Философия C++

Практическое программирование

БРЮС ЭККЕЛЬ
ЧАК ЭЛЛИСОН

ПИТЕР

BRUCE ECKEL
CHUCK ALLISON

Thinking in C++

Volume Two: Practical Programming

БРЮС ЭККЕЛЬ
ЧАК ЭЛЛИСОН

Философия C++

Практическое программирование

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2004

Брюс Эккель, Чак Эллисон

Философия C++. Практическое программирование

Перевел с английского Е. Матвеев

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*А. Крузенштерн*
Научный редактор	*Е. Матвеев*
Литературный редактор	*А. Жданов*
Иллюстрации	*М. Шендерова*
Художник	*М. Соколинская, Н. Биржаков*
Корректоры	*Н. Лукина, И. Смирнова*
Верстка	*А. Келле-Пелле*

ББК 32.973-018.1
УДК 681.3.06

Эккель Б., Эллисон Ч.

Э 38 Философия C++. Практическое программирование. — СПб.: Питер, 2004. — 608 с.: ил.

ISBN 5-469-00043-5

Книга отличается от других учебников по C++ новым подходом к изложению материала, основанным на логике и здравом смысле. Вместо простого перечисления языковых конструкций, снабженных примерами, авторы стремятся научить читателя мыслить категориями C++. Они подробно объясняют, почему проектировщики языка принимали то или иное решение, описывают типичные проблемы и пути их решения.

Во втором томе рассматриваются такие темы, как обработка исключений, стандартная библиотека C++ (включая потоки ввода/вывода, контейнеры и алгоритмы STL), шаблоны, множественное наследование, RTTI, автоматизация тестирования и отладки программ, паттерны проектирования и т. д.

ISBN 5-469-00043-5

ISBN 0130353132 (англ.)

ООО «Питер Принт», 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Лицензия ИД № 05784 от 07.09.01.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.
Подписано в печать 23.06.04. Формат 70×100/16. Усл. п. л. 73,53. Тираж 3500 экз. Заказ № 2808.
Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького Министерства РФ
по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Содержание

Часть 3. Специальные возможности

Тем, кто без устали работал, развивая язык C++

Предисловие

В первом томе книги излагались основы C и C++. В этом томе рассматриваются нетривиальные возможности, при этом особое внимание уделено методике разработки и концепциям построения надежных программ C++.

Предполагается, что читатель уже знаком с материалом, представленным в первом томе.

Цели

Далее перечислены цели, которые авторы ставили при написании книги.

- Представление материала простыми «порциями», чтобы читатель мог легко усвоить каждую концепцию, прежде чем следовать дальше.
- Обучение приемам «практического программирования», используемым в повседневной работе.
- Важность излагаемого материала для понимания языка (это далеко не все, что мы знаем). На взгляд авторов, информация не равноценна по своей важности. Некоторые темы 95 % программистов никогда не понадобятся, лишь запутают их и повысят субъективную сложность языка. Для примера обратимся к C: если выучить таблицу приоритета операторов (чего авторами сделано не было), вы сможете строить более короткие выражения. Но если *вам* придется думать над написанием выражения, то оно наверняка вызовет трудности у читателя программы. Так что забудьте о приоритетах, а в неоднозначных ситуациях ставьте круглые скобки. То же относится к некоторым аспектам языка C++, которые, как нам кажется, больше интересуют разработчиков компиляторов, а не программистов.
- Ограниченный объем разделов, чтобы продолжительность «лекций» оставалась относительно небольшой. Это не только повышает активность и за-

интересованность читателей, но и способствует созданию у них чувства продвижения к цели.

- Отсутствие привязки к конкретной реализации C++. Программы были протестированы на всех доступных нам компиляторах (об этом далее), и если какая-либо реализация упорно отказывалась работать из-за неполного соответствия стандарту C++, мы помечали этот факт в примере (см. пометки в исходных текстах), чтобы исключить его из процесса построения.
- Автоматизированная компиляция и тестирование примеров.

Главы

Ниже приводится краткое описание глав книги.

Часть I. Разработка устойчивых систем

- **Глава 1. Обработка исключений.** Обработка ошибок вечно была источником проблем при программировании. Даже если ваша функция исправно возвращала код ошибки или устанавливала флаг, ничто не мешало вызывающей стороне проигнорировать ошибку. Механизм обработки исключений является одной из важнейших новых возможностей C++: при возникновении критической ошибки ваша функция «запускает» наружу объект исключения. Для разных ошибок запускаются разные типы объектов, и вызывающая сторона «перехватывает» эти объекты в разных обработчиках ошибок. Запущенное исключение невозможно проигнорировать, и вы можете быть уверены в том, что в случае ошибки *что-то* будет сделано. Наличие механизма обработки исключений оказывает положительное влияние на архитектуру программ.
- **Глава 2. Защитное программирование.** Многие ошибки можно предотвратить. Под «защитным программированием» понимается такая методология построения программ, которая обеспечивает обнаружение и исправление ошибок еще до причинения ими вреда. *Утверждения* являются самым полезным средством проверки кода на стадии разработки; кроме того, их можно рассматривать как дополнительную документацию, раскрывающую логику программиста во время написания программы. Перед распространением программа должна пройти жесткое тестирование. Система автоматизированного модульного тестирования оказывает незаменимую помощь в повседневной разработке программных продуктов.

Часть II. Стандартная библиотека C++

- **Глава 3. Строки.** Обработка текста относится к числу наиболее распространенных задач программирования. Строковые классы C++ избавляют программиста от хлопот с управлением памятью и в то же время предоставляют в его распоряжение мощные средства обработки текста. В C++ также поддерживаются расширенные кодировки и локальные контексты для создания интернационализированных приложений.

- **Глава 4. Потоки ввода-вывода.** Потоки ввода-вывода C++ создавались для замены библиотеки stdio.h языка C. Они проще в использовании, более гибки и лучше расширяются, в частности, вы можете адаптировать их для пользовательских классов. Из материала этой главы читатель узнает, как наиболее эффективно задействовать библиотеку потоков ввода-вывода для выполнения стандартного и файлового ввода-вывода, для форматирования данных в памяти.
- **Глава 5. Шаблоны.** Характерной особенностью «современного языка C++» является широкое применение шаблонов. Шаблоны не ограничиваются построением обобщенных контейнеров — с их помощью также создаются надежные, универсальные, высокопроизводительные библиотеки. Шаблоны образуют своего рода «субъязык» внутри языка C++ и расширяют степень контроля процесса компиляции программистом. Не будет преувеличением сказать, что шаблоны произвели настоящую революцию в области программирования на C++.
- **Глава 6. Обобщенные алгоритмы.** Алгоритмы занимают центральное место в обработке данных. Поддержка шаблонов в языке C++ предоставляет в распоряжение программиста впечатляющий арсенал мощных, эффективных и удобных обобщенных алгоритмов. Поведение стандартных алгоритмов также специализируется посредством объектов функций. В этой главе приводятся описания всех алгоритмов библиотеки (в главах 6 и 7 рассматривается часть стандартной библиотеки C++, известная как STL).
- **Глава 7. Обобщенные контейнеры.** В C++ поддерживаются все стандартные структуры данных, безопасные по отношению к типам. Программисту никогда не приходится беспокоиться о том, что именно хранится в таких контейнерах, — однородность элементов гарантирована. Итераторы позволяют отделить механизм перебора элементов от самого контейнера, это еще одно преимущество шаблонов. Подобная архитектура позволяет максимально просто и гибко применять алгоритмы к содержимому контейнеров.

Часть III. Специальные возможности

- **Глава 8. RTTI.** Механизм идентификации типов в процессе исполнения (RTTI) позволяет узнать фактический тип объекта по указателю или ссылке на базовый тип. Обычно в полиморфных иерархиях фактический тип объекта намеренно игнорируется, а правильное поведение для этого типа выбирается при помощи механизма виртуальных функций. Но в отдельных случаях (например, при написании специализированных отладочных программ) бывает полезно знать фактический тип объекта, поскольку при наличии такой информации некоторые операции выполняются более эффективно. В этой главе объясняется, что собой представляет механизм RTTI и как им пользоваться.
- **Глава 9. Множественное наследование.** На первый взгляд концепция выглядит просто: новый класс создается производным от нескольких существующих классов. Тем не менее множественное наследование ста-

новится причиной многих неоднозначных ситуаций и ведет к дублированию объектов базовых классов. Проблему удается решить при помощи виртуальных базовых классов, но остается другой, более серьезный вопрос: когда следует применять множественное наследование? Оно абсолютно необходимо только в том случае, если вам требуется работать с объектом через разные базовые классы. В этой главе объясняется синтаксис множественного наследования и продемонстрированы альтернативные решения (в частности, показано, как одна из типичных задач решается с использованием шаблонов). В качестве примера полезного применения множественного наследования рассматривается «исправление» интерфейса класса.

- **Глава 10. Паттерны проектирования.** Самым революционным достижением в программировании с момента изобретения объектов являются *паттерны проектирования*. Паттерн представляет собой формулировку решения стандартной задачи программирования, которая не зависит от языка и может применяться в разных контекстах. Такие паттерны, как Синглет, Фабричный метод и Посетитель, входят в повседневный арсенал программистов. В этой главе показано, как некоторые из наиболее полезных паттернов реализуются и используются в C++.
- **Глава 11. Многопоточное программирование.** Пользователи хотят, чтобы ваши программы обладали реактивным интерфейсом, который бы создавал иллюзию одновременного выполнения нескольких задач. Современные операционные системы позволяют запускать в процессах дополнительные программные потоки, совместно использующие адресное пространство процесса. Тем не менее, многопоточное программирование требует особого подхода и порождает специфические проблемы. Данная глава на примере свободно распространяемой библиотеки ZThreads[1] показывает, как организовать эффективное управление многопоточными приложениями в C++.

Упражнения

Мы убедились, что самостоятельная работа исключительно важна для полноценного понимания темы, поэтому каждая глава завершается набором упражнений.

Многие упражнения достаточно просты, чтобы их можно было выполнить за разумный промежуток времени в классе или лаборатории под наблюдением инструктора и убедиться в том, что все учащиеся усвоили материал. Некоторые упражнения имеют более высокую сложность и ориентируются на более опытных учащихся. И все же в большинстве случаев упражнения решаются быстро и предназначаются для проверки существующих познаний. Предполагается, что более сложные задачи вы найдете самостоятельно или, что более вероятно, они сами найдут вас.

---

[1] Разработчик — Эрик Крэхен (Eric Crahen) из IBM.

Исходные тексты

Исходные тексты программ, приводимых в книге, распространяются бесплатно, но с соблюдением авторских прав. В соответствии с авторскими правами запрещается воспроизведение кода в печатных изданиях без разрешения, но разрешается его использование во многих других ситуациях.

В каталоге, выбранном для распаковки архива, находится файл с текстом уведомления об авторских правах. Вы можете использовать эти программы в своих проектах и при проведении учебных занятий с соблюдением перечисленных условий.

Компиляторы

Возможно, ваш компилятор не поддерживает некоторые из возможностей, рассматриваемых в книге (особенно при использовании старой версии компилятора). Реализация такого языка, как C++, — титанический труд, поэтому последовательное введение поддержки тех или иных возможностей оправданно. Если при построении одного из примеров компилятор выдаст многочисленные сообщения об ошибках, это не обязательно свидетельствует об ошибке компилятора; возможно, соответствующие возможности просто не реализованы в вашем конкретном компиляторе.

Примеры, приводимые в книге, тестировались на разных компиляторах. Мы стремились убедиться в том, что наши программы соответствуют стандарту C++ и работают с максимально возможным количеством компиляторов. К сожалению, не все компиляторы соответствуют стандарту C++, поэтому некоторые файлы пришлось исключить из сборки для этих компиляторов. Эти исключения отражены в make-файлах, автоматически генерируемых для пакета программ. Соответствующие пометки есть в комментариях, находящихся в начале каждого листинга; это поможет вам определить, будет ли данная программа работать с конкретным компилятором (в некоторых случаях программа компилируется, но работает неверно; такие случаи тоже исключаются).

Далее перечислены пометки, исключающие компиляторы из сборки.

`{-dmc}`

Компилятор Digital Mars Уолтера Брайта (Walter Bright) для системы Windows. Бесплатно загружается с сайта wwwDigitalMars.com. Компилятор отличается высокой степенью соответствия стандарту, поэтому данная пометка встречается в книге очень редко.

`{-g++}`

Свободно распространяемый компилятор Gnu C++ 3.3.1, входящий в большинство пакетов Linux и Macintosh OSX. Также является частью Cygwin для Windows (см. далее). Версии для большинства других платформ загружаются с сайта www.gnu.org.

`{-msc}`

Компилятор Microsoft из пакета Visual C++ .NET (распространяется только в составе Visual Studio .NET).

{-bor}
Borland C++ версии 6 (достаточно современный компилятор, но бесплатно не распространяется).

{-edg}
EDG (Edison Design Group) C++. Настоящий паттерн соответствия стандарту. Пометка встречается только один раз из-за проблем, связанных с библиотекой, и то только потому, что мы используем интерфейс EDG с реализацией библиотеки от Dinkumware, Ltd. Само по себе применение этого компилятора не приводит к ошибкам компиляции.

{-mwcc}
Metrowerks Code Warrior для Macintosh OS X. Обратите внимание: в поставку OS X входит компилятор Gnu C++.

В архив примеров включены make-файлы для построения программ с помощью перечисленных компиляторов. Мы используем свободно распространяемую утилиту GNU make, которая входит в комплект поставки Linux и Cygwin (бесплатную оболочку Unix, работающую в системе Windows; см. www.cygwin.com), а также может устанавливаться на вашей платформе — см. www.gnu.org/software/make (другие варианты утилиты make могут оказаться совместимыми с этими файлами, но мы их не сопровождаем). После установки введите в командной строке команду make, и вы получите инструкции по установке примеров книги для этих компиляторов.

Учтите, что пометки отражают состояние версий компиляторов на момент написания книги. Вполне возможно, что за время, прошедшее с момента публикации, фирма-разработчик усовершенствует свой компилятор. Также нельзя исключать, что при таком большом количестве примеров мы использовали неправильную комбинацию настроек для конкретного компилятора, поэтому самый надежный путь — попробовать самостоятельно откомпилировать программу.

Языковые стандарты

Говоря о соответствии стандарту ANSI/ISO C, мы имеем в виду стандарт 1989 года; для краткости мы будем просто называть его «стандартом C». Различия между стандартом C и старыми версиями, предшествовавшими появлению стандарта, будут делаться только там, где это необходимо. Стандарт C99 в книге не рассматривается.

Комитет ANSI/ISO C++ давно завершил работу над первым стандартом C++, или сокращенно «C++98». В книге термин «стандарт C++» будет использоваться для обозначения этого стандартизированного языка. Если мы просто упоминаем о C++, это тоже означает соответствие стандарту. Комитет по стандартизации C++ продолжает решать проблемы, важные для сообщества C++. Когда-нибудь его работа воплотится в C++0x — будущий стандарт C++, который, вероятно, появится лишь через несколько лет.

Благодарности

Второй том книги несколько лет пребывал в незаконченном состоянии, пока Брюс занимался другими вещами — Java, паттернами и особенно Python (www.python.org). Если бы Чак (по своей глупости, как ему иногда казалось) не пожелал закончить свою половину и довести книгу до ума, скорее всего, она бы никогда не увидела свет. Найдется не так уж много людей, которым бы Брюс со спокойной совестью доверил судьбу книги. Стремление Чака к точным, правильным и понятным объяснениям сделало книгу такой, какая она есть.

Джейми Кинг (Jamie King) был ассистентом Чака в процессе завершения книги. Он внес немалый вклад в работу — и не только комментариями, но и прежде всего своими непрестанными вопросами и придирками к любой мелочи, которая вызывала хотя бы малейшие сомнения. Если вы найдете в этой книге ответ на свой вопрос, скорее всего, за это стоит благодарить Джейми, который уже успел этот вопрос задать. Джейми также усовершенствовал некоторые примеры программ и разработал многие упражнения, приведенные в конце каждой главы. Скотт Бейкер (Scott Baker), другой ассистент Чака, работа которого была оплачена MindView, Inc., участвовал в подготовке упражнений для главы 3.

Эрик Крэхен (Eric Crahen) из IBM помогал в написании главы 11. В поисках инструментария для многопоточного программирования мы искали пакет, который показался бы нам наиболее простым и интуитивно понятным, но при этом достаточно надежным. Благодаря Эрику мы нашли все это, и не только — он основательно помог нам и использовал наши комментарии для усовершенствования своей библиотеки, а мы, в свою очередь, тоже пользовались плодами его размышлений.

Мы благодарны нашему техническому редактору Питу Бейкеру (Pete Becker). Лишь немногие сравнятся с ним в проницательности и умении скрывать свои мысли, не говоря уже о глубочайших познаниях в C++ и программировании вообще. Мы также благодарны Бьёрну Карлссону (Bjorn Karlsson) за великодушную и своевременную техническую поддержку — он просмотрел всю рукопись за предельно короткий срок.

Уолтер Брайт (Walter Bright) приложил титанические усилия к тому, чтобы все примеры в книге компилировались его компилятором Digital Mars C++. Этот компилятор можно бесплатно загрузить с сайта http://www.DigitalMars.com. Спасибо, Уолтер!

Идеи и концепции этой книги берут свое начало во множестве других источников, в числе авторов которых мои друзья Андреа Провальо (Andrea Provaglio), Дэн Сакс (Dan Saks), Скотт Мейерс (Scott Meyers), Чарльз Петцольд (Charlez Petzold) и Майкл Уилк (Michael Wilk); пионеры языка Бьярн Страуструп (Bjarne Stroustrup), Эндрю Кениг (Andrew Koenig) и Роб Мюррей (Rob Murray); члены комитета по стандартизации C++ Натан Майерс (Nathan Myers), который особенно щедро поделился с нами своими мыслями о книге, Херб Саттер (Herb Sutter), Пи Джей Плаугер (PJ Plauger), Кевлин Хенни (Kevlin Henney), Дэвид Абрахамс (David Abrahams), Том Плам (Tom Plum), Рег Чарни (Reg Charney), Том Пенелло (Tom Penello), Сэм Друкер (Sam Druker), Уве Стейнмюллер (Uwe Steinmueller), Джон

Спайсер (John Spicer), Стив Адамчик (Steve Adamczyk) и Дэвид Вандевоорде (David Vandevoorde); люди, с которыми мы общались на секции C++ во время конференции по разработке программного обеспечения; коллеги Чака Майкл Сивер (Michael Seaver), Хастон Франклин (Huston Franklin), Дэвид Вагстафф (David Wagstaff) и участники семинаров, вопросы которых были нам так нужны, чтобы материал стал более понятным.

