополнительная информация, имеющаяся в ***Int_overflow***, недоступна.

Как всегда, можно использовать указатели или ссылки во избежание потери информации. Мы могли бы, например, написать:

```
int add (int x, int y)
{
        if ( (x>0 && y>0 && x>INT_MAX - y) || (x<0 && y<0 && x<INT_MIN - y))
                throw Int_overflow ("+", x, y);

        return x+y;             // в результате x+y не произойдет переполнение
}

void f ()
{
        try {
                int i1 = add (1, 2);
                int i2 = add (INT_MAX, -2);
                int i3 = add (INT_MAX, 2);          // Приехали!
        }
        catch (Matherr& m) {
                // ...
                m.debug_print ();
        }
}
```

Последний вызов ***add*** () приведет к исключению, которое вызовет ***Int_overflow::debug_print*** (). Если бы исключение перехватывалось по значению, а не по ссылке, была бы вызвана функция ***Matherr::debug_print*** ().

14.2.2. Композиция исключений

Не каждая группа исключений является древообразной структурой. Довольно часто исключение принадлежит сразу двум группам. Например:

```
// ошибка, связанная с файлом в сети
class Netfile_err : public Network_err, public File_system_err { /* ... */ };
```

Netfile_err может перехватываться функциями, работающими с исключениями в сети:

```
void f ()
{
        try {
                // что-нибудь
        }
```

```
        catch (Network_err& e) {
            //...
        }
    }
```

а также функциями, работающими с исключениями файловой системы:

```
    void g()
    {
        try {
            //...
        }
        catch (File_system_err& e) {
            //...
        }
    }
```

Такая неиерархическая организация обработки ошибок имеет большое значение в тех случаях, когда службы (например, сетевые) прозрачны для пользователя. В нашем примере автор ***g*** () мог и не подозревать о существовании сети (см. также § 14.6).

14.3. Перехват исключений

Рассмотрим пример:

```
    void f()
    {
        try {
            throw E();
        }
        catch (H) {
            // когда мы сюда попадем?
        }
    }
```

Обработчик будет вызван:

[1] Если ***H*** того же типа, что и ***E***.
[2] Если ***H*** является однозначной открытой базой ***E***.
[3] Если ***H*** и ***E*** являются указателями, и [1] или [2] выполняется для типов, на которые они ссылаются.
[4] Если ***H*** является ссылкой, и [1] или [2] выполняется для типа, на который ***H*** ссылается.

Кроме того, мы можем добавить модификатор ***const*** к типу, используемому для перехвата исключения, точно так же, как мы можем добавить его к аргументу функции. Это не изменит набор исключений, который мы сможем перехватить, а только воспрепятствует модификации перехваченного исключения.

В принципе, исключение в момент его генерации копируется, поэтому обработчик имеет копию исходного исключения. В действительности, исключение может скопироваться несколько раз до того, как оно будет перехвачено. Следовательно, мы не можем сгенерировать исключение, которое не может быть скопировано. Реализа-

циям позволительно применять широкий набор стратегий хранения и передачи исключений. Однако гарантируется, что оператору ***new*** всегда хватит памяти для генерации стандартного исключения нехватки памяти ***bad_alloc*** (§ 14.4.5).

14.3.1. Повторная генерация

Перехватив исключение, обработчик часто решает, что он не может полностью обработать ошибку. В этом случае обработчик делает то, что может, после чего вновь генерирует исключение. Таким образом, ошибка может быть обработана в наиболее подходящем месте. Это происходит даже в том случае, когда информация, требуемая для корректной обработки ошибки, не сосредоточена в каком-то одном месте, так что действия по восстановлению лучше распределить по нескольким обработчикам. Например:

```
void h()
{
    try {
        // код, который может привести к математическим ошибкам
    }
    catch (Matherr) {
        if (can_handle_it_completely) {
            // в состоянии полностью обработать ошибку?
            // обработка Matherr

            return;
        }
        else {
            // делается то, что можно сделать здесь

            throw;      // повторная генерация исключения
        }
    }
}
```

Факт повторной генерации отмечается отсутствием операнда у ***throw***. Если осуществлена попытка повторной генерации при отсутствии исключения, будет вызвана функция ***terminate*** () (§ 14.7). Компилятор может обнаружить некоторые такие случаи, но не все.

Повторно генерируемое исключение является исходным исключением, а не просто его частью, которая была доступна как ***Matherr***. Другими словами, если было сгенерировано ***Int_overflow***, то функция, вызвавшая ***h*** (), по-прежнему может перехватить ***Int_overflow***, которое было перехвачено в ***h*** () в виде ***Matherr*** и повторно сгенерировано.

14.3.2. Перехват всех исключений

Описываемая ниже вырожденная форма техники ***catch-throw*** может иметь большое значение. Также как и в функциях, многоточие ... означает «любой аргумент» (§ 7.6), поэтому ***catch*** (...) означает «перехват всех исключений». Например:

```
void m()
{
    try {
```

```
            // что-нибудь
        }
        catch (...) {       // обработка всех исключений
            // очистка
            throw;
        }
}
```

То есть если исключение возникает в результате выполнения главной части *m* (), вызывается часть обработчика, осуществляющая очистку. После того, как локальная очистка завершена, исключение, приведшее к ней, повторно генерируется для дальнейшей обработки ошибки. См. в § 14.6.3.2 описание метода получения информации об исключения, перехваченной обработчиком с ....

Одним важным аспектом обработки ошибок вообще и обработки исключений в частности является поддержка инвариантов, предполагаемых программой (§ 24.3.7.1). Например, если предполагается, что функция *m* () оставляет определенные указатели в том состоянии, в котором она их нашла, то мы можем написать в обработчике код, задающий им приемлемые значения. Таким образом, обработчик перехватывающий все исключения, можно использовать для поддержки произвольных инвариантов. Однако во многих важных случаях такой обработчик является не самым элегантным решением данной проблемы (см. § 14.4).

14.3.2.1. Порядок записи обработчиков

Ввиду того, что производные исключения могут быть перехвачены обработчиками, предназначенными для более чем одного вида исключений, порядок, в котором записаны обработчики в инструкции *try*, имеет большое значение. Эти обработчики проверяются по порядку. Например:

```
void f()
{
    try {
        // ...
    }
    catch (std::ios_base::failure) {
        // обработка любой ошибки в потоке ввода/вывода (§ 14.10)
    }
    catch (std::exception& e) {
        // обработка любого исключения стандартной библиотеки (§ 14.10)
    }
    catch (...) {
        // обработка любого другого исключения (§ 14.3.2)
    }
}
```

Так как компилятор знает иерархию классов, он может обнаружить многие логические ошибки. Например:

```
void g()
{
    try {
```

```
            // ...
        }
        catch (...) {
            // обработка любого другого исключения (§ 14.3.2)
        }
        catch (std::exception& e) {
            // обработка любого исключения стандартной библиотеки (§ 14.10)
        }
        catch (std..bad_cast) {
            // обработка сбоя dynamic_cast (§ 15.4.2)
        }
}
```

В этом примере вариант с ***exception*** никогда не будет рассматриваться. Даже если бы мы убрали обработчик, «перехватывающий все», ***bad_cast*** никогда бы не рассматривался, потому что он является производным от ***exception***.

14.4. Управление ресурсами

В тех случаях, когда функции требуется некоторый ресурс (например, функция открывает файл, выделяет свободную память, блокирует доступ к ресурсу), для дальнейшего функционирования системы очень важно, чтобы ресурс был правильно освобожден. Довольно часто это «правильное освобождение» ресурса осуществляется той же самой функцией, которая его затребовала, перед возвратом. Например:

```
void use_file (const char* fn)
{
    FILE* f = fopen (fn, "w");
    // использование f
    fclose (f);
}
```

Это выглядит приемлемо до тех пор, пока вы не поймете, что если что-нибудь произойдет после вызова ***fopen*** () и до вызова ***fclose*** (), исключение может привести к выходу из ***use_file*** () без вызова ***fclose*** (). Точно такая же проблема существует и в языках, не поддерживающих обработку исключений. Например, функция стандартной библиотеки C ***longjmp*** () может привести к подобной ситуации. Даже обыкновенная инструкция ***return*** может завершить ***use_file*** без закрытия *f*.

Первая попытка сделать функцию ***use_file*** () устойчивой к ошибкам выглядит следующим образом:

```
void use_file (const char* fn)
{
    FILE* f = fopen (fn, "r");
    try {
        // использование f
    }
    catch ( ...) {
        fclose (f);
        throw;
```

```
        }
        fclose ();
}
```

Код, использующий файл, заключен в блок ***try***. Соответствующий блок ***catch*** перехватывает все исключения, закрывает файл и повторно генерирует исключение.

Проблема с этим решением состоит в том, что оно очень многословное, утомительное и потенциально дорогостоящее. Более того, любое многословное и утомительное решение подвержено ошибкам, потому что возня с ним надоедает программисту. К счастью, существует более элегантное решение. Общая постановка проблемы выглядит следующим образом:

```
void acquire ()          // выделение ресурсов
{
        // выделение ресурса 1
        // ...
        // выделение ресурса n

        // использование ресурсов

        // освобождение ресурса n
        // ...
        // освобождение ресурса 1
}
```

Часто бывает важно, чтобы ресурсы освобождались в порядке обратном тому, в котором они были выделены. Это напоминает поведение локальных объектов, создаваемых конструкторами и уничтожаемых деструкторами. Следовательно, мы можем решить проблемы выделения и освобождения ресурсов при помощи объектов классов с конструкторами и деструкторами. Например, мы можем определить класс ***File_ptr***, который ведет себя наподобие ***FILE****:

```
class File_ptr {
        FILE* p;
public:
        File_ptr (const char* n, const char* a) { p = fopen (n, a); }
        File_ptr (FILE* pp) { p = pp; }
        ~File_ptr () { if (p) fclose (p); }          // допустимые операции копирования

        operator FILE* () { return p; }
};
```

Мы можем создать ***File_ptr*** либо при наличии ***FILE****, либо по аргументам, требуемым для ***fopen*** (). В любом случае, ***File_ptr*** будет уничтожен в конце его области видимости, и его деструктор закроет файл. Наша функция сократилась теперь до минимума:

```
void use_file (const char* fn)
{
        File_ptr f (fn, "r");
        // использование f
}
```

Деструктор будет вызван независимо от того, завершится ли функция нормально или по исключению. То есть механизмы обработки исключений позволяют нам уда-

лить код обработки ошибки из основного алгоритма. Результирующий код проще и менее подвержен ошибкам, чем его традиционный вариант.

Процесс просмотра стека и поиска обработчика для исключения обычно называют «раскручиванием стека» (stack unwinding). По мере раскручивания вызываются деструкторы созданных локальных объектов.

14.4.1. Использование конструкторов и деструкторов

Техника управления ресурсами с использованием локальных объектов обычно называется принципом «выделение ресурса есть инициализация». Это общая техника, она полагается на свойства конструкторов и деструкторов и на их взаимодействие с обработкой исключений.

Объект не считается созданным до тех пор, пока не завершится выполнение его конструктора. После этого и только после этого механизм раскручивания стека вызовет деструктор для объекта. Процесс конструирования объекта, содержащего в себе другие объекты, продолжается до тех пор, пока не будут сконструированы все входящие объекты. Массив создается до тех пор, пока не будут созданы его элементы (и только полностью сконструированные элементы уничтожаются в процессе раскручивания стека).

Конструктор пытается обеспечить полное и корректное создание объекта. Когда это невозможно, хорошо написанный конструктор восстанавливает — настолько, насколько это возможно — состояние системы до конструирования объекта. В идеале, конструктор всегда достигает одной из этих альтернатив и не оставляет объекты в некотором «наполовину созданном» состоянии. Этого можно достигнуть при помощи принципа «выделение ресурса есть инициализация», применяемого к членам.

Рассмотрим класс ***X***, для которого конструктор должен затребовать два ресурса: файл ***x*** и блокировку ***y***. Процесс «оприходования» может завершиться неуспешно и сгенерировать исключение. Конструктор класса ***X*** не должен завершиться, открыв файл, но не получив блокировку. Более того, выполнение этого требования должно быть достигнуто без взваливания соответствующих проблем на программиста. Мы используем объекты двух классов, ***File_ptr*** и ***Lock_ptr***, для представления требуемых ресурсов. Выделение ресурса происходит в виде инициализации локального объекта, представляющего ресурс:

```
class X{
        File_ptr aa;
        Lock_ptr bb,
public.
        X(const char* x, const char* y)
                : aa(x, "rw"),          // берем файл x
                : bb(y)                 // берем блокировку y
        {}
        //...
};
```

Теперь, как и в случае с локальным объектом, реализация может взять на себя все заботы об учете. Пользователю совершенно не требуется отслеживать этот процесс. Например, если исключение возникнет после создания ***aa***, но до конструирования ***bb***, будет вызван деструктор для ***aa***, но не для ***bb***.

Итак, там, где годится подобная простая модель выделения ресурсов, автору конструктора нет необходимости писать явный код обработки исключений.

Наиболее часто ресурсом, запрашиваемым разного рода способами, является память. Например:

```
class Y {
        int* p;
        void init ();
public:
        Y (int s) { p = new int[s]; init (); }
        ~Y () { delete[] p; }
        // ...
};
```

Такая практика является распространенной и ведет к утечке памяти. Если в *init* () будет сгенерировано исключение, затребованная память не будет освобождена — деструктор не будет вызван, потому что объект сконструирован не полностью. Более безопасным вариантом будет:

```
class Z {
        vector<int> p;
        void init ();
public:
        Z (int s) : p (s) { init (); }
        // ...
};
```

Память, выделенная для *p*, теперь управляется в *vector*. Если *init* () сгенерирует исключение, захваченная память будет освобождена, когда вызовется (неявно) деструктор для *p*.

14.4.2. auto_ptr

Стандартная библиотека предоставляет шаблон класса *auto_ptr*, поддерживающий технику «выделение ресурса есть инициализация». Как правило, *auto_ptr* инициализируется указателем и может быть разыменован аналогично указателю. Кроме того, объект, на который он указывает, будет неявно удален в конце области видимости *auto_ptr*. Например:

```
// не забыть удалить pb при выходе
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb)
{
        auto_ptr<Shape> p (new Rectangle (p1, p2));    // p указывает на прямоугольник
        auto_ptr<Shape> pbox (pb);

        p->rotate (45);          // использует auto_ptr<Shape> точно так же, как Shape*
        // ...
        if (in_a_mess) throw Mess ();
        // ...
}
```

В нашем примере *Rectangle*, указатель *pb* на *Shape* и указатель *pc* на *Circle* будут удалены независимо от того, будет ли сгенерировано исключение.

Для достижения этой *семантики владения* (называемой также *семантикой деструктивного копирования*) шаблоны *auto_ptr* имеют семантику копирования, которая радикально отличается от семантики копирования обычных указателей: если один указатель типа *auto_ptr* копируется в другой, то исходный указатель после этого ни на что не указывает. Поскольку копирование переменной типа *auto_ptr* изменяет ее саму, то переменная типа *const auto_ptr* не может быть скопирована.

Шаблон *auto_ptr* объявлен в <*memory*> и может быть описан следующим образом:

```
template<class X> class Std::auto_ptr {
        template<class Y> struct auto_ptr_ref {/* ... */};      // вспомогательный класс
        X* ptr;
public:
        typedef X element_type;

        // throw() означает «ничего не генерировать»; см. § 14.6
        explicit auto_ptr (X* p=0) throw () {ptr=p;}
        ~auto_ptr () throw () {delete ptr;}

        // отметим, что копирующие конструкторы и присваивания
        // получают неконстантные аргументы:
        auto_ptr (auto_ptr&a) throw ();                         // копирование, далее a.ptr=0
        template<class Y> auto_ptr (auto_ptr<Y>a) throw ();     // копирование, далее a.ptr=0
        auto_ptr& operator=(auto_ptr& a) throw ();              // копирование, далее a.ptr=0
        // копирование, далее a.ptr=0
        template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw ();

        X& operator* () const throw () {return *ptr;}
        X* operator-> () const throw () {return ptr;}
        X* get () const throw () {return ptr;}                  // получаем указатель
        X* release () throw () {X* t=ptr; ptr=0; return t;}     // передача права собственности
        void reset (X* p=0) throw () {if (p!=ptr) {delete ptr; ptr=p;}}

        auto_ptr_ref<X>) throw () ;                             // копирует из auto_ptr_ref
        template<class Y> operator auto_ptr_ref<Y>() throw ();  // копирует в auto_ptr_ref
        // деструктивное копирование из auto_ptr
        template<class Y> operator auto_ptr<Y>() throw ();
};
```

Назначение *auto_ptr_ref* — реализовать семантику деструктивного копирования для обычного типа *auto_ptr*, делая невозможным копирование ***const auto_ptr***. Если ***D**** может быть преобразован в ***B****, то шаблонный конструктор и шаблонное присваивание могут (явно или неявно) преобразовывать ***auto_ptr<D>*** в ***auto_ptr<B>***. Например:

```
void g (Circle* pc)
{
        auto_ptr<Circle> p2 (pc);       // p2 несет ответственность за удаление
        auto_ptr<Circle> p3 (p2);       // теперь p3 несет ответственность
                                        // за удаление (а p2 — нет)
        p2->m= 7;                       // ошибка программиста: p2.get()==0
        Shape* ps= p3.get();            // получаем указатель от auto_ptr
        auto_ptr<Shape>aps (p3);        // передача владения и преобразование типа
        auto_ptr<Circle>p4 (pc);        // ошибка программиста: теперь и p4 несет
                                        // ответственность за удаление
}
```

Эффект «принадлежности» объекта более чем одному ***auto_ptr*** не определен; наиболее вероятно — объект будет удален дважды (с печальными последствиями).

Отметим: семантика деструктивного копирования для ***auto_ptr*** означает, что он не удовлетворяет требованиям к элементам стандартного контейнера или требованиям стандартных алгоритмов, подобных ***sort***(). Например:

```
vector< auto_ptr <Shape>> & v;      // опасно: использует auto_ptr в контейнере
// ...
```

```
sort(v.begin(), v.end());          // так не делать: сортировка, возможно, испортит v
```

Ясно, что ***auto_ptr*** является общим «интеллектуальным» (smart) указателем. Однако он обеспечивает тот сервис, для которого он был разработан — безопасность обработки исключений.

14.4.3. Предостережение

Не все программы должны обладать гибкостью по отношению к любым ошибкам и не все ресурсы являются настолько критичными, чтобы оправдать усилия, требуемые при использовании принципа «выделение ресурса есть инициализация», ***auto_ptr*** и ***catch*** (...). Например, в большинстве программ, которые просто читают из потока ввода и завершают свое выполнение, наиболее подходящей реакцией на серьезную ошибку во время выполнения будет прекращение процесса (после вывода подходящего сообщения). При этом система освободит все выделенные ресурсы, а пользователь может запустить программу заново с подходящим набором входных параметров. Обсуждаемая здесь стратегия предназначена для приложений, в которых такая упрощенная реакция на ошибки времени выполнения неприемлема. В частности, разработчик библиотеки, как правило, не может сделать разумных предположений о требованиях устойчивости к ошибкам в программе, использующей библиотеку, и поэтому вынужден избегать безусловного прекращения выполнения программы и освобождать все ресурсы перед возвратом в вызывающую программу. Стратегия «выделение ресурса есть инициализация» совместно с использованием исключений для сообщений об ошибке в большинстве случаев подходит для написания таких библиотек.

14.4.4. Исключения и оператор new

Рассмотрим пример:

```
void f(Arena& a, X* buffer)
{
    X* p1 = new X;
    X* p2 = new X[10];
    X* p3 = new (&buffer[10]) X,        // поместить X в буфер — не требуется
                                        // освобождения памяти
    X* p4 = new (&buffer[11]) X[10];
    X* p5 = new (a) X;                  // выделение памяти из «арены» a
                                        // (освобождение из a)
    X* p6 = new (a) X[10];
}
```

Что произойдет, если конструктор ***X*** сгенерирует исключение? Освобождается ли память, выделенная ***operator new*** ()? В обычном случае ответ положителен, поэтому инициализация ***p1*** и ***p2*** не вызывают утечки памяти.

Когда используется синтаксис размещения (§ 10.4.11), ответ не может быть таким простым. При некотором использовании этого синтаксиса выделенная память затем освобождается, при некотором — нет. Более того, целью синтаксиса размещения является использование нестандартного выделения памяти, поэтому обычно требуется и нестандартное освобождение. Следовательно, предпринимаемые действия зависят от исполь-

зованного оператора выделения памяти. Если использовался оператор ***Z::operator new*** (), то при выходе применяется ***Z::operator delete*** (); в противном случае не производится попытка освобождения памяти. Массивы обрабатываются аналогичным образом (§ 15.6.1). Эта стратегия корректно работает в случае с оператором размещения стандартной библиотеки ***new*** (§ 10.4.11), также как и в любом другом случае, когда программист реализовал комплиментарную пару функций выделения/освобождения.

14.4.5. Исчерпание ресурсов

Периодически возникает вопрос: что делать, когда попытка получения ресурса завершилась неуспешно? Например, мы жизнерадостно открыли файлы (используя ***fopen*** ()) и затребовали память в куче (используя ***operator new***), не побеспокоившись о том, что будет, если файла не оказалось на месте или нет свободной памяти. Встретившись с такими проблемами, программисты используют решения двух типов:

Возобновление: Попросить вызвавшую функцию устранить проблему и продолжить.

Завершение: Прекратить выполнение и возвратиться в вызвавшую программу.

В первом случае вызывающая функция должна быть готова оказать помощь при возникновении (в неизвестном фрагменте кода) проблемы с выделением ресурсов. Во втором — вызывающая функция должна быть готова к провалу попытки выделения ресурса. Второй вариант в большинстве случаев намного проще и позволяет обеспечить лучшее разделение уровней абстракции. Обратите внимание, что при использовании второго варианта, прекращается выполнение не всей программы, а только конкретного фрагмента. «Завершение» — традиционный термин, означающий стратегию возврата из места, где вычисления закончились неуспешно, в обработчик ошибок, связанный с вызывающей функцией (которая может попытаться заново вызвать неуспешно завершившиеся вычисление), а не попытку устранения проблемы и возобновление выполнения с места, в котором обнаружена проблема.

В C++ модель возобновления поддерживается механизмом вызова функций, а модель завершения — механизмом обработки исключений. Обе модели можно продемонстрировать на простом примере реализации и использования стандартного библиотечного ***operator new*** ():

```
void* operator new (size_t size)
{
    for (;;) {
        if (void* p = malloc (size)) return p;          // попытка найти память
        if (_new_handler == 0) throw bad_alloc ();      // нет обработчика —
                                                        // прекратить
        _new_handler ();                                // обратиться за помощью
    }
}
```

В этом примере для поиска свободной памяти я использовал стандартную функцию библиотеки C ***malloc*** (); другие реализации ***operator new*** () могут выбрать иной подход. Если память найдена, ***operator new*** () возвращает указатель на нее. В противном случае ***operator new*** () вызывает ***_new_handler***. Если ***_new_handler*** может где-то найти память для функции ***malloc*** () — все прекрасно. В противном случае обработчик не может возвратиться в ***operator new*** (), не приводя при этом к бесконечному циклу. В результате ***_new_handler*** () может сгенерировать исключение, предоставив вызывающей программе разбираться с ситуацией:

```
void my_new_handler ()
{
        int no_of_bytes_found = find_some_memory ();
        if (no_of_bytes_found < min_allocation)
                throw bad_alloc (),      // сдаюсь
}
```

Где-то должен быть блок **try** с соответствующим обработчиком:

```
try {
        //...
}
catch (bad_alloc) {
        // реакция на исчерпание памяти
}
```

В реализации **operator new** () используется **_new_handler**, который является указателем на функцию, задаваемую стандартной функцией **set_new_handler** (). Если я хочу, чтобы **my_new_handler** () использовался в качестве **_new_handler**, я могу написать:

```
set_new_handler (&my_new_handler);
```

Если, кроме того, я хочу перехватывать **bad_alloc**, я могу написать:

```
void f ()
{
        void (*oldnh) () = set_new_handler (&my_new_handler),
        try {
                //...
        }
        catch (bad_alloc) {
                //...
        }
        catch ( ... ){
                set_new_handler (oldnh),     // переустановить обработчик
                throw,                       // повторная генерация
        }
        set_new_handler (oldnh);             // переустановить обработчик
}
```

А еще лучше вместо **catch** (...) применить технику «выделение ресурса есть инициализация», описанную в § 14.4, к **_new_handler** (§ 14.12[1]).

При использовании **_new_handler** не передается никакой дополнительной информации из места обнаружения ошибки в функцию-помощник. Можно довольно просто передать больше информации. Однако чем больше информации передается между кодом, обнаружившим ошибку времени выполнения, и функцией, помогающей ее устранить, тем больше два фрагмента кода начинают зависеть друг от друга. Из этого следует, что изменения одного фрагмента кода требуют понимания или может быть даже изменения другого. Для того чтобы отдельные фрагменты кода программы были действительно отдельными, лучше сводить к минимуму подобные зависимости. Механизм обработки исключений лучше поддерживает такое разделение, чем вызовы функций-помощников, предоставляемых вызывающей процедурой.

Как правило, мудрым решением является разбиение выделения ресурсов на слои (уровни абстракции) и устранение зависимости одного слоя от помощи со стороны

слоя, который его вызвал. Практический опыт создания крупных систем показывает, что успешно работающие системы развиваются в этом направлении.

Для генерации исключения требуется объект. К реализации C++ предъявляется требование: объем свободной памяти всегда должен быть достаточен для генерации ***bad_alloc*** в случае исчерпания памяти. Однако вполне возможно, что генерация некоторых других исключений приведет к исчерпанию памяти.

14.4.6. Исключения в конструкторах

Исключения предоставляют способ решить проблему: как сообщить об ошибке из конструктора. Ввиду того, что конструктор не возвращает отдельного значения, которое вызывающая функция могла бы проверить, традиционными (то есть без обработки исключений) альтернативами остаются:

[1] Возвратить объект в «неправильном» состоянии и полагаться на то, что пользователь проверит его состояние.

[2] Присвоить значение нелокальной переменной (например, ***errno***) для указания на неуспешное создание объекта и полагаться на то, что пользователь его проверит.

[3] Не осуществлять никакой инициализации в конструкторе и полагаться на то, что пользователь вызовет функцию инициализации до первого использования.

[4] Пометить объект как неинициализированный и при первом вызове функции-члена для этого объекта осуществить инициализацию (такая функция — не конструктор — может вернуть сообщение об ошибке в случае неуспешной инициализации).

Обработка исключений позволяет передать информацию о неуспешной инициализации из конструктора. Например, простой класс ***Vector*** мог бы защититься от запроса слишком большого количества памяти следующим образом:

```
class Vector {
public:
    class Size { },
    enum { max = 32000 };
    Vector (int sz)
    {
        if (sz < 0 || max < sz) throw Size ();
        //...
    }
    //...
};
```

Код, создающий вектора, теперь может перехватывать ошибки ***Vector::Size***, и мы можем попытаться сделать с ними что-нибудь осмысленное:

```
Vector* f (int i)
{
    try {
        Vector* p = new Vector (i);
        //...
        return p;
    }
    catch (Vector::Size) {
        // обработка ошибки размера вектора
    }
}
```

Естественно, сам обработчик ошибок может использовать стандартный набор основных способов выдачи сообщений об ошибке и восстановления. Каждый раз, когда исключение передается в вызывающую функцию, взгляд на то, что пошло неправильно, меняется. Если с исключением передается подходящая информация, количество информации, имеющей отношения к проблеме, может увеличиться. Другими словами, фундаментальной целью техники обработки исключений является передача информации об ошибке из точки ее обнаружения в место, где имеется достаточно информации для восстановления после возникновения проблемы и для осуществления этого надежным и удобным способом.

Техника «выделение ресурса есть инициализация» является самым безопасным и элегантным способом обработки ошибок в конструкторах, которые нуждаются более чем в одном ресурсе (§ 14.4). По существу, эта техника сводит проблему работы с несколькими ресурсами к повторному применению (простой) техники работы с одним ресурсом.

14.4.6.1. Исключения и инициализация членов

Что происходит, когда код, инициализирующий член (непосредственно или косвенно), генерирует исключение? По умолчанию исключение передается туда, где вызван конструктор для класса этого члена. Однако сам конструктор может перехватывать такие исключения, помещая все тело функции — включая список инициализаторов членов — в блок ***try***. Например:

```
class X{
        Vector v;
        //...
public:
        X(int);
        //...
};

X::X(int s)
try
        :v(s)                   // инициализация v при помощи s
{
        //...
}
catch (Vector::Size){           // здесь перехватываются исключения,
                                // сгенерированные при инициализации v
        //...
}
```

14.4.6.2. Исключения и копирование

Подобно другим конструкторам, копирующий конструктор может сигнализировать об отказе посредством исключения. В этом случае объект не создается. Например, копирующему конструктору вектора часто необходимо выделять память и копировать элементы (§ 16.3.4, § Д.3.2), и это может вызвать генерацию исключения. Перед генерацией исключения копирующему конструктору необходимо освободить все занятые им ресурсы. Детальное обсуждение обработки исключений и управления ресурсами для контейнеров приведено в § Д.2 и § Д.3.

Копирующее присваивание похоже на копирующий конструктор тем, что оно может иметь захваченные ресурсы и может закончиться генерацией исключения. Перед генерацией исключения прсваивание должно убедиться, что оба его операнда находятся в корректном состоянии. С другой стороны, могут быть нарушены требования стандартной библиотеки, что даст в результате непредсказуемое поведение (§ Д.2, § Д.3.3).

14.4.7. Исключения в деструкторах

С точки зрения обработки исключений деструктор может вызываться одним из двух способов:

[1] *Нормальный вызов:* в результате нормального выхода имени из области видимости (§ 10.4.3), использования оператора ***delete*** (§ 10.4.5) и т. д.

[2] *Вызов в процессе обработки исключения:* в процессе раскручивания стека (§ 14.4) механизм обработки исключения приводит к выходу из области видимости, содержащей объект с деструктором.

Во втором случае исключение не должно покинуть сам деструктор. Если все-таки покинет, это считается ошибкой механизма обработки исключений и вызывается ***std::terminate*** () (§ 14.7). Все же не существует общего способа определить, в праве ли механизм обработки исключений или деструктор проигнорировать одно исключение ради обработки другого. Выход из деструктора через генерацию исключения также является нарушением требований стандартной библиотеки (§ Д.2).

Деструктор в состоянии защитить себя, если он вызывает функции, которые могут сгенерировать исключения. Например:

```
X ~X()
try {
        f(),                    // может сгенерировать исключение
}
catch ( ){
        // некоторые действия
}
```

Функция стандартной библиотеки ***uncaught_exception*** () возвращает ***true***, если было сгенерировано исключение, которое еще не перехвачено. Она позволяет программисту определять различные действия в деструкторе в зависимости от того, уничтожен объект нормально или в процессе раскручивания стека.

14.5. Исключения, не являющиеся ошибками

Если исключение ожидается и перехвачено таким образом, что не оказывает отрицательного воздействия на поведение программы, почему это может считаться ошибкой? Только потому, что программист думает о такой ситуации как об ошибке, а о механизме обработки исключений — как о способе отреагировать на ошибку. С другой стороны, можно рассматривать механизмы обработки исключений, как просто еще одну управляющую структуру. Например:

```
void f(Queue<X>& q)
{
        try {
                for (;;){
```

```
                X m = q.get();          // генерирует Empty,
                                        // если очередь пуста
                //...
            }
        }
        catch (Queue<X>::Empty) {
            return;
        }
    }
```

Такое решение в самом деле имеет некоторое очарование, и это как раз тот случай, когда не совсем понятно, что нужно считать ошибкой, а что — нет.

Обработка исключений является менее структурированным механизмом, чем локальные управляющие структуры, такие как ***if*** и ***for***; к тому же обработка исключений часто менее эффективна, если исключение действительно генерируется. Поэтому исключениями следует пользоваться в тех случаях, когда традиционные управляющие структуры являются неэлегантным решением, или ими невозможно воспользоваться. Обратите внимание, что стандартная библиотека предлагает очередь из произвольных элементов ***queue*** без использования исключений (§ 17.3.2).

Исключения в качестве альтернативного механизма возврата могут оказаться элегантным методом завершения функций поиска, особенно глубоко рекурсивных, таких как поиск по дереву. Например:

```
void fnd(Tree* p, const string& s)
{
    if (s == p->str) throw p;           // найдено
    if (p->left) fnd(p->left, s);
    if (p->right) fnd(p->right, s);
}

Tree* find(Tree* p, const string& s)
{
    try {
        fnd(p, s);
    }
    catch (Tree* q) {                   // q->str == s
        return q;
    }
    return 0;
}
```

Однако неумеренное употребление исключений ведет к непонятному коду. Всегда, когда это разумно, следует придерживаться точки зрения «обработка исключений является обработкой ошибок». При таком подходе код оказывается понятным образом разделен на две категории: обыкновенный код и код обработки ошибок. Программа становится более осмысленной. К сожалению, реальный мир не разлагается на части также просто. Организация программ будет (и до некоторой степени должна) отражать этот факт.

Обработка ошибок традиционно сложна. Следует ценить все, что помогает реализовать понятную модель ошибки и предоставляет способ ее обработки.

14.6. Спецификации исключений

Генерация и перехват исключений изменяют способ взаимодействия между функциями. Поэтому небесполезно указать (как часть объявления) набор исключений, которые могут быть сгенерированы функцией. Например:

```
void fu (int a) throw (x2, x3);
```

Такое объявление означает, что *f*() может сгенерировать только исключения ***x2***, ***x3*** и исключения, являющиеся производными от этих типов, но не другие. Указывая таким способом, какие исключения она может сгенерировать, функция предоставляет пользователям действенные гарантии. Если во время выполнения функция попытается нарушить взятые на себя обязательства, эта попытка будет трансформирована в вызов ***std::unexpected*** (). Смыслом ***unexpected*** () по умолчанию является вызов ***std::terminate*** (), которая в свою очередь вызывает ***abort*** (); подробности см. в § 9.4.1.1.

На самом деле

```
void f() throw (x2, x3)
{
    // нечто
}
```

эквивалентно:

```
void f()
try
{
    // нечто
}
catch (x2) { throw; }          // повторная генерация
catch (x3) { throw; }          // повторная генерация
catch ( ){
    std::unexpected();         // unexpected() не возвратит управление
}
```

Самым важным преимуществом является то, что *объявление* функции принадлежит интерфейсу, который видят те, кто ее вызывает. *Определение* функции, с другой стороны, не всегда доступно. Даже когда у нас есть доступ к исходному коду всех наших библиотек, мы предпочитаем не заглядывать в него слишком часто. Кроме того, функция со *спецификацией исключений* короче и проще, чем приведенная выше эквивалентная версия, написанная вручную.

Предполагается, что функция, объявленная без спецификации исключений, может сгенерировать любое исключение. Например:

```
int f();                       // может сгенерировать любое исключение
```

Функцию, которая не генерирует исключений, можно объявить с пустым списком:

```
int g() throw();               // не генерирует исключений
```

Можно предложить, чтобы умолчанием был запрет на генерацию исключений. Такое правило, однако, может потребовать спецификации исключений практически для каждой функции; оно станет причиной частых перекомпиляций и воспрепятствует взаимодействию с программами, написанными на других языках. Все это будет сти-

мулировать программистов к «подрыву» механизмов обработки исключений и написанию (не нужного ни для чего другого) подавляющего их кода. А это, в свою очередь, создаст ложное чувство безопасности у людей, не заметивших подобной «подрывной» деятельности.

14.6.1. Проверка спецификаций исключений

На этапе компиляции невозможно обнаружить все случаи нарушения спецификации интерфейса. Тем не менее солидная проверка на этапе компиляции все же осуществляется. О спецификациях исключений можно думать как о предположении, что функция сгенерирует все исключения, которые ей дозволены. Правила проверки спецификаций исключений во время компиляции легко выявляют нелепости.

Если объявление функции содержит спецификацию исключений, то и каждое объявление этой функции (включая определение) должно иметь спецификацию исключений с точно тем же набором типов исключений. Например:

```
int f() throw (std::bad_alloc);

int f()        // ошибка: отсутствует спецификация исключений
{
    //...
}
```

Существенно, что не требуется проверять спецификацию исключений за пределами единицы компиляции. Естественно, компилятор мог бы это сделать, однако для большинства больших и долго живущих систем важно, чтобы этого не делалось — а если и делалось, то чтобы сообщения о серьезной ошибке возникали только в тех случаях, когда нарушения нельзя обнаружить на этапе выполнения.

Смысл состоит в том, чтобы добавление исключения не приводило к необходимости полного обновления связанных спецификаций исключений и перекомпиляции всего потенциально зависящего от этого кода. Тогда система сможет функционировать в частично модифицированном состоянии, полагаясь на динамическое (времени выполнения) обнаружение не ожидавшихся исключений. Это имеет большое значения для сопровождения больших систем, в которых значительные модификации являются очень дорогим удовольствием и не весь исходный код доступен.

Виртуальная функция может быть замещена функцией с не менее ограничительной спецификацией исключений, чем ее собственная. Например:

```
class B {
public:
    virtual void f();                  // может сгенерировать любое исключение
    virtual void g() throw (X, Y);
    virtual void h() throw (X);
};

class D : public B {
    void f() throw (X);                // правильно
    void g() throw (X);                // правильно: D::g() имеет более ограничительную
                                       // спецификацию, чем B::g()
    void h() throw (X, Y);             // ошибка: D::h() имеет менее ограничительную
                                       // спецификацию, чем B::h()
};
```

Это правило соответствует здравому смыслу. Если производный класс генерирует исключение, которое не указано в спецификации исходной функции, вызывающая функция может не ожидать его. С другой стороны, замещенная функция, генерирующая меньше исключений, очевидно подчиняется спецификации исключений замещаемой функции.

Аналогичным образом вы можете присвоить указатель на функцию с более ограничительной спецификацией исключений указателю на функцию с менее ограничительной спецификацией исключений, но не наоборот. Например:

```
void f() throw (X);
void (*pf1)() throw (X, Y) = &f;      // правильно
void (*pf2)() throw () = &f;          // ошибка
```

В частности, вы не можете присвоить указатель на функцию без спецификации исключений указателю на функцию, который ее имеет:

```
void g();                             // может сгенерировать любое исключение
void (*pf3)() throw (X) = &g;         // ошибка
```

Спецификация исключений не является частью типа функции, и *typedef* не может ее содержать. Например:

```
typedef void (*PF)() throw (X);       // ошибка
```

14.6.2. Неожидаемые исключения

Спецификация исключений может привести к вызовам *unexpected* (). Как правило, кроме как на этапе тестирования, такие вызовы нежелательны. От них можно избавится путем тщательной организации исключений и спецификаций интерфейса. С другой стороны, вызовы *unexpected* () можно перехватывать таким образом, что они становятся безвредными.

Все исключения хорошо определенной подсистемы *Y* зачастую являются производными от некоего класса *Yerr*. Например, при наличии определения

```
class Some_Yerr : public Yerr { /* ... */ };
```

функция, объявленная

```
void f() throw (Xerr, Yerr, exception);
```

передаст любое *Yerr* вызывающей функции. В частности, *f* () будет обрабатывать *Some_yerr*, передавая его в вызывающую функцию. Таким образом, никакое исключение *Yerr* в *f* () не запустит *unexpected* ().

Все исключения, генерируемые стандартной библиотекой, являются производными от класса *exception* (§ 14.10).

14.6.3. Отображение исключений

Политика завершения программы при обнаружении неожидаемого исключения иногда оказывается слишком жесткой. В таких случаях поведение *unexpected* () должно быть изменено, чтобы добиться чего-то более приемлемого.

Простейшим способом достижения этой цели является добавление исключения стандартной библиотеки ***std::bad_exception*** в спецификацию исключений. В этом случае ***unexpected*** () вместо вызова функции просто сгенерирует ***bad_exception***. Например:

```
class X{};
class Y{};

void f() throw (X, std::bad_exception)
{
        //...
        throw Y();          // генерация «плохого» исключения
}
```

Спецификация исключений отловит неприемлемое исключение ***Y*** и сгенерирует исключение типа ***bad_exception***.

В ***bad_exception*** нет ничего особенно плохого (bad); оно просто предоставляет механизм, который менее радикален, чем вызов ***terminate*** (). Однако он еще достаточно сырой. В частности, информация о том, какое исключение вызвало проблему, теряется.

14.6.3.1. Отображение исключений пользователем

Рассмотрим функцию ***g*** (), написанную для несетевого окружения. Предположим далее, что ***g*** () была объявлена со спецификацией исключений таким образом, что она может сгенерировать исключения, имеющие отношение только к ее «подсистеме ***Y***»:

```
void g() throw (Yerr);
```

Теперь предположим, что нам потребовалось вызвать ***g*** () в сетевом окружении.

Естественно, ***g*** () не будет ничего знать о сетевых исключениях и вызовет ***unexpected*** (), когда ей встретится одно из них. Для использования ***g*** () в распределенной среде мы должны либо реализовать код, который обрабатывает сетевые исключения, либо переписать ***g*** (). Предполагая, что переписывание является нежелательным или невозможным, мы решим эту проблему, заместив смысл ***unexpected*** ().

Исчерпание памяти обрабатывается ***_new_handler***, задаваемым ***set_new_handler*** (). Аналогично, реакция на неожидаемые исключения определяется ***unexpected_handler***, задаваемым ***set_unexpected*** () из <***exception***>:

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler)();
```

Для того чтобы обрабатывать неожидаемые исключения так, как нас это устраивает, мы сначала определим класс, который позволит нам использовать технику «выделение ресурса есть инициализация» для функции ***unexpected*** ():

```
class STC{        // сохранить и восстановить
        unexpected_handler old;

public:
        STC(unexpected_handler f){ old = set_unexpected(f); }
```

```
        ~STC () { set_unexpected (old); }
};
```

Теперь мы определим функцию с тем смыслом, который хотим для ***unexpected*** () в данном случае:

```
class Yunexpected : public Yerr { };

void throw Y () throw (Yunexpected) { throw Yunexpected (); }
```

Будучи использованной в качестве ***unexpected*** (), функция ***throwY*** () отображает все неожидаемые исключения в ***Yunexpected***.

Наконец, мы предоставляем версию функции ***g*** (), работающую в сетевом окружении:

```
void networked_g () throw (Yerr)
{
        STC xx (&throwY);       // теперь unexpected()
                                // генерирует Yunexpected
        g ();
}
```

Так как ***Yunexpected*** является производным от ***Yerr***, спецификация исключений не нарушается. Если бы ***throwY*** () генерировала исключение, которое нарушает спецификацию исключений, была бы вызвана функция ***terminate*** ().

Сохраняя и восстанавливая ***_unexpected_handler***, мы предоставляем нескольким подсистемам возможность управлять обработкой неожидаемых исключений, не входя в зависимость друг от друга. В основном техника отображения неожидаемых исключений в ожидаемые является более гибким вариантом того, что предлагает система в форме ***bad_exception***.

14.6.3.2. Восстановление типа исключения

Отображение неожидаемых исключений в ***Yunexpected*** позволит пользователю ***networked_g*** () узнать, что некоторое неожидаемое исключение отображено в ***Yunexpected***. Однако пользователь не будет знать, какое именно исключение отображено. Эта информация была потеряна в ***throwY*** (). Простой метод позволяет записать эту информацию и передать ее. Например, мы можем собрать информацию о ***Network_exception*** следующим образом:

```
class Yunexpected : public Yerr {
public:
        Network_exception* pe;
        Yunexpected (Network_exception* p) : pe (p?p->clone() : 0) { }
        ~Yunexpected () {delete p;}
};

void throw Y () throw (Yunexpected)
{
        try {
                throw;                  // сгенерировать заново,
                                        // чтобы немедленно перехватить!
        }
        catch (Network_exception& p) {
                throw Yunexpected (&p); // сгенерировать отображенное исключение
        }
```

```
        catch (...){
            throw Yunexpected(0);
        }
    }
```

Повторная генерация исключения и его перехват позволяют нам создать обработчик исключений любых типов, имена которых мы знаем. Функция ***throwY*** () вызывается из ***unexpected*** (), которая концептуально вызывается из обработчика ***catch*** (...). Поэтому некоторое исключение определенно существует и его можно сгенерировать заново. Функция ***unexpected*** () не может проигнорировать исключение и вернуть управление. Если она попытается, сама ***unexpected*** () сгенерирует ***bad_exception*** (§ 14.6.3). Функция ***clone***() используется для размещения копии исключения в свободной области памяти. Эта копия будет существовать и после того, как обработчик исключений очистит локальные переменные.

14.7. Неперехваченные исключения

Если исключение сгенерировано, но не перехвачено, вызывается функция ***std::terminate*** (). Функция ***terminate*** () будет также вызвана, если механизм обработки исключения обнаружит, что стек разрушен, и если деструктор, вызванный во время обусловленной исключением раскрутки стека, пытается завершить свою работу при помощи исключения.

Неожидаемые исключения обрабатываются ***_unexpected_handler***, задаваемым функцией ***set_unexpected_handler*** (). Подобным образом реакция на неперехваченные исключения определяются ***_uncaught_handler***, задаваемым функцией ***set_terminate*** ():

```
typedef void (*terminate_handler) ();
terminate_handler set_terminate (terminate_handler);
```

Возвращаемым значением является предыдущая функция, переданная ***set_terminate*** ().

Суть ***terminate*** () заключается в том, что вместо обработки исключений иногда следует использовать менее изощренную технику обработки ошибок. Например, ***terminate*** () могла бы использоваться для прекращения процесса или для повторной инициализации системы. Функция ***terminate*** () является радикальным средством, применяемым, когда стратегия восстановления после ошибок, реализованная при помощи механизма обработки исключений, потерпела неудачу, и пришло время перейти на другой уровень борьбы с ошибками.

По умолчанию ***terminate*** () вызовет функцию ***abort*** () (§ 9.4.1.1). Это умолчание является хорошим вариантом для большинства пользователей — особенно во время отладки.

Предполагается, что обработчик ***_uncaught_handler*** не возвращает управление в вызвавшую программу. Если он попытается, ***terminate*** () вызовет ***abort*** ().

Обратите внимание, что вызов функции ***abort*** () означает ненормальный выход из программы. Можно воспользоваться функцией ***exit*** () для выхода из программы с возвращаемым значением, которое укажет внешней системе, был ли выход нормальным или нет (§ 9.4.1.1).

Вызываются ли деструкторы при прекращении выполнения программы из-за неперехваченного исключения, зависит от реализации. Для некоторых систем важно, чтобы деструкторы не вызывались: тогда программа сможет возобновить

выполнение в среде отладчика. В других системах с точки зрения архитектуры почти невозможно *не* вызывать деструкторы во время поиска подходящего обработчика.

Если вы хотите гарантировать очистку при возникновении неперехваченных исключений, вы можете включить в ***main*** () обработчик, «перехватывающей все» (§ 14.3.2) в дополнение к обработчикам исключений, которые действительно имеют для вас значение. Например:

```
int main ()
try {
        //...
}
catch (std range_error)
{
        cerr << "ошибка диапазона\n"
}
catch (std bad_alloc)
{
        cerr << "new исчерпал память\n",
}
catch ( ){
        //..
}
```

Эта конструкция перехватит любое исключение, не считая тех, которые сгенерированы при конструировании и уничтожении глобальных переменных. Не существует способа перехвата исключений, сгенерированных во время инициализации глобальных переменных. Единственным способом обеспечения контроля в случае генерации исключения в инициализаторе нелокального статического объекта является ***set_unexpected*** () (§ 14.6.2). Это еще одна причина избегать, где возможно, глобальных переменных.

При перехвате исключения точное место, где оно сгенерировано, вообще говоря неизвестно. То есть информация теряется по сравнению с тем, что знает о состоянии программы отладчик. Поэтому для определенных людей, работающих с определенными программами в определенных средах разработки C++, может оказаться предпочтительным *не* перехватывать исключения, на восстановление от которых программа не рассчитана.

14.8. Исключения и эффективность

В принципе, обработку исключений можно реализовать таким образом, что если исключение не сгенерировано, то во время выполнения не возникает никаких дополнительных накладных расходов. Кроме того, обработку можно реализовать так, что генерация исключения окажется не намного дороже вызова функции. Добиться этого без использования значительного количества дополнительной памяти, поддерживая при этом совместимость с последовательностью вызова, принятой в C, и соглашения, необходимые для отладчиков, и т. д., можно, но достаточно сложно. Однако не забывайте, что решения, альтернативные исключениям, тоже даются не даром. Не так уж редко встречаются традиционные системы, половина кода которых занята обработкой ошибок.

Рассмотрим простую функцию *f*(), которая вроде бы не имеет никакого отношения к обработке исключений:

```
void g(int);

void f()
{
        string s;
        //...
        g(1);
        g(2);
}
```

Но *g*() может сгенерировать исключение, поэтому *f*() должна содержать код, гарантирующий, что в случае возникновения исключения *s* будет корректно уничтожена. С другой стороны, если бы *g*() не генерировала исключений, она должна была бы сообщить об ошибке каким-либо другим способом. Следовательно, аналогичный фрагмент, использующий обычный код обработки ошибок (а не исключения), сводится не к приведенной выше простой программе, а к чему-то вроде:

```
bool g(int);

bool f()
{
        string s;
        //...
        if (g(1))
                if (g(2))
                        return true;
                else
                        return false;
        else
                return false;
}
```

Все-таки обычно люди не обрабатывают ошибки столь систематически, да такая систематичность и не всегда важна. Однако если существует необходимость в тщательной и систематической обработке ошибок, соответствующую работу лучше предоставить компьютеру, а именно механизму обработки исключений.

Спецификации исключений (§ 14.6) оказывают значительную помощь в улучшении качества генерируемого кода. Если бы вы явно указали, что *g*() не может генерировать исключения:

```
void g(int) throw();
```

качество сгенерированного для *f*() кода могло бы повыситься. Стоит иметь в виду, что ни одна функция традиционного C не генерирует исключений, поэтому в большинстве программ каждая функция C может быть объявлена с пустой спецификацией ***throw***(). В частности, реализация знает, что только некоторые (немногие) функции стандартной библиотеки C (такие как ***atexit***() и ***qsort***()) могут генерировать исключения, и может воспользоваться этим фактом для генерации более качественного кода.

До того как задать «функции С» пустую спецификацию исключений *throw* (), задумайтесь на минутку, не может ли она сгенерировать исключение. Например, она могла быть модифицирована с использованием оператора C++ *new*, который может сгенерировать *bad_alloc*, или она может вызывать библиотечную функцию C++, которая генерирует исключения.

14.9. Альтернативные методы обработки ошибок

Целью механизмов обработки исключений является предоставление средств для передачи из одной части программы в другую информации о том, что возникли «исключительные обстоятельства». Предполагается, что обе части программы написаны независимо, и что та часть программы, которая обрабатывает исключение, как правило может сделать что-нибудь осмысленное с возникшей ошибкой.

Для эффективного использования обработчиков нам нужна общая стратегия. То есть различные части программы должны договориться о том, как используются исключения и где обрабатываются ошибки. Механизмы обработки исключений не локальны по своей сути, поэтому строгое следование общей стратегии является очень важным. Из этого следует, что стратегию обработки ошибок лучше всего планировать на самых ранних фазах проектирования. Из этого также следует, что стратегия должна быть простой (по сравнению со сложностью самой программы) и явной. Никто не будет последовательно придерживаться сложной стратегии в таких и без того трудных вопросах, как восстановление системы после возникновения ошибок.

Прежде всего следует отвергнуть идею, что единственный механизм или метод может обработать все ошибки — это привело бы к черезмерной сложности. Удачные, устойчивые к ошибкам системы являются многоуровневыми. Каждый уровень разбирается с таким количеством ошибок, которое он в состоянии «переварить», а прочие оставляет вышестоящим уровням. С целью поддержки этой точки зрения вводится *terminate* (), предоставляющая выход в том случае, если сам механизм обработки исключений разрушился, или если он не полностью использован, и остались не перехваченные исключения. Аналогично, *unexpected* () предоставляет выход в тех случаях, когда защитная стратегия использования спецификаций исключений терпит неудачу.

Не все функции должны защищаться по «полной программе». В большинстве систем невозможно снабдить каждую функцию достаточным количеством проверок, чтобы гарантировать, что она завершится успешно или потерпит неудачу хорошо определенным образом. Причины, по которым это неосуществимо, различаются от программы к программе и от программиста к программисту. Однако для больших программ:

[1] Объем работы, необходимой для обеспечения таким образом понятой надежности слишком велик для того, чтобы быть выполненным полностью.

[2] Накладные расходы времени и памяти неприемлемо велики (появляется склонность многократно проверять одни и те же ошибки, такие как недопустимые аргументы).

[3] Функции, написанные на других языках, не подчиняются установленным правилам игры.

[4] Такое чисто локальное понятие «надежности» приводит к сложностям, которые становятся реальным бременем при обеспечении надежности системы в целом.

Тем не менее, разбиение программы на раздельные подсистемы, которые либо завершаются успешно, либо терпят неудачу строго определенным образом, представляется важным, осуществимым и экономичным делом. Основные библиотеки, подсистемы и ключевые функции должны быть разработаны подобным образом. Спецификации исключений применяются как часть интерфейсов таких библиотек и подсистем.

Как правило мы лишены роскоши проектирования всего кода системы с нуля. Соответственно, для реализации общей стратегии обработки ошибок во всех частях программы нам придется принять во внимание фрагменты, написанные с использованием стратегий, отличных от нашей. Чтобы сделать это, мы должны рассмотреть множество подходов, связанных со способами управления ресурсами в различных фрагментах программы и состоянием, в котором они оставляют систему после ошибки. Цель состоит в том, чтобы заставить фрагмент программы работать так, как если бы он следовал общей стратегии обработки ошибок, даже если внутри он придерживается другой схемы.

Иногда возникает необходимость в смене стиля обработки ошибок. Например, после вызова библиотеки C мы можем проверить ***errno*** и, возможно, сгенерировать исключение, или, наоборот, перехватить исключения и установить ***errno*** перед возвращением в C-программу из библиотеки C++:

```
void callC () throw (C_blewit)
{
        errno = 0;
        c_function ();
        if (errno) {
                // очистка, если это возможно и необходимо
                throw C_blewit (errno);
        }
}

extern "C" void call_from_C () throw ()
{
        try {
                c_plus_plus_function ();
        }
        catch (...) {
                // очистка, если это возможно и необходимо
                errno = E_CPLPLFCTBLEWIT;
        }
}
```

В подобных случаях важно быть достаточно систематичным для обеспечения полного преобразования стиля обработки ошибок.

Обработка ошибок должна быть — насколько это возможно — иерархической. Если функция обнаруживает ошибку времени выполнения, она не должна обращаться за помощью к вызвавшей ее функции для восстановления или выделения ресурса. Такие запросы вводят циклы зависимостей в системе. Это в свою очередь делает программы сложными для понимания и порождает возможность бесконечных циклов в коде обработки ошибок и восстановления.

Чтобы сделать код обработки ошибок более последовательным, следует применять упрощающие методы, такие как «выделение ресурса есть инициализация», и упроща-

ющие предположения, такие как «исключение представляет ошибку». В § 24.3.7.1 изложены идеи использования инвариантов и утверждений для того, чтобы сделать вызов исключений более регулярным.

14.10. Стандартные исключения

Приведем таблицу стандартных исключений, а также функций, операторов и общих средств, которые их генерируют :

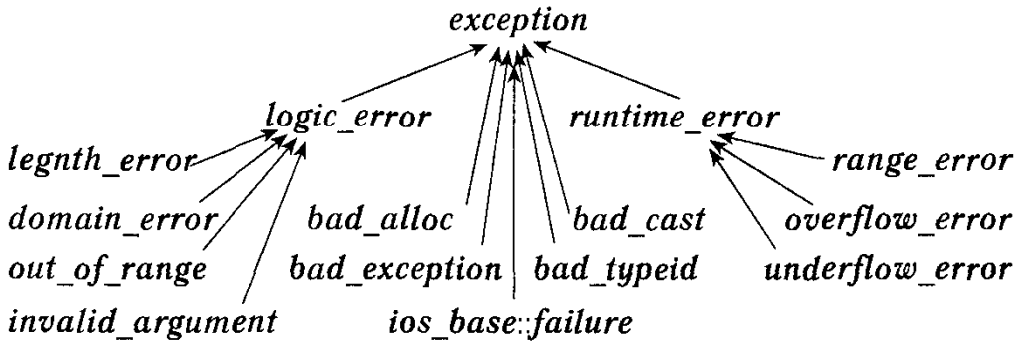
Стандартные исключения (генерируются языком)

Имя	Чем генерируется	Ссылка	Заголовочный файл
bad_alloc	*new*	§ 6.2.6.2, § 19.4.5	<*new*>
bad_cast	*dynamic_cast*	§ 15.4.1.1	<*typeinfo*>
bad_typeid	*typeid*	§ 15.4.4	<*typeinfo*>
bad_exception	*спецификация исключения*	§ 14.6.3	<*exception*>
out_of_range	*at* ()	§ 3.7.2, § 16.3.3, § 20.3.3	<*stdexcept*>
	bitset<>::operator[] ()	§ 17.5.3	<*stdexcept*>
invalid_argument	*конструктор bitset*	§ 17.5.3.1	<*stdexcept*>
overflow_error	*bitset<>::to_ulong* ()	§ 17.5.3.3	<*stdexcept*>
ios_base::failure	*ios_base::clear* ()	§ 21.3.6	<*ios*>

Библиотечные исключения являются частью иерархии классов, вершина которой — стандартный библиотечный класс исключений ***exception***, представленный в <*stdexcept*>:

```
class exception {
public:
        exception () throw ();
        exception (const exception&) throw ();
        exception& operator= (const exception&) throw ();
        virtual ~exception () throw ();

        virtual const char* what () const throw ();
private:
        // ...
};
```

Эта иерархия выглядит следующим образом:



Это выглядит избыточно для организации восьми стандартных исключений. Однако данная иерархия пытается обеспечить основу и для исключений за пределами стан-

дартной библиотеки. Логическими (***logic_error***) являются ошибки, которые в принципе можно перехватить либо до запуска программы, либо путем тестирования функций и конструкторов. Ошибками этапа выполнения (***runtime_error***) являются все остальные ошибки. Некоторые люди считают это полезной основой для всех ошибок и исключений. Я — нет.

Классы исключений стандартной библиотеки не добавляют функции к набору, обеспечиваемому ***exception***; они просто определяют подходящим образом требуемые виртуальные функции. Таким образом, мы можем написать:

```
void f()
try {
        // использование стандартной библиотеки
}
catch (exception& e) {
        cout << "исключение стандартной библиотеки" << e.what() << "\n";
        // ...
}
catch (...) {
        cout << "другое исключение\n";
        // ...
}
```

Стандартные исключения являются производными от ***exception***. Однако этого нельзя сказать про все исключения, так что попытка перехвата всех исключений при помощи перехвата ***exception*** была бы ошибочной. Аналогично, будет ошибкой предполагать, что каждое исключение, производное от ***exception***, является исключением стандартной библиотеки: программисты могут добавлять свои собственные исключения в иерархию ***exception***.

Обратите внимание, что операции ***exception*** сами не генерируют исключений. В частности из этого следует, что генерация исключения стандартной библиотеки не приведет к исключению ***bad_alloc***. Механизм обработки исключений резервирует некоторое количество памяти для себя в целях хранения исключений (возможно, в стеке). Естественно, можно написать код, который постепенно приведет к исчерпанию всей памяти в системе, вызвав таким образом сбой. Например, ниже приведена функция, которая, если будет вызвана, проверяет, кто первым исчерпает память: вызов функции или обработка исключений:

```
void perverted()
{
        try {
                throw exception();          // рекурсивная генерация исключения
        }
        catch (exception& e) {
                perverted();                // рекурсивный вызов функции
                cout << e.what();
        }
}
```

Назначение оператора вывода состоит просто в том, чтобы предотвратить повторное использование компилятором памяти, занимаемой исключением с именем *e*.

14.11. Советы

[1] Пользуйтесь исключениями для обработки ошибок; § 14.1.
[2] Не пользуйтесь исключениями в случаях, когда достаточно локальных управляющих структур; § 14.1.
[3] Пользуйтесь принципом «выделение ресурса есть инициализация» для управления ресурсами; § 14.4.
[4] Не каждая программа должна быть безопасной с точки зрения исключений; § 14.4.3.
[5] Пользуйтесь принципом «выделение ресурса есть инициализация» и обработчиками исключений для поддержки инвариантов; § 14.3.2.
[6] Сводите к минимуму использование блоков ***try***. Вместо явного кода обработки пользуйтесь принципом «выделение ресурса есть инициализация»; § 14.4.
[7] Не в каждой функции требуется обрабатывать все возможные ошибки; § 14.9.
[8] Генерируйте исключения для указания на ошибку в конструкторе; § 14.4.6.
[9] Перед генерацией исключения из выражения присваивания убедитесь в том, что все операнды останутся в корректном состоянии; § 14.4.6.2.
[10] Избегайте генерации исключений в деструкторах; § 14.4.7.
[11] Заставьте ***main*** () перехватывать все исключения и сообщать о них; § 14.7.
[12] Разделяйте обычный код и код обработки ошибок; § 14.4.5, § 14.5.
[13] Гарантируйте, что каждый ресурс, выделенный в конструкторе, освобождается при возникновении исключения в этом конструкторе; § 14.4.
[14] Управление ресурсами должно быть иерархическим; § 14.4.
[15] Пользуйтесь спецификацией исключений для важных интерфейсов; § 14.9.
[16] Опасайтесь утечек памяти, вызываемых выделением памяти при помощи оператора ***new*** без последующего освобождения при возникновении исключений; § 14.4.1, § 14.4.2, § 14.4.4.
[17] Считайте, что каждое исключение, которое может быть сгенерировано функцией, будет сгенерировано; § 14.6.
[18] Не предполагайте, что все исключения являются производными от класса ***exception***; § 14.10.
[19] Библиотека не должна волевым решением прекращать выполнение программы — ей следует сгенерировать исключение и позволить принять решение вызывающей функции; § 14.1.
[20] Библиотека не должна выводить диагностическую информацию конечному пользователю — ей следует сгенерировать исключение и позволить принять решение вызывающей функции; § 14.1.
[21] Разрабатывайте стратегию обработки ошибок на ранних этапах проектирования; § 14.9.

14.12 Упражнения

1. (*2) Обобщите класс ***STC*** (§ 14.6.3.1) до шаблона, который может использовать технику «выделение ресурса есть инициализация» для хранения и переустановки функций различных типов.

2. (*3) Завершите класс ***Ptr_to_T*** из § 11.11 в виде шаблона, который использует исключения для сигнализации об ошибках времени выполнения.
3. (*3) Напишите функцию, осуществляющую поиск значения в узлах двоичного дерева из ***char****. Если узел, содержащий слово «здравствуй», найден, функция ***find*** (*"здравствуй"*) возвратит указатель на этот узел. Воспользуйтесь исключением для индикации «не найдено».
4. (*3) Определите класс ***Int***, который ведет себя точно также, как встроенный тип ***int***, за исключением того, что он генерирует исключения, не допуская переполнения сверху или снизу.
5. (*2.5) Возьмите базовые операции открытия, закрытия, чтения и записи из интерфейса C к вашей операционной системе и реализуйте эквивалентные функции на C++, которые в случае возникновения ошибок генерирует исключения.
6. (*2.5) Напишите законченный шаблон ***Vector*** с исключениями ***Range*** (диапазон) и ***Size*** (размер).
7. (*1) Напишите цикл, вычисляющий сумму ***Vector*** из § 14.12[6], не проверяя размер вектора. Почему это плохая идея?
8. (*2.5) Подумайте об использовании класса ***Exception*** в качестве базового для всех классов, применяемых в качестве исключений. Как это может выглядеть? Как это следует использовать? К чему хорошему это может привести? Какие недостатки возникают из-за требования использовать такой класс?
9. (*1) Имеется

   ```
   int main () { /* ... */ }
   ```

   Внесите такие изменения, чтобы все исключения перехватывались, преобразовывались в сообщения об ошибках, и затем выполнение завершалось по ***abort*** (). Подсказка: функция ***call_from_C*** () в § 14.9 не полностью обрабатывает все случаи.
10. (*2) Напишите класс или шаблон, подходящий для реализации обратного вызова.
11. (*2.5) Напишите класс ***Lock*** (блокировка) для некоторой параллельной системы.

15

Иерархии классов

Абстракция — это выборочное невежество.
— Эндрю Кёниг

Множественное наследование — разрешение неоднозначности — наследование и ***using-объявления*** — повторяющиеся базовые классы — виртуальные базовые классы — использование множественного наследования — управление доступом — защищенные члены — доступ к базовым классам — информация о типе на этапе выполнения — динамическое приведение ***dynamic_cast*** — статические и динамические приведения — приведение из виртуального базового класса — ***typeid*** — расширенная информация о типе — правильное и неправильное использование информации о типе на этапе выполнения — указатели на члены — свободная память — виртуальные конструкторы — советы — упражнения.

15.1. Введение и обзор

В этой главе обсуждается, как производные классы и виртуальные функции взаимодействуют с другими средствами языка, такими как: управление доступом, поиск имен, управление свободной памятью, конструкторы, указатели и преобразования типов. Глава состоит из пяти основных разделов:

§ 15.2 Множественное наследование.
§ 15.3 Управление доступом.
§ 15.4 Определение типа во время выполнения.
§ 15.5 Указатели на члены.
§ 15.6 Использование свободной памяти.

В общем случае, класс создается из решетки базовых классов. Ввиду того, что исторически большинство таких решеток были деревьями, *решетку классов* часто называют *иерархией классов*. Мы пытаемся проектировать классы таким образом, чтобы пользователя без острой необходимости не интересовало, каким образом класс составляется из других классов. В частности, механизм виртуальных вызовов гарантирует, что когда мы вызываем функцию *f*() для некоторого объекта, вызывается одна и та же функция, независимо от того, какой класс в иерархии содержит объявление *f*(), использованное для вызова. В этой главе особое внимание уделяется построению решеток классов, управлению доступом к частям классов и средствам навигации по решеткам классов во время компиляции и во время выполнения.

15.2. Множественное наследование

Как показано в § 2.5.4 и § 12.3, класс может иметь более одного непосредственного базового класса, то есть после «:» в объявлении класса может быть указано несколько классов. Рассмотрим задачу моделирования, в которой параллельные задачи представлены классом ***Task*** (задача), а сбор и вывод данных осуществляется при помощи класса ***Displayed*** (отображаемый). Затем мы определим класс моделируемых сущностей, класс ***Satellite*** (спутник):

```
class Satellite : public Task, public Displayed {
        // ...
};
```

Использование более одного непосредственного базового класса обычно называется *множественным наследованием*. В противоположность этому наличие одного базового класса называется *одиночным наследованием*.

Кроме операций, определенных для ***Satellite***, мы можем воспользоваться объединением операций ***Task*** и ***Displayed***. Например:

```
void f(Satellite& s)
{
        s.draw();               // Displayed::draw()
        s.delay(10);            // Task::delay()
        s.transmit();           // Satellite::transmit()
}
```

Аналогичным образом, ***Satellite*** можно передать в функции, которые ожидают ***Task*** или ***Displayed***. Например:

```
void highlight(Displayed*);
void suspend(Task*);

void g(Satellite* p)
{
        highlight(p);           // передать указатель на Displayed-часть Satellite
        suspend(p);             // передать указатель на Task-часть Satellite
}
```

Реализация этого механизма очевидно подразумевает (простые) методы компиляции для обеспечения того, что функции, ожидающие ***Task***, увидят часть ***Satellite***, отличную от той, которую видят функции, ожидающие ***Displayed***. Виртуальные функции работают как обычно. Например:

```
class Task {
        // ...
        virtual void pending() = 0;
};

class Displayed {
        // ...
        virtual void draw() = 0;
};

class Satellite : public Task, public Displayed {
        // ...
```

```
        void pending();          // замещение Task::pending()
        void draw();             // замещение Displayed::draw()
};
```

Такой подход гарантирует, что функции ***Satellite::draw*** () и ***Satellite::pending*** () будут вызваны для ***Satellite***, проинтерпретированного как ***Displayed*** и ***Task*** соответственно.

Обратите внимание, что при наличии только одиночного наследования выбор реализации для классов ***Displayed***, ***Task*** и ***Satellite*** у программиста будет ограниченным. ***Satellite*** может быть ***Task*** или ***Displayed***, но не обоими (если только ***Task*** не является производным от ***Displayed*** или наоборот). Выбор любой альтернативы ведет к потере гибкости.

Кому может понадобиться класс ***Satellite***? Вопреки предположениям некоторых людей пример ***Satellite*** является реальным. В действительности, существовала — и, может быть, существует до сих пор — программа, построенная способом, очень похожим на тот, что используется здесь для описания множественного наследования. Она использовалась для изучения проектирования коммуникационных систем, включающих спутники, наземные станции и т. д. При наличии такой модели мы можем ответить на вопросы об интенсивности трафика, определить правильные действия при блокировании одной из наземных станций (например, в результате ливня), правильно распределить трафик между спутниковой и наземной связью и т. д. Подобное моделирование включает в себя множество отладочных функций и операций, отображающих ту или информацию. Кроме того, мы должны хранить состояние таких объектов как ***Satellite*** и их подкомпоненты для анализа, отладки и восстановления после ошибок.

15.2.1. Разрешение неоднозначности

Два базовых класса могут иметь функции-члены с одинаковым именем. Например:

```
class Task {
        // ...
        virtual debug_info* get_debug();
};

class Displayed {// ...
        virtual debug_info* get_debug();
};
```

При использовании ***Satellite*** неоднозначности для этих функций должны быть устранены:

```
void f(Satellite* sp)
{
        debug_info* dip = sp->get_debug();      // ошибка: неоднозначно
        dip = sp->Task::get_debug();            // правильно
        dip = sp->Displayed::get_debug();       // правильно
}
```

Однако явное устранение неоднозначности довольно неудобно (слишком пространно), поэтому лучше разрешить эти проблемы, определив новую функцию в производном классе:

```
class Satellite : public Task, public Displayed {
        //...
        // замещение Task::get_debug() и Displayed::get_debug()
        debug_info* get_debug ()
        {
                debug_info* dip1 = Task::get_debug ();
                debug_info* dip2 = Displayed::get_debug ();
                return dip1->merge (dip2);
        }
};
```

Это локализует информацию о базовых классах ***Satellite***. Так как ***Satellite::get_debug*** () замещает функции ***get_debug*** () из обоих классов, ***Satellite::get_debug*** () вызывается при каждом вызове ***get_debug*** () для объекта ***Satellite***.

Квалифицированное имя ***Telstar::draw*** может ссылаться на ***draw***, объявленную либо в ***Telstar***, либо в одном из его базовых классов. Например:

```
class Telstar : public Satellite {
        //...
        void draw ()
        {
                draw ();                        // ошибка: рекурсивный вызов
                Satellite::draw (),             // находит Displayed::draw
                Displayed..draw ();
                Satellite::Displayed:.draw ();  // избыточная двойная квалификация
        }
};
```

Другими словами, если ***Satellite::draw*** () не разрешается в ***draw*** (), объявленную в ***Satellite***, компилятор рекурсивно осуществляет поиск в базовых классах; то есть смотрит ***Task::draw*** () и ***Displayed::draw*** (). Если найдено ровно одно соответствие, используется найденное имя. В противном случае, функция ***Satellite::draw*** () либо не найдена, либо неоднозначна.

15.2.2. Наследование и using-объявления

Разрешение перегрузки не пересекает границ областей видимости классов (§ 7.4). В частности, неоднозначности между функциями из различных базовых классов не разрешаются на основе типов аргументов.

При создании комбинации существенно различных классов, таких как ***Task*** и ***Dispalyed*** в примере ***Satellite***, сходство имен обычно не означает сходство назначения. Подобные конфликты имен часто являются полной неожиданностью для программиста. Например:

```
class Task {
        //...
        void debug (double p);      // вывести информацию только в том случае,
                                    // если приоритет ниже p
};

class Displayed {
```

```
        //...
        void debug(int v);          // чем больше v, тем больше отладочной
                                    // информации выводится
};
class Satellite : public Task, public Displayed {
        //...
};
void g(Satellite* p)
{
        p->debug(1);                // ошибка: неоднозначность
                                    // Displayed::debug(int) или Task::debug(double)?
        p->Task::debug(1);          // правильно
        p->Displayed::debug(1);     // правильно
}
```

Что если использование одного и того же имени в различных базовых классах явилось результатом тщательно продуманного проектного решения и пользователь хотел, чтобы выбор осуществлялся по типам аргумента? В этом случае, ***using-объявления*** (§ 8.2.2) могут ввести обе функции в общую область видимости. Например:

```
class A {
public:
        int f(int);
        char f(char);
        //...
};
class B {
public:
        double f(double);
        //...
};
class AB: public A, public B {
public:
        using A::f;
        using B::f;
        char f(char);           // скрывает A::f(char)
        AB f(AB);
};
void g(AB& ab)
{
        ab.f(1);                // A::f(int)
        ab.f('a');              // AB::f(char)
        ab.f(2.0);              // B::f(double)
        ab.f(ab);               // AB::f(AB)
}
```

Объявления ***using*** позволяют программисту создавать набор перегруженных функций из базовых и производных классов. Функции, объявленные в производном классе, скрывают функции из базового класса, которые в противном случае были бы доступны. Виртуальные функции базового класса можно замещать как обычно (§ 15.2.3.1).

using-объявление (§ 8.2.2) в определении класса должно относиться к членам базового класса. *using-объявление* нельзя использовать для члена класса вне этого класса, его производных классов или их функций-членов. *using-директиву* (§ 8.2.3) нельзя поместить в определение класса, и она не может использоваться для класса.

using-объявление не может использоваться для получения доступа к дополнительной информации. Оно просто является механизмом предоставления более удобного доступа к информации, доступ к которой в принципе разрешен (§ 15.3.2.2).

15.2.3. Повторяющиеся базовые классы

При задании более чем одного базового класса возникает вероятность того, что какой-либо класс дважды окажется базовым для другого класса. Например, если бы каждый из классов ***Task*** и ***Displayed*** был производным от класса ***Link***, у ***Satellite*** было бы два ***Link***:

```
struct Link {
        Link* next;
};

class Task : public Link {
        // Link используется для хранения списка всех задач Task (список планировщика)
        // ...
};

class Displayed : public Link {
        // Link используется для хранения объектов всех отображаемых
        // (Displayed) объектов (список того, что отображается)
        // ...
};
```

Это не вызывает никаких проблем. Используются два отдельных объекта ***Link*** для представления связей, и эти два списка не взаимодействуют друг с другом. Естественно, обращаясь к членам класса ***Link***, вы рискуете получить неоднозначность (§ 15.2.3.1). Объект ***Satellite*** можно представить в графическом виде следующим образом:

В тех случаях, когда общий базовый класс не должен быть представлен в виде двух отдельных объектов, нужно воспользоваться виртуальным базовым классом (§ 15.2.4).

Как правило, базовый класс, который повторяется (как ***Link*** в нашем примере), является деталью реализации, которую не следует использовать вне непосредственно производных от него классов. Если к такому базовому классу нужен доступ из места, где видна более чем одна копия базового класса, во избежание неоднозначности ссылка должна быть явно квалифицирована. Например:

```
void mess_with_links (Satellite* p)
{
        p->next = 0;                    // ошибка: неоднозначно, какой Link?
```

```
    p->Link::next = 0;          // ошибка: неоднозначно, какой Link?
    p->Task::next = 0;          // правильно
    p->Displayed::next = 0;     // правильно
    //...
}
```

Это в точности тот же механизм, который используется для разрешения неоднозначности при обращении к членам (§ 15.2.1).

15.2.3.1. Замещение

Виртуальная функция повторяющегося базового класса может быть замещена (единственной) функцией в производном классе. Например, можно следующим образом предоставить объекту возможность считывать себя из файла и записывать обратно в файл:

```
class Storable {                    // хранимый
public:
    virtual const char* get_file () = 0;
    virtual void read () = 0;
    virtual void write () = 0;
    virtual ~Storable () { }
};
```

Естественно, несколько программистов могут воспользоваться этим для проектирования классов, которые могут применяться независимо или в комбинации для построения более специализированных классов. Например, одним из способов завершения и возобновления моделирования является сохранение компонент моделирования с последующим их восстановлением. Эту идею можно реализовать следующим образом:

```
class Transmitter : public Storable {           // передатчик
public:
    void write ();
    //...
};

class Receiver : public Storable {              // приемник
public:
    void write ();
    //...
};

class Radio : public Transmitter, public Receiver { // радио
public:
    const char* get_file ();
    void read ();
    void write ();
    //...
};
```

Довольно часто замещающая функция вызывает ее версию из базового класса и затем выполняет действия, специфичные для производного класса:

```
void Radio::write ()
{
        Transmitter::write ();
        Receiver::write ();
        // запись информации, специфичной для радио
}
```

Приведение из повторяющегося базового класса в производный класс обсуждается в § 15.4.2. Метод замещения каждой функции ***write*** () отдельными функциями из производных классов см. в § 25.6.

15.2.4. Виртуальные базовые классы

Пример с ***Radio*** из предыдущего подраздела работает, потому что можно безопасно, удобно и эффективно реплицировать ***Storable***. Часто этот подход не годится в случае классов, создаваемых в качестве строительных блоков для других классов. Например, мы могли бы определить ***Storable*** таким образом, чтобы он содержал имя файла, используемого для хранения объекта:

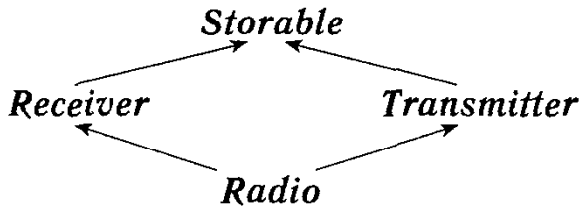
```
class Storable {
public:
        Storable (const char* s);
        virtual void read () = 0;
        virtual void write () = 0;
        virtual ~Storable () { write (); }
private:
        const char* store;

        Storable (const Storable&);
        Storable& operator= (const Storable&);
};
```

Изменив ***Storable*** внешне незначительно, мы должны изменить подход к проектированию ***Radio***. Все части объекта должны совместно использовать единственную копию ***Storable***; в противном случае неоправданно сложно становится избегать хранения нескольких копий объекта. Одним из механизмов задания такого совместного использования является виртуальный базовый класс. Каждый виртуальный базовый класс в производном классе представлен одним и тем же (совместно используемым) объектом. Например:

```
class Transmitter : public virtual Storable {
public:
        void write ();
        // ...
};

class Receiver : public virtual Storable {
public:
        void write ();
        // ...
};

class Radio : public Transmitter, public Receiver {
```

```
public:
        void write ();
        // ...
};
```

Или в графическом виде:



Сравните эту диаграмму с представлением объекта ***Satellite*** в § 15.2.3, чтобы увидеть разницу между обычным и виртуальным наследованием. В графе наследования все виртуальные базовые классы с одним именем будут представлены одним единственным объектом этого класса. С другой стороны, каждый невиртуальный базовый класс будет иметь отдельный подобъект, представляющий его.

15.2.4.1. Программирование виртуальных базовых классов

При определении функций класса с виртуальным базовым классом программист, в общем случае, не может знать, будет ли базовый класс использоваться совместно с другими производными классами. Это может представлять некоторую проблему при реализации алгоритмов, которые требуют, чтобы функция базового класса вызывалась ровно один раз. Например, язык гарантирует, что конструктор виртуального базового класса вызывается только один раз. Конструктор виртуального базового класса вызывается (явно или неявно) из конструктора объекта (конструктора самого «нижнего» производного класса). Например:

```
class A {           // нет конструктора
        // ...
};

class B {
public:
        B ();       // конструктор по умолчанию
        // ...
};

class C {
public:
        C (int);    // нет конструктора по умолчанию
};

class D : virtual public A, virtual public B, virtual public C
{
        D () { /* ... */ }                  // ошибка: нет конструктора по умолчанию для C
        D (int i) : C (i) { /* ... */ };    // правильно
        // ...
};
```

Конструктор виртуального базового класса вызывается до конструкторов производных классов.

При необходимости программист может смоделировать эту схему, вызывая функцию виртуального базового класса только из самого «нижнего» производного класса. Например, предположим, что у нас есть базовый класс ***Window***, который знает, как рисовать свое содержание:

```
class Window {
        // какой-то код
        virtual void draw ();
};
```

Кроме того, у нас есть различные способы оформления окна и введения дополнительных средств:

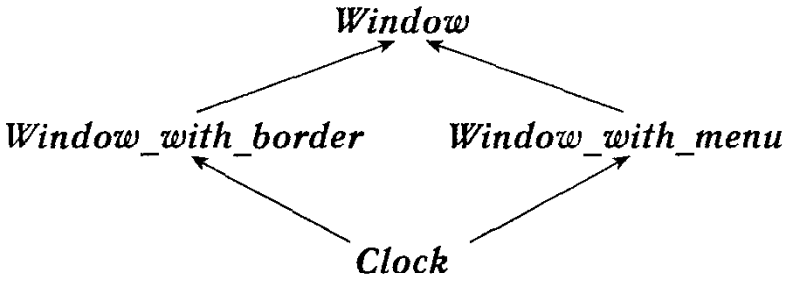
```
class Window_with_border : public virtual Window {        // окно с рамкой
        // код, имеющий отношение к рамке
        void own_draw ();                                 // отобразить рамку
        void draw ();
};

class Window_with_menu : public virtual Window {          // окно с меню
        // код, имеющий отношение к меню
        void own_draw ();                                 // отобразить меню
        void draw ();
};
```

Функции ***own_draw*** () не обязаны быть виртуальными, потому что предполагается, что они будут вызваны из виртуальной функции ***draw*** (), которая «знает» тип объекта, для которго она вызвана.

Из этого мы можем создать вполне приемлемый класс ***Clock*** (часы):

```
class Clock : public Window_with_border, public Window_with_menu {
        // код, имеющий отношение к часам
        void own_draw ();          // отобразить циферблат и стрелки
        void draw ();
};
```

Или в графическом виде:



Теперь функции ***draw*** () можно написать с использованием функций ***own_draw*** () таким образом, чтобы код, вызывающий любую ***draw*** (), косвенно вызывал при этом ***Window***::***draw*** () ровно один раз. Это осуществляется независимо от типа окна ***Window***, для которого вызывается ***draw*** ():

```
void Window_with_border::draw ()
{
        Window::draw ();
```

```
        own_draw ();            // отобразить рамку
}

void Window_with_menu::draw ()
{
        Window::draw ();
        own_draw (),            // отобразить меню
}

void Clock : public Window_with_border::draw ()
{
        Window::draw ();
        Window_with_border::own_draw ();
        Window_with_menu::own_draw ();
        own_draw (),            // отобразить циферблат и стрелки
}
```

Приведение из виртуального базового класса в производный обсуждается в § 15.4.2.

15.2.5. Использование множественного наследования

Простейшим и наиболее очевидным применением множественного наследования является «склеивание» двух никаким другим образом не связанных классов вместе в качестве части реализации третьего класса. Класс ***Satellite***, созданный на основе классов ***Task*** и ***Displayed*** в § 15.2, является характерным примером. Такое использование множественного наследования достаточно примитивно, эффективно, имеет большое значение, но не очень интересно. В основном оно позволяет программисту избежать написания большого количества функций, которые переадресуют вызовы друг другу. Эта техника не оказывает большого воздействия на проект программы в целом и иногда вступает в конфликт с желанием скрыть детали реализации. Однако для того чтобы быть полезной, техника не обязана быть слишком «умной».

Использование множественного наследования для реализации абстрактных классов является более фундаментальной задачей, которая оказывает воздействие на проектирование программы. Класс ***BB_ival_slider*** (§ 12.3) может служить примером:

```
class BB_ival_slider :
        public Ival_slider ,    // интерфейс
        protected BBslider      // реализация
{
        // реализация функций, требуемых Ival_slider и BBslider
        // с использованием средств, предоставляемых BBslider
},
```

В этом примере два базовых класса с логической точки зрения играют совершенно различную роль. Один базовый класс является открытым абстрактным классом, обеспечивающим интерфейс, а другой — защищенным конкретным классом, представляющим «детали» реализации. Эти роли отражены и в стиле наименования классов и в предоставленном доступе. Множественное наследование в этом примере существенно, потому что производный класс должен заместить виртуальные функции обоих классов — и интерфейса и реализации.

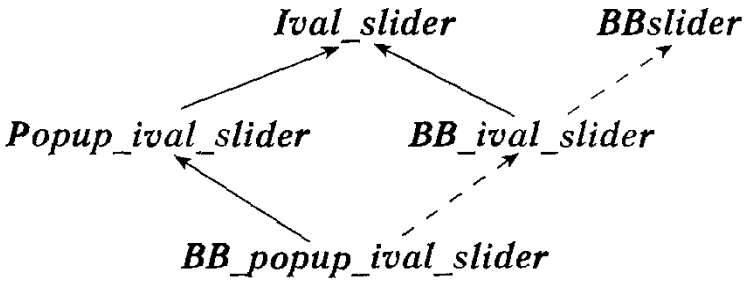
Множественное наследование позволяет «братским» классам совместно использовать информацию без введения зависимости от единственного общего базового класса в программе. Это является случаем так называемого *ромбовидного наследования* (например, ***Radio*** (§ 15.2.4) и ***Clock*** (§ 15.2.4)). Виртуальный базовый класс в противоположность обыкновенному базовому классу требуется в тех случаях, когда базовый класс не должен повторяться.

Я считаю ромбовидное наследование хорошо управляемым в тех случаях, когда либо виртуальный базовый класс либо классы, производные непосредственно от него, являются абстрактными классами. В качестве примера рассмотрите еще раз классы семейства ***Ival_box*** из § 12.4. В конце концов, я сделал классы ***Ival_box*** абстрактными, чтобы отразить их чисто интерфейсную роль. Такое решение позволило мне поместить детали реализации в конкретные классы. Кроме того, совместное использование деталей реализации было выполнено в форме классической иерархии оконной системы, использовавшейся при реализации.

Имело бы определенный смысл, если бы класс, реализующий ***Popup_ival_slider***, использовал большую часть реализации совместно с простым ***Ival_slider***. В конце концов, эти классы реализации могли бы совместно использовать почти все, кроме обработки ввода от пользователя. Однако тогда представляется естественным исключить повторение объектов ***Ival_slider*** в получающихся объектах, реализующих ползунки ***sliders***. Поэтому мы могли бы сделать ***Ival_slider*** виртуальным базовым классом:

```
class BB_ival_slider : public virtual Ival_slider, protected BBslider { /* ... */ },
class Popup_ival_slider : public virtual Ival_slider { /* ... */ },
class BB_popup_ival_slider : public virtual Popup_ival_slider,
    protected BB_ival_slider { /* ... */ };
```

или в графическом виде:



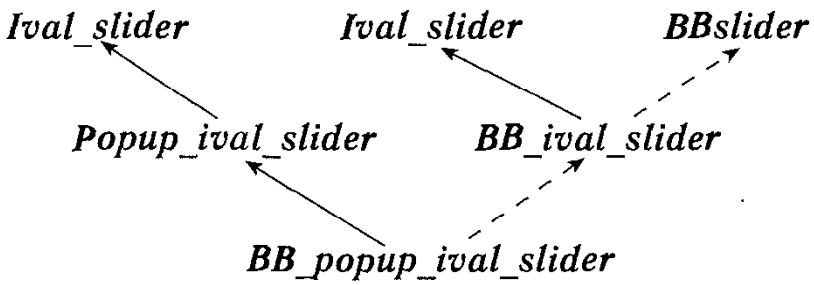
Легко представить себе производные от ***Popup_ival_slider*** интерфейсы и последующие классы реализации, производные от таких классов и ***BB_popup_ival_slider***.

Если мы доведем эту идею до логического конца, все классы, которые формируют интерфейсы нашего приложения и являются производными от абстрактных классов, должны стать виртуальными. Такой подход действительно кажется самым логичным, общим и гибким. Причины, по которым я так не сделал, отчасти имеют исторический характер, а отчасти связаны с тем, что наиболее очевидные и широко используемые техники реализации концепции виртуальных базовых классов требуют дополнительных затрат времени и памяти, что делает их интенсивное использование в классе непривлекательным. При учете этих затрат для анализа во всем остальном приемлемого подхода, очерченного выше, обратите внимание, что объект класса ***Ival_slider*** обычно хранит только указатель на таблицу виртуальных функций. Как отмечено в § 15.2.4, такой абстрактный класс, не хранящий переменных данных, можно делать повторяю-

щимся, не опасаясь неприятных побочных эффектов. Таким образом, мы можем заменить виртуальный базовый класс обычным:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
class BB_popup_ival_slider : public Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

или в графическом виде:



Скорее всего, это является жизнеспособной оптимизацией более понятной альтернативы и вполне допустимой, представленной выше. Потенциальная проблема состоит в том, что теперь ***BB_popup_ival_slider*** не может быть явно преобразован к ***Ival_slider***.

15.2.5.1. Замещение функций виртуальных базовых классов

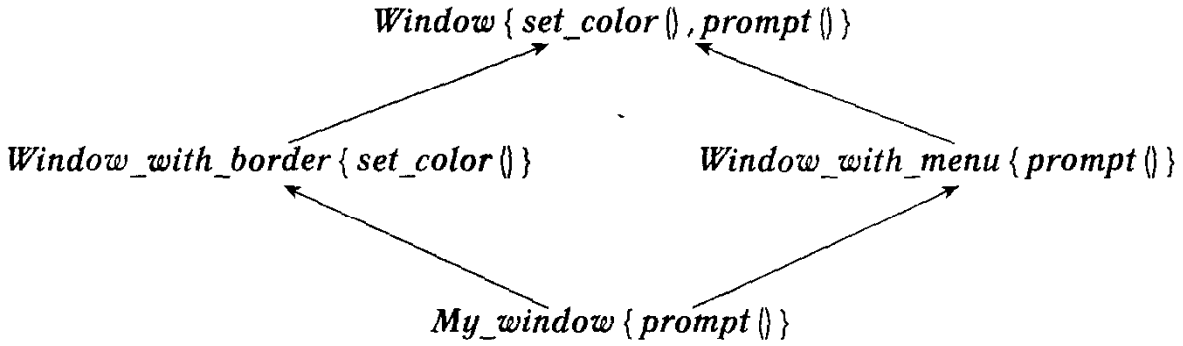
Производный класс может заместить виртуальную функцию своего непосредственного или косвенного виртуального базового класса. В частности, два различных класса могут заместить различные виртуальные функции виртуального базового класса. Таким способом несколько производных классов могут внести свой вклад в реализацию интерфейса, представленного в виртуальном базовом классе. Например, класс ***Window*** мог бы иметь функции ***set_color*** () (установить цвет) и ***prompt*** () (выдать приглашение на ввод). В этом случае ***Window_with_border*** мог бы заместить функцию ***set_color*** () для управления цветовой схемой, а ***Window_with_menu*** мог бы заместить функцию ***prompt*** () для управления интерактивным взаимодействием с пользователем:

```
class Window {
        //...
        virtual set_color (Color) = 0;          // установка цвета фона
        virtual void prompt () = 0;
};

class Window_with_border : public virtual Window {
        //...
        set_color (Color);                      // управление цветом фона
},

class Window_with_menu . public virtual Window {
        //...
        void prompt ();                         // управление взаимодействием с пользователем
};

class My_window : public Window_with_menu, public Window_with_border {
        //...
};
```

Что если различные производные классы заместят одну и ту же функцию? Тогда если мы порождаем от этих классов новый производный класс, мы должны в нем заместить эту функцию. То есть, одна функция должна замещать все остальные. Например, ***My_window*** мог бы заместить ***prompt*** () для улучшения интерфейса ***Window_with_menu***:

```
class My_window  public Window_with_menu, public Window_with_border {
    //...
    void prompt ();          // не оставляем взаимодействие с пользователем
                             // базовому классу
};
```

или в графическом виде:



Если два класса замещают функцию базового класса, то порождение от них производного класса, не замещающего эту функцию, недопустимо. В этом случае не может быть создана таблица виртуальных функций, потому что вызов этой функции с завершенным объектом будет неоднозначен. Например, если бы класс ***Radio*** из § 15.2.4 не объявил ***write*** (), объявления ***write*** () в ***Receiver*** и ***Transmitter*** вызвали бы ошибку при определении ***Radio***. Как и в случае с ***Radio***, такой конфликт разрешается путем добавления замещающей функции в самый «нижний» производный класс.

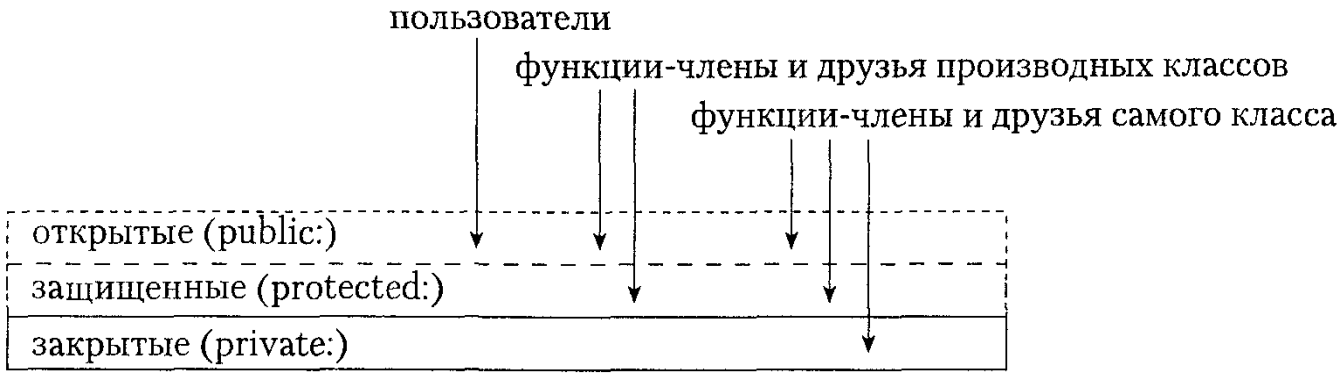
Класс, который обеспечивает некоторую часть — но не всю — реализации виртуального базового класса, часто называют «примесью».

15.3. Управление доступом

Член класса может быть закрытым (***private***), защищенным (***protected***) или открытым (***public***):

- Если он закрыт, его имя может использоваться только в функциях-членах и друзьях класса, в котором он объявлен.
- Если он защищен, его имя может использоваться только в функциях-членах и друзьях класса, в котором он объявлен, и классов, производных от него (см. § 11.5).
- Если он открыт, его именем может пользоваться любая функция.

Это отражает ту точку зрения, что существует три вида функций, имеющих доступ к классу: функции, реализующие класс (члены и друзья), функции, реализующие производные классы (друзья и члены производных классов) и все остальные функции. Это можно изобразить графически:



Управление доступом применяется одинаковым образом ко всем именам. То, к чему относится имя, не оказывает влияние на управление его использованием. Это означает, что мы можем иметь закрытые функции-члены, типы, константы и т. д., так же как закрытые члены данных. Например, эффективному неинтрузивному (§ 16.2.1) классу списков часто требуются структуры данных для учета элементов. Такую информацию лучше сделать закрытой:

```
template<class T> class List {
private:
        struct Link { T val; Link* next; };
        struct Chunk {
                enum { chunk_size = 15 };
                Link v[chunk_size];
                Chunk* next;
        };

        Chunk* allocated;
        Link* free;
        Link* get_free ();
        Link* head;
public:
        class Underflow { };            // класс исключения

        void insert (T);
        T get ();
        // ...
};

template<class T> void List<T>::insert (T val)
{
        Link* lnk = get_free ();
        lnk->val = val;
        lnk->next = head;
        head = lnk;
}

template<class T> List<T>::Link* List<T>::get_free ()
{
        if (free == 0) {        // если свободной памяти больше нет
                                // выделить новый кусок (chunk) памяти
                                // и поместить его связи Link в список свободной памяти
        }
```

```
        Link* p = free;
        free = free->next;
        return p;
}
template<class T> T List<T>::get ()
{
        if (head == 0) throw Underflow ();

        Link* p = head;
        head = p->next;
        p->next = free;
        free = p;
        return p->val;
}
```

Область видимости `List<T>` начинается с `List<T>::` в определении функции-члена. Так как возвращаемый тип функции `get_free ()` упоминался до имени `List<T>::get_free ()`, должно использоваться полное имя `List<T>::Link`, а не сокращенная форма `Link<T>`.

Функции-не-члены (за исключением друзей) не имеют подобного доступа:

```
void would_be_meddler (List<T>* p)
{
        List<T>::Link* q = 0;                  // ошибка: List<T>::Link — закрытое имя

        q = p->free;                           // ошибка: List<T>::Link — закрытое имя
        // ...
        if (List<T>::Chunk::chunk_size > 31) {  // ошибка: List<T>::Chunk::chunk_size —
                                               // закрытое имя
                // ...
        }
}
```

В классе члены по умолчанию закрытые, а в структуре — открытые (§ 10.2.8).

15.3.1. Защищенные члены

Рассмотрим пример с `Window` из § 15.2.4.1. Функции `own_draw ()` спроектированы в качестве строительных блоков для использования в производных классах, и их небезопасно использовать в остальных случаях. Эту разницу можно выразить разбиением интерфейса классов `Window` на две части — защищенную и открытую:

```
class Window_with_border {
public:
        virtual void draw ();
        // ...
protected:
        void own_draw ();
        // ...какой-то вспомогательный код
private:
        // представление и т. д.
};
```

Производный класс может осуществлять доступ к защищенным членам базового класса только для объектов его собственного типа:

```
class Buffer {
protected:
        char a[128];
        // ...
};

class Linked_buffer : public Buffer { /* ... */ };

class Cyclic_buffer : public Buffer {
        // ...
        void f(Linked_buffer* p) {
                a[0] = 0;           // правильно: доступ к собственному защищенному
                                    // члену класса Cyclic_buffer
                p->a[0] = 0;        // ошибка: доступ к защищенному члену другого типа
        }
};
```

Такой подход предохраняет от довольно тонких ошибок, которые могли бы привести к тому, что один производный класс разрушал данные, принадлежащие другому производному классу.

15.3.1.1. Использование защищенных членов

Простая модель сокрытия данных «открытый/закрытый» хорошо работает для конкретных типов (§ 10.3). Однако при использовании производных классов существуют два вида пользователей класса: производные классы и «простая публика». Члены и друзья, реализующие операции класса, работают с объектами классов от имени этих пользователей. Модель «открытый/закрытый» позволяет программисту различать между разработчиками и всеми остальными, но она не предусматривает особого обслуживания для производных классов.

Члены, объявленные ***protected***, более подвержены злоупотреблениям, чем те, которые объявлены ***private***. В частности, объявление данных защищенными, обычно свидетельствует об ошибке на этапе проектирования. Помещение значительной части данных в общий класс, доступный для всех производных классов, приводит к риску разрушения этих данных. Более того, так же как и открытые, защищенные данные не просто реструктурировать ввиду сложности нахождения всех случаев их использования. Таким образом, защищенные данные приводят к проблемам сопровождения.

К счастью, вы не обязаны использовать защищенные данные; члены классов по умолчанию закрыты и, как правило, это является наилучшим вариантом. В моей практике всегда находились лучшие альтернативы, чем помещение значительного количества информации в общий класс для непосредственного доступа к ней из производных классов.

Обратите внимание, что все эти возражения не имеют большого значения для защищенных *функций*; защищенность — это прекрасный способ задания операций для использования в производных классах. Хорошим примером может служить ***Ival_slider*** из § 12.4.2. Если бы в этом примере класс реализации (точнее, его функции) был закрытым, дальнейшее создание производных классов было бы невозможно.

Примеры, иллюстрирующие доступ к членам, можно найти в § B.11.1.

15.3.2. Доступ к базовым классам

Аналогично членам класса базовый класс можно объявить закрытым (*private*), защищенным (*protected*) или открытым (*public*). Например:

```
class X : public B { /* ... */ };
class Y : protected B { /* ... */ };
class Z : private B { /* ... */ };
```

Открытое наследование делает производный класс подтипом базового; это наиболее распространенная форма наследования. Защищенное и закрытое наследование используются для выражения деталей реализации. Защищенные базовые классы полезны в иерархиях классов, в которых дальнейшее построение производных классов является нормой. Хорошим примером может служить *Ival_slider* из § 12.4.2. Закрытые базовые классы полезны в основном при определении класса, который предоставляет большие гарантии, чем его базовый класс, как бы «ужесточая» интерфейс базового класса. Например, шаблон класса вектора из указателей *Vector* добавляет проверку типа к своему базовому классу *Vector<void*>* (§ 13.5). Кроме того, если мы хотим убедиться, что проверяется каждый случай доступа к *Vec* (§ 3.7.2), мы должны определить базовый класс *Vec* закрытым (для предотвращения преобразования *Vec* в непроверяемый базовый класс):

```
template<class T> class Vec : private vector<T> { /* ... */ };  // вектор с проверкой выхода
                                                                // за диапазон
```

Спецификатор доступа может быть опущен. В этом случае базовый класс по умолчанию будет закрытым для производного класса и открытым для производной структуры. Например:

```
class XX  : B { /* ... */ },      // B — закрытое наследование
struct YY : B { /* ... */ },      // B — открытое наследование
```

Чтобы сделать код более читаемым, лучше использовать явный спецификатор.

Спецификатор доступа к базовому классу управляет доступом к членам базового класса и преобразованием указателей и ссылок из типа производного класса в тип базового класса. Рассмотрим класс *D*, производный от базового класса *B*:

- Если *B* является закрытым базовым классом, его открытые и защищенные члены могут быть использованы только функциями-членами и друзьями *D*. Только друзья и члены *D* могут преобразовать *D** в *B**.
- Если *B* является защищенным базовым классом, его открытые и защищенные члены могут быть использованы только функциями-членами и друзьями класса *D* и его производных классов. Только друзья и члены *D* и его производных классов могут преобразовать *D** в *B**.
- Если *B* является открытым базовым классом, его открытые члены могут быть использованы любой функцией. Кроме того, его защищенные члены могут быть использованы членами и друзьями класса *D* и его производных. Любая функция может преобразовать *D** в *B**.

Это в основном является переформулировкой правил для доступа к членам (§ 15.3). Мы выбираем способ доступа к базовым классам из тех же соображений, что и для членов. Например, я предпочел сделать *BBwindow* защищенным базовым классом для

Ival_slider (§ 12.4.2), потому что ***BBwindow*** является частью реализации *Ival_slider*, а не частью его интерфейса. Однако я не мог полностью скрыть ***BBwindow***, сделав его закрытым базовым классом, потому что я хотел оставить возможность дальнейшего построения классов, производных от *Ival_slider*, и этим производным классам потребовался бы доступ к реализации.

Примеры, иллюстрирующие доступ к базовым классам, можно найти в § B.11.2.

15.3.2.1. Множественное наследование и управление доступом

Если доступ к имени или базовому классу может быть осуществлен при помощи нескольких путей в решетке классов с множественным наследованием, то доступ разрешен только в том случае, если он разрешен по каждому из возможных путей. Например:

```
struct B {
        int m;
        static int sm;
        // ...
};

class D1 : public virtual B { /* ... */ };
class D2 : public virtual B { /* ... */ };
class DD  public D1, private D2 { /* ... */ };

DD* pd = new DD;
B* pb = pd;                 // правильно: доступ через D1
int i1 = pd->m;             // правильно: доступ через D1
```

Даже если доступ к некоторой сущности может быть осуществлен несколькими способами, тем не менее на нее можно в принципе ссылаться без появления неоднозначности. Например:

```
class X1 : public B { /* ... */ };
class X2 : public B { /* ... */ };
class XX : public X1, public X2 { /* ... */ };

XX* pxx = new XX;

int i1 = pxx->m;        // ошибка, неоднозначность: XX::X1::B::m или XX::X2::B::m
int i2 = pxx->sm;       // правильно: в XX есть только один B::sm
```

15.3.2.2. using-объявления и управление доступом

using-объявление не может быть использовано для получения доступа к дополнительной информации. Оно является механизмом предоставления уже доступной информации в более удобном для использования виде. С другой стороны, если доступ к информации имеется, его можно предоставить и другим пользователям. Например:

```
        int c;
};

class D : public B {
public:
        using B::a;         // ошибка: B::a закрыт
        using B::b;         // B::b становится общедоступным через D
};
```

Комбинации *using-объявления* с закрытым или защищенным наследованием можно использовать для предоставления только части интерфейса, предлагаемого классом. Например:

```
class BB : private B {          // предоставляет доступ к B::b и B::c, но не к B::a
        using B::b;
        using B::c;
};
```

См. также § 15.2.2.

15.4 Информация о типе на этапе выполнения

Вероятным использованием семейства классов ***Ival_box***, определенных в § 12.4, будет передача их в систему, управляющую экраном, которая передаст эти объекты обратно в программу при наступлении некоторого события. Так работает множество пользовательских интерфейсов. Однако система пользовательского интерфейса не знает ничего о нашем семействе ***Ival_box***. Интерфейсы системы задаются в терминах классов и объектов системы, а не в терминах классов нашего приложения. Так оно и должно быть. Однако это имеет неприятный побочный эффект, заключающийся в том, что мы теряем информацию о типе объектов, переданных системе и затем возвращенных нам.

Восстановление «потерянного» типа объекта требует, чтобы мы могли каким-либо образом спросить объект о его типе. При любой операции над объектом нам требуется указатель или ссылка подходящего типа. Следовательно, наиболее очевидной и полезной операцией по определению типа объекта на этапе выполнения является операция преобразования типа, которая возвращает корректный указатель, если объект имеет ожидаемый тип, и нулевой указатель — в противном случае. Оператор ***dynamic_cast*** именно это и делает. Например, предположим, что «система» вызывает обработчик ***my_event_handler*** () с указателем на то окно ***BBwindow***, где произошло некоторое событие. Я затем мог бы вызвать некоторый код в моем приложения, например с использованием функции ***do_something*** () класса ***Ival_box***:

```
void my_event_handler(BBwindow* pw)
{
        // pw указывает на Ival_box?
        if (Ival_box* pb = dynamic_cast<Ival_box*>(pw))
                pb->do_something();
        else {
                // Проблема! Неожиданное событие
        }
}
```

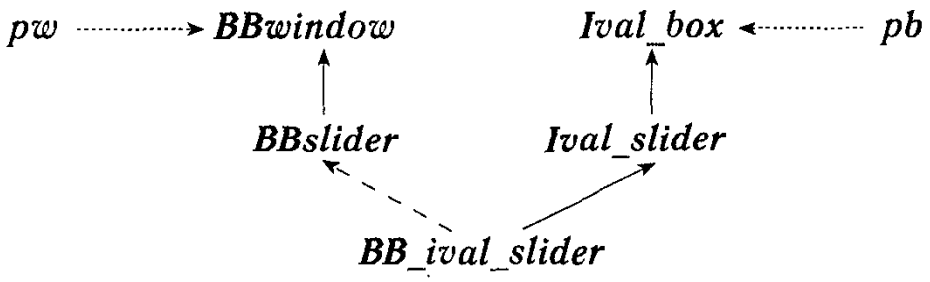
Происходящее можно объяснить следующим образом: ***dynamic_cast*** переводит с языка системы пользовательского интерфейса на язык приложения. Обратите внимание

на то, что в примере явно *не* указан действительный тип объекта. Объект будет иметь тип одного из классов семейства ***Ival_box***, скажем ***Ival_slider***, реализованный одним из видов ***BBwindow***, например, ***BBslider***. Не является необходимым, да это и нежелательно, делать действительный тип объекта явным во взаимодействии между «системой» и приложением. Интерфейс существует для представления существенных моментов взаимодействия. В частности, хорошо спроектированный интерфейс скрывает несущественные детали.

Графически, действие

```
pb = dynamic_cast<Ival_box*>(pw)
```

можно представить следующим образом:



Стрелки от ***pw*** и ***pb*** представляют указатели на передаваемый объект, а остальные стрелки представляют отношения наследования между различными частями передаваемого объекта.

Использование информации о типе во время выполнения обычно называют «информацией о типе на этапе выполнения» или сокращенно, RTTI (Run-Time Type Information).

Приведение из базового класса в производный часто называют *понижающим приведением* (downcast), потому что принято изображать дерево наследования растущим вниз из корня наверху. Аналогичным образом, приведение из производного класса в базовый называют *повышающим приведением* (upcast). Приведение между производными классами одного базового класса называют *перекрестным приведением* (crosscast).

15.4.1. Динамическое приведение dynamic_cast

Оператор ***dynamic_cast*** имеет два операнда: тип, заключенный в угловые скобки, и указатель (или ссылка), заключенный в круглые скобки.

Сначала рассмотрим случай с указателем:

```
dynamic_cast<T*>(p)
```

Если ***p*** типа ***T**** или типа ***D****, где ***T*** является базовым классом для ***D***, результат будет точно такой же, как при простом присваивании ***p*** указателю типа ***T****. Например:

```
class BB_ival_slider : public Ival_slider, protected BBslider {
    // ...
};

void f(BB_ival_slider* p)
{
    Ival_slider* pi1 = p;                                    // правильно
    Ival_slider* pi2 = dynamic_cast<Ival_slider*>(p);        // правильно
```

```
        BBslider* pbb1 = p;          // ошибка: BBslider — защищенный базовый класс
        BBslider* pbb2 = dynamic_cast<BBslider*> (p);    // правильно: pbb2 станет 0
}
```

Этот случай не представляет интереса. Однако утешительно сознавать, что ***dynamic_cast*** не допускает случайных нарушений правил доступа к закрытым и защищенным базовым классам.

Приведение ***dynamic_cast*** используется в тех случаях, когда правильность преобразования не может быть определена компилятором. В этом случае,

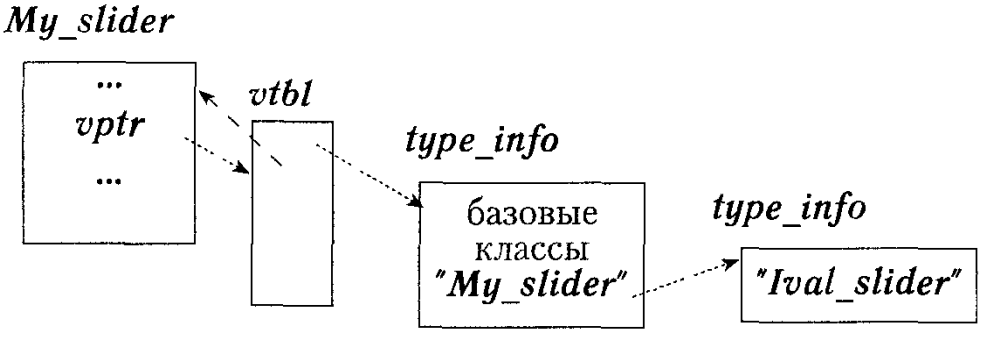
```
dynamic_cast<T *> (p)
```

смотрит на объект, на который указывает ***p*** (если указывает). Если это объект класса ***T*** или он имеет единственный базовый класс типа ***T***, ***dynamic_cast*** возвращает указатель типа ***T*** * на этот объект; в противном случае, возвращается ноль. Если значение ***p*** равно нулю, ***dynamic_cast<T *> (p)*** возвращает ноль. Обратите внимание на требование, что преобразование должно производится в однозначно идентифицируемый объект. Можно сконструировать примеры, когда преобразование не удастся, и возвратится ***0***, потому что объект, на который указывает ***p***, имеет более одного подобъекта, представляющего базовые классы типа ***T*** (см. § 15.4.2).

Для выполнения понижающего или перекрестного приведения требуется, чтобы аргумент ***dynamic_cast*** был ссылкой или указателем на полиморфный тип. Например:

```
class My_slider: public Ival_slider {      // полиморфный базовый класс
                                           // (Ival_slider имеет виртуальные функции)
        //...
},
class My_date  public Date {               // базовый класс не полиморфен
                                           // (у Date нет виртуальных функций)
        //...
};
void g (Ival_box* pb, Date* pd)
{
        My_slider* pd1 = dynamic_cast<My_slider*> (pb);    // правильно
        My_date* pd2 = dynamic_cast<My_date*> (pd);        // ошибка: Date —
                                                           // не полиморфный тип
}
```

Требование, что тип указателя должен быть полиморфным, упрощает реализацию ***dynamic_cast***, потому что в этом случае легче найти место для хранения необходимой информации о типе объекта. В типичной реализации «объект, содержащий информацию о типе» будет добавлен к самому объекту путем помещения указателя на информацию о типе в таблицу виртуальных функций объекта (§ 2.5.5). Например:



Пунктирная линия означает смещение, которое позволяет найти начало объекта при наличии только указателя на полиморфный подобъект. Ясно, что ***dynamic_cast*** можно реализовать эффективно. Все, что для этого нужно — несколько сравнений объектов ***type_info***, представляющих базовые классы; дорогостоящий поиск или сравнение строк не требуется.

Использование ***dynamic_cast*** только с полиморфными типами имеет смысл и с логической точки зрения. Если у объекта нет виртуальных функций, им нельзя безопасно манипулировать, не зная его конкретный тип. Следовательно, должны быть приняты меры предосторожности, чтобы такой объект не оказался в контексте, в котором его тип неизвестен. Если его тип известен, нет необходимости в использовании ***dynamic_cast***.

Результирующий тип ***dynamic_cast*** не обязан быть полиморфным. Это позволяет нам «завернуть» конкретный тип в полиморфный с целью, скажем, передачи объекта системе ввода/вывода и возврата его обратно (см. § 25.4.1) с последующим его извлечением. Например:

```
class Io_obj {                                  // базовый класс системы ввода/вывода
        virtual Io_obj* clone () = 0;
};

class Io_date : public Date, public Io_obj { };

void f (Io_obj* pio)
{
        Date* pd = dynamic_cast<Date*> (pio);
        //...
}
```

Приведение в ***void**** с помощью ***dynamic_cast*** можно использовать для определения адреса начала объекта полиморфного типа. Например:

```
void g (Ival_box* pb, Date* pd)
{
        void* pd1 = dynamic_cast<void*> (pb);   // правильно
        void* pd2 = dynamic_cast<void*> (pd);   // ошибка: Date — не полиморфный тип
}
```

Это полезно только при взаимодействии с низкоуровневыми функциями.

15.4.1.1. Динамическое приведение ссылок

Для осуществления полиморфного поведения доступ к объекту должен производиться через указатель или ссылку. При использовании ***dynamic_cast*** с типами указателей ***0*** означал неуспешное преобразование. Такое поведение нежелательно и недопустимо применительно к ссылкам.

Если результат является указателем, мы должны учитывать вероятность того, что он равен ***0***, то есть указатель не указывает ни на какой объект. Следовательно, результат применения ***dynamic_cast*** к указателю нужно всегда явно проверять. Для указателя ***p*** приведение ***dynamic_cast<T *> (p)*** можно интерпретировать как вопрос: «Объект, на который указывает ***p***, имеет тип ***T***?».

С другой стороны, мы законно можем предположить, что ссылка ссылается на некий объект. Следовательно, для ссылки ***r*** приведение ***dynamic_cast<T&> (r)*** является

не вопросом, а утверждением: «Объект, на который ссылается *r*, имеет тип *T*». Результат применения ***dynamic_cast*** к ссылке неявно проверяется самой реализацией ***dynamic_cast***. Если операнд динамического приведения к ссылке не принадлежит ожидаемому типу, генерируется исключение ***bad_cast***. Например:

```
void f(Ival_box* p, Ival_box& r)
{
        // p указывает на некий Ival_slider?
        if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {
                // использование is
        }
        else {
                // *p не ползунок (то есть не из семейства Ival_slider)
        }
        Ival_slider& is = dynamic_cast<Ival_slider&>(r);   // r ссылается на некий Ival_slider!
        // использование is
}
```

Различие между результатами неуспешного динамического преобразования указателя и ссылки отражает фундаментальное отличие между самими указателями и ссылками. Если пользователь хочет защититься от неудачных приведений к ссылке, необходимо предоставить подходящий обработчик. Например:

```
void g()
{
        try {
                // аргументы передаются как объекты Ival_box
                f(new BB_ival_slider, *new BB_ival_slider);
                // аргументы передаются как объекты Ival_box
                f(new Bbdial, *new Bbdial);
        }
        catch (bad_cast) {              // § 14.10
                // ...
        }
}
```

Первый вызов *f*() завершится нормально, а второй — генерирует исключение ***bad_cast***, которое будет перехвачено в *g*().

Явное сравнение указателя с нулем может быть (а значит, время от времени — будет) пропущено. Если это вас беспокоит, можете написать преобразующую функцию, которая генерирует исключение, а не возвращает *0* (§ 15.8[1]) в случае неудачного выполнения.

15.4.2. Навигация по иерархии классов

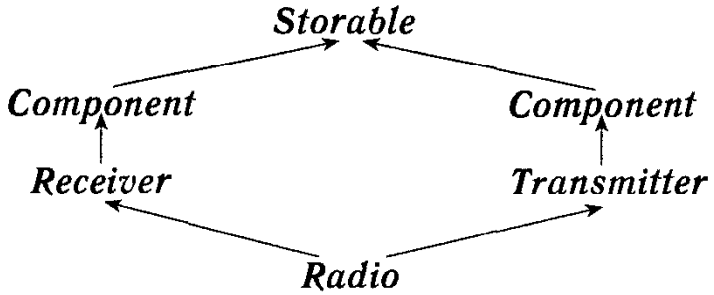
Когда используется только одиночное наследование, класс совместно с его производными классами образует дерево с корнем в единственном базовом классе. Это просто, но часто слишком ограничительно. При использовании множественного наследования не существует единственного корня. Само по себе это не слишком усложняет ситуацию. Однако если класс появляется в иерархии более одного раза, нам следует проявлять некоторую осторожность при обращении к объекту или объектам, которые представляют этот класс.

Естественно, мы пытаемся конструировать иерархии настолько простыми, насколько нам позволяет наше приложение (но не проще). Однако, создав нетривиальную иерархию, мы тут же сталкиваемся с необходимостью навигации по ней с целью нахождения класса с подходящим интерфейсом. Эта потребность проявляется в двух аспектах. Иногда мы хотим явно назвать объект базового класса или член базового класса; примерами могут служить § 15.2.3 и § 15.2.4.1. В других случаях мы хотим получить указатель на объект, представляющий базовый или производный класс объекта, при наличии указателя на завершенный объект или некоторый подобъект; примерами могут служить § 15.4 и § 15.4.1.

Рассмотрим способы навигации по иерархии классов с использованием приведений типа для получения указателя желаемого типа. В качестве иллюстрации имеющихся механизмов и правил их использования рассмотрим решетку классов, содержащую повторяющийся базовый класс и виртуальный базовый класс:

```
class Component : public virtual Storable { /* ... */ };
class Receiver : public Component { /* ... */ };
class Transmitter : public Component { /* ... */ };
class Radio : public Receiver, public Transmitter { /* ... */ };
```

Или в графическом виде:



В этом примере объект **Radio** содержит два подобъекта класса **Component**. Следовательно, динамическое приведение из **Storable** в **Component** внутри **Radio** неоднозначно и результатом будет **0**. Просто не существует способа распознать, какой **Component** программист имел в виду:

```
void h1 (Radio& r)
{
    Storable* ps = &r;
    // ...
    Component* pc = dynamic_cast<Component*> (ps);    // pc = 0
}
```

Эту неоднозначность в общем случае нельзя обнаружить на этапе компиляции:

```
void h2 (Storable* ps)        // ps может указывать на Component,
                              // а может и не указывать
{
    Component* pc = dynamic_cast<Component*> (ps);
    // ...
}
```

Такое определение неоднозначности требуется только для виртуальных базовых классов. В случае обычных базовых классов всегда существует единственный подобъект

(либо его вообще не существует) данного понижающего приведения (то есть приведения в сторону производного класса; § 15.4). Аналогичная неоднозначность возникает при повышающем приведении (то есть приведения в сторону базового класса; § 15.4), и такие неоднозначности обнаруживаются на этапе компиляции.

15.4.2.1. Статическое и динамическое приведение

Динамическое приведение ***dynamic_cast*** может преобразовать полиморфный виртуальный базовый класс в производный или «братский» класс (§ 15.4.1). Статическое приведение ***static_cast*** (§ 6.2.7) не анализирует объект, который оно приводит, поэтому оно не может этого сделать:

```
void g(Radio& r)
{
        // Receiver — обычный базовый класс для Radio
        Receiver* prec = &r;
        // правильно, не проверяется
        Radio* pr = static_cast<Radio*>(prec);
        // правильно, проверяется на этапе выполнения
        pr = dynamic_cast<Radio*>(prec);

        // Storable является виртуальным базовым классом для Radio
        Storable* ps = &r;
        // ошибка: нельзя выполнить приведение
        // из виртуального базового класса
        pr = static_cast<Radio*>(ps);
        // правильно, проверяется на этапе выполнения
        pr = dynamic_cast<Radio*>(ps);
}
```

Динамическое приведение требует наличия полиморфного операнда, потому что в неполиморфном объекте нет информации, которая может быть использована для нахождения объектов, для которых он представляет базовый класс. В частности, объект типа с ограничениями на способ размещения в памяти, задаваемыми некоторыми языками, — такими как Fortran или C — может использоваться в качестве виртуального базового класса. Для объектов таких типов доступна только статическая информация о типе. Однако информация, требуемая для определения типа на этапе выполнения, включает в себя информацию, необходимую для реализации ***dynamic_cast***.

Почему может возникнуть потребность в использовании ***static_cast*** при навигации по иерархии классов? При использовании ***dynamic_cast*** дополнительные накладные расходы на этапе выполнения невелики (§ 15.4.1). Более важно то, что существуют миллионы строк кода, написанных до появления ***dynamic_cast***. Этот код использует альтернативные способы обеспечения корректности преобразования, поэтому проверка осуществляемая ***dynamic_cast*** кажется избыточной. Однако такой код обычно написан с использованием приведений в стиле C (§ 6.2.7), и поэтому в нем часто остаются скрытые ошибки. Там где это возможно, используйте более безопасное динамическое приведение.

Компилятор не может делать предположений о памяти, на которую указывает ***void****. Из этого следует, что динамическое приведение, которое должно проанализировать объект для определения его типа, не может осуществить приведение из ***void****. В этом случае требуется статическое приведение. Например:

```
Radio* f(void* p)
{
    Storable* ps = static_cast<Storable*>(p);      // на совести программиста
    return dynamic_cast<Radio*>(ps);
}
```

Приведения ***dynamic_cast*** и ***static_cast*** учитывают модификатор ***const*** и управление доступом. Например:

```
class Users : private set<Person> { /* ... */ };

void f(Users* pu, const Receiver* pcr)
{
    static_cast<set<Person>*>(pu);          // ошибка: нарушение доступа
    dynamic_cast<set<Person>*>(pu);         // ошибка: нарушение доступа

    static_cast<Receiver*>(pcr);            // ошибка: нельзя «снять» const
    dynamic_cast<Receiver*>(pcr);           // ошибка: нельзя «снять» const

    Receiver* pr = const_cast<Receiver*>(pcr);   // правильно
    //...
}
```

Невозможно осуществить приведение в закрытый базовый класс, а «снятие ***const***» (или ***volatile***) требует ***const_cast*** (§ 6.2.7). И даже в этом случае, использование результата безопасно только в том случае, если объект не был изначально объявлен константным (или ***volatile***) (§ 10.2.7.1).

15.4.3. Конструирование и уничтожение объектов класса

Объект класса — это нечто большее, чем просто область памяти (§ 4.9.6). Объект класса строится из «сырой памяти» своими конструкторами и она снова становится «сырой памятью» после выполнения его деструкторов. Создание идет снизу вверх, уничтожение — сверху вниз; при этом объект класса является объектом в той мере, в какой он был создан или уничтожен. Этот факт отражен в правилах, описывающих RTTI, обработку исключений (§ 14.4.7) и виртуальные функции.

Исключительно неразумно полагаться на порядок создания и уничтожения, но тем не менее этот порядок можно наблюдать при помощи вызовов виртуальных функций, динамического приведения или использования ***typeid*** (§ 15.4.4) в момент, когда объект еще не завершен. Например, если конструктор класса ***Component*** из иерархии § 15.4.2 вызывает виртуальную функцию, будет вызвана версия, определенная в ***Storable*** или ***Component***, но не в ***Receiver***, ***Transmitter*** или ***Radio***. На этом этапе создания объект еще не является ***Radio***; он пока является частично созданным объектом. Лучше не вызывать виртуальные функции на этапе создания и уничтожения.

15.4.4. Typeid и дополнительная информация о типе

Оператор ***dynamic_cast*** удовлетворяет большинство потребностей в информации о типе объекта на этапе выполнения. Важным моментом является то, что он гарантирует, что код, написанный с его использованием, работает правильно с классами, производными от классов, явно указываемых программистом. Таким образом, ***dynamic_cast*** сохраняет гибкость и способность к расширению, присущую виртуальным функциям.

Однако в некоторых случаях очень важно знать точный тип объекта. Например, мы могли бы захотеть узнать имя класса объекта или способ его размещения в памяти. Для этой цели служит оператор ***typeid***. Он выдает объект, представляющий тип его операнда. Если бы оператор ***typeid*** был функцией, ее объявление выглядело бы примерно следующим образом:

```
class type_info,
const type_info& typeid (type_name) throw (),          // псевдообъявление
const type_info& typeid (expression) throw (bad_typeid), // псевдообъявление
```

То есть ***typeid*** () возвращает ссылку на тип стандартной библиотеки, называемый ***type_info***, определенный в <***typeinfo***>. По своему операнду *имя-типа* ***typeid*** () возвращает ссылку на ***type_info*** (информация о типе), который представляет *имя-типа*. При наличии операнда *выражения* (expression), ***typeid*** () возвращает ссылку на ***type_info***, который представляет тип объекта, обозначенного *выражением*. Оператор ***typeid*** () наиболее часто используется для нахождения типа объекта, на который указывает указатель или ссылка:

```
void f(Shape& r, Shape* p)
{
    typeid (r),     // тип объекта, на который ссылается r
    typeid (*p),    // тип объекта, на который указывает p
    typeid (p),     // тип указателя, то есть, Shape*
                    // (редко используется — скорее всего ошибка)
}
```

Если значение указателя или ссылки полиморфного типа равно ***0***, ***typeid*** () генерирует исключение ***bad_typeid***. Если операнд ***typeid*** () имеет неполиморфный тип или не является ***lvalue***, результат определяется во время компиляции без вычисления выражения операнда.

Независимая от реализации часть ***type_info*** выглядит следующим образом:

```
class type_info {
public
    virtual ~type_info ()                          // полиморфный

    bool operator== (const type_info&) const,      // можно сравнивать
    bool operator!= (const type_info&) const;
    bool before (const type_info&) const,          // упорядочение

    const char* name () const,                     // имя типа
private
    type_info (const type_info&),                  // предохраняет от копирования
    type_info& operator= (const type_info&),       // предохраняет от копирования
},
```

Функция ***before*** () позволяет сортировать информацию о типе ***type_info***. Нет никакой связи между отношениями упорядочения, определяемыми ***before*** (), и отношениями наследования.

Не гарантируется, что только один объект ***type_info*** существует для каждого типа в системе. В действительности, при использовании динамически компонуемых библиотек приложению сложно бывает избежать дублирования объектов ***type_info***. Сле-

довательно мы должны использовать == с объектами *type_info* для проверки равенства, а не применять == с указателями на такие объекты.

Иногда нам требуется знать точный тип объекта для выполнения некоторых стандартных действий со всем объектом (а не только с некоторой частью объекта, унаследованной от базового класса). В идеале такие действия должны быть представлены в виде виртуальных функций, что позволило бы не знать точный тип объекта. В некоторых случаях нельзя предполагать наличие общего интерфейса для всех объектов, с которыми мы имеем дело, поэтому становится необходима точная информация о типе (§ 15.4.4.1). Другим, намного более простым, примером использования имени класса может служить диагностический вывод:

```
#include<typeinfo>

void g(Component* p)
{
    cout << typeid(*p).name();
}
```

Символьное представление имени класса зависит от реализации. Эта С-строка располагается в системной области памяти, поэтому программист не должен пытаться уничтожить ее при помощи *delete*[].

15.4.4.1. Дополнительная информация о типе

Как правило, нахождение точной информации о типе является только первым шагом получения и использования более детальной информации об этом типе.

Давайте рассмотрим как приложение или инструментальное средство может сделать информацию о типе доступной пользователям на этапе выполнения. Допустим у меня есть инструментальное средство, которое генерирует описание размещения объектов каждого используемого класса. Я могу поместить эти описания в ассоциативный массив, чтобы пользователи могли получить доступ к этой информации:

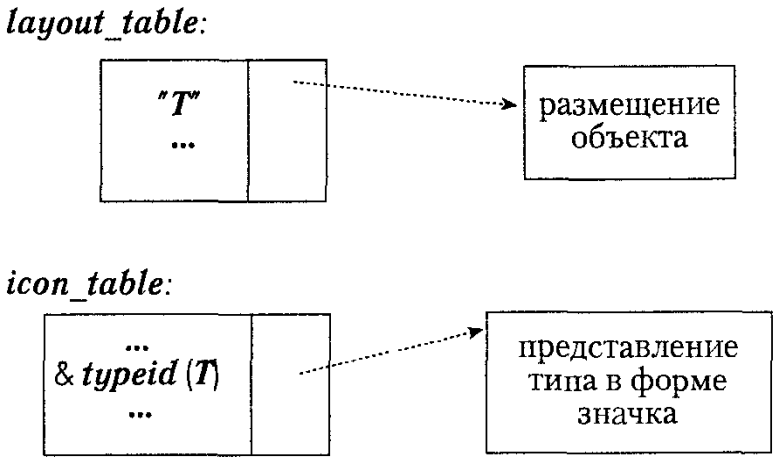
```
map<string, Layout> layout_table;

void f(B* p)
{
    Layouts& x = layout_table[typeid(*p).name()];
    // использование x
}
```

Кто-то может предоставить информацию совершенно другого характера:

```
struct TI_eq {
    bool operator()(const type_info* p, const type_info* q) { return *p==*q; }
};

struct TI_hash {
    int operator()(const type_info* p);        // вычислить хэш-значение (§ 17.6.2.2)
};

hash_map<const type_info*, Icon, hash_fct, TI_hash, TI_eq> icon_table;        // § 17.6
```

```
void g (B* p)
{
        Icon& i = icon_table[&typeid (*p)];
        // использование i
}
```

Такой способ ассоциации идентификаторов ***typeid*** с информацией предоставляет возможность разным людям или инструментальным средствам связывать различную информацию с типами совершенно независимо друг от друга:



Этот факт имеет большое значение, потому что вероятность того, что можно предоставить единственный набор информации, который удовлетворит всех пользователей, равна нулю.

15.4.5. Разумное и неразумное использование RTTI

Информацию о типе на этапе выполнения следует использовать только в случае реальной необходимости. Статическая (на этапе компиляции) проверка безопасней, влечет меньшие накладные расходы и как правило, приводит в результате к лучше структурированным программам. Можно использовать RTTI, например для написания тонко замаскированных ***switch-инструкций:***

```
// неразумное использование информации о типе во время выполнения:
void rotate (const Shape& r)
{
        if (typeid (r) == typeid (Circle)) {
                // ничего не делать
        }
        if (typeid (r) == typeid (Triangle)) {
                // повернуть треугольник
        }
        if (typeid (r) == typeid (Square)) {
                // повернуть квадрат
        }
        //...
}
```

Использование ***dynamic_cast*** вместо ***typeid*** незначительно улучшит этот код.

К сожалению, это не надуманный пример — такой код на самом деле пишется. Многие люди, пользовавшиеся языками типа C, Pascal, Modula-2 или Ada, не могут устоять перед искушением организовать код в виде ***switch-инструкции***. Как правило, следует пытаться противостоять этому стремлению. В большинстве случаев, когда на этапе выполнения требуются различные действия в зависимости от типа, вместо RTTI лучше воспользоваться виртуальными функциями (§ 2.5.5, § 12.2.6).

Много примеров разумного использования RTTI встречается, когда некоторый служебный код выражается в терминах одного класса, и пользователь хочет добавить функциональность при помощи производного класса. Примером может служить ***Ival_box*** из § 15.4. Если пользователь имеет возможность и желание модифицировать определения библиотечных классов, скажем ***BBwindow***, то можно избежать использования RTTI, в противном случае без него не обойтись. Даже если пользователь имеет желание модифицировать базовые классы, подобные модификации могут породить дополнительные проблемы. Например, возможно придется сделать фиктивные реализации виртуальных функций в классах, в которых эти функции либо не нужны либо не имеют смысла. Эта проблема обсуждается более подробно в § 24.4.3. Пример использования RTTI для реализации простой системы объектного ввода/вывода можно найти в § 25.4.1.

Люди с большим опытом использования языков, которые в значительной степени полагаются на динамическую проверку типов, таких как Smalltalk или Lisp, испытывают искушение воспользоваться RTTI совместно с чрезмерно общими типами. Рассмотрим пример:

```
// неразумное использование информации о типе во время выполнения:
class Object { /* ... */ };                    // полиморфный
class Container : public Object {
public:
        void put (Object*);
        Object* get ();
        // ...
};
class Ship : public Object { /* ... */ };
Ship* f (Ship* ps, Container* c)
{
        c->put (ps);
        // ...
        Object* p = c->get ();
        if (Ship* q = dynamic_cast<Ship*> (p)) {  // проверка во время выполнения
                return q;
        }
        else {
                // сделать что-либо еще (как правило, обработка ошибки)
        }
}
```

В этом примере, класс ***Object*** — ненужный артефакт реализации. Он является чрезмерно общим, потому что не соответствует какой-либо абстракции прикладной области и заставляет прикладного программиста пользоваться абстракциями уровня реализации. Проблемы такого рода часто решаются значительно лучше при помощи шаблонов контейнеров, которые содержат указатели только одного вида:

```
Ship* f(Ship* ps, list<Ship*>& c)
{
        c.push_front(ps);
        //...
        return c.pop_front();
}
```

Совместно с виртуальными функциями такой метод годится в большинстве случаев.

15.5. Указатели на члены

Многие классы предоставляют простые и очень общие интерфейсы, которые предполагается вызывать несколькими различными способами. Например, во многих «объектно-ориентированных» пользовательских интерфейсах определяется набор запросов, на которые каждый представленный на экране объект должен быть готов отреагировать. Кроме того, такие вызовы могут осуществляться, непосредственно или косвенно, из других программ. Рассмотрим простой вариант этой идеи:

```
class Std_interface {
public:
        virtual void start() = 0;
        virtual void suspend() = 0;
        virtual void resume() = 0;
        virtual void quit() = 0;
        virtual void full_size() = 0;
        virtual void small() = 0;

        virtual ~Std_interface() { }
};
```

Точное значение каждой операции определяется объектом, для которого она вызывается. Часто между пользователем или программой, выдающей запросы, и объектом, их получающим, находится программная прослойка. В идеале, этот промежуточный код не должен ничего знать об индивидуальных операциях, таких как ***resume*** () и ***full_size*** (). В противном случае, промежуточный код пришлось бы модифицировать при каждом изменении набора операций. Следовательно, подобный промежуточный код должен просто передавать получателю запроса некоторые данные, представляющие операции, которые должны быть вызваны, от его источника.

Одним из простых способов реализации этого подхода является пересылка строкового представления операции, которая должна быть вызвана. Например, чтобы вызвать ***suspend*** (), мы могли бы послать строку ***"suspend"***. Однако, кто-то должен создать эту строку и кто-то — декодировать, чтобы определить, какой операции она соответствует (или никакой не соответствует). Порой это кажется неудобным и сложным способом. Вместо этого мы могли бы посылать просто целые числа, соответствующие операциям. Например, ***2*** может означать ***suspend*** (). Однако, хотя числа и удобны для компьютера, для людей их значение неочевидно. И мы по-прежнему вынуждены писать код для определения того, что ***2*** означает ***suspend*** (), и для вызова ***suspend*** ().

C++ предлагает средство косвенной ссылки на член класса. Указатель на член является значением, идентифицирующим член класса. Вы можете его рассматривать как позицию члена в объекте класса, но конечно же, компилятор принимает в расчет различия между данными, виртуальными функциями, невиртуальными функциями и т. д.

Рассмотрим интерфейс ***Std_interface***. Если я хочу вызвать функцию ***suspend*** () для некоторого объекта, при этом не указывая явно ***suspend*** (), мне потребуется указатель на член ***Std_interface::suspend*** (). Кроме того, мне потребуется указатель или ссылка на объект, к которому нужно применить операцию ***suspend*** (). Рассмотрим тривиальный пример:

```
typedef void (Std_interface::*Pstd_mem)(),    // указатель на функцию-член
void f(Std_interface* p)
{
    Pstd_mem s = &Std_interface::suspend;
    p->suspend(),                  // прямой вызов
    (p->*s)(),                     // вызов через указатель на член
}
```

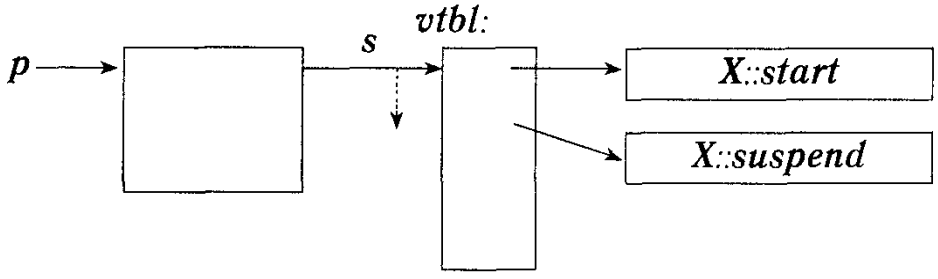
Указатель на член можно получить при помощи применения оператора получения адреса & к полностью квалифицированному имени члена класса, например, ***&Std_interface::suspend***. Переменная типа «указатель на член класса ***X***» объявляется с использованием формы ***X***::*.

С целью компенсации нечитабельности синтаксиса объявлений С обычно пользуются ***typedef***. Однако обратите внимание, что синтаксис объявления ***X***::* точно соответствует традиционному использованию * в объявлении.

Указатель на член ***m*** можно использовать в комбинации с объектом. Операторы ->* и .* позволяют программисту выразить такую комбинацию. Например, ***p->*m*** связывает ***m*** с объектом, на который указывает ***p***, а ***obj.*m*** связывает ***m*** с объектом ***obj***. Результат можно использовать в соответствии с типом ***m***. Невозможно сохранить результат операций ->* или .* для его дальнейшего использования.

Естественно, если бы мы знали, какой член нам нужно вызвать, мы могли бы вызвать его непосредственно, а не затевать эту путаницу с указателями. Также как и указатели на обычные функции, указатели на функции-члены используются тогда, когда возникает необходимость сослаться на функцию, имя которой неизвестно. Однако в отличие от указателя на переменную или обыкновенную функцию, указатель на член не является просто указателем на область памяти. Он больше соответствует смещению в структуре или индексу в массиве. Сочетание указателя на член с указателем на соответствующий объект дает то, что идентифицирует конкретный член конкретного объекта.

Это можно представить в графическом виде следующим образом:



Так как указатель на виртуальную функцию-член (в нашем примере, ***s***) является в некотором смысле смещением, он не зависит от расположения объекта в памяти. Поэтому указатель на виртуальный член можно безопасно передавать из одного адресного пространства в другое, при условии что в них обоих объект размещен

одинаковым образом. Также как и указатели на обыкновенные функции, указатели на невиртуальные функции-члены нельзя передавать в другое адресное пространство.

Обратите внимание на то, что функции, вызываемые через указатели, могут быть виртуальными. Например, когда мы вызываем ***suspend*** () через указатель на функцию, вызывается версия ***suspend*** (), соответствующая объекту, к которому применялся указатель на функцию. Это является существенным аспектом механизма указателей на функции.

Интерпретатор может использовать указатели на члены для вызова функций, представленных в строковом виде:

```
map<string, Std_interface*> variable;
map<string, Pstd_mem> operation;

void call_member(string var, string oper)
{
        (variable[var]->*operation[oper])();          // var.oper()
}
```

Пример небезопасного использования указателей на функции-члены можно найти в ***mem_fun*** () (§ 3.8.5, § 18.4).

Статический член не ассоциируется с конкретным объектом, так что указатель на статический член похож на обыкновенный указатель. Например:

```
class Task {
        // ...
        static void schedule();
};

void (*p)() = &Task::schedule;                // правильно
void (Task::* pm)() = &Task::schedule;        // ошибка: обычный указатель
                                              // присваивается указателю на член
```

Указатели на члены данных описаны в § B.12.

15.5.1. Базовые и производные классы

Производный класс по крайней мере содержит члены, которые он унаследовал от базовых классов. Часто он содержит много дополнительных членов. Из этого следует, что мы можем безопасно присвоить указатель на член базового класса указателю на член производного класса, но не наоборот. Это свойство часто называют *контравариацией*. Например:

```
class text : public Std_interface {
public:
        void start();
        void suspend();
        // ...
        virtual void print();
private:
        vector s;
};

void (Std_interface::* pmi)() = &text::print;     // ошибка
void (text::*pmt)() = &Std_interface::start;      // правильно
```

Создается впечатление, что правило контрвариации имеет смысл, противоположный правилу, которое гласит, что мы можем присвоить указатель на производный класс указателю на его базовый класс. В действительности, оба правила нужны для фундаментальной гарантии того, что указатель ни при каких условиях не укажет на объект, который не реализует (по крайней мере) свойств, подразумеваемых типом этого указателя. В нашем случае ***Std_interface***::* можно использовать с любым ***Std_interface***, но в общем случае не все эти объекты имеют тип ***text***. Следовательно, у них нет члена ***text***::***print***, с помощью которого мы пытались инициализировать ***pmi***. Отказывая в инициализации, компилятор предохраняет нас от ошибки на этапе выполнения.

15.6. Свободная память

Можно самостоятельно управлять памятью для класса, определив собственные ***operator new*** () и ***operator delete*** () (§ 6.2.6.2). Однако замена глобальных ***operator new*** () и ***operator delete*** () — занятие не для слабонервных. В конце концов, какой-нибудь другой пользователь может рассчитывать на некоторые аспекты поведения по умолчанию или даже реализовать собственные версии этих функций.

Более ограниченный и часто более надежный подход состоит в реализации этих операций для конкретного класса. Этот класс может являться базовым для множества производных классов. Например, мы могли бы реализовать специализированные операции выделения и освобождения памяти для класса ***Employee*** из § 12.2.6 и всех его производных классов:

```
class Employee {
        // ...
public:
        // ...
        void operator new (size_t);
        void operator delete (void*, size_t);
};
```

Операторы-члены ***new*** () и ***delete*** () неявно являются статическими членами. Следовательно, у них нет указателя ***this*** и они не изменяют объект. Они предоставляют память, которую конструктор может инициализировать, а деструктор — очистить.

```
void* Employee.operator new (size_t s)
{
        // выделить s байт памяти и вернуть указатель на выделенную область
}

void Employee::operator delete (void* p, size_t s)
{
        if (p) {        // удаляем только если p!=0; см. § 6.2.6, § 6.2.6.2
                        // полагаем, что p указывает на s байтов памяти, выделенной
                        // Employee::operator new() и освобождаем эту память
                        // для дальнейшего использования
        }
}
```

Использование загадочного до сих пор аргумента типа ***size_t*** становится очевидным. Он является размером фактически уничтожаемого объекта. При удалении «просто» ***Employee*** аргумента имеет значение ***sizeof*** (***Employee***); при удалении ***Manager*** значе-

ние аргумента равно ***sizeof(Manager)***. Это позволяет конкретной реализации операции выделения памяти не хранить информацию о размере при каждом выделении памяти. Естественно, реализация этой операции для конкретного класса может хранить эту информацию (стандартная функция выделения памяти должна это делать) и игнорировать аргумент типа ***size_t*** в ***operator delete()***. Однако такой подход затрудняет более эффективную, в смысле скорости и требуемой памяти, реализацию распределителя памяти общего назначения.

Как компилятор узнает, как передать истинный размер в ***operator delete()***? Если в операции ***delete*** указан тип, соответствующий реальному типу объекта, — задача проста. Однако, это не всегда так:

```
class Manager : public Employee {
        int level;
        // ...
};

void f()
{
        Employee* p = new Manager;      // проблема: истинный тип потерян
        delete p;
}
```

В этом случае компилятор не знает реального размера. Так же как и при удалении массива, требуется помощь от пользователя. Это производится добавлением виртуального деструктора к базовому классу, ***Employee***:

```
class Employee {
public:
        void* operator new(size_t);
        void operator delete(void*, size_t);
        virtual ~Employee();
        // ...
};
```

Подойдет даже «пустой» деструктор:

```
Employee::~Employee() { }
```

В принципе, освобождение памяти осуществляется тогда внутри деструктора (который знает размер). Более того, наличие деструктора в ***Employee*** дает гарантию, что каждый производный класс будет иметь деструктор (и выдавать правильную информацию о размере), даже если в нем отсутствуют явно определенные пользователем деструкторы. Например:

```
void f()
{
        Employee* p = new Manager;
        delete p;          // теперь правильно (Employee — полиморфен)
}
```

Выделение памяти осуществляется при помощи (сгенерированного компилятором) вызова:

```
Employee::operator new(sizeof(Manager))
```

а освобождение памяти — при помощи (тоже сгенерированного компилятором) вызова:

```
Employee::operator delete (p, sizeof (Manager))
```

Другими словами, если вы хотите иметь пару для выделения/освобождения памяти, которая корректно работает с производными классами, вы должны либо предоставить виртуальный деструктор в базовом классе, либо воздержаться от использования аргумента типа *size_t* при освобождении памяти. Естественно, можно было спроектировать этот аспект языка таким образом, чтобы вы не имели перечисленных проблем. Однако при этом вы бы не имели и возможности оптимизации, присущей менее безопасным системам.

15.6.1. Выделение памяти под массив

Функции *operator new* () и *operator delete* () дают возможность пользователю заместить выделение и освобождение памяти для индивидуальных объектов; *operator new*[] () и *operator delete*[] () играют ту же роль для массивов. Например:

```
class Employee {
public:
        void* operator new[] (size_t);
        void operator delete[] (void*);
        // ...
};

void f(int s)
{
        employee* p = new Employee[s];
        // ...
        delete[] p;
}
```

Память будет выделяться при помощи вызова,

```
Employee::operator new[] (s*sizeof (Employee)+delta)
```

где *delta* — некоторая необходимая дополнительная память, объем которой зависит от реализации. Освобождение памяти будет осуществляться при помощи вызова:

```
Employee::operator delete[] (p);     // освобождает s*sizeof(Employee)+delta байт
```

Количество элементов (*s*) система «помнит». В отличие от одноаргументной двухаргументная форма функции *delete*[] () позволяет вызывать ее со вторым аргументом типа *s*sizeof (Employee)+delta*.

15.6.2. «Виртуальные конструкторы»

После знакомства с виртуальными деструкторами возникает очевидный вопрос: «Может ли конструктор быть виртуальным?». Краткий ответ — нет; более подробный — тоже нет, но вы легко может получить желаемый эффект.

Для того чтобы создать объект, конструктор должен знать его точный тип. Следовательно, конструктор не может быть виртуальным. Более того, конструктор является не совсем обычной функцией. В частности, он взаимодействует с процедурами управления памятью способом, недоступным обычным функциям-членам. Как следствие, вы не можете получить указатель на конструктор.

Оба ограничения можно обойти, определив функцию, которая вызывает конструктор и возвращает созданный объект. Это хорошо, так как часто возникает необходимость в создании объекта, точный тип которого не известен. Примером класса, специально спроектированного для этой цели, может служить ***Ival_box_maker*** (§ 12.4.4). Сейчас я представлю другую вариацию этой идеи, где объекты класса могут создавать клоны (копии) самих себя или новые объекты своего типа. Рассмотрим пример:

```
class Expr {
public:
        Expr ();                        // конструктор по умолчанию
        Expr (const Expr&);             // копирующий конструктор

        virtual Expr* new_expr () { return new Expr (); }
        virtual Expr* clone () { return new Expr (*this); }
        // ...
},
```

Так как функции вроде ***new_expr*** () и ***clone*** () являются виртуальными, и они (косвенно) создают объекты, их часто называют «виртуальными конструкторами» — пример странности естественного языка. На самом деле, каждая из них просто использует конструктор для создания подходящего объекта.

Для возвращения объекта собственного типа производный класс может заместить ***new_expr*** () и/или ***clone*** ():

```
class Cond . public Expr {
public:
        Cond ();
        Cond (const Cond&);

        Cond* new_expr () { return new Cond (); }
        Cond* clone () { return new Cond (*this); }
        // ...
};
```

Это означает, что если имеется объект класса ***Expr***, пользователь может создать новый объект «точно такого же типа». Например:

```
void user (Expr* p)
{
        Expr* p2 = p->new_expr ();
        // ...
}
```

Указатель, присвоенный ***p2***, имеет корректный, но неизвестный тип.

Значения, возвращаемые ***Cond::new_expr*** () и ***Cond::clone*** (), имеют тип ***Cond****, а не ***Expr****. Это позволяет создавать копии ***Cond*** без потери информации о типе. Например:

```
void user2 (Cond* pc, Expr* pe)
{
        Cond* p2 = pc->clone ();
        Cond* p3 = pe->clone ();        // ошибка
        // ...
}
```

Тип замещающей функции должен быть такой же, как тип виртуальной функции, которую она замещает, за исключением того, что допускаются некоторые ослабления по отношению к типу возвращаемого значения. Например, если исходный тип возвращаемого значения был ***B****, то тип возвращаемого значения замещающей функции может быть ***D**** при условии, что ***B*** является открытым базовым классом для ***D***. Аналогично, вместо ***B***& тип возвращаемого значения может быть ослаблен до ***D***&.

Обратите внимание, что подобные правила «ослабления типа» для аргументов привели бы к нарушениям типов (см. § 15.8[12]).

15.7. Советы

[1] Пользуйтесь обычным множественным наследованием, чтобы выразить объединение свойств; § 15.2, § 15.2.5.

[2] Пользуйтесь множественным наследованием для отделения деталей реализации от интерфейса; § 15.2.5.

[3] Используйте виртуальные базовые классы для выражения некой общности, присущей некоторым, но не всем классам иерархии; § 15.2.5.

[4] Избегайте явного преобразования типа (приведения); § 15.4.5.

[5] Пользуйтесь динамическим приведением ***dynamic_cast***, когда навигация по иерархии неизбежна; § 15.4.1.

[6] Отдавайте предпочтение ***dynamic_cast*** по отношению к идентификатору типа ***typeid***; § 15.4.4.

[7] Отдавайте предпочтение ***private*** по отношению к ***protected***; § 15.3.1.1.

[8] Не объявляйте члены данных защищенными; § 15.3.1.1.

[9] Если класс определяет ***operator delete*** (), в нем должен присутствовать виртуальный деструктор; § 15.6.

[10] Не вызывайте виртуальные функции в процессе создания или уничтожения объекта; § 15.4.3.

[11] Умеренно пользуйтесь явными квалификаторами для разрешения перегрузки имен членов и старайтесь применять их в замещающих функциях; § 15.2.1.

15.8. Упражнения

1. (*1) Напишите шаблон ***ptr_cast***, который работает аналогично ***dynamic_cast*** за тем исключением, что вместо возвращения ***0***, он возбуждает исключение ***bad_cast***.
2. (*2) Напишите программу, которая иллюстрирует влияние последовательности вызовов конструкторов на состояние объекта (с точки зрения RTTI). Аналогичным образом проиллюстрируйте уничтожение объекта.
3. (*3.5) Реализуйте версию настольной игры Реверси (Reversi/Otello). Каждый игрок может быть либо человеком, либо компьютером. Сфокусируйтесь на том, чтобы программа работала корректно, и поднимите качество игры компьютера до достойного уровня.
4. (*3) Сделайте более качественный интерфейс для игры из § 15.8[3].
5. (*3) Определите класс графических объектов с разумным набором операций, который мог бы служить в качестве базового класса библиотеки графических объектов; посмотрите какие операции реализованы в реальной графической библиотеке. Определите класс объектов для работы с базами данных с подходящим набором опера-

ций, который мог бы использоваться в качестве базового класса для объектов, хранящихся как последовательность полей в базе данных; посмотрите какие операции реализованы в реальной библиотеке баз данных. Определите графический объект, связанный с базой данных, с использованием и без использования множественного наследования и обсудите относительные преимущества каждого подхода.

6. (*2) Напишите версию операции ***clone*** () из § 15.6.2, которая может поместить скопированный объект в заданную область памяти ***Arena*** (см. § 10.4.11), переданную ей в качестве аргумента. Реализуйте класс для работы с фиксированными областями памяти («аренами») в виде класса, производного от ***Arena***.
7. (*2) Не заглядывая в эту книгу, напишите как можно больше ключевых слов C++.
8. (*2) Напишите корректную программу на C++, содержащую последовательность по крайней мере из десяти идущих подряд различных ключевых слов, не разделенных идентификаторами, операторами, символами пунктуации и т. д.
9. (*2.5) Изобразите правдоподобное распределение памяти для объектов класса ***Radio*** из § 15.2.3.1. Объясните, как мог бы быть реализован вызов виртуальной функции.
10. (*2) Изобразите правдоподобное распределение памяти для объектов класса ***Radio*** из § 15.2.4. Объясните, как мог бы быть реализован вызов виртуальной функции.
11. (*3) Подумайте, как можно реализовать динамическое приведение ***dynamic_cast***. Спроектируйте и реализуйте шаблон ***dcast***, который ведет себя как ***dynamic_cast***, но использует при этом только данные и функции, определенные вами. Убедитесь, что вы можете добавить новые классы в систему, не меняя при этом определение ***dcast*** или ранее написанных классов.
12. (*2) Предположим, что правила проверки типов для аргументов были ослаблены, аналогично правилам для типов возвращаемых значений таким образом, что функция с аргументом ***Derived**** (указатель на производный класс) может заместить функцию с аргументом ***Base**** (указатель на базовый класс). Затем напишите программу, которая может разрушить объект класса ***Derived*** без использования приведения типа. Опишите безопасное ослабление правил замещения для типов аргументов.

ЧАСТЬ ТРЕТЬЯ

Стандартная библиотека

В этой части описывается стандартная библиотека C++. Читатель знакомится с построением библиотеки и с ключевыми приемами для ее реализации. Цель этой главы — объяснить, как следует пользоваться стандартной библиотекой, в общих чертах продемонстрировать полезные приемы проектирования и программирования, а также показать, каким образом расширять библиотеку в соответствии с замыслом ее создания.

16

Организация библиотеки и контейнеры

Это было ново. Это было своеобразно.
Это было просто. Это должно было получиться!
— Г. Нельсон

Критерии проектирования стандартной библиотеки — организация библиотеки — стандартные заголовочные файлы — языковая поддержка — проектирование контейнеров — итераторы — базовые контейнеры — STL-контейнеры — вектор (*vector*) — итераторы — доступ к элементам — конструкторы — модификаторы — операции со списками — размер и емкость — *vector<bool>* — советы — упражнения.

16.1. Проектирование стандартной библиотеки

Что должно быть в стандартной библиотеке? Для программиста идеалом было бы найти в библиотеке все интересные, нужные и достаточно общие классы, функции, шаблоны и т. д. Однако вопрос не в том «Что должно быть в *некоторой* библиотеке?», но «Что должно быть в *стандартной* библиотеке?» Ответ «Все!» в первом приближении имеет смысл для первого вопроса, но не для второго. Стандартная библиотека — это нечто такое, что должно быть обеспечено в каждой реализации языка, чтобы потом все программисты могли на нее опираться.

Стандартная библиотека C++:

[1] Обеспечивает поддержку свойств языка, таких как управление памятью (§ 6.2.6) и информация о типах во время выполнения (§ 15.4).

[2] Предоставляет информацию о зависящих от реализации аспектах языка, таких, например, как максимальное значение *float* (§ 22.2).

[3] Предоставляет функции, которые не могут быть написаны оптимально для всех систем собственно на языке C++, например *sqrt*() (§ 22.3) или *memmove*() (§ 19.4.6).

[4] Предоставляет программисту нетривиальные средства, на которые он сможет рассчитывать, заботясь о переносимости такие, как списки (§ 17.2.2), отображения (ассоциативные массивы) (§ 17.4.1), функции сортировки (§ 18.7.1) и потоки ввода/вывода (глава 21).

[5] Дает основу для расширения своих возможностей, такую как соглашения и средства поддержки, которые позволяют пользователю обеспечить ввод/вывод для определяемых им типов в стиле ввода/вывода для встроенных типов.

[6] Служит общим фундаментом для других библиотек.

Кроме того, стандартная библиотека предоставляет некоторые программные средства — например, генератор случайных чисел (§ 22.7) — просто потому, что так принято, и это удобно.

Проект библиотеки первоначально определялся последними тремя ролями, которые тесно связаны между собой. Например, переносимость обычно является важным критерием проектирования для специальных библиотек, а общие контейнерные типы, такие как списки и ассоциативные массивы, существенны для удобной связи между раздельно разрабатываемыми библиотеками.

С точки зрения проектирования особенно важна последняя роль, поскольку она помогает ограничить область применения стандартной библиотеки и налагает ограничения на ее средства. Например, стандартной библиотекой предоставляются строки и списки. Если бы их не было, независимо разработанные библиотеки могли бы общаться между собой, только используя встроенные типы. Однако распознавание образов и графика стандартной библиотекой не обеспечиваются. Эти возможности, очевидно, весьма полезны, но они редко участвуют непосредственно во взаимодействии независимо разработанных библиотек.

Если какое-то средство не нужно для обеспечения хотя бы одной из вышеперечисленных ролей, его можно оставить за пределами стандартной библиотеки. К счастью или нет, оставление чего-то за пределами стандартной библиотеки открывает другим библиотекам возможность конкурировать при ее реализации.

16.1.1. Проектные ограничения

Задачи стандартной библиотеки налагают на нее ряд проектных ограничений. Предлагаемые стандартной библиотекой C++ средства спроектированы так, чтобы:

[1] Быть важными и доступными для каждого студента и профессионального программиста, в том числе для создателей других библиотек.

[2] Прямо или косвенно использоваться всеми программистами для решения всех задач, которые связаны с целями библиотеки.

[3] Быть достаточно эффективными, чтобы при реализации будущих библиотек обеспечить достойную альтернативу функциям, классам и шаблонам, программируемым вручную.

[4] Быть независимыми от алгоритмов или предоставлять пользователю возможность задавать алгоритм в качестве аргумента.

[5] Быть примитивными в математическом смысле. То есть компонента, которая играет две мало связанные между собой роли, будет почти наверняка вызывать лишние затраты по сравнению с двумя отдельными компонентами, призванными играть каждый свою одну четко определенную роль.

[6] Быть удобными, эффективными и достаточно безопасными при использовании в большинстве типичных случаев.

[7] Быть завершенными в том, что делают. Стандартная библиотека может оставить множество функций другим библиотекам, но если уж она взялась за какую-то задачу, то должна обеспечить достаточную функциональность, чтобы отдельным пользователям и разработчикам не приходилось заменять ее средства.

[8] Хорошо сочетаться и дополнять встроенные типы и операции.

[9] Быть безопасными по умолчанию с точки зрения типов.

[10] Поддерживать общепринятые стили программирования.
[11] Быть способными к расширению, чтобы работать с типами, определяемыми пользователем, так же, как со встроенными типами и типами из стандартной библиотеки.

Например, встраивание критериев сравнения в функцию сортировки недопустимо, поскольку те же данные могут сортироваться в соответствии с какими-либо другими критериями. Вот почему стандартная библиотечная функция C ***qsort***() получает функцию сравнения в качестве аргумента, а не использует нечто фиксированное — скажем, оператор < (§ 7.7). С другой стороны, излишние затраты, налагаемые вызовом функции для каждого сравнения, компрометируют ***qsort***() с точки зрения дальнейшего построения библиотеки. Почти для каждого типа данных легко реализовать сравнение без лишних затрат на вызов функции.

Серьезны ли эти затраты? Вероятно, в большинстве случаев нет. Однако в некоторых алгоритмах вызовы функций могут занять значительное время и заставить пользователя искать альтернативу. Задание критерия для сравнения через аргумент шаблона, описанное в § 13.4, решает эту проблему. Данный пример иллюстрирует конфликт между эффективностью и универсальностью. От стандартной библиотеки не просто требуется выполнять свои задачи, она также должна выполнять их достаточно эффективно, чтобы у пользователя не возникло соблазна использовать собственные механизмы. Иначе разработчики более продвинутых средств, чтобы остаться конкурентоспособными, будут вынуждены отказаться от стандартной библиотеки. Это добавило бы хлопот разработчику библиотеки и серьезно усложнило бы жизнь пользователям, желающим оставаться независящими от платформы или использовать несколько раздельно разработанных библиотек.

Требования «примитивности» и «удобства при типовом использовании» кажутся противоречивыми. Последнее требование запрещает излишнюю оптимизацию стандартной библиотеки для общих случаев. Однако необходимо иметь возможность включать в стандартную библиотеку (в дополнение к примитивным возможностям, а не взамен) компоненты, служащие часто встречающимся (но не примитивным) целям. Культ ортогональности не должен удерживать нас от того, чтобы сделать жизнь новичка или случайного пользователя удобной. И мы не должны из соображений ортогональности оставлять действия компонент по умолчанию неясными или опасными.

16.1.2. Организация стандартной библиотеки

Средства стандартной библиотеки определены в пространстве имен ***std*** и представлены набором заголовочных файлов. Они идентифицируют основные части библиотеки. Таким образом, их перечисление дает представление о библиотеке и предоставляет способ ее описания в этой и последующих главах.

Ниже в этом подразделе приведен список заголовочных файлов, сгруппированных по функциям, с краткими пояснениями и ссылками, где они рассматриваются. Файлы сгруппированы так, чтобы соответствовать организации стандарта. Ссылка на стандарт (как, например, § s.18.1) означает, что данная возможность здесь не рассматривается.

Стандартный заголовочный файл, имя которого начинается с буквы ***c***, эквивалентен заголовочному файлу в стандартной библиотеке C. Для каждого заголовочного файла <***X.h***>, определяющего часть стандартной библиотеки C в глобальном пространстве имен, а также в пространстве имен ***std***, существует заголовок <***cX***>, определяющий эти же имена только в пространстве имен ***std*** (см. § 9.2.2).

Контейнеры		
<vector>	*одномерный массив элементов T*	§ 16.3
<list>	*двусвязный список элементов T*	§ 17.2.2
<deque>	*очередь элементов T с двумя концами*	§ 17.2.3
<queue>	*очередь элементов T*	§ 17.3.2
<stack>	*стек элементов T*	§ 17.3.1
<map>	*ассоциативный массив элементов T*	§ 17.4.1
<set>	*набор элементов T*	§ 17.4.3
<bitset>	*набор булевских переменных*	§ 17.5.3

Ассоциативные контейнеры ***multimap*** и ***multiset*** можно найти соответственно в *<map>* и *<set>*. Контейнер ***priority_queue*** (очередь с приоритетом) объявлен в *<queue>*.

Основные утилиты		
<utility>	*операторы и пары*	§ 17.1.4, § 17.4.1.2
<functional>	*объекты-функции*	§ 18.4
<memory>	*распределители памяти для контейнеров*	§ 19.4.4
<ctime>	*время и дата в стиле C*	§ s.20.5

В заголовочном файле *<memory>* также содержится шаблон ***auto_ptr***, который в основном используется для сглаживания взаимодействия между указателями и исключениями (§ 14.4.2).

Итераторы		
<iterator>	*итераторы и их поддержка*	Глава 19

Итераторы обеспечивают механизм приспособления стандартных алгоритмов для работы со стандартными контейнерами и другими подобными типами (§ 2.7.2, § 19.2.1).

Алгоритмы		
<algorithm>	*основные алгоритмы*	Глава 18
<cstdlib>	***bsearch*** () ***qsort*** ()	§ 18.11

Типичный универсальный алгоритм может быть применим к любой последовательности (§ 3.8, § 18.3) элементов любого типа. Функции ***bsearch*** () и ***qsort*** () из стандартной библиотеки C применимы к встроенным массивам элементов, тип которых не имеет копирующих конструкторов и деструкторов, определенных пользователем (§ 7.7).

Диагностика		
<exception>	*класс исключений*	§ 14.10
<stdexcept>	*стандартные исключения*	§ 14.10
<cassert>	*макросы утверждений*	§ 24.3.7.2
<cerrno>	*обработка ошибок в стиле C*	§ 20.4.1

Утверждения, основанные на исключениях, описаны в § 24.3.7.1.

Строки		
<string>	*строка элементов T*	Глава 20
<cctype>	*классификация символов*	§ 20.4.2
<cwctype>	*классификация символов из расширенного набора*	§ 20.4.2
<cstring>	*функции над строками в стиле C*	§ 20.4.1
<cwchar>	*функции над строками символов из расширенного набора в стиле C*	§ 20.4
<cstdlib>	*функции над строками в стиле C*	§ 20.4.1

Заголовочный файл *<cstring>* объявляет семейство функций ***strlen*** (), ***strcpy*** () и т. п. Заголовочный файл *<cstdlib>* объявляет функции ***atof*** () и ***atoi*** (), которые преобразуют символьные С-строки в численные значения.

Ввод/вывод		
<iosfwd>	*предварительные объявления средств ввода/вывода*	§ 21.1
<iostream>	*стандартные объекты и операции с потоками ввода/вывода*	§ 21.2.1
<ios>	*базовые классы потоков ввода/вывода*	§ 21.2.1
<streambuf>	*буферизация потоков*	§ 21.6
<istream>	*шаблон потока ввода*	§ 21.3.1
<ostream>	*шаблон потока вывода*	§ 21.3.1
<iomanip>	*манипуляторы*	§ 21.4.6.2
<sstream>	*потоки в строки/из строк*	§ 21.5.3
<cctype>	*функции для работы с символами*	§ 20.4.2
<fstream>	*потоки ввода/вывода в файлы*	§ 21.5.1
<cstdio>	*семейство функций ввода/вывода* ***printf*** ()	§ 21.8
<cwchar>	*ввод/вывод символов из расширенного набора в стиле* ***printf*** ()	§ 21.8

Манипуляторы — это объекты, которыми пользуются, чтобы манипулировать состоянием потока (например изменить формат вывода чисел с плавающей точкой) (§ 21.4.6).

Локализация		
<locale>	*представляет культурные различия*	§ 21.7
<clocale>	*представляет культурные различия в стиле C*	§ 21.7

Заголовочный файл ***locale*** локализует различия — например, в формате вывода дат, в символах для обозначения валют и критериях сравнения строк — в человеческих языках и культурах.

Поддержка языка		
<limits>	*числовые ограничения*	§ 22.2
<climits>	*макросы пределов скалярных чисел в стиле C*	§ 22.2.1

Поддержка языка *(продолжение)*		
<*cfloat*>	*макросы пределов чисел с плавающей точкой в стиле C*	§ 22.2.1
<*new*>	*динамическое распределение памяти*	§ 16.1.3
<*typeinfo*>	*поддержка идентификации типов во время выполнения*	§ 15.4.1
<*exception*>	*поддержка обработки исключений*	§ 14.10
<*cstddef*>	*языковая поддержка библиотеки C*	§ 6.2.1
<*cstdarg*>	*поддержка функций с переменым числом аргументов*	§ 7.6
<*csetjmp*>	*раскрутка стека в стиле C*	§ s.18.7
<*cstdlib*>	*завершение программ*	§ 9.4.1.1
<*ctime*>	*системные часы*	§ Г.4.4.1
<*csignal*>	*обработка сигналов в стиле C*	§ s.18.7

Заголовочный файл <*stddef*> определяет: тип значений, которые возвращает функция ***sizeof*** (), ***size_t***; тип результата вычитания указателей и индексов массива, ***ptrdiff_t*** (§ 6.2.1) и злосчастный макрос ***NULL*** (§ 5.1.1).

Раскрутка стека в стиле C (с использованием ***setjmp*** и ***longjmp*** из <*csetjmp*>) несовместима с обработкой исключений (§ 8.3, глава 14, приложение Д) и лучше ее избегать.

Числа		
<*complex*>	*комплексные числа и операции с ними*	§ 22.5
<*valarray*>	*вектора из чисел и операции с ними*	§ 22.4
<*numeric*>	*распространенные числовые операции*	§ 22.6
<*cmath*>	*общие математические функции*	§ 22.3
<*cstdlib*>	*случайные числа в стиле C*	§ 22.7

Так уж исторически сложилось, что ***abs*** () и ***div*** () находятся в <*cstdlib*>, а не в <*cmath*> с остальными математическими функциями.

Пользователю и разработчику библиотеки не разрешается добавлять или убирать объявления из стандартных заголовочных файлов. Также недопустимо пытаться изменить содержимое заголовочных файлов, определяя макросы до вставки заголовочных файлов, или пытаться изменить смысл объявлений в заголовочных файлах путем объявлений в их контексте (§ 9.2.3). Любая программа или реализация, играющая в подобные игры, не согласуется со стандартом, и код, основанный на таких трюках, не переносим. Даже если сегодня он работает, следующая версия любой части реализации может их разрушить. Избегайте таких трюков.

Для использования возможностей стандартной библиотеки должен быть включен ее заголовочный файл. Написание соответствующих объявлений самостоятельно *не* соответствует стандарту. Дело в том, что некоторые реализации языка C++ оптимизируют процесс компиляции, основываясь на включении стандартных заголовочных файлов, а другие обеспечивают оптимизированную реализацию средств стандартной библиотеки, вводимых заголовочными файлами. В общем, программисты не могут, да и не должны знать, как разработчики данной реализации используют стандартные заголовочные файлы.

Однако программисты могут приспосабливать вспомогательные шаблоны, такие как ***swap*** () (§ 16.3.9), для нестандартных библиотек и типов, определяемых пользователем.

16.1.3. Поддержка языка

Небольшую часть стандартной библиотеки занимает поддержка языка — то есть средства, которые должны присутствовать для запуска программ.

Библиотечные функции, поддерживающие операторы ***new*** и ***delete***, обсуждаются в § 6.2.6, § 10.4.11, § 14.4.4 и § 15.6. Они представлены в заголовочном файле <***new***>.

Идентификация типов во время выполнения опирается на класс ***type_info***, который описан в § 15.4.4 и представлен в <***typeinfo***>.

Стандартные классы исключений рассматриваются в § 14.10 и представлены в заголовочных файлах <***new***>, <***typeinfo***>, <***ios***>, <***exception***> и <***stdexcept***>.

Запуск программы и ее завершение рассматриваются в § 3.2, § 9.4 и § 10.4.9.

16.2. Проектирование контейнеров

Контейнер — это объект, содержащий другие объекты. Примерами контейнеров являются списки, вектора и ассоциативные массивы. Как правило, в контейнер можно добавлять объекты и удалять их из него.

Естественно, пользователям эту идею можно представить по-разному. Стандартные библиотечные контейнеры C++ были построены в соответствии с двумя критериями: обеспечение максимума свободы в построении индивидуальных контейнеров, и в то же время предоставление для пользователей общего интерфейса. Это позволяет оптимизировать реализацию контейнеров и позволяет пользователям писать код, независящий от того, какой конкретный контейнер используется.

Проектирование контейнеров, как правило, отвечает одному из указанных двух критериев. Контейнерную и алгоритмическую часть стандартной библиотеки (ее часто называют STL) можно рассматривать как решение проблемы одновременного обеспечения универсальности и эффективности. Следующие разделы покажут сильные и слабые стороны двух традиционных стилей контейнеров, чтобы понять проектное решение, выбранное для стандартных контейнеров.

16.2.1. Специализированные контейнеры и итераторы

Очевидным приближением к реализации вектора и списка является их определение таким образом, который лучше всего отвечает цели их использования:

```
template <class T> class Vector {       // оптимальный
public:
        explicit Vector (size_t n);     // инициализация, чтобы заполнить
                                        // n объектов значением T()
        T& operator[] (size_t n);       // обращение по индексу
        // ...
};
template<class T> class List {          // оптимальный
public:
        class Link { /* ... */ };
        List ();                        // первоначально пустой
        void put (T *);                 // разместить перед первым элементом
        T * get ();                     // получить текущий элемент
        // ...
};
```

Оба класса обеспечивают операции, близкие к идеальным в использовании, и для обоих классов мы можем выбрать подходящее представление, не беспокоясь о других видах контейнеров. Это позволяет реализовать операции почти что оптимально. В частности, наиболее употребительные операции, такие как ***put*** () для ***List*** и ***operator***[] () для ***Vector***, невелики и легко встраиваются.

Обычное использование большинства видов контейнеров — это проход (итерация) по контейнеру, перебор элементов одного за другим. Обычно это делается путем определения класса итератора, подходящего для данного вида контейнеров (см. § 11.5 и § 11.14[7]).

Однако пользователю, перебирающему контейнер, часто нет дела, хранятся данные в объекте ***List*** или ***Vector***. В этом случае код с итерацией не должен зависеть от того, используется ***List*** или ***Vector***. Желательно, чтобы в обоих случаях работал один и тот же фрагмент кода.

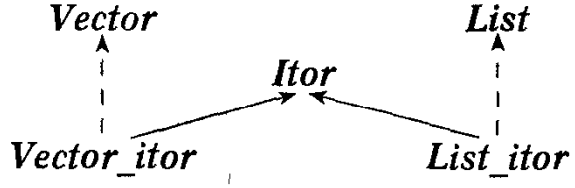
Решение таково: определить класс итератора, обеспечивающий операцию «получение следующего элемента», которую можно реализовать для любого контейнера. Например:

```
template<class T> class Itor {     // общий интерфейс (абстрактный класс § 2.5.4, § 12.3)
public:
// возвратить 0, если элементов больше нет

        virtual T * first () = 0;       // указатель на первый элемент
        virtual T * next () = 0;        // указатель на следующий элемент
};
```

Теперь мы можем обеспечить реализацию для векторов и списков:

```
template<class T> class Vector_itor : public Itor<T> {          // реализация Vector
        Vector<T>& v;
        size_t index;       // индекс текущего элемента
public:
        Vector_itor (Vector<T>& vv) : v (vv), index (0) { }
        T * first () { return (v.size () ? &v[index=0] : 0; }
        T * next () { return (++index<v.size () ? &v[index] : 0; }
};

template<class T> class List_itor : public Itor<T> {            // реализация List
        List<T>& lst;
        List<T>::Link p;    // указывает на текущий элемент
public:
        List_itor (List<T>&);
        T * first ();
        T * next ();
};
```

Или графически, используя пунктирные линии для выражения отношения «реализовано с использованием»:



Внутренняя структура двух итераторов совершенно различна, но пользователей это не касается. Теперь мы можем написать программу, итерирующую все что угодно, для чего мы реализовали ***Itor***. Например:

```
int count(Itor<char>& ii, char term)
{
    int c = 0;
    for (char* p = ii.first(); p; p=ii.next()) if (*p==term) c++;
    return c;
}
```

Однако здесь есть одна загвоздка. Операции над итератором ***Itor*** просты, и все же они привносят затраты на вызов (виртуальной) функции. Во многих случаях эти затраты незначительны по сравнению с другими действиями. Однако итерация простого контейнера во многих быстродействующих системах — критическая операция, а вызов функции во много раз дороже, чем целочисленное сложение или разыменование указателя, которые выполняет функция ***next*** () для ***vector*** и ***list***. Следовательно, эта модель для стандартной библиотеки не подходит или, по крайней мере, не идеальна.

Тем не менее эта контейнерно-итераторная модель успешно используется во многих системах. В течение многих лет она была моей излюбленной для большинства прикладных программ. Ее достоинства и недостатки могут быть подытожены следующим образом:

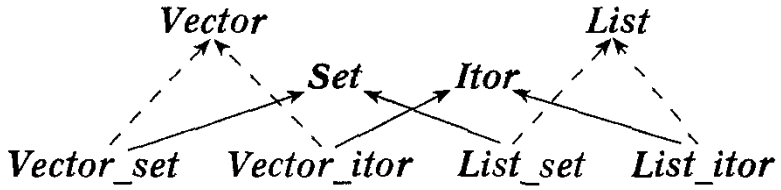
+ Индивидуальные контейнеры просты и эффективны.
+ От контейнеров не требуется большой общности. Итераторы и классы-оболочки (§ 25.7.1) могут использоваться для помещения независимо разработанных контейнеров в «общие рамки».
+ Общность использования обеспечивается через итераторы (а не через общий контейнерный тип; § 16.2.2).
+ Для одних и тех же контейнеров можно определить разные итераторы, служащие разным потребностям.
+ Контейнеры по умолчанию безопасны с точки зрения типов и однородны (то есть все элементы в контейнере относятся к одному и тому же типу). Разнородные контейнеры можно создать как однородные контейнеры указателей на общий базовый класс.
+ Контейнеры неинтрузивны (чтобы стать элементом контейнера объект не обязан иметь особый базовый класс или поле связи). Неинтрузивные контейнеры хорошо работают со встроенными типами и структурами, заданными внешне жестким образом.
– Каждый доступ к итератору приводит к вызову виртуальной функции. По сравнению с простой встроенной функцией эти затраты могут быть значительны.
– Иерархия классов итераторов имеет тенденцию становиться запутанной.
– Контейнеры не имеют ничего общего между собой, и объекты в контейнерах также не имеют ничего общего. Это усложняет создание универсальных служб, например, таких как персистентность и объектный ввод/вывод.

Здесь + обозначает достоинство, а – обозначает недостаток.

Я придаю большое значение гибкости, которую обеспечивают итераторы. Общий интерфейс, такой как ***Itor***, может быть создан гораздо позже проектирования и реализации контейнеров (здесь это ***Vector*** и ***List***). При проектировании мы обычно сначала

изобретаем что-нибудь достаточно конкретное. Например, мы проектируем массив и изобретаем список. Только позднее мы открываем абстракцию, которая объединяет в данном контексте и массивы, и списки.

Фактически мы можем делать такую «позднюю абстракцию» неоднократно. Предположим, мы хотим представить множество. Множество — это отличная от ***Itor*** абстракция, и все же мы можем создать интерфейс ***Set*** для контейнеров ***Vector*** и ***List*** способом, очень похожим на тот, которым мы ввели интерфейс ***Itor*** для ***Vector*** и ***List***.



Таким образом, поздняя абстракция с использованием абстрактных классов позволяет нам обеспечить разные реализации понятия, даже когда между реализациями нет большого сходства. Например, списки и вектора имеют некоторую очевидную общность, но мы можем также легко реализовать ***Itor*** и для ***istream***.

По логике два последних пункта в списке являются главными недостатками данного подхода. То есть даже если устранить для итераторов и подобных контейнерных интерфейсов затраты времени на вызов функций (что в некоторых случаях возможно), этот подход не станет идеальным для стандартной библиотеки.

Неинтрузивные контейнеры влекут за собой небольшой перерасход времени и памяти по сравнению с интрузивными. Я не вижу здесь проблемы. Если это станет проблемой, итератор вроде ***Itor*** может быть создан и для интрузивного контейнера (§ 16.5[11]).

16.2.2. Контейнеры с общим базовым классом

Интрузивный контейнер можно определить без использования шаблонов или других способов параметризации объявления типа. Например:

```
struct Link {
        Link* pre;
        Link* suc;
        //...
};

class List {
        Link* head;
        Link* curr;             // текущий элемент
public:
        Link* get ();       // удаляет и возвращает текущий элемент
        void put (Link*);   // вставляет перед текущим элементом
        //...
};
```

Теперь ***List*** — это список структур ***Link***, и он может содержать объекты любых типов, производных от ***Link***. Например:

```
class Ship : public Link { /* ... */ };

void f (List* lst)
```

```
{
    while (Link* po = lst->get()) {
        if (Ship* ps = dynamic_cast<Ship*>(po)) {      // Ship должен быть
                                                       // полиморфным (§ 15.4.1)
            // используем Ship
        }
        else {
        // проблема, делаем что-нибудь другое
        }
    }
}
```

В подобном стиле определяет свои стандартные контейнеры Simula, так что для языков, поддерживающих объектно-ориентированное программирование, этот подход можно считать исконным. В настоящее время общий класс для всех объектов обычно называют ***Object*** или как-нибудь вроде этого. Класс ***Object*** как правило обеспечивает другие общие услуги, а не только служит связью в контейнерах.

Часто, но не обязательно, этот подход распространяют на общие контейнерные типы:

```
class Container : public Object {
public:
    virtual Object* get();                  // удаляет и возвращает текущий элемент
    virtual void put(Object *);             // вставляет перед текущим элементом
    virtual Object*& operator[](size_t);    // индексация
};
```

Отметим, что эти операции, предоставляемые классом ***Container***, виртуальны, так что индивидуальные контейнеры могут их соответствующим образом замещать:

```
class List : public Container {
public:
    Object* get();
    void put(Object *);
    // ...
};

class Vector : public Container {
public:
    Object*& operator[](size_t);
    // ...
};
```

Сразу же возникает одна проблема. Какие операции мы хотим получить от класса ***Container***? Мы можем обеспечить только те операции, которые поддерживают все контейнеры. Однако пересечение наборов операций во всех контейнерах — это до смешного узкий интерфейс. Фактически, во многих представляющих интерес случаях это пересечение вообще пусто. Поэтому, реально оценивая ситуацию, мы должны предоставить объединение ключевых операций по всему множеству контейнеров, которые мы намереваемся поддержать. Такое объединение интерфейсов к набору понятий называют *жирным интерфейсом* (fat interface) (§ 24.4.3).

Для функций в жирном интерфейсе мы можем либо обеспечить реализации по умолчанию, либо заставить реализовывать их производные классы, сделав эти функ-

ции чисто виртуальными. В любом случае мы получим множество функций, которые просто сообщают об ошибках во время выполнения программы. Например:

```
class Container : public Object {
public:
        struct Bad_op {                              // класс исключений
                const char* p;
                Bad_op (const char* pp) : p (pp) { }
        };

        virtual void put (Object*) { throw Bad_op ("put"); }
        virtual Object* get () { throw Bad_op ("get"); }
        virtual Object*& operator[] (int) { throw Bad_op ("[]"); }
        // ...
};
```

Если мы хотим защититься от возможности того, что контейнер не поддерживает функцию ***get*** (), мы должны где-то перехватить исключение ***Container::Bad_op***. Теперь мы можем написать пример класса ***Ship*** таким образом:

```
class Ship : public Object { /* ... */ };
void f1 (Container* pc)
{
        try {
                while (Object* po = pc->get ()) {
                        if (Ship* ps = dynamic_cast<Ship*> (po)) {
                                // используем Ship
                        }
                        else {
                                // ошибка, делаем что-нибудь другое
                        }
                }
        }
        catch (Container::Bad_op& bad) {
                // ошибка, делаем что-нибудь другое
        }
}
```

Это скучно, и поэтому проверка ***Bad_op*** обычно располагается в другом месте. Положившись на перехват исключения где-то в другом месте, мы можем упростить данный пример:

```
void f2 (Container* pc)
{
        while (Object* po = pc->get ()) {
                Ship& s = dynamic_cast<Ship&> (*po);
                // используем объект класса Ship
        }
}
```

Однако я считаю неоправданное упование на проверку во время выполнения неэффективным и свидетельствующим о плохом вкусе. В подобных случаях я предпочитаю альтернативу — статическую проверку:

```
void f3 (Itor<Ship>* i)
{
        while (Ship* ps = i->next ()) {
                // используем Ship
        }
}
```

Достоинства и недостатки подхода к созданию контейнеров на основе общих базовых классов могут быть подытожены следующим образом (см. также § 16.5[10]):

- – Операции над индивидуальными контейнерами приводят к затратам, связанным с вызовами виртуальных функций.
- – Все контейнеры должны быть производными от ***Container***. Это чревато применением «жирных интерфейсов», требует большой степени предвидения и опирается на проверку типов во время выполнения. Подгонка независимо разработанных контейнеров в общие рамки в лучшем случае оказывается затруднительной (см. § 16.5[12]).
- \+ Общий базовый класс ***Container*** облегчает взаимозаменяемость контейнеров, обеспечивающих схожие наборы операций.
- – Контейнеры разнородны и по умолчанию не безопасны с точки зрения типов (все, на что мы можем рассчитывать — это то, что элементы относятся к типу ***Object****). При желании безопасные с точки зрения типов и однородные контейнеры можно определить с помощью шаблонов.
- – Контейнеры интрузивны (то есть каждый элемент должен относиться к производному от ***Object*** типу). Объекты встроенных типов и структуры, жестко заданные внешним образом, нельзя прямо поместить в контейнеры.
- – Извлеченному из контейнера элементу, прежде чем его использовать, нужно задать соответствующий тип, при помощи явного преобразования типов.
- \+ Классы ***Container*** и ***Object*** могут служить основой при реализации служб для всех объектов и контейнеров. Это чрезвычайно облегчает создание универсальных служб, таких как персистентность и объектный ввод/вывод.

Как и раньше (§ 16.2.1), + обозначает достоинство, а – обозначает недостаток.

По сравнению с подходом, использующим несвязанные между собой контейнеры и итераторы, метод, использующий «общего предка всех объектов», является для пользователя излишне сложным и запутанным, приводит к затратам во время выполнения и ограничивает виды объектов, которые можно поместить в контейнер. Вдобавок, для многих классов наследование от ***Object*** означает открытие деталей реализации. Таким образом, при создании стандартной библиотеки этот подход далек от идеального.

Однако не следует недооценивать универсальность и гибкость данного подхода. Наряду с альтернативным его можно успешно использовать во многих прикладных программах. Преимущества этого подхода проявляются там, где не так важна эффективность, как предоставляемая единым интерфейсом ***Container*** простота и служебные функции вроде объектного ввода/вывода.

16.2.3. STL[1]-контейнеры

Контейнеры стандартной библиотеки и итераторы (часто называемые средой разработки STL, § 3.10) можно воспринимать как стремление взять лучшее из двух описанных выше традиционных моделей. Однако библиотека STL была построена

[1] STL, Standard Template Library — стандартная библиотека шаблонов. — *Примеч. ред.*

не так. Она явилась результатом целенаправленного поиска бескомпромиссно эффективных общих алгоритмов.

Стремление к эффективности привело от трудно встраиваемых виртуальных функций к небольшим часто используемым функциям доступа. Поэтому мы не можем представить стандартный интерфейс для контейнеров или стандартный интерфейс итераторов как абстрактный класс. Вместо этого все виды контейнеров поддерживают стандартный набор базовых операций. Чтобы избежать проблем с «жирными» интерфейсами (§ 16.2.2, § 24.4.3), операции, которые не могут быть эффективно реализованы для всех контейнеров, не включены в набор общих операций. Например, обращение по индексу введено для *vector*, но не для *list*. В добавок, каждый вид контейнеров обеспечивает свой собственный итератор, поддерживающий стандартный набор итерационных операций.

Стандартные контейнеры не являются производными от некоторого общего базового класса. Вместо этого каждый контейнер реализует все стандартные контейнерные интерфейсы. Подобным же образом не существует и общего базового класса итераторов. Использование стандартных контейнеров и итераторов не подразумевает никакой явной или неявной проверки типов во время выполнения.

Предоставление общих услуг для всех контейнеров (важный и сложный вопрос) удалось обеспечить не через общий базовый класс, а при помощи «распределителей памяти» (allocators), передаваемых как аргументы шаблона (§ 19.4.3).

Прежде чем входить в детали и приводить примеры программ, достоинства и недостатки STL-подхода можно обобщить следующим образом:

+ Индивидуальные контейнеры просты и эффективны (не совсем так просты, какими могут быть по-настоящему независимые контейнеры, но так же эффективны).
+ Каждый контейнер обеспечивает набор стандартных операций со стандартными именами и смыслом. Дополнительные операции обеспечиваются для частных контейнерных типов по мере необходимости. Кроме того, для помещения независимо разработанных контейнеров в общие рамки (§ 16.5[14]) могут использоваться классы-оболочки (§ 25.7.1).
+ Дополнительная общность использования обеспечивается через стандартные итераторы. Каждый контейнер предоставляет итераторы, поддерживающие набор стандартных операций со стандартными именами и смыслом. Для каждого частного контейнерного типа определен тип итераторов, чтобы эти итераторы были как можно проще и эффективнее.
+ Для удовлетворения разных нужд контейнеров в добавок к стандартным итераторам могут быть определены различные дополнительные итераторы и универсальные интерфейсы.
+ Контейнеры по умолчанию безопасны с точки зрения типов и однородны (то есть все элементы в контейнере одного и того же типа). Разнородные контейнеры могут быть созданы как контейнеры указателей на общий базовый класс.
+ Контейнеры неинтрузивны (то есть чтобы быть членом контейнера, объекты не нуждаются в общем базовом классе или поле связи). Неинтрузивные контейнеры хорошо работают со встроенными типами и жестко заданными структурами.
+ Интрузивные контейнеры могут подгоняться под общие рамки. Естественно, интрузивные контейнеры влекут за собой ограничения на типы элементов.
+ Каждый контейнер имеет аргумент, называемый «распределителем памяти», который можно использовать в качестве основы для реализации услуг, необхо-

димых контейнеру. Это чрезвычайно облегчает создание универсальных служб вроде персистентности и объектного ввода/вывода (§ 19.4.3).
- Не существует стандартного представления времени выполнения для контейнеров и итераторов, которое могло бы задаваться как аргумент функции (хотя там, где нужно, в частных прикладных программах такие представления для стандартных контейнеров и итераторов нетрудно определить; § 19.3).

Как и раньше, + означает достоинство, а – означает недостаток.

Другими словами, контейнеры и итераторы не имеют фиксированного стандартного представления. Зато каждый контейнер обеспечивает стандартный интерфейс в виде набора операций, так что один контейнер можно использовать вместо другого. С итераторами обращаются схожим образом. Это приводит к минимальным затратам времени и памяти, и в то же время позволяет пользователю задействовать общность и на уровне контейнеров (как при подходе с общим базовым классом), и на уровне итераторов (как при подходе со специализированными контейнерами).

STL-подход в большой степени опирается на шаблоны. Чтобы избежать чрезмерного повторения кода, для обеспечения общей реализации контейнеров, содержащих указатели, обычно требуется частичная специализация (§ 13.5).

16.3. Вектора

Здесь вектор описывается как пример полного стандартного контейнера. Если не утверждается обратного, все сказанное о векторе относится ко всем стандартным контейнерам. Глава 17 описывает особенности списков, множеств, ассоциативных массивов и т. п. Возможности, предлагаемые вектором, — и другими подобными контейнерами — описаны довольно подробно. Цель — дать понять, как возможные применения векторов, так и их роль в проекте стандартной библиотеки вообще.

Обзор стандартных контейнеров и предлагаемых ими средств можно найти в § 17.1. Ниже ознакомление с векторами производится по этапам: типы членов, итераторы, доступ к элементам, конструкторы, операции со стеком, операции со списками, размер и емкость, вспомогательные функции и *vector<bool>*.

16.3.1. Типы

Стандартный вектор — это шаблон, определенный в пространстве имен *std* и представленный в заголовочном файле *<vector>*. Сначала он определяет набор стандартных имен типов:

```
template<class T, class A = allocator<T> > class std::vector {
public:
    // типы
    typedef T value_type;                          // тип элемента
    typedef A allocator_type;                      // тип распределителя памяти
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef implementation_dependent1 iterator;    // T*
    typedef implementation_dependent2 const_iterator; // const T*
```

```
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    typedef typename A::pointer pointer;                    // указатель на элемент
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;                // ссылка на элемент
    typedef typename A::const_reference const_reference;

//...
};
```

Каждый стандартный контейнер определяет эти имена типов в качестве членов. Каждый определяет их способом, наиболее подходящим для своей реализации.

Тип элементов в контейнере задается первым аргументом шаблона и известен как ***value_type***. Второй, необязательный аргумент ***allocator_type*** определяет, как ***value_type*** взаимодействует с различными механизмами распределения памяти. В частности, он предоставляет функции, используемые контейнером, чтобы выделять и освобождать память для своих элементов. Распределители памяти обсуждаются в § 19.4. Как правило, ***size_type*** описывает тип, используемый в контейнере для индексации, а ***difference_type*** — означает тип результата вычитания двух итераторов для контейнера. Для большинства контейнеров они соответствуют ***size_t*** и ***ptrdiff_t*** (§ 6.2.1).

В приложении Д обсуждается поведение вектора, если распределение памяти или операции с элементами генерируют исключения.

С итераторами мы познакомились в § 2.7.2, более подробно они описаны в главе 19. Их можно представлять себе, как указатели на элементы контейнера. Каждый контейнер обеспечивает тип под названием ***iterator*** для указания на свои элементы. Он также предоставляет тип ***const_iterator***, который используется, когда элементы не нужно модифицировать. Как и с указателями, мы используем более безопасную версию с ***const***, если нет каких-либо причин поступать иначе. Фактические типы итераторов вектора определяются при реализации. Их очевидные определения для стандартного шаблона ***vector*** были бы соответственно ***T**** и ***const T****.

Типы обратных итераторов для контейнера ***vector*** конструируются из стандартных шаблонов ***reverse_iterator*** (§ 19.2.5). Они представляют последовательность в обратном порядке.

Как показано в § 3.8.1, эти имена типов членов позволяют пользователю писать программу с использованием контейнеров, ничего не зная об их настоящих типах. В частности, они позволяют составить код, который будет работать с любым стандартным контейнером. Например:

```
template<class C> typename C::value_type sum (const C& c)
{
    typename C::value_type s = 0;
    typename C::const_iterator p = c.begin ();      // начинаем с начала
        while (p != c.end ())     {                 // продолжаем до конца
            s += *p;                                // получаем значение элемента
            ++p;                                    // указатель p будет указывать
                                                    // на следующий элемент
    }
    return s;
}
```

Необходимость добавлять ***typename*** перед именами типов членов в параметрах шаблона раздражает. Однако компилятор — вещь неодушевленная, и у него нет

другого способа узнать, что член типа аргумента шаблона является именем типа (§ B.13.5).

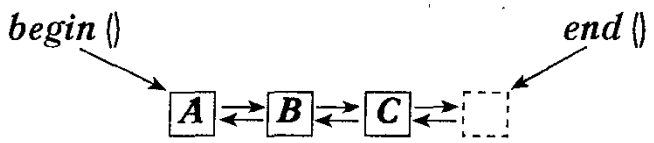
Что касается указателей, префикс * означает разыменование итератора (§ 2.7.2, § 19.2.1), а ++ — его инкремент.

16.3.2. Итераторы

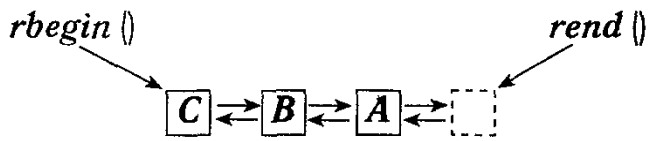
Как показано в предыдущих разделах, при помощи итераторов программист может управлять контейнерами, не зная фактических типов, используемых для идентификации элементов. Несколько ключевых функций-членов позволяют программисту найти концы последовательности элементов:

```
template<class T, class A = allocator<T> > class vector {
public:
        //...
        // итераторы

        // указывает на первый элемент
        iterator begin ();
        const_iterator begin () const;
        // указывает на элемент, следующий за последним
        iterator end ();
        const_iterator end () const;

        // указывет на первый элемент в обратной последовательности
        reverse_iterator rbegin ();
        const_reverse_iterator rbegin () const;
        // указывает на элемент, следущий за последним, в обратной последовательности
        reverse_iterator rend ();
        const_reverse_iterator rend () const;
        //...
};
```

Пара ***begin ()/end ()*** выдает элементы контейнера в обычном порядке, то есть за нулевым элементом следует первый, затем второй и т. д. Пара ***rbegin ()/rend ()*** выдает их в обратном порядке, то есть за ***n–1*** элементом следует ***n–2***, затем ***n–3*** и т. д. Вот как видит последовательность ***iterator***:



А вот как эту же последовательность рассматривает ***reverse_iterator*** (§ 19.2.5):



Это позволяет нам использовать алгоритмы, которым удобнее представлять последовательность в обратном порядке. Например:

```
template<class C> typename C::iterator find_last (C& c, typename C::value_type v)
{
        typename C::reverse_iterator ri = find (c.rbegin (), c.rend (), v);
```

```
        // используем c.end(), чтобы указать «не найдено»
        if (ri == c.rend()) return c.end();
        typename C::iterator i = ri.base();
        return --i;
}
```

Для *reverse_iterator ri.base()* возвращает *iterator*, указывающий на один элемент вперед позиции, на которую указывает *ri* (§ 19.2.5). Без обратных итераторов нам бы пришлось писать явный цикл:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
        typename C::iterator p = c.end();    // ищем от конца к началу
        while (p!=c.begin())
                if (*--p==v) return p;
        return c.end();                      // используем c.end(), чтобы указать «не найдено»
}
```

Обратный итератор — это совершенно обычный итератор, поэтому мы могли бы написать:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
        // просматриваем последовательность в обратном порядке
        typename C::reverse_iterator p = c.rbegin();
        while (p!=c.rend()) {
                if (*p==v) {
                        typename C::iterator i = p.base();
                        return --i;
                }
                ++p;                         // внимание: инкремент (++), а не декремент (--)
        }
        return c.end();                      // используем c.end(), чтобы указать «не найдено»
}
```

Обратите внимание, что *C::reverse_iterator* это не тот же самый тип, что *C::iterator*.

16.3.3. Доступ к элементам

Одним из важных аспектов контейнера *vector* по сравнению с другими контейнерами является легкий и эффективный доступ к его отдельным элементам в произвольном порядке:

```
template<class T, class A = allocator<T> > class vector {
        public:
        // ...
        // доступ к элементам:

        reference operator[](size_type n);              // доступ без проверки
        const_reference operator[](size_type n) const;

        reference at(size_type n);                      // доступ с проверкой
        const_reference at(size_type n) const;
        reference front();                              // первый элемент
```

```
        const_reference front() const;
        reference back();                       // последний элемент
        const_reference back() const;
        //...
};
```

Индексация осуществляется ***operator***[] () и функцией ***at*** (); ***operator***[] () обеспечивает доступ без проверки, в то время как функция ***at*** () осуществляет проверку диапазона и возбуждает исключение ***out_of_range***, если индекс выходит за его пределы. Например:

```
void f(vector<int>& v, int i1, int i2)
try {
        for (int i=0; i < v.size(); i++) {
                // размер уже проверен: используем v[i] без проверки
        }
        v.at(i1) = v.at(i2);                    // проверяем диапазон при доступе
        //...
}
catch (out_of_range) {
        // ошибка диапазона
}
```

Этот пример иллюстрирует одно из возможных использований. То есть, если диапазон уже проверен, можно безопасно использовать оператор индексации [] без проверки; в противном случае разумно использовать функцию ***at*** (), которая проверяет диапазон. Это различие важно, когда большое внимание уделяется эффективности. Когда она не играет роли, или когда не совсем очевидно, точно ли проверен диапазон, безопаснее пользоваться вектором с проверяющим диапазон оператором [] (как ***Vec*** из § 3.7.1) или итератором с проверкой (§ 19.3).

Доступ по умолчанию не производит проверки, чтобы соответствовать массивам. Поверх быстрого доступа без проверки вы можете построить безопасное (с проверкой) средство, но вы не можете ускорить медленное средство.

Операции доступа возвращают значения типа ***reference*** или ***const_reference*** в зависимости от того, применяются ли они к объекту типа ***const*** или нет. Тип ***reference*** удобен для доступа к элементам. В простой и очевидной реализации контейнера ***vector<X>***, ***reference*** — это просто ***X***&, а ***const_reference*** — это просто ***const X***&. Эффект от попытки создания ссылки за пределы диапазона не определен. Например:

```
void f(vector<double>& v)
{
        double d = v[v.size()];         // не определено: плохой индекс

        list<char> lst;
        char c = lst.front();           // не определено: список пуст
}
```

В стандартных последовательностях обращение по индексу поддерживают только ***vector*** и ***deque*** (§ 17.2.3). Причина — желание не запутывать пользователей, предоставляя фундаментально неэффективные операции. Например, обращение по индек-

су может быть обеспечено и для списков ***list*** (§ 17.2.2), но это опасно неэффективно (то есть, ***O*** (*n*)).

Функции-члены ***front*** () и ***back*** () возвращают ссылки на первый и последний элемент соответственно. Они наиболее полезны там, где эти элементы заведомо существуют, и в программах, где эти элементы представляют особый интерес. Очевидный пример — контейнер ***vector***, используемый в качестве контейнера ***stack*** (§ 16.3.5). Отметим, что функция ***front*** () возвращает ссылку на элемент, итератор которого возвращает функция ***begin*** (). Я часто представляю ***front*** () как первый элемент, а ***begin*** () — как указатель на первый элемент. Соотношение между ***back*** () и ***end*** () не так просто: ***back*** () — это последний элемент, а ***end*** () указывает на позицию элемента, следующего за последним.

16.3.4. Конструкторы

Естественно, ***vector*** обеспечивает полный набор (§ 11.7) конструкторов, деструкторов и операций копирования:

```
template<class T, class A = allocator<T> > class vector {
public:
        //...
        // конструкторы и т. п.

        explicit vector (const A& = A ());
        // n копий val
        explicit vector (size_type n, const T& val = T (), const A&=A ());
        // In должен быть итератором для чтения (§ 19.2.1)
        template <class In>
                vector (In first, In last, const A& = A ());    // копирование из [first:last[
        vector (const vector& x);

        ~vector ();

        vector& operator= (const vector& x);

        template <class In>             // In должен быть итератором для чтения (§ 19.2.1)
                void assign (In first, In last);        // копирование из [first:last[
        void assign (size_type n, const T& val);        // n копий val

        //...
};
```

Контейнер ***vector*** обеспечивает быстрый доступ к произвольным элементам, но изменение размера вектора обходится относительно дорого. Поэтому обычно при создании вектора мы задаем его начальный размер. Например:

```
vector<Record> vr (10000);

void f (int s1, int s2)
{
        vector<int> vi (s1);

        vector<double>* p = new vector<double> (s2);
}
```

Элементы размещенного таким образом вектора инициализируются конструкторами по умолчанию для типа элементов. То есть каждый из 10 000 элементов ***vr*** инициализируется конструктором ***Record*** (), а каждый из ***s1*** элементов контейнера ***vi*** инициализируется ***int*** (). Отметим, что для встроенных типов конструкторы по умолчанию выполняют инициализацию нулем соответствующего типа (§ 4.9.5, § 10.4.2).

Если у типа нет конструктора по умолчанию, то невозможно создать вектор с элементами такого типа без явного присваивания каждому элементу какого-то значения. Например:

```
class Num {                          // бесконечная точность
public:
    Num (long);
    // нет конструктора по умолчанию
    //...
};

vector<Num> v1 (1000);               // ошибка: нет конструктора по умолчанию
vector<Num> v1 (1000,Num (0));       // все в порядке
```

Поскольку ***vector*** не может иметь отрицательное число элементов, его размер должен быть неотрицательным. Это отражено в требовании, что его ***size_type*** должен быть типом ***unsigned***, что в некоторых архитектурах допускает больший диапазон размеров векторов. Однако это может и привести к сюрпризам:

```
void f(int i)
{
    vector<char> vc0 (-1);  // компилятору довольно легко выдать предупреждение
    vector<char> vc1 (i);
}

void g ()
{
    f(-1);        // обманываем f(), передавая ей большое положительное число!
}
```

При вызове ***f*** (***–1***) литерал ***–1*** преобразуется в (довольно большое) положительное целое число (§ В.6.3). Если повезет, компилятор пожалуется на это.

Размер вектора можно также указать неявно, задав начальный набор элементов. Это делается путем задания конструктору последовательности значений, из которых конструируется ***vector***. Например:

```
void f(const list<X>& lst)
{
    vector<X> v1 (lst.begin (), lst.end ());        // копирование элементов из списка

    char p[] = "despair";
    vector<char> v2 (p, &p[sizeof (p)-1]);          // копирование символов из С-строки
}
```

И в том, и в другом случае конструктор изменит размер контейнера ***vector*** при копировании элементов из его входной последовательности.

Конструкторы *vector*, которые можно вызвать с единственным аргументом, во избежание случайного преобразования (§ 11.7.1) объявлены ***explicit***. Например:

```
vector<int> v (10);                 // правильно: вектор из 10 int
vector<int> v2 = vector<int> (10);  // правильно: вектор из 10 int
vector<int> v3 = v2;                // правильно: v3 — копия v2
vector<int>v4 = 10;                 // ошибка: попытка неявно
                                    // преобразовать 10 в vector<int>
```

Копирующий конструктор и копирующий оператор присваивания копируют элементы вектора. Для вектора с большим количеством элементов это может быть дорогой операцией, поэтому вектора обычно передаются по ссылке. Например:

```
void f1 (vector<int>&);             // обычный стиль
void f2 (const vector<int>&);       // обычный стиль
void f3 (vector<int>);              // редкий стиль

void h ()
{
    vector<int> v (10000);

    // ...

    f1 (v);     // передача ссылки
    f2 (v);     // передача ссылки
    f3 (v);     // копирование 10 000 элементов в новый vector
                // для использования функцией f3 ()
}
```

Функции присваивания ***assign*** предоставляют альтернативу конструкторам со многими аргументами. Они нужны, поскольку оператор = воспринимает только одиночный правый операнд, так что ***assign*** () используется там, где требуется значение аргумента по умолчанию или диапазон значений. Например:

```
class Book { /* ... */ };

void f (vector<Num>& vn, vector<char>& vc, vector<Book>& vb, list<Book>& lb)
{
    vn.assign (10, Num (0));            // присваивание vn вектора из 10 копий Num(0)
    char s[] = "литерал";
    vc.assign (s, &s[sizeof (s)-1]);    // присваивание vc строки "литерал"

    vb.assign (lb.begin (), lb.end ()); // присваивание элементов списка

    // ...
}
```

Таким образом мы можем инициализировать вектор любой последовательностью элементов его типа и так же присвоить любую такую последовательность. Важно отметить, что это делается без явного введения большого числа конструкторов и функций преобразования. Также отметим, что присваивание полностью заменяет элементы вектора. По замыслу все старые элементы удаляются и вставляются новые. После присваивания размер вектора становится равным числу присвоенных элементов. Например:

```
void f ()
{
    vector<char> v (10, 'x');   // v.size()==10, каждый элемент имеет значение 'x'
```

```
        v.assign (5,'a');              // v.size()==5, каждый элемент имеет значение 'a'
    }
```

Естественно, то, что делает ***assign*** (), может быть неявно выполнено предварительным созданием соответствующего контейнера ***vector*** и последующим присваиванием этого контейнера. Например:

```
void f2 (vector<Book>& vh, list<Book>& lb)
{
    vector<Book> vt (lb.begin (), lb.end ());
    vh = vt;
    // ...
}
```

Однако это может оказаться некрасиво и неэффективно.

Использование конструктора вектора с двумя аргументами одного и того же типа может вызвать кажущуюся неоднозначность.