Мы также хотим поблагодарить щедрых профессионалов из Edison Design Group и Dinkumware, Ltd., предоставивших нам бесплатные экземпляры своего компилятора и библиотеки (соответственно). Без их любезной и компетентной помощи нам не удалось бы протестировать некоторые из примеров книги. Еще мы поблагодарим Ховарда Хиннанта (Howard Hinnant) и коллектив Metrowerks за предоставленный экземпляр компилятора, а Сэнди Смит (Sandy Smith) и коллектив SlickEdit — за поставку первоклассных средств редактирования в течение многих лет. Грег Комо (Greg Comeau) также предоставил копию своего успешного компилятора на базе EDG, Comeau C++.

Отдельное спасибо всем нашим учителям и всем ученикам (которые тоже были нашими учителями).

Эван Кофски (Evan Cofsky) оказывал всестороннюю помощь в работе с сервером, а также в программировании на его любимом языке Python. Шарлин Кобо (Sharlynn Cobaugh) и Пола Стейер (Paula Steuer) обеспечивали дополнительную поддержку и не позволили Брюсу утонуть в потоке проектов.

Дон МакГи (Dawn McGee) наполняла Брюса столь необходимым вдохновением и энтузиазмом. Ниже перечислены лишь некоторые из наших друзей: Марк Уэстерн (Mark Western), Ген Киёка (Gen Kiyooka), Крейг Брокшмидт (Kraig Brockschmidt), Зак Урлокер (Zack Urlocker), Эндрю Бинсток (Andrew Binstock), Нейл Рубенкинг (Neil Rubenking), Стив Синофски (Steve Sinofsky), Джей Ди Хильдебранд (JD Hildebrand), Брайан МакЭльхинни (Brian McElhinney), Бринкли Барр (Brinkley Barr), Билл Гейтс (Bill Gates) из «Midnight Engineering Magazine», Ларри Константайн (Larry Constantine) и Люси Локвуд (Lucy Lockwood), Том Кеффер (Tom Keffer), Грег Перри (Greg Perry), Дэн Путтерман (Dan Putterman), Кристи Вестфаль (Christi Westphal), Джин Ванг (Gene Wang), Дэйв Майер (Dave Mayer), Дэвид Интерсимон (David Intersimone), Клер Сойерс (Claire Sawyers), Андреа Провальо (Andrea Provaglio), Лора Фаллаи (Laura Fallai), Марко Канту (Marco Cantu), Коррадо (Corrado), Ильза и Кристина Густоцци (Ilsa & Christina Giustozzi), Крис и Лора Стрэнд (Chris & Laura Strand), Элмквисты (the Almquists), Брэд Джербик (Brad Jerbic), Джон Крут (John Kruth) и Мэрилин Цвитаник (Marylin Cvitanic), Холли Пэйн (Holly Payne) — да-да, та самая знаменитая писательница!, — Марк Мабри (Mark Mabry), семейства Роббинс (Robbins), Мёльтер (Moelter), Макмиллан (McMillan) и Уилкс (Wilks), Дэйв Стонер (Dave Stoner), Лори Адамс (Laurie Adams), Крэнстоны (Cranstons), Ларри Фогг (Larry Fogg), Майк и Карен Секейра (Mike & Karen Sequeira), Гэри Энтсмингер (Gary Entsminger) и Эллисон Броди (Allison Brody), Честер Андерсен (Chester Andersen), Джо Лорди (Joe Lordi), Дэйв и Бренда Бартлетт (Dave & Brenda Bartlett), Рентшлеры (Rentschlers), Судеки (Sudeks), Линн и Тодд (Lynn & Todd). И конечно, спасибо маме и папе, Сэнди, Джеймсу и Натали, Ким и Джареду, Айзеку и Эбби.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы сможете найти по адресу http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

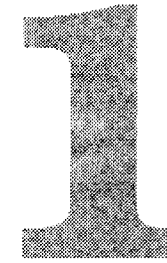
Часть

Разработка устойчивых систем

На проверку готового кода уходит почти столько же времени, сколько на его создание. Качество кода является (или должно являться) главной целью каждого программиста. Научившись устранять проблемы еще до их появления, вы продвинетесь далеко на пути к этой цели. Кроме того, программные системы должны быть достаточно устойчивыми, чтобы разумно реагировать на возникновение непредвиденных внешних проблем.

Исключения были добавлены в C++ как более совершенный механизм обработки ошибок, не загромождающий программу чрезмерными проверками. В главе 1 объясняется, как правильное применение исключений способствует повышению устойчивости кода, а также представляются архитектурные принципы построения программ, безопасных по отношению к исключениям. В главе 2 рассматриваются приемы модульного тестирования и отладки, повышающие качество кода еще до появления окончательной версии программы. В этом плане применение утверждений для выражения и проверки инвариантов программы является верным признаком опытного программиста. Также в главе 2 будет представлена простая система автоматизации модульного тестирования.

Обработка исключений



Одним из самых эффективных путей повышения надежности программ является усовершенствование механизма обработки ошибок.

К сожалению, на практике ошибки часто игнорируются, словно программисты сговорились не обращать на них внимания. Несомненно, это отчасти связано с тем, что проверка многочисленных условий ошибок — занятие скучное и малоинтересное, которое к тому же приводит к разрастанию программ. Например, функция printf() возвращает количество успешно выведенных символов, однако практически никто не проверяет результат ее вызова. Увеличение объема кода само по себе неприятно, не говоря уже о дополнительных сложностях с чтением программ.

У схемы обработки ошибок, использованной в C, имеется один серьезный недостаток — функция так тесно привязывается к коду обработки ошибок, что с ней становится неудобно работать.

К числу важнейших новшеств C++ принадлежит механизм *обработки исключений*. О нем можно сказать следующее.

- Обработка ошибок гораздо проще программируется, а ее код не смешивается с «обычным» кодом. Сначала вы программируете *нормальное* течение событий, а позднее в отдельной секции пишется код для решения проблем. При многократном вызове функции обработка ошибок этой функции производится только один раз и в одном месте.
- Ошибки могут игнорироваться. Если функция должна отправить сообщение об ошибке вызывающей стороне, она «запускает» объект, представляющий эту ошибку. Если вызывающая сторона не «перехватит» ошибку и не обработает ее, то ошибка переходит в следующую внешнюю динамическую область видимости, и т. д. В итоге либо ошибка будет перехвачена, либо программа завершится из-за отсутствия обработчика для данного типа исключения.

В этой главе мы рассмотрим принципы обработки ошибок в языке C и выясним, почему они недостаточно хорошо работают в C и вообще не подходят для C++. Также будут рассмотрены ключевые слова C++ try, throw и catch, используемые при обработке исключений.

Традиционная обработка ошибок

В большинстве приводимых примеров директива assert() используется именно для той цели, для которой она предназначена: для отладки в процессе разработки. В окончательной версии отладочный код обычно отключается директивой #define NDEBUG. Ошибки времени выполнения проверяются функциями из файла require.h (функции assert() и require(), созданные в главе 9 первого тома, приводятся повторно в приложении Б). Эти функции означают примерно следующее: «Имеется проблема, к которой следовало бы отнестись более внимательно, но мы не хотим отвлекаться на нее в данном примере». Для небольших программ функций из файла require.h может быть достаточно, но в более сложных продуктах имеет смысл использовать нетривиальную обработку ошибок.

Если вы точно знаете, что нужно делать, и у вас имеется вся необходимая информация, обработка ошибок выполняется достаточно элементарно. Проблемы начинаются тогда, когда вы *не обладаете* всей информацией в текущем контексте, и вам необходимо передать информацию об ошибке в другой контекст, где эта информация существует. В языке C возможны три варианта действий.

- Можно вернуть информацию об ошибке из функции, а если возвращаемое значение не может использоваться подобным образом, установить глобальный флаг ошибки. В стандартном языке C для этой цели предусмотрены конструкции errno и perror(). Как уже упоминалось, программисты часто игнорируют информацию об ошибках, потому что проверять все возможные ошибки после каждого вызова функции было бы слишком утомительно и неудобно. Кроме того, может оказаться, что возврат из функции, в которой возникла исключительная ситуация, не имеет смысла.
- Можно использовать малоизвестную систему сигналов из стандартной библиотеки C, реализованную функциями signal() (определение реакции на событие) и raise() (инициирование события). Этот вариант тоже требует жесткого связывания основного кода с кодом обработки ошибок, так как пользователь любой библиотеки, генерирующей сигналы, должен знать ее систему сигналов и установить соответствующие обработчики. Кроме того, в больших проектах возможны конфликты номеров сигналов, используемых разными библиотеками.
- Можно использовать *нелокальные версии команды перехода* в виде функций setjmp() и longjmp() из стандартной библиотеки C. Функция setjmp() сохраняет заведомо нормальное состояние в программе, которое при возникновении проблем восстанавливается функцией longjmp(). Но и в этом случае требуется жесткая привязка места сохранения состояния к месту возникновения ошибки.

Обсуждая схемы обработки ошибок в C++, необходимо учитывать еще одно важное обстоятельство: схемы C с сигналами и функциями setjmp()/longjmp() не

вызывают деструкторы, поэтому нормальная зачистка объектов не выполняется (более того, если функция longjmp() выходит за пределы области видимости, в которой должны вызываться деструкторы, это приведет к непредсказуемым последствиям). В результате восстановление после исключительных ситуаций становится практически невозможным, так как позади всегда остаются незачищенные объекты, ставшие недоступными. Следующий пример демонстрирует сказанное для функций setjmp/longjmp:

```
//: C01:Nonlocal.cpp
// setjmp() & longjmp()
#include <iostream>
#include <csetjmp>
using namespace std;

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

jmp_buf kansas;

void oz() {
  Rainbow rb;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home\n";
  longjmp(kansas, 47);
}

int main() {
  if(setjmp(kansas) == 0) {
    cout << "tornado, witch, munchkins...\n";
    oz();
  } else {
    cout << "Auntie Em! "
         << "I had the strangest dream..."
         << endl;
  }
} ///:~
```

Функция setjmp() вообще ведет себя странно. Если вызывать ее напрямую, она сохраняет всю информацию о текущем состоянии процессора (включая содержимое указателей команд и стека) в аргументе jmp_buf и возвращает ноль. В этом случае она ведет себя как обычная функция. Но если вызвать функцию longjmp() с тем же аргументом jmp_buf, все выглядит так, словно управление снова только что было возвращено из функции setjmp() — программа начинает выполняться с команды, следующей за вызовом setmp(). Однако на этот раз возвращаемое значение равно второму аргументу longjmp(), и это позволяет определить, что в действительности произошел возврат из longjmp(). Нетрудно представить, что при наличии нескольких заполненных буферов jmp_buf можно легко перемещаться между разными точками программы. Отличие локальной версии команды перехода по метке (goto) от нелокальной версии состоит в том, что функции setjmp()/longjmp() позволяют вернуться в любую заранее определенную позицию, расположенную выше в стеке (для которой была вызвана функция setjmp()).

Но в C++ возникает проблема: функция longjmp() не думает об объектах; в частности, она не вызывает деструкторы при выходе из области видимости[1]. Вызовы деструкторов абсолютно необходимы, поэтому такое решение не подходит для C++. Более того, в стандарте C++ сказано, что безусловный вход в область видимости по команде goto (в обход вызова конструкторов) или выход из области видимости с помощью функции longjmp() при наличии деструктора у объекта в стеке приводит к непредсказуемым последствиям.

Запуск исключений

Если в программе возникла исключительная ситуация (то есть в текущем контексте не хватает информации для принятия решения о том, как действовать дальше), информацию об ошибке можно передать во внешний, более общий контекст. Для этого в программе создается объект с информацией об исключении, который затем «запускается» из текущего контекста (говорят, что в программе *запускается исключение*). Вот как это выглядит:

```
//: C01:MyError.cpp
class MyError {
  const char* const data;
public:
  MyError(const char* const msg = 0) : data (msg) {}
};

void f() {
   // "Запускаем" объект исключения:
   throw MyError("something bad happened");
}

int main() {
   // Как вскоре будет показано,
   // здесь должен находиться "блок try":
  f();
} ///:~
```

MyError — обычный класс, конструктор которого в нашем примере получает тип char*. При запуске исключения можно использовать произвольный тип (в том числе и встроенные типы), но обычно для этой цели создаются специальные классы.

Ключевое слово throw производит целый ряд полузагадочных манипуляций. Сначала оно создает копию запускаемого объекта и фактически «возвращает» ее из функции, содержащей выражение throw, даже если тип этого объекта не соответствует типу, который положено возвращать этой функции. Обработку исключений можно упрощенно представить себе как альтернативный механизм возврата (но если зайти с этой аналогией слишком далеко, ничего хорошего не выйдет). Генерируя исключения, вы также можете выходить из обычных областей видимости. В любом случае возвращается значение, а управление из функции или области видимости передается наружу.

[1] Возможно, попытка выполнить приведенный пример вас удивит — некоторые компиляторы C++ поддерживают расширенную версию функции longjmp() с уничтожением объектов в стеке. Впрочем, такое поведение зависит от платформы и не является переносимым.

На этом все сходство с командой return завершается — точка, *в которую* происходит возврат, не имеет ничего общего с точкой возврата при обычном вызове функции (управление передается в специальную часть программы, называемую обработчиком исключения; она может находиться далеко от того места, где было запущено исключение). Также уничтожаются все локальные объекты, созданные к моменту запуска исключения. Автоматическое уничтожение локальных объектов часто называется «раскруткой стека».

Стоит сказать и о том, что в программе могут запускаться объекты исключений разнообразных типов. Как правило, для каждой категории ошибок используется свой тип. Предполагается, что информация будет передаваться как внутри объекта, так и в *имени* его класса; благодаря этому в контексте вызова можно будет решить, как поступить с исключением.

Перехват исключений

Как упоминалось ранее, одно из преимуществ механизма обработки исключений C++ состоит в том, что он позволяет программисту сосредоточиться на решаемой задаче, а затем организовать обработку ошибок в другом месте.

Блок try

Если внутри функции происходит исключение, выполнение этой функции завершается. Если вы не хотите, чтобы команда throw приводила к выходу из функции, создайте внутри функции специальный блок, который должен решать проблемы (а возможно — запускать новые исключения). Этот блок, называемый *блоком try*, представляет собой обычную область видимости, перед которой ставится ключевое слово try:

```
try {
  // Программный код, который может генерировать исключения
}
```

Если проверять ошибки по возвращаемому значению функций, вам придется заключать каждый вызов функции между кодом подготовки и кодом проверки даже при многократном вызове одной функции. При обработке исключений выполняемый код помещается в блок try, а обработка исключений производится после блока try. Это существенно упрощает написание и чтение программы, поскольку основной код не смешивается с кодом обработки ошибок.

Обработчики исключений

Конечно, программа должна где-то среагировать на запущенное исключение. Это место называется *обработчиком исключения*; в программу необходимо включить обработчик исключения для каждого типа перехватываемого исключения. Тем не менее полиморфизм распространяется и на исключения: один обработчик может перехватывать как определенный тип исключения, так и исключения классов, производных от этого типа.

Обработчики исключений следуют сразу же за блоком try и обозначаются ключевым словом catch:

```
try {
  // Программный код, который может генерировать исключения
} catch(type1 id1) {
  // Обработка исключений типа type1
} catch(type2 id2) {
  // Обработка исключений типа type2
} catch(type3 id2) {
  // И т. д.
} catch(typeN idN)
  // Обработка исключений типа typeN
}
// Здесь продолжается нормальное выполнение программы...
```

По своему синтаксису секции **catch** напоминают функции, вызываемые с одним аргументом. Идентификатор (**id1**, **id2** и т. д.) может использоваться внутри обработчика по аналогии с аргументом функции, но если он не нужен — не используйте его. Тип исключения обычно дает достаточно информации для его обработки.

Обработчики должны находиться сразу же после блока **try**. Если в программе запускается исключение, механизм обработки исключений начинает искать первый обработчик с аргументом, соответствующим типу исключения. Управление передается в найденную секцию **catch**, и исключение считается обработанным (то есть дальнейший поиск обработчиков прекращается). Выполняется только нужная секция **catch**, а выполнение программы продолжается, начиная с позиции, следующей за последним обработчиком для данного блока **try**.

Обратите внимание: в блоке **try** один тип исключения может генерироваться разными вызовами функций, но обработчик нужен только один.

Для демонстрации работы конструкции **try/catch** в следующей версии файла **Nonlocal.cpp** вызов **setjmp()** заменен блоком **try**, а вызов **longjmp()** — командой **throw**:

```
//: C01:Nonlocal2.cpp
// Демонстрация обработки исключений
#include <iostream>
using namespace std;

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

void oz() {
  Rainbow rb;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home\n";
  throw 47;
}

int main() {
  try {
    cout << "tornado, witch, munchkins...\n";
    oz();
  }
  catch (int) {
    cout << "Auntie Em! "
         << "I had the strangest dream..."
```

```
        << endl;
  }
} ///:~
```

При выполнении команды throw в функции oz() начинается перебор секций catch до тех пор, пока не будет обнаружена секция catch с параметром типа int. Тело этой секции catch продолжает выполнение программы. Важнейшее отличие этой версии от версии из файла Nonlocal.cpp состоит в том, что при выходе из функции oz() по команде throw вызывается деструктор объекта rb.

Завершение и продолжение

В теории обработки исключений существуют две основных модели: обработка с завершением программы и обработка с продолжением программы. В *модели с завершением программы* (поддерживаемой в C++) предполагается, что ошибка настолько серьезна, что автоматически продолжить программу с точки возникновения исключения нельзя. Другими словами, при запуске исключения предполагается, что исправить ситуацию уже невозможно, и возвращаться *нежелательно*.

Альтернативная модель обработки ошибок впервые появилась в языке PL/I в 1960-х годах[1]. Семантика этой модели предполагает, что обработчик исключения каким-то образом исправит ситуацию, после чего сбойный фрагмент кода будет автоматически выполнен заново; причем считается, что вторая попытка может оказаться успешной. Если вы захотите организовать подобную модель поведения в C++, вам придется явно передать управление к точке возникновения ошибки (обычно посредством вызова функции). Нередко блок try помещается в цикл while и выполняется до тех пор, пока результат не окажется удовлетворительным.

История показывает, что программисты, которые работали в операционных системах с поддержкой модели с продолжением, в конечном счете переходили на имитацию модели с завершением. Продолжение программы на первый взгляд выглядит привлекательно, но на практике оно не столь полезно. Возможно, одной из причин является удаленность обработчика от точки, в которой возникло исключение. Просто уйти в удаленный обработчик несложно, тогда как переход в него с последующим возвратом может вызвать концептуальные трудности в больших системах, в которых исключения могут генерироваться во множестве мест.

Поиск подходящего обработчика

Когда в программе генерируется исключение, система обработки исключений начинает просматривать «ближайшие» обработчики в порядке их следования в исходном коде. Если она обнаруживает совпадение, исключение считается обработанным, и поиск на этом прекращается.

Поиск подходящего обработчика не требует идеального соответствия между исключением и его обработчиком. Объект (или ссылка на объект) исключения производного класса считается подходящим для обработчика, работающего с базовым классом. (Однако если обработчик предназначен для объекта, а не для ссылки,

[1] В языке BASIC давно поддерживается ограниченная модель обработки исключений с продолжением программы (команда ON ERROR).