```
vector<int> v (10, 50);   // vector(size,value) или vector(iterator1,iterator2)? vector(size,value)!
```

Однако *int* — не итератор, и реализация должна гарантировать, что на самом деле будет вызван

```
vector (vector<int>::size_type, const int&, const vector<int>::allocator_type&);
```

а не

```
vector (vector<int>::iterator, vector<int>::iterator, const vector<int>::allocator_type&);
```

Библиотека достигает этого за счет подходящей перегрузки конструкторов и аналогичным образом разбирается с похожими неоднозначностями для ***assign*** () и ***insert*** () (§ 16.3.6).

16.3.5. Операции со стеком

Чаще всего мы представляем себе вектор компактной структурой данных, к элементам которой мы имеем доступ через индексы. Однако мы можем игнорировать это конкретное представление и рассматривать вектор как пример абстрактного представления последовательности. Такой взгляд на вектора и рассмотрение обычных применений массивов и векторов делают очевидным, что для векторов имеет смысл использовать операции со стеком:

```
template<class T, class A = allocator<T> > class vector {
    public:
        // ...
        // операции со стеком

    void push_back (const T& x);      // добавление в конец
    void pop_back ();                 // удаление последнего элемента
    // ...
};
```

Функции ***push_back*** () и ***pop_back*** () обращаются с вектором как со стеком, добавляя элементы в конец и удаляя последние элементы. Например:

```
void f(vector<char>& s)
{
    s.push_back ('a');
```

```
        s.push_back('b');
        s.push_back('c');
        s.pop_back();
        if (s[s.size()-1]!='b') error("невозможно!");
        s.pop_back();
        if (s.back()!='a') error("не должно произойти!");
    }
```

Каждый раз при обращении к ***push_back*** () вектор ***s*** увеличивается на один элемент; этот элемент добавляется в конец. Поэтому ***s[s.size()-1]***, также получаемый функцией ***s.back*** () (§ 16.3.3), — это элемент, который был последним размещен (pushed) в векторе.

Если не считать использования слова «вектор» вместо «стек», во всем этом нет ничего необычного. Приписка ***_back*** (назад) используется, чтобы подчеркнуть, что элементы добавляются в конец вектора, а не в начало. Добавление элементов в конец вектора может оказаться дорогой операцией, поскольку требует дополнительной памяти. Однако реализация должна гарантировать, что повторяющиеся стековые операции не приводят к дополнительным расходам при каждом своем вызове[1].

Отметим, что функция ***pop_back*** () не возвращает значения. Она просто снимает (pop) элемент с вершины стека, и если мы хотим узнать, что было на вершине стека перед этим, то должны посмотреть. Я бы не назвал такой стек моим любимым (§ 2.5.3, § 2.5.4), но он имеет основания считаться более эффективным, и это стандарт.

Зачем совершать стековые операции с вектором? Одна очевидная причина — чтобы реализовать стек (§ 17.3.1), но более общая причина состоит в том, чтобы строить постепенно. Например, нам может захотеться прочитать из входного потока последовательность точек, однако мы не знаем, как много точек прочтем, и поэтому не можем сразу выделить память под вектор нужного размера, чтобы сразу прочитать туда данные. Вместо этого мы можем написать:

```
vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;

    while (cin>>buf) {
        if (buf==sentinel) return;
        cities.push_back(buf);          // проверяем новую точку
    }
}
```

Итератор, возвращаемый , указывает на только что вставленный элемент.
Это гарантирует, что вектор расширится насколько нужно. Если нам больше ничего не требуется, кроме как записать в ***vector*** новую точку, мы могли бы инициализировать ***cities*** прямо из входного потока, обеспечив соответствующий конструктор (§ 16.3.4). Однако обычно на входе производят небольшую обработку и по мере развития программы постепенно расширяют структуру данных; ***push_back*** () поддерживает такой подход.

В программах на C чаще всего пользуются функцией ***realloc*** () из стандартной библиотеки C. Таким образом ***vector*** — и вообще любой стандартный контейнер — обеспечивает более универсальную, более изящную и не менее эффективную альтернативу ***realloc*** ().

[1] То есть память выделяется с некоторым запасом (обычно на десять элементов). — *Примеч. ред.*

Размер вектора, возвращаемый функцией ***size*** (), неявно увеличивается функцией ***push_back*** (), так что вектор не может переполниться (до тех пор, пока есть доступная свободная память, см. § 19.4.1). Однако вектор может «переполниться» в другую сторону:

```
void f()
{
        vector<int> v;
        v.pop_back();          // неопределенный эффект: состояние v
                               // становится неопределенным
        v.push_back(7);        // неопределенный эффект: состояние v
                               // не определено, возможно оно плохое
}
```

Эффект переполнения снизу не определен, но очевидная реализация функции ***pop_back*** () приводит к тому, что произойдет запись в область памяти, не принадлежащей вектору. Как и переполнение сверху, переполнения снизу следует избегать.

16.3.6. Операции, характерные для списков

Операции ***push_back*** (), ***pop_back*** () и ***back*** () (§ 16.3.5) позволяют эффективно использовать вектор в качестве стека. Кроме того, иногда бывает полезно добавлять элементы в середину вектора или удалять их:

```
template<class T, class A = allocator<T> > class vector {
public:
        // ...
        // операции, характерные для списков
        iterator insert(iterator pos, const T& x);          // добавление перед pos
        void insert(iterator pos, size_type n, const T& x);

        template<class In>     // In должен быть итератором для чтения (§ 19.2.1)
                void insert(iterator pos, In first, In last);   // вставка элементов
                                                                // из последовательности
        iterator erase(iterator pos);                       // удаление элемента из pos
        iterator erase(iterator first, iterator last);      // удаление последовательности
        void clear();                                       // удаление всех элементов

        // ...
};
```

Итератор, возвращаемый ***insert*** (), указывает на только что вставленный элемент. Итератор, возвращаемый ***erase*** (), указывает на элемент, следующий за последним удаленным элементом.

Чтобы посмотреть, как работают эти операции, давайте проделаем несколько (бессмысленных) действий с вектором фруктов. Сначала определим контейнер ***vector*** и «заселим» его названиями фруктов:

```
vector<string> fruit;

fruit.push_back("peach");
fruit.push_back("apple");
fruit.push_back("kiwifruit");
fruit.push_back("pear");
fruit.push_back("starfruit");
fruit.push_back("grape");
```

Если мне не нравятся фрукты на букву *'p'*, я могу удалить их названия:

```
sort (fruit.begin (),fruit.end ());
vector<string>::iterator p1 = find_if (fruit.begin (),fruit.end (), initial ('p'));
vector<string>::iterator p2 = find_if (p1,fruit.end (), initial_not ('p'));
fruit.erase (p1,p2);
```

Другими словами, мы сортируем *vector*, находим первый и последний фрукты на букву *'p'* и удаляем из *fruit* эти элементы. Как написать проверочную функцию *initial* (*x*) (первая буква *x*?) и *initial_not* (*x*) (первая буква не *'x'*?) объясняется в § 18.4.2.

Операция *erase* (*p1*,*p2*) удаляет элементы с *p1*-го до *p2*-го, не включая последний. Это можно проиллюстрировать графически:

```
fruit[]:
                        p1          p2
                        |           |
                        v           v
    apple   grape   fruit   peach   pear   starfruit
```

Функция *erase* (*p1*, *p2*) удалит *peach* и *pear*, оставив:

```
fruit[]:
    apple   grape   kiwifruit   starfruit
```

Как обычно, последовательность, затрагиваемая операцией, определяется ее первым элементом и элементом, следующим за последним.

Было бы соблазнительно написать так:

```
vector<string>::iterator p1 = find_if (fruit.begin (),fruit.end (), initial ('p'));
vector<string>::reverse_iterator p2 = find_if (fruit.rbegin (),fruit.rend (), initial ('p'));
fruit.erase (p1,p2+1);          // ошибка: неверный тип
```

Однако *vector<fruit>::iterator* и *vector<fruit>::reverse_iterator* вовсе не должны быть одного и того же типа, поэтому мы не можем рассчитывать, что *erase* () скомпилируется. Чтобы использовать *reverse_iterator* вместе с *iterator*, *reverse_iterator* должен быть явно преобразован:

```
fruit.erase (p1,p2.base ());          // извлекаем iterator из reverse_iterator (§ 19.2.5)
```

Удаление элемента вектора изменяет его размер, а элементы после удаленного переносятся в освободившиеся позиции. В данном примере *fruit.size* () становится равным 4, а *starfruit*, бывшая *fruit*[*5*], теперь стала *fruit*[*3*].

Естественно, можно удалить и какой-нибудь один элемент. В этом случае нужен только один итератор для этого элемента (а не пара итераторов). Например:

```
fruit.erase (find (fruit.begin (),fruit.end (),"starfruit"));
fruit.erase (fruit.begin ()+1);
```

удаляет *starfruit* и *grape*, оставляя во *fruit* два элемента:

```
fruit[]:
    apple   kiwifruit
```

Элемент можно вставить в вектор. Например:

```
fruit.insert (fruit.begin ()+1,"cherry");
fruit.insert (fruit.end (),"cranberry");
```

Новый элемент вставляется перед указанной позицией, а элементы до конца сдвигаются, чтобы освободить место. Мы получим:

```
fruit[]:
    apple    cherry    kiwifruit    cranberry
```

Отметим, что *f.insert (f.end (), x)* эквивалентно *f.push_back (x)*.

Мы можем также вставить целую последовательность:

```
fruit.insert (fruit.begin ()+2, citrus.begin (), citrus.end ());
```

Если *citrus* представляет из себя следующий контейнер

```
citrus[]:
    lemon    grapefruit    orange    lime
```

мы получим:

```
fruit[]:
    apple    cherry    lemon    grapefruit    orange    lime    kiwifruit    cranberry
```

Элементы контейнера *citrus* копируются в вектор *fruit* функцией *insert ()*. Значение *citrus* остается неизменным.

Ясно, что *insert ()* и *erase ()* более универсальны, чем операции, затрагивающие только конец вектора (§ 16.3.5). Вообще говоря, они дороже, потому что при вставке или удалении не в конец контейнера приходится копировать элементы. Если вставка и удаление — распространенные операции, возможно, лучше использовать списки, а не вектора. Списки *list* оптимизированы для операций *insert ()* и *erase ()*, а не для обращений по индексу (§ 16.3.3).

Вставка и удаление из вектора (из списка или ассоциированного итератора вроде *map*) потенциально сдвигает элементы. Следовательно, итератор, указывающий на какой-то элемент в векторе, может после *insert ()* или *erase ()* указывать на другой элемент или вообще не на элемент. Чтобы выделить память для нового элемента, *insert ()* может заставить вектор перенестись в новую область памяти. Подобным же образом после удаления элемента функцией *erase ()* итераторы указывают не на те элементы. Никогда не обращайтесь к элементам через неправильный итератор: результат этого не определен и с большой вероятностью приведет к печальным последствиям. В частности, остерегайтесь использовать итератор, который применялся для указания места вставки: *insert ()* делает свой первый аргумент недействительным. Например:

```
void duplicate_elements (vector<string>& f)
{
    for (vector<string>::iterator p = f.begin (); p!=f.end (); ++p)
        f.insert (p, *p);          // нет!
}
```

Просто помните об этом (§ 16.5[15]). Реализация контейнера *vector* может сдвинуть все элементы — или по крайней мере все элементы после *p*, чтобы выделить место для нового элемента.

Операция *clear ()* удаляет все элементы контейнера. Таким образом, *c.clear ()* — это краткая запись для *c.erase (c.begin (), c.end ())*. После *c.clear ()* *c.size ()* равно 0.

16.3.7. Адресация элементов

Чаще всего объектом применения операций *erase ()* и *insert ()* является хорошо известное место в контейнере (например *begin ()* или *end ()*), результат поиска (*find ()*) или место, найденное путем итераций. В таких случаях мы имеем итератор, указывающий на подхо-

дящий элемент. Однако часто мы обращаемся к элементам вектора по индексу. Как нам получить итератор, подходящий в качестве аргумента для функций *erase* () или *insert* (), для элемента с индексом *7* в контейнере *c* типа *vector* (или подобного *vector*)? Поскольку это седьмой элемент от начала, правильный ответ — *c.begin* ()+*7*. Других альтернатив, которые могут казаться разумными по аналогии с массивами, следует избегать. Рассмотрим разные возможности:

```
template<class C> void f(C& c)
{
    c.erase(c.begin()+7);    // правильно (если итератор с поддерживает + (см. § 19.2.1))
    c.erase(&c[7]);          // не универсально
    c.erase(c+7);            // ошибка: прибавление 7 к контейнеру бессмысленно
    c.erase(c.back());       // ошибка: c.back() — ссылка, а не итератор
    c.erase(c.end()-2);      // правильно (пред-пред-последний элемент)
    c.erase(c.rbegin()+2);   // ошибка: векторные reverse_iterator
                             // и iterator — разные типы
    c.erase((c.rbegin()+2).base()); // неясно, но сойдет (см. § 19.2.5)
}
```

Самая соблазнительная альтернатива, &*c*[*7*], действительно работает с очевидной реализацией *vector*, где *c*[*7*] ссылается на элемент, а его адрес является корректным итератором. Однако *c* может оказаться контейнером, итератор которого не является простым указателем на элемент. Напрмер, оператор индексирования для ассоциативных массивов (§ 17.4.1.3) возвращает *mapped_type*&, а не указатель на элемент (*value_type*&).

Не все контейнеры допускают применение операции + к своим итераторам. Например, *list* не допускает даже *c.begin* ()+*7*. Если вам на самом деле необходимо увеличить *list*::*iterator* на *7*, воспользуйтесь многократно ++ или используйте *advance* ().

Варианты *c*+*7* и *c.back* () — просто ошибки в типе. Контейнер — не числовая переменная, к которой можно прибавить *7*, а *c.back* () — это элемент со значением вроде *"pear"*, не определяющий положения груши (pear) в контейнере *c*.

16.3.8. Размеры и емкость

Пока что вектор описывался с минимальным упоминанием о распределении памяти. По мере необходимости вектор растет. Обычно это и все, что нужно знать. Однако можно задать прямой вопрос, каким образом вектор получает память, а при случае стоит прямо вмешаться в этот процесс. Операции таковы:

```
template<class T, class A = allocator<T> > class vector {
public:
    // ...
    // емкость:
    size_type size() const;                  // число элементов
    bool empty() const { return size() == 0; }
    size_type max_size const;                // размер самого длинного
                                             // возможного вектора
    void resize(size_type sz, T val = T());  // добавленные элементы
                                             // инициализируются val

    size_type capacity() const;      // размер выделенной памяти (число элементов)
    void reserve(size_type n);       // выделяет место для всех n элементов;
                                     // не инициализирует
                                     // генерирует length_error, если n>max_size()
```

```
    //...
};
```

В любой момент времени вектор содержит информацию о числе элементов. Это число можно узнать, обратившись к функции ***size*** (), и изменить функцией ***resize*** (). Таким образом пользователь может определить размеры вектора и изменить их, если они кажутся ему недостаточными или избыточными. Например:

```
class Histogram {
    vector<int> count;
public:
    Histogram (int h) : count (max (h, 8)) { }
    void record (int i);
    //...
};

void Histogram::record (int i)
{
    if (i<0) i = 0;
    if (count.size ()<=i) count.resize (i+i);      // выделяет много памяти
    count[i]++;
}
```

Использование ***resize*** () для ***vector*** очень похоже на применение функции ***realloc*** () из стандартной библиотеки C для C-массива, располагаемого в свободной памяти.

Когда размеры вектора изменяются в соответствии с бóльшим (или меньшим) числом элементов, все его элементы могут переместиться в новую область памяти. Следовательно, хранить указатели на элементы в векторе, который может изменять свои размеры — идея неудачная, так как после функции ***resize*** () эти указатели могут указывать на незанятую область памяти. Вместо этого мы можем хранить индексы. Отметим, что операции ***push_back*** (), ***insert*** () и ***erase*** () в неявном виде изменяют размеры вектора.

В дополнение к хранению элементов реализация может выделять некоторую область памяти для потенциального расширения. Программист, считающей, что такое расширение вполне вероятно, может приказать контейнеру ***vector*** зарезервировать место для будущего расширения, используя функцию ***reserve*** (). Например:

```
struct Link {
    Link* next;
    Link (Link* n=0) : next (n) { }
    //...
};

vector<Link> v;

void chain (size_t n)    // заполняет v n элементами типа Link,
                         // так чтобы каждый Link указывал на предшествующий
{
    v.reserve (n);
    v.push_back (Link (0));
    for (int i = 1; i<n; i++) v.push_back (Link (&v[i-1]));
    //...
}
```

Вызов *v.reserve* (*n*) гарантирует, что при увеличении *v* не понадобится никакого выделения памяти, пока *v.size* () не превзойдет *n*.

Резервирование памяти заранее имеет два преимущества. Во-первых, даже самая бесхитростная реализация может сразу выделить достаточно памяти одной операцией, а не требовать все время дополнительного выделения. Во-вторых, во многих случаях существует и логическое преимущество, которое перевешивает соображения эффективности. При возрастании *vector* элементы контейнера потенциально переносятся в другое место в памяти. Таким образом, связи установленные между элементами *v* в предыдущем примере, гарантируются только тем, что обращение к функции *reserve* () обеспечило отсутствие всякого выделения памяти, пока шло построение вектора. Итак, в некоторых случаях *reserve* () кроме преимуществ в эффективности дает гарантию корректности.

Той же гарантией можно воспользоваться, чтобы быть уверенным, что исчерпание памяти и потенциально дорогое перераспределение элементов состоится в предсказуемый момент. Для программ реального времени это может быть очень важно.

Отметим, что функция *reserve* () не изменяет размеры вектора. Таким образом, ей не приходится инициализировать каждый новый элемент. В обоих отношениях эта операция отличается от *resize* ().

Так же, как *size* () выдает текущее число элементов, *capacity* () сообщает нам текущее число зарезервированных мест; *c.capacity* ()–*c.size* () показывает, сколько еще элементов можно вставить без перераспределения памяти.

Уменьшение размеров вектора не уменьшает его емкости. Просто остается больше места для увеличения вектора впоследствии. Чтобы вернуть память системе, нужно сделать следующий трюк:

```
vector tmp = v;          // копия v с емкостью по умолчанию
v.swap (tmp);            // теперь v имеет емкость по умолчанию (см. § 16.3.9)
```

Контейнер *vector* получает необходимую для его элементов память путем вызова функций-членов своего распределителя памяти (§ 19.4.2) (переданного, как параметр шаблона). Распределитель памяти по умолчанию, называемый *allocator* (§ 19.4.1), использует для выделения памяти оператор *new*, так что если памяти больше нет, будет возбуждено исключение *bad_alloc*. Другие распределители памяти могут использовать другую стратегию (см. § 19.4.2).

Функции *reserve* () и *capacity* () уникальны для контейнера *vector* и схожих компактных контейнеров. Контейнеры типа списков ничего подобного не обеспечивают.

16.3.9. Другие функции-члены

Во многих алгоритмах — в том числе и важных алгоритмах сортировки — нужно менять элементы местами (swapping). Очевидный способ такой «перемены мест» (§ 13.5.2) — просто копировать элементы. Однако *vector*, как правило, реализуется структурой, которая действует как дескриптор (handle) для элементов (§ 13.5, § 17.1.3). Таким образом, два вектора могут быть переставлены более эффективно путем обмена их дескрипторов; это делает функция *vector::swap* (). Разница во времени выполнения между этой функцией и *swap* () по умолчанию в важных случаях отличается на несколько порядков:

```
template<class T, class A = allocator<T> > class vector {
```

```
public:
        // ...
        void swap (vector&);
        allocator_type get_allocator () const;
};
```

Функция *get_allocator* () дает программисту возможность управлять размещением вектора (§ 16.3.1, § 16.3.4). Как правило, смысл этого — гарантировать, что данные из прикладной программы, связанные с векторами, распределены в памяти схожим образом (также как и вектора) (§ 19.4.1).

16.3.10. Функции-помощники

Два вектора можно сравнить при помощи операторов == и <:

```
template<class T, class A>
bool std::operator== (const vector<T, A>& x, const vector<T, A>& y);

template<class T, class A>
bool std::operator< (const vector<T, A>& x, const vector<T, A>& y);
```

Два вектора *v1* и *v2* считаются равными (==), если *v1.size* ()==*v2.size* () и *v1*[*n*]==*v2*[*n*] для любого допустимого индекса *n*. Аналогичным образом, < — это лексикографическое упорядочивание. Иными словами, оператор < для векторов можно определить так:

```
template<class T, class A>
inline bool std::operator< (const vector<T, A>& x, const vector<T, A>& y)
{
        return lexicographical_compare (x.begin (), x.end (), y.begin (), y.end ()); // см. § 18.9
}
```

Это означает, что *x* меньше *y*, если первый *x*[*i*], не равный соответствующему элементу *y*[*i*], меньше, чем *y*[*i*], или *x.size* ()<*y.size* () при равенстве всех *x*[*i*] соответствующим *y*[*i*].

Стандартная библиотека также предоставляет операторы !=, <=, > и >= с определениями, соответствующими == и <.

Поскольку функция *swap* () — член класса, она вызывается с синтаксисом *v1.swap* (*v2*). Однако не любой тип имеет функцию-член *swap* (), поэтому в обобщенных алгоритмах пользуются общепринятым синтаксисом *swap* (*a*,*b*). Чтобы заставить это работать и для векторов, стандартная библиотека обеспечивает специализацию:

```
template<class T, class A> void std::swap (vector<T, A>& x, vector<T, A>& y)
{
        x.swap (y);
}
```

16.3.11. vector<bool>

Специализация (§ 13.5) *vector<bool>* вводится как компактный вектор, состоящий из элементов *bool*. К переменной типа *bool* можно обращаться по адресу, поэтому

она должна занимать по крайней мере один байт. Однако нетрудно реализовать *vector<bool>* так, что каждый элемент займет только один бит.

Обычные операции с векторами работают и для *vector<bool>* и сохраняют свой обычный смысл. В частности, обращение по индексу и итерации работают так, как от них и ожидается. Например:

```
void f(vector<bool>& v)
{
    // итерация через индексы
    for (int i = 0; i<v.size(); ++i) cin >> v[i];

    typedef vector<bool>::const_iterator VI;
    // итерация при помощи итераторов
    for (VI p = v.begin(); p!=v.end(); ++p) cout << *p;
}
```

Чтобы добиться этого, реализация должна имитировать адресацию одного бита. Поскольку указатель не может адресовать единицу памяти, меньшую чем байт, итератор *vector<bool>::iterator* не может быть указателем. В частности, нельзя рассматривать *bool** как итератор для *vector<bool>*:

```
void f(vector<bool>& v)
{
    bool* p = v.begin();          // ошибка: несоответствие типов
    // ...
}
```

Техника адресации одного бита в общих чертах описана в § 17.5.3.

Библиотека также обеспечивает *bitset* как множество логических значений с набором логических операций над множествами (§ 17.5.3).

16.4. Советы

[1] Для сохранения переносимости пользуйтесь средствами стандартной библиотеки; § 16.1.

[2] Не пытайтесь переопределить средства стандартной библиотеки; § 16.2.

[3] Не верьте, что стандартная библиотека лучше всего подходит для любых задач.

[4] Создавая новое средство, подумайте, нельзя ли его получить даром в рамках стандартной библиотеки; § 16.3.

[5] Помните, что средства стандартной библиотеки определены в пространстве имен *std*; § 16.1.2.

[6] Объявляйте средства стандартной библиотеки, включая ее заголовочный файл, а не явным объявлением; § 16.1.2.

[7] Пользуйтесь преимуществом поздней абстракции; § 16.2.1.

[8] Избегайте жирных интерфейсов; § 16.2.2.

[9] Предпочитайте алгоритмы с обратными итераторами, а не явные циклы от конца в начало; § 16.3.2.

[10] Для извлечения *iterator* из *reverse_iterator* пользуйтесь *base*(); § 16.3.2.

[11] Передавайте контейнеры по ссылке; § 16.3.4.

[12] Для обращения к элементам контейнера пользуйтесь итераторными типами, такими как *list<char>::iterator*, а не указателями; § 16.3.1.

[13] Если вам не нужно модифицировать элементы контейнера, пользуйтесь константными итераторами; § 16.3.1.
[14] Если хотите проверять диапазон, пользуйтесь операцией ***at*** (), прямо или косвенно; § 16.3.3.
[15] Применяйте ***push_back*** () или ***resize*** () для контейнера, а не ***realloc*** () применительно к массиву; § 16.3.5.
[16] Не применяйте итераторы в векторе, с измененным размером; § 16.3.8.
[17] Чтобы избежать превращения итераторов в недействительные, пользуйтесь ***reserve*** (); § 16.3.8.
[18] При необходимости пользуйтесь ***reserve*** (), чтобы сделать быстродействие предсказуемым; § 16.3.8.

16.5. Упражнения

Решение некоторых задач к этой главе можно найти, просмотрев исходный текст реализации стандартной библиотеки. Сделайте одолжение — попытайтесь найти собственные решения, прежде чем смотреть, как подошел к этим проблемам разработчик вашей библиотеки.

1. (*1.5) Создайте ***vector<char>***, содержащий буквы в алфавитном порядке. Распечатайте элементы этого вектора в прямом и в обратном порядке.
2. (*1.5) Создайте ***vector<srtring>*** и считайте в него список названий фруктов из ***cin***. Отсортируйте список и распечатайте.
3. (*1.5) Используя вектор из § 16.5[2], напишите цикл распечатки названий фруктов, начинающихся на букву ***a***.
4. (*1) Используя вектор из § 16.5[2], напишите цикл, удаляющий все фрукты, названия которых начинаются на букву ***a***.
5. (*1) Используя вектор из § 16.5[2], напишите цикл, удаляющий все цитрусовые.
6. (*1.5) Используя вектор из § 16.5[2], напишите цикл, удаляющий все фрукты, которые вы не любите.
7. (*2) Завершите классы ***Vector***, ***List*** и ***Itor*** из § 16.2.1.
8. (*2.5) Взяв класс ***Itor***, подумайте, как обеспечить итераторы для перебора в возрастающем порядке, в убывающем порядке, для перебора контейнера, способного изменяться при итерации, и перебора неизменяющегося (immutable) контейнера. Организуйте этот набор итераторов, так чтобы пользователь мог взаимозаменяемым образом использовать итераторы, обеспечивающие достаточную функциональность алгоритмов. Минимизируйте повторение фрагментов кода в реализации этих контейнеров. Какие еще итераторы могут понадобиться пользователю? Перечислите достоинства и недостатки вашего подхода.
9. (*2) Завершите классы ***Container***, ***Vector*** и ***List*** из § 16.2.2.
10. (*2.5) Сгенерируйте 10 000 равномерно распределенных случайных чисел в диапазоне от 0 до 1023 и сохраните их: а) в стандартном библиотечном контейнере ***vector***; б) в контейнере ***Vector*** из § 16.5[7]; в) в контейнере ***Vector*** из § 16.5[9]. Во всех случаях сосчитайте арифметическое среднее элементов вектора (как если бы вы его еще не знали). Засеките время. Оцените, измерьте и сравните потребление памяти в этих трех стилях векторов.
11. (*1.5) Напишите итератор, чтобы ***Vector*** из § 16.2.2 использовался как контейнер в стиле § 16.2.1.

12. (*1.5) Напишите класс, производный от ***Container***, чтобы ***Vector*** из § 16.2.1 использовался как контейнер в стиле § 16.2.2.
13. (*2) Напишите классы, чтобы ***Vector*** из § 16.2.1 и ***Vector*** из § 16.2.2 использовались как стандартные контейнеры.
14. (*2) Напишите шаблон, реализующий контейнер с теми же функциями-членами и типами членов, как и стандартный ***vector***, для существующего (нестандартного, неучебного) контейнерного типа. Не модифицируйте уже существующий контейнерный тип. Как бы вы стали работать с функциональностью, предлагаемой нестандартным ***vector***, но не предоставляемой стандартным?
15. (*1.5) Вкратце обрисуйте возможное поведение функции ***duplicate_elements*** () из § 16.3.6 для ***vector<string>***, состоящего из трех элементов: ***don't do this*** (не делай этого).

17

Стандартные контейнеры

Теперь самое время поставить вашу работу
на твердый теоретический фундамент.
— Сэм Морган

Стандартные контейнеры — обзор операций и контейнеров — эффективность — представление — требования к элементам — последовательности — вектора — списки —очереди с двумя концами — адаптеры — стек — очереди — очереди с приоритетами — ассоциативные контейнеры — ассоциативные массивы — сравнения — контейнеры ***multimap*** — множества — множества с дублированием элементов — «почти контейнеры» — битовые наборы — массивы — хэш-таблицы — реализация ***hash_map*** — советы — упражнения.

17.1. Стандартные контейнеры

В стандартной библиотеке определены два типа контейнеров — последовательности и ассоциативные контейнеры. Все последовательности очень похожи на ***vector*** (§ 16.3). Если не сказано обратное, типы членов и функции, упоминавшиеся для векторов, могут с тем же эффектом использоваться и для любого другого контейнера. Но ассоциативные контейнеры дополнительно предоставляют доступ к элементам на основе ключей (§ 3.7.4).

Встроенные массивы (§ 5.2), строки ***string*** (глава 20), ***valarray*** (§ 22.4) и битовые наборы ***bitset*** (§ 17.5.3) содержат элементы, а следовательно могут считаться контейнерами. Однако эти типы не являются полностью разработанными стандартными контейнерами. Если бы они являлись таковыми, это помешало бы их основному предназначению. Например, встроенный массив не может одновременно содержать собственный размер и оставаться совместимым по размещению в памяти с массивами языка С.

Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться ***map*** (ассоциативным массивом) (§ 17.4.1). С другой стороны, если преобладают универсальные операции, характерные для списков, можно воспользоваться контейнером ***list*** (§ 17.2.2). Если добавления и удаления элементов часто производятся в концы контейнера, следует подумать об использовании ***deque*** (очереди с двумя концами, § 17.2.3), ***stack*** (стека, § 17.3.1) или ***queue*** (оче-

реди, § 17.3.2). Кроме того, пользователь может разработать дополнительные контейнеры, вписывающиеся в рамки стандартных контейнеров (§ 17.6). По умолчанию должен использоваться *vector* (§ 16.3); он реализован, чтобы хорошо работать для самого широкого диапазона задач.

Идея обращения с различными видами контейнеров — и в более общем случае со всеми видами источников информации — унифицированным способом, ведет к понятию обобщенного (generic) программирования (§ 2.7.2, § 3.8). Для поддержки этой идеи стандартная библиотека содержит множество обобщенных алгоритмов (глава 18). Такие алгоритмы избавляют программиста от необходимости знать подробности отдельных контейнеров.

17.1.1. Обзор операций

В этом разделе перечислены общие и почти общие члены стандартных контейнеров. Если хотите узнать подробности, прочитайте заголовочные файлы вашей стандартной библиотеки (*<vector>*, *<list>*, *<map>* и т. д.; § 16.1.2).

Типы членов (§ 16.3.1)	
value_type	Тип элемента
allocator_type	Тип распределителя памяти
size_type	Тип индексов, счетчика элементов и т. п.
difference_type	Тип разности между итераторами
iterator	Ведет себя подобно *value_type**
const_iterator	Ведет себя подобно *const value_type**
reverse_iterator	Просматривает контейнер в обратном порядке; как *value_type**
const_reverse_iterator	Просматривает контейнер в обратном порядке; как *const value_type**
reference	Ведет себя подобно *value_type*&
const_reference	Ведет себя подобно *const value_type*&
key_type	Тип ключа (только для ассоциативных контейнеров)
mapped_type	Тип *mapped_value* (отображенного значения); только для ассоциативных контейнеров
key_compare	Тип критерия сравнения (только для ассоциативных контейнеров)

Контейнер может быть последовательно просмотрен или в порядке, определяемом его итератором, или в обратном порядке. Для ассоциативного контейнера порядок основывается на его (контейнера) критерии сравнения (по умолчанию это <).

Итераторы (§ 16.3.2)	
begin ()	Указывает на первый элемент
end ()	Указывает на элемент, следующий за последним
rbegin ()	Указывает на первый элемент в обратной последовательности
rend ()	Указывает на элемент, следующий за последним, в обратной последовательности.

К некоторым элементам можно обратиться прямо:

Доступ к элементам (§ 16.3.3)	
front ()	Первый элемент
back ()	Последний элемент
[]	Обращение по индексу, доступ без проверки (не для списка)
at ()	Обращение по индексу, доступ с проверкой (не для списка)

Векторы и очереди с двумя концами обеспечивают эффективные операции с концом (back) своей последовательности элементов. Кроме того списки и очереди с двумя концами обеспечивают такие же операции в начале (front) своей последовательности.

Операции со стеком и очередями (§ 16.3.5, § 17.2.2.2)	
push_back ()	Добавление в конец
pop_back ()	Удаление последнего элемента
push_front ()	Добавление нового первого элемента (только для списков и очередей с двумя концами)
pop_front ()	Удаление первого элемента (только для списков и очередей с двумя концами)

Контейнеры обеспечивают операции со списками:

Операции со списками (§ 16.3.6)	
insert (*p*, *x*)	Добавление *x* перед *p*
insert (*p*, *n*, *x*)	Добавление *n* копий *x* перед *p*
insert (*p*, *first*, *last*)	Добавление элементов из [*first*:*last*[перед *p*
erase (*p*)	Удаление элемента в позиции *p*
erase (*first*, *last*)	Удаление [*first*:*last*[
clear ()	Удаление всех элементов

В приложении Д обсуждается поведение коттейнера, если выделение памяти или операция над элементами генерирует исключение.

Все контейнеры обеспечивают операции, связанные с получением информации о числе элементов, и несколько других операций:

Другие операции (§ 16.3.8, § 16.3.9, § 16.3.10)	
size ()	Число элементов
empty ()	Контейнер пуст?
max_size ()	Размер самого большого возможного контейнера
capacity ()	Память, выделенная под вектор (только для векторов)
reserve ()	Выделяет память под вектор (только для векторов)
resize ()	Изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)
swap ()	Обмен местами элементов двух контейнеров
get_allocator ()	Получает копию контейнерного распределителя памяти
==	Содержимое двух контейнеров совпадает?
!=	Содержимое двух контейнеров различается?
<	Один контейнер лексикографически меньше другого?

Контейнеры обеспечивают множество разных конструкторов и операций присваивания:

Конструкторы и пр. (§ 16.3.4)	
container ()	Пустой контейнер
container (*n*)	*n* элементов со значением по умолчанию (не для ассоциативных контейнеров)
container (*n*, *x*)	*n* копий *x* (не для ассоциативных контейнеров)
container (*first*, *last*)	Инициализирует элементы из [*first*:*last*[
container (*x*)	Копирующий конструктор; инициализирует элементами из контейнера *x*
~*container* ()	Уничтожает контейнер и все его элементы

Присваивания (§ 16.3.4)	
operator= (*x*)	Копирующее присваивание; берутся элементы из контейнера *x*
assign (*n*, *x*)	Присваивание *n* копий *x* (не для ассоциативных контейнеров)
assign (*first*, *last*)	Присваивание из [*first*:*last*[

Ассоциативные контейнеры обеспечивают поиск элементов по ключам:

Ассоциативные операции (§ 17.4.1)	
operator[] (*k*)	Доступ к элементу с ключом *k* (для контейнеров с уникальным ключом)
find (*k*)	Находит элемент с ключом *k*
lower_bound (*k*)	Находит первый элемент с ключом *k*
upper_bound (*k*)	Находит первый элемент с ключом больше *k*
equal_range (*k*)	Находит *lower_bound* (нижнюю границу) и *upper_bound* (верхнюю границу) элементов с ключом *k*
key_comp ()	Копирует объект с совпавшим ключом
value_comp ()	Копирует объект с совпавшим *mapped_value* (отображенным значением)

В дополнение к этим распространенным операциям большинство контейнеров обеспечивает несколько специализированных операций.

17.1.2. Краткий обзор контейнеров

Стандартные контейнеры вкратце можно описать примерно так:

Стандартные операции с контейнерами					
	[]	Операции со списками	Операции с началом	Операции с концом (стековые)	Итераторы
	§ 16.3.3 § 17.4.1.3	§ 16.3.6 § 20.3.9	§ 17.2.2.2 § 20.3.9	§ 16.3.5 § 20.3.12	§ 19.2.1
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
deque	const	O(n)	const	const	Ran

Стандартные операции с контейнерами *(продолжение)*

	[]	Операции со списками	Операции с началом	Операции с концом (стековые)	Итераторы
	§ 16.3.3 § 17.4.1.3	§ 16.3.6 § 20.3.9	§ 17.2.2.2 § 20.3.9	§ 16.3.5 § 20.3.12	§ 19.2.1
stack		O(n)		const+	
queue		O(n)	const	const+	
priority_queue		O(log(n))	O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set	O(log(n))	O(log(n))+			Bi
multiset		O(log(n))+			Bi
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
valarray	const				Ran
bitset	const				

В столбце *Итераторы* ***Ran*** означает итератор с произвольным доступом, а ***Bi*** — двунаправленный итератор; операции для двунаправленных итераторов представляют собой подмножество операций для итераторов с произвольным доступом (§ 19.2.1). Записи в других ячейках — это степень эффективности операций. Запись ***const*** означает, что операция занимает время, которое не зависит от числа элементов в контейнере. Другое принятое обозначение для *постоянного времени* — ***O*** (*1*). ***O*** (*n*) означает, что время пропорционально числу участвующих в операции элементов. Суффикс + показывает, что иногда операция может оказаться более дорогой. Например, вставка элемента в список имеет фиксированную стоимость (эта операция отмечена ***const***), в то время как та же операция с вектором приводит к перемещению всех элементов после места вставки (для вектора эта операция отмечена как ***O*** (*n*)). Иногда нужно переместить все элементы (поэтому я добавил +). «Большое ***O***» — общепринятая запись. Я добавил + для программистов, заботящихся кроме среднего быстродействия еще и о предсказуемости. Общепринятый термин для ***O*** (*n*)+ — это *амортизированное линейное время*.

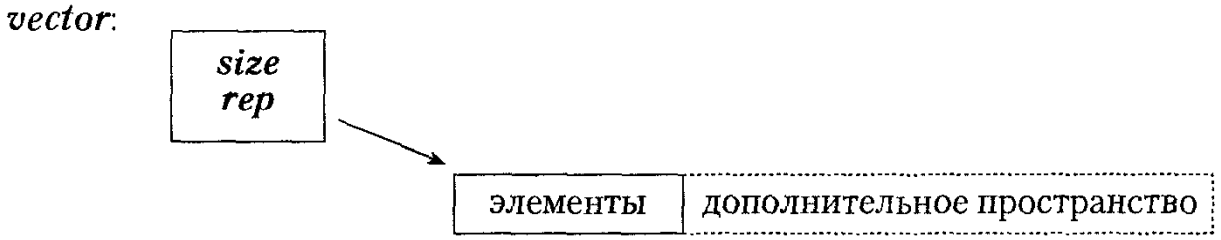
Естественно, если число элементов невелико, то постоянное время может оказаться больше, чем ***O*** (*n*). Однако для больших структур данных ***const*** может считаться «дешево», ***O*** (*n*) — «дорого», а ***O*** (*log* (*n*)) — «довольно дешево». Даже для умеренно больших значений *n* время ***O*** (*log* (*n*)) ближе к константе, чем к ***O*** (*n*). Те, кого заботит цена, должны присмотреться к этому повнимательнее. В частности, вы должны понять, какие элементы учитываются в *n*. Никакая базовая операция не бывает «слишком дорогой», то есть ***O*** (*n*n*) или хуже.

Не считая ***string***, перечисленные оценки стоимости отражают требования к стандарту. Оценки для ***string*** — это мои допущения. Данные для ***stack*** и ***queue*** отражают стоимость реализации по умолчанию с использованием ***deque*** (§ 17.3.1, § 17.3.2).

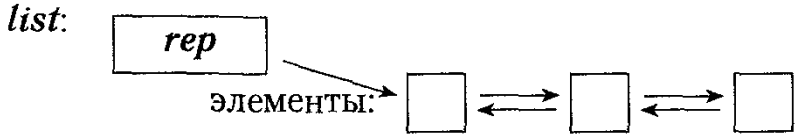
Приведенные оценки сложности и цены являются оценками сверху. Они существуют для того, чтобы дать пользователям какую-то ориентацию в том, что можно ожидать от реализации операций. Естественно, в важных случаях разработчики библиотеки постараются реализовать функции получше.

17.1.3. Представление

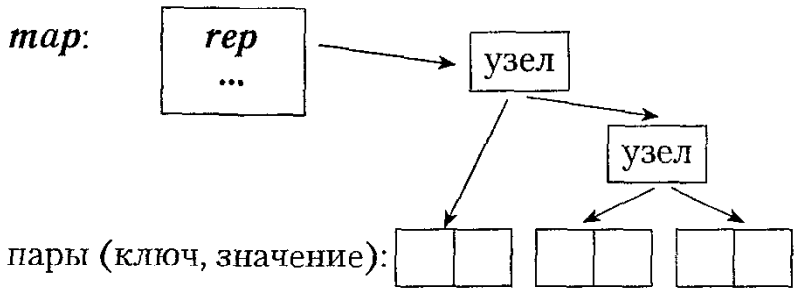
Стандарт не предписывает специального представления для каждого стандартного контейнера. Вместо этого он определяет интерфейсы контейнеров и некоторые требования к сложности. Чтобы удовлетворить общим требованиям, разработчики библиотеки выберут подходящие и часто тщательно оптимизированные реализации. Контейнер почти наверняка будет представлен структурой данных, содержащей элементы, доступ к которым осуществляется через дескриптор, содержащий информацию о размере и емкости. Для векторов структура элементов данных является непрерывной последовательностью этих элементов и очень напоминает массив:



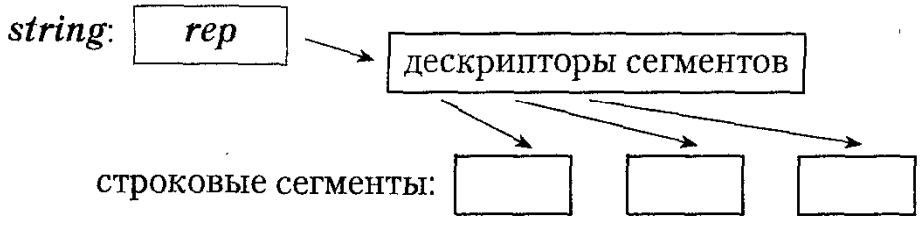
Аналогично, список наилучшим образом представляется набором связей, указывающих на элементы:



Ассоциативный массив скорее всего реализуется как (сбалансированное) дерево узлов, указывающих на пары (ключ, значение):



Строки можно реализовать, как описано в § 11.12, или, например, как последовательность массивов, каждый из которых содержит не слишком много символов:



Здесь *rep* обозначает «представление» (representation).

17.1.4. Требования к элементам

Элементы в контейнере — это копии вставленных объектов. Таким образом, чтобы стать элементом контейнера, объект должен быть такого типа, который позволил бы реализации контейнера скопировать его. Контейнер может копировать элементы при помощи копирующего конструктора или присваивания; в обоих случаях получающаяся копия

должна быть эквивалентным объектом. Грубо говоря, это означает, что любая придуманная вами проверка значений объектов на равенство должна считать копию равной оригиналу. Другими словами, копирование элемента должно работать очень похоже на обычное копирование для встроенных типов (включая указатели). Например:

```
X& X::operator= (const X& a)        // правильный оператор присваивания
{
        // копирование всех членов a в *this
        return *this;
}
```

делает ***X*** подходящим для типа элемента в стандартном контейнере, но

```
void Y::operator= (const Y& a)      // неправильный оператор присваивания
{       // обнуление всех членов a
}
```

обрабатывает ***Y*** неправильным образом, потому что присваивание ***Y*** не имеет ни принятого возвращаемого типа, ни традиционного смысла.

Некоторые нарушения правил для стандартных контейнеров могут быть выявлены компилятором, но другие не выявляются и могут привести к неожиданностям в поведении программы. Например, операция присваивания, генерирующая исключение, может оставить какой-нибудь элемент лишь частично скопированным. Она может оставить даже сам контейнер в состоянии, которое потом принесет неприятности. Такие операции копирования сами по себе являются плохо спроектированными (§ 14.4.6.1, приложение Д).

Когда копирование элементов — это не совсем то, что нужно, альтернативой служит помещение в контейнер вместо самих объектов указателей на объекты. Наиболее очевидный пример — полиморфные типы (§ 2.5.4, § 12.2.6). Например, чтобы сохранить полиморфное поведение, мы используем ***vector<Shape*>***, а не ***vector<Shape>***.

17.1.4.1. Сравнения

Ассоциативные контейнеры требуют, чтобы их элементы могли быть упорядочены. То же самое касается многих операций, которые можно применить к контейнерам (например, ***sort*** ()). По умолчанию для определения порядка используется оператор <. Если < не подходит, программист может обеспечить альтернативу (§ 17.4.1.5, § 18.4.2). Критерий упорядочивания должен определить *строгое слабое упорядочение* (strict weak ordering). Неформально это означает, что «меньше» и «равно» должны быть транзитивными. То есть для критерия упорядочения ***cmp***:

[1] ***cmp*** (***x***, ***x***) есть ***false***.
[2] Если ***cmp*** (***x***, ***y***) и ***cmp*** (***y***, ***z***), то ***cmp*** (***x***, ***z***).
[3] Определим ***equiv*** (***x***, ***y***) как ! (***cmp*** (***x***, ***y***) || (***cmp*** (***y***, ***x***)). Если ***equiv*** (***x***, ***y***) и ***equiv*** (***y***, ***z***), то ***equiv*** (***x***, ***z***).

Рассмотрим:

```
// для сравнения используем <
template<class Ran> void sort (Ran first, Ran last);
// используем cmp
template<class Ran, class Cmp> void sort (Ran first, Ran last, Cmp cmp);
```

Первая версия пользуется оператором <, а вторая — пользовательской функцией сравнения ***cmp*** (). Например, мы могли бы решить отсортировать фрукты ***fruit*** при помощи

сравнения без учета регистра. Мы сделаем это, определив объект-функцию (§ 11.9, § 18.4), который осуществляет сравнение, будучи вызванным для пары строк *string*:

```
class Nocase {    // сравнение без учета регистра
public:
        bool operator () (const string&, const string& y) const;
};

bool Nocase::operator () (const string& x, const string& y) const
// возвращает true, если x лексикографически меньше чем y,
// не принимая в расчет регистр
{
        string::const_iterator p = x.begin ();
        string::const_iterator q = y.begin ();

        while (p!=x.end () && q!=y.end () && toupper (*p)==toupper (*q)) {
                ++p;
                ++q;
        }
        if (p == x.end ()) return q != y.end ();
        if (q == y.end ()) return false;
        return toupper (*p) < toupper (*q);
}
```

Пользуясь этим критерием сравнения, мы можем вызвать ***sort*** ().

Например: возьмем контейнер:

```
fruit:
    apple     pear     Apple     Pear     lemon
```

Отсортировав при помощи ***sort*** (*fruit.begin* (), *fruit.end* (), *Nocase* ()), получим:

```
fruit:
    Apple     apple     lemon     Pear     pear
```

в то время как простое ***sort*** (*fruit.begin* (), *fruit.end* ()) может дать что-то вроде:

```
fruit:
    Apple     Pear     apple     lemon     pear
```

в предположении, что в наборе символов сначала идут заглавные, а потом строчные буквы.

Имейте в виду, что для С-строк (то есть *char**) оператор < не определяет лексикографического порядка (§ 13.5.2). Таким образом, при использовании в качестве ключей С-строк ассоциативные контейнеры будут работать не так, как ожидало бы большинство людей. Чтобы заставить их работать должным образом, нужно пользоваться операцией «меньше», производящей сравнение, основываясь на лексикографическом порядке. Например:

```
struct Cstring_less {
        bool operator () (const char* p, const char* q)
        const { return strcmp (p, q)<0; }
};

map<char*, int, Cstring_less> m;        // ассоциативный массив, который использует
                                        // strcmp() для сравнения ключей const char*
```

17.1.4.2. Другие операторы отношения

По умолчанию контейнеры и алгоритмы, когда нужно произвести сравнение на «меньше», пользуются оператором <. Когда умолчание не подходит, программист может задать свой критерий сравнения. Однако для проверки на равенство никакого механизма не обеспечивается. Вместо этого, когда программист предоставляет функцию сравнения *cmp* (), равенство проверяется при помощи двух сравнений. Например:

```
if (x == y)   // не делается, если пользователь обеспечил свое сравнение
if (!cmp (x, y) && !cmp (y, x))        // вот что делается, если пользователь
                                       // обеспечил свое сравнение
```

Так мы избавляемся от необходимости добавлять к каждому ассоциативному контейнеру и большинству алгоритмов критерий равенства. Это может показаться расточительным, но библиотека не так уж часто обращается к сравнению, а в 50% случаев требуется всего лишь одно обращение к *cmp* ().

Применение отношения равенства, определенного функцией «меньше» (по умолчанию оператором <), а не функцией «равно» (по умолчанию оператором ==), имеет также и практическое применение. Например, ассоциативные контейнеры (§ 17.4) сравнивают ключи при помощи такой проверки на эквивалентность:

```
!(cmp (x, y) || (cmp (y, x))
```

Это подразумевает, что эквивалентные ключи не обязаны быть равны. Например, *multimap* (§ 17.4.2), использующий критерий сравнения без учета регистра, будет считать строки ***Last***, ***last***, ***lAst***, ***laSt*** и ***lasT*** эквивалентными даже при том, что оператор == для строк не считает их равными. Это позволяет нам не принимать во внимание различия, которые мы считаем несущественными при сортировке.

Имея операторы < и == , мы можем легко сконструировать остальные часто применяемые сравнения. Стандартная библиотека определяет их в пространстве имен *std*::***rel_ops*** и представляет в заголовочном файле <***utility***>:

```
template<class T> bool rel_ops::operator!= (const T& x, constT& y)
        { return !(x==y); }
template<class T> bool rel_ops::operator> (const T& x, constT& y)
        { return y<x; }
template<class T> bool rel_ops::operator<= (const T& x, constT& y)
        { return !(y<x); }
template<class T> bool rel_ops::operator>= (const T& x, constT& y)
        { return !(x<y); }
```

Поместив эти операции в ***rel_ops***, мы гарантируем, что при необходимости ими легко воспользоваться, и при этом они не создаются неявно, если только их не «вытащить» из пространства имен:

```
void f()
{
        using namespace std;
        // !=, > и т. д., не порождаются по умолчанию
}
void q()
{
```

```
        using namespace std;
        using namespace std::rel_ops;
        // !=, > и т. д., порождаются по умолчанию
}
```

Операции != и т. д. не определены прямо в ***std*** потому, что они не всегда нужны, и иногда их определение может помешать пользовательской программе. Например, если бы я писал обобщенную математическую библиотеку, мне бы захотелось иметь свои собственные операторы сравнения, а не версии из стандартной библиотеки.

17.2. Последовательности

Последовательности следуют модели, описанной для векторов (§ 16.3). Стандартная библиотека обеспечивает следующие фундаментальные последовательности:

- вектора (***vector***);
- списки (***list***);
- очереди с двумя концами (***deque***).

Из них, предоставлением соответствующего интерфейса, созданы

- стеки (***stack***);
- очереди (***queue***);
- очереди с приоритетами (***priority_queue***).

Эти последовательности называются *контейнерными адаптерами*, *адаптерами последовательностей* или просто *адаптерами* (§ 17.3).

17.2.1. Вектора

Стандартный вектор ***vector*** подробно описан в § 16.3. Средства для резервирования памяти (§ 16.3.8) уникальны для ***vector***. По умолчанию при использовании индексации [] диапазон не проверяется. Если нужна проверка, пользуйтесь ***at*** () (16.3.3), вектором с проверкой (§ 3.7.2) или итератором с проверкой (§ 19.3). Контейнер ***vector*** обеспечивает итераторы с произвольным доступом (§ 19.2.1).

17.2.2. Списки

Список ***list*** — это последовательность, оптимизированная для вставки и удаления элементов. По сравнению с вектором (и ***deque***; § 17.2.2) обращение к элементу списка по индексу оказалось бы чрезвычайно медленным, поэтому оно для него не предоставляется. И поэтому список обеспечивает двунаправленные итераторы (§ 19.2.1), а не итераторы с произвольным доступом. Это приводит к тому, что список, как правило, реализуется при помощи некоторой формы двусвязного списка (см. § 17.8[16]).

Контейнер ***list*** обеспечивает все типы членов и операции, предлагаемые контейнером ***vector*** (§ 16.3), за исключением индексации, ***capacity*** () и ***reserve*** ():

```
template<class T, class A = allocator<T> > class std::list {
public:
        // типы и операции похожие на типы и операции
        // для векторов, кроме [], at(), capacity() и reserve()
        //...
};
```

17.2.2.1. Удаление-вставка, сортировка и слияние

В добавление к основным операциям с последовательностями ***list*** предоставляет операции, специально приспособленные для манипулирования со списками:

```
template<class T, class A = allocator<T> > class list {
public:
        //...
        // операции, специфичные для списков

        // перемещение всех элементов из x на место
        // перед pos в данном списке без копирования
        // (удаление-вставка)
        void splice (iterator pos, list& x);
        // перемещение *p из x на место перед pos без копирования
        void splice (iterator pos, list& x, iterator p);
        void splice (iterator pos, list& x, iterator first, iterator last);

        // объединение отсортированных списков
        void merge (list&);
        template<class Cmp> void merge (list&, Cmp);

        void sort ();
        template<class Cmp> void sort (Cmp);

        //...
};
```

Все эти операции контейнера ***list*** *устойчивы*, то есть сохраняют относительный порядок элементов, имеющих одинаковые значения.

Пример с фруктами из § 16.3.6 работает с контейнером ***fruit***, определенном как список. В добавок мы можем извлекать элементы из одного списка и вставлять в другой одной операцией удаления-вставки (splice). Имея списки

```
fruit:
    apple      pear

citrus:
    orange     grapefruit     lemon
```

мы можем удалить ***orange*** из ***citrus*** и вставить его во ***fruit*** следующим образом:

```
list<string>::iterator p = find_if (fruit.begin (), fruit.end (), initial ('p'));
fruit.splice (p, citrus.begin ());
```

Это приведет к удалению первого элемента из ***citrus*** (***citrus.begin*** ()) и помещению его прямо перед первым элементом списка ***fruit*** на букву *'p'*; тем самым получится:

```
fruit:
    apple      orange      pear
citrus:
    grapefruit     lemon
```

Отметим, что функция ***splice*** () не копирует элементы, как ***insert*** () (§ 16.3.6). Она просто изменяет структуры данных списка, относящиеся к элементу.

Кроме удаления-вставки отдельных элементов и диапазонов, при помощи функции ***splice*** () мы можем удалить-вставить все элементы списка:

```
fruit.splice (fruit.begin (), citrus);
```

Это приведет к следующему результату:

```
fruit:
    grapefruit    lemon    apple    orange    pear

citrus:
    <empty> (нусто)
```

Во всех версиях функция ***splice*** () получает в качестве второго аргумента список, из которого берутся элементы. Это позволяет удалять элементы из их первоначального списка. Просто итератор не позволил бы этого, поскольку не существует универсального способа определить контейнер, содержащий элемент, по одному лишь итератору на этот элемент (§ 18.6).

Естественно, аргумент-итератор должен быть допустимым итератором для списка, на элементы которого он (предположительно) указывает. То есть он должен указывать на элемент того списка или быть его концом (***end*** ()) . В противном случае результат не определен и может привести к неприятностям. Например:

```
list<string>::iterator p = find_if (fruit.begin (), fruit.end (), initial ('p'));
fruit.splice (p, citrus, citrus.begin ());      // правильно
fruit.splice (p, citrus, fruit.begin ());       // ошибка: fruit.begin ( ) не указывает
                                                // внутрь списка citrus
citrus.splice (p, fruit, fruit.begin ());       // ошибка: p не указывает
                                                // внутрь списка citrus
```

С первой функцией ***splice*** () все правильно, даже если список ***citrus*** пуст.

Функция ***merge*** () (слияние) объединяет два сортированных списка, удаляя элементы из одного списка и вставляя в другой с сохранением порядка. Например:

```
f1:
    apple    quince    pear

f2:
    lemon    grapefruit    orange    lime
```

можно отсортировать и объединить таким образом:

```
f1.sort ();
f2.sort ();
f1.merge (f2);
```

В результате получится:

```
f1:
    apple    grapefruit    lemon    lime    orange    pear    quince

f2:
    <empty> (нусто)
```

Если один из объединяемых списков не отсортирован, ***merge*** () все же выдаст список, содержащий объединение элементов этих двух списков. Однако относительно порядка в получившемся списке нет никаких гарантий.

Как и ***splice*** (), ***merge*** () воздерживается от копирования элементов. Вместо этого она удаляет элементы из списка-источника и вставляет их в целевой список. После ***x.merge*** (*y*) список *y* окажется пустым.

17.2.2.2. Операции с начальными элементами

Операции с первыми элементами списков созданы в дополнение к операциям, которые имеют дело с последними элементами, обеспечиваемым всеми последовательностями (§ 16.3.5):

```
template<class T, class A = allocator<T> > class list {
public:
        // ...
        // доступ к элементам:

        reference front ();                 // ссылка на первый элемент
        const_reference front () const;

        void push_front (const T&);         // добавление нового первого элемента
        void pop_front ();                  // удаление первого элемента

        // ...
};
```

Первый элемент контейнера называют ***front***. Для списков операции с началом так же эффективны и удобны, как и операции с концом (§ 16.3.6). Когда есть выбор, лучше предпочесть операции с конечными элементами, а не с начальными. Программы, написанные с использованием последних элементов, применимы как для списков, так и для векторов. Поэтому если существует возможность, что программа, использующая списки, когда-нибудь разовьется в обобщенный алгоритм, применимый к множеству контейнеров, лучше всего выбирать более широко распространенные операции с концами. Это частный случай того правила, что, решая задачу, для достижения максимальной гибкости разумно использовать минимальный набор операций.

17.2.2.3. Другие операции

Вставка и удаление элементов особенно эффективны для списков. Когда такие операции встречаются часто, это, конечно, стимулирует людей предпочитать списки. Что, в свою очередь, делает осмысленным поддержку обычных способов удаления элементов напрямую:

```
template<class T, class A = allocator<T> > class list {
public:
        // ...

        void remove (const T& val);
        template<class Pred> void remove_if (Pred p);

        void unique ();     // удаляет дубликаты при помощи ==
        template<class BinPred> void unique (BinPred b);     // удаляет дубликаты
                                                             // при помощи b

        void reverse ();    // обратный порядок элементов
};
```

Например, получив

```
fruit:
    apple    orange    grapefruit    lemon    orange    lime    pear    quince
```

мы можем удалить все элементы со значением *"orange"* таким образом:

```
fruit.remove ("orange");
```

В результате получим:

```
fruit:
    apple    grapefruit    lemon    lime    pear    quince
```

Часто интереснее удалить все элементы, отвечающие какому-либо критерию, а не просто все элементы с данным значением. Это выполняет операция ***remove_if*** (). Например:

```
fruit.remove_if (initial ('l'));
```

удалит из ***fruit*** все элементы на букву *'l'*, оставив:

```
fruit:
    apple    grapefruit    pear    quince
```

Обычная причина удаления элементов — устранение дубликатов. Это обеспечивает операция ***unique*** (). Например, если бы ***fruit*** содержал такой список:

```
apple    pear    apple    apple    pear
```

простое ***fruit.unique*** () привело бы к результату:

```
apple    pear    apple    pear
```

в то время как предварительная сортировка (а потом вызов ***unique*** ()) даст:

```
apple    pear
```

Если нужно удалить только определенные дубликаты, мы можем ввести предикат (логическое условие), чтобы указать, какие именно дубликаты мы хотим удалить. Например, мы могли бы определить бинарный предикат (§ 18.4.2) ***initial2*** (***x***), чтобы выявлять строки, начинающиеся на букву ***x***, и возвращающий ***false*** для каждой строки, начинающейся на какую-либо другую букву.

В списке

```
pear    pear    apple    apple
```

мы можем удалить последовательные дубликаты каждого фрукта на букву *'p'* следующим вызовом:

```
fruit.unique (initial2 ('p'));
```

Это даст нам:

```
pear    apple    apple
```

Как отмечено в § 16.3.2, иногда мы хотим просмотреть контейнер в обратном порядке. Для списков можно изменить порядок элементов так, чтобы первый стал последним и т. д. без копирования элементов. Это обеспечивается операцией ***reverse*** (). Из списка

```
fruit:
    banana    cherry    lime    strawberry
```

вызов ***fruit.reverse*** () сделает:

```
fruit:
    strawberry    lime    cherry    banana
```

Удаляемый из списка элемент уничтожается. Однако заметим, что уничтожение указателя не влечет за собой удаления (***delete***) элемента, на который он указывает. Если вам нужен контейнер указателей, который при удалении из него указателя (или уничтожения всего контейнера) уничтожает элемент, на который указывает этот указатель, вы должны написать его самостоятельно (§ 17.8[13]).

17.2.3. Очереди с двумя концами

Контейнер ***deque*** (читается «дек») — это очередь с двумя концами. То есть это последовательность, оптимизированная таким образом, что операции с обоими концами почти так же эффективны, как в списке, в то время как обращение по индексу приближается по своей эффективности к вектору:

```
template<class T, class A = allocator<T> > class std::deque {
    // типы и операции, такие же как в vector (§ 16.3.3, § 16.3.5, § 16.3.6),
    // кроме capacity() и reserve() плюс операции с началами (§ 17.2.2.2) как в list
};
```

Вставка и удаление элементов «в середине» здесь так же (не)эффективны, как в векторе, то есть вставка «в середине» ведет себя не так, как в списке. Следовательно, контейнер ***deque*** используют там, где добавление и удаление элементов производится «с краев». Например, мы могли бы воспользоваться очередями с двумя концами для моделирования участка железной дороги или колоды игральных карт:

```
deque<car> siding_no_3;
deque<Card> bonus;
```

17.3. Адаптеры последовательностей

Контейнеры ***vector***, ***list*** и ***deque*** не построить один из другого без потери эффективности. С другой стороны, при помощи этих трех базовых последовательностей можно изящно и эффективно реализовать стеки (***stack***) и очереди (***queue***). Поэтому классы ***stack*** и ***queue*** определены не как отдельные контейнеры, а как адаптеры базовых контейнеров.

Адаптер контейнера предоставляет ограниченный интерфейс к контейнеру. В частности, адаптеры не предоставляют итераторов; предполагается, что они (адаптеры) будут использоваться только через их специализированные интерфейсы.

Методы по созданию из контейнеров контейнерных адаптеров часто полезны для неинтрузивного приспособления интерфейса класса к потребностям пользователя.

17.3.1. Стек

Адаптер ***stack*** определен в заголовочном файле <***stack***>. Он так прост, что лучший способ описать его — это представить реализацию:

```
template<class T, class C = deque<T> > class std::stack {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
```

```
        typedef typename C::size_type size_type;
        typedef C container_type;

        explicit stack (const C& a = C ()) : c (a) { }

        bool empty () const { return c.empty (); }
        size_type size () const { return c.size (); }

        value_type& top () { return c.back (); }
        const value_type& top () const { return c.back (); }

        void push (const value_type& x) { c.push_back (x); }
        void pop () { c.pop_back (); }
};
```

То есть ***stack*** — это просто интерфейс для контейнера, тип которого передается как аргумент шаблона. Все, что ***stack*** делает — это удаляет из интерфейса нестековые операции для своего контейнера и дает операциям ***back*** (), ***push_back*** () и ***pop_back*** () общепринятые имена — соответственно ***top*** (), ***push*** () и ***pop*** ().

По умолчанию стек хранит свои элементы в очереди с двумя концами, но можно использовать любую другую последовательность, обеспечивающую операции ***back*** (), ***push_back*** () и ***pop_back*** (). Например:

```
stack<char> s1;                 // использует deque<char> для хранения
                                // элементов типа char
stack<int, vector<int> > s2;    // использует vector<int> для хранения
                                // элементов типа int
```

Чтобы инициализировать стек, можно предоставить существующий контейнер. Например:

```
void print_backward (vector<int>& v)
{
        stack< int, vector<int> >state (v);    // инициализируем состояние из v
        while (state.size ()) {
                cout << state.top ();
                state.pop ();
        }
}
```

Однако элементы аргумента-контейнера копируются, поэтому указание существующего контейнера в качестве аргумента может оказаться дорогостоящим.

Элементы добавляются в стек, при помощи ***push_back*** () контейнера, используемого для хранения элементов. Следовательно стек не может переполниться, пока в машине есть достаточно памяти для удовлетворения требований этого контейнера (с использованием его распределителя памяти; см. § 19.4).

С другой стороны, стек может переполниться снизу:

```
void f ()
{
        stack<int, vector<int>> s;
        s.push (2);
        if (s.empty ()) {     // можно защититься от переполнения снизу
                // не снимать со стека (pop)
        }
```

```
    else {              // но оно вполне возможно
        s.pop ();       // прекрасно: s.size() становится равным 0
        s.pop ();       // эффект не определен, возможны неприятности
    }
}
```

Отметим, что для использования элемента не нужно применять ***pop*** () (снимать элемент с вершины стека). Вместо этого обращаются к ***top*** () (получить элемент с вершины стека). ***pop*** () применяют, когда элемент больше не нужен. Это не вызывает слишком больших затруднений и более эффективно, когда удаление элемента с вершины стека не нужно:

```
void f(stack<char>& s)
{
    if (s.top () == 'c') s.pop ();      // удаляем возможные начальные 'c'
    //...
}
```

В отличие от полностью разработанных контейнеров, ***stack*** (как и другие контейнерные адаптеры) не имеет среди параметров шаблона распределителя памяти. Вместо этого ***stack*** и его пользователи обращаются к распределителю памяти того контейнера, который используется при реализации ***stack***.

17.3.2. Очереди

Определенный в заголовочном файле <***queue***> контейнер ***queue*** — это интерфейс для контейнера, позволяющего вставлять элементы в конец (***back*** ()) и извлекать элементы из начала (***front*** ()):

```
template<class T, class C = deque<T> > class std::queue {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit queue (const C& a = C ()) : c (a) { }

    bool empty () const { return c.empty (); }
    size_type size () const { return c.size (); }

    value_type& front () { return c.front (); }
    const value_type& front () const { return c.front (); }

    value_type& back () { return c.back (); }
    const value_type& back () const { return c.back (); }

    void push (const value_type& x) { c.push_back (x); }
    void pop () { c.pop_front (); }
};
```

По умолчанию очередь ***queue*** для хранения своих элементов использует ***deque***, но для этой цели может применяться и любая другая последовательность, предоставляющая операции ***front*** (), ***back*** (), ***push_back*** () и ***pop_front*** (). Поскольку вектор не предо-

ставляет операцию ***pop_front*** (), он не может использоваться в качестве базового контейнера для очередей.

Очереди, кажется, встречаются во всех системах. Для простой системы, основанной на передаче сообщений, можно определить сервер (программу, обрабатывающую сообщения) например так:

```
struct Message {        // сообщение
    //...
};

void server (queue<Message>& q)
{
    while (!q.empty ()) {
        Message& m = q.front ();        // получили сообщение
        m.service ();                   // вызвали функцию обслуживания запроса
        q.pop ();                       // уничтожили сообщение
    }
}
```

Сообщения помещаются в очередь при помощи ***push*** ().

Если запрашивающая программа и сервер выполняются в разных процессах или потоках, для доступа к очереди необходима какая-то синхронизация. Например:

```
void server2 (queue<Message>& q, Lock& lck)
{
    while (!q.empty ()) {
        Message m;
        {   LockPtr h (lck);            // блокируем только на время
                                        // извлечения сообщения (§ 14.4.1)
            if (q.empty ()) return;     // сообщение получил кто-то другой
            m = q.front ();
            q.pop ();
        }
        m.service ();                   // вызвали функцию обслуживания запроса
    }
}
```

В C++ да и вообще в мире нет стандартного определения параллельности или блокировки. Посмотрите, что может предложить ваша система, и как это реализовать на C++ (§ 17.8[8]).

17.3.3. Очереди с приоритетом

Очередь с приоритетом (***priopity_queue***) — это такая очередь, где каждому элементу назначен приоритет, определяющий порядок, в котором элементы достигают выхода из очереди (***top*** ()):

```
template <class T, class C = vector<T>,
    class Cmp = less<typename C::value_type> >
class std::priority_queue {
protected:
    C c;
    Cmp cmp;
```

```
public:
        typedef typename C::value_type value_type;
        typedef typename C::size_type size_type;
        typedef C container_type;

        explicit priority_queue (const Cmp& a1 = Cmp (), const C& a2 = C ())
                : c (a2), cmp (a1) {make_heap (c.begin (), c.end (), cmp); }          // см. § 18.8
        template<class In>
        priority_queue (In first, In last, const Cmp& = Cmp (), const C& = C ());

        bool empty () const { return c.empty (); }
        size_type size () const { return c.size (); }

        const value_type& top () const { return c.front (); }

        void push (const value_type&);
        void pop ();
};
```

Объявление ***priority_queue*** находится в заголовочном файле <***queue***>.

По умолчанию ***priority_value*** просто сравнивает элементы при помощи оператора <, и ***top*** () возвращает наибольший элемент:

```
struct Message {
        int priority;
        bool operator< (const Message& x) const { return priority < x.priority; }
        // ...
};

void server (priority_queue<Message>& q, Lock& lck)
{
        while (!q.empty ()) {
                Message m;
                { LockPtr h (lck);              // блокируем только на время
                                                // извлечения сообщения (§ 14.4.1)
                        if (q.empty ()) return; // сообщение получил кто-то другой
                        m = q.top ();
                        q.pop ();
                }
                m.service ();                   // вызвали функцию обслуживания запроса
        }
}
```

Этот пример отличается от примера с очередью (§ 17.3.2) тем, что сообщения с более высоким приоритетом будут обслуживаться первыми. Порядок, в котором элементы с равными приоритетами достигнут выхода из очереди, не определен. Считается, что два элемента имеют равный приоритет, если ни у одного из них приоритет не выше, чем у другого (§ 17.4.1.5).

Альтернативу оператору < для сравнения элементов можно указать в аргументе шаблона. Например, мы могли бы отсортировать строки без учета регистра, поместив их в

```
priority_queue<string, vector<string>, Nocase> pq;      // используем Nocase для
                                                        // сравнения (§ 17.4.1)
```

при помощи ***pq.push*** (), а затем получив обратно при помощи ***pq.top*** () и ***pq.pop*** ().

Объекты, определенные шаблонами с разными аргументами, относятся к разным типам (§ 13.6.3.1). Например:

```
priority_queue<string>& pq1 = pq;          // ошибка: типы не совпадают
```

Мы можем создать критерий сравнения, не затрагивая типа ***priopity_queue***, задав объект сравнения подходящего типа в качестве аргумента конструктора. Например:

```
struct String_cmp {          // тип, используемый для выражения критерия
                             // сравнения во время выполнения
    String_cmp (int n = 0); // использовать критерий сравнения n
    //...
};

typedef priority_queue<string, vector<string>, String_cmp > Pqueue;

void g (Pqueue& pq)          // pq использует String_cmp() для сравнений
{
    Pqueue pq2 (String_cmp (nocase));
    pq = pq2;                // правильно: pq и pq2 одного и того же типа
                             // pq теперь также использует String_cmp(nocase)
}
```

Поддержание порядка элементов не бесплатно, но и не так уж дорого. Один полезный способ реализации ***priority_queue*** заключается в использовании дерева, чтобы отслеживать относительное положение элементов. Это приводит к тому, что обе операции ***push*** () и ***pop*** () обходятся как ***O*** (***log*** (***n***)).

По умолчанию очередь с приоритетом использует для хранения своих элементов вектор, но в этом качестве может использоваться и любая другая последовательность, предоставляющая операции ***front*** (), ***push_back*** (), ***pop_back*** () и итераторы с произвольным доступом. Очередь с приоритетом скорее всего реализуется при помощи кучи (***heap***) (§ 18.8).

17.4. Ассоциативные контейнеры

Ассоциативный массив — это один из самых полезных и универсальных типов, определяемых пользователем. Фактически, в языках, занимающихся главным образом обработкой текстов и символов, это зачастую встроенный тип. Ассоциативный массив, часто называемый *отображением* (map), а иногда *словарем* (dictionary), содержит пары значений. Зная одно значение, называемое *ключом* (key), мы можем получить доступ к другому, называемому *отображенным значением* (mapped value). Ассоциативный массив можно представить как массив, для которого индекс не обязательно должен иметь целочисленный тип:

```
template<class K, class V> class Assoc {
public:
    V& operator[] (const K&);    // возвращает ссылку на V, соответствующий K
    //...
}
```

Таким образом, ключ типа ***K*** «именует» отображенное значение ***V***.

Ассоциативные контейнеры — это обобщение понятия ассоциативного массива. ***map*** — это традиционный ассоциативный массив, где по значению ключа можно

однозначно найти значение. Контейнер ***multimap*** — это ассоциативный массив, позволяющий дублировать элементы для данного ключа, а ***set*** (множество) и ***multiset*** можно рассматривать как вырожденные ассоциативные массивы, в которых ключу не соответствует никакого значения.

17.4.1. Ассоциативные массивы

Ассоциативный массив ***map*** — это последовательность пар (ключ, значение), которая обеспечивает быстрое получение значения по ключу. Каждому ключу соответствует максимум одно значение. Иными словами, каждый ключ в ассоциативном массиве уникален. Контейнер ***map*** предоставляет двунаправленные итераторы (§ 19.2.1).

Ассоциативный массив требует, чтобы для типов ключа (§ 17.4.1) существовала операция «меньше» (§ 17.1.4.1); он хранит свои элементы отсортированными (по ключу) так, что перебор происходит по порядку. Для элементов, не имеющих очевидного понятия порядка, или в том случае, когда нет необходимости держать контейнер отсортированным, мы можем рассмотреть использование ***hash_map*** (§ 17.6).

17.4.1.1. Типы

Ассоциативный массив имеет обычные типы членов контейнера (§ 16.3.1) и несколько, отражающих его специфику:

```
template<class Key, class T, class Cmp = less<Key>, class A = allocator<pair <const Key, T> > >
class std::map {
public:
    // типы:

    typedef Key key_type;
    typedef T mapped_type;

    typedef pair<const Key, T> value_type;

    typedef Cmp key_compare;
    typedef A allocator_type;

    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;

    typedef implementation_defined1 iterator;
    typedef implementation_defined2 const_iterator;

    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    // ...
};
```

Отметим, что ***value_type*** контейнера ***map*** — это ***pair***, пара (ключ,значение). Тип отображенных значений обозначен как ***mapped_type***. Таким образом, ***map*** — это последовательность пар ***pair<const Key, mapped_type>***.

Как обычно, фактические типы итераторов определяются при реализации (implementation-defined). Поскольку ***map***, скорее всего, реализована с использованием какой-нибудь формы дерева, эти итераторы обычно обеспечивают некоторый способ прохода по дереву.

Реверсивные итераторы конструируются из стандартных шаблонов ***reverse_iterator*** (§ 19.2.5).

17.4.1.2. Итераторы и пары

Контейнер ***map*** обеспечивает обычный набор функций, возвращающих итераторы (§ 16.3.2):

```
template<class Key, class T, class Cmp = less<Key>, class A = allocator<pair <const Key, T> > >
        class map {
public:
        //...
        // итераторы:

        iterator begin ();
        const_iterator begin () const;

        iterator end ();
        const_iterator end () const;

        reverse_iterator rbegin ();
        const_reverse_iterator rbegin () const;

        reverse_iterator rend ();
        const_reverse_iterator rend () const;

        //...
};
```

Итерация по ***map*** — это просто итерация по последовательности пар ***pair<const Key, mapped_type>***. Например, мы могли бы распечатать записи в телефонной книге:

```
void f(map<string, number>& phone_book)
{
        typedef map<string, number>::const_iterator CI;
        for (CI p = phone_book.begin (); p!=phone_book.end (); ++p)
                cout << p->first << '\t' << p->second << '\n';
}
```

Итератор для ассоциативного массива предоставляет элементы в порядке возрастания их ключей (§ 17.4.1.5), поэтому пункты телефонной книги ***phone_book*** будут выведены в лексикографическом порядке.

Мы обращаемся к первому элементу любой пары ***pair*** как к ***first***, а ко второму как к ***second*** независимо от того, какого типа они в действительности:

```
template<class T1, class T2> struct std::pair {
        typedef T1 first_type;
        typedef T2 second_type;

        T1 first;
        T2 second;
```

```
        pair() :first(T1()), second(T2()) { }
        pair(const T1& x, const T2& y) : first(x), second(y) { }
        template<class U, class V>
            pair(const pair<U, V>& p) :first(p.first), second(p.second) { }
};
```

Последний конструктор существует для того, чтобы позволить преобразования в инициализаторе (§ 13.6.2). Например:

```
pair<int, double> f(char c, int i)
{
    pair<char, int> x(c, i)
    //...
    return x;          // требуется преобразование pair<char, int> в pair<int, double>
}
```

В отображении ключ является первым элементом в паре, а отображенное значение — вторым.

Полезность класса ***pair*** не ограничивается реализацией ***map***; он является полноправным классом стандартной библиотеки. Определение ***pair*** находится в заголовочном файле <***utility***>. Также обеспечена функция, позволяющая удобным образом создавать ***pair***:

```
template<class T1, class T2> pair<T1, T2> std::make_pair(T1 t1, T2 t2)
{
    return pair<T1, T2>(t1, t2);
}
```

Пара ***pair*** по умолчанию инициализируется значениями по умолчанию для типов ее элементов. В частности, это приводит к тому, что элементы встроенных типов инициализируются нулями (§ 5.1.1), а строки ***string*** — пустыми строками (§ 20.3.4). Тип, не имеющий конструктора по умолчанию, может быть элементом пары ***pair*** только при условии, что она явно инициализирована.

17.4.1.3. Индексация

Типичная операция с ассоциативным массивом — это ассоциативный поиск при помощи оператора индексации ([]):

```
template<class Key, class T, class Cmp = less<Key>, class A = allocator<pair<const Key, T> > >
class map {
public:
    //...

    mapped_type& operator[] (const key_type& k);        // доступ к элементу с ключом k
    //...
};
```

Оператор индексации выполняет поиск по ключу, заданному в качестве индекса, и возвращает соответствующее значение. Если ключ не найден, в ассоциативный массив вставляется элемент с этим ключом и значением по умолчанию типа ***mapped_type***. Например:

```
void f()
{
```

```
    map<string, int> m;          // сначала ассоциативный массив пуст
    int x = m["Генри"];          // создаем новую запись для "Генри",
                                 // инициализируем его нулем и возвращаем 0
    m["Гарри"] = 7;              // создаем новую запись для "Гарри",
                                 // инициализируем его нулем и присваиваем 7
    int y = m["Генри"];          // возвращаем значение из записи "Генри"
    m["Гарри"] = 9;              // заменяем значение записи "Гарри" на 9
}
```

Как чуть более реалистичный пример, рассмотрим программу, подсчитывающую общую стоимость предметов, представленных в форме пар (название предмета, стоимость), например:

```
nail 100 hammer 2 saw 3 saw 4 hammer 7 nail 1000 nail 250
```

и также подсчитывающую сумму по каждому предмету. Основная работа может быть выполнена при считывании пар в ассоциативный массив:

```
void readitems (map<string, int>& m)
{
    string word;
    int val = 0;
    while (cin >> word >> val) m[word] += val;
}
```

Операция индексации **m[word]** определяет соответствующую пару (**string, int**) и возвращает ссылку на ее часть **int**. Эта программа пользуется тем, что в новом элементе по умолчанию значение **int** устанавливается в **0**.

Ассоциативный массив, сконструированный функцией **readitem** (), затем может быть выведен при помощи обычного цикла:

```
int main ()
{
    map<string, int> tbl;
    readitems (tbl);

    int total = 0;
    typedef map<string, int>::const_iterator CI;

    for (CI p = tbl.begin (); p!=tbl.end (); ++p) {
        total += p->second;
        cout << p->first << '\t' << p->second << '\n'
    }

    cout << "----------------\ntotal\t" << total << '\n';

    return !cin;
}
```

При вышеуказанном входе мы получим на выходе:

```
hammer    9
nail      1350
saw       7
----------------
total     1366
```

Отметим, что предметы печатаются в лексикографическом порядке (§ 17.4.1, § 17.4.1.5).

Операция индексации должна найти в ассоциативном массиве ключ. Это, конечно, не так дешево, как индексация массива целым числом. Цена равна ***O*** (***log*** (***size_of_map***)), что для многих прикладных программ приемлемо. Для тех, кому это не годится, выходом может послужить хэшируемый контейнер (§ 17.6).

Когда ключ не находится, операция индексации добавляет элемент по умолчанию. Поэтому для константных ассоциативных массивов не существует версии ***operator***[] (). Более того, индексация может использоваться, только если ***mapped_type*** (тип отображенного значения) имеет значение по умолчанию. Если программист просто хочет посмотреть, существует ли такой ключ, для поиска ключа ***key*** без изменения ассоциативного массива можно воспользоваться операцией ***find*** () (§ 17.4.1.6).

17.4.1.4. Конструкторы

Класс ***map*** обеспечивает обычный комплект конструкторов и т. п. (§ 16.3.4):

```
template<class Key, class T, class Cmp = less<Key>, class A = allocator<pair <const Key, T> > >
class map {
public:
        //...
        // сконструировать/скопировать/уничтожить:

        explicit map (const Cmp& c = Cmp (), const A& = A ());
        template<class In> map (In first, In last,
                const Cmp& c= Cmp (), const A& = A ());
        map (const map&);

        ~map ();

        map& operator= (const map&);

        //...
};
```

Копирование контейнера требует выделения памяти для элементов и создания копии каждого элемента (§ 16.3.4). Это может оказаться очень дорого, и делать это нужно лишь при необходимости. Поэтому контейнеры, такие как ассоциативные массивы, часто передают по ссылке.

Конструктор члена шаблона принимает последовательность пар ***pair<const Key, T>***, описанных парой итераторов для чтения. Функцией ***insert*** () (§ 17.4.1.7) конструктор вставляет элементы из последовательности в ассоциативный массив.

17.4.1.5. Сравнения

Чтобы найти в ассоциативном массиве элемент по данному ключу, операции массива должны сравнивать ключи. К тому же итераторы перемещаются по ассоциативному массиву в порядке возрастания значений ключа, и при вставке, как правило, ключи тоже будут сравниваться (чтобы поместить элемент в структуру дерева, представляющего ассоциативный массив).

По умолчанию используемое для ключей сравнение — это < («меньше»), но можно обеспечить альтернативу в виде параметра шаблона или аргумента конструктора (см. § 17.3.3). Данное сравнение — это сравнение ключей, но ***value_type*** контейнера ***map*** —

это пара (ключ,значение). Поэтому функция ***value_comp*** () определена так, чтобы сравнивать пары, используя функцию сравнения ключей:

```
template<class Key, class T, class Cmp = less<Key>, class A = allocator<pair <const Key, T> > >
class map {
public:
        //...

        typedef Cmp key_compare;
        class value_compare : public binary_function<value_type, value_type, bool> {
        friend class map;
        protected:
                Cmp cmp;
                value_compare (Cmp c) : cmp (c) { }
        public:
                bool operator () (const value_type& x, const value_type& y) const
                        { return cmp (x.first, y.first); }
        };

        key_compare key_comp () const;
        value_compare value_comp () const;

        //...
};
```

Например:

```
map<string, int> m1;
map<string, int, Nocase> m2;                    // определяет тип сравнения (§ 17.1.4.1)
map<string, int, String_cmp> m3;                // определяет тип сравнения (§ 17.1.4.1)
map<string, int, String_cmp> m4 (String_cmp (literary));    // передается сравнение
```

Функции-члены ***key_comp*** () и ***value_comp*** () дают возможность запросить ассоциативный массив о способе сравнения ключей и значений. Обычно это делается, чтобы обеспечить тот же критерий сравнения для некоторых других контейнеров или алгоритмов. Например:

```
void f (map<string, int>& m)
{
        map<string, int> m2;                    // по умолчанию сравнивается при помощи <
        map<string, int> m3 (m.key_comp ());    // сравнивается так, как это делает m
        //...
}
```

Как определить особое сравнение, см. в § 17.1.4.1; описание объектов-функций приведено в § 18.4.

17.4.1.6. Операции с ассоциативными массивами

Главная идея ассоциативных массивов, а в действительности и всех ассоциативных контейнеров, — получать информацию по ключу. Для этого введено несколько специализированных операций:

```
template<class Key, class T, class Cmp = less<Key>, class A = allocator<pair <const Key, T> > >
class map {
public:
```

```
        //...
        // операции с ассоциативными массивами
        // находит элемент с ключом k
        iterator find (const key_type& k);
        const_iterator find (const key_type& k) const;
        // находит число элементов с ключом k
        size_type count (const key_type& k) const;
        // находит первый элемент с ключом k
        iterator lower_bound (const key_type& k);
        const_iterator lower_bound (const key_type& k) const;
        // находит первый элемент с ключом больше чем k
        iterator upper_bound (const key_type& k);
        const_iterator upper_bound (const key_type& k) const;
        pair<iterator, iterator> equal_range (const key_type&);
        pair<const_iterator, const_iterator> equal_range (const key_type&) const;
        //...
};
```

Операция ***m.find (k)*** просто выдает итератор, соответствующий элементу с ключом ***k***. Если такого элемента не существует, возвращенным итератором будет ***m.end ()***. Для контейнера с уникальными ключами, такого как ***map*** или ***set***, результирующий итератор будет указывать на единственный элемент с ключом ***k***. Для контейнера с неуникальными ключами, такого как ***multimap*** или ***multiset***, результирующий итератор будет указывать на первый элемент с таким ключом. Например:

```
void f (map<string, int>& m)
{
        map<string, int>::iterator p = m.find ("Золото");
        if (p!=m.end ()) {                              // если "Золото" найдено
                //...
        }
        else if (m.find ("Серебро")!=m.end ()) {        // ищем "Серебро"
                //...
        }
        //...
}
```

Для ***multimap*** (§ 17.4.2) нахождение первого вхождения редко так полезно, как нахождение всех вхождений; функции ***m.lower_bound (k)*** и ***m.upper_bound (k)*** дают начало и конец подпоследовательности элементов контейнера ***m***, имеющих ключ ***k***. Обычно конец последовательности — это итератор, указывающий на элемент, следующий за последним в последовательности. Например:

```
void f (multimap<string, int>& m)
{
        multimap<string, int>::iterator lb = m.lower_bound ("Золото");
        multimap<string, int>::iterator ub = m.upper_bound ("Золото");
        for (multimap<string, int>::iterator p = lb; p!=ub; ++p) {
                //...
        }
}
```

Нахождение верхней и нижней границ двумя отдельными операциями не изящно и не эффективно. Операция ***equal_range*** () предназначена для предоставления обоих. Например:

```
void f(multimap<string, int>& m)
{
    typedef multimap<string, int>::iterator MI;
    pair<MI, MI> g = m.equal_range ("Золото");
    for (MI p = g.first; p!=g.second; ++p) {
        //...
    }
}
```

Если ***lower_bound*** (***k***) не находит ***k***, она возвращает итератор на первый элемент с ключом больше ***k*** или ***end*** (), если такого элемента не существует. Этот способ сообщения о неудаче также используется функциями ***upper_bound*** () и ***egual_range*** ().

17.4.1.7. Операции со списками

Вводить значения в ассоциативный массив принято простым присваиванием с использованием индексации. Например:

```
phone_book["Отдел заказов"] = 8226339;
```

Это гарантирует, что об отделе заказов будет занесена соответствующая запись, независимо от того, было ли в телефонной книге что-либо об отделе заказов до того. Также можно вставить элемент функцией ***insert*** () и удалить функцией ***erase*** ():

```
template<class Key, class T, class Cmp = less<Key>,
    class A = allocator<pair <const Key, T> > >
    class map {
public:
    //...
    // операции со списками:

    // вставляет пару (ключ, значение)
    pair<iterator, bool> insert (const value_type& val);
    // pos — это только подсказка, с какого места искать
    iterator insert (iterator pos, const value_type& val);
    // вставляет элементы из последовательности
    template<class In> void insert (In first, In last);

    // удаляет элемент, на который указывает итератор
    void erase (iterator pos);
    // удаляет элемент с ключом k (если такой элемент существует)
    size_type erase (const key_type& k);
    // удаляет диапазон
    void erase (iterator first, iterator last);
    void clear ();

    //...
};
```

Операция ***m.insert*** (***val***) пытается добавить в ***m*** пару ***val*** типа (***Key***, ***T***). Поскольку ассоциативные массивы — это контейнеры с уникальными ключами, вставка производится, только если в ***m*** не существует элемента с таким ключом. Возвращаемое

значение *m.insert (val)* — это пара *pair<iterator,bool>*. Переменная типа *bool* принимает значение *true*, если *val* действительно вставилось. Итератор указывает на тот элемент контейнера *m*, который имеет ключ *val.first*. Например:

```
void f(map<string, int>& m)
{
    pair<string, int> p99 ("Павел", 99);
    pair<map<string, int>::iterator, bool> p = m.insert (p99);
    if (p.second) {
        // "Павел" был вставлен
    }
    else {
        // "Павел" уже был там
    }
    map<string, int>::iterator i = p.first;      // указывает на m["Павел"]
    // ...
}
```

Обычно нас не заботит, вставлен ключ заново или уже присутствовал в ассоциативном массиве до выполнения *insert ()*. Когда это нас все же интересует, то лишь потому, что мы хотим зарегистрировать тот факт, что значение оказалось в массиве откуда-то извне. Две другие версии *insert ()* не возвращают признака, действительно ли значение вставилось.

Указание позиции *pos* в *insert (pos, val)* — это просто подсказка реализации, откуда начать поиск ключа *val.first*. Если подсказка хороша, то можно значительно выиграть в производительности. Если нет, вам лучше обойтись без нее, с точки зрения как эффективности, neat и краткости в записи. Например:

```
void f(map<string, int>& m)
{
    m["Дмитрий"] = 3;                          // хотя, возможно, менее эффективно
    m.insert (m.begin (), make_pair (const string ("Давид"), 99));      // плохо
}
```

Фактически, оператор [] — это больше, чем просто более удобное обозначение *insert ()*. Результат *m[k]* эквивалентен (*(m.insert (make_pair (k,V ())).first)).second*, где *V ()* — значение по умолчанию для отображенного типа. Поняв эту эквивалентность, вы, вероятно, поймете суть ассоциативных контейнеров.

Поскольку оператор [] всегда пользуется *V ()*, нельзя использовать индексацию с типом значения, не имеющим значения по умолчанию. Это печальное ограничение для стандартных ассоциативных контейнеров. Однако требование иметь значение по умолчанию не является фундаментальным свойством ассоциативных контейнеров вообще (см. § 17.6.2).

Элементы с заданным ключом можно удалять. Например:

```
void f(map<string, int>& m)
{
    int count = phone_book.erase ("Роман");
    // ...
}
```

Возвращенное значение типа *int* — это число удаленных элементов. В частности, если нет ни одного элемента с ключом *"Роман"*, *count* становится равным *0*. Для контейне-

ров *multimap* и *multiset* это значение может оказаться больше *1*. В качестве альтернативы можно удалить элемент, на который указывает итератор, или удалить целый диапазон элементов, зная последовательность. Например:

```
void g(map<string, int>& m)
{
    m.erase(m.find("Екатерина"));
    m.erase(m.find("Алиса"), m.find("Владимир"));
}
```

Естественно, быстрее удалить элемент, для которого уже имеется итератор, чем сначала найти элемент по ключу, и лишь потом удалить. После операции *erase* () этот итератор не может использоваться снова, поскольку элемента, на который он указывает, больше не существует. Вызов *m.erase* (*b*,*e*), где *e* есть *m.end* (), безопасен (если *b* ссылается на элемент *m* или на *m.end* ()). С другой стороны, вызов *m.erase* (*p*), где *p* есть *m.end* (), является серьезной ошибкой, способной погубить контейнер.

17.4.1.8. Другие функции

Наконец, контейнер *map* обеспечивает простые функции, выдающие число элементов, и предоставляющие специализированную форму «перемены мест» *swap* ():

```
template<class Key, class T, class Cmp = less<Key>, class A = allocator<pair<const Key, T> > >
class map {
public:
    // ...
    // емкость:

    size_type size() const;              // число элементов
    size_type max_size() const;          // размер максимально возможного map
    bool empty() const { return size() == 0; }

    void swap(map&);
};
```

Возвращаемые функциями *size* () и *max_size* () значения — это число элементов.

Вдобавок, *map* предоставляет операторы ==, !=, <, >, <=, >= и функцию *swap* () как функции-не-члены:

```
template<class Key, class T, class Cmp, class A>
bool operator== (const map<Key, T, Cmp, A>&, const map<Key, T, Cmp, A>&);

// аналогично != , < , > , <= и >=

template<class Key, class T, class Cmp, class A>
void swap(map<Key, T, Cmp, A>&, map<Key, T, Cmp, A>&);
```

Зачем кому-то сравнивать два ассоциативных массива? Когда мы особым образом сравниваем два ассоциативных массива, обычно мы хотим узнать, не только, различаются ли они, но также, чем именно они различаются. В таком случае мы не пользуемся == или !=. Однако, обеспечив для каждого контейнера ==, < и *swap* (), мы сделаем возможным написание алгоритмов, приложимых ко всем контейнерам. Например, эти функции позволяют нам рассортировать функцией *sort* () вектор из ассоциативных массивов и получить множество с элементами типа ассоциативных массивов.

17.4.2. Контейнер multimap

Контейнер ***multimap*** похож на ***map*** (ассоциативный массив), если не считать, что он позволяет дублировать ключи:

```
template<class Key, class T, class Cmp = less<Key>, class A = allocator<pair<const Key, T> > >
class std::multimap {
public:
    // как и map, за исключением:
    iterator insert(const value_type&);      // возвращает итератор, не пару
    // нет оператора индексации []
};
```

Например (используя ***Cstring_less*** из § 17.1.4.1 для сравнения С-строк):

```
void f(map<char*, int, Cstring_less>& m,
    multimap<char*, int, Cstring_less>& mm)
{
m.insert(make_pair("x", 4));
m.insert(make_pair("x", 5));      // эффекта не имеет: уже есть запись для x (§ 17.4.1.7)
// теперь m["x"] == 4

mm.insert(make_pair("x", 4));
mm.insert(make_pair("x", 5));
// теперь mm содержит и ("x", 4) и ("x", 5)
}
```

Это приводит к тому, что ***multimap*** не может поддерживать индексацию по значению ключа, как это делает ***map***. Операции ***equal_range*** (), ***lower_bound*** () и ***upper_bound*** () (§ 17.4.1.6) являются первичными средствами доступа к нескольким значениям с одним и тем же ключом.

Естественно, в тех случаях, когда может существовать несколько значений с одним и тем же ключом, ***multimap*** выглядит предпочтительнее ***map***. Это случается гораздо чаще, чем кажется, когда впервые слышишь о ***multimap***. Кое в чем ***multimap*** даже совершеннее и изящнее ***map***.

Поскольку у человека запросто может быть несколько телефонных номеров, хорошим примером применения ***multimap*** является телефонная книга. Я мог бы распечатать номера моих телефонов так:

```
void print_numbers(const multimap< string, int>& phone_book)
{
    typedef multimap< string, int>:: const_iterator I;
    pair<I, I> b = phone_book.equal_range("Страуструп");
    for (I i = b.first; i != b.second; ++i) cout << (*i).second << '\n';
}
```

В контейнере ***multimap*** аргумент операции ***insert*** () всегда вставляется. Следовательно, ***multimap***::***insert*** () возвращает итератор, а не пару ***pair<iterator,bool>***, как в случае с ***map***. Из соображений единообразия библиотека могла бы обеспечить универсальную форму операции ***insert*** () как для ***map***, так и для ***multimap***, несмотря на то, что ***bool*** оказался бы для ***multimap*** лишним. Другое проектное решение

обеспечило бы простую ***insert*** (), которая не возвращала бы ***bool*** в обоих случаях, и потом предоставляла бы пользователю ***map*** каким-то образом выяснять, не вставлен ли ключ заново. Это тот случай, когда сталкиваются разные идеи по проектированию интерфейса.

17.4.3. Множества

Множества можно рассматривать как ассоциативные массивы (§ 17.4.1), в которых значения не играют роли, так что мы отслеживаем только ключи. Это ведет лишь к небольшим изменениям в пользовательском интерфейсе:

```
template<class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::set {
public:
        // как map, за исключением:
        typedef Key value_type;                 // сам ключ является значением
        typedef Cmp value_compare;
        // никаких операторов индексации []
};
```

Определение ***value_type*** как типа ***key_type*** — это хитрость, позволяющая программе, в которой используются множества и ассоциативные массивы, оставаться идентичной во многих случаях.

Отметим, что множества основываются на операции сравнения (по умолчанию <), а не равенства (==). Это приводит к тому, что равенство элементов определяется неравенством (§ 17.1.4.1), и итерации по контейнеру ***set*** имеют строго определенный порядок.

Как и ***map***, класс ***set*** предоставляет операторы ==, !=, <, >, <=, >= и функцию ***swap*** ().

17.4.4. Контейнер multiset

Контейнер ***multiset*** — это множество, допускающее одинаковые ключи:

```
template<class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::multiset {
public:
        // так же, как set, за исключением:
        iterator insert (const value_type&);    // возвращает итератор, а не пару
};
```

Операции ***equal_range*** (), ***lower_bound*** () и ***upper_bound*** () (§ 17.4.1.6) — основные средства доступа к нескольким вхождениям ключа.

17.5. Почти контейнеры

Встроенные массивы (§ 5.2), строки ***string*** (глава 20), массивы ***valarray*** (§ 22.4) и битовые наборы ***bitset*** (§ 17.5.3) содержат элементы и, следовательно, во многих отношениях могут считаться контейнерами. Однако все они имеют те или иные недостатки в отношении стандартного контейнерного интерфейса, так что эти «почти контейнеры» не совсем взаимозаменяемы с полностью разработанными контейнерами, такими как вектора и списки.

17.5.1. Строки

Класс *basic_string* обеспечивает индексацию, итераторы с произвольным доступом и большинство удобных обозначений для контейнеров (глава 20). Однако *basic_string* не предоставляет такого большого выбора типов элементов. Он оптимизирован для использования в качестве строки символов и, как правило, применяется совсем не так, как контейнеры.

17.5.2. Valarray

Массив *valarray* (§ 22.4) — это вектор, оптимизированный для численных вычислений. Следовательно, *valarray* не претендует на то, чтобы быть универсальным контейнером. Класс *valarray* обеспечивает много полезных численных операций, однако из стандартных контейнерных операций (§ 17.1.1) он предлагает только *size*() и оператор индексации [] (§ 22.4.2). Указатель на элемент массива *valarray* является итератором с произвольным доступом (§ 19.2.1).

17.5.3. Битовые наборы

Часто некоторые аспекты какой-либо системы, такие как состояние входного потока (§ 21.3.3), представляются в качестве набора признаков («флажков»), показывающих бинарные условия вроде «хорошо/плохо», «правда/ложь» и «включено/выключено». Язык C++ эффективно поддерживает понятие небольших наборов признаков через битовые операции над целыми числами (§ 6.2.4). Эти операции включают в себя & (логическое И), | (логическое ИЛИ), ^ (исключающее ИЛИ), << (сдвиг влево) и >> (сдвиг вправо). Класс *bitset<N>* обобщает это понятие и предлагает большие удобства, обеспечивая операции с набором из *N* бит, индексированных от *0* до *N–1*, где *N* известно во время компиляции. Для битовых наборов, не укладывающихся в диапазон чисел типа *long int*, удобнее использовать *bitset*, чем оперировать непосредственно с числами. Для наборов поменьше может быть как выигрыш, так и проигрыш в эффективности. Если вы хотите задать битам имена, а не нумеровать их, то альтернативой может быть использование множеств (§ 17.4.1), перечислений (§ 4.8) или битовых полей (§ B.8.1).

Класс *bitset<N>* — это массив из *N* бит. Он отличается от *vector<bool>* (§ 16.3.1.1) тем, что имеет фиксированную длину, а от множества (§ 17.4.3) — тем, что его биты проиндексированы целыми числами, а не ассоциативно — значениями. Кроме того, битовый набор обеспечивает операции для манипулирования битами.

К одному биту невозможно обратиться прямо по встроенному указателю (§ 5.1), поэтому *bitset* вводит тип, ссылающийся на бит. Это действительно универсально полезный прием для адресации объектов, для которых встроенный указатель по каким-то причинам не подходит:

```
template<size_t N>class std::bitset {
public:
    class reference {                    // ссылка на одиночный бит
        friend class bitset;
        reference();
```

```
    public:                                    // b[i] относится к (i+1)-му биту
        ~reference ();
        reference& operator= (bool x);         // для b[i]=x;
        reference& operator= (const reference&); // для b[i]=b[j];
        bool operator~ () const;               // инвертирует b[i]
        operator bool () const;                // для x=b[i]
        reference& flip ();                    // для b[i].flip();
    };
    //...
};
```

Шаблон ***bitset*** определен в пространстве имен ***std*** и представлен в <***bitset***>.

Так уж исторически сложилось, что ***bitset*** по стилю несколько отличается от классов стандартной библиотеки. Например, если индекс (также называемый *битовой позицией*) выходит за пределы, генерируется исключение ***out_of_range***. Никакие итераторы не обеспечиваются. Битовые позиции нумеруются справа налево — точно так же, как нумеруют разряды, так что значение ***b[i]*** — это ***pow*** (***2***, ***i***). Таким образом, битовый набор можно представить себе как *N*-разрядное двоичное число:

позиция:	9	8	7	6	5	4	3	2	1	0
bitset<10>(989):	1	1	1	1	0	1	1	1	0	1

17.5.3.1. Конструкторы

Битовый набор можно сконструировать со значениями по умолчанию из битов в числе типа ***unsigned long int*** или из строки ***string***:

```
template<size_t N> class bitset {
public:
    //...
    // конструкторы:

    bitset ();                          // N-битный нуль
    bitset (unsigned long val);         // биты из val

    // Tr — это свойство символа (§ 20.2):
    template<class Ch, class Tr, class A>
    // биты из строки str:
    explicit bitset (const basic_string<Ch, Tr, A>& str,
        basic_string<Ch, Tr, A>::size_type pos = 0,
        basic_string<Ch, Tr, A>::size_type n = basic_string<Ch, Tr, A>::npos);

    //...
};
```

Значение бита по умолчанию равно ***0***. Когда задается аргумент типа ***unsigned long int***, каждый бит в этом целом используется для инициализации соответствующего бита в битовом наборе (если такой есть). Аргумент типа ***basic_string*** (глава 20) делает то же самое, только символ *'0'* дает битовое значение ***0***, символ *'1'* — битовое значение *1*, а другие символы приводят к генерации исключения ***invalid_argument***. По умолчанию для инициализации используется полная строка. Однако в стиле конструктора ***basic_string*** (§ 20.3.4) пользователь может определить, что должен использоваться только диапазон символов от ***pos*** до конца строки или до ***pos+n***. Например:

```
void f()
{
    bitset<10> b1;                          // все 0

    bitset<16> b2 = 0xaaaa;                 // 1010101010101010
    bitset<32> b3 = 0xaaaa;                 // 00000000000000001010101010101010

    bitset<10> b4 (string ("1010101010"));              // 1010101010
    bitset<10> b5 (string ("10110111011110"), 4);       // 0111011110
    bitset<10> b6 (string ("10110111011110"), 2, 8);    // 0011011101

    bitset<10> b7 (string ("n0g00d"));  // invalid_argument (недопустимый аргумент)
    bitset<10> b8 = "n0g00d";           // ошибка: нет преобразования string в bitset
}
```

Ключевая идея при проектировании ***bitset*** заключается в том, что для битовых наборов, умещающихся в одно слово, может быть обеспечена оптимизированная реализация. Интерфейс отражает это допущение.

17.5.3.2. Операции с битами

Класс ***bitset*** обеспечивает операции для доступа к отдельным битам и для манипулирования всеми битами в наборе:

```
template<size_t N> class std::bitset {
public:
    // ...
    // операции с битовым набором:

    reference operator[] (size_t pos);          // b[i]
    bitset& operator&= (const bitset& s);       // И
    bitset& operator|= (const bitset& s);       // ИЛИ
    bitset& operator^= (const bitset& s);       // исключающее ИЛИ
    bitset& operator<<= (size_t n);             // логический сдвиг влево
                                                // (с заполнением нулями)
    bitset& operator>>= (size_t n);             // логический сдвиг вправо
                                                // (с заполнением нулями)
    bitset& set ();                             // установка всех битов в 1
    bitset& set (size_t pos, int val = 1);      // b[pos]=1
    bitset& reset ();                           // установка всех битов в 0
    bitset& reset (size_t pos);                 // b[pos]=0
    bitset& flip ();                            // изменение значения каждого бита
    bitset& flip (size_t pos);                  // изменение значения b[pos]
    bitset operator~ () const
        { return bitset<N> (*this).flip (); }   // создание дополнительного набора
    bitset operator<< (size_t n) const
        { return bitset<N> (*this)<<=n; }       // создание сдвинутого набора
    bitset operator>> (size_t n) const
        { return bitset<N> (*this)>>=n; }       // создание сдвинутого набора

    // ...
};
```

Оператор индексации [] генерирует исключение ***out_of_range***, если индекс выходит за пределы набора. Непроверяемой индексации нет.

Значение *bitset*&, возвращаемое этими операциями, — это **this*. Оператор, возвращающий *bitset* (а не *bitset*&), делает копию **this*, применяет свою операцию к копии и возвращает результат. В частности, >> и << являются операторами сдвига, а не операциями ввода/вывода. Оператор вывода для *bitset* — это <<, принимающий *ostream* и *bitset* (§ 17.5.3.3).

При сдвиге битов применяется логический (а не циклический) сдвиг. Это приводит к тому, что некоторые биты «выпадают из конца», а некоторые позиции заполняются значениями по умолчанию — нулями. Отметим, что поскольку *size_t* — беззнаковый тип, на отрицательное число сдвинуть невозможно. Однако это ведет к тому, что *b<<-1* пытается сдвинуть *b* на очень большое положительное число и тем самым обнуляет все биты *b*. Ваш компилятор должен пожаловаться на это.

17.5.3.3. Другие операции

Кроме того, класс *bitset* поддерживает такие операции как *size* (), ==, ввод/вывод и т. д.:

```
template<size_t N> class bitset {
public:
    //...
    unsigned long to_ulong () const;

    template<class Ch, class Tr, class A> basic_string<Ch, Tr, A> to_string () const;

    size_t count () const;              // число бит со значением 1
    size_t size () const { return N; }  // число бит

    bool operator== (const bitset& s) const;
    bool operator!= (const bitset& s) const;

    bool test (size_t pos) const;       // true, если b[pos] равно 1
    bool any () const;                  // true, если хотя бы один бит равен 1
    bool none () const;                 // true, если ни один бит не равен 1
};
```

Операции *to_ulong* () и *to_string* () обеспечивают обращение операций конструирования. Этим операциям отдали предпочтение перед преобразованиями, чтобы избежать неочевидных преобразований. Если в значении *bitset* так много значащих битов, что их не представить в *unsigned long*, операция *to_ulong* () сгенерирует исключение *overflow_error*.

Операция *to_string* () производит строку желаемого типа, содержащую последовательность символов *'0'* и *'1'*; *basic_string* — это шаблон, используемый для реализации строк (глава 20). Мы можем использовать *to_string*, чтобы написать двоичное представление целого:

```
void binary (int i)
{
    bitset<8*sizeof (int)> b = i;       // предполагается 8-битный байт
                                        // (см. также § 22.2)
    cout << b.template to_string < char, char_traits<char>, allocator<char> > () << '\n';
}
```

К сожалению, вызов явно квалифицированного шаблона члена требует довольно сложного и редкого синтаксиса (§ B.13.6).

Вдобавок к функциям-членам *bitset* обеспечивает двоичное & (логическое «И»), | (логическое «ИЛИ»), ^ («исключающее ИЛИ») и обычные операторы ввода/вывода:

```
template<size_t N> bitset<N>
    std::operator& (const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N>
    std::operator| (const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N>
    std::operator^ (const bitset<N>&, const bitset<N>&);

template<class char T, class Tr, size_t N>
basic_istream<char T, Tr>&
    std operator>> (basic_istream<char T, Tr>&, bitset<N>&);
template<class char T, class Tr, size_t N>
    basic_ostream<char T, Tr>&
    std::operator<< (basic_ostream<char T, Tr>&, const bitset<N>&);
```

Поэтому мы можем выдать битовый набор без преобразования его в строку. Например:

```
void binary (int i)
{
    bitset<8*sizeof(int)> b = i;    // предполагается 8-битный байт (см. также § 22.2)
    cout << b << '\n';
}
```

Эта программа выводит биты, представляя их символами *'0'* и *'1'* слева направо, самый старший бит — самый левый.

17.5.4. Встроенные массивы

Встроенный массив предоставляет индексацию и итераторы с произвольным доступом в виде обычных указателей (§ 2.7.2). Однако сам массив не знает собственных размеров, поэтому следить за его размерами должны пользователи. Вообще говоря, массивы не обеспечивают стандартных типов и операций-членов.

Возможно, и иногда полезно, представить обычный массив в таком виде, который обеспечил бы удобства стандартного контейнера без изменения его природы на низком уровне:

```
template<class T, int max> struct c_array {
    typedef T value_type;

    typedef T * iterator;
    typedef const T * const_iterator;

    typedef T& reference;
    typedef const T& const_reference;

    T v[max];
    operator T * () { return v; }

    reference operator[] (ptrdiff_t i) { return v[i]; }
    const_reference operator[] (ptrdiff_t i) const { return v[i]; }

    iterator begin () { return v; }
    const_iterator begin () const { return v; }
```

```
        iterator end () { return v+max; }
        const_iterator end () const { return v+max; }

        ptrdiff_t size () const { return max; }
};
```

Для совместимости с массивами я воспользовался знаковым типом ***ptrdiff_t*** (§ 16.1.2), а не беззнаковым ***size_t*** в качестве типа индекса. Применение могло бы привести к тонким двусмысленностям при использовании [ ] с объектами ***c_array***.

Шаблон ***c_array*** не является частью стандартной библиотеки, он представлен здесь в качестве примера «впихивания» чужеродного контейнера в рамки стандартных контейнеров. Его можно использовать со стандартными алгоритмами (глава 18), применяя ***begin*** (), ***end*** () и т. п. Такой контейнер можно разместить в стеке без всякого неявного динамического распределения памяти. Его также можно передавать функциям, написанным в стиле C, аргумент которых — указатель. Например:

```
void f(int* p, int sz);                    // функция в стиле C

void g()
{
        c_array<int, 10> a;
        f(a, a.size());                    // используется стиль C
        c_array<int, 10>::iterator
                p = find(a.begin(), a.end(), 777);   // стиль C++/STL
        // ...
}
```

17.6. Определение нового контейнера

Стандартные контейнеры обеспечивают основу, к которой пользователь может добавлять свои механизмы. Здесь я покажу, как создать контейнер, чтобы в использовании он был взаимозаменяем со стандартными контейнерами. Его реализация будет реалистичной, но не оптимальной. Интерфейс выбирается из тех соображений, чтобы он был поближе к существующим, широко доступным и высококачественным реализациям понятия ***hash_map***. Используйте приведенный здесь ***hash_map*** для изучения общих вопросов, а потом применяйте поддерживаемый ***hash_map***.

17.6.1. hash_map

Контейнер ***map*** — это ассоциативный контейнер, принимающий элементы почти любого типа. Он обеспечивает такую возможность, опираясь для сравнения элементов только на операцию «меньше» (§ 17.4.1.5). Однако если мы знаем больше о типе ключа, то нередко можем ускорить поиск элементов, введя хэш-функцию и реализовав контейнер как хэш-таблицу.

Хэш-функция — это функция, которая по значению быстро вычисляет индекс таким образом, что два различных значения редко соответствуют одинаковому индексу. В основном хэш-таблицы реализуются размещением значения по его индексу, если там уже не расположилось другое значение, и «рядом» в противном случае. Поиск элемента, расположенного по своему индексу, занимает мало времени, а поиск элемента, расположенного «рядом», если такой существует, тоже не очень долог, при условии,что проверка на равенство проводится сравнительно быстро. И потому не так уж странно,

что для больших контейнеров, где поиск играет очень большую роль, ***hash_map*** работает в 5–10 раз быстрее, чем ***map***. С другой стороны, ***hash_map*** с плохо подобранными хэш-функциями может оказаться гораздо медленнее, чем ***map***.

Есть много способов построения хэш-таблицы. Интерфейс ***hash_map*** разработан так, чтобы отличаться от интерфейса стандартных ассоциативных контейнеров только тогда, когда это необходимо для выигрыша в быстродействии за счет хэширования. Самое фундаментальное различие между ***map*** и ***hash_map*** заключается в том, что ***map*** требует от типа своих элементов оператор <, в то время как ***hash_map*** требует == и хэш-функцию. Таким образом, ***hash_map*** должен отличаться от ***map*** при создании не по умолчанию. Например:

```
map<string, int> m1;                  // сравнение строк при помощи оператора <
map<string, int, Nocase> m2;          // сравнение строк при помощи
                                      // Nocase() (§ 17.1.4.1)

hash_map<string, int> hm1;            // хэширование при помощи Hash<string>()
                                      // (§ 17.6.2.3), сравнение при помощи ==
hash_map<string, int, hfct> m2;       // хэширование при помощи hfct(),
                                      // сравнение при помощи ==
hash_map<string, int, hfct, eql> m2;  // хэширование при помощи hfct(),
                                      // сравнение при помощи eql()
```

Контейнер, использующий хэшированный поиск, реализуется при помощи одной или более таблиц. Кроме хранения своих элементов контейнер должен следить, какие значения связаны с каждым хэшированным значением («индекс» в объяснении, приведенном выше); это делается при помощи хэш-таблицы. Большинство реализаций хэш-таблиц существенно снижают быстродействие, если эта таблица становится «слишком полной» — заполненной, скажем, на 75%. Поэтому определенный следующим образом ***hash_map*** автоматически изменяет размер, когда становится слишком полным. Однако изменение размера может оказаться дорогим, поэтому полезно иметь возможность задать первоначальный размер.

Итак, в первом приближении ***hash_map*** выглядит следующим образом:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T> > >
class hash_map {
        // как map, за исключением:

        typedef H Hasher;
        typedef EQ key_equal;

        hash_map(const T& dv = T(), size_type n=101, const H& hf=H(), const EQ& = EQ());
        template<class In> hash_map(In first, In last,
                const T& dv = T(), size_type n = 101, const H& hf = H(), const EQ& = EQ());
};
```

В своей основе это интерфейс ***map*** (§ 17.4.1.4) с <, замененным на ==, и хэш-функцией.

Примеры применения ***map***, приведенные в этой книге до сих пор (§ 3.7.4, § 6.1, § 17.4.1), могут быть преобразованы к использованию ***hash_map*** простой заменой имени ***map*** на ***hash_map***. Часто изменение ***map*** на ***hash_map*** можно облегчить, используя ***typedef***. Например:

```
typedef hash_map<string, record> Map;
Map dictionary;
```

Оператор *typedef* также полезен для сокрытия от пользователей точного типа словаря.

Хотя это не совсем корректно, я представляю выигрыш и потери от применения *map* и *hash_map* как просто выигрыш во времени за счет памяти и наоборот. Если эффективность не играет роли, не стоит тратить время, чтобы сделать выбор между ними: хорошо работать будут и *map*, и *hash_map*. Для больших и интенсивно используемых таблиц *hash_map* имеет явное преимущество в скорости и должен использоваться, если нет дефицита памяти. И даже тогда , прежде чем выбрать «просто» *map*, я бы рассмотрел другие способы экономии памяти. Нужно произвести непосредственные замеры, чтобы не оптимизировать неверную программу.

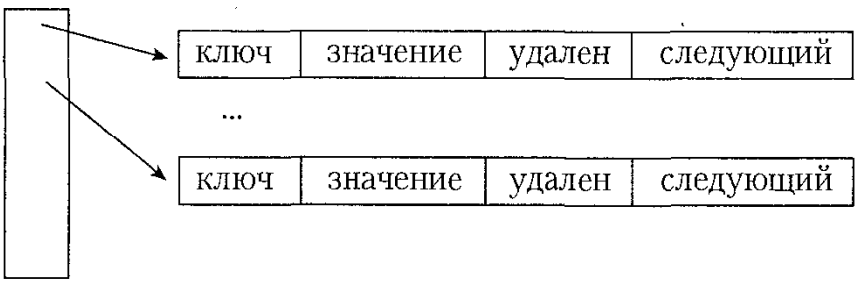
Ключ к эффективному хэшированию кроется в качестве хэш-функций. Если хорошая хэш-функция недостижима, *map* может легко превзойти *hash_map* по эффективности. Хэширование, основанное на С-строке, строке *string* или целом числе, обычно очень эффективно. Однако стоит помнить, что эффективность хэш-функций чрезвычайно зависит от собственно хэшируемых значений (§ 17.8[35]). Контейнер *hash_map* нужно использовать там, где оператор < не определен или не подходит к предполагающемуся ключу. И наоборот, хэш-функция не определяет упорядочения так, как это делает <, поэтому *map* нужно использовать, когда важно держать элементы отсортированными.

Как и *map*, *hash_map* обеспечивает операцию *find* (), чтобы позволить программисту определить, был ли ключ вставлен.

17.6.2. Представление и конструирование

Возможно много разных реализаций *hash_map*. Здесь я привожу одну, которая обеспечивает неплохую скорость, и наиболее важные операции которой довольно просты. Ключевые операции — это конструкторы, поиск (оператор []), изменение размеров и удаление элемента (*erase* ()).

Простая реализация, выбранная здесь, основана на хэш-таблице, представляющей собою вектор указателей на записи (entries). Каждая запись хранит ключ *key*, значение *value*, указатель на следующую запись (если она есть) с тем же хэш-значением, и бит *erased* («удален»):



Выраженное в объявлениях, это выглядит следующим образом:

```
template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<pair <const Key, T> > >
class hash_map {
    //...
private:                    // представление
    struct Entry {
```

```
            key_type key;
            mapped_type val;
            bool erased;
            Entry* next;        // связь переполнения хэша
            Entry (key_type k, mapped_type v, Entry* n)
                : key (k), val (v), erased (false), next (n) { }
        };
        vector<Entry> v;        // действительные записи
        vector<Entry*> b;       // хэш-таблица: указатели на v
        //...
};
```

Отметим бит ***erased***. Способ, которым здесь обрабатывается несколько значений с одинаковым хэш-значением, затрудняет удаление элемента. Поэтому вместо действительного удаления при вызове функции ***erase*** () я просто помечаю элемент ***erased*** («удален») и не обращаю на него внимания, пока таблица не изменит размеры.

В дополнение к главной структуре данных ***hash_map*** нуждается в некотором количестве административных данных. Естественно, каждому конструктору нужно установить все это. Например:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair <const Key, T> > >
class hash_map {
        //...
        hash_map (const T& dv = T (), size_type n=101, const H& h = H (),
            const EQ& e = EQ ()) : default_value (dv), b (n),
            no_of_erased (0), hash (h), eq (e)
        {
            set_load ();                        // по умолчанию
            v.reserve (max_load*b.size ());     // резервируем память для роста
        }
        void set_load (float m = 0.7, float g = 1.6) { max_load = m; grow = g; }
        //...
private:
        float max_load;             // удерживает v.size()<=b.size()*max_load
        float grow;                 // при необходимости изменяем размеры
                                    // resize(bucket_count()*grow)
        size_type no_of_erased;     // число записей в v, занятых
                                    // стертыми элементами
        Hasher hash;                // хэш-функция
        key_equal eq;               // равенство
        const T default_value;      // значение по умолчанию, используемое
                                    // оператором []
};
```

Стандартные ассоциативные контейнеры требуют, чтобы отображенный тип имел значение по умолчанию (§ 17.4.1.7). Это ограничение не является логически необ-

ходимым и может причинить неудобства. Беря в качестве аргумента значение по умолчанию, можно написать:

```
hash_map<string, Number> phone_book1;                  // по умолчанию: Number()
hash_map<string, Number> phone_book2 (Number (411));   // по умолчанию: Number(411)
```

17.6.2.1. Поиск

Наконец мы можем ввести важнейшие операции поиска:

```
template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<pair <const Key, T> > >
class hash_map {
    //...
    mapped_type& operator[] (const key_type& k);

    iterator find (const key_type&);
    const_iterator find (const key_type&) const;
    //...
};
```

Чтобы найти значение, ***operator[] ()*** использует хэш-функцию с целью нахождения индекса в хэш-таблице для ключа. Затем он просматривает записи, пока не найдет совпадающий ключ. Значение в этой записи и есть то, что мы ищем. Если найти его не удается, вводится значение по умолчанию:

```
template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<pair <const Key, T> > >
hash_map<Key, T, H, EQ, A>::mapped_type&
    typename hash_map<Key, T, H, EQ, A>::operator[]    (const key_type& k)
{
    size_type i = hash (k)%b.size ();              // хэш

    for (Entry* p=b[i]; p; p = ->next)             // поиск среди записей, хэшированных по i
        if (eq (k, p->key)) {                      // найдено
            if (p->erased) {                       // повторная вставка
                p->erased = false;
                no_of_erased--;
                return p->val = default_value;
            }
            return p->val;
        }
    // не найдено

    if (size_type (b.size ()*max_load) <= v.size ()) {   // если слишком заполнено
        resize (b.size ()*grow);                         // расширяем
        return operator[] (k);                           // повторное хэширование
    }

    v.push_back (Entry (k, default_value, b[i]));   // добавление записи
    b[i] = &v.back ();                              // указывает на новый элемент

    return b[i]->val;
}
```

В отличие от ***map***, ***hash_map*** не опирается на проверку равенства, синтезированную из операции «меньше» (§ 17.1.4.1). Это связано с вызовом функции ***eq*** () в цикле для

поиска элементов с тем же хэш-значением. Данный цикл является решающим для производительности поиска, а для обычных очевидных типов ключа, таких как ***string*** и С-строка, перерасход времени на лишние сравнения может быть значительным.

Для представления набора значений с одинаковым хэш-значением я мог бы воспользоваться *set<Entry>*. Однако если у нас есть хорошая хэш-функция (***hash*** ()) и хэш-таблица подходящего размера, большинство таких множеств (***set***) будут иметь ровно один элемент. Поэтому я связал элементы этого множества вместе при помощи поля ***next*** в записи ***Entry*** (§ 17.8[27]).

Отметим, что *b* содержит указатели на элементы *v*, и что элементы добавляются к *v*. Вообще говоря, функция ***push_back*** () может привести к перераспределению памяти и тем самым сделать указатели на элементы невёрными (§ 16.3.5), однако в данном случае конструкторы (§ 17.6.2) и ***resize*** () предусмотрительно резервируют (операцией ***reserve*** ()) достаточно памяти, чтобы не случилось неожиданного перераспределения.

17.6.2.2. Удаление и повторное хэширование

Когда таблица становится слишком заполненной, хэшированный поиск становится неэффективным. Чтобы снизить вероятность этого, при вызове оператора индексации ([]) таблица автоматически изменяет размеры (операцией ***resize*** ()). Операция *set_load* () (§ 17.6.2) предоставляет способ контроля над тем, когда и как происходит изменение размеров. Другие функции нужны для того, чтобы позволить программисту следить за состоянием ***hash_map***:

```
template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<pair <const Key, T> > >
class hash_map {
    //...

    void resize (size_type n);          // устанавливает размер хэш-таблицы равным n

    void erase (iterator position);     // удаляет элемент, на который
                                        // указывает итератор

    size_type size () const { return v.size () - no_of_erased; }   // число элементов

    size_type bucket_count () const { return b.size (); }          // размер хэш-таблицы

    Hasher hash_fun () const { return hash; }                      // используемая
                                                                   // хэш-функция
    key_equal key_eq () const { return eq; }                       // используемое
                                                                   // равенство

    //...
};
```

Операция ***resize*** () очень важна, довольно проста и потенциально дорога:

```
template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<pair <const Key, T> > >
void hash_map<Key, T, H, EQ, A>::resize (size_type s)
{
    size_type i=v.size();

    if (s <= b.size()) return;
```

```
        while (no_of_erased) {          // фактически уничтожает удаленные элементы
            if (v[--i].erased) {
                v.erase(&v[i]);
                --no_of_erased;
            }
        }
        b.resize(s);                    // добавляет s указателей на b.size()
        fill(b.begin(), b.end(), 0);    // обнуляет все элементы (§ 18.6.6)
        v.reserve(s*max_load);          // если для v нужно перераспределить память,
                                        // пусть это произойдет сейчас

        for (size_type i = 0; i<v.size(); i++) {          // повторное хэширование:
            size_type ii = hash(v[i].key)%b.size();       // хэширование
            v[i].next = b[ii];                            // связь
            b[ii] = &v.begin() + i;
        }
    }
```

При необходимости пользователь может «вручную» обратиться к ***resize*** (), чтобы гарантировать, что размещение не произойдет в неподходящее время. Я считаю, что функция ***resize*** () очень важна во многих прикладных программах, но она не связана фундаментально с понятием хэш-таблицы. В некоторых стратегиях реализации она вообще не нужна.

Вся реальная работа делается где-то в другом месте (и только если ***hash_map*** меняет размеры), поэтому ***erase*** () реализуется тривиально:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T> > >
void hash_map<Key, T, H, EQ, A>::erase(iterator p);     // удаляет элемент, на который
                                                        // указывает итератор
{
    if (p->erased == false) no_of_erased++;
    p->erased = true;
}
```

17.6.2.3. Хэширование

Чтобы завершить ***hash_map*::operator**[] (), нам нужно определить ***hash*** () и ***eq*** (). По причинам, которые станут яснее в § 18.4, хэш-функцию лучше всего определить как ***operator*** () объекта-функции:

```
template<class T> struct Hash : unary_function<T, size_t> {
    size_t operator() (const T& key) const;
};
```

Хорошая хэш-функция берет ключ и возвращает целое число, так что различные ключи с высокой степенью вероятности отвечают различным числам. Выбор хорошей хэш-функции — это искусство. Однако часто приемлемой является операция исключающего ИЛИ с битами ключа, представленного как целое число:

```
typedef char* Pchar;
template< > size_t Hash<Pchar>::operator() (const Pchar& key) const
{
    size_t res = 0;
```

```
    size_t len = sizeof(T);
    const char* p = reinterpret_cast<const char*>(&key); // доступ к объекту как к
                                                         // последовательности байтов
    while (len--) res = (res<<1)^*p++; // использование целочисленных
                                       // значений символов
    return res;
}
```

Мы можем сделать что-то получше, когда больше узнаем о хэшируемых объектах. В частности, если объект содержит указатель, если объект большой, или если из-за требований к выравниванию членов в представлении остаются пустые места («дыры»), обычно мы можем что-нибудь улучшить (см. § 17.8[29]).

С-строка — это указатель (на символы), а строка ***string*** содержит указатель. Следовательно, уместны специализации:

```
typedef char* Pchar;
template<> size_t Hash<char*>::operator() (const char* key) const
{
    size_t res = 0;
    Pchar p=key;
    while (*p) res = (res<<1)^*p++;          // используются целые значения символов
    return res;
}

template<> size_t Hash<string>::operator() (const string& key) const
{
    size_t res = 0;

    typedef string::const_iterator CI;
    CI p = key.begin();
    CI end = key.end();

    while (p!=end) res = (res<<1)^*p++;      // используются целые значения символов
    return res;
}
```

Реализация *hash_map* включает хэш-функции по крайней мере для целых и строковых ключей. Для более сложных типов ключей пользователю, вероятно, придется прибегнуть к помощи подходящих специализаций. При выборе хэш-функции очень важно экспериментирование и измерение результата. Интуиция в этой области работает плохо.

Чтобы завершить *hash_map*, нам нужно определить итераторы и большое количество второстепенных тривиальных функций; это оставлено в качестве упражнения (§ 17.8[34]).

17.6.3. Другие хэшированные ассоциативные контейнеры

Для согласованности и полноты ***hash_map*** должны бы соответствовать ***hash_set***, ***hash_multimap*** и ***hash_multiset***. Их определения очевидны, исходя из ***hash_map***, ***map***, ***multimap***, ***set*** и ***multiset***, поэтому я оставлю их в качестве упражнений (§ 17.8[34]). Доступны хорошие свободно распространяемые и коммерческие реализации этих хэшированных ассоциативных контейнеров. Для реальных программ их следует предпочитать версиям вроде моих, сконцентрированных на локальных проблемах.

17.7. Советы

[1] По умолчанию, если вам нужен контейнер, используйте вектор (*vector*); § 17.1.2.
[2] Узнайте стоимость (сложность, асимптотическую *O*-оценку) каждой операции, которой вы часто пользуетесь; § 17.1.2.
[3] Интерфейс, реализация и представление контейнера — это различные понятия. Не смешивайте их; § 17.1.3.
[4] Сортировать и искать можно в соответствии со множеством критериев; § 17.1.4.1.
[5] Не пользуйтесь С-строками в качестве ключа, если вы не обеспечили подходящего критерия сравнения; § 17.1.4.1.
[6] Вы можете определить критерий сравнения так, чтобы эквивалентные, но все же отличающиеся значения ключей, отображались в один и тот же ключ; § 17.1.4.1.
[7] При вставке и удалении элементов предпочитайте операции с концом последовательности (*back*-операции); § 17.1.4.1.
[8] Когда нужно производить много вставок и удалений из начала или середины контейнера, используйте списки (*list*); § 17.2.2.
[9] Когда нужно главным образом обращаться к элементам по ключу, пользуйтесь *map* и *multimap*; § 17.4.1.
[10] Для достижения максимальной гибкости пользуйтесь минимальным набором операций; § 17.1.1.
[11] Если элементы должны быть упорядочены, выбирайте *map*, а не *hash_map*; § 17.6.1.
[12] Когда важна скорость поиска, выбирайте *hash_map* , а не *map*; § 17.6.1.
[13] Если для элементов невозможно определить операцию «меньше», между *map* и *hash_map* выбирайте *hash_map*; § 17.6.1.
[14] Для проверки того, есть ли данный ключ в ассоциативном контейнере, пользуйтесь функцией *find* (); § 17.4.1.6.
[15] Чтобы найти в ассоциативном контейнере элементы с данным ключом, пользуйтесь функцией *equal_range* (); § 17.4.1.6.
[16] Когда нужно хранить несколько значений для одного ключа, пользуйтесь *multimap*; § 17.4.2.
[17] Когда сам ключ является единственным значением, которое нужно хранить, пользуйтесь *set* и *multiset*; § 17.4.3.

17.8. Упражнения

Решения к некоторым упражнениям этой главы можно найти, заглянув в исходный текст вашей стандартной библиотеки. Но сделайте одолжение — попытайтесь найти собственные решения, прежде чем смотреть, как подошел к проблеме разработчик библиотеки. Потом посмотрите на собственную версию реализации контейнеров и их операций.

1. (*2.5) Поймите, что означает *O* () (§ 17.1.2). Проделайте измерения для операций стандартных контейнеров, чтобы определить постоянные коэффициенты (множители перед *O* ()).
2. (*2) Многие телефонные номера не умещаются в тип *long*. Напишите тип *phone_number* и класс, обеспечивающий набор полезных операций с контейнером телефонных номеров типа *phone_number*.

3. (*2) Напишите программу, которая бы заносила различные слова в файл в алфавитном порядке. Сделайте две версии: пусть в одной слово будет просто последовательностью символов, разделенной символами-разделителями, а в другой — последовательностью символов, разделенной символами, не являющимися буквами.
4. (*2.5) Реализуйте простую карточную игру «солитер» (пасьянс «косынка»).
5. (*1.5) Реализуйте простую проверку, является ли слово палиндромом («перевертышем», то есть словом, в котором буквы располагаются симметрично, например: «наган», «казак», «боб»). Реализуйте простую проверку, является ли целое число палиндромом. Реализуйте простую проверку, является ли последовательность палиндромом. Обобщите.
6. (*1.5) Определите очередь, используя (только) два стека.
7. (*1.5) Определите стек, похожий на ***stack*** (§ 17.3.1), но чтобы он не копировал свой базовый контейнер и допускал бы итерации по своим элементам.
8. (*3) Понятия потока, задачи и процесса описывают организацию параллельности на вашем компьютере. Поймите, как достигается параллельность. Механизм параллелизма использует блокировку, чтобы предотвратить одновременный доступ двух задач к одной области памяти. Используйте машинный механизм блокировки для реализации класса блокировки.
9. (*2.5) Прочитайте из входа последовательность дат, например ***дек85***, ***дек50***, ***янв76*** и т. д. и потом выведите их так, чтобы более поздние даты шли первыми. Формат даты: три символа — месяц, за ними два символа — год. Считаем, что все годы относятся к одному веку.
10. (*2.5) Обобщите входной формат для дат, чтобы можно было вводить даты вроде ***дек1985***, ***3/12/1990***, ***(30 дек. 1950)***, ***3/6/2001*** и т. п. Измените упражнение § 17.8[9] в соответствии с новым форматом.
11. (*1.5) При помощи ***bitset*** распечатайте двоичные значения чисел, включая ***0***, ***1***, ***-1***, ***18***, ***-18*** и максимальное положительное целое (***int***).
12. (*1.5) Используйте ***bitset*** для хранения информации о том, кто из учеников в данный день присутствует в классе. Прочтите битовые наборы ***bitset*** в сериях по 12 сдней и определите, кто присутствовал в каждый из дней. Определите, кто из учеников присутствовал по крайней мере 8 дней.
13. (*1.5) Напишите список указателей (***List***), который уничтожает (***delete***) объект, на который указывает элемент списка во время уничтожения всего списка или при удалении элемента из списка операцией ***remove*** ().
14. (*1.5) Имея объект типа ***stack***, распечатайте по порядку его элементы (не изменяя значения стека).
15. (*2.5) Завершите шаблон ***hash_map*** (§ 17.6.1). Это подразумевает реализацию функций ***find*** () и ***equal_range*** () и изобретение способа тестирования завершенного шаблона. Проверьте ***hash_map*** с по крайней мере одним типом ключа, для которого хэш-функция по умолчанию не подходила бы.
16. (*2.5) Реализуйте и протестируйте список в стиле стандартного ***list***.
17. (*2) Иногда перерасход памяти для списков может вызвать трудности. Напишите и проверьте односвязный список в стиле стандартного контейнера.
18. (*2.5) Реализуйте список, похожий на стандартный ***list***, но поддерживающий индексацию. Сравните затраты на индексацию для различных списков с затратами на индексацию для вектора той же длины.

19. (*2) Реализуйте шаблон функции, который производит слияние двух контейнеров.
20. (*1.5) Для С-строки определите, не является ли она палиндромом. Определите, не является ли палиндромом начальная последовательность из хотя бы трех слов в строке.
21. (*2) Прочтите последовательность пар (***имя***, ***значение***) и создайте сортированный список четверок (***имя***, ***сумма***, ***среднее значение***, ***медиана***). Распечатайте этот список.
22. (*2.5) Определите расходы памяти для каждого стандартного контейнера в вашей реализации.
23. (*3.5) Подумайте, какая стратегия реализации была бы разумной для ***hash_map***, чтобы минимизировать объем памяти, используемый таким массивом. Подумайте, какая стратегия реализации была бы разумной для ***hash_map***, чтобы минимизировать время поиска. В обоих случаях подумайте, какими операциями вы могли бы пренебречь, чтобы приблизиться к соответствующему идеалу (без лишних затрат памяти и без лишних затрат времени на поиск соответственно).
24. (*2) Придумайте стратегию обработки переполнения в ***hash_map*** (слишком много различных значений хэшируются в одно и тоже хэш-значение), чтобы реализация ***equal_range*** () была тривиальной.
25. (*2.5) Оцените затраты памяти ***hash_map***, а затем измерьте их. Сравните оценку с измерением. Сравните затраты памяти вашего ***hash_map*** и реализованного вами ***map***.
26. (*2.5) Протестируйте (используя профайлер) вашу ***hash_map*** и определите, где тратится время. Проделайте то же самое с реализованным вами ассоциативным массивом ***map*** и каким-нибудь широко распространенным ***hash_map***.
27. (*2.5) Реализуйте ***hash_map***, основанный на ***vector<map<K, V>*>***, чтобы каждый ***map*** содержал все ключи с одинаковым хэш-значением.
28. (*3) Реализуйте ***hash_map***, используя Splay-деревья (см.: D. Sleator and R. E. Tarjan, *Self-Adjusting Binary Search Trees*, JASM, Vol. 32. 1985).
29. (*2) Дана структура данных, описывающая нечто похожее на строку:

```
struct St {
        int size;
        char type_indicator;
        char* buf;              // указывает на size символов
        St (const char* p);     // выделяет память под буфер
                                // и заполняет его
};
```

Создайте 1000 объектов типа ***St*** и используйте их в качестве ключей для ***hash_map***. Напишите программу для измерения быстродействия ***hash_map***. Напишите хэш-функцию (***Hash***; § 17.6.2.3) специально для ключей ***St***.

30. (*2) Предложите хотя бы четыре разных способа для удаления из ***hash_map*** «стертых» (***erased***) элементов. Вам следует использовать стандартный библиотечный алгоритм (§ 3.8, глава 18), чтобы избежать явного цикла.
31. (*3) Реализуйте ***hash_map*** с непосредственным удалением элементов.
32. (*2) Хэш-функции, представленные в § 17.6.2.3, не всегда рассматривают всё представление ключа. Когда часть представления будет проигнорирована? Напишите хэш-функцию, всегда рассматривающую всё представление ключа. Приведите при-

мер, когда было бы разумно не обращать внимания на часть ключа, и напишите хэш-функцию, которая рассчитывала бы хэш-значение, основываясь только на части значимого ключа.

33. (*2.5) Код хэш-функций часто оказывается похожим — в цикле получаются данные и они хэшируются. Определите ***Hash*** (§ 17.6.2.3), который получал бы свои данные повторяющимся вызовом функции, которую пользователь может определить для типа ключа. Например:

```
size_t res = 0;
while (size_t v = hash(key)) res = (res<<3)^v;
```

Здесь пользователь может определить ***hash(K)*** для каждого типа ***K***, который нужно хэшировать.

34. (*3) Дана некоторая реализация ***hash_map***, реализуйте ***hash_multimap***, ***hash_set*** и ***hash_multiset***.
35. (*2.5) Напишите хэш-функцию, предназначенную для того, чтобы отображать равномерно распределенные целочисленные значения в хэш-значения для таблицы размером около 1024. Получив эту функцию, придумайте набор из 1024 ключей, каждый из которых отображается в одно и то же значение.

18

Алгоритмы и объекты-функции

Формальность освобождает.
— Популярная у инженеров поговорка

Введение — обзор стандартных алгоритмов — последовательности — объекты-функции — предикаты — арифметические объекты — связыватели — объекты-функции как члены — ***for_each*** — поиск элементов — ***count*** — сравнение последовательностей — поиск — копирование — ***transform*** — замещение и удаление элементов — заполнение последовательности — изменение порядка — ***swap*** — сортированные последовательности — ***binary_search*** — ***merge*** — операции с наборами — ***min*** и ***max*** — кучи — перестановки — алгоритмы в стиле C — советы — упражнения.

18.1. Введение

Сами по себе контейнеры неинтересны. Чтобы стать поистине полезным, контейнер должен быть снабжен основными операциями, такими как определение его размера, итерирование, сортировка и поиск элементов. К счастью, стандартная библиотека предоставляет алгоритмы для выполенния большинства распространенных операций, которые требуются пользователю контейнера.

В этой главе приведен обзор стандартных алгоритмов и несколько примеров их использования, перечислены ключевые принципы и методы выражения алгоритмов на языке C++, а также подробно разъяснено несколько ключевых алгоритмов.

Объекты-функции обеспечивают механизм, посредством которого пользователь может приспособить стандартный алгоритм к своим нуждам. Объекты-функции дают необходимую информацию, которая нужна алгоритму для работы с данными пользователя. Поэтому упор делается на том, как определять и использовать объекты-функции.

18.2. Обзор алгоритмов стандартной библиотеки

На первый взгляд может показаться, что алгоритмов стандартной библиотеки несметное число. Однако их всего 60. Я видывал классы с бóльшим числом функций-членов. Кроме того, многие алгоритмы имеют много общего в своем поведении и интерфейсе, что облегчает их понимание. Как и с языком программирования, программист должен пользоваться действительно нужными и понятными алгоритмами — и толь-

ко ими. Нет смысла стремиться к тому, чтобы включить в программу возможно большое число стандартных алгоритмов или стараться заменить очевидные простые алгоритмы более сложными. Помните: главная задача при написании программы — это сделать ее ясной и понятной для того, кто будет читать ее; этим человеком можете оказаться вы сами через несколько лет. С другой стороны, проделывая что-то с элементами контейнера, думайте, нельзя ли это действие выразить средствами стандартного библиотечного алгоритма. Может быть, алгоритм уже существует. Если вы не будете стараться рассматривать свои действия в терминах универсальных алгоритмов, вы будете заново изобретать колесо.

Каждый алгоритм выражается шаблоном функции (§ 13.3) или набором шаблонов функций. Таким образом, алгоритм может работать с очень разными последовательностями, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор (§ 19.1), как правило для сообщения о неудаче используют конец входной последовательности. Например:

```
void f(list<string>& ls)
{
    list<string>::const_iterator p = find(ls.begin(), ls.end(), "Федор");
    if (p == ls.end()) {
        // "Федор" не найден
    }
    else {
        // вот он; p указывает на "Федор"
    }
}
```

Алгоритмы не выполняют проверки диапазона на их входе и выходе. Ошибку выхода за пределы диапазона нужно стараться предотвратить другими средствами (§ 18.3.1, § 19.3). Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. В частности, аргументами алгоритма определяется, возвращает ли он константый итератор или неконстантный итератор. Например:

```
void f(list<string>& li, const list<string>& ls)
{
    list<int>::iterator p = find(li.begin(), li.end(), 42);
    list<string>::const_iterator q = find(ls.begin(), ls.end(), "Кольцо");
}
```

Алгоритмы в стандартной библиотеке реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировку, поиск, вставку и удаление элементов. Все стандартные алгоритмы находятся в пространстве имен *std*, а их объявления — в заголовочном файле <*algorithm*>. Интересно, что большинство действительно распространенных алгоритмов столь просты, что шаблонные функции, как правило, являются встроенными (*inline*). Благодаря этой агрессивной оптимизации на уровне каждой функции циклы выполняются значительно быстрее.

Стандартные объекты-функции также находятся в пространстве имен *std*, но их объявления помещены в <*functional*>. Объекты-функции рассчитаны на то, чтобы их было легко встраивать как *inline*.

Немодифицирующие операции с последовательностями используются для извлечения информации из последовательности или для определения положения элемента в последовательности:

Немодифицирующие операции с последовательностями (§ 18.5) <algorithm>	
for_each ()	Выполняет операцию для каждого элемента последовательности
find ()	Находит первое вхождение значения в последовательность
find_if ()	Находит первое соответствие предикату (условию) в последовательности
find_first_of ()	Находит значение из одной последовательности в другой
adjacent_find ()	Находит пару соседних значений
count ()	Подсчитывает количество вхождений данного значения в последовательность
count_if ()	Подстчитывает количество выполнений данного предиката в последовательности
mismatch ()	Находит первый элемент, в котором две последовательности различаются
equal ()	*true*, если элементы в двух последовательностях попарно равны
search ()	Находит первое вхождение последовательности как подпоследовательности
find_end ()	Находит последнее вхождение последовательности как подпоследовательности
search_n ()	Находит *n*-е вхождение значения в последовательность

Большинство алгоритмов дают пользователю возможность определить операцию, выполняемую над каждым элементом или парой элементов. Это делает алгоритмы более универсальными и полезными, чем они кажутся на первый взгляд. В частности, пользователь может сам указать критерий равенства и различия (§ 18.4.2). Где это разумно, наиболее часто встречающиеся и полезные действия обеспечиваются по умолчанию.

Модифицирующие операции с последовательностями имеют между собой мало общего, кроме того очевидного свойства, что все они могут изменять значение элементов в последовательности:

Модифицирующие операции с последовательностями (§ 18.6) <algorithm>	
transform ()	Выполняет операцию над каждым элементом в последовательности
copy ()	Копирует последовательность, начиная с первого элемента
copy_backward ()	Копирует последовательность, начиная с последнего элемента
swap ()	Меняет местами два элемента
iter_swap ()	Меняет местами два элемента, на которые указывают итераторы
swap_ranges ()	Меняет местами элементы двух последовательностей
replace ()	Заменяет элементы с указанным значением
replace_if ()	Заменяет элементы при выполнении предиката
replace_copy ()	Копирует последовательность, заменяя элементы с указанным значением
replace_copy_if ()	Копирует последовательность, заменяя элементы при выполнении предиката
fill ()	Заменяет все элементы данным значением

Модифицирующие операции с последовательностями (§ 18.6) <algorithm> *(продолжение)*

fill_n ()	Заменяет первые *n* элементов указанным значением
generate ()	Заменяет все элементы результатом операции
generate_n ()	Заменяет первые *n* элементов результатом операции
remove ()	Удаляет элементы с данным значением
remove_if ()	Удаляет элементы при выполнении предиката
remove_copy ()	Копирует последовательность, удаляя элементы с указанным значением
remove_copy_if ()	Копирует последовательность, удаляя элементы, удовлетворяющие предикату
unique ()	Удаляет равные соседние элементы
unique_copy ()	Копирует последовательность, удаляя равные соседние элементы
reverse ()	Меняет порядок следования элементов на обратный
reverse_copy ()	Копирует последовательность в обратном порядке
rotate ()	Перемещает элементы циклически
rotate_copy ()	Копирует элементы в циклической последовательности
random_shuffle ()	Перемещает элементы согласно случайному равномерному распределению («тасует» последовательность)

Всякий хороший проект несет на себе следы индивидуальных особенностей и интересов проектировщика. Контейнеры и алгоритмы стандартной библиотеки явно отражают тенденцию построения основы для работы с классическими алгоритмами и структурами данных. Стандартная библиотека обеспечивает не только минимум контейнеров и алгоритмов, необходимых практически каждому программисту, но также включает в себя многие программные средства, используемые этими алгоритмами и необходимые для расширения библиотеки за пределы этого минимума.

Упор здесь делается не на проектировании алгоритмов и даже не на их использовании (за исключением простейших и очевидных алгоритмов). По поводу проектирования и анализа алгоритмов следует обратиться к другой литературе (например, [Knuth, 1968] или [Tarjan, 1983]). В настоящей главе перечислены алгоритмы, предлагаемые стандартной библиотекой, и объяснено, как они выражаются на C++. Такая методика позволит разобравшемуся в алгоритмах программисту правильно использовать библиотеку и расширять ее по тем же приницпам, согласно которым она была построена.

Стандартная библиотека предоставляет большое число операций сортировки, поиска и манипулирования с каким-либо образом упорядоченными последовательностями:

Сортировка последовательностей (§ 18.7) <algorithm>

sort ()	Сортирует с хорошей средней эффективностью
stable_sort ()	Сортирует, сохраняя порядок следования равных элементов
partial_sort ()	Упорядочивает первую часть последовательности
partial_sort_copy ()	Копирует, упорядочивая первую часть результата
nth_element ()	Ставит на нужное место *n*-й элемент
lower_bound ()	Находит первое вхождение значения

Сортировка последовательностей (§ 18.7) <algorithm> *(продолжение)*	
upper_bound ()	Находит первый элемент, больший чем значение
equal_range ()	Находит подпоследовательность с данным значением
binary_search ()	Указанное значение есть в отсортированной последовательности?
merge ()	Объединяет две отсортированные последовательности
inplace_merge ()	Объединяет две последовательно отсортированные последовательности
partition ()	Перемещает вперед элементы, удовлетворяющие предикату
stable_partition ()	Перемещает вперед элементы, удовлетворяющие предикату, сохраняя их относительный порядок следования

Алгоритмы для работы с множествами (§ 18.7.5) <algorithm>	
includes ()	*true*, если последовательность является подпоследовательностью другой
set_union ()	Конструирует отсортированное объединение
set_intersection ()	Конструирует отсортированное пересечение
set_difference ()	Конструирует отсортированную последовательность элементов, входящих в первую, но не входящих во вторую последовательность
set_symmetric_difference ()	Конструирует отсортированную последовательность элементов, присутствующих лишь в одной из двух последовательностей

Операции с кучей поддерживают последовательность в состоянии, облегчающем сортировку (если в ней возникает необходимость):

Операции с кучей (§ 18.8) <algorithm>	
make_heap ()	Подготавливает последовательность к использованию в качестве кучи
push_heap ()	Добавляет в кучу элемент
pop_heap ()	Удаляет из кучи элемент
sort_heap ()	Сортирует кучу

Библиотека обеспечивает несколько основанных на сравнении алгоритмов для выбора элементов:

Минимумы и максимумы (§ 18.9) <algorithm>	
min ()	Меньшее из двух значений
max ()	Большее из двух значений
min_element ()	Наименьшее значение в последовательности
max_element ()	Наибольшее значение в последовательности
lexicographical_compare ()	Лексикографически первая из двух последовательностей

Наконец, библиотека обеспечивает перестановку элементов в последовательности:

Перестановки (§ 18.10) <algorithm>	
next_permutation ()	Следующая перестановка в лексикографическом порядке
prev_permutation ()	Предыдущая перестановка в лексикографическом порядке

Кроме того, несколько обобщенных численных алгоритмов определены в <*numeric*> (§ 22.6).

В описании алгоритмов важны имена параметров шаблона. ***In***, ***Out***, ***For***, ***Bi*** и ***Ran*** означают соответственно: итератор для чтения, итератор для записи, однонаправленный итератор, двунаправленный итератор и итератор с произвольным доступом (§ 19.2.1). ***Pred*** означает унарный предикат (условие), ***BinPred*** — бинарный предикат (§ 18.4.2), ***Cmp*** означает функцию сравнения (§ 17.1.4.1,§ 18.7.1), ***Op*** — унарную операцию, ***BinOp*** — бинарную операцию (§ 18.4). Обычно для аргументов шаблонов использовались гораздо более длинные имена, однако я нахожу, что даже после короткого знакомства со стандартной библиотекой длинные имена ухудшают, а не улучшают читабельность.

Итераторы с произвольным доступом могут использоваться как двунаправленные итераторы, двунаправленные итераторы — как однонаправленные, а однонаправленные итераторы — как итераторы для чтения или для записи (§ 19.2.1). Задание типа, не обеспечивающего требуемой операции, приведет к ошибке инстанцирования шаблона (§ B.13.7). Указание типа, имеющего нужную операцию, но с неверной семантикой, приведет к непредсказуемому поведению программы во время выполнения (§ 17.1.4).

18.3. Последовательности и контейнеры

Есть хороший универсальный принцип: то, что наиболее часто используется, должно быть самым коротким, легко выражаемым и безопасным. Стандартная библиотека нарушает этот принцип во имя универсальности. Для стандартной библиотеки универсальность важнее всего. Например, мы можем найти два первых вхождения числа *42* в списке так:

```
void f(list<int>& li)
{
    list<int>::iterator p = find(li.begin(), li.end(), 42);      // первое вхождение
    if (p != li.end()) {
        list<int>::iterator q = find(++p, li.end(), 42);         // второе вхождение
        //...
    }
    //...
}
```

Если бы функция ***find*** () была выражена как операция над контейнером, для поиска второго вхождения нам бы понадобился какой-то дополнительный механизм. Важно отметить, что обобщение такого «дополнительного механизма» на каждый контейнер и каждый алгоритм не так просто. Вместо этого стандартные библиотечные алгоритмы работают с последовательностями элементов. То есть на вход алгоритма подается пара итераторов, определяющих последовательность (последовательность — это как бы то, что находится между ними). Первый итератор ссылается на первый

элемент последовательности, а второй указывает на «место сразу же за последним элементом» (§ 3.8, § 19.2). Такая последовательность называется «полуоткрытой», так как она включает в себя первое из значений, на которые указывают итераторы, но не включает второе. Полуоткрытая последовательность позволяет выразить большинство алгоритмов, не выделяя в особый случай пустую последовательность.

Последовательность — особенно последовательность, допускающую произвольный доступ — часто называют *диапазоном*. Традиционные математические обозначения для полуоткрытого диапазона — [***первый***, ***последний***) или [***первый***, ***последний***[. Важно, что последовательностью могут быть элементы контейнера или подпоследовательность контейнера. С другой стороны, многие последовательности, такие как потоки ввода/вывода, не являются контейнерами. Однако алгоритмы, выраженные в терминах последовательностей, прекрасно работают и с ними.

18.3.1. Входные последовательности

Чтобы выразить понятие «все элементы контейнера *x*», обычно пишут ***x.begin*** (), ***x.end*** (). Тем не менее такое написание может привести к ошибке. Например, когда используется несколько итераторов, легко создать алгоритм с парой аргументов, не образующих последовательность:

```
void f(list<string>& fruit, list<string>& citrus)
{
    typedef list<string>::const_iterator LI;

    LI p1 = find(fruit.begin(), citrus.end(), "apple");      // неверно!
    LI p2 = find(citrus.begin(), citrus.end(), "pear");      // правильно
    // ...
    LI p3 = find(p1, p2, "peach");                           // неверно!
    // ...
}
```

В приведенном выше примере две ошибки. Первая очевидна (если подозреваешь, что она есть), но компилятору выявить ее нелегко. Вторую нелегко рассмотреть в реальной программе даже опытному программисту. Сократив число явно используемых итераторов, мы облегчим задачу. Здесь я очерчу подход к проблеме, основанный на том, чтобы сделать понятие входной последовательности явным. Однако, чтобы удержать разговор о стандартных контейнерах строго в рамках стандартной библиотеки, далее в этой главе я не стану прибегать к явным входным последовательностям.

Ключевая идея состоит в явном указании того, что на входе — последовательность. Например:

```
template<class In, class T> In find(In first, In last, const T& v)      // стандарт
{
    while (first!=last && *first!=v) ++first;
    return first;
}

template<class In, class T> In find(Iseq<In> r, const T& v)             // расширение
{
    return find(r.first, r.second, v);
}
```

Вообще говоря, когда используется аргумент ***Iseq***, перегрузка (§ 13.3.2) позволяет предпочесть версию алгоритма со входной последовательностью.

Естественно, входная последовательность реализуется парой (§ 17.4.1.2) итераторов:

```
template<class In> struct Iseq : public pair<In, In> {
    Iseq (In i1, In i2) : pair<In, In> (i1, i2) { }
};
```

Мы могли бы явно сделать так, чтобы для вызова второй версии ***find*** (), требовалась входная последовательность:

```
LI p = find (Iseq<LI> (fruit.begin (), fruit.end ()), "apple");
```

Однако, это еще утомительнее, чем прямо вызывать изначальную ***find*** (). Задачу облегчит простая функция-помощник. В частности, ***Iseq*** (входная последовательность) контейнера — это последовательность элементов от его начала (***begin*** ()) до конца (***end*** ()):

```
template<class C> Iseq<C::iterator> iseq (C& c)        // для контейнера
{
    return Iseq<C::iterator> (c.begin (), c.end ());
}
```

Это позволяет нам выразить алгоритм для контейнеров компактно и без повторений. Например:

```
void f (list<string> & ls)
{
    list<string>::iterator p = find (ls.begin (), ls.end (), "стандарт");
    list<string>::iterator q = find (iseq (ls), "расширение");
}
```

Нетрудно создать версию ***iseq*** (), производящую входные последовательности для массивов, потоков ввода и т. п. (§ 18.13[6]).

Основной выгодой от введения ***Iseq*** является то, что это делает понятие входной последовательности явным. Непосредственный практический эффект здесь заключается в том, что ***iseq*** () устраняет большую часть утомительных и ведущих к ошибкам повторений, необходимых, чтобы выразить входные последовательности в виде пары итераторов.

Полезно также и понятие выходной последовательности, однако оно не столь просто и не так нужно, как входная последовательность (§ 18.13[7]; см. также § 19.2.4).

18.4. Объекты-функции

Многие алгоритмы работают с последовательностями при помощи только итераторов и значений. Например, функцией ***find*** () мы можем найти в последовательности первый элемент со значением *7* следующим образом:

```
void f (list<int> & c)
{
    list<int>::iterator p = find (c.begin (), c.end (), 7);
    //...
}
```

Чтобы сделать что-нибудь поинтереснее, мы можем захотеть применить эти алгоритмы для выполнения какого-то своего кода (§ 3.8.4). Например, мы можем найти в последовательности первый элемент со значением меньшим, чем 7, так:

```
bool less_than_7 (int v)
{
    return v<7;
}

void f (list<int> & c)
{
    list<int>::iterator p = find_if (c.begin (), c.end (), less_than_7);
}
```

Функции, заданные в качестве аргумента, могут быть самыми разнообразными: логические предикаты, арифметические операции, операции извлечения информации из элементов и т. п. Писать отдельные функции для каждого случая неудобно и неэффективно. Также нет функции логически достаточной, чтобы выразить все, что мы хотим. Зачастую функция, вызываемая для каждого элемента, нуждается в сохранении данных между вызовами и в возвращении результата многих обращений. Функция-член класса служит таким нуждам лучше, чем обычная функция, поскольку сохранять данные может объект соответствующего класса. Кроме того, класс в состоянии обеспечить операции для инициализации и извлечения таких данных.

Давайте рассмотрим, как написать функцию — или, точнее, функции-подобный класс — для вычисления суммы:

```
template<class T> class Sum {
    T res;
public:
    Sum (T i = 0) : res (i) { }                    // инициализация
    void operator () (T x) { res += x; }           // накопление
    T result () const { return res; };             // возвращение суммы
};
```

Ясно, что класс **Sum** предназначен для арифметических типов, для которых определены инициализация нулем и оператор +=. Например:

```
void f (list<double>& ld)
{
    Sum<double> s;
    s = for_each (ld.begin (), ld.end (), s);      // вызов s() для каждого элемента ld
    cout << "сумма равна " << s.result () << '\n';
}
```

Здесь ***for_each*** () (§ 18.5.1) вызывает ***Sum<double>::operator*** () (***double***) для каждого элемента ***ld***.

Суть заключается в том, что ***for_each*** () в действительности не считает, что ее третьим аргументом должна быть функция. Она полагает , что ее третий аргумент — нечто такое, что можно вызвать с соответствующим аргументом. Подходящим образом определенный объект может подойти тут так же, как функция — а иногда и лучше. Например, встроить (***inline***) обращение к оператору () класса легче, чем функцию,

переданную по указателю. Поэтому объекты-функции зачастую выполняются быстрее, чем обычные функции. Объект какого-либо класса с оператором () (§ 11.9) называется *объектом-функцией*, *функтором*, *объектом типа функция* или *функциональным объектом*.

18.4.1. Базовые классы для объектов-функций

Стандартная библиотека предоставляет множество полезных объектов-функций. Для помощи в написании объектов-функций библиотека предусматривает несколько базовых классов:

```
template<class Arg, class Res> struct unary_function {
        typedef Arg argument_type;
        typedef Res result_type;
};

template<class Arg, class Arg2, class Res> struct binary_function {
        typedef Arg first_argument_type;
        typedef Arg2 second_argument_type;
        typedef Res result_type;
};
```

Назначение этих классов — дать стандартные имена типам аргументам и возвращаемых значений для использования в классах, производных от ***unary_function*** и ***binary_function***. Используя эти базовые классы должным образом (так, как это делает стандартная библиотека), программист избавится от долгих мучений, которые все равно рано или поздно приведут к осознанию необходимости правильного применения базовых классов (§ 18.4.4.1).

18.4.2. Предикаты

Предикат — это объект-функция (или просто функция), возвращающая значение типа ***bool***. Например, заголовочный файл ***<functional>*** содержит такие определения:

```
template<class T> struct logical_not : public unary_function<T, bool> {
        bool operator () (const T& x) const { return !x; }
};

template<class T> struct less : public binary_function<T, T, bool> {
        bool operator () (const T& x, const T& y) const { return x<y; }
};
```

Унарные и бинарные предикаты часто бывают полезны в сочетании с алгоритмами. Например, мы можем сравнить две последовательности в поисках элемента первой последовательности, который был бы не меньше соответствующего элемента во второй последовательности:

```
void f(vector<int> & vi, list<int> & li)
{
        typedef list<int>::iterator LI;
        typedef vector<int>::iterator VI;
        pair<VI, LI> p1 = mismatch (vi.begin (), li.begin (), less<int> ());
        //...
}
```

Алгоритм ***mismatch*** () (несоответствие) многократно применяет свой бинарный предикат к парам соответствующих элементов до первой неудачи (§ 18.5.4). Затем он возвращает итераторы для элементов, не удовлетворивших условию. Поскольку нужен объект, а не тип, используется ***less<int>*** () (со скобками), а не соблазнительное ***less<int>***.

Вместо поиска первого элемента *не меньшего*, чем соответствующий элемент в другой последовательности, мы могли бы захотеть найти первый элемент, *меньший* чем соответствующий элемент в другой последовательности. Мы можем сделать это, отыскивая первую пару, которая не удовлетворяет дополнительному предикату *greater_equal* (больше или равно):

```
p1 = mismatch (vi.begin (), vi.end (), li.begin (), greater_equal<int> ());
```

или представив алгоритму ***mismatch*** () последовательности в обратном порядке и использовав ***less_equal***:

```
pair<LI, VI> p2 = mismatch (li.begin (), li.end (), vi.begin (), less_equal<int> ());
```

В § 18.4.4.4 я покажу, как выразить предикат «не меньше».

18.4.2.1. Обзор предикатов

В заголовочном файле ***<functional>*** стандартная библиотека предоставляет несколько постоянно используемых предикатов:

Предикаты <functional>

equal_to	Бинарный	*arg1* == *arg2*
not_equal_to	Бинарный	*arg1* != *arg2*
greater	Бинарный	*arg1* > *arg2*
less	Бинарный	*arg1* < *arg2*
greater_equal	Бинарный	*arg1* >= *arg2*
less_equal	Бинарный	*arg1* <= *arg2*
logical_and	Бинарный	*arg1* && *arg*
logical_or	Бинарный	*arg1* || *arg2*
logical_not	Унарный	!*arg*

Определения *less* и ***logical_not*** даны в § 18.4.2.

Кроме библиотечных предикатов пользователь может написать собственные. Пользовательские предикаты необходимы для простого и изящного использования стандартных библиотек контейнеров и алгоритмов. В частности, возможность определять предикаты важна, когда нужно воспользоваться алгоритмами для классов, разработанных не на основе стандартной библиотеки и ее алгоритмов. Например, рассмотрим вариант класса ***Club*** из § 10.4.6:

```
class Person { /* ... */ };

struct Club {
    string name;
    list<Person*> members;      // члены клуба
    list<Person*> officers;     // персонал
    // ...
    Club (const string& n);
};
```

Поиск объекта типа *Club* с данным именем *name* в списке *list<Club>* — явно осмысленная вещь (это поиск клуба с данным названием). Однако стандартный алгоритм *find_if* () ничего не знает о типе *Club*. Библиотечные алгоритмы умеют проверять на равенство, но нам не нужно искать объект типа *Club* по его полному значению. Мы хотим использовать в качестве ключа поиска *Club::name*. И чтобы это отразить, мы напишем предикат:

```
class Club_eq : public unary_function<Club, bool> {
        string s;
public:
        explicit Club_eq (const string& ss) : s (ss) { }
        bool operator () { const Club& c) const { return c.name==s; }
};
```

Определить полезные предикаты не трудно. А как только они введены для типов, определяемых пользователем, применение этих типов со стандартными алгоритмами становится так же просто и эффективно, как и в примере с контейнерами простых типов. Например:

```
void f (list<Club>& lc)
{
        typedef list<Club>::iterator LCI;
        LCI p = find_if (lc.begin (), lc.end (), Club_eq ("Философы за обедом"));
        // ...
}
```

18.4.3. Арифметические объекты-функции

Когда имеешь дело с числовыми классами, иногда полезно иметь стандартные арифметические функции, доступные как объекты-функции. Поэтому в заголовочном файле *<functional>* стандартная библиотека предоставляет арифметические операции:

Арифметические операции <functional>		
plus	Бинарный	*arg1 + arg2*
minus	Бинарный	*arg1 – arg2*
multiplies	Бинарный	*arg1 * arg2*
divides	Бинарный	*arg1 / arg2*
modulus	Бинарный	*arg1 % arg2*
negate	Унарный	*– arg1*

Мы могли бы при помощи *multiplies* перемножить элементы двух векторов, тем самым получив третий:

```
void discount (vector<double>& a, vector<double>& b, vector<double>& res)
{
        transform (a.begin (), a.end (), b.begin (),
                back_inserter (res), multiplies<double> ());
}
```

Функция *back_inserter* () описана в § 19.2.4. Несколько численных алгоритмов можно найти в § 22.6.

18.4.4. Связыватели, адаптеры и отрицатели

Мы можем пользоваться предикатами и арифметическими функциями, которые сами написали, а также предикатами и арифметическими функциями из стандартной библиотеки. Однако когда нужен новый предикат, мы часто обнаруживаем, что он лишь незначительно отличается от уже существующего. Стандартная библиотека поддерживает композицию объектов-функций:

§ 18.4.4.1 *Связыватель* (binder) позволяет использовать объект-функцию с двумя аргументами как функцию с одним аргументом путем связывания одного аргумента со значением.

§ 18.4.4.2 *Адаптер функций-членов* позволяет использовать функции-члены как аргументы алгоритмов.

§ 18.4.4.3 *Адаптер указателя на функцию* позволяет использовать указатель на функцию как аргумент алгоритма.

§ 18.4.4.4 *Отрицатель* позволяет нам выразить противоположный предикат.

Все вместе эти объекты-функции называют *адаптерами*. Все они имеют общую структуру, опирающуюся на базовые для объектов-функций классы ***unary_function*** и ***binary_function*** (§ 18.4.1). Для каждого из этих адаптеров существует функция-помощник, принимающая объект-функцию в качестве аргумента и возвращающая другой объект-функцию. Вызванный его оператором, этот объект-функция выполнит желаемое действие. То есть адаптер — это простая форма функции высшего порядка: он берет функцию-аргумент и производит из нее новую функцию.

Связыватели, адаптеры и отрицатели <functional>

Функция	Тип объекта	Действие объекта (operator () ())
bind2nd (*y*)	*bind2nd*	Вызывает бинарную функцию с *y* в качестве второго аргумента
bind1st (*x*)	*bind1st*	Вызывает бинарную функцию с *x* в качестве первого аргумента
mem_fun ()	*mem_fun_t*	Вызывает 0-аргументную функцию-член через указатель
	mem_fun1_t	Вызывает унарную функцию-член через указатель
	const_mem_fun_t	Вызывает 0-аргументную константную функцию-член через указатель
	const_mem_fun1_t	Вызывает унарную константную функцию-член через указатель
mem_fun_ref ()	*mem_fun_ref_t*	Вызывает 0-аргументную функцию-член через ссылку
	mem_fun1_ref_t	Вызывает унарную функцию-член через ссылку
	const_mem_fun_ref_t	Вызывает 0-аргументную константную функцию-член через ссылку
	const_mem_fun1_ref_t	Вызывает унарную константную функцию-член через ссылку

Связыватели, адаптеры и отрицатели <functional> *(продолжение)*

Функция	Тип объекта	Действие объекта (operator () ())
ptr_fun ()	*pointer_to_unary_function*	Вызывает унарный указатель на функцию
ptr_fun ()	*pointer_to_binary_function*	Вызывает бинарный указатель на функцию
not1 ()	*unary_negate*	Заменяет унарный предикат на противоположный
not2 ()	*binary_negate*	Заменяет бинарный предикат на противоположный

18.4.4.1. Связыватели

Бинарные предикаты, такие как *less* (§ 18.4.2), полезны и достаточно гибки. Однако мы скоро обнаружим, что самый полезный вид предиката — тот, который последовательно сравнивает фиксированный аргумент с элементами контейнера. Типичным примером является функция *less_than_7* () (§ 18.4). Операция *less* при каждом обращении требует двух явно указанных аргументов, поэтому ее нельзя применить непосредственно. Нам придется определить:

```
template<class T> class less_than : public unary_function<T, bool> {
        T arg2;
public:
        explicit less_than (const T& x) : arg2 (x) { }
        bool operator () (const T& x) const { return x<arg2; }
};
```

Теперь мы можем написать:

```
void f (list<int>& c)
{
        list<int>::const_iterator p = find_if (c.begin (), c.end (), less_than<int> (7));
        // ...
}
```

Мы должны писать *less_than<int>* (*7*), а не *less_than* (*7*), потому что аргумент шаблона *<int>* не может быть выведен из типа аргумента конструктора (*7*) (§ 13.3.1).

Предикат *less_than* полезен в очень многих случаях. Важно то, что мы определили его, зафиксировав или связав второй аргумент предиката *less*. Такое построение со связыванием аргумента настолько употребительно, полезно и порой утомительно, что стандартная библиотека предоставляет для выполнения этой задачи стандартный класс:

```
template<class BinOp>
class binder2nd
        : public unary_function<typename BinOp::argument_type,
                                typename BinOp::result_type> {
protected:
        BinOp op;
        typename BinOp::second_argument_type arg2;
public:
        binder2nd (const BinOp& x, const typename BinOp::second_argument_type& v)
                : op (x), arg2 (v) { }
```

```
        result_type operator () (const argument_type& x) const { return op (x, arg2); }
};

template<class BinOp, class T> binder2nd<BinOp> bind2nd (const BinOp& op, const T& v)
{
        return binder2nd<BinOp> (op, v);
}
```

Например, мы можем воспользоваться ***bind2nd*** () для создания унарного предиката «меньше чем 7» из бинарного предиката «меньше» и значения 7:

```
void f (list<int>& c)
{
        list<int>::const_iterator p = find_if (c.begin (), c.end (),
                bind2nd (less<int> (), 7));
        // ...
}
```

Читаемо? Эффективно? Даже в среднего качества реализации C++ этот код будет эффективнее по быстродействию и памяти, чем первоначальная версия, использующая функцию ***less_than_7*** () из § 18.4! Сравнение легко встраивается.

Эта система обозначений логична, но к ней надо привыкнуть. Как правило, определение именованной операции со связанным аргументом в конце концов оправдывает все затраты:

```
template<class T> struct less_than : public binder2nd<less<T> > {
        explicit less_than (const T& x) : binder2nd<less<T> > (less<T> (), x) { }
};

void f (list<int>& c)
{
        list<int>::const_iterator p = find_if (c.begin (), c.end (),
                less_than<int> (7));
        // ...
}
```

Важно определить ***less_than*** в терминах ***less***, а не прямым использованием оператора <. Таким образом, ***less_than*** получит все преимущества любой специализации, которая может быть выполнена с ***less*** (§ 13.5, § 19.2.2).

По аналогии с ***bind2nd*** () и ***binder2nd*** () заголовочный файл ***<functional>*** предоставляет ***bind1st*** () и ***binder1st*** для связывания первого аргумента бинарной функции.

Связывание аргумента, ***bind1st*** () и ***bind2nd*** () выполняет услугу, которую обычно называют *Currying* (*Карринг*, по имени математика H. B. Curry).

18.4.4.2. Адаптеры функций-членов

Большинство алгоритмов обращаются к стандартным или пользовательским операциям. Естественно, пользователи часто хотят вызвать функцию-член класса. Например:

```
void draw_all (list<Shape*>& c)
{
        for_each (c.begin (), c.end (), &Shape::draw);      // ошибка!
}
```

Трудность в том, что функцию-член ***mf*** () нужно вызывать для какого-нибудь объекта: ***p->mf*** (). Однако алгоритмы, такие как ***for_each*** (), вызывают свои функции-операнды простым применением скобок: ***f*** (). Следовательно, нам нужен удобный и эффективный способ для создания чего-то такого, что позволит алгоритму вызвать функцию-член. Альтернативой было бы дублирование алгоритмов: одна версия для функций-членов и одна для обычных функций. Хуже того, нам могут понадобиться дополнительные версии алгоритмов для контейнеров с объектами (а не с указателями на объекты). Что касается связывателей (§ 18.4.4.1), эта проблема решается введением функции в класс. Сначала рассмотрим типичный случай, когда мы хотим вызвать функцию-член без аргументов для элементов контейнера с указателями:

```
template<class R, class T> class mem_fun_t : public unary_function<T*, R> {
        R (T::*pmf) ();
public:
        explicit mem_fun_t (R (T::*p) ()) :pmf (p) { }
        R operator () (T* p) const { return (p->*pmf) (); }          // вызов через указатель
};

template<class R, class T> mem_fun_t<R, T> mem_fun (R (T::*f) ())
{
        return mem_fun_t<R, T> (f);
}
```

Этот помогает нам правильно записать пример с ***Shape::draw*** ():

```
void draw_all (list<Shape*>& lsp)        // вызов безаргументной функции-члена
                                         // через указатель
{
        for_each (lsp.begin (), lsp.end (), mem_fun (&Shape::draw));    // рисуем все фигуры
}
```

Кроме того нам нужен некий класс и функция ***mem_fun*** () для обработки функции-члена с аргументом. Нам также нужны версии для прямого (а не через указатель) вызова объекта; они называются ***mem_fun_ref*** (). Наконец, нам нужны версии для константных функций-членов:

```
template<class R, class T>  mem_fun_t <R, T> mem_fun (R (T::*f) ()); // и версии для унарных,
                        // константных и константных унарных
                        // функций-членов (см. таблицу в § 18.4.4)

template<class R, class T>  mem_fun_ref_t <R, T> mem_fun_ref (R (T::*f) ());  // и версии для
                        // унарных, константных и константных унарных
                        // функций-членов (см. таблицу в § 18.4.4)
```

Получив из <***functional***> эти адаптеры функций-членов, мы можем написать:

```
void f (list<string> ls)  // использование безаргументной функции-члена для объекта
{
        typedef list<string>::iterator LSI;
        // поиск ""
        LSI p = find_if (ls.begin (), ls.end (), mem_fun_ref (&string::empty));
}

void rotate_all (list<Shape*>& ls, int angle)
```

```
        // использование функции-члена с одним аргументом через указатель на объект
{
        for_each (ls.begin (), ls.end (), bind2nd (mem_fun (&Shape::rotate), angle));
}
```

Стандартной библиотеке не приходится иметь дело с функциями-членами более чем одного аргумента, поскольку ни один из стандартных библиотечных алгоритмов не принимает в качестве операнда функцию более чем от двух аргументов.

18.4.4.3. Указатели на адаптеры функций

Алгоритму нет дела, является ли «аргумент типа функции» функцией, указателем на функцию или объектом-функцией. Однако связывателю (§ 18.4.4.1) это не безразлично, поскольку ему нужно хранить копию аргумента для дальнейшего использования. Поэтому стандартная библиотека дает нам два адаптера, чтобы позволить использовать указатели на функции со стандартными алгоритмами из *<functional>*. Определение и реализация очень напоминают определение и реализацию адаптеров функций-членов (§ 18.4.4.2). Снова используется пара функций и пара классов:

```
template<class A, class R>
        pointer_to_unary_function<A, R> ptr_fun (R (*f) (A));
template<class A, class A2, class R>
        pointer_to_binary_function<A, A2, R> ptr_fun (R (*f) (A, A2));
```

Получив эти адаптеры указателей на функцию, мы можем воспользоваться обычными функциями вместе со связывателями:

```
class Record {/* ... */};

bool name_key_eq (const Record&, const char*);      // сравнение, основанное на именах
bool ssn_key_eq (const Record&, const long);        // сравнение, основанное на числах

void f (list<Record> & lr)      // использование указателя на функцию
{
        typedef list<Record> ::iterator LI;
        LI p = find_if (lr.begin (), lr.end (), bind2nd (ptr_fun (name_key_eq), "Джон Браун"));
        LI q = find_if (lr.begin (), lr.end (), bind2nd (ptr_fun (ssn_key_eq), 1234567890));
// ...
}
```

Так в списке *lr* ищутся элементы, соответствующие ключам *"Джон Браун"* и *1234567890*.

18.4.4.4. Отрицатели

Предикатные отрицатели сродни связывателям в том, что они принимают операцию и производят из нее другую операцию. Определение и реализация отрицателей следуют модели адаптеров функций-членов (§ 18.4.4.2). Их определения тривиальны, но окончательной простоте мешают длинные стандартные имена:

```
template<class Pred>
class unary_negate : public unary_function<typename Pred::argument_type, bool> {

        Pred op;
```

```
public:
        explicit unary_negate (const Pred& p) : op (p) { }
        bool operator () (const argument_type& x) const { return !op (x); }
};

template<class Pred>
class binary_negate : public binary_function<typename Pred::first_argument_type,
            typename Pred::second_argument_type, bool> {
        typedef first_argument_type Arg;
        typedef second_argument_type Arg2;

        Pred op;
public:
        explicit binary_negate (const Pred& p) : op (p) { }
        bool operator () (const Arg& x, const Arg2& y) const
            { return !op (x, y); }
};

template<class Pred> unary_negate<Pred> not1 (const Pred& p);
// заменяет унарный предикат на противоположный
template<class Pred> binary_negate<Pred> not2 (const Pred& p);
// заменяет бинарный предикат на противоположный
```

Эти классы и функции объявлены в *<functional>*. Имена ***first_argument_type***, ***second_argument_type*** и т. п. взяты из стандартных базовых классов ***unary_function*** и ***binary_function***.

Как и связыватели, отрицатели очень удобно использовать косвенно через их вспомогательные функции. Например, мы можем описать бинарный предикат «не меньше чем» и использовать его для поиска первой пары соответствующих элементов, чей первый элемент больше или равен второму:

```
void f (vector<int> & vi, list<int> & li)          // пересмотренный пример из § 18.4.2
{
        // ...
        p1 = mismatch (vi.begin (), vi.end (), li.begin (), not2 (less<int> ()));
        // ...
}
```

То есть ***p1*** обозначает первую пару элементов, для которых не выполнился предикат «не меньше».

Предикаты имеют дело с логическими условиями, поэтому здесь нет эквивалентов для битовых операций |, &, ^ и ~.

Естественно, связыватели, адаптеры и отрицатели часто сочетают. Например:

```
extern "C" int strcmp (const char*, const char*);      // из <stdlib>

void f (list<char*> & ls)                    // используется указатель на функцию
{
        typedef typename list<char*> ::const_iterator LI;
        LI p = find_if (ls.begin (), ls.end (), not1 (bind2nd (ptr_fun (strcmp), "забавно")));
}
```

Так находится элемент списка ***ls***, содержащий С-строку ***"забавно"***. Отрицатель нужен из-за того, что ***strcmp*** () возвращает ***0*** при равенстве строк.

18.5. Алгоритмы, не модифицирующие последовательность

Немодифицирующие алгоритмы являются основным средством для поиска чего-либо в последовательности без написания цикла. Кроме того, они позволяют получить необходимую информацию об элементах. Эти алгоритмы могут принимать в качестве аргументов константные итераторы (§ 19.2.1) и — за исключением ***for_each*** () — не должны вызывать операции, изменяющие элементы последовательности.

18.5.1. Алгоритмы for_each

Мы пользуемся библиотеками, чтобы воспользоваться работой, уже проделанной кем-то. Использование библиотечных функций, классов, алгоритмов и т. п. избавляет нас от лишнего придумывания, проектирования, написания, отладки и документирования. Использование стандартной библиотеки облегчает восприятие программы другим человеком, знакомым с библиотекой — ему не придется долго разбираться в вашем коде «собственного приготовления».

Основная выгода стандартных библиотечных алгоритмов заключается в том, что они избавляют программиста от написания явных циклов. Циклы могут быть утомительны и чреваты ошибками. Алгоритм ***for_each*** () — простейший в том смысле, что он ничего не делает кроме того, что устраняет необходимость в явном цикле. Он просто вызывает свой операторный аргумент для последовательности:

```
template<class In, class Op> Op for_each (In first, In last, Op f)
{
    while (first != last) f (*first++);
    return f;
}
```

Какие функции имеет смысл вызывать таким образом? Если вы хотите накапливать информацию от элементов, подумайте об ***accumulate*** () (§ 22.6). Если вам надо найти что-нибудь в последовательности, рассмотрите применение ***find*** () и ***find_if*** () (§ 18.5.2). Если вы изменяете или удаляете элементы, используйте ***replace*** () (§ 18.6.4) или ***remove*** () (§ 18.6.5). Вообще, прежде чем прибегать к ***for_each*** (), подумайте, нет ли более специализированного алгоритма, который больше вам подойдет.

Результатом ***for_each*** () является функция или функциональный объект, передаваемый в качестве третьего аргумента. Как показано в примере ***Sum*** (§ 18.4), это позволяет вернуть информацию в вызывающую функцию.

Одно из обычных применений ***for_each*** () — извлечение информации из элементов последовательности. Например, рассмотрим сбор имен неизвестного числа объектов ***Club***:

```
void extract (const list<Club> & lc, list<Person*> & off)
// размещаем персонал из 'lc' в 'off'
{
    for_each (lc.begin (), lc.end (), Extract_officers (off));
}
```

Подобно примеру из § 18.4 и § 18.4.2 мы определяем класс-функцию, которая извлекает желаемую информацию. В этом случае имена, которые будут выданы, находятся в списках *list<Person*>* в нашем списке *list<Club>*. Следовательно, ***Extract_officers*** должна скопировать в наш список весь персонал (***officers***) из списков ***officers*** объектов ***Club***:

```
class Extract_officers {
    list<Person*> & lst;
```

```
public:
    explicit Extract_officers (list<Person*>& x) : lst (x) { }

    void operator () (const Club& c) const
        { copy (c.officers.begin (), c.officers.end (), back_inserter (lst)); }
};
```

Теперь мы можем распечатать имена, снова воспользовавшись ***for_each*** ():

```
void extract_and_print (const list<Club> & lc)
{
    list<Person*> off;
    extract (lc, off);
    for_each (off.begin (), off.end (), Print_name (cout));
}
```

Написание ***Print_name*** оставлено в качестве упражнения (§ 18.13[4]).

Алгоритм ***for_each*** () классифицируется как немодифицирующий, поскольку он явно не изменяет последовательность. Тем не менее, будучи применен к неконстантной последовательности, ***for_each*** () (через третий аргумент) может изменять ее элементы. См., например, использование ***negate*** () в § 11.9.

18.5.2. Семейство find

Алгоритмы ***find*** () просматривают последовательность или две последовательности в поисках значения или соответствия некому предикату. Простейшая версия ***find*** () ищет конкретное значение или производит поиск на соответствие предикату:

```
template<class In, class T> In find (In first, In last, const T& val);
template<class In, class Pred> In find_if (In first, In last, Pred p);
```

Алгоритмы ***find*** () и ***find_if*** () возвращают итератор для первого элемента, удовлетворяющего значению или предикату соответственно. Фактически, ***find*** () можно рассматривать как версию ***find_if*** () с предикатом ==. Почему бы обе эти функции было не назвать ***find*** ()? Причина в том, что перегрузка функций не всегда различает вызовы двух шаблонных функций с одинаковым числом аргументов. Рассмотрим такой пример:

```
bool pred (int);

void f (vector<bool (*f) (int)>& v1, vector<int>& v2)
{
    find (v1.begin (), v1.end (), pred);        // находим pred
    find_if (v2.begin (), v2.end (), pred);     // находим целое, для которого
                                                // pred() возвращает true
}
```

Если бы ***find*** () и ***find_if*** () имели одно и то же имя, возникла бы довольно интересная двусмысленность. Вообще, суффикс ***_if*** используют для обозначения того, что алгоритм работает с предикатом.

Алгоритм ***find_first_of*** () находит в последовательности первый элемент, который имеется и во второй последовательности:

```
template<class For, class For2>
    For find_first_of(For first, For last, For2 first2, For2 last2);
template<class For, class For2, clas BinPred>
    For find_first_of(For first, For last, For2 first2, For2 last2, BinPred p);
```

Например:

```
int x[] = { 1, 3, 4 };
int y[] = { 0, 2, 3, 4, 5 };
void f()
{
    int* p = find_first_of(x, x+3, y, y+5);      // p = &x[1]
    int* q = find_first_of(p+1, x+3, y, y+5);    // q = &x[2]
}
```

Указатель *p* укажет на *x*[*1*], потому что *3* — первый элемент из *x*, имеющийся также в *y*. Подобным образом *q* укажет на *x*[*2*].

Алгоритм ***adjacent_find*** () найдет пару соседних совпадающих значений:

```
template<class For> For adjacent_find(For first, For last);
template<class For, class BinPred> For adjacent_find(For first, For last, BinPred p);
```

Возвращаемым значением является итератор на первый совпадающий элемент. Например:

```
void f(vector<string>& text)
{
    vector<string>::iterator p = adjacent_find(text.begin(), text.end());
    if (p!=text.end() && *p=="the") {      // я снова продублировал «the»!
        text.erase(p);
        // ...
    }
}
```

18.5.3. Алгоритмы count

Алгоритмы ***count*** () и ***count_if*** () считают число вхождений в последовательность некоторых значений:

```
template<class In, class T>
    typename iterator_traits<In>::difference_type count(In first, In last, const T& val);
template<class In, class Pred>
    typename iterator_traits<In>::difference_type count_if(In first, In last, Pred p);
```

Интересен возвращаемый алгоритмом ***count*** () тип. Рассмотрим очевидную и довольно наивную версию ***count*** ():

```
template<class In, class T> int count(In first, In last, const T& val)
{
    int res = 0;
    while (first != last) if (*first++ == val) ++res;
    return res;
}
```

Проблема в том, что *int* может быть неподходящим типом для результата. На машине с маленьким *int* может быть слишком много элементов в какой-нибудь последовательности, чтобы значение *count* () уместилось в *int*. И наоборот, на конкретной машине может быть выполнена более быстродействующая реализация, если счетчик будет иметь тип *short*.

Ясно, что число элементов в последовательности не может быть больше, чем максимальная разность между ее итераторами (§ 19.2.1). Следовательно, очевидный способ попытаться решить проблему — определить тип возвращаемого значения как:

```
typename In::difference_type
```

Однако, стандартный алгоритм должен быть приложим ко встроенным массивам, а также к стандартным контейнерам. Например:

```
void f(char* p, int size)
{
    int n = count (p, p+size, 'e');    // подсчет числа вхождений буквы 'e'
}
```

К сожалению, *const char**::*difference_type* не годится для C++. Эта проблема решается частичной специализацией *iterator_traits* (§ 19.2.2).

18.5.4. Равенство и несовпадение

Алгоритмы *equal* () (равенство) и *mismatch* () (несовпадение) сравнивают две последовательности:

```
template<class In, class In2> bool equal (In first, In last, In2 first2);

template<class In, class In2, class BinPred>
    bool equal (In first, In last, In2 first2, BinPred p);

template<class In, class In2>
    pair<In, In2> mismatch (In first, In last, In2 first2);

template<class In, class In2, class BinPred>
    pair<In, In2> mismatch (In first, In last, In2 first2, BinPred p);
```

Алгоритм *equal* () просто проверяет, равны ли соответствующие пары элементов в последовательностях; *mismatch* () ищет первую пару элементов, которые при сравнении не совпадают, и возвращает итераторы на эти элементы. Для второй последовательности конец не определен, то есть *last2* нет. Вместо этого считается, что во второй последовательности по крайней мере столько же элементов, сколько и в первой, и в качестве *last2* используется *first2* + (*last*–*first*). Этот прием широко применяется в стандартной библиотеке, где для операций над парами элементов используются пары последовательностей.

Как показано в § 18.5.1, эти алгоритмы даже полезнее, чем кажутся на первый взгляд, поскольку пользователь может создавать предикаты, определяющие, что считать равенством и совпадением. Отметим, что последовательности не обязаны быть одного типа. Например:

```
void f(list<int>& li, vector<double>& vd)
{
    bool b = equal (li.begin (), li.end (), vd.begin ());
}
```

Все, что здесь требуется — чтобы элементы последовательности были допустимы как операнды предикатов.

Две версии *mismatch* () отличаются только использованием предикатов. Фактически, мы можем реализовать их одной функцией с аргументом-шаблоном по умолчанию:

```
template<class In, class In2, class BinPred>
pair<In, In2> mismatch (In first, In last, In2 first2,
        BinPred p = equal_to<In::value_type> () )
        // § 18.4.2.1
{
        while (first != last && p (*first, *first2)) {
                ++first;
                ++first2;
        }
        return pair<In, In2> (first, first2);
}
```

Разницу между двумя функциями и одной с аргументом по умолчанию можно увидеть при использовании указателей на функции. Однако, если все множество вариантов стандартных алгоритмов считать просто «версией с предикатом по умолчанию», это на добрую половину сократит число шаблонных функций, которые надо запоминать.

18.5.5. Поиск

Алгоритмы *search* () и *search_n* () и *find_end* () находят одну последовательность как подпоследовательность другой:

```
template<class For, class For2>
        For search (For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
        For search (For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class For2>
        For find_end (For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
        For find_end (For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class Size, class T>
        For search_n (For first, For last, Size n, const T& val);

template<class For, class Size, class T, class BinPred>
        For search_n (For first, For last, Size n, const T& val, BinPred p);
```

Алгоритм *search* () ищет вторую последовательность-аргумент как подпоследовательность в первой. Если таковая найдена, возвращается итератор на первый совпадающий элемент в первой последовательности. Таким образом, возвращенное значение всегда находится в интервале [*first, last*]. Конец последовательности (*last*) возвращается для указания на отрицательные результаты поиска. Например:

```
string quote ("Зачем тратить время на обучение, если невежество дается даром?");

bool in_quote (const string& s)
{
        typedef string::const_iterator SCI;
```

```
        SCIp = search (quote.begin (), quote.end (), s.begin (), s.end ()); // находит в quote строку s
        return p != quote.end ();
}

void g ()
{
        bool b1 = in_quote ("учение");          // b1 = true
        bool b2 = in_quote ("лечение");         // b2 = false
}
```

Таким образом, ***search*** () — это операция для поиска подстроки, универсальная для всех последовательностей. Значит, ***search*** () — очень полезный алгоритм.

Алгоритм ***find_end*** () ищет вторую заданную последовательность (свой второй аргумент) как подпоследовательность первой. Если вторая последовательность найдена, ***find_end*** () возвращает итератор, указывающий на последнее совпадающий элемент в первой строке. Другими словами, ***find_end*** () — это ***search*** () «с остановкой в конце». Этот алгоритм находит последнее, а не первое вхождение второго аргумента в первой строке.

Алгоритм ***search_n*** () находит в последовательности подпоследовательность из по крайней мере ***n*** совпадений со значением аргумента ***value***. Возвращается итератор на первый элемент из ***n*** совпадений.

18.6. Алгоритмы, модифицирующие последовательность

Если вам нужно изменить последовательность, вы можете явно обратиться к ее элементам через итераторы. Затем вы можете изменить значение элемента. Однако, мы предпочитаем избегать такого программирования ради более простого и систематического стиля. Альтернативой являются алгоритмы, которые проходят по последовательности, выполняя указанные действия. Когда мы просто считываем элементы последовательности, этой цели служат немодифицирующие алгоритмы (§ 18.5). Изменяющие последовательность алгоритмы нужны для того, чтобы выполнять наиболее распространенные операции обновления данных. Некоторые алгоритмы обновляют последовательность, в то время как другие создают новую последовательность, основываясь на информации, найденной за время прохода.

Стандартные алгоритмы работают со структурами посредством итераторов. Это приводит к тому, что вставка нового или удаление старого элемента из контейнера вызывают трудности. Например, имея только итератор с указателем на элемент, как нам найти контейнер, из которого нужно удалить элемент? Не считая специальных итераторов (например, вставок, § 3.8, § 19.2.4), операции с итераторами не изменяют размеров контейнера. Вместо вставки и удаления элементов такие алгоритмы изменяют значения элементов, меняют элементы местами и копируют их. Даже операция ***remove*** () работает посредством переписывания элементов, которые нужно удалить (§ 18.6.5). Вообще говоря, фундаментальные изменяющие операции дают на выходе измененные копии входа. Те алгоритмы, которые фактически меняют последовательность, являются вариантами копирующих алгоритмов.

18.6.1. Копирование

Копирование — простейший способ произвести из одной последовательности другую. Определения базовых операций копирования тривиальны:

```
template<class In, class Out> Out copy (In first, In last, Out res)
{
    while (first != last) *res++ = *first++;
    return res;
}

template<class Bi, class Bi2> Bi2 copy_backward (Bi first, Bi last, Bi2 res)
{
    while (first != last) *--res = *--last;
    return res;
}
```

Результат копирующего алгоритма не обязан быть контейнером. Сгодится все, что можно описать итератором для записи (§ 19.2.6). Например:

```
void f(list<Club>& lc, ostream& os)
{
    copy (lc.begin (), lc.end (), ostream_iterator<Club> (os));
}
```

Чтобы прочитать последовательность, нам нужна последовательность, описывающая, где начать и где закончить. Чтобы что-либо записать в нее, нам нужен только итератор, описывающий куда писать. Однако, мы должны соблюдать осторожность, чтобы не произвести запись за конец последовательности. Одним из способов застраховаться от этого является использование вставки (§ 19.2.4), по мере надобности увеличивающей размер последовательности. Например:

```
void f(const vector<char> & vs, vector<char> & v)
{
    vector<char> v;

    copy (vs.begin (), vs.end (), v.begin ());          // можем выйти за конец v
    copy (vs.begin (), vs.end (), back_inserter (v));   // добавляем элементы из vs в конец v
}
```

Входная и выходная последовательности могут перекрываться. Мы пользуемся **copy** (), когда последовательности не перекрываются, или если конец выходной последовательности находится внутри входной. Мы используем **backward_copy** (), когда начало выходной последовательности находится внутри входной. Таким образом, никакие элементы не запишутся поверх других, пока последние не будут скопированы — см. также § 18.13[13].

Естественно, чтобы скопировать что-то по направлению от конца к началу, нам нужен двунаправленный итератор (§ 19.2.1) как для входной, так и для выходной последовательностей. Например:

```
void f(vector<char> & vc)
{
    copy_backward (vc.begin (), vc.end (), ostream_iterator<char> (cout));      // ошибка
    vector<char> v (vc.size ());

    copy_backward (vc.begin (), vc.end (), v.end ());                           // правильно
    copy (v.begin (), v.end (), ostream_iterator<char> (cout));                 // правильно
}
```

Часто нам нужно скопировать только элементы, отвечающие некоторому критерию. К сожалению, ***copy_if*** () как-то выпал из набора алгоритмов стандартной библиотеки (mea culpa[1]). С другой стороны, определить его очень просто:

```
template<class In, class Out, class Pred>
Out copy_if (In first, In last, Out res, Pred p)
{
    while (first != last) {
        if (p (*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

Теперь если мы захотим распечатать элементы со значением больше, чем ***n***, мы можем сделать это так:

```
void f (list<int>& ld, int n, ostream& os)
{
    copy_if (ld.begin (), ld.end (), ostream_itreator<int> (os), bind2nd (greater<int> (), n));
}
```

Посмотрите также ***remove_copy_if*** () (§ 18.6.5).

18.6.2. Алгоритмы transform

Несколько сбивает с толку, что ***transform*** () вовсе не обязательно изменяет (трансформирует) входную последовательность. Вместо этого ***transform*** () создает выход, который является преобразованием входа, основанным на определенной пользователем операции:

```
template<class In, class Out, class Op>
Out transform (In first, In last, Out res, Op op)
{
    while (first != last) *res++ = op (*first++);
    return res;
}

template<class In, class In2, class Out, class BinOp>
Out transform (In first, In last, In2 first2, Out res, BinOp op)
{
    while (first != last) *res++ = op (*first++, *first2++);
    return res;
}
```

Алгоритм ***transform*** (), который читает единственную последовательность, чтобы выдать на выход результат, напоминает ***copy*** (). Вместо записи элементов он записывает результат выполнения указанной операции над элементами. Таким образом, мы могли бы определить ***copy*** () как ***transform*** () с операцией, возвращающей свой аргумент:

```
template<class T> T identity (const T& x) { return x; }      //возвращает аргумент

template<class In, class Out> Out copy (In first, In last, Out res)
{
    return transform (first, last, res, identity <typename iterator_traits<In>::value_type>);
}
```

[1] «Моя вина» (лат.)

Явная квалификация ***identity*** понадобилась, чтобы получить конкретную функцию из шаблона. Для получения типа элемента ***In*** мы применили шаблон ***iterator_traits*** (§ 19.2.2).

Другой взгляд на ***transform*** () — рассмотреть его как вариант ***for_each***, который явно формирует свой выход. Например, мы можем при помощи ***transform*** () создать список имен типа ***string*** из списка элементов типа ***Club***:

```
string nameof(const Club& c)        // извлекает строку с именем
{
    return c.name;
}

void f(list<Club> & lc)
{
    transform (lc.begin (), lc.end (), ostream_iterator<string> (cout), nameof);
}
```

Одна из причин, почему алгоритм ***transform*** () был назван «transform» (изменить), заключается в том, что результат операции часто пишется туда же, откуда пришел аргумент. В качестве примера рассмотрим уничтожение объектов, на которые указывает набор указателей:

```
struct Delete_ptr {      // чтобы применить встраивание, используем объект-функцию
    template<class T> T* operator () (T* p) { delete p; return 0; }
};

void purge (deque<Shape*>& s)
{
    transform (s.begin (), s.end (), s.begin (), Delete_ptr ());
}
```

Алгоритм ***transform*** () всегда выдает выходную последовательность. Здесь я направил результат обратно во входную последовательность, так что ***Delete_ptr*** (***p***) производит тот же эффект, что и ***p = Delete_ptr*** (***p***). Вот почему я решил из ***Delete_ptr::operator***() () вернуть ***0***.

Алгоритм ***transform*** (), имеющий дело с двумя последовательностями, позволяет объединить информацию из двух источников. Например, при анимации может понадобиться процедура, которая меняла бы положение нескольких фигур, применяя операцию сдвига:

```
// shape — фигура, move shape — переместить фигуру

Shape* move_shape (Shape* s, Point p)  // *s += p
{
    s->move_to (s->center ()+p);
    return s;
}

void update_positions (list<Shape*>& ls, vector<Point>& oper)
{
    // вызов операции над соответствующим объектом:
    transform (ls.begin (), ls.end (), oper.begin (),
        ls.begin (), move_shape);
}
```

Нам на самом деле не нужно возвращать значение из ***move_shape*** (). Однако ***transform*** () настаивает на присваивании чему-то результата операции, поэтому я по-

зволил *move_shape* () вернуть свой первый операнд; тем самым я могу записать его туда, откуда получил.

Иногда у нас нет такой возможности. Например, операции, которые писал не я, и которые мне не хочется изменять, могут не возвращать значения. Иногда входная последовательность является константной. В таких случаях мы можем определить *for_each* () для двух последовательностей, чтобы он совпадал с *transform* () для двух последовательностей:

```
template<class In, class In2, class BinOp>
BinOp for_each (In first, In last, In2 first2, BinOp op)
{
    while (first != last) op (*first++, *first2++);
    return op;
}

void update_positions (list<Shape*>& ls, vector<Pointer>& oper)
{
    for_each (ls.begin (), ls.end (), oper.begin (), move_shape);
}
```

В других случаях может быть полезно иметь итератор для записи, который в действительности ничего не пишет (§ 19.6[2]).

Алгоритмов, которые читали бы три последовательности и более, в стандартной библиотеке нет. Впрочем, такие алгоритмы легко написать. Или можно многократно использовать *transform* ().

18.6.3. Алгоритмы unique

При сборе информации часто происходит дублирование данных. Алгоритмы *unique* () и *unique_copy* () удаляют соседние одинаковые значения:

```
template<class For> For unique (For first, For last);
template<class For, class BinPred> For unique (For first, For last, BinPred p);

template<class In, class Out> Out unique_copy (In first, In last, Out res);
template<class In, class Out, class BinPred>
    Out unique_copy (In first, In last, Out res, BinPred p);
```

Алгоритм *unique* () удаляет в последовательности соседние одинаковые элементы, *unique_copy* () создает копию без дубликатов. Например:

```
void f(list<string>& ls, vector<string>& vs)
{
    ls.sort ();                                   // сортировка списка (§ 17.2.2.1)
    unique_copy (ls.begin (), ls.end (), back_inserter (vs));
}
```

ls копируется в *vs*, и в процессе копирования одинаковые значения удаляются. Функция *sort* () нужна для того, чтобы одинаковые значения оказались соседними.

Как и другие стандартные алгоритмы, алгоритм *unique* () работает с итераторами. Он не знает тип контейнера, на который указывают эти итераторы, и потому не может модифицировать контейнер. Он может только изменять значение элементов.

Это приводит к тому, что *unique* () не удаляет одинаковые элементы во входной последовательности, как мы наивно могли бы предположить, а помещает уникальные элементы в начало последовательности и возвращает итератор на конец последовательности уникальных элементов:

```
template<class For> For unique (For list, For last)
{
        first = adjacent_find (first, last);          // § 18.5.2
        return unique_copy (first, last, first);
}
```

Элементы за «уникальной» последовательностью остаются неизменными. Поэтому в векторе одинаковые значения не удаляются:

```
void f(vector<string> & vs)                    // осторожно: опасный код!
{
        sort (vs.begin (), vs.end ());         // сортировка вектора
        unique (vs.begin (), vs.end ());       // удаляет одинаковые значения? нет!
}
```

Фактически, помещая последние элементы последовательности вперед, чтобы удалить одинаковые значения, *unique* () может ввести новые одинаковые значения. Например, результатом

```
int main ()
{
        char v[] = "abbcccde";

        char* p = unique (v, v+strlen (v));
        cout << v << ' ' << p-v << '\n';
}
```

будет

```
abcdecde 5
```

То есть *p* укажет на второе *c*.

Алгоритмы, которые могли бы удалить элементы (но не делают этого), в основном сводятся к двум формам: «простые» версии, перестраивающие элементы как *unique* (), и версии, создающие новую последовательность, похожие на *unique_copy* (). Суффикс *_copy* используется для того, чтобы различать эти два вида алгоритмов.

Чтобы удалить из контейнера одинаковые элементы, мы должны явно сжать его:

```
template<class C> void eliminate_duplicates (C& c)
{
        sort (c.begin (), c.end ());                          // сортируем
        typename C::iterator p = unique (c.begin (), c.end ()); // сжимаем
        c.erase (p, c.end ());                                // укорачиваем
}
```

Отметим, что функция *eliminate_duplicates* () не имела бы смысла для встроенного массива, но *unique* () можно применять и к массивам.

Пример *unique_copy* () можно найти в § 3.8.3.

18.6.3.1. Критерии сортировки

Чтобы удалить одинаковые значения, входную последовательность нужно отсортировать (§ 18.7.1). Оба алгоритма, и ***unique*** (), и ***unique_copy*** () по умолчанию в качестве критерия для сравнения пользуются оператором == и позволяют пользователю ввести собственный критерий. Например, мы могли бы изменить пример из § 18.5.1 так, чтобы удалить одинаковые имена. После извлечения имен обслуживающего персонала из контейнера ***Club***, мы бы остались со списком ***list<Person*>***, названным ***off*** (§ 18.5.1). Мы могли бы удалить одинаковые имена так:

```
eliminate_duplicates (off);
```

Однако, такой метод основывается на сортировке указателей и предполагает, что каждый указатель однозначно определяет человека. Вообще говоря, нам бы пришлось более внимательно отнестись к записям ***Person***, чтобы определить, будем ли мы считать их одинаковыми. Мы могли бы написать:

```
bool operator== (const Person& x, const Person& y)        // равенство для объектов
{
    // сравниваем x и y на равенство
}
bool operator< (const Person& x, const Person& y)         // «меньше» для объектов
{
    // сравниваем x и y
}
bool Person_eq (const Person* x, const Person* y)         // равенство через указатель
{
    return *x == *y;
}
bool Person_lt (const Person* x, const Person* y)         // «меньше» через указатель
{
    return *x < *y;
}
void extract_and_print (const list<Person*>& lc)
{
    list<Person*> off;
    extract (lc, off);
    off.sort (Person_lt);
    list<Club>::iterator p = unique (off.begin (), off.end (), Person_eq);
    for_each (off.begin (), p, Print_name (cout));
}
```

Было бы разумно убедиться, что критерий равенства при сортировке совпадений тот же, что и при удалении одинаковых элементов. Смысл по умолчанию == и < для указателей редко годится для сравнения объектов, на которые эти указатели указывают.

18.6.4. Алгоритмы replace

Алгоритмы ***replace*** () проходят по последовательности, заменяя указанные значения. Эти алгоритмы следуют модели ***find/find_copy*** и ***unique/unique_copy***, что дает четыре варианта алгоритма. И снова, код достаточно прост и не требует пояснений:

```
template<class For, class T>
void replace (For first, For last, const T& val, const T& new_val)
{
        while (first != last) {
                if (*first ==val) *first = new_val;
                ++first;
        }
}
template<class For, class Pred, class T>
void replace_if (For first, For last, Pred p, const T& val)
{
        while (first != last) {
                if (p (*first)) *first = val;
                ++first;
        }
}
template<class In, class Out, class T>
Out replace_copy (In first, In last, Out res, const T& val, const T& new_val)
{
        while (first != last) {
                *res++ = (*first = =val) ? new_val : *first;
                ++first;
        }
        return res;
}
template<class In, class Out, class Pred, class T>
Out replace_copy_if (In first, In last, Out res, Pred p, const T& new_val)
{
        while (first != last) {
                *res+ = p (*first) ? new_val : *first;
                ++first;
        }
        return res;
}
```

Допустим нам понадобилось пройти по списку строк, заменяя обычную английскую транслитерацию имени моего родного города Aarhus его истинным названием Århus:

```
void f (list<string>& towns)
{
        replace (towns.begin (), towns.end (), "Aarhus", "Århus");
}
```

Этот пример использует расширенный набор символов (§ B.3.3).

18.6.5. Алгоритмы remove

Алгоритмы *remove* () перемещают элементы из последовательности, основываясь на значении или предикате:

```
template<class For, class T> For remove (For first, For last, const T& val);
template<class For, class Pred> For remove_if (For first, For last, Pred p);
```

```
template<class In, class Out, class T> Out remove_copy (In first, In last, Out res, const T& val);

template<class In, class Out, class Pred> Out remove_copy_if (In first, In last, Out res, Pred p);
```

Допустив, что **Club** (клуб) имеет адрес, мы можем создать список клубов с адресом в Копенгагене:

```
class located_in : public unary_function <Club, bool> {
        string town;
public:
        located_in (const string& ss) : town (ss) { }
        bool operator () (const Club& c) const { return c.town == town; }
};

void f (list<Club>& lc)
{
        remove_copy_if (lc.begin (), lc.end (),
                ostream_iterator<Club> (cout), not1 (located_in ("København")));
}
```

Таким образом, алгоритм ***remove_copy_if*** () — это ***copy_if*** () (§ 18.6.1) с обратным условием. То есть элемент помещается алгоритмом ***remove_copy_if*** () на выход, если он не удовлетворяет предикату.

«Простой» ***remove*** () записывает все несоответствующие элементы в начало последовательности и возвращает итератор на конец этой уплотненной подпоследовательности (см. также § 18.6.3).

18.6.6. Алгоритмы fill и generate

Алгоритмы ***fill*** () и ***generate*** () созданы для того, чтобы систематически присваивать последовательностям значения:

```
template<class For, class T> void fill (For first, For last, const T& val);
template<class Out, class Size, class T> void fill_n (Out res, Size n, const T& val);

template<class For, class Gen> void generate (For first, For last, Gen g);
template<class Out, class Size, class Gen> void generate_n (Out res, Size n, Gen g);
```

Алгоритм ***fill*** () присваивает указанное значение, алгоритм ***generate*** () присваивает значения путем многократного вызова функции, являющейся его аргументом. Таким образом, ***fill*** () — это просто специальный случай ***generate*** (), в котором функция-генератор многократно возвращает одно и то же значение. Версии с суффиксом ***_n*** производят присваивание только первым ***n*** элементам последовательности.

Например, используя генераторы случайных чисел ***Randint*** и ***Urand*** из § 22.7, можно написать:

```
int v1[900];
int v2[900];
vector v3;

void f ()
{
        // присваивание всем элементам v1 значения 99
        fill (v1, &v1[900], 99);
```

```
        // присваивание случайных значений (§ 22.7)
        generate(v2, &v2[900], Randint());

        // вывод 200 случайных целых чисел: в интервале [0..99]:
        generate_n(ostream_iterator<int>(cout), 200, Urand(100));        // см. § 22.7

        // добавляем в v3 20 элементов со значением 99
        fill_n(back_inserter(v3), 20, 99);
    }
```

Функции ***generate*** () и ***fill*** () скорее присваивают, а не инициализируют. Если вам нужно манипулировать чистой памятью, скажем, чтобы превратить область памяти в объекты хорошо определенного типа и с определенным состоянием, вы должны пользоваться алгоритмами вроде ***uninitialized_fill*** () из <***memory***> (§ 19.4.4), а не алгоритмами из <*algorithm*>.

18.6.7. Алгоритмы reverse и rotate

Порой нам нужно изменить порядок элементов в последовательности:

```
template<class Bi> void reverse(Bi first, Bi last);
template<class Bi, class Out> Out reverse_copy(Bi first, Bi last, Out res);

template<class For> void rotate(For first, For middle, For last);
template<class For, class Out> Out rotate_copy(For first, For middle, For last, Out res);

template<class Ran> void random_shuffle(Ran first, Ran last);
template<class Ran, class Gen> void random_shuffle(Ran first, Ran last, Gen& g);
```

Алгоритм ***reverse*** () изменяет порядок следования элементов на обратный, так что первый элемент становится последним и т. д. Алгоритм ***reverse_copy*** () производит копию входной последовательности в обратном порядке.

Алгоритм ***rotate*** () считает последовательность [***first***, ***last***[кольцом и циклически сдвигает ее элементы, пока бывший средний элемент (***middle***) не окажется на месте первого. То есть элемент из положения ***first*** + ***i*** сдвигается на позицию ***first*** + (***i*** + (***last*** − ***middle***)) % (***last*** − ***first***). Знак % (целочисленное деление) делает сдвиг циклическим, а не просто сдвигом влево. Например:

```
void f()
{
    string v[] = {"Лягушка", "и", "Персик"};
    reverse(v, v+3);        // Персик и Лягушка
    rotate(v, v+1, v+3);    // и Лягушка Персик
}
```

Алгоритм ***rotate_copy*** () копирует свой вход циклически.

По умолчанию ***random_shuffle*** () тасует последовательность при помощи генератора случайных равномерно распределенных чисел. То есть, алгоритм выбирает перестановку элементов в последовательности таким образом, что любая перестановка имеет равную вероятность быть использованной. Если вам нужно другое распределение или вас не устраивает генератор случайных чисел, вы можете создать их сами. При вызове ***random_shuffle***(***b***,***e***,***r***) генератор вызывается с аргументом, равным количеству элементов в последовательности. Например при вызове ***r***(***e*** − ***b***) генератор

должен возвращать значения в диапазоне [*0*, *e–b*). Если *My_rand* — такой генератор, мы могли бы перетасовать колоду карт следующим образом:

```
void f(deque<Card>& dc, My_rand& r)
{
    random_shuffle(dc.begin(), dc.end(), r);
    // ...
}
```

Перемещение элементов выполняется *rotate*() при помощи *swap*() (§ 18.6.8).

18.6.8. Алгоритмы swap

Чтобы делать с элементами контейнера что-нибудь мало-мальски интересное, их нужно менять местами. Такая «перемена мест» выражается лучше всего — то есть наиболее просто и эффективно — алгоритмами *swap*():

```
template<class T> void swap(T& a, T&b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
template<class For, class For2> void iter_swap(For x, For2 y);
template<class For, class For2>
    For2 swap_ranges(For first, For last, For2 first2)
{
    while (first != last) iter_swap(first++, first2++);
    return first2;
}
```

Чтобы поменять элементы местами, вам понадобится переменная для временного хранения элемента. Существуют всякие хитрости, чтобы в особых случаях избавиться от нее, но ради простоты и наглядности лучше даже об этом не говорить. Алгоритм *swap*() специализирован для важных типов, где это имеет существенное значение (§ 16.3.9, § 13.5.2).

Алгоритм *iter_swap*() меняет местами элементы, на которые указывают аргументы-итераторы.

Алгоритм *swap_ranges* меняет местами элементы в двух входных диапазонах.

18.7. Сортированные последовательности

Когда мы собрали некоторые данные, нам часто нужно отсортировать их. Когда последовательность отсортирована, наши возможности манипулирования данными в удобном виде значительно возрастают.

Чтобы отсортировать последовательность, нам необходим способ сравнения элементов. Это делается при помощи бинарных предикатов (§ 18.4.2). По умолчанию операцией сравнения является <.

18.7.1. Сортировка (алгоритмы sort)

Алгоритмы *sort*() требуют итераторов с произвольным доступом (§ 19.2.1). То есть они лучше всего работают с контейнерами типа *vector* (§ 16.3) и ему подобными:

```
template<class Ran> void sort (Ran first, Ran last);
template<class Ran, class Cmp> void sort (Ran first, Ran last, Cmp cmp);

template<class Ran> void stable_sort (Ran first, Ran last);
template<class Ran, class Cmp>
    void stable_sort (Ran first, Ran last, Cmp cmp);
```

Стандартный контейнер ***list*** (§ 17.2.2) не обеспечиваст итераторов с произвольным доступом, поэтому такие контейнеры следует сортировать при помощи специфических операций класса ***list*** (§ 17.2.2.1).

Базовый алгоритм ***sort*** () эффективен (в среднем он имеет показатель ***N*log** (***N***)), но не всегда (в худшем случае время растет как ***O*** (***N*N***)). К счастью, худшие случаи возникают нечасто. Если важно гарантировать поведение алгоритма и в плохих случаях, или требуется устойчивая сортировка, следует использовать ***stable_sort*** (), то есть алгоритм с затратами ***N*loq*** (***N***)***log*** (***N***), который можно улучшить практически до ***N*log*** (***N***), когда система имеет достаточно оперативной памяти. Относительный порядок одинаковых элементов сохраняется алгоритмом ***stable_sort*** (), но не ***sort*** ().

Иногда нужны только первые элементы в отсортированной последовательности. В этом случае имеет смысл сортировать последовательность, только пока не будет приведена в порядок первая часть. Это называется *частичной сортировкой* (partial sort):

```
template<class Ran> void partial_sort (Ran first, Ran middle, Ran last);
template<class Ran, class Cmp>
    void partial_sort (Ran first, Ran middle, Ran last, Cmp cmp);

template<class In, class Ran>
    Ran partial_sort_copy (In first, In last, Ran first2, Ran last2);
template<class In, class Ran, class Cmp>
    Ran partial_sort_copy (In first, In last, Ran first2, Ran last2, Cmp cmp);
```

«Простые» алгоритмы ***partial_sort*** () расставляют по порядку элементы в диапазоне от ***first*** до ***middle***. Алгоритмы ***partial_sort_copy*** () выдают ***N*** элементов, где ***N*** равно меньшей из длин двух последовательностей (входной и выходной). Нам нужно указать и начало, и конец результирующей последовательности, поскольку это определяет, сколько элементов нужно отсортировать. Например:

```
class Compare_copies_sold {                          // «сравнение проданных копий»
public:
    bool operator () (const Book& b1, const Book& b2) const // сортировка в убывающем порядке
        { return b1.copies_sold () < b2.copies_sold (); }
};

void f(vector<Book> & sales)                         // находит десять лучших книг
{
    vector<Book> bestsellers (10);
    partial_sort_copy (sales.begin (), sales.end (),
        bestsellers.begin (), bestsellers.end (), Compare_copies_sold ());
    copy (bestsellers.begin (),
        bestsellers.end (), ostream_iterator<Book> (cout, "\n"));
}
```

Поскольку целевым итератором в алгоритме ***partial_sort_copy*** () должен быть итератор с произвольным доступом, мы не можем сортировать прямо в ***cout***.

Наконец, предоставляются алгоритмы для того, чтобы сортировать только до тех пор, пока *N*-й элемент не займет подобающее ему место, так что никакой элемент не располагается меньше, чем *N*-й элемент, в последовательности после него:

```
template<class Ran> void nth_element (Ran first, Ran nth, Ran last);
template<class Ran, class Cmp>
    void nth_element (Ran first, Ran nth, Ran last, Cmp cmp);
```

Этот алгоритм особенно полезен для тех, кому нужно искать медианы, процентили и т. п., например, для экономистов, социологов и учителей.

18.7.2. Двоичный поиск

Последовательный поиск, такой как алгоритм *find* () (§ 18.5.2), страшно неэффективен для длинных последовательностей, но это лучшее, что мы можем сделать без сортировки и хэширования (§ 17.6). Однако, когда последовательность отсортирована, для определения, есть ли в ней то или иное значение, мы можем воспользоваться двоичным поиском:

```
template<class For, class T> bool binary_search (For first, For last, const T& val);

template<class For, class T, class Cmp>
    bool binary_search (For first, For last, const T& value, Cmp cmp);
```

Например:

```
void f(list<int>& c)
{
    if (binary_search (c.begin (), c.end (), 7)) {        // есть ли 7 в с?
        //...
    }
}
```

binary_search () возвращает значение *bool*, показывающее, присутствует ли в последовательности данное значение. Как и в случае с *find* (), мы часто хотим знать, где находится в последовательности данный элемент. Однако в последовательности может быть много таких элементов, и тогда мы должны указать, что мы хотим: найти первый из них или все. Поэтому разработаны алгоритмы для поиска ряда одинаковых элементов (*equal_range* ()) и для поиска нижней и верхней границ этого ряда (соответственно *lower_bound* () и *upper_bound* ()):

```
template<class For, class T>
    For lower_bound (For first, For last, const T& val);
template<class For, class T, class Cmp>
    For lower_bound (For first, For last, const T& val, Cmp cmp);

template<class For, class T>
    For upper_bound (For first, For last, const T& val);
template<class For, class T, class Cmp>
    For upper_bound (For first, For last, const T& val, Cmp cmp);

template<class For, class T>
    pair<For, For> equal_range (For first, For last, const T& val);
template<class For, class T, class Cmp>
    pair<For, For> equal_range (For first, For last, const T& val, Cmp cmp);
```

Эти алгоритмы согласуются с операциями из *multimap* (§ 17.4.2). Мы можем представлять себе *lower_bound* () как быстрые *find* () и *find_if* () для сортированных последовательностей. Например:

```
void g(vector<int>& c)
{
    typedef vector::iterator VI;

    // вероятно, работает медленно: O(N); вектор с сортировать не нужно
    VI p = find(c.begin(), c.end(), 7);
    // вероятно, работает быстро: O(log(N)); вектор с должен быть отсортирован
    VI q = lower_bound(c.begin(), c.end(), 7);
    // ...
}
```

Если *lower_bound* (*first*, *last*, *k*) не находит *k*, он возвращает итератор на первый элемент с ключом больше *k*, или *last*, если такого элемента не найдено. Такой способ сообщения о неудаче также используется в *upper_bound* () и *equal_range* (). А значит, мы можем пользоваться этими алгоритмами для определения, на какое место вставить в сортированную последовательность новый элемент, чтобы она осталась сортированной.

18.7.3. Слияние (алгоритмы merge)

Имея две отсортированные последовательности, мы можем объединить (слить) их в одну новую отсортированную последовательность при помощи алгоритма *merge* () или объединить две части одной последовательности при помощи *inplace_merge* ():

```
template<class In, class In2, class Out>
    Out merge(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out merge(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class Bi> void inplace_merge(Bi first, Bi middle, Bi last);
template<class Bi, class Cmp>
    void inplace_merge(Bi first, Bi middle, Bi last, Cmp cmp);
```

Отметим, что эти алгоритмы слияния отличаются от слияния списков *list* (§ 17.2.2.1) тем, что не удаляют элементы из своей входной последовательности. Вместо этого элементы копируются.

При равенстве элементы из первого диапазона будут всегда предшествовать элементам второго.

Алгоритм *inplace_merge* () используется главным образом тогда, когда у вас есть последовательность, которую можно отсортировать по нескольким критериям. Например, у вас может быть вектор *vector* с названиями рыб, отсортированный по видам. Если элементы каждого вида отсортированы по весу, вы можете отсортировать весь вектор при помощи *inplace_merge* () и объединить информацию для разных видов (§ 18.13[20]).

18.7.4. Разделители (алгоритмы partition)

Разделить последовательность, значит расположить все элементы, удовлетворяющие предикату, перед всеми элементами, которые ему не удовлетворяют. Стандартная библиотека предоставляет алгоритм *stable_partition* (), который сохраняет относительный

порядок элементов, удовлетворяющих и не удовлетворяющих данному предикату. Кроме того, библиотека предлагает алгоритм ***partition*** (), который не сохраняет относительный порядок, но работает чуть быстрее, когда память ограничена:

```
template<class Bi, class Pred> Bi partition (Bi first, Bi last, Pred p);
template<class Bi, class Pred> Bi stable_partition (Bi first, Bi last, Pred p);
```

Вы можете представлять себе разделитель как некую разновидность сортировки с очень простым критерием. Например:

```
void f(list<Club>& lc)
{
    list<Club>::iterator
        p = partition (lc.begin (), lc.end (), located_in ("København"));
    // ...
}
```

В результате список ***list*** будет отсортирован так, что клубы (объекты ***Club***) с адресом в Копенгагене расположатся в начале. Возвращенное значение (здесь ***p***) укажет либо на первый элемент, не удовлетворяющий предикату (здесь ***located_in*** (***"København"***)), либо на конец списка.

18.7.5. Операции с множествами для последовательностей

Последовательности можно рассматривать как множества. Поэтому имеет смысл обеспечить для последовательностей операции, аналогичные действиям с множествами, например пересечение. Однако эти операции очень неэффективны, если последовательности не отсортированы, поэтому стандартная библиотека обеспечивает операции с множествами только для сортированных последовательностей. В частности, операции с множествами хорошо работают с ***set*** (§ 17.4.3) и ***multiset*** (§ 17.4.4), которые в любом случае отсортированы.

Если эти теоретико-множественные алгоритмы применить к несортированным последовательностям, результирующая последовательность не будет подчиняться обычным правилам теории множеств. Эти алгоритмы не изменяют свою входную последовательность, а их выходные последовательности упорядочены.

Алгоритм ***includes*** () проверяет, является ли каждый член второй последовательности [***first2*** : ***last2***[также и членом первой:

```
template<class In, class In2>
    bool includes (In first, In last, In2 first2, In2 last2);
template<class In, class In2, class Cmp>
    bool includes (In first, In last, In2 first2, In2 last2, Cmp cmp);
```

Назначение алгоритмов ***set_union*** () (объединение) и ***set_intersection*** () (пересечение) очевидно:

```
template<class In, class In2, class Out>
    Out set_union (In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_union (In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
    Out set_intersection (In first, In last, In2 first2, In2 last2, Out res);
```

```
template<class In, class In2, class Out, class Cmp>
    Out set_intersection (In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

Алгоритм ***set_difference*** () выдает последовательность элементов, входящих в первую, но не входящих во вторую входную последовательность. Алгоритм ***set_symmetric_difference*** () (симметрическая разность) выдает последовательность элементов, входящих в одну и только одну из входных последовательностей:

```
template<class In, class In2, class Out>
    Out set_difference (In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_difference (In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
    Out set_symmetric_difference (In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_symmetric_difference (In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

Например:

```
char v1[] = "abcd";
char v2[] = "cdef";

void f(char v3[])
{
    set_difference (v1, v1+4, v2, v2+4, v3);              // v3 = "ab"
    set_symmetric_difference (v1, v1+4, v2, v2+4, v3);    // v3 = "abef"
}
```

18.8. Кучи

Слово «куча» (heap) в разном контексте означает разные вещи. Применительно к алгоритмам, «куча» часто обозначает способ организации последовательности, когда первым является элемент с наибольшим значением. Добавление элемента (при помощи ***push_heap*** ()) и удаление (при помощи ***pop_heap*** ()) производится довольно быстро; в худшем случае быстродействие равно ***O*** (***log*** (***N***)), где ***N*** — число элементов в последовательности. Сортировка (при помощи ***sort_heap*** ()) в худшем случае имеет быстродействие ***O*** (***N**log*** (***N***)). Куча реализуется следующим набором функций:

```
template<class Ran> void push_heap (Ran first, Ran last);
template<class Ran, class Cmp> void push_heap (Ran first, Ran last, Cmp cmp);

template<class Ran> void pop_heap (Ran first, Ran last);
template<class Ran, class Cmp> void pop_heap (Ran first, Ran last, Cmp cmp);

// превращают последовательность в кучу:
template<class Ran> void make_heap (Ran first, Ran last);
template<class Ran, class Cmp> void make_heap (Ran first, Ran last, Cmp cmp);

// превращают кучу в последовательность:
template<class Ran> void sort_heap (Ran first, Ran last);
template<class Ran, class Cmp> void sort_heap (Ran first, Ran last, Cmp cmp);
```

Стиль алгоритмов, работающих с кучами, довольно странен. Естественнее было бы обеспечить класс-адаптер с четырьмя операциями. Получилось бы нечто вроде

priority_queue (§ 17.3.3) (очереди с приоритетами). Фактически, *priority_queue* почти наверняка реализуется с использованием куч.

Значение элемента, помещаемого в кучу алгоритмом *push_heap* (*first, last*), равно * (*last–1*). Принимается допущение, что [*first, last–1*[уже является кучей, так что *push_heap* (), включая следующий элемент, расширяет последовательность до [*first, last*[. Таким образом, серией операций *push_heap* () можно из существующей последовательности построить кучу. И наоборот, *pop_heap* () удаляет первый элемент из кучи, переставляя его с последним (* (*last–1*)) и делая последовательность [*first, last –1*[кучей.

18.9. Алгоритмы min и max

Описанные здесь алгоритмы выбирают значение, основываясь на сравнении. Очевидно, полезно уметь находить максимальное и минимальное из двух значений:

```
template<class T> const T& max (const T& a, const T& b)
{
    return (a<b) ? b : a;
}

template<class T, class Cmp>
    const T& max (const T& a, const T& b, Cmp cmp)
{
    return (cmp (a, b)) ? b : a;
}

template<class T> const T& min (const T& a, const T& b);

template<class T, class Cmp> const T& min (const T& a, const T& b, Cmp cmp);
```

Операции *max* () и *min* () можно обобщить, чтобы очевидным образом применять их к последовательностям:

```
template<class For> For max_element (For first, For last);
template<class For, class Cmp> For max_element (For first, For last, Cmp cmp);

template<class For> For min_element (For first, For last);
template<class For, class Cmp> For min_element (For first, For last, Cmp cmp);
```

И наконец, лексикографическое упорядочивание строк легко обобщить для последовательности значений, которые можно сравнивать:

```
template<class In, class In2>
bool lexicographical_compare (In first, In last, In2 first2, In2 last2);

template<class In, class In2, class Cmp>
bool lexicographical_compare (In first, In last, In2 first2, In2 last2, Cmp cmp)
{
    while (first != last && first2 != last2) {
        if (cmp (*first, *first2)) return true;
        if (cmp (*first2++, *first++)) return false;
    }
    return first == last && first2 != last2;
}
```

Это очень напоминает соответствующую функцию для обычных строк (§ 13.4.1), однако *lexicographical_compare* () сравнивает последовательности вообще, а не только

строки. Кроме того этот алгоритм возвращает ***bool***, а не более распространенный ***int***. Результатом является ***true*** тогда (и только тогда), когда первая последовательность при сравнении со второй оператором < дает ***true***. В частности, результат будет ***false***, если последовательности равны. Например:

```
char v1[] = "да";
char v2[] = "нет";
string s1 = "Да";
string s2 = "Нет";

void f()
{
        bool b1 = lexicographical_compare (v1, v1+strlen (v1), v2, v2+strlen (v2));
        bool b2 = lexicographical_compare (s1.begin (), s1.end (), s2.begin (), s2.end ());

        bool b3 = lexicographical_compare (v1, v1+strlen (v1), s1.begin (), s1.end ());
        bool b4 = lexicographical_compare (s1.begin (), s1.end (), v1, v1+strlen (v1), Nocase ());
}
```

Последовательности не должны быть одного и того же типа: все, что нужно — это сравнивать их элементы, а критерий сравнения можно сообщить. Это делает ***lexicographical_compare*** () более универсальным и потенциально более медленным, чем сравнение строк — см. также § 20.3.8.

18.10. Перестановки (алгоритмы permutations)

Имея последовательность из четырех элементов, мы можем расставить их 4*3*2 способами. Каждый из вариантов называется *перестановкой* (permutation):

```
abcd abdc acbd acdb adbc adcb bacd badc
bcad bcda bdac bdca cabd cadb cbad cbda
cdab cdba dabc dacb dbac dbca dcab dcba
```

Функции ***next_permutation*** () и ***prev_permutation*** () выдают такие перестановки в последовательности:

```
template<class Bi> bool next_permutation (Bi first, Bi last);
template<class Bi, class Cmp> bool next_permutation (Bi first, Bi last, Cmp cmp);

template<class Bi> bool prev_permutation (Bi first, Bi last);
template<class Bi, class Cmp> bool prev_permutation (Bi first, Bi last, Cmp cmp);
```

Перестановки строки ***abcd*** можно произвести таким образом:

```
int main()
{
        char v[] = "abcd";
        cout << v << "\t";
        while (next_permutation (v, v+4)) cout << v << '\t';
}
```

Перестановки производятся в лексикографическом порядке (§ 18.9). Возвращенное функцией ***next_permutation*** () значение показывает, существует ли следующая перестановка. Если нет, возвращается ***false*** и последовательность является такой перестановкой, в которой элементы стоят в лексикографическом порядке. Возвращенное

функцией *prev_permutation* () значение показывает, существует ли предыдущая перестановка. Если нет, возвращается *false* и последовательность является такой перестановкой, в которой элементы стоят в обратном лексикографическом порядке.

18.11. Алгоритмы в стиле C

От стандартной библиотеки C стандартная библиотека C++ унаследовала несколько алгоритмов, оперирующих со строками C (§ 20.4.1), а также быструю сортировку и двоичный поиск. И то и другое применяется только для массивов.

Функции *qsort* () и *bsearch* () представлены в <*cstdlib*> и <*stdlib.h*>. Обе они оперируют массивами из *n* элементов размера *elem_size*, используя функцию сравнения «меньше чем», передаваемую по указателю. Элементы должны иметь тип без пользовательского копирующего конструктора, копирующего оператора присваивания и деструктора:

```
typedef int (*__cmp) (const void*, const void*);

void qsort (void* p, size_t n, size_t elem_size, __cmp);      // сортировка p
void* bsearch (const void* key, void* p, size_t n,
        size_t elem_size, __cmp);                             // находит в p ключ key
```

Использование *qsort* () описано в § 7.7.

Эти алгоритмы предоставляются только для совместимости с языком C; *sort* () (§ 18.7.1) и *search* () (§ 18.5.5) — наиболее универсальные функции и должны быть также более эффективными.

18.12. Советы

[1] Предпочитайте алгоритмы циклам; § 18.5.1.

[2] Когда пишете цикл, подумайте, нельзя ли выразить его универсальным алгоритмом; § 18.2.

[3] Регулярно пересматривайте набор алгоритмов, чтобы увидеть, не стало ли написание новой прикладной программы более очевидным; § 18.2.

[4] Всегда убеждайтесь, что пара аргументов-итераторов действительно определяет последовательность; § 18.3.1.

[5] Разрабатывайте программы так, чтобы наиболее часто употребляемые операции были просты и безопасны; § 18.3, § 18.3.1.

[6] Выражайте условия проверки в такой форме, чтобы их можно было использовать в качестве предикатов; § 18.4.2.

[7] Помните, что предикаты — это функции и объекты, а не типы; § 18.4.2.

[8] Чтобы сделать из бинарных предикатов унарные, вы можете воспользоваться связывателями; § 18.4.4.1.

[9] Для применения алгоритмов к контейнерам пользуйтесь *mem_fun* () и *mem_fun_ref* (); § 18.4.4.2.

[10] Для связывания аргумента функции пользуйтесь *ptr_fun* (); § 18.4.4.3.

[11] Помните, что *strcmp* () отличается от оператора == тем, что при равенстве возвращает *0*; § 18.4.4.4.

[12] Пользуйтесь *for_each* () и *transform* () только тогда, когда для задачи не существует более специфичного алгоритма; § 18.5.1.

[13] Для применения алгоритмов с несколькими критериями сравнения и равенства пользуйтесь предикатами; § 18.4.2.1; § 18.6.3.1.

[14] Используйте предикаты и другие объекты-функции, чтобы применять стандартные алгоритмы в «более широком смысле»; § 18.4.2.

[15] Операторы по умолчанию < и == для указателей редко подходят к стандартным алгоритмам; § 18.6.3.1.

[16] Алгоритмы напрямую не добавляют и не удаляют элементы из своих входных последовательностей (аргументов); § 18.6.

[17] Всегда проверяйте соответствие друг другу что используемых для последовательностей предикатов «меньше» и «равно»; § 18.6.3.1.

[18] Иногда для повышения эффективности и изящества можно воспользоваться сортированными последовательностями; § 18.7.

[19] Пользуйтесь функциями ***qsort*** () и ***bsearch*** () только для обеспечения совместимости; § 18.11.

18.13. Упражнения

Решение некоторых задач к этой главе можно найти, просмотрев исходный текст реализации стандартной библиотеки. Сделайте одолжение, попытайтесь найти собственные решения, прежде чем смотреть, как к этим проблемам подошел разработчик вашей библиотеки.

1. (*2) Разберитесь с обозначением ***O*** (). Дайте реальный пример, где алгоритм ***O*** (***N*N***) быстрее, чем ***O*** (***N***) для некоторого ***N>10***.
2. (*2) Реализуйте и проверьте четыре функции ***mem_fun*** () и ***mem_fun_ref*** () (§ 18.4.4.2).
3. (*1) Напишите алгоритм поиска совпадающей пары ***match*** (), который похож на ***mismatch*** () (несовпадение), за исключением того, что возвращает итераторы на первые два элемента, удовлетворяющие предикату.
4. (*1.5) Реализуйте и проверьте функцию печати имен ***Print_name*** из § 18.5.1.
5. (*1) Отсортируйте список ***list***, используя только стандартные библиотечные алгоритмы.
6. (*2.5) Определите версии ***iseq*** () (§ 18.3.1) для встроенных массивов, ***istream*** и пар итераторов. Определите подходящий набор перегрузок для алгоритмов, немодифицирующих последовательность(§ 18.5), с целью применения к последовательностям ***Iseq***. Подумайте, как лучше всего избежать двусмысленности и резкого увеличения количества функций-шаблонов .
7. (*2) Определите ***oseq*** (), дополняющую ***iseq*** () (***o*** — от ***out***, ***i*** — от ***in***). Выходная последовательность, заданная как аргумент ***oseq*** (), должна заместиться выходной последовательностью алгоритма. Определите подходящий набор перегруженных операторов по крайней мере для трех стандартных алгоритмов по вашему выбору.
8. (*1.5) Создайте вектор ***vector*** квадратов целых чисел от 1 до 100. Распечатайте таблицу результатов. Возьмите квадратный корень от элементов этого вектора и распечатайте получившийся вектор.
9. (*2) Напишите набор объектов-функций, которые производят битовые логические операции над своими операндами. Проверьте эти объекты на векторах с элементами ***char***, ***int*** и ***bitset<67>***.

10. (*1) Напишите связыватель ***binder3*** (), который бы связывал второй и третий аргументы трехаргументной функции для получения унарного предиката. Приведите пример, где ***binder3*** () является полезной функцией.
11. (*1.5) Напишите маленькую программку, которая которая удаляла бы соседние одинаковые слова из из файла файла. Подсказка: программа должна удалить из предыдущего предложения три слова: ***которая***, ***из*** и ***файла***.
12. (*2.5) Определите формат записи для ссылок на статьи и книги. Напишите программу, которая могла бы выбирать из файла записи по времени издания, имени автора, издательству или ключевым словам. Пользователь должен иметь возможность указать один из этих критериев для сортировки выхода.
13. (*2) Реализуйте алгоритм ***move*** () в стиле ***copy*** () таким образом, чтобы входная и выходная последовательности могли перекрываться. Постарайтесь обеспечить приемлемую эффективность, задавая в качестве аргументов итераторы с произвольным доступом.
14. (*1.5) Создайте все анаграммы из слова ***food*** — то есть все четырехбуквенные комбинации ***f***, ***o***, ***o***, ***d***. Обобщите эту программу так, чтобы она получала на вход слово, а на выходе выдавала анаграммы.
15. (*1.5) Напишите программу, которая бы выдавала анаграммы предложений, то есть выдавала бы все перестановки слов в предложении (а не букв в слове).
16. (*1.5) Реализуйте ***find_if*** () (§ 18.5.2), а затем при помощи ***find_if*** () реализуйте ***find*** (). Найдите способ сделать это так, чтобы не возникало необходимости давать функциям различные имена.
17. (*2) Реализуйте ***search*** () (§ 18.5.5). Создайте оптимизированную версию для итераторов с произвольным доступом.
18. (*2) Возьмите алгоритм сортировки (например, ***sort*** () из вашей стандартной библиотеки или сортировку Шелла из § 13.5.2) и вставьте в него код, так чтобы он распечатывал сортируемую последовательность после каждой перестановки элементов.
19. (*2) Для двунаправленных итераторов нет алгоритма ***sort*** (). Существует гипотеза, что копирование последовательности в вектор и последующая сортировка будет быстрее, чем сортировка последовательности при помощи двунаправленных итераторов. Реализуйте универсальную сортировку для двунаправленных итераторов и проверьте эту гипотезу.
20. (*2.5) Представьте, что вы ведете записи о группе спортсменов-рыболовов. Для каждой выловленной рыбы запишите ее вид, длину, вес, дату улова, фамилию рыболова и т. д. Отсортируйте и распечатайте записи в соответствии с разными критериями. Подсказка: ***inplace_merge***.
21. (*2) Создайте список студентов, изучающих математику, английский, французский и биологию. Из 40 человек выберите около 20 имен для каждого предмета. Составьте список студентов, изучающих и математику, и английский. Составьте список студентов, изучающих французский, но не изучающих ни биологию, ни математику. Составьте список студентов, не изучающих науки. Составьте список студентов, изучающих французский и математику, но не изучающих ни английский, ни биологию.
22. (*1.5) Напишите функцию ***remove*** (), которая бы действительно удаляла элементы из контейнера.

19

Итераторы и распределители памяти

Структуры данных и алгоритмы
могут работать вместе только потому, ...
что они ничего не знают друг о друге.
— Алекс Степанов

Итераторы и последовательности — операции над итераторами — свойства итераторов — категории итераторов — вставки — обратные итераторы — итераторы потоков — итераторы с проверкой — исключения и алгоритмы — распределители памяти — стандартный распределитель памяти — пользовательские распределители памяти — низкоуровневые функции для работы с памятью — советы — упражнения.

19.1. Введение

Итераторы — это клей, который удерживает контейнеры и алгоритмы вместе. Итераторы обеспечивают абстрактный взгляд на данные, так что создателю алгоритма нет необходимости заботиться о конкретных деталях всех возможных структур данных. И наоборот, обеспечиваемая итераторами стандартная модель доступа к данным освобождает контейнеры от необходимости предоставлять более широкий набор операций доступа. Подобным же образом, распределители памяти используются для того, чтобы при реализации контейнеров оградить разработчика от знания подробностей доступа к памяти.

Итераторы поддерживают абстрактную модель данных как последовательности объектов (§ 19.2). Распределители памяти обеспечивают отображение низкоуровневой модели, представляющей данные как массив битов, в высокоуровневую объектную модель (§ 19.4). Самая распространенная низкоуровневая модель памяти поддерживается всего несколькими стандартными функциями (§ 19.4.4).

Итераторы — это понятие, с которым должны быть знакомы все программисты. И напротив, распределители памяти — это служебный механизм, о котором программисту редко нужно беспокоиться, и мало кому из программистов когда-либо понадобится писать новый распределитель памяти.

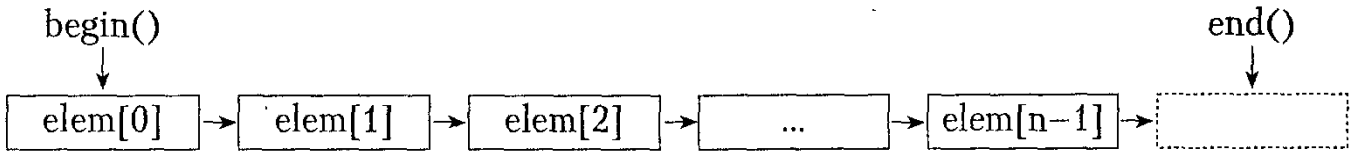
19.2. Итераторы и последовательности

Итератор — это чистая абстракция. То есть все, что ведет себя, как итератор, и есть итератор (§ 3.8.2). Итератор абстрагирует понятие указателя на элемент последовательности. Ключевые концепции итератора таковы:

- «текущий указываемый элемент» (косвенное обращение; представляется операторами * и ->);
- «указать на следующий элемент» (инкремент; представляется оператором ++),
- равенство (представляется оператором ==).

Например, встроенный тип ***int**** — это итератор для ***int***[], а класс ***list<int>::iterator*** — итератор для класса ***list<int>***.

Последовательность — это абстракция понятия «нечто такое, что мы можем перебирать от начала до конца, используя оператор получения следующего элемента»:



Примерами последовательностей служат массивы (§ 5.2), вектора (§ 16.3), односвязные списки (§ 17.8[17]), двусвязные списки (§ 17.2.2), деревья (§ 17.4.1), потоки ввода (§ 21.3.1) и вывода (§ 21.2.1). Каждая последовательность имеет свой вид итераторов.

Итераторные классы и функции объявлены в пространстве имен ***std*** и находятся в заголовочном файле <***iterator***>.

Итератор *не* является обычным указателем. Скорее это абстрагированное понятие указателя на массив. Понятия «нулевой итератор» не существует. Проверку, указывает ли итератор на некоторый элемент или нет, принято делать путем сравнения его с *концом* данной последовательности (а не с ***null***). Это упрощает многие алгоритмы, избавляя от необходимости особого учета конца, и хорошо обобщается до последовательностей произвольных типов.

Итератор, указывающий на какой-то элемент, называется *действительным* и через него можно получить доступ к элементу (при помощи *, [] или ->). Итератор может быть недействительным потому, что:

- не был инициализирован;
- указывает на контейнер, который явно или неявно изменил свои размеры (§ 16.3.6, § 16.3.8),
- контейнер, на который он указывает, уничтожен;
- итератор обозначает конец последовательности.

Конец последовательности можно представлять себе как итератор, указывающий на воображаемый элемент, следующий за последним элементом в последовательности (one-past-the-last element) (§ 18.2).

19.2.1. Операции с итераторами

Не все виды итераторов поддерживают один и тот же набор операций. Например, чтение требует других операций по сравнению с записью, а ***vector*** позволяет получить удобный и эффективный произвольный доступ способом, который был бы очень дорогим для ***list*** или ***istream***. Поэтому мы разобьем итераторы на пять категорий в

соответствии с операциями, которые они могут эффективно обеспечивать (то есть за постоянное время; § 17.1).

Итераторы: операции и категории

Категория: (**Перевод:**)	output для записи	input для чтения	forward однонаправленный	bidirectional двунаправленный	random-access с произвольным доступом
Сокращение:	***Out***	***In***	***For***	***Bi***	***Ran***
Чтение:		=*p	=*p	=*p	=*p
Доступ:		->	->	->	-> []
Запись:	*p=		*p=	*p=	*p=
Итерация:	++	++	++	++ --	++ -- + - += -=
Сравнение:		== !=	== !=	== !=	== != < > >= <=

И чтение, и запись производятся через итератор, разыменовываемый при помощи * :

```
*p = x;     // запись x через p
x = *p;     // считывание в x через p
```

Чтобы быть итераторным, тип данных должен обеспечивать соответствующий набор операций. Эти операции должны иметь именно тот смысл, который обычно подразумевается. То есть каждая операция должна производить тот же эффект, какой она производит на обыкновенный указатель.

Независимо от своей категории, итератор может позволять как константный, так и неконстатный доступ к объекту, на который указывает. При помощи итератора на ***const*** вы не можете писать в элемент, к какой бы категории итератор не относился. Итератор обеспечивает набор операций, но окончательным судьей, решающим, что можно сделать с элементом, является тип элементов, на которые указывает итератор.

При чтении и записи объекты копируются, поэтому типы элементов должны иметь обычную семантику копирования (§ 17.1.4).

Прибавлять (или вычитать) целое для относительной адресации можно только применительно к итераторам с произвольным доступом. Однако, если не считать итераторов для записи, расстояние между двумя итераторами всегда можно найти путем перебора (итерирования) элементов. Для этого предназначена функция ***distance*** ():

```
template<class In> typename iterator_traits<In>::
        difference_type distance (In first, In last)
{
        typename iterator_traits<In>::difference_type d = 0;
        while (first++!=last) d++;
        return d;
}
```

Чтобы выражать расстояние между элементами, для всех итераторов ***In*** определен тип ***iterator_traits<In>::difference_type***, (§ 19.2.2).

Функция, приведенная выше, названа ***distance*** (), а не ***operator-*** (), потому что операция, выполняемая ею, довольно дорога, а все необходимые для итератора операторы выполняются за постоянное время (§ 17.1). Подсчет элементов один за другим — это не тот вид операции, который я могу позволить себе нечаянно вызвать для большой

последовательности. Для итераторов с произвольным доступом библиотека обеспечивает гораздо более эффективную реализацию функции ***distance*** ().

Подобным образом предоставляется функция ***advance*** () в качестве потенциально медленного оператора +=:

```
template<class In, class Dist> void advance (In& I, Dist n); // i+=n
```

19.2.2. Свойства итераторов

Мы пользуемся итераторами, чтобы добраться до информации об объекте, на который они указывают, и о последовательности, на элементы которой они указывают. Например, через итератор мы можем косвенно обратиться к объекту и манипулировать им, а также можем найти номер элемента в последовательности по описывающему его итератору. Чтобы выразить такие операции, нам нужно уметь работать с типами, возникающими при рассмотрении итераторов, как, например, «тип объекта, на который ссылается итератор» и «тип, обозначающий расстояние между двумя итераторами». Связанные с итератором типы описываются небольшим набором объявлений в шаблоне класса ***iterator_traits***:

```
template<class Iter> struct iterator_traits {
        typedef typename Iter::iterator_category iterator_category;     // § 19.2.3
        typedef typename Iter::value_type value_type;                   // тип элемента
        typedef typename Iter::difference_type difference_type;
        typedef typename Iter::pointer pointer;     // возвращаемый тип оператора ->
        typedef typename Iter::reference reference; // возвращаемый тип оператора *()
};
```

Тип ***difference_type*** — это тип для выражения разности между двумя итераторами, а ***iterator_category*** — это тип, показывающий, какие операции итератор поддерживает. Для обыкновенных указателей предоставляются специализации (§ 13.5) для <***T****> и <***const T****>. В частности:

```
template<class T> struct iterator_traits<T*> { // специализация для указателей
        typedef random_access_iterator_tag iterator_category;
        typedef T value_type;
        typedef ptrdiff_t difference_type;
        typedef T* pointer;
        typedef T& reference;
};
```

То есть разность между двумя указателями представляется стандартным библиотечным типом ***ptrdiff_t*** из <***cstddef***> (§ 6.2.1), и указатель обеспечивает произвольный доступ (§ 19.2.3). Имея тип ***iterator_traits***, мы можем написать программу, зависящую от параметра-итератора. Классический пример — алгоритм ***count*** () (сосчитать):

```
template<class In, class T>
typename iterator_traits<In>::difference_type count (In first, In last, const T& val)
{
        typename iterator_traits<In>::difference_type res = 0;
        while (first != last) if (*first++==val) ++res;
        return res;
}
```

Здесь тип результата выражается через *iterator_traits* входа. Причина этого состоит в том, что не существует примитивов языка, которые могли бы непосредственно выразить один произвольный тип через комбинации других типов. В частности, нет способа непосредственно выразить следующее: «тип, является результатом вычитания двух типов *In*».

Вместо *iterator_traits* мы могли бы использовать для указателей специализированный счетчик *count* ():

```
template<class In, class T>
    typename In:: difference_type count (In first, In last, const T& val);

template<class In, class T>
    ptrdiff_t count<T*, T> (T* first, T* last, const T& val);
```

Однако так бы решилась проблема только для *count* (). Если бы мы воспользовались этим приемом для дюжины алгоритмов, информация о типах расстояния вызвала бы дюжину репликаций, а, как правило, лучше представлять принятое проектное решение в одном месте (§ 23.4.2). Тогда, при необходимости, и изменить его можно в одном месте.

Так как тип *iterator_traits<Iterator>* определен для всех итераторов, мы неявно определяем *iterator_traits* каждый раз, когда создаем новый итераторный тип. Если свойства (traits) по умолчанию, порожденные из универсального шаблона *iterator_traits*, не годятся для нашего нового итераторного типа, мы обеспечиваем специализацию примерно так же, как это делается для типов указателей в стандартной библиотеке. Порожденный неявно тип *iterator_traits* предполагает, что итератор является классом с типами членов *difference_type*, *value_type* и т. п. В заголовочном файле *<iterator>* стандартной библиотеки введен базовый тип, который можно использовать для определения типов этих членов:

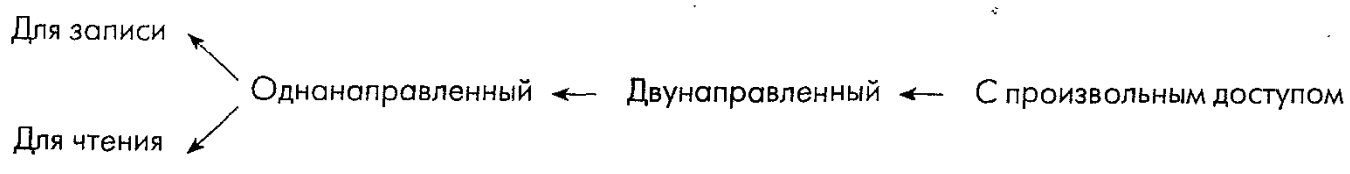
```
template<class Cat, class T, class Dist = ptrdiff_t, class Ptr = T*, class Ref = T&>
struct iterator {
    typedef Cat iterator_category;      // § 19.2.3
    typedef T value_type;               // тип элемента
    typedef Dist difference_type;       // тип разности итераторов
    typedef Ptr pointer;                // возвращаемый тип оператора ->
    typedef Ref reference;              // возвращаемый тип оператора *
};
```

Отметим, что *reference* и *pointer* — не итераторы. Они предназначены для того, чтобы быть типами значений, возвращаемых соответственно операторами *operator** () и *operator ->* (), для некоторого итератора.

Тип *iterator_traits* является ключом к упрощению многих опирающихся на итераторы интерфейсов и к эффективной реализации многих алгоритмов.

19.2.3. Категории итераторов

Различные виды итераторов — обычно называемые категориями итераторов — укладываются в следующую иерархию:



Это не диаграмма наследования классов. Категория итераторов — это характеристика типа, основанная на том, какие операции он обеспечивает. Многие, в других отношениях не связанные между собой типы, могут относиться к одной и той же категории итераторов. Например, и обыкновенные указатели (§ 19.2.2), и объекты класса ***Checked_iter*** (§ 19.3) являются итераторами с произвольным доступом.

Как отмечено в главе 18, различные алгоритмы требуют в качестве аргумента итераторов различных видов. Кроме того, один и тот же алгоритм для различных видов итераторов иногда можно реализовать с различной эффективностью. Чтобы в случае перегрузки компилятор мог различать алгоритмы, основываясь на категориях итераторов, стандартная библиотека обеспечивает пять классов, представляющих пять категорий итераторов:

```
struct input_iterator_tag { };                  // для чтения
struct output_iterator_tag { };                 // для записи
struct forward_iterator_tag :
        public input_iterator_tag { };          // однонаправленный
struct bidirectional_iterator_tag :
        public forward_iterator_tag { };        // двунаправленный
struct random_access_iterator_tag :
        public bidirectional_iterator_tag { };  // с произвольным доступом
```

Глядя на операции, поддерживаемые итераторами для чтения и итераторами для записи (§ 19.2.1), мы можем предположить, что ***forward_iterator_tag*** (последовательный итератор) должен быть производным от ***output_iterator_tag*** и ***input_iterator_tag***. Причины почему это не так, достаточно запутанны и, вероятно, не совсем логичны. Однако, я еще не видел примера, где бы такое наследование упростило реальную программу.

Наследование для всех этих классов-ярлыков (tag — «ярлык») полезно (только) для того, чтобы избавиться от определения различных версий функций, в которых несколько видов итераторов — но не все — могут использовать одни и те же алгоритмы. Рассмотрим, как реализовать ***distance***:

```
template<class In>
typename iterator_traits<In>::difference_type distance (In first, In last);
```

Есть две очевидные альтернативы:

[1] Если ***In*** — это итератор с произвольным доступом, мы можем вычесть ***first*** из ***last***.

[2] В противном случае мы должны инкрементировать итератор от ***first*** до ***last*** и подсчитывать расстояние.

Эти две альтернативы можно выразить парой функций-помощников:

```
template<class In>
typename iterator_traits<In>::difference_type
dist_helper (In first, In last, input_iterator_tag)
{
        typename iterator_traits<In>::difference_type d = 0;
        while (first++!=last) d++;      // используем только инкремент
        return d;
}

template<class Ran>
typename iterator_traits<In>::difference_type
```

```
dist_helper(Ran first, Ran last, random_access_iterator_tag)
{
    return last - first;          // опираемся на произвольный доступ
}
```

Категории итераторов, указанные в качестве аргументов, явно определяют, какой из итераторов ожидается. Категория итератора используется исключительно для разрешения имен при перегрузке; она не принимает участия в действительных вычислениях. Этот механизм относится исключительно к компиляции. Вдобавок к автоматическому выбору функции-помощника этот прием обеспечивает непосредственную проверку типа (§ 13.2.5).

Теперь очень просто определить ***distance*** () путем вызова соответствующей функции-помощника:

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last)
{
    return dist_helper(first, last, iterator_traits<In>::iterator_category());
}
```

Чтобы вызвать ***dist_helper***, используемая категория ***iterator_traits<In>::iterator_category*** должна быть либо ***input_iterator_tag*** либо ***random_access_iterator_tag***. Однако для последовательных и двунаправленных итераторов нет необходимости делать две различные версии ***dist_helper*** (). Благодаря наследованию классов категорий эти случаи обрабатываются функцией-помощником ***dist_helper*** (), которая в качестве аргумента принимает ***input_iterator_tag***. Отсутствие версии с ***output_iterator_tag*** отражает тот факт, что применительно к итераторам для записи ***distance*** () не имеет смысла:

```
void f(vector<int>& vi,
       list<double>& ld,
       istream_iterator<string>& is1, istream_iterator<string>& is2,
       ostream_iterator<string>& os1, ostream_iterator<string>& os2)
{
    distance(vi.begin(), vi.end());     // используем алгоритм с вычитанием
    distance(ld.begin(), ld.end());     // используем алгоритм с инкрементом
    distance(is1, is2);                 // используем алгоритм с инкрементом
    distance(os1, os2);                 // ошибка: неправильная категория итератора,
                                        // тип аргумента для dist_helper неверен
}
```

Впрочем, в реальной программе вызов ***distance*** () для ***istream_iterator***, вероятно, не имеет большого смысла. Эффект будет таков: чтение входа, выбрасывание входных значений и возвращение числа выброшенных значений.

Использование ***iterator_traits<T>::iterator_category*** позволяет программисту обеспечить альтернативную реализацию, чтобы пользователь, которому нет никакого дела до реализации алгоритмов, автоматически получал самую подходящую реализацию для каждой используемой структуры данных. Иными словами, это позволяет нам скрыть детали реализации за соответствующим интерфейсом. Для гарантии того, что это изящество достигается не ценой быстродействия, можно воспользоваться встраиванием.

19.2.4. Вставки (inserters)

Вывод данных через итератор в контейнер приводит к тому, что элементы, следующие за тем, на которые указывает итераторы, могут быть перезаписаны. Это может также привести к переполнению и, следовательно, к порче памяти. Например:

```
void f(vector<int>& vi)
{
    fill_n (vi.begin (), 200, 7);      // присваивание 7 всем элементам vi[0]..vi[199]
}
```

Если в *vi* меньше 200 элементов, будет плохо.

В заголовочном файле *<iterator>* стандартная библиотека предлагает решение этой проблемы: три шаблона классов итераторов плюс три функции (вставки), чтобы эти итераторы было удобно использовать:

```
template<class Cont> back_insert_iterator<Cont> back_inserter (Cont& c);
template<class Cont> front_insert_iterator<Cont> front_inserter (Cont& c);
template<class Cont, class Out> insert_iterator<Cont> inserter (Cont& c, Out p);
```

Функция ***back_inserter*** () добавляет элементы в конец контейнера, ***front_inserter*** () добавляет элементы в начало, а «просто» вставка ***inserter*** () добавляет элементы перед своим аргументом-итератором. В ***inserter*** (*c*, *p*) *p* должно быть действительным итератором для *c*. Естественно, с каждой вставкой значения контейнер растет.

При записи вставка вставляет новый элемент в последовательность с использованием ***push_back*** (), ***push_front*** () или ***insert*** (), вместо того, чтобы перезаписывать существующий. Например:

```
void g (vector<int>& vi)
{
    fill_n (back_inserter (vi), 200, 7);    // добавление 200 семерок в конец vi
}
```

Вставки столь же просты и эффективны, сколь и полезны. Например:

```
template<class Cont>
class insert_iterator : public iterator<output_iterator_tag, void, void, void, void> {
protected:
    Cont& container;                    // контейнер, в который вставляем
    typename Cont::iterator iter;       // указывает на элементы контейнера
public:
    explicit insert_iterator (Cont& x, typename Cont::iterator i)
        : container (x), iter (i) { }

    insert_iterator& operator= (const typename Cont::value_type& val)
    {
        iter = container.insert (iter, val);
        ++iter;
        return *this;
    }

    insert_iterator& operator* () { return *this; }
    insert_iterator& operator++ () { return *this; }        // префиксный ++
```

```
        insert_iterator operator++ (int) { return *this; }        // постфиксный ++
};
```

Ясно, что вставки — это итераторы для записи.

insert_iterator — это особый случай выходной последовательности. По аналогии с *iseq* из § 18.3.1, мы можем определить:

```
template<class Cont>
insert_iterator<Cont>
oseq (Cont& c, typename Cont::iterator first, typename Cont::iterator last)
{
        // erase() объясняется в § 16.3.6
        return insert_iterator<Cont> (c, c.erase (first, last));
}
```

Другими словами, выходная последовательность удаляет свои старые элементы и заменяет их выходными значениями. Например:

```
void f(list<int>& li, vector<int>& vi)
// заменяет вторую половину vi копией из li
{
        copy (li.begin (), li.end (), oseq (vi, vi.begin ()+vi.size ()/2, vi.end ()));
}
```

Аргументом *oseq* должен быть контейнер, поскольку уменьшить размеры контейнера невозможно, имея только итератор на его элементы (§ 18.6, § 18.6.3).

19.2.5. Обратные итераторы

Стандартные контейнеры обеспечивают операции *rbegin* () и *rend* () для перебора элементов в обратном порядке (§ 16.3.2). Эти функции-члены возвращают итераторы типа *reverse_iterator*:

```
template<class Iter>
class reverse_iterator : public iterator<typename iterator_traits<Iter>::iterator_category,
                                         typename iterator_traits<Iter>::value_type,
                                         typename iterator_traits<Iter>::difference_type,
                                         typename iterator_traits<Iter>::pointer,
                                         typename iterator_traits<Iter>::reference> {
protected:
        Iter current;                   // current указывает на элемент,
                                        // следующий за тем, на который ссылается *this
public:
        typedef Iter iterator_type;

        reverse_iterator () : current () { }
        explicit reverse_iterator (Iter x) : current (x) { }
        template<class U> reverse_iterator (const reverse_iterator<U>& x) : current (x.base ()) { }

        Iter base () const { return current; }        // значение текущего итератора

        reference operator* () const { Iter tmp=current; return* --tmp; }
        pointer operator-> () const;
```

```
        reference operator[] (difference_type n) const;

        reverse_iterator& operator++ () { --current; return *this ; }  // отметите: не ++
        reverse_iterator operator++ (int) { reverse_iterator t=current; --current; return t ; }
        reverse_iterator& operator-- () { ++current; return *this ; }  // отметите: не --
        reverse_iterator operator-- (int) { reverse_iterator t=current; ++current; return t ; }

        reverse_iterator operator+ (difference_type n) const;
        reverse_iterator& operator+= ( difference_type n);
        reverse_iterator operator- ( difference_type n)const;
        reverse_iterator& operator-= ( difference_type n);
};
```

Обратный итератор реализуется с использованием итератора, называемого *текущим* (current). Такой итератор может указывать (только) на элементы своей последовательности и на элемент, следующий за последним. Однако, для обратного итератора элемент, следующий за последним, это элемент (недоступный), следующий перед первым в исходной последовательности. Чтобы избежать нарушения доступа, текущий итератор указывает на элемент, следующий за тем, на который ссылается обратный итератор. Это приводит к тому, что * возвращает значение * (***current–1***), и ++ реализуется через --, применяемый к текущему итератору.

Обратный итератор поддерживает операции, поддерживаемые его инициализатором (и только их). Например:

```
void f(vector<int>& v, list<char>& lst)
{
        // правильно: итератор с произвольным доступом
        v.rbegin ()[3] = 7;
        // ошибка: двунаправленный итератор не поддерживает оператор []
        lst.rbegin ()[3] = '4';
        * (++++++lst.rbegin ()) = '4';   // правильно!
}
```

Кроме того, для обратных итераторов библиотека обеспечивает обычные операций сравнения: ==, !=, <, <=, > и >=.

19.2.6. Итераторы потоков

Обычно ввод/вывод выполняется при помощи библиотеки потоков (глава 21), системы графического пользовательского интерфейса (не определяемой стандартом C++) или функциями ввода-вывода языка C (§ 21.8). Эти интерфейсы ввода/вывода в первую очередь направлены на считывание и запись отдельных значений разнообразных типов. Чтобы согласовать операции ввода/вывода с механизмом контейнеров и алгоритмов, стандартная библиотека обеспечивает четыре итераторных типа:

- ***ostream_iterator*** для записи в ***ostream*** (§ 3.4, § 21.2.1);
- ***istream_iterator*** для чтения из ***istream*** (§ 3.6, § 21.3.1);
- ***ostreambuf_iterator*** для записи в буфер потока (§ 21.6.1);
- ***istreambuf_iterator*** для чтения из буфера потока (§ 21.6.2).

Идея заключается просто в том, чтобы представить входные и выходные потоки в виде последовательностей:

```
template<class T, class Ch = char, class Tr = char_traits<Ch> >
class ostream_iterator : public iterator<output_iterator_tag, void, void, void, void> {
public:
        typedef basic_ostream<Ch, Tr> ostream_type;
        typedef Ch char_type;
        typedef Tr traits_type;

        ostream_iterator (ostream_type& s);
        ostream_iterator (ostream_type& s, const Ch* delim);
        ostream_iterator (const ostream_iterator&);
        ~ostream_iterator ();

        ostream_iterator& operator= (const T& val);          // запись в поток вывода

        ostream_iterator& operator* ();
        ostream_iterator& operator++ ();
        ostream_iterator& operator++ (int);
}
```

Этот итератор принимает обычные операции записи и инкрементирования итератора для записи и преобразует их в операции вывода в ***ostream***. Например:

```
void f()
{
        ostream_iterator<int> os (cout);        // запись целых в cout через os
        *os = 7;                                // вывод 7
        ++os;                                   // готово к следующему выводу
        *os = 79;                               // вывод 79
}
```

Операция ++ может осуществить действительную операцию вывода, а может не произвести никакого эффекта. Различные реализации используют разные стратегии. Поэтому, чтобы программа была переносимой, оператор ++ должен появляться между каждыми двумя присваиваниями итератору ***ostream_iterator***. Естественно, все стандартные алгоритмы написаны таким образом — иначе они бы не работали с контейнерами ***vector***. Вот почему ***ostream_iterator*** определен так, как показано выше.

Реализация ***ostream_iterator*** тривиальна, и я оставляю ее в качестве упражнения (§ 19.6[4]). Стандартный ввод/вывод поддерживает различные типы символов; ***char_traits*** (§ 20.2) описывают те аспекты символьных типов, которые могут оказаться важными для ввода/вывода и строк ***string***.

Итератор для чтения для потоков ***istream*** определяется аналогично:

```
template<class T, class Ch = char, class Tr = char_traits<Ch>, class Dist = ptrdiff_t >
class istream_iterator : public iterator<intput_iterator_tag, T, Dist, const T*, const T&> {
public:
        typedef Ch char_type;
        typedef Tr traits_type;
        typedef basic_istream<Ch, Tr> istream_type;
```

```
        istream_iterator();                          // конец ввода
        istream_iterator(istream_type& s);
        istream_iterator(const istream_iterator& x);
        ~istream_iterator();

        const T& operator*() const;
        const T* operator->() const;
        istream_iterator& operator++();
        istream_iterator operator++(int);
};
```

Этот итератор определен так, чтобы обычные операции с контейнером >> приводили к чтению из потока *istream*. Например:

```
void f()
{
        istream_iterator<int> is(cin);          // чтение целых из cin через is
        int i1 = *is;                           // чтение int
        ++is;                                   // готово к следующему вводу
        int i2 = *is;                           // чтение int
}
```

Итератор *istream_iterator* по умолчанию представляет конец ввода, так что мы можем задать входную последовательность:

```
void f(vector<int>& v)
{
        copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
}
```

Чтобы это заработало, стандартная библиотека обеспечивает для итераторов *istream_iterator* операторы == и !=.

Реализация *istream_iterator* не так тривиальна, как *ostream_iterator*, но тоже проста. Я ее оставляю в качестве упражнения (§ 19.6[5]).

19.2.6.1. Буфера потоков

Как описывается в § 21.6, поток ввода/вывода основан на идее о потоках *istream* и *ostream*, заполняющих и освобождающих буфера, ввод/вывод из которых реализован на низком физическом уровне. Можно обойти форматирование стандартных потоков ввода/вывода и работать напрямую с буферами потоков (§ 21.6.4). Такая возможность также обеспечивается для алгоритмов через итераторы *istreambuf_iterator* и *ostreambuf_iterator*:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator
        : public iterator<input_iterator_tag, Ch, typename Tr::offtype, Ch*, Ch&> {
public:
        typedef Ch char_type;
        typedef Tr traits_type;
        typedef typename Tr::int_type int_type;
        typedef basic_streambuf<Ch, Tr> streambuf_type;
        typedef basic_istream<Ch, Tr> istream_type;
```

```
        class proxy;                                    // вспомогательный тип

        istreambuf_iterator() throw();                  // конец буфера
        istreambuf_iterator(istream_type& is) throw();  // чтение из streambuf объекта is
        istreambuf_iterator(streambuf_type*) throw();
        istreambuf_iterator(const proxy& p) throw();    // чтение из streambuf объекта p
        Ch operator*() const;
        istreambuf_iterator& operator++();              // префиксный оператор
        proxy operator++(int);                          // постфиксный оператор

        // оба или ни один из streambuf в конце файла (eof)
        bool equal(istreambuf_iterator&);
};
```

Кроме того обеспечиваются операторы ++ и !=.

Считывание из ***streambuf*** — это операция более низкого уровня, чем считывание из ***istream***, и следовательно, интерфейс итератора ***istreambuf_iterator*** сложнее, чем ***istream_iterator***. Однако, когда ***istreambuf_iterator*** должным образом инициализирован, операторы *, ++ и = имеют свое обычное значение, если их использовать соответствующим образом.

Тип ***proxy*** (заместитель) — это определенный при реализации вспомогательный тип, который позволяет реализовать постфиксный оператор ++ без введения ограничений на реализацию ***streambuf***. При инкременте итератора тип ***proxy*** содержит результирующее значение:

```
template<class Ch, Class Tr = char_traits<Ch> >
class istreambuf_iterator<Ch, Tr>::proxy {
        Ch val;
        basic_streambuf<Ch, Tr>* buf;

        proxy(Ch v, basic_streambuf<Ch, Tr>* b) : val(v), buf(b) { }
public:
        Ch operator*() { return val; }
};
```

ostreambuf_iterator определяется похожим образом:

```
template<class Ch, class Tr = char_traits<Ch> >
class ostreambuf_iterator : public iterator<output_iterator_tag, void, void, void, void> {
public:
        typedef Ch char_type;
        typedef Tr traits_type;
        typedef basic_streambuf<Ch, Tr> streambuf_type;
        typedef basic_ostream<Ch, Tr> ostream_type;

        ostreambuf_iterator(ostream_type& os) throw();      // запись в буфер потока os
        ostreambuf_iterator(streambuf_type*) throw();
        ostreambuf_iterator& operator=(Ch);

        ostreambuf_iterator& operator*();
        ostreambuf_iterator& operator++();
        ostreambuf_iterator& operator++(int);

        bool failed() const throw();                        // true, если встретили Tr::eof()
};
```

19.3. Итераторы с проверкой (checked_iter)

Кроме обеспечиваемых стандартной библиотекой, программист может создать собственные итераторы. Это часто необходимо при создании нового вида контейнера, а иногда новый итератор является хорошим способом поддержки особого использования существующих контейнеров.

Использование стандартных контейнеров снижает затраты на явное распределение памяти. Использование стандартных алгоритмов снижает затраты на явную адресацию элементов в контейнере. Использование стандартной библиотеки и возможностей языка, поддерживающих безопасность с точки зрения типов, очень значительно снижает количество не выявляемых компилятором ошибок по сравнению с традиционным стилем языка C. Однако, задачу предотвращения обращения за пределы контейнера стандартная библиотека по-прежнему оставляет программисту. Если обратиться к элементу некоторого контейнера ***x*** в стиле ***x[x.size ()+7]***, случится нечто непредсказуемое и, как правило, нехорошее. Во многих случаях помогает использование контейнеров ***vector*** с проверкой диапазона, таких как ***Vec*** (§ 3.7.2). В большинстве случаев каждое обращение проверяется через итератор.

Чтобы достичь такого уровня проверки, не возлагая на программиста тяжелого бремени новых обозначений, нам нужны итераторы с проверкой и удобный способ прикрепления их к контейнерам. Как для связывателей (§ 18.4.4.1), вставок (§ 19.2.4) и т. п., для создания ***Checked_iter*** я написал функции:

```
template<class Cont, class Iter> Checked_iter<Cont, Iter> make_checked (Cont& c, Iter i)
{
    return Checked_iter<Cont, Iter> (c, i);
}

template<class Cont> Checked_iter<Cont, typename Cont::iterator>
    make_checked (Cont& c)
{
    return Checked_iter<Cont, typename Cont::iterator> (c, c.begin ());
}
```

Эти функции очень удобны с точки зрения обозначений, позволяя выводить типы из аргументов, а не объявлять такие типы явно. Например:

```
void f(vector<int>& v, const vector<int>& vc)
{
    typedef Checked_iter<vector<int>, vector<int>::iterator> CI;
    CI p1 = make_checked (v, v.begin ()+3);
    CI p2 = make_checked (v);     // по умолчанию: указывает на первый элемент

    typedef Checked_iter<const vector<int>, vector<int>::const_iterator> CIC;
    CIC p3 = make_checked (vc, vc.begin ()+3);
    CIC p4 = make_checked (vc);

    const vector<int>& vv = v;
    CIC p5 = make_checked (v, vv.begin ());
}
```

По умолчанию константные контейнеры имеют константные итераторы, так что их ***Checked_iter*** тоже должны быть константными итераторами. Итератор ***p5*** показывает один из способов получения константного итератора из неконстантного контейнера.

Это демонстрирует, почему ***Checked_iter*** нужно два параметра шаблона: один — для типа контейнера, а второй — чтобы выразить различие константный/неконстантный.

Имена этих типов ***Checked_iter*** становятся довольно длинными и громоздкими, но это не имеет значения, когда итераторы используются как аргументы в обобщенных алгоритмах. Например:

```
template<class Iter> void mysort (Iter first, Iter last);

void f(vector<int>& c)
{
    try {
        mysort (make_checked (c), make_checked (c, c.end ()));
    }
    catch (out_of_bounds) {
        cerr << "Ага: в mysort () ошибка\n";
        abort ();
    }
}
```

Первоначальная версия такого алгоритма располагается именно там, где я более всего склонен подозревать возможную ошибку выхода за пределы диапазона, так что использование итератора с проверкой имеет смысл.

Представлением ***Checked_iter*** является указатель на контейнер и итератор, указывающий внутрь контейнера:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
    Iter curr;   // итератор для текущей позиции
    Cont* c;     // указатель на текущий контейнер
    // ...
};
```

Одним из приемов для определения желаемых типов является создание типов, производных от ***iterator_traits***. Очевидная альтернатива — создание типов, производных от ***iterator*** — в данном случае будет слишком многословной (как это было для ***reverse_iterator***; § 19.2.5). Поскольку нет требования, чтобы итератор был классом, и нет требования, чтобы итераторы, которые являются классами, происходили бы от ***iterator***.

Все операции в ***Checked_iter*** довольно тривиальны:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
    // ...
public:
    void valid (Iter p) const
    {
        if (c->end () == p) return;
        for (Iter pp=c->begin (); pp!=c->end (); ++pp) if (pp==p) return;
        throw out_of_bounds ();
    }

    friend bool operator== (const Checked_iter& i, const Checked_iter& j)
        { return i.c==j.c && i.curr==j.curr; }

    // нет инициализатора по умолчанию
```

```
// используем копирующий конструктор по умолчанию
// и копирующее присваивание по умолчанию

Checked_iter(Cont& x, Iter p) : c(&x), curr(p) { valid(p); }

Checked_iter(Cont& x) : c(&x), curr(x.begin()) { }

reference operator* () const
{
    if (curr==c->end()) throw out_of_bounds();
    return *curr;
}

pointer operator-> () const
{
    return &*curr;                          // проверяется оператором *
}

Checked_iter operator+ (difference d) const  // только для итераторов
                                             // с произвольным доступом
{
    if (c->end()-curr<d || d<-(curr-c->begin())) throw out_of_bounds();
    return Checked_iter(c, curr+d);
}

reference operator[] (difference d) const    // только для итераторов
                                             // с произвольным доступом
{
    if (c->end()-curr<=d || d<-(curr-c->begin())) throw out_of_bounds();
    return curr[d];
}

Checked_iter& operator++ ()                  // префиксный оператор ++
{
    if (curr==c->end()) throw out_of_bounds();
    ++curr;
    return *this;
}

Checked_iter& operator++ (int)               // постфиксный оператор ++
{
    Checked_iter tmp = *this;
    ++*this;                                 // проверяется префиксным ++
    return tmp;
}

Checked_iter& operator-- ()                  // префиксный оператор --
{
    if (curr == c->begin()) throw out_of_bounds();
    --curr;
    return *this;
}

Checked_iter& operator-- (int)               // постфиксный оператор --
{
    Checked_iter tmp = *this;
    --*this;                                 // проверяется префиксным --
    return tmp;
}
```

```
        difference_type index() const          // только для произвольного доступа
            { return curr-c.begin(); }

        Iter unchecked() const { return curr; }

        //+, -, <и т. п. (§ 19.6[6])
};
```

Checked_iter может быть инициализирован только для конкретного итератора, указывающего на конкретный контейнер. В полноценной реализации должна быть обеспечена более эффективная версия функции ***valid*** () для итераторов с произвольным доступом (§ 19.6[6]). Когда ***Checked_iter*** инициализирован, все операции, изменяющие позицию итератора, проверяются: действительно ли итератор по-прежнему указывает внутрь контейнера. Попытка заставить итератор указать за пределы контейнера приводит к генерации исключения ***out_of_bounds***. Например:

```
void f(list<string>& ls)
{
    int count=0;
    try {
        Checked_iter<list<string> > p(ls, ls.begin());
        while (true) {
            ++p;              // рано или поздно мы достигнем конца
            ++count;
        }
    }
    catch (out_of_bounds) {
        cout << "выход за пределы после " << count << " проходов\n"
    }
}
```

Итератор ***Checked_iter*** знает, в какой контейнер указывает, и это позволяет ему отловить некоторые (но не все) случаи, когда итераторы в контейнере становятся недействительными в результате операций над ним (§ 16.3.6, § 16.3.8). Для полной защиты от таких случаев может понадобиться разработать другой, менее экономный итератор (см. § 19.6[7]).

Отметим, что пост-инкремент (постфиксный оператор ++) использует промежуточную переменную, а пре-инкремент (префиксный оператор ++) нет. По этой причине, для итераторов предпочтительнее ++*p*, а не *p*++.

Поскольку ***Checked_iter*** содержит указатель на контейнер, его нельзя использовать для встроенных массивов напрямую. При необходимости можно воспользоваться массивом ***c_array*** (§ 17.5.4).

Для завершения темы об итераторах с проверкой мы должны сделать их простыми в применении. Здесь есть два основных подхода:

[1] Определить тип контейнера с проверкой, ведущего себя так же, как другие контейнеры, но обеспечивающего лишь ограниченный набор конструкторов, и его ***begin*** (), ***end*** () и т. п. предоставляют не обычные итераторы, а ***Checked_iter***.

[2] Определить дескриптор, который может быть инициализирован произвольным контейнером, и который обеспечивает «проверяющие» функции доступа к своему контейнеру (§ 19.6[8]).

Следующий шаблон соединяет итератор с проверкой и контейнер:

```
template<class C> class Checked : public C {
public:
    explicit Checked (size_t n) : C (n) { }
    Checked () : C () { }

    typedef Checked_iter<C> iterator;
    typedef Checked_iter<C, C::const_iterator> const_iterator;

    iterator begin () { return iterator (*this, C::begin ()); }
    iterator end () { return iterator (*this, C::end ()); }
    const_iterator begin () const { return const_iterator (*this, C::begin ()); }
    const_iterator end () const { return const_iterator (*this, C::end ()); }

    typename C::reference_type operator[] (typename C::size_type n)
        { return Checked_iter<C> (*this) [n]; }

    C& base () { return this; }          // получили базовый контейнер
}
```

Это позволяет нам написать:

```
Checked<vector<int> > vec (10);
Checked<list<double> > lst;

void f ()
{
    int v1 = vec[5];                     // все в порядке
    int v2 = vec[15];                    // возбуждает out_of_bounds
    // ...
    lst.push_back (v2);
    mysort (vec.begin (), vec.end ());
    copy (vec.begin (), vec.end (), lst.begin ());
}
```

Очевидно, избыточная функция ***base*** () позволяет сделать интерфейс функции ***Checked*** () аналогичным интерфейсу дискриптора (вспомогательного класса) в контейнерах. Дискриптор контейнера обычно не обеспечивает неявного преобразования в контейнеры. Если контейнер изменил свои размеры, все итераторы в нем — в том числе и с проверкой — могут стать недействительными. В этом случае итераторы типа ***Checked_iter*** могут быть инициализированы заново:

```
void g (vector<int> vi)
{
    Checked_iter<vector<int>> p (vi, vi.begin ());
    // ...
    int i = p.index ();                          // получаем текущую позицию
    vi.resize (100);                             // p становится недействительным
    p = Checked_iter<vector<int>> (vi, vi.begin ()+i);   // восстанавливаем
                                                 // текущую позицию
}
```

Старая — и недействительная теперь — текущая позиция теряется. Я предоставил функцию ***index*** () как средство для извлечения индекса, что позволяет восстанавливать итераторы с проверкой.

19.3.1. Исключения, контейнеры и алгоритмы

Вы могли бы возразить, что использование и стандартных алгоритмов, и итераторов с проверкой напоминает одновременное ношение ремня и подтяжек: безопасность обеспечивает и то, и другое. Однако опыт показывает, что для многих людей и многих прикладных программ доза паранойи не помешает — особенно, когда в программу часто вносятся изменения несколькими людьми.

Один из способов проверки во время выполнения — это держать в коде проверочные фрагменты. Затем, перед сдачей программы, они удаляются. Эту практику сравнивают с надеванием спасательного жилета при плавании вблизи берега и сниманием при выходе в открытое море. Однако, иногда применение проверок во время выполнения действительно ведет к значительным затратам времени и памяти, так что настаивать на них не реалистично. Но, во всяком случае, не очень мудро оптимизировать программу, не проводя измерений, так что прежде чем удалять проверочный код, проведите эксперимент и посмотрите, действительно ли стоит подобным образом улучшать программу. Чтобы провести такой эксперимент, мы должны иметь возможность легко удалять проверки, предназначенные для времени выполнения (см. § 24.3.7.1). Когда оценка произведена, мы можем удалить проверки там, где это наиболее критично с точки зрения быстродействия — и в надежде, что все тщательно протестировали, — а остальные оставить как сравнительно дешевую форму страховки.

Использование итераторов с проверкой позволяет нам выявить многие ошибки. Однако оно не помогает нам легко избавиться от них. Люди редко пишут программы, которые на 100% нечувствительны ко всем ++, --, *, [], -> и =, потенциально генерирующим исключения. Это оставляет нам две очевидные стратегии:

[1] Перехватывать (***catch***) исключения рядом с точками, в которых они генерируются , чтобы обработчик исключений имел шанс узнать, в чем ошибка и предпринять соответствующие действия.

[2] Перехватывать исключения на верхнем уровне программы, прерывая изрядную долю вычислений и оставляя под подозрением все структуры данных, участвовавшие в неудавшемся вычислении (таких структур данных может не оказаться, или их благонадежность можно проверить).

Перехватывать исключения из какой-то неизвестной части программы и продолжать работу, предполагая, что все структуры данных остались в нормальном состоянии, было бы несерьезно, если нет обработки ошибок на следующем уровне, которая бы перехватила соответствующие ошибки. Простой пример этого — окончательная проверка (компьютером или человеком) перед тем, как принять программу. В таких случаях проще и дешевле беспечно продолжать, не пытаясь поймать все ошибки на нижнем уровне. Это может служить примером упрощения, возможного благодаря схеме многоуровневого восстановления после ошибок (§ 14.9).

19.4. Распределители памяти

Распределитель памяти (allocator) используется, чтобы отделить разработчика алгоритмов и контейнеров, которые должны выделять память, от подробностей физической организации памяти. Распределитель памяти обеспечивает стандартные способы выделения и перераспределения памяти, а также стандартные имена типов для указателей и ссылок. Подобно итератору распределитель памяти — это чистая абст-

ракция. Любой тип, ведущий себя, как распределитель памяти, является распределителем памяти.

Стандартная библиотека обеспечивает стандартный распределитель памяти, предназначенный для того, чтобы хорошо служить большинству пользователей данной реализации. Кроме того, пользователь может обеспечить свои распределители памяти, предоставляющие альтернативный доступ к памяти. Например, мы можем написать распределители памяти, работающие с разделяемой памятью, памятью со сборкой мусора, памятью из заранее выделенного пула объектов (§ 19.4.2) и т. д.

Стандартные контейнеры и алгоритмы получают память и обращаются к ней через средства, обеспечиваемые распределителем памяти. Таким образом, предоставляя новый распределитель памяти, мы обеспечиваем стандартные контейнеры способом использования новых видов памяти.

19.4.1. Стандартный распределитель памяти

Стандартный шаблон ***allocator*** из заголовочного файла <***memory***> выделяет память при помощи оператора ***new*** () (§ 6.2.6) и по умолчанию используется всеми стандартными контейнерами:

```
template<class T> class allocator {
public:
        typedef T value_type;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;
        typedef T* pointer;

        typedef const T* const_pointer;

        typedef T& reference;
        typedef const T& const_reference;

        pointer address (reference r) const { return &r; }
        const_pointer address (const_reference r) const { return &r; }

        allocator () throw ();
        template<class U> allocator (const allocator<U>&) throw ();
        ~allocator () throw ();

        // память для n объектов T
        pointer allocate (size_type n, allocator<void>::const_pointer hint = 0);
        // перераспределяет n объектов T, не уничтожает их
        void dealocate (pointer p, size_type n);

        // инициализирует *p значением val
        void construct (pointer p, const T& val) { new (p) T (val); }
        // уничтожает *p, но не перераспределяет
        void destroy (pointer p) { p->~T (); }

        size_type max_size () const throw ();

        // фактически: typedef allocator<U> other
        template<class U>
        struct rebind { typedef allocator<U> other; };
};

template<class T> bool operator== (const allocator<T>&, const allocator<T>&) throw ();
template<class T> bool operator!= (const allocator<T>&, const allocator<T>&) throw ();
```

Операция *allocate* (*n*) выделяет память для *n* объектов, которая может быть снова высвобождена соответствующим вызовом *deallocate* (). Отметим, что *deallocate* () также принимает в качестве аргумента число *n*. Это позволяет создавать близкие к оптимальным распределители, хранящие минимум информации о выделяемой памяти. С другой стороны, такие распределители требуют, чтобы при вызове *deallocate* () пользователь всегда указывал правильное *n*. Отметим: функция *deallocate* () отличается от оператора *delete* () (§ 6.2.6) тем, что указатель, передаваемый ей в качестве аргумента, не может быть равен нулю.

Операция *allocator* по умолчанию использует *operator new* (*size_t*) для выделения памяти и *delete* (*void**) для ее освобождения. Это приводит к тому, что может быть вызвана *new_handler* (), и в случае, если памяти не хватает, генерируется исключение *std*::*bad_alloc*.

Отметим, что *allocate* () не обязана каждый раз вызывать распределитель памяти нижнего уровня. Часто для распределителя памяти лучшей стратегией является поддержка свободного списка областей памяти, готовых к использованию, с минимальными затратами времени (§ 19.4.2).

Необязательный аргумент *hint* (подсказка) для *allocate* () полностью зависит от реализации. Однако, он предназначен в помощь распределителям памяти для систем, где важна локальность. Например, распределитель может попытаться выделить область памяти для связанных между собой объектов в одной и той же странице системы со страничной организацией памяти. Тип аргумента *hint* — указатель из сверхупрощенной специализации:

```
template<> class allocator<void> {
public:
        typedef void* pointer;
        typedef const void* const_pointer;
        // обратите внимание: не ссылка
        typedef void value_type;
        template <class U>
        struct rebind { typedef allocator<U>other; };   // фактически: typedef allocator<U> other
};
```

Тип *allocator<void>*::*pointer* действует как тип универсального указателя и является *void** для всех стандартных распределителей памяти.

Если документация на распределитель памяти не указывает обратного, у пользователя есть два разумных выбора при вызове *allocate* ():

[1] Не давать распределителю никакой подсказки.

[2] В качестве подсказки использовать указатель на объект, который часто применяется вместе с новым объектом — например, предыдущий элемент в последовательности.

Распределители памяти предназначены для избавления разработчиков контейнеров от необходимости иметь дело напрямую с «сырой» памятью. В качестве примера подумайте, как могла бы использовать память реализация класса *vector*:

```
template<class T, class A = allocator<T> > class vector {
public:
        typedef typename A::pointer iterator;
        //...
private:
        A alloc;            // распределитель памяти объекта
        iterator v;         // указатель на элементы
        //...
```

```
public:
    explicit vector (size_type n, const T& val = T (), const A& a=A ()) : alloc (a)
    {
        v = alloc.allocate (n);
        for (iterator p = v; p<v+n; ++p) alloc.construct (p, val);
        // ...
    }
    void reserve (size_type n)
    {
        if (n<=capacity ()) return;
        iterator p = alloc.allocate (n);
        iterator q = v;
        while (q<v+size ()) {                    // копирование существующих элементов
            alloc.construct (p++, *q);
            alloc.destroy (q++);
        }
        alloc.deallocate (q, capacity ());       // освобождаем старое пространство
        v = p-size ();
        // ...
    }
    // ...
};
```

Операции типа ***allocator*** выражаются в терминах определений типов ***pointer*** и ***reference***, чтобы дать пользователю возможность обеспечить альтернативные типы для доступа к памяти. В общем виде сделать это очень трудно. Например, в рамках языка C++ невозможно определить совершенный ссылочный тип. Однако разработчики реализаций языка и библиотеки могут пользоваться этими определениями типов для поддержки типов, которые не в состоянии ввести рядовой пользователь. Примером может служить распределитель, обеспечивающий доступ к долговременной памяти. Другим примером служит тип «длинного» указателя для доступа к главной памяти за пределами того пространства, которое может адресовать указатель по умолчанию (обычно 32-разрядный).

Рядовой пользователь может ввести тип необычного указателя для особых нужд. Этого нельзя сделать для ссылок, но для эксперимента или специализированной системы такое ограничение вполне приемлемо.

Распределитель памяти предназначен для того, чтобы было легче работать с объектами типа, определяемого параметром их шаблона. Однако, для большинства реализаций контейнеров требуются объекты дополнительных типов. Например, разработчику контейнера ***list*** понадобится разместить в памяти объекты ***Link***. Обычно память под них должна выделяться распределителями памяти контейнера ***list***.

Любопытный тип ***rebind*** введен для того, чтобы размещать в памяти объекты произвольного типа. Рассмотрим определение:

```
typedef typename A::rebind<Link>::other Link_alloc;      // «шаблон» см. § B.13.6
```

Если ***A*** — это ***allocator***, то ***rebind<Link>::other*** в определении типа означает ***allocator<Link>***, так что предыдущее определение — это косвенный способ выразить следующее:

```
typedef allocator<Link> Link_alloc;
```

Данная косвенность освобождает нас от необходимости упоминать ***allocator*** прямо. Это выражает тип ***Link_alloc*** как параметр шаблона ***A***. Например:

```
template<class T, class A = allocator<T> > class list {
private:
    class Link { /* ... */ };

    typedef typename A::template rebind<Link>::other Link_alloc;      // allocator<Link>

    Link_alloc a;      // распределитель памяти для link
    A alloc;           // распределитель памяти для list
    // ...
public:
    typedef typename A::pointer iterator;
    // ...

iterator insert (iterator position, const T& x)
{
    Link_alloc::pointer p=a.allocate (1);      // получение Link
    // ...
}
    // ...
};
```

Поскольку ***Link*** — член ***list***, он параметризуется распределителем памяти. Поэтому объекты ***Link*** из контейнеров ***list*** с разными распределителями памяти принадлежат к разным типам, точно так же, как и сами контейнеры ***list*** (§ 17.3.3).

19.4.2. Распределители памяти, определяемые пользователем

Те, кто реализует контейнеры, часто применяют к объектам операции ***allocate*** () или ***deallocate*** () «по одному объекту за раз». При наивной реализации ***allocate*** () это подразумевает много вызовов оператора ***new***, а не все реализации оператора ***new*** эффективны при подобном использовании. Как пример определяемого пользователем распределителя памяти, я приведу схему использования пулов с участками памяти фиксированного размера, из которых распределитель может выделять память при помощи ***allocate*** () эффективнее, чем более распространенный и универсальный оператор ***new*** ().

Мне случилось столкнуться с распределителем из пула, который делал в общем-то правильную вещь, но имел неправильный интерфейс (так как был разработан за несколько лет до того, как придумали распределители памяти). Тот класс ***Pool*** реализовывал понятие о пуле с элементами фиксированного размера, в котором пользователь мог быстро выделять память и освобождать ее. Это был низкоуровневый тип, работавший напрямую с памятью и заботящийся о выравнивании:

```
class Pool {
    struct Link { Link* next; };

    struct Chunk {                          // Chunk — кусок, участок
        enum { size = 8*1024-16 };          // немного меньше 8 K, чтобы уложиться в 8 K
        char mem[size];                     // сначала резервируем область памяти
        Chunk* next;                        // (чтобы обеспечить строгое выравнивание)
    };
    Chunk* chunks;
```

```
        const unsigned int esize;
        Link* head;
        Pool (Pool&);                   // защита от копирования
        void operator= (Pool&);         // защита от копирования
        void grow ();                   // увеличение пула
public:
        Pool (unsigned int n);          // n — это размер элементов
        ~Pool ();

        void* alloc ();                 // выделение памяти под один элемент
        void free (void* b);            // помещение элемента обратно в пул
};

inline void* Pool::alloc ()
{
        if (head==0) grow ();
        Link* p = head;                 // возвратить первый элемент
        head = p->next;
        return p;
}

inline void Pool::free (void* b)
{
        Link* p = static_cast<Link*> (b);
        p->next = head;                 // поместить b обратно как первый элемент
        head = p;
}

Pool::Pool (unsigned int sz)
        : esize (sz<sizeof (Link*) ? sizeof (Link*) : sz)
{
        head=0;
        chunks=0;
}

Pool::~Pool ()                          // высвобождение всех кусков (chunks)
{
        Chunk* n = chunks;
        while (n) {
                Chunk* p = n;
                n = n->next;
                delete p;
        }
}

// выделяет память под новый кусок, организуя его
// как связный список элементов размера esize
void Pool::grow ()
{
        Chunk* n = new Chunk;
        n->next = chunks;
        chunks = n;

        const int nelem = Chunk::size/esize;
        char* start = n->mem;
        char* last = &start[ (nelem-1)*esize];
```

```
        for (char* p = start; p<last; p+=esize)
            reinterpret_cast<Link*>(p)->next = reinterpret_cast<Link*>(p+esize);
        reinterpret_cast<Link*>(last)->next = 0;
        head = reinterpret_cast<Link*>(start);
    };
```

Чтобы добавить чуть-чуть реализма, я буду использовать ***Pool*** неизменным как часть реализации моего распределителя памяти и не буду переписывать его, чтобы ввести правильный интерфейс. Распределитель памяти из пула предназначен для быстрого выделения и высвобождения памяти под один элемент, и это именно то, что мой класс ***Pool*** поддерживает. Расширение этой реализации для выделения памяти под произвольное число объектов и для объектов произвольного размера (как того требует ***rebind***) я оставлю в качестве упражнения (§ 19.6[9]).

Имея ***Pool***, определение распределителя ***Pool_alloc*** становится тривиальным:

```
template<class T> class Pool_alloc {
private:
    static Pool mem;        // пул элементов размером sizeof(T)
public:
    // аналогично стандартному распределителю памяти (§ 19.4.1)
};

template<class T> Pool Pool_alloc<T>::mem(sizeof(T));

template<class T> Pool_alloc<T>::Pool_alloc() { }

template<class T>
T* Pool_alloc<T>::allocate(size_type n, void* = 0)
{
    if (n == 1) return static_cast<T*>(mem_alloc());
    //...
}

template<class T>
void Pool_alloc<T>::deallocate(pointer p, size_type n)
{
    if (n == 1) {
        mem.free(p);
        return;
    }
    //...
}
```

Теперь этот распределитель памяти можно использовать очевидным образом:

памяти, как с одним и тем же. Это может привести к значительным преимуществам в быстродействии. Например, из-за этого ограничения не нужно оставлять память для распределителей в объектах ***Link*** (которые обычно параметризуются распределителем памяти контейнера, связями (links) в котором они являются), и операции, которые могут выполнять доступ к элементам двух последовательностей (такие как ***swap*** ()), не обязаны проверять, все ли участвующие объекты имеют один и тот же распределитель памяти. Однако это ограничение на самом деле приводит к тому, что такие распределители не могут использовать данные конкретного объекта.

Прежде чем применять такой способ оптимизации, убедитесь в необходимости этого. Я полагаю, что многие распределители по умолчанию будут реализовывать в точности этот тип классической оптимизации в C++, что избавит вас от лишних хлопот.

19.4.3. Обобщенные распределители памяти

Распределитель памяти — это упрощенный и оптимизированный вариант идеи о задании контейнеру информации в виде параметра шаблона (§ 13.4.1, § 16.2.3). Например, имеет смысл потребовать, чтобы каждый элемент в контейнере размещался распределителем памяти этого контейнера. Однако, если двум спискам ***list*** одного и того же типа позволить иметь разные распределители памяти, то операция ***splice*** () (удалить–вставить) не может быть реализована посредством «пересылки» одного списка на другой. Вместо этого ***splice*** () пришлось бы копировать элементы для учета тех редких случаев, когда мы хотим объединить элементы из списка ***list*** с одним распределителем памяти с другим списком того же самого типа, но с другим распределителем памяти. Подобным же образом, если распределителям памяти позволить стать совершенно универсальными, механизм ***rebind*** (), позволяющий распределителю памяти размещать элементы произвольных типов, пришлось бы сделать более изощренным. Поэтому считается, что стандартный распределитель памяти не содержит данных на уровне объекта, и при реализации стандартного контейнера этим можно воспользоваться.

Удивительно, что кажущееся драконовским ограничение на пообъектную информацию в распределителях памяти оказывается не слишком серьезным. Большинство распределителей памяти не нуждаются в данных на уровне объекта, и могут работать быстрее без этих данных. Распределители памяти, однако, могут хранить данные на уровне типа. Если нужны разные данные, можно воспользоваться разными типами распределителей. Например:

```
template<class T, class D> class My_alloc {          // распределитель памяти для T,
                                                     // реализованный с использованием D
    D d;  // данные, нужные для my_alloc<T, D>
    // ...
};

typedef My_alloc<int, Persistent_info> Persistent;
typedef My_alloc<int, Shared_info> Shared;
typedef My_alloc<int, Default_info> Default;

list<int, Persistent> lst1;
list<int, Shared> lst2;
list<int, Default> lst3;
```

Списки *lst1*, *lst2* и *lst3* относятся к разным типам. Поэтому, оперируя двумя из этих списков, мы должны использовать универсальный алгоритм (глава 18), а не специализированные операции со списками (§ 17.2.2.1). Это приводит к тому, что происходит копирование, а не пересылка, так что наличие разных распределителей памяти не вызывает проблем.

Ограничение против данных на уровне объекта введено из-за строгих требований к эффективности стандартной библиотеки в смысле быстродействия и памяти. Например, расходы памяти на данные распределителя для списка, вероятно, не так уж значительны, однако они могут стать серьезными, если затраты внесет каждая связь в списке.

Рассмотрим, как прием с распределителем памяти можно использовать, если отказаться от ограничений эффективности для стандартной библиотеки. Это может пригодиться для нестандартной библиотеки, от которой не требуется высокого быстродействия для каждой структуры данных и каждого типа в программе, а также для некоторых специальных реализаций стандартной библиотеки. В таких случаях распределитель памяти можно использовать для хранения той информации, которая часто содержится в универсальных базовых классах (§ 16.2.2). Например, распределитель памяти можно разработать так, чтобы он отвечал на запросы о том, где располагается объект, предоставлял данные о его структуре и отвечал на вопросы вроде «содержится ли этот элемент в этом контейнере?» Он мог бы также обеспечить контроль над контейнером, который действует в качестве кэша для информации в постоянной памяти, обеспечить ассоциативную связь между контейнером и другими объектами и т. д.

Так можно понятным образом ввести произвольные услуги для обычных операций с контейнерами. Однако лучше всего разделять операции, относящиеся к хранению данных, и к их использованию. Последние не относятся к обобщенным распределителям памяти, но их можно предоставить через отдельный аргумент шаблона.

19.4.4. Неинициализированная память

Кроме стандартного типа ***allocator***, заголовочный файл <***memory***> предоставляет несколько функций для работы с неинициализированной памятью. Они обладают опасным, но порой очень важным свойством — использовать имя типа ***T***, чтобы обращаться не собственно к сконструированному объекту типа ***T***, а к пространству памяти, достаточному, чтобы хранить объект типа ***T***.

Библиотека обеспечивает три способа для копирования значений в неинициализированную область:

```
template<class In, class For>
For uninitialized_copy (In first, In last, For res)        // копирование в res
{
        typedef typename iterator_traits<For>::value_type V;

        while (first != last)
        // конструируем в res (§ 10.4.11)
                new (static_cast<void*> (&*res++)) V (*first++);
        return res;
}
```

```
template<class For, class T>
void uninitialized_fill (For first, For last, const T& val)
{
        typedef typename iterator_traits<For>::value_type V;

        // конструируем в first
        while (first != last)
                new (static_cast<void*> (&*first++)) V (val);
}

template<class For, class Size, class T>
void uninitialized_fill_n (For first, Size n, const T& val)
{
        typedef typename iterator_traits<For>::value_type V;

        // конструируем в first
        while (n--)
                new (static_cast<void*> (&*first++)) V (val);
}
```

Эти функции предназначены прежде всего для тех, кто реализует контейнеры и алгоритмы. Например, используя эти функции (§ 19.6[10]), очень легко реализовать ***reserve*** () и ***resize*** () (§ 16.3.8). Будет не очень хорошо, если какой-нибудь неинициализированный объект сбежит из пределов контейнера и попадется обычному пользователю. (См. также § Д.4.4.)

Для приемлемого функционирования алгоритмы часто требуют промежуточного хранилища. Часто такую промежуточную память лучше всего выделять в одной операции, но не инициализировать, пока распределение действительно не понадобится. Поэтому для выделения и высвобождения неинициализированной области памяти библиотека обеспечивает пару функций:

```
// выделяет память, не инициализируя:
template<class T> pair<T*, ptrdiff_t> get_temporary_buffer (ptrdiff_t);
// высвобождает память, не уничтожая:
template<class T> void return_temporary_buffer (T*);
```

Операция ***get_temporary_buffer<X>*** (***n***) пытается выделить область памяти под ***n*** или более объектов типа ***X***. Если ей удается выделить какую-то память, она возвращает указатель на первую неинициализированную область памяти и число объектов типа ***X***, способных уместиться в этой области; в противном случае значение ***second*** в паре равно нулю. Идея заключается в том, чтобы система могла хранить число буферов фиксированного размера, готовых к быстрому выделению, так что запрос памяти для ***n*** объектов может привести к выделению объема памяти большего, чем необходимо для ***n*** объектов. Однако может быть выделено и меньше, поэтому способ использования ***get_temporary_buffer*** () состоит в том, что мы просим побольше, а потом смотрим, сколько окажется в распоряжении.

Буфер, полученный функцией ***get_temporary_buffer*** (), после вызова ***return_temporary_buffer*** () должен быть освобожден перед повторным использованием. Точно так же, как ***get_temporary_buffer*** () выделяет память под объект без его конструирования, ***return_temporary_buffer*** () высвобождает память без уничтожения объекта. Поскольку ***get_temporary_buffer*** () является операцией низкого уровня и, вероятно, оптимизированной для работы с временными буферами, ее не следует использовать вместо ***new*** или ***allocator::allocate*** () для выделения более долговременного хранилища.

Стандартные алгоритмы, которые записывают в последовательность, считают, что элементы последовательности заблаговременно проинициализированы. То есть для записи эти алгоритмы пользуются присваиванием, а не копирующим конструктором. Поэтому мы не можем использовать неинициализированную память как непосредственный выход алгоритма. Это может оказаться неприятным, поскольку присваивание часто оказывается значительно дороже, чем инициализация. Кроме того, нас не интересуют значения, поверх которых мы все равно собираемся произвести запись (иначе мы не стали бы писать поверх). Решение состоит в использовании ***raw_storage_iterator*** из <***memory***>, который производит инициализацию, а не присваивание:

```
template<class Out, class T>
class raw_storage_iterator : public iterator<output_iterator_tag, void, void, void, void> {
        Out p;
public:
        explicit raw_storage_iterator (Out pp) : p (pp) { }
        raw_storage_iterator& operator* () { return *this; }
        raw_storage_iterator& operator= (const T& val)
        {
                T* pp = &*p;
                new (pp) T (val);          // помещает val в pp (§ 10.4.11)
                return *this;
        }
        raw_storage_iterator& operator++ () { ++p; return *this; }
        raw_storage_iterator operator++ (int) {
                raw_storage_iterator t= *this;
                p++;
                return t; }
};
```

Например, мы могли бы написать шаблон, который копирует содержимое контейнера ***vector*** в буфер:

```
template<class T, class A> T* temporary_dup (vector<T, A>& v)
{
        pair<T*, ptrdiff_t> p=get_temporary_buffer<T> (v.size ());
        if (p.second< v.size()) {                    // проверка наличия доступной памяти
                if (p.first != 0) return_temporary_buffer(p.first);
                return 0;
        }
        copy (v.begin (), v.end (), raw_storage_iterator<T*, T> (p.first));
        return p.first;
}
```

Если бы вместо ***get_temporary_buffer*** () использовался оператор ***new***, была бы произведена инициализация. Когда инициализации избегают, для работы с неинициализированной областью необходим ***raw_storage_iterator***. В данном примере за вызов ***destroy_temporary_buffer*** () для указателя, который получила ***temporary_dup*** (), отвечает та функция, которая вызвала ***temporary_dup*** ().

19.4.5. Динамическое распределение памяти

Функции, использованные при реализации операторов ***new*** и ***delete***, объявлены в <***new***> вместе с несколькими связанными с ними средствами:

```
class bad_alloc : public exception { /* ... */ };

struct nothrow_t { };
extern const nothrow_t nothrow;          // индикатор выделения памяти,
                                         // не генерирующего исключений

typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

void* operator new(size_t) throw(bad_alloc);
void operator delete(void*) throw();

void* operator new(size_t, const nothrow_t&) throw();
void operator delete[](void*, const nothrow_t&) throw();

void* operator new[](size_t) throw(bad_alloc);
void operator delete[](void*) throw();

void* operator new[](size_t, const nothrow_t&) throw();
void operator delete[](void*, const nothrow_t&) throw();

void* operator new(size_t, void* p) throw() { return p; }      // размещение (§ 10.4.11)

void operator delete[](void* p, void*) throw() { }

void* operator new[](size_t, void* p) throw() { return p; }
void operator delete(void* p, void*) throw() { }
```

Операторы **new** () или **new** [] () с незаполненной (пустой) *спецификацией исключения* (exception-specification) (§ 14.6) не могут сигнализировать об истощении памяти, генерируя исключение **std::bad_alloc**. Вместо этого, при неудачной попытке выделить память они возвращают 0. В *выражении-new* (§ 6.2.6.2) проверяется значение, возвращаемое аллокатором с незаполненной спецификацией исключения: если возвращаемое значение — 0, никакой конструктор не вызывается и *выражение-new* возвращает 0. В частности, аллокатор **nothrow** возвращает 0, а не генерирует исключение в случае безуспешной попытки выделения памяти.

```
void f()
{
    int* p = new int[100000];                  // может сгенерировать bad_alloc
    if (int* q = new (nothrow) int[100000]) {  // не сгенерирует исключения
        // выделение памяти удалось
    }
    else {
        // выделение памяти не удалось
    }
}
```

Это позволяет нам для выделения памяти использовать стратегию обработки ошибок до генерации исключений.

19.4.6. Выделение памяти в стиле C

От C язык C++ унаследовал функциональный интерфейс для динамического распределения памяти. Его можно найти в <*stdlib*>:

```
// выделяет n байт
```

```
void* malloc (size_t n);
// выделяет n раз по s байтов, инициализированных нулями
void* calloc (size_t n, size_t s);
// высвобождает память, выделенную функциями malloc() или calloc()
void free (void* p);
// изменяет до s размер массива, на который указывает p; если этого не получается,
// выделяет s байт и копирует в них массив, на который указывает p
void* realloc (void* p, size_t s);
```

Этих функций следует избегать и предпочитать ***new***, ***delete*** и стандартные контейнеры. Эти функции работают с неинициализированной памятью. В частности, ***free*** () не вызывает деструкторов для памяти, которую высвобождает. Реализация ***new*** и ***delete*** может пользоваться этими функциями, но нет никакой гарантии, что она будет это делать. Например, размещать объект при помощи ***new***, а уничтожать при помощи ***free*** () означает напрашиваться на неприятность. Если вы чувствуете потребность воспользоваться ***realloc*** (), подумайте, как лучше вместо этого обратиться к стандартному контейнеру; обычно это проще, и столь же эффективно (§ 16.3.5).

Библиотека также обеспечивает набор функций, предназначенных для эффективного манипулирования байтами. Поскольку С первоначально обращался к байтам без типа через указатели ***char****, эти функции находятся в <***cstring***>. Внутри этих функций с указателями ***void**** следует обращаются так, как будто это указатели ***char****:

```
// копирование неперекрывающихся областей
void* memcpy (void* p, const void* q, size_t n);
// копирование потенциально перекрывающихся областей
void* memmove (void* p, const void* q, size_t n);
```

Подобно ***strcpy*** () (§ 20.4.1) эти функции копируют ***n*** байтов из ***q*** в ***p*** и возвращают ***p***. Диапазоны, копируемые функцией ***memmove*** (), могут перекрываться. Однако ***memcpy*** () предполагает, что диапазоны не перекрываются, и обычно оптимизирована с использованием этого допущения:

```
// подобно strchr() (§ 20.4.1): находит b в p[0]..p[n-1]
void* memchr (const void* p, int b, size_t n);
// подобно strcmp(): сравнивает последовательности байтов
int memcmp (const void* p, const void* q, size_t n);
// устанавливает n байтов в b, возвращает p
void* memset (void* p, int b, size_t n);
```

Многие реализации обеспечивают хорошо оптимизированные версии этих функций.

19.5. Советы

[1] При написании алгоритма решите, какой вид итераторов нужен для обеспечения приемлемой эффективности, и выразите алгоритм при помощи (только) операторов, поддерживаемых этими итераторами; § 19.2.1.

[2] Для обеспечения эффективных реализаций алгоритма, когда в качестве аргументов вы имеете итераторы, обеспечивающие нечто большее, чем необходимо, используйте перегрузку; § 19.2.3.

[3] Для выражения алгоритмов, соответствующих разным категориям итераторов, пользуйтесь ***iterator_traits***; § 19.2.2.

[4] Не забывайте употреблять ++ между обращениями к ***istream_iterator*** и ***ostream_iterator***; § 19.2.6.
[5] Чтобы избежать переполнения контейнеров, пользуйтесь вставками; § 19.2.4.
[6] Во время отладки используйте дополнительные проверки и удаляйте их потом только в случае необходимости; § 19.3.1.
[7] Предпочитайте использовать ++*p*, а не *p*++; § 19.3.
[8] Чтобы повысить быстродействие алгоритмов, расширяющих структуры данных, пользуйтесь неинициализированной памятью; § 19.4.4.
[9] Чтобы повысить быстродействие алгоритмов, требующих временной памяти для хранения структур данных, пользуйтесь временными буферами; § 19.4.4.
[10] Прежде чем писать свой распределитель памяти, хорошенько подумайте; § 19.4.
[11] Избегайте функций ***malloc*** (), ***free*** (), ***realloc*** () и т. п.; § 19.4.6.
[12] Вы можете симулировать ***typedef*** с шаблоном, воспользовавшись приемом с ***rebind***; § 19.4.1.

19.6. Упражнения

1. (*1.5) Реализуйте ***reverse*** () из § 18.6.7. Подсказка: См. § 19.2.3.
2. (*1.5) Напишите итератор для записи ***Sink***, который бы в действительности никуда не писал. Когда может пригодиться ***Sink***?
3. (*2) Реализуйте ***reverse_iterator*** (§ 19.2.5).
4. (*1.5) Реализуйте ***ostream_iterator*** (§ 19.2.6).
5. (*2) Реализуйте ***istream_iterator*** (§ 19.2.6).
6. (*2.5) Завершите ***Checked_iter*** (§ 19.3).
7. (*2.5) Переделайте ***Checked_iter***, чтобы проверять недействительные итераторы.
8. (*2) Придумайте и реализуйте вспомогательный класс, который мог бы работать как заместитель контейнера, обеспечивая для пользователей полный интерфейс контейнера. Его реализация должна состоять из указателя на контейнер и реализации контейнерных операций с проверкой диапазона.
9. (*2.5) Завершите или реализуйте заново ***Pool_alloc*** (§ 19.4.2) так, чтобы она обеспечивала все возможности распределителя памяти ***allocator*** (§ 19.4.1) из стандартной библиотеки. Сравните быстродействие ***allocator*** и ***Pool_alloc***, чтобы решить, имеет ли смысл пользоваться ***Pool_alloc*** в вашей системе.
10. (*2.5) Реализуйте ***vector*** при помощи распределителей памяти, а не операторов ***new*** и ***delete***.

20

Строки

Оригинальности предпочитайте стандарт.
— Странк и Уайт

Строки — символы — ***char_traits*** — ***basic_string*** — итераторы — доступ к элементам — конструкторы — обработка ошибок — присваивание — преобразования — сравнения — вставка — конкатенация — поиск и замена — размер и емкость — ввод/вывод строк — С-строки — классификация символов — функции из библиотеки С — советы — упражнения.

20.1. Введение

Строка — это последовательность символов. Стандартный библиотечный класс ***string*** обеспечивает операции для манипулирования строками, такие как индексация (§ 20.3.3), присваивание (§ 20.3.6), сравнение (§ 20.3.8), добавление (§ 20.3.9), конкатенация (§ 20.3.10) и поиск подстрок (§ 20.3.6). Стандарт не обеспечивает для подстрок никаких общих средств, поэтому то, что рассматривается здесь, можно считать стандартным примером использования строк (§ 20.3.11). Стандартная строка может состоять из практически любых символов (§ 20.2).

Опыт показывает, что совершенный класс ***string*** разработать невозможно. Для этого слишком различаются людские вкусы, ожидания и потребности. Поэтому стандартный библиотечный класс ***string*** не идеален. Я бы кое-что решил по-другому, и вы тоже. Однако библиотечный класс ***string*** хорошо служит многим потребностям, легко предоставляются вспомогательные функции, и ***std***::***string*** широко известен и доступен. В большинстве случаев эти факторы важнее, чем второстепенные улучшения, которые мы можем произвести. Написание строковых классов имеет большое образовательное значение (§ 11.12, § 13.2), но для широко используемого кода следует пользоваться стандартным библиотечным классом ***string***.

C++ унаследовал от С понятие строки как оканчивающегося нулем массива элементов типа ***char***, а также набор функций для манипулирования такими С-строками (§ 20.4.1).

20.2. Символы

«Символ» — само по себе интересное понятие. Рассмотрим символ ***C***. ***C***, который вы видите как кривую линию на странице (или экране), я напечатал на моем компьютере много месяцев назад. Там он и живет, как числовое значение 67 в 8-битном

байте. Это третья буква в латинском алфавите, обычная аббревиатура для шестого химического элемента (углерод) и, между прочим, имя языка программирования (§ 1.6). То, что имеет значение в контексте программирования с использованием строк — это наличие связи между закорючками с общепонятными значениями, так называемыми буквами, и числовыми кодами. Чтобы все окончательно запутать, одна и та же буква в разных символьных наборах может иметь разные числовые значения, и не каждый набор имеет значение для той или иной буквы, а в широком использовании находится много различных символьных наборов. Символьный набор — это взаимное соответствие между буквой (некоторого общепонятного символа) и числом.

Программисты на C++ обычно считают доступным стандартный американский символьный набор (ASCII), но C++ допускает использование и некоторых символов, которых может не оказаться в программном окружении. Например, при отсутствии таких символов как [и {, могут использоваться ключевые слова и диграфы (§ B.3.1).

Символьный набор с символами не из ASCII представляет еще одну проблему. Такие языки как китайский, датский, французский, исландский, японский не могут быть адекватно переданы с использованием только символов ASCII. Хуже того, символьные наборы для этих языков могут быть взаимно не согласованы. Например, буквы европейских языков, использующих латинский алфавит, *почти* укладываются в набор из 256 символов. К несчастью, в разных языках по-прежнему используются разные символьные наборы, и некоторые разные символы имеют один и тот же код. Например, французский язык (использующий Latin 1) не слишком хорошо сосуществует с исландским (который поэтому требует Latin 2). Амбициозные попытки представить все известные человеку символы в едином наборе принесли большую пользу, но даже 16-разрядный набор — такой как Unicode — не может удовлетворить всех. 32-разрядные наборы, которые способны — насколько мне известно — содержать в себе все символы, не нашли широкого применения.

В основном подход C++ состоит в том, чтобы позволить программисту пользоваться в строках любым символьным набором. Можно использовать расширенный символьный набор или переносимую числовую кодировку (§ B.3.3)

20.2.1. Особенности символов

Как показано в § 13.1, строка может в качестве типа своих элементов (символов) использовать в принципе любой тип с соответствующей операцией копирования. Однако эффективность может улучшиться, а реализации могут упроститься для типов, не имеющих определяемых пользователем операций копирования. Поэтому стандартная строка ***string*** требует, чтобы используемый в качестве символьного тип, не имел пользовательских операций копирования. Это также помогает сделать простым и эффективным ввод/вывод строк.

Свойства символьного типа определяются его ***char_traits*** (свойствами символов). ***char_traits*** является специализацией шаблона:

```
template<class Ch> struct char_traits { };
```

Все ***char_traits*** определены в ***std***, а стандартные представлены в <***string***>. Собственно универсальные ***char_traits*** не имеют свойств, их имеют только специализации ***char_traits*** для конкретных символьных типов Рассмотрим ***char_traits<char>***:

```
template<> struct char_traits<char> {
        typedef char char_type;                          // тип символа
        static void assign (char_type&, const char_type&); // = для char_type

        // численное представление символов:

        typedef int int_type;                            // тип целочисленных
                                                         // значений символов

        static char_type to_char_type (const int_type&);  // преобразование int в char
        static int_type to_int_type (const char_type&);   // преобразование char в int
        static bool eq_int_type (const int_type&, const int_type&); // ==

        // сравнения char_type:

        static bool eq (const char_type&, const char_type&); // ==
        static bool lt (const char_type&, const char_type&); // <

        // операции над массивами s[n]:

        static char_type* move (char_type* s, const char_type* s2, size_t n);
        static char_type* copy (char_type* s, const char_type* s2, size_t n);
        static char_type* assign (char_type* s, size_t n, char_type a);

        static int compare (const char_type* s, const char_type* s2, size_t n);
        static size_t length (const char_type*);
        static const char_type* find (const char_type* s, int n, const char_type&);

        // функции, связанные с вводом/выводом:

        typedef streamoff off_type;                      // смещение в потоке
        typedef streampos pos_type;                      // позиция в потоке
        typedef mbstate_t state_type;                    // многобайтное состояние потока


        static int_type eof ();                          // конец файла
        static int_type not_eof (const int_type& i);     // i если i не равно eof()
        static state_type get_state (pos_type p);        // многобайтное преобразование
                                                         // состояния символа в p
};
```

Реализация стандартного шаблона строк — ***basic_string*** (§ 20.3) — опирается на эти типы и функции. Тип, используемый для ***basic_string*** в качестве символьного, должен обеспечивать специализацию ***char_traits***, которая бы поддерживала их все.

Чтобы тип был ***char_type***, мы должны уметь получать целочисленное значение, соответствующее каждому символу. Тип этого целого — ***int_type***, и преобразование его в ***char_type*** и обратно выполняется функциями ***to_char_type*** () и ***to_int_type*** (). Для ***char*** это преобразование тривиально.

И ***move*** (*s*, *s2*, *n*), и ***copy*** (*s*, *s2*, *n*) копируют *n* символов из *s2* в *s* при помощи ***assign*** (*s*[*i*], *s2*[*i*]). Разница в том, что ***move*** () работает правильно, даже если диапазоны [*s*, *s*+*n*[и [*s2*, *s2*+*n*[перекрываются. Поэтому ***copy*** () может оказаться быстрее. Это аналог функций из стандартной библиотеки C — ***memcpy*** () и ***memmove*** () (§ 19.4.6). Вызов ***assign*** (*s*, *n*, *x*) при помощи ***assign*** (*s*[*i*], *x*) присваивает *n* копий *x* строке *s*.

Функция ***compare*** () использует для сравнения символов ***lt*** () и ***eq*** (). Она возвращает ***int***, причем ***0*** означает точное совпадение, отрицательное число означает, что

первый аргумент стоит лексикографически до второго, а положительное — что после. Такое использование возвращаемого значения аналогично функции ***strcmp*** () из стандартной библиотеки C (§ 20.4.1).

Функции, связанные с вводом/выводом, используются в реализации низкоуровневого ввода/вывода (§ 21.6.4).

Расширенный символ (символ из расширенного набора) — то есть объект типа ***wchar_t*** (§ 4.3) — похож на ***char***, но занимает два байта или больше. Свойства ***wchar_t*** описывает ***char_traits<wchar_t>***:

```
template<> struct char_traits<wchar_t> {
        typedef wchar_t char_type;
        typedef wint_t int_type;
        typedef streamoff off_type;
        typedef wstreampos pos_type;

        // подобно char_traits<char>
};
```

Тип ***wchar_t***, как правило, используется для хранения символов из 16-разрядного набора — например, Unicode.

20.3. Тип basic_string

Стандартные библиотечные возможности работы со строками базируются на шаблоне ***basic_string***, который предоставляет типы членов и операции, схожие с теми, что обеспечиваются стандартными контейнерами (§ 16.3.1):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
        class std::basic_string {
public:
        //...
};
```

Этот шаблон и все связанные с ним средства определены в пространстве имен ***std*** и представлены заголовочным файлом <***string***>.

Два определения типов обеспечивают для обычных строковых типов общепринятые имена:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Тип ***basic_string*** похож на ***vector*** (§ 16.3), за исключением того, что ***basic_string*** вместо набора операций со списками, который предлагает ***vector***, обеспечивает некоторые типовые операции со строками, такие как поиск подстрок. Многие обычные применения строк лучше обслуживаются теми реализациями, которые минимизируют копирование, не используют свободной памяти для коротких строк, позволяют просто модифицировать более длинные строки и т. п. (см. § 20.6[12]). Большое число функций для работы со строками отражает важность манипуляций со строками, а также тот факт, что некоторые машины предоставляют специальные аппаратные инструкции, связанные со строками. Такие инструкции разработчик библитеки может использовать особенно эффективно, если их смысл близок функциям из стандартной библиотеки.

Как и другие стандартные библиотечные типы, ***basic_string<T>*** является конкретным типом (§ 2.5.3, § 10.3) без виртуальных функций. Его можно использовать

в качестве члена при проектировании классов для более изощренного манипулирования текстами, но он не предназначен для того, чтобы служить базовым для производных классов (§ 25.2.1; см. также § 20.6[10]).

20.3.1. Типы

Подобно ***vector***, ***basic_string*** делает свои родственные типы доступными через набор имен типов членов:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // типы (очень похоже на vector, list и т. п.: § 16.3):

    typedef Tr traits_type;              // специфично для basic_string

    typedef typename Tr::char_type value_type;
    typedef A allocator_type;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;

    typedef implementation_defined iterator;
    typedef implementation_defined const_iterator;

    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // ...
};
```

Понятие ***basic_string*** в добавление к простому ***basic_string<char>***, известному как ***string***, поддерживает строки с разнообразными символами. Например:

```
typedef basic_string<unsigned char> Ustring;

struct Jchar { /* ... */ };                    // тип японских букв
typedef basic_string<Jchar> Jstring;
```

Строками из таких символов можно пользоваться точно так же, как строками из ***char***, пока это позволяет семантика символов. Например:

```
Ustring first_word (Ustring& us)
{
    Ustring::size_type pos = us.find (' ');          // см. § 20.3.11
    return Ustring (us, 0, pos);                     // см.§ 20.3.4
}

Jstring first_word (Jstring& js)
{
    Jstring::size_type pos = js.find (' ');          // см. § 20.3.11
    return Jstring (us, 0, pos);                     // см.§ 20.3.4
}
```

Естественно, также можно пользоваться шаблонами, принимающими строки в качестве аргумента:

```
template<class S> S first_word (S& s)
{
        typename S::size_type pos = s.find (' ');          // см. § 20.3.11
        return S (s, 0, pos);                              // см.§ 20.3.4
}
```

basic_string<Ch> может содержать любой символ из набора *Ch*. В частности, *string* может содержать *0* (ноль). «Символьный» тип *Ch* обязан вести себя подобно настоящим символам. В частности, он не может содержать определяемые пользователем копирующий конструктор, деструктор или оператор копирующего присваивания.

20.3.2. Итераторы

Как и другие контейнеры, *string* предоставляет итераторы для обычных и обратных итераций:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        //...
        // итераторы (так же как vector, list и т. п.; § 16.3.2):

        iterator begin ();
        const_iterator begin () const;
        iterator end ();
        const_iterator end () const;

        reverse_iterator rbegin ();
        const_reverse_iterator rbegin () const;
        reverse_iterator rend ();
        const_reverse_iterator rend () const;

        //...
};
```

Поскольку *string* имеет типы членов и функции, требующиеся для получения итераторов, строки можно использовать со стандартными алгоритмами (глава 18). Например:

```
void f (string& s)
{
        string::iterator p=find (s.begin (), s.end (), 'a');
}
```

Наиболее часто используемые операции над строками обеспечиваются непосредственно самим *string*. Надеюсь эти версии будут оптимизированы для строк помимо общей оптимизации универсальных алгоритмов.

Стандартные алгоритмы (глава 18) не так полезны для строк, как можно было бы подумать. Универсальные алгоритмы имеют тенденцию считать, что элементы контейнера значимы и сами по себе, в изоляции. Как правило, для строк это не так. Значение строки закодировано в точном соблюдении последовательности ее символов. Поэтому

сортировка строки (то есть сортировка символов в строке) лишает ее смысла, в то время как сортировка универсального контейнера обычно делает его более полезным.

Итераторы *string* не осуществляют проверку диапазона.

20.3.3. Доступ к элементам

К отдельным символам в строке можно обратиться по индексу:

```
template<class Ch, class Tr = char_traits<Ch>, class A=allocator<Ch> >
class basic_string {
public:
        //...
        // доступ к элементам (как для vector: § 16.3.3)
        const_reference operator[] (size_type n) const;          // обращение без проверки
        reference operator[] (size_type n);

        const_reference at (size_type n) const;                  // обращение с проверкой
        reference at (size_type n);

        //...
};
```

При обращении за диапазон функция *at* () сгенерирует исключение *out_of_range*.

По сравнению с вектором, строкам не хватает операций *front* () и *back* (). Чтобы обратиться к первому и последнему символам строки, мы должны написать соответственно *s*[*0*] и *s*[*s.length* ()−*1*]. Эквивалентность указателей и массивов (§ 5.3) для строк *string* не выполняется. Если *s* — это строка *string*, то &*s*[*0*] — не то же самое, что *s*.

20.3.4. Конструкторы

Набор операций инициализации и копирования для строк во многом отличается от того, что реализовано для других контейнеров (§ 16.3.4):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        //...
        // конструкторы и пр. (немного похоже на vector и list: § 16.3.4)

        explicit basic_string (const A& a = A ());
        basic_string (const basic_string& s, size_type pos = 0, size_type n = npos, const A& a = A ());
        basic_string (const Ch* p, size_type n = npos, const A& a = A ());
        basic_string (const Ch* p, const A& a = A ());
        basic_string (size_type n, Ch c, const A& a = A ());
        template<class In>basic_string (In begin, In end, const A& a = A ());

        ~basic_string ();

        static const size_type npos;          // маркер «все символы»

        //...
};
```

Строку *string* можно инициализировать С-строкой, другой строкой *string*, подстрокой строки *string* или последовательностью символов. Однако ее нельзя инициализировать символом или числом:

```
void f(char* p, vector<char>& v)
{
    string s0;                          // пустая строка
    string s00 = "";                    // тоже пустая строка

    string s1 = 'a';                    // ошибка: char не преобразуется в string
    string s2 = 7;                      // ошибка: int не преобразуется в string
    string s3 (7);                      // ошибка: нет конструктора
                                        // с одним аргументом int
    string s4 (7,'a');                  // 7 копий 'a'; то есть "aaaaaaa"

    string s5 = "Фродо";                // копия "Фродо"
    string s6 = s5;                     // копия s5

    string s7 (s5, 3, 2);               // s5[3] и s5[4]: то есть "до"
    string s8 (p+7, 3)                  // p[7], p[8] и p[9]
    string s9 (p, 7, 3);                // string(string(p), 7, 3),
                                        // возможно расточительно

    string s10 (v.begin (), v.end ());  // копирование всех символов из v
}
```

Символы нумеруются начиная с ***0***, так что строка — это последовательность символов с номерами от ***0*** до ***length ()−1***.

Длина строки ***length ()*** — это просто синоним ***size ()***; обе функции возвращают число символов в строке. Отметим, что вычисление длины строки не основывается на понятии «завершающего нуля», обрывающего С-строку (§ 20.4.1). Реализация ***basic_string*** хранит длину строки, не полагаясь на завершающий символ (ноль).

Подстроки выражаются как позиция символа плюс число символов. По умолчанию значение ***npos*** инициализируется максимальным возможным числом, означая как бы «все элементы».

Для создания строки из ***n*** неопределенных символов конструктора не существует. Самое большое, что мы можем сделать, чтобы к этому приблизиться, — это предоставить конструктор, который производит ***n*** копий указанного символа. Отсутствие конструктора, принимающего только один символ, и конструктора, принимающего только некоторое количество элементов, позволяет компилятору обнаружить ошибки вроде определений ***s2*** и ***s3*** в приведенном выше примере.

Копирующий конструктор — это конструктор с четырьмя аргументами. Три из них имеют значения по умолчанию. Ради эффективности этот конструктор можно реализовать как два отдельных конструктора. Пользователь не сможет узнать об этом, не посмотрев сгенерированный код.

Конструктор, являющийся членом шаблона, — самый универсальный. Он позволяет инициализировать строку значениями из произвольной последовательности. В частности, он позволяет инициализировать строку элементами разных символьных типов, если для них существует преобразование. Например:

```
void f(string s)
{
    wstring ws (s.begin (), s.end ());  // копирование всех символов из s
    // ...
}
```

Каждый ***wchar_t*** в ***ws*** инициализируется соответствующим ***char*** из ***s***.

20.3.5. Ошибки

Часто строки просто считывают, записывают, распечатывают, хранят, копируют и т. д. Это не вызывает проблем или, в худшем случае, возникают проблемы, связанные с быстродействием. Однако когда мы начинаем манипулировать отдельными подстроками и символами, чтобы составить новую строку из существующих, то рано или поздно делаем ошибки и можем произвести запись за конец строки.

Чтобы сделать доступ к отдельному символу явным, функция ***at*** () проверяет диапазон и, если мы пытаемся обратиться за конец строки, генерирует исключение ***out_of_memory***; [] не делает этого.

Большинство операций со строками принимают позицию символа и число символов. Если позиция больше размера строки, генерируется исключение ***out_of_range***. «Слишком большое» число символов воспринимается просто как «все остальные символы». Например:

```
void f()
{
        string s = "Snobol4";
        string s2 (s, 100, 2);          // позиция символа за пределами строки:
                                        // сгенерируется out_of_range
        string s3 (s, 2, 100);          // число символов «слишком велико»:
                                        // равносильно s3(s, 2, s.size()-2)
        string s4 (s, 2, string::npos); // символы, начиная с s[2]
}
```

Таким образом, «слишком больших» позиций следует избегать, но «слишком большие» числа символов могут пригодиться. Фактически, ***npos*** является максимально возможным значением для ***size_type***.

Мы можем попытаться указать отрицательную позицию или отрицательное число символов:

```
void g (string& s)
{
        string s5 (s, -2, 3);           // большая позиция! Сгенерируется out_of_range
        string s6 (s, 3, -2);           // большое число символов! Правильно
}
```

Однако ***size_type***, используемый для представления позиции и числа символов, имеет тип ***unsigned***, поэтому отрицательное число является просто запутанным способом обозначить большое положительное (§ 16.3.4).

Отметим, что функции для поиска подстрок (§ 20.3.11), если ничего не нашли, возвращают ***npos***. То есть они не возбуждают исключений. Однако при последующем использовании ***npos*** в качестве позиции символа исключение возбудится.

Другой способ обозначить подстроку — пара итераторов. Первый итератор определяет позицию, а разность между двумя итераторами — число символов. Как обычно, итераторы не производят проверки диапазона.

При использовании С-строк проверка диапазона труднее. Когда в качестве аргумента используется С-строка (указатель на ***char***), функции из ***basic_string*** считают, что указатель не равен ***0***. Получая позицию символа для С-строк, они считают эту строку достаточно длинной, чтобы позиция оказалась в ее пределах. Соблюдайте

осторожность! Здесь нужна параноидальная осторожность за исключением случаев, когда используются символьные литералы.

Для всех строк выполняется неравенство ***length*** () < ***npos***. В некоторых случаях, таких как вставка одной строки в другую (§ 20.3.9), может (хотя это и маловероятно) получиться строка слишком длинная, чтобы ее представить. В этом случае сгенерируется ***length_error***. Например:

```
string s (string::npos, 'a');          // возбудится length_error()
```

20.3.6. Присваивание

Естественно, для строк определены операции присваивания:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        //...
        // присваивание (немного похоже на vector и list: § 16.3.4):

        basic_string& operator= (const basic_string& str);
        basic_string& operator= (const Ch* p);
        basic_string& operator= (Ch c);

        basic_string& assign (const basic_string&);
        basic_string& assign (const basic_string& s, size_type pos, size_type n);
        basic_string& assign (const Ch* p, size_type n);
        basic_string& assign (const Ch* p);
        basic_string& assign (size_type n, Ch c);
        template<class In> basic_string& assign (In first, In last);

        //...
};
```

Как и другие стандартные контейнеры, строки имеют семантику значений. То есть когда одной строке присваивается другая, присваиваемая строка копируется, и после присваивания существует две отдельных строки с одинаковыми значениями. Например:

```
void g()
{
        string s1 = "Knold";
        string s2 = "Tot";

        s1 = s2;            // две копии "Tot"
        s2[1] = 'u';        // s2 стала "Tut", s1 по-прежнему "Tot"
}
```

Присваивание строке одного символа поддерживается, несмотря на то, что инициализации с одним символом не существует:

```
void f()
{
        string s = 'a';     // ошибка: инициализация типом char
        s = 'a';            // правильно: это присваивание
        s = "a";
        s = s;
}
```

Возможность присвоить строке символ *char* не очень полезна и даже может рассматриваться как провоцирующая на ошибки, однако приписывание *char* в конец при помощи оператора += порой важно (§ 20.3.9), и было бы странно, если бы могли написать *s+='c'*, а *s=s+'c'* было бы незаконно.

Имя *assign* () используется для присваиваний, которые являются альтернативами конструкторов со многими аргументами (§ 16.3.4, § 20.3.4).

Как упоминалось в § 11.12, *string* можно оптимизировать, чтобы на самом деле копирования не производилось, если только не нужны две копии какой-либо строки. Проект стандартного класса *string* поощряет создание реализаций, минимизирующих действительное копирование. Это делает использование строк, которые только считываются и задаются в качестве аргумента, гораздо более дешевым, чем кто-то мог по наивности предположить. Однако было бы так же наивно со стороны программистов не проверять имеющиеся у них реализации перед написанием кода, который полагается на оптимизацию копирования строк (§ 20.6[13]).

20.3.7. Преобразование в С-строку

Как показано в § 20.3.4, строку *string* можно инициализировать С-строкой, а С-строки можно присваивать строкам *string*. И наоборот, копию символов из *string* можно поместить в массив:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        //...
        // преобразование в С-строку

        const Ch* c_str() const;
        const Ch* data() const;
        size_type copy(Ch* p, size_type n, size_type pos = 0) const;

        //...
};
```

Функция *data* () пишет символы строки в массив и возвращает указатель на него. Массив принадлежит *string*, и пользователь не должен пытаться его уничтожить. Также пользователь не может полагаться на его значение после вызова неконстантной функции над этой строкой. Функция *c_str* () похожа на *data* () но добавляет в конец *0*, традиционно обрывающий С-строку. Например:

```
void f()
{
        string s = "равноденствие";      // s.length() = 13
        const char* p1 = s.data();       // p1 указывает на 13 символов
        printf("p1 = %s\n", p1);         // плохо: не хватает конца строки
        p1[2] = 'a';                     // ошибка: p1 указывает на константный массив
        s[2] = 'a';
        char c = p1[1];                  // плохо: вызов s.data() после изменения s

        const char* p2 = s.c_str();      // p2 указывает на 14 символов
        printf("p2 = %s\n", p2);         // правильно: c_str() добавила
                                         // завершающий символ
}
```

Иными словами, ***data*** () создает массив символов, в то время как ***c_str*** () создает С-строку. Эти функции предназначены прежде всего для того, чтобы упростить использование функций, которые требуют С-строк, и следовательно, ***c_str*** () будет применяться чаще, чем ***data*** (). Например:

```
void f(string s)
{
    int i=atoi(s.c_str());          // по цифрам из строки выдает
                                    // целочисленное значение (§ 20.4.1)
}
```

Как правило, пока символы вам не нужны, лучше всего оставлять их в ***string***. Однако если вы не можете пользоваться символами непосредственно, вы можете скопировать их в массив, а не оставлять в буфере, выдсленном функциями ***c_str*** () и ***data*** (). Для этого введена функция ***copy*** (). Например:

```
char* c_string(const string& s)
{
    char* p = new char[s.length()+1];  // заметьте: +1
    s.copy(p, string::npos);
    p[s.length()] = 0;                 // заметьте: добавляем завершающий символ
    return p;
}
```

Вызовом ***s.copy*** (***p***, ***n***, ***m***) копируется не более чем ***n*** символов в ***p***, начиная с ***s***[***m***]. Если в строке ***s*** символов для копирования меньше, чем ***n***, ***copy*** () просто копирует все символы, какие есть.

Отметим, что ***string*** может содержать и символ с кодом ***0***. Функции, манипулирующие с С-строками, воспримут этот ***0*** как завершающий символ. Будьте осторожны. Не помещайте в строку нулей, если вы применяете функции в стиле С (если только вы не используете ***0*** именно как завершающий символ).

Преобразование в С-строку может быть обеспечено оператором ***operator const char****** (), а не ***c_str*** (). Это обеспечило бы удобство неявного преобразования, но ценой всякого рода сюрпризов в тех случаях, когда такого преобразования не ожидалось.

Если вы обнаружите, что ***c_str*** () появляется в вашей программе чрезвычайно часто, вероятно, вы слишком полагаетесь на интерфейсы в стиле С. Часто бывает доступен аналогичный интерфейс, опирающийся на ***string***, а не на С-строки, и, чтобы избежать преобразований, можно воспользоваться им. Вы можете устранить большинство явных обращений к ***c_str*** () и другим образом — обеспечив дополнительное определение тех функций, которые заставляют вас писать вызовы ***c_str*** ():

```
extern "C" int atoi(const char*);

int atoi(const string& s)
{
    return atoi(s.c_str());
}
```

20.3.8. Сравнения

Строки можно сравнивать со строками их же типа и с массивами символов того же символьного типа:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        //...

        int compare (const basic_string& str) const;          // сочетает > и ==
        int compare (const Ch* p) const;

        int compare (size_type pos, size_type n, const basic_string& str) const;
        int compare (size_type pos, size_type n,
                     const basic_string& str, size_type pos2, size_type n2) const;
        int compare (size_type pos, size_type n, const Ch* p, size_type n2 = npos) const;

        //...
};
```

Если для строки ***compare*** () заданы позиция и размер, используется только указанная подстрока. Например, ***s.compare*** (***s***, ***pos***, ***n***) эквивалентно ***string*** (***pos***, ***n***, ***s2***).***compare*** (***s2***). В качестве критерия сравнения используется ***compare*** () из ***char_traits<Ch>*** (§ 20.2.1). Таким образом, ***s.compare*** (***s2***) возвращает ***0***, если строки имеют одинаковое значение; отрицательное число, если ***s*** лексикографически находится перед ***s2***, и положительное число в противном случае.