объект исключения «усекается» до базового типа при передаче обработчику. Усечение безвредно, но оно приводит к потере всей информации, специфической для производного типа.) По этой причине, а также чтобы предотвратить создание лишней копии объекта исключения, всегда лучше перехватывать исключения *по ссылке*, а не по значению[1]. При запуске указателя обработчик ищется по стандартным правилам преобразования указателей. Тем не менее, в процессе поиска автоматические преобразования одного типа исключения к другому типу не выполняются. Пример:

```
//: C01:Autoexcp.cpp
// Отсутствие преобразований при поиске обработчика
#include <iostream>
using namespace std;

class Except1 {};

class Except2 {
public:
  Except2(const Except1&) {}
};

void f() { throw Except1(); }

int main() {
  try { f();
  } catch (Except2&) {
    cout << "inside catch(Except2)" << endl;
  } catch (Except1&) {
    cout << "inside catch(Except1)" << endl;
  }
} ///:~
```

Хотя на первый взгляд может показаться, что первый обработчик может быть выбран в результате преобразования объекта `Except1` в `Except2` с использованием преобразующего конструктора, при обработке исключений система не выполняет такие преобразования, и в итоге будет выбран обработчик `Except1`.

Следующий пример показывает, как обработчик исключений базового класса перехватывает исключение производного класса:

```
//: C01:Basexcpt.cpp
// Иерархии исключений
#include <iostream>
using namespace std;

class X {
public:
  class Trouble {};
  class Small : public Trouble {};
  class Big : public Trouble {};
  void f() { throw Big(); }
};

int main() {
  X x;
```

[1] Более того, в обработчиках исключений практически всегда следует задавать объекты исключений по константной ссылке (модификация исключения с повторным запуском применяется редко). Тем не менее, мы не настаиваем на этом.

```
  try {
    x.f();
  } catch(X::Trouble&) {
    cout << "caught Trouble" << endl;
  // Скрывается предыдущим обработчиком:
  } catch(X::Small&) {
    cout << "caught Small Trouble" << endl;
  } catch(X::Big&) {
    cout << "caught Big Trouble" << endl;
  }
} ///:~
```

В данном примере механизм обработки исключений всегда будет сопоставлять объект Trouble (и *все, что является частным случаем* Trouble по правилам открытого наследования[1]) с первым обработчиком. Таким образом, второй и третий обработчики вообще никогда не вызываются, поскольку все исключения достаются первому обработчику. Логичнее начать с обработчиков производных типов и переместить обработчик базового типа в конец списка, чтобы перехватить все более общие случаи.

Обратите внимание: в этих примерах исключения перехватываются по ссылке, хотя для данной иерархии это несущественно — производные классы не содержат дополнительных членов, а идентификаторы аргументов в обработчиках все равно не используются. Обычно в обработчиках следует работать с аргументами по ссылке, а не по значению, чтобы избежать усечения информации.

Перехват любых исключений

Иногда требуется написать обработчик для перехвата *любых* типов исключений. Для этой цели используется специальный список аргументов в виде многоточия (...):

```
catch(...) {
  cout << "an exception was thrown" << endl;
}
```

Поскольку такой обработчик перехватывает все исключения, он размещается *в конце* списка обработчиков (иначе следующие за ним обработчики никогда не выполняются).

Универсальный обработчик не может иметь аргументов, поэтому в нем невозможно получить какую-либо информацию об исключении или его типе. Такие секции catch часто освобождают некие ресурсы и перезапускают исключение.

Перезапуск исключения

Перезапуск исключений обычно применяют для освобождения тех или иных ресурсов (скажем, закрытия сетевых подключений или освобождения памяти в куче — за подробностями обращайтесь к разделу «Управление ресурсами»

[1] Только *однозначно выбираемые* и *доступные* базовые классы могут перехватывать исключения производных классов. Это правило сводит к минимуму затраты времени выполнения, необходимые для проверки исключений. Помните, что исключения проверяются на стадии выполнения, а не на стадии компиляции, поэтому обширная информация, доступная при компиляции, во время обработки исключений недоступна.

этой главы). При возникновении исключения иногда бывает не важно, какая ошибка породила его — просто нужно закрыть подключение, открытое ранее, после чего передать обработку исключения в другой контекст, ближе к пользователю (то есть находящийся выше в цепочке вызовов). Конструкция catch(...) идеально подходит для таких случаев. Вы хотите перехватить *любые* исключения, освободить ресурс, а затем перезапустить исключение для последующей обработки. Исключения перезапускаются командой throw без аргумента внутри обработчика:

```
catch(...) {
  cout << "an exception was thrown" << endl;
  // Освобождение ресурсов
  throw;
}
```

Остальные секции catch того же блока try игнорируются — команда throw передает исключение обработчикам следующего контекста. Вся информация объекта исключения сохраняется, поэтому обработчики внешнего контекста, перехватывающие конкретные типы исключений, смогут извлечь любую информацию, содержащуюся в объекте.

Неперехваченные исключения

Как объяснялось в начале главы, обработка исключений лучше традиционной методики с возвратом кода ошибки, поскольку исключения не могут игнорироваться, а обработка исключения отделяется от непосредственно решаемой задачи. Если ни один из обработчиков, следующих за блоком try, не соответствует типу исключения, то исключение передается в контекст следующего уровня, то есть в функцию (или в блок try), в которой находится блок try, не перехвативший исключение, причем местонахождение этого внешнего блока try не всегда очевидно, поскольку он находится на более высоком уровне иерархии. Процесс продолжается до тех пор, пока в какой-то момент для исключения не будет найден подходящий обработчик. В этот момент исключение считается «перехваченным», и дальнейший поиск обработчиков для него не выполняется.

Функция terminate()

Если исключение не будет перехвачено ни одним обработчиком какого-либо уровня, автоматически вызывается стандартная библиотечная функция terminate(). По умолчанию terminate() вызывает функцию abort() из стандартной библиотеки C, что приводит к аварийному завершению программы. В системах семейства Unix функция abort() также выводит дамп памяти. Вызов abort() отменяет стандартную процедуру завершения программы, а это означает, что деструкторы глобальных и статических объектов не вызываются. Функция terminate() также выполняется в том случае, если деструктор локального объекта генерирует исключение в процессе раскрутки стека (во время обработки текущего исключения) или исключение происходит в конструкторе или деструкторе глобального или статического объекта (в общем случае не стоит разрешать запуск исключений в деструкторах).

Функция set_terminate()

Вы можете заменить стандартную функцию terminate() собственной версией при помощи стандартной функции set_terminate(), которая возвращает указатель на заменяемую функцию terminate() (при первом вызове это будет стандартная библиотечная версия) для ее последующего восстановления. Пользовательская функция terminate() должна вызываться без аргументов и возвращать void. Кроме того, устанавливаемые обработчики не могут возвращать управление или запускать исключения, они лишь реализуют некую логику завершения программы. Вызов функции terminate() означает, что проблема неразрешима.

Применение функции set_terminate() продемонстрировано в следующем примере. Сохранение и восстановление возвращаемого значения поможет локализовать фрагмент кода, в котором произошло неперехваченное исключение:

```
//: C01:Terminator.cpp
// Использование set_terminate().
// Программа также демонстрирует обработку неперехваченных исключений.
#include <exception>
#include <iostream>
using namespace std;

void terminator() {
  cout << "I'll be back!" << endl;
  exit(0);
}

void (*old_terminate)()
  = set_terminate(terminator);

class Botch {
public:
  class Fruit {};
  void f() {
    cout << "Botch::f()" << endl;
    throw Fruit();
  }
  ~Botch() { throw 'c'; }
};

int main() {
  try {
    Botch b;
    b.f();
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
} ///:~
```

Определение old_terminate на первый взгляд выглядит несколько странно: мы не только создаем указатель на функцию, но и инициализируем его возвращаемым значением set_terminate(). Хотя после указателя на функцию обычно следует точка с запятой, в действительности это самая обычная переменная, которая может инициализироваться при определении.

Класс Botch запускает исключение не только в f(), но и в деструкторе. Как нетрудно убедиться, это приводит к вызову terminate(). Хотя в обработчике

используется секция catch(...), которая вроде бы должна перехватить все исключения, не оставляя повода для вызова terminate(), функция terminate() все равно будет вызвана. В процессе уничтожения объектов в стеке при обработке первого исключения вызывается деструктор Botch, который генерирует второе исключение и становится причиной вызова terminate(). Следовательно, деструктор, который сам запускает исключения или приводит к их запуску, обычно является признаком плохого проектирования или небрежного программирования.

Зачистка

Одно из преимуществ обработки исключений состоит в том, что нормальный ход программы прерывается, и управление сразу передается в соответствующий обработчик исключений. Но эта передача принесет пользу только в том случае, если в момент запуска исключения будет проведена необходимая деинициализация. Механизм обработки исключений C++ гарантирует, что при выходе из области видимости для всех объектов этой области, *конструкторы которых завершены*, будут вызваны деструкторы.

Следующий пример убеждает в том, что для объектов с незавершенными конструкторами деструкторы не вызываются. Кроме того, он показывает, что происходит при запуске исключения в процессе создания массива объектов:

```
//: C01:Cleanup.cpp
// При запуске исключения уничтожаются только готовые объекты
#include <iostream>
using namespace std;

class Trace {
  static int counter;
  int objid;
public:
  Trace() {
    objid = counter++;
    cout << "constructing Trace #" << objid << endl;
    if(objid == 3) throw 3;
  }
  ~Trace() {
    cout << "destructing Trace #" << objid << endl;
  }
};

int Trace::counter = 0;

int main() {
  try {
    Trace n1;
    // Запуск исключения:
    Trace array[5];
    Trace n2;  // Сюда не попадаем
  } catch(int i) {
    cout << "caught " << i << endl;
  }
} ///:~
```

Класс Trace выводит информацию о создании и уничтожении своих объектов, что позволяет проследить за ходом выполнения программы. Класс подсчитывает созданные объекты в статической переменной counter, а идентификатор конкретного объекта хранится в переменной objid.

Функция main() создает один объект n1 (objid 0), а затем пытается создать массив из пяти объектов Trace, но создание четвертого объекта (идентификатор 3) прерывается. Таким образом, создание объекта n2 так и не завершается. Из выходных данных программы видно, как происходит вызов деструкторов:

```
constructing Trace #0
constructing Trace #1
constructing Trace #2
constructing Trace #3
destructing Trace #2
destructing Trace #1
destructing Trace #0
caught 3
```

Три элемента массива создаются успешно, но в процессе вызова конструктора четвертого элемента запускается исключение. Поскольку в main() конструирование четвертого объекта (array[2]) не было завершено, в программе вызываются деструкторы только для объектов array[1] и array[0]. В конце уничтожается объект n1, но не объект n2, который так и не был создан.

Управление ресурсами

Программируя обработку исключений, всегда следует задавать себе вопрос: «Если произойдет исключение, будут ли освобождены задействованные ресурсы?» Как правило, механизм освобождения ресурсов работает достаточно надежно, но существует специфическая проблема, связанная с конструкторами: если до завершения конструктора будет сгенерировано исключение, то для данного объекта деструктор не вызывается. Это означает, что при написании конструкторов необходимо быть особенно внимательным.

Проблема связана с выделением ресурсов в конструкторах. Если в конструкторе произойдет исключение, то деструктор не сможет освободить этот ресурс. Чаще всего эта проблема проявляется в виде «зависших» указателей. Пример:

```
//: C01:Rawp.cpp
// Зависшие указатели
#include <iostream>
#include <cstddef>
using namespace std;

class Cat {
public:
  Cat() { cout << "Cat()" << endl; }
  ~Cat() { cout << "~Cat()" << endl; }
};

class Dog {
public:
  void* operator new(size_t sz) {
    cout << "allocating a Dog" << endl;
    throw 47;
  }
  void operator delete(void* p) {
```

```
    cout << "deallocating a Dog" << endl;
    ::operator delete(p);
  }
};

class UseResources {
  Cat* bp;
  Dog* op;
public:
  UseResources(int count = 1) {
    cout << "UseResources()" << endl;
    bp = new Cat[count];
    op = new Dog;
  }
  ~UseResources() {
    cout << "~UseResources()" << endl;
    delete [] bp; // Уничтожение массива
    delete op;
  }
};

int main() {
  try {
    UseResources ur(3);
  } catch(int) {
    cout << "inside handler" << endl;
  }
} ///:~
```

Результат:

```
UseResources()
Cat()
Cat()
Cat()
allocating a Dog
inside handler
```

Программа входит в конструктор UseResources, и конструктор Cat успешно завершается для трех объектов массива. Однако при вызове Dog::operator new происходит исключение (имитация нехватки памяти). Внезапно управление передается в обработчик *без вызова* деструктора UseResources — и это логично, потому что конструктор UseResources не завершился. Но это также означает, что объекты Cat, успешно созданные в куче, тоже не будут уничтожены.

Управление ресурсами на уровне объектов

Чтобы предотвратить подобную утечку ресурсов, необходимо отказаться от «низкоуровневого» выделения ресурсов. Существуют два возможных способа:

- перехват исключений внутри конструктора с последующим освобождением ресурса;
- выделение ресурсов только в конструкторе объекта, чтобы освобождение ресурсов могло выполняться внутри деструктора.

Во втором варианте каждая операция выделения ресурсов становится атомарной вследствие того, что эти операции являются частью жизненного цикла локального объекта. Если выделение ресурса завершается неудачей, другие объекты вы-

деления ресурсов будут должным образом уничтожены в процессе раскрутки стека. Эта методика называется *получением ресурсов при инициализации* (Resource Acquisition Is Initialization, RAII), поскольку управление ресурсом (выделение и освобождение) совмещается с основными точками жизненного цикла объекта. Ниже показано, как эта задача решается за счет применения шаблонов в предыдущем примере:

```
//: C01:Wrapped.cpp
// Безопасные, атомарные указатели
#include <iostream>
#include <cstddef>
using namespace std;

// Упрощение. В вашем случае могут использоваться другие аргументы.
template<class T, int sz = 1> class PWrap {
  T* ptr;
public:
  class RangeError {}; // класс исключения
  PWrap() {
    ptr = new T[sz];
    cout << "PWrap constructor" << endl;
  }
  ~PWrap() {
    delete [] ptr;
    cout << "PWrap destructor" << endl;
  }
  T& operator[](int i) throw(RangeError) {
    if(i >= 0 && i < sz) return ptr[i];
    throw RangeError();
  }
};

class Cat {
public:
  Cat() { cout << "Cat()" << endl; }
  ~Cat() { cout << "~Cat()" << endl; }
  void g() {}
};

class Dog {
public:
  void* operator new[](size_t) {
    cout << "Allocating a Dog" << endl;
    throw 47;
  }
  void operator delete[](void* p) {
    cout << "Deallocating a Dog" << endl;
    ::operator delete[](p);
  }
};

class UseResources {
  PWrap<Cat, 3> cats;
  PWrap<Dog> dog;
public:
  UseResources() {
    cout << "UseResources()" << endl;
  }
  ~UseResources() {
```

```
    cout << "~UseResources()" << endl;
  }
  void f() { cats[1].g(); }
};

int main() {
  try {
    UseResources ur;
  } catch(int) {
    cout << "inside handler" << endl;
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
} ///:~
```

В новой версии шаблоны используются как оболочки для указателей и их инкапсуляции в объекты. Конструкторы этих объектов вызываются *до* основного кода конструктора *UseResources*, и *для любого конструктора, завершившегося прежде* возникновения исключения, в процессе раскрутки стека будет вызван соответствующий деструктор.

Шаблон PWrap демонстрирует более типичное применение исключений: в нем определяется вложенный класс RangeError[1], который используется в операторной функции operator[] при выходе аргумента из интервала допустимых значений. Поскольку функция operator[] возвращает ссылку, она не может вернуть ноль (нулевых ссылок не существует). Такая ситуация действительно является исключительной — в текущем контексте нельзя решить, что делать дальше, и возвращать недостоверное значение тоже нельзя. В приведенном примере используется простейший класс исключения RangeError. Предполагается, что вся необходимая информация содержится в имени класса, хотя при необходимости в него можно включить переменную со значением индекса.

На этот раз результат выглядит так:

```
Cat()
Cat()
Cat()
PWrap constructor
allocating a Dog
~Cat()
~Cat()
~Cat()
PWrap destructor
inside handler
```

И снова при выделении памяти для Dog происходит исключение. Однако на этот раз массив объектов Cat уничтожается так, как положено, и утечки памяти не возникает.

auto_ptr

В типичной программе C++ динамическая память является наиболее часто используемым ресурсом. По этой причине в стандарте предусмотрена RAII-оболочка для указателей на память в куче, обеспечивающая автоматическое освобождение памя-

[1] В стандартную библиотеку C++ входит класс исключения std::out_of_range, предназначенный именно для таких ситуаций.

ти. У шаблонного класса auto_ptr, определяемого в заголовочном файле <memory>, имеется конструктор, получающий указатель на тип параметра (то есть тип, непосредственно используемый в программе). Шаблон auto_ptr также перегружает операторы * и -> и выполняет соответствующие операции с исходным указателем, инкапсулированным в объекте auto_ptr. Таким образом, с объектом auto_ptr можно работать так, как если бы он был обычным указателем. Вот как это делается:

```
//: C01:Auto_ptr.cpp
// Демонстрация RAII-природы класса auto_ptr
#include <memory>
#include <iostream>
#include <cstddef>
using namespace std;
class TraceHeap {
  int i;
public:
  static void* operator new(size_t siz) {
    void* p = ::operator new(siz);
    cout << "Allocating TraceHeap object on the heap "
         << "at address " << p << endl;
    return p;
  }
  static void operator delete(void* p) {
    cout << "Deleting TraceHeap object at address "
         << p << endl;
    ::operator delete(p);
  }
  TraceHeap(int i) : i(i) {}
  int getVal() const {
    return i;
  }
};

int main() {
  auto_ptr<TraceHeap> pMyObject(new TraceHeap(5));
  cout << pMyObject->getVal() << endl;  // Выводит 5
} ///:~
```

Класс TraceHeap перегружает операторы new и delete, чтобы они выводили полную информацию о происходящих событиях. Как и в любом другом шаблоне, фактический тип указывается в параметре шаблона.

Однако мы не используем запись TraceHeap* — объект auto_ptr уже знает, что в нем будет храниться указатель на ваш тип. Вторая строка main() позволяет убедиться в том, что в функции operator->() класса auto_ptr происходит косвенное обращение к исходному низкоуровневому указателю. Еще важнее другое: хотя исходный указатель не удаляется в программе, деструктор pMyObject удаляет его в процессе раскрутки стека, как видно из следующего вывода:

```
Allocating TraceHeap object on the heap at address 8930040
5
Deleting Traceheap object at address 8930040
```

Шаблон auto_ptr также удобен при работе с указателями на переменные классов. Поскольку объекты классов, хранимые по значению, всегда уничтожаются, переменные типа auto_ptr при уничтожении внешнего объекта всегда удаляют объект, связанный с хранящимся в них низкоуровневым указателем.