Пользователь не может задать свой критерий сравнения, как это делалось в § 13.4. Когда нужен такой уровень гибкости, мы можем воспользоваться ***lexicographical_compare*** () (§ 18.9), определить функцию, как в § 13.4, или написать явный цикл. Например, функция ***toupper*** () (§ 20.4.2) позволяет нам написать сравнение без учета регистра:

```
int cmp_nocase (const string& s, const string& s2)
{
        string::const_iterator p = s.begin ();
        string::const_iterator p2 = s2.begin ();

        while (p!=s.end () && p2!=s2.end ()) {
                if (toupper (*p) != toupper (*p2))
                        return (toupper (*p) < toupper (*p2)) ? -1 : 1;        // size не имеет знака
                ++p;
                ++p2;
        }

        return (s2.size ()==s.size ()) ? 0 : (s.size () <s2.size ()) ? -1 : 1;
}

void f(const string& s, const string& s2)
{
        if (s == s2) {                          // сравнение s и s2 с учетом регистра
                //...
        }

        if (cmp_nocase (s, s2) == 0) {          // сравнение s и s2 без учета регистра
                //...
        }

        //...
}
```

Для ***basic_string*** введены обычные операторы сравнения ==, !=, >, <, >= и <=:

```
template<class Ch, class Tr, class A>
bool operator== (const basic_string<Ch, Tr, A>&, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
bool operator== (const Ch*, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
bool operator== (const basic_string<Ch, Tr, A>&, const Ch*);

// аналогичные объявления для != , > , < , >= и <=
```

Операторы сравнения — это функции-не-члены, поэтому преобразования одинаково применимы к обоим операндам (§ 11.2.3). Для оптимизации сравнений со строковыми литералами предоставлены версии, принимающие С-строки. Например:

```
void f(const string& name)
{
    if (name == "Obelix" || "Asterix" == name) {      // используется оптимизированное ==
        // ...
    }
}
```

20.3.9. Вставка

Когда строка создана, ею можно манипулировать по-разному. Среди операций, изменяющих значение строки, одной из самых распространенных является «приписывание» к ней— то есть добавление символов в конец. Вставка в другие места строки встречается реже:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // добавление символов после (*this)[length()-1]:

    basic_string& operator+= (const basic_string& str);
    basic_string& operator+= (const Ch* p);
    basic_string& operator+= (const Ch c);
    void push_back (Ch c);

    basic_string& append (const basic_string& str);
    basic_string& append (const basic_string& s, size_type pos, size_type n);
    basic_string& append (const Ch* p, size_type n);
    basic_string& append (const Ch* p);
    basic_string& append (size_type n, Ch c);
    template<class In>basic_string& append (In first, In last);

    // вставка символов перед (*this)[pos]:

    basic_string& insert (size_type pos, const basic_string& str);
    basic_string& insert (size_type pos, const basic_string& s, size_type pos2, size_type n);
    basic_string& insert (size_type pos, const Ch* p, size_type n);
    basic_string& insert (size_type pos, const Ch* p);
    basic_string& insert (size_type pos, size_type n, Ch c);
```

```
        // вставка символов перед p:

        iterator insert (iterator p, Ch c);
        void insert (iterator p, size_type n, Ch c);
        template<class In>void insert (iterator p, In first, In last);

        void push_back (Ch c);
        // ...
};
```

В основном, многочисленные операции, введенные для инициализации строк и присваивания, доступны также и для вставки символов перед некоторой позицией и добавления в конец строки.

Для наиболее распространенной формы добавления предоставлен оператор +=, как удобное и понятное обозначение. Например:

```
string complete_name (const string& first_name, const string& family_name)
{
        // complelte_name — имя и фамилия
        // first_name — имя
        // family_name — фамилия
        string s = first_name;
        s+=' ';
        s+=family_name;
        return s;
}
```

Добавление в конец строки может быть значительно эффективнее, чем вставка в другую позицию. Например:

```
string complete_name2 (const string& first_name, const string& family_name)
        // не слишком хороший алгоритм
{
        string s = family_name;
        s.insert (s.begin (),' ');
        return s.insert (0,first_name);
}
```

Обычно вставка вынуждает реализацию *string* выделять лишнюю память и переписывать символы с места на место.

Поскольку *string* имеет операцию *push_back* (§ 16.3.5), для строк можно использовать *back_inserter* точно так же, как для общих контейнеров.

20.3.10. Конкатенация

Добавление символов — это особая форма конкатенации. *Конкатенация* — конструирование строки из двух расположением одной после другой — обеспечивается оператором +:

```
template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A>
operator+ (const basic_string<Ch, Tr, A>&, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A> operator+ (const Ch*, const basic_string<Ch, Tr, A>&);
```

```
template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A> operator+ (Ch, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A> operator+ (const basic_string<Ch, Tr, A>&, const Ch*);

template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A> operator+ (const basic_string<Ch, Tr, A>&, Ch);
```

Как обычно, оператор + определен как функция-не-член. Для шаблонов с несколькими параметрами это приводит к неудобству записи, поскольку параметры шаблона повторяются.

С другой стороны, использование конкатенации понятно и удобно. Например:

```
string complete_name3 (const string& first_name, const string& family_name)
{
    return first_name + ' ' + family_name;
}
```

Это удобство записи может достигаться ценой некоторого перерасхода времени по сравнению с **complete_name** (). Для **complete_name3** () требуется одна лишняя временная переменная (§ 11.3.2). На своем опыте я убедился, что это редко бывает важно, но об этом стоит помнить, когда пишешь вложенный цикл в программе, в которой требуется быстродействие. В этом случае мы могли бы попробовать устранить вызов функции **complete_name** (), сделав ее встроенной (**inline**), или составлять результирующую строку путем применения операций низкого уровня (§ 20.6[14]).

20.3.11. Поиск

Множество функций для поиска подстрок может обескуражить:

```
template<class Ch, class Tr = char_traits<Ch>,class A = allocator<Ch> >
class basic_string {
public:
    //...
    // поиск подпоследовательности (наподобие search() § 18.5.5):
    size_type find (const basic_string& s, size_type i = 0) const;
    size_type find (const Ch* p, size_type i, size_type n ) const;
    size_type find (const Ch* p, size_type i = 0) const;
    size_type find (Ch c, size_type i = 0) const;

    // поиск подпоследовательности, начиная с конца (подобно find_end(), § 18.5.5):
    size_type rfind (const basic_string& s, size_type i = npos) const;
    size_type rfind (const Ch* p, size_type i, size_type n ) const;
    size_type rfind (const Ch* p, size_type i = npos) const;
    size_type rfind (Ch c, size_type i = npos) const;

    // поиск символа (наподобие find_first_of() в § 18.5.5):
    size_type find_first_of (const basic_string& s, size_type i = 0) const;
    size_type find_first_of (const Ch* p, size_type i, size_type n) const;
    size_type find_first_of (const Ch* p, size_type i = 0) const;
    size_type find_first_of (Ch c, size_type i = 0) const;

    // поиск символа из аргумента, начиная с конца:
    size_type find_last_of (const basic_string& s, size_type i = npos) const;
    size_type find_last_of (const Ch* p, size_type i, size_type n ) const;
```

```
        size_type find_last_of(const Ch* p, size_type i = npos) const;
        size_type find_last_of(Ch c, size_type i = npos) const;

        // поиск символа, которого нет в аргументе:
        size_type find_first_not_of(const basic_string& s, size_type i = 0) const;
        size_type find_first_not_of (const Ch* p, size_type i, size_type n ) const;
        size_type find_first_not_of (const Ch* p, size_type i = 0) const;
        size_type find_first_not_of (Ch c, size_type i = 0) const;

        // поиск символа, которого нет в аргументе, начиная с конца:
        size_type find_last_not_of(const basic_string& s, size_type i = npos) const;
        size_type find_lastst_not_of (const Ch* p, size_type i, size_type n ) const;
        size_type find_larst_not_of (const Ch* p, size_type i = npos) const;
        size_type find_last_not_of (Ch c, size_type i = npos) const;
        // ...
};
```

Все это константные члены. То есть они существуют для того, чтобы локализовать подстроку для какого-либо использования, но не изменяют значения строки, к которой применяются.

Значение функций ***basic_string::find*** можно понять по их аналогам среди универсальных алгоритмов. Рассмотрим пример:

```
void f()
{
        string s="accdcde";

        string::size_type i1 = s.find ("cd");          // i1=2 s[2]=='c' && s[3]=='d'
        string::size_type i2 = s.rfind ("cd");         // i1=4 s[4]=='c' && s[5]=='d'
        string::size_type i3 = s.find_first_of("cd");  // i3=1 s[1]=='c'
        string::size_type i4 = s.find_last_of("cd");   // i4=5 s[5]=='d'
        string::size_type i5 = s.find_first_not_of("cd"); // i5=0 s[0]!='c' && s[0]!='d'
        string::size_type i6 = s.find_last_not_of("cd"); // i6=6 s[6]!='c' && s[6]!='d'
}
```

Если такая функция ***find*** () не находит ничего, она возвращает ***npos***, что означает запрещенную позицию символа. Если ***npos*** используется как позиция символа, сгенерируется ***out_of_range*** (§ 20.3.5).

20.3.12. Замена

Когда позиция в строке определена, можно изменить отдельный элемент при помощи его индекса или заменить целую подстроку новыми символами при помощи функции ***replace*** ():

```
template<class Ch, class Tr = char_traits<Ch>, class A=allocator<Ch> >
class basic_string {
public:
        // ...
        // замена [(*this)[i], (*this)[i+n] [ другими символами:
        basic_string& replace (size_type i, size_type n, const basic_string& str);
        basic_string& replace (size_type i, size_type n,
                               const basic_string& s, size_type i2, size_type n2 );
        basic_string& replace (size_type i, size_type n, const Ch* p, size_type n2);
```

```
        basic_string& replace (size_type i, size_type n, const Ch*p);
        basic_string& replace (size_type i, size_type n, size_type n2, Ch c);

        basic_string& replace (iterator i, iterator i2, const basic_string& str);
        basic_string& replace (iterator i, iterator i2, const Ch* p, size_type n);
        basic_string& replace (iterator i, iterator i2, const Ch* p);
        basic_string& replace (iterator i, iterator i2, size_type n, Ch c);
        template<class In>basic_string& replace (iterator i, iterator i2, In j, In j2);

        // удаление символов из строки («замена ничем»):

        basic_string& erase (size_type i=0, size_type n = npos);
        iterator erase (iterator i);
        iterator erase (iterator first, iterator last);
        void clear ();                                  // удаляет все символы

        // ...
};
```

Отметим, что число символов в новой строке не обязательно должно быть тем же, что было в строке раньше. Размер строки изменяется, чтобы принять в себя новую подстроку. В частности, ***erase*** () просто удаляет подстроку и соответственно изменяет размер строки. Например:

```
void f()
{
        string s = "но я расскажу эту историю, даже если вы ей не поверите";
        s.erase (0, 3);       // удаляем "но "
        s.replace (s.find ("даже"), 4, "лишь");
        s.replace (s.find ("не"), 2, "");  // удаление заменой на ""
}
```

Простой вызов ***erase*** () без аргументов превратит строку в пустую строку. Эта операция для универсальных контейнеров называется ***clear*** () (§ 16.3.6).

В своем разнообразии функции ***replace*** () не уступают присваиваниям. В конце концов, ***replace*** () — это присваивание нового значения подстроке.

20.3.13. Подстроки

Функция ***substr*** () позволяет вам указать подстроку, задав позицию и длину:

```
template<class Ch, class Tr = char_traits<Ch>,
        class A = allocator<Ch> >
class basic_string {
public:
        // ...
        // адрес подстроки:

        basic_string substr (size_type i=0; size_type n = npos) const;
        void clear ();                                  // удаляет все символы
        // ...
};
```

Функция ***substr*** () — это просто способ прочитать часть строки. С другой стороны, ***replace*** () позволяет вам вписать что-либо в подстроку. И та, и другая используют низкоуровневую позицию и указанное число символов. Однако ***find*** () позволяет нам находить подстроку по ее значению. Все вместе они дают возможность определить подстроку, которую можно использовать как для чтения, так и для записи:

```
template<class Ch> class Basic_substring {
public:
        typedef typename basic_string<Ch>::size_type size_type;

        Basic_substring (basic_string<Ch>& s, size_type i, size_type n);      // s[i]..s[i+n-1]
        Basic_substring (basic_string<Ch>& s, const basic_string<Ch>& s2);    // s2 в s
        Basic_substring (basic_string<Ch>& s, const Ch* p);                   // *p в s
        Basic_substring& operator= (const basic_string<Ch>&);                 // запись через *ps
        Basic_substring& operator= (const Basic_substring<Ch>&);
        Basic_substring& operator= (const Ch*);
        Basic_substring& operator= (Ch);

        operator basic_string<Ch> () const;                                   // чтение из *ps
        operator Ch* () const;
private:
        basic_string<Ch>* ps;
        size_type pos;
        size_type n;
};
```

Реализация достаточно тривиальна. Например:

```
template<class Ch>
Basic_substring<Ch>::Basic_substring (basic_string<Ch>& s, const basic_string<Ch>& s2)
        : ps (&s), n (s2.length ())
{
        pos = s.find (s2);
}

template<class Ch> Basic_substring<Ch>&
        Basic_substring<Ch>::operator= (const basic_string<Ch>& s)
{
        ps->replace (pos, n, s);                    // запись через указатель *ps
        return *this;
}

template<class Ch>
Basic_substring<Ch>::operator basic_string<Ch> () const
{
        return basic_string<Ch> (*ps, pos, n);      // копирование из *ps
}
```

Если *s2* не найдена в *s*, *pos* будет равен *npos*. Попытки прочитать или записать по такой позиции будут генерировать *out_of_range* (§ 20.3.5).

Этот *Basic_substring* можно использовать так:

```
typedef Basic_substring<char> Substring;

void f ()
{
        string s = "Наша Маша громко плачет";
        Substring (s, "Маша") = "Миша";
        Substring (s, "громко") = "никогда не";

        strring s2 = "Наш" + Substring (s, s.find (' '), string::npos);
}
```

Естественно, было бы гораздо интереснее, если бы функция *Substring* умела искать по образцу (§ 20.6[7]).

20.3.14. Размер и емкость

Функции, связанные с выделением памяти, очень похожи на аналогичные функции для векторов:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        // ...
        // размер, емкость и пр. (подобно § 16.3.8)
        size_type size () const;                    // число символов (§ 20.3.4)
        size_type max_size () const;                // максимальная длина строки
        size_type length () const { return size (); }
        bool empty () const { return size () == 0; }

        void resize (size_type n, Ch c);
        void resize (size_type n) { resize (n, Ch ()); }

        size_type capacity () const;                // как для vector: § 16.3.8
        void reserve (size_type res_arg = 0);       // как для vector: § 16.3.8

        allocator_type get_allocator () const;
};
```

Вызов *reserve* (*res_arg*) генерирует *length_error*, если *res_arg>max_size* ().

20.3.15. Операции ввода/вывода

Одно из главных применений строк — использование их как приемника при вводе и как источника при выводе. Операторы ввода и вывода для *basic_string* предоставляются <*string*> (но не <*iostream*>):

```
template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& operator>> (basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& getline (basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&, Ch eol);

template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& getline (basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&);
```

Оператор << пишет строку в *ostream* (§ 21.2.1). Оператор >> считывает слово, ограниченное «символом-разделителем» (whitespace) (§ 3.6, § 21.3.1), в указанную строку, расширяя ее по мере необходимости, чтобы вместить все слово. Первые символы-разделители пропускаются, а завершающий символ-разделитель в строку не вводится.

Функция *getline* () считывает строку, оканчивающуюся символом *eol* (end of line — конец строки) в указанную строку, по мере надобности расширяя последнюю. Если не указан аргумент *eol*, ограничителем считается символ новой строки — '\n'. Ограничитель из потока удаляется, но в строку не вводится. Поскольку, чтобы вместить ввод, *string* расширяется, не имеет смысла оставлять символ конца строки в потоке или считать символы, как это делают *get* () и *getline* () для символьных массивов (§ 21.3.4).

20.3.16. Перемена местами

Как и для векторов (§ 16.3.9), функции *swap* () для строк могут оказаться гораздо эффективнее, чем универсальные алгоритмы, поэтому для них введены специальные версии:

```
template<class Ch, class Tr, class A>
void swap (basic_string<Ch, Tr, A>&, basic_string<Ch, Tr, A>&);
```

20.4. Стандартная библиотека C

Стандартная библиотека C++ унаследовала из библиотеки C функции для работы с C-строками. В этот раздел включены несколько самых полезных из них. Описание не предполагается исчерпывающим, и за дополнительной информацией обращайтесь к вашему справочному руководству. Учтите, что разработчики реализаций часто добавляют в стандартные заголовочные файлы свои собственные нестандартные функции, поэтому очень легко запутаться, какие функции гарантированно доступны для всех реализаций, а какие нет.

Заголовочные файлы, представляющие возможности стандартной библиотеки C, перечислены в § 16.1.2. Функции распределения памяти можно найти в § 19.4.6, функции ввода/вывода языка C — в § 21.8, а математическую библиотеку C — в § 22.3. Функции, относящиеся к запуску и завершению программ, описаны в § 3.2 и § 9.4.1.1, а возможности чтения неопределенных аргументов функции представлены в § 7.6. Функции в стиле C для строк с расширенным набором символов находятся в <*cwchar*> и <*wchar.h*>.

20.4.1. C-строки

Функции для манипулирования C-строками находятся в заголовочных файлах <*string.h*> и <*cstring*>:

```
char* strcpy (char* p, const char* q);      // копирование из q в p (включая конец строки)
char* strcat (char* p, const char* q);      // добавление q в p (включая конец строки)
char* strncpy (char* p, const char* q, int n);   // копирование n символов из q в p
char* strncat (char* p, const char* q, int n);   // добавление n символов из q в p

size_t strlen (const char* p);                   // длина p (не считая конца строки)

int strcmp (const char* p, const char* q);       // сравнение p и q
int strncmp (const char* p, const char* q, int n);   // сравнение первых n символов

char* strchr (char* p, int c);                   // поиск первого вхождения c в p
const char* strchr (const char* p, int c);

char* strrchr (char* p, int c);                  // поиск последнего вхождения c в p
const char* strrchr (const char* p, int c);

char* strstr (char* p, const char* q);           // поиск первого вхождения q в p
const char* strstr (const char* p, const char* q);

char* strpbrk (char* p, const char* q);          // поиск в p первого вхождения
                                                 // символа из q
const char* strpbrk (const char* p, const char* q);

size_t strspn (const char* p, const char* q);    // число символов в p до любого
                                                 // символа, не встречающегося в q
size_t strcspn (const char* p, const char* q);   // число символов в p до любого
                                                 // символа, встречающегося в q
```

Считается, что указатель не равен нулю, а массив элементов ***char***, на который он указывает, заканчивается нулем ***0***. Функции ***strn***, если в строке не набирается ***n*** элементов для копирования, останавливаются при обнаружении нуля. Сравнение строк возвращает ***0***, если строки равны, отрицательное число — если первый аргумент лексикографически следует до второго, и положительное число в противном случае.

Естественно, C не обеспечивает пар перегруженных функций. Однако, они нужны в C++ для обеспечения константных вызовов. Например:

```
void f(const char* pcc, char* pc)          // C++
{
    *strchr(pcc,'a') = 'b';                // ошибка: const char нельзя присваивать
    *strchr(pc,'a') = 'b';                 // правильно, но неаккуратно: в pc может
                                           // не оказаться 'a'
}
```

Функция ***strchr*** () в C++ не позволяет записывать в константный аргумент. Однако программы на C могут «воспользоваться преимуществом» более слабого контроля типов в C в функции ***strchr*** ():

```
/* функция из стандартной библиотеки C, не C++ */
char* strchr(const char* p, int c);

/* C, в C++ не скомпилируется */
void g(const char* pcc, char* pc)
{
    /* преобразует const в не-const: правильно на C, ошибка на C++ */
    *strchr(pcc,'a') = 'b';
    /* правильно и на C, и на C++ */
    *strchr(pc,'a') = 'b';
}
```

По мере возможности C-строк лучше избегать, предпочитая им строки ***string***. C-строки и связанные с ними стандартные функции можно использовать для получения очень эффективного кода, но даже опытные программисты на C и C++ порой допускают при работе с ними «глупые ошибки». Однако программист на C++ не может не встретить некоторые из этих функций в старом коде. Здесь я приведу бессмысленный пример, иллюстрирующий наиболее распространенные функции:

```
void f(char* p, char* q)
{
    if (p==q) return;                  // указатели равны
    if (strcmp(p,q)==0) {              // значения строк равны
        int i = strlen(p);             // число символов (не считая конца строки)
        //...
    }
    char buf[200];
    strcpy(buf,p);                     // копирование p в buf (включая конец строки)
                                       // неаккуратно: когда-нибудь переполнится
    strcpy(buf,p,200);                 // копирование 200 символов из p в buf
                                       // неаккуратно: когда-нибудь не удастся
                                       // скопировать завершающий символ
    //...
}
```

Ввод и вывод C-строк обычно производится при помощи семейства функций ***printf*** (§ 21.8).

В <*stdlib.h*> и <*cstdlib*> стандартная библиотека вводит полезные функции для преобразования строк, представляющих численные значения, в сами численные значения:

```
double atof(const char* p);              // преобразует p[] в double ("alpha to floating")
double strtod(const char* p, char** end); // преобразует p[] в double ("string to double")
int atoi(const char* p);                 // преобразует p[] в десятичный int
long atol(const char* p);                // преобразует p[] в десятичный long
long strtol(const char* p, char** end, int b); // преобразует p[] в long по основанию b
```

Символы-разделители в начале игнорируются. Если входная строка не представляет собой число, возвращается ***0***. Например, значение ***atoi("семь")*** равно ***0***.

Если в вызове ***strtol(p, end, b)*** значение ***end*** не равно нулю, положение первого непрочитанного символа во входной строке может быть доступно через присвоение ****end***. Если ***b==0***, число интерпретируется как целый литерал C++ (см. § 4.4.1). Например, префикс ***0x*** означает шестнадцатеричное число, ***0*** — восьмеричное и т. д.

Что произойдет, если через ***atof()***, ***atoi()*** или ***atol()*** преобразовывать величины, которые не имеют представления в соответствующем возвращаемом типе, не определено. Если входная строка ***strtol()*** представляет число, которое не может быть представлено через ***long int***, или если входная строка ***strtod*** представляет число, непредставимое через ***double***, то переменная ***errno*** (§ 16.1.2, § 22.3) устанавливается в ***ERANGE***, и возвращается соответствующее очень большое или очень маленькое значение.

За исключением обработки ошибок, ***atof(s)*** эквивалентно ***strtod(s,0)***, ***atoi(s)*** эквивалентно ***int strtol(s,0,10)*** и ***atol(s)*** эквивалентно ***strtol(s,0,10)***.

20.4.2. Классификация символов

В <*ctype.h*> и <*cctype*> стандартная библиотека предоставляет набор полезных функций для работы с символами из ASCII и других подобных ему наборов:

```
int isalpha(int);   // буква: 'a'..'z' 'A'..'Z' в местном алфавите C (§ 20.2.1, § 21.7)
int isupper(int);   // буква в верхнем регистре: 'A'..'Z'
                    // в местном алфавите C (§ 20.2.1, § 21.7)
int islower(int);   // буква в нижнем регистре: 'a'..'z'
                    // в местном алфавите C (§ 20.2.1, § 21.7)
int isdigit(int);   // '0'..'9'
int isxdigit(int);  // '0'..'9' или буква
int isspace(int);   // символы-разделители
int iscntrl(int);   // управляющие символы (ASCII 0..31 и 127)
int ispunct(int);   // пунктуация: ни один символ из вышеупомянутых
int isalnum(int);   // isalpha() | isdigit()
int isprint(int);   // то, что можно напечатать: ascii ' '..'~'
int isgraph(int);   // isalpha() | isdigit() | ispunct()

int toupper(int c); // эквивалент c в верхнем регистре
int tolower(int c); // эквивалент c в нижнем регистре
}
```

Все они обычно реализуются простым поиском с использованием символа в качестве индекса в таблице атрибутов символов. Это значит, что такие конструкции, как:

```
if ('a'<=c && c<='z') || ('A'<=c && c<='Z') {        // с из алфавита?
        //...
}
```

неэффективны, не говоря о нудности их написания и большой вероятности ошибок (на машинах с символьным набором EBCDIC через это сравнение пройдут и неалфавитные символы).

Эти стандартные функции принимают в качестве аргумента *int* и передаваемое целое должно быть представимо как *unsigned char* или *EOF* (который чаще всего имеет значение *-1*). Это может оказаться проблемой в системах, где тип *char* имеет знак (см. § 20.6[11]).

Аналогичные функции для расширенных символов находятся в заголовочных файлах <*cwctype*> и <*wctype.h*>.

20.5. Советы

[1] Функциям с С-строками предпочитайте операции со *string*; § 20.4.1.

[2] Пользуйтесь типом *string* для переменных и членов, а не как базовым классом; § 20.3, § 25.2.1.

[3] Вы можете передавать строки *string* как аргументы по значению и возвращать их по значению, предоставив системе самой заботиться о распределении памяти; § 20.3.6.

[4] Когда вы хотите проверять диапазон, пользуйтесь *at* (), а не итераторами и []; § 20.3.2, § 20.3.5.

[5] Когда вы хотите оптимизировать скорость, пользуйтесь итераторами или индексацией [], а не *at* (); § 20.3.2, § 20.3.5.

[6] Прямо или косвенно пользуйтесь *substr* (), чтобы считывать подстроки, и *replace* (), чтобы записывать подстроки; § 20.3.12, § 20.3.13.

[7] Чтобы локализовать значение внутри *string*, пользуйтесь операциями *find* () (а не пишите явный цикл); § 20.3.11.

[8] Если к *string* нужно эффективно добавить символы, пользуйтесь *append* (); § 20.3.9.

[9] Если быстродействие некритично, используйте строки *string* для символьного ввода; § 20.3.5.

[10] Для обозначения «остаток строки» пользуйтесь записью *string::npos*; § 20.3.5.

[11] При необходимости реализуйте интенсивно используемые классы строк с помощью операций низкого уровня (не применяйте низкоуровневые структуры данных повсюду); § 20.3.5.

[12] Если вы пользуетесь типом *string*, где-нибудь перехватывайте исключения *length_error* и *out_of_range*; § 20.3.5.

[13] Будьте осторожны: не передавайте *char** со значением *0* в функцию, работающую со строками *string*; § 20.3.5.

[14] Пользуйтесь *c_str* для того, чтобы получить представление строки в виде С-строки только тогда, когда это необходимо; § 20.3.7.

[15] Когда хотите узнать вид символа (алфавитный, число, и т. д.), пользуйтесь *isalpha* (), *isdigit* () и т. п., а не пишите собственные проверки; § 20.4.2.

20.6. Упражнения

Решение некоторых задач к этой главе можно найти, просмотрев исходный текст реализации стандартной библиотеки. Сделайте одолжение — попытайтесь найти собственные решения, прежде чем смотреть, как к этим проблемам подошел разработчик вашей библиотеки.

1. (*2) Напишите функцию, принимающую две строки *string* и возвращающую строку, которая является конкатенацией этих двух строк с точкой между ними. Например, если дано *file* и *write*, функция должна возвратить *file.write*. Сделайте тоже самое с С-строками, используя только возможности С, такие как *malloc* () и *strlen* (). Сравните две функции. Что будет подходящим критерием сравнения?
2. (*2) Перечислите различия между *vector* и *basic_string*. Какие различия существенны?
3. (*2) Средства для работы со строками не слишком регулярны. Например, вы можете присвоить строке значение типа *char*, но не можете инициализировать *string* значением *char*. Составьте список таких нерегулярностей. Какие из них можно устранить, не усложняя использование строк? Какие другие нерегулярности это вызовет?
4. (*1.5) В классе *basic_string* множество членов. Какие из них могли бы быть нечленами без потери эффективности и удобства записи?
5. (*1.5) Напишите версию *back_inserter* () (§ 19.2.4), работающую с *basic_string*.
6. (*2) Завершите тип *Basic_substring* из § 20.3.13 и интегрируйте его с типом *String*, чтобы перегрузить оператор () для обозначения «подстрока из», а в остальном получившийся тип должен действовать как *string*.
7. (*2.5) Напишите функцию *find* (), которая находит первое вхождение простого регулярного выражения в строку. Используйте ? для обозначения «любой символ», * — для обозначения любого числа символов, не совпадающих со следующей частью регулярного выражения, и [*abc*] — для обозначения любого символа из тех, что стоят между квадратными скобками (здесь *a*, *b* и *c*). Другие символы должны совпадать сами с собой. Например, *find* (*s*, "*name*: ") возвращает указатель на первое вхождение *name*: в строку *s*; *find* (*s*, "[*nN*]*ame*: ") возвращает указатель на первое вхождение в строку *s* либо *name*: либо *Name*: ; а *find* (*s*, "[*nN*]*ame* (*)") возвращает указатель на первое вхождение в строку *s Name* или *name* с последующими (возможно, пустыми) символьными последовательностями в скобках.
8. (*2.5) Каких операций, вы считаете, не хватает среди функций, работающих с простыми регулярными выражениями из § 20.6[7]? Определите их и добавьте. Сравните выразительность ваших функций для поиска регулярных выражений с широко распространенными. Сравните скорость работы ваших средств с распространенными.
9. (*2.5) Воспользуйтесь библиотекой регулярных выражений для реализации операций поиска по образцу над классом *String*, имеющим ассоциированный класс *Substring*.
10. (*2.5) Подумайте, как написать «идеальный» класс для универсальной обработки текстов. Назовите его *Text*. Какими возможностями он должен обладать? Какие ограничения на реализацию и какие затраты вызовут ваши «идеальные» средства?

11. (*1.5) Определите набор перегруженных версий для ***alpha*** (), ***isdigit*** () и т. п., так чтобы эти функции правильно работали с ***char***, ***unsigned char*** и ***signed char***.
12. (*2.5) Напишите класс ***String***, оптимизированный для строк не более чем из восьми символов. Сравните его быстродействие со ***String*** из § 11.12 и вашей версии, реализующей стандартную библиотечную ***string***. Можно ли создать строку, сочетающую преимущества строк, оптимизированных для очень малой длины, с преимуществами совершенно универсальных строк?
13. (*2) Оцените скорость копирования строк ***string***. Значительно ли оптимизирует копирование ваша реализация строк?
14. (*2.5) Сравните быстродействие трех функций ***complete_name*** () из § 20.3.9 и § 20.3.10. Попытайтесь написать версию ***complete_name*** (), работающую с максимальной скоростью. Ведите запись ошибок, выявленных во время реализации и проверки.
15. (*2.5) Представьте себе, что считывание строк средней длины (в большинстве своем от 5 до 25 символов) из потока ***cin*** — самое узкое место в вашей системе. Напишите функцию ввода, читающую такие строки с максимальной скоростью (насколько сумеете придумать). Вы можете выбрать интерфейс к этой функции, чтобы оптимизировать быстродействие, а не удобство. Сравните результат с вашей реализацией для строк ***string***.
16. (*1.5) Напишите функцию ***itos*** (***int***), которая возвращала бы строковое представление своего целочисленного аргумента.

21

Потоки

То, что вы видите, — это все, что вы получите.
— Брайан Керниган

Ввод и вывод — потоки ***ostream*** — вывод встроенных типов — вывод типов, определяемых пользователем — виртуальные функции вывода — потоки ***istream*** — ввод встроенных типов — неформатированный ввод — состояние потока — ввод типов, определяемых пользователем — исключения ввода/вывода — связывание потоков — часовые — форматированный вывод целых и чисел с плавающей точкой — поля и выравнивания — манипуляторы — стандартные манипуляторы — манипуляторы, определяемые пользователем — файловые потоки — закрытие потоков — строковые потоки — строковые буферы — буферы потоков — национальные особенности — функции обратного вызова для потоков — ***printf*** () — советы — упражнения.

21.1. Введение

Проектирование и реализация средств универсального ввода/вывода для языка программирования представляет значительную трудность. Традиционно средства ввода/вывода проектировались исключительно для работы с несколькими встроенными типами данных. Однако нетривиальные программы на C++ пользуются множеством типов, определяемых пользователем, и для этих типов нужно осуществлять ввод/вывод. Средства ввода/вывода должны быть просты, удобны и безопасны в использовании, эффективны и гибки, и, самое главное, полны. Никто еще не нашел решения, которое понравилось бы всем. Поэтому должна быть возможность обеспечить альтернативные средства ввода/вывода и расширить стандартный ввод/вывод в соответствии с требованиями прикладных программ.

C++ разрабатывался для того, чтобы дать пользователю возможность определять новые типы, такие же эффективные и удобные в использовании, как и встроенные. И потому имеет смысл потребовать, чтобы средства ввода/вывода для C++ реализовывались на C++ с использованием только тех возможностей, которые доступны каждому программисту. Представленные здесь средства для работы с потоками ввода/вывода являются следствием попытки справиться с этой задачей:

§ 21.2 *Вывод*: То, что разработчик прикладных программ считает выводом, на самом деле является преобразованием объектов некоторого типа, такого как ***int***, ***char****** и ***Employee_record***, в последовательность символов. Здесь

описывается возможность записи на вывод встроенных типов и типов, определяемых пользователем.

§ 21.3 *Ввод*: Здесь представлены средства для запроса на ввод символов, строк и значений других встроенных типов и типов, определяемых пользователем.

§ 21.4 *Форматирование*: На расположение выводимой информации часто накладываются специальные требования. Например, может возникнуть необходимость вывести число ***int*** в десятичном виде, а указатель в шестнадцатеричном, или числа с плавающей точкой должны быть записаны со строго определенной степенью точности. Здесь обсуждается управление форматированием и соответствующие приемы программирования.

§ 21.5 *Файлы и потоки*: По умолчанию каждая программа на C++ может пользоваться стандартными потоками, такими как стандартный вывод (***cout***), стандартный ввод (***cin***) и вывод ошибок (***cerr***). Чтобы пользоваться другими устройствами и файлами, потоки должны создаваться и прикрепляться к этим устройствам и файлам. Здесь описываются механизмы открытия и закрытия файлов, а также прикрепления потоков к файлам и строкам ***string***.

§ 21.6 *Буферизация*: Чтобы сделать ввод/вывод эффективным, мы должны воспользоваться стратегией буферизации, удобной как для записываемых (считываемых) данных, так и для приемника, куда они пишутся (или откуда читаются). Здесь представлены основные методы буферизации потоков.

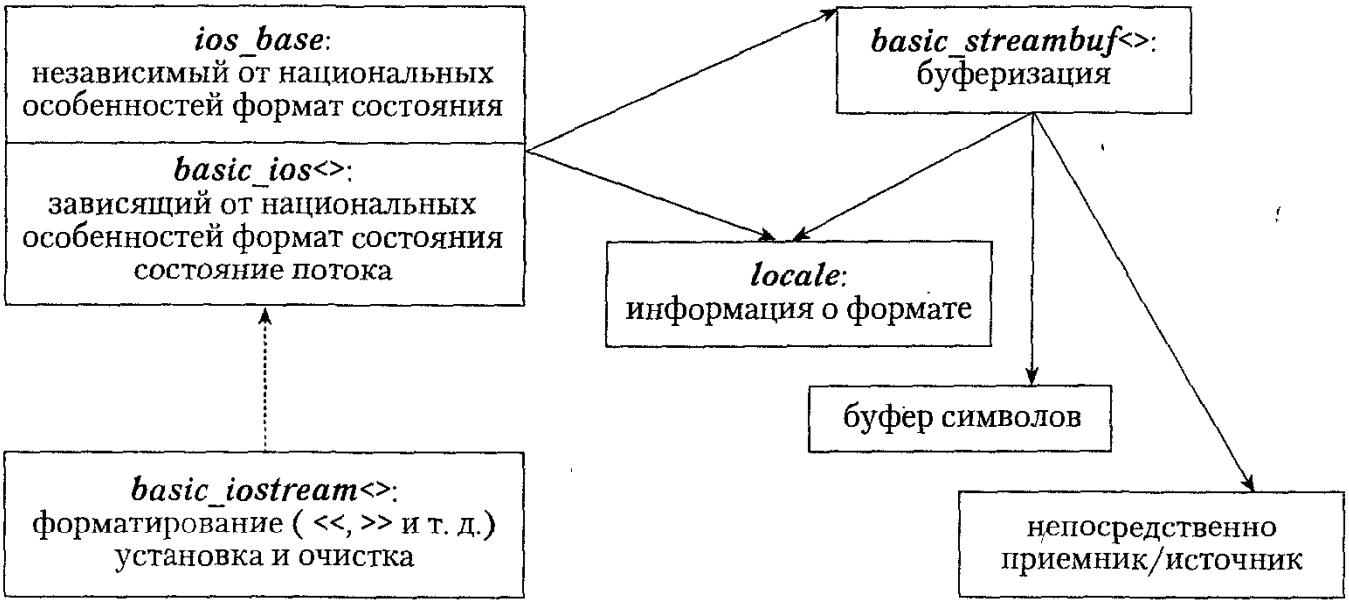
§ 21.7 *Национальные особенности*: Объект ***locale*** описывает, как печатаются числа, какие символы считаются буквами и т. п. Он инкапсулирует многие различия в культурных традициях. Национальные особенности используются системой ввода/вывода в неявном виде и описаны только вкратце.

§ 21.8 *Ввод/вывод в стиле C*: Рассматриваются функции ***printf***() из библиотеки C <***stdio.h***> и связи этой библиотеки с библиотекой C++ <***iostream***>.

Для использования библиотеки не нужно знать методы, примененные при реализации библиотеки потоков. К тому же эти приемы в разных реализациях различаются. Однако реализация ввода/вывода — сложная и интересная задача. Ее решение содержит примеры, которые можно применить ко многим другим задачам программирования и проектирования. Поэтому стоит изучить методы, использованные при реализации ввода/вывода.

В этой главе обсуждается система ввода/вывода в той степени, которая позволит вам оценить структуру этой системы, использовать ее для самых употребительных видов ввода/вывода и распространить ввод/вывод на работу с типами, определяемыми пользователем. Если вам понадобится реализовать стандартные потоки ввода/вывода, ввести новый вид потока или учесть национальные особенности, вам понадобится копия стандарта, хорошее руководство по системе и/или примеры работающего кода в добавок к тем, что представлены здесь.

Ключевые компоненты систем потокового ввода/вывода можно представить графически:



Пунктирная стрелка от *basic_iostream*<> показывает, что *basic_ios*<> является виртуальным базовым классом; сплошные стрелки представляют указатели. Классы, помеченные <>, — это шаблоны, параметризованные символьным типом, содержащие национальные особенности.

Концепция потоков и обеспечиваемая ей универсальная система обозначений могут быть применены к широкому классу задач, связанных с коммуникациями. Потоки используются для передачи объектов между компьютерами (§ 25.4.1), для шифровки сообщений в потоках (§ 21.10[22]), для сжатия данных, для долговременного хранения объектов и еще для очень многих целей. Однако рассмотрение их здесь ограничено только простым, ориентированным на символы вводом и выводом.

Объявления классов потоков ввода/вывода и шаблонов (достаточные для обращения к ним, но не достаточные для выполнения операций над ними) и стандартные определения типов *typedef* представлены в *<iosfwd>*. Этот заголовочный файл порой требуется, когда вы хотите включить некоторые, но не все заголовочные файлы ввода/вывода.

21.2. Вывод

Безопасный с точки зрения типов и универсальный подход как к встроенным, так и к типам, определяемым пользователем, может быть достигнут с использованием единого перегруженного имени для набора функций вывода. Например:

```
put(cerr,"x = "); // cerr — это выходной поток сообщений об ошибках
put(cerr,x);
put(cerr,'\n');
```

Тип аргумента определяет, какая функция *put* будет для него вызвана. Такое решение используется в некоторых других языках. Однако, это приводит к повторениям. Перегрузка оператора << для обозначения «записать в» (put to) дает более удобное написание и позволяет программисту одной инструкцией выводить последовательность объектов. Например:

```
cerr << "x = " << x << '\n';
```

Если *x* относится к типу *int* и имеет значение *123*, такая инструкция выведет:

```
x = 123
```

и далее переведет строку в стандартном потоке ошибок *cerr*. Подобным образом если *x* относится к типу *complex* (§ 22.5) и имеет значение {*1*, *2.4*}, данное выражение выведет в *cerr*:

```
x = {1, 2.4}
```

Таким стилем можно пользоваться, если *x* относится к типу, для которого определен оператор <<, а для нового типа оператор << пользователь может легко определить сам.

Оператор вывода нужен для того, чтобы избежать многословности, которая получается при использовании функций вывода. Но почему именно <<? Новый лексический знак придумать невозможно(§ 11.2). Для ввода и вывода предлагалось использовать оператор присваивания, но, кажется, большинство людей предпочитает разные операторы для ввода и для вывода. Кроме того, оператор = неправильно связывает, то есть *cout=a=b* означает скорее *cout= (a=b)*, чем *(cout=a) =b* (§ 6.2). Я пробовал < и >, но их смысл «меньше чем» и «больше чем» так крепко врос в людские умы, что новые выражения, обозначающие ввод/вывод, для всех практических применений оказались просто нечитабельны.

Операторы << и >> используются для встроенных типов не так часто, чтобы вызвать проблему. Они симметричны в том смысле, что могут использоваться для обозначения «в» и «из». Когда они применяются для ввода/вывода, я обращаюсь к << как к *записать в*, а к >> — как к *прочесть из*. Те, кто предпочитает более «технические» наименования, могут называть их соответственно *инсертерами* и *экстракторами*. Приоритет операции << не очень высок и позволяет вставлять арифметические выражения в операнды без скобок. Например:

```
cout << "a*b+c= " << a*b+c << '\n';
```

Скобки нужно использовать при написании выражений с операторами, чей приоритет ниже чем у << . Например:

```
cout << "a^b|c= " << (a^b|c) << '\n';
```

Оператор сдвига влево (§ 6.2.4) также можно использовать в выражении для вывода, но, конечно, он тоже должен писаться в скобках:

```
cout << "a<<b= " << (a<<b) << '\n';
```

21.2.1. Потоки вывода

ostream — это механизм для преобразования значений различного типа в последовательность символов. Обычно эти символы затем выводятся при помощи низкоуровневых операций вывода. Есть много видов символов (§ 20.2), которые можно охарактеризовать при помощи свойств символов *char_traits* (§ 20.2.1). Следовательно, *ostream* является специализацией для конкретных видов символов универсального шаблона *basic_ostream*:

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
        virtual ~basic_ostream ();
        // ...
};
```

Этот шаблон и ассоциированные с ним операции вывода определены в пространстве имен ***std*** и представлены заголовочным файлом <***iostream***>.

Параметры шаблона ***basic_iostream*** контролируют используемый реализацией тип символов; они не влияют на возможные типы выводимых значений. Потоки, реализованные с использованием обычных ***char***, и те, что реализованы с использованием расширенных символов, непосредственно поддерживаются для каждой реализации:

```
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;
```

Во многих системах запись расширенных символов можно оптимизировать посредством ***wostream*** , но такая оптимизация плохо подойдет для потоков, где единицей ввода/вывода является байт

Можно определить потоки, для которых физический ввод/вывод производится не в виде символов. Однако такие потоки находятся за пределами стандарта C++ и этой книги (§ 21.10[15]).

Базовый класс ***basic_ios*** представлен в <***ios***>. Он контролирует форматирование (§ 21.4), национальные особенности (§ 21.7) и обращение к буферам (§ 21.6). Он также определяет несколько типов, служащих для удобства записи:

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
public:
        typedef Ch char_type;
        typedef Tr traits_type;
        typedef typename Tr::int_type int_type;        // тип целочисленного
                                                       // значения символа
        typedef typename Tr::pos_type pos_type;        // позиция в буфере
        typedef typename Tr::off_type off_type;        // смещение в буфере

        // ... см. также: § 21.3.3, § 21.3.7, § 21.4.4, § 21.6.3 и § 21.7.1
};
```

Класс ***basic_ios*** запрещает копирующие конструктор и присваивание (§ 11.2.2). Соответственно, потоки ***ostream*** и ***istream*** нельзя копировать. Поэтому, если вам нужно изменить приемник потока, следует либо заменить буферы потоков (§ 21.6.4), либо воспользоваться указателями (§ 11.2.2).

Класс ***ios_base*** содержит информацию и операции, не зависящие от типа используемых символов — например, точность для вывода числа с плавающей точкой. Поэтому он не обязан быть шаблоном.

Кроме определений ***typedef*** в ***ios_base*** библиотека потоков ввода/вывода использует интегральный со знаком тип ***streamsize*** для представления размера буферов и числа символов, передаваемых в операциях ввода/вывода.

В <***iostream***> объявлено несколько стандартных потоков:

```
ostream cout;       // стандартный символьный поток вывода
ostream cerr;       // стандартный небуферизованный поток вывода
                    // для сообщений об ошибках
ostream clog;       // стандартный поток вывода для сообщений об ошибках

wostream wcout;     // расширенный поток, соответствующий cout
wostream wcerr;     // расширенный поток, соответствующий cerr
wostream wclog;     // расширенный поток, соответствующий clog
```

Потоки *cerr* и *clog* относятся к одному и тому же получателю данных; они различаются только буферизацией. Поток *cout* записывает туда же, что и *stdout* в C (§ 21.8), в то время как *cerr* и *clog* записывают туда же, куда и *stderr*. По мере необходимости программист может создавать новые потоки (см. § 21.5).

21.2.2. Вывод встроенных типов

Класс *ostream* определяет оператор << («записать в») для управления выводом встроенных типов:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
        // ...

        basic_ostream& operator<< (short n);
        basic_ostream& operator<< (int n);
        basic_ostream& operator<< (long n);

        basic_ostream& operator<< (unsigned short n);
        basic_ostream& operator<< (unsigned int n);
        basic_ostream& operator<< (unsigned long n);

        basic_ostream& operator<< (float f);
        basic_ostream& operator<< (double f);
        basic_ostream& operator<< (long double f);

        basic_ostream& operator<< (bool n);
        basic_ostream& operator<< (const void* p);          // записать значение указателя

        basic_ostream& put (Ch c);                          // записать c
        basic_ostream& write (const Ch* p, streamsize n);   // p[0]..p[n-1]

        // ...
};
```

Функции *put* () и *write* () просто записывают символы. Следовательно, оператор << для вывода может и не быть членом. Функции *operator*<< (), принимающие в качестве аргумента символьный операнд, можно реализовать как не-члены при помощи *put* ():

```
template<class Ch, class Tr>
        basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, Ch);
template<class Ch, class Tr>
        basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, char);
template<class Tr>
        basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, char);
template<class Tr>
        basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, signed char);
template<class Tr>
        basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, unsigned char);
```

Подобным же образом вводится оператор << для записи оканчивающихся нулем символьных массивов:

```
template<class Ch, class Tr>
```

```
        basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, const Ch*);
template<class Ch, class Tr>
        basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, const char*);
template<class Tr>
        basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, const char*);
template<class Tr>
        basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, const signed char*);
template<class Tr>
        basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, const unsigned char*);
```

Операторы вывода для строк представлены в <*string*>, см. §20.3.15.

Функция *operator*<< () возвращает ссылку на *ostream*, для которого она вызывалась, чтобы к ней можно было применить другой *operator*<< (). Например:

```
cerr << "x= " << x;
```

где *x* относится к типу *int*, будет интерпретировано так:

```
operator << (cerr, "x= ").operator << (x);
```

В частности, это приводит к тому, что когда несколько сообщений выводится в одной инструкции, они выведутся в том порядке, как и предполагалось — слева направо. Например:

```
void val (char c)
{
        cout<<"int (' " << c << " ') = " << int (c) << '\n';
}

int main ()
{
        val ('A');
        val ('Z');
}
```

Для реализации, пользующейся символами ASCII, эта программа выведет:

```
int ('A') = 65
int ('Z') = 90
```

Отметим, что символьный литерал имеет тип *char* (§ 4.3.1), так что *cout*<<'*Z*' выведет символ *Z*, а не численное значение *90*.

Значение типа *bool* по умолчанию выведется как *0* или *1*. Если вам это не нравится, вы можете установить флаг форматирования *boolalpha* из <*iomanip*> (§ 21.4.6.2) и получить *true* или *false*. Например:

```
int main ()
{
        cout << true << ' ' << false << '\n';
        cout << boolalpha;                      // использовать для true и false
                                                // символьное представление
        cout << true << ' ' << false << '\n';
}
```

В результате выведется:

```
1 0
true false
```

Точнее, ***boolalpha*** гарантирует, что вы получите зависящее от национальных традиций представление значений типа ***bool***. Установив мои (датские) национальные особенности (§ 21.7), я получу:

```
1 0
sandt falsk
```

Форматирование чисел с плавающей точкой, вывод целых чисел в разной системе счисления и т. д. рассматриваются в § 21.4.

Функция *ostream::operator<<* (*const void**) выводит значение указателя в виде, соответствующем архитектуре машины. Например:

```
int main ()
{
    int* p = new int;
    cout << "локальная память " << &p << ", свободная память " << p << '\n';
}
```

выведет в моей машине

```
локальная память 0x7fffead0, свободная память 0x500c
```

Другие системы обрабатывают значение указателей по-другому.

21.2.3. Вывод типов, определяемых пользователем

Рассмотрим определяемый пользователем тип ***complex*** (§ 11.3):

```
class complex {
public:
    double real () const { return re; }
    double imag () const { return im; }
    // ...
};
```

Оператор << для нового типа ***complex*** можно определить так:

```
ostream& operator<< (ostream& s, complex z)
{
    return s << '(' << z.real () << ',' << z.imag () << ')';
}
```

После этого таким оператором << можно пользоваться точно так же, как оператором << для встроенных типов. Например:

```
int main ()
{
    complex x (1, 2);
    cout << "x = " << x << '\n';
}
```

выведет

```
x = (1, 2)
```

Определение операции вывода для типа, определяемого пользователем, не требует изменений в объявлении класса *ostream*. И это хорошо, поскольку *ostream* определен в <*ostream*>, который пользователь не может и не должен изменять. Запрет на добавления к *ostream* также обеспечивает защиту от случайной порчи его структур данных и позволяет изменять реализацию *ostream* без влияния на пользовательские программы.

21.2.3.1. Виртуальные функции вывода

Члены *ostream* не виртуальны. Операции вывода, которые может добавлять пользователь, не являются членами, поэтому они тоже не могут быть виртуальными. Одна из причин этого — стремление приблизиться к оптимальному быстродействию для простых операций, таких как помещение символа в буфер. Это место, где быстродействие является критическим и встраивание необходимо. Виртуальные функции используются только для достижения гибкости операций, имеющих дело с переполнением буфера сверху и снизу (§ 21.6.4).

Тем не менее, иногда программист хочет вывести объект, для которого известен только базовый класс. Поскольку точный тип не известен, правильность вывода не может быть достигнута просто путем определения оператора << для каждого нового типа. Вместо этого в абстрактном базовом классе можно ввести виртуальную функцию вывода:

```
class My_base {
public:
        // ...
        virtual ostream& put (ostream& s) const = 0;    // запись *this в s
};

ostream& operator<< (ostream& s, const My_base& r)
{
        return r.put (s);                               // использует подходящую функцию put()
}
```

То есть *put* () является виртуальной функцией, которая гарантирует использование правильной операции вывода <<.

С учетом этого мы можем написать:

```
class Sometype : public My_base {
public:
        // ...
        // фактическая функция вывода переопределяет My_base::put()
        ostream& put (ostream& s) const;
};

void f(const My_base& r, Sometype& s)          // использует заново определенный <<
{
        cout << r << s;
}
```

Таким образом виртуальная функция *put* () интегрируется в окружение, предоставленное *ostream* и << . Этот прием универсально полезен для введения операций, дей-

ствующих подобно виртуальным функциям, но с выбором во время выполнения, основанным на втором аргументе операции.

21.3. Ввод

Ввод обрабатывается почти также, как и вывод. Есть класс ***istream***, обеспечивающий оператор ввода >> («прочесть из») для небольшого набора стандартных типов. Функция ***operator>>*** () может быть определена пользователем во вводимых им типах.

21.3.1. Потоки ввода

По аналогии с ***basic_ostream*** (§ 21.2.1) ***basic_istream*** определен в <***istream***> (который содержит относящуюся ко вводу часть <***iostream***>) следующим образом:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_istream : virtual public basic_ios<Ch, Tr> {
public:
        virtual ~basic_istream ();

        // ...
};
```

Базовый класс ***basic_ios*** описан в § 21.2.1.

В <***istream***> вводятся два стандартных потока ввода: ***cin*** и ***wcin***:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

istream cin;            // стандартный поток ввода символов char
wistream wcin;          // стандартный поток ввода символов wchar_t
```

Поток ***cin*** считывает символы из того же источника, что и ***stdin*** в C (§ 21.8).

21.3.2. Ввод встроенных типов

Класс ***istream*** вводит оператор >> для типов, определяемых пользователем:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
        // ...
        // форматированный ввод:

        basic_istream& operator>> (short& n);               // прочесть в n
        basic_istream& operator>> (int& n);
        basic_istream& operator>> (long& n);

        basic_istream& operator>> (unsigned short& u);      // прочесть в u
        basic_istream& operator>> (unsigned int& u);
        basic_istream& operator>> (unsigned long& u);

        basic_istream& operator>> (float& f);               // прочесть в f
        basic_istream& operator>> (double& f);
        basic_istream& operator>> (long double& f);
```

```
        basic_istream& operator>> (bool& b);        // прочесть в b
        basic_istream& operator>> (void*& p);       // прочесть значение
                                                    // указателя в p

        // ...
};
```

Функции ввода ***operator***>> определены в таком стиле:

```
// T — тип, для которого объявляется istream::operator>>
istream& istream::operator>> (T& tvar)
{
        // пропустить символы-разделители, после чего каким-либо образом
        // прочесть переменную типа T в 'tvar'
        return *this;
}
```

Поскольку оператор >> пропускает символы-разделители, вы можете считать последовательность целых чисел, разделенных символами-разделителями, в вектор:

```
// заполняет v, возвращает число считанных чисел
int read_ints (vector<int>& v)
{
        int i = 0;
        while (i<v.size () && cin>>v[i]) i++;
        return i;
}
```

Нецелые числа на входе приведут к тому, что ввод не выполнится, и цикл ввода закончится. Например, ввод

```
1 2 3 4 5.6 7 8.
```

приведет к тому, что ***read_ints*** () считает пять символов:

```
1 2 3 4 5
```

и оставит точку как символ для следующего считывания. Символы-разделители определяются, как стандартные символы-разделители в C (пробел, табуляция, новая строка, новая страница и возврат каретки) вызовом функции ***isspace*** (), определенной в <*cctype*> (§ 20.4.2).

Наиболее распространенная ошибка при использовании потоков ***istream*** — не заметить, что введено не то, что предполагалось, поскольку на входе был не тот формат. Нужно или проверять состояние потока ввода (§ 21.3.3) перед использованием предположительно считанных данных, или задействовать исключения (§ 21.7).

Формат ввода учитывает национальные особенности (§ 21.7). По умолчанию значения ***true*** и ***false*** для типа ***bool*** представляются соответственно как ***1*** и ***0***. Целые числа и числа с плавающей точкой должны быть в том виде, как принято в программах на C++. Заданием ***basefield*** (§ 21.4.2), можно прочесть ***0123*** как восьмеричное число с десятичным значением ***83***, а ***0xff*** как шестнадцетеричное число с десятичным значением ***255***. Формат для считывания указателей полностью зависит от реализации (посмотрите, как это делается в вашей реализации).

Удивительно, но нет функции-члена >> для чтения символа. Причина здесь в том, что >> для символов можно реализовать при помощи операции ввода символа ***get*** ()

(§ 21.3.4), так что ей нет необходимости быть членом. Из потока мы можем считать символ в символьный тип, ассоциированный с данным потоком. Если это символьный тип ***char***, мы можем также считать данные в ***signed char*** и ***unsigned char***:

```
template<class Ch, class Tr>
    basic_istream<Ch, Tr>& operator>> (basic_istream<Ch, Tr>&, Ch&);

template<class Tr>
    basic_istream<char, Tr>& operator>> (basic_istream<char, Tr>&, unsigned char&);

template<class Tr>
    basic_istream<char, Tr>& operator>> (basic_istream<char, Tr>&, signed char&);
```

С точки зрения пользователя не важно, является ли операция >> членом или нет.

Как и остальные операторы >>, эти функции пропускают символы-разделители. Например:

```
void f()
{
    char c;
    cin >> c;
    //...
}
```

Таким образом в *c* будет помещен первый символ не-разделитель из ***cin***.

Кроме того, мы можем считывать в массив символов:

```
template<class Ch, class Tr>
    basic_istream<char, Tr>& operator>> (basic_istream<Ch, Tr>&, Ch*);
template<class Tr>
    basic_istream<char, Tr>& operator>> (basic_istream<char, Tr>&, unsigned char*);
template<class Tr>
    basic_istream<char, Tr>& operator>> (basic_istream<char, Tr>&, signed char*);
```

Эти операции сначала пропускают символы-разделители. Потом они считывают в массив-операнд, пока не встретят символ-разделитель или конец файла. И наконец, они вставляют признак конца строки — ***0***. Ясно, это открывает широкие возможности для переполнения, так что обычно лучше считывать в строку ***string*** (§ 20.3.5). Однако вы можете определить максимальное число символов, считываемых оператором >>: ***is.width*** (*n*) определяет, что следующее >> над ***is*** считает в массив максимум *n*–*1* символов. Например:

```
void g()
{
    char v[4];
    cin.width(4);
    cin >> v;
    cout << "v= " << v << endl;
}
```

Таким образом в *v* считается не более трех символов, и добавится признак конца строки ***0***.

Установка ***width*** () для ***istream*** влияет только на непосредственно следующую за ней операцию >> считывания в массив и не затрагивает считывание в переменные других типов.

21.3.3. Состояние потока

Каждый поток (*istream* или *ostream*) имеет связанное с ним *состояние*. Установкой и соответствующей проверкой этого состояния вылавливаются ошибки и нестандартные условия.

Состояние потока вводится в *basic_ios*, базовом классе класса *basic_stream*, в <*ios*>:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
        //...

        bool good() const;      // следующая операция может выполниться
        bool eof() const;       // виден конец ввода
        bool fail() const;      // следующая операция не выполнится
        bool bad() const;       // поток испорчен

        iostate rdstate() const;                // получение флагов состояния ввода/вывода
        void clear(iostate f = goodbit);        // установка флагов состояния ввода/вывода
        void setstate(iostate f)                // добавление f к флагам состояния
                { clear(rdstate() | f); }

        operator void*() const;                 // не ноль, если !fail()
        bool operator!() const { return fail(); }

        //...
};
```

Если установилось состояние *good* (), значит предыдущие операции успешно выполнились. Если установилось состояние *good* (), может выполниться следующий ввод, в противном случае он не выполнится. Применение операции ввода к потоку, находящемуся не в состоянии *good* (), — это нулевая операция по отношению к считываемой переменной. Если мы пытаемся прочитать в переменную *v*, и операция не выполняется, значение *v* должно остаться неизменным (оно не изменяется, если *v* — это переменная одного из тех типов, которые обрабатываются функциями-членами классов *istream* или *ostream*). Различие между состояниями *fail* () и *bad* () очень незначительно. Когда поток в состоянии *fail* (), но одновременно не в *bad* (), считается, что он не испорчен (но символы могут быть потеряны). Когда он в состоянии *bad* (), ни в чем нельзя быть уверенным.

Состояние потока представляется набором флагов. Как и большинство констант, выражающих поведение потока, эти флаги определены в *ios_base* — базовом классе класса *basic_ios*:

```
class ios_base {
public:
        //...

        typedef implementation_defined2 iostate;
        static const iostate    badbit,     // поток испорчен
                                eofbit,     // виден конец файла
                                failbit,    // следующая операция не выполнится
                                goodbit;    // goodbit==0

        //...

};
```

Флагами состояния ввода/вывода можно непосредственно манипулировать. Например:

```
void f()
{
        ios_base::iostate s = cin.rdstate();        // возвращает набор битов
                                                    // состояния ввода/вывода
        if (s & ios_base::badbit) {
                // возможно, символы в cin потеряны
        }
        //...
        cin.setstate(ios_base::failbit);
        //...
}
```

Когда поток используется как условие, состояние потока проверяется функциями ***operator void****() или ***operator***!(). Проверки считаются успешными, только если состояние потока есть !***fail***() и ***fail***() соответственно. Например, универсальную функцию копирования можно написать так:

```
template<class T> void iocopy(istream& is, ostream& os)
{
        T buf;
        while (is>>buf) os << buf << '\n';
}
```

Операция ***is>>buf*** возвращает ссылку на ***is***, которая проверяется вызовом ***is::operator void****(). Например:

```
void f(istream& i1, itream& i2, istream& i3, istream& i4)
{
        iocopy<complex>(i1, cout);      // копирование комплексных чисел
        iocopy<double>(i2, cout);       // копирование чисел типа double
        iocopy<char>(i3, cout);         // копирование char
        iocopy<string>(i4, cout);       // копирование слов, разделенных
                                        // символами-разделителями
}
```

21.3.4. Ввод символов

Оператор >> предназначен также для форматированного ввода, то есть для считывания объектов ожидаемого типа и в ожидаемом формате. Там, где это нежелательно, и мы хотим считывать символы как символы, а потом проверять их, мы пользуемся функциями ***get***():

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
        //...
        // неформатированный ввод:
        streamsize gcount() const;      // число символов, считанных последней get()
        int_type get();                 // считывание одного Ch
        basic_istream& get(Ch& c);      // считывание одного Ch в c
```

```
        basic_istream& get (Ch* p, streamsize n );   // завершающий символ — новая строка
        basic_istream& get (Ch* p, streamsize n, Ch term);

        basic_istream& getline (Ch* p, streamsize n ); // завершающий символ — новая строка
        basic_istream& get (Ch* p, streamsize n, Ch term);

        basic_istream& ignore (streamsize n = 1, int_type t = Tr::eof());
        basic_istream& read (Ch* p, streamsize n);          // чтение не более n символов

        // ...
};
```

Кроме того, в <*string*> предлагается функция *getline* () для стандартных строк (§ 20.3.15).

Функции *get* () и *getline* () обращаются с символами-разделителями точно так же, как и с другими символами. Они предназначены для операций ввода, где заранее не предполагается, что означают введенные символы.

Функция *istream::get* (*char*&) считывает один символ в свой аргумент. Например, программу посимвольного копирования ввода можно написать так:

```
int main ()
{
        char c;
        while (cin.get (c)) cout.put (c);
}
```

Функция с тремя аргументами *s.get* (*p, n, term*) считывает не более *n–1* символов в *p*[*0*]..*p*[*n–2*]. Вызов *get* () всегда помещает *0* после размещенных в буфере символов, так что *p* должен указывать на массив, содержащий не менее чем *n* символов. Третий аргумент, *term*, определяет завершающий символ. Типичное использование функции *get* () с тремя аргументами — считывание «строки» в буфер фиксированного размера для дальнейшего анализа. Например:

```
void f ()
{
        char buf[100];
        cin >> buf;                  // не очень хорошо: когда-нибудь переполнится
        cin.get (buf, 100, '\n');    // безопасно
        // ...
}
```

Если функции *get* () или *getline* () не удается считать и удалить из потока хотя бы один символ, вызывается *setstate* (*failbit*), так что последующее чтение из потока завершится неудачей (или генерацией исключения, § 21.3.6). Если встречен завершающий символ, он остается первым несчитанным символом в потоке. Никогда не обращайтесь к *get* () два раза подряд, не удалив завершающий символ. Например:

```
void subtle_error ()
{
        char buf[256];

        while (cin) {
                cin.get (buf, 256);      // считывание строки
                cout << buf;             // вывод строки. Ошибка: забыли удалить '\n' из cin —
                                         // следующий вызов get ( ) будет неудачен
        }
}
```

Этот пример показывает, что лучше использовать ***getline*** (), а не ***get*** (). Функция ***getline*** () ведет себя так же, как и соответствующая ***get*** (), но удаляет из ***istream*** встреченный завершающий символ. Например:

```
void f()
{
        char word[MAX_WORD] [MAX_LINE];        // MAX_WORD массив из MAX_LINE
                                               // символов в каждом
        int i=0;
        while (cin.getline (word[i++], MAX_LINE, '\n') && i<MAX_WORD);
        //...
}
```

Когда эффективность не играет большой роли, лучше считывать в строку ***string*** (§ 3.6, § 20.3.15). Тогда не возникнет обычных проблем с распределением памяти и переполнением. Однако функции ***get*** (), ***getline*** () и ***read*** () нужны, чтобы реализовать такие возможности высокого уровня. Относительно запутанный интерфейс — это цена, которую мы платим за скорость, за избавление от необходимости повторно просматривать вход, выясняя, чем закончилась операция ввода, за возможность надежного ограничения числа считываемых символов и т. д.

Обращение к ***read*** (***p***, ***n***) считывает максимум ***n*** символов в ***p***[***0***]..***p***[***n–1***]. Функция ***read*** () не полагается на завершающий символ и не ставит в конец полученной строки ***0***. Следовательно, она действительно может считать ***n*** символов (а не ***n–1***). Другими словами, она просто считывает символы и не пытается превратить считанное в С-строку.

Функция ***ignore*** () считывает символы так же, как и ***read*** (), но нигде не хранит их. Как и ***read*** (), она на самом деле считывает ***n*** символов (а не ***n–1***). По умолчанию число считываемых ею символов равно 1, поэтому вызов ***ignore*** () без аргументов означает «выбросить следующий символ». Как и ***getline*** (), она принимает завершающий символ, если он встретится, и в этом случае удаляет его из потока ввода. Отметим, что по умолчанию завершающим символом для ***ignore*** () является конец файла.

Для всех этих функций не очевидно, что именно должно завершать чтение — и даже трудно запомнить, какой функции соответствует тот или иной завершающий символ. Однако мы всегда проверяем, не достигли ли конца файла (§ 21.3.3). Также введена функция ***gcount*** (), возвращающая число символов, считанных из потока функцией неформатированного ввода в последний раз. Например:

```
void read_a_line (int max)
{
        //...
        if (cin.fail ()) {                  // плохой формат ввода
                cin.clear ();               // очистка флагов ввода (§ 21.3.3)
                cin.ignore (max, ';');      // пропуск до точки с запятой

                if (!cin) {
                        // конец потока
                }
                else if (cin.gcount () == max) {
                        // проблема: прочитали максимальное число символов
                }
                else {
                        // встретили и выбросили точку с запятой
```

```
            }
        }
        //...
    }
```

К несчастью, если максимальное число символов прочитано, то не существует способа определить, был ли достигнут терминатор (в качестве последнего символа).

Функция ***get*** () без аргументов — это версия из <***iostream***>, аналогичная ***getchar*** () в <***cstdio***> (§ 21.8). Она просто считывает символ и возвращает его численное значение. Таким образом, она освобождает нас от предположений, какой тип символа использовался. Если нет никакого символа, ***get*** () возвращает соответствующий маркер «конца файла» (то есть ***traits_type***::***eof*** () этого потока) и устанавливает поток в состояние ***eof*** (§ 21.3.3). Например:

```
void f(unsigned char* p)
{
    int i;
    while ( (i = cin.get ()) && i!=EOF) {
        *p++=i;
        //...
    }
}
```

EOF — это значение ***eof*** () из обычных ***chair_traits*** для ***char***. ***EOF*** представлен в <***iostream***>. Таким образом, этот цикл можно было бы написать как ***read*** (***p, MAX_INT***), но мы специально написали явный цикл, потому что хотели посмотреть на каждый символ при его поступлении. Как уже было сказано, главная сила языка C — в его способности считывать символы и решать, что с ними ничего не надо делать — причем выполнять это быстро. Это действительно важное достоинство, которое нельзя недооценивать, и цель C++ — не утратить его.

Стандартный заголовочный файл <***cctype***> определяет несколько функций, которые могут пригодиться при обработке ввода (§ 20.4.2). Например, функция ***eatwhite*** (), считывающая из входа символы-разделители, может быть определена следующим образом:

```
istream& eatwhite (istream& is)
{
    char c;
    while (is.get (c)) {
        if (!isspase (c)) {          // c — не символ-разделитель?
            is.putback (c);          // поместить с обратно в буфер ввода
            break;
        }
    }
    return is;
}
```

Вызов ***is.putback*** () превращает ***c*** в следующий символ из потока ***is*** (§ 21.6.4).

21.3.5. Ввод типов, определяемых пользователем

Операции ввода для типов, определяемых пользователем, можно ввести точно так же, как это делалось для операций вывода. Однако для операций ввода важно, чтобы второй аргумент не был константной ссылкой. Например:

```
istream& operator>> (istream& s, complex& a)
/*
        форматы ввода для complex (f означает число с плавающей точкой):
                f
                (f)
                (f,f)
*/
{
        double re = 0, im = 0;
        char c = 0;

        s>>c;
        if (c == '(') {
                s >> re >> c;
                if (c == ',') s >> im >> c;
                if (c != ')') s.clear (ios_base::failbit);        // установка состояния
        }
        else {
                s.putback (c);

                s >> re;
        }
        if (s) a = complex (re, im);
        return s;
};
```

Несмотря на краткость кода обработки ошибок, такая программа в действительности может выловить большинство видов ошибок. Локальная переменная *c* инициализируется для того, чтобы ее значение после неудавшейся первой операции >> случайно не оказалось '('. Заключительная проверка состояния потока гарантирует, что значение аргумента *a* изменилось, только если все прошло хорошо. Если найдена ошибка форматирования, состояние потока устанавливается в ***failbit***. Состояние не устанавливается в ***badbit*** потому, что поток сам по себе не разрушен. Пользователь может переустановить поток (используя ***clear*** ()) и, возможно, обойти проблему и извлечь из потока полезные данные.

Операция для установки состояния потока называется ***clear*** () (очистить), поскольку ее самое распространенное назначение — сброс состояния потока в ***good*** (); ***ios_base::goodbit*** — это значение аргумента по умолчанию для ***clear*** () (§ 21.3.3).

21.3.6. Исключения

Проверять наличие ошибок после каждой операции ввода/вывода не слишком удобно, поэтому самая распространенная причина ошибок кроется в том, что этого не делают там, где это важно. В частности, как правило, не проверяются операции вывода, в то время как сбои случаются и при их выполнении.

Единственная функция, которая непосредственно изменяет состояние потока, — это ***clear*** (). Поэтому очевидный способ заметить изменение состояния — это попросить ***clear*** () сгенерировать исключение. Именно этим и занимается член класса ***basic_ios*** функция ***exceptions*** ():

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
        //...
```

```
        class failure;                          // класс исключений (см. § 14.10)

        iostate exceptions () const;            // получает состояние исключения
        void exceptions (iostate except);       // устанавливает состояние исключения

        //...
};
```

Например,

```
cout.exceptions (ios_base::badbit | ios_base::eofbit | ios_base::eofbit);
```

требует, чтобы ***clear*** () сгенерировала исключение ***ios_base::failure***, если ***cout*** придет в состояние ***bad***, ***fail*** или ***eof*** — другими словами, если какая-нибудь из операций над ***cout*** не выполнилась безупречно. Если необходимо, мы можем проверить ***cout*** для точного определения, что произошло. Подобным же образом

```
cin.exceptions (ios_base::badbit | ios_base::failbit);
```

позволит нам перехватить не слишком экзотический случай, когда ввод производится не в ожидаемом нами формате и операция ввода не возвращает значения из потока.

Вызов ***exceptions*** () без аргументов возвращает набор флагов состояния ввода/вывода, которое инициировало исключение. Например:

```
void print_exception (ios_base& ios)
{
        ios_base::iostate s = ios.exceptions ();
        if (s&ios_base::badbit) cout << "генерирует для bad";
        if (s&ios_base::failbit) cout << "генерирует для fail";
        if (s&ios_base::eofbit) cout << "генерирует для eof";
        if (s == 0) cout << "не возбуждает";
}
```

Главное назначение исключений ввода/вывода — вылавливать маловероятные, а значит, часто упускаемые ошибки. Вторая их задача — контролировать ввод/вывод. Например:

```
void readints (vector<int>& s)                  // не мой любимый стиль!
{
        ios.base::iostate old_state = cin.exceptions ();  // сохраняет состояние исключения
        cin.exceptions (ios_base::eofbit);              // возбуждает для eof
        for (;;)
                try {
                        int i;
                        cin >> i;
                        s.push_back (i);
                }
                catch (ios_base::failure) {     // правильно: дошли до конца файла
                        break;
                }
        cin.exceptions (old_state);             // восстанавливаем состояние исключения
};
```

Вопрос, который задается при использовании исключений — «Ошибка ли это?» или «Действительно ли это исключительное событие?». Обычно я считаю, что ответ на оба этих вопроса — «нет». И поэтому я предпочитаю работать напрямую с состоянием потока. То, что можно обработать локальными управляющими структурами внутри функции, редко улучшается за счет использования исключений.

21.3.7. Связывание потоков

Функция *tie* () из *basic_ios* используется для того, чтобы устанавливать и разрывать связи между *istream* и *ostream*:

```
template<class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
        //...
        basic_ostream<Ch, Tr>* tie () const;   // получаем указатель на связанный поток
        basic_ostream<Ch, Tr>* tie (basic_ostream<Ch, Tr>* s);   // привязываем *this к s
        //...
};
```

Рассмотрим:

```
void get_passwd ()
{
        string s;
        cout << "Пароль: ";
        cin >> s;
        //...
}
```

Как нам гарантировать, что **Пароль**: появится на экране прежде, чем выполнится операция считывания? Вывод в ***cout*** буферизуется, так что если ***cin*** и ***cout*** независимы, то **Пароль**: не появится на экране, пока не заполнится буфер вывода. Решение этой задачи заключается в том, что ***cout*** связывается с ***cin*** операцией ***cin.tie*** (&***cout***).

Если ***ostream*** связан с ***istream***, ***ostream*** очищается каждый раз, когда операция ввода над ***istream*** приводит к переполнению; то есть каждый раз, когда нужен новый символ из конечного источника ввода, чтобы завершить операцию ввода. Таким образом,

```
cout << "Пароль: ";
cin >> s;
```

равносильно следующему:

```
cout << "Пароль: ";
cout.flush ();
cin >> s;
```

Поток может иметь самое большое один ***ostream***, связанный с ним в конкретный момент времени. Обращение к ***s.tie*** (***0***) разрывает связь между потоком ***s*** и потоком, с которым он был связан (если был). Как и большинство функций с потоками, устанавливающих значение, функция ***tie*** (***s***) возвращает прежнее значение; то есть она возвращает предыдущий связанный поток или ***0***. При вызове без аргумента ***tie*** () возвращает текущее значение, не изменяя его.

Из стандартных потоков ***cout*** связывается с ***cin***, а ***wcout*** с ***wcin***. Потоки ***cerr*** не нужно связывать, поскольку они не буферизуются, в то время как потоки ***clog*** не предполагают вмешательства пользователя.

21.3.8. Часовые

Когда я писал операторы << или >> для ***complex***, я не беспокоился о связанных потоках (§ 21.3.7) или о том, не приведет ли изменение состояния потока к исключению (§ 21.3.6). Я считал — и правильно — что обеспеченные библиотекой функции сами позаботятся об этом за меня. Но как? Таких функций несколько десятков. Если бы нам пришлось писать замысловатый код для обработки связанных потоков, национальных особенностей (§ 21.7), исключений и т. д. в каждой из них, код мог бы стать довольно запутанным.

Взятый на вооружение подход заключается в обеспечении общего кода через класс ***sentry*** (часовой). Код, который должен выполняться первым («префикс-код») — например, очистка связанного потока — вводится в виде конструктора ***sentry***. Код, который должен выполняться последним («суффикс-код») — например, генерация исключений, вызванных изменением состояния, — обеспечивается через деструктор ***sentry***:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
        //...
        class sentry;
        //...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream<Ch, Tr>::sentry {
public:
        explicit sentry (basic_ostream<Ch, Tr>& s);
        ~sentry ();
        operator bool ();

        //...
};
```

Таким образом, общий код «выносится за скобки», и отдельные функции можно написать следующим образом:

```
template<class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch, Tr>& basic_ostream<Ch, Tr>::operator<< (int i)
{
        sentry s (*this);

        if (!s) {       // проверка, все ли готово для начала вывода
                setstate (failbit);
                return *this;
        }

        // вывод числа

        return *this;
}
```

Этот прием, использующий конструкторы и деструкторы для выполнения инициирующего и заключительного кода для класса, может пригодиться во многих случаях.

Естественно, ***basic_istream*** имеет похожий класс членов «часовых».

21.4. Форматирование

Примеры, приведенные в § 21.2, представляют собой то, что обычно называют *неформатированным выводом*. То есть объект превращался в последовательность символов в соответствии с некоторыми правилами по умолчанию. Часто программисту нужно более детализированное управление. Например, необходимо управлять объемом памяти, выделенным под операции вывода, и форматом вывода чисел. Подобным образом нужно в явном виде управлять и некоторыми аспектами ввода.

Управление форматированием ввода/вывода производится классом ***basic_ios*** и его базовым классом ***ios_base***. Например, класс ***basic_ios*** содержит информацию о системе счисления (восьмеричная, десятичная или шестнадцатеричная) при записи и чтении целых чисел, о точности записываемых и считываемых чисел с плавающей точкой и т. п. Он также содержит функции для установки и проверки этих управляющих переменных для каждого потока.

Класс ***basic_ios*** — это базовый класс для ***basic_istream*** и ***basic_ostream***, так что управление форматом осуществляется «по-поточно» (для каждого потока).

21.4.1. Состояние формата

Форматированием ввода/вывода управляют набор флагов и целочисленные значения в ***ios_base*** потока:

```
class ios_base {
public:
    // ...
    // имена флагов формата:

    typedef implementation_defined1 fmtflags;
    static const fmtflags
        skipws,         // пропуск символов-разделителей на входе
        left,           // выравнивание поля: отступ после значения
        right,          // отступ перед значением
        internal,       // отступ между знаком и значением
        boolalpha,      // вывод символического представления true и false

        dec,            // система счисления для целых чисел: десятичная
        hex,            // шестнадцатеричная
        oct,            // восьмеричная

        scientific,     // обозначение числа с плавающей точкой: d.dddddEddd
        fixed,          // dddd.dd

        showbase,       // на выходе префикс 0 перед восьмеричными
                        // числами и 0x перед шестнадцатеричными
        showpoint,      // выводить незначащие нули
        showpos,        // явный '+' перед положительными целыми
        uppercase,      // 'E', 'X' вместо 'e', 'x'

        adjustfield,    // относится к выравниванию полей (§ 21.4.5)
        basefield,      // относится к системе счисления (§ 21.4.2)
        floatfield,     // относится к выводу чисел с плавающей точкой (§ 21.4.3)

        unitbuf;        // очистка буфера после каждой операции вывода
```

```
        fmtflags flags () const;                    // для чтения флагов
        fmtflags flags (fmtflags f);                // для установки флагов

    fmtflags setf(fmtflags f) { return flags (flags () | f); }   // добавление флага
    // сброс и установка флагов mask
    fmtflags setf(fmtflags f, fmtflags mask) { return flags ((flags ()&~mask) | (f&mask)); }
    void unsetf (fmtflags mask){ flags (flags ()&~mask); }   // сброс флагов

    // ...
};
```

Значения флагов определяются при реализации. Используйте исключительно символические имена, а не специфические численные значения, даже если на сегодня в вашей реализации они вам известны.

Определение интерфейса в виде набора флагов и предоставление операций для их установки и сброса проверены временем, хотя это и выглядит несколько старомодным приемом. Главное достоинство этого метода в том, что пользователь может объединять опции. Например:

```
const ios_base::fmtflags my_opt = ios_base::left | ios_base::oct | ios_base::fixed;
```

Это позволяет нам при необходимости произвольно устанавливать опции. Например:

```
void your_function (ios_base::fmtflags opt)
{
    ios_base::fmtflags old_options = cout.flags (opt);   // сохранение старых
                                                         // опций и установка новых
    // ...
    cout.flags (old_options);                            // восстановление опций
}

void my_function ()
{
    your_function (my_opt);
    // ...
}
```

Функция *flags* () возвращает прежний набор опций.

Имея возможность читать и задавать опции, мы можем устанавливать отдельные флаги. Например:

```
myostream.flags (myostream.flags () | ios_base::showpos);
```

В результате *myostream* будет выдавать явный + перед положительными числами, а остальные опции останутся без изменений. Старые опции считываются, и в набор при помощи логического ИЛИ добавляется *showpos*. Функция *setf* () именно это и делает, так что пример можно переписать следующим образом:

```
myostream.setf(ios_base::showpos);
```

После установки флаг сохраняет свое состояние, пока не будет сброшен.

Управление опциями ввода/вывода путем явной установки и сброса флагов грубо и может привести к ошибкам. Для простых случаев более аккуратный интерфейс обеспечивают манипуляторы (§ 21.4.6). Пользоваться флагами для управления состоянием потока лучше лишь с целью изучения приемов реализации, а не при проектировании интерфейса.

21.4.1.1. Копирование состояния формата

Полное состояние формата потока можно скопировать функцией ***copyfmt*** ():

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
        //...
        basic_ios& copyfmt (const basic_ios& f);
        //...
};
```

Буфер потока (§ 21.6) и состояние буфера функцией ***copyfmt*** () не копируются. Однако все остальное состояние копируется, в том числе требуемые исключения (§ 21.3.6) и все пользовательские добавления к состоянию (§ 21.7.1).

21.4.2. Вывод целых чисел

Прием с добавлением новых опций логическим ИЛИ в функциях ***flags*** () и ***setf*** () работает только тогда, когда какой-то характеристикой управляет один бит. Это не годится для таких опций, как система счисления при выводе целых чисел и вид числа с плавающей точкой. Для подобных опций значение, определяющее вид вывода, не обязательно выражается одним битом или набором независящих друг от друга одиночных битов.

Решение, примененное в <***iostream***>, заключается в том, чтобы предоставить версию ***setf*** () со вторым «псевдоаргументом», который показывает, какой вид опции мы хотим установить в добавок к новому значению. Например,

```
cout.setf(ios_base::oct, ios_base::basefield);      // восьмеричные
cout.setf(ios_base::dec, ios_base::basefield);      // десятичные
cout.setf(ios_base::hex, ios_base::basefield);      // шестнадцатеричные
```

устанавливает систему счисления без всяких побочных эффектов для остальных частей состояния потока. Однажды установленная, система счисления остается неизменной, пока не будет переустановлена. Например,

```
cout << 1234 << ' ' << 1234 << ' ';                 // по умолчанию: десятичная
cout.setf(ios_base::oct, ios_base::basefield);      // восьмеричная
cout << 1234 << ' ' << 1234 << ' ';

cout.setf(ios_base::hex, ios_base::basefield);      // шестнадцатеричная
cout << 1234 << ' ' << 1234 << ' ';
```

выведет ***1234 1234 2322 2322 4d2 4d2.***

Если нам нужно сказать, какая система счисления использовалась для каждого числа, мы можем установить ***showbase***, добавив

```
cout.setf(ios_base::showbase);
```

Тогда выведется ***1234 1234 02322 02322 0x4d2 0x4d2.*** Более изящный способ для определения системы счисления выводимых целых чисел обеспечивают стандартные манипуляторы (§ 21.4.6.2).

21.4.3. Вывод чисел с плавающей точкой

Вывод чисел с плавающей точкой определяется *форматом* и *точностью*:

- *Универсальный* формат дает реализации самой выбирать формат представления числа в том виде, который наилучшим образом представит число в доступном пространстве. Максимальное число цифр определяется точностью. Это соответствует %*g* в *printf*() (§ 21.8).
- *Научный* формат представляет число десятичной дробью с одной цифрой перед точкой и показателем степени. Это соответствует %*e* в *printf*().
- *Фиксированный* формат представляет число как целую часть с дробной частью, отделенной точкой. Точность определяет максимальное число цифр после точки. Это соответствует %*f* в *printf*().

Мы управляем выводом чисел с плавающей точкой при помощи функций манипулирования состоянием. В частности, мы можем установить способ вывода числа с плавающей точкой, не влияя на другие аспекты состояния потока. Например,

```
cout << "по умолчанию:\t" << 1234.56789 << '\n';

cout.setf(ios_base::scientific, ios_base::floatfield);   // научный формат
cout << "научный:\t" << 1234.56789 << '\n';

cout.setf(ios_base::fixed, ios_base::floatfield);        // формат с фиксированной точкой
cout << "фиксированный:\t" << 1234.56789 << '\n';

cout.setf(0, ios_base::floatfield);                      // восстановление формата по умолчанию
                                                         // (то есть универсального)
cout << "по умолчанию:\t" << 1234.56789 << '\n';
```

выведет

```
по умолчанию:    1234.57
научный:         1.2345678e+03
фиксированный:   1234.567890
по умолчанию:    1234.57
```

По умолчанию точность — 6 цифр (для всех форматов). Точностью управляют функции-члены класса *ios_base*:

```
class ios_base {
public:
        //...
        streamsize precision() const;          // возвращает точность
        streamsize precision(streamsize n);    // устанавливает точность
                                               // (и возвращает старую)
        //...
}
```

Вызов *precision*() влияет на все операции ввода/вывода чисел с плавающей точкой в поток и действует до следующего обращения к *precision*(). Таким образом,

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
```

выведет

```
1234.5679   1234.5679   123456
1235   1235   123456
```

Отметим, что числа с плавающей точкой округляются, а не просто урезаются, и что функция ***precision*** () не влияет на вывод целых чисел.

Флаг ***uppercase*** (§ 21.4.1) определяет, будет выводиться ***e*** или ***E*** для обозначения порядка в научном формате.

Более изящный способ для определения вида выводимых чисел с плавающей точкой предоставляют манипуляторы (§ 21.4.6.2).

21.4.4. Поля вывода

Часто мы хотим заполнить текстом определенное место в выходной строке. Скажем, мы хотим использовать ровно *n* символов и не меньше (а больше — только если текст не влезает). Чтобы это сделать, мы определим ширину поля и символ, используемый для его заполнения при необходимости:

```
class ios_base {
public:
        //...
        streamsize width () const;              // получить ширину поля
        streamsize width (streamsize wide);     // установить ширину поля
        //...
};

template<class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
        //...
        Ch fill () const;                       // получить символ-заполнитель
        Ch fill (Ch ch);                        // установить символ-заполнитель
        //...
};
```

Функция ***width*** () определяет минимальное число символов, которые выведутся следующей операцией вывода `<<` стандартной библиотеки для числа, логического значения, строки в стиле C, символа, указателя (§ 21.2.1), строки (§ 20.3.15) или ***bitset*** (§ 17.5.3.3). Например, в результате

```
cout.width (4);
cout << 12;
```

выведется *12* с двумя пробелами впереди.

Символ-заполнитель можно определить функцией ***fill*** (), например:

```
cout.width (4);
cout.fill ('#');
cout << "ab";
```

выдаст строку (##*ab*).

По умолчанию символом-заполнителем является пробел, а величина поля равна *0*, что означает «столько символов, сколько нужно». Размер поля по умолчанию можно восстановить следующим образом:

```
cout.width(0);          // «столько символов, сколько нужно»
```

Функция *width(n)* устанавливает минимальное число символов в *n*. Если строка или число длиннее, выведутся все их символы. Например, в результате

```
cout.width(4);
cout << "abcdef";
```

выведется *abcdef*, а не *abcd*. Обычно лучше получить правильный вывод, ужасный с виду, чем прекрасный внешне, но неверный (см. также § 21.10[21]).

Обращение к функции *width(n)* влияет только на непосредственно следующую за ней операцию вывода <<:

```
cout.width(4);
cout.fill('#');
cout << 12 << ':' << 13;
```

Данные операции выведут *##12:13*, а вовсе не *##12###:####13*, как можно было бы предположить, увидев в последовательности инструкций *width(4)*. Если бы *width()* влияла на все операции по выводу чисел и строк, нам пришлось бы явно указать *width()* практически для всех величин.

Стандартные манипуляторы (§ 21.4.6.2) предоставляют более изящный способ определения ширины поля вывода.

21.4.5. Выравнивание поля

Управлять выравниванием символов в поле можно, вызывая функции *setf()*:

```
cout.setf(ios_base::left, ios_base::adjustfield);      // влево
cout.setf(ios_base::right, ios_base::adjustfield);     // вправо
cout.setf(ios_base::internal, ios_base::adjustfield);  // внутреннее
```

Эти установки выравнивания внутри поля вывода определяются функцией *ios_base::width()* и не влияют на другие части состояния потока.

Выравнивание можно определить следующим образом:

```
cout.fill('#');

cout <<'(';
cout.width(4);
cout << -12 << "), (";

cout.width(4);
cout.setf(ios_base::left, ios_base::adjustfield);
cout << -12 << "), (";

cout.width(4);
cout.setf(ios_base::internal, ios_base::adjustfield);
cout << -12 << ")";
```

В результате выведется: (#−12), (−12#), (−#12). Внутреннее выравнивание располагает символы-заполнители между знаком и значением. Как показано, по умолчанию установлено выравнивание вправо. Что произойдет, если одновременно установлено более одного флага выравнивания, не определено.

21.4.6. Манипуляторы

Чтобы избавить программиста от необходимости иметь дело с состоянием потока посредством флагов, стандартная библиотека предоставляет набор функций для манипулирования этим состоянием. Ключевая идея заключается в том, чтобы вставлять между объектами (записываемыми или читаемыми) операцию, которая изменила бы состояние потока. Например, мы можем явно потребовать, чтобы выходной буфер очистился:

```
cout << x << flush << y << flush;
```

Здесь **cout.flush** () вызывается в соответствующее время. Это делается версией оператора <<, который принимает в качестве аргумента указатель на функцию и вызывает ее:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
        //..

        basic_ostream& operator<< (basic_ostream& (*f) (basic_ostream&)) { return f(*this); }
        basic_ostream& operator<< (ios_base& (*f) (ios_base&));
        basic_ostream& operator<< (basic_ios<Ch, Tr>& (*f) (basic_ios<Ch, Tr>&));

        //...
};
```

Чтобы это заработало, функция должна быть не-членом или статической функцией-членом правильного типа. В частности, **flush** () определяется следующим образом:

```
template<class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch, Tr>& flush (basic_ostream<Ch, Tr>& s)
{
        return s.flush ();          // вызов функции flush(), члена ostream
}
```

Это гарантирует, что

```
cout << flush;
```

разрешается как

```
cout.operator << (flush);
```

что вызывает

```
flush (cout);
```

что в свою очередь вызывает

```
cout.flush ();
```

Весь этот анализ происходит (во время компиляции) для того, чтобы было можно обращаться к **basic_ostream::flush** () при помощи записи **cout<<flush**.

Существует множество операций, которые нам бы хотелось выполнить непосредственно перед или после операции ввода/вывода. Например:

```
cout << x;
cout.flush();
cout << y;

cin.unsetf(ios_base::skipws);        // не пропускать символы-разделители
cin >> x;
```

Когда операции записываются как отдельные инструкции, логические связи между ними не очевидны. При потере логической связи код становится сложнее понимать. Введение манипуляторов позволяет вставлять такие операции, как ***flush*** () и ***noskipws*** () непосредственно в список операций ввода или вывода. Например:

```
cout << x << flush << y << flush;
cin >> unsetf(ios_base::skipws) >> x;
```

Отметим, что манипуляторы находятся в пространстве имен ***std***. Поэтому там, где ***std*** не является частью текущей области видимости, они должны быть явно квалифицированы:

```
std::cout << endl;          // ошибка: endl вне области видимости
std::cout << std::endl;     // правильно
```

Естественно, класс ***basic_istream***, так же, как и ***basic_ostream***, обеспечивает операторы >> для вызова манипуляторов:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    //...

    basic_istream& operator>> (basic_istream& (*pf) (basic_istream&));
    basic_istream& operator>> (basic_ios<Ch, Tr>& (*pf) (basic_ios<Ch, Tr>&));
    basic_istream& operator>> (ios_base& (*pf) (ios_base&));

    //...
};
```

21.4.6.1. Манипуляторы с аргументами

Могут пригодиться и манипуляторы с аргументами. Например, нам может захотеться написать:

```
cout << precision(4) << angle;
```

чтобы вывести значение переменной с плавающей точкой ***angle***, состоящее из четырех цифр.

Чтобы это сделать, ***setprecision*** должна вернуть объект, инициализированный числом ***4***, который при обращении к нему вызывает ***cout.precision(4)***. Таким манипулятором является объект-функция, который вызывается оператором <<, а не (). Точный тип этого объекта-функции определяется при реализации, но его можно определить следующим образом:

```
struct smanip {
    ios_base& (*f)(ios_base&, int);        // функция для вызова
    int i;

    smanip(ios_base& (*ff) (ios_base&, int), int ii) : f(ff), i(ii) { }
};
```

```
template<class Ch, class Tr>
ostream<Ch, Tr>& operator<< (ostream<Ch, Tr>& os, const smanip& m)
{
    return m.f(os, m.i);
}
```

Конструктор *smanip* хранит свои аргументы в *f* и *i*, а ***operator<<*** вызывает *f*(*i*). Теперь мы можем определить ***setprecision*** () следующим образом:

```
ios_base& set_precision (ios_base& s, int n)    // функция-помощник
{
    return s.setprecision (n);                  // вызов функции-члена
}

inline smanip setprecision (int n)
{
    return smanip (set_precision, n);           // создание объекта-функции
}
```

И теперь мы можем написать:

```
cout << setprecision (4) << angle;
```

В случае необходимости программист может определить новые манипуляторы в том же стиле, что и ***smanip*** (§ 21.10[22]). Это не требует внесения изменений в определения шаблонов стандартной библиотеки и таких классов, как ***basic_istream***, ***basic_ostream***, ***basic_ios*** и ***ios_base***.

21.4.6.2. Стандартные манипуляторы ввода/вывода

Стандартная библиотека предоставляет манипуляторы, относящиеся к различным состояниям формата и изменению состояний. Стандартные манипуляторы определены в пространстве имен ***std***. Манипуляторы, использующие ***ios_base***, представлены в <***ios***>. Манипуляторы, использующие ***istream*** и ***ostream***, представлены в <***istream***> и <***ostream***> соответственно, и, кроме того, в <***iostream***>. Остальные стандартные манипуляторы представлены в <***iomanip***>.

```
ios_base& boolalpha (ios_base&); // символьное представление true и false (при вводе и выводе)
ios_base& noboolalpha (ios_base& s);   // str.unsetf(ios_base::boolalpha)

ios_base& showbase (ios_base&);        // включает вывод 0 перед восьмеричными и 0x
                                       // перед шестнадцатеричными числами
ios_base& noshowbase (ios_base&);      // s.unsetf(ios_base::showbase)

ios_base& showpoint (ios_base&);
ios_base& noshowpoint (ios_base& s);   // s.unsetf(ios_base::showpoint)

ios_base& showpos (ios_base&);
ios_base& noshowpos (ios_base& s);     // s.unsetf(ios_base::showpos)

ios_base& skipws (ios_base&);          // пропуск «символов-разделителей»
ios_base& noskipws (ios_base& s);      // s.unsetf(ios_base::skipws)

ios_base& uppercase (ios_base&);       // X и E вместо x и e
ios_base& nouppercase (ios_base& );    // x и e вместо X и E

ios_base& internal (ios_base&);        // выравнивание (§ 21.4.5)
ios_base& left (ios_base&);            // заполнители после значения
```

```
ios_base& right (ios_base&);          // заполнители перед значением

ios_base& dec (ios_base&);            // десятичная система счисления, § 21.4.2
ios_base& hex (ios_base&);            // шестнадцатеричная система счисления
ios_base& oct (ios_base&);            // восьмеричная система счисления

ios_base& fixed (ios_base&);          // формат dddd.dd числа
                                      // с плавающей точкой (§ 21.4.3)
ios_base& scientific (ios_base&);     // научный формат d.ddddEdd

template<class Ch, class Tr>
    basic_ostream<Ch, Tr>& endl (basic_ostream<Ch, Tr>&);  // запись '\n' и очистка
template<class Ch, class Tr>
    basic_ostream<Ch, Tr>& ends (basic_ostream<Ch, Tr>&);  // запись '\0'
template<class Ch, class Tr>
    basic_ostream<Ch, Tr>& flush (basic_ostream<Ch, Tr>&); // очистка буфера потока

template<class Ch, class Tr>
    basic_istream<Ch, Tr>& ws (basic_ostream<Ch, Tr>&); // прочесть и проигнорировать
                                                        // все символы-разделители

smanip resetiosflags (ios_base::fmtflags f);  // сброс флагов (§ 21.4)
smanip setiosflags (ios_base::fmtflags f);    // установка флагов (§ 21.4)
smanip setbase (int b);                       // вывод целых чисел в системе
                                              // счисления с основанием b (§ 21.4.2)
smanip setfill (int c);                       // сделать с символом заполнения (§ 21.4.4)
smanip setprecisionl (int n);                 // n цифр (§ 21.4.3, 21.4.6.1)
smanip setw (int n);                          // в следующем вводе — n символов (§ 21.4.4)
```

Например,

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

выведет *1234, 4d2, 2322*, а

```
cout << '(' << setw (4) << setfill ('#') << '(' << 12 << ")(" << 12 << ")\n";
```

выведет *(##12)(12)*.

При использовании манипуляторов без аргументов *не* ставьте скобок. При использовании стандартных манипуляторов с аргументами, не забудьте включить **#include<iomanip>**. Например:

```
#include<iostream>
using namespace std

int main ()
{
    using namespase std;
    cout << setprecision (4)      // ошибка: setprecision не определена (забыли <iomanip>)
         << scientific ()         // ошибка: ostream<<ostream& (неуместные скобки)
         << 1.41421 << endl;
}
```

21.4.6.3. Манипуляторы, определяемые пользователем

Программист может добавлять собственные манипуляторы, написанные в том же стиле, что и стандартные. Здесь я представляю дополнительный стиль, который, мне кажется, может пригодиться для форматирования чисел с плавающей точкой.

Введенная точность сохраняется для всех операций вывода, но операция ***width*** () применяется только к ближайшей операции численного вывода. Мне же хотелось бы сделать нечто такое, что позволит просто выводить число с плавающей точкой в желаемом формате, не затрагивая последующих операций вывода в поток. Основная идея заключается в определении некоторого класса, который бы представлял форматы, другого, который бы представлял формат и форматируемое значение, и еще одного оператора <<, который в соответствии с форматом выводил бы это значение в ***ostream***. Например:

```
Form gen4 (4);                          // универсальный формат, точность 4

void f (double d)
{
        Form sci8 = gen4;
        sci8.scientific ().precision (8);       // научный формат, точность 8

        cout << d << ' ' << gen4 (d) << ' ' << sci8 (d) << ' ' << d << '\n';

}
```

При вызове *f (1234.56789)* выведется

```
1234.57 1235 1.23456789e+03 1234.57
```

Отметим, что использование ***Form*** не влияет на состояние потока, так что последний вывод числа *d* имеет тот же формат по умолчанию, что и первый.

Вот упрощенная реализация:

```
class Bound_form;       // Form плюс значение
class Form {
        friend ostream& operator<< (ostream&, const Bound_form&);

        int prc;        // точность
        int wdt;        // ширина поля, 0 означает столько цифр, сколько нужно
        int fmt;        // универсальный, научный или фиксированный форматы (§ 21.4.3)
        // ...

public:
        explicit Form (int p=6) : prc (p)               // по умолчанию точность 6
        {
                fmt = 0;                                // универсальный формат (§ 21.4.3)
                wdt = 0;                                // столько цифр, сколько понадобится
        }

        Bound_form operator () (double d) const;  // создание связанной формы для *this и d

        Form& scientific () { fmt = ios_base::scientific; return *this; }
        Form& fixed () { fmt = ios_base::fixed; return *this; }
        Form& general () { fmt = 0; return *this; }

        Form& uppercase ();
        Form& lowercase ();
        Form& precision (int p) { prc = p; return *this; }

        Form& width (int w) { wdt = w; return *this; }  // применяется ко всем типам
        Form& fill (char);

        Form& plus (bool b = true);                     // явный плюс
        Form& trailing_zeros (bool b = true);           // вывод незначащих нулей
        // ...
};
```

Идея заключается в том, чтобы ***Form*** содержал всю информацию, нужную для форматирования одного элемента данных. Выбор по умолчанию должен быть полезен для многих применений, а различные функции-члены можно использовать для задания отдельных аспектов форматирования. Оператор () используется для связывания значения с форматом, который используется для его вывода. Тогда ***Bound_form*** может может быть выведена в данный поток соответствующей функцией <<:

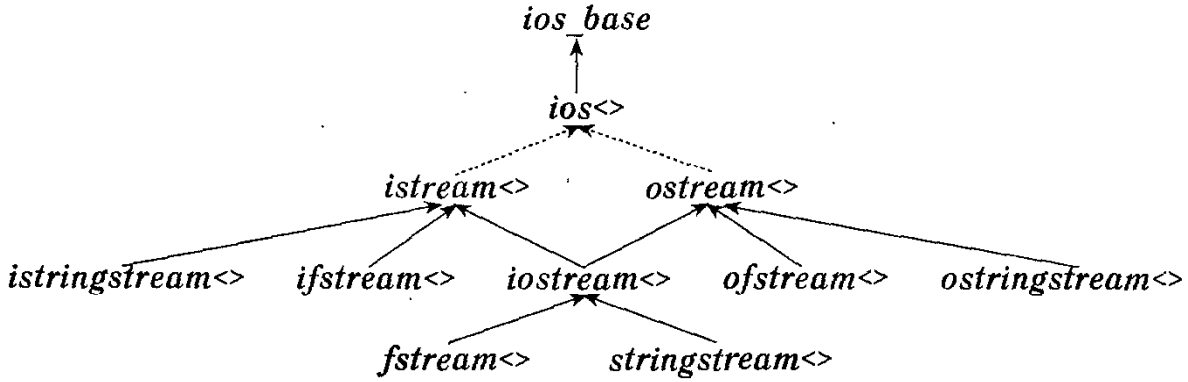
```
struct Bound_form {
    const Form& f;

    double val;
    Bound_form (const Form& ff, double v) : f (ff), val (v) { }
};

Bound_form Form::operator () const (double d) { return Bound_form (*this, d); }

ostream& operator<< (ostream& os, const Bound_form& bf)
{
    ostringstream s;                          // строковые потоки описаны в § 21.5.3
    s.precision (bf.f.prc);
    s.setf (bf.f.fmt, ios_base::floatfield);
    s << bf.val;                              // составление строки в s
    return os << s.str ();                    // вывод s в os
}
```

Написание менее примитивной реализации оператора << я оставляю в качестве упражнения (§ 21.10[21]). Классы ***Form*** и ***Bound_form*** легко расширяются для форматирования целых чисел, строк и т. д. (см. § 21.10[20]).

Отметим, что эти объявления превращают операторы << и () в тернарный оператор; ***cout<<sci4 (d)***, прежде чем производить действительные вычисления, собирает ***ostream***, формат и значение в единую функцию.

21.5. Файловые и строковые потоки

Когда программа на C++ начинает выполняться, можно пользоваться ***cout***, ***cerr***, ***clog***, ***cin*** и их эквивалентами для расширенных символов (§ 21.2.1). Эти потоки доступны по умолчанию, а их связь с устройствами ввода/вывода или файлами определяется «системой». Кроме того вы можете создать свои собственные потоки. В этом случае вы должны указать, к чему их следует прикрепить. Довольно часто поток прикрепляют к файлу или к строке ***string***, и поэтому такие действия поддерживаются стандартной библиотекой. Здесь представлена иерархия классов стандартных потоков:



Классы с суффиксом <> являются шаблонами, параметры которых имеют символьный тип; их имена начинаются с ***basic_***. Пунктирная линия означает виртуальные базовые классы (§ 15.2.4).

Файлы и строки — это примеры контейнеров, из которых можно считывать и в которые можно записывать. Следовательно, вы можете завести поток, поддерживающий как оператор <<, так и >>. Такой поток называется ***iostream***, он определен в пространстве имен ***std*** и представлен в <***iostream***>:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_iostream : public basic_istream<Ch, Tr>, public basic_ostream<Ch, Tr> {
public:
        explicit basic_iostream (basic_streambuf<Ch, Tr>* sb);
        virtual ~basic_iostream ();
};

typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
```

Управление чтением и записью в ***iostream*** осуществляется через две операции над буфером ***streambuf*** потока ***iostream*** — «записать в буфер» и «прочесть из буфера».

21.5.1. Файловые потоки

Здесь представлена полная программа копирования одного файла в другой. Имена файлов берутся из аргументов командной строки:

```
#include <fstream>
#include <cstdlib>
void error (const char* p, const char* p2="")
{
        std::cerr << p << ' ' << p2 << '\n';
        std::exit (1);
}
int main (int argc, char* argv[])
{
        if (argc != 3) error ("неверное число аргументов");

        std::ifstream from (argv[1]);          // открываем входной файл
        if (!from) error ("не открывается входной файл", argv[1]);

        std::ofstream to (argv[2]);            // открываем выходной файл
        if (!to) error ("не открывается выходной файл", argv[2]);

        char ch;
        while (from.get (ch)) to.put (ch);

        if (!from.eof () || !to) error ("случилось что-то странное");
}
```

Файл открывается для ввода созданием объекта класса ***ifstream*** (входной файловый поток) и имеет имя, задаваемое аргументом. Подобным же образом файл для вывода открывается созданием объекта класса ***ofstream*** (выходной файловый поток) и имеет имя, задаваемое аргументом. В обоих случаях мы проверяем состояние созданного объекта, чтобы узнать, благополучно ли открылся файл.

Класс ***basic_ofstream*** объявлен в <***fstream***>:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ofstream : public basic_istream<Ch, Tr> {
public:
        basic_ofstream ();
        explicit basic_ofstream (const char* p, openmode m=out | trunc);

        basic_filebuf<Ch, Tr>* rdbuf () const;

        bool is_open () const;
        void open (const char* p, openmode m= out | trunc);
        void close ();
}
```

Класс ***basic_ifstream*** похож на ***basic_ofstream*** за исключением того, что является производным от ***basic_istream*** и по умолчанию открыт для чтения. Дополнительно стандартная библиотека предлагает класс ***basic_fstream***, аналогичный ***basic_ofstream*** за исключением того, что он производится от ***basic_iostream*** и по умолчанию может быть открыт как на запись, так и на чтение.

Как обычно, для распространенных типов доступны определения типов ***typedef***:

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;

typedef basic_ifstream<wchar> wifstream;
typedef basic_ofstream<wchar> wofstream;
typedef basic_fstream<wchar> wfstream;
```

Класс ***ifstream*** похож на ***ofstream*** за исключением того, что является производным от ***istream*** и по умолчанию открывается на чтение. Кроме того стандартная библиотека предлагает класс ***fstream***, похожий на ***ofstream*** за исключением того, что он производится от ***iostream*** и по умолчанию открывается на запись и чтение.

Второй аргумент в конструкторах файловых потоков описывает, как открывается файл:

```
class ios_base {
public:
        //...
        typedef implementation_defined3 openmode;
        static openmode  app,       // добавление в конец
                         ate,       // открытие и поиск конца файла
                         binary,    // ввод/вывод в двоичном виде (а не текстовом)
                         in,        // открытие на чтение
                         out,       // открытие на запись
                         trunc;     // урезать файл до нулевой длины
        //...
};
```

Фактические значения ***openmode*** и их смысл определяются при реализации. Насчет деталей сверьтесь, пожалуйста, с руководством по вашей системе — и поэкспериментируйте. Комментарии должны дать некоторое представление о том, для чего предназначены соответствующие режимы открытия. Например, мы можем открыть файл, чтобы все добавлять ему в конец:

```
ofstream mystream (name.c_str (), ios_base::app);
```

Также можно открыть файл на запись и чтение:

```
fstream dictionary ("concordance", ios_base::in | ios_base::out);
```

21.5.2. Закрытие потоков

Файл можно закрыть, явно вызвав ***close*** () данного потока:

```
void f(ostream& mystream)
{
    //...
    mystream.close();
}
```

Однако это неявно делается деструктором потока. Поэтому явный вызов ***close*** () требуется только в том случае, если файл нужно закрыть до конца области видимости, в которой объявлен данный поток.

Тогда возникает вопрос: как реализация может гарантировать, что предопределенные потоки ***cout***, ***cin*** и ***cerr*** создадутся до их первого использования и закроются (только) после последнего? Естественно, для достижения этого различные реализации библиотеки потоков <***iostream***> могут использовать различные приемы. В конце концов, пользователю не должно быть видно, как именно реализованы детали. Здесь я привожу лишь один прием, достаточно универсальный для того, чтобы гарантировать соответствующий порядок выполнения конструкторов и деструкторов для глобальных объектов различных типов. Реализации могут быть сделаны лучше за счет использования особенностей компилятора или компоновщика.

Основная идея заключается в том, чтобы определить вспомогательный класс, который является счетчиком, отслеживающим, сколько раз <***iostream***> был включен в отдельно компилируемый исходный файл:

```
class ios_base::Init {
    static int count;
public:
    Init();
    ~Init();
};

namespace {      // в <iostream>, по копии на каждый файл, где есть #include <iostream>
    ios_base::Init __ioinit;
}

int ios_base::Init::count = 0;          // вставить в какой-нибудь файл .c
```

Каждая единица трансляции (§ 9.1) объявляет свой собственный объект с именем ***__ioinit***. Конструктор для объектов ***__ioinit*** использует ***ios_base::Init::count*** как индикатор первого вызова, дабы гарантировать, что действительная инициализация глобальных объектов в библиотеке потоков ввода/вывода выполнится ровно один раз:

```
ios_base::Init::Init()
{
    if (count++ == 0) { /*инициализация cout, cerr, cin и т. д. */ }
}
```

И наоборот, деструктор для объектов ***ios_base::Init*** использует ***ios_base::Init::count*** как индикатор последнего вызова, дабы гарантировать, что потоки закроются:

```
ios_base::Init::~Init()
{
    if (--count == 0)   { /*очистка cout (например, очистка буфера), cerr, cin и т. д. */ }
}
```

Это универсальный прием для работы с библиотеками, требующими инициализации и очистки глобальных объектов. В системах, где весь код во время выполнения располагается в основной памяти, этот прием почти не имеет ограничений. В противном случае могут оказаться значительными затраты на занесение каждого объектного файла в основную память для выполнения его инициализирующих функций. Глобальных объектов по мере возможности следует избегать. Для классов, где каждая операция выполняет важную работу, чтобы гарантировать инициализацию, было бы разумно в каждой операции проверять индикатор первого вызова (как *ios_base::Init::count*). Однако этот подход для потоков может оказаться чересчур дорогим. В функциях, считывающих и записывающих один символ, затраты на индикатор первого вызова окажутся чрезмерными.

21.5.3. Строковые потоки

Поток можно прикрепить к строке. То есть мы можем считывать из строки *string* и писать в такую строку при помощи предоставляемых потоками возможностей форматирования. Такие потоки называют *stringstream*. Они определены в *<sstream>*:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_stringstream : public basic_iostream<Ch, Tr> {
public:
    explicit basic_stringstream (ios_base::openmode m = out | in);
    explicit basic_stringstream (const basic_string<Ch>& s, openmode m = out | in);
    basic_string<Ch> str () const;                  // получение копии string
    void str (const basic_string<Ch>& s);           // установка значения копии s

    basic_stringbuf<Ch, Tr, A>* rdbuf () const;     // получение указателя
                                                    // на текущий файловый буфер
};
```

Класс *basic_istringstream* похож на *basic_stringstream* за исключением того, что он производится от *basic_istream* и по умолчанию открыт для чтения. Класс *basic_ostringstream* похож на *basic_stringstream* за исключением того, что он производится от *basic_ostream* и по умолчанию открыт для записи. Как обычно, функции обеспечивают основные специализации:

```
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;

typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;
```

Например, *ostringstream* можно использовать для форматирования строки сообщения:

```
string compose (int n, const string& cs)
{
    extern const char* std_message[];
    ostringstream ost;
    ost << "ошибка (" << n << ")" << std_message[n] << " (комментарий пользователя:" << cs << ')';
    return ost.str ();
}
```

Проверять на переполнение нет необходимости, поскольку *ost* расширяется автоматически. Этот прием может оказаться очень полезным для копирования в случаях,

когда требуется форматирование более сложное, чем построковый вывод в устройствах телетайпного типа.

Обращение с начальным значением для строкового потока аналогично обращению файлового потока со своим файлом:

```
string compose2 (int n, const string& cs)          // эквивалентно compose()
{
        extern const char* std_message[];
        ostringstream ost ("ошибка (", ios_base::ate);     // пишем в конец
        ost << n << ") "<< std_message[n] << " (комментарий пользователя: " << cs << ')';
        return ost.str ();
}
```

Класс ***istringstream*** — это входной поток, который считывает данные из своей строки инициализации (точно так же, как ***ifilestream*** считывает из файла):

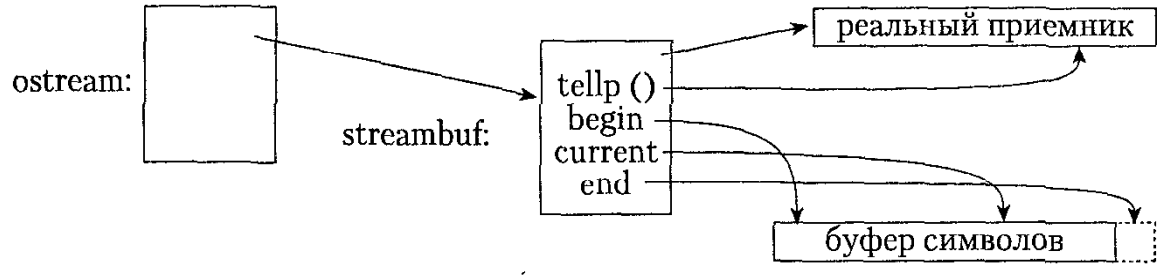
```
#include<sstream>

void word_per_line (const string& s)               // вывод по одному слову в строке
{
        istringstream ist (s);
        string w;
        while (ist>>w) cout << w << '\n';
}

int main ()
{
        word_per_line ("Если вы считаете, что C++ труден,
                попытайтесь выучить английский");
}
```

Строка-инициализатор копируется в ***istringstream***. Вывод заканчивается по символу конца строки.

Можно определить потоки, которые осуществляют непосредственное считывание и запись в массивы символов (§ 21.10[26]). Часто это полезно при работе с устаревшими программами, тем более, что делающие это классы ***ostrstream*** и ***istrstream*** были частью изначальной библиотеки потоков.

21.6. Буферизация

Вообще говоря, выходной поток кладет символы в буфер. Через какое-то время они переписываются туда, куда нужно. Такой буфер называется ***streambuf*** (§ 21.6.4). Его определение находится в <***streambuf***>. Различные реализации буферов ***streambuf*** используют различные стратегии. Как правило, ***streambuf*** хранит символы в массиве, пока переполнение не вынудит его очиститься и записать символы по назначению. Таким образом, ***ostream*** можно представить графически следующим образом:



Набор аргументов шаблона для *ostream* и его *streambuf* должен быть один и тот же; он определяет тип символов в буфере символов.

Класс *istream* похож на *ostream* за исключением того, что символы передаются в другую сторону.

Небуферизованный ввод/вывод — это просто ввод/вывод, в котором буфер сразу же передает каждый символ, не ожидая, пока их наберется достаточно для того, чтобы сделать передачу эффективной.

21.6.1. Потоки вывода и буфера

Класс *ostream* вводит операции для преобразования значений различных типов в последовательность символов в соответствии с общепринятыми соглашениями (§ 21.2.1) и явными директивами форматирования (§ 21.4). Кроме того, *ostream* предоставляет операции, имеющие дело напрямую со *streambuf*:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
        //...

        explicit basic_ostream (basic_streambuf<Ch, Tr>* b);
        pos_type tellp ();                              // получение текущей позиции
        basic_ostream& seekp (pos_type);                // установка текущей позиции
        basic_ostream&
                seekp (off_type, ios_base::seekdir);    // установка текущей позиции

        basic_ostream& flush ();            // очистка буфера (запись в реальный приемник)
        basic_ostream& operator<< (basic_streambuf<Ch, Tr>*b);    // запись из b
};
```

Конструктор класса *ostream* имеет аргументом объект *streambuf*, который определяет, как записанные символы обрабатываются, и куда они в конце концов уходят. Например, *ostringstream* (§ 21.5.3) или *ofstream* (§ 21.5.1) создаются инициализацией *ostream* подходящим буфером *streambuf* (§ 21.6.4).

Функции *seekp* () используются для того, чтобы установить позицию записи в *ostream*. Суффикс *p* указывает, что эта позиция используется для *записи* (putting) символов в поток. Эти функции не оказывают никакого эффекта, если поток не прикреплен к чему-нибудь такому, для чего позиционирование имеет смысл — например, к файлу. *pos_type* представляет позицию символа в файле, а *off_type* — его смещение от точки, на которую указывает *ios_base::seekdir*:

```
class ios_base {
        //...

        typedef implementation_defined4 seekdir;
        static const seekdir beg,           // поиск от начала текущего файла
                             cur,           // поиск от текущей позиции
                             end;           // поиск назад от конца текущего файла

        //...
};
```

Позиции в потоке нумеруются, начиная с нулевой, поэтому мы можем представить себе файл как массив из *n* символов. Например:

```
int f(ofstream& fout)          // fout ссылается на некоторый файл
{
    fout.seekp(10);            // добавляем символ и сдвигаемся на (+1)
    fout << '#';
    fout.seekp(-2, iosbase::cur);
    fout <<'*';
}
```

В результате выполнения этой функции в *file*[*10*] запишется # , а в *file*[*9*] — * . Но никакого подобного способа для произвольного доступа к элементам простых потоков ***istream*** и ***ostream*** не существует (см. 21.10[13]). Попытка поиска до начала или после конца файла приводит поток в состояние ***bad*** () (§ 21.3.3).

Операция ***flush*** () позволяет пользователю очищать буфер, не ожидая переполнения.

Для непосредственной записи ***streambuf*** в ***ostream*** можно воспользоваться оператором <<. Это удобно прежде всего для тех, кто реализует механизмы ввода/вывода.

21.6.2. Потоки ввода и буфера

Класс ***istream*** вводит операции для чтения символов и преобразования их в значения различных типов (§ 21.3.1). Кроме того, ***istream*** предоставляет операции, которые имеют дело напрямую со ***streambuf***:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    //...

    explicit basic_istream(basic_streambuf<Ch, Tr>* b);
    pos_type tellg();                          // получение текущей позиции
    basic_istream& seekg(pos_type);            // установка текущей позиции
    basic_istream&
        seekg(off_type, ios_base::seekdir);    // установка текущей позиции

    basic_istream& putback(Ch c);              // положить с обратно в буфер
    basic_istream& unget();                    // положить обратно в буфер
                                               // последний считанный символ
    int_type peek();                           // посмотреть, какой следующий
                                               // символ прочтется

    int sync();                                // очистить буфер (сброс)

    basic_istream& operator>>(basic_streambuf<Ch, Tr>* b);   // чтение в b
    basic_istream& get(basic_streambuf<Ch, Tr>& b, Ch t=Tr::newline());

    streamsize readsome(Ch* p, streamsize n);  // считывание максимум n символов
};
```

Функции позиционирования работают так же, как и их аналоги из ***ostream*** (§ 21.6.1). Суффикс *g* указывает, что эти позиции используются для *чтения* (getting) символов из потока. Суффиксы ***p*** и ***g*** нужны потому, что мы можем создать класс ***iostream***, производный и от ***istream***, и от ***ostream***, а такой поток должен отслеживать как запись, так и чтение символов по данной позиции.

Функция ***putback*** () позволяет программе возвращать ненужный символ в поток, чтобы прочитать его в другой раз, как показано в § 21.3.5. Функция ***unget*** () помеща-

ет последний считанный символ обратно. К сожалению, возвращение в поток ввода не всегда возможно. Например, попытка вернуть первый считанный символ установит бит *ios_base::failbit*. Все, что гарантируется — это возможность возврата одного символа после благополучного считывания. Функция *peek* () читает следующий символ, но оставляет его в *streambuf*, так что его можно прочитать снова. Таким образом, *c=peek* () означает то же самое, что (*c=get* (), *unget* (), *c*) и (*putback* (*c=get* ()), *c*). Отметим, что установка *failbit* может вызвать исключение (§ 21.3.6).

Синхронизация потока *istream* и буфера осуществляется при помощи *sync* (). Это не всегда выполняется правильно. Для некоторых потоков нам бы пришлось повторно считывать символы с действительного источника, а это не всегда возможно и желательно. Поэтому *sync* () при успешном выполнении возвращает *0*, а в противном случае устанавливает бит *ios_base::badbit* (§ 21.3.3) и возвращает *–1*. И снова напомню, что установка *badbit* может вызвать исключение (§ 21.3.6). Функция *sync* () для буфера, связанного с *ostream*, сбрасывает содержимое буфера на устройство вывода.

Операции >> и *get* (), читающие в *streambuf*, полезны прежде всего для тех, кто реализует средства ввода/вывода, и только им следует напрямую манипулировать со *streambuf*.

Функция *readsome* () является операцией нижнего уровня, которая позволяет пользователю посмотреть, есть ли в потоке символы, доступные для считывания. Это может очень пригодиться, когда нежелательно ждать ввода, например с клавиатуры. Посмотрите также функцию *in_avail* () (§ 21.6.4).

21.6.3. Потоки и буфера

Связь между потоком и его буфером поддерживается в *basic_ios* данного потока:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
        //...
        basic_streambuf<Ch, Tr>* rdbuf () const;        // получение буфера
        // установка и очистка буфера, возвращает указатель на старый буфер
        basic_streambuf<Ch, Tr>* rdbuf (basic_streambuf<Ch, Tr> *b);
        locale imbue (const locale& loc);               // установка национальных
                                                        // особенностей (и получение старых)

        char narrow (char_type c, char d) const;        // получение символьного
                                                        // значения из char_type c
        char_type widen (char c) const;                 // получение значение типа
                                                        // char_type по символу c
        //...
protected:
        basic_ios ();
        void init (basic_streambuf<Ch, Tr>* b);         // установка начального буфера
};
```

В дополнение к чтению и установке буфера данного потока (§ 21.6.4) шаблон *basic_ios* вводит *imbue*, чтобы читать и заново устанавливать национальные особенности потока (§ 21.7) путем вызова *imbue* () для его *ios_base* (§ 21.7.1) и *pubimbue* () для его буфера (§ 21.6.4).

Функции *narrow*() и *widen*() используются для преобразования символа *char* в символьный тип *char_type* буфера и обратно. Второй аргумент функции *narrow*(*c*, *d*) — это символ, возвращаемый, если не существует символа, соответствующего значению *c* типа *char_type*.

21.6.4. Буфера потоков

Операции ввода/вывода определяются без упоминания о типах файлов, но не со всеми устройствами можно обращаться одинаково в смысле буферизации. Например, поток *ostream*, связанный со строкой *string* (§ 21.5.3), нуждается не в таком типе буфера, как *ostream*, связанный с файлом (§ 21.5.1). Эти трудности преодолеваются во время инициализации указанием разных типов буфера для разных потоков. Над этими типами буферов существует всего один набор операций, поэтому функции класса *ostream* не содержат кода, который бы их отличал. Разные типы буферов являются производными от класса *streambuf*. Класс *streambuf* предоставляет виртуальные функции для операций, таких как обработка переполнения буфера сверху или снизу, в которых стратегии буферизации отличаются.

Класс *basic_streambuf* вводит два интерфейса. Открытый интерфейс предназначается главным образом для тех, кто реализует классы потоков, такие как *istream*, *ostream*, *fstream*, *stringstream* и т. д. Кроме того предоставляется защищенный интерфейс для тех, кто реализует новые стратегии буферизации и буфера *streambuf* для новых источников ввода и приемников вывода.

Чтобы понять, что такое *streambuf*, полезно сначала рассмотреть основополагающую модель буферной области, предоставляемую защищенным интерфейсом. Допустим, что *streambuf* имеет *область записи*, в которую пишет оператор <<, и *область чтения*, из которой читает оператор >>. Каждая область описывается начальным указателем, текущим указателем и указателем на элемент, следующий за концом. Эти указатели доступны через следующие функции:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_streambuf{
protected:
    Ch* eback() const;              // начало буфера чтения
    Ch* gptr() const;               // следующий заносимый символ (чтение
                                    // следующегосимвола происходит отсюда)
    Ch* egptr() const;              // элемент, следующий за концом буфера чтения

    void gbump(int n);              // добавление n к gptr()
    void setg(Ch* begin, Ch* next, Ch* end);    // установка eback(), gptr() и egptr
    Ch* pbase() const;              // начало буфера записи
    Ch* pptr() const;               // следующий записываемый символ пойдет сюда
    Ch* epptr() const;              // элемент, следующий за концом буфера записи
    void pbump(int n);              // добавление n к pptr()
    void setp(Ch* begin, Ch* end);  // установка pbase() и pptr() в начало,
                                    // а epptr() в конец
    // ...
};
```

Имея массив символов, функции *setg*() и *setp*() могут соответствующим образом установить указатели. В реализации можно произвести доступ к этим областям следующим образом:

```
template<class Ch, class Tr = char_traits<Ch> >
basic_streambuf<Ch, Tr>::int_type basic_streambuf<Ch, Tr>::snextc ()
    // пропустить текущий символ, затем прочитать следующий
{
    if (1<egptr () -gptr()) {                  // если в буфере есть не менее 2-х символов
        gbump (1);                             // пропустить текущий символ
        return Tr::to_int_type (*gptr ());     // возвращает новый текущий символ
    }
    if (1==egptr () -gptr()) {                 // если в буфере ровно 1 символ
        gbump (1);                             // пропустить текущий символ
        return underflow ();
    }
    // буфер пуст (или нет буфера), попытка заполнить буфер:
    if (Tr::eq_int_type (uflow (), Tr::eof ()) ) return Tr::eof ();
    if (0<egptr() -gptr()) return Tr::to_int_type (*gptr ()); // возвращает новый текущий символ
    return underflow ();
}
```

Доступ к буферу осуществляется через ***gptr*** (); ***egptr*** () отмечает границы полученной области. Символы считываются из реального источника функциями ***uflow*** () и *underflow* (). Вызовы ***traits_type::to_int_type*** гарантируют, что код не зависит от фактического типа символа. Этот код допустим для различных типов буфера потока и учитывает возможность того, что виртуальные функции ***uflow*** () и ***underflow*** () могут ввести новую область (используя ***setg*** ()).

Открытый интерфейс ***streambuf*** выглядит следующим образом:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
public:
    // обычные определения typedef (§ 21.2.1)

    virtual ~basic_streambuf ();

    locale pubimbue (const locale &loc);       // установка национальных
                                               // особенностей (и получение старых)
    locale getloc () const;                    // получение национальных особенностей

    basic_streambuf* pubsetbuf (Ch* p, streamsize n);     // установка памяти
                                                          //для буфера

    // установка позиции (§ 21.6.1):
    pos_type pubseekoff (off_type off, ios_base::seekdir way,
        ios_base::openmode m=ios_base::in | ios_base::out);
    pos_type pubseekpos (pos_type p,
        ios_base::openmode m=ios_base::in | ios_base::out);

    int pubsync ();          // используется при синхронизации ввода sync() (§ 21.6.2)

    template<class Ch, class Tr = char_traits< Ch> > basic_streambuf<Ch, Tr>::int_type
    basic_streambuf<Ch, Tr>::snextc ();          // чтение следующего символа
    int_type sbumpc ();                          // продвижение gptr() на 1
    int_type sgetc ();                           // получение текущего символа
    streamsize sgetn (Ch* p, streamsize n);      // чтение в p[0]..p[n-1]

    int_type sputbackc (Ch c);                   // положить с обратно в буфер
```

```
        int_type sungetc ();                    // отменить чтение последнего символа
        int_type sputc (Ch c);                  // записать c
        streamsize sputn (const Ch* p, streamsize n); // записать p[0]..p [n-1]
        streamsize in_avail ();                 // ввод готов?
        // ...
};
```

Открытый интерфейс содержит функции для вставки символов в буфер и извлечения их из буфера. Эти функции просты и легко встраиваются, а это имеет решающее значение для эффективности.

Функции, реализующие части специфической стратегии буферизации, обращаются к соответствующим функциям защищенного интерфейса. Например, ***pubsetbuf*** () вызывает ***setbuf*** (), которая замещена в производном классе для реализации концепции этого класса по выделению памяти для буферизации символов. Использование двух функций для реализации таких операций как установка буфера ***setbuf*** позволяет разработчику ***iostream*** произвести некоторую «уборку» до и после пользовательской программы. Например, разработчик реализации может сделать оболочку вокруг вызова виртуальной функции и перехватывать исключения, сгенерированные в пользовательском коде.

По умолчанию ***setbuf*** (***0***, ***0***) означает «отсутствие буферизации», а ***setbuf*** (***p***, ***n***) означает использование ***p***[***0***]..***p***[***n***−***1***] для хранения буферизованных символов.

Вызов ***in_avail*** () используется для того, чтобы посмотреть, сколько доступных символов в буфере. Это может пригодиться, чтобы не ждать ввода. Во время считывания из потока, связанного с клавиатурой, ***cin.get*** (***c***) может дожидаться, пока пользователь вернется с обеда. В некоторых системах и для некоторых прикладных программ это следует учитывать. Например:

```
if (cin.rdbuf ().in_avail ()) {          // get() не заблокирует
        cin.get (c);
        // что-то делается
}
else {                                   // get() может заблокировать
        // делается что-то другое
}
```

Отметим, что в некоторых системах трудно определить доступен ли ввод. Так, реализация ***in_aviail***() может (неудачно) устанавливать, что ***in_aviail***() возвращает ***0*** в случаях, когда операция ввода может пройти успешно.

В дополнение к открытому интерфейсу, используемому ***basic_istream*** и ***ostream***, ***basic_streambuf*** предлагает защищенный интерфейс для разработчиков буферов потоков. Именно там и объявляются определяющие тактику действий виртуальные функции:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:
        // ...
        basic_streambuf ();
        virtual void imbue (consr locale &loc);   // установка национальных особенностей
        virtual basic_streambuf* setbuf (Ch* p, streamsize n);
        virtual pos_type seekoff (off_type off, ios_base::seekdir way,
                        ios_base::openmode m = ios_base::in | ios_base::out);
```

```
        virtual pos_type seekpos (pos_type p,
                                  ios_base::openmode m = ios_base::in | ios_base::out);
        virtual int sync ();                                    // синхронизация ввода (§ 21.6.2)
        virtual int showmanyc ();
        virtual streamsize xsgetn (Ch* p, streamsize n);        // чтение n символов
        //повторное заполнение области; возвращает символ или eof
        virtual int_type underflow ();
        //повторное заполнение области; возвращает символ или eof, сдвигает gptr() на 1
        virtual int_type uflow ();
        virtual int_type pbackfail (int_type c = Tr::eof());    // положить обратно не удалось
        virtual streamsize xsputn (const Ch* p, streamsize n); // записать n символов
        virtual int_type overflow (int_type c = Tr::eof());     // записать всю область
};
```

К функциям ***underflow*** () и ***uflow*** () обращаются для получения символа из реального источника, когда буфер пуст. Если ввод из этого источника больше не доступен, поток устанавливается в состояние ***eof*** () (§ 21.3.3). Если это не приводит к генерации исключения, возвращается ***traits_type::eof*** (). После возврата символа функцией ***uflow*** (), но не ***underflow*** (), ***gptr*** () сдвигается на *1*. Помните, что, как правило, в вашей системе больше буферов, чем введено в библиотеке ***iostream***, поэтому вы можете столкнуться с задержками буферизации, даже пользуясь небуферизованным потоком ввода/вывода.

К функции ***overflow*** () обращаются для передачи символов в реальный приемник вывода по заполнении буфера. Вызов ***overflow*** (*c*) выводит содержимое буфера и символ *c*. Если в данный приемник больше невозможно ничего вывести, поток устанавливается в состояние ***eof*** (§ 21.3.3). Если при этом не сгенерируется исключение, возвращается ***traits_type::eof*** ().

Функция ***showmanyc*** () — «показать, сколько символов» («show how many characters») — это дополнительная функция, позволяющая пользователю узнавать о состоянии машинной системы ввода. Эта функция возвращает оценку, сколько символов можно считать «быстро» — скажем, очистив буфера операционной системы, а не дожидаясь чтения с диска. Вызов ***showmanyc*** () возвращает *–1*, если функция не может обещать, что считается хотя бы один символ без получения конца файла. Это (обязательно) — функция низкого уровня, в большой степени зависящая от реализации. Не пользуйтесь ***showmanyc*** (), не ознакомившись внимательно с документацией на вашу систему и не проведя несколько экспериментов.

По умолчанию все потоки получают глобальные национальные особенности (§ 21.7). Вызов ***pubimbue*** (***loc***) или ***imbue*** (***loc***) приводит к тому, что поток в качестве национальных особенностей использует аргумент ***loc***.

Буфер ***streambuf*** для конкретного вида потока является производным от ***basic_streambuf***. Он обеспечивает конструкторы и инициализирующие функции, которые связывают ***streambuf*** с реальным источником (или приемником) символов, и замещает виртуальные функции, определяющие стратегию буферизации. Например:

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_filebuf : public basic_streambuf<Ch, Tr> {
public:
        basic_filebuf ();
        virtual ~basic_filebuf ();
```

```
        bool is_open () const;
        basic_filebuf* open (const char* p, ios_base::openmode mode);
        basic_filebuf* close ();

    protected:
        virtual int showmanyc ();
        virtual int_type underflow ();
        virtual int_type uflow ();

        virtual int_type pbackfail (int_type c = Tr::eof ());
        virtual int_type overflow (int_type c = Tr::eof ());

        virtual basic_streambuf<Ch, Tr>* setbuf (Ch* p, streamsize n);
        virtual pos_type seekoff (off_type off, ios_base::seekdir way,
                                  ios_base::openmode m = iosbase::in | ios_base::out);
        virtual pos_type seekpos (pos_type p,
                                  ios_base::openmode m = iosbase::in | ios_base::out);
        virtual int sync ();
        virtual void imbue (const locale& loc);
    };
```

Функции для манипулирования буферами и т. п. без изменений наследуются от *basic_streambuf*. Только функции, влияющие на инициализацию и политику буферизации, должны быть введены отдельно.

Как обычно, предоставлены очевидные определения *typedef* и их аналоги для расширенных символьных потоков:

```
typedef basic_streambuf<char> streambuf;
typedef basic_stringbuf<char> stringbuf;
typedef basic_filebuf<char> filebuf;

typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
typedef basic_filebuf<wchar_t> wfilebuf;
```

21.7. Национальные особенности

Национальная особенность ***locale*** — это объект, который контролирует классификацию символов на буквы, цифры и т. д., устанавливаст порядок следования символов, управляет видом вводимых и выводимых численных значений. Чаще всего национальные особенности неявно используются в библиотеке потоков ввода-вывода, обеспечивая способы написания, принятые для данного естественного языка, и следование традициям. В этом случае пользователь никогда не видит объекта ***locale***. Однако, изменив национальные особенности для некоторого потока ***stream***, программист может изменить поведение потока ради выполнения соглашений, принятых в другой культуре.

Национальные особенности представляются объектом класса ***locale***, определенного в пространстве имен ***std***, заголовочный файл <***locale***> (§ Г.2):

```
class locale {                          // полное объявление см. в § Г.2
public:
    // ...
    // конструктор по умолчанию копирует текущие глобальные
    locale () throw ;                   //национальные особенности
```

```
        // конструктор, использующий национальные особенности из C
        explicit locale (const char* name);
        // выдает имя конкретных национальных особенностей
        basic_string<char> name () const;

        locale (const locale&) throw ();          // копирует национальные особенности
        const locale& operator= (const locale&) throw (); // копирует национальные особенности

        // установка новых национальных особенностей (получение предыдущих)
        static locale global (const locale&);
        static const locale& classic ();          // получение особенностей, определенных C
};
```

Простейшее использование *locale* — это переключение с одних национальных особенностей на другие. Например:

```
void f()
{
        std::locale loc ("POSIX");      // стандартные национальные особенности для POSIX
        cin.imbue (loc);                // пусть cin пользуется loc
        //...
        // восстановление для cin национальных особенностей по умолчанию
        cin.imbue (std::locale ());
}
```

Функция *imbue* () является членом *basic_ios* (§ 21.7.1).

Как показано в вышеприведенном примере, некоторые распространенные национальные особенности имеют строковые имена. Они, как правило, используются совместно с C.

Можно установить национальные особенности, которые используется по умолчанию для всех вновь созданных потоков:

```
void g (const locale& loc=locale ())     // использует текущие глобальные
                                         // национальные особенности по умолчанию
{
        // теперь национальными особенностями по умолчанию будут loc
        locale old_global = locale::global (loc);
        //...
}
```

Установка глобальных национальных особенностей не изменяет поведения существующих потоков, пользующихся предыдущим значением глобальных национальных особенностей. В частности, это не влияет на *cin*, *cout* и т. д. Если нужно изменить их, к ним нужно явно применить функцию *imbue* ().

Вызов для потока функции *imbue* () и установка новых национальных особенностей изменяет некоторые детали его поведения. Можно напрямую использовать члены *locale*, можно определять новые *locale*, можно дополнять *locale* новыми деталями. Например, национальные особенности можно явно использовать, чтобы изменить для ввода и вывода обозначение денежных единиц, способ написания дат и т. п. (§ 21.10[25]), а также для того, чтобы преобразовывать кодовые наборы. Концепция локализации, классы *locale* и *facet*, а также стандартные национальные особенности и фасеты описаны в приложении Г.

Национальные особенности в стиле C представлены в <*clocale*> и <*locale.h*>.

21.7.1. Функции обратного вызова для потоков

Иногда люди хотят кое-что добавить к состоянию потока. Например, кому-то может захотеться, чтобы поток «знал», в каком виде будут выводиться переменные типа *complex* — в полярных или декартовых координатах. Класс *ios_base* предоставляет функцию *xalloc* (), чтобы выделить память для такой простой информации о состоянии. Значение, возвращаемое *xalloc* (), идентифицирует местоположение, к которому можно обратиться через функции *iword* () и *pword* ():

```
class ios_base {
public:
        //...
        ~ios_base ();
        locale imbue (const locale& loc);   // установка новых и возврат старых
                                            // национальных особенностей, см. § Г.2.3
        locale getloc () const;             // получение национальных особенностей
        static int xalloc ();               // выделяет память для целого и для указателя
                                            // (оба инициализируются нулем)
        long& iword (int i);                // доступ к целому iword(i)
        void*& pword (int i);               // доступ к указателю pword(i)
        // обратные вызовы:
        enum event { erase_event, imbue_event, copyfmt_event };     // тип события
        typedef void (*event_callback) (event, ios_bases&, int i);
        void register_callback (event_callback f, int i);           // прикрепление f к pword(i)
};
```

Иногда разработчику реализации или пользователю нужно знать об изменении в состоянии потока. Функция *register_callback* () «регистрирует» функцию, которая будет вызвана, когда случится «событие». Таким образом, при вызове *imbue* (), *copyfmt* () или *~ios_base* () вызывается функция «зарегистрированная» для *imbue_event*, *copyfmt_event* или *erase_event* соответственно. Когда состояние изменяется, зарегистрированные функции вызываются с аргументом *pword* (*i*), где *i* предоставлено *register_callback* ().

Этот механизм хранения и обратного вызова довольно сложен. Пользуйтесь им, только когда вам крайне необходимо расширить возможности форматирования нижнего уровня.

21.8. Ввод-вывод на C

Поскольку программы на C и C++ часто смешиваются, потоки ввода/вывода C++ часто совмещаются с семейством функций ввода/вывода C — *printf*. Функции ввода/вывода в стиле C представлены в <*cstdio*> и <*stdio.h*>. Поскольку C-функции можно вызывать из C++, некоторые программисты могут предпочесть знакомые C-функции ввода/вывода. Даже если сами вы предпочитаете потоковый ввод/вывод, вы несомненно когда-нибудь наткнетесь на ввод/вывод в стиле C.

Ввод/вывод C и C++ может смешиваться посимвольно. Вызов *sync_with_stdio* () до первой операции с потоками ввода/вывода при выполнении программы гарантирует, что операции ввода/вывода в стиле C и в стиле C++ будут иметь общие буфера. Вызов *sync_with_stdio* (*false*) до первой операции с потоками ввода/вывода предотвращает совместное использование буфера и для некоторых реализаций может улучшить характеристики ввода/вывода:

```
class ios_base {
    //...
    static bool sync_with_stdio (bool sync = true);        // получение и установка
};
```

Вызов *sync_with_stdio (false)* «разъединяет» ввод/вывод в стиле C и C++ и в некоторых реализациях может привести к улучшению производительности.

Главное преимущество функций потокового вывода над функциями *printf* () из стандартной библиотеки C заключается в том, что потоки безопасны с точки зрения типов и имеют общий стиль для вывода объектов как встроенных типов, так и типов, определяемых пользователем.

Основные функции вывода C

```
int printf(const char* format...);          // запись в stdout
int fprintf(FILE*, const char* format...);  // запись в «файл» (stdout, stderr и т. д.)
int sprintf(char* p, const char* format...); // запись в p[0]..
```

производят форматированный вывод произвольной последовательности аргументов под управлением строки формата *format*. Строка формата содержит два типа объектов — простые символы, которые просто копируются в поток вывода, и спецификации преобразования, каждая из которых приводит к преобразованию и выводу следующего аргумента. Перед каждой из спецификаций преобразования пишется знак %. Например:

```
printf("присутствовало %d членов.", num_of_members);
```

Здесь *%d* сообщает, что с *num_of_members* (числом членов) нужно обращаться как с целым, и вывести соответствующей последовательностью десятичных цифр. Если *num_of_members==127*, на выходе получим:

```
присутствовало 127 членов.
```

Набор спецификаций преобразования очень широк и обеспечивает высокую степень гибкости. После % может стоять:

- \- необязательный минус, который сообщает о выравнивании преобразованного значения влево в указанном поле;
- \+ необязательный плюс, который указывает, что перед значением типа со знаком всегда должен стоять + или −;
- 0 необязательный ноль, который сообщает, что лидирующие нули используются для заполнения численного значения. Если установлена спецификация — (минус) или точность, данный 0 игнорируется;
- # необязательный знак #, указывающий, что: числа с плавающей точкой будут выводиться с десятичной точкой, даже если после точки идут одни нули; будут выводиться незначащие нули;перед восьмеричными числами будет выводиться *0*, а перед шестнадцатеричными — *0x* или *0X*;
- d необязательная строка цифр, определяющая ширину поля; если преобразованное число имеет меньше цифр, чем умещается в поле, то поле будет дополнено пробелами слева (или справа, если указан признак выравнивания вправо); если строка начинается с нуля, то вместо пробелов поле заполняется нулями;
- . необязательная точка, которая служит для отделения ширины поля от следующей строки цифр;
- d необязательная строка цифр, которая определяет точность, указывая, сколько цифр выводить после точки для преобразований *e* и *f*, или максимальное число выводимых цифр;

* ширина поля или точность может задаваться *, а не строкой цифр. В этом случае ширину строки или точность указывает целочисленный аргумент;

h необязательный символ ***h***, указывающий, что следующее ***d***, ***o***, ***x*** или ***u*** относится к целочисленному аргументу типа ***short***;

l необязательный символ ***l***, указывающий, что следующее ***d***, ***o***, ***x*** или ***u*** относится к целочисленному аргументу типа ***long***;

% указывает, что следует вывывести символ %; никаких аргументов не используется;

c символ, указывающий тип преобразования, которое нужно применить. Символы преобразования и их значения таковы:

- d Целочисленный аргумент преобразуется в десятичный вид;
- i Целочисленный аргумент преобразуется в десятичный вид;
- o Целочисленный аргумент преобразуется в восьмеричный вид;
- x Целочисленный аргумент преобразуется в шестнадцатеричный вид;
- X Целочисленный аргумент преобразуется в шестнадцатеричный вид;
- f Аргумент типа ***float*** или ***double*** преобразуется в десятичный вид [-]*ddd.ddd*, где количество *d* после десятичной точки равно точности, указанной для данного аргумента. При необходимости число округляется. Если точность не задана, выводится шесть цифр; если точность задана явным указанием ***0*** и # не указано, не выводится ни цифр, ни десятичной точки;
- e Аргумент типа ***float*** или ***double*** преобразуется в десятичный вид в научном стиле [-]*d.ddde+dd* или [-]*d.ddde-dd*, где до точки стоит одна цифра, а число цифр после точки равно указанной для данного аргумента точности. При необходимости число округляется. Если точность не задана, выводится шесть цифр; если точность задана явным указанием ***0***, и # не указано, не выводится ни цифр, ни точки;
- E Подобно *e*, но для обозначения показателя степени используется большое ***E***;
- g Аргументы типа ***float*** или ***double*** выводятся в стиле ***d***, ***f*** или ***e*** в зависимости от того, что из них дает максимальную точность, занимая минимум места;
- G Подобно *g*, но для обозначения показателя степени используется большое ***E***;
- c Выводится символьный аргумент. Нулевые символы игнорируются;
- s Аргумент должен быть строкой (указателем на символ), и символы из строки выводятся, пока не встретится нулевой символ, или пока не выведется число символов, определяемое точностью; однако если точность равна ***0*** или отсутствует, выведутся все символы до нулевого;
- p Аргумент должен быть указателем. Вид печатаемого представления зависит от реализации;
- u Целочисленный аргумент без знака преобразуется в десятичный вид;
- n Количество символов, выведенное на данный момент вызовом ***printf*** (), ***fprintf*** () или ***sprintf*** () и записанное в аргументе типа ***int***, на который указывает указатель на ***int***.

Неуказанная или недостаточная ширина поля ни в коем случае не приведет к урезанию вывода; дополнение символами-заполнителями имеет место, только если указанное поле шире действительного.

Вот более замысловатый пример:

```
char* line_format = "#line %d \"%s\"\n";
int line = 13;
char* file_name = "C++/main.c";
```

```
printf("int a;\n");
printf(line_format, line, file_name);
printf("int b;\n");
```

В результате будет выведено:

```
int a;
#line 13 "C++/main.c"
int b;
```

Использование **printf()** небезопасно в том смысле, что проверки типа не производится. Например, ниже представлен хорошо известный способ, как получить непредсказуемый вывод, вывести содержимое какого-то участка памяти или сделать что-нибудь еще хуже:

```
char x;
// ...
printf("недопустимый вводимый символ: %s", x);        // должно быть %c, а не %s
```

Однако функция **printf()** обеспечивает чрезвычайную гибкость, к тому же она знакома программистам, пишущим на C.

Подобным же образом **getchar()** предоставляет знакомый способ чтения вводимых символов:

```
int i;
while ((i=getchar())!= EOF) {      /* использование i */ }
```

Для того чтобы можно было сравнивать **EOF** как целое, значение, возвращаемое функцией **getchar()**, должно быть типа **int**, а не **char**.

Более подробно с вводом/выводом в стиле C можно ознакомиться в вашем справочном пособии по C или в книге Кернигана и Ричи «Язык программирования C» [Kernighan, 1988].

21.9. Советы

[1] Определяйте операторы << и >> для типов, определяемых пользователем, имеющих значения осмысленного текстового представления; § 21.2.3, § 21.3.5.

[2] Когда вы пишете выражения, содержащие операторы с низким приоритетом, пользуйтесь скобками; § 21.2.

[3] Чтобы добавить новые операторы << и >>, вам не нужно изменять **istream** и **ostream**; § 21.2.3.

[4] Вы можете определить функцию так, чтобы она вела себя, как виртуальная, основываясь на *втором* (или следующем) аргументе; § 21.2.3.1.

[5] Помните, что по умолчанию оператор >> пропускает символы-разделители; § 21.3.2.

[6] Пользуйтесь низкоуровневыми функциями ввода, такими как **get()** и **read()**, в основном при реализации высокоуровневых функций ввода; § 21.3.4.

[7] При использовании **get()**, **getline()** и **read()** будьте внимательны с определением конца ввода; § 21.3.4.

[8] Для установки флагов управления вводом/выводом предпочитайте манипуляторы; § 21.3.3, § 21.4, § 21.4.6.

[9] Пользуйтесь исключениями (только) для перехвата редких ошибок ввода/вывода; § 21.3.6.
[10] Связывайте потоки, используемые для интерактивного ввода и вывода; § 21.3.7.
[11] Чтобы сконцентрировать начальный и завершающий код для разных функций в одном месте, используйте часовых; § 21.3.8.
[12] Не пользуйтесь скобками после манипулятора без аргументов; § 21.4.6.2.
[13] При использовании стандартных манипуляторов не забывайте *#include<iomanip>*; § 21.4.6.2.
[14] Вы можете достичь эффекта (и эффективности) тернарного оператора, определив простую функцию-объект; § 21.4.6.3.
[15] Помните, что спецификация ***width*** применяется только к ближайшей операции ввода/вывода; § 21.4.4.
[16] Помните, что спецификации ***precision*** применяются ко всем последующим операциям вывода чисел с плавающей точкой; § 21.4.3.
[17] Для форматировании в памяти пользуйтесь потоками строк; § 21.5.3.
[18] Вы можете установить способ работы с файловым потоком; § 21.5.1.
[19] При расширении системы ввода/вывода четко различайте форматирование (***iostream***) и буферизацию (***streambuf***); § 21.1, § 21.6.
[20] Реализуйте нестандартные способы передачи значений через буфера потоков; § 21.6.4.
[21] Реализуйте нестандартные способы форматирования значений через операции с потоками; § 21.2.3, § 21.3.5.
[22] Вы можете изолировать и инкапсулировать обращения к пользовательскому коду при помощи пар функций; § 21.6.4.
[23] Вы можете до чтения воспользоваться функцией ***in_avail*** (), чтобы определить, не будет ли операция ввода заблокирована; § 21.6.4.
[24] Различайте простые операции, которые должны быть эффективными, и операции, реализующие тактику действий: первые делайте встроенными, а вторые виртуальными; § 21.6.4.
[25] Для отражения различий в культурных традициях пользуйтесь ***locale***; § 21.7.
[26] Если вы смешиваете ввод/вывод в стиле C и в стиле C++, пользуйтесь ***sync_with_stdio*** (); чтобы «разъединить» ввод/вывод в стиле C и в стиле C++ используйте вызов ***sync_with_stdio*** (***false***); § 21.8.
[27] При вводе/выводе в стиле C опасайтесь ошибок в типе; § 21.8.

21.10. Упражнения

1. (*1.5) Прочитайте файл, содержащий числа с плавающей точкой, скомбинируйте пары прочитанных чисел в комплексные числа и выведите их.
2. (*1.5) Определите тип ***Name_and_address*** (имя и адрес). Определите для него операторы << и >>. Скопируйте поток объектов типа ***Name_and_address***.
3. (*2.5) Скопируйте поток объектов типа ***Name_and_address***, и вставьте в него столько ошибок, сколько сможете придумать (например, ошибки формата и преждевременные концы строк). Обработайте эти ошибки таким образом, чтобы гарантировать, что функция копирования прочитает большинство правильно отфор-

матированных объектов ***Name_and_address***, хотя на вводе будут чередоваться «плохие» значения с «хорошими».

4. (*2.5) Замените формат ввода/вывода объектов ***Name_and_address***, чтобы сделать его менее чувствительным к ошибкам в формате.
5. (*2.5) Напишите несколько функций для запроса и чтения информации разного типа. Например: целые числа, числа с плавающей точкой, имена файлов, почтовые адреса, анкетные данные и т. п. Попытайтесь сделать их защищенными от ошибок.
6. (*1.5) Напишите программу, которая выводит: (а) все буквы нижнего регистра, (б) все буквы, (в) все буквы и цифры, (г) все символы, которые могут встретиться в идентификаторе C++ для вашей системы, (д) все знаки препинания, (е) коды всех управляющих символов, (ж) все символы-разделители, (з) коды всех символов-разделителей и наконец (и) все печатаемые символы.
7. (*2) Прочитайте последовательность текстовых строк (lines) в символьный буфер фиксированного размера. Удалите все символы-разделители и замените все буквы алфавита следующими за ними в алфавите (***z*** заменяется на ***a***, а ***9*** на ***0***). Выведите получившуюся строку.
8. (*3) Напишите «миниатюрную» систему потокового ввода/вывода, предоставляющую классы ***iostream***, ***ostream***, ***ifstream***, ***ofstream***, функции ***operator***<< () и ***operator***>> () для целых чисел и такие операции, как ***open*** () и ***close*** (), для файлов.
9. (*4) Реализуйте стандартную библиотеку ввода/вывода C (<***stdio.h***>) при помощи стандартной библиотеки ввода/вывода C++ (<***iostream***>).
10. (*4) Реализуйте стандартную библиотеку ввода/вывода C++ (<***iostream***>) при помощи стандартной библиотеки ввода/вывода C (<***stdio.h***>).
11. (*4) Реализуйте эти библиотеки на C и C++ так, чтобы ими было можно пользоваться одновременно.
12. (*2) Реализуйте класс, для которого оператор [] перегружен, чтобы выполнять чтение символов из указанной позиции файла.
13. (*3) Повторите § 21.10[12], но сделайте оператор [] применимым как для чтения, так и для записи. Подсказка: сделайте так, чтобы оператор [] возвращал объект «дескрипторного типа», для которого присваивание означало бы «записать в файл через дескриптор», а неявное преобразование в ***char*** означало бы «прочитать из файла через дескриптор».
14. (*2) Повторите § 21.10[14], но пусть оператор [] индексирует объекты произвольного типа, а не только символы.
15. (*3.5) Реализуйте версии ***istream*** и ***ostream*** для чтения и записи чисел в двоичном виде без преобразования их в символы. Рассмотрите достоинства и недостатки этого подхода по сравнению с подходом, основанным на символьном представлении.
16. (*3.5) Спроектируйте и реализуйте операцию ввода по шаблону. Для задания шаблона воспользуйтесь строками формата в стиле ***printf***. К одному и тому же введенному значению можно потенциально применить несколько шаблонов, перед тем как установить его действительный формат. Можно сделать класс для ввода по шаблону производным от ***istream***.
17. (*4) Спроектируйте (и реализуйте) значительно лучший вид шаблона для проверки соответствия с шаблоном. Определитесь с тем, что считать лучшим.
18. (*2) Определите манипулятор вывода ***based***, имеющий два аргумента — систему счисления и целое значение — и выводящий целое число в соответствии с системой счисления. Например, ***based*** (***2***, ***9***) должно вывести ***1001***.

19. (*2) Напишите манипуляторы, которые бы включали и выключали эхо при вводе символов.
20. (*2) Реализуйте ***Bound_form*** из § 21.4.6.3 для обычного набора встроенных типов.
21. (*2) Реализуйте ***Bound_form*** из § 21.4.6.3 так, чтобы операция вывода никогда не приводила к переполнению ее ширины (***width*** ()). Программисту нужно гарантировать, что вывод никогда (незаметным образом) не обрежется и будет достигнута указанная точность.
22. (*3) Реализуйте манипулятор ***encrypt*** (***k***), который бы гарантировал зашифрованный вывод в ***ostream*** с ключом ***k***. Обеспечьте аналогичный манипулятор ***decrypt*** () для ***istream***. Введите средства, отключающие шифрование потока, чтобы последующий ввод/вывод осуществлялся открытым текстом.
23. (*2) Проследите путь символа через вашу систему от клавиатуры до экрана в следующих инструкциях:

```
char c;
cin >> c;
cout << c << endl;
```

24. (*2) Измените функцию ***readints*** () (§ 21.3.6) для обработки всех исключений. Подсказка: «выделение ресурса есть инициализация».
25. (*2.5) Существует стандартный способ чтения, записи и представления дат в соответствии с национальными особенностями. Найдите его в документации на вашу реализацию и напишите маленькую программку чтения и записи дат при помощи этого механизма. Подсказка: ***struct tm***.
26. (*2.5) Определите некий поток вывода под именем ***ostrstream***, который было бы можно прикрепить к массиву символов (С-строке) аналогично тому, как ***ostringstream*** прикрепляется к ***string***. Однако не копируйте массив в или из ***ostrstream***. Поток ***ostrstream*** должен просто обеспечить способ для записи в массив, являющийся его аргументом. Это можно использовать для форматирования в памяти следующим образом:

```
char buf[message_size];
ostrstream ost (buf, message_size);
do_something (argument, ost);          // вывод в буфер buf через ost
cout << buf;                           // ost добавляет завершающий 0
```

Операция вроде ***do_something*** может писать в поток ***ost***, передавать ***ost*** своим подоперациям и т. д. при помощи стандартных операций вывода. Проверять, нет ли переполнения, не нужно, поскольку ***ost*** знает свои размеры и по заполнении перейдет в состояние ***fail*** (). Наконец, ***display*** () может записать сообщение в «настоящий» поток вывода. Этот прием может очень пригодиться в случаях, когда окончательная операция ***display*** () связана с записью в какое-то более замысловатое устройство, чем традиционное, ориентированное на вывод строк. Например, текст из ***ost*** может быть помещен в область на экране с фиксированными размерами. Аналогично определите класс ***istrstream*** как поток ввода строк, считывающий из заканчивающейся нулем строки символов. Интерпретируйте завершающий нуль как конец файла. Такие строковые потоки входили в первоначальную библиотеку потоков, и их часто можно найти в <***strstream.h***>.
27. (*2.5) Реализуйте манипулятор ***general*** (), который бы восстанавливал изначальный (универсальный) формат потока, так же как ***scientific*** () (§ 21.4.6.2) заставляет поток использовать научный формат.

22

Численные методы

Цель вычисления — понимание, а не числа.
— Р. В. Хэмминг

...но часто для учащихся
числа — лучший путь к пониманию.
— А. Ролстон

Введение — предельные значения — математические функции — ***valarray*** — векторные операции — срезы — ***slice_array*** — устранение временных массивов — ***gslice_array*** — ***mask_array*** — ***indirect_array*** — ***complex*** — обобщенные алгоритмы — случайные числа — советы — упражнения.

22.1. Введение

Редко удается написать реальную программу, в которой не производились бы какие-нибудь расчеты. Однако по большей части в коде не требуется каких-либо серьезных математических вычислений кроме арифметических. Данная глава представляет средства, которые стандартная библиотека предлагает тем, кто этим не ограничивается.

Ни C, ни C++ не проектировались специально для численных расчетов. Однако численные расчеты, как правило, возникают в контексте других задач — доступ к базам данных, работа с сетями, управление устройствами, графика, моделирование, финансовый анализ и т. д. И C++ стал привлекательным средством вычислений, которые выполняются как часть большой системы. Более того, численные методы вышли далеко за рамки простых циклов над векторами чисел с плавающей точкой. А там, где нужны более сложные структуры данных, становятся полезными мощные средства C++. Научные и инженерные расчеты все чаще стали проводиться с использованием C++. Поэтому появились специальные средства и приемы для поддержки таких расчетов. В этой главе описываются те части стандартной библиотеки, которые предназначены для поддержки численных методов. Здесь представлено несколько приемов решения проблем, традиционно возникающих, когда люди выражают математические вычисления на C++. Я не пытаюсь учить численным методам. Вычисления — это сама по себе захватывающая тема. Чтобы вникнуть в нее, вам нужно пройти хороший курс по численным методам или, по крайней мере, прочитать хороший учебник — а не просто руководство по языку и учебное пособие.

22.2. Предельные значения

Чтобы сделать с числами что-нибудь интересное, нам, как правило, нужно знать кое-что о свойствах встроенных типов, определяемое реализацией, а не фиксируемое правилами самого языка (§ 4.6). Например, чему равно самое большое *int*? Какое самое маленькое *float*? *double* округляется или обрезается при преобразовании во *float*? Сколько бит в *char*?

Ответы на эти вопросы дают специализации шаблона *numeric_limits*, представленного в заголовочном файле <*limits*>. Например:

```
void f(double d, int i)
{
    if (numeric_limits<unsigned char>::digits != 8) {
        // необычные байты (с числом бит не 8)
    }

    if (i < numeric_limits<short>::min() || numeric_limits<short>::max() < i) {
        // i не может храниться в short без потери точности
    }

    if (0<d && d<numeric_limits<double>::epsilon()) d = 0;

    if (numeric_limits<Quad>::is_specialized) {
        // информация о предельных значениях для типа Quad
    }
}
```

Каждая специализация дает необходимую информацию о типе ее аргумента. Таким образом, универсальный шаблон *numeric_limits* — это просто средство записи для набора констант и встроенных функций:

```
template<class T> class numeric_limits {
public:
    static const bool is_specialized = false;  // доступна ли информация
                                                // для numeric_limits<T>?
    // неинтересные установки по умолчанию
};
```

Реальная информация находится в специализациях. Каждая реализация стандартной библиотеки обеспечивает специализацию *numeric_limits* для каждого фундаментального типа (символьных типов, целых типов, типов для чисел с плавающей точкой и логических типов), но не для других возможных кандидатов — например, *void*, перечислений или библиотечных типов (таких как *complex<double>*).

Для интегральных типов, таких как *char*, представляет интерес только некоторая часть информации. Вот *numeric_limits<char>* для реализации, в которой *char* состоит из 8 битов и имеет знак:

```
class numeric_limits<char> {
public:
    static const bool is_specialized = true;    // да, у нас есть информация

    static const int digits = 7;      // число битов («двоичных цифр»), исключая знак
    static const bool is_signed = true;                 // char имеет знак
    static const bool is_integer = true;                // char — это интегральный тип
```

```
        inline static char min () throw () { return -128; }     // наименьшее значение
        inline static char max () throw () { return 127; }      // наибольшее значение

        // множество объявлений, не имеющих отношения к char
};
```

Отметим, что для целого со знаком количество цифр на 1 больше, чем количество бит, необходимое для хранения переменной данного типа.

Большинство членов шаблона ***numeric_limits*** предназначено для описания чисел с плавающей точкой. Например, вот как описывается возможная реализация для ***float***:

```
class numeric_limits<float> {
public:
        static const bool is_specialized = true;

        static const int radix = 2;             // основание показателя степени (целого)
        static const int digits = 24;           // количество (двоичных) цифр в мантиссе
        static const int digits10 = 6;          // количество десятичных цифр в мантиссе

        static const bool is_signed = true;
        static const bool is_integer = false;
        static const bool is_exact = false;

        static float min () throw () { return 1.17549435E-38F; }
        static float max () throw () { return 3.40282347E+38F; }

        static float epsilon () throw () { return 1.19209290E-07F; }
        static float round_error () throw () { return 0.5F; }

        static float infinity () throw () { return /*какое-то значение*/ ; }
        static float quiet_NaN () throw () { return /*какое-то значение*/ ; }
        static float signaling_NaN () throw () { return /*какое-то значение*/ ; }
        static float denorm_min () throw () { return min (); }

        static const int min_exponent = -125;
        static const int min_exponent10 = -37;
        static const int max_exponent = +128;
        static const int max_exponent10 = +38;

        static const bool has_infinity = true;
        static const bool has_quiet_NaN = true;
        static const bool has_signaling_NaN = true;
        static const float_denorm_style has_denorm
                = denorm_absent;                // элемент перечисления из <limits>
        static const bool has_denorm_loss = false;

        static const bool is_iec559 = true;     // соответствует IEC-559
        static const bool is_bounded = true;
        static const bool is_modulo = false;
        static const bool traps = true;
        static const bool tinyness_before = true;

        static const float_round_style round_style
                = round_to_nearest;             // элемент перечисления из <limits>
};
```

Отметим, что ***min*** () — это самое маленькое *положительное* нормализованное число, а ***epsilon*** — самое маленькое положительное число с плавающей точкой такое, что *1+epsilon−1* больше нуля.

Определяя некий скалярный тип в духе встроенных типов, полезно ввести соответствующую специализацию ***numeric_limits***. Например, если бы я написал тип ***Quad*** для чисел с учетверенной точностью, или если бы поставщик ввел целые числа с расширенной точностью ***long long***, пользователь был бы в праве ожидать, что для них существуют ***numeric_limits<Quad>*** и ***numeric_limits<long long>***.

Можно представить себе специализацию ***numeric_limits***, которая описывала бы свойства типов, определяемых пользователем, мало похожих на числа с плавающей точкой. В таких случаях для описания свойств типа обычно лучше пользоваться общей техникой, чем специализировать ***numeric_limits*** с не входящими в стандарт свойствами.

Значения с плавающей точкой представлены как встроенные функции. Однако, целые значения в ***numeric_limits*** должны быть представлены в такой форме, которая позволила бы использовать их в константных выражениях. Это значит, что они должны иметь инициализаторы внутри класса (§ 10.4.6.2). Если вы пользуетесь для этого статическими константными членами, а не перечислениями, не забудьте определить эти статические члены.

22.2.1. Макросы для предельных значений

Для описания свойств целых чисел C++ унаследовал из C макросы. Эти макросы находятся в ***<climits>*** и ***<limits.h>***, они имеют имена наподобие ***CHAR_BIT*** и ***INT_MAX***. Подобным образом в ***<cfloat>*** и ***<float.h>*** определены макросы, описывающие свойства чисел с плавающей точкой. Их имена выглядят как ***DBL_MIN_EXP***, ***FLT_RADIX*** и ***LDBL_MAX***.

Как всегда, макросов лучше стараться избегать.

22.3. Стандартные математические функции

Заголовочные файлы ***<cmath>*** и ***<math.h>*** предоставляют то, что обычно называют «обычными математическими функциями»:

```
double abs (double);          // абсолютное значение; в C отсутствует;
                              // то же, что fabs()
double fabs (double);         // абсолютное значение

double ceil (double d);       // наименьшее целое, не меньшее d
double floor (double d);      // наибольшее целое, не большее d

double sqrt (double d);       // квадратный корень из d,
                              // d должно быть неотрицательным

double pow (double d, double e);  // d в степени e; ошибка, если d==0 и e<=0,
                                  // или если d<0 и e не целое
double pow (double d, int i);     // d в степени i; в C отсутствует

double cos (double);          // косинус
double sin (double);          // синус
double tan (double);          // тангенс

double acos (double);         // арккосинус
double asin (double);         // арксинус
double atan (double);         // арктангенс
double atan2 (double x, double y);  // atan(x/y)

double sinh (double);         // гиперболический синус
double cosh (double);         // гиперболический косинус
```

```
double tanh (double);          // гиперболический тангенс

double exp (double);           // экспонента, основание e
double log (double d);         // натуральный (с основанием e) логарифм,
                               // d должно быть положительным
double log10 (double d);       // десятичный логарифм,
                               // d должно быть положительным

double modf (double d, double* p);  // возвращает дробную часть d,
                                    // целая часть помещается в *p
double frexp (double d, int* p);    // находит x в [.5, 1) и y такие, что
                                    // d=x*pow(2,y), возвращает x и сохраняет y в *p
double fmod (double d, double m);   // остаток от деления d на m, того же знака, что d
double ldexp (double d, int i);     // d*pow(2,i)
```

Кроме того, <*cmath*> и <*math.h*> предоставляют эти функции для аргументов типа *float* и *long double*.

Там, где результат неоднозначен — как в случае с функцией *asin* () — возвращается значение, ближайшее к *0*. Функция *acos* () возвращает неотрицательный результат.

Ошибки индицируются установкой *errno* из <*cerrno*> в состояние *EDOM* для ошибок, связанных с выходом из области определения, и в состояние *ERANGE* — для ошибок выхода за пределы диапазона. Например:

```
void f()
{
    errno=0;    // очистка старого состояния, индицирующего ошибку
    sqrt (-1);
    if (errno == EDOM)
        cerr << "sqrt () не определена для отрицательных аргументов";
    pow (numeric_limits<double>::max (), 2);
    if (errno == ERANGE)
        cerr << "результат pow () слишком велик для типа double";
}
```

Так уж исторически сложилось, что несколько математических функций находятся не в <*cmath*>, а в заголовочном файле <*cstdlib*>:

```
int abs (int);          // абсолютное значение
long abs (long);        // абсолютное значение (в C отсутствует)
long labs (long);       // абсолютное значение

struct div_t { implementation_defined quot, rem; }
struct ldiv_t { implementation_defined quot, rem; }

div_t div (int n, int d);          // деление n на d, возвращает (частное, остаток)
ldiv_t ldiv (int n, int d);        // деление n на d, возвращает (частное, остаток)
                                   // (в C отсутствует)
ldiv_t ldiv (long int n, long int d);  // деление n на d, возвращает (частное, остаток)
```

22.4. Векторная арифметика

Значительная часть работы с числами связана с относительно простыми одномерными векторами чисел с плавающей точкой. В частности, такие вектора хорошо поддерживаются в архитектурах высокопроизводительных машин, для таких векторов написаны широко распространенные библиотеки, и во многих областях деятельности

придается большое значение агрессивной оптимизации программ, применяющих подобные вектора. Поэтому стандартная библиотека предоставляет вектор — который называется *valarray* — разработанный специально для ускорения распространенных численных операций над векторами.

Глядя на свойства вектора *valarray* следует помнить, что они рассчитаны на сравнительно низкоуровневые быстрые вычисления. В частности, главным критерием проектирования считалась не легкость применения, а эффективность использования высокопроизводительных компьютеров на основе приемов агрессивной оптимизации. Если ваша главная цель — достижение гибкости и универсальности, а не эффективности, то, возможно, вам лучше строить вычисления на основе контейнеров из глав 16 и 17, чем стараться втиснуться в рамки простого, эффективного и подчеркнуто традиционного вектора *valarray*.

Кто-нибудь может сказать, что *valarray* следовало бы назвать *vector* (вектор), поскольку он является традиционным математическим вектором, а *vector* (§ 16.3) следовало бы назвать *array* (массив), однако терминология развивалась иначе. *valarray* — это вектор, оптимизированный для численных расчетов, а *vector* — это гибкий контейнер, разработанный для хранения объектов различного типа и манипулирования с ними, массив же — это низкоуровневый встроенный тип.

Тип *valarray* поддерживается четырьмя вспомогательными типами, служащими для определения подмножеств *valarray*:

- *slice_array* и *gslice_array* представляют понятие срезов (slice) (§ 22.4.6, § 22.4.8);
- *mask_array* определяет подмножество, по-разному помечая элементы вектора (§ 22.4.9);
- *indirect_array* содержит не сами элементы, а их индексы (§ 22.4.10).

22.4.1. Конструирование valarray

Тип *valarray* и связанные с ним средства определены в пространстве имен *std* и представлены в <*valarray*>:

```
template<class T> class std::valarray {
        // представление
public:
        typedef T value_type;

        valarray ();                                  // valarray с размером size()==0
        explicit valarray (size_t n);                 // n элементов со значением T()
        valarray (const T& val, size_t n);            // n элементов со значением val
        valarray (const T* p, size_t n);              // n элементов со значениями p[0], p[1], ...
        valarray (const valarray& v);                 // копия v

        valarray (const slice_array<T>&);             // см. § 22.4.6
        valarray (const gslice_array<T>&);            // см. § 22.4.8
        valarray (const mask_array<T>&);              // см. § 22.4.9
        valarray (const indirect_array<T>&);          // см.§ 22.4.10

        ~valarray ();

        //...
};
```

Этот набор конструкторов позволяет нам инициализировать *valarray*, используя числовые массивы вспомогательных типов или отдельные значения. Например:

```
valarray<double> v0;                    // занимаем место; мы можем присвоить
                                        // что-нибудь v0 позже
valarray<float> v1 (1000);              // 1000 элементов со значением float()==0.0F

valarray<int> v2 (-1, 2000);            // 2000 элементов со значением -1
valarray<double> v3 (100, 9.8064);      // ошибка: размер valarray задан
                                        // числом с плавающей точкой

valarray<double> v4=v3;                 // v4 имеет v3.size() элементов
```

В двухаргументных конструкторах значение указывается перед числом элементов. В этом отличие от общепринятой записи для стандартных контейнеров (§ 16.3.4).

При копировании размер получающегося ***valarray*** определяется числом элементов в аргументе, который должен быть скопирован.

Большинство программ нуждается в данных, введенных или взятых из таблицы, это поддерживается конструктором, который копирует элементы из встроенного массива. Например:

```
const double vd[] = { 0, 1, 2, 3, 4 };
const int vi[] = { 0, 1, 2, 3, 4 };

valarray<double> v3 (vd, 4);    // 4 элемента: 0, 1, 2, 3
valarray<double> v4 (vi, 4);    // ошибка в типе: vi не является указателем на double
valarray<double> v5 (vd, 8);    // не определено: в инициализаторе слишком
                                // мало элементов
```

Такая форма инициализации очень важна, поскольку есть много численных программ, которые выдают данные в виде больших (обычных) массивов.

Тип ***valarray*** и его вспомогательные средства разработаны для ускорения вычислений. Это отражается в некоторых ограничениях для пользователя и в жестких требованиях для разработчиков. В сущности, разработчику ***valarray*** приходится использовать все мыслимые приемы оптимизации. Например, операции можно делать встроенными; считается, что операции с ***valarray*** не влияют ни на какие объекты (кроме, конечно, своих явных аргументов). Также считается, что массивы ***valarray*** не имеют альтернативных имен, и при сохранении базовой семантики допускается введение вспомогательных типов и устранение временных массивов. Таким образом, объявления в <***valarray***> могут выглядеть не так, как здесь (и в стандарте), но они должны обеспечивать те же самые операции с тем же самым смыслом для всех программ, написанных в рамках правил. В частности, элементы ***valarray*** должны иметь обычную семантику копирования (§ 17.1.4).

22.4.2. Индексация и присваивание для valarray

Индексация ***valarray*** применяется как для того, чтобы обращаться к отдельным членам, так и для того чтобы получать подмассивы:

```
template<class T> class valarray {
public:
    // ...
    valarray& operator= (const valarray& v);    // копирование v
    valarray& operator= (const T& val);         // присваивание val каждому элементу

    T operator[] (size_t) const;
    T& operator[] (size_t);
```

```
        valarray operator[] (slice) const;                    // см. § 22.4.6
        slice_array<T> operator[] (slice);

        valarray operator[] (const gslice&) const;            // см. § 22.4.8
        gslice_array<T> operator[] (const gslice&);

        valarray operator[] (const valarray<bool>&) const;    // см. § 22.4.9
        mask_array<T> operator[] (const valarray<bool>&);

        valarray operator[] (const valarray<size_t>&) const;  // см. § 22.4.9
        indirect_array<T> operator[] (const valarray<size_t>&);

        valarray& operator= (const slice_array<T>&);          // см. § 22.4.6
        valarray& operator= (const gslice_array<T>&);         // см. § 22.4.8
        valarray& operator= (const mask_array<T>&);           // см. § 22.4.9
        valarray& operator= (const indirect_array<T>&);       // см. § 22.4.10

        // ...
};
```

Массив *valarray* можно присвоить другому того же размера. Как и предполагалось, *v1=v2* копирует каждый элемент *v2* в соответствующую позицию объекта *v1*. Если вектора имеют разный размер, результат присваивания не определен. Поскольку *valarray* придуман для оптимизации скорости, было бы наивно предполагать, что присваивание объекта *valarray* неправильного размера приведет к легко выявляемой ошибке (такой как исключение) или какому-либо другому «осмысленному» поведению.

Кроме такого общепринятого присваивания, массиву *valarray* можно присвоить скаляр. Например, *v=7* присваивает каждому элементу *v* значение *7*. Это может показаться странным; лучше всего понимать это как иногда применяемый вырожденный случай оператора присваивания (§ 22.4.3).

Индексация целым числом ведет себя обычным образом и не выполняет проверки диапазона.

Кроме того для выделения отдельных элементов, индексация *valarray* предоставляет четыре способа извлечения подмассивов (§ 22.4.6). И наоборот, присваивания (и конструкторы § 22.4.1) принимают такие подмассивы в качестве операндов. Набор операций присваивания для массивов *valarray* избавляет от необходимости преобразовывать вспомогательные массивы, такие как *slice_array*, в *valarray* перед их присваиванием. Для обеспечения эффективности подобным же образом в реализации могут вводиться другие векторные операции, такие как + и *. Кроме того, для векторных операций существует много действенных приемов оптимизации, в том числе над срезами типа *slice* и другими вспомогательными векторными типами.

22.4.3. Операции-члены

Вводятся очевидные, а также и не совсем очевидные функции-члены:

```
template<class T> class valarray {
public:
        // ...

        valarray& operator*= (const T& arg);    // v[i]*=arg для каждого элемента
        // аналогично: /=, %=, +=, -=, ^=, &=, |=, <<= и >>=

        T sum () const;                         // сумма элементов, использует += для сложения
```

```
    T min () const;                 // наименьшая величина, использует < для сравнения
    T max () const;                 // наибольшая величина, использует < для сравнения

    valarray shift (int i) const;   // логический сдвиг (влево для i<0 и вправо для i>0)
    valarray cshift (int i) const;  // цикический сдвиг (влево для i<0 и вправо для i>0)

    valarray apply (T f(T)) const;  // result[i]=f(v[i]) для каждого элемента
    valarray apply (T f(const T&)) const;

    valarray operator- () const;    // result[i]=-v[i] для каждого элемента
    // аналогично: +, ~, !

    size_t size () const;                       // число элементов
    void resize (size_t n, const T& val = T ()); // n элементов со значением val
};
```

Если *size_t ==0*, величины *sum ()*, *min ()* и *max ()* неопределены.

Например, если *v* — это массив *valarray*, то его можно масштабировать следующим образом: *v*=.2* или *v/=1.3*. То есть применение к вектору скаляра означает применение этого скаляра к каждому элементу вектора. Обычно, легче оптимизировать *=, чем комбинацию * и = (§ 11.3.1).

Отметим, что неприсваивающие операции конструируют новый *valarray*. Например:

```
double incr (double d) { return d+1; }          // инкремент

void f (valarray<double>& v)
{
    valarray<double> v2 = v.apply (incr);       // производит
                                                // инкрементированный val_array
}
```

Значение *v* не изменяется. К сожалению, *apply ()* не принимает в качестве аргумента (§ 22.9[1]) объекты-функции (§ 18.4).

Функции логического и циклического сдвига *shift ()* и *cshift ()* возвращают новый *valarray* с соответствующим образом сдвинутыми элементами, а исходный объект остается без изменений. Например, циклический сдвиг *v2=v.cshift (n)* выдаст *v[2]* такой, что *v2[i]==v[(i+n)%v.size ()]*. Логический сдвиг *v3=v.shift (n)* выдаст *v[3]*, такой, что *v3[i]* равен *v[i+n]*, если *i+n* — допустимый индекс для *v*, а иначе *v3[i]* примет значение по умолчанию. Считается, что и *shift ()*, и *cshift ()* сдвигают влево, если им задан положительный аргумент, и вправо — если отрицательный. Например:

```
void f ()
{
    int alpha[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

    valarray<int> v (alpha, 8);     // 1, 2, 3, 4, 5, 6, 7, 8
    valarray<int> v2 = v.shift (2); // 3, 4, 5, 6, 7, 8, 0, 0
    valarray<int> v3 = v<<2;        // 4, 8, 12, 16, 20, 24, 28, 32
    valarray<int> v4 = v.shift (-2);// 0, 0, 1, 2, 3, 4, 5, 6
    valarray<int> v5 = v>>2;        // 0, 0, 0, 1, 1, 1, 1, 2
    valarray<int> v6 = v.cshift (2);// 3, 4, 5, 6, 7, 8, 1, 2
    valarray<int> v7 = v.cshift (-2);// 7, 8, 1, 2, 3, 4, 5, 6
}
```

Для массивов ***valarray*** операторы << и >> — это операторы битового сдвига, а не операторы сдвига элементов или ввода/вывода (§ 22.4.4). Следовательно, чтобы сдвигать биты внутри элементов интегрального типа, можно пользоваться операторами <<= и >>=. Например:

```
void f(valarray<int> vi, valarray<double> vd)
{
    vi <<= 2;    // vi[i]<<=2 для всех элементов vi
    vd <<= 2;    // ошибка: для значений с плавающей точкой сдвиг неопределен
}
```

Размер массива ***valarray*** можно изменить, однако ***resize*** () — *не* та операция, которая предназначена для превращения ***valarray*** в структуру данных, способную динамически расти, как вектора или строки. Для ***valarray resize*** () — это операция повторной инициализации, которая заменяет существующее содержимое ***valarray*** набором значений по умолчанию. Старые значения утрачиваются.

Часто ***valarray*** после вызова функции ***resize*** () становится тем, что мы создаем в качестве пустого вектора. Рассмотрим, как можно инициализировать ***valarray*** из ввода:

```
void f()
{
    int n=0;
    cin >> n;                                   // читаем размер массива
    if (n<=0) error("неверные границы массива");

    valarray<double> v(n);                      // создаем массив подходящего размера
    int i=0;
    while (i<n && cin>>v[i++]);                 // заполняем массив
    if (i!=n) error("введено слишком мало элементов");

    // ...
}
```

Если мы захотим обработать ввод в отдельной функции, это можно сделать так:

```
void initialize_from_input(valarray<double>& v)
{
    int n=0;
    cin >> n;                                   // читаем размер массива
    if (n<=0) error("неверные границы массива");
    v.resize(n);                                // создаем массив подходящего размера
    int i=0;
    while (i<n && cin>>v[i++]);                 // заполняем массив
    if (i!=n) error("введено слишком мало элементов");
}

void g()
{
    valarray<double> v;                         // создаем массив по умолчанию
    initialize_from_input(v);                   // инициализация v с правильным размером
                                                // и допустимыми элементами
    // ...
}
```

Это позволяет избежать копирования больших объемов данных.

Если нам нужно, чтобы массив ***valarray*** с ценными данными динамически рос, мы должны воспользоваться временным массивом для хранения:

```
void grow(valarray<int>& v, size_t n)
{
    if (n<=v.size()) return;
    valarray<int> tmp(n);              // временный массив; n элементов по умолчанию
    copy(&v[0], &v[v.size()], &tmp[0]); // копирующий алгоритм из § 18.6.1
    v.resize(n);
    copy(&tmp[0], &tmp[v.size()], &v[0]);
}
```

Не предполагается, что массивы ***valarray*** будут использоваться таким образом. Тип ***valarray*** рассчитан на то, что после инициализации размер массива не изменяется.

Элементы ***valarray*** образуют последовательность; то есть $v[0]..v[n-1]$ находятся в памяти друг за другом. Это означает, что ***T**** является для ***valarray*** <***T***> итератором с произвольным доступом (§ 19.2.1), так что можно пользоваться стандартными алгоритмами, такими как ***copy*** (). Однако более соответствует духу ***valarray*** выражение копирования через присваивания и подмассивы:

```
void grow2(valarray& v, size_t n)
{
    if (n<=v.size()) return;

    valarray<int> tmp = v;
    slice s(0, v.size(), 1);          // подмассив из v.size() элементов (см. § 22.4.5)

    v.resize(n);                      // при изменении размера значения не сохраняются
    v[s] = tmp;                       // восстанавливаем значения первого параметра
}
```

Если по какой-либо причине ввод данных организован так, что для того, чтобы узнать необходимый размер массива, приходится пересчитывать элементы, то обычно лучше занести введенные данные сначала в ***vector*** (§ 16.3.5), а потом скопировать его элементы в массивы ***valarray***.

22.4.4. Операции

Введены обычные бинарные операторы и математические функции:

```
template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T*, const valarray<T>&);

// аналогично: /, +, –, ^, &, |, <<, >>, &&, ||, %, ==, !=, <, >, <=, >=, atan2 и pow

template<class T> valarray<T> abs(const valarray<T>&);
// аналогично: acos, asin, atan, cos, cosh, exp, log, log10, sin, sinh, sqrt, tan и tanh
```

Бинарные операции определены для массивов ***valarray*** и для комбинации ***valarray*** с (соответствующим ему) скалярным типом. Например:

```
void f(valarray<double>& v, valarray<double>& v2, double d)
```

```
{
    valarray<double> v3 = v*v2;          // v3[i]=v[i]*v2[i] для всех i
    valarray<double> v4 = v*d;           // v4[i]=v[i]*d для всех i
    valarray<double> v5 = d*v2;          // v5[i]=d*v2[i] для всех i

    valarray<double> v6 = cos(v);        // v6[i]=cos(v[i]) для всех i
}
```

Все эти векторные операции применяют свои операции к каждому элементу операнда (операндов) способом, показанным выше для * и *cos* (). Естественно, операция может использоваться, только если соответствующая операция определена для типа аргумента шаблона. Иначе при попытке специализировать шаблон компилятор выдаст сообщение об ошибке (§ 13.5).

Там, где результат имеет тип *valarray*, длина получающегося массива та же, что и у операндов типа *valarray*. Если их длины неодинаковы, результат бинарной операции над двумя массивами *valarray* не определен.

Довольно любопытно, что для *valarray* не предоставляется никаких операций ввода/вывода (§ 22.4.3); << и >> — это операции сдвига. Однако версии << и >> для ввода/вывода *valarray* легко определяются (§ 22.9[5]).

Отметим, что эти операции над *valarray* возвращают новый *valarray*, а не модифицируют свои операнды. Это может оказаться дорого (а может и нет, если применяется агрессивная оптимизация (см., например, § 22.4.7)).

Все эти операторы и математические функции над массивами *valarray* можно применять к *slice_array* (§ 22.4.7), *gslice_array* (§ 22.4.8), *mask_array* (§ 22.4.9), *indirect_array* (§ 22.4.10) и их комбинациям. Однако при реализации допускается перед выполнением требуемой операции преобразовывать операнды не типа *valarray* в *valarray*.

22.4.5. Срезы

Срез (slice) — это абстракция, которая позволяет нам эффективно манипулировать вектором как матрицей произвольной размерности. Это ключевое понятие для векторов Fortran и библиотеки BLAS (Basic Linear Algebra Subprograms — библиотека основных подпрограмм линейной алгебры), которое является основой для многих численных расчетов. В сущности, срез — это каждый *n*-й элемент некоторой части *valarray*:

```
class std::slice {
    // начальный индекс, длина и шаг
public:
    slice();
    slice(size_t start, size_t size, size_t stride);

    size_t start() const;          // индекс первого элемента
    size_t size() const;           // число элементов
    size_t stride() const;         // n-й элемент находится
                                   // по индексу: (индекс первого)+ n*stride()
};
```

Шаг (stride) — это расстояние (число элементов) между двумя элементами среза. Таким образом, срез описывает последовательность целых чисел. Например:

```
// отображение i в соответствующий индекс
size_t slice_index (const slice& s, size_t i)
{
    return s.start ()+i*s.stride ();
}

// печать элементов среза s
void print_seq (const slice& s)
{
    for (size_t i=0; i<s.size (); i++) cout << slice_index (s, i) << ",";
}

void f ()
{
    print_seq (slice (0, 3, 4));      // строка 0
    cout << ",";
    print_seq (slice (1, 3, 4));      // строка 1
    cout << ",";
    print_seq (slice (0, 4, 1));      // столбец 0
    cout << ",";
    print_seq (slice (4, 4, 1));      // столбец 1
}
```

выведет ***0 4 8 , 1 5 9 , 0 1 2 3 , 4 5 6 7.***

Другими словами, срез описывает отображение неотрицательных целых в индексы. Число элементов (*size* ()) не влияет на это отображение (адресацию), а просто позволяет нам найти конец последовательности. Это отображение можно использовать для моделирования двухмерного массива внутри одномерного (такого как *valarray*) эффективным, универсальным и довольно удобным способом. Рассмотрим матрицу 3 на 4, как мы часто представляем ее (§ B.7):

00	01	02
10	11	12
20	21	22
30	31	32

Следуя правилам языка Fortran, мы можем расположить ее в памяти так:

0				4				8			
00	10	20	30	01	11	21	31	02	12	22	32
0	1	2	3								

В C++ массивы располагаются *не* так (см. § B.7). Однако мы должны иметь возможность представить концепцию через ясный и логичный интерфейс, а затем выбрать представление, соответствующее налагаемым задачей ограничениям. Здесь я выбрал расположение в стиле Fortran, чтобы облегчить взаимодействие с программами численных расчетов, следующих этому соглашению. Однако я не зашел настолько далеко, чтобы начать индексацию с *1*, а не с *0*; это я оставил в качестве упражнения (§ 22.9[9]). Многие вычисления делаются и будут делаться с применением нескольких языков и с использованием разнообразных библиотек. Часто важно иметь возможность манипу-

лировать данными в разнообразных форматах, определенных этими библиотеками и стандартами языков.

Строку ***x*** можно описать как ***slice*** (***x***, ***3***, ***4***). То есть первым элементом строки ***x*** является ***x***-й элемент вектора, следующим — ***x+4***-й элемент и т. д., а в каждой строке 3 элемента. На рисунке ***slice*** (***0***, ***3***, ***4***) описывает строку с ***00***, ***01*** и ***02***.

Столбец ***y*** можно описать как ***slice*** (***4*y***, ***4***, ***1***). То есть первым элементом столбца ***y*** является ***4*y***-й элемент вектора, следующим — (***4*y+1***)-й и т. д.; в каждом столбце 4 элемента. На рисунке ***slice*** (***0***, ***4***, ***1***) описывает столбец с ***00***, ***10***, ***20*** и ***30***.

Кроме моделирования двухмерного массива, срез может описать еще много других последовательностей. Это довольно универсальный способ для определения простых последовательностей. Далее это понятие будет рассмотрено в § 22.4.8.

Один из способов представить себе срез — рассматривать его как некий странного вида итератор: срез позволяет нам описать последовательность индексов для массива ***valarray***. На этой основе мы можем построить настоящий итератор:

```
template<class T> class Slice_iter {
        valarray<T>* v;
        slice s;
        size_t curr;                    // индекс текущего элемента

        T& ref(size_t i) const { return  (*v)[s.start()+i*s.stride()]; }
public:
        Slice_iter(valarray<T>* vv, slice ss) : v(vv), s(ss), curr(0) { }

        Slice_iter end()
        {
                Slice_iter t = *this;
                t.curr = s.size();      // позиция элемента, следующего за последним
                return t;
        }

        Slice_iter& operator++() { curr++; return *this; }
        Slice_iter& operator++(int) { Slice_iter t = *this; curr++; return t; }

        T& operator[](size_t i) { return ref(curr=i); }          // индексация в стиле C
        T& operator()(size_t i) { return ref(curr=i); }          // индексация в стиле Fortran
        T& operator*() { return ref(curr); }                     // текущий элемент

        //...
};
```

Поскольку срез имеет размер, мы можем даже обеспечить проверку диапазона. Здесь я воспользовался ***slice::size***(), чтобы ввести операцию ***end***(), предоставляющую итератор для элемента, следующего за последним в массиве ***slice***.

Поскольку срез может описывать либо строку, либо столбец, ***Slice_iter*** позволяет нам проходить через ***valarray*** или по строкам, или по столбцам.

Чтобы итератор ***Slice_iter*** был полезным, нужно определить операторы ==, != и <.

```
template<class T> bool operator==(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
        return p.curr==q.curr && p.s.stride() == q.s.stride() && p.s.start() == q.s.start();
}
```

```
template<class T> bool operator!= (const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return !(p==q);
}

template<class T> bool operator< (const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return p.curr<q.curr && p.s.stride() == q.s.stride() && p.s.start() == q.s.start();
}
```

22.4.6. Массив slice_array

Из массива ***valarray*** и среза мы можем построить нечто похожее на ***valarray***, но на самом деле являющееся всего лишь способом обратиться к подмножеству массива, описываемому срезом. Такой ***slice_array*** определяется следующим образом:

```
template<class T> class std::slice_array {
public:
    typedef T value_type;

    void operator= (const valarray<T>&);
    void operator= (const T& v);                    // присваивает v каждому элементу
    void operator*= (const T& val);                 // v[i]*=val для каждого элемента
    // аналогично: /=, %=, +=, -=, ^=, &=, |=, <<=, >>=

    ~slice_array();
private:
    slice_array();                                  // во избежание конструирования
    slice_array(const slice_array&);                // во избежание копирования
    slice_array& operator= (const slice_array&);    // во избежание копирования

    valarray<T>* p;                                 // определяемое в реализации
                                                    // представление
    slice s;
};
```

Пользователь не может напрямую создавать ***slice_array***. Вместо этого он индексирует массив ***valarray***, чтобы создать ***slice_array*** для данного среза. Когда ***slice_array*** инициализирован, все обращения к нему косвенно пересылаются на ***valarray***, для которого был создан данный ***slice_array***. Например, мы можем создать нечто, что представляет каждый второй элемент массива:

```
void f(valarray<double>& d)
{
    slice_array<double>& v_even = d[slice(0, d.size()/2+d.size()%2, 2)];
    slice_array<double>& v_odd = d[slice(1, d.size()/2, 2)];

    v_even *= v_odd;        // попарно перемножает элементы и сохраняет
                            // результат в четных элементах
    v_odd = 0;              // присваивает 0 каждому нечетному элементу d
}
```

Запрет на копирование ***slice_array*** необходим, чтобы разрешить оптимизацию, опирающуюся на отсутствие альтернативных имен. Это может оказаться очень стеснительно. Например:

```
slice_array<double> row (valarray<double>& d, int i)
{
        slice_array<double> v=d[slice (0, 2, d.size ()/2)];    // ошибка: попытка скопировать
        return d[slice (i%2, i, d.size ()/2)];                 // ошибка: попытка скопировать
}
```

Часто разумной альтернативой копированию ***slice_array*** является копирование срезов ***slice***.

Срезами можно пользоваться для выражения разнообразных подмножеств массива. Например, мы могли бы воспользоваться срезами, чтобы манипулировать непрерывными подмассивами:

```
inline slice sub_array (size_t first, size_t count)      // [first:first+count[
{
        return slice (first, count, 1);
}

void f (valarray<double>& v)
{
        size_t sz = v.size ();
        if (sz<2) return;
        size_t n = sz/2;
        size_t n2 = sz-n;

        valarray<double> half1 (n);
        valarray<double> half2 (n2);

        half1 = v[sub_array (0, n)];                     // копирование первой половины v
        half2 = v[sub_array (n, n2)];                    // копирование второй половины v

        // ...
}
```

Стандартная библиотека не обеспечивает класса матриц. Назначение массивов ***valarray*** и срезов — обеспечить инструмент для построения матриц, оптимизированных для разнообразных потребностей. Давайте рассмотрим, как при помощи ***valarray*** и ***slice_array*** мы могли бы реализовать простую двухмерную матрицу:

```
class Matrix {
        valarray<double>* v;
        size_t d1, d2;
public:

        Matrix (size_t x, size_t y);          // отметим: нет конструктора по умолчанию
        Matrix (const Matirx&);
        Matrix& operator= (const Matirx&);
        ~Matrix ();

        size_t size () const { return d1*d2; }
        size_t dim1 () const { return d1; }
        size_t dim2 () const { return d2; }

        Slice_iter<double> row (size_t i);
        Cslice_iter<double> row (size_t i) const;
        Slice_iter<double> column (size_t i);
```

```
        Cslice_iter<double> column (size_t i) const;

        double& operator () (size_t x, size_t y);          // индексация в стиле Fortran
        double operator () (size_t x, size_t y) const;

        Slice_iter<double> operator () (size_t i) { return row (i); }
        Cslice_iter< double> operator () (size_t i) const { return row (i); }

        Slice_iter<double> operator[] (size_t i) { return row (i) };     // индексация в стиле C
        Slice_iter<double> operator[] (size_t i) const { return row (i) }

        Matrix& operator*= (double);
        Matrix& operator= (const Matrix&);

        valarray<double>& array () { return *v; }
};
```

Здесь матрица ***Matrix*** представляется массивом ***valarray***. Мы «добавляем» размерность к массиву с помощью срезов. При необходимости мы можем рассматривать это представление как одномерное, двухмерное, трехмерное и т. д. тем же способом, как мы ввели двухмерное представление через строки и столбцы. ***Slice_iter*** используются для того, чтобы обойти запрет копировать массивы ***slice_array***. Я не смог бы возвратить *slice_array*:

```
slice_array row (size_t i) { return (*v) [slice (i, d2, d1)]; }          // ошибка
```

поэтому я возвращаю не ***slice_array***, а итератор, содержащий указатель на ***valarray*** и собственно срез.

Чтобы выразить различие между срезом для константной матрицы и неконстантной матрицы, нам нужен дополнительный класс — «итератор для среза констант»:

```
inline Slice_iter<double> Matrix::row (size_t i)
{
        return Slice_iter<double> (v, slice (i, d2, d1));
}

inline Cslice_iter<double> Matrix::row (size_t i) const
{
        return Cslice_iter<double> (v, slice (i, d2, d1));
}

inline Slice_iter<double> Matrix::column (size_t i)
{
        return Slice_iter<double> (v, slice (i*d2, d2, 1));
}

inline Cslice_iter<double> Matrix::column (size_t i) const
{
        return Cslice_iter<double> (v, slice (i*d2, d2, 1));
}
```

Определение итератора ***Cslice_iter*** идентично определению ***Slice_iter*** за исключением того, что первый возвращает константные ссылки на элементы своего среза.

Остальные операции-члены довольно тривиальны:

```
Matrix::Matrix (size_t x, size_t y)
```

```
{
    // проверка, что x и y имеют смысл
    d1 = x;
    d2 = y;
    v = new valarray<double> (x*y);
}
double& Matrix::operator () (size_t x, size_t y)
{
    return row (x) [y];
}
double mul (Cslice_iter<double>& v1, const valarray<double>& v2)
{
    double res=0;
    for (size_t i = 0; i<v1.size; i++) res+=v1[i]*v2[i];
    return res;
}
valarray<double> operator* (consr Matrix& m, const valarray<double>& v)
{
    valarray<double> res (m, dim1 ());
    for (size_t i = 0; i<m.dim1 (); i++) res [i]=mul (m.row (i), v);
    return res;
}
Matrix& Matrix::operator*= (double d)
{
    (*v) *= d;
    return *this;
}
```

Для индексации матрицы я ввел оператор `(i, j)`, потому что `()` — это один оператор, и в сообществе вычислителей такое обозначение многим хорошо знакомо. Понятие строки обеспечивает более знакомое (в сообществе C и C++) обозначение `[i][j]`:

```
void f (Matrix& m)
{
    m (1, 2) = 5;              // индексация в стиле Fortran
    m.row (1) (2) = 6;
    m.row (1)[2] = 7;
    m[1] (2) = 8;              // нежелательное смешение стилей (но это работает)
    m[1][2] = 9;               // индексация в стиле C++
}
```

Использование ***slice_array*** для выражения индексации подразумевает наличие хорошего оптимизатора.

Обобщение этого подхода для n-мерной матрицы произвольных элементов и с осмысленным набором операций я оставляю в качестве упражнения (§ 22.9[7]).

Вашей первой мыслью для двухмерного вектора может оказаться такая:

```
class Matrix {
    valarray<valarray<double> > v;
public:
    //...
};
```

Это тоже заработало бы (§ 22.9[10]). Однако будет непросто достичь эффективности и совместимости, нужной для высокопроизводительных вычислений, не опускаясь на нижний и более обычный уровень, представленный массивами *valarray* и срезами.

22.4.7. Временные массивы, копирование и циклы

Если вы строите векторный или матричный класс, то вскоре обнаружите, что для удовлетворения пользователей, обеспокоенных быстродействием, вам придется столкнуться с тремя (взаимосвязанными) задачами:

[1] Минимизировать число временных массивов.
[2] Минимизировать копирование матриц.
[3] Минимизировать вложенные циклы над одними и теми же данными в составных операциях.

Эти проблемы не связаны напрямую со стандартной библиотекой, однако я опишу прием, которым можно воспользоваться для достижения хорошо оптимизированной реализации.

Рассмотрим выражение $U=M*V+W$, где U, V и W — вектора, а M — матрица. Бесхитростная реализация для $M*V$ и $M*V+W$ введет временные вектора и будет копировать результаты $M*V$ и $M*V+W$. Реализация поумнее вызовет функцию *mul_add_and_assign* (&*U*, &*M*, &*V*, &*W*), которая не вводит временных векторов, не копирует векторов и обращается к элементам матрицы минимальное количество раз.

Такая степень оптимизации редко необходима для более чем нескольких выражений, поэтому простое решение проблемы эффективности состоит во введении функции вроде *mul_add_and_assign* (&*U*, &*M*, &*V*, &*W*), чтобы пользователь вызывал ее, когда важна эффективность. Однако можно спроектировать *Matrix* так, чтобы для выражений соответствующего вида такая оптимизация применялась автоматически. То есть мы можем обращаться с $U=M*V+W$, используя один оператор с четырьмя операндами. Соответствующие общие приемы были продемонстрированы для манипуляторов *ostream* (§ 21.4.6.3). Как правило, или можно пользоваться, чтобы комбинация n бинарных операторов работала как один $(n+1)$-арный оператор. Обработка $U=M*V+W$ требует введения двух вспомогательных классов. Однако в некоторых системах данный прием может привести к внушительному повышению быстродействия (скажем, раз в 30), благодаря тому, что становится возможна мощная техника оптимизации.

Сначала мы определим результат умножения *Matrix* на *Vector*:

```
struct MVmul {
        const Matrix& m;
        const Vector& v;

        MVmul (const Matrix& mm, const Vector& vv) : m (mm), v (vv) { }

        operator Vector ();      // вычислить и вернуть результат
};

inline MVmul operator* (const Matrix& mm, const Vector& vv)
{
        return MVmul (mm, vv);
}
```

Это «умножение» ничего не делает, только хранит ссылки на свои операнды; вычисление $M*V$ откладывается. Объект, полученный оператором *, тесно связан с тем, что

во многих технических сообществах называют *closure* (замыкающим объектом). Подобным же образом мы можем разобраться с тем, что получится, если мы добавим *Vector*:

```
struct MVmulVadd {
        const Matrix& m;
        const Vector& v;
        const Vector& v2;

        MVmulVadd (const MVmul& mv, const Vector& vv) :
        m (mv.m), v (mv.v), v2 (vv) { }

        operator Vector ();              // вычислить и вернуть результат
};

inline MVmulVadd operator+ (const MVmul& mv, const Vector& vv)
{
        return MVmulVadd (mv, vv);
}
```

Таким образом вычисление *M*V+W* откладывается. Теперь нам надо гарантировать, что при присваивании этого вектору все будет вычисляться по хорошему алгоритму:

```
class Vector {
        // ...
public:
        Vector (const MVmulVadd& m)              // инициализация результатом mvm
        {
                // размещение элементов и т. п.
                mul_add_and_assign (this, &m.m, &m.v, &m.v2);
        }

        Vector& operator= (const MVmulVadd& m)    // присваивание *this результата m
        {
                mul_add_and_assign (this, &m.n, &m.v, &m.v2);
                return *this;
        }
        // ...
};
```

Теперь *U=M*V+W* автоматически раскрывается в

```
U.operator= (MVmulVadd (MVmul (M, V), W))
```

который из-за встраивания разрешается в желаемый простой вызов

```
mul_add_and_assign (&U, &M, &V, &W)
```

Ясно, что это устраняет копирование и временные массивы. Кроме того, мы могли бы оптимизировать саму функцию *mul_add_and_assign* (). Однако даже если мы и не будем этого делать, все равно форма кода оставит огромные возможности для оптимизатора.

Я ввел новый класс *Vector* (а не воспользовался *valarray*), потому что мне нужно было определить присваивание (а присваивание должно быть функцией-членом). Однако, *valarray* — это лучшая кандидатура для внутреннего представления *Vector*.

Важность этого приема заключается в том, что большинство критичных по времени вычислений с векторами и матрицами выполняются при помощи относительно простых синтаксических форм. Как правило, не ставится цели оптимизировать таким образом выражения из полдюжины операторов; для них хватает более привычных методов (§ 11.6).

Данный прием основывается на идее использования анализа времени компиляции и замыкающих объектов, чтобы перенести вычисление подвыражений в объект, представляющий составную операцию. Такой подход можно применять к разнообразным проблемам, имеющим следующий общий признак: перед тем, как производить вычисление, информацию нужно собрать в одну функцию. Объекты, порожденные для откладывания вычислений, я называю *объектами, замыкающими композицию*, или просто *композиторами*.

22.4.8. Обобщенные срезы

Пример с ***Matrix*** в § 22.4.6 показывает, как два среза можно использовать для описания строк и столбцов двухмерного массива. В общем случае срез может описывать любые строки и столбцы ***n***-мерного массива (§ 22.9[7]). Однако иногда нам нужно извлечь подмассив, не являющийся строкой или столбцом. Например, нам может понадобиться матрица 2×3 из левого верхнего угла матрицы 3×4:

00	01	02
10	11	12
20	21	22
30	31	32

К сожалению, эти элементы расположены так, что их не может описать один срез:

0	1	2									
00	10	20	30	01	11	21	31	02	12	22	32
				4	5	6					

gslice — это «обобщенный срез», который содержит (почти) всю информацию из ***n*** срезов:

```
class std::gslice {
    // вместо одного шага и одного размера (как для среза),
    // gslice содержит n шагов и n размеров
public:
    gslice ();
    gslice (size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

    size_t start () const;                  // индекс первого элемента
    valarray<size_t>size () const;          // число элементов в измерении
    valarray<size_t>stride () const;        // шаг для индекса[0], индекса[1], ...
};
```

Дополгительные значения позволяют ***gslice*** определить соответствие между ***n*** целыми числами и индексом для адресации элементов в массиве. Например, мы можем описать размещение матрицы 2×3 одной парой пар (длина,шаг). Как показано в

§ 22.4.5, при использовании размещения в стиле Fortran длина 2 и шаг 4 описывает два элемента строки матрицы 3×4. Аналогично, длина 3 и шаг 1 описывает три элемента столбца. Вместе они описывают все элементы подматрицы 2×3. Чтобы перечислить эти элементы, мы можем написать:

```
size_t gslice_index (const gslice& s, size_t i, size_t j)
{
    return s.start ()+i*s.stride ()[0]+j*s.stride ()[1];
}

size_t len[] = {2, 3 };                    // (len[0], str[0]) описывает строку
size_t str[] = {4, 1 };                    // (len[1], str[1]) описывает столбец

valarray<size_t> lengths (len, 2);
valarray<size_t> strides (str, 2);

void f ()
{
    gslice s (0, lengths, strides);

    for (int i = 0; i<s.size ()[0]; i++) cout << gslice_index (s, i, 0) << " ";      // строка
    cout << ", ";
    for (int j = 0; j<s.size ()[1]; j++) cout << gslice_index (s, 0, j)<< " ";      // столбец
}
```

Будет выведено *0 4 , 0 1 2*.

Таким способом ***gslice*** с двумя парами (длина,шаг) описывает подмассив двумерного массива, ***gslice*** с тремя парами (длина,шаг) описывает подмассив трехмерного массива и т. д. Использование ***gslice*** в качестве индекса для ***valarray*** создаст ***gslice_array***, состоящий из элементов, которые описывает ***gslice***. Например:

```
void f (valarray<float>& v)
{
    gslice m (0, lengths, strides);
    v[m] = 0;                              // обнуление v[0], v[1], v[2], v[4], v[5], v[6]
}
```

Тип ***gslice_array*** предлагает тот же набор членов, что и ***slice_array***. В частности, пользователь не может напрямую конструировать ***gslice_array***, и ***gslice_array*** нельзя копировать (§ 22.4.6). Вместо этого ***gslice_array*** является результатом использования ***gslice*** как индекса для массива ***valarray*** (§ 22.4).

22.4.9. Маски

Массив ***mask_array*** предоставляет еще один способ определения подмножества массива ***valarray*** и представления результата в виде, схожем с ***valarray***. В контексте массивов ***valarray***, маска — это просто ***valarray<bool>***. Когда маска применяется для индексирования ***valarray***, бит ***true*** обозначает, что соответствующий элемент массива ***valarray*** следует включать в результат. Это позволяет нам производить операции над подмножествами ***valarray***, даже если нет простой модели (вроде среза), которая описывала бы эти подмножества. Например:

```
void f(valarray<double>& v)
{
    bool b[] = { true, false, false, true, false, true };
    vlarray<bool> mask (b, 6);                  // элементы 0, 3 и 5

    valarray<double> vv = cos (v[mask]);        // vv[0]==cos(v[0]),
                                                // vv[1]==cos(v[3]),
                                                // vv[2]==cos(v[5])
}
```

mask_array предлагает тот же набор членов, что и ***slice_array***. В частности, пользователь не может напрямую конструировать ***mask_array***, и ***mask_array*** нельзя копировать (§ 22.4.6). Он получается в результате использования ***valarray<bool>*** как «индекса» для ***valarray*** (§ 22.4.2). Число элементов массива ***valarray***, используемого в качестве маски, не должно превышать числа элементов массива ***valarray***, который он индексирует.

22.4.10. Косвенные массивы

Массив ***indirect_array*** предоставляет способ произвольного выделения подмножества из ***valarray*** и переупорядочивания ***valarray***. Например:

```
void f(valarray<double> &v)
{
    size_t i[] = { 3, 2, 1, 0 };                // первые четыре элемента в обратном
                                                // порядке
    valarray<size_t> index (i, 4);              // элементы 3, 2, 1, 0 (в таком порядке)

    valarray<double> vv = log (v[index]);       // vv[0]==log(v[3]), vv[1]==log(v[2]),
                                                // vv[2]==log(v[1]), vv[3]==log(v[0])
}
```

Если индекс указан дважды, значит, мы сослались на соответствующий элемент ***valarray*** дважды в одной и той же операции. Это именно тот вид альтернативных имен, который для ***valarray*** запрещен, поэтому если индекс повторяется, поведение ***indirect_array*** не определено.

Тип ***indirect_array*** предлагает тот же набор членов, что и ***slice_array***. В частности, ***indirect_array*** (пользователю) нельзя напрямую конструировать, и его нельзя копировать (§ 22.4.6). Массив ***indirect_array*** получается в результате использования ***valarray<size_t>*** в качестве индекса для ***valarray*** (§ 22.4.2). Число элементов массива ***valarray***, используемого в качестве индекса, не должно превышать числа элементов массива ***valarray***, который он индексирует.

22.5. Комплексная арифметика

Стандартная библиотека предоставляет шаблон ***complex*** в духе того класса ***complex***, который был описан в § 11.3. Библиотечный класс ***complex*** должен быть шаблоном, чтобы комплексные числа можно было основывать на разных скалярных типах. В частности, для ***complex*** введены специализации, пользующиеся скалярными типами ***float***, ***double*** и ***long double***.

Шаблон ***complex*** определен в пространстве имен ***std*** и представлен в <***complex***>:

```
template<class T> class std::complex {
    T re, im;
public:
    typedef T value_type;

    complex (const T& r=T (), const T& i=T ()) : re (r), im (i) { }
    template<class X> complex (const complex<X>& a) : re (a.re), im (a.im) { }
    T real () const { return re; }
    T imag ()const { return im; }

    complex<T>& operator= (const T& z);          // присваивание complex(z, 0)
    template<class X> complex<T>& operator= (const complex<X>&);
    // аналогично: +=, -=, *=, /=
};
```

Это представление и встроенные функции приведены для иллюстрации. Можно вообразить (хотя и с трудом) стандартный библиотечный класс ***complex*** в другом представлении. Отметим использование шаблонов членов для инициализации и присваивания ему любого другого типа ***complex*** любым другим (§ 13.6.2).

На протяжении этой книги я использовал ***complex*** в качестве класса, а не шаблона. Это приемлемо, так как я использовал фокус с пространством имен, чтобы получить тип ***complex*** с ***double***, который я обычно предпочитаю:

```
typedef std::complex<double> complex;
```

Определены обычные унарные и бинарные операции:

```
template<class T> complex<T> operator+ (const complex<T>&, const complex<T>&);
template<class T> complex<T> operator+ (const complex<T>&, const T&);
template<class T> complex<T> operator+ (const T&, const complex<T>&);

// аналогично: -, *, /, == и !=

template<class T> complex<T> operator+ (const complex<T>&);
template<class T> complex<T> operator- (const complex<T>&);
```

Введены координатные функции:

```
template<class T> T real (const complex<T>&);
template<class T> T imag (const complex<T>&);

template<class T> complex<T> conj (const complex<T>&);

// конструирует из полярных координат (abs(), arg()):
template<class T> complex<T> polar (const T& rho, const T& theta);

template<class T> T abs (const complex<T>&);      // иногда называется rho (ρ)
template<class T> T arg (const complex<T>&);      // иногда называется theta (θ)

template<class T> T norm (const complex<T>&);     // квадрат abs()
```

Обеспечивается обычный набор математических функций:

```
template<class T> complex<T> sin (const complex<T>&);
// аналогично: sinh, tan, tanh, cos, cosh, exp, log и log10
```

```
template<class T> complex<T> pow (const complex<T>&, int);
template<class T> complex<T> pow (const complex<T>&, const T&);
template<class T> complex<T> pow (const complex<T>&, const complex<T>&);
template<class T> complex<T> pow (const T&, const complex<T>&);
```

И наконец, предоставляется потоковый ввод/вывод:

```
template<class T, class Ch, class Tr>
basic_istream<Ch, Tr>& operator>> (basic_istream<Ch, Tr>&, complex<T>&);
template<class T, class Ch, class Tr>
basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, const complex<T>&);
```

Комплексное число записывается в формате (x, y) и может быть считано в формате x, (x) и (x, y) (§ 21.2.3, § 21.3.5). Специализации ***complex<float>***, ***complex<double>*** и ***complex<long double>*** введены для сокращения преобразований (§ 13.6.2) и чтобы предоставить возможность оптимизации. Например:

```
template<> class complex< double> {
    double re, im;
public:
    typedef double value_type;

    complex (double r = 0.0, double i = 0.0) : re (r), im (i) { }
    complex (const complex<float>& a) : re (a.real ()), im (a.imag ()) { }
    explicit complex (const complex<long double>& a) : re (a.real ()), im (a.imag ()) { }

    //...
};
```

Теперь ***complex<float>*** может быть преобразовано в ***complex<double>***, в то время как ***complex<long double>*** не может. Аналогично специализации гарантируют, что ***complex<float>*** и ***complex<double>*** могут быть «тихо» преобразованы в ***complex<long double>***, но ***complex<long double>*** не может быть неявно преобразовано в ***complex<double>*** или ***complex<float>***, а ***complex<double>*** не преобразуется неявно в ***complex<float>***. Забавно, что присвоение не предлагает такой же защиты, как конструкторы. Например:

```
void f (complex<float> cf, complex<double> cd, complex<long double> cld, complex<int> ci)
{
    complex<double> c1=cf;          // хорошо
    complex<double> c2 = cd;        // хорошо
    complex<double> c3= cld;        // ошибка: возможно урезание
    complex<double> c4 (cld);       // правильно: вы попросили урезать
    complex<double> c5= ci;         // ошибка: нет преобразования

    c1 = cld;                       // правильно, но будте осторожны
    c1 = cf;                        // правильно
    c1 = ci;                        // правильно
}
```

22.6. Обобщенные числовые алгоритмы

В <*numeric*> стандартная библиотека предоставляет несколько обобщенных численных алгоритмов в стиле нечисленных алгоритмов из <***algorithm***> (глава 18):

Обобщенные числовые алгоритмы <numeric>	
accumulate ()	Накапливает результаты операции над последовательностью
inner_product ()	Накапливает результаты операции над двумя последовательностями
partial_sum ()	Генерирует последовательность по операции над последовательностью
adjacent_difference ()	Генерирует последовательность по операции над последовательностью

Эти алгоритмы обобщают обычные операции, такие как вычисление суммы, позволяя применять их ко всем видам последовательностей и задавая в виде параметров операцию, которую необходимо применить к элементам последовательностей. Для каждого алгоритма универсальная версия дополняется версией, применяющей наиболее распространенный для данного алгоритма оператор.

22.6.1. Накопление

Алгоритм *accumulate* () можно представить как обобщение суммы элементов вектора. Алгоритм *accumulate* () определяется в пространстве имен *std* и представлен в <*numeric*>:

```
template<class In, class T> T accumulate (In first, In last, T init)
{
    while (first != last) init = init + *first++;          // плюс
    return init;
}
template<class In, class T, class BinOp>
    T accumulate (In first, In last, T init, BinOp op)
{
    while (first != last) init = op (init, *first++);      // универсальная операция
    return init;
}
```

Простая версия *accumulate* () складывает элементы последовательности, используя их оператор +. Например:

```
void f (vector< int>& price, list< float>& incr)
{
    int i = accumulate (price.begin (), price.end (), 0);      // накапливаем в int
    double d = 0;
    d = accumulate (incr.begin (), incr.end (), d);            // накапливаем в double
    // ...
}
```

Обратите внимание на то, как тип начального переданного значения определяет тип результата.

Не все элементы, которые мы хотим добавить, доступны как элементы последовательности. Если это действительно так, зачастую можно обеспечить операцию, которая вызывалась бы из *accumulate* () для создания добавляемых элементов по мере по-

ступления. Среди кандидатов на роль передаваемой операции самым очевидным является извлечение значения из структуры данных. Например:

```
struct Record {
        // ...
        int unit_price;
        int number_of_units;
};

long price (long val, const Record& r)
{
        return val + r.unit_price * r.number_of_units;
}

void f(const vector<Record>& v)
{
        cout << "Итого: " << accumulate (v.begin (), v.end (), 0, price) << '\n';
}
```

Операции, подобные ***accumulate***, в некоторых сообществах называются ***reduce*** и ***reduction*** (приведение).

22.6.2. Алгоритм inner_product

Накопление из последовательности очень распространено, но накопление из двух последовательностей тоже не является редкостью. Алгоритм ***inner_product*** () определен в пространстве имен ***std*** и представлен в <***numeric***>:

```
template<class In, class In2, class T>
T inner_product (In first, In last, In2 first2, T init)
{
        while (first != last) init = init + *first++ * *first2++;
        return init;
}

template<class In, class In2, class T, class BinOp, class BinOp2>
T inner_product (In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
        while (first != last) init = op (init, op2 (*first++, *first2++));
        return init;
}
```

Как обычно, в качестве аргумента передается только начало второй входной последовательности. Считается, что вторая входная последовательность по крайней мере такой же длины, что и первая.

Алгоритм ***inner_product*** является ключевой операцией при умножении матрицы ***Matrix*** на вектор ***valarray***:

```
valarray<double> operator* (const Matrix& m, valarray<double>& v)
{
        valarray<double> res (m.dim2 ());

        for (size_t i = 0; i<m.dim2 (); i++) {
                const Cslice_iter<double>& ri = m.row (i);
```

```
            res [i] = inner_product (ri, ri.end (), &v[0], double (0));
        }
        return res;
    }

    valarray<double> operator* (valarray<>& v, const Matrix& m)
    {
        valarray<double> res (m.dim1 ());

        for (size_t i = 0; i<m.dim1 (); i++) {
            const Cslice_iter<double>& ci = m.column (i);
            res [i] = inner_product (ci, ci.end, &v[0], double (0));
        }
        return res;
    }
```

Некоторые разновидности алгоритма ***inner_product*** часто называют «скалярным произведением» (dot product).

22.6.3. Инкрементное изменение

Алгоритмы ***partial_sum*** () и ***adjacent_difference*** () являются обратными по отношению друг к другу и связаны с понятием инкрементного изменения. Они определены в пространстве имен ***std*** и представлены в <***numeric***>:

```
template<class In, class Out>
    Out adjacent_difference (In first, In last, Out res);

template<class In, class Out, class BinOp>
    Out adjacent_difference (In first, In last, Out res, BinOp op);
```

Получив последовательность ***a***, ***b***, ***c***, ***d*** и т. д., ***adjacent_difference*** выдаст ***a***, ***b–a***, ***c–b***, ***d–c*** и т. д.

Рассмотрим вектор с последовательностью замеров температуры. Мы можем преобразовать его в вектор изменений температуры:

```
vector<double> temps;

void f ()
{
    adjacent_difference (temps.begin (), temps.end (), temps.begin ());
}
```

Например, ***17***, ***19***, ***20***, ***20***, ***17*** превратится в ***17***, ***2***, ***1***, ***0***, ***–3***.

Алгоритм ***partial_sum***, наоборот, позволяет нам вычислить конечный результат набора инкрементных изменений:

```
template<class In, class Out, class BinOp>
Out partial_sum (In first, In last, Out res, BinOp op)
{
    if (first == last) return res;
    *res = *first;
    T val = *first;
    while (++first != last) {
        val = op (val, *first);
```

```
            *++res = val;
        }
        return ++res;
    }

    template<class In, class Out> Out partial_sum (In first, In last, Out res)
    {
        return partial_sum (first, last, res, plus);        // § 18.4.3
    }
```

Получив последовательность *a*, *b*, *c*, *d* и т. д., ***partial_sum*** выдаст *a*, *a*+*b*, *a*+*b*+*c*, *a*+*b*+*c*+*d* и т. д. Например:

```
    void f()
    {
        partial_sum (temps_changes.begin (), temps_changes.end (), temps.begin ());
    }
```

Обратите внимание на то, как ***partial_sum*** () увеличивает ***res*** перед присваиванием нового значения через него. Это позволяет результату быть той же последовательностью, что и входная; ***adjacent_difference*** () ведет себя аналогично. Так

```
    partial_sum (v.begin (), v.end (), v.begin ());
```

превратит последовательность *a*, *b*, *c*, *d* в *a*, *a*+*b*, *a*+*b*+*c*, *a*+*b*+*c*+*d* и т. д., а

```
    adjacent_difference (v.begin (), v.end (), v.begin ());
```

превратит ее в изначальную. В частности, ***partial_sum*** () снова превратит *17*, *2*, *1*, *0*, *–3* в *17*, *19*, *20*, *20*, *17*.

Эти операции полезны для анализа последовательности изменений. Например, анализ изменений цен на акции включает в себя две такие операции.

22.7. Случайные числа

Случайные числа очень важны во многих видах моделирования и в играх. В <***cstdlib***> и <***stdlib.h***> стандартная библиотека предоставляет простые необходимые средства для генерирования случайных чисел:

```
    #define RAND_MAX implementation_defined      /* большое положительное целое */

    int rand ();                    // псевдослучайные числа от 0 до RAND_MAX
    void srand (unsigned int i);    // инициализирует генератор случайных чисел числом i
```

Создать хороший генератор псевдослучайных чисел не так легко, и, к сожалению, не все системы снабжены хорошим ***rand*** (). В частности, младшие биты в случайном числе часто вызывают подозрение, поэтому ***rand*** ()%*n* — нельзя назвать хорошим переносимым способом генерирования случайных чисел от ***0*** до ***n–1***. Часто приемлемый результат получается в результате (***double*** (***rand*** ())/***RAND_MAX***)**n*. Однако для серьезного использования этой формулы мы должны позаботиться о малой вероятности того, что результат может быть *n*.

Вызов ***srand*** () начинает новую последовательность случайных чисел с *семени* (seed), заданного в качестве аргумента. Для отладки часто важно, чтобы псевдослучайная последовательность от данного семени повторялась. Однако часто нам нужно

начинать каждое новое реальное выполнение программы с новым семенем. Чтобы сделать игру фактически непредсказуемой, часто полезно выбирать семя из окружения программы. Для таких программ хорошее семя получается из битов счетчика реального времени.

Если вы должны написать собственный генератор случайных чисел, обязательно тщательно его проверьте (§ 22.9[14]).

Генератор случайных чисел часто оказывается удобнее, если его представить как класс. В этом случае генераторы случайных чисел с разными видами распределений легко строятся следующим способом:

```
class Randint {    // равномерное распределение, предполагает 32-битный тип long
        unsigned long randx;
public:
        Randint (long s = 0) { randx = s; }
        void seed (long s) { randx = s; }

        // приведенные ниже магические числа нужны
        // для выделения 31 бита из 32-битного числа типа long:
        long abs (long x) { return x&0x7fffffff; }
        static double max () { return 2147483648.0; }          // внимание: double
        long draw () { return randx = randx*1103515245 + 12345; }

        double fdraw () { return abs (draw ())/max (); }       // интервал [0, 1]

        long operator () () { return abs (draw); }             // интервал [0, pow(2,31)]
}
class Urand : public Randint {
        // равномерное распределение в интервале [0:n[
        int n;
public:
        Urand (int nn) { n = nn; }

        int operator () () { int r = n*fdraw (); return (r == n) ? n-1 : r; }
};
class Erand : public Randint {
        // генератор случайных чисел с экспоненциальным распределением
        int mean;                                              // среднее
public:
        Erand (int m) { mean = m; }
        int operator () () { return -mean * log ( (max () - draw ())/max () + .5; }
};
```

Вот простой тест:

```
int main ()
{
        Urand draw (10);
        map<int, int> bucket;
        for (int i =0; i<1000000; i++) bucket[draw ()]++;
        for (int j =0; j<10; j++) cout << bucket[j] >> '\n';
}
```

Если все значения ***bucket*** не равны приблизительно 100 000, где-то в программе ошибка.

Эти генераторы случайных чисел являются слегка подредактированными версиями того, что я поставлял вместе с самой первой библиотекой C++ (в действительности, с библиотекой «C с классами», § 1.4).

22.8. Советы

[1] Численные проблемы часто очень тонки. Если вы не на 100% уверены в математическом аспекте задачи, воспользуйтесь советом специалиста или поэкспериментируйте; § 22.1.

[2] Чтобы определить свойства встроенных типов, пользуйтесь *numeric_limits*; § 22.2.

[3] Для скалярных типов, определяемых пользователем, специализируйте *numeric_limits*; § 22.2.

[4] Для математических вычислений, когда быстродействие важнее гибкости по отношению к операциям и типам элементов, пользуйтесь массивами *valarray*; § 22.4.

[5] Операции над частью массива выражайте при помощи срезов, а не циклов; § 22.4.

[6] Для достижения эффективности, чтобы избавиться от временных массивов и использовать лучшие алгоритмы, пользуйтесь композиторами; § 22.4.7.

[7] Для арифметики с комплексными числами пользуйтесь *std::complex*; § 22.5.

[8] Старый код, использующий класс *complex*, при помощи *typedef* вы можете преобразовать для работы с шаблоном *std::complex*; § 22.5.

[9] Прежде чем писать цикл для вычисления значения по списку, рассмотрите применение алгоритмов *accumulate* (), *inner_product* (), *partial_sum* () и *adjacent_difference* (); § 22.6.

[10] Для конкретного вида распределения предпочитайте класс случайных чисел, а не пользуйтесь напрямую *rand* (); § 22.7.

[11] Убедитесь в том, что ваши случайные числа действительно случайны; § 22.7.

22.9. Упражнения

1. (*1.5) Напишите функцию, которая вела бы себя как *apply* () из § 22.4.3, за исключением того, что была бы функцией-не-членом и принимала бы в качестве аргументов объекты-функции.
2. (*1.5) Напишите функцию, которая вела бы себя как *apply* () из § 22.4.3, за исключением того, что была бы функцией-не-членом, принимала бы в качестве аргументов объекты-функции и изменяла свой аргумент *val_array*.
3. (*2) Завершите *Slice_iter* (§ 22.4.5). Будьте особенно внимательны при определении деструктора.
4. (*1.5) Перепишите программу из § 17.4.1.3, применив *accumulate* ().
5. (*2) Реализуйте операторы ввода/вывода << и >> для *val_array*. Реализуйте функцию *get_array* (), которая создавала бы массив *val_array* с размером, который определялся бы при вводе.
6. (*2.5) Определите и реализуйте трехмерную матрицу с соответствующими операциями.

7. (*2.5) Определите и реализуйте *n*-мерную матрицу с соответствующими операциями.
8. (*2.5) Реализуйте класс, подобный *valarray*, и определите для него операторы + и *. Сравните их быстродействие с реализацией *valarray* в вашей версии C++. Подсказка: Включите в проверку выражение *x=0.5 *(x+y)−z* и попробуйте его с разными размерами векторов *x*, *y* и *z*.
9. (*3) Реализуйте массив *Fort_array* в стиле Fortran; где индексы начинаются с *1*, а не с *0*.
10. (*3) Реализуйте *Matrix*, используя для представления элементов члены *valarray* (а не указатель или ссылку на *valarray*).
11. (*2.5) При помощи композиторов (§ 22.4.7) реализуйте эффективное многомерное индексирование с использованием обозначения []. Например, *v1*[*x*], *v2*[*x*][*y*], *v2*[*x*], *v3*[*x*][*y*][*z*], *v3*[*x*][*y*] и *v3*[*x*] должны выдать соответствующие элементы и подмассивы при помощи простых вычислений с индексом.
12. (*2) Обобщите идею из программы в § 22.7 в функцию, которая, получив в качестве аргумента генератор случайных чисел, вывела бы простое графическое представление их распределения, что могло бы стать грубой наглядной проверкой правильности работы генератора.
13. (*1) Если *n* относится к типу *int*, каково распределение (*double* (*rand* ())/ *RAND_MAX*)**n*?
14. (*2.5) Выведите на экран точки в квадратной области. Пары координат для точек должны выдаваться генератором *Urand* (*N*), где *N* — число пикселей вдоль соответствующей стороны области вывода.
15. (*2) Реализуйте генератор *Nrand*, выдающий случайные числа с нормальным распределением.

ЧАСТЬ ЧЕТВЕРТАЯ

Проектирование с использованием C++

Эта часть знакомит читателя с методами C++, облегчающими разработку программного обеспечения. Упор делается на проектирование и эффективную реализацию проекта в терминах конструкций языка.

«... Только сейчас я начинаю открывать для себя всю сложность выражения идей на бумаге. Пока речь идет исключительно об описании, работать просто; но лишь только в игру вступает рассуждение, как соблюдение должных взаимосвязей, ясности и относительной легкости восприятия оказывается той упомянутой выше сложностью, о которой я раньше и понятия не имел...»

— Чарльз Дарвин

23

Разработка и проектирование

Не существует серебряной пули.
— Ф. Брукс

Построение программного продукта — цели и средства — процесс разработки — цикл разработки — этапы проектирования — выявление классов — определение операций — определение зависимостей — определение интерфейсов — реорганизация иерархии классов — модели — экспериментирование и анализ — тестирование — сопровождение программного продукта — эффективность — руководство разработкой — повторное использование — масштаб — важность личностей— гибридное проектирование — библиография — советы.

23.1. Обзор

Эта глава является первой из трех, призванных подробно познакомить вас с производством программного продукта, начиная с относительно общего взгляда на проектирование и заканчивая специфичными для C++ приемами и концепциями проектирования. После введения и короткого обсуждения целей и средств разработки программного обеспечения в § 23.3 эта глава состоит из двух основных частей:

§ 23.4 Процесс разработки программного продукта
§ 23.5 Практические суждения по поводу организации разработки программного продукта

В главе 24 обсуждается связь между проектированием и языком программирования. Глава 25 знакомит вас с некоторыми ролями, которые играют классы в организации программы с точки зрения проектирования. В целом три главы части IV ставят своей целью соединить два метода проектирования: тот, который мнит себя независимым от языка и тот, который близоруко фокусируется на деталях. В большом проекте присутствуют обе эти крайние точки зрения, но во избежание неприятностей и чрезмерной цены разработки они должны стать составной частью единого континуума, включающего в себя и программирование, и все необходимые практические аспекты.

23.2. Введение

Создание любого нетривиального программного продукта представляет собой сложную и зачастую устрашающую задачу. Даже для отдельного программиста простое написание программных конструкций является лишь частью процесса. Как правило, задача написания и отладки отдельных фрагментов конечной программы кажется простой по сравнению с анализом проблемы, общим проектированием программы, созданием документации, тестированием и сопровождением, а также руководством всем этим. Естественно, можно назвать всю эту деятельность «программированием» и заявить: «Я не занимаюсь проектированием, что вы! Я только программирую». Но как бы ни называть эту активность, иногда важно сфокусироваться на ее отдельных частях — и точно так же важно иногда рассмотреть процесс целиком. Ни детали, ни общая картина не должны теряться из виду за стремлением поскорее сдать систему — хотя довольно часто именно так и случается.

Данная глава посвящена тем частям процесса разработки программ, которые не связаны с написанием и отладкой отдельных фрагментов конечной программы. Представленное обсуждение менее конкретно и подробно, чем описание отдельных особенностей языка и специфических приемов программирования, представленных в других разделах этой книги. Однако, это неизбежно, так как для создания хорошего программного продукта не может быть общих рецептов. Подробное описание типа «Как изготовить программу» можно привести для специфических хорошо понятных прикладных случаев, но не для общих областей применения. В программировании ничем не заменишь ума, опыта и вкуса. И потому данная глава предлагает только общие рекомендации, различные подходы и предостережения.

Обсуждение затрудняется абстрактной природой программного продукта и тем фактом, что приемы, работающие в сравнительно небольших проектах (скажем, где один или два программиста пишут 10 000 строчек программы), не обязательно распространяются на средние и большие проекты. По этой причине некоторые мнения сформулированы применительно к менее абстрактным инженерным дисциплинам, а не к программированию. Пожалуйста, помните, что «доказательство по аналогии» обманчиво, и аналогии здесь используются только для иллюстрации. Рассуждения о проектировании, выраженные в терминах C++, с примерами можно найти в главах 24 и 25. Идеи, высказанные в данной главе, отражены и в языке C++, и в отдельных примерах на протяжении всей книги.

Помните также, что из-за огромного разнообразия областей применения, людей и средств разработки программного обеспечения вам не следует ожидать, что каждое сделанное здесь высказывание будет напрямую применимо к вашей текущей проблеме. Приведенные наблюдения сделаны в реальных жизненных проектах и применимы к самым разным ситуациям, но их нельзя считать универсальными. Отнеситесь к ним со здравой долей скептицизма.

C++ можно использовать просто как улучшенный C. Однако при этом останутся неиспользованными самые мощные средства и особенности языка, и будет достигнута лишь малая толика тех выгод, которые дает C++. Эта глава фокусируется на подходе к проектированию, который дает возможность эффективного использования средств абстракции данных и объектно-ориентированного программирования, предлагаемых C++; такой подход часто называют *объектно-ориентированным проектированием*.

В этой главе можно выделить несколько главных тем:

- Самое главное при разработке программного обеспечения — ясно понимать, что вы пытаетесь построить.
- Успешная разработка программного обеспечения — это дело, требующее времени.
- Конструируемые нами системы стремятся оказаться на пределе сложности, доступной нам и нашим средствам.
- В проектировании и программировании не существует универсальных рецептов, которые могли бы заменить ум, опыт и хороший вкус.
- При разработке всякого нетривиального программного обеспечения незаменимо экспериментирование.
- Проектирование и программирование — итеративные процессы.
- Нельзя строго разделить фазы создания программы, такие как проектирование, программирование и тестирование.
- Нельзя рассматривать программирование и проектирование, не затрагивая вопросы менеджмента.

Легко недооценить приведенные утверждения — и, как правило, это дорого обходится. Абстрактные идеи, которые они воплощают, трудно претворить в практику. Необходимо отметить неизбежность экспериментирования. Также как судостроение, езда на велосипеде или программирование, проектирование — не то искусство, которым можно овладеть, изучая только теорию.

Эта глава касается проектирования систем, которые находятся на грани возможностей опыта и ресурсов разработчиков. Похоже, это в природе людей и целых организаций — стараться спроектировать что-то на пределе собственных возможностей. Системы, не бросающие такого вызова, не нуждаются в особом обсуждении. У таких проектов уже есть свои устоявшиеся средства разработки, от которых не стоит отказываться. Новые, лучшие средства и методы нужны только для более честолюбивых попыток. А проекты, которые «мы знаем, как делать», можно передать относительным новичкам, которые этого не знают.

Для проектирования и построения системы не существует «единственно правильного способа». Я бы счел веру в «единственно правильный способ» детской болезнью, если бы ей так часто не страдали опытные программисты и проектировщики. Пожалуйста, помните, что если прием сработал у вас в прошлом году для одного проекта, это ничуть не значит, что он будет неизменно работать у кого-нибудь другого или у вас в другом проекте. Очень важно держать ум открытым.

Ясно, что многое в приведенном здесь обсуждении относится к разработке программного обеспечения промышленного масштаба. Читатели, не вовлеченные в подобные разработки, могут развалиться в кресле и с удовольствием посмотреть на ужасы, которых они избежали. Или они могут поискать темы, относящиеся к их сфере деятельности. Нет нижнего предела размеров программ, для которых, прежде чем писать код, имеет смысл создать проект. Однако есть нижний предел, для которого годится любой частный подход к проектированию и документированию. Для рассмотрения вопросов масштабирования обратитесь к § 23.5.2.

Самая фундаментальная проблема в разработке программного продукта — это взаимосвязь его частей. Существует лишь один основной метод борьбы со сложно-

стью — разделяй и властвуй. Если задачу можно разделить на две подзадачи, с которыми можно справиться по отдельности, то таким разделением проблема уже более чем наполовину решена. Этот простой принцип применим к удивительно большому множеству случаев. В частности, использование модулей и классов при проектировании системы разделяет программу на две части — реализацию и ее пользователей, — и эти две части связываются лишь одним (в идеальном случае) хорошо определенным интерфейсом. Это фундаментальный подход к борьбе с присущей программам сложностью. Аналогично, процесс проектирования программы можно разбить на независимые действия с четко определенным (в идеальном случае) взаимодействием между участвующими в проекте людьми. Это основной подход в борьбе со сложностью, внутренне присущей процессу разработки и связанным с ним человеческим отношениям.

В обоих случаях вычленение частей и определение интерфейсов между ними как раз и является тем местом, где необходим опыт и вкус. Такое вычленение — не простой механический процесс; как правило, он требует проникновения в суть, которого можно достичь только через полное понимание системы и выход на соответствующий уровень абстрагирования (см. § 23.4.2, § 24.3.1 и § 25.3). Близорукий взгляд на программу или на процесс разработки программного продукта часто приводит к серьезным изъянам. Отметим также, что *разделение* производится сравнительно легко как для программ, так и для людей. Труднее обеспечить *связь* между частями, находящимися по разные стороны барьера, не разрушив сам барьер и не задушив общения, необходимого для взаимодействия.

Эта глава знакомит вас с подходом, а не с полным методом проектирования. Полный формальный метод проектирования находится за пределами рассмотрения данной книги. Представленный здесь подход можно использовать с разной степенью формализации; он может также служить основой для различных формализаций. Данная глава — не обзор литературы; она не пытается затронуть все темы, относящиеся к разработке программных продуктов, или познакомить вас со всеми точками зрения на разработку. Это остается за пределами данной книги. Обзор литературы можно найти в [Booch, 1994]. Отметим, что используемые здесь термины являются довольно универсальными и общепринятыми. Особо «интересные» термины, такие как *проектирование* (design), *прототип* (prototype) и *программист* (programmer) имеют в литературе несколько разных и зачастую противоречащих друг другу определений. Пожалуйста, будьте осторожны, и, основываясь на узко понятых терминах или понятиях, смысл которых вы осознаете не вполне точно, не прочитайте здесь чего-нибудь такого, что я ни в коей мере не намеревался говорить.

23.3. Цели и средства

Целью профессионального программирования является поставка готового продукта, который удовлетворил бы пользователей. Главные средства для этого — создать программу с ясной внутренней структурой и вырастить группу проектировщиков и программистов, достаточно квалифицированных и имеющих стимул, чтобы быстро реагировать на новые методы и возможности.

Почему? Внутренняя структура программы и процесс ее создания в идеале не должны волновать конечного пользователя. Я скажу больше: если конечному пользователю

приходится беспокоиться о том, как писалась программа, то с этой программой что-то неладно. Так почему же важно, какова структура программы, и какие люди ее создавали?

Программа должна иметь ясную внутреннюю структуру, чтобы ее было легче:

- тестировать;
- переносить на другие системы;
- обслуживать;
- расширять;
- изменять;
- понимать.

Главное в том, чтобы каждый успешно разработанный программный продукт имел свою жизнь, на протяжении которой с ним бы работали разные программисты и проектировщики, он бы переносился на новую аппаратуру, приспосабливался к непредвиденным задачам и так или иначе переделывался. За время жизни программного продукта должны рождаться его новые версии с приемлемым уровнем ошибок и с разумными затратами. Не планируя этого, вы планируете неудачу.

Отметим, что хотя конечные пользователи в идеале не должны знать внутреннюю структуру системы, в действительности, они могут ею заинтересоваться. Например, пользователь может захотеть узнать кое-что о проекте системы, чтобы оценить ее надежность или способность к переделкам и расширению. Если рассматриваемый программный продукт не представляет собой завершенной системы — например, это набор библиотек для построения других программ — то пользователи захотят узнать больше «подробностей», чтобы получше применить библиотеки или воспользоваться ими как источником идей.

Должен быть найден баланс между отсутствием общего проекта и слишком большим упором на структуру. Первое ведет к бесконечному срезанию углов («мы только сдадим эту программу, а проблему решим в следующей версии»). Второе ведет к слишком усложненному проекту, в котором сущность теряется в формализме, и к ситуациям, когда реализация задерживается из-за постоянных реорганизаций программы («но эта новая структура *гораздо* лучше старой; люди подождут»). Это также часто приводит к тому, что система начинает требовать столько ресурсов, что большинство потенциальных пользователей не смогут себе позволить ее приобрести. Такое балансирование — самый трудный аспект проектирования, и именно здесь проявляются талант и опыт. Этот выбор труден для отдельного проектировщика или программиста, и еще труднее его сделать в больших проектах, где участвует много людей разной квалификации и опыта.

Программа должна производиться и обслуживаться организацией, способной делать это несмотря на смену персонала, руководства и изменение структуры управления. Обычный подход к этой проблеме заключается в том, чтобы свести разработку системы к задачам относительно низкого уровня, подгоняемых под грубую общую схему. То есть идея заключается в том, что создается класс быстро обучаемых (дешевых) взаимозаменяемых программистов низкого уровня («кодировщиков») и класс не таких дешевых, но столь же взаимозаменяемых (и стало быть столь же малоценных) проектировщиков. Предполагается, что кодировщики не принимают проектных решений, в то время как проектировщики не утруждают себя черной работой по вниканию в детали кодирования. Такой подход часто приводит к неудаче. Там, где он работает, в результате получаются чрезмерно громоздкие системы с низкой производительностью.

Проблемы этого подхода таковы:

- недостаточная связь между реальными разработчиками и проектировщиками, что ведет к упущенным возможностям, задержкам, неэффективности и повторяющимся проблемам, так как никто не учится на ошибках;
- узкие рамки для проявления инициативы со стороны реальных разработчиков, что ведет к недостаточному профессиональному росту, заглушению инициативы, разболтанности и высокой текучести кадров.

По сути дела, такой системе не хватает механизма обратной связи, чтобы люди учились на чужом опыте. Это растрата человеческого таланта. Создание среды разработки, в рамках которой человек может проявлять разнообразные таланты, развивать новые способности, вносить идеи и радоваться своей работе — это не просто цель, достойная сама по себе, но вещь, приносящая практическую экономическую выгоду.

С другой стороны, нельзя строить систему, обслуживать ее и разрабатывать документацию без какой-либо бюрократической структуры. Просто найти лучших людей и дать им наброситься на проблему самым подходящим с их точки зрения образом часто является идеальным началом для проектов, требующих новшеств. Однако по мере развития проекта становится необходимым уделять внимание календарному планированию, специализации и формальным связям между участниками проекта. Слишком жесткая система может остановить рост и задушить новшества. Здесь как раз и испытываются талант и опыт руководителя. Для отдельной личности подобная дилемма заключается в выборе, где попытаться проявить ум, а где просто «действовать по инструкции».

Рекомендация — думать не только о ближайшем выпуске текущей программы, но и о долгосрочных целях. Беспокоиться только о ближайшем выпуске программы значит планировать неудачу. Мы должны разрабатывать организацию и стратегию разработки программы, нацеленные на производство и обслуживание многих выпусков многих проектов, мы должны планировать серию успехов.

Цель «проектирования» — создать ясную и сравнительно простую внутреннюю структуру программы, иногда называемую *архитектурой*. Другими словами, мы хотим создать среду, в которую естественно впишутся отдельные части программы, и которая сама направляла бы написание фрагментов кода.

Проект — это конечный продукт процесса проектирования (если в итеративном процессе существует *конечный* продукт). На нем фокусируется связь между проектировщиком и программистом, а также между программистами. Здесь важно иметь чувство меры. Если я — отдельный программист и разрабатываю небольшую программку, которую собираюсь реализовать завтра, соответствующий уровень точности и подробности проекта может быть на уровне нескольких набросков на обратной стороне конверта. С другой стороны, разработка системы с участием сотен проектировщиков и программистов может потребовать томов спецификаций, тщательно написанных в соответствии со стандартом и нормами. Определение соответствующего уровня подробности, точности и формальности проекта — сама по себе непростая техническая и управленческая задача.

В этой и следующих главах я предполагаю, что проект системы выражается набором объявлений классов (как правило, опуская закрытые части объявлений как ненужные подробности) и связей между ними. Это упрощение. В конкретный проект входит гораздо больше всего; например, согласование классов и функций, организация программы,

позволяющая минимизировать перекомпиляцию, вопросы долговременного хранения и использования многих компьютеров. Однако для рассмотрения на данном уровне детализации необходимо упрощение, и главной целью проектирования с применением C++ являются классы. Некоторые из перечисленных аспектов упоминаются на протяжении этой главы, а некоторые, прямо влияющие на проектирование программ на C++, рассматриваются в главах 24 и 25. Для более детального обсуждения и рассмотрения примеров объектно-ориентированного проектирования обратитесь к [Booch, 1994].

Различие между анализом и проектированием я оставлю не до конца уточненным, поскольку обсуждение этой темы выходит за пределы книги и очень чувствительно к различиям в конкретных методах проектирования. Главное — выбрать метод анализа, соответствующий стилю проектирования, и стиль проектирования, соответствующий стилю программирования и применяемому языку.

23.4. Процесс разработки

Разработка программного продукта — итеративный и последовательный процесс. За время разработки многократно возвращаются к каждой стадии процесса, и каждый раз конечный результат на данной стадии улучшается. Вообще говоря, этот процесс не имеет начала и конца. Проектируя и реализуя систему, вы начинаете с основы, заложенной другими людьми,— с их проектов, библиотек и прикладных программ. Заканчивая, вы оставляете проект и программу другим для улучшения, пересмотра, расширения и переноса на другие системы. Естественно, специфический проект может иметь определенные начало и конец, и необходимо (хотя часто довольно трудно) ясно и точно ограничить проект по времени. Однако, считая, что все заканчивается с «последним выпуском», вы можете породить серьезные проблемы для ваших преемников (часто для вас самих в другой роли).

Одним из следствий данного подхода является то, что последующие разделы можно читать в любом порядке, поскольку аспекты проектирования и реализации в реальном проекте могут почти произвольно перемежаться. То есть «проект» почти всегда переделывается, основываясь на предыдущих проектах и опыте других реализаций. Кроме того, проект ограничивается календарными планами, квалификацией участников, совместимостью отдельных частей и т. д. Главная трудность для проектировщика/менеджера/программиста заключается в создании процесса, не удушающего новшеств и не разрушающего обратной связи, необходимых для успешной разработки.

Процесс разработки имеет три этапа:

- анализ: определение границ решаемой проблемы;
- проектирование: создание общей структуры системы;
- реализация: написание и тестирование программы.

Пожалуйста, помните об итеративной природе процесса — эти этапы намеренно не пронумерованы. Отметим, что некоторые важные аспекты в разработке программы не проявляются как отдельные стадии, поскольку должны пронизывать весь процесс, а именно:

- экспериментирование;
- тестирование;
- анализ проекта и реализации;
- документирование;
- менеджмент.

«Сопровождение» программы просто подразумевает дополнительные проходы по циклу разработки (§ 23.4.6).

Очень важно, чтобы анализ, проектирование и реализация не слишком разделялись, и чтобы участники разработки имели некую общую культуру, позволяющую эффективно общаться. Слишком часто в больших проектах все это отсутствует. В идеале люди на протяжении процесса переходят от одного этапа к другому: лучший способ переноса нетривиальной информации — это хранить ее в голове одного человека. К сожалению, организации часто устанавливают барьеры против такого последовательного перехода, например, предоставляя проектировщикам более высокий статус и/или более высокий оклад, чем «простым программистам». Если среди сотрудников не практикуется хождение туда-сюда для того, чтобы учить и учиться, их нужно по крайней мере поощрять регулярно беседовать с участниками «других» этапов разработки.

Для небольших и средних проектов часто не делается различий между анализом и проектированием; эти две фазы сливаются в одну. В маленьких проектах часто не делают различия между проектированием и программированием. Этим решаются проблемы общения. Важно придать данному проекту подходящий уровень формализации и поддерживать соответствующий уровень разделения между его фазами (§ 23.5.2). Здесь не существует единственно верного рецепта.

Описанная модель разработки программного продукта в корне отличается от традиционной «модели водопада». В модели водопада процесс разработки в организованной линейной манере поэтапно проходит все фазы от анализа до тестирования. Модель водопада имеет фундаментальный изъян, заключающийся в том, что информация течет лишь в одном направлении. Когда проблемы находятся «внизу по течению», часто возникает сильное методологическое и организационное давление, призванное обеспечить их решение непосредственно «по месту выявления»; то есть не затрагивая предыдущие этапы процесса. Этот дефицит обратной связи ведет к ущербности проекта, а исправления «по месту выявления» приводят к «корявой» реализации. В случаях, когда информация течет назад, к источнику, и приводит к изменениям в проекте, процесс замедляется, и поэтому люди противятся изменениям, неохотно и медленно реагируя на информацию «снизу». В качестве довода в пользу «неизменности» и решения проблем «на месте» часто говорят, что одно подразделение не должно переделывать свою работу из-за прихоти другого. В частности, к тому времени, когда находится серьезная ошибка, часто порочным решением уже порождено столько бумаг, что по сравнению с усилиями, связанными с изменением документации, изменения в программе кажутся ничтожными. Таким образом, в разработке программного продукта проблема, связанная с «бумажной работой», играет не последнюю роль. Естественно, как ни организуй разработку большой системы, подобные проблемы могут возникать и возникают. В конце концов, *какие-то* бумаги необходимы. Однако стремление к линейной модели разработки (модели водопада) сильно увеличивает вероятность того, что эти проблемы выйдут из-под контроля.

Недостаток модели водопада заключается в недостаточной обратной связи и неспособности реагировать на изменения. Опасность же описанного здесь итеративного подхода — в соблазне принять серию несвязных изменений за реальную мысль и прогресс. Обе проблемы легче выявить, чем решить, и в любом деле легко и соблазни-

тельно счесть бурную деятельность гарантом движения к цели. Естественно, с развитием проекта те или иные этапы становятся более или менее важными. Сначала упор делается на анализ и проектирование, а программирование привлекает меньше внимания. С течением времени акцент перемещается на проектирование и программирование, а затем на программирование и тестирование. Однако, никогда нельзя фокусироваться только на одной стадии процесса «анализ/проектирование/программирование», полностью забывая про остальные.

Помните, что ни внимание к деталям, ни применение соответствующих приемов менеджмента, ни передовые методы вам не помогут, если у вас нет ясного представления, чего вы хотите достичь. Большая часть проектов кончается неудачей именно из-за отсутствия четко определенной и реалистичной цели, чем из-за каких-либо других причин. Что бы вы ни делали и как бы ни плутали, ясно понимайте свою цель, определите осязаемые вехи и верстовые столбы, и не ищите технических решений для социальных проблем. С другой стороны, используйте всякие *подходящие* доступные методы — даже если они требуют инвестиций; с подходящими средствами и в хорошем окружении люди работают лучше. Не думайте, что следовать этому совету легко.

23.4.1. Цикл разработки

Разработка системы является итеративным процессом. Главный цикл состоит из многократных повторений следующих этапов:

[0] исследование проблемы;
[1] создание общего проекта;
[2] выявление стандартных компонент;
  - приспособление компонент для данного проекта;

[3] создание новых стандартных компонент;
  - приспособление компонент для данного проекта;

[4] сборка проекта.

В качестве аналогии рассмотрим автомобильный завод. Чтобы запустить производство, нужен общий проект нового типа автомобиля. Первый набросок будет основан на некотором анализе и определении данного автомобиля в общих словах, относящихся больше к его назначению, а не к тому, как достичь соответствующих свойств. Решение о том, какие свойства желательны (или, в идеале, выработка относительно простой схемы для определения того, какие свойства желательны), часто является самым трудным. Правильное принятие решения — это как правило, дело одной «проницательной» личности, и часто это называют *ви́дением* (vision). Проекты, когда они не имеют такой ясной цели, практически всегда обречены на неудачу.

Скажем, мы хотим построить средних размеров легковую машину с четырьмя дверями и довольно мощным двигателем. Первая стадия в проектировании, несомненно, состоит не в том, чтобы начинать сходу проектировать автомобиль (и все входящие в него агрегаты). Проектировщику программного продукта или программисту в подобной ситуации было бы тоже неразумно браться сразу за проектирование и программирование.

Первая стадия заключается в выяснении того, какие компоненты уже имеются на заводе или у надежных поставщиков. Найденные таким образом компоненты не обязаны в точности подходить для нового автомобиля. Есть способы приспособить ком-

поненты. Также не стоит упускать возможность повлиять на характеристики «следующего выпуска» таких компонент, чтобы сделать их более подходящими для данного проекта. Например, может быть, имеется двигатель с нужными свойствами, но с чуть-чуть недостаточной мощностью. Или мы, или поставщик, могли бы незначительно подправить двигатель, чтобы восполнить этот недостаток без изменения основного проекта. Отметим, что такая подгонка «без изменения основного проекта» вряд ли возможна, если первоначальный проект не предусматривал такую возможность. Такая подгонка, как правило, требует сотрудничества между вами и поставщиком двигателей. Проектировщик и программист находятся в схожих отношениях. В частности, полиморфные классы и шаблоны могут эффективно использоваться для подгонки. Однако, не ожидайте, что сможете произвольно менять все, что захотите, если разработчик класса не предусмотрел подобных возможностей.

Определив все стандартные компоненты, проектировщик автомобиля не бросается проектировать оптимальные новые компоненты для новой машины. Это было бы слишком дорого. Допустим, нигде нет подходящего кондиционера, а под капотом есть подходящее Г-образное пространство. Одним из решений было бы спроектировать такой Г-образный кондиционер. Однако слишком низка вероятность, что такой странный кондиционер пригодится для установки на другие автомобили — даже после подгонки. Это означает, что наш автомобильный проектировщик не сможет разделить стоимость производства такого кондиционера с другими типами автомобилей, и кондиционер будет слишком уникален, чтобы быть рентабельным. Таким образом, стоило бы спроектировать кондиционер более широкого применения — более простой формы и более пригодный для подгонки, чем Г-образный. Это, вероятно, потребует больше работы, чем разработка конкретного, хотя и странного Г-образного кондиционера, и повлечет внесение изменений в общий проект автомобиля, чтобы приспособить его к разрабатываемому более стандартному кондиционеру. Поскольку новый кондиционер разработан для более широкого применения, чем наше Г-образное чудо, вероятно, его придется немного подогнать, чтобы он в точности подошел к нашему исправленному проекту. И снова очевидна аналогия с проектировщиком программного продукта и программистом. Вместо написания специальной подпрограммы для данного проекта они могут разработать новый универсальный компонент, который будет хорошим кандидатом в стандартные средства общего назначения.

И, наконец, исчерпав потенциал стандартных компонент, мы собираем «окончательный» проект. Мы должны использовать наименьшее возможное число специально разработанных деталей, поскольку на следующий год нам снова придется проектировать новую модель, и специально разработанные детали нам, весьма вероятно, придется переделывать или выбросить. Печально, но опыт традиционного проектирования программ говорит, что не многие части системы можно хотя бы выделить как отдельные компоненты, и совсем незначительная их часть находит применение вне изначального проекта.

Я не утверждаю, что все проектировщики автомобилей так рациональны, как я обрисовал в приведенной аналогии, или что все проектировщики программных продуктов совершают упомянутые ошибки. Напротив, эту модель можно применить для создания программных продуктов. В частности, данная глава и следующая познакомят вас с тем, как данная модель реализуется в C++. Однако я утверждаю, что из-за неосязаемой природы программ при проектировании программных продуктов избежать по-

добных ошибок труднее (§ 24.3.1, § 24.3.4), а в § 23.5.3 я показываю, что от использования приведенной здесь модели людей часто удерживают корпоративные традиции.

Отметим, что эта модель разработки хороша в долгосрочной перспективе. Если ваш горизонт простирается только до ближайшего выпуска, создание и сопровождение стандартных компонент не имеет смысла. Все это будет рассматриваться как излишние затраты. Данная модель предлагается для организаций, жизни которых хватает на несколько проектов, и достаточно крупных, чтобы произвести необходимые дополнительные инвестиции в инструментарий (для проектирования, программирования и менеджмента проекта) и образование (проектировщиков, программистов и менеджеров). Наш набросок завода по производству программных продуктов (что любопытно) только масштабом отличается от практики лучших независимых программистов, которые годами создают запас приемов, проектов, инструментов и библиотек, чтобы повысить свою личную эффективность. Но в жизни, похоже, большинству организаций не удалось воспользоваться лучшими методами личной практики из-за отсутствия виденья и неспособности руководить такой практикой, разве что в очень малых масштабах.

Отметим, что неразумно ожидать от «стандартных компонент» чрезмерной универсальности. Появятся несколько международных стандартных библиотек, однако большинство компонент останутся стандартными (только) внутри одной страны, отрасли промышленности, фирмы, производственной линии, отдела, прикладной области и т. д. Мир просто слишком велик, чтобы универсальные стандарты были реалистичны, или чтобы они действительно стали основной целью всех компонент и инструментов.

Стремление к универсальности в первоначальном проекте означает создание проекта, который никогда не будут завершен. Одна из причин, почему цикл разработки является циклом, в том, что необходимо иметь работающую систему, из которой можно черпать опыт (§ 23.4.3.6).

23.4.2. Цели проектирования

Какие задачи должно решить проектирование? Одна из них, конечно, — достижение простоты, но простоты в каком смысле? Допустим, проект должен развиваться. То есть систему придется расширять, переносить, настраивать, вообще по-разному изменять, и заранее нельзя предвидеть, как именно. Следовательно, мы должны стремиться к проекту и реализации системы, которые имели бы простые средства для внесения в них изменений. Кроме того, разумно предположить, что со времени первоначального проекта до первой сдачи системы требования к системе несколько раз изменятся.

Из этого вытекает, что система должна быть спроектирована так, чтобы *оставаться* как можно проще после серии внесенных в нее изменений. Мы должны закладывать изменения в проект; то есть мы должны стремиться к:

- гибкости;
- способности к расширению;
- переносимости.

Лучше всего это достигается попытками инкапсулировать те области системы, которые вероятно будут изменяться, и обеспечением такого способа работы, когда последующие проектировщики/программисты вносят изменения в одну область, не затрагивая другие. Это обеспечивается путем определения для данной разработки ключевых

понятий и придания каждому классу исключительной ответственности за содержание всей информации, относящейся к отдельному понятию. В этом случае изменение можно внести, модифицировав только один класс. В идеале изменение одного понятия может быть произведено созданием производного класса (§ 23.4.3.5) или заданием другого аргумента шаблона. Естественно, этот идеал легче провозгласить, чем ему следовать.

Рассмотрим пример. В моделировании, связанном с метеорологическими явлениями, мы хотим показать дождевую тучу. Как нам это сделать? Мы не можем иметь общую процедуру для показа тучи, поскольку ее внешний облик зависит от ее внутреннего состояния, а это состояние должно быть ответственностью только тучи.

Первое решение проблемы — дать туче возможность самой показывать себя. Такой стиль решения приемлем во многих конкретных случаях. Однако он не универсален, потому что существует множество способов представления тучи: например, как подробной картины, как грубого силуэта или как условного значка на карте. Другими словами, то, как выглядит туча, зависит не только от нее самой, но и от окружения.

Второе решение проблемы — сделать так, чтобы туча знала о своем окружении и сама себя отображала. Это решение приемлемо для еще более широкого ряда случаев. Однако это по-прежнему не универсальное решение. Знание тучей всех деталей своего окружения нарушает установку на то, что класс должен отвечать только за одну вещь, и за каждую вещь должен отвечать какой-то один класс. Может оказаться невозможным достичь согласованности в понятии «окружение тучи», потому что, вообще говоря, внешний вид тучи зависит как от нее самой, так и от наблюдателя. Так в реальной жизни, то, как я вижу тучу, сильно зависит от того, как я на нее смотрю: невооруженным глазом, через поляризованный фильтр или с помощью экрана метеорологического радара. Кроме того наблюдателю и туче нужно принимать в расчет «общий фон» — например, относительное положение солнца. Еще больше усложняет дело добавление других объектов, таких как соседние тучи и самолеты. Чтобы окончательно усложнить жизнь проектировщику, добавим возможность одновременного присутствия нескольких наблюдателей.

Третье решение — заставить тучу (и другие объекты, такие как самолеты и солнце) описывать самих себя для наблюдателя. Это решение достаточно универсально[1]. Однако, оно может привести к повышению сложности и потере быстродействия. Например, как нам сделать так, чтобы наблюдатель понял описания, выдаваемые тучами и другими объектами?

Дождевые тучи не очень часто встречаются в программах (но для примера см. § 15.2), однако объекты, которым нужно участвовать во множестве операций ввода/вывода, встречаются постоянно. Это делает тучу значимым примером для программ вообще и проектирования библиотек в частности. Программу на C++ для схожего по логике примера можно найти в манипуляторах, используемых для форматированного потокового ввода/вывода (§ 21.4.6, § 21.4.6.3). Отметим, что третье решение не является «правильным», оно просто самое универсальное. Проектировщик должен балансировать между множеством различных потребностей, чтобы выбрать уровень универсальности и абстрагирования, подходящий для данной задачи в данной системе. Эм-

[1] Даже эта модель вряд ли будет удовлетворительной для случаев, когда требуется графика высокого качества, основанная на расчете хода лучей. Подозреваю, что для удовлетворения таких подробностей проектировщику потребуется перейти на другой уровень абстрагирования.

пирическое правило таково: для программ, рассчитанных на долгую жизнь, правильным уровнем абстракции следует считать самый универсальный из тех, что вы можете себе представить и позволить, а *не* абсолютно самый универсальный. Чрезмерная универсализация, выходящая за рамки данного проекта и человеческого понимания, может причинить вред; то есть привести к задержкам, неприемлемо низкой эффективности, неуправляемости и просто к ошибкам.

Чтобы сделать такие приемы управляемыми и экономичными, мы должны также закладывать в проект возможность повторного использования (§ 23.5.1) и не совсем забывать про эффективность (§ 23.4.7).

23.4.3. Этапы проектирования

Рассмотрим проектирование отдельного класса. Как правило, это *не* слишком хорошая идея. Концепции (понятия) *не* существуют изолированно; они определяются в контексте других концепций. Точно также , и класс не существует изолированно, а определяется вместе с логически связанными классами. Как правило, мы работаем с набором взаимосвязанных классов. Такой набор часто называют *библиотекой классов* (class library) или *компонентой* (component). Иногда все классы в компоненте составляют иерархию классов, иногда все они входят в одно пространство имен, а иногда являются просто произвольным набором объявлений (§ 24.4).

Набор классов объединяется в компоненту некоторым логическим критерием, часто благодаря общему стилю, а иногда — из-за зависимости от одних и тех же услуг. Таким образом, компонента является единицей проекта, документации, владения и, часто, повторного использования. Это не означает, что если вы пользуетесь одним классом из компоненты, то должны понимать и использовать все ее классы или иметь код всех ее классов. Наоборот, мы, как правило, бьемся за то, чтобы использовать класс лишь с минимальными затратами машинных ресурсов и человеческих усилий. Однако, чтобы пользоваться какой-либо частью компоненты, нам нужно понять определяющий ее логический критерий (прекрасно, когда он вполне ясен из документации), принятые условности, стиль, воплощенный в проекте компоненты, документацию, а также общие услуги (если таковые есть).

Итак, рассмотрим, как можно подойти к проектированию компоненты. Поскольку зачастую это непростая задача, стоит разбить ее на этапы, чтобы логичным образом сфокусироваться на разных подзадачах. Как обычно, для этого нет универсального правильного способа. Однако, вот последовательность действий, которая кое-кому помогла:

[1] Выявите понятия/классы и их самые фундаментальные взаимосвязи.

[2] Уточните классы, определив набор операций над ними.
- Классифицируйте эти операции. В частности, рассмотрите потребность в конструкторах, деструкторах и копировании.
- Рассмотрите минимальность, полноту и удобство.

[3] Уточните классы, определив их взаимозависимость.
- Рассмотрите параметризацию, наследование и воспользуйтесь зависимостью.

[4] Определите интерфейсы.
- Разделите функции на открытые и защищенные.
- Определите точный тип операций над классом.

Отметим, что это этапы итеративного процесса. Как правило, чтобы получить проект, которым удобно пользоваться при первоначальной или повторной реализации, требуется несколько проходов по указанному циклу. Одно из преимуществ хорошо выполненных анализа и абстракции данных заключается в том, что становится относительно просто перетасовывать классы даже после того, как программа написана. Впрочем, это всегда нетривиальная задача.

После этого мы реализуем классы, возвращаемся назад и пересматриваем проект, уже обогащенные опытом его реализации. В следующих подразделах я обсуждаю эти этапы один за другим.

23.4.3.1. Этап 1: выявление классов

Выявляем понятия/классы и их самые фундаментальные взаимосвязи. Ключ к хорошему проекту — смоделировать некоторый аспект «реальности» напрямую, то есть понять концепции программы как классы, представить отношения между ними четко определенным способом, таким как наследование, и повторить это многократно на разных уровнях абстракции. Но как нам найти эти концепции? Как на практике решить, какие классы нам нужны?

Лучше всего начать поиски с самого приложения, а не рыться в портфеле с абстракциями и концепциями компьютерного ученого. Послушайте того, кто собирается стать профессиональным пользователем системы, когда она будет реализована, или того пользователя, который не удовлетворен существующей системой, подлежащей замене. Обратите внимание на термины, которыми они пользуются.

Часто говорят (и часто это действительно так), что классам и объектам в программе будут соответствовать существительные. Однако, это ни в коем случае еще не конец. Глаголы могут подсказать операции над объектами, традиционные (глобальные) функции, которые предоставляют новые значения, основываясь на аргументах, или даже классы. Как пример последнего, отметим объекты-функции (§ 18.4) и манипуляторы (§ 21.4.6). Такие слова как «несколько раз проходить» или «выполнять» можно представить соответственно объектами-итераторами и объектами, представляющими транзакции в базе данных. Часто с пользой для дела классами можно представить даже прилагательные. Рассмотрим прилагательные «хранящиеся», «одновременные», «зарегистрированные» и «ограниченные». Они могут стать классами, позволяющими проектировщику или программисту выбрать подходящие атрибуты классов, которые будут спроектированы в будущем, путем указания соответствующих виртуальных базовых классов (§ 15.2.4).

Не все классы соотносятся с понятиями на уровне программы. Например, некоторые классы представляют ресурсы системы и абстракции на уровне реализации (§ 24.3.1). Также важно избегать слишком похожего моделирования старой системы. Например, мы не хотим, чтобы система управления базами данных точно воспроизводила все аспекты старой системы, существующей только для того, чтобы помогать людям управляться с перекладыванием бумаг «вручную».

Наследование используется для того, чтобы выразить общность понятий. Очень важно, чтобы оно использовалось для отражения иерархической организации, основывающейся на поведении классов, представляющих отдельные понятия (§ 1.7, § 12.2.6, § 24.3.2). Иногда это называют *классификацией* или даже *таксономией*.

Нужно активно искать общность. Универсализация и классификация — это деятельность верхнего уровня, требующая проникновения в сущность системы, чтобы дать полезные, а не сиюминутные результаты. Общая база должна представлять собой более глубокую, а не просто схожую концепцию, которая требует для своего представления меньше данных.

Отметим, что классифицировать нужно те аспекты концепций, которые мы моделируем в нашей системе, а не те, которые могут иметь смысл в других областях. Например, в математике окружность — это частный случай эллипса, но в большинстве программ окружность не нужно выводить из эллипса, или делать эллипс потомком окружности. Часто приводимые аргументы «потому что так в математике» или «потому что окружность является частным случаем эллипса» неубедительны или даже ошибочны. Дело в том, что для многих программ ключевое свойство окружности — равноудаленность ее центра от всех точек периметра. Это свойство должно поддерживаться всем поведением (всеми операциями) окружности (то есть это свойство является инвариантом, § 24.3.7.1). В свою очередь эллипс характеризуется двумя точками фокуса, которые во многих программах могут независимо друг от друга изменяться. Если эти точки фокуса совмещаются, эллипс становится похож на окружность, но это не окружность, поскольку его операции не сохраняют инварианта окружности. В большинстве систем это различие будет отражено разным набором операций для эллипса и окружности, и один набор не будет являться подмножеством другого.

Мы не просто придумываем набор классов и отношений между классами и используем его в окончательной системе. Нет, мы создаем первоначальный набор классов и их взаимоотношений, а затем многократно совершенствуем его (§ 23.4.3.5), чтобы получить набор классов достаточно универсальный, гибкий и стабильный для оказания помощи в дальнейшей эволюции системы

Лучшим инструментом для выявления первоначальных понятий/классов является классная доска. Лучший способ для их усовершенствования,— обсуждение со специалистами в области применения и несколькими друзьями. Обсуждения необходимы для развития жизнеспособного словаря и базовой системы понятий. Не многие могут сделать это в одиночку. Одним из способов развить набор полезных классов из первоначального набора кандидатов является моделирование системы, где роль классов играют проектировщики. Это может выявить абсурдность первоначальных идей, стимулировать споры об альтернативах и создать общее понимание развивающегося проекта. Эта деятельность может поддерживаться и документироваться заметками на каталожных карточках. Такие карточки, в соответствии с информацией, которую на них записывают, обычно называют CRC-карточками («Class, Responsibility, and Collaborators», «класс, ответственность, сотрудники» [Wirfs-Brock, 1990]). *Пример использования* — это описание частного использования системы. Вот простой пример использования телефонной системы: берем трубку, набираем номер, на другом конце звонит телефон, там снимают трубку. Выявление таких типичных случаев использования может оказаться очень ценным на всех стадиях разработки. Сначала описание случаев использования поможет нам понять, что мы собираемся построить. В процессе проектирования их можно использовать (например, при помощи CRC-карточек), чтобы проверить, что относительно постоянное описание системы в терминах классов и объектов действительно имеет смысл с точки зрения пользователя. Во время программирования и тестирования примеры исполь-

зования становятся источником тестов. Таким образом примеры использования обеспечивают ортогональный взгляд на систему и служат проверкой соответствия с реальностью.

Примеры использования описывают систему как (динамическую) работающую сущность. Поэтому они могут соблазнить проектировщика функциональным анализом системы и отвлечь от важной задачи по поиску полезных концепций, которые можно отобразить в классах. Упор на примерах использования, особенно у того, кто имеет опыт в структурном анализе и не имеет опыта в объектно-ориентированном программировании/проектировании, ведет к функциональной декомпозиции. Набор примеров использования — это не проект. Фокусирование на использовании системы должно уравновешиваться анализом ее структуры.

Команда разработчиков может заняться бесплодными по своей природе попытками описать *все* примеры использования. Такие ошибки дорого обходятся. Также как при поиске кандидатов в классы для системы, приходит время, когда нужно сказать: «Достаточно. Пришло время испытать, что у нас есть, и посмотреть, что получится». Только используя подходящие наборы классов и примеры использования в дальнейшем развитии, мы можем получить обратную связь, столь необходимую для построения хорошей системы. Всегда трудно понять, когда же пора остановить полезную деятельность. Особенно трудно понять, когда остановиться, если мы знаем, что впоследствии должны вернуться и завершить эту задачу.

Сколько нужно примеров? На этот вопрос нельзя ответить однозначно. Однако в каждом конкретном проекте приходит время, когда становится ясно, что большая часть обычного функционирования системы описана, а часть не столь обычного функционирования и обработки ошибок уже изучена. Значит, пора приступать к следующему этапу проектирования и программирования.

Когда вы пытаетесь оценить процент покрытия системы набором примеров использования, полезно разделить примеры на первичные и вторичные. Первичные описывают наиболее общие и «нормальные» действия, а вторичные — необычные действия и сценарии обработки ошибок. Примером вторичного примера использования может послужить вариант с телефонным звонком, когда номер занят, и оттуда звонят на этот конец. Часто говорят, что когда покрыто 80% первичных и несколько вторичных примеров, сбор примеров пора заканчивать, но поскольку мы заранее не знаем, что такое 100%, это просто эмпирическое приближенное правило. Тут много значат опыт и хороший вкус.

Упомянутые здесь концепции, операции и взаимосвязи естественно вытекают из нашего понимания области применения или появляются при дальнейшей работе над структурой классов. Они представляют собой наше фундаментальное понимание приложения. Часто они являются классификацией фундаментальных концепций. Например, «машина с раздвижной лестницей» является частным случаем пожарной машины, которая представляет собой разновидность грузовика, который, в свою очередь, представляет собой разновидность транспортного средства вообще. Подразделы § 23.4.3.2 и § 23.4.5 знакомят с несколькими способами взгляда на классы и иерархию классов с точки зрения их улучшения.

Опасайтесь наглядных схем! На определенной стадии вас попросят познакомить кого-нибудь с проектом, и вы выдадите набор диаграмм, объясняющих структуру строящейся системы. Это может оказаться очень полезным упражнением, поскольку помо-

жет сосредоточить внимание на том, что в системе важно, и заставит выразить свои идеи в терминах, понятных для других. Представление системы другим — это важный инструмент проектирования. Подготовка к презентации с целью передачи понимания людям, заинтересованным и способным к конструктивной критике, — это упражнение в концептуализации и ясном выражении идей.

Однако формальное представление проекта также очень опасно, поскольку существует сильный соблазн представить идеальную систему — систему, которую вы хотели бы суметь построить, систему, которую хотело бы получить ваше высокое начальство, — а не ту, которая у вас есть и которую вы могли бы произвести в разумные сроки. Когда конкурируют разные подходы, а начальство на самом деле не понимает «деталей» или не интересуется ими, презентация может превратиться в соревнование во лжи, где каждая команда представляет самые грандиозные прожекты ради сохранения рабочих мест. В таких случаях ясное выражение идей часто заменяется «тяжелым» жаргоном и аббревиатурами. Если вы присутствуете на такой презентации — а особенно, если вам предстоит принять решение, и вы распоряжаетесь средствами на разработку — очень важно отличить благие пожелания от реалистичного планирования. Высококачественные материалы для презентации не гарантируют качества описываемой системы. Фактически, я часто обнаруживал, что организации, сосредоточенные на реальной проблеме, когда дело доходит до представления результатов, проигрывают организациям, менее обеспокоенным производством реальных систем.

При поиске концепций для представления в качестве классов, учтите, что существуют важные свойства системы, которые нельзя представить классами. Например, надежность, производительность и способность к тестированию — важные свойства системы, которые можно измерить. Однако даже вполне объектно-ориентированная система не локализует свою надежность в объекте. Врожденные свойства системы можно определить, их можно закладывать в проект и в конце концов проверять измерением. Забота об этих свойствах должна красной нитью проходить через все классы, и ее можно отразить в правилах проектирования и реализации для отдельных классов и компонент.

23.4.3.2. Этап 2: определение операций

Уточните классы, определив набор операций над ними. Естественно, невозможно отделить выявление классов от выяснения того, какие операции понадобятся. Однако практический метод состоит в том, что поиск классов сосредотачивается на ключевых концепциях и преднамеренно не делает упора на операциях, в то время как определение операций фокусируется на нахождении полного и удобного для применения набора действий. Очень часто слишком трудно рассматривать и то, и другое одновременно, особенно когда родственные классы нужно разрабатывать вместе. Когда приходит время рассматривать и то, и другое, часто оказываются полезными CRC-карточки (§ 23.4.3).

При выяснении того, какие функции надо обеспечить, можно высказать несколько общих соображений. Я предлагаю следующую стратегию:

[1] Подумайте, как объект класса должен конструироваться, копироваться (если должен) и уничтожаться.

[2] Определите *минимальный* набор операций, которые требует представляющая класс концепция. Как правило, эти операции становятся функциями-членами (§ 10.3).

[3] Подумайте, какие операции можно добавить для удобства. Включите только несколько действительно важных. Часто эти операции становятся не членами, а «функциями-помошниками» (§ 10.3.2).

[4] Подумайте, какие операции должны быть виртуальными — то есть операциями, для которых класс может действовать как интерфейс реализации, обеспечиваемой производным классом.

[5] Подумайте, какой общности в именовании и функциональности можно достичь для всех классов компоненты.

Это — явный минимализм. Гораздо легче добавить все функции, которые могут когда-нибудь пригодиться, и сделать все операции виртуальными. Однако, чем больше функций, тем вероятнее, что они останутся неиспользованными, и тем вероятнее, что они затруднят реализацию и дальнейшее развитие системы. В частности, функции, непосредственно читающие или записывающие часть состояния объекта, часто ограничивают класс единственной стратегией реализации и жестко лимитируют возможность перепроектирования. Такие функции понижают уровень абстракции от концепции к ее реализации. Добавление функций также добавляет работы и тому, кто реализует класс, и проектировщику при следующем проектировании. *Гораздо* легче добавить функцию, когда она явно понадобится, чем удалять ее.

Смысл требования сделать функцию виртуальной явно, а не по умолчанию или как деталь реализации, заключается в том, что превращение функции в виртуальную критически влияет на использование класса и на отношения между классами. Объекты класса даже с одной виртуальной функцией имеют нетривиальную структуру по сравнению с объектами на языках вроде C или Fortran. Класс даже с единственной виртуальной функцией потенциально действует как интерфейс к еще не определенному классу, а виртуальные функции подразумевают зависимость от еще не определенных классов (§ 24.3.2.1).

Учтите, что минимализм требует от проектировщика больше работы, а не меньше.

Выбирая операции, важно сосредоточится на том, что нужно делать, а не на том, как это делать. То есть, мы должны сосредоточиться на желаемом поведении, а не на проблемах реализации.

Иногда полезно классифицировать операции над классом в соответствии с их использованием внутреннего состояния объектов:

- основные операции: конструкторы, деструкторы и копирование;
- инспекторы: операции, не изменяющие состояния объекта;
- модификаторы: операции, изменяющие состояние объекта;
- преобразования: операции, производящие объект другого типа, основываясь на значении (состоянии) объекта, к которому они прилагаются;
- итераторы: операции, служащие для доступа к хранящимся в контейнере объектам или для их использования.

Эти категории не ортогональны. Например, итератор можно спроектировать так, чтобы он был или инспектором, или модификатором. Эти категории являются просто классификацией, помогающей людям приблизиться к проектированию интерфейса классов. Естественно, возможны другие классификации. Они особенно полезны для поддержания совместимости набора классов в компоненте.

C++ обеспечивает поддержку различения инспекторов и модификаторов в форме функций-членов ***const*** и не-***const***. Аналогично, прямо поддерживаются понятия конструкторов, деструкторов, операций копирования и преобразования.

23.4.3.3. Этап 3: определение взаимозависимостей

Уточните классы, определив их взаимозависимости. Различные взаимозависимости обсуждаются в § 24.3. Ключевые зависимости для рассмотрения в контексте проектирования — это параметризация, *наследование* и отношение *использования*. Каждая из них требует рассмотрения того, что она означает для класса, отвечающего за одно из свойств системы. Отвечать определенно не значит, что класс должен содержать собственно все данные, или что его функции-члены должны непосредственно выполнять все необходимые операции. Напротив, каждый класс с единственной областью ответственности гарантирует, что большая часть работы осуществляется прямыми запросами «куда-то еще» для обработки каким-то другим классом, отвечающим за частную подзадачу. Однако учтите, что злоупотребление этим приемом может привести к неэффективному и непонятному проекту, размножив классы и объекты до такой степени, что уже никакая работа не будет выполняться без каскада предшествующих запросов на предоставление услуг. Что можно выполнить прямо здесь и сейчас, и должно быть выполнено именно так.

Необходимость рассматривать наследование и отношение использования на данном этапе проектирования (а не во время реализации) ведет напрямую от использования классов к представлению концепций. Это также подразумевает, что единицей проекта является компонента (§ 23.4.3, § 24.4), а не отдельный класс.

Параметризация — часто ведущая к использованию шаблонов — это способ сделать неявные зависимости явными, чтобы было можно представить несколько альтернативных решений, не добавляя новых концепций. Часто существует выбор: оставить что-то зависящим от контекста, представить это как ветвь в дереве наследования или воспользоваться параметром (§ 24.4.1).

23.4.3.4. Этап 4: определение интерфейсов

Определите интерфейсы. Закрытые (private) функции обычно не нужно рассматривать на данной стадии проектирования. Вопросы реализации, которые нужно рассмотреть на данной стадии, связаны с зависимостями, возникшими на этапе 3. Более того: я вывел для себя эмпирическое правило, что если не возможны по крайней мере две различные реализации класса, то с этим классом что-то не так. То есть это не представление соответствующей концепции, а просто замаскированная реализация. Во многих случаях раздумья о том, доступна ли для класса какая-нибудь форма эволюции, являются хорошим способом ответа на вопрос: «Достаточно ли независим от реализации интерфейс этого класса?»

Отметим, что открытые базовые классы и друзья являются частью открытого интерфейса класса; см. также § 11.5 и § 24.4.2. Обеспечение раздельных интерфейсов для наследования и для универсальных клиентов путем раздельного определения защищенных и открытых интерфейсов может быть очень полезным.

На данном шаге обдумываются и определяются точные типы аргументов. В идеале нужно максимально возможное число интерфейсов статически типизировать типами уровня прикладной программы; см. также § 24.2.3 и § 24.4.2.

Определяя интерфейсы, выявляйте классы, где кажется, что операции поддерживают несколько уровней абстракции. Например, некоторые функции-члены класса ***File*** могут принимать аргументы типа ***File_descriptor*** и другие строковые аргументы,

означающие имя файла. Операции с ***File_descriptor*** действуют на другом уровне абстракции, чем те, что работают с именами файлов, поэтому следует подумать, стоит ли заносить их в один класс. Может быть, лучше завести два файловых класса: один, поддерживающий понятие файлового дескриптора, и другой, поддерживающий понятие имени файла. Как правило, все операции над классом должны поддерживать один и тот же уровень абстракции. Когда это не так, следует подумать о реорганизации класса и родственных ему классов.

23.4.3.5. Реорганизация иерархии классов

На этапах 1 и 3 мы снова проверяем классы и иерархию классов, чтобы выяснить, удовлетворяют ли они нашим требованиям. Как правило, оказывается, что не удовлетворяют, и нам приходится заниматься реорганизацией, чтобы улучшить эту структуру и/или спроектировать новую реализацию.

Самые распространенные случаи реорганизации иерархии классов — вычленение общей части двух классов в новый класс и разбиение старого класса на два новых. В обоих случаях в результате получается три класса: базовый и два производных. Когда производить такую реорганизацию? Каков обычно признак того, что такая реорганизация окажется полезной?

К сожалению, на эти вопросы нет простых универсальных ответов. И в этом нет ничего удивительного, поскольку мы говорим не о незначительных деталях реализации, а об изменении базовой концепции системы. Фундаментальная — и нетривиальная — операция состоит в том, чтобы найти общность между классами и вычленить их общую часть. Точный критерий общности невозможно определить, но он должен отражать общность в концепции системы, а не просто служить удобству реализации. Признаками того, что несколько классов имеют нечто общее, что можно вычленить в общий базовый класс, служат общие модели использования, сходство наборов операций, сходство реализации, и просто то, что эти классы часто одновременно всплывают на совещаниях по проектированию. И наоборот, класс может стать хорошим кандидатом для расщепления на два, если подмножества операций этого класса имеют различные модели использования, реализованные по-разному, и если о классе можно устроить две независимые дискуссии. Иногда превращение множества схожих классов в шаблон является способом систематического представления необходимых альтернатив (§ 24.4.1).

Из-за близкого родства между классами и концепциями проблемы с организацией иерархии классов часто всплывают при обсуждении названий классов и использования их имен. Если имена классов и классификация, порожденная иерархией классов, звучат неуклюже, значит, вероятно, существует возможность улучшить иерархию. Заметьте: я хочу сказать, что два человека гораздо лучше справляются с анализом иерархии классов, чем один. Если у вас не окажется никого, с кем можно обсудить проект, то полезной альтернативой может стать написание методического руководства по проекту с использованием имен классов.

Одна из самых важных задач проектирования — обеспечить интерфейсы, которые останутся стабильными при изменении самих классов. Часто это лучше всего достигается тем, что класс, от которого зависит много других классов и функций, делается абстрактным классом, представляющим собой самые общие операции. Детали лучше всего оставить более специализированным производным классам, от которых непосредственно зависит меньшее количество функций и классов. Я скажу больше: чем

больше классов зависит от класса, тем более универсальным должен быть этот класс, и тем меньше деталей он должен предоставлять

Существует сильный соблазн — добавлять операции (и данные) в используемый многими класс. Часто это выглядит как способ сделать класс более полезным и менее нуждающимся в изменениях (в будущем). В результате получается класс с «жирным» интерфейсом (§ 24.4.3) и с членами, поддерживающими слабо связанные между собой функции. Это снова является признаком того, что при значительных изменениях в одном из других классов, которые этот класс поддерживает, его придется переделывать. Это, в свою очередь, влечет за собой переделку казалось бы независимых пользовательских и производных классов. Вместо усложнения класса, являющегося для проекта основным, обычно лучше сделать его универсальным и абстрактным. В случае необходимости особые возможности будут представлены производными классами. Для примера обратитесь к [Martin, 1995].

Такой способ приводит к иерархии абстрактных классов, где классы рядом с корнем являются самыми общими, и от них зависит большинство других классов и функций. Классы-листья представляют собой самые специализированные классы, и от них зависит очень немного частей программы. В качестве примера разберите окончательную версию иерархии ***Ival_box*** (§ 12.4.3, § 12.4.4).

23.4.3.6. Использование моделей

Собираясь написать статью, я стараюсь подыскать подходящую модель, которой можно следовать. То есть я не сразу начинаю набирать текст, а ищу публикации по схожей теме, чтобы посмотреть, не найдется ли чего-нибудь такого, что может стать образцом для моей статьи. Если выбранным образцом оказывается моя собственная статья по данной теме, я могу даже оставить части текста без изменений и добавить новую информацию только там, где этого требует логика новой статьи. Например, подобным образом написана данная книга, основанная на первой и второй редакциях. Крайняя степень такого способа написания — форма (шаблон) письма. В этом случае я просто заполняю имя и, может быть, добавляю несколько строчек, чтобы «персонализировать» послание. По сути дела, когда я пишу такие письма, я просто указываю отличия от базовой модели.

Аналогичное использование существующих систем в качестве модели для новых проектов является скорее нормой, чем исключением во всех формах творческой деятельности. Когда возможно, проектирование и программирование должны базироваться на уже проделанной работе. Это ограничивает для проектировщика степени свободы и позволяет сосредоточить внимание лишь на нескольких вопросах. Начинать большой проект «совершенно с чистого листа», может быть, замечательно, однако, стремление к возможно более точному описанию оказывает «отравляющее» действие, и в результате вы начинаете, как пьяный, блуждать среди альтернатив проектирования. Наличие модели не обременительно, и модель не требует, чтобы ей рабски следовали — она просто освобождает проектировщика, позволяя ему рассматривать в данный момент только один аспект проекта.

Учтите, что использование модели неизбежно, потому что любой проект вытекает из опыта проектировщиков. Наличие явно выраженных моделей позволяет сделать сознательный выбор, ясным образом выявить предположения, определить общий набор терминов, придать проекту первоначальную форму и повысить вероятность общности подходов среди проектировщиков.

Естественно, выбор исходной модели — само по себе важное проектное решение, и часто оно может быть принято только после исследования потенциально допустимых моделей и тщательной оценки всех альтернатив. Более того, во многих случаях модель подходит только с учетом того, что в ней необходимы серьезные изменения для приспособления к новому приложению. Проектирование программного продукта — тяжелая работа, и нам нужна всяческая помощь. Мы не должны отвергать использование моделей из-за неуместного презрения к «подражательству». Подражание — это самая искренняя форма лести, и использование для вдохновения моделей и предыдущих работ — в рамках законов о собственности и авторском праве — является допустимым приемом творческой работы во всех областях: что было хорошо для Шекспира, хорошо и для нас. Некоторые называют такое применение моделей в проектировании «повторным использованием проектов» (design reuse).

Очевидной идеей является документирование основных элементов, появляющихся во многих проектах, а также описание некоторых проблем проектирования, которые эти элементы решают, и условий, при которых их можно использовать — по крайней мере об этом стоит подумать. Для описания таких универсальных и полезных элементов проекта часто используется слово *образец* (pattern), и существует литература по документированию образцов и их использованию (например, [Gamma, 1994] и [Coplien, 1995]).

Проектировщику лучше познакомиться с распространенными образцами в данной прикладной области. Как программист, я предпочитаю образцы со связанными с ними конкретными примерами программ. Подобно большинству людей я лучше всего понимаю основную идею (в данном случае образца), когда передо мной есть конкретный пример (в данном случае фрагмент программы, иллюстрирующий применение образца). Люди, активно применяющие образцы, для общения друг с другом используют специальный набор терминов. К сожалению, это может стать своего рода жаргоном, отсекающим от понимания всех, кто не входит в круг посвященных, а, как всегда, очень важно обеспечить общение между людьми, работающими над разными частями проекта (§ 23.3), а также между проектировщиками и программистами.

Все удачные большие системы являются результатом перепроектирования некоторых меньших работающих систем. Я не знаю исключений из этого правила. Можно привести множество примеров проектов, закончившихся неудачей, над которыми мучились годами, тратили большие средства, и наконец успешно заканчивали, но спустя годы после намеченного срока. В таких проектах сначала непреднамеренно строилась неработающая система, которую потом пытались трансформировать в работающую, но в конце концов приходилось перепроектировать все по-новому, приближаясь к первоначально поставленным целям. Это говорит о том, что наивно браться за большую систему «с нуля», чтобы точно следовать новейшим идеям. Чем больше и интереснее система, к которой мы стремимся, тем важнее иметь модель, с которой можно начать. Для больших систем единственно приемлемая модель — это относительно небольшая, родственная *работающая* система.

23.4.4. Экспериментирование и анализ

Начиная разработку амбициозного проекта, мы не знаем, каков лучший способ построить систему. Часто мы даже точно не знаем, что система должна делать, поскольку частности проясняются только после попыток построения, тестирования и эксплуа-

тации системы. Не говоря уж о построении всей системы, как нам получить необходимую информацию, чтобы понять, какие проектные решения являются ключевыми, и оценить их последствия?

Мы проводим эксперимент. Мы также анализируем проект и реализацию, как только у нас появляется что анализировать. Чаще всего и с большой пользой мы обсуждаем альтернативные проект и реализацию. Во всех случаях, кроме редчайших, проектирование — это коллективная деятельность, в которой проекты развиваются путем их представления и обсуждения. Часто самым важным инструментом в проектировании выступает классная доска; без нее находящиеся в зародышевом состоянии концепции не могут развиться и стать общим достоянием проектировщиков и программистов.

Похоже, что самая распространенная форма эксперимента состоит в построении прототипа, то есть уменьшенной версии всей системы или ее части. Прототип не имеет строгих критериев быстродействия, и ресурсов оборудования, как правило, для него вполне хватает, а проектировщики и программисты получают образование, опыт и стимул к работе. Идея заключается в том, чтобы как можно быстрее произвести работающую версию, и тем самым сделать возможным исследование проекта и выбор реализации.

При хорошем исполнении этот подход может оказаться очень удачным. С другой стороны, он может служить оправданием небрежности. Трудность заключается в том, что цель прототипа может легко сместиться от «исследования альтернатив» на «получение какой-нибудь работающей системы как можно скорее». Это запросто уведет нас от интереса к внутренней структуре прототипа («в конце концов, это всего лишь прототип»), и чревато пренебрежением к проектированию ради забав с реализацией прототипа. Опасность таится в том, что подобная реализация может выродиться в худший сорт пожирателя ресурсов и кошмар при сопровождении, в то же время давая иллюзию «почти завершенной» системы. По определению прототип не имеет внутренней структуры, эффективности и инфраструктуры сопровождения, позволяющих спроецировать его на реальное приложение. Поэтому «прототип», превратившись в «почти продукт», поглощает время и энергию, которые было бы лучше потратить на сам продукт. Существует соблазн как для разработчиков, так и для руководства, превратить прототип в готовый продукт и отложить «вопросы производительности» до следующего выпуска. Такая порочная практика перечеркивает все принципы проектирования.

Родственная проблема порождается тем, что разработчики прототипа влюбляются в свои средства. Они могут забыть, что производственная система не всегда может позволить себе оплачивать стоимость этих средств, и что свободу от ограничений и формальностей, царящую в их маленьком исследовательском коллективе, нелегко поддержать для более многочисленной группы разработчиков, работающих по жесткому графику.

С другой стороны, прототип может принести неоценимую пользу. Рассмотрим разработку интерфейса. В данном случае внутренняя структура части системы, не взаимодействующей напрямую с пользователем, в самом деле не имеет значения, и нет другого реального пути получить опыт, кроме как дать пользователю возможность посмотреть и «пощупать систему руками». Другой пример — прототип, спроектированный именно для того, чтобы изучить внутреннюю работу системы. Здесь рудиментом окажется пользовательский интерфейс — вполне можно вместо реальных пользователей имитировать условных.

Создание прототипа — это вид экспериментирования. Желаемый результат — не сам прототип, а проникновение в проблему, даваемое его построением. Может быть, самым важным критерием для прототипа является то, что он должен остаться настолько незавершенным, чтобы было очевидным, что это всего лишь экспериментальный аппарат, который нельзя превратить в готовый продукт без серьезной переделки проекта и реализации. Оставление прототипа «неоконченным» поможет сфокусироваться на эксперименте и минимизировать опасность превращения прототипа в готовое изделие. Это также уменьшает соблазн попытаться строить проект продукта слишком похожим на проект прототипа, забывая о присущей ему ограниченности или не придавая ей значения. После использования прототип нужно выбросить.

Нужно помнить, что во многих случаях существуют приемы экспериментирования, альтернативные построению прототипа. Когда можно воспользоваться ими, они часто предпочтительнее, поскольку строже и требуют от проектировщика меньше времени, а от системы меньше ресурсов. Примером могут служить математические модели и различные виды симуляторов. По существу, можно говорить о непрерывном переходе от математической модели ко все более и более детализированным симуляторам, затем к частичной реализации и, наконец, к завершенной системе.

Это ведет к мысли о росте системы от первоначального проекта и реализации посредством многократных повторных проектов и реализаций. Это идеальная стратегия, но она может предъявить большие требования к средствам проектирования и реализации. К тому же, этот подход рискует быть отягощенным первоначальными проектными решениями до такой степени, что лучший проект окажется просто невозможно реализовать. На нынешний день эта стратегия, кажется, ограничивается применением в маленьких и средних системах, где большие изменения всего проекта маловероятны, а также в перепроектировании и новых реализациях после первого выпуска системы, где такая стратегия неизбежна.

Кроме экспериментов, рассчитанных на понимание возможных путей проектирования, важным источником дальнейшего развития является анализ проекта и/или собственно реализации. Например, изучение различных взаимозависимостей между классами может оказать большую помощь (§ 24.3), и нельзя пренебрегать такими традиционными инструментами разработчика реализации, как графы вызовов, измерение быстродействия и т. д.

Отметим, что спецификации (получаемые в результате анализа) и проекты подвержены ошибкам, также как и реализация. И по сути дела даже больше, поскольку они менее конкретны, зачастую определенно менее точны, их нельзя исполнить, и, как правило, они не поддерживаются изощренными средствами, сравнимыми с теми, которые доступны для тестирования и анализа реализации. Возрастающая формальность языка/обозначений при выражении проекта дает проектировщику возможность воспользоваться подобными средствами, но этого не следует делать за счет выхолащивания используемого для реализации языка программирования (§ 24.3.1). К тому же формальные обозначения могут сами внести путаницу и проблемы. Такое случается, когда: формализм плохо подходит к той практической проблеме, к которой он приложен; строгость формализма превышает математическую подготовку и зрелость проектировщиков и программистов; формальное описание системы неадекватно системе, которую оно предположительно описывает.

Проектирование по природе своей склонно к ошибкам, и его трудно поддерживать эффективными средствами. Главное в проектировании — опыт и обратная связь. Следовательно, рассматривать разработку программного продукта как линейный поступательный процесс, начинающийся с анализа и заканчивающийся тестированием — фундаментальное заблуждение. Для установления достаточной обратной связи на разных стадиях разработки и получения опыта необходимо концентрировать внимание на итеративной природе проектирования и реализации.

23.4.5. Тестирование

Программы, которые не тестировали, не работают. Такое идеальное проектирование и/или проверка программы, чтобы она заработала с первого раза, достижимы разве что в самых тривиальных случаях. Мы должны стремиться к идеалу, но не следует думать, что тестировать легко.

«Как тестировать?» — на этот вопрос нет общего ответа. Однако на вопрос «Когда тестировать?» можно дать общий ответ: как можно раньше и как можно чаще. Стратегия тестирования должна быть частью проекта и реализации, или по крайней мере должна разрабатываться параллельно с ними. Как только система может быть запущена, должно начинаться тестирование. Откладывание серьезного тестирования до тех пор, «пока не завершится реализация» — это верный рецепт срыва календарного графика и получения изъянов в конечном продукте.

Когда представляется возможность, систему следует специально проектировать так, чтобы ее было относительно легко тестировать. В частности, механизмы для тестирования часто могут встраиваться в саму систему. Иногда этого не делают из опасения получить слишком расточительную программу во время исполнения или из опасения, что избыточность, необходимая для проверки, чрезмерно расширит структуры данных. Такие страхи, как правило, неуместны, поскольку большую часть тестирующего кода и избыточности при необходимости можно убрать из программы перед выпуском системы. Иногда здесь полезен механизм утверждений (§ 24.3.7.2).

Важнее конкретных тестов является идея о том, что структура системы должна быть таковой, чтобы у нас была возможность убедить себя и/или заказчика, что мы можем устранить ошибки сочетанием статической проверки, статического анализа и тестирования. Там, где разрабатывается стратегия устойчивости к ошибкам (§ 14.9), стратегию тестирования можно разрабатывать как дополнительный и тесно связанный с ней аспект общего проектирования.

Если на стадии проектирования вопросы тестирования были совершенно забыты, то возникнут проблемы с тестированием, сроками сдачи и сопровождением. Обычно хорошим местом для начала работы над стратегией тестирования являются интерфейсы классов и взаимозависимость между классами (как описано в § 24.3 и § 24.4.2).

Обычно трудно определить, сколько нужно тестировать систему. Однако нехватка тестирования — более распространенная проблема, чем избыток. Точное количество ресурсов, которые должны быть выделены для тестирования по сравнению с проектированием и реализацией, естественно зависит от природы системы и методов ее конструирования. Однако, в качестве эмпирического правила я могу предположить, что на тестирование системы следует потратить больше времени, усилий и та-

ланта, чем на конструирование первоначальной реализации. Тестирование должно сосредоточиться на проблемах, которые могут повлечь за собой ужасные последствия, и тех, которые встречаются чаще всего.

23.4.6. Сопровождение программ

«Сопровождение программ» — это недоразумение. Слово «сопровождение» (maintenance) вызывает ассоциацию с обслуживанием техники. Но программу не нужно смазывать, заменять износившиеся части, в ней не появляются трещины, куда затекает вода, вызывая ржавчину. Программу можно скопировать один к одному и передать на большие расстояния в течение нескольких минут. Программа — это не аппаратура.

Деятельность под названием «сопровождение программ» на самом деле означает перепроектирование и повторную реализацию; таким образом она относится к обычному циклу разработки программного продукта. Когда в проекте уделяют внимание гибкости, способности к расширению и переносимости, то это как раз поможет избежать традиционных проблем с сопровождением.

Подобно тестированию, рассмотрение вопросов сопровождения не следует оставлять на потом, и не нужно отделять их от разработки в целом. В частности, важно иметь в коллективе «непрерывный переход» от лиц, занимающихся чистым проектированием к тем, кто ответственны за сопровождение. Нелегко успешно передать сопровождение новому (и, как правило, не имеющему такого опыта) коллективу без связи с первоначальными разработчиками. Когда необходимы серьезные изменения в коллективе, нужно постараться передать новым сотрудникам понимание структуры системы и ее задач. Если «обслуживающий персонал» не понимает архитектуру системы, или ему приходится догадываться о назначении компонент системы, исходя из реализации, структура системы может стремительно ухудшиться под потоком «локальных» исправлений. Документация, как правило, скорее освещает детали, чем помогает новым людям разобраться в ключевых идеях и принципах.

23.4.7. Эффективность

Дональд Кнут заметил, что «несвоевременная оптимизация — корень всех зол». Некоторые слишком хорошо усвоили этот урок и считают всякую заботу об эффективности злом. Напротив, эффективность нельзя упускать из виду при проектировании и реализации. Однако, это не значит, что проектировщик должен заботиться о «микро-эффективностях»; рассматривать нужно только эффективность «первого порядка».

Лучшая стратегия достижения эффективности — создать ясный и простой проект. Только такой проект может остаться относительно стабильным за время жизни системы и послужить базой для улучшения быстродействия. Очень важно избегать гаргантюанизма, который губит большие проекты. Слишком часто люди добавляют возможности к системе «просто на всякий случай» (§ 23.4.3.2, § 23.5.3) и заканчивают тем, что ради сохранения архитектурных излишеств вдвое или даже вчетверо увеличивают размеры программы при таком же снижении быстродействия. Хуже того, такие чрезмерно изощренные системы часто очень трудно анализировать, так что становится нелегко различить, где можно избежать затрат, а где нельзя. Таким образом отбивается всякое желание анализа и оптимизации. Оптимизация должна рождаться

в результате анализа и измерения быстродействия, а не из игр с программой. «Интуиция» проектировщика или программиста, особенно когда речь идет об относительно больших системах, — ненадежное средство в том, что касается эффективности.

Важно избегать конструкций, неэффективных по своей природе, и тех, которые для достижения приемлемого уровня быстродействия потребуют много времени и умственных усилий. Также важно минимизировать использование непереносимых конструкций и средств, поскольку при их применении проект будет обречен вечно работать со старыми (менее мощными и/или более дорогими) компьютерами.

23.5. Менеджмент

Когда это имеет хоть какой-то смысл, большинство людей делают то, за что их поощряют. В частности, если в контексте проекта вы получаете определенные рычаги для управления другими лицами и их наказания, только исключительный программист и проектировщик будет рисковать своей карьерой, чтобы перед лицом неодобрения, равнодушия или волокиты со стороны руководства делать то, что он считает правильным[1]. Из этого следует, что в организации должна быть введена система поощрений, соответствующая установленным целям проектирования и программирования. Однако, слишком часто такого не происходит: серьезного изменения в стиле программирования можно достичь лишь через соответствующее изменение стиля проектирования, а и то, и другое, чтобы быть эффективным, как правило, требует изменения в стиле менеджмента. Умственная и организационная инерция слишком легко приводит к локализации изменений, которая не соответствует глобальным переменам, требующимся для достижения успеха. Довольно характерный пример — переход на новый язык, который бы поддерживал объектно-ориентированное программирование, вроде C++, без соответствующего изменения в стратегии проектирования для получения преимуществ от объектно-ориентированного подхода (см. также § 24.2). Другой пример — переход на «объектно-ориентированное проектирование», оставаясь со старым не объектно-ориентированным языком программирования.

23.5.1. Повторное использование

Увеличение повторного использования уже готовых проектов и программ часто называют в качестве главной причины для выбора нового языка программирования или стратегии проектирования. Однако большинство организаций поощряет отдельных работников и целые коллективы, предпочитающие изобретать велосипед. Например, производительность программиста часто измеряют по числу строк кода; будет ли он писать маленькие программы, основанные на стандартных библиотеках, жертвуя своим доходом и, возможно, положением? Менеджеру могут платить в зависимости от количества людей в его подразделении; станет ли он пользоваться программами, разработанными в другом месте, когда вместо этого можно нанять еще пару программистов в свое подразделение? Фирма может получить госзаказ, где доход определяется процентом от затрат; станет ли эта фирма минимизировать свой доход, используя самые эффективные средства разработки? Стимулировать повтор-

[1] В организации, которая обращается со своими программистами, как с полными идиотами, вскоре будут работать только те программисты, которые желают и способны вести себя как полные идиоты.

ное использование трудно, но если руководство не найдет способа поощрять и вознаграждать за это, повторного использования не будет.

Повторное использование — это социальная проблема. Я могу воспользоваться чьей-то программой, если:

[1] Она работает: чтобы годиться к повторному использованию, программа должна быть пригодной к использованию вообще.

[2] Она понятна: очень важны структура программы, комментарии, документация и руководство по применению.

[3] Она может сосуществовать с программами, написанными без учета необходимости взаимодействия с ней.

[4] Она сопровождается (или я хочу сам сопровождать ее; как правило, я не хочу).

[5] Она экономична (могу ли я разделить затраты на разработку и сопровождение с другими пользователями?).

[6] Я могу ее найти.

К этому мы можем добавить, что компонента бывает не готова к повторному использованию, пока кто-то не «использует ее повторно». Задача подгонки компоненты к новой среде, как правило, ведет к усовершенствованию операций, универсализации ее поведения и улучшению ее способности сосуществовать с другими программами. Пока все эти действия не будут проделаны хотя бы раз, даже компоненты, спроектированные и реализованные с величайшим вниманием, будут иметь неожиданные острые углы.

Мой опыт говорит, что условия, необходимые для повторного использования, возникнут только тогда, когда кто-то займется этим вплотную. В маленьких коллективах это, как правило, означает, что кто-то (намеренно или случайно) становится «хранителем» общих библиотек и документации. В больших организациях это означает, что группе или отделу поручается собирать, строить, документировать, популяризировать и обслуживать программы для использования несколькими группами.

Важность таких групп по «стандартным компонентам» нельзя переоценить. Заметьте, что в первом приближении система отражает организацию, произведшую ее. Если организация не имеет механизма поддержки и поощрения кооперации и совместного использования, кооперация и совместное использование станут редкостью. Группы по стандартным компонентам должны активно продвигать свои компоненты. Это подразумевает, что наличие традиционной хорошей документации необходимо, но не достаточно. Кроме нее группа стандартных компонент должна выпускать руководства по применению и другую информацию, которая позволит потенциальному пользователю найти компоненту и понять, зачем она может пригодиться. Это означает, что группа по стандартным компонентам должна заняться деятельностью, традиционно ассоциирующейся с маркетингом и повышением квалификации сотрудников.

По возможности члены этой группы должны работать в тесном контакте с производителями приложений, поскольку только тогда они смогут понять потребности пользователей и осознать возможности совместного использования компонент в разных разработках. Из этого следует, что в таких организациях должны быть обеспечены консультации с группой по стандартным компонентам и передача информации в нее и из нее.

Успех групп по стандартным компонентам должен измеряться успехом их клиентов. Если их успех оценивается просто количеством средств и инструментов, в полезности которых удалось убедить организации, такие группы деградируют и стано-

вятся просто распространителями коммерческих программ и поборниками вечных переделок.

Не все программы нуждаются в повторном использовании, и способность к повторному использованию не является универсальной характеристикой. Утверждение, что компонент «годится к повторному применению», не означает, что его повторное использование не потребует серьезной работы. В большинстве случаев переход к другой среде разработки потребует значительных усилий. В этом отношении возможность повторного использования очень напоминает переносимость. Важно отметить, что повторное использование — это результат проектирования, направленного на повторное использование, уточнения компонент, основанного на опыте, и сознательных усилий, направленных на поиск компонент, годящихся для повторного использования. Повторное использование не возникает по волшебству из бездумного применения особенностей нового языка или приемов кодирования. Такие особенности C++, как классы, виртуальные функции и шаблоны, позволяют выразить проект так, что повторное использование становится легче (и следовательно более вероятно), но сами по себе эти средства не гарантируют возможности повторного использования.

23.5.2. Масштаб

Отдельная личность или организация легко поддаются стремлению «делать вещи правильно». В официальном переложении это часто звучит как «разработка и строгое соблюдение соответствующих процедур». В обоих случаях здравый смысл может стать первой жертвой искреннего и пылкого стремления улучшить принятый ход вещей. К несчастью, когда теряется здравый смысл, нет пределов непреднамеренно наносимому вреду.

Рассмотрим стадии процесса разработки, перечисленные в § 23.4, и этапы проектирования, перечисленные в § 23.4.3. Относительно легко переработать эти соображения в методику проектирования, где каждый этап определен точно и имеет четко обозначенный вход и выход, а также полуформальное обозначение для выражения этого входа и выхода. Можно разработать список проверок для гарантии того, что методики придерживаются, или выработать механизмы для обязательного соблюдения процедур и условных обозначений. Далее, глядя на приведенную в § 24.3 классификацию зависимостей, можно объявить, что одни зависимости хороши, а другие плохи, и обеспечить механизм анализа, гарантирующий, что эти соображения применяются во всем проекте. Чтобы завершить это «укрепление дисциплины» в процессе производства программного продукта, нужно определить стандарты документации (включая орфографические и грамматические правила, а также способ набора) и унифицировать облик программы (включая спецификации на то, какие особенности языка можно применять, а какие нельзя, какими видами библиотек можно пользоваться, а какими нельзя, какие делать отступы, как называть функции, переменные, типы и т. д.).

Многое из этого может способствовать успеху проекта. Во всяком случае, было бы безрассудно, не имея более или менее определенной и довольно жесткой всесторонне продуманной схемы, браться за проектирование системы, которая в конечном итоге будет иметь десять миллионов строк кода, в разработке которой будут заняты сотни людей, а лет через десять тысячи людей будут заниматься сопровождением и поддержкой.

К счастью, большинство систем не попадает в эту категорию. Однако, как только принята идея, что такой-то метод проектирования, кодирования и ведения докумен-

тации «правилен», возникает желание применять его повсеместно и неукоснительно. В небольших проектах это может привести к нелепым ограничениям и бессмысленным затратам. В частности, такой подход ведет к тому, что в качестве оценки успеха будет рассматриваться передвижение бумаг и заполнение форм, а не продуктивная работа. Если это произошло, настоящие программисты и проектировщики покинут проект, и их заменят бюрократы.

Поскольку в программистском сообществе уже имели место такие смешные ошибки в приложении метода проектирования (надо надеяться, вполне обоснованные), подобные неудачи порой служат оправданием для несоблюдения никаких формальностей в процессе разработки. Это, в свою очередь, ведет к неразберихе и неудачам, которые стремился предотвратить предыдущий метод.

Действительная проблема заключается в том, чтобы для разработки конкретного проекта выбрать соответствующую степень формальности. Не думайте, что вы легко найдете решение этой проблемы. Для маленького проекта годится по сути дела любой подход. Хуже того, похоже, что и для больших проектов годится по сути дела любой подход — как бы плохо он ни был задуман и как бы жестоко ни относился к вовлеченным в него людям — если вы готовы потратить на разработку неограниченное количество времени и денег.

Ключевая проблема при любой разработке заключается в том, как сохранить целостность проекта. Эта проблема растет в нелинейной зависимости от масштаба. Только отдельная личность или небольшая группа может сохранять понимание общих задач всего проекта. Большинство же должно затрачивать столько времени на его составные части, технические детали, повседневное администрирование и т. п., что задачи проекта в целом легко забываются или подчиняются более локальным и непосредственно важным целям. Отсутствие личности или группы с явно поставленной задачей поддерживать целостность проекта часто означает неудачу. Еще один рецепт неудачи — не дать возможности этой личности или группе повлиять на проект в целом.

Отсутствие согласованной долговременной цели приносит проекту и организации больше вреда, чем какой-либо частный недостаток. Работа нескольких личностей должна заключаться в формулировании такой общей цели, в постоянном размышлении о ней, в написании ключевых документов по проекту в целом и в том, чтобы помочь другим держать эту цель в голове.

23.5.3. Личности

Описанный здесь метод проектирования выводит на первое место умелых проектировщиков и программистов. Таким образом, выбор проектировщиков и программистов имеет критическое значение для успеха организации.

Руководители часто забывают, что организация состоит из личностей. Распространено мнение о том, что все программисты одинаковы и взаимозаменяемы. Подобное заблуждение может развалить организацию, приведя к изгнанию самых эффективных личностей, и обрекая оставшихся на работу гораздо ниже своих способностей. Личности взаимозаменяемы, только если им не давать воспользоваться своими способностями выше того абсолютного минимума, которого требует данная задача. Таким образом, миф о взаимозаменяемости негуманен и по сути своей расточителен.

Большинство способов измерения производительности труда программиста поощряет расточительную практику и не принимает в расчет решающего индивидуального вклада. Самый наглядный пример — относительно широко распространенная практика подсчета производительности труда по числу строк кода, страниц документации, проведенных тестов и т. п. Такие графики хороши на картинках, но имеют весьма отдаленное отношение к реальности. Например, если продуктивность измеряется числом строк кода, успешное повторное использование оказывает отрицательное влияние на показатели продуктивности программиста. Успешное применение лучших принципов для перепроектирования большого фрагмента программы, как правило, производит тот же эффект.

Измерить качество произведенной работы гораздо труднее, чем валовое количество, и все же отдельных личностей и группы нужно награждать, основываясь на качестве их результатов, а не на грубых количественных показателях. К сожалению, разработка практических способов измерить качество — насколько я понимаю — едва ли ведется Кроме того, показатели, описывающие состояние проекта не полностью, имеют тенденцию искажать действительность. Люди приспосабливаются к установленным промежуточным срокам и привыкают улучшать индивидуальную и коллективную производительность, определяемую показателями. Прямой результат этого состоит в уменьшении целостности системы и производительности труда. Например, если определяется, сколько ошибок должно быть ликвидировано в срок (или сколько должно остаться), мы увидим, что в срок укладываются за счет понижения быстродействия или увеличения требований к аппаратуре, необходимой для работы системы. И наоборот, если оценивается только быстродействие, количество ошибок несомненно возрастет, когда разработчики будут стараться оптимизировать систему для скорейшего прохождения данного теста. Отсутствие хороших и понятных показателей качества выдвигает высокие требования к технической квалификации руководителей, но альтернатива этому — систематическая тенденция награждать за любую бурную деятельность, а не за реальное продвижение к цели. Не забывайте, что руководители — тоже люди. Им нужны по крайней мере те же знания о новых методах, что и людям, которыми они руководят.

Как и в других областях, при разработке программ мы должны рассматривать долгосрочную перспективу. Совершенно невозможно судить о производительности индивидуума по его работе в течение года. Однако, большинство работников имеют достаточно долгий послужной список, по которому можно надежно оценить техническую зрелость работника и его производительность в последнее время. Отказ от учета такого списка — как это делается там, где личностей рассматривают как взаимозаменяемые винтики — оставляет руководителей на милость сбивающих с толку количественных показателей.

Одно из последствий «долгосрочного» подхода и отказа от создания «школы управления взаимозаменяемыми идиотами» заключается в том, что личностям (и разработчикам, и руководителям) нужно дольше врастать в интересную сложную задачу. Это отбивает охоту перепрыгивать с одной работы на другую и переходить из одного отдела в другой «из соображений карьеры». Целью должна быть низкая текучесть кадров как среди технического персонала, так и среди ключевых руководителей. Ни один руководитель не может добиться успеха без взаимопонимания с главными проектировщиками и программистами и без новых знаний в соответствующей технической области. И наоборот, никакая группа проектировщиков и разработчиков не может в конечном итоге добиться успеха без поддержки со стороны компетен-

тных руководителей и без хотя бы минимального выхода за рамки узкотехнического понимания проблемы, над которой они работают.

Там где требуются нововведения, старшие технические работники, аналитики, проектировщики, программисты и т. п. сталкиваются с критически важной и трудной ролью по ознакомлению с новыми методами. Эти люди должны изучить новую технику и во многих случаях отказаться от старых привычек. Это нелегко. Такие люди, как правило, внесли большой личный вклад в старые методы работы и опираются на свои успехи и репутацию, достигнутые при помощи этих методов. То же самое относится и ко многим техническим руководителям.

Естественно, часто такие люди боятся перемен. Это может привести к завышенной оценке сложности, связанной с нововведениями, и неохотному ознакомлению с проблемами старого подхода. Так же естественно, что люди, ратующие за перемены, склонны к завышению выгоды от новых методов и недооценке проблем, связанных с изменениями. Эти две группы личностей *должны* общаться, научиться говорить на одном языке и помочь друг другу выработать модель перехода. Иначе — организационный паралич и уход наиболее способных личностей из обеих групп. Обе стороны должны помнить, что самые преуспевшие «консерваторы» — это вчерашние «радикалы». Получив возможность приобрести опыт без унижения своего достоинства, более опытные программисты и проектировщики могут стать самыми надежными и успешными проводниками перемен. Их здравый скептицизм, знание запросов пользователей, знакомство с организационными барьерами могут оказаться незаменимы. Те, кто ратует за скорые и радикальные перемены, должны понять, что чаще всего необходим такой переход к новой технике, который требует постепенного привыкания. И наоборот, те, у кого нет желания видеть перемены, должны найти такие области, где перемены не нужны, а не вести ожесточенные арьергардные бои в областях, где новые требования уже значительно изменили условия достижения успеха.

23.5.4. Гибридное проектирование

Введение в организацию новых методов работы может быть болезненным. Потрясения для организации и работающих в ней людей могут быть значительны. В частности, резкие перемены, когда на следующий же день опытные работники «старой школы» превращаются в новичков в «новой школе», как правило, неприемлемы. Однако серьезные цели редко достигаются без перемен, а значительные перемены, как правило, связаны с риском.

C++ разрабатывался для того, чтобы снизить риск, позволяя постепенно привыкать к нововведениям. Хотя и очевидно, что величайшие выгоды от C++ достигаются посредством абстракции данных, объектно-ориентированного программирования и объектно-ориентированного проектирования, совсем не очевидно, что скорейший способ для достижения этих целей — радикально порвать с прошлым. Порой такой «чистый» разрыв достижим. Но чаще желание улучшений смягчается — или должно смягчаться — тревогой о том, как справиться с переходом. Учтите следующие соображения:

- Проектировщикам и программистам нужно время, чтобы приобрести новые навыки.
- Новые программы должны взаимодействовать со старыми.
- Старые программы нужно сопровождать (часто неопределенно долго).

- Работу над существующими проектами и программами нужно завершить (в срок).
- Средства, поддерживающие новые приемы, нужно приспособить к местному окружению.

Эти факторы естественным образом ведут к гибридному стилю проектирования — даже там, где у проектировщиков нет такого намерения. Первые два пункта легко недооценить.

Поддерживая несколько парадигм программирования, C++ облегчает переход организации на новые методы благодаря следующим обстоятельствам:

- Во время изучения C++ программисты могут продолжать продуктивно работать.
- C++ может принести значительные выгоды в бедном инструментами окружении.
- Фрагменты программ на C++ могут взаимодействовать с кодом на C и других традиционных языках.
- C++ имеет большое совместимое с C подмножество.

Идея заключается в том, что программист может перейти к C++ от традиционных языков, понемногу осваивая C++, и в то же время сохраняя традиционный (процедурный) стиль программирования. Потом можно воспользоваться абстракцией данных. И наконец — когда язык и связанные с ним средства освоены — можно переходить на объектно-ориентированное и обобщенное программирование. Заметьте, что хорошо разработанную библиотеку гораздо легче использовать, чем проектировать и реализовать, поэтому новичок может воспользоваться передовыми методами абстрагирования даже на ранних стадиях освоения.

Идея постепенного обучения объектно-ориентированному проектированию, объектно-ориентированному программированию и C++ поддерживается возможностями C++ смешивать программы на C++ с кодом на языках, не поддерживающих те средства C++, которые касаются абстракции данных и объектно-ориентированного программирования (§ 24.2.1). Многие интерфейсы можно просто оставить процедурными, поскольку превращение их во что-нибудь более сложное не принесет непосредственных выгод. Для многих библиотек это уже сделано поставщиком библиотеки, так что программист на C++ может остаться в неведении о действительном языке реализации. Использование библиотек, написанных на таких языках, как C, явилось для C++ первой и очень важной формой повторного использования.

Следующая стадия — необходимая только там, где действительно нужна более изощренная техника — представить процедуры, написанные на языках C и Fortran, в виде классов, инкапсулировав структуры данных и функции в интерфейсные классы C++. Простой пример возвышения семантики от уровня «процедуры плюс структура данных» до уровня абстракции данных — это строковый класс из § 11.12. Там инкапсуляция представления символьных строк в стиле C и стандартные функции C для работы со строками используются для того, чтобы создать строковый тип, гораздо более простой в использовании.

Подобный прием можно использовать, чтобы «втиснуть» встроенный или отдельный тип в иерархию классов (§ 23.5.1). Это позволяет проектам на C++ развиться до использования абстракции данных и иерархии классов в присутствии программ, написанных на языках, в которых эти концепции отсутствуют, и даже при том ограничении, что к результирующей программе должны обращаться из процедурных языков.

23.6. Аннотированная библиография

Эта глава лишь касается проблем, связанных с проектированием программных продуктов и с соответствующими проблемами менеджмента. По этой причине я привожу краткую аннотированную библиографию. Расширенную аннотированную библиографию можно найти в [Booch, 1994].

[Anderson, 1990] Bruce Anderson and Sanjiv Gossain: *An Iterative Design Model for Reusable Object-Oriented Software.* Proc. OOPSLA'90, Оттава, Канада. Описание модели итеративного проектирования и перепроектирования со специфическими примерами и обсуждением опыта.

[Booch, 1994] Grady Booch: *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings. 1994. ISBN 0-8053-5340-2. Содержит подробное описание проектирования, особых методов проектирования с графическими обозначениями и несколько подробных примеров проектирования, выраженных на C++. Это превосходная книга, которой данная глава многим обязана. Она предоставляет более глубокий взгляд на многие вопросы, затронутые в данной главе.

[Booch, 1996] Grady Booch: *Object Solutions.* Benjamin/Cummings. 1996. ISBN 0-8053-0594-7. Описывает разработку объектно-ориентированных систем с точки зрения проблем менеджмента. Содержит подробные примеры программ на C++.

[Brooks, 1982] Fred Brooks: The Mythical Man Month. Addison-Wesley. 1982. Эту книгу все должны перечитывать каждые пару лет. Предостережение от высокомерия. Книга немного устарела в техническом отношении, но ничуть не утратила своей актуальности в том, что относится к человеческому фактору, организациям и масштабированию. Переиздана с дополнениями в 1997, ISBN 1-201-83595-9.

[Brooks, 1987] Fred Brooks: No Silver Bullet. IEEE Computer, Vol. 20, No. 4, April 1987. Краткое изложение подходов к разработке программ промышленного масштаба с очень нужными предостережениями против веры в чудесные средства («серебряные пули»).

[Coplien, 1995] James O. Coplien and Douglas C. Shmidt (editors): *Pattern Languages of Program Design.* Addison-Wesley. 1995. ISBN 1-201-60734-4.

[Gamma, 1994] Eric Gamma, et. al.: *Design Patterns.* Addison-Wesley. 1994. ISBN 0-2-1-63361-2. Перечень практических приемов для создания адаптируемого программного обеспечения с возможностью повторного использования, содержит нетривиальные хорошо объясненные примеры. В книге можно найти много примеров кода на C++.

[DeMarco, 1987] T. DeMarco and T. Lister: *Peopleware.* Dorset House Publishing Co. 1987. Одна из немногих книг, уделяющих большое внимание роли людей в производстве программного продукта.

Обязательна для всех руководителей. Написана достаточно гладко для чтения перед сном. Противоядие от многих глупостей.

[Jacobson, 1992] Ivar Jacobson et. al.: *Object-Oriented Software Engineering.* Addison-Wesley. 1992. ISBN 0-201-54435-0. Полное и практическое описание разработки промышленных программ с упором на случаи использования (§ 23.4.3.1). Не совсем правильно оценивает C++, описывая его в варианте «урожая 1987 г.».

[Kerr, 1987] Ron Kerr: *A Materialistic View of the Software «Engineering» Analogy.* In SIGPLAN Notices, March 1987. Использование аналогий в данной и следующей главах многим обязано этой статье и изложенным в ней представлениям, а также предшествующим беседам с Роном.

[Liskov, 1987] Barbara Liskov: *Data Abstraction and Hierarchy.* Proc. OOPSLA'87 (Addendum). Orlando, Florida. Обсуждение того, как использованием наследования можно скомпрометировать абстракцию данных. Отметим, что C++ имеет специальные средства для избежания подобных проблем (§ 24.3.4).

[Martin, 1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method.* Prentice-Hall. 1995. ISBN 0-13-203837-4. Показывает, как достаточно систематическим образом переходить от постановки задачи к коду на C++. Представляет альтернативные проекты и принципы выбора между ними. Более практична и конкретна, чем большинство других книг по проектированию. Содержит подробные примеры программ на C++.

[Parkinson, 1957] C. N. Parkinson: *Parkinson's Law and other Studies in Administration.* Houghton Mifflin. Boston. 1957. Одно из самых забавных и язвительных описаний тех бед, которые несет с собой административный процесс.

[Meyer, 1988] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall. 1988. Стр. 1–64 и 323–324 дают хорошее представление об одном взгляде на объектно-ориентированное программирование и проектирование и содержат много здравых практических советов. Остальная книга описывает язык Eiffel. Наблюдается тенденция запутать Eiffel универсальными принципами.

[Shlaer, 1988] S. Shlaer and S. J. Mellor: *Object-Oriented Systems Analysis and Object Lifecycles.* Yourdon Press. ISBN 0-13-629023-X and 0-13-629940-7. Представляет взгляд на анализ, проектирование и программирование, сильно отличающийся от того, что изложен здесь и воплощен в C++, но делает это с использованием терминологии, делающей этот взгляд похожим на наш.

[Snyder, 1986] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages.* Proc. OOPSLA'86. Portland, Oregon. Be-

роятно, первое хорошее описание взаимосвязи между инкапсуляцией и наследованием. Содержит также интересное обсуждение некоторых понятий множественного наследования.

[Wirfs-Brock, 1990] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener: Designing Object-Oriented Software. Prentice Hall. 1990. Описывает метод антропоморфического проектирования, основанный на ролях с использованием CRC-карточек (Classes, Responsibilities, and Collaboration, классы, ответственности и сотрудничество). Если не сам метод, то по крайней мере текст имеет склонность к Smalltalk.

Русские переводы:

[Booch, 1994] Г. Буч. *Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е издание.* СПб. «Невский Диалект». 1998.

[Parkinson, 1957] С. Н. Паркинсон. *Законы Паркинсона.* Минск. «Поппури». 1997.

23.7. Советы

[1] Поймите, чего вы хотите добиться; § 23.3.

[2] Всегда помните, что разработка программного продукта — человеческая деятельность; § 23.2, § 23.5.3.

[3] Доказательство по аналогии — это обман; § 23.4.

[4] Поставьте себе четкие и осязаемые цели; § 23.4.

[5] Не применяйте технические приемы для решения социальных задач; § 23.4.

[6] В проектировании и в обращении с людьми учитывайте деятельность за большой период времени; § 23.4.1, § 23.5.3.

[7] Нет нижнего предела размера программ, для которых имеет смысл создать проект, прежде чем начинать писать код; § 23.2.

[8] Проектируйте процесс так, чтобы поощрять обратную связь; § 23.4.

[9] Не путайте бурную деятельность с продвижением к цели; § 23.3, § 23.4.

[10] Не обобщайте больше, чем нужно, больше, чем вам позволяет опыт, и больше, чем вы можете протестировать; § 23.4.1, § 23.4.2.

[11] Представляйте основные понятия в виде классов; § 23.4.2, § 23.4.3.1.

[12] Существуют свойства системы, которые нельзя представить в виде классов; § 23.4.3.1.

[13] Представляйте иерархические взаимоотношения между понятиями в виде иерархии классов; § 23.4.3.1.

[14] Активно ищите общность в понятиях прикладной области и реализации и представляйте найденные более общие понятия в виде базовых классов; § 23.4.3.1, § 23.4.3.5.

[15] Классификация в других областях не обязательно является полезной классификацией в модели наследования прикладной программы; § 23.4.3.1.

[16] Проектируйте иерархию классов, основываясь на поведении и инвариантах; § 23.4.3.1, § 23.4.3.5, § 24.3.7.1.

[17] Рассмотрите случаи использования; § 23.4.3.1.
[18] Рассмотрите возможность использования CRC-карточек; § 23.4.3.1.
[19] Используйте в качестве модели существующие системы, это послужит источником вдохновения и хорошей отправной точкой; § 23.4.3.6.
[20] Опасайтесь конструирования на уровне наглядных схем; § 23.4.3.1.
[21] Отбросьте прототип до того, как он станет помехой; § 23.4.4.
[22] Закладывайте в проект возможность изменения; сосредоточьтесь на гибкости, способности к расширению, переносимости и повторному использованию; § 23.4.2.
[23] Сосредоточьтесь на проектировании компонент; § 23.4.3.
[24] Воспользуйтесь классами для представления основных понятий; § 23.4.3.1.
[25] Проектируйте стабильность при неизбежности изменений; § 23.4.2.
[26] Стабилизируйте проект, сделав интенсивно используемые интерфейсы минимальными, общими и абстрактными; § 23.4.3.2, § 23.4.3.5.
[27] Не давайте проекту разрастись. Не добавляйте свойств «просто на всякий случай»; § 23.4.3.2.
[28] Всегда рассматривайте альтернативные представления класса. Если нет альтернативных представлений, то класс, по всей вероятности, не представляет собой чистого понятия; § 23.4.3.4.
[29] Многократно пересматривайте и уточняйте проект и реализацию; § 23.4.3.4.
[30] Для тестирования и анализа проблемы, проекта и реализации пользуйтесь самыми лучшими доступными средствами; § 23.3, § 23.4.1, § 23.4.4.
[31] Экспериментируйте, анализируйте и тестируйте как можно раньше и как можно чаще; § 23.4.4, § 23.4.5.
[32] Не забывайте об эффективности; § 23.4.7.
[33] Уровень формальности должен соответствовать масштабу проекта; § 23.5.2.
[34] Убедитесь, что кто-то отвечает за проект в целом; § 23.5.2.
[35] Документируйте, пропагандируйте и поддерживайте повторно используемые компоненты; § 23.5.1.
[36] Документируйте задачи и принципы наравне с деталями; § 23.4.6.
[37] В качестве части документации напишите руководство для новых разработчиков; § 23.4.6.
[38] Награждайте и поощряйте повторное использование проектов, библиотек и классов; § 23.5.1.

24

Проектирование и программирование

Пусть это будет просто:
просто, как только можно,
но не проще.
— А. Эйнштейн

Проектирование и язык программирования — классы — наследование — проверка типов — программирование — что представляют классы? — иерархия классов — зависимости — включение — включение и наследование — альтернативы проектирования — использование отношений — запрограммированные отношения — инварианты — утверждения — инкапсуляция — компоненты — шаблоны — интерфейсы и реализации — советы.

24.1. Обзор

В этой главе рассматривается, каким образом языки программирования вообще и C++ в частности могут поддержать проектирование:

- § 24.2 Фундаментальная роль классов, иерархии классов, проверки типа и собственно программирования.
- § 24.3 Использование классов и иерархии классов, фокусирующееся на взаимозависимости между разными частями программы.
- § 24.4 Понятие *компоненты*, которая является основной единицей проектирования, и некоторые практические размышления о том, как выражать интерфейсы.

Более общее рассмотрение вопросов проектирования приведено в главе 23, а различные применения классов подробно рассматриваются в главе 25.

24.2. Проектирование и язык программирования

Если бы мне пришлось строить мост, я бы серьезно задумался, из какого материала его построить. Сам проект моста в большой степени зависит от выбора материала и наоборот. Разумный проект каменного моста отличается от разумного проекта стального моста, деревянного моста и т. п. Не думаю, что я смог бы выбрать правильный материал, ничего не зная о различных материалах и их использовании. Естественно, чтобы спроектировать деревянный мост, вам не нужно быть искусным плотником, но чтобы выбрать между деревом и сталью для постройки моста, нужно знать основные

сведения о деревянных конструкциях. Более того, несмотря на то, что для проектирования моста вам самому не нужно быть хорошим плотником, вам понадобится тонкое знание свойств дерева и нравов плотников.

Аналогично, чтобы выбрать язык для некоторого программного продукта, вам нужно быть знакомым с несколькими языками, а чтобы успешно спроектировать часть программного продукта, необходимо довольно хорошо знать выбранный для реализации язык — даже если сами вы не напишете ни строчки программы. Хороший проектировщик мостов учитывает свойства материалов и использует их для улучшения проекта. Аналогично, хороший проектировщик программных продуктов строит проект, полагаясь на сильные стороны языка реализации, и — насколько возможно — избегает такого использования данного языка, которое может вызвать затруднения у тех, кто реализует проект.

Кто-то может подумать, что эта чувствительность к выбору языка приходит естественным путем, когда в разработке участвует единственный проектировщик/программист. Однако даже в таких случаях программист может соблазниться не тем языком из-за недостаточного опыта или чрезмерного уважения к стилю программирования, привычного для совершенно других языков. Когда проектировщик отличается от программиста — и особенно, если они не воспитаны в рамках некой общей культуры — можно говорить не о вероятности, а предопределенности того, что в получившейся системе будут ошибки, а сама система получится неизящной и неэффективной.

Так что же значит для проектировщика язык программирования? Он может предоставить механизмы, которые позволят выразить проектные понятия прямо на языке программирования. Это облегчает реализацию, поддержание соответствия между проектом и реализацией, лучшее взаимопонимание между проектировщиками и исполнителями реализации, а также позволяет создать лучшие инструментальные средства в помощь тем и другим.

Например, большинство методов проектирования тщательно изучают взаимозависимости между разными частями программы (обычно стремясь свести их к минимуму и сделать так, чтобы они были четко определены и понятны). Язык, поддерживающий явные интерфейсы между разными частями программы, может отразить соответствующие понятия проекта. Он может гарантировать, что существуют только ожидаемые зависимости. Когда многие зависимости выражены явно в коде, написанном на таком языке, можно предоставить средства, которые бы читали программу и составляли карту зависимостей. Это облегчает работу проектировщиков и всех тех, кому нужно разобраться в структуре программы. Такой язык программирования как C++, можно использовать для уменьшения разрыва между проектированием и программированием, а следовательно, для уменьшения возможной путаницы и устранения недоразумений.

Ключевое понятие C++ — класс. Класс в C++ — это тип. Наряду с пространствами имен классы также являются механизмом сокрытия информации. Программы можно определить в терминах типов, определяемые пользователем, и иерархии таких типов. И встроенные типы, и типы, определяемые пользователем, подчиняются правилам статической проверки типов. Виртуальные функции обеспечивают механизм динамического связывания без нарушения правил статической типизации. Шаблоны поддерживают проектирование параметризованных типов. Исключения обеспечивают более регулярный способ обработки ошибок. Данные особенности C++ можно использовать,

не внося дополнительных затрат времени и памяти по сравнению с программами на C. Это основные свойства C++, и проектировщик должен их понять и учитывать. Кроме того, сильно повлиять на проектирование могут общедоступные основные библиотеки — такие как библиотеки матриц, интерфейсы к базам данных, библиотеки графического интерфейса пользователя и библиотеки поддержки параллелизма.

Страх перед новым ведет к неоптимальному использованию C++. К этому же ведет неправильное применение опыта, накопленного в других языках, системах и прикладных областях. Ухудшить проект могут также неадекватные средства проектирования. Стоит упомянуть пять ошибок проектирования, которые не позволяют воспользоваться преимуществами языка и учесть его ограничения:

[1] Отказ от классов; в результате проект выражается так, что при реализации приходится пользоваться только подмножеством языка, совместимым с C.
[2] Отказ от производных классов и виртуальных функций, использование только подмножества языка, связанного с абстракцией данных.
[3] Отказ от статической проверки типов; в результате проект выражается так, что приходится реализовать его, симулируя динамическую проверку типов.
[4] Отказ от программирования и выражение системы таким образом, чтобы исключить программистов.
[5] Отказ от всего, кроме иерархии классов.

Эти ошибки характерны соответственно для проектировщиков:

[1] имеющих опыт работы с C, традиционным CASE-проектированием и структурированным проектированием;
[2] имеющих опыт работы с Ada83 и Visual Basic или активно использовавших абстракцию данных;
[3] имеющих опыт работы со Smalltalk или Lisp;
[4] имеющих опыт работы в нетехнических или сугубо специализированных областях;
[5] сильно увлекающихся упором на «чистое» объектно-ориентированное программирование.

В каждом случае следует задуматься, правильно ли выбран язык программирования, правильно ли выбран метод проектирования, и освоил ли проектировщик имеющиеся средства проектирования.

В таком несоответствии нет ничего необычного или постыдного. Просто это несоответствие приводит к неоптимальному проекту и приносит программистам трудности, которых можно было бы избежать. То же самое можно сказать и о проектировщиках, когда концептуально метод проектирования оказывается значительно беднее, чем идеология C++. Поэтому по мере возможности следует избегать таких несоответствий.

Следующие рассуждения построены в виде ответов на возражения, поскольку они часто возникают в реальной жизни.

24.2.1. Отказ от классов

Рассмотрим проектирование без применения классов. Конечная программа на C++ будет в основном совпадать с программой на C, которая получилась бы в том же процессе проектирования — а эта программа будет опять же примерно соответствовать программе на COBOL, которая получилась бы при следовании этому процессу проектирования. По сути проект получается «независимым от языка программирования»

ценой того, что программист вынужден пользоваться общим подмножеством C и COBOL. Этот подход действительно имеет свои преимущества. Например, строгое разделение данных и кода, что позволяет легче пользоваться традиционными базами данных, которые спроектированы для таких программ. Поскольку используется минимальный язык программирования, от программистов требуется меньшая квалификация — или, по крайней мере, меньше разных навыков. Для многих приложений — скажем, для традиционных последовательных программ обновления баз данных — такой образ мыслей вполне оправдан, и для них вполне годятся традиционные приемы, развивавшиеся десятилетиями.

Однако предположим, что прикладная программа существенно отличается от традиционной последовательной обработки записей (или символов), или сложность программы выше, чем обычно, — скажем, в интерактивной CASE-системе.

Отсутствие языковой поддержки абстракции данных, вызванное решением не применять классы, принесет вред. Внутренняя сложность проявится где-нибудь в приложении, а если программа реализована на достаточно бедном языке, то код не будет напрямую отражать проект. Код будет слишком объемным, будет отсутствовать проверка типов, и вообще программа не будет способна к взаимодействию со средствами программирования. А это верный рецепт превращения сопровождения в кошмар.

Обычное временное решение этой проблемы заключается в построении специальных средств, которые поддерживали бы основные понятия проектирования. Эти средства предоставляют конструкции высокого уровня и проверку, чтобы компенсировать недостатки (преднамеренно обедненного) языка реализации. Таким образом, метод проектирования становится специализированным и, как правило, внутрифирменным языком программирования. Такие языки в большинстве случаев плохо заменяют широко распространенные универсальные языки, поддерживаемые разнообразными средствами проектирования.

Самая распространенная причина неиспользования классов в проекте — простая инерция. Традиционные языки программирования не поддерживают понятия класса, а традиционные методы проектирования вынуждены отражать это отсутствие. Проектирование, как правило, сводилось к тому, чтобы разбить проблему на множество процедур, выполняющих нужные действия. Такой подход, названный в главе 2 процедурным программированием, в контексте проектирования часто называют *функциональной декомпозицией*. Обычно задают вопрос: «А можно использовать C++ вместе с методом проектирования, основанном на функциональной декомпозиции?» Можно, но скорее всего, вы придете к тому, что будете пользоваться C++ как просто улучшенным C, и перед вами встанут вышеупомянутые проблемы. Это может быть приемлемо в переходный период для уже завершенных проектов и для подсистем, где классы вроде бы не принесут больших выгод (учитывая опыт конкретных исполнителей). Однако в более долгосрочном плане политика сдерживания широкомасштабного применения классов, вызванная привязанностью к функциональной декомпозиции, не совместима с эффективным использованием C++ и других языков, поддерживающих абстракцию.

Процедурно-ориентированный и объектно-ориентированный взгляды на программирование фундаментально различаются и, как правило, ведут к радикально различным решениям одной и той же проблемы. Это наблюдение одинаково верно как для фазы проектирования, так и для фазы реализации: вы можете сфокусировать проект

на выполняемых действиях или на представляемых сущностях, но не на том и другом одновременно.

Так почему же следует предпочесть «объектно-ориентированное» проектирование традиционным методам, основанным на функциональной декомпозиции? В первую очередь потому, что функциональная декомпозиция ведет к недостаточной абстракции данных. Из этого следует, что получившийся в результате проект будет:

- менее податлив к изменениям;
- менее совместим с инструментальными средствами;
- менее приспособлен к параллельной разработке;
- менее приспособлен для совместного выполнения.

Проблема тут в том, что функциональная декомпозиция заставляет нас делать интересные данные глобальными, поскольку, когда система структурирована в виде дерева функций, любые данные, с которыми работают хотя бы две функции, должны быть глобальными по отношению к обеим этим функциям. Это приводит к тому, что «интересные» данные всплывают все выше и выше к корню дерева по мере того, как все новым и новым функциям требуется доступ к ним (как обычно, считается, что дерево растет от корня вниз). Точно такой же процесс можно наблюдать в однокорневой иерархии классов, где «интересные» данные и функции имеют тенденцию всплывать к корневому классу (§ 24.4). Фокусирование на описании классов и инкапсуляции данных призвано решить эту проблему, делая зависимости между разными частями программы явными и понятными. И что более важно — снижается число взаимозависимостей в системе за счет того, что улучшается локализация обращений к данным.

Однако некоторые проблемы лучше решать написанием набора процедур. Суть объектно-ориентированного подхода к проектированию не в том, чтобы в программе не было функций-не-членов, или чтобы ни одна из частей программы не была процедурно-ориентированной. Нет, суть в том, чтобы разделить разные части программы так, чтобы они лучше отражали понятия прикладной области. Как правило, это достигается, когда проект фокусируется на классах, а не на функциях. Применение процедурного стиля должно быть сознательным решением, а не просто использоваться по умолчанию. И классы, и процедуры нужно применять в зависимости от приложения, а не просто как следствия жесткого метода проектирования.

24.2.2. Отказ от наследования

Рассмотрим проект, не применяющий наследования. Тогда программы просто не смогут воспользоваться основным преимуществом C++, в то же время некие выгоды от C++ по сравнению с C, Pascal, Fortran, COBOL и т. д. будут получены. Распространенные причины отказа от наследования — не считая инерции — состоят в заявлениях, что «наследование — это деталь реализации», «наследование нарушает сокрытие информации» и «наследование затрудняет взаимодействие с другими программами».

Взгляд на наследование просто как на деталь реализации не учитывает, что иерархия классов может непосредственно моделировать ключевые взаимоотношения между понятиями в данной прикладной области. Такие взаимоотношения должны быть выражены в проекте явно.

Можно оправдать исключение наследования из части программ на C++, которые должны взаимодействовать с программами, написанными на других языках. Однако

это *не* достаточная причина для отказа от наследования во всей системе; это просто причина для тщательной спецификации и инкапсуляции интерфейса программы с «внешним миром». Аналогично, беспокойства, связанные с нарушением принципа сокрытия информации при наследовании (§ 24.3.2.1), должны просто заставить нас аккуратнее пользоваться виртуальными функциями и защищенными членами (§ 15.3). Это не причина, чтобы отказываться от наследования вообще.

Во многих случаях никакого реального выигрыша от наследования нет. Однако в больших проектах политика «нет наследованию» приводит к менее понятным и менее гибким системам, в которых наследование симулируется при помощи других языковых и проектных конструкций. Более того, я подозреваю, что, несмотря на такую политику, наследование в конце концов все равно будет использоваться, поскольку программисты найдут убедительные аргументы в пользу применения основанных на наследовании проектных решений в различных частях системы. Поэтому политика «нет наследованию» приведет лишь к тому, что будет утрачена связная общая архитектура, и использование иерархий классов будет ограничено отдельными подсистемами.

Иными словами, держите ум открытым. Иерархии классов не являются необходимой частью всякой хорошей программы, но они могут помочь как в понимании приложения, так и в выражении решения задачи. Тот факт, что наследование может быть неправильно применено — это лишь повод быть внимательным, а не причина для отказа.

24.2.3. Отказ от статической проверки типов

Рассмотрим проект, который отказывается от статической проверки типов на этапе проектирования. Обычно это обосновывается следующим образом: «типы — это артефакт[1] языка программирования», «естественнее думать об объектах, не беспокоясь о типах», и «статическая проверка типов вынуждает нас слишком рано задумываться о вопросах реализации». Такая позиция хороша, пока все идет хорошо. На такие мелочи, как проверка типа, можно не обращать внимания на стадии проектирования; на стадии анализа и ранних стадиях проектирования часто безопасно начисто игнорировать вопросы типов. Однако классы и иерархии классов при проектировании очень полезны. В частности, они позволяют определиться в концепциях, уточнить их взаимосвязь и понять смысл. При продвижении проекта эта точность обретает форму все более точных высказываний относительно классов и их интерфейсов.

Важно понять, что точно определенные и строго типизированные интерфейсы являются фундаментальным инструментом проектирования. C++ проектировался с таким учетом. Строго типизированный интерфейс гарантирует (до известного предела), что компилируются и компонуются только совместимые части программ, таким образом позволяя этим частям делать довольно сильные предположения относительно друг друга. Выполнение этих предположений гарантируется системой типов. Эффект от этого проявляется в минимизации тестирования программы, что повышает эффективность и значительно упрощает фазу интеграции в проекте, над которым работало множество людей. По сути дела, интеграция не стала главной темой данной главы лишь потому, что в интеграции систем со строго типизированными интерфейсами накоплен большой положительный опыт.

[1] То есть нечто искуственное, привнесенное, обусловленное не сутью вещей, а нашим подходом к ним.— *Примеч. ред.*

Рассмотрим аналогию. В физическом мире мы все время что-то стыкуем, и кажется, что число стандартов на разъемы бесконечно. Очевидно, разъемы специально спроектированы так, чтобы было невозможно соединить две части, если они не предназначены для этого, и чтобы они соединялись только правильным образом. Вам не воткнуть низковольтную электробритву в высоковольтную сеть. Если бы вам это удалось, то результатом было бы либо обуглившаяся бритва, либо обуглившийся труп того, кто хотел побриться. Немало изобретательности потрачено на то, чтобы разные части аппаратуры не вставлялись одна в другую. Альтернатива использованию несовместимых разъемов — детали, которые сами защищают себя от нежелательного поведения других деталей, воткнутых в их разъемы. Хорошим примером этого является предохранитель от выбросов напряжения. Поскольку совершенную совместимость невозможно гарантировать на «уровне совместимости разъемов», порой нам требуется более сложная схема защиты, которая бы динамически подстраивалась для защиты от нескольких источников угрозы.

Аналогия почти полная. Статическая проверка типа равносильна совместимости разъемов, а динамическая проверка относится к защите/подстройке цепей. Если обе проверки не сработали — как в физическом мире, так и в программном мире — может возникнуть серьезная авария. В больших системах используются обе формы проверок. На ранних стадиях проектирования можно просто сказать: «Эти две детали должны стыковаться». Однако как именно они должны стыковаться, становится относительно ясно лишь позже. Какие гарантии дает разъем относительно своего поведения? Какие условия приведут к возникновения ошибки? Каковы приблизительные оценки стоимости?

Использование «статической типизации» не ограничивается физическим миром. Использование единиц измерения (например, метров, килограммов и секунд), чтобы избежать смешения разнородных сущностей, широко распространено как в физике, так и в технике.

Согласно схеме проектирования из § 23.4.3 вопросы о типе возникают на этапе 2 (предположительно после поверхностного рассмотрения на этапе 1) и становятся главными на этапе 4.

Статически проверяемые интерфейсы — это основной способ гарантировать взаимодействие между программами на C++, разработанными разными группами. Документирование этих интерфейсов (включая точную информацию о типах) является основным способом общения между различными группами программистов. Такие интерфейсы являются одним из самых важных результатов процесса проектирования, и на них фокусируется общение между проектировщиками и программистами.

Недостаточное внимание к вопросам типов при рассмотрении интерфейсов приводит к проектам с неясной структурой, а выявление ошибок откладывается до этапа выполнения программы. Например, интерфейс может определяться в терминах самоидентифицирующихся объектов:

```
// Пример, предполагающий динамическую проверку типа, а не статическую
Stack s;    // стек может содержать указатели на объекты любого типа
void f()
{
```

```
        s.push (new Saab900);
        s.push (new Saab37B);

        s.pop ()->takeoff ();      // правильно; (takeoff означает взлет)
                                   // Saab 37B — это самолет
        s.pop ()->takeoff ();      // ошибка во время выполнения:
                                   // автомобиль Saab900 не умеет летать
}
```

Здесь недостаточно строгое определение интерфейса (функции ***Stack*::*push*** ()) приводит к динамической проверке вместо статической. Стек ***s*** предназначался для хранения самолетов (plane), но это осталось неявным в коде, так что гарантировать выполнение этого требования становится обязанностью пользователя.

Более точное определение — шаблон плюс виртуальные функции вместо ничем не ограниченной динамической проверки типов — переместит выявление ошибки на стадию компиляции:

```
Stack<Plane*> s;              // стек может содержать только указатели на самолеты
void f ()
{
        s.push (new Saab900); // ошибка: Saab900 — не самолет
        s.push (new Saab37B);

        s.pop ()->takeoff ();      // правильно: Saab 37B — это самолет
        s.pop ()->takeoff ();
}
```

Схожий пример приведен также в § 16.2.2. Разница в затратах времени (выполнения) на динамическую и статическую проверку может быть значительной. Расходы на динамическую проверку по сравнению со статической обычно больше в 3–10 раз.

Впрочем, не надо впадать и в другую крайность. Статической проверкой невозможно выловить все ошибки. Например, даже программа с самой полной статической проверкой уязвима к сбоям аппаратной части. Посмотрите также пример из § 25.4.1, где полная статическая проверка может оказаться недостижима. Однако нужно стремиться к тому, чтобы статически на уровне прикладной программы проверялось подавляющее большинство интерфейсов; § 24.4.2.

Другая проблема заключается в том, что проект может быть безупречен в абстракции, но вызывает серьезные проблемы из-за того, что не принимает в расчет ограничений, налагаемых основными средствами программирования, в данном случае — C++. Например, функция ***f*** (), которой нужно выполнить операцию ***turn_right*** () (повернуть вправо) над некоторым аргументом, может сделать это только при условии, что все ее аргументы принадлежат одному типу:

```
class Plane {
        // ...
        void turn_right ();
};

class Car {
        // ...
        void turn_right ();
};
```

```
void f(X* p)// каким типом должен быть X?
{
    p->turn_right();
    //...
}
```

Некоторые языки (такие как Smalltalk и CLOS) позволяют использовать два типа (если они имеют те же операции) взаимозаменяемым образом, обращаясь к каждому типу через общий подобъект базового класса и откладывая разрешение имени до выполнения программы. Однако C++ (преднамеренно) поддерживает это понятие только через шаблоны и разрешение имен только во время компиляции. Функция (не шаблон) может принимать аргументы двух типов, только если они могут быть неявно преобразованы в общий тип. Таким образом, в предыдущем примере ***X*** должен быть общим базовым классом для ***Plane*** (самолет) и ***Car*** (автомобиль) — например, классом ***Vehicle*** (транспортное средство).

Как правило, примеры, вдохновленные чуждыми C++ понятиями, *можно* отобразить в C++, явно выразив соответствующие допущения. Например, имея ***Plane*** и ***Car*** (без общего базового класса), мы можем по-прежнему создать иерархию классов, которая позволит нам передавать в *f*(***X****) объекты, содержащие либо автомобили, либо самолеты (§ 25.4.1). Однако для этого часто требуется много технических приемов и сообразительности. Для отображения таких концепций в понятия C++ часто полезны шаблоны. Несоответствие между понятиями проекта и C++, как правило, ведет к «неестественному виду» кода и его неэффективности. Программисты, занимающиеся сопровождением, обычно недолюбливают неидиоматический код, возникающий из-за таких несоответствий.

Несоответствие между техникой проектирования и языком реализации можно сравнить с дословным переводом с одного человеческого языка на другой. Например, английский язык с немецкой грамматикой так же неуклюж, как немецкий с английской, и оба оказываются почти непонятны для того, кто хорошо владеет только одним из этих языков.

Классы в программе — это конкретное представление проектных понятий. Следовательно, затушевывая отношения между классами, мы делаем менее понятными фундаментальные концепции проекта.

24.2.4. Отказ от программирования

Программирование дорого и непредсказуемо по сравнению со многими другими видами деятельности, и полученная программа часто не на 100% надежна. Программирование трудоемко, и — по многим причинам — многие серьезные проекты задерживаются из-за неготовности кода. Так почему бы программирование как род деятельности совсем не устранить из процесса?

Многим руководителям кажется, что избавление от нахальных, недисциплинированных, слишком высоко оплачиваемых, одержимых специальными терминами, несоответствующим образом одевающихся и т. п. программистов,[1] может принести значительную дополнительную прибыль. Для программистов это может звучать абсурдно. Однако в некоторых важных областях в самом деле существуют альтернативы традиционному программированию. В некоторых прикладных областях можно сгенериро-

[1] Да, я сам программист.

вать код прямо из спецификаций высокого уровня. В других областях код можно получить, манипулируя фигурами на экране. Например, удобный пользовательский интерфейс можно построить непосредственным манипулированием, и это займет малую толику того времени, которое было бы затрачено на построение этого интерфейса при написании традиционного кода. Аналогично, размещение баз данных и код для доступа к данным в соответствии с этим размещением можно сгенерировать из спецификаций, что гораздо проще, чем выражение этих операций непосредственно на C++ или любом другом языке программирования общего назначения. Конечные автоматы, которые будут проще, быстрее и правильнее, чем того способен добиться программист, можно сгенерировать из спецификаций или непосредственным манипулированием.

Подобные приемы хорошо работают в специфических областях, где есть либо крепкий теоретический фундамент (например, математика, конечные автоматы и реляционные базы данных), либо общий каркас, в который можно встраивать маленькие прикладные фрагменты (например, графические пользовательские интерфейсы, моделирование сетей и схемы баз данных). Очевидное удобство такого приема в ограниченных — и, как правило, принципиально ограниченных — областях может подтолкнуть кое-кого к мысли, что замена традиционного программирования такими приемами уже не за горами. Но это не так. Причина здесь в том, что расширение техники спецификаций за области с крепким теоретическим фундаментом подразумевает, что в языке спецификаций проявится сложность языка программирования общего назначения. А это лишает язык спецификаций его основных преимуществ — ясности и хорошей обоснованности с прикладной точки зрения.

Иногда забывают, что каркас, позволяющий ликвидировать традиционное программирование в какой-либо области — это система или библиотека, которая была спроектирована, запрограммирована и протестирована традиционным образом. Фактически, C++ и описанные в данной книге приемы широко применяются именно при проектировании и построении подобных систем.

Компромисс, обеспечивающий лишь малую часть той выразительности, какую имеет язык программирования общего назначения, при применении вне строго ограниченной специальной области, является наихудшим вариантом. Проектировщиков, привязавшихся к точке зрения моделирования высокого уровня, раздражает все возрастающая сложность, и они производят такие спецификации, из которых получается ужасный код. Программисты, применяющие традиционные методы программирования, вынуждены бороться с трудностями, возникающими из-за отсутствия языковой поддержки, и улучшают код только путем чрезвычайных усилий и отказом от моделей высокого уровня.

Лично я не вижу признаков, что программирование как род деятельности может быть успешно ликвидировано вне областей, где имеется основательный теоретический базис, или где основы программирования обеспечиваются средой разработки. В обоих случаях применяемые приемы теряют эффективность, когда выходят за первоначальные рамки и применяются к более универсальным задачам. Притворяться, что это не так — соблазнительно и опасно. И наоборот, может оказаться глупостью не принимать во внимание технику спецификаций высокого уровня и приемы прямой манипуляции в тех областях, где это хорошо обосновано и опробовано.

Средства проектирования, библиотеки и среды разработки являются одной из высших форм проектирования и программирования. Конструирование полезной математической модели прикладной области является одной из высших форм анализа.

Таким образом, выработка инструментов, языка, среды разработки и т. п. — всего того, что делает результат подобной работы доступным тысячам людей — это способ для программиста и проектировщика не стать ремесленниками, которые владеют лишь одним инструментом.

Очень важно, чтобы система спецификаций или библиотека основных классов могла эффективно взаимодействовать с языком программирования общего назначения. Иначе этот подход станет очень ограничительным. Это подразумевает, что системы спецификаций и прямого манипулирования, генерирующие код на достаточно высоком уровне в принятый язык программирования общего назначения, имеют огромное преимущество. Собственный язык является долговременным преимуществом только для его поставщика. Если произведенный код получается такого низкого уровня, что добавляемый общий код приходится писать без выгод абстрагирования, это приводит к потере надежности и экономии, а также к ухудшению сопровождения. По сути дела, генерирующая система должна быть спроектирована так, чтобы сочетать достоинства спецификаций высокого уровня и языка программирования высокого уровня. Исключить то или другое — значит пожертвовать интересами разработчиков системы ради интересов разработчика средства. Удачные большие системы являются многоуровневыми и модульными, кроме того они развиваются со временем. И потому успех попыток произвести такую систему подразумевает привлечение разнообразных языков, библиотек, инструментов и методов программирования.

24.2.5. Использование исключительно иерархий классов

Когда мы обнаруживаем, что что-то новое действительно работает, мы часто «входим в раж» и применяем это новое без разбора. Другими словами, удачное решение некоторой проблемы часто кажется удачным решением для почти всех проблем. Иерархии классов и операции, полиморфные по отношению к их (одному) объекту, предоставляют удачное решение многих проблем. Однако не любое понятие лучше всего выражается в виде части иерархии, и не любая компонента программы лучше всего представляется в виде иерархии классов.

Почему? Иерархия классов выражает взаимоотношения между своими классами, а класс представляет понятие. Теперь скажите, что общего между улыбкой, драйвером моего привода CD-ROM, записью «Дон Жуана» Рихарда Штрауса, строчкой текста, спутником, моей медицинской картой и часами реального времени? Помещение всех их в единую иерархию, когда их единственной общностью является лишь то, что все они — артефакты программирования (они все — «объекты»), имеет мало фундаментального смысла и может привести к путанице (§ 15.4.5). Загоняя все в единую иерархию, вы можете ввести искусственное сходство и затушевать реальное. Иерархию следует применять, только если анализ вскрыл концептуальную общность, или если проектирование и программирование обнаружили полезную общность в структурах, используемых для реализации понятий. В последнем случае нужно очень осторожно различать истинную общность (которую следует отразить в виде подтипизации с использованием открытого наследования) и полезные упрощения реализации (которые следует отразить в виде закрытого наследования; § 24.3.2.1).

Такой образ мыслей приводит к программе, которая имеет несколько несвязанных или слабо связанных иерархий классов, каждая из которых представляет множество

тесно связанных между собой понятий. Это также приводит к понятию конкретного класса (§ 25.2), который не входит в иерархию, поскольку помещение его в иерархию скомпрометирует его производительность и независимость от остальной системы.

Чтобы быть эффективными, большинство критических операций над классом, входящим в иерархию, должны быть виртуальными функциями. Кроме того, многие данные класса нужно сделать защищенными, а не скрытыми. Это делает класс уязвимым к изменениям в производных классах и может серьезно усложнить тестирование. Там, где с точки зрения проектирования имеет смысл более строгая инкапсуляция, нужно использовать невиртуальные функции и закрытые данные (§ 24.3.2.1).

Если один элемент операции (тот, что обозначает «объект») особый, это может привести к искажениям в проекте. Если с несколькими аргументами мы обращаемся одинаково, операцию лучше представить в виде функции-не-члена. Это не означает, что такие функции должны быть глобальными. Фактически, почти все такие «самостоятельные» функции должны быть членами пространства имен (§ 24.4).

24.3. Классы

Самое фундаментальное понятие объектно-ориентированного проектирования и программирования заключается в том, что программа является моделью некоторых аспектов реальности. Классы в программе представляют собой фундаментальные понятия прикладной области и, в частности, фундаментальные понятия моделируемой «реальности». Объекты реального мира и артефакты реализации представляются объектами классов.

Анализ отношений между классами и внутри частей класса — это центральное место в проектировании системы. Следует учитывать:

§ 24.3.2 Отношения наследования
§ 24.3.3 Отношения включения
§ 24.3.5 Отношения использования
§ 24.3.6 Запрограммированные отношения
§ 24.3.7 Отношения внутри класса

Поскольку класс в C++ — это тип, классы и взаимоотношения поддерживаются компилятором и в основном доступны статическому анализу.

Чтобы вписаться в проект, класс не просто должен представлять собой полезное понятие; он также должен обеспечить подходящий интерфейс. Идеальный класс имеет минимальную и строго определенную зависимость от остального мира и интерфейс, предоставляющий минимум информации о классе остальному миру (§ 24.4.2).

24.3.1. Что представляют классы?

Есть два основных вида классов:

[1] классы, прямо отражающие прикладные понятия; то есть концепции, непосредственно используемые конечными пользователями для описания своих проблем и их решений;

[2] классы, являющиеся артефактами реализации; то есть они выражают концепции, использующиеся проектировщиками и программистами для описания своих приемов реализации.

Некоторые из классов, являющихся артефактами реализации, также представляют собой понятия реального мира. Например, аппаратные и программные ресурсы системы могут стать хорошими кандидатами в прикладные классы. В этом отражается тот факт, что систему можно рассматривать с разных точек зрения; то, что для одного — деталь реализации, для другого — характерная черта прикладной области. Хорошо спроектированная система содержит классы, поддерживающие логически разные взгляды на систему. Например:

[1] классы, представляющие прикладные понятия (например, легковые машины и грузовики);
[2] классы, представляющие обобщения прикладных понятий (например, транспортные средства);
[3] классы, представляющие аппаратные ресурсы (например, класс, управляющий распределением памяти);
[4] классы, представляющие ресурсы системы (например, потоки вывода);
[5] классы, используемые для реализации других классов (например, списки, очереди, блокировки);
[6] встроенные типы данных и управляющие структуры.

В более сложных системах сохранять разделение между логически разными типами классов и не смешивать уровни абстракции становится нетривиальным делом. В простых системах можно выделить три уровня абстракций:

[1+2] уровень прикладного представления системы;
[3+4] уровень, отражающий компьютер, на котором выполняется моделирование;
[5+6] уровень, отражающий низкоуровневую (языка программирования) реализацию.

Как правило, чем больше система, тем больше уровней абстракции требуется для ее описания, и тем труднее становится определять и поддерживать эти уровни. Отметим, что такое разделение на уровни абстракции имеет прямые соответствия в природе и в других видах человеческих конструкций. Например, можно считать, что дом состоит из:

[1] атомов;
[2] молекул;
[3] бревен и кирпичей;
[4] полов, стен и потолков;
[5] комнат.

Пока эти уровни абстракции разделены, вы можете поддерживать согласованный взгляд на дом. Однако, если их смешать, возникают нелепости. Например, высказывание «Мой дом состоит из нескольких тысяч фунтов углерода, нескольких сложных полимеров, 5000 кирпичей, двух ванных комнат и 13 потолков» звучит глупо. Учитывая абстрактную природу программ, равносильное по глупости высказывание о сложной системе не всегда так легко узнается.

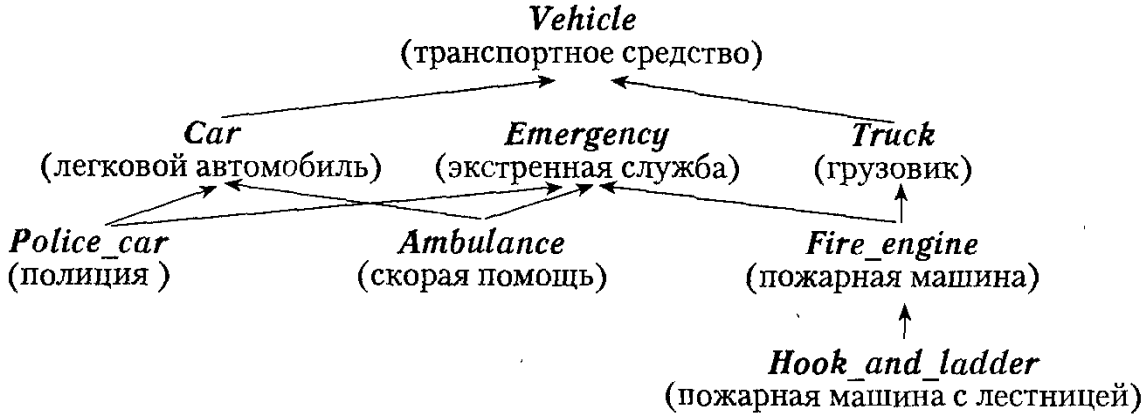
Перевод понятия прикладной области в класс проекта — не простая механическая операция. Она часто требует понимания. Учтите, что понятия прикладной области сами по себе являются абстракциями. Например, «налогоплательщики», «монахи» и «сотрудники» на самом деле в природе не существуют; эти понятия — просто ярлыки, которые навешивают на личностей, чтобы классифицировать их в соответствии с некоторой системой. Реальный или даже воображаемый мир (например, литература, особенно научная фантастика) иногда выступает просто в качестве источника поня-

тий, которые при преобразовании в классы радикально меняются. Например, экран моего персонального компьютера не напоминает поверхность рабочего стола, несмотря на то, что он был разработан, чтобы служить метафорой рабочего стола[1], а окна на экране имеют лишь самое отдаленное отношение к тем затейливым приспособлениям, которые вызывают сквозняки у меня в кабинете. Смысл моделирования реальности не в том, чтобы рабски копировать то, что мы видим, а скорее использовать это в качестве отправной точки для проектирования, в качестве источника вдохновения и якоря, за который можно зацепиться, когда неосязаемая природа программ угрожает превзойти нашу способность понимания собственного продукта.

Предостережение: начинающим часто оказывается трудно «найти классы», но эта трудность вскоре преодолевается без долговременных неприятных последствий. Однако потом часто наступает фаза, когда классы — и их наследственные отношения — начинают неуправляемым образом размножаться. Это может вызвать долговременные проблемы со сложностью, понятностью и эффективностью получающийся программы. Не надо каждую сиюминутную деталь представлять отдельным классом, и не всякую взаимосвязь нужно выражать как наследование. Постарайтесь помнить, что цель проекта — смоделировать систему с *соответствующим* уровнем детализации и на *соответствующем* уровне абстракции. Нахождение баланса между простотой и общностью — не простое дело.

24.3.2. Иерархии классов

Рассмотрим моделирование уличного городского движения, чтобы примерно определить наиболее вероятное время, требуемое машинам экстренных служб (полиция, скорая помощь, пожарные), чтобы добраться до места аварии. Ясно, что нам нужно представить легковые машины, грузовики, машины скорой помощи, пожарные машины, полицейские автомобили, автобусы и т. д. Появляется наследование, поскольку понятия реального мира не существуют изолированно друг от друга; они связаны друг с другом множеством связей. Без осознания этих взаимосвязей мы не воспринимаем понятия. Поэтому модель, не выражающая этих взаимосвязей, не адекватно отражает наши понятия. То есть нам нужно, чтобы классы выражали понятия, но этого не достаточно. Нам также нужны способы представления связей между классами. Одним из мощных средств непосредственного представления иерархических связей является наследование. В нашем примере мы, вероятно, особо выделим автомобили экстренных служб, а также захотим различать легковые и грузовые машины. Это приведет к следующей иерархии классов:



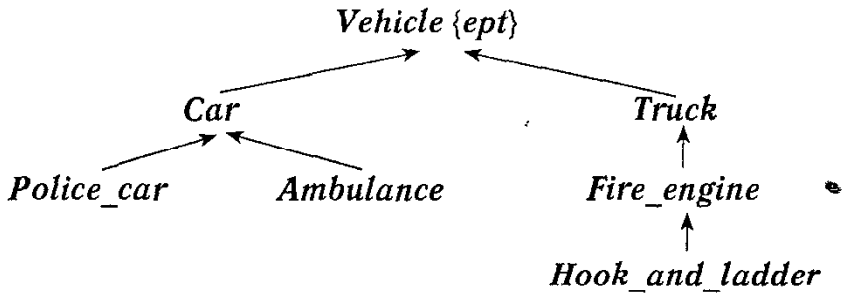
[1] Я все равно не смог бы вынести такого беспорядка на экране.

Здесь ***Emergency*** представляет понятие транспортного средства некоторой экстренной службы, существенное для нашего моделирования: оно может нарушать кое-какие правила дорожного движения, в случае движения по вызову имеет преимущества на перекрестках, его контролирует диспетчер и т. д.

Вот версия на C++:

```
class Vehicle { /* ... */ };
class Emergency { /* ... */ };
class Car : public Vehicle { /* ... */ };
class Truck : public Vehicle { /* ... */ };
class Police_car : public Car, protected Emergency { /* ... */ };
class Ambulance : public Car, protected Emergency { /* ... */ };
class Fire_engine : public Truck, protected Emergency { /* ... */ };
class Hook_and_ladder : public Fire_engine { /* ... */ };
```

Наследование — это связь самого высшего уровня, которая выражается на C++; на ранних стадиях проектирования оно играет важнейшую роль. Часто для выражения связи существует выбор между наследованием и членством. Рассмотрим альтернативное понятие того, что значит быть транспортным средством экстренной службы: транспортное средство является экстренным, если у него есть проблесковый фонарь («мигалка»). Это позволило бы упростить иерархию классов, заменив класс ***Emergency*** членом в классе *Vehicle*:



Здесь *eptr* означает «указатель на экстренность». Теперь класс ***Emergency*** просто используется как член в классах, которым может понадобиться действовать как транспортные средства экстренной службы:

```
class Emergency { /* ... */ };
class Vehicle { protected: Emergency* eptr; /* ... */ };
class Car :public Vehicle { /* ... */ };
class Truck :public Vehicle { /* ... */ };
class Police_car :public Car { /* ... */ };
class Ambulance :public Car { /* ... */ };
class Fire_engine :public Truck { /* ... */ };
class Hook_and_ladder :public Fire_engine { /* ... */ };
```

Здесь, транспортное средство является транспортным средством экстренной службы, если ***Vehicle::eptr*** не равен нулю. «Простые» легковые автомобили и грузовики инициализируются с ***Vehicle::eptr*** равным нулю, в остальных случаях ***Vehicle::eptr*** не равен нулю. Например:

```
Car::Car()                    // конструктор легкового автомобиля
{
    eptr = 0;
}
```

```
Police_car::Police_car ()          // конструктор полицейской машины
{
    eptr = new Emergency;
}
```

Определение вещей таким образом делает возможным простое преобразование из транспортного средства экстренной службы в обычное и наоборот:

```
void f(Vehicle* p)
{
    delete p->eptr;
    p->eptr=0;                     // теперь это не транспортное средство
                                   // экстренной службы
    //...
    p->eptr = new Emergency;       // снова появилась «экстренность»
}
```

Итак, какой вариант иерархии классов лучше? Общий ответ таков: «Лучше та программа, которая лучше моделирует интересующие нас аспекты реального мира». То есть, выбирая между моделями при неизбежных ограничениях эффективности и простоты мы должны стремиться к большему реализму. В данном случае легкость преобразования обычных транспортных средств в транспортные средства экстренных служб мне кажется неестественной. Пожарные машины и машины скорой помощи имеют особую конструкцию, укомплектованы обученным персоналом и работают под управлением диспетчера, что требует специального оборудования для связи. Это соображение показывает, что понятие «транспортное средство экстренной службы» должно быть фундаментальным и представляться в программе непосредственно, чтобы улучшить проверку типов и сделать осмысленным использование инструментальных средств. Если бы мы моделировали некую задачу, где роли автомобилей менее четко определены — скажем, области, где для доставки спецперсонала к месту происшествия используются частные автомашины, и где связь обеспечивается портативными радиопередатчиками — более подходящим может оказаться другой способ моделирования.

Для тех, кому моделирование уличного движения кажется надуманным примером, может быть, стоит напомнить, что задача выбора между наследованием и членством возникает при проектировании постоянно. См. также эквивалентный пример с полосой прокрутки из § 24.3.3.

24.3.2.1. Зависимости внутри иерархии классов

Естественно, производные классы зависят от базовых. Не так часто замечают, что может быть справедливо и обратное[1]. Если в классе есть виртуальная функция, он зависит от производных классов, выполняющих часть его обязанностей, при условии, что эти производные классы замещают его виртуальные функции. Если какой-то член базового класса вызывает виртуальную функцию этого класса, то базовый класс зависит от своих производных в самой своей реализации. Аналогично, если какой-то

[1] Это наблюдение было сформулировано так: «Безумие наследственно. Вы заражаетесь им от своих детей».

класс пользуется защищенными членами, то он опять же зависит от своих производных классов в самой своей реализации. Рассмотрим пример:

```
class B {
        // ...
protected:
        int a;
public:
        virtual int f();
        int g() { int x = f(); return x-a; }
};
```

Что делает *g*()? Ответ критическим образом зависит от определения *f*() в некотором производном классе. Вот версия, которая гарантирует, что *g*() вернет *1*:

```
class D1 : public B {
        int f() { return a+1; }
};
```

а вот версия, которая заставит *g*() вывести *"Здравствуй, мир!"* и возвратить *0*:

```
class D2 : public B {
        int f() { cout << "Здравствуй, мир!\n"; return a; }
};
```

Этот пример просто иллюстрирует важнейшее свойство виртуальных функций. Почему это должно выглядеть глупо? Почему бы какому-нибудь программисту когда-нибудь не написать что-либо подобное? Ответ заключается в том, что виртуальная функция входит в интерфейс базового класса, и этот класс предположительно может использоваться без каких-либо знаний о его производных классах. Следовательно, должна быть предоставлена возможность описать ожидаемое поведение объекта базового класса таким образом, чтобы можно было составить программу, ничего не зная о производном классе. Каждый класс, заменяющий виртуальную функцию, должен реализовать вариант этого поведения. Например, виртуальная функция ***rotate*** () (повернуть) из класса ***Shape*** (фигура) поворачивает фигуру. Функции ***rotate*** () в таких производных классах как ***Circle*** (круг) и ***Triangle*** (треугольник) должны поворачивать соответствующую фигуру; в противном случае, фундаментальное предположение о классе ***Shape*** нарушается. Такого предположения не делается относительно класса ***B*** или его производных классов ***D1*** и ***D2***; таким образом, пример бессмыслен. Даже имена ***B***, ***D1***, ***D2***, *f* и *g* выбраны так, что понять их предназначение невозможно. Спецификация ожидаемого поведения виртуальных функций — это *главный* аспект проектировании классов. Выбор хороших имен для классов и функций очень важен — и не всегда легок.

Зависимость от неизвестного (возможно, еще не написанного) производного класса — это хорошо или плохо? Естественно, все зависит от намерений программиста. Если цель в том, чтобы изолировать класс от всех внешних влияний, дабы гарантировать его специфическое поведение, то защищенных членов и виртуальных функций лучше избегать. Если, однако, наши намерения состоят в том, чтобы обеспечить каркас, к которому следующий программист (или тот же самый программист через несколько недель) сможет добавлять код, то виртуальные функции часто являются наиболее элегантным средством для достижения этого; и защищенные функции-члены доказали свою пригодность для использования в таких целях. Этот прием приме-

няется в библиотеке потокового ввода/вывода (§ 21.6) и проиллюстрирован окончательной версией иерархии ***Ival_box*** (§ 12.4.2).

Если виртуальная функция предназначается только для косвенного использования производным классом, ее можно оставить закрытой. Например, рассмотрим простой шаблон буфера:

```
template<class T> class Buffer {
public:
        void put (T);       // вызывает overflow(T) (переполнение сверху), если буфер полон
        T get ();           // вызывает underflow() (переполнение снизу), если буфер пуст
        //...
private:
        virtual int overflow (T);
        virtual int underflow ();
        //...
};
```

Функции ***put*** () и ***get*** () вызывают соответственно виртуальные функции ***overflow*** () и ***underflow*** (). Теперь пользователь, чтобы удовлетворить потребность в разнообразных буферах, может реализовать несколько буферных типов, заместив ***overflow*** () и ***underflow*** ():

```
// циклический буфер
template<class T> class Circular_buffer : public Buffer<T> {
        int overflow (T);       // если переполнение, начнем заполнять сначала
        int underflow (T);
        //...
};

// расширяющийся буфер
template<class T> class Expanding_buffer : public Buffer<T> {
        int overflow (T);       // если переполнение, увеличим размер буфера
        int underflow ();
        //...
};
```

Только если бы производному классу было нужно обращаться к ***overflow*** () и ***underflow*** () напрямую, эти функции нужно было бы сделать защищенными, а не закрытыми.

24.3.3. Отношения включения

При применении включения (содержания в себе) для представления объекта класса ***X*** есть две основные альтернативы:

[1] Объявить член типа ***X***.

[2] Объявить член типа ***X**** или ***X***&.

Если значение указателя никогда не изменяется, эти варианты равносильны (не учитывая эффективности и способа написания конструкторов и деструкторов):

```
class X {
public:
        X (int);
        //...
};
```

```
class C {
        X a;
        X* p;
        X& r;
public:
        C (int i, int j, int k) : a (i), p (new X (j)), r (*new X (k)) { }
        ~C () { delete p; delete &r; }
};
```

В таких случаях предпочтительнее членство собственно объекта, как в случае ***C::a***, поскольку оно эффективнее по быстродействию, памяти и нажатиям клавиш. Оно также меньше подвержено ошибкам, поскольку связь между содержащимся и содержащим объектами описывается правилами конструирования и уничтожения (§ 10.4.1, § 12.2.2, § 14.4.1). Однако см. также § 24.4.2 и § 25.7.

Решение с указателем можно применять тогда, когда за время жизни «содержащего» объекта нужно изменить указатель на «содержащийся» объект. Например:

```
class C2 {
        X* p;
public:
        C2 (int i) : p (new X (i)) { }

        ~C2 () { delete p; }
        X* change (X* q)
        {
                X* t = p;
                p = q;
                return t;
        }
};
```

Другое соображение в пользу применения члена-указателя может состоять в том, чтобы позволить задавать «содержащийся» член в качестве аргумента:

```
class C3 {
        X* p;
public:
        C3 (X* q) : p (q) { }
        // ...
};
```

Имея объекты, включающие указатели на другие объекты, мы создаем то, что часто называют *иерархиями объектов*. Они являются альтернативой и дополнением к иерархии классов. Как показано в примере с транспортными средствами экстренных служб (§ 24.3.2), при проектировании часто нелегко выбрать между представлением свойства класса в виде базового класса, или в виде члена. Необходимость в замещении является признаком того, что лучше выбрать первое. И наоборот, если вам нужно иметь возможность представлять свойство множеством типов — это признак того, что лучше выбрать последнее. Например:

```
class XX : public X { /* ... */ };

class XXX : public X { /* ... */ };

void f()
```

```
{
    C3* p1 = new C3 (new X);        // C3 "содержит в себе" X
    C3* p2 = new C3 (new XX);       // C3 "содержит в себе" XX
    C3* p3 = new C3 (new XXX);      // C3 "содержит в себе" XXX
    // ...
}
```

Такое нельзя смоделировать, сделав ***C3*** производным от ***X*** или включив в ***C3*** член типа ***X***, поскольку нужно использовать точный тип члена. Это важно для классов с виртуальными функциями, таких как класс фигур (§ 2.6.2) или класс абстрактных множеств (§ 25.3).

Ссылки могут использоваться для упрощения классов, основанных на членстве указателей, когда за время существования содержащего объекта ссылаются только на один объект. Например:

```
class C4 {
    X& r;
public:
    C4 (X& q) : r (q) { }
    // ...
};
```

Члены-указатели и члены-ссылки также нужны, когда объект является совместно используемым:

```
X* p = new XX;
C4 obj1 (*p);
C4 obj2 (*p);       // obj1 и obj2 теперь совместно используют новый XX
```

Естественно, обращение с совместно используемыми объектами требует чрезвычайной осторожности — особенно в параллельных системах.

24.3.4. Включение и наследование

С учетом важности отношений наследования не покажется удивительным, что ими часто злоупотребляют, и их нередко неправильно понимают. Когда производный класс ***D*** порожден открытым образом от базового класса ***B***, часто говорят, что ***D*** *есть* (*is a*) ***B***:

```
class B { /* ... */ };
class D : public B { /* ... */ };  // D — это разновидность B
```

Иначе это выражается словами, что наследование является отношением *is-a*, или — несколько более точно — что ***D*** *есть разновидность* ***B***. С другой стороны, о классе ***D***, имеющем член другого класса ***B***, часто говорят, что он *имеет* (*has a*) ***B*** или *содержит* ***B***. Например:

```
class D {           // D содержит B
public:
    B b;
    // ...
};
```

Иначе это выражается словами, что членство является отношением *has-a*.

Для данных классов ***B*** и ***D*** как нам выбрать между наследованием и членством? Рассмотрим классы ***Airplane*** (самолет) и ***Engine*** (двигатель). Новичкам часто приходит в голову сделать ***Airplane*** производным от ***Engine***. Это — плохая идея, поскольку самолет *не является* двигателем, он *имеет* двигатель. Один из способов увидеть это — задуматься, может ли самолет иметь несколько двигателей? Поскольку это представляется возможным (даже если в нашей программе все самолеты одномоторные), нам следует воспользоваться членством, а не наследованием. Во многих случаях, когда есть сомнения, полезен вопрос «может ли у него их быть несколько?» Как всегда, именно неосязаемая природа программ приводит к тому, что приходится анализировать такие «очевидные» вопросы. Если бы все классы было так легко наглядно представить, как самолет и двигатель, было бы легко избежать тривиальных ошибок, когда ***Airplane*** делают производным от ***Engine***. Однако такие ошибки случаются весьма часто — в частности, их допускают люди, считающие наследование просто механизмом комбинирования конструкций уровня языка программирования. Несмотря на удобство и краткость записи, которые обеспечивает наследование, его следует использовать почти исключительно для выражения отношений, четко определенных в проекте. Рассмотрим пример:

```
class B {
public:
        virtual void f();
        void g();
};

class D1 {                      // D1 содержит B
public:
        B b;
        void f();               // не замещает b.f()
};

void h1(D1* pd)
{
        B* pb = pd;             // ошибка: нет преобразования D1* в B*
        pb = &pd->b;
        pb->g();                // вызывает B::g
        pd->g();                // ошибка: D1 не имеет члена g()
        pd->b.g();
        pb->f();                // вызывает B::f (не замещенную D1::f)
        pd->f();                // вызывает D1::f
}
```

Отметим, что не существует неявного преобразования из класса в один из его членов, и что класс, содержащий член другого класса, не замещает виртуальных функций того класса. Это очевидно контрастирует со случаем открытого наследования:

```
class D2 : public B {           // D2 является разновидностью B
public:
        void f();               // замещает B::f()
};

void h2(D2* pd)
```

```
{
    B* pb = pd;          // правильно: неявное преобразование D2* в B*
    pb->g();             // вызывает B::g
    pd->g();             // вызывает B::g
    pb->f();             // виртуальный вызов: обращение к D2::f
    pd->f();             // обращается к D2::f
}
```

Удобство записи в примере с ***D2*** по сравнению с примером с ***D1*** является поводом к злоупотреблению. Надо помнить, что за удобство записи приходится платить возросшей зависимостью между ***B2*** и ***D2*** (см. § 24.3.2.1). В частности, легко забыть о неявном преобразовании из ***D2*** в ***B***. Если такие преобразования не являются допустимой частью семантики ваших классов, *открытого* наследования следует избегать. Когда же класс используется для представления понятия, а наследование используется для представления отношения *is-a*, такие преобразования очень часто оказываются именно тем, что нужно.

Существуют случаи, когда вам нравится наследование, но вы не можете себе позволить таких преобразований. Рассмотрим написание класса ***Cfield*** (управляемое поле), который — кроме всего прочего — обеспечивает динамическое (на этапе выполнения) управление доступом для другого класса ***Field*** (поле). На первый взгляд, сделать ***Cfield*** производным от ***Field*** вполне разумно:

```
class Cfield : public Field { /* ... */ };
```

Это выражает тот факт, что ***Cfield*** является разновидностью ***Field***, допускает более удобную запись при написании функций класса ***Cfield***, которые пользуются членами из части ***Cfield***, и — самое важное — позволяет классу ***Cfield*** замещать виртуальные функции класса ***Field***. Неприятность заключается в том, что преобразование ***Cfield**** в ***Field****, подразумеваемое в объявлении ***Cfield***, делает бессмысленными все попытки управлять доступом к ***Field***:

```
void g(Cfield* p)
{
    *p = "asdf";         // доступ к полю, контролируемый оператором
                         // присваивания из Cfield: p->Cfield::operator=("asdf")
    Field* q = p;        // неявное преобразование Cfield* в Field*
    *q = "asdf";         // ОШИБКА! никакого контроля
}
```

Решением могло бы стать определение ***Cfield*** таким образом, чтобы ***Field*** было его членом, но это не даст возможности ***Cfield*** замещать виртуальные функции ***Field***. Лучшее решение — использовать *закрытое* наследование:

```
class Cfield : private Field { /* ... */ };
```

С точки зрения проектирования закрытое наследование равносильно включению, если не считать (иногда очень важного) вопроса с замещением. Важное применение такого подхода — прием с открытым наследованием из абстрактного базового класса, определяющего интерфейс, и одновременным защищенным или закрытым наследованием от конкретного класса для предоставления реализации (§ 2.5.4, § 12.3, § 25.3). Поскольку построение производных классов, подразумеваемое *закрытым* и *защищенным* наследованием, является деталью реализации, которая не влияет на тип производного

класса, его иногда называют *наследованием реализации*, и оно противопоставляется открытому наследованию, посредством которого интерфейс базового класса наследуется и неявное преобразование в базовый тип становится допустимым. Последнее иногда называют *подтипизацией* или *наследованием интерфейса*.

Другой способ выразить то же самое — сказать, что объектом производного класса можно пользоваться там, где допустим объект открытого базового класса. Это иногда называют «принципом подстановки Лисков[1]» (§ 23.6 [Liskov, 1987]). Различение между открытым, защищенным и закрытым наследованием непосредственно поддерживает это для полиморфных типов, к которым мы обращаемся посредством указателей и ссылок.

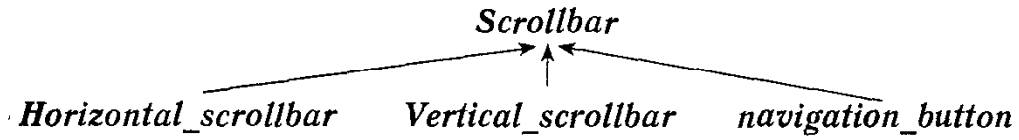
24.3.4.1. Альтернатива «членство/иерархия»

Чтобы еще глубже разобраться с выбором между включением и наследованием, рассмотрим, как представить полосу прокрутки в интерактивной графической системе, и как связать эту полосу с окном. Нам нужны две полосы прокрутки: горизонтальная и вертикальная. Мы можем представить это либо двумя типами — ***Horizontal_scrollbar*** и ***Vertical_scrollbar*** — либо одним типом ***Scrollbar*** с аргументом, указывающим, вертикальная ли это полоса прокрутки или горизонтальная. Первый вариант подразумевает необходимость введения третьего типа, просто полосы прокрутки ***Scrollbar***, выступающего в качестве базового класса для двух частных типов полос. Второй вариант предполагает необходимость введения дополнительного аргумента для указания типа полосы и выбора значений, представляющих эти два типа. Например:

```
enum Orientation { horizontal, vertical };
```

Когда выбор сделан, он определяет, какие изменения нужны для расширения системы. В примере с полосами прокрутки нам может понадобиться ввести третий тип полосы. Сначала мы могли подумать, что нужно лишь два вида таких полос прокрутки («в конце концов, окно имеет только два измерения»). Однако в данном случае — как и в большинстве других— возможны расширения, которые проявляются при перепроектировании. Например, кому-то может захотеться вместо двух полос сделать одну «навигационную кнопку» (navigation button). Такая кнопка будет приводить к прокрутке в разных направлениях в зависимости от того, с какой стороны пользователь ее нажал. Щелчок в середине ее верхней части будет означать «прокрутку вверх», щелчок в середине левой части — «прокрутку влево», щелчок в левом верхнем углу — «прокрутку вверх и влево». Такие кнопки встречаются. Их можно рассматривать как развитие идеи полос прокрутки; они особенно удобны в тех прикладных программах, где прокручиваемая информация — не просто текст, а изображения более общего вида.

Добавление навигационной кнопки в программу с иерархией классов с тремя полосами прокрутки влечет за собой добавление еще одного класса, но не требует изменений в коде старой полосы прокрутки:



Это приятный аспект «иерархического» решения.

[1] Барбары Лисков. — *Примеч. ред.*

Задание ориентации полосы прокрутки в качестве аргумента подразумевает наличие полей типа в объектах, отвечающих полосам прокрутки, и использование инструкции ***switch*** в коде функций-членов класса ***Scrollbar***. То есть мы столкнулись с выбором между выражением этого аспекта структуры системы посредством объявлений или посредством кода. Первый выбор улучшает статическую проверку и увеличивает объем информации, доступной инструментальным средствам. Второй вариант откладывает решение на время выполнения программы и позволяет вносить изменения, модифицируя отдельные функции и не касаясь общей структуры системы, как ее видят система контроля типов и другие инструментальные средства. Лично я рекомендую, в большинстве случаев, для прямого моделирования иерархических отношений между понятиями пользоваться иерархией классов.

Решение с одним типом полосы прокрутки облегчает хранение и передачу информации, задающей вид прокрутки:

```
void helper (Orientation oo)
{
        //...
        p = new Scrollbar (oo);
        //...
}
void me ()
{
        helper (horizontal);
        //...
}
```

Такое представление также упростило бы переориентацию полос во время выполнения программы. Вряд ли это имеет большое значение для полос прокрутки, но есть много схожих примеров, где это важно. Главное в том, что в этом вопросе всегда существует выбор, причем зачастую трудный.

24.3.4.2. Альтернатива «включение/иерархия»

Теперь рассмотрим, как привязать полосу прокрутки к окну. Если мы считаем класс ***Window_with_scrollbar*** (окно с полосой прокрутки) чем-то таким, что является и окном, и полосой прокрутки, то получим следующее:

```
class Window_with_scrollbar : public Window, public Scrollbar {
        //...
}
```

Это позволит любому ***Window_with_scrollbar*** действовать и как ***Scrollbar***, и как ***Window***, но вынуждает нас использовать решение с одним типом полосы прокрутки.

С другой стороны, если мы рассматриваем ***Window_with_scrollbar*** как окно, имеющее полосу прокрутки, то получим следующее:

```
class Window_with_scrollbar : public Window {
        //...
        Scrollbar* sb;
public:
        Window_with_scrollbar (Scrollbar* p, /* ... */ ):
```

```
        Window {/* ... */}, sb (p) {/* ... */}
};
```

Это позволяет нам применить решение с иерархией полос прокрутки. Задание полосы в качестве аргумента позволяет окну не знать точный тип своей полосы. Мы можем даже передавать ***Scrollbar*** так же, как мы передавали ориентацию ***Orientation*** (§ 24.3.4.1). Если нам понадобится, чтобы ***Window_with_scrollbar*** действовало как полоса прокрутки, мы можем добавить оператор преобразования:

```
Window_with_scrollbar::operator Scrollbar& ()
{
    return *sb;
}
```

По-моему, предпочтительнее, когда окно содержит в себе полосу прокрутки. Мне легче вообразить окно, *имеющее* полосу прокрутки, чем окно, которое, будучи окном, *является* еще и полосой прокрутки. По сути дела, моя излюбленная стратегия проектирования состоит в том, чтобы сделать полосу прокрутки специальной разновидностью окна, а затем включить ее в окно, которое нуждается в услугах полосы прокрутки. Подобная стратегия приводит к решению в пользу включения. Другой аргумент в пользу того же решения проистекает из эмпирического правила «может ли оно иметь их две?» (§ 24.3.4). Поскольку нет никаких логических причин, почему бы окно не могло иметь две полосы прокрутки (ведь на самом деле многие окна имеют горизонтальную и вертикальную полосы прокрутки), класс ***Window_with_scrollbar*** не следует делать производным от ***Scrollbar***.

Отметим, что нельзя создать производный класс от неизвестного класса. Во время компиляции должен быть известен точный тип базового класса (§ 12.2). С другой стороны, если атрибут класса передается его конструктору в качестве аргумента, то где-то в классе должен быть представляющий его член. Однако если этот член — указатель или ссылка, мы можем передать объект класса производного по отношению к классу, указываемому этим членом. Например, член ***sb*** типа ***Scrollbar**** в предыдущем примере может указывать на ***Scrollbar*** такого типа как ***Navigation_button***, неизвестного пользователям ***Scrollbar****.

24.3.5. Отношения использования

Знание того, какие другие классы используются данным классом и как именно, часто критически важно для выражения и понимания проекта. Такие зависимости поддерживаются C++ только неявно. Класс может использовать только имена, которые были (где-то) объявлены, но список использованных имен не обеспечивается исходным кодом C++. Для извлечения такой информации необходимы инструментальные средства программирования (или при отсутствии соответствующих средств — внимательное чтение). Способы, которыми класс ***X*** может пользоваться другим классом ***Y***, можно классифицировать по-разному. Например, так:

- ***X*** пользуется именем ***Y***;
- ***X*** пользуется ***Y***;
  - ***X*** вызывает функцию-член ***Y***;
  - ***X*** читает член класса ***Y***;
  - ***X*** записывает член класса ***Y***;

- *X* создает *Y*;
  - *X* выделяет память под автоматические и статические переменные класса *Y*;
  - *X* создает *Y* при помощи *new*;
  - *X* принимает размеры класса *Y*.

Принятие в качестве аргумента размеров объекта классифицируется как создание, поскольку это требует знания полного объявления класса. С другой стороны, пользование именем *Y* классифицируется как отдельная зависимость, поскольку просто употребление имени — например, в объявлении *Y** или упоминание *Y* в объявлении внешней функции — совсем не требует доступа к объявлению *Y* (§ 5.7):

```
class Y;          // Y — имя некоторого класса
Y* p;
extern Y f(const Y&);
```

Часто важно различать зависимости интерфейса класса (объявления класса) и зависимости реализации класса (определения членов класса). В хорошо спроектированной системе, последние, как правило, имеют еще много зависимостей, и они гораздо менее интересны для пользователя, чем зависимости объявлений класса (§ 24.4.2). Как правило, цели проектирования заключаются в минимизации зависимостей интерфейса, так как они становятся зависимостями пользователей класса (§ 8.2.4.1, § 9.3.2, § 12.4.1.1, § 24.4).

От тех, кто реализует класс, C++ не требует детально описывать, какие используются другие классы и как именно. Одна из причин этого заключается в том, что большинство важных классов зависят от стольких других, что для читабельности перечень этих классов пришлось бы сократить — например, директивой *#include*. Другая причина в том, что классификация и мельчайшие подробности таких зависимостей не представляются вопросом, который должен решаться языком программирования. С другой стороны, то, как именно рассматриваются зависимости *использования*, зависит от целей проектировщика, программиста или инструментальных средств. И наконец, какие зависимости представляют интерес, также может зависеть от деталей реализации языка.

24.3.6. Запрограммированные отношения

Язык программирования не может — и не должен — напрямую поддерживать все концепции всех методов проектирования. Подобным образом и язык проектирования не должен поддерживать все особенности всех языков программирования. Язык проектирования должен быть богаче и менее озабочен деталями, чем язык для программирования систем. И наоборот, язык программирования должен уметь поддерживать различные философии проектирования, или он потерпит неудачу из-за неспособности приспосабливаться к ним.

Когда язык программирования не обеспечивает средств для прямого представления понятий проекта, необходимо использовать удобное отображение конструкций проекта в конструкции языка программирования. Например, метод проектирования может пользоваться понятием делегирования. То есть проект может специфицировать, что все операции, не определенные для класса *A*, должны обслуживаться объектом класса *B*, на который указывает указатель *p*. C++ не может выразить это напрямую. Однако можно представить стилизацию этой идеи на C++ и написать программу, генерирующую соответствующий код. Рассмотрим пример:

```
class B {
        // ...
        void f();
        void g();
        void h();
};
class A {
        B* p;
        // ...
        void f();
        void ff();
};
```

Спецификация того, что ***A*** делегирует полномочия ***B*** через ***A*::*p***, даст следующий код:

```
class A {
        B* p;                   // делегирование через p
        // ...
        void f();
        void ff();
        void g() { p->g(); }    // делегируем g()
        void h() { p->h(); }    // делегируем h()
};
```

Программисту ясно, что здесь делается, но имитация проектного понятия в коде далека от отношения взаимнооднозначного соответствия. Такие «запрограммированные» отношения не «понимаются» языком программирования, и поэтому они менее доступны для инструментальных средств. Например, стандартные инструментальные средства не распознают «делегирование» от ***A*** к ***B*** через ***A*::*p***, как нечто отличное от других использований ***B****.

Взаимнооднозначное соответствие между проектными понятиями и понятиями языка программирования нужно использовать при всякой возможности. Взаимнооднозначное соответствие обеспечивает простоту и гарантирует, что проект действительно отображается в программе, так что программисты и инструментальные средства могут воспользоваться этим.

Операторы преобразования предоставляют языковой механизм для выражения класса запрограммированных отношений. То есть оператор преобразования ***X*::*operator Y*()** говорит, что всегда, когда допустим ***Y***, можно использовать ***X*** (§ 11.4.1). Конструктор ***Y*::*Y*(*X*)** выражает то же самое отношение. Отметим, что оператор преобразования (и конструктор) производят новый объект, а не изменяют тип существующего. Объявление функции преобразования в ***Y*** — это просто способ запросить *неявное* применение функции, которая вернет ***Y***. Поскольку неявное применение преобразований, определяемых конструкторами и операторами преобразования, может порой оказаться предательским, иногда при проектировании полезно анализировать их отдельно.

Важно гарантировать, что графы преобразований для программы не содержат циклов. Если они содержат циклы, ошибки неоднозначности сделают невозможным комбинирование входящих в цикл типов. Например:

```
class Rational;          // рациональное число
class Big_int {          // большое целое
public:
```

```
        friend Big_int operator+ (Big_int, Big_int);
        operator Rational ();
        // ...
};

class Rational {
public:
        friend Rational operator+ (Rational, Rational);
        operator Big_int ();
        // ...
};
```

Типы ***Rational*** и ***Big_int*** взаимодействуют не так гладко, как кто-то мог бы надеяться:

```
void f(Rational r, Big_int i)
{
        g (r+i);                    // ошибка, неоднозначность: operator+(r, Rational(i))
                                    // или operator+(Big_int(r), i)?
        g (r+Rational (i));         // одно явное разрешение
        g (Big_int (r)+i);          // другое явное разрешение
}
```

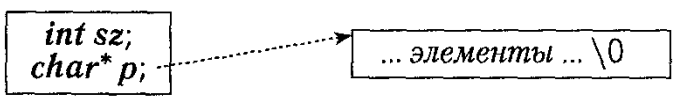
Можно избежать таких «взаимных» преобразований, сделав хотя бы некоторые из них явными. Например, преобразование ***Big_int*** в ***Rational*** вместо оператора преобразования можно определить как ***make_Rational*** (), и тогда сложение разрешилось бы как ***g (Big_int (r), i)***. Там, где операторов «взаимного» преобразования не избежать, нужно разрешать получающиеся конфликты либо явными преобразованиями, как показано выше, либо определять много различных версий бинарных операторов, таких как +.

24.3.7. Отношения внутри класса

Класс может скрывать почти неограниченное количество деталей реализации и почти неограниченное количество мусора — и иногда ему приходится это делать. Однако сами объекты большинства классов имеют регулярную структуру, и легко описать, как ими манипулировать. Объект класса — это набор других подобъектов (которые часто называют членами), и многие из них являются указателями и ссылками на другие объекты. Таким образом, объект можно представить себе как корень дерева объектов, а упомянутые выше объекты образуют как бы «иерархию объектов», дополняющую иерархию классов, как описано в § 24.3.2.1. Например, рассмотрим очень простой класс ***String***:

```
class String {
        int sz;
        char* p;
public:
        String (const char* q);
        ~String ();
        // ...
};
```

Объект ***String*** можно представить графически следующим образом:



24.3.7.1. Инварианты

Значения членов и объектов, на которые ссылаются члены, собирательно называются *состоянием* объекта (или просто его *значением*). Главная забота при проектировании класса — привести объект в четко определенное состояние (инициализация/конструирование), поддерживать четко определенное состояние при выполнении операций и наконец изящно уничтожить объект. Свойство, делающее состояние объекта четко определенным, называется *инвариантом*.

Таким образом, целью инициализации является приведение объект в состояние, для которого выполняется инвариант. Как правило, это делается конструктором. Каждая операция над классом может полагаться на то, что она найдет инвариант истинным при входе, и она должна оставить его истинным при выходе. Деструктор окончательно отменяет инвариант, уничтожая объект. Например, конструктор ***String***::***String*** (***const char****) гарантирует, что ***p*** указывает на массив по крайней мере из ***sz+1*** элементов, где ***sz*** имеет осмысленное значение, и ***p*[*sz*]==*0***. Каждая строковая операция должна оставлять это утверждение истинным.

Искусство проектирования классов включает в себя умение сделать класс достаточно простым, чтобы при реализации он имел полезный и просто выражаемый инвариант. Просто заявить, что каждому классу нужен инвариант. Труднее получить полезный инвариант, который легко понять, и который не подразумевает неприемлемых ограничений на реализацию или эффективность операций. Отметим, что «инвариант» здесь можно понимать также как фрагмент кода, который можно выполнить для проверки состояния объекта. Вполне возможно более строгое, математическое понятие, и в некоторых контекстах оно более уместно. Инвариант, который обсуждается здесь — это практическая — и поэтому, как обычно, экономная и логически незавершенная — проверка состояния объекта.

Понятие инварианта ведет свое происхождение от работы Флойда, Наура и Хоара (Floyd, Naur, and Hoare) о предусловиях и постусловиях, и в последние 30 лет оно присутствует практически во всех работах по абстрактным типам данных и верификации программ. Оно также является основой отладки в С.

Как правило, инвариант не сохраняется во время выполнения функции-члена. Функции, которые кто-то может вызвать в то время, когда инвариант не выполняется, не должны быть частью открытого интерфейса. Для этой цели могут служить закрытые и защищенные функции.

Как нам выразить понятие инварианта в программе на С++? Простой путь — определить функцию проверки инварианта и в открытые операции вставить ее вызов. Например:

```
class String {
        int sz;
        char* p;
public:
        class Range { };                    // классы исключений
        class Invariant { };

        void check ();                      // проверка инварианта

        String (const char* q);
        String (const String&);
        ~String ();
```

```
        char& operator[] (int i);
        int size () { return sz; }
        //...
};
void String::check ()
{
        if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz-1])
                throw Invariant ();
}
char& String::operator[] (int i)
{
        check ();                               // проверка на входе
        if (i<0 || sz<=i) throw Range ();       // работаем
        check ();                               // проверка на выходе
        return p[i];
}
```

Это будет неплохо работать и вряд ли представляет какую-либо сложность для программиста. Однако для простого класса вроде ***String*** проверка инварианта займет большую часть времени выполнения и, может быть, даже большую часть кода. Поэтому программисты часто выполняют проверку инварианта только во время отладки:

```
inline void String::check ()
{
#ifndef NDEBUG
        if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz]) throw Invariant ();
#endif
}
```

Здесь макрос ***NDEBUG*** используется аналогично тому, как он применяется в стандартном макросе C ***assert*** (). Принято устанавливать ***NDEBUG*** в знак того, что отладка *не* производится.

Простое действие по определению инвариантов и использование их во время отладки оказывает неоценимую помощь в получении правильного кода и — что более важно — в том, чтобы выраженные классами понятия были четко определены и регулярны. Дело в том, что когда вы будете проектировать инварианты, класс будет рассматриваться с другой точки зрения, и код будет содержать избыточность. И то, и другое увеличивает вероятность выявления ошибок, несовместимостей и оплошностей.

24.3.7.2. Утверждения

Инвариант — это особая форма утверждения. Утверждение — это просто высказывание о том, что некий логический критерий должен выполняться. Вопрос в том, что делать, когда он не выполняется.

Стандартная библиотека C — и стало быть, стандартная библиотека C++ — предоставляет в заголовочных файлах <***cassert***> или <***assert.h***> макрос ***assert*** (). Макрос ***assert*** () вычисляет свой аргумент, и если результат равен нулю (***false***), вызывает ***abort*** (). Например:

```
void f(int* p)
```

```
{
    assert(p!=0);       // прекращение программы, если p равно нулю
    //...
}
```

Перед прекращением программы, ***assert*** () выводит имя исходного файла и номер строки, где прекратилось выполнение. Этим ***assert*** () оказывает большую помощь при отладке. ***NDEBUG*** обычно устанавливается опциями компилятора для каждой единицы компиляции отдельно. Это подразумевает, что ***assert*** () не следует использовать во встроенных функциях и шаблонах функций, которые включаются в несколько единиц трансляции, разве что тщательно следя за тем, что ***NDEBUG*** везде установлен одинаково (§ 9.2.3). Как и вся остальная «макросомагия», ***NDEBUG*** относится к слишком низкому уровню, слишком запутан и провоцирует ошибки. Кроме того, бывает полезно оставить активными по крайней мере некоторые средства проверки даже в отлично проверенной программе, а ***NDEBUG*** не очень хорошо для этого подходит. Более того, вызов ***abort*** () редко приемлем в промышленном коде.

Однако часто полезно оставить хотя бы некоторые проверки активными даже в прекрасно проверенной программе, а ***NDEBUG*** не очень хорошо для этого подходит. К тому же вызов ***abort*** () редко приемлем для готовых программ.

Альтернатива заключается в использовании шаблона ***Assert*** (), который генерирует исключение, а не прекращает программу, так что утверждения можно оставить в готовой программе, если это желательно. К сожалению, стандартная библиотека не предоставляет ***Assert*** (). Однако он очень просто определяется:

```
template<class X, class A> inline void Assert(A assertion)
{
    if (!assertion) throw X();
}
```

Assert () генерирует исключение ***X*** (), если ***assertion*** ложно. Например:

```
class Bad_arg { };

void f(int* p)
{
    Assert<Bad_arg>(p!=0);          //проверки p!=0; генерация Bad_arg, если p!=0
    //...
}
```

При этом стиле утверждений условия записываются явно, поэтому, если мы хотим проверять только во время отладки, мы должны так и сказать. Например:

```
void f2(int* p)
{
    Assert<Bad_arg>(NDEBUG || p!=0);    // либо нет отладки, либо p!=0
    //...
}
```

Употребление в утверждении || в отличие от && может показаться удивительным. Однако ***Assert*** <***E***>(***a***||***b***) проверяет !(***a***||***b***), что эквивалентно !***a***&&!***b***).

Использование ***NDEBUG*** в этом случае требует, чтобы мы определили ***NDEBUG*** со значением, говорящим, находимся мы в режиме отладки или нет. Реализация C++ не делает этого для нас по умолчанию, поэтому лучше пользоваться значением. Например:

```
#ifdef NDEBUG
const bool ARG_CHECK = false;   // не отлаживаем — запретить проверки
#else
const bool ARG_CHECK = true;    // отлаживаем
#endif

void f3 (int *p)
{
    Assert<Bad_arg> ( !ARG_CHECK || p !=0 );      // либо нет отладки, либо p!=0
}
```

Если исключение, связанное с утверждением, не перехвачено, неудавшийся ***Assert*** () заканчивает программу (функцией ***terminate*** ()) почти так же, как макрос ***assert*** () вызывает ***abort*** (). Однако обработчик исключений может предпринять не такие радикальные действия.

В любой программе осмысленных размеров для тестирования я включаю и выключаю утверждения в группах. Функция ***assert*** () языка С — это просто самая грубая форма данного приема. На ранних этапах разработки большинство утверждений включены, однако после сдачи программы включенными остаются только основные проверки. Такой стиль использования легче всего управляется, если действительное утверждение делится на две части, где первая является разрешающим условием (переменной, разрешающей ту или иную проверку, такой как ***ARG_CHECK***), а вторая — собственно утверждением.

Если разрешающее условие — это константное выражение, то когда оно не включено, все утверждение просто не будет скомпилировано. Однако разрешающее условие может быть и переменным, так что утверждение можно включать и выключать во время выполнения программы, как того требует отладка. Например:

```
bool string_check = true;

inline void String::check ()
{
    Assert<Invariant> (!string_check || (p && 0<=sz && sz<TOO_LARGE && p[sz]==0));
}

void f ()
{
    String s;
    // здесь строки проверяются
    string_check=false;
    // здесь строки не проверяются
}
```

Естественно, в таких случаях код будет генерироваться, поэтому если мы активно пользуемся утверждениями, то должны следить, не начнется ли разбухание кода.

Инструкция

```
Assert<E> (a);
```

это просто другой способ сказать

```
if (!a) throw E ();
```

Так зачем же связываться с ***Assert*** (), а не написать прямо эту инструкцию? Дело в том, что применение ***Assert*** () делает намерение проектировщика явным. Оно говорит, что это утверждение о чем-то таком, что всегда предполагается истинным. Это не обычная часть программной логики. И, тем самым, ценная информация для тех,

кто будет читать программу. Более практическое преимущество состоит в том, что в тексте проще искать ***assert*** () или ***Assert*** () нежели осуществлять нетривиальный поиск условных выражений, генерирующих исключения.

Assert () можно обобщить для генерации исключений с аргументами и переменных исключений:

```
template<class A, class E> inline void Assert (A asserion, E except)
{
        if (!assertion) except;
}
struct Bad_g_arg {
        int* p;
        Bad_g_arg (int* pp) : p (pp) { }
};
bool g_check = true;
int g_max = 100;
void g (int* p, exception e)
{
        Assert (!g_check || p!=0, e);                          // указатель допустим
        Assert (!g_check || (0<*p&&*p<=g_max), Bad_g_arg (p)); // значение допустимо
        // ...
}
```

Во многих программах критически важно, чтобы там, где утверждение можно вычислить во время компиляции, никакой код для ***Assert*** () не генерировался. К сожалению, некоторые компиляторы не могут добиться этого для обобщенного ***Assert*** (). Поэтому двухаргументный шаблон ***Assert*** () следует применять, только когда исключение представлено не в форме ***E*** (), и когда допустимо, чтобы некоторый код генерировался независимо от значения утверждения.

В § 23.4.3.5 упоминалось, что две самые распространенные формы реорганизации иерархии классов — это расщепление класса на два и выделение общей части двух классов в базовый класс. В обоих случаях ключ к потенциальной реорганизации могут дать четко определенные инварианты. В созревшем для расщепления классе сравнение инварианта с кодом операций выявит большую часть избыточных проверок инварианта. В таких случаях подмножества операций будут иметь доступ только к соответствующим подмножествам состояния объекта. И наоборот — классы, созревшие для объединения, будут иметь схожие инварианты, даже если детали их реализации различаются.

24.3.7.3. Предусловия и постусловия

Одним из распространенных применений утверждений является выражение предусловий и постусловий для функции. То есть проверка того, что выполняются основные предположения при входе в функцию (например, о входных данных), и что функция оставляет мир в правильном состоянии на выходе. К сожалению, утверждения, которые нам хотелось бы проверить, часто оказываются на более высоком уровне, чем язык программирования позволяет нам выразить удобным и эффективным образом. Например:

```
template<class Ran> void sort (Ran first, Ran last)
{
        Assert<Bad_sequence>                    // плохая последовательность?
        ("[first, last) - допустимая последовательность"); // псевдокод
```

```
    // сортирующий алгоритм
    Assert<Failed_sort>                    // неправильная сортировка?
        ("[first, last) располагается в порядке возрастания");    // псевдокод
}
```

Это фундаментальная проблема. То, что мы хотим сказать о программе, лучше всего выражается на языке высокого уровня, основанном на математике, а не на алгоритмическом языке программирования, *на котором* мы пишем программу.

Что касается инвариантов, то для того, чтобы перевести идеальное утверждение, которое нам хотелось бы высказать, в нечто такое, что можно алгоритмически проверить, нужна определенная доля интеллекта. Например:

```
template<class Ran> void sort (Ran first, Ran last)
{
    // [first, last) — допустимая последовательность:
    // проверяем правдоподобие
    Assert<Bad_sort_sequence> (NDEBUG || first>last);

    // сортирующий алгоритм

    // [first, last) располагается в порядке возрастания:
    // проверяем образец
    Assert<Failed_sort> (NDEBUG ||
        (last-first)<2 || (*first<=last[-1]
            && *first<=first[ (last-first)/2] && first[ (last-first)/2]<= last[-1]));
}
```

Часто я обнаруживаю, что проще писать обычный код с проверкой аргументов и результатов, чем составлять утверждения. Однако важно попытаться выразить реальные (идеальные) предусловия и постусловия — и хотя бы документировать их в комментариях, — прежде чем свести их к чему-то менее абстрактному, что можно выразить на языке программирования.

Проверка предусловий может легко выродиться в простую проверку значений аргументов. Поскольку один аргумент часто передается нескольким функциям, эта проверка может многократно повторяться и стать довольно дорогой. Однако простое утверждение, что все аргументы-указатели во всех функциях не равны нулю, не очень помогает и может вызвать ложное чувство безопасности — особенно если тесты выполняются только во время отладки, дабы предотвратить излишние затраты. Это главная причина того, почему я советую сосредоточить внимание на инвариантах.

24.3.7.4. Инкапсуляция

Отметим, что в C++ единицей инкапсуляции является класс — а не отдельный объект. Например:

```
class List {
    List* next;
public:
    bool on (List*);
    //...
};
bool List::on (List* p)
{
```

```
        if (p == 0) return false;
        for (List* q = this; q; q=q->next) if (p == q) return true;
        return false;
}
```

Доступ к закрытому указателю ***List::next*** разрешен, поскольку функция ***List::on*** () имеет доступ ко всем объектам класса ***List***, на которые она может как-то ссылаться. Там, где это неудобно, можно упростить положение, не пользуясь преимуществом доступа к представлению других объектов из функции-члена. Например:

```
bool List::on (List* p)
{
        if (p == 0) return false;
        if (p == this) return true;
        if (next == 0) return false;
        return next->on (p);
}
```

Однако это превращает итерацию в рекурсию, что может нанести серьезный удар по быстродействию, если компилятор не способен оптимизировать и превратить рекурсию обратно в итерацию.

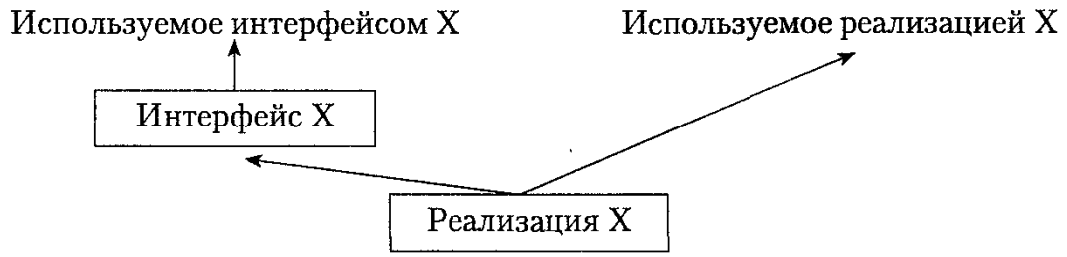
24.4. Компоненты

Единица проектирования — это множество классов, функций и т. д. , а не отдельный класс. Такое множество, часто называемое *библиотекой* или *средой разработки* (§ 25.8), является также единицей повторного использования (§ 23.5.1), сопровождения и т. д. Для выражения понятия о наборе возможностей, объединенных неким логическим критерием, C++ предоставляет три механизма:

[1] классы, содержащие наборы данных, функций, шаблонов и типизированных членов;
[2] иерархии классов, содержащие набор классов;
[3] пространства имен, содержащее наборы данных, функций, шаблонов и типизированных членов.

Класс обеспечивает множество средств, делающих более удобным создание объектов того типа, который он определяет. Однако механизм создания объектов единственного типа не лучшим образом описывает многие важные компоненты. Иерархия классов выражает понятие множества родственных типов. Однако отдельные члены компоненты не всегда лучше всего выражаются классами, и не все классы обладают достаточным сходством, требующимся, чтобы вписать их в осмысленную иерархию классов (§ 24.2.5). Поэтому самым прямым и универсальным воплощением идеи «компонентности» является пространство имен. Компоненту часто называют «категорией классов». Однако не все элементы компоненты являются или должны являться классами.

В идеале компонента описывается набором интерфейсов, которыми она сама пользуется для своей реализации, плюс набор интерфейсов, которые она предоставляет пользователям. Все прочее — это «детали реализации», скрытые от остальной системы. Такой набор в самом деле может служить для проектировщика описанием компоненты. Чтобы реализовать ее, программисту нужно отобразить эти интерфейсы в объявления. Классы и иерархии классов предоставляют интерфейсы, а пространства имен позволяют программисту сгруппировать интерфейсы и отделить используемые интерфейсы от предоставляемых интерфейсов.



При использовании техники, описанной в § 8.2.4.1, это превращается в следующее:

```
namespace A {    // некоторые средства, используемые интерфейсом X
        //...
}
namespace X {                    // интерфейс компоненты X
        using namespace A;       // зависит от объявлений из A
        //...
        void f();
}
namespace X_impl {               // средства, нужные для реализации X
        using namespace X;
        //...
}
void X::f()
{
        using namespace X_impl;  // зависит от объявлений из X_impl
}
```

Универсальный интерфейс ***X*** не должен зависеть от интерфейса реализации, то есть от ***X_impl***.

Компонента может иметь много классов, не предназначенных для общего использования. Такие классы следует «прятать» внутрь классов реализации или пространств имен:

```
namespace X_impl {    // детали реализации компоненты X
        class Widget {
                //...
        };
        //...
};
```

Это гарантирует, что ***Widget*** не используется другими частями программы. Однако, классы, представляющие согласованные понятия, часто являются кандидатами на повторное использование, и поэтому следует рассмотреть включение их в интерфейс компоненты.

```
class Car {
        class Wheel {
                //...
        };
        Wheel flw, frw, rlw, rrw;
        //...
```

```
public:
        //...
};
```

В большинстве применений нам нужно спрятать действительные колеса (wheels), чтобы сохранить абстракцию автомобиля (когда вы пользуетесь автомобилем, вы не можете независимо оперировать колесами). Однако класс колеса ***Wheel*** сам по себе кажется хорошей кандидатурой для более широкого использования, поэтому, может быть, его лучше вынести из класса ***Car*** (автомобиль):

```
class Wheel {
        //...
};
class Car {
        Wheel flw, frw, rlw, rrw;
public:
        //...
};
```

Решение, вставлять один класс в другой или нет, зависит от целей проектирования и универсальности участвующих в процессе понятий. И вставка, и «не-вставка» класса являются широко применяемыми проектными приемами. По умолчанию класс следует максимально локализовать, пока не появится необходимость сделать его более доступным.

У «интересных» функций есть отвратительная тенденция «всплывать» в глобальное пространство имен, в широко используемые пространства имен или в высший базовый класс в иерархии. Это может запросто привести к непреднамеренному раскрытию деталей реализации и к проблемам, связанным с глобальными данными и функциями. С большой вероятностью это случается в иерархиях с одним корнем и в программах, где используется очень мало пространств имен. В контексте иерархии классов для борьбы с этим явлением можно воспользоваться виртуальными базовыми классами (§ 15.2.4). Для избежания проблем в контексте пространств имен главным средством являются маленькие «реализационные» пространства имен.

Отметим, что заголовочные файлы предоставляют мощный механизм обеспечения разного представления компоненты для разных пользователей, а также для исключения классов, которые с точки зрения пользователя являются частью реализации (§ 9.3.2).

24.4.1. Шаблоны

С точки зрения проектирования шаблоны служат двум, мало связанным между собой целям:

- обобщенному программированию;
- политике параметризации

На ранних стадиях проектирования операции являются просто операциями. Позже, когда настает время описать типы операндов, для статически типизированных языков программирования, каковым является C++, шаблоны приобретают огромную

важность. Без шаблонов определения функций стали бы повторяться, или проверка типов неизбежно отражалась бы на времени выполнения программы (§ 24.2.3). Операция, реализующая алгоритм для операндов разных типов, является кандидатом на реализацию в виде шаблона. Если все операнды вписываются в единую иерархию классов, и особенно если есть необходимость добавлять новые типы операндов во время выполнения программы, тип операнда лучше всего представить классом — часто абстрактным классом. Если типы операнда не вписываются в единую иерархию, и особенно если критично быстродействие, эти операции лучше реализовать в виде шаблона. Стандартные контейнеры и поддерживающие их алгоритмы являются примером того, когда необходимость принимать операнды множества не связанных между собой типов в сочетании с требованиями быстродействия приводят к использованию шаблонов (§ 16.2).

Чтобы сделать альтернативу шаблон/иерархия более конкретной, рассмотрим, как обобщить простую итерацию:

```
void print_all (Iter_for_T x)
{
        for (T* p=x.first (); p; p = x.next ()) cout << *p;
}
```

Здесь предполагается, что итератор ***Iter_for_T*** обеспечивает операции, дающие ***T****.

Мы можем сделать итератор ***Iter_for_T*** параметром шаблона:

```
template<class Iter_for_T> void print_all (Iter_for_T x)
{
        for (T* p=x.first (); p; p = x.next ( )) cout << *p;
}
```

Это позволяет нам использовать разнообразные не связанные между собой итераторы, если все они предоставляют ***first*** () и ***next*** () с правильным смыслом, и если во время компиляции мы знаем тип итератора для каждого вызова ***print_all*** (). На этой идее основываются контейнеры и алгоритмы стандартной библиотеки.

С другой стороны, мы можем воспользоваться наблюдением, что ***first*** () и ***next*** () образуют интерфейс для итераторов. Тогда для представления этого интерфейса мы можем определить класс:

```
class Iter {
public:
        virtual T* first () const = 0;
        virtual T* next () = 0;
};

void print_all2 (Iter& x)
{
        for (T* p = x.first (); p; p = x.next ()) cout << *p;
}
```

Теперь мы можем пользоваться любыми итераторами, производными от ***Iter***. Используем ли мы для представления параметризации шаблоны или иерархию классов, это не влияет на действительный код — отличаются только быстродействие, скорость повторной компиляции и т. п. В частности, класс ***Iter*** является кандидатом на использование в качестве аргумента шаблона:

```
void f(Iter& i)
{
    print_all(i);      // использование шаблона
    print_all2(i);
}
```

Следовательно, эти два подхода мы можем рассматривать как дополняющие друг друга.

Часто шаблону как частью своей реализации нужно пользоваться функциями и классами. Многие из них для сохранения универсальности и эффективности сами должны быть шаблонами. В этом смысле алгоритмы становятся обобщенными по множеству типов. Этот стиль использования шаблонов называется *обобщенным программированием* (§ 2.7). Когда мы вызываем ***std*::*sort*** () для вектора, элементы вектора служат операндами сортировки ***sort*** (); таким образом, алгоритм ***sort*** является обобщенным по типу сортируемых элементов. Кроме того, стандартная сортировка является обобщенной по типам контейнеров, поскольку к ней можно обращаться для произвольных контейнеров, удовлетворяющих стандартам (§ 16.3.1).

Алгоритм ***sort*** () также параметризуется критерием сравнения (§ 18.7.1). С точки зрения проектирования, это отличается от принятия операции в качестве аргумента и от обобщения алгоритма по типу операнда. Решение параметризовать алгоритм для объекта (или операции) так, чтобы управлять действиями алгоритма, принадлежит гораздо более высокому уровню проектирования. Это решение дает проектировщику/программисту возможность влиять на политику выполнения определенных действий в алгоритме. Однако с точки зрения языка программирования здесь нет никакой разницы.

24.4.2. Интерфейсы и реализации

Идеальный интерфейс

- представляет пользователю полный и согласованный набор понятий;
- совместим по всем частям компоненты;
- не открывает пользователю подробностей реализации;
- может быть реализован несколькими способами;
- статически типизируется;
- выражается при помощи типов прикладного уровня;
- ограничен и строго определенным образом зависит от других интерфейсов.

Отметив необходимость согласованности по классам, представляющим для внешнего мира интерфейс компоненты (§ 24.4), давайте забудем об этом и упростим обсуждение, рассматривая лишь один класс.

```
class Y{ /* ... */};      // нужен для X
class Z{ /* ... */};      // нужен для X
class X{                  // пример плохого интерфейса
    Y a;
    Z b;
public:
    void f(const char* ...);
    void g(int[], int);
    void set_a(Y&);
    Y& get_a();
};
```

Этот интерфейс заключает в себе несколько потенциальных проблем:

- Он пользуется типами ***Y*** и ***Z*** так, что для компиляции требуется знать их объявления.
- Функция ***X***::*f* принимает произвольное число аргументов неизвестного типа (вероятно под управлением некоторой «форматирующей строки», заданной в качестве первого аргумента; § 21.8).
- Функция ***X***::*g* принимает аргумент *int*[]. Это может быть допустимо, но обычно является признакам того, что уровень абстракции слишком низок. Массив целых чисел не описывает сам себя, поэтому неочевидно, сколько элементов он в себе содержит.
- Функции *set_a* () и *get_a* () с большой вероятностью открывают представление объектов класса ***X***, позволяя прямой доступ к ***X***::*a*.

Эти функции-члены предоставляют интерфейс на очень низком уровне абстракции. По сути дела классы с интерфейсами такого уровня принадлежат к деталям большей компоненты — если они вообще чему-либо принадлежат. В идеале аргумент интерфейсной функции несет в себе достаточно информации, чтобы описать самого себя. Эмпирическое правило таково: должна быть возможность передать удаленному серверу как можно более краткий запрос на обслуживание.

C++ позволяет программисту выставить представление класса наружу как часть интерфейса. Это представление может быть спрятано (при помощи ***private*** или ***protected***), но оно доступно компилятору, чтобы распределить память под автоматические переменные, позволить встраивание функций и т. д. Отрицательный эффект от этого состоит в том, что использование классов-типов в представлении класса может внести нежелательные взаимозависимости. Является ли проблемой использование членов типа ***Y*** и ***Z***, зависит от того, какими типами ***Y*** и ***Z*** являются на самом деле. Если это простые типы, такие как *list*, *complex* и *string* , их использование как правило вполне оправдано. Такие типы можно считать стабильными, и необходимость включать их в объявления классов — приемлемая нагрузка для компилятора. Однако если бы ***Y*** и ***Z*** сами были интерфейсными классами важных компонент, таких как графическая система или система управления банковскими расчетами, было бы разумно не зависеть от них «слишком непосредственно». В подобных случаях лучшим решением является использование члена-указателя или ссылки:

```
class Y;
class Z;

class X {     // X имеет доступ к Y и Z только через указатели и ссылки
        Y* a;
        Z& b;
        // ...
};
```

Это разъединяет определение ***X*** и определения ***Y*** и ***Z***; то есть определение ***X*** зависит только от имен ***Y*** и ***Z***. Реализация ***X*** будет, конечно же, по-прежнему зависеть от определений ***Y*** и ***Z***, но это не будет вредить пользователям ***X***.

Это иллюстрирует важную мысль: интерфейс, скрывающий значительную часть информации, — как и должен делать хороший интерфейс — будет иметь гораздо меньше зависимостей, чем скрываемая им реализация. Например, определение класса ***X***

можно откомпилировать без доступа к определениям ***Y*** и ***Z***. Однако определения функций-членов класса ***X***, манипулирующих объектами ***Y*** и ***Z***, будут нуждаться в доступе к определениям ***Y*** и ***Z***. Когда анализируются зависимости, зависимости интерфейса и реализации должны рассматриваться раздельно. В обоих случаях, чтобы облегчить понимание и тестирование системы, было бы идеально, если бы графы зависимости системы были направленными ациклическими графами. Однако этот идеал гораздо более важен и гораздо чаще достижим для интерфейсов, чем для реализаций.

Отметим, что класс может определять три интерфейса:

```
class X{
private:        // закрытый
                // доступны только для членов и друзей
protected:      // защищенный
                // доступны только для членов и друзей,
                // а также членов и друзей производных классов
public:         // открытый
                // доступны для широкой публики
};
```

Кроме того, друзья ***friend*** являются частью открытого интерфейса (§ 11.5).

Член должен быть частью самого ограничительного интерфейса. То есть, член должен быть закрытым, если нет причины сделать его более доступным. Если он все же должен быть более доступным, его нужно сделать защищенным, если только нет причины его открывать. Практически во всех случаях делать члены данных открытыми или защищенными очень плохо. Не члены данных, а функции и классы, составляющие открытый интерфейс, должны представлять взгляд на класс, согласующийся с его ролью, как выразителя некоторой концепции.

Отметим, что для следующего уровня сокрытия представления можно использовать абстрактные классы (§ 2.5.4, § 12.3, § 25.3).

24.4.3. Жирные интерфейсы

В идеале интерфейс должен предлагать только те операции, которые имеют смысл и могут быть хорошо реализованы каждым производным классом, реализующим этот интерфейс. Однако это не всегда легко. Рассмотрим списки, массивы, ассоциативные массивы, деревья и т. п. Как показано в § 16.2.2, соблазнительно и иногда полезно обобщить все эти типы в один — обычно называемый *контейнером* — который можно было бы использовать в качестве интерфейса ко всем ним. Это (казалось бы) избавит пользователя от необходимости иметь дело с деталями всех этих контейнеров. Однако определить интерфейс универсального контейнерного класса не так просто. Допустим, мы хотим определить ***Container*** как абстрактный тип. Какие операции должен он предоставить? Мы можем ввести только те операции, которые может поддержать каждый контейнер (пересечение множеств операций), но это до смешного узкий интерфейс. Фактически, во многих интересных случаях это пересечение представляет собой пустое множество. В качестве альтернативы мы можем ввести объединение всех множеств операций и выдавать ошибку во время выполнения, если через интерфейс к объекту будет применена «несуществующая операция». Интерфейс, представляющий собой такое объединение интерфейсов к множеству концепций, называется *жирным интерфейсом*. Рассмотрим «универсальный контейнер» объектов типа ***T***:

```
class Container {
public:
        struct Bad_oper {        // класс исключений
                const char* p;
                bad_oper (const char* pp) : p (pp) { }
        };

        virtual void put (const T*) { throw Bad_oper ("Container::put"); }
        virtual T* get () { throw Bad_oper ("Container::get"); }

        virtual T*& operator[] (int) { throw Bad_oper ("Container::[] (int)"); }
        virtual T*& operator[] (const char*) { throw Bad_oper ("Container::[] (char*)"); }
        // ...
};
```

Тогда конкретные контейнеры можно объявить так:

```
class List_container : public Container, private list {
public:
        void put (const T*);
        T* get ();
        // ... нет оператора operator[] ...
};

class Vector_container : public Container, private vector {
public:
        T*& operator[] (int);
        T*& operator[] (const char*);
        // ... нет функций put() и get() ...
};
```

Пока мы внимательны, все в порядке:

```
void f ()
{
        List_container sc;
        Vector_container vc;
        // ...
        user (sc, vc);
}

void user (Container& c1, Container& c2)
{
        T* p1 = c1.get ();
        T* p2 = c2[3];
        // не используются c2.get() и c1[3]
        // ...
}
```

Однако не многие структуры данных хорошо поддерживают и индексацию, и операции в стиле списков. Поэтому, вероятно, это не очень хорошая идея — определить интерфейс, требующий и то, и другое. Это приводит к интенсивным опросам во время выполнения (§ 15.4) или к обработке исключений (глава 14), чтобы избежать ошибок времени выполнения. Например:

```
void user2 (Container& c1, Container& c2)     // обнаружение легко, но исправление
                                              // может оказаться тяжелым
```

```
{
    try {
        T* p1 = c1.get();
        T* p2 = c2[3];
        //...
    }
    catch (Container::Bad_oper& bad) {
        // Ошибка!
        // И что теперь?
    }
}
```

или

```
void user3 (Container& c1, Container& c2)    // раннее обнаружение утомительно;
                                             // исправление по-прежнему
                                             // может быть тяжелым
{
    if (dynamic_cast<List_container*> (&c1) &&
        dynamic_cast<Vector_container*> (&c2)) {
    T* p1 = c1.get();
    T* p2 = c2[3];
    //...
    }
    else {
        // Ошибка!
        // И что теперь?
    }
}
```

В обоих случаях может пострадать быстродействие, и сгенерированный код может оказаться на удивление большим. В результате программисты будут соблазняться не обращать внимания на потенциальные ошибки и надеяться, что на самом деле их не возникнет, когда программа окажется у пользователей. Проблема с данным подходом состоит в том, что исчерпывающее тестирование также тяжело и дорого.

Поэтому жирных интерфейсов лучше избегать, когда на первое место ставится быстродействие, когда требуются строгие гарантии корректности, и вообще когда есть хорошая альтернатива. Использование жирных интерфейсов ослабляет взаимосвязь между концепциями и классами и таким образом открывает простор для использования наследования как просто удобства реализации.

24.5. Советы

[1] Активнее используйте абстракцию данных и объектно-ориентированное программирование; § 24.2

[2] По мере надобности (только) используйте методы и приемы C++; § 24.2.

[3] Стремитесь к соответствию стилей программирования и проектирования; § 24.2.1.

[4] Фокусируйте проектирование прежде всего на классах/концепциях, а не на функциях/обработке; § 24.2.2.

[5] Пользуйтесь классами для представления понятий; § 24.2.1, § 24.3.

[6] Пользуйтесь наследованием (только) для представления иерархических взаимосвязей между понятиями; § 24.2.2, § 24.2.5, § 24.3.2.

[7] Выражайте строгие гарантии относительно интерфейсов в терминах статических типов прикладного уровня; § 24.2.3.
[8] Для облегчения решения четко определенных задач пользуйтесь генераторами программ и средствами прямого манипулирования; § 24.2.4.
[9] Избегайте генераторов программ и средств прямого манипулирования, которые плохо стыкуются с универсальными языками программирования; § 24.2.4.
[10] Удерживайте разные уровни абстракции раздельными; § 24.3.1.
[11] Фокусируйте внимание на проектировании компонент; § 24.4.
[12] Убедитесь, что виртуальные функции имеют четко определенный смысл, и что все замещающие функции реализуют желаемое поведение; § 24.3.4, § 24.3.2.1.
[13] Для выражения отношения *is-a* (является) пользуйтесь открытым наследованием; § 24.3.4.
[14] Для выражения отношения *has-a* (содержит) пользуйтесь членством; § 24.3.4.
[15] Для выражения простого включения (содержания в себе) предпочитайте непосредственное членство, а не с указатель на объект; § 24.3.3, 24.3.4.
[16] Убедитесь, что зависимости *использования* понятны, не циклические (по возможности) и минимальны; § 24.3.7.2.
[17] Для всех классов определяйте инварианты; § 24.3.7.1.
[18] Предусловия/постусловия и другие утверждения явно выражайте утверждениями (возможно, с использованием ***Assert***()); § 24.3.7.2.
[19] Определяйте интерфейсы так, чтобы они открывали (необходимый) минимум информации; § 24.4.
[20] Минимизируйте зависимости интерфейса от других интерфейсов; § 24.4.2.
[21] Поддерживайте строгую типизацию интерфейсов; § 24.4.2.
[22] Выражайте интерфейсы в терминах типов прикладного уровня; § 24.4.2.
[23] Выражайте интерфейсы так, чтобы запрос к интерфейсу можно было передать на удаленный сервер; § 24.4.2.
[24] Избегайте жирных интерфейсов; § 24.4.2.
[25] Где только возможно применяйте закрытые члены данных и функции-члены; § 24.4.2.
[26] Пользуйтесь различением «защищенный/открытый» для учета различий потребностей проектировщиков производных классов и конечных пользователей; § 24.4.2.
[27] Пользуйтесь шаблонами для обобщенного программирования; § 24.4.1.
[28] Пользуйтесь шаблонами для параметризации политики выполнения алгоритмов; § 24.4.1.
[29] Пользуйтесь шаблонами, когда необходимо, чтобы разрешение типов имен производилось во время компиляции; § 24.4.1.
[30] Пользуйтесь иерархией классов, когда необходимо, чтобы разрешение типов производилось во время выполнения; § 24.4.1.

25

Роли классов

Кое-что лучше изменить, ...
но фундаментальные темы должны
упиваться своим упорством.
— Стефен Дж. Гоулд

Разновидности классов — конкретные типы — абстрактные типы — узлы — изменяющиеся интерфейсы — объектный ввод/вывод — действия — интерфейсные классы — вспомогательные классы-дескрипторы — использование счетчиков — прикладные среды разработки — советы — упражнения.

25.1. Разновидности классов

Класс в C++ — это конструкция языка программирования, которая служит разнообразным потребностям проектирования. Я нахожу, что многие запутанные проблемы проектирования решаются введением какого-то нового класса, представляющего некоторое понятие, которое не было явным в предыдущем эскизе проекта (возможно, также приходится удалять некоторые классы). Огромное разнообразие ролей, которые может играть класс, приводит к множеству разновидностей классов, каждая из которых хорошо приспособлена к решению частных задач. В этой главе описываются несколько классов-архетипов, их достоинства и недостатки:

§ 25.2 Конкретные типы
§ 25.3 Абстрактные типы
§ 25.4 Узлы
§ 25.5 Операции
§ 25.6 Интерфейсы
§ 25.7 Вспомогательные классы
§ 25.8 Прикладные среды разработки

Эти «разновидности классов» являются понятиями проектирования, а не языковыми конструкциями. Недостигнутый и, вероятно, недостижимый идеал заключается в том, чтобы получить минимальный набор простых и ортогональных разновидностей классов, из которых можно конструировать полезные, хорошо ведущие себя классы. Важно отметить, что в проекте встречаются все разновидности, и ни одна из них по природе своей не лучше других во всех отношениях. Большая путаница при обсуждении вопросов проектирования и программирования исходит от людей, старающихся использовать исключительно одну или две разновидности классов. Обычно это дела-

ется во имя простоты, однако приводит к искаженному и неестественному применению этих излюбленных ими разновидностей классов.

Приведенное здесь описание делает упор на чистые формы этих разновидностей классов. Естественно, используются и гибридные формы. Однако гибрид должен появляться в результате проектного решения, основанного на оценке инженерных альтернатив, а не в результате ведущей в никуда попытки избежать принятия решений. «Откладывание решения» слишком часто является завуалированной формой «отказа думать». Новичкам в проектировании обычно лучше держаться подальше от гибридов и следовать стилю существующих компонент, имеющих свойства, схожие с теми, что они стремятся достичь. Только опытным программистам следует пытаться написать универсальную компоненту или библиотеку, и каждый разработчик библиотеки «обречен» на использование, документирование и поддержку своего создания в течение нескольких лет. См. также, пожалуйста, § 23.5.1.

25.2. Конкретные типы

Такие классы как *vector* (§ 16.3), *list* (§ 17.2.2), *Date* (§ 10.3) и *complex* (§ 11.3, § 22.5) являются *конкретными* в том смысле, что каждый из них представляет собой относительно простое понятие со всеми операциями, существенными для поддержки этого понятия. Также в каждом из этих классов существует взаимнооднозначное соответствие между интерфейсом и реализацией, и ни один из этих классов не предназначен для создания производных классов. Как правило, конкретные типы не вписываются в иерархию родственных классов. Каждый конкретный тип можно понять в отрыве от других классов, с минимальной связью с другими классами. Если конкретный тип хорошо реализован, использующие его программы по размеру и быстродействию сравнимы с программами, написанными вручную, или специализированными версиями для реализации той же концепции. Аналогично, если реализация значительно изменяется, обычно изменяется и интерфейс, чтобы отразить это изменение. Во всем этом конкретный тип напоминает встроенные типы. Естественно, все встроенные типы тоже конкретные. Определяемые пользователем конкретные типы, такие как комплексные числа, матрицы, сообщения об ошибках, символические ссылки, часто являются фундаментальными типами для некоторых прикладных областей.

Точная природа интерфейса класса определяет, какие изменения в реализации существенны в данном контексте; более абстрактные интерфейсы оставляют больше простора для изменения реализаций, но могут снизить быстродействие. Кроме того, хорошая реализация не зависит от других классов больше, чем это абсолютно необходимо, так что класс можно использовать без расходов времени компиляции и выполнения, вызванных включением в программу других «схожих» классов.

Подведем итог. Класс, представляющий конкретный тип, предназначен для того, чтобы:

[1] точно соответствовать данному частному понятию и стратегии реализации;
[2] обеспечивать быстродействие и затраты памяти, сравнимые с «написанным вручную» кодом за счет применения встраивания и операций, полностью использующих особенности самого понятия и его реализации;
[3] в минимальной степени зависеть от других классов;
[4] быть понятным и пригодным к использованию независимо от других классов.

Результат — тесная связь между пользовательским кодом и кодом реализации. Если реализация как-либо изменяется, пользовательский код придется перекомпилировать, поскольку пользовательский код почти всегда содержит обращения к встроенным функциям и локальным переменным конкретного типа.

Название «конкретный тип» было выбрано по контрасту с распространенным термином «абстрактный тип». Взаимосвязь между конкретными и абстрактными типами обсуждается в § 25.3.

Конкретные типы не могут непосредственно выражать общность. Например, *list* и *vector* обеспечивают схожие наборы операций и могут взаимозаменяемо использоваться в каких-нибудь шаблонах функций. Однако между *list<int>* и *vector<int>* или между *list<Shape*>* и *list<Circle*>* (§ 13.6.3) никакой связи нет, хотя *мы* можем распознать их сходство.

Для бесхитростно спроектированных конкретных типов это приводит к тому, что код, использующий два разных типа схожим образом, будут выглядеть по-разному. Например, итерации по списку *List* при помощи итератора *next* () резко отличаются от итерации по вектору *Vector* при помощи индексации:

```
void my (List& sl)
{     // «естественная» итерация по списку
      for (T* p=sl.first (); p; p=sl.next ()) {
            // мой код
      }
      //...
}

void your (Vector& v)
{     // «естественная» итерация по вектору
      for (int i=0; i<v.size (); i++) {
            // ваш код
      }
      //...
}
```

Разница в стиле итераций естественна в том смысле, что операция получения следующего элемента является основной для понятия списка (но это не так для вектора), а индексация является неотъемлемой чертой вектора (но не списка). Доступность операций, которые «естественны» для избранной стратегии реализации, часто оказывается решающей для эффективности и облегчения написания программы.

Очевидная сложность здесь заключается в том, что программы для фундаментально схожих операций, таких как два приведенных выше цикла, могут выглядеть непохоже, а код, использующий разные конкретные типы для схожих операций, не взаимозаменяем. В реальных примерах приходится много думать, чтобы найти сходство, и тщательно перепроектировать, чтобы добиться выгоды от сходства, если оно найдено. Стандартные контейнеры и алгоритмы — вот пример тщательного продумывания, которое позволило задействовать сходство между конкретными типами, не теряя их эффективности и изящества (§ 16.2).

Чтобы принимать конкретный тип в качестве аргумента, именно этот конкретный тип должен быть описан в функции как тип аргумента. Не будет никаких отношений наследования, которые можно использовать, чтобы сделать объявление аргумента ме-

нее специфическим. Поэтому попытка воспользоваться сходством между конкретными типами приводит к применению шаблонов и обобщенного программирования, как описано в § 3.8. При использовании стандартной библиотеки, итерации выглядят так:

```
template<class C> void ours (C& c)
{
    for (typename C::iterator p=c.begin (); p!=c.end (); ++p) {
        // итерация согласно стандартной библиотеке
        // ...
    }
}
```

Здесь использовано фундаментальное сходство между контейнерами, что, в свою очередь, открывает возможность дальнейшей эксплуатации сходства в стандартных алгоритмах (глава 18).

Чтобы хорошо использовать конкретный тип, пользователь должен понять его частные детали. В библиотеке (как правило) не существует общих свойств для всех конкретных типов, на которые можно было бы положиться, чтобы избавить пользователя от необходимости вникать в отдельные классы. Это плата за быстродействие и эффективность. Иногда эта цена окупается, иногда нет. Также может случиться, что легче понять и использовать индивидуальный конкретный класс, чем более универсальный (абстрактный). Подобное часто происходит с классами, представляющими хорошо известные типы данных, такие как массивы и списки.

Однако отметим, что в идеале все же лучше насколько возможно скрывать реализацию, не внося серьезных ухудшений в производительность. В данном контексте большой выигрыш могут принести встроенные функции. Когда мы не скрываем члены данных, делая их открытыми или вводя функции, которые устанавливают и записывают их значения, это позволяет пользователю напрямую ими манипулировать — такую идею очень редко можно назвать удачной (§ 24.4.2). Конкретные типы должны оставаться типами, а не просто мешками битов с несколькими функциями, введенными просто для удобства.

25.2.1. Повторное использование конкретных типов

Конкретные типы редко приносят пользу в качестве базовых классов для новых производных классов. Каждый конкретный тип нацелен на обеспечение ясного и эффективного представления одного понятия. Класс, хорошо с этим справляющийся, редко является хорошей кандидатурой для создания других, но родственных классов, через открытое наследование. Такие классы чаще приносят больше пользы в качестве членов или закрытых базовых классов. Тут они могут использоваться эффективно, не смешивая свои интерфейсы и реализации с интерфейсами и реализациями новых классов, и не компрометируя их. Рассмотрим создание производного класса от класса ***Date***:

```
class My_date : public Date {
    // ...
};
```

Допустимо ли использовать ***My_date*** вместо простого ***Date***? Это, конечно, зависит от того, что представляет собой ***My_date***, но согласно моему опыту редко можно найти конкретный тип, который бы являлся хорошим базовым классом без модификации.

Конкретные типы «повторно используются» не модифицированным образом так же, как и встроенные типы, наподобие ***int*** (§ 10.3.4). Например:

```
class Date_and_time {
private:
        Date d;
        Time t;
public:
        //...
};
```

Такая форма использования (повторного?) обычно проста, эффективна и продуктивна.

Может быть, было ошибкой не проектировать ***Date*** так, чтобы его было легко модифицировать через наследование? Иногда заявляют, что *каждый* класс должен быть открыт для модификации путем замещения функций-членов и доступа функций-членов производных классов. Эта точка зрения приводит примерно к такому варианту ***Date***:

```
class Date2 {
public:
        // открытый интерфейс, состоящий в основном из виртуальных функций
protected:
        // другие детали реализации (возможно, включающие некоторое представление)
private:
        // представление и другие детали реализации
};
```

Чтобы сделать замещение функций, устанавливающих значения, легким и эффективным, представление объявлено защищенным. Благодаря этому, класс ***Date2*** полностью готов к созданию произвольных классов, и в то же время его пользовательский интерфейс остается неизменным. Однако цена этого такова:

[1] *Менее эффективные базовые операции.* Обращение к виртуальным функциям в C++ немного медленнее, чем к обычным, виртуальные функции нельзя делать встроенными также часто, как невиртуальные, и класс с виртуальными функциями, как правило, приводит к перерасходу памяти по одному слову на объект.

[2] *Требуется использовать свободную память.* Цель введения ***Date2*** — позволить использовать объекты разных классов, производных от ***Date2***, взаимозаменяемым образом. Поскольку размеры этих производных классов различны, очевидно следует располагать их в свободной памяти и обращаться к ним через указатели и ссылки. Таким образом, использование истинно локальных переменных резко уменьшается.

[3] *Неудобство для пользователей.* Чтобы получить выгоду от полиморфизма, обеспечиваемого виртуальными функциями, доступ к ***Date2*** должен осуществляться через указатели или ссылки.

[4] *Относительно слабая инкапсуляция.* Виртуальные операции можно замещать, и защищенными данными можно манипулировать из производных классов (§ 12.4.1.1).

Естественно, эта цена не всегда значительна, и поведение определенного подобным образом класса часто оказывается именно тем, что мы хотели (§ 25.3, § 25.4). Однако для простого конкретного типа, такого как ***Date2***, эта цена, вероятно, достаточно высока и к тому же необязательна.

Наконец, четко спроектированный конкретный тип часто является идеальным представлением для более податливых типов. Например:

```
class Date3 {
public:
        // открытый интерфейс, состоящий в основном из виртуальных функций
private:
        Date d;
};
```

Именно этот способ следует использовать при необходимости вписать конкретные типы (в том числе встроенные) в иерархию классов. См. также § 25.10[1].

25.3. Абстрактные типы

Простейший способ ослабить связь между пользователями класса и его разработчиками, а также между создающим объекты кодом и кодом, пользующимся такими объектами, — ввести абстрактный класс, представляющий собой интерфейс ко множеству реализаций общего понятия. Рассмотрим бесхитростный класс ***Set*** (множество):

```
template<class T> class Set {
public:
        virtual void insert (T*) = 0;          // вставить
        virtual void remove (T*) = 0;          // удалить

        virtual int is_member (T*) = 0;

        virtual T* first () = 0;
        virtual T* next () = 0;

        virtual ~Set () { }
};
```

Так определяется интерфейс к множеству со встроенным понятием итерации по элементам. Типично отсутствие конструктора и наличие виртуального деструктора (§ 12.4.2). Возможно несколько реализаций (§ 16.2.1). Например:

```
template<class T> class List_set : public Set<T>, private list<T> {
        //...
};

template<class T> class Vector_set : public Set<T>, private vector<T> {
        //...
};
```

Абстрактный класс предоставляет общий интерфейс к реализациям. Это означает, что мы можем пользоваться классом ***Set***, не зная, какая из реализаций используется. Например:

```
void f (Set<Plane*>& s)
{
        for (Plane** p = s.first (); p; p = s.next ())
        {
                // мой код
        }
```

```
        //...
}

List_set<Plane*> sl;
Vector_set<Plane*> v(100);

void g()
{
        f(sl);
        f(v);
}
```

Для конкретных типов от нас бы потребовалось перепроектирование реализации классов, чтобы выразить общность, и использование шаблона для того, чтобы ее задействовать. Здесь же мы должны спроектировать общий интерфейс (в данном случае — ***Set***), но больше никакой общности, кроме необходимой для реализации этого интерфейса, не требуется.

Более того, пользователи класса ***Set*** не знают объявлений ***List_set*** и ***Vector_set***, поэтому пользователи не зависят от этих объявлений, и их (объявления) не нужно заново компилировать или как-либо изменять, если были модифицированы ***List_set*** или ***Vector_set***, или даже если была введена новая реализация ***Set*** — скажем, ***Tree_set***. Все зависимости содержатся в функциях, которые явно пользуются классом, производным от ***Set***. В частности, предполагая общепринятое использование заголовочных файлов, программисту, пишущему *f*(***Set***&), нужно включить только ***Set.h***, а не ***List_set.h*** или ***Vector_set.h***. «Реализационный заголовочный файл» нужен только там, где создаются ***List_set*** и ***Vector_set***. Реализацию можно еще дальше изолировать от действительных классов, введя абстрактный класс, обрабатывающий запросы на создание объектов («фабрику»; § 12.4.4).

Отделение интерфейса от реализаций подразумевает отсутствие доступа к операциям, «естественным» для частной реализации, но не достаточно универсальным, чтобы стать частью интерфейса. Например, поскольку в множестве нет порядка, в интерфейсе ***Set*** мы не можем поддержать индексацию, даже если нам придется реализовать частный случай шаблона ***Set***, использующий массив. Это подразумевает потери времени выполнения за счет отсутствия ручной оптимизации. Более того, встраивание функций, как правило, становится недостижимым (разве что в локальном контексте, когда компилятор знает реальный тип), и все интересные операции над интерфейсом становятся вызовами виртуальных функций. Как и в случае с конкретными типами, иногда затраты на абстрактный тип окупаются, иногда нет. Подводя итог, можно сказать, что абстрактные типы предназначены для того, чтобы:

[1] определить одно понятие таким образом, чтобы позволить сосуществовать в программе нескольким его реализациям;
[2] обеспечить приемлемое быстродействие и затраты памяти, благодаря использованию виртуальных функций;
[3] минимизировать зависимость каждой реализации от других классов;
[4] быть понятными сами по себе.

Абстрактные типы не лучше, чем конкретные, это просто другие типы. Для пользователя важно, но и трудно сделать правильный выбор. Разработчик библиотеки может уйти от проблемы выбора, предоставив и те, и другие типы, и тем самым оставив выбор за пользователем. Важно ясно представлять, какому миру принадлежит класс.

Ограничение общности абстрактного типа в попытках конкурировать с конкретным типом в быстродействии, как правило, заканчивается неудачей. Это ухудшает возможность использования взаимозаменяемых реализаций без значительной перекомпиляции после внесения изменений. Аналогично, попытки обеспечить «общность» в конкретных типах, также обычно приводят к неудаче. Это снижает эффективность и удобство применения простых классов. Эти два понятия могут сосуществовать — на самом деле они *должны* сосуществовать, поскольку конкретные типы обеспечивают реализацию абстрактных типов, — но их не следует смешивать.

Абстрактные типы часто не предназначены для создания дальнейших производных классов. Порождение классов чаще всего используется для реализации. Однако из абстрактного класса можно сконструировать новый интерфейс, создав производный от него расширенный абстрактный класс. Тогда новый абстрактный класс должен в свою очередь быть реализован путем создания производного неабстрактного класса (§ 15.2.5).

Почему мы не сделали ***List*** и ***Vector*** производными от ***Set*** сразу, ведь тогда удалось бы избежать введения классов ***List_set*** и ***Vector_set***? Иными словами, зачем иметь конкретные типы, когда можно иметь абстрактные?

[1] *Эффективность.* Нам нужны конкретные типы, такие как ***vector*** и ***list***, без затрат, связанных с отделением реализации от интерфейса (что подразумевается стилем абстрактных типов).

[2] *Повторное использование.* Нам нужен механизм вписывания типов (вроде ***vector*** или ***list***), спроектированных «где-то в другом месте», в новую библиотеку или прикладную программу, путем введения для них нового интерфейса (а не переписывая их).

[3] *Множественные интерфейсы.* Использование единого класса в качестве базового для многих классов приводит к жирным интерфейсам (§ 24.4.3). Для класса, созданного для новых задач, часто лучше обеспечить новый интерфейс (как интерфейс ***Set*** для ***vector***), а не изменять старый для подгонки под множество целей.

Естественно, эти пункты связаны между собой. Подробно они обсуждались для примера с ***Ival_box*** (§ 12.4.2, §.15.2.5) и в контексте проектирования контейнеров (§ 16.2). Использование базового класса ***Set*** привело бы к решению с контейнером, являющимся базовым классом, опирающимся на узловые классы (§ 25.4).

В разделе § 25.7 описывается итератор, более гибкий в том смысле, что связав его с реализацией, мы можем специфицировать объекты в точке инициализации и изменять их во время выполнения.

25.4. Узловые классы

Иерархия классов строится с точки зрения наследования, отличного от разделения интерфейс/реализация, характерного для абстрактных типов. Здесь класс рассматривается как фундамент для строительства. Даже если это абстрактный класс, он обычно имеет некоторое представление и предоставляет некоторые услуги своим производным классам. В качестве примеров узлового класса можно привести ***Polygon*** (§ 12.3), первоначальный ***Ival_slider*** (§ 12.4.1) и ***Satellite*** (§ 15.2).

Как правило, класс в иерархии представляет общее понятие, для которого производные классы могут считаться специализациями. Типичный класс, спроектированный как составная часть иерархии, *узловой класс*, опирается на услуги базовых клас-

сов, чтобы обеспечить свои собственные услуги. То есть он вызывает функции-члены базового класса. Типичный узловой класс обеспечивает не просто реализацию интерфейса, описанного его базовым классом (как это делает класс реализации для абстрактного типа). Он также сам добавляет новые функции, обеспечивая таким образом более широкий интерфейс. Рассмотрим класс **Car** из примера с моделированием уличного движения в § 24.3.2:

```
class Car : public Vehicle {
public:
        // passengers — пассажиры, size — размер,
        // weight — вес, fuel capacity — емкость бака
        Car (int passengers, Size_category size, int weight, int fc)
                : Vehicle (passengers, size, weight), fuel_capacity (fc) { /* ... */ }

        // замещение соответствующих виртуальных функций из Vehicle:

        void turn (Direction);          // turn — поворот, direction — направление
        // ...

        // добавление специфических для автомобиля функций:
        virtual void add_fuel (int amount);     // автомобилю для езды нужно топливо
        // ...
};
```

Важные функции являются конструкторами, посредством которых программист описывает основные свойства, существенные для модели, и (виртуальные) функции, которые позволяют имитирующим процедурам манипулировать автомобилем, не зная его точного типа. Класс **Car** можно создать и использовать следующим образом:

```
void user ()
{
        // ...
        Car* p = new Car (3, economy, 1500, 60);
        drive (p, bs_home, MH);         // вход во фрагмент, имитирующий
                                        // уличное движение
};
```

Узловому классу обычно нужны конструкторы, и часто непростые. Этим узловой класс отличается от абстрактных типов, которые редко имеют конструкторы.

Операции над классом **Car** в своих реализациях будут, как правило, пользоваться операциями из базового класса **Vehicle**. Кроме того пользователь класса **Car** опирается на услуги его базовых классов. Например, **Vehicle** обеспечивает базовые функции, относящиеся к весу и размерам, так что классу **Car** не приходится этим заниматься:

```
bool Bridge::can_cross (const Vehicle& r) {
// bridge — мост,
// can cross — можно переехать?
        if (max_weight<r.weight ()) return false;
        // ...
}
```

Это позволяет программисту создавать новые классы, такие как **Car** и **Truck**, из узлового класса **Vehicle**, описывая и реализуя только то, что должно отличаться от **Vehicle**. Это

часто называют «программированием отличий» или «программированием по расширению».

Подобно многим узловым классам, сам класс ***Car*** является хорошей кандидатурой для создания дальнейших производных классов. Например, в классе ***Ambulance*** (скорая медицинская помощь) нужны дополнительные данные и операции, связанные с экстренностью службы:

```
class Ambulance : public Car, public Emergency {
public:
        Ambulance ();

        // замещение виртуальных функций изCar:

        void turn (Direction);
        // ...

        // замещение виртуальных функций из Emergency:
        // dispatch to — «направить в», location — «место»
        virtual void dispatch_to (const Location&);
        // ...

        // добавление функций, специфичных для скорой помощи:

        virtual int patient_capacity ();      // количество носилок
        // ...
};
```

Подводя итог, можно сказать, что узловой класс:

[1] опирается на свои базовые классы как для своей реализации, так и для предоставления услуг своим пользователям;
[2] предоставляет более широкий интерфейс (то есть интерфейс с большим числом открытых функций-членов) по сравнению с базовыми классами;
[3] в своем открытом интерфейсе опирается в первую очередь (но не обязательно только) на виртуальные функции;
[4] зависит от всех своих (прямых и непрямых) базовых классов;
[5] понятен только в контексте своих базовых классов;
[6] может служить базовым для дальнейшего порождения классов;
[7] может быть использован для создания объектов.

Не все узловые классы будут удовлетворять пунктам 1, 2, 6 и 7, но большинство — удовлетворяют. Класс, не удовлетворяющий пункту 6, напоминает конкретный тип, и его можно назвать *конкретным узловым классом*. Например, конкретный узловой класс может использоваться для реализации абстрактного класса (§ 12.4.2), а переменные такого класса могут располагаться в памяти статически и в стеке. Такие классы иногда называют *классами-листьями*. Однако любой код, оперирующий с указателями или ссылками на класс с виртуальными функциями, должен учитывать возможность наличия дальнейших производных классов (или допустить, без языковой поддержки, что дальнейших производных классов нет). Класс, не удовлетворяющий пункту 7, напоминает абстрактный тип, и его можно назвать *абстрактным узловым классом*. Вследствие неудачной традиции многие узловые классы имеют хотя бы несколько защищенных членов, чтобы предоставить не столь ограниченный интерфейс для производных классов (§ 12.4.1.1).

Пункт 4 подразумевает, что для компиляции узлового класса программист должен включить объявления всех его прямых и непрямых базовых классов, а также все объявления, от которых в свою очередь зависят последние. Этим узловой класс также отличается от абстрактного типа. Пользователь абстрактного типа не зависит от классов, использующихся для его реализации, и может не включать их в текст для компиляции.

25.4.1. Изменение интерфейсов

Узловой класс по определению является частью иерархии. Не все классы в иерархии должны предлагать один и тот же интерфейс. В частности, производный класс может предоставить новые функции-члены, а его «братский» класс — совершенно другой набор функций. С точки зрения проектирования можно рассматривать динамическое приведение ***dynamic_cast*** (§ 15.4) как механизм запроса у объекта, обеспечивает ли он данный интерфейс.

Для примера разберем простую систему объектного ввода/вывода. Пользователи хотят читать объекты из потока, определять, относятся ли они к ожидаемому типу, и пользоваться ими. Например:

```
void user()
{
        // открываем файл в предположении, что он содержит фигуры (shape),
        // и прикрепляем ss в качестве потока ввода к этому файлу
        Io_obj* p = get_obj(ss); // чтение объекта из потока
        if (Shape* sp=dynamic_cast<Shape*>(p)) {
                sp->draw();             // использование фигуры
                //...
        }
        else {
                // ошибка: в файле встретилась не фигура
        }
}
```

Функция ***user*** () работает с фигурами исключительно посредством абстрактного класса ***Shape*** и поэтому может пользоваться любыми фигурами. Использование динамического приведения существенно, так как система объектного ввода/вывода может работать со многими другими видами объектов, и пользователь может по ошибке открыть файл с вполне нормальными объектами, о которых он никогда не слышал.

Такая система объектного ввода/вывода предполагает, что каждый записанный или считанный объект относится к классу, производному от ***Io_obj***. Класс ***Io_obj***, чтобы позволить нам пользоваться динамическим приведением, должен быть полиморфным типом. Например:

```
class Io_obj {
public:
        virtual Io_obj* clone() const = 0;    // полиморфный
        virtual ~Io_obj() { }
};
```

Важнейшей функцией в системе объектного ввода/вывода является ***get_obj*** (), которая считывает данные из потока ***istream*** и на основании этих данных создает объекты

класса. Допустим, что данные, представляющие объект в потоке ввода, предваряются строкой, описывающей класс данного объекта. Работа функции ***get_obj*** () заключается в чтении этой строки-префикса и вызове функции, способной создать и инициализировать объект соответствующего класса. Например:

```
typedef Io_obj* (*PF)(istream&);          // указатель на функцию, возвращающую Io_obj*
map<string, PF> io_map;                   // отображает строки в функции создания
bool get_word (istream& is, string& s);   // чтение слова из is в s
Io_obj* get_obj (istream& s)
{
    string str;
    bool b = get_word (s, str);           // чтение начального слова в str
    if (b == false) throw No_class ();    // проблема с форматом ввода/вывода

    PF f = io_map[str];                   // поиск str для получения функции
    if (f == 0) throw Unknown_class ();   // нет вхождения str

    return f (s);
}
```

Функция ***map***, которую вызывает ***io_map***, содержит пары из имен строк и функций, способных сконструировать объекты классов с этим именем.

Мы могли бы определить класс ***Shape*** обычным образом, за исключением того, что необходимо сделать его производным от ***Io_obj***, как того требует функция ***user*** ():

```
class Shape : public Io_obj {
    //...
};
```

Однако интереснее (и для многих случаев более реалистично) было бы воспользоваться уже определенным ***Shape*** (§ 2.6.2) без изменений:

```
class Io_circle : public Circle, public Io_obj {   // Circle — окружность
public:
    Io_circle* clone () const                      // используется копирующий конструктор
        { return new Io_circle (*this); }
    Io_circle (istream&);                          // инициализация из потока ввода
    static Io_obj* new_circle (istream& s) { return new Io_circle (s); }
};
```

Этот пример в первую очередь показывает, как класс можно вписать в иерархию при помощи абстрактного класса с меньшей предусмотрительностью чем та, которая бы потребовалась для того, чтобы построить его в виде узлового класса (§ 12.4.2, § 25.3). Конструктор ***Io_circle*** (***istream***&) инициализирует объект данными из своего аргумента ***istream***. Функция ***new_circle*** помещается в ***io_map***, чтобы дать знать системе объектного ввода/вывода о соответствующем классе. Например:

```
io_map["Io_circle"]=&Io_circle::new_circle;
```

Другие фигуры конструируются точно так же:

```
class Io_triangle : public Triangle, public Io_obj {   // Triangle — треугольник
    //...
};
```

Если построение «подмостков» для объектного ввода/вывода становится утомительным, нам поможет шаблон:

```
template<class T> class Io : public T, public Io_obj {
public:
    Io* clone () const
        { return new Io (*this); }        // замещение Io_obj::clone()
    Io (istream&);                        // инициализация из потока ввода
    static Io* new_io (istream& s) { return new Io (s); }
    //...
};
```

Теперь мы можем определить ***Io_circle***:

```
typedef Io<Circle> Io_circle;
```

Впрочем, нам по-прежнему нужно явно определить ***Io<Circle>::Io*** (*istream*&), так как этой функции нужно знать подробности о ***Circle***.

Шаблон ***Io*** является примером того, как вписать конкретные типы в иерархию классов, введя вспомогательный класс, которой являлся бы узлом в данной иерархии. Этот класс наследует от своего параметра-шаблона, чтобы позволить приведение из типа ***Io_obj***. К сожалению, это приводит к тому, что ***Io*** нельзя использовать со встроенными типами:

```
typedef Io<Date> Io_date;     // оболочка для конкретного типа
typedef Io<int> Io_int;       // ошибка: нельзя создать производный класс
                              // от встроенного типа
```

С этой проблемой можно справиться, введя отдельный шаблон для встроенных типов, или воспользовавшись классом, представляющим встроенный тип (§ 25.10[1]).

Эта простая система объектного ввода/вывода не может делать все на свете, но она почти умещается на одной странице, а ключевые механизмы, использовавшиеся в ней, имеют множество применений. Вообще говоря, эти приемы можно использовать для вызова функций, основываясь на строке, предоставляемой пользователем, и для манипулирования объектами неизвестного типа посредством интерфейсов, установленных при помощи идентификации типа во время выполнения.

25.5. Действия

В C++ самый простой и очевидный способ охарактеризовать действие — написать функцию. Однако если действие нужно отложить, передать до выполнения «куда-нибудь», если оно требует собственных данных, должно комбинироваться с другими действиями (§ 25.10 [18, 10]) и т. п., то часто хочется оформить его в виде класса, который мог бы выполнить желаемое действие, а также предоставить другие услуги. Очевидным примером такого действия является объект-функция, использующийся со стандартными алгоритмами (§ 18.4), а также манипуляторы, применяемые с потоками ***iostream*** (§ 21.4.6). В первом случае собственно действие выполняется применяющимся оператором, а во втором — операторами << и >>. В случае с ***Form*** (§ 21.4.6.3) и ***Matrix*** (§ 22.4.7) классы-композиторы используются для откладывания действия, пока не будет собрано достаточно информации для его эффективного выполнения.

Обычная форма класса-действия — это просто класс, содержащий только одну виртуальную функцию (обычно она называется как-нибудь вроде «do_it» — «сделай это»):

```
class Action {                          // действие
public:
        virtual int do_it (int) = 0;
        virtual ~Action () { }
};
```

С таким классом мы можем написать код — скажем, меню — который сможет хранить действия для последующего выполнения, не используя указатели на функции, не имея информации о вызываемых объектах и даже не зная имен вызываемых операций. Например:

```
class Write_file : public Action {          // записать в файл
        File& f;
public:
        int do_it (int) { return f.write ().succeed (); }
};
class Error_response : public Action {      // ответ на ошибку
        string message;                     // сообщение
public:
        Error_response (const string &s) : message (s) {}
        int do_it (int);
};
int Error_response::do_it (int)
{
        Response_box db (message.c_str (), "продолжить", "отменить", "повторить");
        switch (db.get_response ()) {
        case 0:
                return 0;
        case 1:
                abort ();
        case 2:
                current_operation.redo ();
                return 1;
        }
}
Action* actions[] = {                       // действия
        new Write_file (f),
        new Error_response ("вам снова не повезло"),
        // ...
};
```

Пользователь класса ***Action*** может совершенно ничего не знать о производных классах, таких как ***Write_file*** и ***Error_response.***

Это — мощная техника, и людям, привыкшим к функциональной декомпозиции, следует применять ее с известной осторожностью. Если слишком много классов ста-

новятся похожими на ***Action***, может оказаться, что весь проект системы ухудшился, превратившись в нечто излишне функциональное.

И, наконец, класс может кодировать операцию для выполнения на удаленной машине или для хранения, чтобы выполнить ее в будущем.

25.6. Интерфейсные классы

Одна из самых важных разновидностей классов — это скромные интерфейсные классы, на которые, как правило, смотрят свысока. Интерфейсный класс мало что делает — если бы делал, то не был бы интерфейсным классом. Он просто приспосабливает внешнее представление некоторых услуг к местным потребностям. Поскольку в принципе невозможно все время одинаково хорошо удовлетворять все потребности, интерфейсные классы очень важны для того, чтобы позволить совместное использование без необходимости надевать на всех пользователей одну смирительную рубашку.

Совершенно чистая форма интерфейса даже не вызывает генерирования кода. Рассмотрим специализацию контейнера ***Vector*** из § 13.5:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
        typedef Vector Base;

        Vector() { }
        Vector(int i) : Base(i) { }

        T*& operator[](int i)
                        { return reinterpret_cast<T*&>(Base::operator[](i)); }
        // ...
};
```

Эта (частичная) специализация превращает небезопасный ***Vector<void>*** в более полезное семейство векторных классов, безопасных с точки зрения типов. Часто для того, чтобы сделать интерфейсные классы приемлемыми, необходимо прибегать к встроенным функциям. В случаях вроде приведенного выше, когда встроенные перенаправляющие функции только подгоняют тип, не происходит никаких лишних расходов времени или памяти.

Естественно, абстрактный базовый класс, представляющий абстрактный тип, реализованный конкретными типами (§ 25.2), — это форма интерфейсного класса, также как и промежуточные классы из § 25.7. Однако здесь мы сфокусируемся на тех классах, которые не имеют никаких других специфических функций кроме приспособления интерфейса.

Рассмотрим проблему слияния двух иерархий, с использованием множественного наследования. Что можно сделать, если имеет место конфликт имен, то есть в двух классах виртуальные функции, выполняющие совершенно разные операции, имеют одинаковые имена? Например, рассмотрим видеоигру «Дикий Запад», где взаимодействие с пользователем осуществляется через общий оконный класс:

```
class Window {
        // ...
        virtual void draw();           // вывести образ
};
```

```
class Cowboy {
        // ...
        virtual void draw ();                    // вынуть револьвер из кобуры
};

class Cowboy_window : public Cowboy, public Window {
        // ...
};
```

Cowboy_window представляет анимацию ковбоя в игре, и через этот класс игрок осуществляет управление ковбоем. Мы бы предпочли множественное наследование, чем объявлять ***Window*** или ***Cowboy*** членом, поскольку есть много служебных функций, определенных и для ***Window***, и для ***Cowboy***. Мы бы хотели передавать ***Cowboy_window*** в такие функции без специальных действий со стороны программиста. Однако это ведет к проблеме определения версий ***Cowboy::draw*** () и ***Window::draw*** ().

В ***Cowboy_window*** может быть только одна функция с именем ***draw*** (). Поскольку служебные функции манипулируют с классами ***Window*** и ***Cowboy***, ничего не зная о классе ***Cowboy_window***, последний должен заместить ***draw*** () и для класса ***Cowboy***, и для класса ***window***. Но замещение обеих функций одной функцией ***draw*** () будет неправильным — несмотря на общее имя, функции ***draw*** () не связаны между собой и не могут быть замещены одной общей функцией.

Кроме того, нам бы хотелось, чтобы класс ***Cowboy_window*** имел разные, однозначные имена для унаследованных функций ***Cowboy::draw*** () и ***Window::draw*** ().

Чтобы решить эту проблему, нам нужно ввести по дополнительному классу для ***Cowboy*** и для ***Window***. Эти классы вводят два новых имени для функций ***draw*** () и гарантируют, что вызов функций ***draw*** () в классах ***Cowboy*** и ***Window*** вызовет функции с новыми именами:

```
class CCowboy : public Cowboy {          // интерфейс к Cowboy с переименованной draw()
public:
        virtual int cow_draw () = 0;
        void draw () { cow_draw (); }    // замещение Cowboy::draw
};

class WWindow : public Window {          // интерфейс к Window с переименованной draw()
public:
        virtual int win_draw () = 0;
        void draw () { win_draw (); }    // замещение Window::draw
};
```

Теперь мы можем составить ***Cowboy_window*** из интерфейсных классов ***CCowboy*** и ***WWindow*** и заместить ***cow_draw*** () и ***win_draw*** (), чтобы добиться желаемого результата:

```
class Cowboy_window : public CCowboy, public WWindow {
        // ...
        void cow_draw ();
        void win_draw ();
};
```

Отметим, что эта проблема была серьезной только потому, что две функции ***draw*** () имели одинаковый тип аргумента. Если бы типы аргументов различались, обычные

правила перегрузки гарантировали бы, что проблем не возникнет, несмотря на то, что несвязанные между собой функции имеют одно и то же имя.

Для каждого использования интерфейсного класса можно представить себе специализированное расширение языка, которое выполняло бы желаемое приспособление чуть более эффективно или изящно. Однако интерфейсные классы применяются нечасто, и поддержка их специальными языковыми конструкциями вызвала бы чрезмерное усложнение. В частности, конфликты имен, возникающие из-за слияния иерархий классов, нельзя назвать обычным явлением (по сравнению с тем, насколько часто программист пишет классы), и они, как правило, появляются при объединении иерархий, порожденных в разных культурах программирования — таких как игры и операционные системы. Слияние таких непохожих иерархий — непростая задача, и разрешение конфликтов имен чаще всего будет не самой сложной из встающих перед программистом проблем; например, таких, как несхожесть стратегий обработки ошибок, инициализации и управления памятью. Разрешение конфликтов имен обсуждается здесь лишь потому, что прием с введением интерфейсных классов, имеющих функцию, переадресующую вызов другой функции, имеет множество других применений. Его можно использовать не только для замены имен, но также для изменения аргументов и типов возвращаемых значений, для введения проверки во время выполнения и т. п.

Поскольку функции ***CCowboy::draw*** () и ***WWindow::draw*** () переадресуют свой вызов виртуальным функциям, их нельзя оптимизировать простым встраиванием. Однако компилятор может распознать в них просто переадресующие функции и оптимизировать, убрав их из цепочки вызовов.

25.6.1. Приспосабливающие интерфейсы

Главное применение интерфейсных функций — приспосабливание интерфейса, чтобы он лучше соответствовал ожиданиям пользователей, путем перемещения в интерфейс кода, который иначе был бы разбросан по всей пользовательской программе. Например, отсчет индексов в стандартном векторе ***vector*** идет от нуля. Пользователи, желающие, чтобы их вектора индексировались не от ***0*** до ***size–1***, должны приспособить вектора для своих целей. Например:

```
void f()
{
    vector v<int> (10);        // диапазон [0:9]
    // притворимся, что v имеет диапазон [1:10]
    for (int i=1; i<=10; i++) {
        v[i–1] = 7;            // не забудьте подправить индекс
        //...
    }
}
```

Лучшим решением является введение вектора с произвольными границами:

```
class Vector : public vector<int> {
    int lb;
public:
    Vector (int low, int high) : vector<int> (high–low+1) { lb=low; }
```

```
        int& operator[] (int i) { return vector<int>::operator[] (i-lb); }

        int low () { return lb; }
        int high () { return lb+size ()-1; }
};
```

Vector можно использовать следующим образом:

```
void g ()
{
        Vector v (1, 10);          // диапазон [1:10]

        for (int i=1; i<10; i++) {
                v[i]=7;
                // ...
        }
}
```

Это не вызывает никаких затрат по сравнению с предыдущим примером. Ясно, что версия ***Vector*** легче читается и пишется и менее подвержена ошибкам.

Интерфейсные классы обычно невелики и (по определению) мало что делают. Однако они становятся необходимы, когда программные продукты, написанные в разных традициях, должны объединиться, поскольку тогда требуется посредник между разными стилями написания кода. Например, интерфейсные классы часто используются для того, чтобы обеспечить интерфейс C++ к программам на других языках, а также чтобы изолировать прикладной код от подробностей библиотеки (оставив открытой возможность замены одной библиотеки другой).

Другое важное применение интерфейсных классов — предоставление проверяемых и ограниченных интерфейсов. Например, довольно часто предполагается, что целые переменные имеют значения в определенных пределах. Это можно гарантировать (во время выполнения программы) простым шаблоном:

```
template<int low, int high> class Range {
        int val;
public:
        class Error { };                    // класс исключений

        Range (int i) { Assert<Error> (low<=i&&i<high); val = i; }      // см. § 24.3.7.2

        Range operator= (int i) { return *this=Range (i); }

        operator int () { return val; }
        // ...
};

void f (Range<2, 17>);
void g (Range<-10, 10>);

void h (int x)
{
        Range<0, 2001> i = x;          // может сгенерировать Range::Error
        int i1 = i;

        f (3);
        f (17);                        // генерирует Range::Error
```

```
    g(-7);
    g(100);                    // генерирует Range::Error
}
```

Шаблон ***Range*** легко расширяется для работы с диапазонами произвольных скалярных типов (§ 25.10[7]).

Интерфейсный класс, который контролирует доступ к другому классу или приспосабливает его интерфейс, иногда называют *оболочкой*.

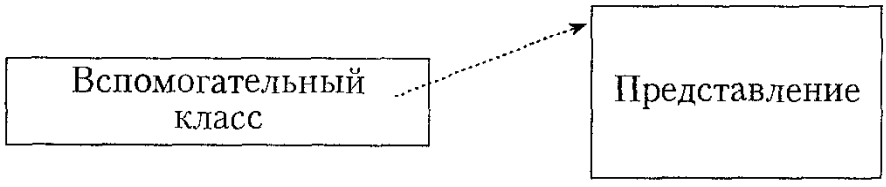
25.7. Вспомогательные классы

Абстрактный тип обеспечивает эффективное разделение интерфейса и его реализаций. Однако, если следовать § 25.3, получается неразрывная связь между интерфейсом, предоставляемым абстрактным типом, и его реализацией, обеспечиваемой конкретным типом. Например, невозможно «отвязать» абстрактный итератор от одного источника — скажем, множества, и привязать к другому — например, потоку, когда первоначальный источник исчерпан.

Более того, если не манипулировать объектом, релизующим абстрактный класс, через указатели или ссылки, выгоды от виртуальных функций теряются. Пользовательская программа может стать зависимой от деталей классов реализации, поскольку абстрактный тип (в том числе, если это объект, принятый в качестве аргумента, переданного по значению), если его размеры неизвестны, нельзя расположить статически в памяти или в стеке. Использование указателей и ссылок подразумевает, что бремя распределения памяти ложится на пользователя.

Другое ограничение подхода с применением абстрактных классов заключается в том, что объект класса имеет фиксированные размеры. Однако классы используются для представления понятий, требующих различное количество памяти для своей реализации.

Распространенным методом решения этих трудностей является разделение единого объекта на два: вспомогательного, обеспечивающего пользовательский интерфейс, и представления, содержащего все или большую часть состояния объекта. В качестве связи между вспомогательным классом и представлением, как правило, выступает указатель во вспомогательном классе. Часто вспомогательные классы имеют чуть больше данных, чем просто указатель на представление, но не намного. Это подразумевает, что расположение в памяти объектов вспомогательного класса, как правило, неизменно, даже когда представление изменяется, и что вспомогательные классы достаточно малы для свободной пересылки между функциями, так что пользователю не нужно применять указатели и ссылки.



Простым примером вспомогательного класса является ***String*** из § 11.12. Вспомогательный класс обеспечивает интерфейс, контроль за доступом и управление памятью для представления. В данном случае и вспомогательный класс, и представление являются конкретными типами, хотя класс представления часто бывает абстрактным.

Рассмотрим абстрактный тип множеств *Set* из § 25.3. Как ввести для него вспомогательный класс, какие выгоды это принесет, и какой ценой? Для класса множеств вспомогательный класс можно определить, просто перегрузив оператор ->:

```
template<class T> class Set_handle {
        Set<T>* rep;
public:
        Set<T>* operator-> () { return rep; }
        Set_handle (Set<T>* pp) : rep (pp) { }
};
```

Это не очень существенно влияет на то, как используются множества; просто повсюду вместо *Set*& или *Set** передаются *Set_handle*. Например:

```
void f (Set_handle<int> s)
{
        for (int* p = s->first (); p; p = s->next ())
        {
                //...
        }
}

void user ()
{
        Set_handle<int>sl (new List_set<int>);
        Set_handle<int> v (new Vector_set<int> (100));

        f (sl);
        f (v);
}
```

Часто нам нужно, чтобы вспомогательный класс не только обеспечивал доступ. Например, если классы *Set* и *Set_handle* проектируются вместе, легко ввести счетчик ссылок, включив соответствующие счетчики в каждый класс множеств. Вообще говоря, нам не нужно проектировать вспомогательный класс вместе с тем, чему он будет «помогать», поэтому любую информацию, которая будет использоваться совместно со вспомогательным классом, нам придется сохранять в отдельном объекте. Иными словами, кроме интрузивных вспомогательных классов нам бы хотелось иметь и неинтрузивные. Например, вот вспомогательный класс, который удаляет объект, когда исчезает его последний вспомогательный объект:

```
template<class X> class Handle {
        X* rep;
        int* pcount;
public:
        X* operator-> () { return rep; }

        Handle (X* pp) : rep (pp), pcount (new int (1)) { }
        Handle (const Handle& r) : rep (r.rep), pcount (r.pcount) { (*pcount)++; }

        Handle& operator= (const Handle& r)
        {
                if (rep == r.rep) return *this;
                if (-- (*pcount) == 0) {
```

```
                delete rep;
                delete pcount;
            }
            rep = r.rep;
            pcount = r.pcount;
            (*pcount)++;
            return *this;
        }

        ~Handle () { if (-- (*pcount) == 0) { delete pcount; } }

        // ...
};
```

Такие вспомогательные объекты можно свободно передавать.

```
void f1 (Handle<Set>);
Handle<Set> f2 ()
{
        Handle<Set> h (new List_set<int>);
        // ...
        return h;
}

void g ()
{
        Handle<Set> hh=f2 ();
        f1 (hh);
        // ...
}
```

Здесь множество, созданное в *f2* (), будет удалено на выходе из *g* () (если только *f1* () его не скопирует); программисту нет нужды заботиться об этом.

Естественно, за это удобство приходится платить, но для многих прикладных программ цена хранения и обслуживания счетчика использований приемлема.

Иногда указатель на представление полезно извлечь из вспомогательного класса и воспользоваться им напрямую — например, если нужно передать объект в функцию, которая не знает о вспомогательном классе. Это хорошо работает, если вызываемая функция не разрушает переданный ей объект или сохраняет указатель на него для использования после возврата в вызвавшую функцию. Также может пригодиться операция привязывания вспомогательного класса к новому представлению:

```
template<class X> class Handle {
        // ...

        X* get_rep () { return rep; }

        void bind (X* pp)
        {
                if (pp != rep) {
                        if (-- (*pcount) == 0) {
                                delete rep;
                                *pcount = 1;            // обновить pcount
                        }
```

```
            else
                pcount = new int (1);          // новый pcount
            rep = pp;
        }
    }
};
```

Отметим, что построение производных классов от вспомогательного класса ***Handle*** не особенно полезно. Это конкретный тип без виртуальных функций. Идея заключается в том, чтобы иметь один вспомогательный класс для семейства классов, определяемых базовым классом. А вот создание производных классов от этого базового класса может оказаться мощным приемом. Он применим и к узловым классам, и к абстрактным типам.

Как уже сказано, вспомогательный класс ***Handle*** не связан с наследованием. Чтобы получить класс, действующий как истинный указатель, считающий число использований, ***Handle*** нужно скомбинировать с ***Ptr*** из § 13.6.3.1 (см. § 25.10[2]).

Вспомогательный класс, обеспечивающий интерфейс, почти идентичный классу, для которого он является «вспомогательным», часто называют *заместителем* (proxy). Такой прием часто используется для классов, ссылающихся на объекты на удаленной машине.

25.7.1. Операции во вспомогательных классах

Перегрузка оператора –>дает возможность вспомогательному классу получать управление и выполнять некоторую работу при каждом обращении к объекту. Например, можно собрать статистику о числе использований объекта через вспомогательный класс:

```
template<class T> class Xhandle {
    T* rep;
    int no_of_accesses;                    // число обращений
public:
    T* operator–> () { no_of_accesses++; return rep; }
    //...
};
```

Вспомогательные классы, для которых требуется выполнение работы и *до*, и *после* обращения, требуют более изощренного программирования. Например, может понадобиться множество с блокировкой во время вставки или удаления элемента. Здесь очень важно, чтобы интерфейс класса-представления был повторен во вспомогательном классе:

```
template<class T> class Set_controller {
    Set<T>* rep;
    Lock lock;
    //...
public:
    void insert (T* p) { Lock_ptr x (lock); rep–>insert (p); }      // см. § 14.4.1
    void remove (T* p) { Lock_ptr x (lock); rep–>remove (p); }

    int is_member (T* p) { return rep–>is_member (p); }

    T* get_first () { T* p = rep–>first (); return p? *p : T (); }
    T* get_next () { T* p = rep–>next (); return p? *p : T (); }
```

```
        T first () { Lock_ptr x (lock); T tmp = *rep->first (); return tmp; }
        T next () { Lock_ptr x (lock); T tmp = *rep->next (); return tmp; }

        // ...
};
```

Введение этих переадресующих функций утомительно (и поэтому здесь, в известном смысле, кроется источник ошибок), хотя оно нетрудно и не дорого с точки зрения выполнения программы.

Отметим, что только некоторые из функций *set* требуют блокировки. По своему опыту я могу сказать, что, как правило, класс, нуждающийся в выполнении пред-действий и пост-действий, требует этого лишь для нескольких функций-членов. В случае с блокировкой, блокировка всех операций — как это делается для мониторов в некоторых системах — ведет к излишним блокировкам и иногда приводит к значительной потере параллельности.

Одно из преимуществ тщательного определения всех операций вспомогательного класса через перегрузку оператора -> заключается в том, что можно создавать классы, производные от *Set_controller*. К сожалению, некоторые выгоды вспомогательных классов становятся сомнительными, если в производный класс добавляются члены данных. В частности, объем совместно используемого кода (во «вспомогаемых» классах) уменьшается по сравнению с объемом кода в каждом вспомогательном классе.

25.8. Прикладные среды разработки

Компоненты, построенные из классов разных видов, описанных в §25.2–§25.7, поддерживают проектирование и повторное использование кода, предоставляя и строительные блоки, и способы их соединения; создатель прикладной программы проектирует единую среду разработки, в которую включаются эти общие строительные блоки. Альтернативный, и иногда более амбициозный подход к поддержке проектирования и повторного использования — сначала предоставить общую среду разработки, в которую бы создатели прикладных программ вставляли специфичные для приложения фрагменты кода в качестве строительных блоков. Такой подход часто называют *прикладной средой разработки*. Классы, образующие подобную среду, зачастую имеют такой жирный интерфейс, что вряд ли являются классами в обычном понимании. Они приближаются к идеалу завершенных прикладных программ, если не считать того, что они ничего не делают. Конкретные действия задаются прикладным программистом.

В качестве примера рассмотрим фильтр, то есть программу, которая считывает поток ввода, (возможно) выполняет какие-то действия, основанные на вводе, (возможно) выдает поток вывода и (возможно) представляет окончательный результат. Бесхитростная среда разработки для таких программ обеспечивает набор операций, которыми бы мог воспользоваться прикладной программист:

```
class Filter {
public:
        class Retry {
        public:
                virtual const char* message () { return 0; }
        };
```

```
        virtual void start () { }
        virtual int read () = 0;
        virtual void write () { }
        virtual void compute { }
        virtual int result () = 0;

        virtual int retry (Retry& m) { cerr << m.message () << '\n'; return 2; }

        virtual ~Filter () { }
};
```

Функции, которые будут использоваться в производном классе, объявлены как чисто виртуальные; другие просто определены так, что ничего не делают.

Среда разработки также обеспечивает главный цикл и рудиментарный механизм обработки ошибок:

```
int main_loop (Filter* p)              // главный цикл
{
    for (;;) {
        try {
            p->start ();
            while (p->read ()) {
                p->compute ();
                p->write ();
            }
            return p->result ();
        }

        catch (Filter::Retry& m) {
            if (int i = p->retry (m)) return i;
        }

        catch (...) {
            cerr << "Фатальная ошибка в фильтре\n";
            return 1;
        }
    }
}
```

И наконец, я мог бы написать свою программу следующим образом:

```
class My_filter : public Filter {
        istream& is;
        ostream& os;
        int nchar;
public:
        int read () { char c; is.get (c); return is.good (); }
        void compute () { nchar++; };
        int result () { os << nchar << "символов считано\n"; return 0; }

        My_filter (istream& ii, ostream& oo) : is (ii), os (oo), nchar (0) { }
};
```

и активизировать ее таким образом:

```
int main ()
{
        My_filterf (cin, cout);
        return main_loop (&f);
}
```

Естественно, чтобы быть полезной, среда разработки должна обеспечивать больше структур и служебных функций, чем этот простой пример. В частности, среда разработки, как правило, является иерархией узловых классов. Благодаря тому, что прикладному программисту остается только ввести классы-листья, «растущие» глубоко в иерархии, достигается общность между прикладными программами и повторное использование служебных функций. Среда разработки также будет поддерживаться библиотекой, которая предоставляет классы, нужные прикладному программисту при описании конкретных, «действующих» классов.

25.9. Советы

[1] Решение, каким образом класс должен использоваться, должно быть сознательным (осознанным и проектировщиком, и пользователем); § 25.1.

[2] Учитывайте альтернативы, связанные с разными видами классов; § 25.1.

[3] Для представления простых независимых понятий пользуйтесь конкретными типами; § 25.2.

[4] Для представления понятий, где важна эффективность, близкая к оптимальной, пользуйтесь конкретными типами; § 25.2.

[5] Не стройте от конкретных классов производные; § 25.2.

[6] Для представления интерфейсов в тех случаях, когда представления объектов могут меняться, пользуйтесь абстрактными классами; § 25.3.

[7] Там, где должны сосуществовать разные представления объектов, для представления интерфейсов пользуйтесь абстрактными классами; § 25.3.

[8] Для представления новых интерфейсов к существующим типам пользуйтесь абстрактными классами; § 25.3.

[9] Там, где схожие понятия имеют существенные общие детали реализации, применяйте узловые классы; § 25.4.

[10] Используйте узловые классы для постепенного расширения реализации; § 25.4.

[11] Чтобы ввести для объекта новый интерфейс, пользуйтесь определением типа во время выполнения; § 25.4.1.

[12] Для представления действий с ассоциированным с ними состоянием пользуйтесь классами; § 25.5.

[13] Для представления действий, которые нужно сохранять, передавать или откладывать пользуйтесь, классами; § 25.5.

[14] Для приспособления классов к новому виду использования (без изменения самого класса) применяйте интерфейсные классы; § 25.6.

[15] Пользуйтесь интерфейсными классами, чтобы ввести проверку; § 25.6.1.

[16] Чтобы избежать прямого использования указателей и ссылок, применяйте вспомогательные классы; § 25.7.

[17] Чтобы управлять общими представлениями, пользуйтесь вспомогательными классами; § 25.7.

[18] В тех прикладных областях, где управляющую структуру можно предопределить заранее, пользуйтесь прикладными средами разработки; § 25.8.

25.10. Упражнения

1. (*1) Шаблон ***Io*** из § 25.4.1 не работает для встроенных типов. Измените его так, чтобы работал.
2. (*1.5) Шаблон вспомогательного класса ***Handle*** из § 25.7 не отражает отношений наследования классов, которым он «помогает». Измените его так, чтобы отражал. То есть вы должны сделать так, чтобы можно было присваивать объект ***Handle<Circle>*** объекту ***Handle<Shape>***, но не наоборот.
3. (*2.5) Имея класс ***String***, определите другой строковый класс, использующий его как представление и предоставляющий свои операции как виртуальные функции. Сравните быстродействие двух классов. Попытайтесь найти осмысленный класс, который бы лучше всего реализовывался открытым наследованием из класса строк с виртуальными функциями.
4. (*4) Изучите две широко используемые библиотеки. Классифицируйте библиотечные классы в терминах конкретных и абстрактных типов, узловых, вспомогательных и интерфейсных классов. Используются ли абстрактные узловые и конкретные узловые классы? Есть ли для классов этих библиотек более подходящая классификация? Используются ли жирные интерфейсы? Какие возможности в них предоставляются (если вообще предоставляются) для получения информации о типе во время выполнения? Какова стратегия распределения памяти?
5. (*2) Воспользуйтесь средой разработки ***Filter*** для реализации программы, которая удаляет повторяющиеся соседние слова из потока ввода, а в остальном копирует ввод на вывод.
6. (*2) Воспользуйтесь средой разработки ***Filter*** для реализации программы, которая подсчитывает частоту слов в потоке ввода и выдает на выход список пар (слово, счетчик) в порядке возрастания частот.
7. (*1.5) Напишите шаблон ***Range***, принимающий в качестве параметров шаблона и диапазон, и тип элементов.
8. (*1) Напишите шаблон ***Range***, принимающий диапазон в качестве аргумента конструктора.
9. (*9) Напишите простой строковый класс, не выполняющий проверки ошибок. Напишите другой класс, проверяющий доступ к первому. Рассмотрите все «за» и «против» разделения базовых функций и проверки ошибок.
10. (*2.5) Реализуйте систему объектного ввода/вывода из § 25.4.1 для нескольких типов, включая по крайней мере целые числа, строки и какую-нибудь иерархию классов по вашему выбору.
11. (*2.5) Определите класс ***Storable***[1] как абстрактный базовый класс с виртуальными функциями ***write_out*** () (записать) и ***read_in*** () (прочесть). Для простоты допустите, что символьной строки достаточно для описания места, где перманентно хранится объект. Воспользуйтесь классом ***Storable***, чтобы обеспечить возмож-

[1] Название подразумевает, что объекты этого класса умеют записывать себя в некую долговременную память (скажем, на диск) и считывать себя оттуда. — *Примеч. ред.*

ность записывать на диск объекты классов, производных от ***Storable***, и считывать такие объекты с диска. Проверьте это на паре классов по вашему выбору.

12. (*4) Определите базовый класс ***Persistent***[1] с операциями ***save*** () и ***no_save*** (), которые контролировали бы, записывается ли объект деструктором в долговременную память или нет. Кроме ***save*** () и ***no_save*** (), какие еще операции можно с пользой для дела ввести в ***Persistent***? Проверьте класс ***Persistent*** на паре классов по вашему выбору. Является ли он узловым классом, конкретным типом или абстрактным типом? Почему?
13. (*3) Напишите класс ***Stack*** (стек), реализацию которого можно изменять во время выполнения программы. Подсказка: «Всякая проблема решается еще одним перенаправлением».
14. (*3.5) Определите класс ***Oper***, который содержал бы идентификатор типа ***Id*** (это может быть ***string*** или С-строка) и некую операцию (указатель на функцию или некоторый объект-функцию). Определите класс ***Cat_object***, содержащий список операций ***Oper*** и указателей ***void****. Введите в ***Cat_object*** функцию ***add_oper*** (***Oper***), добавляющую в список операцию ***Oper***; ***remove_oper*** (***Id***), удаляющую из списка операцию, идентифицируемую строкой ***Id***; и ***operator*** () (***Id, arg***), вызывающий операцию ***Oper*** по ее идентификатору ***Id***. Реализуйте стек объектов ***Cat*** при помощи ***Cat_object***. Напишите маленькую программку, чтобы проверить эти классы.
15. (*3) Определите шаблон ***Object*** на основе класса ***Cat_object***. Воспользуйтесь ***Object*** для реализации стека строк ***String***. Напишите маленькую программку, чтобы проверить этот шаблон.
16. (*2.5) Определите вариант класса ***Object***, называемый ***Class***, который гарантировал бы, что объекты с одинаковыми операциями имеют один и тот же список операций. Напишите маленькую программку, чтобы проверить этот шаблон.
17. (*2) Определите шаблон ***Stack***, который бы предоставлял удобный и безопасный с точки зрения типов интерфейс к стеку, реализованному шаблоном ***Object***. Сравните этот стек со стековыми классами из предыдущих упражнений. Напишите маленькую программку, чтобы проверить этот шаблон.
18. (*3) Напишите класс для представления операций, которые должны передаваться на выполнение другому компьютеру. Проверьте его либо путем действительной пересылки команд другой машине, либо предварительной записью команд в файл с их последующим чтением и выполнением.
19. (*2) Напишите класс для составления операций, представленных в виде объектов-функций. При наличии двух объектов-функций f и g ***Compose*** (f, g) должен произвести объект, который можно вызвать с аргументом x, допустимым для g, и возвращающий $f(g(x))$, если возвращаемое функцией g () значение является допустимым аргументом для f ().

[1] Название подразумевает, что объекты этого класса будут «долгоживущими», то есть они (в частности) могут существовать после завершения выполнения программы. Близко к понятию «хранимый» (storable). — *Примеч. ред.*

Приложения и предметный указатель

В приложениях описана грамматика C++, обсуждается совместимость между C++ и C, между стандартом C++ и предыдущими версиями C++, разного рода технические подробности языка, особенности локализации, а также вопросы безопасности при генерации исключений. Предметный указатель весьма обширен и является неотъемлемой частью данной книги.

Приложение A

Грамматика

Для учителя нет большей опасности,
чем учить словам, а не вещам.
— Марк Блок

Введение — ключевые слова — лексические соглашения — программы — выражения — инструкции — объявления — объявители-классы — производные классы — особые функции-члены — перегрузка — шаблоны — обработка исключений — директивы препроцессора.

A.1. Введение

Приведенное ниже описание синтаксиса C++ должно помочь пониманию языка. Оно не является точным определением языка C++. В частности, описанная здесь грамматика допускает надмножество допустимых конструкций C++. Чтобы отличать выражения от объявлений, нужно применять правила устранения неоднозначностей (§ A.5, § A.7). Кроме того, для «прополки» синтаксически правильных, но бессмысленных конструкций нужно использовать правила контроля доступа, разрешения неоднозначности и типов.

Стандартные грамматики C и C++ выражают самые малейшие различия синтаксически, а не через ограничения. Это способствует точности, но не всегда улучшает читаемость.

A.2. Ключевые слова

Новые зависящие от контекста ключевые слова вводятся в программу при помощи объявлений ***typedef*** (§ 4.9.9), пространств имен (§ 8.2), классов (глава 10), перечислений (§ 4.8) и шаблонов ***template*** (глава 13).

имя-typedef:
    идентификатор

имя-пространства-имен:
    оригинальное-имя-пространства-имен
    псевдоним-пространства-имен

оригинальное-имя-пространства-имен:
    идентификатор

псевдоним-пространства-имен:
    идентификатор

имя-класса:
    идентификатор
    идентификатор-шаблона

имя-перечисления:
идентификатор

имя-шаблона:
идентификатор

Отметим, что название класса через *имя-typedef* — это тоже *имя-класса*.

Если идентификатор явно не объявлен, как обозначающий имя типа, считается, что он обозначает нечто, не являющееся типом (см. § В.13.5).

Ключевые слова C++ таковы:

Ключевые слова C++

and	*continue*	*goto*	*public*	*try*
and_eq	*default*	*if*	*register*	*typedef*
asm	*delete*	*inline*	*reinterpret_cast*	*typeid*
auto	*do*	*int*	*return*	*typename*
bitand	*double*	*long*	*short*	*union*
bitor	*dynamic_cast*	*mutable*	*signed*	*unsigned*
bool	*else*	*namespace*	*sizeof*	*using*
break	*enum*	*new*	*static*	*virtual*
case	*explicit*	*not*	*static_cast*	*void*
catch	*export*	*not_eq*	*struct*	*volatile*
char	*extern*	*operator*	*switch*	*wchar_t*
class	*false*	*or*	*template*	*while*
compl	*float*	*or_eq*	*this*	*xor*
const	*for*	*private*	*throw*	*xor_eq*
const_cast	*friend*	*protected*	*true*	

А.3. Лексические соглашения

Стандартные грамматики C и C++ представляют лексические соглашения как грамматические продукции. Это повышает точность, но приводит к более громоздким грамматикам и не всегда улучшает читаемость:

шестнадцатеричная-четверка:
шестнадцатеричная-цифра шестнадцатеричная-цифра
шестнадцатеричная-цифра шестнадцатеричная-цифра[1]

универсальное-символьное-имя:
\u *шестнадцатеричная-четверка*
\U *шестнадцатеричная-четверка шестнадцатеричная-четверка*

токен-препроцессирования:
имя-заголовочного-файла
идентификатор
pp-число
символьный-литерал
строковый-литерал
оператор-препроцессирования-или-знак-препинания
всякий символ-не-разделитель, не относящийся к указанным выше

[1] К сожалению, ограниченная ширина страницы книги не позволяет уместить некоторые грамматические конструкции C++ в одной строке. В таких случаях они переносятся на новую строку с дополнительным отступом слева. — *Примеч. ред.*

токен:
идентификатор
ключевое-слово
литерал
оператор
знак-препинания

имя-заголовочного-файла:
<h-символьная-последовательность>
"q-символьная-последовательность"

h-символьная-последовательность:
h-символ
h-символьная-последовательность h-символ

h-символ:
любой элемент исходного набора символов кроме перевода-строки и >

q-символьная-последовательность:
q-символ
q-символьная-последовательность q-символ

q-символ:
любой элемент исходного набора символов кроме-перевода строки и "

pp-число:
цифра
. цифра
pp-число цифра
pp-число не-цифра
pp-число e *знак*
pp-число E *знак*
pp-число .

идентификатор:
не-цифра
идентификатор не-цифра
идентификатор цифра

не-цифра: одно из
универсальное-символьное-имя

_ a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

цифра: одно из

0 1 2 3 4 5 6 7 8 9

оператор-препроцессирования-или-знак-препинания: одно из

{	}	[	]	#	##	(	)	<:	:>	<%	%>	%:%:
%:	;	:	?	::	.	.*	+	–	*	/	%	^
&	|	~	!	=	<	>	+=	–=	*=	/=	%=	^=
&=	|=	<<=	>>=	<<	>>	==	!=	<=	>=	&&	||	++
––	,	–>	–>*	...	new	delete		and	and_eq		bitand	
bitor		compl		not	or	not_eq		xor	or_eq		xor_eq	

литерал:
целый-литерал
символьный-литерал
литерал-с-плавающей-точкой
строковый-литерал
логический-литерал

целый-литерал:
десятичный-литерал целый-суффикс$_{opt}$[1]
восьмеричный-литерал целый-суффикс$_{opt}$
шестнадцатеричный-литерал целый-суффикс$_{opt}$

десятичный-литерал:
ненулевая-цифра
десятичный-литерал-цифра

восьмеричный-литерал:
0
восьмеричный-литерал восьмеричная-цифра

шестнадцатеричный-литерал:
0x *шестнадцатеричная-цифра*
0X *шестнадцатеричная-цифра*
шестнадцатеричный-литерал шестнадцатеричная-цифра

ненулевая-цифра: одно из
1 2 3 4 5 6 7 8 9

восьмеричная-цифра: одно из
0 1 2 3 4 5 6 7

шестнадцатеричная-цифра: одно из
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

целочисленный-суффикс:
беззнаковый-суффикс длинный-суффикс$_{opt}$
длинный-суффикс беззнаковый-суффикс$_{opt}$

беззнаковый-суффикс: одно из
u U

длинный-суффикс: одно из
l L

символьный-литерал:
'с-символьная-последовательность'
L*'с-символьная-последовательность'*

с-символьная-последовательность:
с-символ
с-символьная-последовательность с-символ

с-символ:
любой элемент исходного набора символов кроме апострофа ('), обратной косой черты (\) и перевода строки

[1] Здесь и далее $_{opt}$ означает «optional», то есть «необязательно». — *Примеч. ред.*

escape-последовательность
универсальное-символьное-имя

escape-последовательность:
простая-escape-последовательность
восьмеричная-escape-последовательность
шестнадцатеричная-escape-последовательность

простая-escape-последовательность: одно из
\' \" \? \\ \a \b \f \n \r \t \v

восьмеричная-escape-последовательность:
\ *восьмеричная-цифра*
\ *восьмеричная-цифра восьмеричная-цифра*
\ *восьмеричная-цифра восьмеричная-цифра восьмеричная-цифра*

шестнадцатеричная-escape-последовательность:
\x *шестнадцатеричная-цифра*
шестнадцатеричная-escape-последовательность
шестнадцатеричная цифра

литерал-с-плавающей-точкой:
дробная-константа показатель$_{opt}$ *плавающий-суффикс*$_{opt}$
цифровая-последовательность показатель плавающий-суффикс$_{opt}$

дробная-константа:
цифровая-последовательность$_{opt}$. *цифровая-последовательность*
цифровая-последовательность .

показатель:
e *знак*$_{opt}$ *цифровая-последовательность*
E *знак*$_{opt}$ *цифровая-последовательность*

знак: одно из
\+ –

цифровая-последовательность:
цифра
цифровая-последовательность цифра

плавающий-суффикс: одно из
f l F L

строковый-литерал:
"*s-символьная-последовательность*$_{opt}$"
L" *s-символьная-последовательность*$_{opt}$"

s-символьная-последовательность:
s-символ
s-символьная-последовательность s-символ

s-символ:
любой элемент исходного набора символов кроме двойных кавычек (") обратной косой черты (\) и перевода строки
escape-последовательность
универсальное-символьное-имя

логический-литерал:
false
true

А.4. Программы

Программа — это набор *единиц-трансляции*, комбинируемых посредством компоновки (§ 9.4). *Единица-трансляции*, которую часто называют *исходным файлом*, — это последовательность *объявлений*:

единица-трансляции:
    последовательность-объявлений$_{opt}$

А.5. Выражения

Выражения описываются в главе 6 и в обобщенном виде в § 6.2. Определение *списка-выражений* идентично определению *выражения*. Существуют два правила для отличия запятой в качестве разделителя в списке аргументов функции и оператора запятая (последовательность, § 6.2.2).

первичное-выражение:
    литерал
    this
    :: *идентификатор*
    :: *идентификатор-оператора-функции*
    :: *квалифицированный-идентификатор*
    (*выражение*)
    идентификатор-выражение

идентификатор-выражение:
    неквалифицированный-идентификатор
    квалифицированный-идентификатор

неквалифицированный-идентификатор:
    идентификатор
    идентификатор-функции-оператора
    идентификатор-функции-преобразования
    ~ *имя-класса*
    идентификатор-шаблона

квалифицированный-идентификатор:
    спецификатор-вложенного-имени template$_{opt}$
        неквалифицированный идентификатор

спецификатор-вложенного-имени:
    имя-класса-или-пространства-имен ::
        спецификатор-вложенного-имени$_{opt}$
    имя-класса-или-пространства-имен ::
        template *спецификатор-вложенного-имени*

имя-класса-или-пространства-имен:
    имя-класса
    имя-пространства-имен

постфиксное-выражение:
    первичное-выражение
    постфиксное-выражение [*выражение*]
    постфиксное-выражение (*список-выражений*$_{opt}$)
    спецификатор-простого-типа (*список-выражений*$_{opt}$)
    typename ::$_{opt}$ *спецификатор-вложенного-имени идентификатор*
        (*список-выражений*$_{opt}$)

typename ::$_{opt}$ *спецификатор-вложенного-имени*
template$_{opt}$ *идентификатор-шаблона* (*список-выражений*$_{opt}$)
постфиксное-выражение . template$_{opt}$ ::$_{opt}$ *идентификатор-выражение*
постфиксное-выражение–>template$_{opt}$ ::$_{opt}$ *идентификатор-выражение*
постфиксное-выражение . *имя-псевдодеструктора*
постфиксное-выражение–>*имя-псевдодеструктора*
постфиксное-выражение ++
постфиксное-выражение ––
dynamic_cast < *идентификатор-типа* > (*выражение*)
static_cast < *идентификатор-типа* > (*выражение*)
reinterpret_cast < *идентификатор-типа* > (*выражение*)
const_cast < *идентификатор-типа* > (*выражение*)
typeid (*выражение*)
typeid (*идентификатор-типа*)

список-выражений:
выражение-присваивания
список-выражений , *выражение-присваивания*

имя-псевдодеструктора:
::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *имя-типа* :: ~ *имя-типа*
::$_{opt}$ *спецификатор-вложенного-имени* template
template *идентификатор-шаблона* :: ~ *имя-типа*
::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ ~ *имя-типа*

унарное-выражение:
постфиксное-выражение
++ *выражение-приведения*
–– *выражение-приведения*
унарный-оператор выражение-приведения
sizeof *унарное-выражение*
sizeof (*идентификатор-типа*)
new-выражение
delete-выражение

унарный-оператор: одно из
* & + – ! ~

new-выражение:
::$_{opt}$ new *новое-размещение*$_{opt}$ *идентификатор-нового-типа*
новый-инициализатор$_{opt}$
::$_{opt}$ new *новое-размещение*$_{opt}$ (*идентификатор- типа*)
новый-инициализатор$_{opt}$

новое-размещение:
(*список-выражений*)

идентификатор-нового-типа:
последовательность-спецификаторов-типа новый-объявитель$_{opt}$

новый-объявитель:
ptr-оператор новый-объявитель$_{opt}$
непосредственно-новый-объявитель

непосредственно-новый-объявитель:
[*выражение*]
непосредственно-новый-объявитель [*константное-выражение*]

новый-инициализатор:
(*список-выражений*$_{opt}$)

delete-выражение
::$_{opt}$ delete *выражение-приведения*
::$_{opt}$ delete [ ] *выражение-приведения*

выражение-приведения:
унарное-выражение
(*идентификатор-типа*) *выражение-приведения*

pm-выражение:
выражение-приведения
pm-выражение .* *выражение-приведения*
pm-выражение –>* *выражение-приведения*

мультипликативное-выражение:
pm-выражение
мультипликативное-выражение * *pm-выражение*
мультипликативное-выражение / *pm-выражение*
мультипликативное-выражение % *pm-выражение*

аддитивное-выражение:
мультипликативное-выражение
аддитивное-выражение + *мультипликативное-выражение*
аддитивное-выражение – *мультипликативное-выражение*

выражение-сдвига:
аддитивное-выражение
выражение-сдвига << *аддитивное-выражение*
выражение-сдвига >> *аддитивное-выражение*

выражение-отношения:
выражение-сдвига
выражение-отношения < *выражение-сдвига*
выражение-отношения > *выражение-сдвига*
выражение-отношения <= *выражение-сдвига*
выражение-отношения >= *выражение-сдвига*

выражение-равенства
выражение-отношения
выражение-равенства == *выражение-отношения*
выражение-равенства != *выражение-отношения*

выражение-И:
выражение-равенства
выражение-И & *выражение-равенства*

выражение-исключающего-ИЛИ:
выражение-И
выражение-исключающего-ИЛИ ^ *выражение-И*

выражение-включающего-ИЛИ:
выражение-исключающего-ИЛИ
выражение-включающего-ИЛИ | *выражение-исключающего-ИЛИ*

выражение-логического-И:
выражение-включающего-ИЛИ
выражение-логического-И && *выражение-включающего-ИЛИ*

выражение-логического-ИЛИ:
выражение-логического-И
выражение-логического-ИЛИ || *выражение-логического-И*

условное-выражение:
выражение-логического-ИЛИ
выражение-логического-ИЛИ ? выражение : выражение-присваивания

выражение-присваивания:
условное-выражение
выражение-логического-ИЛИ оператор-присваивания выражение-присваивания
выражение-throw

оператор-присваивания: одно из
= *= /= %= += -= >>= <<= &= ^= |=

выражение:
выражение-присваивания
выражение , выражение-присваивания

константное-выражение:
условное-выражение

Грамматические неоднозначности возникают из-за сходства между функциеподобным приведением и объявлением. Например:

```
int x;
void f()
{
    char(x);    // преобразование x в char или объявление char с именем x?
}
```

Все эти неоднозначности разрешаются в объявления. То есть «если нечто можно понять как объявление, это и есть объявление». Например:

```
T(a)->m;        // выражение-инструкция
T(a)++;         // выражение-инструкция

T(*e)(int(3));  // объявление
T(f)[4];        // объявление

T(a);           // объявление
T(a)=m;         // объявление
T(*b)();        // объявление
T(x), y, z = 7; // объявление
```

Это избавление от неоднозначностей чисто статическое. Единственная информация, используемая для имени, — это известно ли оно как имя типа или как имя шаблона. Если это не возможно определить, считается что имя относится к чему-то такому, что не является ни шаблоном, ни типом.

Конструкция ***template*** *неквалифицированный-идентификатор* используется для того, чтобы заявить, что в контексте, где это не может быть определено, *неквалифицированный-идентификатор* является именем шаблона (см. § В.13.5).

А.6. Инструкции

См. § 6.3.

инструкция:
помеченная-инструкция
инструкция-выражение
составная-инструкция

инструкция-выбора
инструкция-итерации
инструкция-перехода
инструкция-объявления
блок-try

помеченная-инструкция:
идентификатор : *инструкция*
case *константное-выражение* : *инструкция*
default : *инструкция*

инструкция-выражение:
выражение$_{opt}$;

составная-инструкция:
{ *последовательность-инструкций*$_{opt}$ }

последовательность-инструкций:
инструкция
последовательность-инструкций инструкция

инструкция-выбора:
if (*условие*) *инструкция*
if (*условие*) *инструкция* else *инструкция*
switch (*условие*) *инструкция*

условие:
выражение
последовательность-спецификаторов-типа объявитель = *выражение-присваивания*

инструкция-итерации:
while (*условие*) *инструкция*
do *инструкция* while (*выражение*) ;
for (*инструкция-инициализации-for условие*$_{opt}$; *выражение*$_{opt}$) *инструкция*

инструкция-инициализации-for:
инструкция-выражение
простое-объявление

инструкция-перехода:
break ;
continue ;
return *выражение*$_{opt}$;
goto *идентификатор* ;

инструкция-объявления:
объявление-блока

А.7. Объявления

Структура объявлений описана в главе 4, перечисления — в § 4.8, указатели и массивы — в главе 5, функции — в главе 7, пространства имен — в § 8.2, директивы компоновки — в § 9.2.4, классы хранения — в § 10.4.

последовательность-объявлений:
объявление
последовательность-объявлений объявление

объявление:
объявление-блока
определение-функции

объявление-шаблона
явное-инстанцирование
явная-специализация
спецификация-компоновки
определение-пространства-имен

объявление-блока:
простое-объявление
asm-определение
определение-псевдонима-пространства-имен
using-объявление
using-директива

простое-объявление:
последовательность-спецификаторв-объявлений$_{opt}$
список-инициализирующих-объявителей$_{opt}$;

спецификатор-объявления:
спецификатор-класса-хранения
спецификатор-типа
спецификатор-функции
friend
typedef

последовательность-спецификаторов-объявлений:
последовательность-спецификаторов-объявлений$_{opt}$
спецификатор-объявления

спецификатор-класса-хранения:
auto
register
static
extern
mutable

спецификатор-функции:
inline
virtual
explicit

имя-typedef:
идентификатор

спецификатор-типа:
спецификатор-простого-типа
спецификатор-класса
спецификатор-перечисления
спецификатор-сложного-типа
cv-квалификатор

спецификатор-простого-типа:
::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *имя-типа*
::$_{opt}$ *спецификатор-вложенного-имени* template$_{opt}$ *идентификатор-шаблона*
char
wchar_t
bool
short
int
long

signed
unsigned
float
double
void

имя-типа:
имя-класса
имя-перечисления
имя-typedef

спецификатор-сложного-типа:
ключевое-слово-класса ::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *идентификатор*
enum ::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *идентификатор*
typename ::$_{opt}$ *спецификатор-вложенного-имени идентификатор*
typename ::$_{opt}$ *спецификатор-вложенного-имени* template$_{opt}$
идентификатор-шаблона

имя-перечисления:
идентификатор

спецификатор-перечисления:
enum *идентификатор*$_{opt}$ { *список-элементов-перечисления*$_{opt}$ }

список-элементов-перечисления:
определение-элемента-перечисления
список-элементов-перечисления , определение-элемента-перечисления

определение-элемента-перечисления:
элемент-перечисления
элемент-перечисления = константное-выражение

элемент-перечисления:
идентификатор

имя-пространства-имен:
оригинальное-имя-пространства-имен
псевдоним-пространства-имен

оригинальное-имя-пространства-имен:
идентификатор

определение-пространства-имен:
определение-именованного-пространства-имен
определение-неименованного-пространства-имен

определение-именованного-пространства-имен:
определение-оригинального-пространства-имен
определение-расширенного-пространства-имен

определение-оригинального-пространства-имен:
namespace *идентификатор* { *тело-пространства-имен* }

определение-расширенного-пространства-имен:
namespace *имя-оригинального-пространства-имен*
{ *тело-пространства-имен* }

определение-неименованного-пространства-имен:
namespace { *тело-пространства-имен* }

тело-пространства-имен:
последовательность-объявлений$_{opt}$

псевдоним-пространства-имен:
идентификатор

определение-псевдонима-пространства-имен:
namespace *идентификатор* =
квалифицированный-спецификатор-пространства-имен

квалифицированный-спецификатор-пространства-имен
::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *имя-пространства-имен*

using-объявление:
using typename$_{opt}$ ::$_{opt}$ *спецификатор-вложенного-имени*
неквалифицированный-идентификатор;
using :: *неквалифицированный-идентификатор*;

using-директива:
using namespace ::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$
имя-пространства-имен;

asm-определение:
asm (*строковый-литерал*) ;

спецификация-компоновки:
extern *строковый-литерал* { *список-объявлений*$_{opt}$ }
extern *строковый-литерал объявление*

Грамматика допускает произвольную вложенность объявлений. Однако есть некоторые семантические ограничения. Например, не разрешаются вложенные функции (функции, определенные локально относительно других функций).

Перечень спецификаторов, которые начинают объявление, не может быть пустым (нет «неявного ***int***», § Б.2) и состоит из самой длинной возможной последовательности спецификаторов. Например:

```
typedef int I;
void f(unsigned I) { /* ... */ }
```

Здесь ***f***() принимает неименованный аргумент типа ***unsigned int***.

asm() — это вставка ассемблерного текста. Смысл определяется при реализации, но идея состоит в том, чтобы эта строка была частью ассемблерного кода, который будет вставлен в генерируемую программу в указаное место.

Можно объявить переменную регистровой (***register***). Это будет служить подсказкой компилятору оптимизировать постоянные обращения к переменной, что, впрочем, большинство современных компиляторов прекрасно понимают сами.

А.7.1. Объявители

См. § 4.9.1, главу 5 (указатели и массивы), § 7.7 (указатели на функции) и § 15.5 (указатели на члены).

список-инициализирующих-объявителей:
инициализирующий-объявитель
список-инициализирующих-объявителей , *инициализирующий-объявитель*

инициализирующий-объявитель:
объявитель инициализатор$_{opt}$

объявитель:
непосредственно-объявитель
ptr-оператор объявитель

непосредственно-объявитель:
объявитель-идентификатор
непосредственно-объявитель (*конструкция-объявления-параметра*)
последовательность-cv-квалификаторов$_{opt}$ *спецификация-исключения*$_{opt}$
непосредственно-объявитель [*константное-выражение*$_{opt}$]
(*объявитель*)

ptr-оператор:
* *последовательность-cv-квалификаторов*$_{opt}$
&
::$_{opt}$ *спецификатор-вложенного-имени* *
последовательность-cv-квалификаторов$_{opt}$

последовательность-cv-квалификаторов:
cv-квалификатор последовательность-cv-квалификаторов$_{opt}$

cv-квалификатор:
const
volatile

объявитель-идентификатор:
::$_{opt}$ *идентификатор-выражение*
::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *имя-типа*

идентификатор-типа:
последовательность-спецификаторов-типа абстрактный-объявитель$_{opt}$

последовательность-спецификаторов-типа:
спецификатор-типа последовательность-спецификаторов-типа$_{opt}$

абстрактный-объявитель:
ptr-оператор абстрактный-объявитель$_{opt}$
непосредственно-абстрактный-объявитель

непосредственно-абстрактный-объявитель:
непосредственно-абстрактный-объявитель$_{opt}$
(*конструкция-объявления-параметра*)
последовательность-cv-квалификаторов$_{opt}$
спецификация-исключения$_{opt}$
непосредственно-абстрактный-объявитель$_{opt}$
[*константное-выражение*$_{opt}$]
(*абстрактный-объявитель*)

конструкция-объявления-параметра:
список-объявления-параметров$_{opt}$ . . .$_{opt}$
список-объявления-параметров , . . .

список-объявления-параметров:
объявление-параметра
список-объявления-параметров , *объявление-параметра*

объявление-параметра:
последовательность-спецификаторов-объявлений объявитель
последовательность-спецификаторов-объявлений объявитель = выражение-присваивания
последовательность-спецификаторов-объявлений абстрактный-объявитель$_{opt}$
последовательность-спецификаторов-объявлений абстрактный-объявитель$_{opt}$ *= выражение-присваивания*

определение-функции:
последовательность-спецификаторов-объявлений$_{opt}$ *объявитель ctor-инициализатор*$_{opt}$ *тело-функции*
последовательность-спецификаторов-объявлений$_{opt}$ *объявитель try-блок-функции*

тело-функции:
составная-инструкция

инициализатор:
= конструкция-инициализатора
(список-выражений)

конструкция-инициализатора:
выражение-присваивания
{ список-инициализаторов ,$_{opt}$ *}*
{ }

список-инициализаторов:
конструкция-инициализатора
список-инициализаторов , конструкция-инициализатора

Спецификатор ***volatile*** — это подсказка компилятору, что объект может изменять свое значение не описанным в языке образом, так что агрессивной оптимизации следует избегать. Например, счетчик реального времени можно объявить так:

```
extern const volatile long clock;
```

Два последовательных считывания ***clock*** могут дать разные результаты.

А.8. Классы

См. главу 10.

имя-класса:
идентификатор
идентификатор-шаблона

спецификатор-класса:
заголовок-класса { спецификация-членов$_{opt}$ *}*

заголовок-класса:
ключевое-слово-класса идентификатор$_{opt}$ *конструкция-базы*
ключевое-слово-класса спецификатор-вложенного-имени идентификатор конструкция-базы$_{opt}$
ключевое-слово-класса спецификатор-вложенного-имени$_{opt}$ *идентификатор-шаблона конструкция-базы*$_{opt}$

ключевое-слово-класса:
class
struct
union

спецификация-члена:
объявление-члена спецификация-члена$_{opt}$
спецификатор-доступа : спецификация-члена$_{opt}$

объявление-члена:
последовательность-спецификаторов-объявлений$_{opt}$
список-объявителей-членов$_{opt}$;
определение-функции ;$_{opt}$
::$_{opt}$ *спецификатор-вложенного-имени* template$_{opt}$
неквалифицированный-идентификатор ;
using-объявление
объявление-шаблона

список-объявителей-членов:
объявитель-члена
список-объявителей-членов , объявитель-члена

объявитель-члена:
объявитель спецификатор-чистоты$_{opt}$
объявитель константный-инициализатор$_{opt}$
идентификатор$_{opt}$ *: константное-выражение*

спецификатор-чистоты:
= 0

константный-инициализатор:
= *константное-выражение*

Чтобы сохранить совместимость с C, в одной области видимости (§ 5.7) могут быть объявлены класс и не-класс с одинаковыми именами. Например:

```
struct stat { /* ... */ };
int stat (char* name, struct stat* buf);
```

В этом случае простое имя (***stat***) — это имя не-класса. На класс надо ссылаться с использованием префикса *ключевое-слово-класса*.

Константные выражения определены в § B.5.

A.8.1. Производные классы

См. главы 12 и 15.

конструкция-базы:
: список-спецификаторов-баз

список-спецификаторов-баз:
спецификатор-базы
список-спецификаторов-баз , спецификатор-базы

спецификатор-базы:
::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *имя-класса*

virtual *спецификатор-доступа*$_{opt}$ ::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *имя-класса*
спецификатор-доступа virtual$_{opt}$ ::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *имя-класса*

спецификатор-доступа:
private
protected
public

А.8.2. Особые функции-члены

См. § 11.4 (операторы преобразования), § 10.4.6 (инициализация членов класса) и § 12.2.2 (инициализация подобъектов базовых классов).

идентификатор-функции-преобразования:
operator *идентификатор-преобразования-типа*

идентификатор-преобразования-типа:
последовательность-спецификаторов-типа объявитель-преобразования$_{opt}$

объявитель-преобразования:
ptr-оператор объявитель-преобразоавния$_{opt}$

ctor-инициализатор:
: *список-инициализаторов-памяти*

список-инициализаторов-памяти:
инициализатор-памяти
инициализатор-памяти , *список-инициализаторов-памяти*

инициализатор-памяти:
идентификатор-инициализатора-памяти (*список-выражений*$_{opt}$)

идентификатор-инициализатора-памяти:
::$_{opt}$ *спецификатор-вложенного-имени*$_{opt}$ *имя-класса*
идентификатор

А.8.3. Перегрузка

См. главу 11.

идентификатор-функции-оператора:
operator *оператор*

оператор: одно из

new	delete		new[]		delete[]							
+	–	*	/	%	^	&	|	~	!	=	<	>
+=	–=	*=	/=	%=	^=	&=	|=	<<	>>	>>=	<<=	==
!=	<=	>=	&&	||	++	––	,	–>*	–>	()	[]	

А.9. Шаблоны

Шаблоны объясняются в главе 13 и § В.13.

объявление-шаблона:
export$_{opt}$ template < *список-параметров-шаблона* > *объявление*

список-параметров-шаблона:
параметр-шаблона
список-параметров-шаблона , параметр-шаблона

параметр-шаблона:
параметр-тип
объявление-параметра

параметр-тип:
class *идентификатор*$_{opt}$
class *идентификатор*$_{opt}$ = *идентификатор-типа*
typename *идентификатор*$_{opt}$
typename *идентификатор*$_{opt}$ = *идентификатор-типа*
template < *список-параметров-шаблона* > class *идентификатор*$_{opt}$
template < *список-параметров-шаблона* > class *идентификатор*$_{opt}$ =
идентификатор-выражение

идентификатор-шаблона:
имя-шаблона < *список-аргументов-шаблона*$_{opt}$ >

имя-шаблона:
идентификатор

список-аргументов-шаблона:
аргумент-шаблона
список-аргументов-шаблона , аргумент-шаблона

аргумент-шаблона:
выражение-присваивания
идентификатор-типа
идентификатор-выражение

явное-инстанцирование:
шаблон объявление

явная-специализация:
template < > *объявление*

Явная специализация аргумента шаблона открывает возможность для завуалированной синтаксической двусмысленности. Рассмотрим пример:

```
void h ()
{
    f<1>(0);    // неоднозначность: ((f)<1)>(0) или (f<1>)(0)?
                // разрешение: вызывается f<1> с аргументом 0
}
```

Разрешение просто и эффективно: если *f* — имя шаблона, то *f*< — начало квалифицированного имени шаблона, и последующие токены должны интерпретироваться с учетом этого; в противном случае < означает знак «меньше». Аналогично, первый невложенный знак > заканчивает список аргументов шаблона. Если требуется знак «больше», нужно использовать скобки:

```
f<a>b>(0);  // синтаксическая ошибка
f<(a>b)>(0); // правильно
```

Подобная же лексическая двусмысленность возникает, когда завершающие знаки > оказываются слишком близко друг к другу. Например:

```
list<vector<int>>lv1;      // синтаксическая ошибка: неожиданный >>
                           // (сдвиг вправо)
list< vector<int> >lv2;    // правильно: список векторов
```

Пробел между двумя > забывать нельзя; >> — это оператор сдвига вправо. Это в самом деле может доставить неприятности.

А.10. Обработка исключений

См. § 8.3 и главу 14.

блок-try:
try *составная-инструкция последовательность-обработчиков*

try-блок-функции:
try ctor-инициализатор$_{opt}$ *тело-функции последовательность-обработчиков*

последовательность-обработчиков:
обработчик последовательность-обработчиков$_{opt}$

обработчик:
catch (*объявление-исключения*) *составная-инструкция*

объявление-исключения:
последовательность-спецификаторов-типа объявитель
последовательность-спецификаторов-типа абстрактный-объявитель
последовательность-спецификаторов-типа
...

выражение-throw:
throw *выражение-присваивания*$_{opt}$

спецификация-исключения:
throw (*список-идентификаторов-типа*$_{opt}$)

список-идентификаторов-типа:
идентификатор-типа
список-идентификаторов-типа , идентификатор-типа

А.11. Директивы препроцессора

Препроцессор — это относительно простой обработчик макросов, работающий в основном с лексическими токенами, а не с отдельными символами. В дополнение к возможности определять и использовать макросы (§ 7.8), препроцессор предоставляет механизмы для включения текстовых и стандартных заголовочных файлов (§ 9.2.1), а также для условной компиляции, основываясь на макросах (§ 9.3.3). Например:

```
#if OPT==4
#include "header4.h"
#elif 0<OPT
#include "someheader.h"
#else
#include<cstdlib>
#endif
```

Все директивы препроцессора начинаются с символа #, который должен быть первым символом-не-разделителем в данной строке.

файл-препроцессора:
группа$_{opt}$

группа:
часть-группы
группа часть-группы

часть-группы:
pp-токены$_{opt}$ *перевод-строки*
фрагмент-if
управляющая-строка

фрагмент-if:
группа-if группы-elif$_{opt}$ *группа-else*$_{opt}$ *строка-endif*

группа-if:
if *константное-выражение перевод-строки группа*$_{opt}$
ifdef *идентификатор перевод-строки группа*$_{opt}$
ifdef *идентификатор перевод-строки группа*$_{opt}$

группы-elif:
группа-elif
группы-elif группа-elif

группа-elif:
elif *константное-выражение перевод-строки группа*$_{opt}$

группа-else:
else *перевод-строки группа*$_{opt}$

строка-endif:
endif *перевод-строки*

управляющая-строка:
include *pp-токены перевод-строки*
define *идентификатор список-подстановок перевод-строки*
define *идентификатор левая-скобка список-идентификаторов*$_{opt}$)
список-подстановок перевод-строки
undef *идентификатор перевод-строки*
line *pp-токены перевод-строки*
error *pp-токены*$_{opt}$ *перевод-строки*
pragma *pp-токены*$_{opt}$ *перевод-строки*
перевод-строки

левая-скобка:
символ левой круглой скобки
без символа-разделителя перед ней

список-подстановок:
pp-токены$_{opt}$

pp-токены:
токен-препроцессора
pp-токены токен-препроцессора

перевод-строки:
символ перевода строки

список-идентификаторов:
идентификатор
список-идентификаторов , идентификатор

Приложение Б

Совместимость

Вы продолжайте следовать своим обычаям,
а я буду следовать своим.
— Ч. Нэпьер

Совместимость C/C++ — «тихие» различия между C и C++ — код на C, не являющийся кодом на C++ — нежелательные особенности — код на C++, не являющийся кодом на C — устаревшие реализации C++ — заголовочные файлы — стандартная библиотека — пространства имен — ошибки распределения памяти — шаблоны — инициализаторы *for-инструкции*.

Б.1. Введение

В этом приложении обсуждаются несовместимости между C и C++, а также между стандартом C++ и более ранними версиями языка C++. Цель данной главы — документировать различия, которые могут вызвать проблемы у программиста, и указать пути их преодоления. Большинство проблем с совместимостью возникает: когда кто-то пытается обновить программу на C, чтобы сделать ее программой на C++, при попытках перенести написанную на C++ программу с одной нестандартной версии C++ на другую и при попытках скомпилировать программу с новейшими особенностями языка на старом компиляторе. Моя цель — не утопить вас в деталях каждой проблемы совместимости, когда-либо возникавшей в какой-либо реализации, а просто перечислить наиболее часто встречающиеся трудности и представить стандартные решения.

При анализе вопросов совместимости очень важно рассмотреть диапазон реализаций, в которых программа должна работать. Для изучения C++ имеет смысл использовать наиболее полную и удобную реализацию. Для конечного продукта правильной является более консервативная стратегия, призванная максимально расширить число систем, в которых этот продукт сможет работать. В прошлом это служило оправданием бегства от новшеств C++. Однако реализации сближаются, и необходимость переносимости с одной системы на другую теперь не требует той чрезвычайной осторожности, как пару лет назад.

Б.2. Совместимость C/C++

За минимальными исключениями C++ является надмножеством C. Наиболее значительные различия возникают из-за большей строгости C++ при проверке типов. Хорошо написанные программы на C как правило оказываются и программами на C++. Все различия между C++ и C может выявить компилятор.

Б.2.1. «Тихие» различия

За немногими исключениями, программы, являющиеся программами и на C, и на C++, имеют в обоих языках одинаковый смысл. К счастью, эти исключения («тихие» различия) сравнительно незначительны:

В C размер символьной константы и перечисления равны *sizeof(int)*. В C++ *sizeof('a')* равно *sizeof(char)*, и реализациям C++ разрешено выбирать размер, лучше всего подходящий для перечисления (§ 4.8).

C++ использует для комментариев знак //; C не позволяет этого (хотя многие реализации C тоже допускают такой комментарий в качестве расширения языка). Это различие можно использовать, чтобы конструировать программы, ведущие себя на разных языках по-разному. Например:

```
int f(int a, int b)
{
        return a // * довольно невероятно */ b
                ; /* нереалистично: точка с запятой в отдельной строке
                 во избежание синтаксической ошибки */
}
```

ISO C (международный стандарт C) пересматривается, чтобы ввести комментарии // в C.

Имя структуры, объявленное во внутренней области видимости, может скрыть имя объекта, функции, перечисления или типа во внешней области видимости. Например:

```
int x[99];
void f()
{
        struct x { int a; };
        sizeof(x);          // размер массива на C, размер структуры на C++
}
```

Б.2.2. Код на C, не являющийся кодом на C++

Несовместимости C/C++, вызывающие большинство реальных проблем, не очень изощренны. Многие из них легко вылавливаются компиляторами. В данном разделе приведены примеры кода на C, который не является допустимым кодом на C++. Многие из них для современного C считаются дурным тоном или даже устаревшими.

На C большинство функций можно вызывать без предварительного объявления:

```
main()                                          /* дурной тон в C. Не C++ */
{
        double sq2 = sqrt(2);                   /* вызов необъявленной функции */
        printf("корень из 2 равен %g\n", sq2);  /* вызов необъявленной функции */
}
```

Полное и последовательное использование объявлений (прототипов) функций, вообще говоря, рекомендуется и для C. Там, где этому разумному совету следуют, и особенно там, где компилятор C предоставляет опции обязательного объявления, программа на C соблюдает правила C++. Там, где вызываются необъявленные функции, вам нужно довольно хорошо знать функции и правила C, чтобы понять, где вы сделали ошибку или породили проблему, связанную с переносимостью. Например, приведенная выше функция *main()* как программа на C++ содержит по крайней мере две ошибки.

В языке C функция, объявленная без описания типов аргумента, может принимать любое число аргументов любого типа. Подобное использование признано в Стандарте C устаревшим, но оно встречается:

```
void f();                       /* типы аргумента не упоминаются */
```

```
void g()
{
    f(2);                    /* дурной тон в С. Не С++ */
}
```

В С функции можно определять с использованием синтаксиса, который (не обязательно) описывает типы аргументов после перечня аргументов:

```
void f(a, p, c) char* p, char c; { /* ... */ }  /* С, не С++ */
```

Такие определения надо переписать:

```
void f(int a, char* p, char c) { /* ... */ }
```

В С и старых версиях С++, предшествовавших стандарту, спецификатором типа по умолчанию считается ***int***. Например:

```
const a = 7; /* В С подразумевается тип int. Не С++ */
```

Стандарт ISO С пересматривается, чтобы тоже не допускать «неявного ***int***», как и в С++.

С допускает определения структур в возвращаемом типе и в объявлении типа аргумента. Например:

```
struct S { int x, y; } f();          /* С, не С++ */
void g(struct S { int x, y; } y);    /* С, не С++ */
```

Правила С++ определения типа делают такие объявления бесполезными и не допускают их.

В С переменным типа перечислений можно присваивать целые числа:

```
enum Direction { up, down };
enum Direction d = 1; /*ошибка: int присваивается Direction; в С — допустимо */
```

В С++ гораздо больше ключевых слов, чем в С. Если какое-нибудь из них появится в программе на С как идентификатор, эту программу нужно будет переделать для совместимости с С++:

Ключевые слова С++, которые не являются ключевыми словами С

and	*class*	*false*	*not_eq*	*reinterpret_cast*	*typeid*
and_eq	*compl*	*friend*	*operator*	*static_cast*	*typename*
asm	*const_cast*	*inline*	*or*	*template*	*using*
bitand	*delete*	*mutable*	*or_eq*	*this*	*virtual*
bitor	*dynamic_cast*	*namespace*	*private*	*throw*	*wchar_t*
bool	*explicit*	*new*	*protected*	*true*	*xor*
catch	*export*	*not*	*public*	*try*	*xor_eq*

В С некоторые из ключевых слов С++ являются макросами, определенными в стандартных заголовочных файлах:

Ключевые слова С++, являющиеся макросами на С

and	*bitand*	*compl*	*not_eq*	*or_eq*	*xor*
and_eq	*bitor*	*not*	*or*	*wchar_t*	*xor_eq*

Это подразумевает, что в С они могут проверяться при помощи ***#ifdef***, замещаться и т. п.

В языке С глобальные объекты данных можно объявлять в одной единице трансляции несколько раз без спецификатора ***extern***. Если инициализатор встречается только в одном из таких объявлений, объект считается определенным лишь один раз. Например:

```
int i; int i;          /* определяет или объявляет одну целую переменную i; не C++ */
```

В C++ переменная должна быть определена ровно один раз; § 9.2.3.

В C++ класс не может иметь то же имя, что и имя, определенное при помощи ***typedef***, для обозначения другого типа в той же области видимости; § 5.7.

В С ***void**** можно использовать в качестве правого операнда в присваивании или при инициализации переменной любого указательного типа; в C++ этого делать нельзя (§ 5.6). Например:

```
void f(int n)
{
        int* p = malloc(n*sizeof(int));    /* не C++. Выделяйте память в C++ при помощи new */
}
```

Язык С разрешает переходы (***goto***) в обход инициализации; C++ не разрешает.

В С имена вложенных структур размещаются в той же области видимости, что и структура, в которую они вложены. Например:

```
struct S {
        struct T { /* ... */
        // ...
};

struct T x;                /* В С правильно и означает: S::T x. Не C++ */
```

В С массив может инициализироваться инициализатором, в котором элементов больше, чем требуется для этого массива. Например:

```
char v[5] = "Оскар";    /* Допустимо в С, завершающий строку 0 не используется. Не C++ */
```

Б.2.3. Нежелательные особенности

Используя термин «нежелательные особенности», комитет по стандарту выражает пожелание эти особенности убрать. Однако у комитета нет права удалять широко используемые свойства — как бы избыточны и опасны они не были. Таким образом, нежелательность является для пользователей всего лишь настоятельным советом избегать этих особенностей.

Ключевым словом ***static***, обычно означающим «статически распределяемый», можно воспользоваться как индикатором, что функция или объект локальны для данной единицы трансляции. Например:

```
// файл1
        static int glob;

// файл2
        static int glob;
```

Эта программа на самом деле имеет две целочисленных переменных с именем ***glob***. Каждая ***glob*** используется исключительно функциями, определенными в ее единице трансляции.

Использование *static* для индикации «локальная для единицы трансляции» в C++ нежелательно. Вместо этого пользуйтесь неименованными пространствами имен (§ 8.2.5.1).

Неявное преобразование строкового литерала в (неконстантный) *char** нежелательно. Используйте именованые массивы *char* или избегайте присваивания строковых литералов переменным типа *char** (§ 5.2.2).

Приведения типа в стиле C тоже стали нежелательны после введения приведений в новом стиле. Программисты должны серьезно подумать над запрещением приведения в стиле C в своих программах. Там где необходимо явное преобразование, то же самое могут сделать *static_cast*, *reinterpret_cast*, *const_cast* или их сочетания. Следует предпочитать приведения типа в новом стиле, так как они более явные, и их лучше видно (§ 6.2.7).

Б.2.4. Код на C++, не являющийся кодом на C

В данном разделе перечисляются возможности C++, не обеспечиваемые C. Эти особенности рассортированы по назначению. Однако их можно классифицировать поразному, и многие особенности имеют несколько назначений, так что к данной классификации не следует относиться слишком серьезно.

- Особенности, предназначенные прежде всего для удобства записи:
  [1] комментарии с помощью // ; в настоящее время добавляется к C;
  [2] поддержка ограниченного набора символов (§ B.3.1);
  [3] поддержка расширенного набора символов (§ B.3.3); в настоящее время добавляется к C;
  [4] неконстантные инициализаторы для объектов со статическим хранением (§ 9.4.1);
  [5] *const* в константных выражениях;
  [6] объявления как инструкции (§ 6.3.1);
  [7] объявления в инициализаторах *for-инструкции* и условных инструкциях (§ 6.3.3, § 6.3.2.1);
  [8] имена структур не обязательно предварять словом *struct* (§ 5.7).
- Особенности, предназначенные прежде всего для усиления системы типов:
  [1] проверка типа аргумента в функции (§ 7.1); была добавлена в C (§ Б.2.2);
  [2] компоновка, безопасная с точки зрения типов (§ 9.2, § 9.2.3);
  [3] распределение свободной памяти при помощи *new* и *delete* (§ 6.2.6, § 10.4.5, § 15.6);
  [4] *const* (§ 5.4, § 5.4.1); было добавлено в C;
  [5] логический тип *bool* (§ 4.2).
  [6] новый синтаксис приведения (§ 6.2.7).
- Средства, предназначенные для поддержки типов, определяемых пользователем:
  [1] классы (глава 10);
  [2] функции-члены (§ 10.2.1) и классы-члены (§ 11.12);
  [3] конструкторы и деструкторы (§ 10.2.3, § 10.4.1);
  [4] производные классы (главы 12 и 15);
  [5] виртуальные функции и абстрактные классы (§ 12.2.6, § 12.3);
  [6] контроль доступа, открытый/защищенный/закрытый (*public/protected/private*) (§ 10.2.2, § 15.3, § B.11);

[7] друзья (§ 11.5);
[8] указатели на члены (§ 15.5, § B.12);
[9] статические члены (§ 10.2.4);
[10] члены ***mutable*** (§ 10.7.2);
[11] перегрузка операторов (глава 11);
[12] ссылки (§ 5.5).