Блоки try уровня функций

Исключения при выполнении конструкторов происходят достаточно часто. Допустим, вы хотите обработать исключения, происходящие при инициализации вложенных объектов или подобъектов базовых классов. Для этого инициализация таких подобъектов заключается в *блок try уровня функции*. В отличие от обычного синтаксиса, блоком try для инициализаторов конструкторов является само тело конструктора, а блок catch следует за телом конструктора, как в следующем примере.

```
//: C01:InitExcept.cpp {-bor}
// Обработка исключений в подобъектах
#include <iostream>
using namespace std;

class Base {
  int i;
public:
  class BaseExcept {};
  Base(int i) : i(i) { throw BaseExcept(); }
};

class Derived : public Base {
public:
  class DerivedExcept {
    const char* msg;
  public:
    DerivedExcept(const char* msg) : msg(msg) {}
    const char* what() const { return msg; }
  };
  Derived(int j)
  try
    : Base(j) {
    // Тело конструктора
    cout << "This won't print" << endl;
  }
  catch (BaseExcept&) {
    throw DerivedExcept("Base subobject threw");;
  }
};

int main() {
  try {
    Derived d(3);
  }
  catch (Derived::DerivedExcept& d) {
    cout << d.what() << endl;  // "Base subobject threw"
  }
} ///:~
```

Обратите внимание: список инициализаторов в конструкторе Derived следует после ключевого слова try, но перед телом конструктора. Возникшее исключение означает, что хранящийся объект не сконструирован, поэтому возвращаться к тому коду, где он создавался, бессмысленно. По этой причине возможен только один разумный вариант: запустить исключение в секции catch уровня функции.

Как показывает следующий пример, в C++ также допускается создание блоков try уровня функции для *любых* функций (впрочем, особенно полезной эту возможность не назовешь):

```
//: C01:FunctionTryBlock.cpp {-bor}
// Блоки try уровня функций
// {RunByHand} (Не запускать автоматически)
#include <iostream>
using namespace std;

int main() try {
  throw "main";
} catch(const char* msg) {
  cout << msg << endl;
  return 1;
} ///:~
```

В этом случае блок catch возвращает управление по аналогии с тем, как оно обычно возвращается из тела функции. Данная разновидность блоков try уровня функции практически не отличается от простого заключения кода функции в конструкцию try/catch.

Стандартные исключения

Исключения стандартной библиотеки C++ также могут использоваться в ваших программах. Как правило, проще и удобнее начать со стандартного класса исключения, чем пытаться определять собственный класс. Даже если стандартный класс делает не совсем то, что нужно, на его основе всегда можно создать производный класс.

Все стандартные классы исключений в конечном счете являются производными от общего предка exception, определенного в заголовке <exception>. Два основных производных класса logic_error и runtime_error определяются в заголовке <stdexcept> (который включает <exception>). Класс logic_error представляет ошибки в логике программирования — например, передачу недопустимых аргументов. К категории runtime_error относятся те ошибки, которые возникают в результате непредвиденных обстоятельств (скажем, сбоев оборудования или нехватки памяти). В обоих классах, runtime_error и logic_error, определен конструктор с аргументом std::string. Он позволяет сохранить сообщение в объекте исключения и извлечь его позднее с помощью функции exception::what(), как это сделано в следующем примере:

```
//: C01:StdExcept.cpp
// Создание класса исключения, производного от std::runtime_error
#include <stdexcept>
#include <iostream>
using namespace std;

class MyError : public runtime_error {
public:
  MyError(const string& msg = "") : runtime_error(msg) {}
};

int main() {
  try {
    throw MyError("my message");
  }
```

```
  catch (MyError& x) {
    cout << x.what() << endl;
  }
} ///:~
```

Хотя конструктор runtime_error сохраняет сообщение в подобъекте std::exception, класс std::exception не содержит конструктора с аргументом std::string. Обычно классы исключений объявляются производными не от std::exception, а от runtime_error или logic_error (или одного из их потомков).

Ниже кратко описаны основные классы исключений.

`exception`

Базовый класс для всех исключений, генерируемых стандартной библиотекой C++. Функция what() возвращает необязательную строку, указанную при инициализации исключения.

`logic_error`

Класс, производный от exception. Передает информацию о логических ошибках в программе. Предполагается, что такие ошибки обнаруживаются путем анализа некоторых условий.

`runtime_error`

Класс, производный от exception. Передает информацию об ошибках времени выполнения. Предполагается, что такие ошибки обнаруживаются только во время выполнения программы.

Класс исключений потоков ввода-вывода ios::failure тоже является производным от exception, но других подклассов не имеет.

Следующие классы (производные от logic_error и от runtime_error) можно использовать «как есть» или же создать на их основе производные классы для более специализированных типов исключений.

Далее перечислены классы исключений, производные от logic_error.

`domain_error`

Сообщает о нарушении предусловия.

`invalid_argument`

Указывает, что функции, запустившей исключение, был передан недопустимый аргумент.

`length_error`

Сообщает о попытке создания объекта, размер которого больше или равен npos (размер наибольшего представимого значения контекстного типа, обычно std::size_t).

`out_of_range`

Сообщает о выходе аргумента из интервала допустимых значений.

`bad_cast`

Выдается при выполнении недопустимых выражений dynamic_cast в подсистеме RTTI (см. главу 8).

`bad_typeid`

Сообщает об использовании null-указателя p в выражении typeid(*p). Также используется подсистемой RTTI (см. главу 8).

А здесь перечислены классы исключений, производные от runtime_error.

`runtime_error`

Сообщает о нарушении постусловия.

`overflow_error`

Сообщает о возникновении математического переполнения.

`bad_alloc`

Сообщает о неудачной попытке выделения памяти.

Спецификации исключений

Вообще говоря, вы не обязаны сообщать пользователям вашей функции, какие исключения она может запускать. Однако такое поведение считается нецивилизованным — оно означает, что пользователи не будут знать, как написать код перехвата потенциальных исключений. При наличии исходных текстов они смогут просмотреть их и поискать команды throw, однако библиотеки не всегда поставляются с исходными текстами. Хорошая документация поможет решить проблемы, но много ли найдется хорошо документированных программных проектов? Специальный синтаксис C++ позволяет сообщить пользователю, какие исключения запускаются данной функцией, чтобы он мог обработать их. Речь идет о необязательной *спецификации исключений*, указываемой в объявлении функции после списка аргументов.

Спецификация исключений состоит из ключевого слова throw, за которым в круглых скобках перечисляются типы всех потенциальных исключений, которые могут запускаться данной функцией. Объявление функции может выглядеть примерно так:

```
void f() throw(toobig, toosmall, divzero);
```

В отличие от этого объявления традиционное объявление функции означает, что функция может запускать исключения *любых* типов:

```
void f();
```

Однако следующая конструкция говорит о том, что функция не может запускать *никаких* исключений (проследите за тем, чтобы функции, находящиеся на очередном уровне в цепочке вызовов, не передавали исключения наверх!):

```
void f() throw();
```

Если хороший стиль программирования, полнота документации и удобства работы с функцией вам не безразличны, обязательно включайте спецификации исключений в те функции, которые их запускают (впрочем, эта рекомендация будет обсуждаться далее в этой главе).

Функция unexpected()

Итак, в спецификации перечисляются исключения, которые могут запускаться функцией. Но что произойдет, если функция запустит исключение, отсутствующее в списке? В этом случае вызывается специальная функция unexpected(), которая по умолчанию вызывает функцию terminate(), упоминавшуюся ранее.

Функция set_unexpected()

По аналогии с terminate() механизм вызова unexpected() позволяет вам назначить собственную функцию для обработки непредвиденных исключений. Задача решается при помощи функции set_unexpected(). Этой функции (как и set_terminate()) передается адрес функции без аргументов, возвращающей void. Функция set_unexpected() возвращает предыдущее значение указателя unexpected() для последующего восстановления. Чтобы использовать функцию set_unexpected(), необходимо включить в программу заголовочный файл <exception>. Простейшее применение функции set_unexpected() продемонстрировано в следующем примере:

```
//: C01:Unexpected.cpp
// Спецификации исключений и unexpected()
//{-msc} (ненормальное завершение)
#include <exception>
#include <iostream>
using namespace std;

class Up {};
class Fit {};
void g();

void f(int i) throw (Up, Fit) {
  switch(i) {
    case 1: throw Up();
    case 2: throw Fit();
  }
  g();
}

// void g() {} // Версия 1
void g() { throw 47; } // Версия 2

void my_unexpected() {
  cout << "unexpected exception thrown" << endl;
  exit(0);
}

int main() {
  set_unexpected(my_unexpected);
  // (возвращаемое значение игнорируется)
  for(int i = 1; i <=3; i++)
    try {
      f(i);
    } catch(Up) {
      cout << "Up caught" << endl;
    } catch(Fit) {
      cout << "Fit caught" << endl;
    }
} ///:~
```

Классы Up и Fit предназначены только для запуска в качестве исключений. Обычно классы исключений имеют небольшие размеры, но конечно, они могут содержать дополнительную информацию, которую могут запрашивать обработчики исключений.

Функция f() в своей спецификации исключений обещает запускать исключения только типов Up и Fit. Судя по определению функции, это действительно так —

первая версия g(), вызываемая из f(), не запускает никаких исключений, поэтому обещание выполняется. Но если кто-нибудь изменит функцию g() так, что она будет запускать исключения другого типа (например, вторая версия в нашем примере запускает исключение int), спецификация исключений f() будет нарушена.

В соответствии с критериями пользовательских функций unexpected(), функция my_unexpected() не имеет аргументов и возвращаемого значения. Она просто выводит сообщение о своем вызове, а затем завершает программу (команда exit(0) используется для того, чтобы при построении примеров книги процесс make не завершался аварийно). Новая функция unexpected() не может содержать команды return.

В функции main() блок try заключен в цикл for, чтобы продемонстрировать все возможные варианты обработки исключений. Так обеспечивается некое подобие восстановления: блок try выполняется в цикле for, while или do, вы перехватываете все исключения и пробуете решить проблему, а затем снова пытаетесь выполнить блок try.

Перехватываются только исключения Up и Fit, потому что создатель f() обещает, что функция запускает только эти исключения. Вторая версия g() приводит к вызову my_unexpected(), так как f() в этом случае запускает исключение типа int.

Хотя значение, возвращаемое при вызове set_unexpected(), проигнорировано, его также можно сохранить в указателе на функцию и восстановить позднее, как это было сделано в примере set_terminate() ранее в этой главе.

Типичный обработчик unexpected сохраняет информацию об ошибке и завершает программу вызовом exit(). Впрочем, он также может запустить другое (или перезапустить текущее) исключение, либо вызвать функцию abort(). Если обработчик запустит исключение типа, входящего в спецификацию, то дальнейший поиск продолжается с точки *вызова* функции с данной спецификацией исключений (такое поведение характерно только для unexpected()).

Если исключение, запущенное из обработчика unexpected, недопустимо по исходной спецификации, возможен один из двух вариантов действий.

- Если в спецификации исключений функции присутствует объект std::bad_exception, то исключение, запущенное из обработчика unexpected, заменяется объектом std::bad_exception, после чего поиск продолжается.
- Если в исходной спецификации нет объекта std::bad_exception, вызывается функция terminate().

Следующая программа поясняет сказанное:

```
//: C01:BadException.cpp {-bor}
#include <exception>    // Для std::bad_exception
#include <iostream>
#include <cstdio>
using namespace std;

// Классы исключений:
class A {};
class B {};

// Обработчик terminate()
void my_thandler() {
  cout << "terminate called\n";
  exit(0);
```

```
}

// Обработчики unexpected()
void my_uhandler1() {
  throw A();
}
void my_uhandler2() {
  throw;
}

// Если включить эту команду throw в f или g,
// компилятор распознает нарушение и сообщит об ошибке,
// поэтому мы выделяем ее в отдельную функцию.
void t() {
  throw B();
}

void f() throw(A) {
  t();
}
void g() throw(A, bad_exception) {
  t();
}

int main() {
  set_terminate(my_thandler);
  set_unexpected(my_uhandler1);
  try {
    f();
  }
  catch (A&) {
    cout << "caught an A from f\n";
  }
  set_unexpected(my_uhandler2);
  try {
    g();
  }
  catch (bad_exception&) {
    cout << "caught a bad_exception from g\n";
  }
  try {
    f();
  }
  catch (...) {
    cout << "This will never print\n";
  }
} ///:~
```

Обработчик my_uhandler() запускает допустимое исключение (A), поэтому выполнение успешно продолжается в первой секции catch. Обработчик my_uhandler2() запускает недопустимое исключение (B), но в спецификации g() присутствует класс bad_exception, поэтому исключение B заменяется объектом bad_exception, и выполнение успешно продолжается во второй секции catch. Поскольку спецификация f() не содержит объекта bad_exception, функция my_thandler() вызывается как обработчик завершения. Результат выполнения программы выглядит так:

```
caught an A from f
caught a bad_exception from g
terminate called
```

Улучшим спецификации исключений?

На первый взгляд может показаться, что существующий синтаксис спецификаций исключений не очень надежен, и что следующая запись *должна* означать, что функция не запускает никаких исключений:

```
void f();
```

Если программист хочет указать, что функция может запускать произвольные исключения, ему *следовало бы* записать это объявление так:

```
void f() throw(...);   // Только не в C++
```

Наверное, такой синтаксис был бы более логичным, поскольку объявления функции стали бы более однозначными. К сожалению, при взгляде на код функции не всегда понятно, запускает ли она исключения, например, исключение может возникнуть из-за нехватки памяти. Что еще хуже, существующие функции, написанные до появления в языке обработки исключений, могут непреднамеренно запускать исключения из-за вызываемых ими функций (при их включении в новые версии программ с обработкой исключений). Из-за этого следующая неинформативная запись означает: «Возможно, я буду запускать исключения... а может, и нет»:

```
void f();
```

Подобная неоднозначность неизбежна, чтобы не препятствовать эволюции программ. Если вы хотите указать, что функция f() не запускает исключения, задайте спецификацию с пустым списком:

```
void f() throw();
```

Спецификации исключений и наследование

Каждая открытая функция класса участвует в формировании контракта с пользователем; если передать ей некоторые аргументы, функция выполнит те или иные операции и/или вернет результат. Контракт должен соблюдаться и в производных классах, в противном случае нарушится основное правило «производный класс является частным случаем базового класса». Поскольку спецификации исключений являются логической частью объявления функции, они тоже должны сохраняться в иерархиях наследования. Например, если функция базового класса объявляет, что она запускает только исключения типа A, переопределение этой функции в производном классе не должно добавлять новые типы исключений в спецификацию, поскольку это нарушит работу любых программ, зависящих от интерфейса базового класса. С другой стороны, количество запускаемых исключений можно *уменьшить* и даже вовсе *запретить* их, так как это не приведет к расширению интерфейса базового класса. В спецификации производной функции вместо A также можно использовать «тип, производный от A». Рассмотрим пример:

```
// C01:Covariance.cpp {-xo}
// Должна происходить ошибка компиляции. {-mwcc}{-msc}
#include <iostream>
using namespace std;

class Base {
public:
  class BaseException {};
```

```
  class DerivedException : public BaseException {};
  virtual void f() throw (DerivedException) {
    throw DerivedException();
  }
  virtual void g() throw (BaseException) {
    throw BaseException();
  }
};

class Derived : public Base {
public:
  void f() throw (BaseException) {
    throw BaseException();
  }
  virtual void g() throw (DerivedException) {
    throw DerivedException();
  }
}; ///:~
```

Компилятор должен отвергнуть переопределение Derived::f() как ошибочное (или по крайней мере выдать предупреждение), поскольку оно изменяет спецификацию исключений Base::f(). Спецификация Derived::g() допустима, потому что DerivedException является частным случаем BaseException (а не наоборот). Base/Derived и BaseException/DerivedException можно рассматривать как параллельные иерархии классов; в контексте Derived ссылки на BaseException в спецификациях исключений и возвращаемых значениях могут заменяться на DerivedException. Такое поведение называется *ковариантным* (оба набора классов одновременно изменяются при перемещении по соответствующим иерархиям). Как было показано в первом томе, типы параметров *не ковариантны* — вы не можете изменять сигнатуры переопределенных виртуальных функций.

Когда спецификации исключений не используются

Просматривая объявления функций в стандартной библиотеке C++, вы не обнаружите в ней ни одной спецификации исключений! Хотя это может показаться странным, на самом деле такая внешняя небрежность объясняется вескими причинами: библиотека состоит в основном из шаблонов, а ее автору неизвестно, как будут работать конкретные типы или функции. Допустим, вы создаете обобщенный шаблон стека и пытаетесь включить в функцию pop() спецификацию исключений:

```
T pop() throw(logic_error);
```

Единственная потенциальная ошибка, которую можно себе представить, — попытка извлечения элемента из пустого стека, поэтому кажется, что вполне безопасно задать исключение logic_error или другого подходящего типа. Но копирующий конструктор типа T тоже может запустить исключение! В этом случае будет вызвана функция unexpected(), а программа завершится. Не стоит давать обещания, которые невозможно выполнить. Если вы не знаете, какие исключения могут возникнуть в программе, не используйте спецификации исключений. Вот почему в шаблонах, составляющих основную часть стандартной библиотеки C++, отсутствуют спецификации исключений — известные им исключения указываются *в документации*, а остальное остается на ваше усмотрение. Спецификации исключений требуются главным образом в нешаблонных классах.

Безопасность исключений

В главе 7 мы подробно рассмотрим контейнеры стандартной библиотеки, в том числе контейнер stack. В частности, вы увидите, что объявление функции pop() выглядит так:

```
void pop();
```

Может показаться странным, что функция pop() не возвращает значение, а просто выталкивает верхний элемент из стека. Чтобы получить значение этого элемента, приходится вызывать top() перед pop(). Однако такое поведение объясняется вескими причинами, связанными с *безопасностью исключений* — одним из критически важных факторов при проектировании библиотек. Существует несколько уровней безопасности исключений, но важнее другое: безопасность исключений означает правильную семантику при возникновении исключений.

Предположим, вы реализуете стек на базе динамического массива (назовем его data, а счетчик элементов — count), и пытаетесь написать функцию pop() так, чтобы она возвращала значение. Код такой функции pop() будет выглядеть примерно так:

```
template<class T> T stack<T>::pop() {
  if(count == 0)
    throw logic_error("stack underflow");
  else
    return data[--count];
}
```

Что произойдет, если копирующий конструктор, вызываемый для возвращаемого значения в последней строке, запустит исключение? Извлеченный элемент из-за исключения не возвращается, но счетчик count уже уменьшился, поэтому верхний элемент теряется навсегда! Проблема заключается в том, что функция пытается одновременно, во-первых, вернуть значение, во-вторых, изменить состояние стека. Лучше разделить эти операции на две разные функции класса, что и делается в стандартном классе stack (другими словами, соблюдается принцип *связности* — каждая функция решает одну четко сформулированную задачу). Код, безопасный по отношению к исключениям, оставляет объекты в логически целостном состоянии и не приводит к утечке ресурсов.