Особенности, предназначенные в первую очередь для организации программ (в добавление к классам):

[1] шаблоны (глава 13, § B.13);
[2] встроенные функции (§ 7.1.1);
[3] аргументы по умолчанию (§ 7.5);
[4] перегрузка функций (§ 7.4);
[5] пространства имен (§ 8.2);
[6] явное разрешение области видимости (оператор ::; § 4.9.4);
[7] обработка исключений (§ 8.3, глава 14);
[8] определение типа во время выполнения (§ 15.4).

Ключевыми словами, добавленными в C++ (§ Б.2.2) можно пользоваться для выявления специфических средств C++. Однако некоторые средства, такие как перегрузка функций и ***const*** в константных выражениях, не идентифицируются по ключевым словам. Кроме перечисленных здесь языковых особенностей библиотека C++ (§ 16.1.2) по большей части специфична для C++.

Для определения того, каким компилятором должна компилироваться программа — C или C++, — можно пользоваться макросом ***__cplusplus*** (§ 9.2.4).

Б.3. Старые реализации C++

C++ постоянно использовался, начиная с 1983 года (§ 1.4). С тех пор было определено несколько версий, и появилось несколько независимо разработанных реализаций. Фундаментальная цель попыток стандартизации заключалась в том, чтобы разработчики реализаций и пользователи смогли работать с единым определением C++. Однако, до тех пор, пока это определение на распространится повсеместно в сообществе C++, нам придется смириться с тем фактом, что не каждая реализация обеспечивает все средства, описанные в этой книге.

К сожалению, не редкость, когда первое серьезное знакомство с C++ происходит с реализацией пятилетней давности. Как правило, это вызвано тем, что такие реализации широко доступны и свободно распространяются. Имея выбор, ни один уважающий себя профессионал не будет работать на таком антиквариате. А для новичка устаревшие реализации оборачиваются серьезными скрытыми затратами. Отсутствие языковых средств и поддержки со стороны библиотеки означает, что новичку придется бороться с проблемами, устраненными в новых реализациях. Кроме того, использование сравнительно бедных устаревших реализаций наносит вред стилю программирования новичка и порождает искаженный взгляд на то, чем является C++ на самом деле. Лучшее подмножество C++ для первоначального изучения — *не* набор возможностей низкого уровня (и не общее подмножество C и C++; § 1.2). В частности, чтобы облегчить обучение и получить хорошее первое впечатление о программировании на C++, я рекомендую опираться на стандартную библиотеку и шаблоны.

Первый коммерческий выпуск C++ состоялся в конце 1985 года. Язык определялся первой редакцией данной книги. На том этапе C++ не предлагал множественного насле-

дования, шаблонов, информации о типе во время выполнения, исключений и пространств имен. Сегодня я не вижу смысла пользоваться реализацией, не обеспечивающей хотя бы части этих возможностей. В 1989 году я добавил к определению множественное наследование, шаблоны и исключения. Однако ранняя поддержка шаблонов и исключений была шероховатой и зачастую довольно убогой. Если в старых реализациях у вас возникнут проблемы с шаблонами или исключениями, поскорее обновите версию.

Вообще говоря, по мере возможности разумно применять реализацию, совпадающую со стандартом, и минимизировать использование аспектов языка, зависящих от реализации или неопределенных. Проектируйте так, как будто вам доступен весь язык, а затем ищите нужные средства. Это ведет к лучше организованным программам, которые легче сопровождать, по сравнению с проектированием, использующим подмножество C++, отвечающее «наименьшему общему знаменателю». Также будьте осторожны в использовании расширений, специфических для данной реализации языка, и применяйте их только в случае крайней необходимости.

Б.3.1. Заголовочные файлы

Традиционно все заголовочные файлы имели расширение *.h*. Таким образом, реализации C++ вводили заголовочные файлы, такие как *<map.h>* и *<iostream.h>*. Из соображений совместимости большинство реализаций по-прежнему так и делают.

Когда комитету по стандартам понадобились заголовочные файлы для новых версий стандартных библиотек и для добавляемых возможностей библиотеки, именование этих заголовочных файлов стало проблемой. Использование старых имен с расширением *.h* породило бы трудности с совместимостью. И было решено для имен стандартных заголовочных файлов отказаться от суффикса *.h*. Он все равно избыточен, поскольку обозначение < > и так указывает, что речь идет об имени стандартного заголовочного файла.

Таким образом, стандартная библиотека предоставляет заголовочные файлы без расширения, такие как *<iostream>* и *<map>*. Объявления в этих файлах располагаются в пространстве имен *std*. Устаревшие заголовочные файлы располагают свои объявления в глобальном пространстве имен и используют расширение *.h*. Рассмотрим пример:

```
#include<iostream>
int main ()
{
    std::cout << "Здравствуй, мир!\n";
}
```

Если в вашей реализации это не удается скомпилировать, попробуйте более традиционную версию:

```
#include<iostream.h>
int main ()
{
    cout << "Здравствуй, мир!\n";
}
```

Несколько самых серьезных проблем с переносимостью возникает из-за непереносимых заголовочных файлов. Стандартные заголовочные файлы вносят в эту несовместимость лишь малую долю. Часто программы зависят от большого числа заголовочных

файлов, которые присутствуют не на всех системах, от большого числа объявлений, которые не на всех системах располагаются в тех же заголовочных файлах, и от объявлений, которые считаются стандартными (поскольку находятся в заголовочных файлах со стандартными именами), но не являются частью какого-либо стандарта.

Вполне удовлетворительных подходов к решению вопросов переносимости перед лицом несогласованности заголовочных файлов не существует. Главная идея состоит в том, чтобы избежать прямых зависимостей от несогласованных заголовочных файлов и локализовать оставшиеся зависимости. То есть мы пытаемся добиться переносимости посредством косвенности и локализации. Например, если нужные нам объявления в разных системах вводятся в разных заголовочных файлах — мы можем включить один специфичный для данной прикладной программы заголовочный файл, который в свою очередь включит соответствующие заголовочные файлы, для каждой системы разные. Аналогично, если какие-то возможности в разных системах предоставляются в несколько отличной форме, мы можем обращаться к этим средствам через специфичные для прикладной программы интерфейсные классы и функции.

Б.3.2. Стандартная библиотека

Естественно, реализации C++, предшествовавшие стандарту, могут не иметь некоторых частей стандартной библиотеки. В большинстве из них будут потоки ввода/вывода, нешаблонный класс ***complex***, различные строковые классы и стандартная библиотека C. Однако могут отсутствовать ассоциативные массивы (***map***), списки (***list***), ***valarray*** и т. д. В таких случаях пользуйтесь доступными библиотеками (как правило, являющимися собственностью той или иной фирмы), которые бы позволили впоследствии перейти к стандарту при обновлении вашей реализации. Обычно лучше пользоваться нестандартными строками, списками и ассоциативными массивами, чем при отсутствии этих классов в стандартной библиотеке возвращаться к программированию в стиле C. Также доступны для свободной загрузки хорошие реализации STL-части стандартной библиотеки (главы 16, 17, 18 и 20).

Ранние реализации стандартной библиотеки были неполными. Например, некоторые имеют контейнеры, которые не поддерживают распределители памяти, а другие требуют, чтобы распределители памяти были явно указаны для каждого класса. Схожие проблемы возникают для других «аргументов, определяющих политику алгоритма», таких как критерии сравнения. Например:

```
list<int> li;   // правильно, но некоторые реализации требуют распределителя памяти
list<int, allocator<int> > li2;      // правильно, но некоторые реализации
                                     // не реализуют распределителей памяти

map<string, Record> m1;              // правильно, но некоторые реализации требуют
                                     // указания операции «меньше» (less)
map<string, Record, less<string> > m2;
```

Пользуйтесь той версией, которую позволяет реализация. В конце концов реализации позволят все.

Ранние реализации C++ предоставляли ***istrstream*** и ***ostrstream***, определенные в ***<strstream.h>***, а не ***istringstream*** и ***ostringsteram***, определяемые в ***<sstream>***. Потоки ***strstream*** работали прямо с ***char***[] (см. § 21.10[26]).

Потоки в реализациях C++, предшествовавших стандарту, не параметризовались. В частности, шаблоны с префиксом ***basic_*** являются новыми для стандарта, а класс ***basic_ios*** обычно назывался ***ios***. Довольно любопытно, что ***iostate*** обычно назывался ***io_state***.

Б.3.3. Пространства имен

Если ваша реализация не поддерживает пространств имен, для выражения логической структуры программы пользуйтесь исходными файлами (глава 9). Аналогично, для выражения интерфейсов, которые вы предоставляете для реализаций, или которые являются общими с C, пользуйтесь заголовочными файлами.

В отсутствие пространств имен, чтобы компенсировать недостаток в неименованных пространствах имен, пользуйтесь ***static***. Также пользуйтесь идентифицирующими префиксами перед глобальными именами, чтобы отличать ваши имена от имен из других частей кода. Например:

```
// для использования в реализациях, предшествовавших пространствам имен
class bs_string { /* ... */ };          // строка Бьерна
typedef int bs_bool;                    // логический тип Бьерна
class john_string;                      // строка Джона
enum john_bool {john_false, john_true}; // логический тип Джона
```

Будьте осторожны при выборе префикса. Существующие библиотеки C и C++ переполнены такого рода префиксами.

Б.3.4. Ошибки распределения памяти

В C++ до появления обработки исключений оператор ***new*** возвращал ***0***, когда выделение памяти не удавалось. В стандарте C++ ***new*** по умолчанию генерирует исключение ***bad_alloc***.

Как правило, лучше стремиться к сходству со стандартом. В данном случае это означает, что следует изменить программу так, чтобы перехватывать ***bad_alloc***, а не проверять на ***0***. В обоих случаях при исчерпании памяти сделать что-либо, кроме выдачи сообщения об ошибке, не так-то просто в большинстве систем.

Однако, когда преобразование программы от проверки на ***0*** к перехватыванию ***bad_alloc***, нереалистично, вы можете в некоторых случаях изменить программу, чтобы вернуться к поведению, соответствующему старым реализациям. Если не задан обработчик ***_new_handler***, использование распределителя ***nothrow*** приведет к возвращению ***0*** в случае неудачного распределения:

```
X* p1 = new X;              // возбуждает bad_alloc, если нет памяти
X* p2 = new (nothrow) X;    // возвращает 0, если нет памяти
```

Б.3.5. Шаблоны

Стандарт вводит новые средства для работы с шаблонами и вносит ясность в правила работы с некоторыми старыми средствами.

Если ваша реализация не поддерживает частичную специализацию, пользуйтесь отдельным именем для шаблона, который иначе был бы специализацией. Например:

```
template <class T> class plist :
        private list<void*> {       // должно было бы быть list<T*>
        // ...
};
```

Если ваша реализация не поддерживает шаблоны членов, некоторые методы станут недоступны. В частности, шаблоны членов позволяют программисту задавать конструиро-

вание и преобразование с гибкостью, недостижимой без них. Иногда в качестве альтернативы можно рассмотреть введение функции-не-члена, которая конструирует объект:

```
template <class T> class X {
        //...
        template<class A> X(const A& a);
};
```

При отсутствии шаблонов-членов, мы должны ограничить себя определенными типами:

```
template<class T> class X {
        //...
        X(const A1& a);
        X(const A2& a);
        //...
};
```

Большинство ранних реализаций генерировали определения для всех функций-членов, определенных внутри шаблона класса, когда шаблон инстанцировался. Это могло приводить к ошибкам в неиспользуемых функциях-членах (§ B.13.9.1). Решение заключается в том, чтобы разместить определение функций-членов после объявления класса. Например, вместо:

```
template<class T> class Container {
        //...
public:
        void sort() { /* используем <*/ }    // определение внутри класса
};

class Glob { /* для Glob нет <*/ };

Container<Glob> sg;          // некоторые реализации, предшествовавшие стандарту,
                             // пытаются определить Container<Glob>::sort()
```

используйте

```
template<class T> class Container {
        //...
public:
        void sort()
};

template<class T> void Container<T>::sort() { /* используем <*/ } // определение вне класса

class Glob { /* для Glob нет <*/ };

Container<Glob> cg;                  // проблем не возникает, пока не будет вызвана cg.sort
```

Ранние реализации C++ не понимают членов, определенных в классе позже. Например:

```
template<class T> class Vector {
public:
        T& operator[] (size_t i) { return v[i]; }          // v объявлено ниже
        //...
private:
        T* v;                                              // не найдено!
        size_t sz;
};
```

В таких случаях или упорядочите объявления членов, чтобы избежать проблем, или расположите определение функций-членов после объявления класса.

Некоторые реализации C++, предшествовавшие стандарту, не допускают для шаблона аргументов по умолчанию (§ 13.4.1). В этом случае каждый параметр шаблона нужно задать как явный аргумент. Например:

```
template<class T, class LT = less<T> > class map {
        //...
};

map<string int> m;          // увы: аргументы шаблона по умолчанию не реализованы
map<string int, less<int> > m2;          // пойдем в обход: сделаем аргумент явным
```

Б.3.6. Инициализаторы for-инструкции

Рассмотрим следующий код:

```
void f(vector<char>& v, int m)
{
        for (int i=0; i<v.size() && i<=m; ++i) cout << v[i];
        if (i==m) {                                  // ошибка: обращение к i после for
                //...
        }
}
```

Такой код обычно работал, потому что в изначальном определении C++ область видимости контролируемых переменных распространялась до конца области видимости, в которую входила *for-инструкция*. Если вы нашли такой код, просто объявите контролируемую переменную перед *for-инструкцией*:

```
void f2(vector<char>& v, int m)
{
        int i=0;                                     // i понадобится после цикла
        for (; i<v.size() && i<=m; ++i) cout << v[i];
        if (i==m) {
                //...
        }
}
```

Б.4. Советы

[1] Для изучения C++ используйте самую последнюю и полную из доступных вам реализаций стандартного C++; § Б.3.

[2] Общее подмножество C и C++ не есть лучшее для изучения начальное подмножество; § 1.6., § Б.3.

[3] При создании кода помните: не каждая реализация C++ полностью обновлена. До того, как использовать новое средство в промышленном коде, попробуйте его в небольшой программе для проверки соответствия стандарту и производительности реализации, которую вы планируете использовать. Для примера см. § 8.5[6–7], § 16.5[10], § Б.5[7].

[4] Избегайте «неодобряемых» возможностей, типа глобальных статических переменных. Избегайте приведения типов в стиле C; § 6.2.7, § Б.2.3.

[5] Неявный *int* был выкинут, поэтому явно определяйте тип каждой функции, переменной, *const* и пр.; § Б.2.2.

[6] При преобразовании программы с C на C++ сначала убедитесь, что последовательно используются объявления функций (прототипы) и стандартные заголовочные файлы; § Б.2.2.

[7] При преобразовании программы с C на C++ переименуйте переменные C, которые совпадают с ключевыми словами C++; § Б.2.2.

[8] При преобразовании программы с C на C++ приводите результат, возвращаемый *malloc* () к соответствующему типу, или замените все вызовы *malloc* () на *new*; § Б.2.2.

[9] При преобразовании *malloc* () и *free* () к *new* и *delete* подумайте об использовании *vector*, *push_back* () и *reserve* (), вместо *realloc* (); § 3.8, § 16.3.5.

[10] При преобразовании программы с C на C++ помните, что нет неявного преобразования от типа *int* к перечислениям. При необходимости используйте явное преобразование типов; § 4.8.

[11] Средства из пространства имен *std* определяются в заголовке без суффикса (т. е. *std*::*cout* объявлена в <*iostream*>). Старые реализации содержат средства стандартной библиотеки в глобальном пространстве имен, объявленные в заголовочных файлах с суффиксом *.h* (т. е. ::*cout* объявлена в <*iostream.h*>); § 9.2.2, § Б.3.1.

[12] В старом коде проверки на равенство 0 результата, возвращаемого *new*, должны быть заменены на перехват исключения *bad_alloc* или на использование *new* (*nothrow*); § Б.3.4.

[13] Если ваша реализация не поддерживает аргументы шаблонов по умолчанию, используйте аргументы явно. Часто *typedef* может быть использовано для предотвращения повторения аргументов шаблона (аналогично тому, как *typedef string* позволяет вам не использовать *basic_string*< *char*, *char _traits*<*char*>, *allocator*<*char*> >); § Б.3.5.

[14] Используйте <*string*> для доступа к *std*::<*string*> (<*string.h*> содержит функции для строк в стиле C); § 9.2.2, § Б.3.1.

[15] Для каждого стандартного заголовочного файла языка C <*X.h*>, который помещает имена в глобальное пространство имен, заголовочный файл <*cX*> помещает имена в пространство имен *std*; § Б.3.1.

[16] Многие системы содержат заголовочный файл "*String.h*", определяющий строковый тип. Отметим, что такие строки отличаются от типа *string* из стандартной библиотеки.

[17] Предпочитайте стандартные средства нестандартным; § 20.1, §Б.3, В.2.

[18] Используйте *extern "C"* при объявлении C-функции; § 9.2.4.

Б.5. Упражнения

1. (*2.5) Возьмите программу на C и преобразуйте ее в программу на C++. Составьте список использованных конструкций, не являющихся конструкциями C++, и определите, соответствуют ли они стандарту ANSI C. Сначала приведите программу в строгое соответствие с ANSI C (добавьте прототипы и т. д.), затем преобразуйте в программу на C++. Оцените время, необходимое для такого преобразования программы в 100 000 строк.
2. (*2.5) Напишите программу, помогающую конвертировать код из C в C++ путем переименования переменных, совпадающих с ключевыми словами C++, замены вызовов *malloc* () на *new* и т. д. Подсказка: не пытайтесь достичь совершенства.
3. (*2) В C++ программе, написанной в стиле C (возможно той же, что преобразовывалась в упражнении 1) замените все используемые вызовы *malloc* () на вызовы *new*. Подсказка: см. § Б.4[8–9].
4. (*2.5) В C++ программе (возможно той же, что преобразовывалась в упражнении 1) минимизируйте использование макросов, глобальных переменных, инициализируемых переменных и приведение типов в стиле C .
5. (3*) Возьмите C++ программу, являющуюся результатом «грубого» преобразования из C, и покритикуйте ее с точки зрения C++ на локальность информации, абстракцию, удобочитаемость, расширяемость и на возможность повторного использования отдельных частей. Проведите в программе какое-либо значительное изменение, основанное на вашей критике.
6. (*2) Возьмите небольшую (около 500 строк) C++ программу и преобразуйте ее в программу на C. Сравните исходную и преобразованную программы по размерам и возможностям сопровождения.
7. (3*) Напишите небольшой набор текстовых программ для определения, поддерживает ли ваша реализация C++ «наипоследнейше» средства из стандарта. Например: какова область видимости переменной, описанной в инициализаторе *forstatement* (§ Б.3.6)? Поддерживаются ли аргументы шаблонов по умолчанию (§ Б.3.5) и члены шаблона (§ 13.6.2)? Поддерживается ли поиск имен по аргументам (§ 8.2.6)? Подсказка: см. § Б.2.4.
8. (2.5*) Возьмите C++ программу, использующую <*X.h*> и преобразуйте ее на использование <*X*> и <*cX*>. Минимизируйте использование директив *using*.

Приложение B

Технические подробности

Глубоко, в самом сердце
сознания и Вселенной —
есть смысл.
— Слартибартфаст

Что гарантирует стандарт? — наборы символов — целые литералы — константные выражения — продвижения и преобразования — многомерные массивы — поля и объединения — управление памятью — сборка мусора — пространства имен — контроль доступа — указатели на члены данных — шаблоны — статические члены — друзья — шаблоны в качестве параметров шаблона — выведение аргумента шаблона — ***typename*** и шаблоны — инстанцирование — связывание имен — шаблоны и пространства имен — явное инстанцирование — советы.

B.1. Введение и обзор

Эта глава описывает технические детали и примеры, которые не совсем вписываются в мою презентацию основных черт языка C++ и их использования. Представленные здесь подробности могут оказаться важными, когда вы будете писать программы, и очень важными, когда вы будете читать программы, написанные с их применением. Однако я считаю их техническими деталями, которые не должны отвлекать студентов от главной цели, — научиться хорошо применять язык C++, а программистов от того, чтобы стараться выразить идеи как можно яснее и непосредственнее, насколько позволяет C++.

B.2. Стандарт

Вопреки общему мнению, строгое следование стандартам языка C++ и библиотеки не гарантирует получения хорошей или даже переносимой программы. Стандарт ничего не говорит о том, хорош или плох данный фрагмент программы; он просто говорит, на что программист может (или не может) рассчитывать в реализации. Можно написать совершенно ужасающую программу в полном соответствии со стандартом, а большинство программ реального мира опираются на особенности, не описываемые стандартом.

Очень важные вещи стандарт считает *определенными в реализации*. Это означает, что каждая реализация должна обеспечить специфическое четко определенное поведение ряда конструкций и задокументировать его. Например:

```
unsigned char c1 = 64;      // четко определено: char имеет по крайней
                            // мере 8 бит и всегда может содержать число 64
unsigned char c2 = 1256;    // определяется в реализации: урезание, если
                            // char имеет только 8 бит
```

Инициализация ***c1*** четко определена, поскольку в ***char*** должно быть по крайней мере 8 бит. Однако поведение инициализации ***c2*** зависит от реализации, так как число битов в ***char*** определяется в реализации. Если ***char*** имеет только 8 бит, значение 1256 будет урезано до 232 (§ В.6.2.1). Большинство определяемых в реализации особенностей относится к различиям в «железе», на котором будет работать программа.

Во время написания реальной программы обычно приходится опираться на поведение, определяемое в реализации. Это является ценой, которую мы платим за возможность эффективно работать с различными системами. Например, язык был бы гораздо проще, если бы все ***char*** были 8-битовыми, а все ***int*** — 32 битовыми. Однако не так уж необычны 16-битовые и 32-битовые символьные наборы — так же как и числа, слишком большие, чтобы уместиться в 32 бита. Например, многие компьютеры теперь имеют диски емкостью более 32 Гбайт, так что для представления адреса на диске уместны 48-битовые и 64-битовые числа.

Чтобы максимально повысить переносимость, разумно явно выразить, на какие определяемые в реализации свойства мы опираемся, и самые сомнительные места изолировать в четко определенных местах программы. Типичный пример такой практики — представлять все зависимости от аппаратуры в виде констант и определений типов в некотором заголовочном файле. Чтобы поддержать такую технику, стандартная библиотека предоставляет ***numeric_limits*** (§ 22.2).

Неопределенное поведение неприятнее. Конструкция считается *не определенной* стандартом, если от реализации не требуется какого-либо осмысленного поведения. Как правило, некоторые очевидные приемы реализации приводят к тому, что программы, использующие неопределенности, ведут себя очень плохо. Например:

```
const int size = 4*1024;
char page[size];

void f()
{
    page[size+size] = 7;        // не определено
}
```

Вероятный результат этого фрагмента кода — запись поверх данных, не относящихся к программе, и генерирование аппаратной ошибки/исключения. От реализации не требуется выбирать между вероятными результатами. Там, где используются мощные оптимизаторы, эффект от неопределенного поведения может стать совершенно непредсказуемым. Если есть набор удовлетворительных и легко реализуемых альтернатив, особенность считается не неопределенной, а определяемой в реализации.

Вполне себя оправдывают значительные время и силы, затраченные на гарантии того, что программа не использует ничего не определенного стандартом. Во многих случаях для этого существуют инструментальные средства.

В.3. Символьные наборы

Примеры в этой книге[1] написаны с использованием американского (США) варианта международного 7-битного символьного набора ISO 646-1983, называемого ASCII (ANSI3.4-1968). У тех, кто применяет C++ в среде, использующей другой символьный набор, это может вызвать три проблемы:

[1] ASCII содержит символы пунктуации и операторные символы — такие как [, { и !; для некоторых наборов эти символы недоступны.

[2] Нам нужны обозначения для символов, не имеющих соответствующего символьного представления (например, перевод строки и «символ со значением 17»).

[3] ASCII не содержит символов, использующихся в других языках (не английском) — например, ξ, æ или П.

В.3.1. Сокращенный символьный набор

Специальные символы ASCII [,], {, }, | и \ занимают позиции в символьном наборе, которые определены ISO как алфавитно-цифровые. В большинстве европейских символьных наборов ISO-646 эти позиции заняты буквами, которых нет в английском алфавите. Например, датский национальный набор символов на их месте содержит гласные ***Æ***, ***æ***, ***Ø***, ***ø***, ***Å*** и ***å***. Без этих символов по-датски не напишешь сколько-нибудь осмысленного текста.

Для того чтобы выражать национальные буквы переносимым образом при помощи по-настоящему стандартного минимального символьного набора, введен набор триграфов. Это может пригодиться для обмена программами, но не облегчает людям чтение таких программ. Естественно, долговременное решение этой проблемы — дать программистам на C++ такое оборудование, которое поддерживает и их родной язык, и C++. К сожалению, для некоторых это представляется недостижимым, и введение нового обрудования может оказаться до огорчения медленным процессом. Чтобы помочь программистам обходиться с неполным набором символов, C++ обеспечивает альтернативы:

Ключевые слова		Диграфы		Триграфы	
and	&&	<%	{	??=	#
and_eq	&=	%>	}	??(	[
bitand	&	<:	[	??<	{
bitor	|	:>	]	??/	\
compl	~	%:	#	??)	]
not	!	%:%:	##	??>	}
or	||			??'	^
or_eq	|=			??!	|
xor	^			??-	~
xor_eq	^=				
not_eq	!=				

Программы с ключевыми словами и диграфами гораздо лучше читаемы чем эквивалентные программы, написанные с использованием триграфов. Однако если такие

[1] Имеется ввиду американское издание. — *Примеч. ред.*

символы как { недоступны, триграфы необходимы, чтобы не «потерять» символы в строках и символьных константах. Например, '{' превращается в '??<'.

Некоторые предпочитают вместо традиционного обозначения операторов пользоваться ключевыми словами, такими как ***and***.

В.3.2. Escape-символы

Несколько символов имеют стандартные имена, использующие символ обратной косой черты \ в качестве escape-символа:

Название	Название ASCII	Название C++
перевод строки	NL (LF)	\n
горизонтальная табуляция	HT	\t
вертикальная табуляция	VT	\v
забой	BS	\b
возврат каретки	CR	\r
перевод страницы	FF	\f
гудок	BEL	\a
обратная косая черта	\	\\
знак вопроса	?	\?
апостроф	'	\'
двойная кавычка	"	\"
восьмеричное число	*ooo*	*ooo*
шестнадцатеричное число	*hhh*	\x*hhh* ...

Несмотря на свой внешний вид, это одиночные символы.

Символ можно представить в виде одной, двух или трех восьмеричных цифр (с \ перед ними) или шестнадцатеричным числом (с ***x** перед цифрами). На число шестнадцатеричных цифр в последовательности нет ограничений. Последовательность восьмеричных или шестнадцатеричных цифр заканчивается первым символом, не являющимся соответственно восьмеричной или шестнадцатеричной цифрой. Например:

Восьмеричная	Шестнадцатеричная	Десятичная	ASCII
'\6'	'\x6'	6	ACK
'\60'	'\x30'	48	'0'
'\137'	'\x05f'	95	'_'

Это позволяет представить любой символ из машинного символьного набора и, в частности, вставить такие символы в символьные строки (см. § 5.2.2). Использование любых цифровых обозначений для символов делает программу непереносимой с одной машины на другую, на которой применяется другой символьный набор.

Символьный литерал можно записать несколькими символами, например '***ab***'. Такое применение архаично, зависит от реализации, и его лучше избегать.

При вставке в строку цифровой константы в восьмеричном обозначении всегда разумно пользоваться трехзначными числами. Подобные обозначения достаточно трудно читать, не задумываясь, цифра стоит после константы или не цифра. Для

шестнадцатеричных констант пользуйтесь двумя цифрами. Рассмотрим следующие примеры:

```
char v1[] = "a\xah\129";      // 6 символов: 'a' '\xa' 'h' '\12' '9' '\0'
char v2[] = "a\xah\127";      // 5 символов: 'a' '\xa' 'h' '\127' '\0'
char v3[] = "a\xad\127";      // 4 символа: 'a' '\xad' '\127' '\0'
char v4[] = "a\xad\0127";     // 5 символов: 'a' '\xad' '\012' '7' '\0'
```

В.3.3. Большие символьные наборы

Программа на C++ может быть написана и представлена пользователю в символьных наборах, гораздо более богатых, чем 127-символьный набор ASCII. Там, где реализация поддерживает большие наборы символов, в идентификаторах, комментариях, символьных константах и строках могут содержаться такие символы как å, β и Г. Однако из соображений переносимости реализация должна отображать эти символы в кодировку с использованием только тех символов, которые будут доступны любому пользователю C++. В принципе, этот перевод в основной набор символов для C++ (использующийся в данной книге) происходит до того, как компилятор производит какую-либо другую обработку. Поэтому отображение не влияет на смысл программы.

Стандартное кодирование символов из большого набора менее широким, непосредственно поддерживаемым C++, представляется последовательностью из четырех или восьми шестнадцатеричных цифр:

универсальное-имя-символа:
\U*XXXXXXXX*
\u*XXXX*

Здесь ***X*** представляет шестнадцатеричную цифру. Например, ***\u1e2b***. Более короткое обозначение ***\uXXXX*** равносильно ***\U0000XXXX***. Количество шестнадцатеричных цифр, отличное от четырех и от восьми, является лексической ошибкой.

Программист может прямо пользоваться этим кодированием символов. Однако в первую очередь оно задумано как способ для реализации, которая, пользуясь внутри себя маленьким набором символов, должна работать с большим набором, видимым программисту.

Если для использования расширенного набора символов в идентификаторах вы опираетесь на специальное окружение, программа становится менее переносимой. Если вы не понимаете использованного для идентификаторов и комментариев естественного языка, программу труднее читать. Поэтому для программ, использующихся в разных странах, обычно лучше придерживаться английского языка и ASCII.

В.3.4. Знаковые и беззнаковые символы

Имеет просто ***char*** знак, или не имеет, — это зависит от реализации. Таким образом открывается возможность для неприятных сюрпризов и зависимости от реализации. Например:

```
char c = 255;      // 255 — «все единицы», шестнадцатеричное 0xFF
int i = c;
```

Каково будет значение *i*? К сожалению, ответ не определен. Во всех известных мне реализациях он зависит от смысла «всех единиц» в битовом наборе ***char*** при расши-

рении в *int*. На машине SGI Challenge *char* беззнаковый, так что ответ будет *255*. На Sun SPARC или IBM PC, где *char* имеет знак, ответ будет *-1*. В этом случае компилятор может предупредить о преобразовании литерала *255* в значение *-1*. Однако C++ не предлагает универсального механизма для выявления подобных проблем. Одно из решений — избегать простого *char* и пользоваться только определенными (*signed* или *unsigned*) типами *char*. К несчастью, некоторые стандартные библиотечные функции, такие как *strcmp* (), принимают только просто *char* (§ 20.4.1).

Тип *char* должен вести себя одинаково по отношению к *signed char* и *unsigned char*. Однако все три типа *char* различаются, поэтому указатели на разные *char* смешивать нельзя. Например:

```
void f(char c, signed char sc, unsigned char uc)
{
    char* pc = &uc;              // ошибка: указатель не преобразуется
    signed char* psc = pc;       // ошибка: указатель не преобразуется
    unsigned char* puc = pc;     // ошибка: указатель не преобразуется
    psc = puc;                   // ошибка: указатель не преобразуется
}
```

Переменные трех типов *char* можно свободно присваивать одну другой. Однако присваивание слишком большого значения переменной *signed char* (§ В.6.2.1) по-прежнему не определено. Например:

```
void f(char c, signed char sc, unsigned char uc)
{
    c = 255;          // определяется реализацией, если просто char имеет знак
                      // и содержит 8 битов

    c = sc;           // правильно
    c = uc;           // определяется реализацией, если просто char имеет знак,
                      // а значение uc слишком велико
    sc = uc;          // определяется реализацией, если значение uc слишком велико
    uc = sc;          // правильно: преобразование в unsigned
    sc = c;           // определяется реализацией, если просто char не имеет знака,
                      // а значение c слишком велико
    uc = c;           // правильно: преобразование в unsigned
}
```

Ни одна из этих потенциальных проблем не возникнет, если вы везде будете пользоваться просто *char*.

В.4. Типы целых литералов

Как правило, тип целого литерала зависит от его формы, значения и суффикса:

- Если он десятичный и не имеет суффикса, то относится к первому из следующих типов, который окажется достаточным для представления его значения: *int*, *long int*, *unsigned long int*.
- Если он восьмеричный и не имеет суффикса, то относится к первому из следующих типов, который окажется достаточным для представления его значения: *int*, *unsigned int*, *long int*, *unsigned long int*.
- Если он имеет суффикс *u* или *U*, его тип — первый из двух, в котором можно представить его значение: *unsigned int*, *unsigned long int*.

- Если он имеет суффикс *l* или *L*, его тип — первый из двух, в котором можно представить его значение: ***long int***, ***unsigned long int***.
- Если он имеет суффикс ***ul***, ***lu***, ***uL***, ***Lu***, ***Ul***, ***lU***, ***UL*** или ***LU***, его тип — ***unsigned long int***.

Например, на машине с 32-разрядными целыми ***int*** число ***100000*** относится к типу ***int***, но на машине с 16-разрядными целыми и 32-разрядными ***long int*** оно относится к типу ***long int***. Аналогично ***0XA000*** относится к типу ***unsigned int*** на машинах с 32-разрядными целыми и к типу ***unsigned int*** на машинах с 16-разрядными целыми. Этих зависимостей от реализации можно избежать, используя суффиксы: ***100000L*** относится к типу ***long int*** на всех машинах, а ***0XA000U*** на всех компьютерах относится к ***unsigned int***.

B.5. Константные выражения

В таких местах как границы массива (§ 5.2), метки в инструкции ***case*** (§ 6.3.2) и инициализаторы перечислений (§ 4.8) C++ требует *константных выражений*. Константное выражение вычисляется в интегральную константу или константу перечисления. Такое выражение составляется из литералов (§ 4.3.1, § 4.4.1, § 4.5.1), элементов перечислений (§ 4.8) и констант, инициализированных константными выражениями. В шаблоне также можно использовать целый параметр шаблона (§ B.13.3). Литералы с плавающей точкой (§ 4.5.1) можно использовать, только если явно преобразовать их в интегральный тип. Как операнды в операторе ***sizeof*** (§ 6.2) только функции, объекты классов, указатели и ссылки можно использовать.

Интуитивно, константные выражения — это просто выражения, которые компилятор может вычислить до компоновки программы (§ 9.1) и ее запуска.

B.6. Неявное преобразование типов

Интегральные типы и типы с плавающей точкой (§ 4.1.1) в присваиваниях и арифметических выражениях можно свободно смешивать. Когда возможно, значения преобразуются так, чтобы не потерять информации. К сожалению, преобразования, приводящие к потере значения, также могут осуществляться неявно. В этом разделе дано описание правил преобразования, перечислены проблемы, связанные с преобразованием, и приведены их решения.

B.6.1. Продвижения

Неявные преобразования, сохраняющие значение, обычно называют *продвижениями*. Перед тем, как выполнить арифметическую операцию, используется *интегральное продвижение* — для того, чтобы создать переменные типа ***int*** из переменных более «коротких» целых типов. Отметим, что эти продвижения не продолжаются далее в ***long*** (если операнд не относится к типу ***wchar_t*** или не является перечислением, которое уже больше чем ***int***). В этом отражается изначальная цель подобных продвижений в C: привести операнды к «естественным» размерам для арифметических операций.

Интегральные продвижения таковы:

- ***char***, ***signed char***, ***unsigned char***, ***short int*** или ***unsigned short int*** преобразуются в ***int***, если ***int*** может представить все значения исходных типов; в противном случае они преобразуются в ***unsigned int***.

- *wchar_t* (§ 4.3) или типы перечислений (§ 4.8) преобразуются в первый из следующих типов, который сможет представить все значения их базовых типов: ***int***, ***unsigned int***, ***long*** или ***unsigned long***.
- Битовое поле (§ В.8.1) преобразуется в ***int***, если ***int*** может представить все значения битового поля; в противном случае оно преобразуется в ***unsigned int***, если ***unsigned int*** может представить все значения битового поля. В противном случае никакого целочисленного продвижения не применяется.
- ***bool*** преобразуется в ***int***; ***false*** становится ***0***, ***true*** становится ***1***.

Продвижения используются как часть обычных арифметических преобразований (§ В.6.3).

В.6.2. Преобразования

Фундаментальные типы могут преобразовываться один в другой невероятно большим количеством способов. По-моему, дозволяется слишком много преобразований. Например:

```
void f(double d)
{
    char c = d;        // осторожно: число с плавающей точкой двойной
                       // точности преобразуется в char
}
```

При написании программы вы всегда должны стремиться избегать неопределенного поведения и преобразований, которые незаметно «выбрасывают» информацию. Компилятор может предупредить о многих сомнительных преобразованиях — и к счастью, многие компиляторы действительно предупреждают.

В.6.2.1. Интегральные преобразования

Целое (*int*) может преобразовываться в другой целый тип. Значение перечислимого типа (*enum*) можно преобразовать в целый тип.

Если конечный тип беззнаковый, результирующее значение будет просто состоять из стольких битов, сколько уместится в конечном типе (при необходимости старшие разряды отбрасываются). Точнее говоря, результат будет наименьшим беззнаковым целым, соответствующим остатку от деления исходного целого числа на ***2*** в степени ***n***, где ***n*** — это число битов, применяемых для представления беззнакового типа. Например:

```
unsigned char uc = 1023;    // двоичное 111111111: uc становится двоичным
                            // 11111111, то есть 255
```

Когда конечный тип знаковый, значение остается неизменным, если его можно представить этим типом, в противном случае оно определяется в реализации.

```
signed char sc = 1023;      // определяется в реализации
```

Возможный результат — ***127*** или ***-1*** (§ В.3.4)

Логический тип или значение перечислимого типа могут неявно преобразовываться в свой целочисленный эквивалент (§ 4.2, § 4.8).

В.6.2.2. Преобразования чисел с плавающей точкой

Число с плавающей точкой может преобразовываться в число с плавающей точкой другого типа. Если исходное значение можно точно представить в конечном типе, результат будет изначальным численным значением. Если исходное значение лежит между двумя соседними значениями в конечном типе, результат будет одним из этих значений. В противном случае результат не определен. Например:

```
float f = FLT_MAX;          // наибольшее значение с плавающей точкой
double d = f;               // правильно: d == f
float f2 = d;               // правильно: f2 == f
double d3 = DBL_MAX;        // наибольшее значение double
float f3 = d3;              // не определено, если FLT_MAX<DBL_MAX
```

В.6.2.3. Преобразования указателей и ссылок

Любой указатель на объектный тип может быть неявно преобразован в ***void**** (§ 5.6). Указатель (ссылку) на производный класс можно неявно преобразовать в указатель (ссылку) на доступный и недвусмысленно определяемый базовый класс (§ 12.2). Отметим, что указатель на функцию или указатель на член не может быть неявно преобразован в ***void****.

Константное выражение (§ В.5), которое при своем вычислении равно ***0***, можно неявно преобразовать в любой указательный тип или тип указателя на член (§ 5.1.1). Например:

```
int* p =
          !      ! !     !  !      !
        ! !   !  !     !  !      !
       !    ! !   !     !  !      !
      !     !     !!!!!!  !!!!!  !!!!1;
```

T* может неявно преобразоваться в ***const T**** (§ 5.4.1). Аналогично, ***T***& может неявно преобразоваться в ***const T***&.

В.6.2.4. Преобразования указателей на члены

Указатели и ссылки на члены могут неявно преобразовываться, как описано в § 15.5.1.

В.6.2.5. Преобразования в логические переменные

Указатели, интегральные значения и числа с плавающей точкой могут неявно преобразовываться в логический тип ***bool*** (§ 4.2). Ненулевое значение преобразуется в ***true***, а нулевое — в ***false***. Например:

```
void f(int* p, int i)
{
    bool is_not_zero = p;       // true, если p!=0
    bool b2 = i;                // true, если i!=0
}
```

В.6.2.6. Преобразования чисел с плавающей точкой в целые и обратно

Когда число с плавающей точкой преобразуется в целое, его дробная часть отбрасывается. Иными словами, преобразование из числа с плавающей точкой в целое, урезает число. Например, значение ***int*** (***1.6***) равно ***1***. Если урезанное значение не может быть представлено конечным типом, поведение не определено. Например:

```
int i = 2.7;          // i становится равным 2
char b = 2000.7       // поведение не определено для 8-битных char:
                      // 2000 не представить 8 битами
```

Преобразование из целого типа в тип с плавающей точкой настолько математически корректно, насколько позволяет аппаратура. Потеря точности происходит, если целочисленное значение нельзя точно представить типом с плавающей точкой. Например,

```
int i = float(1234567890);
```

оставит *i* со значением ***1234567936***, если в машине и ***int***, и ***float*** представляются 32 битами.

Ясно, что потенциально разрушающих значение неявных преобразований лучше избегать. Фактически, компилятор может выявить некоторые очевидно опасные преобразования и предупредить о них — например, о преобразовании числа с плавающей точкой в целое или ***long int*** в ***char***. Однако полное выявление во время компиляции нереалистично, поэтому программист должен быть осторожен. Когда просто «быть осторожным» недостаточно, он должен вставлять явные проверки. Например:

```
class check_failed { };

char checked(int i);
{
    char c = i;          // опасно: не переносимо (§ B.6.2.1)
    if (i != c) throw check_failed();
    return c;
}

void my_code(int i)
{
    char c = checked(i);
    //...
}
```

Чтобы урезать число с гарантией переносимости, нужно воспользоваться ***numeric_limits*** (§ 22.2).

B.6.3. Обычные арифметические преобразования

Эти преобразования выполняются над операндами бинарного оператора, чтобы привести их к общему типу, который потом используется как тип результата:

[1] Если один из операндов относится к типу ***long double***, другой тоже преобразуется в ***long double***.
- В противном случае, если один из операндов относится к типу ***double***, другой тоже преобразуется в ***double***.
- В противном случае, если один из операндов относится к типу ***float***, другой тоже преобразуется в ***float***.
- В противном случае над обоими операндами производится интегральное продвижение (§ B.6.1).

[2] Затем, если один из операндов относится к типу ***unsigned long***, другой тоже преобразуется в ***unsigned long***.
- В противном случае, если один из операндов относится к типу ***long int***, а другой к типу ***unsigned int***, то если ***long int*** может представить все значения типа

unsigned int, *unsigned int* преобразуется в *long int*; иначе оба операнда преобразуются в *unsigned long int*.

- В противном случае, если один из операндов относится к типу *long*, другой тоже преобразуется в *long*.
- В противном случае, если один из операндов относится к типу *unsigned*, другой тоже преобразуется в *unsigned*.
- В противном случае оба операнда *int*.

В.7. Многомерные массивы

Не так уж редко нам бывает нужен вектор векторов, вектор векторов из векторов и т. д. Вопрос в том, как представляются такие многомерные вектора в C++. Здесь я сначала покажу, как пользоваться стандартным библиотечным классом *vector*. Потом я познакомлю вас с многомерными массивами, как они выглядят в программах на C и C++ при использовании только встроенных средств.

В.7.1. Вектора

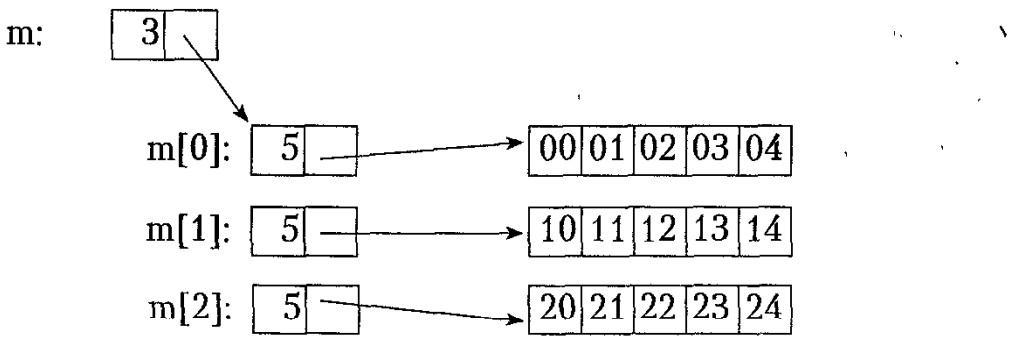
Стандартный *vector* (§ 16.3) предоставляет вполне общее решение:

```
vector<vector<int> > m(3, vector<int>(5));
```

Так создается вектор из трех векторов с пятью целыми элементами, в котором каждый из 15 целых элементов по умолчанию равен 0. Мы можем присвоить новые значения целым элементам следующим образом:

```
void init_m()
{
    for (int i = 0; i<m.size(); i++) {
        for (int j = 0; j<m[i].size(); j++) m[i][j] = 10*i+j;
    }
}
```

или графически:



Каждая реализация шаблона *vector* содержит указатель на его элементы плюс число элементов. Как правило, элементы содержатся в массиве. Для иллюстрации я задал каждому *int* начальное значение, представляющее его координаты.

Доступ к элементам осуществляется путем двойного индексирования. Например, *m*[*i*][*j*] — это *j*-й элемент *i*-го вектора. Мы можем распечатать *m* следующим образом:

```
void print_m()
{
```

```
        for (int i = 0; i<m.size (); i++) {
            for (int j = 0; j<m[i].size (); j++) cout << m[i][j] << '\t';
            cout << '\n';
        }
}
```

Получится:

```
0    1    2    3    4
10   11   12   13   14
20   21   22   23   24
```

Отметим, что *m* является вектором из векторов, в отличие от простого многомерного массива. Поэтому, в частности, возможно провести изменение размера (§ 16.3.8) элементов. Например:

```
void reshape_m (int_ ns)
{
    for (int i = 0; i<m.size (); i++) m[i].resize (ns);
}
```

Нет необходимости, чтобы вектора *vector<int>* в *vector< vector <int> >* имели одинаковый размер.

В.7.2 Массивы

Встроенные массивы являются главным источником ошибок — особенно когда они используются для построения многомерных массивов. Для новичков они также являются главным источником смущения и непонимания. По возможности пользуйтесь шаблонами *vector*, *valarray*, *string* и т. п.

Многомерные массивы представляются как массивы массивов. Массив 3×5 объявляется следующим образом:

```
int ma[3][5];       // 3 массива по 5 целых в каждом
```

Для массивов размерность должна задаваться как часть определения. Мы можем инициализировать *ma* следующим образом:

```
void init_ma ()
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) ma[i][j] = 10*i+j;
    }
}
```

или графически:

ma:

00	01	02	03	04	10	11	12	13	14	20	21	22	23	24

Массив *ma* — это просто 15 целых чисел, к которым мы обращаемся, как если бы здесь были 3 массива по 5 чисел. В частности, в памяти нет единого объекта, который был бы матрицей *ma* — хранятся только элементы. Размерности 3 и 5 существуют только в исходном коде. При написании программы наша обязанность — каким-то образом запомнить их. Например, мы могли бы распечатать *ma* следующим образом:

```
void print_ma ()
```

```
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) cout << ma[i][j] <<'\t';
        cout << '\n';
    }
}
```

Запятую, которая в некоторых языках используется для границ массива, в C++ применять нельзя, поскольку запятая (,) — это оператор последовательности (§ 6.2.2). К счастью, большинство ошибок вылавливается компилятором. Например:

```
int bad[3, 5];          // ошибка: в константных выражениях запятая не допускается
int good[3][5];         // 3 массива по 5 целых чисел в каждом
int ouch = good[1, 4];  // ошибка: int, инициализируется int*
                        // (good [1, 4] означает good[4], то есть int*)
int nice = good[1][4];
```

В.7.3. Передача многомерных массивов

Рассмотрим определение функции, которая бы манипулировала двумерной матрицей. Если размеры известны во время компиляции, проблем не возникает:

```
void print_m35 (int m[3][5])
{
    for (int i = 0; i<3; i++) {
        for (int j=0; j<5; j++) cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

Матрица, представленная как многомерный массив, передается как указатель (а не копируется, § 5.3). Первая размерность массива не влияет на проблему нахождения местоположения элемента; она просто говорит о том, сколько присутствует элементов (в данном случае 3) соответствующего типа (здесь ***int*[5]**). Например, взглянув на предыдущее представление ***ma***, мы заметим, что зная только вторую размерность 5 мы можем найти местоположение ***ma*[*i*][5]** для любого ***i***. Поэтому первую размерность можно передать в качестве аргумента:

```
void print_mi5 (int m[][5], int dim1)
{
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<5; j++) cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

Трудность возникает, когда нужно передать обе размерности. «Очевидное решение» просто не работает:

```
void print_mij (int m[][], int dim1, int dim2)    // работает не так, как
{                                                 // большинство людей могло бы подумать
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<dim2; j++) cout << m[i][j] << '\t';        // сюрприз!
        cout << '\n';
    }
}
```

Во-первых, объявление *m*[][] незаконно, так как чтобы найти элемент, должна быть известна вторая размерность двухмерного массива. Во-вторых, выражение *m*[*i*][*j*] интерпретируется (правильно) как * (* (*m*+*i*)+*j*), хотя это вряд ли то, чего хотел программист. Правильное решение таково:

```
void print_mij(int* m, int dim1, int dim2)
{
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<dim2; j++)
            cout << m[i*dim2+j] << '\t';          // не слишком ясно
        cout << '\n';
    }
}
```

Выражение для доступа к членам в ***print_mij*** () эквивалентно тому, что компилятор генерирует, когда ему известна последняя размерность.

Чтобы вызвать эту функцию, мы передаем матрицу как обычный указатель:

```
int main()
{
    int v[3][5] =
        { {0, 1, 2, 3, 4},
        {10, 11, 12, 13, 14},
        {20, 21, 22, 23, 24} };

    print_m35(v);
    print_mi5(v, 3);
    print_mij(&v[0][0], 3, 5);
}
```

Обратите внимание на использование &*v*[*0*][*0*] в последнем вызове; сработало бы и *v*[*0*], так как это равносильно, но *v* будет ошибкой типа. Такой топкий и довольно запутанный код лучше всего прятать. Если вам нужно работать напрямую с многомерными массивами, подумайте, как инкапсулировать соответствующий код, использующий их. Таким образом вы облегчите работу с вашей программой следующему программисту. Введение типа многомерного массива с соответствующим оператором индексации избавит многих пользователей от забот о расположении данных в массиве (§ 22.4.6).

Стандартный ***vector*** (§ 16.3) этими проблемами не страдает.

В.8. Экономия памяти

При написании нетривиальных программ часто наступает время, когда вам требуется памяти больше, чем имеется или чем это приемлемо. Есть два способа «выжимания» дополнительной памяти из имеющейся:

[1] Помещать в один байт несколько маленьких объектов.

[2] В разное время использовать одну и ту же область памяти для хранения разных объектов.

Первое достигается использованием *битовых полей (field)*, а последнее — *объединений (union)*. Эти конструкции описываются в последующих разделах. Многие применения полей и объединений являются чистой оптимизацией, которые часто основываются на непереносимых способах распределения памяти. Поэтому прежде чем их применять, программисту следует хорошенько подумать. Часто лучшим под-

ходом является изменение способа управления данными — например, больше применять динамическое распределение памяти (§ 6.2.6) и меньше пользоваться заранее распределяемой (статической) памятью.

B.8.1. Поля

Для представления бинарной переменной — например, для указания положения выключателя включен/выключен — представляется странным использовать целый байт (или ***char*** или ***bool***), но ***char*** в C++ — это самый маленький объект, под который можно выделить память, и который можно независимо адресовать (§ 5.1). Однако несколько таких крохотных переменных можно связать вместе как *поля* в структуре. Член определяется как поле указанием количества битов, которые оно занимает. Допускаются неименованные поля. Они не влияют на смысл именованных полей, но ими можно пользоваться, чтобы лучше расположить данные в машинной памяти (зависимым от машины образом):

```
struct PPN {                          // физический номер страницы для R6000
        unsigned int PFN : 22         // номер страничного кадра
        int : 3;                      // не используются
        unsigned int CCA : 3;         // алгоритм согласования с кэш-памятью
        bool nonreachable : 1;
        bool dirty : 1;
        bool valid : 1;
        bool global : 1;
};
```

Этот пример также иллюстрирует другое важное применение полей: именование частей внешне заданного размещения в памяти. Поле должно относиться к интегральному или перечислимому типу (§ 4.1.1). Для поля нельзя получить адрес. Однако в остальном полем можно пользоваться точно так же, как другими переменными. Отметим, что поле ***bool*** на самом деле может быть представлено одним битом. В ядре операционной системы или отладчиках тип PPN можно применить следующим образом:

```
void part_of_VM_system (PPN* p)
{
        //...
        if (p->dirty) {     // содержимое изменилось
                            // копирование на диск
                p->dirty = 0;
        }
        //...
}
```

Как это ни удивительно, использование полей для упаковки нескольких переменных в один байт не обязательно экономит память. Этим уменьшается область данных, но размер кода, манипулирующего такими переменными, на большинстве машин возрастает. Известно, что когда бинарные переменные преобразуются из битовых полей в символы, программы значительно сжимаются! Кроме того, обращение к ***char*** и ***int*** намного быстрее, чем к полю. Поля — это просто удобное средство для битовых логических операторов (§ 6.2.4), чтобы извлекать информацию из части слова и помещать ее туда.

В.8.2. Объединения

Объединение (*union*) — это структура (*struct*), в которой все члены расположены по одному и тому же адресу, так что все объединение занимает столько места, сколько занимает самый большой его член. Естественно, в данный момент времени объединение может хранить значение только одного из членов. Например, рассмотрим запись в таблице символов, содержащую имя и значение:

```
enum Type { S, I };

struct Entry {
        char* name;
        Type t;
        char* s;      // используем s, если t == S
        int i;        // используем i, если t == I
};

void f(Entry* p)
{
        if (p->t == S) cout << p->s;
        // ...
}
```

Членами *s* и *i* нельзя пользоваться одновременно, так что место под них тратится впустую. Этого можно просто избежать, описав обе переменные так, чтобы они стали членами объединения:

```
union Value {
        char* s;
        int i;
};
```

Язык не следит за тем, значения какого вида хранятся в объединении, так что программист должен по-прежнему писать так:

```
struct Entry {
        char* name;
        Type t;
        Value v;      // используем v.s, если t==S;
                      // используем v.i, если t == I
};

void f(Entry* p)
{
        if (p->t == S) cout << p->v.s;
        // ...
}
```

К сожалению, введение объединения вынуждает нас переписать код, чтобы заменить простое *s* на *v.s*. Этого можно избежать при помощи *анонимного объединения* — объединения, которое не имеет имени и следовательно не определяет типа. Вместо этого оно просто гарантирует, что все его члены располагаются по одному и тому же адресу:

```
struct Entry {
        char* name;
```

```
        Type t;
        union {
                char* s;        // используем s, если t == S
                int i;          // используем i, если t == I
        };
};

void f(Entry* p)
{
        if (p->t == S) cout << p->s;
        // ...
}
```

Так весь код, использующий ***Entry***, останется без изменений.

Применение объединения таким образом, что его значение всегда читается при помощи члена, через который он был записан — это чистая оптимизация. Однако не всегда легко гарантировать, что объединение используется только таким образом, и из-за этого могут возникать тонкие ошибки. Чтобы их избежать, объединение можно инкапсулировать, чтобы гарантировалось соответствие между полем типа и доступом к членам объединения (§ 10.6[20]).

Иногда объединения ошибочно используют для «преобразования типов». Такое главным образом практикуют программисты, работавшие с языками, не умеющими преобразовывать типы явно, и в которых приходится хитрить. Следующий пример демонстрирует, как ***int*** «преобразуется» в ***int****, просто исходя из побитовой эквивалентности:

```
union Fudge {           // вздор!
        int i;
        int *p;
};

int* cheat (int i)      // жульничество
{
        Fudge a;
        a.i = i;
        return a.p;     // плохо
}
```

На самом деле это никакое не преобразование. На некоторых машинах ***int*** и ***int**** занимают разный объем памяти, а на других ***int*** не может иметь нечетный адрес. Такое применение объединений опасно и непереносимо, а для преобразования типов есть явный и переносимый способ (§ 6.2.7).

Иногда объединения преднамеренно используются для того, чтобы избежать преобразования типов. Например, ***Fudge*** можно использовать для того, чтобы определить представление нулевого указателя:

```
int main ()
{
        Fudge foo;
        foo.p = 0;
        cout << "целое значение указателя 0 равно " << foo.i << '\n';
}
```

В.8.3. Объединения и классы

Многие нетривиальные объединения имеют некоторые члены, которые значительно больше, чем часто используемые члены. Поскольку размеры объединения по крайней мере равны размерам наибольшего члена, память расходуется зря. Часто этого можно избежать, пользуясь вместо объединений производными классами.

Класс с конструктором, деструктором или операцией копирования не может быть типом члена объединения (§ 10.4.12), так как компилятор не может узнать, какой член уничтожать.

В.9. Управление памятью

В C++ есть три основных способа использования памяти:

Статическая память, в которую компоновщик помещает объект на все время выполнения программы. В статической памяти располагаются глобальные переменные и переменные из пространств имен, статические члены классов (§ 10.2.4) и статические переменные из функций (§ 7.1.2). Объект, размещаемый в статической памяти, конструируется один раз и сохраняется до окончания программы. Он всегда имеет один и тот же адрес. Статические объекты могут вызвать проблемы в программах, использующих потоки (при параллелизме с разделяемым адресным пространством), поскольку такие объекты используются совместно и для правильного доступа к ним требуется блокировка.

Автоматическая память, в которой располагаются аргументы функций и локальные переменные. Каждый блок или функция получает свою копию. Такая память автоматически создается и уничтожается, отсюда и название — автоматическая память. Ее также называют «памятью в стеке». Если вы хотите соблюсти абсолютную ясность, C++ предоставляет необязательное ключевое слово ***auto***.

Свободная память, которую явно требует программа при размещении объектов, и которую она может освободить после того, как память больше не нужна (при помощи ***new*** и ***delete***). Когда программе требуется еще свободной памяти, ***new*** запрашивает ее у операционной системы. Как правило, свободная память (также называемая *динамической памятью* или *кучей*) за время жизни программы разрастается, поскольку память никогда не возвращается операционной системе для использования другими программами.

Насколько это касается программиста, автоматическая и статическая память используется просто, очевидно и неявно. Представляет интерес, как распределять свободную память. Выделение памяти (при помощи ***new***) просто, но если мы не имеем последовательной политики возвращения памяти диспетчеру свободной памяти, вся память заполнится — особенно в долго работающих программах.

Простейшая стратегия — для управления соответствующими объектами в свободной памяти пользоваться автоматическими объектами. Поэтому многие контейнеры реализуются как обработчики элементов, хранящихся в свободной памяти (§ 25.7). Например, автоматическая строка ***String*** (§ 11.12) управляет последовательностью символов в свободной памяти и автоматически освобождает эту память, когда выходит из области видимости. Все стандартные контейнеры (§ 16.3, главы 17 и 20, § 22.4) удобно реализовать таким образом.

В.9.1. Автоматическая сборка мусора

Когда регулярного подхода недостаточно, программист может воспользоваться диспетчером памяти. Этот диспетчер находит объекты, к которым программа больше не может обратиться, и забирает их память обратно, чтобы можно было хранить в ней другие объекты. Обычно это называют *автоматической сборкой мусора* или просто *сборкой мусора*. Естественно, такой диспетчер памяти называют *сборщиком мусора*.

Основная идея сборки мусора заключается в том, что объект, на который больше нет ссылок в программе, больше не доступен, и занимаемую им память можно безопасно использовать под какие-то другие объекты. Например:

```
void f()
{
        int* p = new int;
        p = 0;
        char* q = new char;
}
```

Здесь присваивание **p=0** оставляет целую переменную без ссылки, так что ее память можно использовать под какой-нибудь другой объект. Таким образом, следующую переменную **char** можно расположить в той же памяти, так что **q** будет иметь то же значение, что первоначально имело **p**.

Стандарт не требует, чтобы реализация обеспечивала сборщик мусора, но сборщики мусора все больше используются в C++ в областях, где они выгоднее ручного управления свободной памятью. При сравнении затрат учитывайте время выполнения, занимаемую память, надежность, переносимость, денежную стоимость программирования, денежную стоимость сборщика мусора и предсказуемость производительности.

В.9.1.1. Замаскированные указатели

На объект нет ссылок — что это должно означать? Рассмотрим пример:

```
void f()
{
        int* p = new int;
        long i1 = reinterpret_cast<long> (p)&0xFFFF0000;
        long i2 = reinterpret_cast<long> (p)&0x0000FFFF;
        p = 0;
        // точка #1: указателя на целую переменную здесь нет

        p = reinterpret_cast<int*> (i1|i2);
        // теперь снова появляется ссылка на целую переменную
}
```

Указатели, хранящиеся в программе как не указатели, часто называют «замаскированными указателями». В частности, указатель, первоначально находящийся в **p**, замаскирован в целых **i1** и **i2**. Однако замаскированные указатели не должны беспокоить сборщик мусора. Если сборщик мусора дойдет до точки **#1**, память, хранящую целую переменную, можно взять обратно. Фактически, такие программы не могут гарантированно работать даже без использования сборщика мусора, поскольку применение **reinterpret_cast** для преобразования целых в указатели в лучшем случае определяется в реализации.

Объединение, которое может содержать и указатели, и не указатели, представляет особую проблему для сборщика мусора. Вообще говоря, невозможно знать, содержит такое объединение указатель или нет. Рассмотрим:

```
union U {          // объединение с членами-указателями и не указателями
        int* p;
        int i;
};

void f(U u, U u2, U u3)
{
        u.p = new int;
        u2.i = 999999;
        u.i = 8;
        //...
}
```

Безопасное предположение состоит в том, что любое значение, которое появляется в таком объединении, — это значение указателя. Умный сборщик мусора может сделать кое-что получше. Например, он может заметить, что (в данной реализации) целые переменные не располагаются по нечетным адресам, и что никакие объекты не располагаются по таким маленьким адресам, как *8*. Заметив это, сборщик мусора не предположит, что объекты, содержащие адреса *999999* и *8*, используются функцией *f*().

B.9.1.2. delete

Если реализация предоставляет автоматическую уборку мусора, операторы *delete* и *delete*[] больше не нужны, чтобы освобождать память для потенциального повторного использования. Таким образом, пользователь, опирающийся на сборку мусора, может просто отказаться от использования этих операторов. Однако кроме высвобождения памяти *delete* и *delete*[] вызывают деструкторы.

При наличии сборщика мусора

```
delete p;
```

вызывает деструктор для объекта, на который указывает *p* (если он есть). Однако повторное использование памяти может быть отложено до тех пор, пока она не будет собрана. Возвращение памяти из под многих объектов сразу помогает ограничить фрагментацию (С.9.1.4). Также в важных случаях, когда деструктор просто уничтожает выделенную память, сборщик мусора делает безвредной ошибку по удалению одного объекта дважды, которая в противном случае является довольно серьезной.

Как всегда, доступ к объекту после его уничтожения не определен.

B.9.1.3. Деструкторы

Когда сборщик мусора собирается вернуть память, занимаемую объектом, есть две альтернативы:

[1] Вызвать для объекта деструктор (если таковой есть).

[2] Отнестись к объекту, как к «сырой» памяти (не вызывать его деструктор).

По умолчанию сборщик мусора должен выбрать вариант 2, так как объекты, созданные при помощи *new* и не удаленные при помощи *delete*, никогда не уничтожаются.

Таким образом, сборщик мусора можно рассматривать как механизм имитации бесконечной памяти.

Можно спроектировать сборщик мусора так, чтобы он вызывал деструктор для объектов, которые специально «зарегистрированы» сборщиком. Однако нет стандартного способа «регистрации» объектов. Отметим, что всегда важно уничтожать объекты в таком порядке, чтобы гарантировать, что деструктор для одного объекта не ссылается на объект, который уже был удален. Этого нелегко добиться от сборщика мусора без помощи со стороны программиста.

В.9.1.4. Фрагментация памяти

Когда память выделяется и освобождается для множества объектов разного размера, она *фрагментируется*. То есть большая часть памяти разбивается на кусочки, которые слишком малы, чтобы их эффективно использовать. Причина в том, что универсальный распределитель памяти не всегда может найти для объекта область памяти в точности нужного размера. Использовать область чуть побольше — значит оставить кусочек памяти неиспользованным. Через некоторое время работы программы с простым распределителем памяти не так уж редко оказывается, что половина доступной памяти занята фрагментами, слишком маленькими для повторного использования.

Для борьбы с фрагментацией существует несколько приемов. Простейший из них — запрашивать у распределителя только большие куски памяти и использовать каждый кусок для объектов одного размера (§ 15.3, § 19.4.2). Поскольку большинство размещений и переразмещений производится для маленьких объектов таких типов, как узлы деревьев, связи и т. п., этот прием может оказаться очень эффективным. Иногда распределитель памяти может применять схожие приемы автоматически. В обоих случаях фрагментация еще более снижается, если все относительно большие «куски» имеют один размер (скажем, в одну страницу), так что их самих можно размещать и переразмещать без фрагментации.

Существует два стиля сборщиков мусора:

[1] *Копирующий сборщик* для сжатия фрагментированной области перемещает объекты в памяти.

[2] *Консервативный сборщик* размещает объекты так, чтобы минимизировать фрагментацию.

С точки зрения C++ предпочтительнее консервативные сборщики, так как очень трудно (вероятно, в реальных программах и невозможно) переместить объект и правильно модифицировать все указатели на него. Кроме того консервативный сборщик позволяет фрагментам программы на C++ сосуществовать с программами, написанными на таких языках, как C. Традиционно копирующие сборщики пользовались популярностью среди тех, кто программировал на языках, имеющих дело с объектами не непосредственно, а через уникальные указатели и ссылки (такими языками являются, например, Lisp и Smalltalk). Однако для относительно больших программ, где объем копирования и взаимодействие между распределителем памяти и страничной системой приобретают большую важность, современные консервативные сборщики, похоже, не менее эффективны, чем копирующие. Для программ поменьше часто достижим просто идеальный вариант, когда сборщик мусора никогда не вызывается, — особенно на C++, где многие объекты естественным образом являются автоматическими.

В.10. Пространства имен

Этот раздел знакомит вас с подробностями, относящимися к пространствам имен, которые выглядят техническими мелочами, однако в дискуссиях и реальных программах часто всплывают на поверхность.

В.10.1. Удобство против безопасности

using-объявление добавляет имя к локальной области видимости. ***using-директива*** не добавляет имени, она просто делает доступными имена в той области видимости, в которой она указана. Например:

```
namespace X {
        int i, j k;
}

int k;

void f1 ()
{
        int i = 0;
        using namespace X;      // делает имена из X доступными
        i++;                    // локальная i
        j++;                    // X::j
        k++;                    // ошибка: X::k или глобальная k?
        ::k++;                  // глобальная k
        X::k++;                 // k из X
}

void f2 ()
{
        int i = 0;
        using X::i;             // ошибка: i объявлено в f2() дважды
        using X::j;
        using X::k;             // скрывает глобальную k

        i++;
        j++;                    // X::j
        k++;                    // X::k
}
```

Локально объявленное имя (объявленное или обычным объявлением или ***using-объявлением***) скрывает нелокальные объявления с тем же именем, и в точке объявления выявляются все незаконные перегрузки этого имени.

Обратите внимание на ошибку неоднозначности для ***k++*** в ***f1*** (). Глобальные имена не имеют приоритета перед именами из пространств имен, ставшими доступными в глобальной области видимости. Этим обеспечивается значительная защищенность от случайных конфликтов имен и — что важно — гарантируется отсутствие каких-либо выгод от замусоривания глобального пространства имен.

Когда библиотека, объявляющая много имен, делается доступной посредством ***using-директив***, очень важным достоинством является то, что конфликты неиспользуемых имен не рассматриваются как ошибки.

Глобальная область видимости — это просто еще одно пространство имен. Оно отличается лишь тем, что вам нет необходимости упоминать его имя явно. То есть ::*k* означает «искать *k* в глобальном пространстве имен и в пространствах имен, упомянутых в *using-директивах* в глобальном пространстве имен», в то время как *X*::*k* означает «*k*, объявленное в пространстве имен *X* или пространствах имен, упомянутых в *using-директивах* в *X*» (§ 8.2.8).

Надеюсь, что в новых программах с использованием пространств имен я увижу радикальное снижение использования глобальных имен по сравнению с традиционными программами на C и C++. Правила работы с пространствами имен разрабатывались специально для того, чтобы «ленивые» любители глобальных имен не получали никаких преимуществ перед теми, кто старается не замусоривать глобальную область видимости.

B.10.2. Вложение пространств имен

Одним из очевидных примеров использования пространств имен является помещение полного набора объявлений и определений в отдельное пространство имен:

```
namespace X {
    // все мои объявления
}
```

Список объявлений в общем случае будет содержать пространства имен. Таким образом возникают вложенные пространства имен. Это допускается из практических соображений, а также из той простой идеи, что конструкциям следует быть вложенными, если только нет сильных аргументов против этого. Например:

```
void h();

namespace X {
    void g();
    //...
    namespace Y {
        void f();
        void ff();
        //...
    }
}
```

Здесь применяются обычные правила видимости и квалификации:

```
void X::Y::ff()
{
    f(); g(); h();
}

void X::g()
{
    f();          // ошибка: в X нет f()
    Y::f();       // правильно
}
```

```
void h ()
{
        f ();          // ошибка: нет глобальной f()
        Y::f ();       // ошибка: нет глобального Y
        X::f ();       // ошибка: в X нет f()
        X::Y::f ();    // правильно
}
```

В.10.3. Пространства имен и классы

Пространство имен — это именованная область видимости. Класс — это тип, определенный именованной областью видимости, которая описывает, как создаются и используются объекты данного типа. Таким образом, пространство имен — это более простое понятие, нежели класс, и было бы идеально определить класс как пространство имен, включающее несколько дополнительных возможностей. Это почти что так. Пространство имен открыто (§ 8.2.9.3), но класс закрыт. Это различие проистекает из того соображения, что классу нужно определять расположение объектов, а это лучше делать в одном месте. Кроме того, *using-объявления* и *using-директивы* применимы к классам только очень ограниченным способом (§ 15.2.2).

Пространства имен предпочтительнее классов, когда требуется только инкапсулировать имена. В этом случае аппарат классов для проверки типов и создания объектов не нужен; достаточно более простой концепции пространств имен.

В.11. Контроль доступа

В этом разделе приведено несколько технических примеров, иллюстрирующих контроль доступа в дополнение к примерам из § 15.3.

В.11.1. Доступ к членам

Рассмотрим следующий фрагмент кода:

```
class X {
// по умолчанию private
        int priv;
protected:
        int prot;
public:
        int publ;
        void m ();
};
```

Член *X::m* имеет неограниченный доступ:

```
void X::m ()
{
        priv = 1;    // правильно
        prot =2;     // правильно
        publ = 3;    // правильно
}
```

Члены производного класса имеют доступ к защищенным и открытым членам (§ 15.3):

```
class Y: public X{
        void mderived ();
};

void Y::mderived ()
{
        priv = 1;    // ошибка: priv — закрытый член
        prot = 2;    // правильно: prot — защищенный член, а mderived() —
                     // член производного класса Y
        prot = 3;    // правильно: publ — это открытый член
}
```

Глобальная функция может обращаться только к открытым членам:

```
void f(Y* p)
{
        p->priv = 1;     // ошибка: priv — закрытый член
        p->prot = 2;     // ошибка: priv — защищенный член, а f() —
                         // не друг и не член класса X или Y
        p->publ = 3;     // правильно: publ — это открытый член
}
```

В.11.2. Доступ к базовым классам

Как и члены, базовый класс можно объявить закрытым (***private***), защищенным (***protected***) или открытым (***public***). Рассмотрим пример:

```
class X{
public:
        int a;
        //...
};

class Y1 : public X{};
class Y2 : protected X{};
class Y3 : private X{};
```

Поскольку ***X*** является для ***Y1*** открытым базовым классом, любая функция может (неявно) преобразовывать ***Y1**** в ***X****, там, где это нужно, и обращаться к открытым членам класса ***X***. Например:

```
void f(Y1* py1, Y2* py2, Y3* py3)
{
        X* px = py1;     // правильно: X для Y1 — это открытый базовый класс
        py1->a = 7;      // правильно

        px = py2;        // ошибка: X для Y2 — защищенный базовый класс
        py2->a = 7;      // ошибка

        px = py3;        // ошибка: X для Y3 — закрытый базовый класс
        py3->a = 7;      // ошибка
}
```

Рассмотрим следующий фрагмент:

```
class Y2 : protected X { };
class Z2 : public Y2 { void f(Y1*, Y2*, Y3*); };
```

Поскольку ***X*** — это защищенный базовый класс для ***Y2***, только члены и друзья ***Y2***, а также члены и друзья производных от ***Y2*** классов (например, ***Z2***), могут (неявно) обращаться к открытым и защищенным членам класса ***X***. Например:

```
void Z2::f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;      // правильно: X — открытый базовый класса для Y1
    py1->a = 7;       // правильно

    px = this;        // правильно: X — защищенный базовый класс для Y2,
                      // а Z2 — производный класс от Y2
    a = 7;            // правильно

    px = py2;         // ошибка: X — защищенный базовый класс для Y2,
                      // а Z2 — производный класс от Y2, но нам не известно,
                      // что py2 принадлежит к Z2 или
                      // как Y2::X используется в объектах, не принадлежащих к Z2
    py2->a = 7;       // ошибка: Z2 не знает, как Y2::a используется в объектах,
                      // не принадлежащих к Z2

    px = py3;         // ошибка: X — это закрытый базовый класс для Y3
    py3->a = 7;       // ошибка
}
```

И наконец рассмотрим:

```
class Y3 : private X { void f(Y1*, Y2*, Y3*); };
```

Поскольку ***X*** является для ***Y3*** закрытым базовым классом, преобразовывать (неявно) ***Y3**** в ***X**** могут только члены и друзья ***Y3***; они также могут обращаться к открытым и защищенным членам класса ***X***. Например:

```
void Y3::f(Y1* py1; Y2* py2, Y3* py3)
{
    X* px = py1;      // правильно: X — открытый базовый класс для Y1
    py1->a = 7;       // правильно

    px = py2;         // ошибка: X — защищенный базовый класс для Y2
    py2->a = 7;       // ошибка

    px = py3;         // правильно: X — это закрытый базовый класс для Y3,
                      // а Y3::f — член класса Y3
    py3->a = 7;       // правильно
}
```

В.11.3. Доступ из членов-классов

Члены класса-члена не имеют особого доступа к членам внешнего класса. Аналогично, члены внешнего класса не имеют особого доступа к членам вложенного класса; нужно соблюдать обычные правила доступа (§ 10.2.2). Например:

```
class Outer {                    // внешний
    typedef int T;
    int i;
public:
    int i2;
```

```
    static int s;
    class Inner {              // внутренний
        int x;
        T y;                   // ошибка: Outer::T — закрытый
    public:
        void f(Outer* p, int v);
    };
    int g(Inner* p);
};
void Outer::Inner::f(Outer* p, int v)
{
    p->i = v;                  // ошибка: Outer::i — закрытый
    p->i2 = v;                 // правильно: Outer::i2 — открытый
}

int Outer::g(Inner* p)
{
    p->f(this, 2);             // правильно: Inner::f() — открытый
    return p->x;               // ошибка: Inner::x — закрытый
}
```

Однако часто полезно позволить классу-члену обращаться к его внешнему классу. Этого можно добиться, сделав член другом (*friend*). Например:

```
class Outer {
    typedef int T;
    int i;
public:
    class Inner;               // предварительное объявление класса члена
    friend class Inner;        // разрешаем доступ к Outer::Inner

    class Inner {
        int x;
        T y;                   // правильно: Inner — друг
    public:
        void f(Outer* p, int v);
    };
};
void Outer::Inner::f(Outer* p, int v)
{
    p->i = v;                  // правильно: Inner — друг
}
```

В.11.4. Дружба

Дружба не наследуется и не транзитивна. Например:

```
class A {
    friend class B;
    int a;
};
class B {
    friend class C;
};
```

```
class C {
        void f(A* p)
        {
                p->a++;     // ошибка: C — не друг класса A, несмотря на то,
                            // что приходится другом другу класса A
        }
};
class D : public B {
        void f(A* p)
        {
                p->a++;     // ошибка: D — не друг класса A, несмотря на то,
                            // что является производным от друга класса A
        }
};
```

В.12. Указатели на члены данных

Естественно, понятие указателя на член (§ 15.5) применяется к членам данных и к функциям-членам с аргументом и типом возвращаемого значения. Например:

```
struct C {
        const char* val;
        int i;
        void print(int x) { cout << val << x << '\n'; }
        void f1(int);
        int f2();
        C(const char* v) { val = v; }
};

typedef void (C::*PMFI)(int);       // указатель на функцию-член класса C
                                    // с целым аргументом
typedef const char* C::*PM;         // указатель на член данных класса C
                                    // типа char*

void f(C& z1, C& z2)
{
        C* p = &z2;
        PMFI pf = &C::print;
        PM pm = &C::val;

        z1.print(1);
        (z1.*pf)(2);
        z1.*pm = "nv1";
        p->*pm = "nv2";
        z2.print(3);
        (p->*pf)(4);

        pf = &C::f1;        // ошибка: несоответствие типа возвращаемого значения
        pf = &C::f2;        // ошибка: несоответствие типа аргумента
        pm = &C::i;         // ошибка: несоответствие типа
        pm = pf;            // ошибка: несоответствие типа
}
```

Тип указателя на функцию проверяется точно так же, как любой другой тип.

В.13. Шаблоны

Шаблон класса описывает, как сгенерировать класс по заданному подходящему набору аргументов шаблона. Аналогично, шаблон функции описывает, как сгенерировать функцию по соответствующему набору аргументов шаблона. Таким образом, шаблоны можно использовать для генерирования типов и исполнимого кода. Вместе с такой выразительной мощью возникает некоторая сложность. Большая ее часть относится к разнообразию контекстов определения и использования шаблонов.

В.13.1. Статические члены

Шаблон класса может иметь статические (*static*) члены. Каждый класс, сгенерированный по шаблону, имеет собственную копию статических членов. Статические члены должны определяться отдельно и могут быть специализированы. Например:

```
template<class T> class X {
        //...
        static T def_val;
        static T* new_X(T a = def_val);
};

template<class T> T X<T>::def_val(0, 0);
template<class T> T* X<T>::new_X(T a) { /* ... */ }

template<> int X<int>::def_val<int> = 0;
template<> int* X<int>::new_X<int>(int i) { /* ... */ }
```

Если вы хотите сделать объект или функцию общими для всех членов любого сгенерированного по шаблону класса, вы можете поместить их в нешаблонный базовый класс. Например:

```
struct B {
        static B* nil;      // должен использоваться как общий нулевой указатель
                            // во всех классах, производных от B
}

template<class T> class X : public B {
        //...
};

B* B::nil = 0;
```

В.13.2. Друзья

Как и другие классы, шаблоны классов могут иметь друзей. Рассмотрим примеры ***Matrix*** и ***Vector*** из § 11.5. Обычно как ***Matrix***, так и ***Vector*** будут шаблонами:

```
template<class T> class Matrix;

template<class T> class Vector {
        T v[4];
public:
        friend Vector operator*<>(const Matrix<T>&, const Vector&);
        //...
};
```

```
template<class T> class Matrix {
        Vector <T> v[4];
public:
        friend Vector<T> operator*<> (const Matrix&, const Vector<T>&);
        // ...
};
```

Угловые скобки <> после имени функции-друга указывают на то, что речь идет о шаблонной функции. Без этих скобок будет подразумеваться нешаблонная функция. Определенный ниже оператор умножения имеет прямой доступ к данным классов ***Vector*** и ***Matrix***:

```
template<class T> Vector<T> operator* (const Matrix<T>& m, const Vector<T>& v)
{
        // ... использует m.v[i] и v.v[i] для прямого доступа к элементам
}
```

Друзья не влияют на область видимости, в которой определен шаблонный класс; также они не влияют на область видимости, где этот шаблон используется. Вместо этого функции-друзья и операторы-друзья находятся путем поиска, основанного на типе аргумента (§ 11.2.4, § 11.5.1). Как и функции-члены, функции-друзья инстанцируются (§ В.13.9.1) только при их вызове.

В.13.3 Шаблоны как параметры шаблона

Иногда в качестве аргументов шаблона полезно передавать шаблоны — а не классы или объекты. Например:

```
template<class T, template<class> class C> class Xrefd {
        C<T> mems;
        C<T*> refs;
        // ...
};

Xrefd<Entry, vector> x1; // хранит перекрестные ссылки для записей (Entry) в векторе
Xrefd<Record, set> x2; // хранит перекрестные ссылки для записей (Record) во множестве set
```

Чтобы объявить шаблон как параметр шаблона, мы должны определить требующиеся ему аргументы. Например, мы определяем, что ***C***, являющийся параметром шаблона ***Xref***, — это класс-шаблон с одним аргументом-типом. Если мы не сделаем этого, мы не сможем использовать специализации параметра ***C***. Точка использования шаблона в качестве параметра шаблона обычно является тем местом, где мы хотим выполнить его инстанцирование различными типами аргументов (такими как ***T*** и ***T**** в предыдущем примере). То есть мы хотим выразить объявления членов шаблона в терминах другого шаблона, но нам нужно, чтобы другой шаблон был параметром, который может задаваться пользователем.

В обычном случае, когда шаблону нужен контейнер, чтобы хранить элементы типа его аргумента, лучше передавать тип контейнера (§ 13.6, § 17.3.1).

Аргументами шаблонов могут быть только шаблоны классов.

В.13.4. Выведение аргументов шаблона функции

Компилятор может сам вывести (логически определить) тип аргумента шаблона, ***T*** или ***TT***, и аргумент шаблона, не являющийся типом, ***I***, из аргумента шаблона функции с типом, составленным из следующих конструкций:

T	*const T*	*volatile T*
*T**	*T*&	*T*[*constant_expression*]
type[*I*]	*class_template_name*<*T*>	*class_template_name*<*I*>
TT<*T*>	*T*<*I*>	*T*<>
T type::*	*T T*::*	*type T*::*
T(*) (*args*)	*type* (*T*::*) (*args*)	*T* (*type*::*) (*args*)
type (*type*::*) (*args_TI*)	*T* (*T*::*) (*args_TI*)	*type* (*T*::*) (*args_TI*)
T (*type*::*) (*args_TI*)	*type* (*) (*args_TI*)	

Здесь ***args_TI*** — это список параметров, по которому можно определить ***T*** или ***I*** при помощи рекурсивного применения этих правил, а ***args*** — это список параметров, которые не допускают выведения. Если не все параметры можно вывести таким образом, вызов неоднозначен. Например:

```
template<class T, class U> void f(const T*, U (*) (U));

int g(int);

void h(const char* p)
{
    f(p, g);    // T — это char, U — int
    f(p, h);    // ошибка: невозможно вывести U
}
```

Глядя на аргумент первого вызова *f*(), мы легко выводим аргументы шаблона. Глядя на второй вызов *f*(), мы видим, что ***h*** не соответствует образцу ***U*** (*) (***U***), поскольку типы аргумента функции ***h*** () и возвращаемого ей значения различаются.

Если параметр шаблона можно вывести из более чем одного аргумента функции, результатом каждого выведения должен быть один и тот же тип. Иначе вызов будет ошибочным. Например:

```
template<class T> void f(T i, T* p);

void g(int i)
{
    f(i, &i);          // правильно
    f(i, "Помни!");    // ошибка, двусмысленность: T — это int или const char?
}
```

В.13.5. typename и шаблоны

Чтобы сделать обобщенное программирование более простым и общим, контейнеры стандартной библиотеки предоставляют набор стандартных функций и типов (§ 16.3.1). Например:

```
template<class T> class vector {
public:
    typedef T* iterator;

    iterator begin();
    iterator end();

    //...
}
```

```
template<class T> class list {
        class link { /* ... */};
public:
        typedef link* iterator;

        iterator begin ();
        iterator end ();

        // ...
};
```

Это искушает нас написать:

```
template<class C> void f (C& v)
{
        C::iterator i = v.begin ();          // синтаксическая ошибка
        // ...
}
```

К сожалению, от компилятора нельзя требовать здравого смысла, и он не понимает, что ***C::iterator*** — это имя типа. В некоторых случаях «умный» компилятор мог бы догадаться: понимается ли имя как наименование типа, или же чего-нибудь, что типом не является (подобно функции или шаблону). Но в общем случае это невозможно. Рассмотрим пример, в котором нет явных подсказок о его смысле:

```
int y;

template<class T> void g (T& v)
{
        T::x (y);       // вызов функции или объявление переменной?
}
```

T::x — это функция, вызываемая с нелокальной переменной ***y*** в качестве аргумента? Или мы объявляем переменную ***y*** типа ***T::x*** и, извращаясь, поставили необязательные скобки? Можно представить программу, в которой ***X::x*** (***y***) окажется вызовом функции, а ***Y::x*** (***y***) — объявлением.

Разрешение производится просто: если не указано обратного, идентификатор считается относящимся к чему-то такому, что не является типом или шаблоном. Если мы хотим сказать, что с чем-то нужно обращаться как с типом, мы можем воспользоваться ключевым словом ***typename***:

```
template<class C> void h (C& v)
{
        typename C::iterator i = v.begin ();
        // ...
}
```

Чтобы заявить о том, что именованная сущность — это тип, перед квалифицированным именем можно поставить ключевое слово ***typename***. В этом отношении ***typename*** напоминает ***struct*** и ***class***.

Ключевое слово ***typename*** необходимо, если имя типа зависит от параметров шаблона. Например:

```
template<class T>
void k (vector<T> & v)
```

```
{
    vector<T>:: iterator i=v.begin (); // синтаксическая ошибка: отсутствует typename
    typename vector<T>:: iterator i=v.begin (); // правильно
    // ...
}
```

В этом случае компилятору возможно удастся определить, что ***iterator*** является именем типа в каждом экземпляре типа ***vector***, но от компилятора этого не требуется. Такое действие является нестандартным и непереносимым расширением языка. Единственный контекст, в котором компилятор может предположить, что имя, зависящее от аргумента шаблона, является именем типа,— это те несколько случаев, когда только имена типов разрешены грамматикой. Например, это имеет место в *спецификаторе-базы* § А.8.1).

Ключевое слово ***typename*** можно также использовать как альтернативу слову ***class*** в объявлении шаблона. Например:

```
template<typename T> void f(T);
```

Лично я неважно печатаю, и мне вечно не хватает места на экране, поэтому я предпочитаю более короткое:

```
template<class T> void f(T);
```

В.13.6. template как квалификатор

Необходимость в квалификаторе ***typename*** возникает потому, что мы можем обращаться и к членам, которые являются типами, и к членам, которые таковыми не являются. Аналогично может возникнуть необходимость различать имя члена шаблона от других имен членов. Рассмотрим возможный интерфейс к универсальному диспетчеру памяти:

```
class Memory { // некоторый распределитель памяти (Allocator)
public:
    template<class T> T* get_new ();
    template<class T> void release (T&);
    // ...
}

template<class Allocator> void f(Allocator& m)
{
    int* p1 = m.get_new<int> (); // синтаксическая ошибка:
                                 // int после оператора «меньше»
    int* p2 = m.template get_new<int> (); // явная квалификация
    // ...
    m.release (p1); // аргумент шаблона выводится,
                    // явная квалификация не нужна
    m.release (p2);
}
```

Явная квалификация ***get_new*** () необходима потому, что ее параметр шаблона не может быть выведен. В таком случае нужно использовать префикс ***template***, чтобы сообщить компилятору (и читающему программу человеку), что ***get_new*** — это шаблон

члена, так что возможна явная квалификация желаемым типом элемента. Без квалификации с ключевым словом ***template*** мы получим синтаксическую ошибку, поскольку символ < будет воспринят как оператор «меньше». Необходимость в квалификации с ключевым словом ***template*** возникает редко, потому что большинство параметров шаблона выводится.

В.13.7. Инстанцирование

Имея определение шаблона и его использование, генерация правильного кода становится задачей реализации языка. Из шаблона-класса и набора аргументов шаблона компилятору нужно сгенерировать определение класса и определения его используемых функций-членов. Из шаблонов функций нужно сгенерировать функции. Этот процесс обычно называют *инстанцированием шаблона.*

Сгенерированные классы и функции называются *специализациями.* Когда нужно различать сгенерированные специализации и специализации, написанные программистом явно (§ 13.5), их называют соответственно *сгенерированными специализациями* и *явными специализациями.* Явные специализации иногда также называют *специализациями, определяемыми пользователем,* или просто *пользовательскими.*

Чтобы использовать шаблоны в нетривиальных программах, программист должен понимать, как имена, используемые в определении шаблона, связываются с объявлениями, и как можно организовать исходный код (§ 13.7).

По умолчанию компилятор генерирует классы и функции из используемых шаблонов по правилам связывания имен (§ В.13.8). То есть программист не обязан явно указывать, какие версии каких шаблонов нужно сгенерировать. Это важно, потому что программисту нелегко точно понять, какие версии каких шаблонов нужны. Часто в реализациях библиотек используются шаблоны, о которых он даже не слышал, а иногда знакомые программисту шаблоны используются с аргументами шаблона неизвестного типа. Как правило, набор функций, которые нужно сгенерировать, можно узнать только рекурсивной проверкой шаблонов, используемых в прикладных программах. Для такого анализа лучше приспособлены компьютеры, а не люди.

Однако иногда программисту важно иметь возможность точно указать, где из шаблона должен генерироваться код(§ В.13.10). Сделав это, программист получает полный контроль над контекстом инстанцирования. В большинстве окружений компиляции это также подразумевает контроль над тем, когда именно производится инстанцирование. В частности, можно воспользоваться явным инстанцированием, чтобы получать ошибки компиляции в предсказуемое время, а не тогда, когда реализация определит, что нужно сгенерировать специализацию. Для некоторых пользователей необходим хорошо предсказуемый процесс построения программы.

В.13.8. Связывание имен

Важно определить шаблоны функций так, чтобы они имели как можно меньше зависимостей от нелокальной информации. Дело в том, что потом шаблон будет использоваться для генерирования функций и классов, основываясь на неизвестных типах и в неизвестном контексте. Всякая тонкая зависимость от контекста, скорее всего, всплывет для какого-нибудь программиста в виде проблемы при отладке — и вряд ли он захочет разбираться в деталях реализации шаблона. К универсальному правилу —

избегать по мере возможности глобальных имен — нужно с особой серьезностью отнестись в коде шаблона. Таким образом мы стараемся сделать определения шаблонов по мере возможности самодостаточными и превратить максимум из того, что в противном случае стало бы глобальным контекстом, в форму параметров шаблона (например, свойств; § 13.4, § 20.2.1).

Однако нужно использовать и некоторые нелокальные имена. В частности, чаще всего пишут набор взаимодействующих между собой шаблонов функций, а не один самодостаточный шаблон. Иногда такие взаимодействующие функции могут быть членами класса, но не всегда. Иногда лучшим решением являются нелокальные функции. Типичный пример этого — вызов в функциях ***sort*** () функций ***swap*** () и ***less*** () (§ 13.5.2). Много соответствующих примеров предоставляют алгоритмы стандартной библиотеки (глава 18).

Другой источник использования нелокальных имен при определении шаблона — операции с общепринятыми именами и смыслом, такие как + , *, [] и ***sort*** (). Рассмотрим пример:

```
#include<vector>

bool tracing;

//...

template<class T> T sum (std::vector<T>& v)
{
        T t = 0;
        if (tracing) cerr << "sum (" << &v << ")\n";
        for (int i = 0; i<v.size (); i++) t = t + v[i];
        return t;
}

//...

#include<quad.h>

void f (std::vector<Quad>& v)
{
        Quad c = sum (v);
}
```

Невинный с виду шаблон функции ***sum*** () зависит от оператора +. В данном примере + определен в <***quad.h***>:

```
Quad operator+ (Quad, Quad);
```

Важно то, что когда определяется ***sum*** (), в области видимости нет ничего, относящегося к ***Quad***-числам, и нельзя предполагать, что автор функции ***sum*** () знает о классе ***Quad***. В частности, оператор + может быть определен после ***sum*** () в тексте программы, и даже позже по времени.

Процесс поиска объявлений для каждого имени, явно или неявно используемого в шаблоне, называется *связыванием имен*. Главная проблема со связыванием имен шаблона состоит в том, что в инстанцировании шаблона замешаны три контекста, которые нельзя четко разделить:

[1] контекст определения шаблона;
[2] контекст объявления типа аргумента;
[3] контекст использования шаблона.

В.13.8.1. Зависимые имена

При определении шаблона функции нам нужно убедиться, что контекста достаточно, чтобы определение шаблона имело смысл в терминах его фактических аргументов, не захватывая «случайный» мусор из окружения в точке использования. Чтобы помочь этому, язык разделяет используемые в определении шаблона имена на две категории:

[1] Имена, зависящие от аргумента шаблона. Такие имена связываются в некоторой точке инстанцирования (§ В.13.8.3). В примере с функцией ***sum*** () определение оператора + можно найти в контексте инстанцирования, поскольку она принимает операнды типа аргумента шаблона.

[2] Имена, не зависящие от аргументов шаблона. Такие имена связываются в точке определения шаблона (§ В.13.8.2). В примере с ***sum*** () шаблон ***vector*** определен в заголовочном файле <***vector***>, и когда компилятор встречает определение ***sum*** (), логическая переменная ***tracing*** находится в области видимости.

Простейшим определением «***N*** зависит от параметра шаблона ***T***» было бы «***N*** является членом ***T***». К сожалению, этого не вполне достаточно; сложение чисел с квадратичной точностью ***Quad*** (§ В.13.8) является примером обратного. Поэтому говорят, что вызов функции *зависит от* аргумента шаблона, если и только если выполняется одно из следующих условий:

[1] Тип фактического аргумента зависит от параметра шаблона ***T*** согласно правилам выведения типа (§ 13.3.1). Например, *f*(***T***(*1*)), *f*(*t*), *f*(*g*(*t*)) и *f*(&*t*), считая, что *t* — это ***T***.

[2] Вызываемая функция имеет формальный параметр, который зависит от ***T*** по правилам выведения типа (§ 13.3.1). Например, *f*(***T***), *f*(***list***<***T***>&) и *f*(***const T****).

В основном имя вызываемой функции зависимо, если оно очевидно зависимо при просмотре ее аргументов или формальных параметров.

Вызов, который случайно имеет аргумент, соответствующий фактическому типу параметра шаблона, не является зависимым. Например:

```
template<class T> T f(T a)
{
    return g(1);        // ошибка: в области видимости нет g(), и g(1) не зависит от T
}

void g(int);

int z = f(2);
```

Неважно, что при вызове *f*(*2*) типом ***T*** оказался ***int***, и аргумент *g*() тоже оказался ***int***. Если бы мы рассматривали *g*(*1*) как зависимый, его смысл для читающего определение шаблона был бы абсолютно непонятным и таинственным. Если программист хочет, чтобы *g*(***int***) вызывалась, определение *g*(***int***) должно располагаться до определения *f*(), так чтобы при анализе *f*() определение *g*(***int***) было в области видимости. Это то же самое правило, что и для определения функций, не являющихся шаблонами.

В добавление к именам функций имя переменной, типа, *const* и т. д. может быть зависимым, если их тип зависит от параметра шаблона. Например:

```
template<class T> void fct(const T& a)
{
    typename T::Memtype p = a.p;        // p и Memtype зависят от T
    cout << a.i << ' ' << p->j;         // i и j зависят от T
}
```

В.13.8.2. Связывание в точке определения

Когда компилятор видит определение шаблона, он решает, какие имена являются зависимыми (§ В.13.8.3). Если имя зависимо, поиск его объявления нужно отложить до инстанцирования (§ В.13.8.3).

Имена, не зависящие от аргумента шаблона, должны находиться в области видимости (§ 4.9.4) в точке определения. Например:

```
int x;

template<class T> T f(T a)
{
        x++;                    // правильно
        y++;                    // ошибка: в области видимости нет y,
                                // и y не зависит от T
        return a;
}

int y;

int z = f(2);
```

Если объявление найдено, это объявление используется, даже если впоследствии может найтись объявление «получше». Например:

```
void g(double);

template<class T> class X : public T {
public:
        void f() { g(2); }      // вызов g(double);
        // ...
};

void g(int);

class Z { };

void h(X<Z> x)
{
        x.f();
}
```

Когда генерируется определение для `X<Z>::f()`, `g(int)` не рассматривается, поскольку она объявлена после `X`. Не важно, что `X` используется после объявления `g(int)`. Также вызов, не являющийся зависимым, нельзя «угонять» в базовый класс:

```
class Y { public: void g(int); };

void h(X<Y> x)
{
        x.f();
}
```

И снова `X<Y>::f()` вызовет `g(double)`. Если бы программист хотел вызвать `g()` из базового класса `T`, об этом следовало сказать в определении `f()`:

```
template<class T> class XX: public T {
        void f() { T::g(2); }        // вызывает T::g()
        // ...
};
```

Тут, конечно, применяется эмпирическое правило, что определение шаблона должно быть настолько самодостаточно, насколько это возможно.

В.13.8.3. Связывание в точке инстанцирования

Каждое применение шаблона для данного набора аргументов шаблона определяет точку инстанцирования. Это точка находится в ближайшей глобальной области видимости или области видимости пространства имен, охватывающей ее использование, прямо перед объявлением, содержащим это использование. Например:

```
struct X { X(int); /* ... */ };
void g(X);
template<class T> void f(T a) { g(a); }
void h()
{
        extern void g(int);
        f(2);                        // вызов f(X(2)); эквивалентно f<X>(X(2))
}
```

Здесь точка инстанцирования для *f* находится прямо перед **h**(), так что функция **g**(), вызываемая в *f*(), является глобальной **g**(**X**), а не локальной **g**(**int**). Определение «точки инстанцирования» подразумевает, что параметр шаблона никогда не может быть связан с локальным именем или членом класса. Например:

```
void f()
{
        struct X { /* ... */ };      // локальная структура
        vector<X> v;                 // ошибка: нельзя в качестве параметра
                                     // шаблона использовать локальную структуру
        // ...
}
```

Также неквалифицированное имя, используемое в шаблоне, не может быть связано с локальным именем. И наконец, даже если шаблон впервые используется внутри класса, используемые в шаблоне неквалифицированные имена не будут связаны с членами этого класса. Игнорирование локальных имен существенно для предотвращения многих макросоподобных неприятностей. Например:

```
template<class T> void sort(vector<T>& v)
{
        sort(v.begin(), v.end());    // используется sort()
                                     // из стандартной библиотеки
                                     // (причем явно не указывается std::)
}

class Container {
        vector<int> v;               // элементы
public:
        void sort()          // сортировка элементов
```

```
        {
            ::sort(v);    // sort(vector<int>&) вызывает std::sort(), а не Container::sort()
        }
        //...
};
```

Пусть ***sort*** (***vtctor<T>***&) вызывает ***sort*** (),используя ***std***::***sort*** (). Результат был бы тем же, а код был бы прозрачнее.

Если точка инстанцирования для шаблона, определенного в пространстве имен, находится в другом пространстве имен, для связывания доступны имена из обоих пространств. Как всегда, для выбора между именами из разных пространств имен используется разрешение перегрузки (§ 8.2.9.2).

Отметим, что шаблон, используемый несколько раз с одним и тем же набором аргументов, имеет несколько точек инстанцирования. Если связывания независимых имен различаются, программа неверна. Однако эту ошибку трудно выявить в реализации, особенно если точки инстанцирования находятся в разных единицах трансляции. При связывании имен лучше избегать сложностей, минимизируя применение в шаблонах нелокальных имен и пользуясь заголовочными файлами, чтобы контексты использования были согласованы.

В.13.8.4. Шаблоны и пространства имен

Когда вызывается функция, ее объявление может быть найдено, даже если его нет в области видимости, при условии, что она объявлена в том же пространстве имен, что и один из ее аргументов (§ 8.2.6). Это очень важно для функций, вызываемых в определениях шаблонов, поскольку при помощи этого механизма во время инстанцирования находятся зависимые функции.

Специализация шаблона может быть сгенерирована в любой точке инстанцирования (§ В.13.8.3), в любой точке, следующей за ней в единице трансляции, или в единице трансляции, специально созданной для генерации специализаций. Это отражает три очевидных стратегии, которые могут использоваться в реализации для генерации специализаций:

[1] Генерировать специализацию в первый же раз, когда встретится вызов.
[2] В конце единицы трансляции генерировать все специализации, нужные для этой единицы трансляции.
[3] Когда все единицы трансляции просмотрены, генерировать все специализации, нужные для программы.

Все три стратегии имеют свои достоинства и недостатки; также возможно сочетание этих стратегий.

В любом случае связывание независимых имен производится в точке определения шаблона. Связывание зависимых имен производится путем просмотра:

[1] имен в области видимости в точке, где определяется шаблон;
[2] имен в пространстве имен аргумента зависимого вызова (считая, что глобальные функции находятся в пространстве имен встроенных типов).

Например:

```
namespace N{
    class A{ /* ... */ };

    char f(A);
}
```

```
char f(int);

template<class T> char g(T t) { return f(t); }

char c = g(N::A());        // приводит к вызову N::f(N::A)
```

Здесь вызов `f(t)` очевидно зависим, так что в точке определения мы не можем связать `f` с `f(N::A)` или с `f(int)`. Чтобы сгенерировать специализацию для `g<N::A>(N::A)`, реализация просматривает пространство имен `N` в поисках функций с именем `f()` и находит `N::f(N::A)`.

Программу следует считать неправильной, если, выбрав разные точки инстанцирования или разные контексты пространств имен в разных возможных контекстах для генерирования специализации, можно получить две разные функции. Например:

```
namespace N {
    class A { /* ... */ };

    char f(A, int);
}

template<class T, class T2> char g(T t, T2 t2) { return f(t, t2); }

char c = g(N::A(), 'a');        // ошибка (возможны разные разрешения f(t))

namespace N {                   // добавление в пространство имен N (§ 8.2.9.3)
    void f(A, char);
}
```

Мы могли бы сгенерировать специализацию в точке инстанцирования и получить вызов `f(N::A, int)`. Или могли бы подождать, сгенерировать специализацию в конце единицы трансляции и получить вызов `f(N::A, char)`. Следовательно, вызов `g(N::A(), 'a')` является ошибочным.

Это очень плохое программирование — вызывать перегруженную функцию между двумя ее объявлениями. Просматривая большую программу, программист не ожидает встретить здесь подвох. В данном конкретном случае компилятор мог бы выловить неоднозначность. Однако схожие проблемы могут возникать в разных единицах трансляции, и тогда выявление ошибки становится гораздо труднее. Реализация не обязана вылавливать подобные ошибки.

Большинство проблем с неоднозначным разрешением вызова функции связано со встроенными типами. Поэтому большинство средств лечения опирается на более осторожное использование аргументов, относящихся к встроенным типам.

Встроенные типы не имеют ассоциированного пространства имен. Следовательно, зависимое разрешение имен не обеспечивает переопределение разрешения между объявлениями, видимыми до и после определения шаблона. Например:

```
int f(int);

void ff(int);
void ff(char);

template T g(T t) { ff(t); return f(t); }

char f(char);

char c = g('a');        // вызов ff(char); вызов f(int) - f(char) не рассматривается
```

Очевидно, подобных хитростей лучше избегать.

В.13.9. Когда нужна специализация?

Специализацию шаблона класса нужно генерировать, только если возникает потребность в определении класса. В частности, чтобы объявить указатель на какой-либо класс, определения класса не нужно. Например:

```
class X;
X* p;          // правильно: определения X не нужно
X a;           // ошибка: требуется определение X
```

При определении шаблонов классов это различие может оказаться критическим. Шаблон класса *не* инстанцируется, пока действительно не понадобится его определение. Например:

```
template<class T> class Link {
        Link* suc;      // правильно: определения Link не нужно (пока)
        //...
};

Link<int>* pl;          // инстанцирования Link<int> не требуется

Link<int> lnk;          // теперь Linl<int> нужно инстанцировать
```

Точка инстанцирования находится там, где впервые понадобилось определение.

В.13.9.1. Инстанцирование шаблонов функций

Реализация инстанцирует шаблон функции только если эта функция используется. В частности, инстанцирование шаблона класса не влечет инстанцирования всех его членов, или даже всех членов, определенных в объявлении шаблона класса. Это позволяет программисту иметь достаточную гибкость при определении шаблона класса. Например:

```
template<class T> class List {
        //...
        void sort ();
};

class Glob { /* нет операторов сравнения */ };

void f(List<Glob>& lb, List<string>& ls)
{
        ls.sort ();
        // используем операции над lb, но не lb.sort()
}
```

Здесь инстанцируется ***List<string>::sort*** (), но не ***List<Glob>::sort*** (). Это и уменьшает объем генерируемого кода, и избавляет нас от необходимости перепроектировать программу. Если бы ***List<Glob>::sort*** () было сгенерировано, нам пришлось бы либо добавить в ***Glob*** операции, требуемые ***List::sort*** (), заместить ***sort*** () так, чтобы она не была функцией-членом ***List***, либо использовать для объектов ***Glob*** какой-либо другой контейнер.

В.13.10. Явное инстанцирование

Запрос на явное инстанцирование — это объявление специализации после ключевого слова *template* (за которым не следует символ <):

```
template class vector<int>;                    // класс
template int& vector<int>::operator[] (int);   // член
template int convert<int, double> (double);    // функция
```

Объявление шаблона начинается с ***template<***, в то время как просто ***template*** означает начало запроса на инстанцирование. Отметим, что ***template*** пишется перед полным объявлением; простого написания имени не достаточно:

```
template vector<int>::operator[];              // синтаксическая ошибка
template convert<int, double>;                 // синтаксическая ошибка
```

Как и при вызове шаблонов функций, аргументы шаблона, которые можно вывести, исходя из аргументов функций, можно опустить (§ 13.3.1). Например:

```
template int convert<int, double> (double);    // правильно (избыточно)
template int convert<int> (double);            // правильно
```

Когда шаблон класса явно инстанцируется, инстанцируются и все его функции-члены.

Обратите внимание, что явным инстанцированием можно пользоваться как проверкой ограничений. Например:

```
template<class T> class Calls_foo {
    void constraints (T t) { foo (t); }        // вызывается из каждого конструктора
    // ...
};

template class Calls_foo<int>;                 // ошибка: foo(int) не определена
template void Calls_foo<Shape*>::constraints (Shape*);  // ошибка: foo(Shape*)
                                               // не определена
```

Запросы на инстанцирование значительно влияют на время компоновки и эффективность повторной компиляции. Я знаю примеры, когда помещение большей части инстанцирований шаблонов в одну единицу трансляции сокращало время компиляции с нескольких часов до такого же количества минут.

Иметь два определения для одной специализации — это ошибка. Не важно, является ли такая множественная специализация определяемой пользователем (§ 13.5), сгенерированной неявно (§ В.13.7) или явно запрошенной. Однако компилятор не обязан выявлять множественное инстанцирование в отдельных единицах компиляции. Это позволяет «умным» реализациям игнорировать избыточные инстанцирования и таким образом избегать проблем, связанных с составлением программы из библиотек с использованием явного инстанцирования (§ В.13.7). Однако реализации не обязаны быть «умными». Пользователи «менее умных» реализаций должны избегать множественного инстанцирования. Однако самое плохое, что может случиться, если они все-таки применят ее,— программа просто не загрузится; никакого изменения смысла «в тихую» не произойдет.

Язык не требует от пользователя явного инстанцирования. Явное инстанцирование — это необязательный механизм оптимизации и управления процессом компиляции и компоновки «вручную» (§ В.13.7).

В.14. Советы

[1] Фокусируйтесь на разработке программного продукта, а не на технических деталях; § В.1.

[2] Даже следование стандарту не гарантирует переносимости; § В.2.

[3] Избегайте неопределенного поведения (включая внутрифирменные расширения); § B.2.
[4] Локализуйте поведение, определяемое в реализации; § B.2.
[5] В системах, где отсутствуют {, }, [,], | или ! для представления программ пользуйтесь ключевыми словами и диграфами, а там где отсутствует \ применяйте триграфы; § B.3.1.
[6] Чтобы облегчить общение, для представления программ пользуйтесь символами ANSI; § B.3.3.
[7] Численному представлению символов предпочитайте escape-символы; § B.3.2.
[8] Не полагайтесь на наличие или отсутствие знака у простого ***char***; § B.3.4.
[9] При сомнении относительно типа целого литерала пользуйтесь суффиксом; § B.4.
[10] Избегайте неявных преобразований типа, приводящих к потере значения; § B.6.
[11] Предпочитайте вектора ***vector*** массивам; § B.7.
[12] Избегайте объединений; § B.8.2.
[13] Для представления заданного расположения в памяти пользуйтесь битовыми полями; § B.8.1.
[14] Помните об альтернативах при выборе между разными стилями управления памятью; § B.9.
[15] Не засоряйте глобальное пространство имен; § B.10.1
[16] Там, где нужна область видимости (модуль), а не тип, выбирайте ***namespace***, а не ***class***; § B.10.3.
[17] Не забывайте определять статические члены шаблонов классов; § B.13.1.
[18] Чтобы устранить неоднозначность в типе членов параметра шаблона, пользуйтесь ключевым словом ***typename***; § B.13.5.
[19] Там, где необходима явная квалификация аргументами шаблона, чтобы устранить неоднозначность в членах шаблона класса, пользуйтесь ключевым словом ***template***; § B.13.6.
[20] Пишите определения шаблонов с минимальной зависимостью от контекста их инстанцирования; § B.13.8.
[21] Если инстанцирование шаблонов отнимает слишком много времени, попробуйте инстанцировать их явно; § B.13.10.
[22] Если порядок компиляции должен быть абсолютно предсказуем, попробуйте использовать явное инстанцирование; § B.13.10.

Приложение Г

Локализация

В чужой монастырь
со своим уставом не ходят.

— Пословица

Учет культурных различий — класс ***locale*** — именованные локализации — создание локализаций — копирование и сравнение локализаций — локализации ***global***() и ***classic***() — сравнение строк — класс ***facet*** — доступ к фасетам локализаций — простой определяемый пользователем фасет — стандартные фасеты — сравнение строк — ввод/вывод чисел — ввод/вывод денег — ввод/вывод дат и времени — низкоуровневые операции со временем — класс ***Date*** — классификация символов — преобразование кода символа — каталоги сообщений — советы — упражнения.

Г.1. Учет культурных различий

Объект ***locale*** (локализация) определяет набор культурных (национальных) особенностей, таких как способ сравнения строк, удобный для чтения формат вывода чисел и представление символов во внешней памяти. Понятие локализации расширяемо, так что программист может добавлять в локализацию новые фасеты[1] (***facet***) для отражения местных особенностей, не поддерживаемых стандартной библиотекой непосредственно, как, например, почтовые индексы и телефонные номера. Основной задачей использования локализаций в стандартной библиотеке является управление отображением информации, записываемой в поток вывода (***ostream***), и форматами, воспринимаемыми потоком ввода (***istream***).

В разделе § 21.7 описано, как можно изменить локализацию для потока. В данном приложении рассказывается, как локализация конструируется из фасетов, и объясняются механизмы, посредством которых локализация воздействует на свой поток. В приложении речь также идет об определении фасета, перечисляются стандартные фасеты, определяющие специфические свойства потока, изложены технологии реализации и использования локализаций и фасетов. При описании ввода/вывода

[1] Слово «фасет» означает грань или аспект чего-либо. В контексте информационных технологий термин «фасет» означает совокупность сведений, которую при классификации удобно рассматривать как единое целое. Заинтересованный читатель может отыскать дальнейшие сведения о фасетной классификация информации в Интернете или в специализированной литературе. — *Примеч. ред.*

дат рассматриваются средства стандартной библиотеки для представления даты и времени.

Обсуждение локализаций и их фасетов организовано следующим образом:

§ Г.1. Вводит основные идеи представления культурных различий с помощью локализаций.

§ Г.2. Описывает класс ***locale*** (локализация).

§ Г.3. Описывает класс ***facet*** (фасет локализации).

§ Г.4. Содержит обзор стандартных фасетов с указанием деталей каждого из них:

- § Г.4.1. Сравнение строк.
- § Г.4.2. Ввод и вывод числовых значений.
- § Г.4.3. Ввод и вывод денежных значений.
- § Г.4.4. Ввод и вывод дат и времени.
- § Г.4.5. Классификация символов.
- § Г.4.6. Преобразования кодов символов.
- § Г.4.7. Каталоги сообщений.

Понятие локализации не является «собственностью» C++ — в большинстве операционных систем и сред разработки приложений имеются подобные понятия. В принципе, локализация используется в системе всеми программами совместно, независимо от того, на каком языке программирования они написаны. Таким образом, идею локализации стандартной библиотеки можно рассматривать в качестве стандартного и переносимого способа доступа из программ на C++ к информации, имеющей существенно различное представление в различных системах. Среди прочего, класс ***locale*** C++ является общим интерфейсом к системной информации, представленной несовместимым образом в различных системах.

Г.1.1. Программирование культурных различий

Подумаем о написании программы, которой придется пользоваться в нескольких странах. Стиль написания подобных программ часто называют «интернационализацией» (делая акцент на использовании программы во многих странах) или «локализацией» (упирая на адаптацию программы к местным условиям). Многие объекты, которыми манипулирует программа, в различных странах принято выводить по-разному. Мы можем написать процедуры ввода/вывода с учетом этих требований. Например:

```
void print_date(const Date& d)      // вывод в подходящем формате
{
    switch(where_am_I) {            // определенный пользователем индикатор стиля
    case DK:                        // например, 7. marts 1999
        cout << d.day() << ". " << dk_month[d.month()] << " " << d.year();
        break;
    case UK:                        // например, 7 / 3 / 1999
        cout << d.day() << " / " << d.month() << " / " << d.year();
        break;
    case US:                        // например, 3/7/1999
        cout << d.month() << "/" << d.day() << "/" << d.year();
        break;
    // ...
    }
}
```

Такой подход решает задачу. Однако это скверный стиль, причем нам придется последовательно его придерживаться, чтобы корректно адаптировать весь вывод к местным особенностям. Хуже того, если мы захотим добавить новый способ записи даты, придется модифицировать код. Казалось бы, с проблемой можно справиться путем создания иерархии классов (§ 12.2.4). Однако сама информация в ***Date*** не зависит от того, в каком виде мы хотим ее видеть. Поэтому нам не подходит иерархия типов ***Date***, наподобие ***US_Date***, ***UK_Date*** и ***JP_Date***. Мы хотели бы располагать разнообразными способами вывода дат: «вывод в американском стиле», «вывод в английском стиле». «вывод в японском стиле» и т. д.; см. § Г.4.4.5.

Подход «пусть пользователь пишет функции ввода/вывода, учитывающие культурные различия» порождает и другие трудности:

[1] Без помощи стандартной библиотеки программист, пишущий приложение, не может изменить «внешний вид» встроенных типов достаточно простым, переносимым и эффективным способом.

[2] Поиск каждой операции ввода/вывода (и каждой операции, готовящей данные для ввода/вывода с учетом местных особенностей) в большой программе не всегда осуществим.

[3] Иногда у нас нет возможности переписать программу с учетом новых соглашений, но даже если бы и была, мы бы предпочли иное решение, не требующее переписывания кода.

[4] Слишком накладно заставлять всех пользователей разрабатывать и реализовывать учет культурных особенностей.

[5] Различные программисты по-разному будут реализовывать низкоуровневые национальные особенности, и программы, работающие с одной и той же информацией, будут отличаться без существенных на то причин. Таким образом, программистам, сопровождающим код из различных источников, придется изучить множество программных соглашений. Довольно утомительно и сопряжено с ошибками.

Стандартная библиотека предоставляет расширяемый способ учета национальных соглашений. На эту схему работы опирается библиотека потоков ввода/вывода (§ 21.7), обрабатывая как встроенные, так и определяемые пользователем типы. Например, рассмотрим простой цикл копирования пар (***Date***, ***double***), которые могут представлять собой последовательные измерения или транзакции:

```
void cpy(istream& is, ostream& os)      // копирование потока пар (дата, число)
{
    Date d;
    double volume;

    while (is >> d >> volume) os << d << ' ' << volume << '\n';
}
```

Само собой разумеется, реальная программа выполнила бы какие-нибудь операции над записями и, в идеале, хоть чуть-чуть позаботилась бы об обработке ошибок.

Как заставить эту программу прочитать файл, соответствующий французским соглашениям (для отделения дробной части используется запятая; например, ***12,5*** означает «двенадцать с половиной»), и переписать его в американском формате? Мы можем определить локализации и операции ввода/вывода так, чтобы функцию ***cpy()*** можно было применить для преобразования между двумя форматами:

```
void f(istream& fin, ostream& fout, istream& fin2, ostream& fout2);
{     fin.imbue(locale("en_US"));          // американский английский
      fout.imbue(locale("fr"));            // французский
      cpy(fin, fout);                      // чтение на американском английском,
                                           // запись на французском

      fin2.imbue(locale("fr"));            // французский
      fout2.imbue(locale("en_US"))         // американский английский
      cpy(fin2, fout2);                    // чтение на французском,
                                           // запись на американском английском
}
```

Пусть имеются потоки:

```
Apr 12, 1999      1000.3
Apr 13, 1999      345.45
Apr 14, 1999      9688.321
...
3 juillet 1950    10,3
3 juillet 1951    134,45
3 juillet 1952    67,9
...
```

на выходе программы будет:

```
12 avril 1999     1000,3
13 avril 1999     345,45
14 avril 1999     9688,321
...
July 3, 1950      10.3
July 3, 1951      134.45
July 3, 1952      67.9
...
```

Оставшаяся часть данного приложения в основном описывает механизмы, делающие возможной подобную схему работы, и объясняет, как их применять. Отметим, что у большинства программистов нет ни малейших причин разбираться в деталях локализаций. Многие программисты никогда явно не будут работать с ***locale***, а большинство из тех, что все-таки будут, просто извлекут стандартную локализацию и закрепят (imbue) ее за потоком (§ 21.7). Механизмы создания локализаций и обеспечения простоты их использования представляют из себя небольшой язык программирования.

Если программа или система оказались успешны, они будут использованы людьми с потребностями и склонностями, которые проектировщики и программисты не предвидели. Большинство успешных программ будут исполнятся в странах, где (естественные) языки и наборы символов отличаются от тех, что знакомы проектировщикам и программистам. Широкое использование программы является признаком успеха, так что проектирование и программирование, нацеленные на преодоление языковых и культурных барьеров, являются подготовкой к успеху.

Концепция локализации (интернационализации) проста. Но практические ограничения изрядно запутывают проектирование и реализацию ***locale***:

[1] В локализации инкапсулированы культурные особенности, такие как форма отображения дат. Подобные соглашения неуловимым и несистематическим об-

разом отличаются друг от друга. Эти правила не имеют ничего общего с языками программирования, поэтому язык программирования не может их стандартизовать.

[2] Концепция локализации должна быть расширяемой, потому что невозможно перечислить все особенности, присущие разным культурам, и представляющие интерес для различных пользователей C++.

[3] Локализация ***locale*** используется в операциях ввода/вывода, от которых требуют эффективности на этапе исполнения.

[4] Локализация ***locale*** должна быть невидима для большинства программистов, желающих воспользоваться «правильно работающим» потоком ввода/вывода, и не желающих разбираться в том, что это значит и как достигается.

[5] Локализация ***locale*** должна быть доступна разработчикам средств, зависящих от культурных особенностей, помимо библиотеки потоков ввода/вывода.

Проектирование программы, осуществляющей ввод/вывод, связано с выбором между управлением форматированием при помощи «обычного кода» или посредством локализаций. Первый (традиционный) подход приемлем в случаях, когда есть уверенность, что каждая операция ввода может быть легко преобразована из одной системы соглашений в другую. Однако, если требуется изменить формы отображения встроенных типов, или нужны различные наборы символов, или приходится иметь в виду расширяемый набор связанных со вводом-выводом соглашений, механизм ***locale*** выглядит более привлекательно.

Локализация (***locale***) состоит из фасетов (***facet***), управляющих отдельными соглашениями, такими как символ-разделитель при отображении чисел с плавающей точкой (***decimal_point***(); § Г.4.2) и формат, используемый при чтении денежных величин (***moneypunct***; § Г.4.3). Фасет является объектом класса, производного от класса ***locale***::***facet*** (§ Г.3). Мы можем рассматривать ***locale*** как контейнер фасетов ***facet*** (§ Г.2, § Г.3.1).

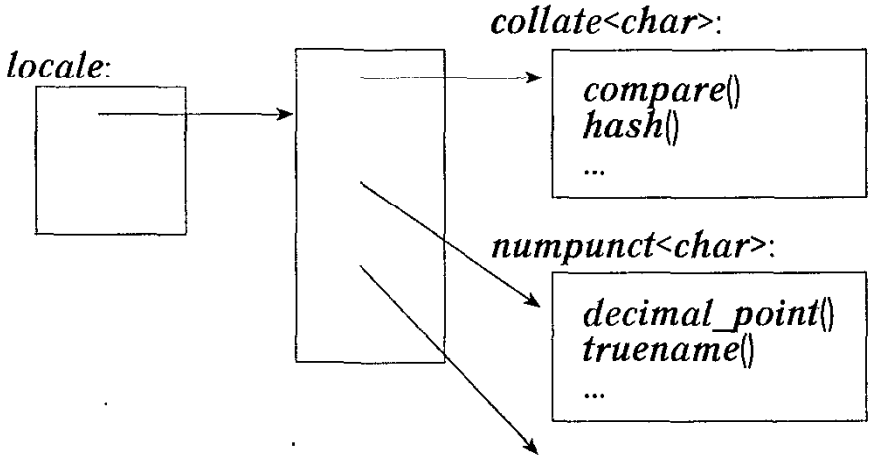
Г.2. Класс locale

Класс ***locale*** и связанные с ним средства представлены в <***locale***>:

```
class std::locale {
public:
    class facet;              // тип представления фасетов локализации; § Г.3
    class id;                 // тип для идентификации локализации; § Г.3
    typedef int category;     // тип для разбиения фасетов на категории

    static const category     // настоящие значения определяются реализацией
        none = 0,
        collate = 1,          // сравнения
        ctype = 1<<1,
        monetary = 1<<2,
        numeric = 1<<3,
        time = 1<<4,
        messages = 1<<5,
        all = collate | ctype | monetary | numeric | time | messages;

    locale() throw();                    // копия глобальной локализации (§ Г.2.1)
    locale(const locale& x) throw();     // копия x
```

```
    explicit locale(const char* p);        // копия локализации с именем p (§ Г.2.1)
    ~locale() throw();
    locale(const locale& x, const char* p, category c);   // копия x плюс фасеты
                                                          // категории c из p
    locale(const locale& x, const locale& y, category c); // копия x плюс фасеты
                                                          // категории c из y
    template<class Facet> locale(const locale& x, Facet* f);  // копия x плюс фасет f
    template<class Facet> locale combine(const locale& x);    // копия *this плюс фасет
                                                              // Facet из x
    const locale& operator=(const locale& x) throw();
    bool operator==(const locale&) const;  // сравнение локализаций
    bool operator!=(const locale&) const;
    string name() const;                   // имя данной локализации
    template <class Ch, class Tr, class A> // сравнение строк в данной локализации
    bool operator() (const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&) const;
    static locale global(const locale&);   // устанавливает новую глобальную
                                           // локализацию и возвращает старую
    static const locale& classic();        // возвращает "классическую"
                                           // локализацию в стиле C
private:
    // представление
};
```

Класс ***locale*** можно рассматривать как интерфейс к ***map<id,facet*>***, а именно, в качестве средства, позволяющего нам пользоваться ***locale::id*** для поиска соответствующего объекта класса, производного от ***locale::facet***. Настоящая реализация ***locale*** является эффективным вариантом этой идеи. Схема выглядит примерно так:



Здесь ***collate<char>*** и ***numpunct<char>*** — стандартные библиотечные фасеты (§ Г.4). Как и все фасеты, они произведены от ***locale::facet***.

Предполагается, что локализации можно легко и дешево копировать. Следовательно, почти наверняка локализация реализована в виде вспомогательного класса (handle) по отношению к специализированному ***map<id, facet*>***, представляющему собой основную часть реализации. Доступ к фасстам локализации должен осуществляться быстро, поэтому специализированный ***map<id, facet*>*** оптимизируется для

обеспечения быстрого доступа наподобие массива. Доступ к фасету локализации осуществляется при помощи нотации ***use_facet<Facet>(loc)***; см. § Г.3.1.

Стандартная библиотека предоставляет богатый набор фасетов. Чтобы помочь программистам обращаться с фасетами, стандартные фасеты подразделяются на категории, такие как ***numeric*** или ***collate*** (§ Г.4).

Программист может заменять фасеты в существующих категориях (§ Г.4, § Г.4.2.1). Однако невозможно добавить новые категории; не существует способа определить новую категорию программным путем. Понятие «категория» применимо только к фасетам стандартной библиотеки и не является расширяемым. Таким образом, фасет не обязан принадлежать к какой-либо категории, и существует множество определенных пользователями фасетов, не принадлежащих ни к одной из них.

По большей части локализации используются (неявно) в потоке ввода/вывода. У каждого потока ввода ***istream*** и потока вывода ***ostream*** имеется собственная локализация ***locale***. По умолчанию, локализацией потока является глобальная локализация (§ Г.2.1) при его создании. Задать локализацию потока можно при помощи операции ***imbue()***, а извлекается копия локализации потока посредством ***getloc()*** (§ 21.6.3).

Г.2.1. Именованные локализации

Локализация конструируется из некоторой другой локализации и из фасетов. Самым простым способом создания локализации является копирование существующей. Например:

```
locale loc0;                    // копия текущей глобальной локализации (§ Г.2.3)

locale loc1 = locale();         // копия текущей глобальной локализации (§ Г.2.3)
locale loc2("");                // копия "предпочитаемой пользователем локализации"
locale loc3("C");               // копия локализации "C"
locale loc4 = locale::classic(); // копия локализации "C"

locale loc5("POSIX");           // копия определяемой реализацией локализации "POSIX"
```

Значение ***locale("C")*** определено стандартом в качестве «классической» локализации C; именно эта локализация подразумевалась на протяжении всей книги. Все другие имена локализаций определяются реализацией.

Выражение ***locale("")*** считается «предпочитаемой пользователем локализацией». Эта локализация устанавливается неязыковыми средствами в среде исполнения программы.

В большинстве операционных систем имеются способы установки локализации для программы. Часто подходящая для оператора локализация выбирается в момент начала работы с системой. Например, я полагаю, что пользователь, настроившийся на аргентинский испанский в качестве языка по умолчанию, обнаружит, что ***locale("")*** означает для него ***locale("es_AR")***. Быстрая проверка обнаружила в одной из моих систем 51 локализацию с мнемоническими именами, такими как ***POSIX***, ***de***, ***en_UK***, ***en_US***, ***es***, ***es_AR***, ***fr***, ***sv***, ***da***, ***pl*** и ***iso_8859_1***. POSIX рекомендует имена, начинающиеся с названия языка строчными буквами, за которым может следовать название страны прописными буквами, за которым может следовать спецификатор кодировки. Например, ***jp_JP.jit***. Однако эти имена не стандартизованы для всех платформ. В другой системе, среди многих других имен локализаций, я обнаружил ***g***, ***uk***, ***us***, ***s***, ***fr***,

sw и *da*. Стандарт C++ не определяет смысл локализации для данной страны или языка, хотя на конкретных платформах такие стандарты возможны. Поэтому при обращении к именованным локализациям программисту следует полагаться на системную документацию и опыт.

Как правило, нецелесообразно включать в текст программы строки имен локализаций. Ссылка на имя файла или системную константу в тексте программы ограничивает переносимость этой программы и часто вынуждает программиста, желающего адаптировать программу к новой среде, искать и изменять подобные значения. Использование строки с именем локализации влечет аналогичные последствия. С другой стороны, локализации можно получить из среды исполнения программы (например, воспользовавшись ***locale***(""), или программа может предоставить опытному пользователю возможность задания альтернативной локализации путем ввода соответствующей строки. Например:

```
void user_set_locale(const string& question_string)
{
        cout << question_string;          // например, "Если требуется другая
                                          // локализация, введите ее имя"
        string s;
        cin >> s;
        locale::global(locale(s.c_str()));  // установить в качестве глобальной
                                            // указанную пользователем локализацию
}
```

Как правило, лучшим решением для менее опытного пользователя будет предоставление списка альтернатив. Соответствующая процедура должна знать, где и как система хранит свои локализации.

Если строковый аргумент не ссылается на реально определенную локализацию, конструктор генерирует исключение ***runtime_error*** (§ 14.10). Например:

```
void set_loc(locale& loc, const char* name)
try
{
        loc = locale(name);
}
catch (runtime_error) {
        cerr << "локализация \"" << name << "\" не определена\n";
        // ...
}
```

Если ***locale*** имеет имя, его можно получить в качестве возвращаемого значения функции ***name***(). Если нет, ***name***() вернет ***string***("*"). Строка имени, во-первых, служит средством для обращения к локализации, имеющейся в среде исполнения. Во-вторых, строку имени можно использовать для отладки. Например:

```
void print_locale_names(const locale& my_loc)
{
        cout << "имя текущей глобальной локализации: " << locale().name() << "\n";
        cout << "имя классической локализации C: " << locale::classic().name() << "\n";
        cout << "имя предпочитаемой пользователем локализации: "
             << locale("").name() << "\n";
```

```
    cout << "имя моей локализации: " << my_loc.name() << "\n";
}
```

Локализации с одинаковыми и отличными от значения по умолчанию *string*("*") строками имен полагаются совпадающими. Однако операторы == и != обеспечивают более непосредственный способ сравнения локализаций.

Копия локализации, имеющей строку имени, получает то же самое имя, поэтому у нескольких локализаций имена могут совпадать. Это разумно, потому что локализации неизменяемы и все копии определяют один и тот же набор национальных особенностей.

Вызов *locale*(*loc*, "*Foo*", *cat*) создает локализацию, подобную *loc*, за тем исключением, что новая локализация берет фасеты категории *cat* из *locale*("*Foo*"). У новой локализации есть строка имени в том и только в том случае, если таковая имеется у *loc*. Стандарт не определяет, какую конкретно строку имени получит новая локализация, но предполагается, что новая строка будет отличаться от строки, соответствующей *loc*. Одно из очевидных решений — сборка нового имени из строки имени *loc*, плюс "*Foo*". Например, если строкой имени *loc* является *en_UK*, новая локализация могла бы получить строку имени "*en_UK:Foo*".

Сведения о строках имен вновь созданных локализаций можно обобщить следующим образом:

Локализация	Строка имени
locale("*Foo*")	"*Foo*"
locale(*loc*)	*loc.name*()
locale(*loc*, "*Foo*", *cat*)	Новая строка имени, если таковая имеется у *loc*; в противном случае — *string*("*")
locale(*loc*, *loc2*, *cat*)	Новая строка имени, если таковые имеются у *loc* и *loc2*; в противном случае — *string*("*")
locale(*loc*, *Facet*)	*string*("*")
loc.combine(*loc2*)	*string*("*")

Не существует средств, которыми программист мог бы задать имя новой локализации С-строкой. Строки имен либо определены в среде исполнения программы, либо собираются из таких имен конструкторами локализаций.

Г.2.1.1. Создание новых локализаций

Новая локализация создается из существующей посредством добавления к ней или замены в ней фасетов. Как правило, новая локализация лишь немного видоизменяет существующую. Например:

```
void f(const locale& loc, const My_money_io* mio)  // My_money_io определен в § Г.4.3.1
{
    locale loc1(locale("POSIX"), loc, locale::monetary);   // использование денежных
                                                           // фасетов из loc
    locale loc2 = locale(locale::classic(), mio);          // классика плюс mio
    // ...
}
```

В нашем примере *loc1* является модифицированной копией локализации *POSIX*, использующей денежные фасеты из *loc* (§ Г.4.3). Аналогично, *loc2* является модификацией

локализации C, использующей ***My_money_io*** (§ Г.4.3.1). Если аргумент ***Facet**** (в нашем случае, ***My_money_io***) равен ***0***, новая локализация будет просто копией аргумента ***locale***.

В записи

```
locale(const locale& x, Facet* f);
```

аргумент ***f*** должен идентифицировать конкретный тип фасета. Просто ***facet**** недостаточно. Например:

```
void g(const locale::facet* mio1, const My_money_io* mio2)
{
    locale loc3 = locale(locale::classic(), mio1);    // ошибка: не известен тип фасета
    locale loc4 = locale(locale::classic(), mio2);    // правильно: тип фасета известен
    // ...
}
```

Причина в том, что ***locale*** использует тип аргумента ***Facet**** для определения типа фасета на этапе компиляции. Именно, реализация локализации применяет идентификатор типа фасета ***facet::id*** (§ Г.3) для поиска фасета в локализации (§ Г.3.1).

Обратите внимание, что конструктор

```
template <class Facet> locale(const locale& x, Facet* f);
```

является единственным механизмом языка, предоставляющим программисту возможность задания фасета, который будет использоваться в локализации. Ряд локализаций предоставляются реализациями в виде именованных локализаций (§ Г.2.1). Эти именованные локализации можно извлечь из среды исполнения программы. Программист, понимающий механизмы извлечения, присущие конкретной реализации, может добавлять таким способом новые локализации (§ Г.6[11,12]).

Набор конструкторов локализации спроектирован таким образом, чтобы тип каждого фасета становился известен либо путем выведения типа (из параметра ***Facet*** шаблона), либо благодаря тому, что фасет получен из другой локализации (которая знает его тип). Задание аргумента ***category*** косвенно определяет тип фасета, потому что локализация знает тип фасетов в категориях. Это подразумевает, что класс ***locale*** может (и должен) отслеживать типы фасетов, что позволяет манипулировать ими с минимальными накладными расходами.

Для определения типов фасетов локализации используют тип-член ***locale::id*** (§ Г.3).

Иногда полезно создать локализацию, являющуюся копией существующей, за исключением того, что некоторые ее фасеты взяты из третьей. Это реализуется шаблонной функцией-членом ***combine***(). Рассмотрим пример:

```
void f(const locale& loc, const locale& loc2)
{
    locale loc3 = loc.combine<My_money_io>(loc2);
    // ...
}
```

Новая ***loc3*** ведет себя подобно ***loc***, но использует копию ***My_money_io*** (§ Г.4.3.1) из ***loc2*** для форматирования ввода/вывода денежных единиц. Если в ***loc2*** нет ***My_money_io*** для передачи новой локализации, ***combine***() сгенерирует исключение ***runtime_error*** (§ 14.10). Результат ***combine***() не имеет строки имени.

Г.2.2. Копирование и сравнение локализаций

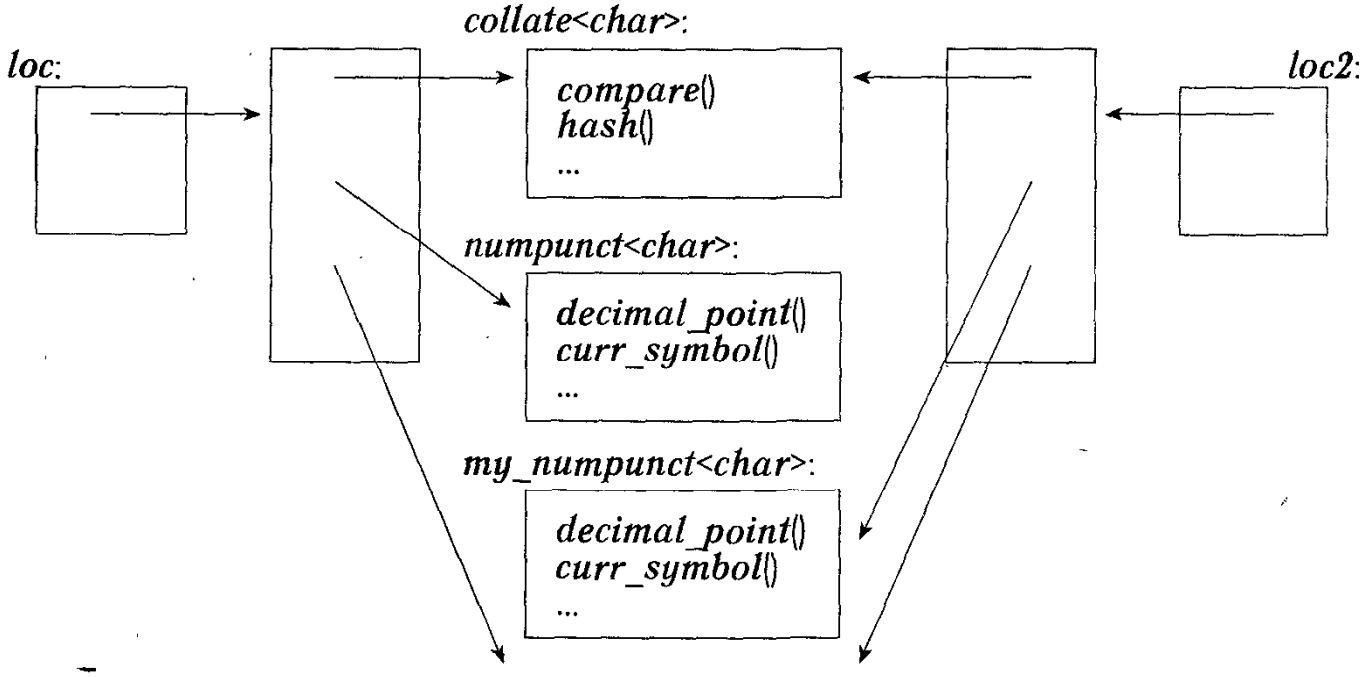
Локализацию можно скопировать инициализацией и присваиванием. Например:

```
void swap(locale& x, locale& y)          // как и std::swap()
{
        locale temp = x;
        x = y;
        y = temp;
}
```

Копия локализации успешно проверяется на равенство оригиналу, но копия является независимым и отдельным объектом. Например:

```
void f(locale* my_locale)
{
        locale loc = locale::classic();          // локализация "C"

        if (loc != locale::classic()) {
                cerr << "ошибка реализации: сообщите поставщику\n";
                exit(1);
        }
        if(&loc != &locale::classic()) cout << "ничего удивительного -- адреса
отличаются\n";
        locale loc2 = locale(loc, my_locale, locale::numeric);
        if(loc == loc2) {
                cout << "мои числовые фасеты те же, что и у classic()\n"
                // ...
        }
        // ...
}
```

Если в локализацию ***my_locale*** имеется отвечающий за пунктуацию чисел фасет ***my_numpunct<char>***, отличающийся от стандартного ***numpunct<char>*** из ***classic()***, то полученные локализации можно изобразить следующим образом:



Не существует способа модификации *locale*. Но операции над локализациями позволяют создавать новые локализации из существующих. Тот факт, что локализация неизменяема после ее создания, является существенным для обеспечения эффективности на этапе исполнения. Это позволяет пользователю локализации вызывать виртуальные функции фасетов и кэшировать возвращаемые значения. Например, *istream* может знать, какой символ используется для представления десятичной точки, и как представляется значение *true*, не вызывая *decimal_point*() каждый раз при чтении числа и *truename*() каждый раз при чтении логической переменной (§ Г.4.2). Действительно, значения, возвращаемые подобными вызовами, могут измениться лишь при вызове *imbue*() для потока (§ 21.6.3).

Г.2.3. Локализации global() и classic()

Понятие текущей локализации программы обеспечивается функциями *locale*(), возвращающей копию текущей локализации, и *locale*::*global*(*x*), устанавливающей локализацию *x* в качестве текущей. Текущую локализацию обычно называют «глобальной локализацией», отражая этим названием ее вероятную реализацию глобальным (или статическим) объектом.

Глобальная локализация неявно используется при инициализации потока. То есть каждому новому потоку придается (*imbue*, § 21.1, § 21.6.3) копия *locale*(). При запуске, глобальной локализацией является стандартная локализация C, *locale*::*classic*().

Статическая функция-член *locale*::*global*() позволяет программисту задавать глобальную локализацию. Возвращает же *global*() копию предыдущей глобальной локализации, что обеспечивает возможность восстановить прежнюю глобальную локализацию. Например:

```
void f(const locale& my_loc)
{
    ifstream fin1(some_name);                  // за fin1 закрепляется глобальная
                                               // локализация
    locale& old_global = locale::global(my_loc); // устанавливается новая
                                               // глобальная локализация
    ifstream fin2(some_other_name);            // за fin2 закрепляется локализация
                                               // my_loc
    // ...
    locale::global(old_global);                // восстановление предыдущей
                                               // глобальной локализации
}
```

Если в локализации *x* имеется строка имени, *locale*::*global*(*x*) также устанавливает глобальную локализацию C. Имеется в виду, что если программа на C++ вызывает функцию из стандартной библиотеки C, зависящую от локализации, то работа с локализацией будет согласованна в смешанной C/C++ программе.

В том случае, когда локализация *x* не имеет строки имени, не определено, влияет ли *locale*::*global*(*x*) на глобальную локализацию C. Подразумевается, что программа на C++ не может надежным и переносимым способом задать в качестве локализации C такую локализацию, которая не была получена из среды исполнения. Не существует стандартного способа, которым программа на C могла бы установить глобальную

локализацию C++ (кроме как вызвать для этого функцию на C++). В смешанной C/C++ программе наличие глобальной локализации C, отличающейся от ***global***(), провоцирует ошибки.

Задание глобальной локализации не влияет на существующие потоки ввода/вывода; они по прежнему используют локализации, закрепленные за ними до изменения глобальной локализации. Например, ***fin1*** остался неизменной в результате действий с глобальной локализацией, приведших к закреплению за ***fin2*** локализации ***my_loc***.

Изменение глобальной локализации страдает теми же недостатками, что и любые другие технологии, использующие изменение глобальных данных: по сути невозможно узнать, на что влияют изменения. Поэтому лучше свести использование ***global***() к минимуму и сосредоточить эти изменения в нескольких разделах кода, подчиняющихся простой стратегии подобных изменений. Дело упрощается благодаря возможности закрепить конкретные локализации за отдельным потоками (§ 21.6.3). Однако изобилие разбросанных по программе явных ссылок на локализации и фасеты также приведет к трудностям сопровождения.

Г.2.4. Сравнение строк

Сравнение двух строк в соответствии с конкретной локализацией является, вероятно, наиболее типичным *явным* использованием ***locale***, поэтому эта операция предоставляется непосредственно локализацией, чтобы пользователям не пришлось создавать своих собственных функций сравнения из фасета ***collate*** (§ Г.4.1). Функция сравнения определена как ***operator***()() локализации, так что ее можно непосредственно применять как предикат (§ 18.4.2). Например:

```
void f(vector<string>& v, const locale& my_locale)
{
        sort(v.begin(), v.end()); // сортировка с использованием < для сравнения элементов
        // ...
        sort(v.begin(), v.end(), my_locale);        // сортировка в соответствии
                                                    // с правилами my_locale
        // ...
}
```

При сравнении функция ***sort***() стандартной библиотеки по умолчанию использует оператор < с численными значениями набора символов конкретной реализации (§ 18.7, § 18.6.3.1).

Обратите внимание, что локализации сравнивают объекты типа ***basic_string***, а не C-строки.

Г.3. Фасеты локализаций

Фасет (***facet***) является объектом класса, производного от класса-члена ***facet*** класса ***locale***:

```
class std::locale::facet {
protected:
        explicit facet(size_t r = 0);       // "r==0": локализация управляет временем
                                            // жизни фасета
```

```
    virtual ~facet();                    // внимание: защищенный деструктор
private:
    facet(const facet&);                 // не определен
    void operator=(const facet&);        // не определен

    // представление класса
};
```

Операции копирования объявлены закрытыми и оставлены неопределенными для предотвращения копирования (§ 11.2.2).

Класс ***facet*** разработан в качестве базового, в нем нет открытых функций. Конструктор этого класса объявлен защищенным для предотвращения создания объектов типа «просто ***facet***», а деструктор является виртуальным для обеспечения корректного уничтожения объектов производных классов.

Предполагается, что объекты ***locale*** будут обращаться с ***facet*** при помощи указателей. Аргумент ***0*** конструктора ***facet*** означает, что ***locale*** обязан удалить ***facet*** после исчезновения последней ссылки на него. Напротив, ненулевой аргумент конструктора гарантирует, что ***locale*** никогда не удалит ***facet***. Ненулевое значение аргумента задумано на тот редкий случай, когда время жизни фасета управляется не косвенно локализацией, а непосредственно программистом. Например, мы можем попытаться создать объекты стандартного типа фасета ***collate_byname<char>*** (§ Г.4.1.1) следующим образом:

```
void f(const string& s1, const string& s2)
{
    // обычный случай: (по умолчанию) аргумент 0 означает, что за уничтожение
    // отвечает локализация:
    collate<char>* p = new collate_byname<char>("pl");
    locale loc(locale(), p);

    // редкий случай: аргумент 1 означает, что за уничтожение отвечает
    // пользователь:
    collate<char>* q = new collate_byname<char>("ge", 1);

    collate_byname<char> bug1("sw");     // ошибка: нельзя уничтожить
                                         // локальную переменную
    collate_byname<char> bug2("no", 1);  // ошибка: нельзя уничтожить
                                         // локальную переменную

    // ...

    // q нельзя удалить: деструктор collate_byname<char> защищен
    // и нельзя удалить p: за удаление *p отвечает локализация
}
```

Таким образом, стандартные фасеты полезны, когда управляются локализациями (как базовые классы), и очень редко — в других случаях.

Фасет ***_byname()*** является фасетом из именованной локализации среды исполнения (§ Г.2.1).

Для того, чтобы поиск фасета в локализации можно было осуществлять при помощи ***has_facet()*** и ***use_facet()*** (§ Г.3.1), для каждого фасета должен быть определен идентификатор ***id***:

```
class std::locale::id {
public:
        id();
private:
        id(const id&);                  // не определен
        void operator=(const id&);      // не определен

        // представление
};
```

Операции копирования объявлены закрытыми и оставлены неопределенными для предотвращения копирования (§ 11.2.2).

Класс *id* задуман для того, чтобы пользователь мог определить статический член типа *id* в каждом классе, предоставляющем новый интерфейс фасета (например, см. § Г.4.1). Механизмы локализации используют объекты *id* для идентификации фасетов (§ Г.2, § Г.3.1). В очевидной реализации локализации, *id* применяется как индекс вектора указателей на фасеты, реализуя таким образом эффективный *map<id, facet*>*.

Данные, используемые для определения (производных) фасетов, определяются в производном классе, а не в самом базовом классе *facet*. Это подразумевает, что программист, определяющий фасет, имеет полный контроль над данными, и что для реализации представляемого фасетом понятия может быть использован произвольный объем данных.

Обратите внимание, что все функции-члены определенного пользователем фасета должны быть *const*-членами. Как правило, фасет предполагается неизменяемым (§ Г.2.2).

Г.3.1. Доступ к фасетам локализации

Доступ к фасетам локализации осуществляется при помощи шаблонной функции *use_facet*. Мы может выяснить, содержит ли локализация данный конкретный фасет, при помощи шаблонной функции *has_facet*:

```
template <class Facet> bool has_facet(const locale&) throw();
template<class Facet> const Facet& use_facet(const locale&);    // может сгенерировать
                                                                 // bad_cast
```

Можно считать, что эти шаблонные функции осуществляют поиск шаблонного параметра *Facet* в аргументе *locale*. Или посмотрим на *use_facet* как на своего рода явное преобразование (приведение) типа локализации в тип конкретного фасета. Такая возможность обеспечена тем, что у локализации может быть только один фасет данного типа. Например:

```
void f(const locale& my_locale)
{
        char c = use_facet< numpunct<char> >(my_locale).decimal_point();
                                                // использование стандартного фасета
        // ...
        if(has_facet<Encrypt>(my_locale)) {      // в my_locale имеется фасет Encrypt?
                const Encrypt& f = use_facet<Encrypt>(my_locale);    // извлечение фасета
                                                                     // Encrypt
```

```
            const Crypto c = f.get_crypto();      // использование фасета Encrypt
            // ...
        }
        // ...
    }
```

Обратите внимание, что ***use_facet*** возвращает ссылку на константный фасет, поэтому мы не можем присвоить результат ***use_facet*** не-константе. Это разумно, потому что подразумевается, что фасет неизменяем и содержит только константные члены.

Если мы напишем ***use_facet<X>(loc)***, а фасета ***X*** в ***loc*** нет, ***use_facet()*** сгенерирует исключение ***bad_cast*** (§ 14.10). Гарантируется, что стандартные фасеты доступны во всех локализациях (§ Г.4), так что для стандартного фасета нет необходимости обращаться к ***has_facet***. Со стандартными фасетами ***use_facet*** не сгенерирует ***bad_cast***.

Как можно реализовать ***use_facet*** и ***has_facet***? Вспомните, что мы вправе рассматривать ***locale*** как ***map<id, facet*>*** (§ Г.2). Располагая типом ***facet*** в качестве шаблона аргумента ***Facet***, реализация ***has_facet*** или ***use_facet*** в состоянии обратиться к ***Facet::id*** и использовать его для поиска соответствующего фасета. Весьма наивная реализация функций ***has_facet*** и ***use_facet*** могла бы выглядеть следующим образом:

```
// псевдореализация: представим, что локализация содержит
// ассоциативный массив map<id, facet*> с именем facet_map
template <class Facet> bool has_facet(const locale& loc) throw()
{
    const locale::facet* f = loc.facet_map[Facet::id];
    return f ? true : false;
}

template <class Facet> const Facet& use_facet(const locale& loc)
{
    const locale::facet* f = loc.facet_map[Facet::id];
    if (f) return static_cast<const Facet&>(*f);
    throw bad_cast();
}
```

По-другому механизм ***facet::id*** можно интерпретировать как реализацию полиморфизма этапа компиляции. Очень близкий к ***use_facet*** результат получается применением ***dynamic_cast***. Однако специализированная ***use_facet*** реализуется куда эффективнее более общего механизма ***dynamic_cast***.

В действительности, ***id*** идентифицирует скорее интерфейс и поведение, чем класс. То есть если интерфейсы двух классов фасетов в точности совпадают, и оба класса реализуют одну и ту же семантику (насколько это имеет отношение к локализации), они должны идентифицироваться одним и тем же ***id***. Например, ***collate<char>*** и ***collate_byname<char>*** взаимозаменяемы в локализации, поэтому они оба идентифицируются ***collate<char>::id*** (§ Г.4.1).

При определении фасета с новым интерфейсом — вроде ***Encrypt*** в ***f()*** — мы должны определить и соответствующий ***id*** для идентификации фасета (см. § Г.3.2 и § Г.4.1).

Г.3.2. Простой определяемый пользователем фасет

Стандартная библиотека предоставляет стандартные фасеты, учитывающие наиболее существенные области культурных различий, такие как набор символов и ввод/вывод чисел. Чтобы познакомиться с механизмом фасетов отдельно от всякого рода сложностей, связанных с широко используемыми типами и соображениями производительности, позвольте мне сначала описать фасет для тривиального определяемого пользователем типа:

```
enum Season { spring, summer, fall, winter}; // весна, лето, осень, зима
```

Это простейший пользовательский тип, какой я только сумел выдумать. Очерченный ниже стиль написания ввода/вывода может использоваться (с минимальными вариациями) для большинства простых определяемых пользователем типов:

```
class Season_io : public locale::facet {
public:
    Season_io(int i = 0) : locale::facet(i) { }
    ~Season_io() { }        // чтобы обеспечить возможность уничтожения
                            // объектов типа Season_io (§ Г.3)

    virtual const string& to_str(Season s) const = 0;      // строковое представление s

    // поместить Season, соответствующий s, в x:
    virtual bool from_str(const string& s, Season& x) const = 0;

    static locale::id id;   // объект-идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)
};

locale::id Season_io::id;   // определение объекта-идентификатора
```

Для простоты данный фасет ограничивается представлениями, использующими *char*.

Класс *Season_io* обеспечивает общий абстрактный интерфейс для всех своих фасетов. Для определения представления ввода/вывода *Season* в конкретной локализации, мы наследуем класс от *Season_io*, определив при этом подходящие *to_str()* и *from_str()*.

Вывести *Season* просто. Если у потока имеется фасет *Season_io*, мы можем воспользоваться последним для преобразования времени года в строку. Если нет, выведем целое значение *Season*:

```
ostream& operator<<(ostream& s, Season x)
{
    const locale& loc = s.getloc();          // извлечем локализацию потока (§ 21.7.1)
    if (has_facet<Season_io>(loc)) return s << use_facet<Season_io>(loc).to_str(x);
    return s << int(x);
}
```

Обратите внимание, что реализация оператора << применяет его же, но с другими типами. Благодаря этому мы можем воспользоваться преимуществами простоты вызова << по сравнению с прямым доступом к буферам *ostream*; преимуществами чувствительности << к локализациям и, наконец, преимуществами обработки ошибок, обеспечиваемой для <<. Стандартные фасеты склонны непосредственно обращаться к буферам потока (§ Г.4.2.2, § Г.4.2.3) для достижения максимальной эффективности и гибкости, но для многих простых определяемых пользователем типов нет необходимости опускаться на уровень абстракции *streambuf*.

Как обычно, ввод немного сложнее вывода:

```
istream& operator>>(istream& s, Season& x)
{
        const locale& loc = s.getloc();        // извлечем локализацию потока (§ 21.7.1)

        if(has_facet<Season_io>(loc)) {        // чтение символьного представления
                const Season_io& f = use_facet<Season_io>(loc);
                string buf;
                if(!(s>>buf && f.from_str(buf, x))) s.setstate(ios_base::failbit);
                return s;
        }

        int i;                                 // чтение числового представления
        s >> i;
        x = Season(i);
        return s;
}
```

Обработка ошибок проста и следует стилю обработки ошибок встроенных типов. То есть если строка ввода не является допустимым значением ***Season*** в выбранной локализации, поток устанавливается в состояние ***fail*** (ошибочное состояние). Если допустимы исключения, то подразумевается, что сгенерируется исключение ***ios_base::failure*** (§ 21.3.6).

Вот тривиальная тестовая программа:

```
int main()          // тривиальный тест
{
        Season x;

        // использование локализации по умолчанию (фасет Season_io отсутствует)
        // подразумевает ввод/вывод целых чисел:
        cin >> x;
        cout << x << endl;

        locale loc(locale(), new US_season_io);  // US_season_io определен ниже
        cout.imbue(loc);                         // локализация с фасетом Season_io
        cin.imbue(loc);                          // локализация с фасетом Season_io

        cin >> x;
        cout << x << endl;
}
```

Пусть имеется вход:

```
2
summer
```

на выходе программы будет:

```
2
summer
```

Для получения этого результата мы должны определить ***US_season_io*** с целью задания строкового представления времен года и замещения (override) функций ***Season_io***, преобразующих между строковыми представлениями и перечислениями:

```
class US_season_io : public Season_io {
        static const string seasons[];
public:
        const string& to_str(Season) const;
        bool from_str(const string&, Season&) const;

        // обратите внимание, что US_season_io::id отсутствует
};

const string US_season_io::seasons[] = { "spring", "summer", "fall", "winter" };

const string& US_season_io::to_str(Season x) const
{
        if(s<spring || winter<s) {
                static const string ss = "нет такого времени года";
                return ss;
        }
        return seasons[x];
}

bool US_season_io::from_str(const string& s, Season& x) const
{
        const string* beg = &seasons[spring];
        const string* end = &seasons[winter] + 1;       // указатель "за последний элемент"
        const string* p = find(beg, end, s);            // § 3.8.1, § 18.5.2
        if(p==end) return false;
        x = Season(p - beg);
        return true;
}
```

Заметьте, что поскольку ***US_season_io*** является просто реализацией интерфейса ***Season_io***, я не определял ***id*** для ***US_season_io***. В действительности, если мы захотим использовать ***US_season_io*** в качестве ***Season_io***, мы не вправе задавать для ***US_season_io*** его собственный ***id***. Операции с локализациями, такие как ***has_facet*** (§ Г.3.1), полагают, что фасеты, реализующие одни и те же понятия, идентифицируются одним и тем же ***id*** (§ Г.3).

Единственный интересный вопрос реализации: что делать, если требуется вывести некорректный ***Season***. Естественно, этого не должно произойти. Однако случаи появления неверных значений простых определяемых пользователем классов не являются необычными, поэтому целесообразно принять во внимание подобную возможность. Я мог бы сгенерировать исключение. Но когда имеешь дело с простым выводом, предназначенным для человеческих глаз, полезно подготовить представление «вне диапазона» для недопустимых значений. Обратите внимание, что при вводе стратегия обработки ошибок отдана на откуп оператору >>, тогда как при выводе политика обработки ошибок реализуется функцией фасета ***to_str***(). Это сделано с целью иллюстрации возможных альтернатив проектирования. В «промышленном проекте» функции фасета либо реализуют обработку ошибок и для ввода, и для вывода, либо сообщают об ошибках операторам >> и <<, которые и заниматься обработкой ошибок.

Обсуждаемый проект ***Season_io*** полагается на то, что специфические для конкретной локализации строки предоставляются производными классами. Альтернативный подход предполагает, что ***Season_io*** сам извлекает строки из соответствующего храни-

лица (см. § Г.4.7). Создание класса ***Season_io***, которому строки времен года передаются как аргументы конструктора, оставляем в качестве упражнения (§ Г.6[2]).

Г.3.3. Использование локализаций и фасетов

Локализации в первую очередь используются в потоках ввода/вывода стандартной библиотеки. Однако механизм локализации является общим и расширяемым средством представления информации, зависящей от культурных особенностей. Класс ***messages*** (§ Г.4.7) является примером фасета, не имеющего никакого отношения к потокам ввода/вывода. Преимуществами локализаций могут воспользоваться расширения потоковой библиотеки ввода/вывода и даже средства ввода/вывода, не основанные на потоках. Кроме того, пользователь может применять локализации в качестве удобного способа организации произвольной информации, связанной с культурными различиями.

Благодаря общности механизмов локализаций и фасетов, возможности определяемых пользователем фасетов не ограничены. Вероятными кандидатами на представление фасетами являются даты, часовые пояса, телефонные номера, номера социального страхования (персональные идентификационные номера), коды товаров, температуры, произвольные пары вида (единица измерения, значение), почтовые коды, размеры одежды и международные номера книг ISBN.

Как и другими мощными механизмами, фасетами нужно пользоваться с осторожностью. Из того факта, что нечто может быть представлено в виде фасета, не следует, что такое представление будет наилучшим. При выборе представления культурных зависимостей ключевыми являются, как всегда, следующие вопросы: каким образом различные решения оказывают влияние на сложность написания кода; насколько легко читается код; просто ли поддерживать программу; и насколько эффективны операциями ввода/вывода в плане времени выполнения и используемой памяти.

Г.4. Стандартные фасеты

Стандартная библиотека предоставляет в *<locale>* следующие фасеты для локализации *classic*():

Стандартные фасеты (локализации *classic*())

	Категория	Назначение	Фасеты
§ Г.4.1	*collate*	Сравнение строк	*collate<Ch>*
§ Г.4.2	*numeric*	Ввод/вывод чисел	*numpunct<Ch>* *num_get<Ch>* *num_put<Ch>*
§ Г.4.3	*monetary*	Ввод/вывод денег	*moneypunct<Ch>* *moneypunct<Ch, true>* *money_get<Ch>* *money_put<Ch>*
§ Г.4.4	*time*	Ввод/вывод времени	*time_get<Ch>* *time_put<Ch>*
§ Г.4.5	*ctype*	Классификация символов	*ctype<Ch>* *codecvt<Ch, char, mbstate_t>*
§ Г.4.7	*messages*	Выборка сообщений	*messages<Ch>*

В этой таблице *Ch* означает *char* или *wchar_t*. Пользователь, которому требуется, чтобы стандартный ввод/вывод работал с другим символьным типом *X*, должен предоставить подходящие варианты фасетов *X*. Например, для управления преобразованиями между *X* и *char* может потребоваться *codecvt<X, char, mbstate_t>* (§ Г.4.6). Тип *mbstate_t* используется для представления состояния индикатора смещения в многобайтном представлении символов (§ Г.4.6); *mbstate_t* определен в *<cwchar>* и *<wchar.h>*. Эквивалентом *mbstate_t* для произвольного символьного типа *X* является *char_traits<X>::state_type*.

Кроме перечисленных выше, стандартная библиотека предоставляет в *<locale>* следующие фасеты:

Стандартные фасеты

	Категория	Назначение	Фасеты
§ Г.4.1	*collate*	Сравнение строк	*collate_byname<Ch>*
§ Г.4.2	*numeric*	Ввод/вывод чисел	*numpunct_byname<Ch>* *num_get<C, In>* *num_put<C, Out>*
§ Г.4.3	*monetary*	Ввод/вывод денег	*moneypunct_byname<Ch, International>* *money_get<C, In>* *money_put<C, Out>*
§ Г.4.4	*time*	Ввод/вывод времени	*time_put_byname<Ch, Out>*
§ Г.4.5	*ctype*	Классификация символов	*ctype_byname<Ch>*
§ Г.4.7	*messages*	Выборка сообщений	*messages_byname <Ch>*

При инстанцировании фасета из этой таблицы, *Ch* может быть *char* или *wchar_t*; *C* может быть любым символьным типом (§ 20.1). Параметр *International* может иметь значение «истина» или «ложь»; «истина» означает, что используется «интернациональное» четырехсимвольное представление денежного знака (§ Г.4.3.1). Тип *mbstate_t* применяется для представления состояния индикатора смещения в многобайтном представлении символов (§ Г.4.6); *mbstate_t* определен в *<cwchar>* и *<wchar.h>*.

Параметры *In* и *Out* — итераторы ввода и вывода соответственно (§ 19.1, § 19.2.1). Снабжение фасетов *_put* и *_get* этими шаблонными аргументами позволяет программисту реализовывать фасеты, которые осуществляют доступ к нестандартным буферам (§ Г.4.2.2). Буфера, связанные с потоками ввода/вывода, являются буферами потока, поэтому их итераторы имеют тип *ostreambuf_iterator* (§ 19.2.6.1, § Г.4.2.2). Следовательно, для обработки ошибок доступна функция *failed*() (§ 19.2.6.1).

Фасет *F_byname* является производным от фасета *F*. Фасет *F_byname* предоставляет интерфейс, идентичный интерфейсу *F*, за исключением того, что добавлен конструктор со строковым аргументом, именующим локализацию (см. § Г.4.1). *F_byname(name)* предоставляет подходящую семантику для *F*, определенного в *locale(name)*. Идея состоит в извлечении версии стандартного фасета из именованной локализации (§ Г.2.1) среды исполнения программы. Например:

```
void f(vector<string>& v, const locale& loc)
{
    locale d1(loc, new collate_byname<char>("da"));      // сравнение датских строк
    locale dk(dl, new ctype_byname<char>("da"));   // классификация датских символов
    sort(v.begin(), v.end(), dk);
}
```

Эта новая локализация ***dk*** будет пользоваться датскими строками, но в ней оставлены без изменений соглашения по умолчанию, касающиеся чисел. Заметим, что поскольку второй аргумент фасета по умолчанию равен ***0***, временем жизни фасета, созданного при помощи оператора ***new***, управляет локализация(§ Г.3).

Подобно конструкторам локализации со строковыми аргументами, конструкторы ***_byname*** осуществляют доступ к среде исполнения программы. Соответственно, они значительно медленнее, чем конструкторы, которым не нужно обращаться к среде окружения. Почти всегда быстрее сначала создать локализацию, а затем осуществлять доступ к ее фасетам, чем пользоваться фасетами ***_byname*** во многих местах программы. Таким образом, как правило хорошей идеей является однократное чтение фасета из среды окружения с последующим многократным использованием его копии в главной памяти (программы). Например:

```
locale dk("da");        // однократное чтение датской локализации (всех ее фасетов)
                        // затем используем dk и ее фасеты
                        // по мере необходимости

void f(vector<string>& v, const locale& loc)
{
    const collate<char>& col = use_facet<collate<char> >(dk);
    const collate<char& ctyp = use_facet<ctype<char> >(dk);

    locale d1(loc, col);        // сравнение датских строк
    locale d2(d1, ctyp);        // классификация датских символов
                                // и сравнение датских строк

    sort(v.begin(), v.end(), d2);
    // ...
}
```

Понятие категории даст болсе простой способ манипулирования стандартными фасетами локализаций. Например, имея локализацию ***dk***, мы можем сконструировать локализацию, которая читает и сравнивает строки в соответствии с датскими правилами (где по сравнению с английским имеются три дополнительные гласные), но в которой сохраняется синтаксис чисел, используемый в C++:

```
locale dk_us(locale::classic(), dk, collate | ctype);      // датские буквы, американские числа
```

При описании конкретных стандартных фасетов приводятся и другие примеры их использования. В частности, обсуждение ***collate*** (§ Г.4.1) выявляет многие общие решения относительно архитектуры фасетов.

Заметим, что стандартные фасеты часто зависят друг от друга. Например, ***num_put*** зависит от ***numpunct***. Только детально зная индивидуальные фасеты, вы сможете успешно комбинировать фасеты или создавать новые версии стандартных фасетов. Другими словами, помимо простых операций, упоминавшихся в § 21.7, механизмы локализаций предназначены не для новичков.

Разработка индивидуального фасета — обычно неблагодарный труд. Причина отчасти в том, что фасеты должны отражать неупорядоченные культурные соглашения, неподконтрольные разработчику библиотеки, а отчасти в том, что средства стандартной библиотеки C++ должны оставаться в значительной степени совместимыми со средствами стандартной библиотеки C и различными стандартами, зависящими от платформ. Например, POSIX предоставляет средства локализации, игнорировать которые разработчику библиотеки было бы неразумно.

С другой стороны, среда, обеспечиваемая локализациями и фасетами, — очень общая и гибкая. Можно спроектировать фасет для хранения произвольных данных, а операции фасета могут обеспечить любые желаемые действия с этими данными. Если поведение нового фасета не слишком ограничено обилием соглашений, его проект, вероятно, окажется простым и ясным (§ Г.3.2).

Г.4.1. Сравнение строк

Стандартный фасет ***collate*** (сравнение) обеспечивает сравнение массивов символов типа ***Ch***:

```
template <class Ch>
class std::collate : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<ch> string_type;

    explicit collate(size_t r = 0);

    int compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const
        { return do_compare(b, e, b2, e2); }

    long hash(const Ch* b, const Ch* e) const {return do_hash(b, e); }
    string_type transform(const Ch* b, const Ch* e) const {return do_transform(b, e); }

    static locale::id id;       // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)

protected:
    ~collate();                 // внимание: защищенный деструктор

    virtual int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    virtual string_type do_transform(const Ch* b, const Ch* e) const;
    virtual long do_hash(const Ch* b, const Ch* e) const;
};
```

Как и все фасеты, ***collate*** открыто наследуется от ***facet*** и предоставляет конструктор, аргумент которого указывает, отвечает ли класс ***locale*** за время жизни фасета (§ Г.3).

Заметьте, что деструктор защищен. Фасет ***collate*** не предназначен для непосредственного использования, а служит базовым классом для всех (производных) классов сравнения строк под управлением ***locale*** (§ Г.3). Под интерфейс ***collate*** прикладные программисты, поставщики реализаций и библиотек могут писать различные фасеты сравнения строк.

Базовой функцией сравнения строк по правилам конкретного ***collate*** является ***compare***(); она возвращает ***1***, если первая строка лексикографически больше второй, ***0*** — если строки идентичны, и ***–1*** — если вторая строка больше первой. Например:

```
void f(const string& s1, const string& s2, collate<char>& cmp)
{
    const char* cs1 = s1.data();  // потому что compare() работает с char[]
    const char* cs2 = s2.data();
    switch (cmp.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size())) {
    case 0:                       // идентичные с точки зрения стр строки
        // ...
        break;
    case -1:                      // s1 < s2
        // ...
        break;
    case 1:                       // s1 > s2
        // ...
        break;
    }
}
```

Обратите внимание, что функции-члены ***collate*** сравнивают массивы ***Ch***, а не строки ***basic_string*** или завершающиеся нулем С-строки. В частности, ***Ch*** с числовым значением ***0*** воспринимается как обычный символ, а не терминатор. Кроме того, ***compare***() отличается от ***strcmp***() тем, что возвращает конкретно ***–1***, ***0*** и ***1***, а не ***0*** и (произвольные) отрицательные и положительные значения (§ 20.4.1).

Стандартный библиотечный тип ***string*** не чувствителен к локализации. То есть сравнение производится с использованием правил, принятых для набора символов данной реализации (§ В.2). Более того, стандартный тип ***string*** не позволяет непосредственно задать критерий сравнения (глава 20). Для осуществления сравнений, зависящих от локализации, мы воспользуемся функцией ***compare***() фасета ***collate***. С точки зрения формы записи, быть может удобнее задействовать ***collate***() косвенно, через ***operator***() локализации (§ Г.2.4). Например:

```
void f(const string& s1, const string& s2, const char* n)
{
    bool b = s1 == s2;            // сравнение (набор символов реализации)

    const char* cs1 = s1.data();  // потому что compare работает с массивом char[]
    const char* cs2 = s2.data();

    typedef collate<char> Col;

    const Col& glob = use_facet<Col>(locale());          // из текущей глобальной
                                                         // локализации
    int i0 = glob.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size());

    const Col& my_coll = use_facet<Col>(locale(""));     // из моей любимой локализации
    int i1 = my_coll.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size());

    const Col& coll = use_facet<Col>(locale(n));         // из локализации по имени n
    int i2 = coll.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size());

    int i3 = locale()(s1, s2);          // в текущей глобальной локализации
    int i4 = locale("")(s1, s2);        // в моей любимой локализации
    int i5 = locale(n)(s1, s2);         // в локализации по имени n

    // ...
}
```

В примере *i0==i3*, *i1==i4* и *i2==i5*. Нетрудно представить случаи, когда *i2*, *i3* и *i4* различаются. Рассмотрим последовательность слов из немецкого словаря:

Dialekt, Diät, dich, dichten, Dichtung

По соглашению, существительные (и только) пишутся с заглавной буквы, но порядок не зависит от регистра.

Зависящая от регистра сортировка расположила бы слова, начинающиеся с D, перед словами, начинающимися с d:

Dialekt, Diät, Dichtung, dich, dichten

Символ *ä* (умляут *a*) интерпретируется как «нечто вроде *a*», поэтому он стоит раньше *c*. Однако в большинстве распространенных наборов символов числовое значение *ä* больше числового значения *c*. Соответственно, *int*('*c*') < *int*('*ä*') и на выходе простой сортировки, основанной на числовых значениях, будет:

Dialekt, Dichtung, Diät, dich, dichten

Написание функции сравнения, упорядочивающей эту последовательность правильно согласно словарю, является довольно интересным упражнением (§ Г.6[3]).

Функция *hash*() вычисляет значение хеширования (§ 17.6.2.3), которое, очевидно, может оказаться полезным при построении таблиц хеширования.

Функция *transform*() производит строку, сравнение которой с другими строками дает тот же результат, что получился бы при сравнении с ее входной строкой. Функция *transform* предназначена для оптимизации кода, в котором одна строка сравнивается со многими. Это полезно при осуществлении поиска одной или нескольких строк во множестве строк.

Открытые функции *compare*(), *hash*() и *transform*() реализованы обращениями к защищенным виртуальным функциям *do_compare*(), *do_hash*() и *do_transform*() соответственно. В производных классах «*do*_функции» можно заместить. Такая стратегия «двух функций» позволяет разработчику библиотеки, реализующему невиртуальные функции, предоставить некоторую общую функциональность для всех вызовов, вне зависимости от того, чем на самом деле занимаются предоставляемые пользователями «*do*_функции».

Использование виртуальных функций предохраняет полиморфную природу фасета, но потенциально накладно. Во избежание лишних вызовов функций, *locale* может выявить конкретный применяемый *facet* и кэшировать значения, нужные ей для повышения производительности (§ Г.2.2).

Статический член *id* типа *locale::id* идентифицирует фасет (§ Г.3). Стандартные функции *has_facet* и *use_facet* полагаются на соответствие этих идентификаторов и фасетов (§ Г.3.1). Два фасета, предоставляющие локализации идентичный интерфейс и семантику, должны иметь одно и то же значение *id*. Например, у *collate<char>* и *collate_byname<char>* — одинаковые *id*. И наоборот, два фасета, выполняющие различные функции (по отношению к локализации), должны иметь различные идентификаторы. Например, у *numpunct<char>* и *num_put<char>* —разные значения *id* (§ Г.4.2).

Г.4.1.1. Именованные фасеты сравнения

Фасет *collate_byname* является вариацией *collate* для конкретной локализации с именем, заданным аргументом конструктора:

```
template <class Ch>
class std::collate_byname : public collate<ch> {
public:
    typedef basic_string<Ch> string_type;

    explicit collate_byname(const char*, size_t r = 0);    // создание из именованной
                                                           // локализации

    // обратите внимание: ни нового id, ни новых функций

protected:
    ~collate_byname();    // внимание: защищенный деструктор

    // заместим виртуальные функции collate<ch>:

    int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    string_type do_transform(const Ch* b, const Ch* e) const;
    long do_hash(const Ch* b, const Ch* e) const;
};
```

Таким образом, фасетом *collate_byname* можно пользоваться для извлечения *collate* из локализации, именованной в среде исполнения программы (§ Г.4). Очевидный способом хранения фасетов в среде исполнения — запись соответствующих данных в файл. Менее гибкая альтернатива — представление фасета в виде текста программы и данных в фасете *_byname*.

Класс *collate_byname<char>* является примером фасета, у которого нет собственного идентификатора *id* (§ Г.3). В рамках *locale* фасеты *collate_byname<Ch>* и *collate<Ch>* взаимозаменяемы. Фасеты *collate* и *collate_byname* для одной локализации отличаются только дополнительным конструктором, предоставляемым *collate_byname*, и семантикой, предлагаемой *collate_byname*.

Обратите внимание, что деструктор *_byname* является защищенным. Значит нельзя объявить фасету *_byname* как локальную переменную. Например:

```
void f()
{
    collate_byname<char> my_coll("");    // ошибка: невозможно уничтожить my_coll
    // ...
}
```

Это отражает точку зрения, что локализациями и фасетами лучше пользоваться на достаточно высоком уровне программы, чтобы они воздействовали на значительную часть кода. Примером является установка глобальной локализации (§ Г.2.3) или закрепление локализации за потоком (§ 21.6.3, § Г.1). При необходимости мы можем унаследовать от *_byname* производный класс с открытым деструктором и объявить локальные переменные этого класса.

Г.4.2. Ввод и вывод чисел

Вывод чисел осуществляется фасетом *num_put* путем записи в буфер потока (§ 21.6.4). Ввод чисел выполняется фасетом *num_get* путем чтения из буфера потока.

Формат, используемый ***num_put*** и ***num_get***, определяется фасетом «пунктуации чисел» ***numpunct***.

Г.4.2.1. Пунктуация чисел

Фасет ***numpunct*** (пунктуация чисел) определяет формат ввода/вывода встроенных типов, таких как ***bool***, ***int*** и ***double***:

```
template<class Ch>
class std::numpunct : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit numpunct(size_t r = 0);

    Ch decimal_point() const;          // '.' в classic()
    Ch thousands_sep() const;          // ',' в classic()
    string grouping() const;           // "" в classic(), то есть отсутствие
                                       // группировки

    string_type truename() const;      // "true" в classic()
    string_type falsename() const;     // "false" в classic()

    static locale::id id;              // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)
protected:
    ~numpunct();

    // виртуальные "do_функции" (см. § Г.4.1)
};
```

Символы строки, возвращаемой ***grouping()***, читаются как последовательность малых целых значений. Каждое число задает количество цифр в группе. Символ номер ***0*** определяет крайнюю справа группу (наименее значимые цифры), символ номер ***1*** определяет группу слева от нее и т. д. Таким образом, ***"\004\002\003"*** описывает формат числа вида ***123-45-6789*** (при условии, что вы используете ***'-'*** в качестве символа-разделителя). Если есть необходимость, последнее число образца (pattern) группировки используется повторно, то есть ***"\003"*** эквивалентно ***"\003\003\003"***. Само название разделительного символа ***thousands_sep()*** (разделитель тысяч) подсказывает, что группировка чаще всего применяется для более читаемого отображения больших целых. Функции ***grouping()*** и ***thousands_sep()*** определяют формат как ввода, так и вывода целых. Кроме того, они определяют формат для целой части чисел с плавающей точкой, но не цифр после ***decimal_point()***.

Мы задаем новый стиль пунктуации, создавая производный от ***numpunct*** класс. Например, я мог бы определить фасет ***My_punct*** для записи целых значений тройками цифр с пробелом между ними, а значений с плавающей точкой в европейском стиле — отделяя дробную часть запятой:

```
class My_punct : public std::numpunct<char> {
public:
    typedef char char_type;
    typedef string string_type;

    explicit My_punct(size_t r = 0) : std::numpunct<char>(r) { }
```

```
protected:
        char do_decimal_point() const { return ',' }      // запятая
        char do_thousands_sep() const { return ' '; }     // пробел
        string do_grouping() const {return "\003"; }      // группировка по 3 цифры
};

void f()
{
        cout << "setprecision (4) << fixed;
        cout << "формат A: " << 12345678 << " *** " << 1234.5678 << '\n';

        locale loc(locale(), new My_punct);
        cout.imbue(loc);
        cout << "формат B: " << 12345678 << " *** " << 1234.5678 << '\n';
}
```

На выходе будет:

```
формат A: 12345678 *** 1234.5678
формат B: 12 345 678 *** 1 234,5678
```

Отметим, что ***imbue()*** хранит копию аргумента в своем потоке. Так что поток может использовать приданную ему локализацию даже после уничтожения исходной копии этой локализации. Если установлен флаг ***boolalpha*** потока ввода/вывода (§ 21.2.2, § 21.4.1), то для символьного представления ***true*** и ***false*** используются строки, возвращаемые ***truename()*** и ***falsename()*** соответственно; в противном случае используются ***1*** и ***0***.

Имеется и ***_byname*** версия (§ Г.4, § Г.4.1) ***numpunct***:

```
template <class Ch>
class std::numpunct_byname : public numpunct<Ch> { /* ... */ };
```

Г.4.2.2. Вывод чисел

При записи в буфер потока (§ 21.6.4), ***ostream*** использует фасет ***num_put***:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::num_put : public locale::facet {
public:
        typedef Ch char_type;
        typedef Out iter_type;

        explicit num_put(size_t r = 0);

        // поместить значение "v" в позицию "b" буфера потока "s"
        Out put(Out b, ios_base& s, Ch fill, bool v) const;
        Out put(Out b, ios_base& s, Ch fill, long v) const;
        Out put(Out b, ios_base& s, Ch fill, unsigned long v) const;
        Out put(Out b, ios_base& s, Ch fill, double v) const;
        Out put(Out b, ios_base& s, Ch fill, long double v) const;
        Out put(Out b, ios_base& s, Ch fill, const void* v) const;

        static locale::id id;      // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)

protected:
```

```
    ~num_put();

    // виртуальные "do_функции" (см. § Г.4.1)
};
```

Аргумент ***Out*** (итератор вывода, см § 19.1, 19.2.1) определяет позицию буфера потока (§ 21.6.4), в которую ***put***() помещает символы, соответствующие выводимому числовому значению. Возвращаемое значение ***put***() — итератор, указывающий на позицию, следующую за последним записанным символом.

Отметьте, что специализация по умолчанию ***num_put*** (та, чей итератор доступа к символам имеет тип ***ostreambuf_iterator<Ch>***), является частью стандартной локализации (§ Г.4). Чтобы воспользоваться другой специализацией, придется создать ее самостоятельно. Например:

```
template<class Ch>
class String_numput : public std::num_put<Ch, typename basic_string<Ch>::iterator> {
public:
    String_numput() : std::num_put<Ch, typename basic_string<Ch>::iterator>(1) { }
};

void f(int i, string* s, int pos) // форматирование i в s, начиная с позиции pos
{
    String_numput<char> f;
    ios_base& xxx = cout;
    f.put(s.begin() + pos, xxx, ' ', i);      // форматирование i в s
}
```

Аргумент ***ios_base*** используется для получения информации о состоянии форматирования и локализации. Например, если требуется заполнить место «водой», используется символ ***fill***, как того требует аргумент ***ios_base***. Обычно буфер потока, куда пишут через ***b***, является буфером, ассоциированным с ***ostream***, для которого ***s*** является базовым классом. Заметим, что сконструировать объект ***ios_base*** не просто. Он, в частности, управляет различными сторонами форматирования, которые должны быть согласованы для получения приемлемого вывода. Соответственно, у ***ios_base*** нет открытого конструктора (§ 21.3.3).

Функция ***put***() также пользуется аргументом ***ios_base*** для доступа к ***locale***() потока. Эта локализация нужна для определения пунктуации (§ Г.4.2.1), символьного представления булевых значений и преобразований к ***Ch***. Например, предположим, что ***s*** является ***ios_base*** аргументом ***put***(); тогда в коде ***put***() мог бы присутствовать примерно такой фрагмент:

```
const locale& loc = s.getloc();
// ...
wchar_t w = use_facet<ctype<char> >(loc).widen(c);      // преобразование char в Ch
// ...
string pnt = use_facet<numpunct<char> >(loc).decimal_point(); // по умолчанию '.'
// ...
string flse = use_facet<numpunct<char> >(loc).falsename();    // по умолчанию "false"
```

Стандартные фасеты, вроде ***num_put<char>***, обычно используются неявно — в функциях стандартного ввода/вывода, — поэтому большинству программистов нет необ-

ходимости знать о них. Тем не менее, применение подобных фасетов функциями стандартной библиотеки представляет интерес, потому что иллюстрирует работу потоков ввода-вывода и способы обращения с фасетами. Как всегда, стандартная библиотека предоставляет образцы любопытных технологий программирования.

Используя ***num_put***, разработчик потока вывода ***ostream*** мог бы реализовать его следующим образом:

```
template<class Ch, class Tr>
ostream& std::basic_ostream<Ch, Tr>::operator<<(double d)
{
        sentry guard(*this);            // см. § 21.3.8
        if(!guard) return *this;

        try {
                if(use_facet<num_put<Ch> >(getloc()).put(*this, *this, this->fill(), d).failed())
                        setstate(badbit);
        }
        catch (...) {
                handle_oiexception(*this);
        }
        return *this;
}
```

В примере происходит много интересных вещей. Класс ***sentry*** (часовой) обеспечивает гарантию выполнения всего «префиксного» и «суффиксного» кода (§ 21.3.8). Мы получаем локализацию потока вывода, вызывая функцию-член ***getloc***() (§ 21.7) и извлекаем из локализации фасет ***num_put*** при помощи ***use_facet*** (§ Г.3.1). После этого мы обращаемся к надлежащей функции ***put***(), которая и выполняет работу. Для несложного изготовления двух первых аргументов ***put***(), мы конструируем итератор ***ostreambuf_iterator*** из ***ostream*** (§ 19.2.6) и неявно преобразуем ***ostream*** в его базовый класс ***ios_base*** (§ 21.2.1).

Функция ***put***() возвращает свой аргумент, являющийся итератором вывода. Этот итератор вывода извлекается из ***basic_ostream***, поэтому он является ***ostreambuf_iterator***. Следовательно, для проверки на сбой и для установки потока в нужное состояние нам доступна ***failed***() (§ 19.2.6.1).

Я не пользовался ***has_facet***, потому что гарантируется, что стандартные фасеты (§ Г.4) имеются в каждой локализации. Если эта гарантия не выполняется, генерируется исключение ***bad_cast*** (§ Г.3.1).

Функция ***put***(), в свою очередь, вызывает виртуальную функцию ***do_put***(), а это может привести к исполнению пользовательского кода, так что ***operator<<***() должен быть готов к обработке исключения, генерируемого замещенной ***do_put***(). Кроме того, для некоторых символьных типов может отсутствовать ***num_put***, поэтому ***use_facet***() может сгенерировать ***std::bad_cast*** (§ Г.3.1). Поведение << для встроенных типов, таких как ***double***, определено стандартом C++. Стало быть вопрос не в том, *что* должна делать функция ***handle_ioexception***(), а в том, *как* она исполнит то, что предписано стандартом. Если в этом исключительном состоянии ***ostream*** установлен ***badbit*** (§ 21.3.6), исключение просто сгенерируется повторно. В противном случае обработка исключения сводится к установке состояния потока и продолжению исполнения. В обоих случаях следует установить флаг ***badbit*** (§ 21.3.3):

```
template<class Ch, class Tr>
void handle_ioexception(std::basic_ostream<Ch, Tr>& s)    // вызывается из блока catch
{
    if (s.exceptions() & ios_base::badbit) {
        try{s.setstate(ios_base::badbit); } catch (...) { }
        throw;                          // повторная генерация
    }
    s.setstate(ios_base::badbit);  // может сгенерировать basic_ios::failure
}
```

Потребовался блок *try*, потому что *setstate()* может сгенерировать *basic_ios::failure* (§ 21.3.3, § 21.3.6). Однако, если в исключительном состоянии потока установлен *badbit*, оператор << обязан повторно сгенерировать исключение, приведшее к вызову *handle_ioexception()* (а не просто *basic_ios::failure*).

Реализация операции << для встроенных типов, таких как *double*, должна непосредственно писать в буфер потока. При написании << для определяемого пользователем типа, часто удается избежать избыточной сложности путем выражения вывода определяемого пользователем типа в терминах вывода существующих типов (§ Г.3.2).

Г.4.2.3. Ввод чисел

При чтении из буфера потока (§ 21.6.4), поток ввода *istream* использует фасет *num_get*:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::num_get : public locale::facet{
public:
    typedef Ch char_type;
    typedef In iter_type;

    explicit num_get(size_t r=0);

    // чтение [b:e) в v с использованием правил форматирования из s;
    // сообщаем об ошибке путем установки r:
    In get(In b, In e, ios_base& s, ios_base::iostate& r, bool& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned short& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned int& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, float& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, void*& v) const;

    static locale::id id;       // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)

protected:
    ~num_get();

    // виртуальные "do_функции" (см. § Г.4.1)
};
```

В основном, ***num_get*** организована так же, как и ***num_put*** (§ Г.4.2.2). Поскольку ***get***() читает, а не пишет, ей требуется пара итераторов ввода, а аргумент, обозначающий объект для чтения, является ссылкой. Для уведомления о состоянии потока устанавливается переменная ***r*** типа ***iostate***. Если значение желаемого типа не удается прочитать, в ***r*** устанавливается ***failbit***; по достижении конца ввода в ***r*** устанавливается ***eofbit***. Оператор ввода воспользуется ***r*** для принятия решения об установке состояния потока. Если ошибок не было, считанное значение присваивается ***v***, в противном случае переменная ***v*** остается неизменной.

Реализация ***istream*** может выглядеть следующим образом:

```
template<class Ch, class Tr>
istream& std::basic_istream<Ch, Tr>::operator>>(double& d)
{
        sentry guard(*this);                    // см. § 21.3.8
        if(!guard) {
                setstate(failbit);
                return *this;
        }

        iostate state = 0;                      // все хорошо
        istreambuf_iterator<Ch> eos;
        double dd;

        try {
                use_facet<num_get<Ch> >(getloc()).get(*this, eos, *this, state, dd);
        }
        catch (...) {
                handle_ioexception(*this);      // см. § Г.4.2.2
                return *this;
        }
        if(state==0 || state==eofbit) d = dd;   // значение устанавливается только
                                                // при успешном завершении get()
        setstate(state);
        return *this;
}
```

В случае ошибки функция ***setstate***() сгенерирует исключения, заданные для ***istream*** (§ 21.3.6).

Определив фасет ***numpunct***, такой как ***My_punct*** из § Г.4.2, мы можем читать с использованием нестандартной пунктуации. Например:

```
void f()
{
        cout << "формат A: ";
        int i1;
        double d1;
        cin >> i1 >> d1;    // чтение с использованием стандартного формата "12345678"

        locale loc(locale::classic(), new My_punct);
        cin.imbue(loc);
        cout << "формат B: "
        int i2;
        double d2;
```

```
        cin >> i1 >> d2;    // чтение с использованием формата "12 345 678"
}
```

Чтобы прочитать более экзотические числовые форматы, нам придется заместить ***do_get***(). Например, мы могли бы определить ***num_get*** для чтения римских чисел типа XXI или MM (§ Г.6[15]).

Г.4.3. Ввод и вывод денежных значений

Форматирование денежных сумм технически сходно с форматированием «обычных» чисел (§ Г.4.2). Однако представление денежных величин еще более чувствительно к национальным особенностям. Например, отрицательные значения (убыток, дебит), такие как ***–1.25***, в некоторых контекстах должны быть представлены как положительные значения, заключенные в круглые скобки: (***1.25***). А иногда для распознавания отрицательных сумм используется цвет.

Не существует стандартного «денежного типа». Напротив, предполагается, что программист явно применит денежные фасеты к тем числам, которые, как ему известно, представляют денежные суммы. Например:

```
class Money {     // простой тип для хранения денежных величин
        long int amount;
public;
        Money(long int i) : amount(i) { }
        operator long int() const { return amount; }
};

// ...

void f(long int i)
{
        cout << "значение = " << i << " сумма= " << Money(i) << endl;
}
```

Задача денежных фасетов состоит в предоставлении разумно простого способа записи оператора вывода для ***Money*** таким образом, чтобы сумма печаталась с учетом местных соглашений (см. § Г.4.3.2). Формат вывода будет зависеть от локализации, закрепленной за ***cout***. Возможные варианты форматов вывода:

```
значение= 1234567 сумма= $12345.67
значение= 1234567 сумма= 12345,67 DKK
значение= –1234567 сумма= $–12345.67
значение= –1234567 сумма= –$12345.67
значение= –1234567 сумма= (CHF12345,67)
```

Для денежных сумм, как правило, важно точно учитывать даже самую мелкую монету. Поэтому я принял обычное соглашение об использовании целого значения для представления количества центов (пенсов, эрэ, копеек и т. д.), а не количества долларов (фунтов, крон, динар, евро и т. д.). Это соглашение поддерживается функцией ***frac_digits***() фасеты ***moneypunct*** (§ Г.4.3.1). Вид «десятичной точки» определяется при помощи ***decimal_point***().

Фасеты ***money_get*** и ***money_put*** предоставляют функции, выполняющие ввод/вывод на основе формата, определенного фасетом ***money_base***.

Наш простой тип ***Money*** можно использовать для управления форматами ввода/вывода или для хранения денежных значений. В первом случае, мы приводим (другие) типы, используемые для хранения денежных сумм, к ***Money*** перед записью и читаем в переменные ***Money*** перед преобразованием в другие типы. Меньше провоцирует ошибки хранение денежных значений только в переменных типа ***Money***; в этом случае невозможно забыть преобразовать значение в ***Money*** перед его записью, и не возникнут ошибки ввода при попытках чтения денежных величин без учета местных особенностей. Однако внедрение типа ***Money*** в систему, которая для этого не приспособлена, может оказаться невозможным. В таких случаях необходимо применять преобразования (приведения) ***Money*** в операциях ввода и вывода.

Г.4.3.1. Пунктуация денег

Фасет, управляющий представлением денежных величин, ***moneypunct***, естественно похож на фасет, управляющий обыкновенными числами, ***numpunct*** (§ Г.4.2.1):

```
class std::money_base{
public:
enum part {none, space, symbol, sign, value}; // части формата вывода
                                               // {ничего, заполнитель, символ,
                                               // знак, значение}
    struct pattern { char field[4]; };         // спецификация формата
};

template <class Ch, bool International = false>
class std::moneypunct : public locale::facet, public money_base {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit moneypunct(size_t r = 0);

    Ch decimal_point() const;                  // '.' в classic()
    Ch thousands_sep() const;                  // ',' в classic()
    string grouping() const;                   // "" в classic(), то есть без группировки

string_type curr_symbol() const;               // "$" в classic()
    string_type positive_sign() const;         // "" в classic()
    string_type negative_sign() const;         // "-" в classic()

    int frac_digits() const;                   // количество знаков после десятичной
                                               // точки; 2 в classic()
    pattern pos_format() const;                // { symbol, sign, none, value } в classic()
    pattern neg_format() const;                // { symbol, sign, none, value } в classic()

    static const bool intl = International;   // использование международных
                                               // денежных форматов

    static locale::id id;                      // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)

protected:
    ~moneypunct();

    // виртуальные "do_функции" (см. § Г.4.1)
};
```

Средства, предлагаемые ***moneypunct***, в основном предназначены для использования разработчиками реализаций фасетов ***money_put*** и ***money_get*** (§ Г.4.3.2, § Г.4.3.3).

Функции-члены ***decimal_point***(), ***thousands_sep***() и ***grouping***() ведут себя подобно своим собратьям из ***numpunct***.

Члены ***curr_symbol***(), ***positive_sign***() и ***negative_sign***() возвращают строку, используемую в качестве денежного символа (например, ***$***, ***¥***, ***FRF***, ***DKK***), знак плюс и минус соответственно. Если аргумент шаблона ***International*** равен ***true***, член ***intl*** тоже окажется равен ***true***, и будет использоваться «международное» обозначение денежного символа. Так называемое «международное» обозначение является четырехсимвольной строкой. Например:

```
"USD"
"DKK"
"EUR"
```

Последним символом является завершающий ноль. Стандарт ISO-4217 определяет трехбуквенный идентификатор валюты. Когда ***International*** есть ***false***, можно использовать местные денежные символы, такие как ***$***, ***£*** и ***¥***.

Объект типа ***pattern*** (образец), возвращаемый ***pos_format***() или ***neg_format***(), состоит из четырех частей (***part***), определяющих последовательность, в которой отображаются числовое значение, денежный символ, символ знака и заполнитель. Большинство форматов тривиальным образом задаются с использованием этого просто шаблона. Например:

```
+$123.45      // {sign, symbol, space, value}, где positive_sign() возвращает "+"
$+123.45      // {symbol, sign, value, none}, где positive_sign() возвращает "+"
$123.45       // {symbol, sign, value, none}, где positive_sign() возвращает ""
$123.45-      // {symbol, value, sign, none}
-123.45 DKK   // {sign, value, space, symbol}
($123.45)     // {sign, symbol, value, none}, где negative_sign() возвращает "()"
(123.45DKK)   // {sign,value,symbol,none}, где negative_sign() возвращает "()"
```

Представление отрицательного числа с использованием скобок достигается, когда ***negative_sign***() возвращает строку, содержащую два символа (). Первый символ строки знака помещается в позицию, где в образце формата находится ***sign***, а остальная часть строки знака помещается после всех остальных частей образца. Наиболее часто это средство применяется для представления отрицательных чисел в виде значения, заключенного в скобки, как это принято в финансовых кругах, но возможно и другое использование. Например:

```
-$123.45          // {sign, symbol, value, none}, где negative_sign() возвращает "-"
*$123.45 чепуха   // {sign, symbol, value, none}, где negative_sign() возвращает "* чепуха"
```

Каждое из значений ***sign***, ***value*** и ***symbol*** должно появиться в образце ровно один раз. Оставшееся значение может быть либо ***space***, либо ***none***. В том месте, где в образце расположен ***space***, в представлении значения появится по крайней мере один заполнитель, а возможно их будет несколько. Там же, где находится ***none*** (но не в конце образца), в представлении появится ноль или более заполнителей.

Обратите внимание, что эти строгие правила запрещают такие на первый взгляд разумные образцы, как:

```
pattern pat = { sign, value, none, none }; // ошибка: отсутствует symbol
```

Функция ***frac_digits***() задает, где помещается ***decimal_point***(). Часто денежные суммы представляются в самых мелких денежных единицах (§ Г.4.3). Эти единицы обычно в сто раз меньше основных (например, цент составляет одну сотую доллара), поэтому ***frac_digits***(), как правило, возвращает ***2***.

Приведем пример простого формата, оформленного в виде фасета:

```
class My_money_io : public moneypunct<char, true> {
public:
    explicit My_money_io(size_t r = 0) : moneypunct<char, true>(r) { }

    char_type do_decimal_point() const { return "."; }
    char_type do_thousands_sep() const { return ","; }
    string do_grouping() const { return "\003\003\003"; }

    string_type do_curr_symbol() const { return "USD "; }
    string_type do_positive_sign() const { return ""; }
    string_type do_negative_sign() const { return "()"; }

    int do_frac_digits() const { return 2; }      // две цифры после десятичной точки

    pattern do_pos_format() const
    {
        static pattern pat = { sign, symbol, value, none };
        return pat;
    }

    pattern do_neg_format() const
    {
        static pattern pat = { sign, symbol, value, none };
        return pat;
    }
};
```

Этот фасет используется в операциях ввода и вывода ***Money***, определенных в § Г.4.3.2 и § Г.4.3.3.

Имеется и ***_byname*** версия (§ Г.4, § Г.4.1) ***moneypunct***:

```
template <class Ch, bool Intl = false>
class Std::moneypunct_byname : public moneypunct<Ch, Intl> { /* ... */ };
```

Г.4.3.2. Вывод денежных значений

Фасет ***money_put*** записывает денежные суммы в соответствии с форматом, заданным ***moneypunct***. В частности, ***money_put*** предоставляет функции ***put***(), которые помещают подходящим образом отформатированное строковое представление в буфер потока:

```
template<class Ch, class Out = ostreambuf_iterator<Ch> >
class std::money_put : public std::locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;
    typedef basic_string<Ch> string_type;

    explicit money_put(size_t r = 0);
```

```
        // поместить значение "v" в позицию "b" буфера:
        Out put(Out b, bool intl, ios_base& s, Ch fill, long double v) const;
        Out put(Out b, bool intl, ios_base& s, Ch fill, const string_type& v) const;

        static locale::id id; // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)

    protected:
        ~money_put();

        // виртуальные "do_функции" (см. § Г.4.1)
    };
```

Аргументы ***b***, ***s***, ***fill*** и ***v*** используются также, как и для ***put***()-функций фасета ***num_put*** (§ Г.4.2.2). Аргумент ***intl*** определяет, используется ли четырехсимвольный «международный» денежный символ или «местный» символ (§ Г.4.3.1).

Располагая ***money_put***, мы можем определить оператор вывода для ***Money*** (§ Г.4.3) следующим образом:

```
ostream& operator<<(ostream& s, Money m)
{
    ostream::sentry guard(s);          // см. § 21.3.8
    if(!guard) return s;
    try {
        const money_put<char>& f = use_facet<money_put<char> >(s.getloc());
        if(m==static_cast<long double>(m)) { // m можно представить как long double
            if(f.put(s, true, s, s.fill(), m).failed()) s.setstate(ios_base::badbit);
        }
        else {
            ostringstream v;
            v<<m;       // преобразование в строковое представление
            if(f.put(s, true, s, s.fill(), v.str()).failed()) s.setstate(ios_base::badbit);
        }
        catch(...) {
        handle_ioexception(s); // см. § Г.4.2.2
    }
    return s;
}
```

Если для точного представления денежных величин разрядов ***long double*** не хватает, я преобразовываю значение в его строковое представление и вывожу его функцией ***put***(), принимающей строковый аргумент.

Г.4.3.3. Ввод денежных значений

Фасет ***money_get*** читает денежные значения в соответствии с форматом, заданным ***moneypunct***. В частности, ***money_get*** предоставляет функции ***get***(), которые извлекают подходящим образом отформатированные символьные представления из буфера потока:

```
template <class Ch, class In=istreambuf_iterator<Ch> >
class std::money_get : public std::locale::facet {
public:
    typedef Ch char type;
    typedef In iter_type;
```

```
        typedef basic_string<Ch> string_type;

        explicit money_get(size_t r = 0);

        // читаем [b:e) в v с использованием правил форматирования из s,
        // сообщая об ошибках путем установки r:
        In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, long double& v) const;
        In get(in b, In e, bool intl, ios_base& s, ios_base::iostate& r, string_type& v) const;

        static locale::id id;       // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)
    protected:
        ~money_get();

        // виртуальные "do_функции" (см. § Г.4.1)
    };
```

Аргументы *b*, *e*, *s*, *fill* и *v*, используются также, как и *get*()-функциями ***num_get*** (§ Г.4.2.3). Аргумент ***intl*** определяет, будет ли использован «интернациональный» денежный символ или «местный» (§ Г.4.3.1).

Корректно определенная пара фасетов ***money_get*** и ***money_put*** обеспечит вывод в форме, которая может быть обратно считана без ошибок или потери информации. Например:

```
int main()
{
    Money m;
    while(cin>>m) cout << m << "\n";
}
```

Вывод этой простой программы должен быть приемлем в качестве ее ввода. Более того, вывод, полученный в результате второго запуска с выводом первого запуска на входе, должен быть идентичен вводу.

Оператор ввода для ***Money*** может выглядеть так:

```
istream& operator>>(istream&s, Money& m)
{
    istream::sentry guard(s);                          // см. § 21.3.8
    if (guard) try {
        ios_base::iostate state = 0;                   // все хорошо
        istreambuf_iterator< char> eos;
        string str;

        use_facet< money_get< char> >(s.getloc()).get(s, eos, true, state, str);

        if (state==0 || state==ios_base::eofbit) {     // установка значения только при
                                                       //успешном завершении get()
        long int i = strtol(str.c_str(), 0, 0);        //strtol() см. в § 20.4.1
            if (errno==ERANGE)
                state |= ios_base::failbit;
            else
                m = i;   // установка только если преобразование в long int успешно
            s.setstate(state);
        }
    }
    catch (...) {
```

```
            handle_ioexception(s);                    // см. § Г.4.2.2
        }
        return s;
    }
```

Здесь использована функция *get*(), которая читает в строку, т. к. чтение в ***double*** с последующим преобразованием в ***long int*** может привести к потере точности.

Г.4.4. Ввод и вывод дат и времени

К сожалению, стандартная библиотека C++ не предоставляет удовлетворительного типа даты. Однако она наследует от стандартной библиотеки C низкоуровневые средства работы с датами и интервалами времени. Эти средства C являются основой для средств C++, позволяющих работать со временем системно-независимым образом.

В следующих разделах демонстрируется, каким образом представление дат и времени можно сделать зависимым от локализации. Приводится пример того, как определяемый пользователем тип (***Date***) может вписаться в среду потоков ввода/вывода (глава 21) и локализаций (§ Г.2). Реализация ***Date*** демонстрирует технологии, которые могут оказаться полезными при работе со временем, если у вас нет готового типа ***Date***.

Г.4.4.1. Часы и таймеры

На самом низком уровне большинства систем имеется таймер, отсчитывающий очень маленькие интервалы времени. Стандартная библиотека предоставляет функцию ***clock***(), которая возвращает значение зависящего от реализации типа ***clock_t***. Результат, возвращаемый ***clock***(), можно откалибровать с помощью макроса ***CLOCKS_PER_SEC***. При отсутствии надежной утилиты измерения времени, нетрудно написать собственный измерительный цикл наподобие следующего:

```
int main(int argc, char* argv[])          // § 6.1.7
{
    int n = atoi(argv[1]);                // § 20.4.1
    clock_t t1 = clock();
    if(t1==clock_t(-1)) {                 // clock_t(-1) означает "clock() не работает"
        cerr << "извините, нет часов\n";
        exit(1);
    }
    for(int i=0; i<n; i++) do_something(); // измерительный цикл
    clock_t t2 = clock();
    if(t2==clock_t(-1)) {
        cerr << "извините, часы зашкалило\n";
        exit(2);
    }
    cout << "выполнение do_something() " << n << " раз заняло "
        << double(t2-t1)/CLOCKS_PER_SEC << " секунд"
        << " (частота таймера: " << CLOCKS_PER_SEC << " в секунду)\n";
}
```

Явное преобразование ***double***(*t2–t1*) перед операцией деления необходимо, потому что значение ***clock_t*** может быть целым. Точное значение, с которого начинается работа ***clock***(), зависит от реализации; назначением ***clock***() является измерение интерва-

лов времени в течение одного запуска программы. Если имеются *t1* и *t2*, полученные в результате вызовов ***clock***(), ***double***(*t2*–*t1*) / ***CLOCK_PER_SEC*** является наилучшей возможной оценкой времени в секундах между двумя вызовами.

Если у данного процессора отсутствует таймер, или интервал времени слишком велик для измерения, ***clock***() возвращает значение ***clock_t***(–*1*).

Функция ***clock***() предназначена для измерения интервалов длительностью от доли секунды до некоторого количества секунд. Например, если ***clock_t*** означает 32-битный ***int*** со знаком и ***CLOCKS_PER_SEC*** равно ***1000000***, мы можем пользоваться ***clock***() для измерения интервалов от 0 до 2000 секунд (порядка получаса) в микросекундах.

Заметим, что получение разумных результатов измерения времени выполнения программы может оказаться непростым делом. Другие выполняемые на машине программы могут оказывать значительное влияние на время исполнения; время на кэширование и межзадачный обмен трудно предугадать, да и алгоритмы обработки могут зависеть от данных самым неожиданным образом. Пытаясь засечь время, сделайте несколько замеров и отбросьте результаты как недействительные, если они значительно варьируются от запуска к запуску.

Чтобы управиться с более продолжительными интервалами и с календарным временем, стандартная библиотека предоставляет тип ***time_t*** для указания момента времени и структуру ***tm*** для его разделения на составные части:

```
typedef implementation_defined time_t;      // зависящий от реализации
                                            // арифметический тип (§ 4.1.1)
                                            // способный хранить период времени
                                            // зачастую — 32-битное целое

struct tm {
    int tm_sec;     // секунда минуты [0,61]; 60 и 61 зарезервированы
                    // для корректировочных секунд
    int tm_min;     // минута часа [0,59]
    int tm_hour     // час суток [0,23]
    int tm_mday;    // день месяца [1,31]
    int tm_mon;     // месяц года [0,11]; 0 означает январь (внимание: не [1,12])
    int tm_year;    // год от 1900; 0 означает 1900 год, а 102 — 2002 год
    int tm_wday;    // день недели [0,6]; 0 означает воскресенье
    int tm_yday;    // день года [0,365]; 0 означает 1 января
    int tm_isdst;   // часы летнего времени
};
```

Стандарт гарантирует только то, что в ***tm*** имеются перечисленные выше целые члены. Стандарт не гарантирует, что они расположены именно в таком порядке, или что отсутствуют другие поля.

Типы ***time_t*** и ***tm*** вместе с базовыми средствами их использования объявлены в <***ctime***> и <*time.h*>. Примеры:

```
clock_t clock();                        // количество "тиков" с момента запуска
                                        // программы

time_t time(time_t* pt);                // текущее календарное время
double difftime(time_t t2, time_t t1);  // t2–t1 в секундах

tm* localtime(const time_t* pt);        // местное время для *pt
```

```
tm* gmtime(const time_t* pt);          // Время по Гринвичу (GMT), соответствующее
                                       // *pt или 0
// (официально называется скоординированным универсальным временем, UTC)
time_t mktime(tm* ptm)                 // time_t для *ptm или time_t(-1)
char* asctime(const tm* ptm);          // представление времени *ptm в виде С-строки,
                                       // например "Sun Sep 16 01:03:52 1973\n"
char* ctime(const time_t* t) { return asctime(localtime(t)); }
```

Будьте осторожны: как ***localtime***(), так и ***gmtime***() возвращают указатель ***tm**** на статически размещенный объект, так что последующие вызовы функции изменят значение объекта. Либо используйте возвращаемое значение немедленно, либо скопируйте ***tm*** в управляемую вами область памяти. Аналогично, ***asctime***() возвращает указатель на статически размещенный массив символов.

Структура ***tm*** может представлять даты в диапазоне по меньшей мере десятков тысяч лет (примерно [−32000, 32000] в случае минимального размера ***int***). С другой стороны, ***time_t***, как правило, является 32-битным целым (со знаком). Это позволяет ***time_t*** представлять в секундах диапазон длинною чуть более 68 лет в обе стороны от базового года. Этим базовым годом обычно является 1970, точным базовым временем при этом является 0:00, 1 января по Гринвичу. При 32-битном целом (со знаком) ***time_t***, в 2038 году мы окажемся «вне времени», если не изменим ***time_t*** на больший целый тип, что уже сделано на некоторых системах.

Механизм ***time_t*** задуман в первую очередь для представления «близкого к настоящему времени». Таким образом, мы не вправе ожидать, что ***time_t*** сумеет хранить даты вне пределов диапазона [1902, 2038]. Что еще хуже, не все реализации функций времени одинаково воспринимают отрицательное время. Из соображений переносимости значение, которое должно быть представимо и как ***tm***, и как ***time_t***, обязано лежать в диапазоне [1902, 2038]. Чтобы представить даты вне этого диапазона, придется придумать некий дополнительный механизм.

Одно из последствий вышесказанного — возможный сбой ***mktime***(). Если аргумент ***mktime***() не может быть представлен как ***time_t***, будет возвращен индикатор ошибки ***time_t(−1)***.

Хронометраж в «долгоиграющей» программе можно реализовать так:

```
int main(int argc, char* argv[])                    // § 6.1.7
{
        time_t t1 = time(0);
        do_a_lot(argc, argv);
        time_t t2 = time(0);
        double d = difftime(t2, t1);
        cout << "выполнение do_a_lot() заняло " << d << " секунд\n";
}
```

Если аргумент-указатель ***time***() не равен ***0***, текущее время также записывается по адресу указателя на ***time_t***. Если календарное время недоступно (например, в специализированном процессоре), возвращается значение ***time_t(−1)***. Можно попытаться осторожно определить сегодняшнюю дату следующим образом:

```
int main()
{
```

```
        time_t t;
        if(time(&t)==time_t(-1)) {          // time_t(-1) означает, что "time()
                                            // не работает"
            cerr << "Плохое время...\n";
            exit(1)
        }
        tm* gt = gmtime(&t);
        cout << gt->tm_mon + 1 << '/' << gt->tm_mday << '/' << 1900 + gt->tm_year << endl;
}
```

Г.4.4.2. Класс Date

Как отмечалось в § 10.3, маловероятно, чтобы единственный тип ***Date*** смог удовлетворить все потребности. Различные цели использования информации о дате диктуют наличие множества представлений, а календарные сведения до 19-го столетия сильно зависят от капризов истории. Тем не менее, в качестве примера мы можем определить тип ***Date*** примерно как в § 10.3, реализуя его через ***time_t***:

```
class Date {
public:
    enum Month{jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};

    class Bad_date { };                 // исключение

    Date(int dd, Month mm, int yy);
    Date();

    friend ostream& operator<<(ostream& s, const Date& d);
    // ...
private:
    time_t d;    // стандартное представление даты и времени
};

Date::Date(int dd, Month mm, int yy)
{
    tm x = { 0 };
    if(dd<0 || 31<dd) throw Bad_date();     // чрезмерно упрощенно: см § 10.3.1
    x.tm_mday = dd;
    if(mm<jan || dec<mm) throw Bad_date();
    x.tm_mon = mm - 1;                      // tm_mon отсчитывают с нуля
    x.tm_year = yy - 1900;                  // tm_year отсчитывают с 1900
    d = mktime(&x);
}

Date::Date()
{
    d = time(0);                // значение по умолчанию для Date (сегодня)
    if(d == time_t(-1)) throw Bad_date();
}
```

Перед нами стоит задача реализовать зависящие от локализации операторы << и >> для ***Date***.

Г.4.4.3. Вывод даты и времени

Также как и *num_put* (§ Г.4.2), *time_put* предоставляет *put*()-функции для записи в буферы через итераторы:

```
template<class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;

    explicit time_put(size_t r = 0);

    // поместить t в буфер потока s через b, используя формат fmt:
    Out put(Out b, ios_base& s, Ch fill, const tm* t, const Ch* fmt_b, const Ch* fmt_e)
const;

    Out put(Out b, ios_base& s, Ch fill, const tm* t, char fmt, char mod = 0) const
        {return do_put(b, s, fill, t, fmt, mod); }

    static locale::id id;     // идентификатор фасета (§ Г.4, § Г.3, § Г.3.1)

protected:
    ~time_put();

    virtual Out do_put(Out, ios_base&, Ch, const tm*, char, char) const;
};
```

Вызов *put*(*b*, *s*, *fill*, *t*, *fmt_b*, *fmt_e*) помещает информацию о дате из *t* в буфер потока *s* через *b*. При необходимости в качестве заполнителя используется символ *fill*. Формат вывода задается «*printf*()-подобной» строкой форматирования [*fmt_b*, *fmt_e*). Этот *printf*()-подобный формат (§ 21.8) производит выходную строку и может содержать следующие специальные спецификаторы формата:

%*a*	сокращенное название дня недели (например, Sat)
%*A*	полное название дня недели (например, Saturday)
%*b*	сокращенное название месяца (например, Feb)
%*B*	полное название месяца (например, February)
%*c*	дата и время (например, Sat Feb 06 21:46:05 1999)
%*d*	число [01, 31] (например, 06)
%*H*	24-часовой формат времени [00,23] (например, 21)
%*I*	12-часовой формат времени [01,12] (например, 09)
%*j*	день года [001, 366] (например, 037)
%*m*	месяц [01, 12] (например, 02)
%*M*	минута часа [00, 59] (например, 48)
%*p*	a.m./p.m. (до/после полудня), индикатор для 12-часового формата (например, PM)
%*S*	секунда в минуте [00, 61] (например, 40)
%*U*	неделя года [00, 53] начиная с воскресенья (например, 05); первое воскресенье начинает первую неделю
%*w*	день недели [0, 6]; 0 означает воскресенье (например, 6);
%*W*	неделя года [00, 53] начиная с понедельника (например, 05); первый понедельник начинает первую неделю
%*x*	дата (например, 02/06/99)

%*X* время (например, 21:48:40)
%*y* год без века [00, 99] (например, 99)
%*Y* год (например, 1999)
%*Z* индикатор часового пояса (например, EST) если часовой пояс известен

Столь длинный список весьма специальных правил форматирования может послужить аргументом в пользу выбора какой-либо расширяемой системы ввода/вывода. И все-таки, как и большинство специальных нотаций, эта форма записи решает задачу и часто даже удобна.

Кроме перечисленных директив форматирования, большинство реализаций поддерживают «модификаторы», например целые, указывающие ширину поля (§ 21.8), *%10X*. Модификаторы форматов дат и времени не являются частью стандарта C++, но стандарты некоторых платформ, например POSIX, требуют их. Поэтому трудно избежать применения модификаторов, хотя их использование не полностью переносимо.

Подобная *sprintf* (§ 21.8) функция *strftime*() из <*ctime*> или <*time.h*> осуществляет вывод с использованием директив форматирования времени и дат:

```
size_t strftime(char* s, size_t max, const char* format, const tm* tmp);
```

Эта функция помещает максимум *max* символов из **tmp* и *format* в **s*. Например:

```
int main()
{
        char buf[20];       // небрежно: надежда на то, что strftime__() 
                            // никогда не вернет более 20 символов
        time_t t = time(0);
        strftime(buf, 20, "%A\n", localtime(&t));
        cout << buf;
}
```

В среду результатом вывода будет *Wednesday* в локализации по умолчанию *classic*() (§ Г.2.3) и *onsdag* в датской локализации.

Символы, не являющиеся частью задаваемого формата, такие как перевод на новую строку в примере, просто копируются в первый аргумент (*s*).

Когда *put*() обнаруживает символ форматирования *f* (и необязательный модификатор *m*), для реального выполнения форматирования она вызывает виртуальную функцию *do_put*() с соответствующими параметрами: *do_put*(*b, s, fill, t, f, m*).

Вызов *put*(*b, s, fill, t, f, m*) является упрощенной формой *put*(), где символ форматирования (*f*) и символ модификатора (*m*) заданы явно. Таким образом, вызов

```
const char fmt[] = "%10X";
put(b, s, fill, t, fmt, fmt+sizeof(fmt));
```

можно сократить:

```
put(b, s, fill, t, 'X', 10);
```

Если формат содержит многобайтные символы, он должен и начинаться, и заканчиваться в состоянии по умолчанию (§ Г.4.6).

Мы можем воспользоваться *put*(), чтобы реализовать зависящий от локализации оператор вывода для *Date*:

```
ostream& operator<<(ostream& s, const Date& d)
{
        ostream::sentry guard(s);          // см. § 21.3.8
        if(!guard) return s;

        tm* tmp = localtime(&d.d);
        try {
                if(use_facet<time_put<char> >(s.getloc()).put(s, s, s.fill(), tmp, 'x').failed())
                        s.setstate(ios_base::failbit);
        }
        catch(...) {
                handle_ioexception(s);     // см. § Г.4.2.2
        }
        return s;
}
```

Ввиду отсутствия стандартного типа ***Date***, отсутствует и форма ввода/вывода дат по умолчанию. В примере я задал формат %***x*** путем передачи символа '***x***' в качестве символа форматирования. Формат %***x*** является форматом по умолчанию для ***get_time***() (§ Г.4.4.4), так что ничего «более стандартного», вероятно, и не придумать. См. в § Г.4.4.5 пример использования альтернативных форматов.

Кроме того, имеется и ***_byname*** версия (§ Г.4, § Г.4.1) функции ***time_put***:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put_byname : public time_put<Ch, Out> { /* ... */ };
```

Г.4.4.4. Ввод дат и времени

Как всегда, ввод сложнее вывода. При написании кода для вывода значения часто имеется выбор из различных форматов. А при написании кода для ввода мы еще должны уметь обрабатывать ошибки и решать, с каким из нескольких возможных форматов мы столкнулись.

Фасет ***time_get*** реализует ввод дат и времени. Идея состоит в том, что ***time_get*** конкретной локализации может читать время и даты, выведенные ***time_put*** этой же локализации. Однако ввиду отсутствия стандартных классов даты и времени, программист может воспользоваться локализацией для вывода в соответствии со множеством форматов. Например, все следующие представления могут быть получены при помощи единственной инструкции вывода с использованием ***time_put*** (§ Г.4.4.5) различных локализаций:

January 15th 1999
Thursday 15th January 1999
15 Jan 1999AD
Thurs 15/1/99

Стандарт C++ поощряет разработчиков ***time_get*** воспринимать форматы дат и времени в соответствии с POSIX и другими стандартами. Проблема состоит в том, что сложно стандартизовать потребность в чтении дат и времени во всевозможных форматах, принятых в рамках данной культуры. Полезно поэкспериментировать, чтобы узнать, что предоставляет данная локализация (§ Г.6[8]). Если формат не воспринимается, программист может реализовать подходящий альтернативный фасет ***time_get***.

Стандартный фасет ввода времени ***time_get*** является производным от ***time_base***:

```
class std::time_base {
public:
        enum dateorder {
                no_order,       // не упорядочено; возможно, много элементов
                                // (таких как день недели)
                dmy,            // день–месяц–год
                mdy,            // месяц–день–год
                ymd,            // год–месяц–день
                ydm             // год–день–месяц
        };
};
```

Разработчик может воспользоваться этим перечислением для упрощения разбора форматов даты.

Подобно ***num_get***, функция ***time_get*** осуществляет доступ к своему буферу при помощи пары итераторов ввода:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class time_get : public locale::facet, public time_base {
public:
        typedef Ch char_type;
        typedef In iter_type;

        explicit time_get(size_t r=0);

        dateorder date_order() const { return do_date_order(); }

        // чтение [b,e) в d с использованием правил форматирования из s и сообщением
        // об ошибке путем установки r:
        In get_time(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
        In get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
        In get_year(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

        In get_weekday(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
        In get_monthname(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

        static locale::id id;       // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)
protected:
        ~time_get();

        // виртуальные "do_функции" (см. § Г.4.1)
};
```

Функция ***get_time***() вызывает ***do_get_time***(). По умолчанию ***get_time***() читает время, как оно выводится ***time_put::put***() данной локализации с использованием формата ***%X*** (§ Г.4.4). Точно так же функция ***get_date***() вызывает ***do_get_date***(). По умолчанию, ***get_date***() читает дату так, как она выводится ***time_put::put***() данной локализации с использованием формата ***%x*** (§ Г.4.4).

Таким образом, простейший оператор ввода для ***Date*** выглядит примерно так:

```
istream& operator>>(istream& s, Date& d)
{
        istream::sentry guard(s);           // см. § 21.3.8
```

```
        if(!guard) return s;

        ios_base::iostate res = 0;
        tm x = { 0 };
        istreambuf_iterator<char, char_traits<char> > end;
        try {
                use_facet<time_get<char> >(s.getloc()).get_date(s, end, s, res, &x);
        }
        catch(...) {
                handle_ioexception(s); // см. § Г.4.2.2
                return s;
        }
        d = Date(x.tm_mday, Date::Month(x.tm_mon+1), x.tm_year+1900);
        return s;
}
```

Вызов *get_date(s, end, s, res, &x)* полагается на два неявных преобразования из *istream*: в качестве первого аргумента, *s* используется для создания *istreambuf_iterator*; в качестве третьего аргумента, *s* преобразуется в базовый класс *ios_base*.

Этот оператор ввода будет корректно работать с датами из диапазона, представимого *time_t*. Пример тривиального теста:

```
int main()
try {
        Date today;
        cout << today <<endl;                    // запись в формате %x
        Date d(12, Date::may, 1998);

        cout << d << endl;
        Date dd;
        while (cin >> dd) cout << dd << endl;    // чтение дат в формате %x
}
catch (Date::Bad_date) {
        cout << "выход: неправильная дата\n";
}
```

Имеется и *_byname* версия (§ Г.4, § Г.4.1) функции *time_get*:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::time_get_byname : public time_get<Ch, In> { /* ... */ };
```

Г.4.4.5. Более гибкий класс Date

Попытка воспользоваться *Date* из § Г.4.4.2 с вводом/выводом из § Г.4.4.3 и § Г.4.4.4 вскоре выявила бы ограниченность класса *Date*:

[1] Он работает только с датами, представимыми в форме *time_t*, что как правило означает диапазон [1970, 2038].

[2] Он воспринимает даты только в стандартном формате — каким бы тот ни был.

[3] Его реакция на ошибки ввода неприемлема.

[4] Он поддерживает только потоки *char* (а не потоки произвольных символьных типов).

Более интересный и полезный оператор ввода должен воспринимать более широкий диапазон дат, распознавать несколько распространенных форматов, надежно и со-

держательно сообщать об ошибках. Для достижения этих целей нам придется расстаться с *time_t*:

```
class Date {
public:
    enum Month {jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};

    struct Bad_date {
        const char* why;            // в чем проблема?
        Bad_date(const char* p) : why(p) { }
    };

    Date(int dd, Month mm, int yy, int day_of_week = 0);
    Date();

    void make_tm(tm* t) const;  // поместить tm-представление Date в *t
    time_t make_time_t() const; // возвратить time_t-представление Date

    int year() const { return y; }
    Month month() const { return m; }
    int day() const { return d; }

    // ...
private:
    char d;
    Month m;
    int y;
};
```

Для простоты я вернулся к представлению (*d*, *m*, *y*) (§ 10.2).

Конструктор можно определить следующим образом:

```
Date::Date(int dd, Month mm, int yy, int day_of_week) : d(dd), m(mm), y(yy)
{
    if(d==0 && m==Month(0) && y==0) return;     // Date(0,0,0) является "нулевой
                                                // датой"
    if(mm<jan || dec<mm) throw Bad_date("недопустимый номер месяца");
    if(dd<1 || 31<dd)                           // слишком упрощенно; см. § 10.3.1
        throw Bad_date("недопустимое число");
    if(day_of_week && day_in_week(yy, mm, dd) != day_of_week)
        throw Bad_date("недопустимый день недели");
}

Date::Date() : d(0), m(0), y(0) { }             // "нулевая дата"
```

Алгоритм ***day_in_week***() (день недели) нетривиален и не имеет непосредственного отношения к механизмам локализации, поэтому я оставил его за пределами книги. Если вам нужен этот алгоритм, он наверняка отыщется где-нибудь в вашей системе.

Для типов наподобие ***Date*** всегда полезно располагать операциями сравнения:

```
bool operator==(const Date& x, const Date& y)
{
    return x.year()==y.year() && x.month()==y.month() && x.day()==y.day();
}
```

```
bool operator!=(const Date& x, const Date& y)
{
        return !(x==y);
}
```

Поскольку мы расстались со стандартными форматами *tm* и *time_t*, нам потребуются функции преобразования для взаимодействия с программами, которые ожидают эти типы:

```
void Date::make_tm(tm* p) const   // поместить дату в *p
{
        tm x = { 0 };
        *p = x;
        p->tm_year = y - 1900;
        p->tm_mday = d;
        p->tm_mon = m - 1;
}

time_t Date::make_time_t() const
{
        if (y<1970 || 2038<y)                    // упрощенный вариант
                throw Bad_date("дата находится вне диапазона time_t");
        tm x;
        make_tm(&x);
        return mktime(&x);
}
```

Г.4.4.6. Задание формата даты

C++ не определяет стандартный формат вывода дат (%*x* является наиболее вероятным претендентом; § Г.4.4.3). Но даже если бы стандарт и существовал, нам бы наверняка понадобились альтернативы. Их можно обеспечить, предоставив «формат по умолчанию» и способ изменить его. Например:

```
class Date_format {
        static char fmt[];          // формат по умолчанию
        const char* curr;           // указатели на текущий формат
        const char* curr_end;
public
        Date_format() : curr(fmt), curr_end(fmt + strlen(fmt)) { }

        const char* begin() const { return curr; }
        const char* end() const { return curr_end; }

        void set(const char* p, const char* q) { curr = p; curr_end = q; }
        void set(const char* p) { curr = p; curr_end = curr + strlen(p); }

        static const char* default_fmt() { return fmt; }
};

const char Date_format::fmt[] = "%A, %B %d, %Y";  // например, Friday, February 5, 1999

Date_format date_fmt;
```

Чтобы иметь возможность воспользоваться форматом ***strftime***() (§ Г.4.4.3), я воздержался от параметризации класса ***Date_format*** используемым типом символов. Таким образом, данное решение допускает только те формы представления дат, формат которых может быть выражен как ***char***[]. Кроме того, я воспользовался глобальным объектом форматирования (***date_fmt***) для предоставления формата ***Date*** по умолчанию. За счет того, что значение ***date_fmt*** можно изменять, мы получаем грубый способ управления форматированием ***Date***, подобный тому, как для управления форматированием может использоваться ***global***() (§ Г.2.3).

Более общим решением было бы определение фасетов ***Date_in*** и ***Date_out*** для управления чтением и записью в поток. Это подход демонстрируется в § Г.4.4.7.

При наличии ***Date_format***, ***Date***::***operator***<<() можно написать следующим образом:

```
template<class Ch, class Tr>
basic_ostream<Ch, Tr>& operator<<(basic_ostream<Ch, Tr>& s, const Date& d)
// запись в формате, заданном пользователем
{
        typename basic_ostream<Ch, Tr>::sentry guard(s);    // см. § 21.3.8
        if (!guard) return s;

        tm t;
        d.make_tm(&t);
        try {
                const time_put<Ch>& f = use_facet<time_put<Ch> >(s.getloc());
                if (f.put(s, s, s.fill(), &t, date_fmt.begin(), date_fmt.end()).failed())
                        s.setstate(ios_base::failbit);
        }
        catch (...) {
                handle_ioexception(s);          // см. § Г.4.2.2
        }
        return s;
}
```

Я мог бы воспользоваться ***has_facet*** чтобы удостовериться, что в локализации имеется фасет ***time_put<Ch>***. Однако в данном случае проще выглядит вариант с перехватом исключения, сгенерированного ***use_facet***.

Приведем простую тестовую программу, управляющую форматом вывода при помощи ***date_fmt***:

```
int main()
try {
        Date dd;
        while (cin >> dd && dd != Date()) cout << dd << endl;    // запись в формате
                                                                 // по умолчанию date_fmt
        date_fmt.set("%Y/%m/%d");
        while (cin >> dd && dd != Date()) cout << dd << endl;    // запись в формате
                                                                 // "%Y/ %m/%d"
}
catch (Date::Bad_date e) {
        cout << "перехвачена неправильная дата: " << e.why << endl;
}
```

Г.4.4.7. Фасет ввода даты

Как всегда, ввод немного сложнее вывода. Однако ввиду того, что интерфейс к низкоуровневому вводу ограничен *get_date*(), и благодаря тому, что определенный в § Г4.4.4 *operator*>>() для *Date* не осуществляет непосредственного доступа к представлению *Date*, мы могли бы воспользоваться этим *operator*>>() без изменений. Вот шаблонная версия, соответствующая *operator*<<():

```
template<class Ch, class Tr>
istream<Ch, Tr>& operator>>(istream<Ch, Tr>& s, Date& d)
{
        typename istream<Ch, Tr>::sentry guard(s);
        if(!guard) return s;

        ios_base::iostate res = 0;
        tm x = { 0 };
        istreambuf_iterator<Ch, Tr> end;
        try {
                use_facet<time_get<Ch> >(s.getloc()).get_date(s, end, s, res, &x);
        }
        catch (...) {
                handle_ioexception(s);          // см. § Г.4.2.2
                return s;
        }
        d = Date(x.tm_mday, Date::Month(x.tm_mon+1), x.tm_year+1900, x.tm_wday);
        if (res == ios_base::badbit) s.setstate(res);
        return s;
}
```

Этот оператор ввода *Date* вызывает *get_date*() фасета *time_get* из потока *istream*. Значит можно обеспечить иную и более гибкую форму ввода путем определения нового фасета, производного от *time_get*:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class Date_in : public std::time_get<Ch, In> {
public:
        Date_in(size_t r = 0) : std::time_get<Ch, In>(r) { }

protected:
        In do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const;

private:
        enum Vtype {novalue, unknown, dayofweek, month};// нет значения, неизвестное
                                                        // значение, день недели, месяц
        In getval(In b, In e, ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const;
};
```

Функция *getval*() должна прочитать год, месяц, число, возможно день недели и трансформировать полученное в *tm*.

Названия месяцев и дней недели зависят от локализации. Следовательно, мы не можем упоминать их непосредственно в нашей функции ввода. Так что мы будем распознавать месяцы и дни недели путем вызова функций, которые *time_get* предоставляет для этой цели: *get_monthname*() и *get_weekday*() (§ Г.4.4.4).

Год, число и возможно месяц, представлены в виде целых чисел. К сожалению, на числе не написано, означает ли оно день, месяц или еще что. Например, *7* может означать «июль», или «7-е число», или даже «2007 год». Для разрешении подобных двусмысленностей предназначена ***date_order***() из ***time_get***().

Стратегия ***Date_in*** состоит в чтении значений, их классификации и вызове ***date_order***(), чтобы узнать, имеют ли (и какой) смысл введенные значения. Закрытая функция ***getval***() осуществляет непосредственное чтение буфера потока ввода и начальную классификацию:

```
template<class Ch, class In>
In Date_in< Ch, In>::getval(In b, In e, ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const
    // Читаем часть даты: число, день недели или месяц.
    // Пропускаем заполнители и пунктуацию.
{
    const ctype< Ch>& ct = use_facet< ctype< Ch> >(s.getloc()); // ctype определен в Г.4.5
    Ch c;

    *res = novalue;                          // значение не найдено

    for (;;) {                               // пропускаем заполнители и пунктуацию
        if (b == e) return e;
        c = *b;
        if (!(ct.is(ctype_base::space, c) || ct.is(ctype_base::punct, c))) break;
        ++b;
    }

    if (ct.is(ctype_base::digit, c)) { // читаем целые, игнорируя punct
        int i = 0;

        do {  // преобразуем цифру из произвольного символьного типа
              // в ее числовое значение:
            static char const digits[] = "0123456789";
            i = i*10 + find(digits, digits+10, ct.narrow(c,' ')) - digits;
            c = *++b;
        } while (ct.is(ctype_base::digit, c));

        *v = i;
        *res = unknown; // целое, но неизвестно, что оно означает
        return b;
    }

    if (ct.is(ctype_base::alpha, c)) {       // поиск названия месяца или дня недели
        basic_string< Ch> str;
        while (ct.is(ctype_base::alpha, c)) { // читаем символы в строку
            str += c;
            if (++b == e) break;
            c = *b;
        }

        tm t;
        basic_stringstream< Ch> ss(str);
        typedef istreambuf_iterator< Ch> SI;  // тип итератора для буфера
                                              // потока ss
```

```
            get_monthname(ss.rdbuf(), SI(), s, r, &t); // читаем из находящегося в памяти
                                                       // буфера потока
            if ((r&(ios_base::badbit|ios_base::failbit))==0) {
                *v= t.tm_mon;
                *res = month;
                r = 0;
                return b;
            }

            r = 0;              // очищаем состояние перед попыткой повторного чтения
            get_weekday(ss.rdbuf(), SI(), s, r, &t);    // читаем из находящегося в памяти
                                                        // буфера потока
            if ((r&ios_base::badbit)==0) {
                *v = t.tm_wday;
                *res = dayofweek;
                r = 0;
                return b;
            }
        }
        r|= ios_base::failbit;
        return b;
    }
```

Главный трюк в том, чтобы отличить месяцы от дней недели. Мы читаем через итераторы ввода, поэтому мы не может прочитать [*b*,*e*) дважды, пытаясь прочитать первый раз месяц, а второй — день недели. С другой стороны, мы не можем принять решение, имея информацию в каждый момент времени только об одном символе, потому что только ***get_monthname***() и ***get_weekday***() знают, какая последовательность является названием месяца и дня недели в данной локализации. Решение, которое я выбрал, состоит в чтении строк символов в строку ***string***, создании потока ***stringstream*** из этой строки и повторном чтении из буфера ***streambuf*** созданного потока.

Для фиксации ошибок непосредственно устанавливаются биты состояния, такие как ***ios_base::badbit***. Это необходимо, потому что более удобные функции манипулирования состоянием потока, такие как ***clear***() и ***set_state***(), определены в ***basic_ios***, а не в его базовом классе ***ios_base*** (§ 21.3.3). Затем при необходимости оператор >> использует информацию об ошибках, сообщенную ***get_date***(), для переустановки состояния потока ввода.

Наличие ***getval***() позволяет сначала прочитать значения и только затем проверить, осмыслены ли они. Функция ***date_order***() может иметь решающее значение:

```
template<class Ch, class In >
In Date_in< Ch, In>::do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const
// необязательный день недели, за которым следует ymd, dmy, mdy или ydm
{
    int val[3];                         // для значений числа, месяца и года
                                        // в некотором порядке
    Vtype res[3] = { novalue };         // для классификации значений

    for (int i=0; b!=e && i<3; ++i) {   // прочитать число, месяц и год
        b = getval(b, e, s, r, *val[i], *res[i]);
        if (r) return b;                // ошибка
```

```
            if (res[i]==novalue) {              // неполная дата
                r |= ios_base::badbit;
                return b;
            }
            if (res[i]==dayofweek) {
                tmp->tm_wday = val[i];
                --i;                            // осторожно: не день, не месяц и не год
            }
        }
        time_base::dateorder order = date_order();  // теперь попытаемся придать
                                                    // смысл прочитанным значениям
        if (res[0] == month) {                      // mdy или ошибка
            // ...
        }
        else if (res[1] == month) {
            tmp->tm_mon = val[1];
            switch(order) {
            case dmy:
                tmp->tm_mday = val[0];
                tmp->tm_year = val[2];
                break;
            case ymd:
                tmp->tm_year = val[0];
                tmp->tm_mday = val[2];
                break;
            default:
                r |= ios_base::badbit;
                return b;
            }
        }
        else if (res[2] == month) {             // ydm или ошибка
            // ...
        }
        else {                                  // соответствует dateorder или ошибка
            // ...
        }
        tmp->tm_year -= 1900;                   // привести базовый год в соответствие
                                                // соглашениям tm
        return b;
    }
```

Я опустил фрагменты кода, не имеющие отношение к демонстрации локализаций, дат и управления вводом. Написание лучших и более общих функций ввода даты оставлено в качестве упражнений (§ Г.6[9–10]).

Вот простая тестовая программа:

```
int main()
try {
    cin.imbue(loc(locale(), new Date_in));      // чтение дат с помощью Date_in
```

```
        while (cin >> dd && dd != Date()) cout << dd << endl;
    }
    catch (Date::Bad_date e) {
        cout << "Некорректная дата: " << e.why << endl;
    }
```

Обратите внимание, что ***do_get_date()*** воспримет и бессмысленные даты, такие как

```
Thursday October 7, 1998        // 7 октября 1998 года, четверг
```

и

```
1999/Feb/31
```

Проверка сочетаемости года, месяца, числа и (необязательного) дня недели осуществляется в конструкторе ***Date***. Именно класс ***Date*** должен знать, что является корректной датой, ***Date_in*** не обязан разделять это знание.

Можно написать ***getval()*** или ***do_get_date()*** таким образом, чтобы они строили догадку о смысле числовых значений. Например очевидно, что

```
12 мая 1922
```

означает не 1922 число 12 года. То есть мы могли бы «догадаться», что числовое значение, которое не может быть числом введенного месяца, является годом. Такие «догадки» могут оказаться полезными в конкретном ограниченном контексте. Однако они не являются «волшебной палочкой» для более общих случаев:

```
12 май 15
```

может указывать на любой год из следующих: 12, 15, 1912, 1915, 2012 или 2015. Иногда лучшим подходом является расширение нотации подсказками, которые однозначно определяют год и число. Например, «***1-е***» и «***15-е***» очевидно являются числами. Аналогично, «***751 до н.э.***») и «***1453 н.э.***» явно означают годы.

Г.4.5. Классификация символов

При чтении символов часто возникает задача их классификации с целью придания смысла прочитанному. Например, чтобы прочесть число, процедура ввода должна знать, какие символы являются цифрами. Это похоже на пример из § 6.1.2, который демонстрирует использование стандартных функций классификации символов при синтаксическом разборе ввода.

Естественно, классификация символов зависит от используемого алфавита. Для решения задач классификации символов в локализации имеется фасет ***ctype***.

Символ классифицируется в соответствии с перечислением по имени ***mask***:

```
class std::ctype_base {
public:
    enum mask {                 // действительные значения зависят от реализации
        space = 1,              // разделители (в локализации "C": ' ', '\n', '\t', ...)
        print = 1<<1,           // символы управления печатью
        cntrl = 1<<2,           // управляющие символы
        upper = 1<<3,           // символы в верхнем регистре
        lower = 1<<4,           // символы в нижнем регистре
```

```
            alpha = 1<<5,          // буквы алфавита
            digit = 1<<6,          // десятичные цифры
            punct = 1<<7,          // символы пунктуации
            xdigit = 1<<8,         // шестнадцатеричные цифры
            alnum=alpha | digit,   // алфавитно-цифровые символы
            graph=alnum | punct
        };
};
```

Тип ***mask*** не зависит от конкретного типа символов. Поэтому перечисление помещено в (не являющийся шаблоном) базовый класс.

Очевидно, что ***mask*** отражает традиционную для C и C++ классификацию (§ 20.4.1). Однако для различных наборов символов различные значения символов попадут в разные классы. Например, для набора символов ASCII целое значение ***125*** означает символ '}', являющийся знаком пунктуации (***punct***). А в датском национальном наборе символов ***125*** означает гласную «***å***», которая в датской локализации должна классифицироваться как ***alpha***.

Классификация называется «маской» (mask), потому что традиционной эффективной реализацией классификации символов для небольших наборов символов является таблица, в которой для каждого элемента хранится битовое представление классификации. Например:

```
table['a'] == lower | alpha | xdigit
table['1'] == digit
table[' '] == space
```

При такой реализации ***table***[***c***] & ***m*** отлично от нуля, если ***c*** является ***m***, и ***0*** — в противном случае.

Фасет ***ctype*** определяется следующим образом:

```
template <class Ch>
class std::ctype : public locale::facet, public ctype_base {
public:
    typedef Ch char_type;
    explicit ctype(size_t r = 0);

    bool is(mask m, Ch c) const;  // "c" является "m"?

    // поместить классификацию каждого Ch из [b:e) в v:
    const Ch* is(const Ch* b, const Ch* e, mask* v) const;

    const Ch* scan_is(mask m, const Ch* b, const Ch* e) const;    // найти m
    const Ch* scan_not(mask m, const Ch* b, const Ch* e) const; // найти не m
    Ch toupper(Ch c) const;
    const Ch* toupper(Ch* b, const Ch* e) const;                  // преобразовать [b:e)
    Ch tolower(Ch c) const;
    const Ch* tolower(Ch* b, const Ch* e) const;

    Ch widen(char c) const;
    const char* widen(const char* b, const char* e, Ch* b2) const;
    char narrow(Ch c, char def) const;
    const Ch* narrow(const Ch* b, const Ch* e, char def, char* b2) const;

    static locale::id id;      // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)
```

```
protected:
        ~ctype();

        // виртуальные "do_функции" (см. § Г.4.1)
};
```

Вызов *is*(*m*, *c*) осуществляет проверку принадлежности символа *c* к классификации *m*. Например:

```
int count_spaces(const string& s, const locale& loc)
{
        const ctype<char>& ct = use_facet<ctype<char> >(loc);
        int i = 0;
        for(string::const_iterator p = s.begin(); p != s.end(); ++p)
                if (ct.is(ctype_base::space,*p)) ++i;          // разделитель в соответствии с ct
        return i;
}
```

Обратите внимание, что *is*() можно также воспользоваться для проверки принадлежности символа к одной из нескольких классификаций:

```
ct.is(ctype_base::space | ctype_base::punct, c);          // является ли с в ct разделителем
                                                          // или знаком препинания?
```

Вызов *is*(*m*, *b*, *v*) определяет классификацию каждого символа из [*b*, *e*) и помещает ее в соответствующую позицию массива *v*.

Вызов *scan_is*(*m*, *b*, *e*) возвращает указатель на первый символ в [*b*, *e*), являющийся *m*. Если ни один символ не относится к классификации *m*, возвращается *e*. Как всегда в случае стандартных фасетов, открытые функции-члены реализованы путем вызова своих виртуальных «*do*_функций» . Простая реализация может выглядеть так:

```
template <class Ch>
const Ch* std::ctype<Ch>::do_scan_is(mask m, const Ch* b, const Ch* e) const
{
        while (b!=e && !is(m,*b)) ++b;
        return b;
}
```

Вызов *scan_not*(*m*, *b*, *e*) возвращает указатель на первый символ из [*b*, *e*), который не является *m*. Если все символы принадлежат классификации *m*, возвращается *e*.

Вызов *toupper*(*c*) возвращает *c* в верхнем регистре, если это возможно в используемом наборе символов, и сам *c* — в противном случае.

Вызов *toupper*(*b*, *e*) преобразует каждый символ диапазона [*b*, *e*) в верхний регистр и возвращает *e*. Простая реализация может выглядеть следующим образом:

```
template <class Ch>
const Ch* std::ctype<Ch>::to_upper(Ch* b, const Ch* e)
{
        for ( ; b!=e; ++b) *b = toupper(*b);
        return e;
}
```

Функция *tolower*() аналогична *toupper*() за исключением того, что она преобразует в нижний регистр.

Вызов *widen(c)* преобразует символ *c* в соответствующее значение *Ch*. Если набор символов *Ch* предоставляет несколько символов, соответствующих *c*, стандарт определяет, что должно использоваться «простейшее разумное преобразование». Например,

```
wcout << use_facet<ctype<wchar_t> >(wcout.getloc()).widen('e');
```

выведет разумный эквивалент символа *e* в локализации потока *wcout*.

С помощью *widen()* можно осуществлять и преобразования между не связанными друг с другом представлениями символов, такими как ASCII и EBCDIC. Для примера предположим, что существует локализация *ebcdic*:

```
char EBCDIC_e = use_facet<ctype<char> >(ebcdic).widen('e');
```

Вызов *widen(b, e, v)* берет каждый символ из диапазона [*b*, *e*) и помещает «расширенную» версию в соответствующую позицию массива *v*.

Вызов *narrow(ch, def)* выдает значение *char*, соответствующее символу *ch* типа *Ch*. И снова используется «простейшее разумное преобразование». Если не существует *char*, соответствующего *Ch*, возвращается *def*.

Вызов *narrow(b, e, def, v)* берет каждый символ из диапазона [*b*, *e*) и помещает «суженную» версию в соответствующую позицию массива *v*.

Общая идея состоит в том, что *narrow()* преобразует из большего набора символов в меньший, а *widen()* выполняет обратную операцию. Для символа *c* из меньшего набора символов мы ожидаем, что:

```
c == narrow(widen(c), 0)      // не гарантируется
```

Так и есть при условии, что символ *widen(c)*, имеет только одно представление в «меньшем наборе символов». А это не гарантируется. Если символы *char* не являются подмножеством большего набора символов (*Ch*), следует ожидать нестыковок и потенциальных проблем в коде, работающем с символами в общем виде.

Точно так же и для символа *ch* из большего набора символов мы могли бы ожидать:

```
widen(narrow(ch, def)) == ch || widen(narrow(ch, def))==widen(def)    // не гарантируется
```

Однако, хотя часто так и бывает, нельзя дать подобных гарантий для символа, представляемого несколькими значениями в большем наборе символов, но только одним — в меньшем. Например, цифра *7* часто имеет несколько различных представлений в большем наборе символов. Причина в том, что как правило, большие наборы символов содержат несколько обычных наборов символов в качестве подмножеств, причем символы из меньших наборов повторяются для удобства преобразования.

Для каждого символа из основного набора (§ В.3.3) гарантируется, что

```
widen(narrow(ch_lit, 0)) == ch_lit
```

Например:

```
widen(narrow('x', 0)) == 'x'
```

Функции *narrow()* и *widen()* соблюдают классификацию символа, когда это возможно. Например, если *is(alpha, c)*, то *is(alpha, narrow(c, 'a'))* и *is(alpha, widen(c))*, при условии что *alpha* является корректной маской для используемой локализации.

Главной целью использования фасета *ctype* вообще и функций *narrow*() и *widen*() в частности является написание кода, осуществляющего ввод/вывод и обработку строк для любого набора символов; то есть мы хотим обеспечить общность кода в отношении наборов символов. Подразумевается, что реализации *iostream* решающим образом зависят от описанных в данном разделе средств. Полагаясь на <*iostream*> и <*string*>, пользователь, как правило, может избежать непосредственного использования фасета *ctype*.

Имеется и *_byname* версия (§ Г.4, § Г.4.1) *ctype*:

```
template<class Ch> class std::ctype_byname : public ctype<Ch> { /* ... */ };
```

Г.4.5.1. Некоторые удобные интерфейсы

Наиболее типичным использованием фасета *ctype* является проверка принадлежности символа к некоторой классификации. Для решения этой задачи имеется следующий набор функций:

```
template <class Ch> bool isspace(Ch c, const locale& loc);
template <class Ch> bool isprint(Ch c, const locale& loc);
template <class Ch> bool iscntrl(Ch c, const locale& loc);
template <class Ch> bool isupper(Ch c, const locale& loc);
template <class Ch> bool islower(Ch c, const locale& loc);
template <class Ch> bool isalpha(Ch c, const locale& loc);
template <class Ch> bool isdigit(Ch c, const locale& loc);
template <class Ch> bool ispunct(Ch c, const locale& loc);
template <class Ch> bool isxdigit(Ch c, const locale& loc);
template <class Ch> bool isalnum(Ch c, const locale& loc);
template <class Ch> bool isgraph(Ch c, const locale& loc);
```

Эти функции тривиально реализуются при помощи фасета *use_facet*. Например:

```
template<class Ch>
inline bool isspace(Ch c, const locale& loc)
{
    return use_facet<ctype<Ch> >(loc).is(space, c);
}
```

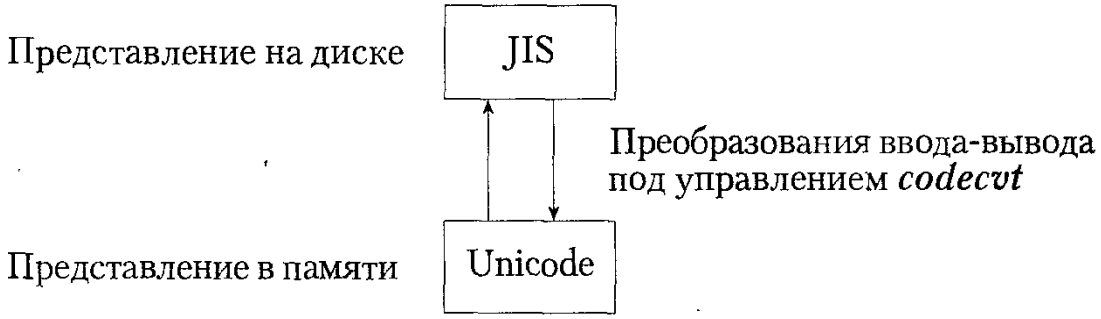
Версии перечисленных функций с одним аргументом, представленные в (§ 20.4.2), просто являются теми же самыми функциями для текущей глобальной локализации C (не для глобальной локализации C++ *locale*()). За исключением редких случаев, когда глобальные локализации C и C++ различаются (§ Г.2.3), мы можем рассматривать функции с одним аргументом как двухаргументные функции, использующие *locale*(). Например:

```
inline int isspace(int i)
{
    return isspace(i, locale());          // это почти правда
}
```

Г.4.6. Преобразование кода символа

Иногда представление символов, хранящихся в файле, отличается от желательного представления этих же символов в памяти. Например, японские символы часто хра-

нят в файлах, в которых особые индикаторы («shifts» — «смещения») указывают, к какому из четырех распространенных наборов символов (kanji, katakana, hiragana или romaji) принадлежит данная последовательность символов. Такой способ несколько неуклюж, так как значение каждого байта зависит от «состояния смещения» (shift state). Зато экономится память, потому что только kanji требует более одного байта на символ. В памяти компьютера символы проще обрабатывать, когда они представлены в виде многобайтного набора символов, в котором размеры представления всех символов одинаковы. Такие символы (например, коды Unicode) обычно хранятся в виде расширенных символов (***wchar_t***; § 4.3). Вот почему фасет ***codecvt*** предоставляет механизм преобразования символов из одного представления в другое в процессе их чтения и записи. Например:



Этот механизм преобразования кода является достаточно общим для реализации произвольных преобразований представлений символов. Он позволяет писать программы, использующие удобное внутреннее представление символов (хранимых в виде ***char***, ***wchar_t*** или как угодно) и воспринимающие различные представления символов потока, настраивая локализацию потоков ввода/вывода. Альтернативой является модификация самой программы или преобразование файлов из одного множества форматов в другое.

Фасет ***codecvt*** обеспечивает преобразования между различными наборами символов при перемещении символа между буфером потока и внешним устройством:

```
class std::codecvt_base {
public:
    enum result { ok, partial, error, noconv };      // индикаторы результата
                                                     // (удачно, частично, ошибка,
                                                     // без преобразования)
};

template <class I, class E, class State>
class std::codecvt : public locale::facet, public codecvt_base {
public:
    typedef I intern_type;
    typedef E extern_type;
    typedef State state_type;

    explicit codecvt(size_t r = 0);

    result in(State&, const E* from, const E* from_end, const E*& from_next,   // чтение
        I* to, I* to_end, I*& to_next) const;
```

```
        result out(State&, const I* from, const I* from_end, const I*& from_next,    // запись
                   E* to, E* to_end, E*& to_next) const;

        result unshift(State&, E* to, E* to_end, E*& to_next) const;    // конец последова-
                                                                        // тельности символов

        int encoding() const throw();          // характеризует основные свойства
                                               // кодирования
    bool always_noconv() const throw();        // можно выполнять ввод/вывод
                                               // без преобразования кодов?

        int length(const State&, const E* from, const E* from_end, size_t max) const;
        int max_length() const throw();        // максимально возможная length()

        static locale::id id;                  // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)

    protected:
        ~codecvt();

        // виртуальные "do_функции" (см. § Г.4.1)
    };
```

Фасет *codecvt* используется *basic_filebuf* (§ 21.5) для чтения и записи символов. Этот фасет *basic_filebuf* получает из локализации потока (§ 21.7.1).

Аргумент шаблона *State* является типом, используемым для хранения состояния смещения преобразуемого потока. Тип *State* также может использоваться для идентификации различных преобразований путем задания специализации (specialization). Последний механизм полезен, поскольку символы из разнообразных кодировок (наборов символов) могут храниться в объектах одного и того же типа. Например:

```
class JISstate { /* .. */ };

p = new codecvt<wchar_t, char, mbstate_t>;   // стандартный char в wchar_t
q = new codecvt<wchar_t, char, JISstate>;    // JIS в wchar_t
```

Без аргумента *State* не существовало бы способа сообщить фасету, в какой кодировке воспринимать поток символов *char*. Тип *mbstate_t* из <*cwchar*> или <*wchar.h*> идентифицирует стандартное системное преобразование между *char* и *wchar_t*.

Можно также создать новый *codecvt* в качестве производного класса, идентифицируемого по имени. Например:

```
class JIScvt : public codecvt<wchar_t, char, mbstate_t> { /* ... */ };
```

Вызов *in*(*s*, *from*, *from_end*, *from_next*, *to*, *to_end*, *to_next*) читает каждый символ диапазона [*from*, *from_end*) и пытается преобразовать его. Если символ можно преобразовать, *in*() записывает преобразованное значение в соответствующую позицию диапазона [*to*, *to_end*); если нет — *in*() завершает выполнение в этом месте. По завершении, *in*() сохраняет указатель на символ, следующий за последним прочитанным, в *from_next*, а указатель на символ, следующий за последним записанным, — в *to_next*. Значение *result*, возвращаемое *in*(), указывает, какая часть работы проделана:

ok:	преобразованы все символы диапазона [*from*, *from_end*)
partial:	преобразованы не все символы диапазона [*from*, *from_end*)
error:	функция *in*() встретила символ, который не смогла преобразовать
noconv:	преобразования не потребовалось

Обратите внимание, что результат ***partial*** не обязательно свидетельствует об ошибке. Возможно, придется прочитать больше символов, чем планировалось, чтобы завершился многобайтный символ, и чтобы его можно было записать; или следует очистить буфер вывода, чтобы освободить место для других символов.

Аргумент *s* типа ***State*** сообщает о состоянии последовательности символов ввода в момент вызова ***in***(). Это имеет значение в тех случаях, когда внешнее представление символов использует состояния смещения. Обратите внимание, что *s* является (не ***const***) ссылкой: по завершении вызова, *s* содержит состояние смещения последовательности ввода. Это позволяет программисту работать с ***partial*** (частичными) преобразованиями и преобразовывать длинные последовательности несколькими вызовами ***in***().

Вызов ***out***(*s*, ***from***, ***from_end***, ***from_next***, ***to***, ***to_end***, ***to_next***) преобразует [***from***, ***from_end***) из внутреннего представления во внешнее аналогично тому, как ***in***() преобразует из внешнего представления во внутреннее.

Поток символов должен начинаться и заканчиваться в «нейтральном» (несмещенном) состоянии («neutral», unshifted state). Как правило, это состояние ***State***(). Вызов ***unshift***(*s*, ***to***, ***to_end***, ***to_next***) считает состояние *s* «нейтральным» и помещает символы в [***to***, ***to_end***) так, чтобы вернуть последовательность символов в это нейтральное состояние. Результат работы ***unshift***() и использование аргумента ***to_next*** аналогичны ***out***().

Вызов ***length***(*s*, ***from***, ***from_end***, ***max***) возвращает количество символов из [***from***, ***from_end***), которое ***in***() может преобразовать.

Функция ***encoding***() возвращает

−1	если кодировка внешнего набора символов использует состояние (state) (например, использует последовательности символов, означающие установку и сброс смещения)
0	если кодировка использует различное количество байт для представления отдельных символов (например, представление символов может использовать бит в байте для указания, один или два байта используются в представлении данного символа)
n	если каждый символ во внешнем представлении занимает *n* байт

Вызов ***always_noconv***() возвращает ***true***, если не требуется никакого преобразования между внутренним и внешним наборами символов, и ***false*** — в противном случае. Очевидно, что ***always_noconv***()==***true*** открывает возможность для обеспечения максимально эффективной реализации, которая просто не задействует функции преобразования.

Вызов ***max_length***() возвращает максимальное значение, которое может вернуть ***length***() при корректном наборе аргументов.

Простейшие преобразование кодировки, которое я смог придумать — это преобразование входных символов из нижнего регистра в верхний. Так что ниже представлен, вероятно, простейший из возможных классов ***codecvt***, который, однако, делает нечто полезное:

```
class Cvt_to_upper : public codecvt<char, char, mbstate_t> {      // преобразование
                                                                  // в верхний регистр
    explicit Cvt_to_upper(size_t r = 0) : codecvt(r) { }
```

```
protected:
        // чтение внешнего представление и запись во внутреннее:
        result do_in(State& s, const char* from, const char* from_end, const char*&
                                        from_next, char* to, char* to_end, char*& to_next) const;

        // чтение внутреннего представление и запись во внешнее:
        result do_out(State& s, const char* from, const char* from_end, const char*&
                                        from_next, char* to, char* to_end, char*& to_next) const
{
        return codecvt<char, char, mbstate_t>::do_out(s,from,from_end,from_next,
                                                                to, to_end, to_next);
}

        result do_unshift(State&, E* to, E* to_end, E*& to_next) const { return ok; }

        int do_encoding() const throw() { return 1; }
        bool do_always_noconv() const throw() { return false; }

        int do_length(const State&, const E* from, const E* from_end, size_t max) const;
        int do_max_length() const throw();
};

codecvt<char, char, mbstate_t>::result
Cvt_to_upper::do_in(State& s, const char* from, const char* from_end, const char*&
                                        from_next, char* to, char* to_end, char*& to_next) const
{
        // ... § Г.6[16] ...
}

int main()   // тривиальный тест
{
        locale ulocale(locale(), new Cvt_to_upper);

        cin.imbue(ulocale);

        char ch;
        while(cin>>ch) cout << ch;
}
```

Имеется и версия *codecvt* вида *_byname* (§ Г.4, § Г.4.1):

```
template <class I, class E, class State>
class std::codecvt_byname : public codecvt<I,E,State> { /* ... */ };
```

Г.4.7. Сообщения

Большинство пользователей предпочитают взаимодействовать с программой на своем родном языке. Однако мы не можем предложить стандартный механизм, обеспечивающий в общем виде зависящее от локализации взаимодействие с программой. Вместо этого, библиотека предоставляет нехитрый механизм хранения зависящих от локализации наборов строк, из которых программист может создавать простые сообщения. В сущности, класс *messages* (сообщения) реализует тривиальную базу данных, доступную только для чтения:

```
class std::messages_base {
public:
```

```
        typedef int catalog;    // тип идентификатора каталога
};

template <class Ch>
class std::messages : public locale::facet, public messages_base {
public:
        typedef Ch char_type;
        typedef basic_string<Ch> string_type;

        explicit messages(size_t r = 0);
        catalog open(const basic_string<char>& fn, const locale&) const;
        string_type get(catalog c, int set, int msgid, const string_type& d) const;
        void close(catalog c) const;

        static locale::id id;    // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)

protected:
        ~messages();

        // виртуальные "do_функции" (см. § Г.4.1)
};
```

Вызов ***open***(***s***, ***loc***) открывает «каталог» (catalog) по имени ***s*** сообщений для локализации ***loc***. Каталог — это набор строк, организованных зависящим от реализации способом и доступных посредством функции ***messages***::***get***(). Если открыть каталог по имени ***s*** невозможно, возвращается отрицательное значение. Каталог должен быть открыт до первого использования ***get***().

Вызов ***close***(***cat***) закрывает каталог, идентифицируемый через ***cat***, и освобождает все ресурсы, связанные с данным каталогом.

Вызов ***get***(***cat***, ***set***, ***id***, ***"не вышло!"***) осуществляет поиск сообщения (***set***, ***id***) в каталоге ***cat***. Если строка найдена, ***get***() возвращает эту строку; в противном случае ***get***() возвращает строку по умолчанию (здесь — ***string***(***"не вышло!"***)).

Приведем пример фасета ***messages*** для реализации, в которой каталог сообщений выполнен в виде вектора множеств «сообщений», а само «сообщение» является строкой:

```
struct Set {
        vector<string> msgs;
};
struct Cat {
        vector<Set> sets;
};

class My_messages : public messages<char> {
        vector<Cat>& catalogs;
public:
        explicit My_messages(size_t = 0) : catalogs(*new vector<Cat>) { }

        catalog do_open(const string& s, const locale& loc) const;    // открыть каталог s
        string do_get(catalog c, int s, int m, const string&) const;  // получить сообщение
                                                                       // (s,m) в c
        void do_close(catalog cat) const
        {
                if (catalogs.size()>c) catalogs.erase(catalogs.begin()+c);
        }
```

```
    ~My_messages() { delete &catalogs; }
};
```

Все функции-члены ***messages*** объявлены ***const***, поэтому структура данных каталога (***vector<Set>***) хранится вне фасета.

Сообщение выбирается путем задания каталога, множества внутри каталога и строки сообщения во множестве. Строка, передаваемая в качестве аргумента, используется в качестве результата по умолчанию в случае, если в каталоге не найдено сообщение:

```
string My_messages::do_get(catalog cat, int set, int msg, const string& def) const
{
    if (catalogs.size()<=cat) return def;
    Cat& c = catalogs[cat];
    if (c.sets.size()<=set) return def;
    Set& s = c.sets[set];
    if (s.msgs.size()<=msg) return def;
    return s.msgs[msg];
}
```

Открытие каталога состоит в считывании текстового представления с диска в структуру *Cat*. В примере я выбрал тривиально читаемое представление. Множество ограничено <<< и >>>, а каждое сообщение является строкой текста:

```
messages<char>::catalog My_messages::do_open(const string& n, const locale& loc) const
{
    string nn = n + locale().name();
    ifstream f(nn.c_str());
    if (!f) return -1;

    catalogs.push_back(Cat());          // создать ядро каталога
    Cat& c = catalogs.back();
    string s;
    while (f>>s && s=="<<<") {          // чтение множества Set
        c.sets.push_back(Set());
        Set& ss = c.sets.back();
        while(getline(f,s) && s != ">>>") ss.msgs.push_back(s);   // чтение сообщения
    }
    return catalogs.size() - 1;
}
```

Пример тривиального использования:

```
int main()
{
    if (!has_facet<My_messages>(locale())) {
        cerr << "не найден фасет сообщений в " << locale().name() << '\n';
        exit(1);
    }

    const messages<char>& m = use_facet<My_messages>(locale());
    extern string message_directory;  // место, где я храню свои сообщения
    int cat = m.open(message_directory, locale());
    if(cat<0) {
```

```
            cerr << "не найден каталог\n";
            exit(1);
        }
        cout << m.get(cat, 0, 0, "А вот и не нашел!") << endl;
        cout << m.get(cat, 1, 2, " А вот и не нашел!") << endl;
        cout << m.get(cat, 1, 3, " А вот и не нашел!") << endl;
        cout << m.get(cat, 3, 0, " А вот и не нашел!") << endl;
}
```

Если каталог выглядит следующим образом:

```
<<<
привет!
пока!
>>>
<<<
да
нет
может быть
>>>
```

на выходе этой программы будет:

```
привет!
может быть
А вот и не нашел!
А вот и не нашел!
```

Г.4.7.1. Использование сообщений из других фасетов

Кроме того, что сообщения являются хранилищем зависящих от локализации строк, предназначенных для общения с пользователями, класс ***messages*** может содержать строки для других фасетов. Например, фасет ***Season_io*** (§ Г.3.2) можно написать так:

```
class Season_io : public locale::facet {
        const messages<char>& m;          // справочник сообщений (directory)
        int cat;                          // каталог сообщений (catalog)
public:
        class Missing_messages { };

        Season_io(int i = 0)
                : locale::facet(i),
                m(use_facet<Season_messages>(locale())),
                cat(m.open(message_directory, locale()))
        { if(cat<0) throw Missing_messages(); }

        ~Season_io() { }          // чтобы иметь возможность уничтожить объекты
                                  // Season_io (§ Г.3)

        const string& to_str(Season x) const;                  // строковое представление x

        bool from_str(const string& s, Season& x) const;       // поместить Season,
                                                               // соответствующий s, в x

        static locale::id id;          // идентификатор фасета (§ Г.2, § Г.3, § Г.3.1)
};
```

```
locale::id Season_io::id;          // определение объекта идентификатора

const string& Season_io::to_str(Season x) const
{
    return m->get(cat, x, "нет такого времени года");
}

bool Season_io::from_str(const string& s, Season& x) const
{
    for (int i = Season::spring; i<=Season::winter; i++)
        if(m->get(cat, i, "нет такого времени года") == s) {
            x = Season(i);
            return true;
        }
    return false;
}
```

Решение с использованием сообщений отличается от исходного (§ Г.3.2) тем, что у разработчика набора строк ***Season*** для новой локализации должна быть возможность добавлять их в справочник ***messages***. Это не представляет сложности для того, кто включает новую локализацию в среду исполнения. Однако, поскольку ***messages*** предоставляют интерфейс только для чтения, добавление нового набора названий времен года может оказаться за рамками возможностей прикладного программиста.

Имеется и версия ***messages*** вида ***_byname*** (§ Г.4, § Г.4.1):

```
template<class Ch>
class std::messages_byname : public messages<Ch> { /* ... */ };
```

Г.5. Советы

[1] Готовьтесь к тому, что каждая нетривиальная программа или система, непосредственно взаимодействующая с людьми, будет использована в нескольких странах; § Г.1.

[2] Не полагайтесь на то, что все используют тот же набор символов, что и вы; § Г.4.1.

[3] Предпочитайте ***locales*** написанию специального («ad hoc») кода национально-зависимого ввода/вывода; § Г.1.

[4] Избегайте прямого упоминания в тексте программы строковых имен локализаций; § Г.2.1.

[5] Минимизируйте использование глобальной информации о формате; § Г.2.3, § Г.4.4.7.

[6] Отдавайте предпочтение зависящему от локализации сравнению строк и сортировке; § Г.2.4, § Г.4.1.

[7] Делайте фасеты неизменяемыми; § Г.2.2, § Г.3.

[8] Сосредоточьте смену ***locale*** в нескольких фрагментах программы; § Г.2.3.

[9] Предоставьте локализации управлять временем жизни фасетов; § Г.3.

[10] При написании функций ввода/вывода, зависящих от локализации, не забывайте обрабатывать исключения, генерируемые предоставляемыми пользователями (замещенными) функциями; § Г.4.2.2.

[11] Пользуйтесь простым типом ***Money*** для хранения денежных величин; § Г.4.3.

[12] Пользуйтесь простыми определяемыми пользователем типами для хранения значений, требующих зависящего от локализации ввода/вывода (в противовес приведению значений встроенных типов или приведению к встроенным типам); § Г.4.3.
[13] Не верьте замерам времени исполнения, пока не оцените все влияющие на них факторы; § Г.4.4.1.
[14] Помните об ограничениях, накладываемых *time_t*; § Г.4.4.1, § Г.4.4.5.
[15] Пользуйтесь процедурой ввода даты, воспринимающей различные входные форматы; § Г.4.4.5.
[16] Отдавайте предпочтение функциям классификации, в которых локализация присутствует явно; § Г.4.5, § Г.4.5.1.

Г.6. Упражнения

1. (*2.5) Определите ***Season_io*** (§ Г.3.2) для языка, отличного от американского английского.
2. (*2) Определите класс ***Season_io*** (§ Г.3.2), принимающий набор строк названий времен года в качестве аргумента конструктора, так чтобы названия времен года различных локализаций могли быть представлены в виде объектов этого класса.
3. (*3) Напишите функцию ***collate<char>::compare()*** со словарным упорядочиванием. Предпочтительно сделать это для языка типа немецкого или французского, в алфавитах которых букв больше, чем в английском.
4. (*2) Напишите программу, которая читает и пишет логические значения в виде чисел, английских слов и слов любого другого языка по вашему выбору.
5. (*2.5) Определите тип ***Time*** для представления времени дня. Определите тип ***Date_and_time*** (дата и время) с помощью типов ***Time*** и ***Date***. Обсудите плюсы и минусы этого подхода по сравнению с ***Date*** из (§ Г.4.4). Реализуйте зависящий от локализации ввод/вывод для ***Time*** и ***Date_and_time***.
6. (*2.5) Спроектируйте и реализуйте фасет почтового индекса. Сделайте это как минимум для двух стран с непохожими соглашениями относительно написания адресов. Например: *NJ 07932* и *CB21QA*.
7. (*2.5) Спроектируйте и реализуйте фасет телефонных номеров. Сделайте это как минимум для двух стран с непохожими соглашениями относительно написания телефонных номеров. Например: *(973) 360–8000* и *1223 343000*.
8. (*2.5) Поэкспериментируйте и посмотрите, какие форматы ввода и вывода данных используются в вашей реализации.
9. (*2.5) Определите ***get_time()***, которая «догадывается» о смысле дат, таких как 12 5 1995, но при этом отвергает все или почти все ошибочные данные. Будьте точны в определении того, какие именно «догадки» приемлемы; обсудите вероятность ошибки.
10. (*2) Определите ***get_time()***, которая воспринимает значительно больше форматов, чем приведенная в § Г.4.4.5.
11. (*2) Составьте список локализаций, поддерживаемых вашей системой.
12. (*2.5) Выясните, где в вашей системе хранятся именованные локализации. Если у вас есть доступ к той части системы, где хранятся локализации, создайте новую именованную локализацию. Будьте очень осторожны, чтобы не испортить существующие локализации.

13. (*2) Сравните две реализации ***Season_io*** (§ Г.3.2 и § Г.4.7.1).
14. (*2) Напишите и протестируйте фасет ***Date_out***, который записывает даты типа ***Date*** с помощью формата, передаваемого в качестве аргумента конструктора. Обсудите плюсы и минусы такого подхода по сравнению с глобальным форматом, предоставляемым ***date_fmt*** (§ Г.4.4.6).
15. (*2.5) Реализуйте ввод/вывод римских чисел (вида *XI* и *MDCLII*).
16. (*2.5) Реализуйте и протестируйте ***Cvt_to_upper*** (§ Г.4.6).
17. (*2.5) Воспользуйтесь ***clock***() для определения средней «стоимости»: (1) вызова функции, (2) вызова виртуальной функции, (3) чтения ***char***, (4) чтения ***int*** из одной цифры, (5) чтения ***int*** из 5 цифр, (6) чтения ***double*** из 5 цифр, (7) ***string*** из одного символа, (8) ***string*** из 5 символов и (9) ***string*** из 40 символов.
18. (*6.5) Выучите еще один естественный язык.

Приложение Д

Безопасность исключений и стандартная библиотека

Все произойдет в точности так,
как Вы и ожидали,
если только Ваши ожидания не ошибочны.

— Хайман Роузен

Безопасность исключений — безопасные при исключениях методы **реализации** — представление ресурсов — присваивание — ***push_back***() — **конструкторы** и инварианты — гарантии стандартных контейнеров — вставка и **удаление** элементов — гарантии и компромиссы — ***swap***() — инициализация **и итераторы** — ссылки на элементы — предикаты — ***string***, потоки, алгоритмы, ***valarray*** и ***complex*** — стандартная библиотека C — значение для пользователей библиотеки — советы — упражнения.

Д.1. Введение

Функции стандартной библиотеки часто вызывают операции, предоставляемые пользователем в виде аргументов функции или шаблона. Естественно, некоторые из этих предоставленных пользователем операций время от времени генерируют исключения. Другие функции, например распределители памяти, также могут генерировать исключения. Рассмотрим пример:

```
void f(vector<X>& v, const X& g)
{
    v[2]=g;                 // присваивание X может сгенерировать исключение
    v.push_back(g);         // распределитель для vector<X> может
                            // сгенерировать исключение
    sort(v.begin(), v.end()); // операция X "меньше чем" может
                            // сгенерировать исключение
    vector<X> u = v;        // копирующий конструктор X может
                            // сгенерировать исключение
    // ...

    // "u" здесь уничтожается: мы должны гарантировать,
    // что деструктор X сможет отработать правильно
}
```

Что произойдет, если при попытке копирования ***g*** присваивание сгенерирует исключение? Вектор ***v*** будет оставлен с недействительным элементом? Что произойдет, если конструктор, который используется в ***v.push_back()*** для копирования ***g***, сгенерирует ***std::bad_alloc***? Число элементов изменится? В контейнер добавится недействительный элемент? Что случится, если во время сортировки оператор ***X*** «меньше чем» сгенерирует исключение? Элементы будут отсортированы частично? Может алгоритм сортировки удалить элемент из контейнера и не вернуть его туда?

Составление полного списка возможных исключений в обсуждаемом примере оставлено в качестве упражнения (§ Д.8[1]). Объяснение того, насколько корректно ведет себя подобный код для каждого хорошо определенного типа ***X*** — даже такого ***X***, который генерирует исключения, — является одной из целей приложения Д. Естественно, львиная доля объяснения сводится к приданию смысла понятиям «хорошо (корректно) ведущий себя» («well behaved») и «хорошо (корректно) определенный» («well defined») в контексте исключений, а также к выработке содержательной терминологии.

Цели данного приложения:

[1] указать, как пользователь может проектировать типы, которые удовлетворяют требованиям стандартной библиотеки;
[2] сформулировать гарантии, предоставляемые стандартной библиотекой;
[3] сформулировать требования стандартной библиотеки к предоставленному пользователем коду;
[4] продемонстрировать эффективные методы построения безопасных при исключениях (exception-safe) и эффективных контейнеров; и
[5] представить несколько общих правил безопасного при исключениях программирования.

Обсуждение безопасности исключений (exception safety) обязательно сосредотачивается на поведении в самом худшем случае. Где исключение создаст больше всего проблем? Как стандартная библиотека защищает себя и своих пользователей от возможных затруднений? Как сами пользователи могут помочь предотвратить проблемы? И пожалуйста, не позвольте этому приложению отвлечь вас от того центрального факта, что генерация исключения есть наилучший способ сообщить об ошибке (§ 14.1, § 14.9). Обсуждение понятий, методов и гарантий стандартной библиотеки организовано следующим образом:

§ Д.2. Содержит обсуждение понятия безопасности исключений.
§ Д.3. Представляет методы реализации эффективных и безопасных при исключениях контейнеров и операций.
§ Д.4. Очерчивает гарантии, предоставляемые для контейнеров стандартной библиотеки и их операций.
§ Д.5. Резюмирует вопросы безопасности исключений для не контейнерных частей стандартной библиотеки.
§ Д.6. Содержит обзор безопасности исключений с точки зрения пользователя стандартной библиотеки.

Как и во всем, стандартная библиотека демонстрирует примеры различных рисков, которые следует иметь в виду при разработке подверженных подобным рискам приложений. Методы, использованные в стандартной библиотеке для обеспечения безопасности исключений, применимы к широкому диапазону проблем.

Д.2. Безопасность исключений

Говорят, что операция на объекте *безопасна при исключениях* (exception safe), если эта операция оставляет объект в действительном состоянии (valid state), когда завершается генерацией исключения. Это действительное состояние может быть ошибочным состоянием, требующим очистки, но оно должно быть хорошо определено, чтобы имелась возможность написать разумный код обработки ошибок для данного объекта. Например, обработчик исключений мог бы уничтожить объект, восстановить его, повторить вариант операции, просто возобновить работу и т. д.

Иными словами, у объекта есть инвариант (§ 24.3.7.1), конструкторы устанавливают этот инвариант, все дальнейшие операции соблюдают инвариант (даже и при генерации исключений), а деструктор выполняет заключительную очистку. Перед генерацией исключения операция обязана позаботиться о соблюдении инварианта, чтобы объект остался в действительном состоянии. Однако весьма вероятно, что это действительное состояние окажется неудовлетворительным для приложения. Например, оставленная пустой строка или контейнер, брошенный неотсортированным. «Восстановить» — значит дать объекту значение, которое является более подходящим/желательным для приложения нежели то, с которым объект был оставлен после неудачной операции. В контексте стандартной библиотеки наиболее интересные объекты — контейнеры.

Мы рассмотрим, при каких условиях операции на контейнерах стандартной библиотеки могут считаться безопасными при исключениях. Имеются всего две действительно простые концептуально стратегии:

[1] «*Без гарантий*»: если сгенерировалось исключение, обрабатываемый контейнер, возможно, испорчен.

[2] «*Сильная гарантия*»: если сгенерировалось исключение, обрабатываемый контейнер остается в состоянии, в котором он находился перед началом операции стандартной библиотеки.

К сожалению, оба ответа слишком просты для реального использования. Альтернатива [1] неприемлема, потому что подразумевает, что после того, как в контейнерной операции сгенерировалось исключение, к контейнеру нельзя обратиться; его даже нельзя уничтожить без риска вызвать ошибку этапа выполнения. Альтернатива [2] неприемлема, потому что она навязывает стоимость семантики отката каждой операции стандартной библиотеки.

Чтобы разрешить эту дилемму, стандартная библиотека C++ обеспечивает набор таких гарантий безопасности при исключениях, которые распределяют бремя создания правильных программ между разработчиками стандартной библиотеки и ее пользователями:

[3а] «*Основная гарантия* (basic guarantee) для всех операций»: соблюдаются основные инварианты стандартной библиотеки, нет утечек ресурсов, например памяти.

[3б] «*Сильная гарантия* (strong guarantee) для ключевых операций»: в дополнение к обеспечению основной гарантии, устанавливается, что операция или достигает цели, или ни на что не влияет. Эта гарантия предоставляется для ключевых библиотечных операций, типа ***push_back***(), одноэлементных ***insert***() для *list* и ***uninitialized_copy***() (§ Д.3.1, § Д.4.1).

[3в] «*Гарантия отсутствия исключений* (nothrow guarantee) для некоторых операций»: в дополнение к обеспечению основной гарантии, устанавливается, что некоторые операции не генерируют исключений. Эта гарантия обеспечивается для нескольких простых операций, типа ***swap***() и ***pop_back***() (§ Д.4.1).

Как основная, так и сильная гарантии обеспечиваются при условии, что предоставленные пользователем операции (такие как присваивания или функция перестановки ***swap***()) не оставляют элементы контейнеров в недействительных состояниях, не создают утечек ресурсов, и что деструкторы не генерируют исключений. Например, рассмотрим два вспомогательных (handle) (§ 25.7) класса:

```
template<class T> class Safe {
        T* p;                           // p указывает на T, распределенный с помощью new
public:
        Safe() : p(new T) { }
        ~Safe() { delete p; }
        Safe& operator= (const Safe& a) { *p= *a.p; return *this; }
        // ...
};

template<class T> class Unsafe {  // небрежный и опасный код
        T* p;                           // p указывает на T
public:
        Unsafe(T* pp) : p(pp) { }
        ~Unsafe() { if (!p->destructible()) throw E(); delete p; }
                        // если нельзя уничтожить — генерируем исключение
        Unsafe& operator= (const Unsafe& a)
        {
                p->~T();                // уничтожение старого значения (§ 10.4.11)
                new(p) T(a.p);          // создание копии a.p в *p (§ 10.4.11)
                return *this;
        }

        // ...
};

void f(vector< Safe<Some_type> >& vg, vector< Unsafe<Some_type> >& vb)
{
        vg.at(1) = Safe<Some_type>();
        vb.at(1) = Unsafe<Some_type>(new Some_type);
        //...
}
```

В этом примере создание ***Safe*** происходит успешно, только если успешно создается ***T***. Создание ***T*** может не состояться из-за неудачного распределения памяти (с генерацией ***std::bad_alloc***) или из-за генерации исключения конструктором T. Но в каждом успешно созданном ***Safe***, ***p*** укажет на успешно созданный ***T***; если конструктор терпит неудачу, объект ***T*** (а равно и ***Safe***) не создается. Точно так же оператор присваивания ***T*** может сгенерировать исключение, заставляя оператор присваивания ***Safe*** неявно сгенерировать это исключение снова. Затруднений не возникает при условии, что оператор присваивания ***T*** всегда оставляет свои операнды в хорошем состоянии. Поэтому ***Safe*** — корректно ведущий себя класс, и, следовательно, каждая операция стандартной библиотеки с ***Safe*** будет иметь разумный и хорошо определенный результат.

С другой стороны, ***Unsafe***() написан небрежно (вернее наоборот, написан тщательно, чтобы показать, как поступать *не* надо). Создание ***Unsafe*** не потерпит неудачу. Напротив, операции с ***Unsafe***, типа присваивания и уничтожения, оставлены наедине с множеством потенциальных проблем. Оператор присваивания может потерпеть неудачу при генерации исключения в копирующем конструкторе ***T***. Это оставило бы ***T*** в неопределенном состоянии, потому что старое значение ****p*** уже разрушено, а никакое новое значение его не заменило. Вообще говоря, результаты подобного поведения непредсказуемы. Деструктор ***Unsafe*** отражает непродуманную попытку защититься от нежелательного уничтожения объекта: генерация исключения во время обработки исключения вызовет обращение к ***terminate***() (§ 14.7), тогда как стандартная библиотека требует, чтобы деструктор завершался нормально после уничтожения объекта. Стандартная библиотека не дает — и не может дать — никаких гарантий, если пользователь предоставляет объекты, ведущие себя столь ужасно.

С точки зрения обработки исключений, ***Safe*** и ***Unsafe*** отличаются тем, что ***Safe*** использует свой конструктор, чтобы установить инвариант (§ 24.3.7.1), который позволяет реализовать операции класса просто и безопасно. Если установить инвариант не удается, исключение генерируется прежде, чем появится недействительный объект. ***Unsafe***, в свою очередь, действует наобум, без содержательного инварианта, а отдельные операции генерируют исключения в отсутствие общей стратегии обработки ошибок. Естественно, это приводит к нарушениям (разумных) предположений стандартной библиотеки относительно поведения типов. Например, ***Unsafe*** может оставить в контейнере недействительные элементы после генерации исключения в ***T*::*operator*=**(), а также способен сгенерировать исключение в своем деструкторе.

Заметим, что гарантии стандартной библиотеки относительно предоставленных пользователем неблагополучных операций аналогичны гарантиям языка относительно нарушений основной системы типов. Если основная операция применяется не в соответствии с ее спецификациями, поведение не определено. Например, если вы генерируете исключение в деструкторе элемента ***vector***, вы имеете не большие основания надеяться на разумный результат, чем когда вы разыменовываете указатель, инициализированный случайным числом:

```
class Bomb {
public:
        //...
        ~Bomb() { throw Trouble(); };
};

vector<Bomb> b(10);                          // приводит к неопределенному поведению

void f()
{
        int* p=reinterpret_cast<int*>(rand());    // приводит к неопределенному поведению
        *p=7;
}
```

Решительно утверждаем: если вы руководствуетесь основными правилами языка и стандартной библиотеки, библиотека будет вести себя хорошо, даже когда вы генерируете исключения.

Помимо обеспечения безопасности исключений как таковой, обычно стремятся также избегать утечек ресурсов. То есть операция, которая генерирует исключение. должна не только оставить свои операнды в хорошо определенном состоянии, но и обеспечить, чтобы каждый захваченный ресурс (в конечном счете) освободился. Например в точке, где сгенерировано исключение, вся распределенная память должна или освобождаться, или принадлежать объекту, который в свою очередь обязан обеспечить корректное освобождение памяти.

Стандартная библиотека гарантирует отсутствие утечек ресурсов при условии, что предоставленные пользователем операции, вызываемые библиотекой, также избегают утечек ресурсов. Пример:

```
void leak(bool abort)
{
        vector<int> v(10);                              // нет утечек
        vector<int>* p = new vector<int>(10);           // возможная утечка памяти
        auto_ptr< vector<int> > q(new vector<int>(10)); // нет утечек (§ 14.4.2)

        if (abort) throw Up();
        // ...
        delete p;
}
```

После генерации исключения ***vector*** с именем *v* и ***vector***, контролируемый ***q***, будут правильно уничтожены, а их ресурсы освободятся. Но ***vector***, на который указывает ***p***, не защищен от исключений и не будет уничтожен. Чтобы сделать эту часть кода безопасной, мы должны или явно удалить ***p*** перед генерацией исключения, или удостовериться, что ***vector*** принадлежит объекту вроде ***auto_ptr*** (§ 14.4.2), который должным образом уничтожит вектор при возникновении исключения.

Обратите внимание, что языковые правила частичного создания и уничтожения гарантируют, что исключения, сгенерированные при построении подобъектов и членов, будут правильно обработаны без особых мер со стороны кода стандартной библиотеки (§ 14.4.1). Это правило — фундамент всех технических приемов, имеющих дело с исключениями.

Помните также, что память — не единственный ресурс, который может теряться. Открытые файлы, блокировки, сетевые соединения и нити (threads) — вот только некоторые примеры системных ресурсов, которые функция должна будет освободить или передать какому-либо объекту перед генерацией исключения.

Д.3. Безопасные при исключениях методы реализации

Как обычно, стандартная библиотека демонстрирует примеры проблем, встречающихся во многих других контекстах, а также широко применяемые решения. Основные инструментальные средства, доступные для написания безопасного при исключениях кода, это:

[1] ***try***-блок (§ 8.3.1) и

[2] поддержка методики «выделение ресурса есть инициализация» (§ 14.4).

Общие принципы, которым надо следовать:

[1] никогда не теряйте информацию, пока не обеспечите ей замену; и

[2] всегда оставляйте объекты в действительном состоянии при генерации или повторной генерации исключений.

Действуя в таком духе, мы всегда сможем восстановиться после ошибочной ситуации. Практическая трудность следования этим принципам заключается в том, что невинно выглядящие операции (типа <, =, и ***sort***()) могут генерировать исключения. Чтобы знать, чего ожидать от приложения, требуется опыт.

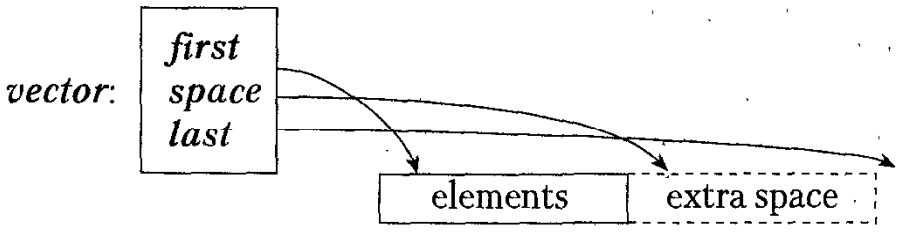
Если вы пишете библиотеку, в идеале следует стремиться к сильной гарантии безопасности исключений (§ Д.2) и всегда обеспечивать основную гарантию. При написании конкретной программы о безопасности исключений обычно беспокоятся меньше. Например, если я пишу простую программу анализа данных для личного использования, меня вполне устроит аварийное завершение в маловероятном случае исчерпания виртуальной памяти. Тем не менее, правильность и основы безопасности исключений тесно связаны.

Методы обеспечения основ безопасности исключений, такие как определение и проверка инвариантов (§ 24.3.7.1), подобны методам, которые полезны, чтобы сделать программу компактной и правильной. Следовательно, накладные расходы на обеспечение основ безопасности исключений (основная гарантия; § Д.2) — и даже на обеспечение сильной гарантии — могут оказаться минимальными, а то и вовсе незначительными; см. § Д.8[17].

Далее мы рассмотрим реализацию стандартного контейнера ***vector*** (§ 16.3) и выясним, что требуется для достижения этого идеала, и где мы могли бы довольствоваться менее жесткими требованиями к безопасности.

Д.3.1. Простой вектор

Типичная реализация ***vector*** (§ 16.3) включает дескриптор, содержащий указатели на первый элемент, на элемент, следующий за последним, и на «элемент», следующий за концом распределенного пространства (§ 17.1.3) (или эквивалентную информацию, представленную как указатель и смещения):



Вот объявление ***vector*** — упрощенное, чтобы показать только то, что необходимо для обсуждения безопасности исключений и предотвращения утечек ресурсов:

```
template< class T, class A = allocator<T> > class vector {
public:
    T* v;                       // начало распределенной памяти
    T* space;                   // конец последовательности элементов,
                                // начало свободного пространства,
                                // распределенного для возможного расширения
    T* last;                    // конец распределенного пространства
    A alloc;                    // распределитель памяти

    explicit vector(size_type n, const T& val = T(), const A& = A());
```

```
    vector(const vector& a);                    // копирующий конструктор
    vector& operator=(const vector& a);         // копирующее присваивание

    ~vector();

    size_type size() const { return space - v; }
    size_type capacity() const { return last - v; }

    void push_back(const T&);

    //...
};
```

Рассмотрим сначала наивную реализацию конструктора:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
                                            // предупреждение: наивная реализация
    : alloc(a)                              // копирование распределителя памяти
{
    v = alloc.allocate(n);                  // получение памяти для элементов (§ 19.4.1)
    space = last = v + n;
    for (T* p = v; p != last; ++p) alloc.construct(p, val);   // создание копии val в *p
                                                              // (§ 19.4.1)
}
```

Здесь имеются три источника исключений:

[1] *allocate*() генерирует исключение, указывающее, что нет доступной памяти;
[2] копирующий конструктор распределителя памяти генерирует исключение;
[3] копирующий конструктор для элемента типа *T* генерирует исключение, потому что не может скопировать *val*.

Во всех случаях никакого объекта не создается, так что деструктор *vector* не вызывается (§ 14.4.1).

Когда *allocate*() терпит неудачу, *throw* завершит выполнение прежде чем будут выделены какие-либо ресурсы, так что все хорошо.

Когда копирующий конструктор *T* терпит неудачу, нами уже была оприходована некоторая память, которую следует освободить, чтобы избежать утечки. Более серьезная трудность состоит в том, что копирующий конструктор для *T* мог бы сгенерировать исключение после правильного создания нескольких элементов, но перед созданием их всех.

Чтобы справиться с затруднением, мы могли бы отследить, какие элементы были созданы, и уничтожить их (и только их) в случае ошибки:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
                                            // усовершенствованная реализация
    : alloc(a)                              // копирование распределителя памяти
{
    v = alloc.allocate(n);                  // получение памяти для элементов
    iterator p;
    try {
        iterator end = v + n;
        for (p = v; p != end; ++p) alloc.construct(p, val);   // создание элемента
                                                              // (§ 19.4.1)
```

```
            last = space = p;
        }
        catch(...) {
            for (iterator q = v; q != p; ++q) alloc.destroy(q);   // уничтожение созданных
                                                                  // элементов
            alloc.deallocate(v, n);                               // освобождение памяти
            throw;                                                // повторная генерация
                                                                  // исключения
        }
    }
```

Накладные расходы здесь связаны с ***try***-блоком. В хорошей реализации C++ эти накладные расходы пренебрежимо малы сравнительно со стоимостью распределения памяти и инициализации элементов. Для реализаций, где вход в ***try***-блок накладен, заслуживает внимания проверка ***if*** (***n***) перед ***try*** и отдельная обработка случая пустого вектора.

Основная часть этого конструктора — безопасная при исключениях реализация ***uninitialized_fill***():

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
    For p;
    try {
        for (p = beg; p != end; ++p)
            new (static_cast<void*>(&*p)) T(x);  // создание копии x в *p (§ 10.4.11)
    }
    catch (...) {                                // уничтожение созданных элементов
                                                 // и повторная генерация исключения
        for (For q = beg; q != p; ++q) (&*q)->~T();  // (§ 10.4.11)
        throw;
    }
}
```

Любопытная конструкция &****p*** «присматривает» за итераторами, которые не являются указателями. В этом случае мы берем адрес разыменованного элемента, чтобы получить указатель. Явное приведение к ***void****** гарантирует, что используется стандартная библиотечная функция размещения (§ 19.4.5), а не определенный пользователем ***operator new***() для ***T****. Этот код работает на довольно низком уровне, где написание действительно общего кода может быть затруднено.

К счастью, мы не должны повторно реализовывать ***uninitialized_fill***(), потому что стандартная библиотека обеспечивает для нее желанную сильную гарантию (§ Д.2). Зачастую важно, чтобы операции инициализации либо завершались успешно, проинициализировав каждый элемент, либо терпели неудачу, не оставляя после себя созданных элементов. Вот почему алгоритмы ***uninitialized_fill***(), ***uninitialized_fill_n***(), и ***uninitialized_copy***() (§ 19.4.4) стандартной библиотеки обеспечивают подобную сильную гарантию безопасности исключений (§ Д.4.4).

Обратите внимание, что алгоритм ***uninitialized_fill***() не защищает от исключений, сгенерированных деструкторами элементов или итераторными операциями (§ Д.4.4). Такая защита была бы непозволительно дорогой (см. § Д.8 [16–17]).

Алгоритм ***uninitialized_fill***() может применяться ко многим видам последовательностей. Соответственно, он использует однонаправленный итератор (§ 19.2.1) и не может гарантировать уничтожение элементов в порядке, обратном их созданию.

Используя ***uninitialized_fill***(), мы можем написать:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)   // неряшливая реализация
        : alloc(a)                                           // копирование распределителя памяти
{
        v = alloc.allocate(n);                               // получение памяти для элементов
        try {
                uninitialized_fill(v, v + n, val);           // копирование элементов
                space = last = v + n;
        }
        catch (...) {
                alloc.deallocate(v, n);                      // освобождение памяти
                throw;                                       // повторная генерация исключения
        }
}
```

Лично я не стал бы вызывать этот славный код. Следующий раздел продемонстрирует, как существенно упростить его.

Заметим что конструктор повторно генерирует перехваченное исключение. Цель понятна — сделать ***vector*** прозрачным для исключений, чтобы пользователь мог определить истинную причину проблемы. Все контейнеры стандартной библиотеки обладают этим свойством. Прозрачность для исключений — часто наилучшая политика для шаблонов и других «тонких» слоев программного обеспечения. Напротив, главные части системы («модули») обязаны, вообще говоря, принимать на себя ответственность за все сгенерированные исключения. То есть разработчик модуля должен быть готов перечислить все исключения, которые могут генерироваться модулем. Достичь этого можно группировкой исключений (§ 14.2), отображением исключений от подпрограмм низшего уровня в собственные исключения модуля (§ 14.6.3) или спецификацией исключений (§ 14.6).

Д.3.2. Явное представление памяти

Опыт показывает, что написать правильный безопасный при исключениях код с помощью явных *try-блоков* гораздо труднее, чем многим кажется. На самом деле эти трудности не оправданы, потому что имеется альтернатива. Методика «выделение ресурса есть инициализация» (§ 14.4) позволяет уменьшить объем кода и сделать код более элегантным. В данном случае ключевой ресурс, требуемый ***vector***, — память для хранения элементов. Обеспечив дополнительный класс для представления понятия памяти, используемой в ***vector***, мы можем упростить код. При этом мы меньше рискуем случайно забыть об освобождении памяти:

```
template<class T, class A = allocator<T>>
struct vector_base {
        A alloc;        // распределитель памяти
        T* v;           // начало распределения
        T* space;       // конец последовательности элементов, начало свободного
                        // пространства, распределенного для возможного расширения
```

```
        T* last;            // конец распределенного пространства

        vector_base(const A& a, typename A::size_type n)
            : alloc(a), v(a.allocate(n)), space(v+n), last(v+n) { }
        ~vector_base() {alloc.deallocate(v, last-v); }
};
```

Пока *v* и *last* правильны, ***vector_base*** может быть уничтожен. Класс ***vector_base*** имеет дело с памятью для типа ***T***, а не с объектами типа ***T***. Следовательно, пользователь ***vector_base*** должен уничтожить все объекты, созданные в отведенной ***vector_base*** памяти, прежде чем уничтожать сам ***vector_base***.

Естественно, сам ***vector_base*** написан так, что при генерации исключения (копирующим конструктором распределителя памяти или функцией ***allocate***()) объект ***vector_base*** не создается, и память не теряется.

При наличии ***vector_base***, ***vector*** может быть определен примерно так:

```
template<class T, class A = allocator<T>>
class vector : private vector_base<T, A> {
        void destroy_elements() {for (T* p = v; p != space; ++p) p->~T(); space=v; }   // § 10.4.11
public
        explicit vector(size_type n, const T& val = T(), const A& = A());

        vector(const vector& a);                        // копирующий конструктор
        vector& operator=(const vector& a);             // копирующее присваивание

        ~vector() { destroy_elements(); }

        size_type size() const { return space - v; }
        size_type capacity() const { return last - v; }

        void push_back(const T&);

        //...
};
```

Деструктор ***vector*** явно вызывает для каждого элемента деструктор ***T***. Значит, если деструктор элемента генерирует исключение, уничтожение объекта ***vector*** терпит неудачу. Это может оказаться бедствием, если случится в ходе раскрутки стека, вызванной исключением; все закончится обращением к ***terminate***() (§ 14.7). В случае нормального уничтожения объекта генерация исключения из деструктора обычно ведет к утечке ресурсов и к непредсказуемому поведению кода, полагающегося на разумное поведение объектов. Не существует действенной защиты от исключений, сгенерированных деструкторами, так что библиотека не дает никаких гарантий, если деструктор элемента генерирует исключение (§ Д.4).

Теперь конструктор может быть определен просто:

```
template<class T, class A>
vector<T, A>::vector(size_type n, const T& val, const A& a)
        : vector_base (a, n)                    // распределение пространства для n элементов
{
        uninitialized_fill(v, v+n, val);        // копирование элементов
}
```

Копирующий конструктор отличается использованием ***uninitialized_copy***() вместо ***uninitialized_fill***():

```
template<class T, class A>
vector<T, A>::vector(const vector<T, A>& a)
    : vector_base(a, a.size())
{
    uninitialized_copy(a.begin(), a.end(), v);
}
```

Отметим, что такой стиль конструктора полагается на фундаментальное правило языка: если исключение сгенерировано в конструкторе, полностью созданные подобъекты (такие как члены базового класса) будут должным образом уничтожены (§ 14.4.1). Алгоритм ***uninitialized_fill()*** и его собратья (§ Д.4.4) обеспечивают эквивалентную гарантию для частично созданных последовательностей.

Д.3.3. Присваивание

Как обычно, присваивание отличается от создания тем, что нужно позаботиться о старом значении. Рассмотрим прямолинейную реализацию:

```
template<class T, class A>
vector<T, A>& vector<T, A>::operator=(const vector& a)          // предлагает сильную
{                                                               // гарантию (§ Д.2)
    vector_base<T, A> b(alloc, a.size());                       // получение памяти
    uninitialized_copy(a.begin(), a.end(), b.v);                // копирование элементов
    destroy_elements();
    alloc.deallocate(v, last - v);                              // освобождение старой памяти
    vector_base::operator=(b);                                  // инсталляция нового представления
    b.v = 0;                                                    // предотвращение освобождения
    return *this;
}
```

Такое симпатичное присваивание безопасно при исключениях. Однако оно повторяет много кода из конструкторов и деструкторов. Избежать этого можно так:

```
template<class T, class A>
vector<T, A>& vector<T, A>::operator=(const vector& a)          // предлагает сильную
{                                                               //гарантию (§ Д.2)
    vector temp(a);                                             // копирование a
    swap<vector_base<T, A>>(*this, temp);                       // обмен местами представлений
    return *this;
}
```

Однако реализация ***swap()*** по умолчанию не удовлетворяет нашим требованиям для ***vector_base***, т. к. она копирует и уничтожает ***vector_base***. Следовательно, предоставляем специализацию:

```
template< class T> void swap(vector_base< T>& a, vector_base& b)
{
    swap(a.a, b.a); swap(a.v, b.v); swap(a.space, b.space); swap(a.last, b.last);
}
```

Старые элементы уничтожаются деструктором ***temp***, а память, использованная для их хранения, освобождается деструктором ***vector_base*** объекта ***temp***.

Производительность этих двух версий должна быть эквивалентна. По существу они лишь демонстрируют два способа определить один и тот же набор операций, но вторая

реализация короче и не копирует код родственных векторных функций, поэтому написанное таким образом присваивание менее подвержено ошибкам и проще сопровождается.

Обратите внимание на отсутствие традиционной проверки на присваивание самому себе (§ 10.4.4):

```
if (this == &a) return *this;
```

Приведенные выше реализации присваивания сначала создают копии, а затем меняют местами представления. При этом присваивание самому себе происходит, очевидно, правильно. Я решил, что выигрыш, полученный от проверки в редком случае присваивания самому себе, с запасом перекрывается потерями в общем случае, когда присваивается другой ***vector***.

В обоих примерах упущены две возможности значительной оптимизации:

[1] если емкость выходного вектора достаточно велика, чтобы вместить входной ***vector***, нет нужды выделять новую память;

[2] присваивание элементу может дать выигрыш по сравнению с уничтожением и последующим созданием элемента.

Оптимизируя код, получим:

```
template<class T, class A>
vector<T, A>& vector<T, A>::operator=(const vector& a)      // оптимизированная версия,
                                                            // основная гарантия (§ Д.2)
{
    if (capacity() < a.size()) {        // распределение нового векторного представления:
        vector temp(a);                                     // копирование a
        swap< vector_base<T, A> >(*this, temp); // обмен местами представлений
        return *this;
    }

    if (this == &a) return *this;   // защита от присваивания самому себе (§ 10.4.4)

                                    // присваивание старым элементам:

    size_type sz = size();
    size_type asz = a.size();
    alloc = a.get_allocator();      // копирование распределителя памяти
    if (asz <= sz) {
        copy(a.begin(), a.begin() + asz, v);
        for (T* p = v+sz; p != space; ++p) p->~T();         // уничтожение лишних
                                                            // элементов (§ 10.4.11)
    }
    else {
        copy(a.begin(), a.begin() + sz, v);
        uninitialized_copy(a.begin() + sz, a.end(), space); // создание дополнительных
                                                            // элементов
    }
    space = v + asz;
    return *this;
}
```

Эти оптимизации не бесплатны. Алгоритм ***copy***() (§ 18.6.1) *не* предлагает сильной гарантии безопасности исключений. Он не гарантирует, что оставит свой выходной объект неизменным, если в ходе копирования сгенерируется исключение. Таким образом, если во время работы ***copy***() присваивание ***T*::operator=**() сгенерирует исклю-

чение, выходной ***vector*** не обязательно будет копией входного и не обязательно останется неизмененным. Например, первые пять элементов могут оказаться копиями элементов присваиваемого вектора, а остальные могут остаться неизмененными. Также вероятно, что тот элемент, который копировался, когда ***T*::*operator*=()** сгенерировал исключение, окажется со значением, не являющимся ни старым значением, ни копией соответствующего элемента присваиваемого вектора. И все же, если ***T*::*operator*=()** при генерации исключения оставляет свои операнды в действительном состоянии, то и ***vector*** останется в действительном состоянии — пусть и не в том, на которое мы надеялись.

В примере я скопировал распределитель памяти с помощью присваивания. На самом деле не требуется, чтобы каждый распределитель памяти поддерживал присваивание (§ 19.4.3); см. также § Д.8[9].

Присваивание для ***vector*** из стандартной библиотеки предлагает более слабую безопасность исключений, соответствующую последней обсуждаемой реализации, — а заодно и потенциальный выигрыш в производительности. Таким образом присваивание ***vector*** обеспечивает основную гарантию, что отвечает взглядам большинства людей на безопасности исключений. Однако оно не обеспечивает сильную гарантию (§ Д.2). Если вы нуждаетесь в присваивании, которое при генерации исключения оставляет ***vector*** неизмененным, следует либо воспользоваться библиотечной реализацией, обеспечивающей сильную гарантию, либо предоставить собственную операцию присваивания. Например:

```
template<class T, class A>
void safe_assign { vector<T, A>& a, const vector<T, A>& b)   // "очевидное" a = b
{
        vector<T, A> temp(a.get_allocator());
        temp.reserve(b.size());
        for (typename vector<T, A>::iterator p = b.begin(); p != b.end(); ++p)
                temp.push_back(*p);
        swap(a, temp);
}
```

При нехватке памяти для создания ***temp*** размером в ***b.size()*** элементов сгенерируется ***std*::*bad_alloc***, причем до того, как изменится ***a***. Точно так же, если ***push_back()*** терпит неудачу по любой причине, ***a*** останется нетронутым, потому что мы применяем ***push_back()*** к ***temp***, а не к ***a***. В этом случае элементы ***temp***, созданные функцией ***push_back()***, будут уничтожены до повторной генерации исключения, вызвавшего сбой.

Перестановка ***swap()*** не копирует элементы ***vector***. Она просто меняет местами члены данных ***vector***, то есть меняет местами подобъекты ***vector_base***. Поэтому она не генерирует исключений, даже если операции с элементами могли бы (§ Д.4.3). Таким образом, ***safe_assign()*** не делает побочных копий элементов и разумно эффективна.

Как это часто бывает, для очевидной реализации имеются альтернативы. Пусть библиотека займется копированием во временную переменную:

```
template<class T, class A> void safe_assign(vector<T, A>& a, const vector<T, A>& b)
                                                        // простое a = b
{
        vector<T, A> temp(b);          // копирование элементов b во временную переменную
        swap(a, temp);
}
```

На самом деле проще воспользоваться вызовом по значению (§ 7.2):

```
template<class T, class A>
void safe_assign(vector<T, A>& a, vector<T, A> b)                    // простое a = b
{                                          // (заметьте: b передается по значению)
    swap(a, b);
}
```

Последние два варианта ***safe_assign***() не копируют распределитель памяти ***vector***. Это — допустимая оптимизация; см. § 19.4.3.

Д.3.4. push_back()

С точки зрения безопасности исключений, ***push_back***() в одном отношении подобна присваиванию: мы должны позаботиться о неизменности ***vector*** в случае неудачного добавления нового элемента:

```
template<class T,class A>
void vector<T, A>::push_back(const T& x)
{
    if (space == last) {              // нет свободного пространства; перераспределение:
        vector_base b(alloc, size() ? 2*size() : 2);    // удвоить размер
        uninitialized_copy(v, space, b.v);
        new(b.space) T(x);                // поместить копию x в *b.space (§ 10.4.11)
        ++b.space;
        destroy_elements();
        swap<vector_base<T, A>>(b, *this);        // поменять местами представления
        return;
    }
    new(space) T(x);                      // поместить копию x в *space (§ 10.4.11)
    ++space;
}
```

Разумеется копирующий конструктор, использованный для инициализации ****space***, может сгенерировать исключение. Если это случится, значение ***vector*** останется неизменным, а ***space*** не увеличится. Причем элементы ***vector*** не перемещались в памяти, так что ссылающиеся на них итераторы остаются действительными. Таким образом, данная реализация обеспечивает сильную гарантию, а исключение, сгенерированное распределителем памяти или даже предоставленным пользователем копирующим конструктором, оставит ***vector*** неизменным. Стандартная библиотека предлагает именно такую гарантию для ***push_back***() (§ Д.4.1).

Обратите внимание на отсутствие ***try***-блока (кроме скрытого в ***uninitialized_copy***()). За счет тщательного упорядочения операций гарантируется, что при генерации исключения ***vector*** останется неизменным.

Обеспечение безопасности исключений путем упорядочения и следования методике «выделение ресурса есть инициализация» (§ 14.4) зачастую оказывается изящнее и производительнее, чем явная обработка ошибок с использованием ***try***-блоков. Из-за того что программист неудачно упорядочивает код, возникает больше проблем с безопасностью исключений, чем от недостатка особого кода обработки исключений. Основное правило упорядочения — не уничтожать информацию до того, как

будет создана походящая замена, пригодная для присваивания без риска возникновения исключений.

Исключения создают почву для неожиданностей по части управления последовательностью выполнения инструкций. Во фрагментах кода с простым локальным управлением, где преобладают, например, ***operator=***(), ***safe_assign***() и ***push_back***(), возможности для сюрпризов ограничены. Довольно просто взглянуть на такой код и спросить себя: «Может ли эта строка программы сгенерировать исключение, и что произойдет, если она это сделает?». Для больших функций со сложным управлением, где имеются вложенные циклы и сложные условные инструкции, подобный прогноз сделать трудно. Добавление ***try***-блоков увеличивает сложность управления и потому может стать источником недоразумений и ошибок (§ 14.4). Я полагаю, что упорядочение и методика «выделение ресурса есть инициализация» обладают преимуществом по сравнению с широким использованием ***try***-блоков: упрощение управления. Простой, стильный код проще и понять, и сделать правильным.

Реализация ***vector*** приведена здесь в качестве примера как трудностей, которые могут создать исключения, так и методов преодоления этих трудностей. Стандарт не требует, чтобы реализация в точности соответствовала одной из представленных выше. Что на самом гарантирует стандарт — тема § Д.4.

Д.3.5. Конструкторы и инварианты

С точки зрения безопасности исключений, другие операции вектора или эквивалентны уже исследованным (потому что они захватывают и освобождают ресурсы подобными же способами), или тривиальны (потому что они не выполняют операций, требующих искусства обеспечения действительных состояний). Однако для большинства классов именно подобные «тривиальные» функции составляют львиную долю кода. Трудность написания таких функций зависит главным образом от среды, которую устанавливает для их деятельности конструктор. Иными словами, сложность «обычных функций-членов» весьма чувствительна к выбору хорошего инварианта класса (§ 24.3.7.1). Изучение «тривиальных» функций векторов поможет нам вникнуть в суть интересного вопроса о том, *что* делает инвариант класса хорошим, и как следует писать конструкторы, чтобы они устанавливали хорошие инварианты.

Операции типа индексирования ***vector*** (§ 16.3.3) писать просто, потому что они могут полагаться на инвариант, установленный конструкторами и поддерживаемый всеми функциями, которые захватывают или освобождают ресурсы. В частности, оператор индексирования вправе полагаться на то, что *v* ссылается на массив элементов:

```
template<class T, class A>
T& vector<T,A>::operator[](size_type i)
{
        return v[i];
}
```

Фундаментально важно, чтобы конструкторы захватили ресурсы и установили простой инвариант. Разберемся почему, рассмотрев альтернативное определение ***vector_base***:

```
template<class T, class A = allocator<T>>      // неуклюжее использование конструктора
class vector_base {
```

```
public:
    A alloc;                    // распределитель памяти
    T* v;                       // начало отведенной памяти
    T* space;                   // конец последовательности элементов, начало свободного
                                // пространства, отведенного для возможного расширения
    T* last;                    // конец отведенного пространства
    vector_base(const A& a, typename A::size_type n) : alloc(a), v(0), space(0), last(0)
    {
        v = alloc.allocate(n);
        space = last = v + n;
    }
    ~vector_base() { if (v) alloc.deallocate(v, last - v); }
};
```

Здесь я создаю ***vector_base*** в два хода: сначала задаю «безопасное состояние», где *v*, ***space*** и ***last*** установлены в ***0***. И лишь вслед за этим я пробую отвести память. Такая тактика вызвана следующим неуместным опасением: если во время распределения памяти под элементы произойдет исключение, конструктор может оставить после себя частично созданный объект. Это опасение неуместно, поскольку невозможно «оставить после себя» частично созданный объект и позднее обратиться к нему. Правила для статических объектов, автоматических объектов, объектов-членов и элементов контейнеров стандартной библиотеки предотвращают подобное развитие событий. С другой стороны, это могло/может случиться в «достандартных» библиотеках, которые использовали/используют выделение памяти через ***new*** (§ 10.4.11) для создания объектов в контейнерах, разработанных без учета безопасности исключений. Трудно сломать старые привычки.

Отметим, что такая попытка написать код «побезопаснее» усложняет инвариант класса: больше не гарантируется, что *v* указывает на отведенную память. Теперь *v* может быть ***0***, что немедленно приводит к издержкам. Требования стандартной библиотеки для распределителей памяти не гарантируют безопасности освобождения указателя со значением ***0*** (§ 19.4.1). Этим распределители памяти отличаются от ***delete*** (§ 6.2.6). Так что мне пришлось предусмотреть в деструкторе проверку. Кроме того каждый элемент сначала инициализируется ***0***, а уж затем ему присваивается значение. Стоимость выполнения дополнительной работы может быть существенна для тех типов, для которых присваивание нетривиально, вроде ***string*** и ***list***.

Двухступенчатый конструктор — не такой уж необычный стиль. Иногда двухступенчатость даже делается явной, так что конструктор производит только некоторую «простую и безопасную» инициализацию, помещая объект в уничтожимое состояние. Реальное же создание объекта оставлено функции ***init***(), которую пользователь должен явно вызвать. Например:

```
template<class T>               // архаичный ("пред-стандартный,
                                //до-исключительный") стиль
class vector_base {
public:
    T* v;                       // начало отведенной памяти
    T* space;                   // конец последовательности элементов, начало свободного
                                // пространства, отведенного для возможного расширения
    T* last;                    // конец отведенного пространства
```

```
    vector_base() : v(0), space(0), last(0) { }
    ~vector_base() { free(v); }

    bool init(size_t n)        // возвращает true, если инициализация успешна
    {
        if (v = (T*) malloc(sizeof(T)*n)) {
            uninitialized_fill(v, v + n, T());
            space = last = v + n;
            return true;
        }
        return false;
    }
};
```

Кажущаяся ценность данного стиля состоит в том, что:

[1] Конструктор не генерирует исключений, а успешность инициализации *init()* можно проверить «обычными» (то есть без исключений) средствами.

[2] Существует тривиальное действительное состояние. При наличии серьезной проблемы операция может придать объекту это состояние.

[3] Выделение ресурсов отложено до момента, когда на самом деле понадобится полностью инициализированный объект.

Эти пункты изучаются в следующих подразделах, где показывается, почему двухступенчатая методика создания не приносит ожидаемых выгод. Она может также оказаться источником проблем.

Д.3.5.1. Использование функций init()

Первый пункт (использование функции *init()* вместо полноценного конструктора) — липа. Применение конструкторов и обработка исключений — более общий и систематический способ запроса ресурсов и работы с ошибками инициализации (§ 14.1, § 14.4). Рассматриваемый двухступенчатый стиль — пережиток C++ в варианте до введения механизма исключений.

Тщательно написанные коды с использованием двух сравниваемых стилей — приблизительно эквивалентны. Вот примеры:

```
int f1(int n)
{
    vector<X> v;
    // ...
    if (v.init(n)) {
        // использование v как вектора из n элементов
    }
    else {
        // обработка проблемы
    }
}
```

и

```
int f2(int n)
try {
    vector v<X> v(n);
```

```
        // ...
        // использование v как вектора из n элементов
    }
    catch(...) {
        // обработка проблемы
    }
```

При этом наличие отдельной функции *init*() — это возможность:
[1] забыть вызвать *init*() (§ 10.2.3);
[2] забыть проверить успешность *init*();
[3] забыть, что *init*() может сгенерировать исключение; и
[4] использовать объект до вызова *init*().
Определение *vector<T>::init*() иллюстрирует [3].

В хорошей реализации C++ *f2*() чуть-чуть быстрее *f1*() потому что в общем случае избегает проверки.

Д.3.5.2. Надежда на действительное состояние по умолчанию

Второй пункт (наличие просто конструируемого действительного состояния по умолчанию) в принципе правилен, но в случае *vector* он ведет к ненужным расходам. Теперь *vector_base* может содержать *v==0*, так что реализация *vector* должна повсеместно защищаться от этого. Например:

```
template<class T>
T& vector<T>::operator[](size_t i)
{
    if (v) return v[i];
    // обработка ошибки
}
```

Наличие возможности *v==0* делает стоимость индексирования без проверки на соответствие диапазону эквивалентной стоимости доступа с проверкой диапазона:

```
template<class T>
T& vector<T>::at(size_t i)
{
    if (i < v.size()) return v[i];
    throw out_of_range("vector index");
}
```

Главное здесь то, что вводя возможности *v==0*, я усложнил основной инвариант *vector_base*. Соответственно усложнился и основной инвариант *vector*. В результате, чтобы справиться с новыми инвариантами усложнился весь код в *vector* и *vector_base*. Это — источник возможных ошибок, трудностей поддержки и накладных расходов этапа выполнения. Обратите внимание, что условные инструкции могут быть неожиданно дорогостоящими на машинах с современной архитектурой. Там где важна производительность, решающее значение может иметь реализация ключевых операций (типа индексирования вектора) без условных инструкций.

Интересно что первоначальное определение *vector_base* уже содержало просто создаваемое действительное состояние. Объект *vector_base* не мог существовать, если начальное распределение памяти не было успешным. Следовательно, разработчик *vector* мог написать функцию «запасного выхода» подобную следующей:

```
template<class T, class A>
void vector<T, A>::emergency_exit()
{
        space = v;                          // установление размера *this в 0
        throw Total_failure();
}
```

Это излишне «круто»: действуя так, не удается вызвать деструкторы элементов и освободить пространство элементов, удерживаемое ***vector_base***. То есть не обеспечивается основная гарантия (§ Д.2). Если мы хотим доверять значениям ***v*** и ***space*** и деструкторам элементов, следует предотвратить возможную утечку ресурсов:

```
template<class T, class A>
void vector<T, A>::emergency_exit()
{
        destroy_elements();                 // очистка
        throw Total_failure();
}
```

Стандартный ***vector***, заметим, спроектирован настолько аккуратно, что он минимизирует проблемы, вызываемые двухфазным созданием. Функция ***init***() примерно эквивалентна ***resize***(), и почти везде возможность ***v==0*** уже закрыта проверками ***size***()==***0***. Отрицательные эффекты, описанные для двухфазного создания, становятся более заметными при рассмотрении прикладных классов, захватывающих значительные ресурсы, типа сетевых соединений и файлов. Такие классы редко являются частью среды, которая направляет их использование и реализацию тем же образом, каким требования стандартной библиотеки руководят определением и использованием ***vector***. Проблемы также обостряются по мере усложнения соответствия между понятиями приложения и ресурсами, требуемыми для их реализации. Немногие классы отображаются на системные ресурсы так же непосредственно, как ***vector***.

Идея «безопасного состояния» — в принципе хороша. Если мы не можем поместить объект в действительное состояние без риска сгенерировать исключение до завершения операции, у нас и в самом деле проблемы. Однако «безопасное состояние» должно быть естественной частью семантики класса, а не артефактом реализации, усложняющим инвариант класса.

Д.3.5.3. Задержка выделения ресурса

Подобно второму пункту (§ Д.3.5.2), третий (о задержке выделения ресурса до момента, когда он потребуется) неправильно применяет хорошую идею, так что издержки есть, а выгод нет. Во многих случаях, особенно для контейнеров вроде ***vector***, лучший способ отложить выделение ресурса состоит в том, что программист откладывает создание объектов до момента, когда они понадобятся. Рассмотрим наивное использование ***vector***:

```
void f(int n)
{
        vector<X> v(n);          // создание n объектов по умолчанию типа X
        //...
        v[3]=X(99);              // реальная "инициализация" v[3]
```

```
        //...
    }
```

Создание ***X*** только для того, чтобы присвоить ему позже новое значение, расточительно — особенно если присваивание ***X*** дорого. Поэтому может показаться привлекательным двухфазное создание ***X***. Например, тип ***X*** может сам быть ***vector***, так что мы могли бы рассмотреть двухфазное создание ***vector***, чтобы оптимизировать создание пустых векторов. Однако создание заданных по умолчанию (пустых) векторов и так эффективно, поэтому усложнение реализации с частным случаем для пустого вектора представляется бесполезным. Вообще, «очистка» конструкторов элементов от сложной инициализации редко является наилучшим разрешением проблемы фиктивной инициализации. А вот пользователь вполне может создавать элементы только когда необходимо. Например:

```
void f2(int n)
{
    vector<X> v;            // создание пустого вектора
    //...
    v.push_back(X(99));     // создание элемента при необходимости
    //...
}
```

Подводя итог: двухфазное создание ведет к более сложным инвариантам и, как правило, к менее изящному, более подверженному ошибкам и более сложному для сопровождения коду. Следовательно всякий раз, когда это возможно, следует отдавать преимущество поддерживаемому языком «конструкторному подходу» по сравнению с «*init*()-подходом». То есть ресурсы должны выделяться в конструкторах, если только отложенное выделение ресурса не вынужденно унаследованной семантикой класса.

Д.4. Гарантии стандартных контейнеров

Если сама библиотечная операция генерирует исключение, она может удостовериться, что объекты, с которыми она работает, оставлены в хорошо определенном состоянии. Библиотечные функции так и поступают. Например ***at***(), генерирующая ***out_of_range*** для ***vector*** (§ 16.3.3), не создает трудностей с обеспечением безопасности исключений в классе ***vector***. Автор ***at***() без проблем убедится, что перед генерацией ***vector*** находится в хорошо определенном состоянии. Трудности для разработчиков библиотеки, для пользователей библиотеки и для людей, пытающихся разобраться в коде, начинаются, когда исключение генерирует предоставленная пользователем функция.

Контейнеры стандартной библиотеки предлагают основную гарантию (§ Д.2): соблюдаются основные инварианты библиотеки, и нет никаких утечек ресурсов, пока код пользователя ведет себя как подобает. То есть предоставленные пользователем операции не должны оставлять элементы контейнеров в недействительных состояниях или генерировать исключения в деструкторах. Я имею в виду «операции», используемые реализацией стандартной библиотеки, типа конструкторов, присваиваний, деструкторов и операций с итераторами (§ Д.4.4).

Программисту довольно просто обеспечить, чтобы такие операции соответствовали ожиданиям библиотеки. Фактически даже наивно написанный код зачастую соответствует требованиям библиотеки. Следующие типы определенно удовлетворяют требованиям стандартной библиотеки к типам контейнерных элементов:

[1] встроенные типы — включая указатели;
[2] типы без определенных пользователем операций;
[3] классы с операциями, которые не генерируют исключений и не оставляют операнды в недействительных состояниях;
[4] классы, деструкторы которых не генерируют исключений, и для которых просто проверить, что операции, используемые стандартной библиотекой (типа конструкторов, присваиваний, <, == и *swap*()), не оставляют операнды в недействительных состояниях.

В каждом случае мы должны также удостовериться, что нет никаких утечек ресурсов. Например:

```
void f(Circle* pc, Triangle* pt, vector<Shape*>& v2)
{
    vector<Shape*> v(10);          // либо создает vector, либо генерирует bad_alloc
    v[3] = pc;                     // исключения не генерируются
    v.insert(v.begin() + 4, pt);   // либо вставляет pt, либо ничего не делает с v
    v2.erase(v2.begin() + 3);      // либо удаляет v2[3], либо ничего не делает с v2

    v2 = v;                        // либо копирует v, либо ничего не делает с v2
    //...
}
```

Когда *f*() завершится, *v* должным образом уничтожится, а *v2* останется в действительном состоянии. Этот фрагмент не указывает, кто ответственен за удаление *pc* и *pt*. Если ответственна *f*(), она может либо перехватывать исключения и выполнять требуемое удаление, либо присваивать указатели локальным переменным типа *auto_ptr*.

Более интересный вопрос: когда библиотечные операции предлагают сильную гарантию, так что они либо успешно заканчиваются, либо не влияют на свои операнды? Например:

```
void f(vector<X>& vx)
{
    vx.insert(vx.begin() + 4, X(7));        // добавление элемента
}
```

Вообще говоря, операции *X* и распределитель памяти *vector<X>* могут генерировать исключения. Что можно сказать об элементах *vx*, когда *f*() завершается из-за исключения? Основная гарантия заверяет, что нет утечек ресурсов, и что *vx* содержит набор действительных элементов. Но каких именно элементов? Изменился ли *vx*? Могло ли добавиться значение *X* по умолчанию? Мог ли элемент быть удален, потому что это был единственный способ для *insert*() восстановиться с соблюдением основной гарантии? Иногда недостаточно знать, что контейнер находится в хорошем состоянии; мы также хотим знать, что это за состояние. После перехвата исключения обычно необходимо удостовериться, что элементы — точно те, какие и предполагались, в противном случае следует приступить к восстановлению после ошибок.

Д.4.1. Вставка и удаление элементов

Вставка элемента в контейнер и удаление его оттуда — очевидные примеры операций, которые могут оставить контейнер в непредсказуемом состоянии, если сгенерировано исключение. Причина в том, что вставка и удаление вызывают множество операций, которые вправе генерировать исключения:

[1] В контейнер копируется новое значение.
[2] Должен быть уничтожен элемент, удаленный (вычеркнутый) из контейнера.
[3] Иногда под новый элемент должна быть отведена память.
[4] Иногда элементы *vector* и *deque* должны копироваться в новое место.
[5] Ассоциативные контейнеры вызывают функции сравнения для элементов.
[6] Многие вставки и удаления включают операции с итераторами.

Во всех подобных случаях могут генерироваться исключения.

Если деструктор генерирует исключение, не дается никаких гарантий (§ Д.2), так как в подобном случае они обошлись бы непозволительно дорого. Однако библиотека может защитить и защищает себя — и своих пользователей — от исключений, сгенерированных другими предоставленными пользователем операциями.

При обработке связной структуры данных наподобие *list* или *map*, элементы могут добавляться и удаляться без воздействия на другие элементы контейнера. Иначе обстоит дело для контейнера, реализованного с помощью непрерывного размещения элементов, типа *vector* или *deque*. Там элементы иногда должны перемещаться на новые места.

В дополнение к основной гарантии, стандартная библиотека предлагает сильную гарантию для нескольких операций, которые вставляют или удаляют элементы. Поскольку контейнеры, реализованные как связные структуры данных, ведут себя иначе, чем контейнеры с непрерывным размещением элементов, стандарт обеспечивает слегка различные гарантии для разных видов контейнеров:

[1] Гарантии для *vector* (§ 16.3) и *deque* (§ 17.2.3):
- Если исключение сгенерировано в *push_back*() или *push_front*(), функция ни на что не влияет.
- Если исключение сгенерировано в *insert*(), но не копирующим конструктором и не оператором присваивания типа элемента, функция ни на что не влияет.
- *erase*() не генерирует исключений, кроме случаев возникновения исключения в копирующем конструкторе и в операторе присваивания типа элемента.
- *pop_back*() и *pop_front*() не генерируют исключений.

[2] Гарантии для *list* (§ 17.2.2):
- Если исключение сгенерировано *push_back*() или *push_front*(), функция ни на что не влияет.
- Если исключение сгенерировано *insert*(), функция ни на что не влияет.
- *erase*(), *pop_back*(), *pop_front*(), *splice*(), и *reverse*() не генерируют исключений.
- Функции-члены *list*: *remove*(), *remove_if*(), *unique*(), *sort*() и *merge*() не генерируют исключений, кроме случаев возникновения исключения в предикате или в функции сравнения.

[3] Гарантии для ассоциативных контейнеров (§ 17.4):

— Если исключение сгенерировано *insert*() при вставке отдельного элемента, функция ни на что не влияет.

— *erase*() не генерирует исключений.

Обратите внимание, что если для операции на контейнере обеспечивается сильная гарантия, все итераторы, указатели и ссылки на элементы при генерации исключения остаются действительными.

Эти правила можно объединить в таблицу:

Гарантии операций с контейнерами

	vector	*deque*	*list*	*map*
clear()	не генерирует (копирование)	не генерирует (копирование)	не генерирует	не генерирует
erase()	не генерирует (копирование)	не генерирует (копирование)	не генерирует	не генерирует
1-элементная *insert*()	сильная (копирование)	сильная (копирование)	сильная	сильная
N-элементная *insert*()	сильная (копирование)	сильная (копирование)	сильная	основная
merge()	—	—	не генерирует (сравнение)	—
push_back()	сильная	сильная	сильная	—
push_front()	—	сильная	сильная	—
pop_back()	не генерирует	не генерирует	не генерирует	—
pop_front()	—	не генерирует	не генерирует	—
remove()	—	—	не генерирует (сравнение)	—
remove_if()	—	—	не генерирует (предикат)	—
reverse()	—	—	не генерирует	—
splice()	—	—	не генерирует	—
swap()	не генерирует	не генерирует	не генерирует	не генерирует (копирование сравнения)
unique()	—	—	не генерирует (сравнение)	—

В этой таблице:

основная	означает, что операция обеспечивает только основную гарантию (§ Д.2)
сильная	означает, что операция обеспечивает сильную гарантию (§ Д.2)
не генерирует	означает, что операция не генерирует исключений (§ Д.2)
—	означает, что операция не обеспечивается как член данного контейнера

Там где гарантия требует, чтобы некоторые предоставленные пользователем операции не генерировали исключений, эти операции указаны в круглых скобках под гарантией. Эти требования точно сформулированы в тексте, предшествующем таблице.

Функция *swap*() отличается от других упомянутых функций тем что не является членом. Гарантия для *clear*() получена из предлагаемой для *erase*() (§ 16.3.6). Таблица перечисляет гарантии, предоставляемые в дополнение к основной гарантии. Соответственно в таблицу не вошли операции, подобные *reverse*() и *unique*(), для *vector*, которые обеспечиваются только как алгоритмы для всех последовательностей без дополнительных гарантий.

«Почти контейнер» *basic_string* (§ 17.5, § 20.3) предлагает основную гарантию для всех операций (§ Д.5.1). Стандарт также гарантирует, что для *basic_string* функции *erase*() и *swap*() не генерируют исключений, а на функции *insert*() и *push_back*() распространяется сильная гарантия.

Помимо неизменности контейнера, операция, обеспечивающая сильную гарантию, также гарантирует действительность всех оставляемых ею итераторов, указателей, и ссылок. Например:

```
void update(map<string, X>& m, map<string, X>::iterator current)
{
        X x;
        string s;
        while (cin>>s>>x)
        try {
                current = m.insert(current, make_pair(s, x));
        }
        catch (...) {
                // здесь current все еще обозначает текущий элемент
        }
}
```

Д.4.2. Гарантии и компромиссы

Причудливая смесь дополнительных гарантий отражает подлинную сущность реализации. Программисты предпочитают сильную гарантию с как можно меньшим числом оговорок, но они также настаивают, чтобы каждая операция стандартной библиотеки была наиболее эффективна. Оба желания разумны, но для многих операций невозможно удовлетворить их одновременно. Чтобы дать лучшее представление о затронутых компромиссах, я исследую способы добавления одного или нескольких элементов к *list*, *vector* и *map*.

Рассмотрим добавление единственного элемента к *list* или *vector*. Как всегда, *push_back*() обеспечивает самый простой способ:

```
void f(list<X>& lst, vector<X>& vec, const X& x)
{
        try {
                lst.push_back(x);                    // добавление к списку
        }
        catch (...) {
                // lst не изменен
                return;
        }
        try {
                vec.push_back(x);                    // добавление к вектору
```

```
        }
        catch(...) {
            // vec не изменен
            return
        }

        // lst и vec содержат по новому элементу со значением x
    }
```

Обеспечение сильной гарантии в этих случаях просто и дешево. Она очень полезна, потому что предоставляет полностью безопасный при исключениях способ добавления элементов. Однако ***push_back***() не определена для ассоциативных контейнеров — у ***map*** нет ***back***(). В конце концов, последний элемент ассоциативного контейнера определяется отношением порядка, а не позицией.

Гарантии для ***insert***() немного запутаннее. Причина в том, что иногда ***insert***() приходится помещать элемент в «середину» контейнера. Это несложно для связной структуры данных, типа ***list*** или ***map***. С другой стороны, если за ***vector*** зарезервировано свободное пространство, очевидная реализация ***vector<X>::insert***() скопирует элементы после точки вставки, чтобы освободить место. Это наиболее эффективно, зато нет простого пути восстановления ***vector***, если присваивание копии ***X*** или копирующий конструктор генерирует исключение (см. § Д.8[10–11]). Следовательно ***vector*** обеспечивает гарантию, которая обусловлена тем, чтобы операция копирования элементов не генерировала исключений. А ***list*** и ***map*** не нуждаются в подобном условии; они запросто могут присоединить новые элементы уже после выполнения любого необходимого копирования.

Как пример предположим, что присваивание копии и копирующий конструктор ***X*** генерируют ***X::cannot_copy***, если они не могут успешно создать копию:

```
void f(list<X>& lst, vector<X>& vec, map<string, X>& m, const X& x, const string& s)
{
    try {
        lst.insert(lst.begin(), x);          // добавление к списку
    }
    catch (...) {
        // lst не изменен
        return;
    }

    try {
        vec.insert(vec.begin(), x);          // добавление к вектору
    }
    catch (X::cannot_copy) {
        // Стоп: vec может содержать, а может и не содержать новый элемент
    }
    catch (...) {
        // vec не изменен
        return;
    }
    try {
        m.insert(make_pair(s, x));           // добавление к map
    }
```

```
        catch (...) {
            // m не изменен
            return;
        }
        // lst и vec содержат по новом элементу со значением x
        // m содержит элемент со значением (s, x)
}
```

Если перехвачено ***X::cannot_copy***, новый элемент либо вставился, либо не вставился в ***vec***. Если новый элемент был вставлен, объект окажется в действительном состоянии, но каково его значение — точно не определено. Возможно, после ***X::cannot_copy*** некоторый элемент окажется «мистически» продублирован (см. § Д.8[11]). Или ***insert()*** может быть реализована с удалением «хвостовых» элементов, чтобы в контейнере наверняка не осталось «посторонних».

К сожалению, обеспечение сильной гарантии для ***insert()*** в ***vector*** без предостережения об исключениях, сгенерированных операциями копирования, не выполнимо. Стоимость полной защиты от исключения при перемещении элементов в ***vector*** значительна сравнительно с обеспечением в этом случае лишь основной гарантии.

Типы элементов с операциями копирования, которые могут генерировать исключения, не редки. Примеры из самой стандартной библиотеки — ***vector<string>***, ***vector<vector<double>>*** и ***map<string, int>***.

Контейнеры ***list*** и ***vector*** обеспечивают одинаковые гарантии при вставке как одного, так и многих элементов. Причина в том, что реализация ***insert()*** для ***vector*** и ***list*** применяет одну и ту же стратегию вставки как одного, так и нескольких элементов. В то же время ***map*** обеспечивает сильную гарантию для одноэлементной ***insert()***, но лишь основную гарантию для многоэлементной ***insert()***. Легко реализовать вставку одного элемента в ***map***, которая обеспечивает сильную гарантию. Однако очевидная стратегия для осуществления многоэлементной вставки в ***map*** состоит в том, чтобы вставлять новые элементы один за другим, а при этом не просто обеспечить сильную гарантию. Проблема в том, что нет простого способа отката предыдущих успешных вставок, если вставка очередного элемента терпит неудачу.

Функцию вставки, которая обеспечивает сильную гарантию (либо все элементы успешно добавлены, либо операция ни на что не повлияла), можно при желании построить, создав новый контейнер с последующим обменом ***swap()***:

```
template<class C, class Iter>
void safe_insert(C& c, typename C::const_iterator i, Iter begin, Iter end)
{
    C tmp(c.begin(), i);                          // копирование начальных
                                                  // элементов в tmp
    copy(begin, end, inserter(tmp, tmp.end()));   // копирование новых
                                                  // элементов
    copy(i, c.end(), inserter(tmp, tmp.end()));   // копирование конечных
                                                  // элементов
    swap(c, tmp);
}
```

Как всегда, этот код может дурно себя вести, если деструктор элемента генерирует исключение. Однако если исключение генерируется операцией копирования элемента, контейнер-аргумент не изменяется.

Д.4.3. Перестановка swap()

Подобно копирующим конструкторам и присваиваниям, операции ***swap***() необходимы для многих стандартных алгоритмов и часто предоставляются пользователями. Например ***sort***() и ***stable_sort***() обычно переупорядочивают элементы с помощью ***swap***(). Таким образом, если функция ***swap***() генерирует исключение при перестановке значений из контейнера, этот контейнер может остаться с непереставленными элементами или с дублированным элементом, а не с парой элементов, поменявшихся местами.

Рассмотрим очевидное определение функции ***swap***() стандартной библиотеки (§ 18.6.8):

```
template<class T> void swap(T& a, T& b)
{
        T tmp = a;
        a = b;
        b = tmp;
}
```

Ясно что ***swap***() не генерирует исключений, если ни копирующий конструктор типа элемента, ни копирующее присваивание не делают этого.

С одним незначительным исключением для ассоциативных контейнеров, относительно функций ***swap***() стандартных контейнеров гарантируется, что они не генерируют исключений. В сущности контейнеры переставляют элементы, меняя местами структуры данных, которые действуют как дескрипторы элементов (§ 13.5, § 17.1.3). Так как сами элементы не перемещаются, конструкторы и присваивания не вызываются, так что им и не сгенерировать исключений. Кроме того стандарт гарантирует, что никакая функция ***swap***() стандартной библиотеки не лишает законной силы ссылки, указатели или итераторы, ссылающиеся на элементы переставляемых контейнеров. Имеется лишь один потенциальный источник исключений: объект сравнения в ассоциативном контейнере копируется как часть дескриптора. Единственное возможное исключение в ***swap***() стандартных контейнеров генерируетсякопирующим конструктором или присваиванием объекта сравнения контейнера (§ 17.1.4.1). К счастью, операции копирования объектов сравнения обычно тривиальны и не имеют возможности сгенерировать исключение.

Предоставленная пользователем ***swap***() должна быть написана так, чтобы обеспечить те же самые гарантии. Это относительно просто сделать, если автор не забывает менять местами типы, представленные дескрипторами, перестановкой дескрипторов, а не медленным и детальным копированием информации, на которую упомянутые дескрипторы ссылаются (§ 13.5, § 16.3.9, § 17.1.3).

Д.4.4. Инициализация и итераторы

Распределение памяти для элементов и инициализация выделенной памяти — основные части реализации контейнера (§ Д.3). Поэтому гарантируется, что стандартные алгоритмы для построения объектов в неинициализированной памяти — ***uninitialized_fill***(), ***uninitialized_fill_n***() и ***uninitialized_copy***() (§ 19.4.4) — не оставляют после себя созданных объектов в случае генерации ими исключений. При этом обеспечивается сильная гарантия (§ Д.2). Поскольку инициализация памяти иногда

включает уничтожение элементов, требование к деструкторам не генерировать исключений является существенным для алгоритмов данного типа; см. § Д.8[14]. Кроме того требуется чтобы итераторы, передаваемые в качестве аргументов этих алгоритмов, вели себя корректно. То есть они должны быть действительными итераторами, ссылаться на действительные последовательности, а итераторным операциям (типа ++, != и *), примененным к действительным итераторам, не разрешается генерировать исключений.

Итераторы — примеры объектов, которые свободно копируются стандартными алгоритмами и операциями со стандартными контейнерами. Соответственно, копирующие конструкторы и копирующие присваивания итераторов не должны генерировать исключений. В частности стандарт гарантирует, что копирующий конструктор или оператор присваивания итератора, возвращаемого стандартным контейнером, не генерируют исключений. Например итератор, возвращаемый *vector<T>::begin()*, можно копировать, не опасаясь исключений.

Обратите внимание, что ++ и -- для итератора могут генерировать исключения. Например, *istreambuf_itterator* (§ 19.2.6) может обоснованно сгенерировать исключение, чтобы указать на ошибку ввода, а итератор с проверкой диапазона вправе сгенерировать исключение в ответ на попытку покинуть разрешенный для него диапазон (§ 19.3). Однако операции инкремента и декремента не могут сгенерировать исключения при перемещении итератора от одного элемента последовательности к другому, не нарушив этим определение ++ и -- для итератора. Таким образом, *uninitialized_fill()*, *uninitialized_fill_n()* и *uninitialized_copy()* предполагают, что ++ и -- для их итераторных параметров не будут генерировать исключений; если все же генерируют, значит либо переданные «итераторы», согласно стандарту, — вовсе и не итераторы, либо итерируемая ими «последовательность» не является настоящей последовательностью. И вновь стандартные контейнеры не защищают пользователя от его собственного неопределенного поведения (§ Д.2).

Д.4.5. Ссылки на элементы

Когда ссылка, указатель или итератор на элемент передается некоторому коду, этот код может испортить какой-либо контейнер, повредив один из его элементов. Например:

```
void f(const X& x)
{
        list<X> lst;
        lst.push_back(x);
        list<X>::iterator i = lst.begin();
        *i = x;                          // копирование x в список
        // ...
}
```

Если *x* испорчен, деструктор списка, возможно, будет не способен должным образом уничтожить *lst*. Например:

```
struct X {
        int* p;
        X() { p = new int; }
```

```
        ~X() { delete p; }
        //...
};

void malicious()                        // злонамеренная функция
{
        X x;
        x.p = reinterpret_cast<int*>(7);    // испортили x
        f(x);                               // бомба с часовым механизмом
}
```

Когда выполнение доходит до конца *f()*, вызывается деструктор *list<X>*, а он в свою очередь вызывает деструктор *X* для испорченного значения. Результат выполнения *delete p*, когда *p* не *0* и не указывает на *X*, не определен, вплоть до немедленного краха. Или в результате свободная память может оказаться испорченной так, что много позже возникнут трудно отслеживаемые проблемы в очевидно независимой части программы.

Возможность подобных разрушений не должна останавливать нас от манипулирования элементами контейнеров через ссылки и итераторы; именно этот образ действий зачастую оказывается самым простым и наиболее эффективным. Однако будет мудро проявить дополнительную предосторожность по отношению к ссылкам на элементы контейнеров. Когда целостность контейнера имеет решающее значение, заслуживает внимания мысль предложить менее опытным пользователям безопасную альтернативу. Например мы могли бы предоставить операцию, которая проверяет действительность нового элемента перед его копированием в важный контейнер. Естественно, такая проверка возможна только при знании прикладных типов.

Вообще если элемент контейнера испорчен, последующие операции с контейнером могут потерпеть неудачу самым скверным образом. Это — не особенность контейнеров. Любой объект, оставленный в плохом состоянии, может вызывать последующий сбой.

Д.4.6. Предикаты

Многие стандартные алгоритмы и операции со стандартными контейнерами полагаются на предикаты, которые предоставляются пользователями. В частности, все ассоциативные контейнеры зависят от предикатов как при поиске, так и при вставке.

Предикат, используемый операцией стандартного контейнера, может сгенерировать исключение. На этот случай каждая операция стандартной библиотеки обеспечивает основную гарантию, а некоторые операции, наподобие вставки отдельного элемента, обеспечивают сильную гарантию (§ Д.4.1). Если предикат генерирует исключение из операции с контейнером, окончательный набор элементов контейнера может оказаться не совсем таким, какой хотел пользователь, но это будет набор действительных элементов. Например, если == генерирует исключение, когда вызывается из *list::unique()* (§ 17.2.2.3), пользователь не вправе полагать, что в списке нет дубликатов. Каждый элемент в списке действителен — и это все, что пользователь может уверенно ожидать (см. § Д.5.3).

К счастью, предикаты редко делают что-нибудь, что могло бы сгенерировать исключение. Однако при рассмотрении безопасности исключений следует принять во внимание определяемые пользователем предикаты <, ==, и !=.

При обращении к *swap*() копируется объект сравнения ассоциативного контейнера (§ Д.4.3). Следовательно, имеет смысл обеспечить, чтобы операции копирования предикатов, которые могли бы использоваться как объекты сравнения, не генерировали исключений.

Д.5. Другие части стандартной библиотеки

Решающий элемент обеспечения безопасности исключений состоит в поддержке согласованности объектов: мы должны соблюдать основные инварианты для индивидуальных объектов и согласованность наборов объектов. В контексте стандартной библиотеки объекты, для которых труднее всего обеспечить безопасность исключений, — это контейнеры. С точки зрения безопасности исключений остальная часть стандартной библиотеки менее интересна. Однако отметим, что в свете безопасности исключений встроенный массив является контейнером, который может быть испорчен опасной операцией.

Вообще функции стандартной библиотеки генерируют только исключения, дозволенные им спецификациями, плюс генерируемые пользовательскими операциями, которые вызываются из библиотечных функций. Кроме того любая функция, которая (прямо или косвенно) распределяет память, может сгенерировать исключение, указывающее на исчерпание памяти (как правило, *std*::*bad_alloc*).

Д.5.1. Строки

Операции со строками *string* могут генерировать самые разнообразные исключения. Однако *basic_string* манипулирует своими символами через функции, предоставленные *char_traits* (§ 20.2), а этим функциям не позволено генерировать исключения. То есть *char_traits*, предоставленный стандартной библиотекой, не генерирует исключений, но никаких гарантий не дается на случай, если операция определенного пользователем *char_traits* генерирует исключение. В частности обратите внимание, что типу, используемому как элементный (символьный) тип для *basic_string*, не дозволено иметь определяемый пользователем копирующий конструктор или определяемое пользователем копирующее присваивание. Тем самым мы избавляемся от важного возможного источника исключений.

Класс *basic_string* очень похож на стандартный контейнер (§ 17.5, § 20.3). В самом деле, его элементы составляют последовательность, к которой можно обращаться с помощью *basic_string*<*Ch*, *Tr*, *A*>::*iterator* и *basic_string*<*Ch*, *Tr*, *A*>::*const_iterator*. Соответственно, реализация строки предлагает основную гарантию (§ Д.2), а гарантии для *erase*(), *insert*(), *push_back*() и *swap*() (§ Д.4.1) относятся и к *basic_string*. Например, *basic_string*<*Ch*,*Tr*,*A*>::*push_back*() обеспечивает сильную гарантию.

Д.5.2. Потоки

Если требуется, функции *iostream* генерируют исключения, чтобы сообщить об изменениях состояния (§ 21.3.6). Соответствующая семантика хорошо определена и не создает трудностей по части безопасности исключений. Генерация исключения пользовательскими *operator*<<() или *operator*>>() для самого пользователя может выглядеть так, словно исключение сгенерировала библиотека *iostream*. Однако та-

кое исключение не повлияет на состояние потока (§ 21.3.3). Дальнейшие операции потока быть может и не найдут ожидаемых данных — потому что предыдущая операция сгенерировала исключение вместо нормального завершения — но сам по себе поток не повреждается. Как всегда при возникновении проблем ввода/вывода, перед продолжением чтения или записи может потребоваться вызов *clear*() (§ 21.3.3, § 21.3.5).

Подобно *basic_string*, при манипуляциях с символами потоки *iostream* полагаются на *char_traits* (§ 20.2.1, § Д.5.1). Таким образом реализация может предполагать, что операции с символами не генерируют исключений, и не дается никаких гарантий, если пользователь нарушает это предположение.

Чтобы сделать возможной значительную оптимизацию кода, предполагается что классы *locale* (§ Г.2) и *facet* (§ Г.3) не генерируют исключений. Если бы они генерировали, использующий их поток мог бы повредиться. В то же время наиболее вероятное исключение, *std*::*bad_cast* из *use_facet* (§ Г.3.1), может сгенерироваться только в предоставленном пользователем коде вне стандартной реализации потока. В худшем случае это приведет к неполному выводу или неудачному чтению, но не к повреждению самого потока *ostream* (или *istream*).

Д.5.3. Алгоритмы

Кроме *uninitialized_copy*(), *uninitialized_fill*(), и *uninitialized_fill_n*() (§ Д.4.4), стандарт предлагает для алгоритмов только основную гарантию (§ Д.2). То есть при условии, что предоставленные пользователем объекты ведут себя корректно, алгоритмы соблюдают все инварианты стандартной библиотеки и не приводят к утечке ресурсов. Чтобы избежать неопределенного поведения, предоставленные пользователем операции должны всегда оставлять свои операнды в действительных состояниях, а деструкторы не должны генерировать исключений.

Сами алгоритмы не генерируют исключений. Вместо этого они сообщают об ошибках и сбоях через свои возвращаемые значения. Например поисковые алгоритмы в качестве сообщения «не найдено» обычно возвращают конец последовательности (§ 18.2). Таким образом исключения, сгенерированные в стандартных алгоритмах, на самом деле возникают в предоставленной пользователем операции. Эти исключения происходят либо в операции на элементе, такой как предикат (§ 18.4), присваивание или *swap*(), либо в распределителе памяти (§ 19.4).

Если подобная операция генерирует исключение, алгоритм немедленно завершается, а обработка исключения остается на совести тех функций, которые вызвали алгоритм. Некоторые алгоритмы допускают генерацию исключения, когда контейнер находится в плохом с точки зрения пользователя состоянии. Так ряд алгоритмов сортировки временно копируют элементы в буфер и позже возвращают их обратно в контейнер. Такой *sort*() мог бы скопировать элементы из контейнера (планируя записать их назад в надлежащем порядке позже), начать записывать что-то поверх них, а затем сгенерировать исключение. С точки зрения пользователя, контейнер испорчен. Однако все элементы находятся в действительном состоянии, так что восстановление должно быть довольно прямолинейным.

Обратите внимание, что стандартные алгоритмы обращаются к последовательностям через итераторы. То есть стандартные алгоритмы никогда не работают с контейнерами непосредственно, только с элементами контейнеров. То, что стандартный ал-

горитм никогда непосредственно не добавляет и не удаляет элементы из контейнера, упрощает анализ последствий исключений. Точно так же, если к структуре данных обращаются только через итераторы, указатели и ссылки на ***const*** (например, через ***const Rec****), обычно тривиально проверяется, что исключение не вызывает никаких нежелательных эффектов.

Д.5.4. Valarray и Complex

Числовые функции явно не генерируют исключений (глава 22). Однако ***valarray*** должен распределить память и, значит, может сгенерировать ***std::bad_alloc***. Кроме того, шаблонам ***valarray*** и ***complex*** можно передать тип элементов (скалярный тип), который генерирует исключения. Как всегда, стандартная библиотека обеспечивает основную гарантию (§ Д.2), но никаких особых гарантий не дается относительно результатов вычисления, завершившегося исключением.

Подобно ***basic_string*** (§ Д.5.1), классам ***valarray*** и ***complex*** разрешается полагать, что у типа аргумента их шаблона нет определенных пользователем операций копирования, так что допустимо побитное копирование. Как правило, эти числовые типы стандартной библиотеки оптимизированы по быстродействию в предположении, что тип их элементов (скалярный тип) не генерирует исключений.

Д.5.5. Стандартная библиотека Си

Операция стандартной библиотеки без спецификации исключений может генерировать исключения определяемым реализацией способом. С другой стороны, функции стандартной библиотеки Си не генерируют исключений, если у них нет параметра-функции, которая генерирует. Ведь функции — часть Си, а в Си нет исключений. Реализация вправе объявить стандартные функции Си с пустой спецификацией исключений ***throw()***, чтобы помочь компилятору сгенерировать лучший код.

Функции типа ***qsort()*** и ***bsearch()*** (§ 18.11) принимают в качестве аргумента указатель на функцию. Поэтому они могут сгенерировать исключение, если это делают их параметры. Основная гарантия (§ Д.2) распространяется и на эти функции.

Д.6. Значение для пользователей библиотеки

На безопасность исключений в контексте стандартной библиотеки можно смотреть так: у нас не будет никаких проблем, если мы не создадим их на собственную голову. Библиотека будет функционировать правильно, пока предоставленные пользователем операции удовлетворяют основным требованиям стандартной библиотеки (§ Д.2). В частности исключение, сгенерированное операцией стандартного контейнера, не вызовет утечки памяти из контейнера и не оставит контейнер в недействительном состоянии. Таким образом, перед пользователем библиотеки стоит следующий вопрос: как мне определить свои типы таким образом, чтобы они не вызвали неопределенного поведения или утечки ресурсов? Вот основные правила:

[1] При модификации объекта не уничтожайте его старое представление до того, как полностью будет создано новое представление и оно сможет заменить старое без риска исключений. Например, см. реализацию ***vector::operator=()***, ***safe_assign()*** и ***vector::push_back()*** в § Д.3.

[2] Перед генерацией исключения освободите все захваченные ресурсы, которые не принадлежат какому-нибудь (другому) объекту.

[2а] Методика «выделение ресурса есть инициализация» (§ 14.4) и правило языка, что частично созданные объекты уничтожаются в той степени, в какой они были созданы (§ 14.4.1), могут быть здесь наиболее полезны. Например, см. ***leak***() в § Д.2.

[2б] Алгоритм ***uninitialized_copy***() и его собратья обеспечивают автоматическое освобождение ресурсов при невозможности завершить создание набора объектов (§ Д.4.4).

[3] Перед генерацией исключения удостоверьтесь, что каждый операнд находится в действительном состоянии. То есть оставляйте каждый объект в состоянии, которое позволяет обратиться к нему и уничтожить его, не вызвав неопределенного поведения и генерации исключения из деструктора. Например, см. присваивание ***vector*** в § Д.3.2.

[3а] Конструкторы особенны тем, что при генерации в них исключения не оставляют после себя объектов для последующего уничтожения. Соответственно, перед генерацией исключения мы не должны устанавливать инвариант, но обязаны убедиться, что освободили все ресурсы, захваченные в ходе неудавшегося создания.

[3б] Деструкторы особенны тем, что сгенерированное в них исключение почти всегда ведет к нарушению инвариантов и/или вызову ***terminate***().

Следовать этим правилам на практике может оказаться неожиданно трудно. Основная причина в том, что исключения иногда генерируются там, откуда их никто не ждет. Хороший пример — ***std***::***bad_alloc***. Каждая функция, которая прямо или косвенно использует ***new*** или ***allocator*** для выделения памяти, может сгенерировать ***bad_alloc***. В некоторых программах мы можем решить эту частную проблему, не исчерпывая памяти. Однако в случае программ, которые рассчитаны на длительную непрерывную работу или на произвольный объем входных данных, мы должны быть готовы к обработке самых разнообразных отказов при запросе ресурсов. Таким образом, каждую функцию следует подозревать в способности сгенерировать исключение, пока не доказано обратное.

Простой способ попытаться избежать неожиданностей состоит в использовании контейнеров элементов, которые не генерируют исключений (типа контейнеров указателей и контейнеров простых конкретных типов), или связных контейнеров (например ***list***), которые обеспечивают сильную гарантию (§ Д.4). Другой (дополняющий) подход — положиться прежде всего на операции вроде ***push_back***(), которые предоставляют сильную гарантию, так что операция либо успешно заканчивается, либо ни на что не влияет (§ Д.2). Однако сами по себе эти подходы недостаточны для предотвращения утечек ресурсов; они могут привести к очень специальному, чрезмерно ограничительному и пессимистическому методу обработки ошибок и восстановления. Например ***vector<T****> — тривиально безопасный при исключениях тип, если операции с ***T*** не генерируют исключений. Однако если указываемые объекты нигде не удаляются, исключение из ***vector*** приведет к утечке ресурсов. Введение вспомогательного класса ***Handle*** для освобождения ресурсов (§ 25.7) и применение ***vector<Handle<T>>*** вместо ***vector<T****>, вероятно, улучшит жизнеспособность кода.

При написании нового кода можно принять более систематический подход и добиться, чтобы каждый ресурс представлялся классом с инвариантом, обеспечивающим основную гарантию (§ Д.2). Такой подход позволяет выделить наиболее важные объекты приложения и обеспечить семантику отката для операций с ними (то есть сильную гарантию — возможно, при каких-то условиях).

Большинство приложений содержат структуры данных и код, которые написаны без учета безопасности исключений. Где необходимо, такой код может быть встроен в безопасную при исключениях среду либо после проверки, что он не генерирует исключений (как стандартная библиотека Си; § Д.5.5), либо с помощью интерфейсных классов, для которых поведение при исключениях и управление ресурсами может быть точно определено.

При проектировании типов, предназначенных для использования в безопасной при исключениях среде, следует обратить особое внимание на операции, используемые стандартной библиотекой: конструкторы, деструкторы, присваивания, сравнения, функции перестановки, предикаты и операции на итераторах. Наилучшее решение состоит в определении инварианта класса, который просто устанавливается всеми конструкторами. Иногда необходимо проектировать инвариант класса так, чтобы можно было перевести объект в состояние, в котором он может быть уничтожен, даже если операция терпит неудачу в «неудобной» точке. В идеале такое состояние не является артефактом, введенным для поддержки обработки исключений, а естественным образом вытекает из семантики типа (§ Д.3.5).

При рассмотрении безопасности исключений акцент должен делаться на определении действительных состояний для объектов (инвариантов) и на надлежащем освобождении ресурсов. Поэтому важно представлять ресурсы как классы. Класс ***vector_base*** (§ Д.3.2) — простой тому пример. Конструкторы таких «ресурсных» классов захватывают ресурсы нижнего уровня (типа необработанной памяти для ***vector_base***) и устанавливают инварианты (скажем, надлежащим образом инициализируют указатели ***vector_base***). Деструкторы подобных классов неявно освобождают ресурсы нижнего уровня. Правила для частичного создания (§ 14.4.1) и методика «выделение ресурса есть инициализация» (§ 14.4) поддерживают этот путь управления ресурсами.

Хорошо написанный конструктор устанавливает инвариант класса для объекта (§ 24.3.7.1). То есть конструктор присваивает объекту значение, которое делает написание последующих операций простым делом и позволяет этим операциям успешно завершаться. Тем самым предполагается, что конструктору часто приходится захватывать ресурсы. Если сделать этого не удается, конструктор может сгенерировать исключение, чтобы мы могли заняться проблемой еще до создания объекта. Такой подход непосредственно поддерживается языком и стандартной библиотекой (§ Д.3.5).

Требование освободить ресурсы и поместить операнд в действительное состояние перед генерацией исключения означает, что бремя обработки исключений разделяется между генерирующей функцией, всеми функциями в цепочке вызовов до обработчика и обработчиком. Генерация исключения не делает обработку ошибки «не моей проблемой». Функции, генерирующие или передающие исключения, обязаны освободить ресурсы, которыми владеют, и поместить операнды в непротиворечивые состояния. Если они этого не делают, обработчик исключений может разве что попробовать изящно завершить программу.

Д.7. Советы

[1] Определитесь со степенью безопасности исключений, которая вам нужна; § Д.2.

[2] Безопасность исключений должна быть частью общей стратегии отказоустойчивости; § Д.2.

[3] Обеспечьте основную гарантию для всех классов, то есть соблюдайте инвариант и не допускайте утечек ресурсов; § Д.2. § Д.3.2, § Д.4.

[4] Где возможно и не слишком накладно, обеспечьте сильную гарантию, чтобы операция либо успешно завершалась, либо оставляла все операнды неизменными; § Д.2, § Д.3.

[5] Не генерируйте исключений в деструкторах; § Д.2, § Д.3.2, § Д.4.

[6] Не генерируйте исключений в итераторах, продвигающихся по действительной последовательности; § Д.4.1, § Д.4.4.

[7] Безопасность исключений требует тщательного исследования отдельных операций; § Д.3.

[8] Разрабатывайте шаблоны прозрачными к исключениям; § Д.3.1.

[9] Отдавайте преимущество «конструкторному подходу» к запросу ресурсов по сравнению с использованием функции *init*(); § Д.3.5.

[10] Определяйте инвариант для класса, чтобы было ясно, что является действительным состоянием; § Д.2, § Д.6.

[11] Удостоверьтесь, что объект всегда может быть помещен в действительное состояние без риска генерации исключения; § Д.3.2, § Д.6.

[12] Придерживайтесь простых инвариантов; § Д.3.5.

[13] Перед генерацией исключения оставляйте все операнды в действительных состояниях; § Д.2, § Д.6.

[14] Избегайте утечек ресурсов; § Д.2, § Д.3.1, § Д.6.

[15] Представляйте ресурсы непосредственно; § Д.3.2, § Д.6.

[16] Помните, что *swap*() может иногда быть альтернативой копированию элементов; § Д.3.3.

[17] Где возможно, полагайтесь на упорядочение операций, а не на явное использование *try*-блоков; § Д.3.4.

[18] Не уничтожайте «старую» информацию, пока не будет благополучно произведена ее замена; § Д.3.3, § Д.6.

[19] Полагайтесь на методику «выделение ресурса есть инициализация»; § Д.3, § Д.3.2, § Д.6.

[20] Удостоверьтесь, что операции сравнения для ассоциативных контейнеров могут копироваться; § Д.3.3.

[21] Выделяйте наиболее важные структуры данных и предоставляйте для них операции, которые обеспечивают сильную гарантию; § Д.6

Д.8. Упражнения

1.(*1) Перечислите все исклюния, которые могли бы сгенерироваться в *f*() из § Д.1.

2.(*1) Ответьте на вопросы после примера из § Д.1.

3.(*1) Определите класс ***Tester***, который иногда генерирует исключения в основных операциях, например в копирующем конструкторе. С помощью ***Tester*** проверьте контейнеры вашей стандартной библиотеки.

4. (*1) Найдите ошибку в «неряшливой» версии конструктора ***vector*** (§ Д.3.1) и напишите программу, вызывающую его крах. Подсказка: сначала реализуйте деструктор ***vector***.
5. (*2) Реализуйте простой список, обеспечивающий основную гарантию. Будьте очень конкретны относительно того, что требует список от своих пользователей, чтобы обеспечить гарантию.
6. (*3) Реализуйте простой список, обеспечивающий сильную гарантию. Тщательно проверьте этот список. Обоснуйте, почему люди должны верить, что он будет безопасным.
7. (*2.5) Еще раз реализуйте ***String*** из § 11.12, чтобы он был столь же безопасным, как стандартный контейнер.
8. (*2) Сравните время выполнения различных версий присваивания ***vector*** и ***safe_assign***() (§ Д.3.3).
9. (*1.5) Скопируйте распределитель памяти без использования оператора присваивания (как это необходимо для улучшения ***operator***=() в § Д.3.3).
10. (*2) Добавьте к ***vector*** функции ***erase***() и ***insert***(), удаляющие/вставляющие один или нескольких элементов, которые обеспечивают основную гарантию (§ Д.3.2).
11. (*2) Добавьте к ***vector*** функции ***erase***() и ***insert***(), удаляющие/вставляющие один или нескольких элементов, которые обеспечивают сильную гарантию (§ Д.3.2). Сравните стоимость и сложность этих решений и решений из упражнения 10.
12. (*2) Напишите ***safe_insert***() (§ Д.4.2), которая вставляет элементы в существующий ***vector*** (а не копирует во временную переменную). Какие ограничения придется наложить на операции?
13. (*2) Напишите ***safe_insert***() (§ Д.4.2), которая вставляет элементы в существующий ***map*** (а не копирует во временную переменную). Какие ограничения придется наложить на операции?
14. (*2.5) Сравните размер, сложность и производительность функций ***safe_insert***() из упражнений 12 и 13 с ***safe_insert***() из § Д.4.2.
15. (*2.5) Напишите лучший (более простой и быстрый) ***safe_insert***() только для ассоциативных контейнеров. С помощью свойств (traits) напишите ***safe_insert***(), который автоматически выбирает оптимальный ***safe_insert***() для контейнера. Подсказка: § 19.2.3.
16. (*2.5) Попробуйте переписать ***uninitialized_fill***() (§ 19.4.4, § Д.3.1) так, чтобы он справлялся с деструкторами, которые генерируют исключения. Это возможно? Если да, какова стоимость? Если нет, почему?
17. (*2.5) Попробуйте переписать ***uninitialized_fill***() (§ 19.4.4, § Д.3.1) так, чтобы он справлялся с итераторами, которые генерируют исключения для ++ и --. Это возможно? Если да, какова стоимость? Если нет, почему?
18. (*3) Выберите библиотечный контейнер, но не из стандартной библиотеки. Ознакомьтесь с его документацией и выясните, какие гарантии безопасности исключений он обеспечивает. Проделайте несколько испытаний, чтобы увидеть, насколько он устойчив при исключениях, сгенерированных распределением памяти и предоставленным пользователем кодом. Сравните с соответствующим контейнером стандартной библиотеки.
19. (*3) Попробуйте оптимизировать ***vector*** из § Д.3, пренебрегая возможностью исключений. Например, удалите все ***try***-блоки. Сравните производительность

с версией из § Д.3 и с реализацией ***vector*** из стандартной библиотеки. Сравните также размер и сложность исходного кода этих различных векторов.

20. (*1) Определите инварианты для ***vector*** (§ Д.3) с возможностью ***v == 0*** и без нее (§ Д.3.5).
21. (*2.5) Прочитайте исходный текст реализации ***vector***. Какие гарантии осуществлены для присваивания, многоэлементный ***insert***() и ***resize***()?
22. (*3) Напишите версию ***hash_map*** (§ 17.6) столь же безопасную, как стандартный контейнер.