Осторожность также потребуется при написании пользовательских операторов присваивания. В главе 12 первого тома было показано, что оператор = должен работать по следующей схеме:

1. Убедиться в том, что объект не присваивается сам себе. Если это происходит, перейти к шагу 6 (проверка выполняется исключительно с целью оптимизации).
2. Выделить новую память для переменных-указателей.
3. Скопировать данные из старой памяти в новую.
4. Освободить старую память.
5. Обновить состояние объекта, присвоив переменным-указателям новые указатели на блоки, выделенные из кучи.
6. Вернуть *this.

Важно, чтобы состояние объекта не изменялось до того момента, когда все компоненты будут успешно созданы и инициализированы. Шаги 2 и 3 обычно оформляются в виде отдельной функции, которая часто называется clone(). В следующем примере это делается для класса, содержащего две переменные-указателя: theString и theInts:

```
//: C01:SafeAssign.cpp
// Оператор =, безопасный по отношению к исключениям
#include <iostream>
#include <new>       // Для std::bad_alloc
#include <cstring>
using namespace std;

// Класс с двумя переменными, содержащими указатели на память в куче
class HasPointers {
  // Класс Handle для хранения данных
  struct MyData {
    const char* theString;
    const int* theInts;
    size_t numInts;
    MyData(const char* pString, const int* pInts,
           size_t nInts)
    : theString(pString), theInts(pInts),
    numInts(nInts) {}
  } *theData;  // Манипулятор
  // Функции clone и cleanup
  static MyData* clone(const char* otherString,
      const int* otherInts, size_t nInts){
    char* newChars = new char[strlen(otherString)+1];
    int* newInts;
    try {
      newInts = new int[nInts];
    } catch (bad_alloc&) {
      delete [] newChars;
      throw;
    }
    try {
      // В данном примере используются встроенные типы, поэтому
      // исключения не генерируются. Однако при использовании
      // классов исключения возможны, поэтому блок try используется
      // для демонстрационных целей (для чего и нужен пример!)
      strcpy(newChars, otherString);
      for (size_t i = 0; i < nInts; ++i)
        newInts[i] = otherInts[i];
    } catch (...) {
      delete [] newInts;
      delete [] newChars;
      throw;
    }
    return new MyData(newChars, newInts, nInts);
  }
  static MyData* clone(const MyData* otherData) {
    return clone(otherData->theString,
                 otherData->theInts,
                 otherData->numInts);
  }
  static void cleanup(const MyData* theData) {
```

```
    delete [] theData->theString;
    delete [] theData->theInts;
    delete theData;
  }
public:
  HasPointers(const char* someString, const int* someInts,
              size_t numInts) {
    theData = clone(someString, someInts, numInts);
  }
  HasPointers(const HasPointers& source) {
    theData = clone(source.theData);
  }
  HasPointers& operator=(const HasPointers& rhs) {
    if (this != &rhs) {
      MyData* newData =
      clone(rhs.theData->theString,
            rhs.theData->theInts,
            rhs.theData->numInts);
      cleanup(theData);
      theData = newData;
    }
    return *this;
  }
  ~HasPointers() {
    cleanup(theData);
  }
  friend ostream& operator<<(ostream& os,
                             const HasPointers& obj) {
    os << obj.theData->theString << ": ";
    for (size_t i = 0; i < obj.theData->numInts; ++i)
      os << obj.theData->theInts[i] << ' ';
    return os;
  }
};

int main() {
  int someNums[] = {1, 2, 3, 4};
  size_t someCount = sizeof someNums / sizeof someNums[0];
  int someMoreNums[] = {5, 6, 7};
  size_t someMoreCount =
  sizeof someMoreNums / sizeof someMoreNums[0];
  HasPointers h1("Hello", someNums, someCount);
  HasPointers h2("Goodbye", someMoreNums, someMoreCount);
  cout << h1 << endl;  // Hello: 1 2 3 4
  h1 = h2;
  cout << h1 << endl;  // Goodbye: 5 6 7
} ///:~
```

Для удобства HasPointers использует класс MyData как манипулятор для работы с указателями. Когда требуется выделить дополнительную память (в результате конструирования или присваивания), в конечном счете для решения этой задачи вызывается первая функция clone. Если первый вызов оператора new завершается неудачей, автоматически генерируется исключение bad_alloc. Если неудача происходит при втором выделении памяти (для theInts), память theString необходимо освободить — для этого и нужен блок try, перехватывающий исключение bad_alloc. Второй блок try в данном случае не принципиален, поскольку копируются только числа int и указатели (так что исключений не будет), но при любом копировании

объектов их операторы присваивания могут породить исключение, которое потребует освобождения выделенных ресурсов. Обратите внимание: в обоих обработчиках мы *перезапускаем* исключение. Это объясняется тем, что наши обработчики всего лишь выполняют необходимые операции по управлению ресурсами; пользователь все равно должен узнать о возникших проблемах, поэтому исключение передается дальше по динамической цепочке. Библиотеки, которые не ограничиваются молчаливым «поглощением» исключений, называются *нейтральными по отношению к исключениям*. Всегда стремитесь к тому, чтобы ваши библиотеки были как безопасными, так и нейтральными по отношению к исключениям.

Внимательно просматривая ранее приведенную программу, можно заметить, что ни одна из операций delete не запускает исключения. От этого факта зависит работа программы. Вспомните: при вызове delete для объекта вызывается деструктор этого объекта. Оказывается, написать код, безопасный по отношению к исключениям, в принципе невозможно без предположения о невозможности исключений в деструкторах. Не позволяйте деструкторам запускать исключения! (Мы еще раз напомним об этом в конце главы[1].)

Программирование с учетом исключений

Для большинства программистов (а особенно программистов C) исключения являются новшеством, к которому придется привыкать. Далее приводятся рекомендации по программированию с учетом исключений.

Когда лучше обойтись без исключений

Исключения — не панацея; не злоупотребляйте ими. В этом разделе рассматриваются ситуации, в которых применять исключения *не рекомендуется*.

Принимая решение об использовании исключений, лучше всего руководствоваться правилом: исключения запускаются только тогда, когда поведение функции не соответствует ее спецификации.

Асинхронные события

Система signal() из стандартного языка C и все аналогичные системы обрабатывают асинхронные события, то есть события, которые происходят вне нормальной последовательности выполнения программы и появление которых невозможно предугадать. Исключения C++ не могут использоваться для обработки асинхронных событий, потому что исключение и его обработчик принадлежат к одному стеку вызова. Другими словами, работа исключений основана на динамической цепочке вызовов функций в стеке программы (они имеют «динамическую видимость»), тогда как асинхронные события должны обрабатываться совершенно отдельным кодом (как правило — процедурами обработки прерываний или циклами событий), который не входит в нормальную последовательность выполнения программы. Не запускайте исключения из обработчиков прерываний!

---

[1] Библиотечная функция uncaught_exception() возвращает true в процессе раскрутки стека, так что теоретически можно сравнить uncaught_exception() с false и обработать возникшее исключение внутри деструктора. Тем не менее, мы еще не видели ни одной хорошей архитектуры, основанной на подобном решении, поэтому такая возможность упоминается лишь в сноске.

Впрочем, это не значит, что асинхронные события не могут *ассоциироваться* с исключениями. Просто обработчик прерывания должен отработать как можно быстрее и вернуть управление. Обычно для этого в обработчике прерывания устанавливается флаг, синхронно проверяемый в основном коде программы.

Устранимые ошибки

Если вы располагаете достаточной информацией для обработки ошибки, запускать исключение не требуется. Решите проблему в текущем контексте, не передавая ее в контекст более высокого уровня.

Кроме того, исключения C++ не запускаются для событий машинного уровня (таких, как деление на ноль[1]). Предполагается, что такие события обрабатываются другими средствами, например операционной системой или оборудованием. При таком подходе исключения C++ работают с разумной эффективностью, а их применение ограничивается условиями программного уровня.

Управление последовательностью выполнения

На первый взгляд кажется, что исключения представляют собой альтернативный механизм возврата управления или отдаленное подобие команды switch. У некоторых программистов возникает искушение использовать исключения вместо стандартных языковых механизмов, но делать этого не стоит. Прежде всего, обработка исключений значительно уступает по эффективности нормальному выполнению программы. Не стоит идти на затраты, связанные с исключениями, в обычной программе. Кроме того, исключения, причиной которых не являются ошибки, собьют с толку пользователей вашего класса или функции.

Исключения не обязательны

Некоторые программы (например, несложные утилиты, ограничивающиеся получением ввода и его элементарной обработкой) достаточно просты. В таких программах тоже могут происходить сбои: неудачные попытки выделения памяти, открытия файлов и т. д. В таких программах можно вывести сообщение и поручить системе «прибрать за программой», вместо того чтобы самостоятельно перехватывать все исключения и освобождать все ресурсы. Короче говоря, если ваша программа может обойтись без исключений — не используйте их.

Новые исключения, старый код

Еще одна характерная ситуация возникает при модификации существующих программ, не поддерживающих исключения. Допустим, к системе подключается библиотека, в которой *используются* исключения; нужно ли изменять весь существующий код? Если в системе уже имеется нормальная схема обработки ошибок, самое тривиальное решение — заключить максимально большой блок, в котором используется новая библиотека (возможно, это будет весь код main()), в блок try с catch(...) с выводом простейших сообщений об ошибках. Представленная схема может уточняться до произвольной степени за счет добавления специализированных обработчиков, но в любом случае объем нового кода должен быть минимальным. Еще лучше изолировать код, генерирующий исключения, в блоке try, и написать обработчики для перевода исключений в существующую схему обработки ошибок.

---

[1] Некоторые компиляторы генерируют исключения в подобных случаях, но обычно у них предусмотрен специальный ключ для подавления этого (нестандартного) поведения.

Очень важно помнить об исключениях при создании библиотеки, которой будут пользоваться другие — особенно если вы не знаете, как им потребуется реагировать на критические ошибки (вспомните, что уже говорилось о безопасности исключений и о том, почему в стандартной библиотеке C++ отсутствуют спецификации исключений).

Типичные применения исключений

Используйте исключения для решения следующих задач:

- проблемы с повторным вызовом функции, в которой произошло исключение;
- восстановление и продолжение программы без повторного вызова функции;
- выполнение всех возможных действий в текущем контексте и перезапуск *того же* исключения в контексте следующего уровня;
- выполнение всех возможных действий в текущем контексте и запуск *другого* исключения в контекст следующего уровня;
- завершение программы;
- создание оболочек для функций (особенно библиотечных функций C), использующих обычные схемы обработки ошибок;
- упрощение программы. Громоздкие и неудобные системы обработки ошибок нередко усложняют работу программ. Возможно, исключения упростят обработку ошибок и повысят ее эффективность;
- повышение надежности библиотеки и программы. Выигрыш достигается как краткосрочный (отладка), так и долгосрочный (повышение устойчивости приложения).

Когда использовать спецификации исключений

Спецификация исключений похожа на прототип функции: она указывает пользователю, что он должен написать код обработки ошибок, и сообщает, какие исключения нужно обрабатывать. По спецификации компилятор узнает, какие исключения могут быть сгенерированы функцией, чтобы обнаружить возможные нарушения на стадии выполнения.

Простой просмотр кода не всегда позволяет определить, какие исключения могут быть запущены той или иной функцией. Иногда вызываемая функция порождает непредвиденные исключения, а иногда старая функция, не запускавшая исключений, заменяется новой функцией с исключениями, что приводит к вызову `unexpected()`. Каждый раз, когда вы используете спецификации исключений или вызываете функции, в которых они есть, подумайте о создании собственной функции `unexpected()`. Такая функция регистрирует сообщение, а затем либо запускает исключение, либо завершает программу.

Как объяснялось ранее, не стоит использовать спецификации исключений в шаблонных классах, поскольку у вас нет информации о том, какие типы исключений могут запускаться классами-параметрами.

Начинайте со стандартных исключений

Прежде чем создавать собственные исключения, ознакомьтесь с исключениями стандартной библиотеки C++. Если стандартное исключение делает то, что вам нужно, скорее всего, будет проще и удобнее работать с ним.

Если нужный тип исключения отсутствует в стандартной библиотеке, попробуйте определить его наследованием от одного из существующих исключений. Хорошо, если ваши пользователи в своих программах всегда смогут рассчитывать на доступность функции what(), определенной в интерфейсе класса exception.

Вложение специализированных исключений

Если вы создаете исключения для своего конкретного класса, желательно вложить классы исключений либо внутрь класса, либо внутрь пространства имен, содержащего класс. Тем самым вы четко сообщаете читателю программы, что исключение используется только вашим классом. Кроме того, вложенные исключения не загромождают глобальное пространство имен.

Вложенными могут быть даже исключения, производные от стандартных исключений C++.

Иерархии исключений

Иерархии исключений являются хорошим средством классификации типов критических ошибок, возникающих в классе или библиотеке. Такая классификация сообщает полезную информацию пользователям, помогает в организации программного кода, а также дает возможность игнорировать конкретные типы исключений и перехватывать исключения базового типа. Последующее добавление новых исключений, производных от того же базового класса, не потребует модификации всего существующего кода — обработчик базового класса успешно перехватит новое исключение.

Стандартные исключения C++ дают хороший пример иерархии исключений. Если удастся, попробуйте создать свои исключения на их базе.

Множественное наследование

Как будет показано в главе 9, *необходимость* во множественном наследовании возникает лишь тогда, когда указатель на объект требуется повышать до двух разных базовых классов (то есть вам требуется полиморфное поведение в отношении обоих базовых классов). Оказывается, применение множественного наследования в иерархиях исключений вполне уместно, потому что исключение может быть обработчиком любого из «корней» иерархии множественного наследования.

Перехват по ссылке

Как было показано в разделе «Поиск подходящего обработчика», исключения стоит перехватывать по ссылке, а не по значению, по двум причинам:

- чтобы избежать ненужного копирования объекта исключений при передаче обработчику;
- чтобы предотвратить усечение объекта при перехвате производного исключения как объекта базового класса.

Хотя вы также можете запускать и перехватывать указатели, это лишь уменьшает свободу действий: обе стороны (запускающая и перехватывающая) должны

согласовать процедуры выделения и освобождения памяти. Это может вызвать проблемы, поскольку само исключение могло произойти из-за нехватки свободной памяти в куче. При запуске объектов-исключений система обработки исключений берет все хлопоты с памятью на себя.

Запуск исключений в конструкторах

Так как конструктор не имеет возвращаемого значения, раньше об ошибках конструирования можно было сообщить двумя способами:

- установить нелокальный флаг и надеяться, что пользователь проверит его;
- вернуть частично созданный объект и надеяться, что пользователь проверит его.

Возникает серьезная проблема: программисты С склонны полагать, что объекты всегда создаются успешно. В С такие ожидания вполне оправданны благодаря примитивности типов. Но дальнейшее выполнение после неудачного конструирования в C++ кончится катастрофой, поэтому конструкторы являются одним из основных кандидатов на запуск исключений — в вашем распоряжении появляется надежный, эффективный способ обработки ошибок конструирования. Однако при этом необходимо следить за указателями внутри объектов и за тем, как происходит освобождение ресурсов при запуске исключений в конструкторах.

Исключения в деструкторах запрещены

Деструкторы вызываются в процессе запуска исключений, поэтому никогда не следует запускать исключения в деструкторах или выполнять какие-либо действия, которые могут привести к запуску исключений в деструкторе. Если это произойдет, новое исключение может быть запущено *раньше* достижения секции catch текущего исключения, что приведет к вызову terminate().

Если в деструкторе вызываются функции, которые могут запускать исключения, эти вызовы должны быть заключены в блок try внутри деструктора, и деструктор должен обработать все исключения сам. Ни одно исключение не должно выйти за пределы деструктора.

Избегайте низкоуровневых указателей

Вернитесь к примеру Wrapped.cpp, представленному ранее в этой главе. Низкоуровневый указатель, для которого в конструкторе выделяются ресурсы, обычно создает потенциальную угрозу утечки памяти. Указатель не имеет деструктора, поэтому ресурсы не освобождаются при возникновении исключения в конструкторе. В качестве указателей, ссылающихся на память в куче, лучше использовать указатель auto_ptr или другие типы умных указателей[1].

Издержки обработки исключений

Запуск исключения сопряжен с существенными издержками (но это *полезные* затраты, потому что объекты зачищаются автоматически!). По этой причине исклю-

[1] Ознакомьтесь с умными указателями по адресу http://www.boost.org/libs/smart_ptr/index.htm. Некоторые из них рекомендованы к включению в следующую версию стандартного языка C++.

чения никогда не должны использоваться как часть обычной последовательности выполнения программы, как бы эффектно и умно это ни выглядело. Исключения должны происходить очень редко, чтобы на издержки приходилось идти лишь в особых случаях, но не при обычном выполнении программы. Одной из важных задач, учитывавшихся при проектировании механизма исключений, было сохранение прежней скорости выполнения программы. Другими словам, если программа не запускает исключения, она должна работать так же быстро, как без обработки исключений. Насколько успешно решена эта задача — зависит от конкретной реализации компилятора, используемой вами (см. описание «модели с нулевыми затратами» далее в этом разделе).

Выражение throw может рассматриваться как вызов специальной системной функции, которая получает объект исключения как аргумент и осуществляет возврат в текущей цепочке вызовов. Чтобы этот механизм работал, компилятор должен сохранить в стеке специальную информацию, используемую при раскрутке стека времени выполнения. Впрочем, для понимания сути происходящего нужно познакомиться со стеком времени выполнения (далее — просто стек).

При вызове функции информация о ней заносится в стек в виде *экземпляра активационной записи* (Activation Record Instance, ARI), также называемый *кадром стека*. Типичный кадр стека состоит из адреса вызывающей функции (по которому возвращается управление), указателя на ARI статического родителя функции (области видимости, лексически содержащей вызванную функцию, для обращения к переменным, глобальным по отношению к функции), и указателя на вызвавшую функцию (*динамического родителя*). Путь, образуемый ссылками на динамических родителей всех уровней, называется *динамической цепочкой*, или *цепочкой вызовов* (этот термин уже встречался ранее в этой главе). Именно этот механизм делает возможным возврат управления при запуске исключения, и он же позволяет разрабатывать независимые компоненты с возможностью обмена информацией об ошибках во время выполнения программы.

Чтобы раскрутка стека при обработке исключений стала возможной, в каждый кадр стека необходимо включить дополнительную информацию о каждой функции. Эта информация указывает, какие деструкторы должны быть вызваны (для освобождения локальных объектов), а также сообщает, имеет ли текущая функция блок try и какие исключения обрабатываются соответствующими секциями catch. На хранение дополнительной информации расходуется память, поэтому программы с поддержкой исключений обычно занимают больше памяти, чем программы, в которых исключения не используются[1]. Даже откомпилированная программа с обработкой исключений имеет больший размер, поскольку компилятору приходится генерировать логику построения кадров стека с дополнительной информацией.

Для демонстрации следующая программа с поддержкой исключений и без нее была откомпилирована в Borland C++ Builder и Microsoft Visual C++[2]:

```
//: C01: HasDestructor.cpp {O}
struct HasDestructor {
```

[1] При сравнении также следует учитывать объем кода проверки возвращаемого значения, который потребовался бы, если бы обработка исключений отсутствовала.

[2] Borland разрешает исключения по умолчанию, а для их запрета используется ключ компилятора –x. Microsoft по умолчанию запрещает исключения, и они включаются ключом –GX. В обоих компиляторах ключ –c активизирует режим «только компиляция» (без сборки).

```
  ~HasDestructor(){}
};

void g();      // Функция g может запускать исключения

void f() {
   HasDestructor h;
   g();
} ///:~
```

При включенной обработке исключений компилятор должен хранить в кадре стека f() информацию о доступности ~HasDestructor() во время выполнения программы (чтобы при возникновении исключения в g() можно было правильно уничтожить h). В табл. 1.1 приведены данные по объемам откомпилированных файлов (.obj) в байтах.

Таблица 1.1. Размеры откомпилированных файлов в программах с поддержкой и без поддержки исключений

Компилятор	С поддержкой исключений	Без поддержки исключений
Borland	616	234
Microsoft	1162	680

Не придавайте слишком большого значения различиям между двумя режимами. Помните, что система обработки исключений обычно составляет минимальную часть программы, так что реальные затраты памяти оказываются гораздо меньше (обычно от 5 до 15 %).

Лишние операции замедляют работу программы, но умная реализация компилятора решает проблему. Поскольку сведения об обработке исключений и смещениях локальных объектов могут быть вычислены на стадии компиляции, они хранятся не в кадре стека, а в конкретном месте, связанном с каждой функцией. В сущности, вы выводите лишнюю информацию из кадра стека, а следовательно — экономите время на ее занесении в стек. Такое решение называется моделью *обработки исключений с нулевыми затратами*, а оптимизированное место хранения данных вне стека называется *теневым стеком*.

Итоги

Восстановление работы после возникновения ошибок является одной из основных проблем практически в любой программе. Это восстановление особенно важно в C++ при написании программных компонентов, которые будут использоваться другими программистами. Чтобы система в целом была устойчивой к ошибкам, устойчивым должен быть каждый из ее компонентов.

Механизм обработки исключений в C++ позволяет упростить разработку больших надежных программ, содержащих минимально возможный объем вспомогательного кода и снижающих вероятность возникновения необработанных ошибок. Поставленная цель достигается с незначительными потерями эффективности и с незначительным влиянием на работу существующих программ.

Освоить элементарную обработку исключений совсем несложно; после освоения начинайте использовать исключения в своих программах как можно раньше.

Исключения принадлежат к числу тех возможностей, польза от которых весьма существенна.

Упражнения

1. Напишите три функции. Первая функция возвращает код ошибки, вторая — устанавливает значение errno, а третья — использует вызов signal(). Напишите программу с вызовами этих функций и обработкой ошибок. Затем напишите четвертую функцию, которая запускает исключение, вызовите ее и перехватите исключение. Опишите различия между этими четырьмя решениями и объясните преимущества схемы с обработкой исключений.
2. Создайте класс, функции которого запускают исключения. Определите внутри него вложенный класс для использования в качестве объекта исключения. При конструировании класс исключения получает один аргумент const char*, представляющий строку описания. Создайте функцию класса, запускающую это исключение (укажите это в спецификации исключений функции). Напишите блок try с вызовом этой функции и секцию catch, которая выводит строку описания.
3. Перепишите класс Stash из главы 13 первого тома так, чтобы оператор [] генерировал исключения out_of_range.
4. Напишите обобщенную функцию main(), которая бы перехватывала все исключения и сообщала о них, как об ошибках.
5. Создайте класс с собственным оператором new. Оператор должен выделять память для десяти объектов, а на одиннадцатом объекте запускать исключение «из-за нехватки памяти». Включите в класс статическую функцию для освобождения этой памяти. Затем напишите функцию main() с блоком try и секцией catch, которая бы вызывала функцию освобождения памяти. Поместите их в цикл while и продемонстрируйте восстановление после исключения с продолжением работы программы.
6. Создайте деструктор, запускающий исключение. Напишите программу, которая бы показывала, что запуск нового исключения до завершения обработки предыдущего приводит к вызову terminate().
7. Убедитесь в том, что все объекты исключений (запускаемых) должным образом уничтожаются.
8. Создайте объект исключения в куче и запустите указатель на него. Убедитесь в том, что этот объект не будет уничтожаться.
9. Напишите функцию со спецификацией исключений, запускающую исключения типов char, int, bool, а также пользовательского типа. Перехватите все исключения в main() и покажите, что они действительно перехватываются. Определите пользовательский класс исключения производным от стандартного исключения. Напишите функцию таким образом, чтобы программа восстанавливалась после ошибки, и попробуйте выполнить ее снова.

10. Измените предыдущий пример так, чтобы функция запускала исключение типа double, нарушающее спецификацию исключений. Перехватите нарушение в пользовательском обработчике unexpected(), который выводит сообщение и корректно завершает программу (то есть без вызова abort()).
11. Напишите класс Garage с вложенным классом Car, содержащим вложенный класс Motor. Используйте блок try уровня функции в конструкторе класса Garage для перехвата исключения из класса Motor при инициализации Car. Запустите другое исключение из тела обработчика конструктора Garage и перехватите его в main().

Защитное программирование

Возможно, «идеальная программа» — не более чем абстрактное понятие для разработчиков. И все же некоторые приемы защитного программирования, применяемые в повседневной работе, существенно повысят качество вашего кода.

Сложность большинства программных проектов гарантирует, что специалисты по тестированию никогда не останутся без работы. И все же хочется верить, что вы будете стремиться создавать программы без дефектов. Объектно-ориентированные приемы проектирования помогают держать под контролем сложность больших проектов, но в конечном счете вам все равно придется писать циклы и функции. Эти мелочи «повседневного программирования» превращаются в строительные блоки для более крупных компонентов ваших разработок. Если счетчик цикла принимает лишнее значение или функции возвращают правильное значение «в большинстве случаев», дело кончится плохо даже при самой хитроумной методологии. В этой главе рассматриваются приемы, которые повышают надежность кода независимо от сложности проекта.

Среди прочего, ваша программа является выражением вашего подхода к решению некоторой задачи. Читателю программы (в том числе и вам) должно быть абсолютно ясно, о чем вы думали при написании того или иного цикла. В некоторых точках программы вы должны быть способны уверенно заявить об истинности тех или иных предположений (а если нет, значит, задача еще не решена). Такие утверждения называются *инвариантами*, потому что они неизменно должны быть истинными в конкретной точке программы. Невыполнение инварианта говорит либо о плохой архитектуре проекта, либо о неточном отражении ее программой.

Рассмотрим программу для игры «угадай число». Один игрок задумывает число от 1 до 100, а другой пытается угадать его (пусть этим занимается компилятор). Игрок, загадавший число, сообщает угадывающему, как связана его догадка с загаданным числом — меньше, больше или равна ему. Оптимальная стратегия угадывания основана на *бинарном поиске*, при котором выбирается средняя точка в интервале искомых чисел. По ответу «больше-меньше» угадывающий определяет,

в какой половине интервала находится искомое число, после чего процесс повторяется. При каждой итерации размер интервала поиска уменьшается вдвое. Как же должен выглядеть правильный цикл угадывания числа? Поскольку зловредный пользователь может солгать, и дальнейшее угадывание затянется до бесконечности, недостаточно просто написать:

```
bool guessed = false;
while(!guessed) {
  ...
}
```

Какое предположение, каким бы простым оно ни было, делается при каждой итерации? Иначе говоря, какое условие *благодаря архитектуре приложения* должно автоматически выполняться при каждой итерации цикла?

Это простое предположение заключается в следующем: загаданное число принадлежит текущему активному интервалу непроверенных чисел. Обозначим конечные точки этого интервала переменными low и high. Если в начале каждой итерации число принадлежало интервалу [low, high], то в конце ее мы вычисляем новый интервал так, чтобы он по-прежнему содержал загаданное число.

Наша цель — выразить инвариант цикла в программе, чтобы его нарушения обнаруживались во время ее выполнения. Компилятор не знает загаданное число, поэтому прямая проверка невозможна, но для начала можно вставить обычный комментарий:

```
while(!guessed) {
  // ИНВАРИАНТ: число находится в интервале [low, high]
  ...
}
```

Что произойдет, если ответ пользователя «больше» или «меньше» не соответствует действительности? Загаданное число будет исключено из нового интервала. Поскольку одна ложь всегда влечет за собой другую, со временем мы придем к пустому интервалу (интервал каждый раз уменьшается вдвое, а загаданное число в нем отсутствует). Это условие выражается в следующей программе:

```
//: C02:HiLo.cpp {RunByHand}
// Демонстрация инварианта цикла на примере игры "угадай число"
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;

int main() {
  cout << "Think of a number between 1 and 100\n";
  cout << "I will make a guess; ";
  cout << "tell me if I'm (H)igh or (L)ow\n";
  int low = 1, high = 100;
  bool guessed = false;
  while (!guessed) {
    // Инвариант: число находится в интервале [low, high]
    if (low > high) {  // Нарушение инварианта
      cout << "You cheated! I quit\n";
      return EXIT_FAILURE;
    }
    int guess = (low + high) / 2;
    cout << "My guess is " << guess << ". ";
    cout << "(H)igh, (L)ow, or (E)qual? ";
```

```
    string response;
    cin >> response;
    switch(toupper(response[0])) {
      case 'H':
        high = guess - 1;
        break;
      case 'L':
        low = guess + 1;
        break;
      case 'E':
        guessed = true;
        break;
      default:
        cout << "Invalid response\n";
        continue;
    }
  }
  cout << "I got it!\n";
  return EXIT_SUCCESS;
} ///:~
```

Нарушение инварианта обнаруживается проверкой условия if(low>high). Если пользователь всегда говорит правду, то загаданное число рано или поздно будет найдено.

В этой программе также используется стандартный для языка C способ передачи информации о состоянии программы в контекст вызова: возврат разных значений функцией main(). Команда return 0; на любой платформе обозначает успешное завершение, однако не существует переносимого кода неудачного завершения. По этой причине мы используем макрос EXIT_FAILURE, объявленный для этой цели в файле <cstdlib>. Ради единства стиля также используется макрос EXIT_SUCCESS, хотя он всегда определяется как 0.

Утверждения

Критерий проверки в программе «угадай число» зависит от пользовательского ввода, поэтому предотвратить его нарушение невозможно. Однако чаще инварианты зависят только от кода, написанного вами, и при правильной реализации исходной архитектуры всегда остаются истинными. В таких случаях лучше воспользоваться положительным *утверждением*, в котором отражено решение, принятое в процессе проектирования.

Допустим, вы реализуете целочисленный вектор — массив, размеры которого изменяются по мере необходимости. Функция добавления элемента в вектор должна сначала убедиться в том, что в базовом массиве имеется свободная позиция; в противном случае она должна запросить дополнительную память в куче и скопировать в нее существующие элементы (а также удалить старый массив). Такая функция выглядит примерно так:

```
void MyVector::push_back(int x) {
  if (nextSlot == capacity)
    grow();
  assert(nextSlot < capacity);
  data[nextSlot++] = x;
}
```

В приведенном примере data — динамический массив с элементами типа int и емкостью capacity, в котором используется nextSlot элементов. Функция grow() должна расширить data так, чтобы новое значение capacity было строго больше nextSlot. От этого решения, обусловленного архитектурой приложения, зависит правильность работы массива MyVector, и если прочие части программы не содержат ошибок, данное условие всегда должно выполняться. Мы организуем его проверку макросом assert(), определенным в заголовочном файле <cassert>.

Макрос assert() стандартной библиотеки C лаконичен, предельно конкретен и работает на всех платформах. Если результат выражения, передаваемого в параметре, отличен от нуля, то программа продолжает выполняться. В противном случае в стандартный поток ошибок выводится текст проверяемого условия с именем исходного файла и номером строки, а программа аварийно завершается. Не слишком ли радикально? Практика показывает, что продолжение работы программы с нарушением базовых условий приводит к гораздо худшим последствиям. Программу необходимо исправлять.

Если все идет нормально, к моменту выхода окончательной версии продукта ваш код будет протестирован с проверкой всех утверждений (мы поговорим о тестировании далее). В зависимости от характера приложения может оказаться, что проверка всех утверждений слишком сильно отразится на его эффективности. Но в этом случае существует простой выход: автоматически удалить код утверждений. Для этого достаточно определить макрос NDEBUG и построить приложение заново.

Чтобы вы лучше поняли, как это делается, рассмотрим типичную реализацию assert():

```
#ifdef NDEBUG
  #define assert(cond) ((void)0)
#else
  void assertImpl(const char*, const char*, long);
  #define assert(cond) \
  ((cond) ? (void)0 : assertImpl(???))
#endif
```

При определенном макросе NDEBUG этот фрагмент вырождается в выражение (void)0, поэтому в потоке компиляции фактически остается лишь пустая команда — результат присутствия символа точки с запятой (;) после каждого вызова assert(). Если символическая переменная NDEBUG не определена, assert(cond) расширяется до условной команды, которая при нулевом условии cond вызывает функцию, специфическую для компилятора (assertImpl()). Строковый аргумент этой функции представляет текст cond, имя файла и номер строки программы, в которой проверялось утверждение. (В нашем примере использован заполнитель ???, но на самом деле возвращаемая строка генерируется в точке вызова макроса. Вопрос о том, откуда берутся необходимые данные, не относится к обсуждаемой теме.) Чтобы разрешать или запрещать проверку утверждений в различных точках программы, необходимо не только включать директивы #define NDEBUG или #undef NDEBUG, но и заново включать файл <cassert>. Макросы обрабатываются по мере их обнаружения препроцессором и поэтому используют текущее состояние NDEBUG для точки включения. Чаще всего NDEBUG определяется сразу для всей программы при помощи ключа компилятора: либо в настройках проекта в визуальной среде, либо в командной строке вида

```
mycc -DNDEBUG myfile.cpp
```

Большинство компиляторов использует флаг -D для определения макросов в командной строке (замените mycc именем своего компилятора). Преимущество такого подхода состоит в том, что вы можете оставить утверждения в исходном тексте программы как бесценную документацию, и при этом полностью избавиться от лишних затрат на стадии выполнения. Поскольку код утверждения исчезает при определении NDEBUG, очень важно, чтобы в утверждениях *не выполнялась никакая полезная работа* — только проверка условий, не влияющих на состояние программы.

Стоит ли использовать макрос NDEBUG в окончательной версии программы? На эту тему до сих пор нет единого мнения. Тони Хоар (Tony Hoare), один из самых авторитетных программистов всех времен и народов[1], считает, что отключение проверок на стадии выполнения напоминает энтузиаста-яхтсмена, который носит спасательный жилет на суше, но снимает его перед выходом в море. Если вдруг окажется, что утверждение не выполняется в готовом продукте, у вас возникнут серьезные проблемы... куда более серьезные, чем небольшое снижение быстродействия. Короче, выбирайте разумно.

Однако не все условия должны проверяться при помощи утверждений. Как подробно объяснялось в главе 1, о пользовательских ошибках и сбоях ресурсов лучше сообщать запуском исключений. В процессе черновой разработки программы возникает искушение — использовать утверждения для большинства ошибок, чтобы позднее заменить многие из них более надежной обработкой исключений. Как и при любом искушении, здесь необходима осторожность, потому что позднее вы можете забыть обо всех необходимых изменениях. Помните: утверждения предназначены для проверки условий, определенных архитектурой приложения, которые могут быть нарушены лишь из-за логических ошибок программиста. В идеале все нарушения утверждений должны быть обнаружены на стадии разработки. Не используйте утверждения для проверки условий, не находящихся под вашим полным контролем (например, зависящих от пользовательского ввода). В частности, не используйте утверждения для проверки аргументов функций; лучше запустите исключение logic_error.

Применение утверждений как средства проверки правильности программы было формализовано Бертраном Мейером (Bertrand Meyer) в предложенной им методологии «проектирования по контракту». Каждая функция заключает со своими клиентами неявный контракт, в соответствии с которым выполнение некоторых *предусловий* гарантирует выполнение некоторых *постусловий*. Иначе говоря, предусловия определяют требования к использованию функции (например, передача аргументов со значениями в определенных интервалах), а постусловия определяют результаты, выдаваемые функцией (возвращаемое значение или побочные эффекты).

Если клиентская программа предоставляет недействительные входные данные, необходимо сообщить ей о нарушении контракта. Завершать программу все же не стоит (хотя вы имеете на это полное право, поскольку контракт был нарушен), правильнее запустить исключение. Именно в таких случаях стандартная библиотека C++ генерирует исключения, производные от logic_error, — такие, как out_of_range[2].

---

[1] В частности, он изобрел алгоритм быстрой сортировки.

[2] С концептуальной точки зрения происходящее эквивалентно проверке утверждения, но поскольку выполнение программы не должно прерываться, макрос assert() в данном случае не подходит. Например, в Java 1.4 при нарушении утверждения запускается исключение.

Но если речь идет о функции, которую вызываете только вы и никто другой (например, закрытая функция в спроектированном вами классе), макрос assert() оказывается более уместным. Вы в полной мере контролируете ситуацию и, несомненно, хотите отладить свою программу перед распространением окончательной версии.

Нарушение постусловий свидетельствует об ошибке в программе. Утверждения уместно использовать *для любых инвариантов в любой момент*, включая проверку постусловия в конце функции. В частности, это относится к функциям классов, поддерживающим состояние объекта. Например, в приведенном выше примере класса MyVector разумный инвариант для всех открытых функций класса мог бы выглядеть так:

```
assert(0 <= nextSlot && nextSlot <= capacity);
```

А если nextSlot является беззнаковым целым, можно так:

```
assert(nextSlot <= capacity);
```

Такой инвариант называется *инвариантом класса*, а его соблюдение вполне может обеспечиваться проверкой утверждения. Производные классы играют роль *субконтрагентов* по отношению к своим базовым классам, потому что они должны сохранить исходный контракт между базовым классом и его клиентами. По этой причине предусловия производных классов не должны выдвигать дополнительных требований, выходящих за рамки базового контракта, а постусловия должны выполняться, по крайней мере, в заданном объеме[1].

С другой стороны, проверка результатов, возвращаемых клиенту, представляет собой не что иное, как *тестирование*, поэтому проверка постусловных утверждений в этом случае приводит лишь к дублированию усилий. Да, утверждения помогают документировать программу, но слишком многие разработчики ошибочно полагали, будто проверка постусловных утверждений способна заменить модульное тестирование.

Простая система модульного тестирования

Все программирование, в конечном счете, сводится к выполнению требований[2]. Создать требования нелегко, к тому же они могут меняться. На еженедельном совещании по проекту может выясниться, что всю неделю вы работали не совсем над тем, чего от вас хотел заказчик.

Человек не может сформулировать четкие требования к программе, если в его распоряжении не будет развивающегося, работающего прототипа. Вы понемногу формулируете требования, понемногу программируете и понемногу тестируете. Затем вы оцениваете результат, после чего все повторяется заново. Возможность проведения интерактивной разработки такого рода принадлежит к числу важней-

---

[1] Для запоминания этого принципа существует хорошая формулировка: «Требовать не больше; обещать не меньше», впервые выдвинутая Маршаллом Клайном (Marshall Kline) и Грегом Ломоу (Greg Lomow). Поскольку предусловия могут ослабляться в производных классах, их иногда называют *контравариантными*, тогда как постусловия, наоборот, являются *ковариантными* (кстати, это объясняет упоминание ковариантности спецификаций исключений в главе 1).

[2] Данный раздел основан на статье Чака Эллисона «The Simplest Automated Unit Test Framework That Could Possibly Work» в журнале «C/C++ Users Journal», сентябрь 2000 г.

ших преимуществ объектно-ориентированных технологий, но для нее необходимы творчески мыслящие программисты, способные создавать гибкий код. Изменяться всегда трудно.

Другой импульс к изменению дает сам программист. Художник внутри вас хочет постоянно совершенствовать архитектуру ваших программ. Какой программист, которому доводилось заниматься сопровождением чужих творений, не проклинал устаревающий продукт, превратившийся в сплошной клубок запутанной логики с навешанными там и сям заплатками? Однако руководство предпочитает, чтобы вы даже не пытались совершенствовать кое-как работающую систему, и это мешает вам внести в код столь нужную гибкость. Известное правило «Не сломано — не чини» постепенно заменяется другим: «Чинить бесполезно — проще переписать». Изменения неизбежны.

К счастью, наша отрасль постепенно привыкает к понятию *переработки* — искусству внутренней реструктуризации кода с целью улучшения его архитектуры без изменения его поведения. К числу таких усовершенствований относится выделение одной функции из другой или наоборот, объединение функций; замена функции класса объектом; параметризация отдельных функций или целых классов, а также замена условных проверок полиморфизмом. Переработка способствует эволюции программ.

Откуда бы ни исходило стремление к изменениям, от пользователей или программистов, сегодняшние модификации могут нарушить то, что еще вчера успешно работало. Нам нужна методика построения программ, адаптирующихся к изменениям и совершенствующихся с течением времени.

Экстремальное программирование[1] (eXtreme Programming, XP) — всего лишь одна из многих вариаций на тему ускорения разработки программного обеспечения. В этом разделе мы рассмотрим то, что, как нам кажется, является одним из ключевых факторов гибкой, последовательной разработки: простую и удобную систему автоматизации модульного тестирования. (Впрочем, *тестеры* — профессионалы, зарабатывающие на жизнь тестированием чужих программ — по-прежнему незаменимы. Здесь всего лишь описывается методика, которая помогает разработчикам писать более качественные программы.)

Разработчики пишут *модульные тесты*, чтобы с уверенностью сделать два самых важных заявления в работе любого программиста:

- я понимаю предъявленные требования;
- моя программа (насколько мне известно) соответствует этим требованиям.

Лучший способ убедиться в том, что вы правильно понимаете, как должна работать программа — начать с написания модульных тестов. Это простое упражнение поможет вам сосредоточиться на предстоящей работе и, скорее всего, быстрее приведет к конечному результату, чем если вы просто возьметесь за программирование. В терминологии XP это выражается так:

тестирование + программирование быстрее, чем просто программирование

---

[1] См.: Бек К., Фаулер М. Экстремальное программирование: планирование. Библиотека программиста. СПб.: Питер, 2003; и Ауэр К., Миллер Р. Экстремальное программирование: постановка процесса. С первых шагов и до победного конца». СПб.: Питер, 2004. — *Примеч. перев.*

Предварительное написание тестов также помогает защититься от граничных условий, способных нарушить работу программы, так что программа становится более надежной.

Если программа прошла все тесты, вы знаете, что в случае неработоспособности системы причина, скорее всего, кроется не в вашем коде. Заявление «Все тесты проходят нормально» — весьма убедительный аргумент.

Автоматизация тестирования

Как же выглядит модульный тест? Слишком часто разработчики берут «хорошие» входные данные, получают ожидаемые результаты и бегло просматривают их. У такого подхода есть два недостатка. Прежде всего, программы не всегда получают «хорошие» данные. Все знают, что входные данные нужно протестировать по граничным условиям, но об этом трудно думать, когда вы просто добиваетесь хоть какой-то работоспособности программы. Если написать тест для функции перед тем, как браться за ее программирование, вы сможете представить себя на месте тестера и спросить себя: «Как бы это сломать?» Напишите тест, который бы доказывал, что программируемая вами функция действительно работает, а потом снова перевоплощайтесь в программиста и пишите саму функцию. Программа получится более качественной, чем без предварительного написания теста.

Вторая опасность состоит в том, что визуальный просмотр выходных данных утомителен и чреват ошибками. Компьютер способен взять на себя большую часть нетворческой работы, выполняемой человеком, но без присущих человеку ошибок. Лучше сформулировать тесты в виде набора *логических выражений* и заставить тестовую программу сообщить обо всех нарушениях.

Предположим, вы создаете класс Date, который должен обладать следующими свойствами:

- дата может инициализироваться строкой (ГГГГММДД), тремя целыми числами (Г, М, Д) или «ничем» (для текущей даты);
- объект даты по запросу возвращает год, месяц, день или строку в формате «ГГГГММДД»;
- класс поддерживает все операции сравнения, а также вычисление промежутков между двумя датами (в годах, месяцах и днях);
- сравниваемые даты могут быть разделены произвольным количеством веков (например, 1600–2200).

Класс содержит три целочисленные переменные для хранения года, месяца и дня (проследите за тем, чтобы год представлялся минимум 16 битами для выполнения последнего пункта в списке). Интерфейс класса Date выглядит примерно так:

```
//: C02:Date1.h
// Первая версия Date.h
#ifndef DATE1_H
#define DATE1_H
#include <string>

class Date {
public:
```

```
  // Структура для представления промежутков времени
  struct Duration {
    int years;
    int months;
    int days;
    Duration(int y, int m, int d)
    : years(y), months(m) ,days(d) {}
  };
  Date();
  Date(int year, int month, int day);
  Date(const std::string&);
  int getYear() const;
  int getMonth() const;
  int getDay() const;
  std::string toString() const;
  friend bool operator<(const Date&, const Date&);
  friend bool operator>(const Date&, const Date&);
  friend bool operator<=(const Date&, const Date&);
  friend bool operator>=(const Date&, const Date&);
  friend bool operator==(const Date&, const Date&);
  friend bool operator!=(const Date&, const Date&);
  friend Duration duration(const Date&, const Date&);
};
#endif // DATE1_H ///:~
```

Прежде чем браться за реализацию класса, стоит укрепить свое понимание требований и написать прототип тестовой программы. Вероятно, эта программа могла бы выглядеть примерно так:

```
//: C02:SimpleDateTest.cpp
//{L} Date
#include <iostream>
#include "Date.h"   // Из приложения
using namespace std;

// Механизм тестирования
int nPass = 0, nFail = 0;
void test(bool t) {
  if(t) nPass++; else nFail++;
}

int main() {
  Date mybday(1951, 10, 1);
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
}
/* Ожидаемый вывод:
Passed: 3, Failed: 0
*/ ///:~
```

В этом тривиальном случае функция test() содержит глобальные переменные nPass и nFail. Вся визуальная проверка сводится к чтению результатов тестирования. Если тест завершился неудачей, более сложная версия test() выводит соответствующее сообщение. В системе, описанной далее в этой главе, среди прочего будет присутствовать такая тестовая функция.

Теперь можно реализовать часть класса **Date**, достаточную для прохождения тестов, а затем последовательно расширять ее функциональность до соблюдения всех требований. Предварительное написание тестов поможет вам лучше представить граничные случаи, которые могут нарушить будущую реализацию. Итоговая версия теста для класса **Date** могла бы выглядеть примерно так:

```
//: C02:SimpleDateTest2.cpp
//{L} Date
#include <iostream>
#include "Date.h"
using namespace std;

// Механизм тестирования
int nPass = 0, nFail = 0;
void test(bool t) { if(t) nPass++; else nFail++; }

int main() {
  Date mybday(1951, 10, 1);
  Date today;
  Date myevebday("19510930");

  // Тестирование операторов
  test(mybday < today);
  test(mybday <= today);
  test(mybday != today);
  test(mybday == mybday);
  test(mybday >= mybday);
  test(mybday <= mybday);
  test(myevebday < mybday);
  test(mybday > myevebday);
  test(mybday >= myevebday);
  test(mybday != myevebday);

  // Тестирование функций
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  test(myevebday.getYear() == 1951);
  test(myevebday.getMonth() == 9);
  test(myevebday.getDay() == 30);
  test(mybday.toString() == "19511001");
  test(myevebday.toString() == "19510930");

  // Тестирование промежутков времени
  Date d2(2003, 7, 4);
  Date::Duration dur = duration(mybday, d2);
  test(dur.years == 51);
  test(dur.months == 9);
  test(dur.days == 3);

  // Вывод результатов:
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
} ///:~
```

Этот тест можно было бы доработать; в частности, мы не проверяем правильность работы с длинными интервалами времени. Пока остановимся на этом, чтобы

вы поняли общую идею. Полная реализация класса Date содержится в файлах Date.h и Date.cpp в приложении[1].

Система TestSuite

Некоторые автоматизированные средства модульного тестирования C++ (такие как CppUnit[2]) свободно распространяются через Интернет. Наша задача — представить читателю тестовый механизм, который был бы не только прост в использовании, но в котором также можно было бы легко разобраться и даже изменить в случае необходимости. Итак, в соответствии с принципом: «Сделайте самое простое, что может работать»[3], мы разработали *систему TestSuite* — пространство имен TestSuite с двумя ключевыми классами Test и Suite.

Класс Test представляет собой абстрактный базовый класс, на основе которого создаются тестовые объекты. Он отслеживает количество успешно пройденных и неудачных тестов, и выводит текст любого сбойного условия. Вам остается лишь переопределить функцию run(), которая в свою очередь должна вызвать макрос test_() для каждого из определяемых логических условий теста.

Чтобы определить комплекс тестов для класса Date с использованием системы TestSuite, мы объявляем класс, производный от Test, как показано в следующей программе:

```
//: C02:DateTest.h
#ifndef DATE_TEST_H
#define DATE_TEST_H
#include "Date.h"
#include "../TestSuite/Test.h"

class DateTest : public TestSuite::Test {
  Date mybday;
  Date today;
  Date myevebday;
public:
  DateTest() : mybday(1951, 10, 1), myevebday("19510930") {
  }
  void run() {
    testOps();
    testFunctions();
    testDuration();
  }
  void testOps() {
    test_(mybday < today);
    test_(mybday <= today);
    test_(mybday != today);
    test_(mybday == mybday);
    test_(mybday >= mybday);
    test_(mybday <= mybday);
    test_(myevebday < mybday);
    test_(mybday > myevebday);
    test_(mybday >= myevebday);
    test_(mybday != myevebday);
  }
```

[1] Наш класс Date также поддерживает расширенную кодировку символов (эта тема будет представлена в конце следующей главы).

[2] За дополнительной информацией обращайтесь по адресу http://sourceforge.net/projects/cppunit.

[3] Основной принцип экстремального программирования.

```
  void testFunctions() {
    test_(mybday.getYear() == 1951);
    test_(mybday.getMonth() == 10);
    test_(mybday.getDay() == 1);
    test_(myevebday.getYear() == 1951);
    test_(myevebday.getMonth() == 9);
    test_(myevebday.getDay() == 30);
    test_(mybday.toString() == "19511001");
    test_(myevebday.toString() == "19510930");
  }
  void testDuration() {
    Date d2(2003, 7, 4);
    Date::Duration dur = duration(mybday, d2);
    test_(dur.years == 51);
    test_(dur.months == 9);
    test_(dur.days == 3);
  }
};
#endif ///:~
```

Запуск теста сводится к простому созданию объекта **DateTest** и вызову его функции **run()**:

```
//: C02:DateTest.cpp
// Автоматизация тестирования
//{L} Date ../TestSuite/Test
#include <iostream>
#include "DateTest.h"
using namespace std;

int main() {
  DateTest test;
  test.run();
  return test.report();
}
/* Вывод:
Test "DateTest":
        Passed: 21.      Failed: 0
*/ ///:~
```

Функция **Test::report()** отображает приведенную выше сводку и возвращает количество сбоев, поэтому ее результат удобно использовать в качестве возвращаемого значения функции **main()**.

Для получения имени класса, включаемого в отчет (в нашем примере — **DateTest**), класс **Test** использует механизм RTTI (RunTime Type Identification — идентификация типов в процессе исполнения)[1]. Также имеется функция **setStream()** на тот случай, если вы захотите отправить результаты теста в файл вместо стандартного выходного потока (по умолчанию). Реализация класса **Test** будет приведена позднее в этой главе.

Макрос **test_()** извлекает текст логического условия, не прошедшего проверку, вместе с именем файла и номером строки[2]. Если вас интересует, что происходит

---

[1] Конкретно используется функция name() класса typeinfo (см. главу 9). Если вы работаете с компилятором Microsoft Visual C++, необходимо задать ключ компиляции /GR; в противном случае во время выполнения произойдет ошибка доступа.

[2] При этом используется *преобразование в строку* (препроцессорный оператор #) и стандартные макросы **__FILE__** è **__LINE__**. Код приводится далее в этой главе.

при невыполнении условий, включите в программу намеренную ошибку, например, измените условие в первом вызове test_() в DateTest::testOps() на противоположное. Выходные данные точно сообщают, какое условие не прошло проверку, и где случилась ошибка:

```
DateTest failure: (mybday > today) , DateTest.h (line 31)
Test "DateTest":
        Passed: 20      Failed: 1
```

Помимо функции test_() в систему тестирования включены функции succeed_() и fail_() для тех случаев, когда логическое условие не подходит. Эти функции применяются, когда тестируемый класс может генерировать исключения. Для тестирования создается набор входных данных, благодаря которым должно произойти исключение. Если исключение не запускается, значит, в программе произошла ошибка; тогда вызывается функция fail_() для вывода сообщения и обновления счетчика сбоев. Если, как и предполагалось, запускается исключение, вызывается функция succeed() для обновления счетчика удачных проверок.

Рассмотрим конкретный пример. Допустим, спецификация двух конструкторов Date была изменена так, чтобы конструкторы запускали исключение DateError (тип, вложенный по отношению к Date и производный от std::logic_error), если их входные параметры не представляют допустимую дату:

```
Date(const string& s) throw(DateError);
Date(int year, int month, int day) throw(DateError);
```

Теперь функция DateTest::run() может вызвать следующую функцию для тестирования обработки исключений:

```
void testExceptions() {
  try {
    Date d(0,0,0);  // Недопустимая дата
    fail_("Invalid date undetected in Date int ctor");
  } catch (Date::DateError&) {
    succeed_();
  }
  try {
    Date d("");  // Недопустимая дата
    fail_("Invalid date undetected in Date string ctor");
  } catch (Date::DateError&) {
    succeed_();
  }
}
```

В обоих случаях отсутствие исключения является ошибкой. Обратите внимание: логическое условие в программе не проверялось, поэтому при вызове fail_() сообщение приходится передавать вручную.

Комплексы тестов

Реальные проекты обычно состоят из множества классов. Нам понадобится способ группировки тестов, который бы позволял протестировать весь проект нажатием одной кнопки[1]. Класс Suite объединяет тесты в функциональные группы.

[1] Для решения этой задачи также хорошо подходят пакетные файлы и сценарии командного интерпретатора. Класс Suite предназначен для группировки взаимосвязанных тестов средствами C++.

Отдельные объекты Test включаются в контейнер Suite функцией addTest(); также предусмотрена функция addSuite() для включения содержимого целого контейнера. В следующем примере программы из главы 3, использующие класс Test, объединяются в одну группу. Учтите, что этот файл находится в подкаталоге главы 3:

```
//: C03:StringSuite.cpp
//{L} ../TestSuite/Test ../TestSuite/Suite
//{L} TrimTest
// Группировка тестов на примере кода из главы 3
#include <iostream>
#include "../TestSuite/Suite.h"
#include "StringStorage.h"
#include "Sieve.h"
#include "Find.h"
#include "Rparse.h"
#include "TrimTest.h"
#include "CompStr.h"
using namespace std;
using namespace TestSuite;

int main() {
    Suite suite("String Tests");
    suite.addTest(new StringStorageTest);
    suite.addTest(new SieveTest);
    suite.addTest(new FindTest);
    suite.addTest(new RparseTest);
    suite.addTest(new TrimTest);
    suite.addTest(new CompStrTest);
    suite.run();
    long nFail = suite.report();
    suite.free();
    return nFail;
}
/* Вывод:
s1 = 62345
s2 = 12345
Suite "String Tests"
====================
Test "StringStorageTest":
   Passed: 2   Failed: 0
Test "SieveTest":
   Passed: 50  Failed: 0
Test "FindTest":
   Passed: 9   Failed: 0
Test "RparseTest":
   Passed: 8   Failed: 0
Test "TrimTest":
   Passed: 11  Failed: 0
Test "CompStrTest":
   Passed: 8   Failed: 0
*/ ///:~
```

Пять из этих тестов полностью находятся в заголовочных файлах. Тест TrimTest отличается от них: он содержит статические данные, которые должны определяться в файле реализации. Первые две строки вывода содержат трассировочные данные из теста StringStorage. Комплексу тестов должно быть присвоено имя, которое передается в аргументе конструктора. Функция Suite::run() вызывает функцию Test::run() для каждого теста, входящего в комплекс. Практически то же самое происходит в функции Suite::report(), если не считать того, что отчеты отдельных тес-

тов могут направляться в другие выходные потоки. Если тесту, переданному при вызове addSuite(), уже назначен указатель на выходной поток, он сохраняется за этим тестом. В противном случае поток назначается объектом Suite (как и в классе Test, у конструктора Suite имеется необязательный второй аргумент, по умолчанию равный std::cout). Деструктор Suite не выполняет автоматического освобождения указателей на хранящиеся в нем объекты Test, поскольку они не обязаны храниться в куче; эта задача решается функцией Suite::free().

Код TestSuite

Код системы автоматизированного тестирования находится в подкаталоге TestSuite архива примеров. Чтобы использовать его в своих программах, включить путь к каталогу TestSuite в заголовочный файл, скомпонуйте объектные файлы и включите подкаталог TestSuite в перечень путей к библиотекам. Заголовочный файл Test.h выглядит так:

```
//: TestSuite:Test.h
#ifndef TEST_H
#define TEST_H
#include <string>
#include <iostream>
#include <cassert>
using std::string;
using std::ostream;
using std::cout;

// Символ подчеркивания в имени fail_() предотвращает конфликты
// с ios::fail(). Ради единства стиля в имена test_() и succeed_()
// также были включены символы подчеркивания.

#define test_(cond) \
  do_test(cond, #cond, __FILE__, __LINE__)
#define fail_(str) \
  do_fail(str, __FILE__, __LINE__)

namespace TestSuite {

class Test {
  ostream* osptr;
  long nPass;
  long nFail;
  // Запрещенные операции:
  Test(const Test&);
  Test& operator=(const Test&);
};
protected:
  void do_test(bool cond, const string& lbl,
    const char* fname, long lineno);
  void do_fail(const string& lbl,
    const char* fname, long lineno);
public:
  Test(ostream* osptr = &cout) {
    this->osptr = osptr;
    nPass = nFail = 0;
  }
  virtual ~Test() {}
  virtual void run() = 0;
```

```
  long getNumPassed() const { return nPass; }
  long getNumFailed() const { return nFail; }
  const ostream* getStream() const { return osptr; }
  void setStream(ostream* osptr) { this->osptr = osptr; }
  void succeed_() { ++nPass; }
  long report() const;
  virtual void reset() { nPass = nFail = 0; }
};

} // namespace TestSuite
#endif // TEST_H ///:~
```

Класс Test содержит три виртуальные функции:

- виртуальный деструктор;
- функцию reset();
- чисто виртуальную функцию run().

Как объяснялось в первом томе, удаление объекта производного класса через указатель на базовый класс является ошибкой, если у базового класса нет виртуального деструктора. Любой класс, который предполагается использовать в качестве базового (на что, очевидно, указывает присутствие хотя бы одной «обычной» виртуальной функции), должен иметь виртуальный деструктор. По умолчанию Test::reset() обнуляет счетчики успешных и неудачных проверок. Возможно, вы захотите переопределить эту функцию и организовать сброс данных в производном объекте; только не забудьте явно вызвать функцию Test::reset() из переопределенной версии, чтобы сбросить счетчики. Функция Test::run() является чисто виртуальной, поэтому ее переопределение в производном классе обязательно.

Макросы test_() и fail_() могут включать информацию об имени файла и номере строки, полученную от препроцессора. Изначально символы подчеркивания в этих именах отсутствовали, но макрос fail() конфликтовал с ios::fail() и вызывал ошибки компиляции.

Далее приведена реализация остальных функций Test:

```
//: TestSuite:Test.cpp {O}
#include "Test.h"
#include <iostream>
#include <typeinfo>  // Примечание: для Visual C++ необходим ключ /GR
using namespace std;
using namespace TestSuite;

void Test::do_test(bool cond,
  const std::string& lbl, const char* fname,
  long lineno) {
  if (!cond)
    do_fail(lbl, fname, lineno);
  else
    succeed_();
}

void Test::do_fail(const std::string& lbl,
  const char* fname, long lineno) {
  ++nFail;
  if (osptr) {
    *osptr << typeid(*this).name()
           << "failure: (" << lbl << ") , "
```

```
            << fname
            << " (line " << lineno << ")" << endl;
  }
}

long Test::report() const {
  if (osptr) {
    *osptr << "Test \"" << typeid(*this).name()
           << "\":\n\tPassed: " << nPass
           << "\tFailed: " << nFail
           << endl;
  }
  return nFail;
} ///:~
```

Класс Test хранит информацию о количестве успешных и неудачных проверок, а также выходной поток, в который функция Test::report() должна выводить результаты. Макросы test_() и fail_() получают текущие имя файла и номер строки от препроцессора; имя файла передается do_test(), а номер строки — do_fail(). Эти функции занимаются непосредственным выводом сообщений и обновлением счетчиков. Трудно представить, кому и зачем могло бы понадобиться копировать и присваивать объекты Test, поэтому мы запретили эти операции. Для этого их прототипы объявлены закрытыми, а тела функций не определяются.

```
//: TestSuite:Suite.h
#ifndef SUITE_H
#define SUITE_H
#include <vector>
#include <stdexcept>
#include "../TestSuite/Test.h"
using std::vector;
using std::logic_error;

namespace TestSuite {

class TestSuiteError : public logic_error {
public:
  TestSuiteError(const string& s = "")
    : logic_error(s) {}
};

class Suite {
  string name;
  ostream* osptr;
  vector<Test*> tests;
  void reset();
  // Запрещенные операции:
  Suite(const Suite&);
  Suite& operator=(const Suite&);
public:
  Suite(const string& name, ostream* osptr = &cout)
    : name(name) { this->osptr = osptr; }
  string getName() const { return name; }
  long getNumPassed() const;
  long getNumFailed() const;
  const ostream* getStream() const { return osptr; }
  void setStream(ostream* osptr) { this->osptr = osptr; }
  void addTest(Test* t) throw (TestSuiteError);
  void addSuite(const Suite&);
```

```
  void run();  // Многократно вызывает Test::run()
  long report() const;
  void free();  // Уничтожает объекты тестов
};

} // namespace TestSuite
#endif // SUITE_H ///:~
```

Класс Suite хранит указатели на объекты Test в векторе. Обратите внимание на спецификацию исключений в объявлении функции addTest(). При включении нового объекта теста в контейнер функция Suite::addTest() убеждается в том, что переданный указатель отличен от null; в противном случае генерируется исключение TestSuiteError. Поскольку это делает невозможным добавление null-указателя в контейнер, addSuite() проверяет это условие в каждой из своих проверок; то же происходит и в других функциях, перебирающих вектор с объектами тестов (см. реализацию ниже). Как и в классе Test, операции копирования и присваивания в классе Suite запрещены.

```
//: TestSuite:Suite.cpp {O}
#include "Suite.h"
#include <iostream>
#include <cassert>
#include <cstddef>
using namespace std;
using namespace TestSuite;

void Suite::addTest(Test* t) throw(TestSuiteError) {
  // Проверка действительности теста и наличия выходного потока:
  if (t == 0)
    throw TestSuiteError("Null test in Suite::addTest");
  else if (osptr && !t->getStream())
    t->setStream(osptr);
  tests.push_back(t);
  t->reset();
}

void Suite::addSuite(const Suite& s) {
  for (size_t i = 0; i < s.tests.size(); ++i) {
    assert(tests[i]);
    addTest(s.tests[i]);
  }
}

void Suite::free() {
  for (size_t i = 0; i < tests.size(); ++i) {
    delete tests[i];
    tests[i] = 0;
  }
}

void Suite::run() {
  reset();
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->run();
  }
}

long Suite::report() const {
  if (osptr) {
    long totFail = 0;
    *osptr << "Suite \"" << name
           << "\"\n=======";
```

```
    size_t i;
    for (i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=" << endl;
    for (i = 0; i < tests.size(); ++i) {
      assert(tests[i]);
      totFail += tests[i]->report();
    }
    *osptr << "=======";
    for (i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=" << endl;
    return totFail;
  }
  else
    return getNumFailed();
}

long Suite::getNumPassed() const {
  long totPass = 0;
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totPass += tests[i]->getNumPassed();
  }
  return totPass;
}

long Suite::getNumFailed() const {
  long totFail = 0;
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totFail += tests[i]->getNumFailed();
  }
  return totFail;
}

void Suite::reset() {
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->reset();
  }
} ///:~
```

Мы будем использовать систему TestSuite там, где это будет уместно.

Методика отладки

Утверждения, о которых рассказывалось в начале этой главы, приносят огромную пользу при отладке; они помогают обнаружить логические ошибки до того, как те принесут реальный вред. В этом разделе собраны полезные советы и приемы, применяемые в процессе отладки.

Трассировочные макросы

Иногда бывает удобно выводить код каждой выполняемой команды в поток cout или в трассировочный файл. Вот как выглядит препроцессорный макрос для решения этой задачи:

```
#define TRACE(ARG) cout << #ARG << endl; ARG
```

Остается лишь заключить трассируемые команды в этот макрос. Впрочем, при этом могут возникнуть проблемы. Для примера возьмем следующий цикл:

```
for(int i = 0; i < 100; i++)
  cout << i << endl;
```

Каждая строка заключается в макрос TRACE():

```
TRACE(for(int i = 0; i < 100; i++))
TRACE(  cout << i << endl);
```

После расширения будет получен следующий результат:

```
cout << "for(int i = 0; i < 100; i++)" << endl;
for(int i = 0; i < 100; i++)
  cout << "cout << i <<  endl;" << endl;
cout << i <<  endl;
```

Не совсем то, что требовалось. А значит, при использовании этого приема необходима осторожность.

Ниже приведена другая разновидность макроса TRACE:

```
#define D(a) cout << #a "=[" << a << "]" << endl;
```

При вызове макроса D передается выражение, которое требуется вывести. Макрос выводит это выражение, а за ним — вычисленное значение (предполагается, что для типа результата существует перегруженный оператор <<). Например, вызов может выглядеть так: D(a+b). Используйте макрос D для проверки промежуточных результатов.

Эти два макроса реализуют две основные операции, выполняемые в процессе отладки: трассировку программного кода и вывод значений. Хороший отладчик значительно повышает эффективность труда программиста, но иногда отладчик недоступен или им может быть неудобно пользоваться в конкретной ситуации. Приемы, описанные в этом разделе, работают всегда.

Трассировочный файл

ВНИМАНИЕ

Программный код, приведенный в этом и следующем разделах, не соответствует официальному стандартному языку C++. В частности, мы переопределяем cout и new с применением макросов; малейшая неосторожность приведет к удивительным последствиям. Тем не менее примеры работали на всех компиляторах, используемых нами, и поставляли полезную информацию. Это единственное место в книге, где мы отклоняемся от священных канонов «программирования по Стандарту». Учтите, что для работы этих приемов необходимо объявление using, чтобы имя cout не снабжалось префиксом пространства имен; иначе говоря, std::cout работать не будет.

Следующая программа создает трассировочный файл и отправляет в него весь вывод, который в обычных условиях передается в cout. Все, что для этого нужно, — вставить в программу директиву #define TRACEON и включить заголовочный файл (конечно, с таким же успехом можно просто включить две ключевые строки прямо в файл):

```
//: C03:Trace.h
// Создание трассировочного файла
#ifndef TRACE_H
#define TRACE_H
#include <fstream>

#ifdef TRACEON
ofstream TRACEFILE__("TRACE.OUT");
```

```
#define cout TRACEFILE__
#endif

#endif // TRACE_H ///:~
```

Простая тестовая программа для этого файла выглядит так:

```
//: C03:Tracetst.cpp {-bor}
#include <iostream>
#include <fstream>
#include "../require.h"
using namespace std;

#define TRACEON
#include "Trace.h"

int main() {
  ifstream f("Tracetst.cpp");
  assure(f, "Tracetst.cpp");
  cout << f.rdbuf(); // Вывод содержимого файла в трассировочный файл
} ///:~
```

Поскольку Trace.h заменяет имя cout на уровне исходного текста, все команды вывода в cout в вашей программе теперь направляют информацию в трассировочный файл. В частности, это позволяет сохранить выходные данные программы в файле, если в операционной системе отсутствуют нормальные средства перенаправления вывода.

Поиск утечки памяти

Следующие тривиальные приемы отладки рассматривались в первом томе книги.

- Чтобы организовать автоматическую проверку индекса для массива любого типа, воспользуйтесь шаблоном Array из примера C16:Array3.cpp. Когда все будет готово к построению окончательной версии программы, проверку можно отключить для повышения эффективности (впрочем, остаются проблемы с получением указателя на массив).
- Проверьте невиртуальные деструкторы в базовых классах.

К числу стандартных проблем с памятью относятся ошибочный вызов оператора delete для памяти, не находящейся в куче; повторное удаление памяти в куче; и самое частое — неудаленный указатель. В этом разделе рассматривается система, которая помогает отслеживать подобные ошибки.

Дополнительное предупреждение (к тому, о чем говорилось в предыдущем разделе): наш способ перегрузки оператора new может не работать на некоторых платформах. Кроме того, он функционирует только в программах, не вызывающих операторную функцию operator new() напрямую. В этой книге мы стремились представлять только те программы, которые полностью соответствуют стандартному языку C++, однако в данном случае было сделано исключение по следующим причинам.

- Хотя эта методика является «технически незаконной», она работает во многих компиляторах[1].
- Попутно демонстрируются некоторые полезные приемы.

[1] Наш главный технический рецензент, Пит Бекер (Pete Becker) из Dinkumware. Ltd, указал на недопустимость замены ключевых слов C++ при помощи макросов. Его замечание выглядело так: «Это грязный фокус. Такие грязные фокусы иногда помогают разобраться в том, почему не работает программа, поэтому полностью отказываться от них не стоит, но не используйте их в окончательной версии программы».

Чтобы использовать систему проверки памяти, включите заголовочный файл MemCheck.h, скомпонуйте файл MemCheck.obj со своим приложением для перехвата всех вызовов new и delete и активизируйте трассировку памяти макросом MEM_ON() (см. далее). Информация обо всех операциях выделения и освобождения памяти будет направляться в стандартный выходной поток (через stdout). При использовании этой системы для всех вызовов new сохраняется имя файла и номер строки, содержащей вызов. Для этого мы задействуем *синтаксис размещения* оператора new[1]. Хотя обычно синтаксис размещения применяется в тех случаях, когда объекты должны находиться по определенному адресу памяти, он также позволяет создать функцию operator new() с любым количеством аргументов. В следующем примере дополнительные аргументы используются для сохранения результатов макросов __FILE__ и __LINE__ при каждом вызове new:

```
//: C02:MemCheck.h
#ifndef MEMCHECK_H
#define MEMCHECK_H
#include <cstddef>  // для size_t

// Перехват оператора new (в версиях для скаляров и массивов)
void* operator new(std::size_t, const char*, long);
void* operator new[](std::size_t, const char*, long);
#define new new (__FILE__, __LINE__)

extern bool traceFlag;
#define TRACE_ON() traceFlag = true
#define TRACE_OFF() traceFlag = false

extern bool activeFlag;
#define MEM_ON() activeFlag = true
#define MEM_OFF() activeFlag = false

#endif // MEMCHECK_H ///:~
```

Этот файл должен включаться во все исходные файлы, для которых требуется отслеживать операции с кучей, но включаться он должен обязательно *последним* (после всех остальных директив #include). Большинство заголовочных файлов стандартной библиотеки содержит шаблоны, а поскольку многие компиляторы используют при компиляции шаблонов *модель с включением*, макрос, заменяющий new в MemCheck.h, узурпирует все вхождения оператора new в исходном коде библиотеки (что, скорее всего, приведет к ошибкам компиляции). Кроме того, нас интересуют наши собственные ошибки в нашем коде, а не в библиотеках.

В следующем файле, содержащем реализацию системы отслеживания операций с памятью, весь ввод-вывод осуществляется стандартными средствами C (вместо потоков C++). В принципе, это должно быть несущественно, поскольку мы не мешаем потокам ввода-вывода работать с кучей, но некоторые компиляторы были против. С другой стороны, версия <cstdio> никаких нареканий не вызывала.

```
//: C02:MemCheck.cpp {O}
#include <cstdio>
#include <cstdlib>
#include <cassert>
#include <cstddef>
using namespace std;
```

[1] Спасибо Регу Чарни (Reg Charney) из Комитета по стандартизации C++, предложившему этот прием.

```
#undef new

// Глобальные флаги, устанавливаемые макросами MemCheck.h
bool traceFlag = true;
bool activeFlag = false;

namespace {

// Тип записи в карте памяти
struct Info {
  void* ptr;
  const char* file;
  long line;
};

// Данные карты памяти
const size_t MAXPTRS = 10000u;
Info memMap[MAXPTRS];
size_t nptrs = 0;

// Поиск адреса в карте
int findPtr(void* p) {
  for (int i = 0; i < nptrs; ++i)
    if (memMap[i].ptr == p)
      return i;
  return -1;
}

void delPtr(void* p) {
  int pos = findPtr(p);
  assert(p >= 0);
  // Удаление указателя из карты
  for (size_t i = pos; i < nptrs-1; ++i)
    memMap[i] = memMap[i+1];
  --nptrs;
}

// Фиктивный тип для статического деструктора
struct Sentinel {
  ~Sentinel() {
    if (nptrs > 0) {
      printf("Leaked memory at:\n");
      for (size_t i = 0; i < nptrs; ++i)
        printf("\t%p (file: %s, line %ld)\n",
          memMap[i].ptr, memMap[i].file, memMap[i].line);
    }
    else
      printf("No user memory leaks!\n");
  }
};

// Статический фиктивный объект
Sentinel s;

} // Конец анонимного пространства имен

// Перегрузка скалярной версии new
void*
operator new(size_t siz, const char* file, long line) {
  void* p = malloc(siz);
  if (activeFlag) {
```

```
    if (nptrs == MAXPTRS) {
      printf("memory map too small (increase MAXPTRS)\n");
      exit(1);
    }
    memMap[nptrs].ptr = p;
    memMap[nptrs].file = file;
    memMap[nptrs].line = line;
    ++nptrs;
  }
  if (traceFlag) {
    printf("Allocated %u bytes at address %p ", siz, p);
    printf("(file: %s, line: %ld)\n", file, line);
  }
  return p;
}

// Перегрузка версии new для массивов
void* operator new[](size_t siz, const char* file, long line) {
  return operator new(siz, file, line);
}

// Перегрузка скалярной версии delete
void operator delete(void* p) {
  if (findPtr(p) >= 0) {
    free(p);
    assert(nptrs > 0);
    delPtr(p);
    if (traceFlag)
      printf("Deleted memory at address %p\n", p);
  }
  else if (!p && activeFlag)
    printf("Attempt to delete unknown pointer: %p\n", p);
}

// Перегрузка версии delete для массива
void operator delete[](void* p) {
  operator delete(p);
} ///:~
```

Логические флаги traceFlag и activeFlag являются глобальными, поэтому их состояние может изменяться в программе макросами TRACE_ON(),TRACE_OFF(), MEM_ON() и MEM_OFF(). Обычно в пару макросов MEM_ON()—MEM_OFF() заключается весь код main(), чтобы операции с памятью отслеживались всегда. Трассировка, выводящая информацию о работе заменителей operator new() и operator delete(), включена по умолчанию, но ее можно отключить макросом TRACE_OFF(). В любом случае итоговый результат выводится всегда (см. примеры тестов далее в этой главе).

Система MemCheck отслеживает операции с памятью, для чего все адреса, выделенные оператором new, сохраняются в массиве структур Info. В массиве также хранятся имена файлов и номера строк, в которых находился вызов new. Чтобы предотвратить конфликты с именами глобального пространства имен, мы стараемся сохранить как можно больше информации в анонимном пространстве. Класс Sentinel существует только для вызова деструктора статического объекта при завершении программы. Деструктор просматривает memMap и смотрит, остались ли в карте памяти неудаленные указатели (что является признаком утечки памяти).

Наша функция operator new() получает память с помощью функции malloc(), после чего включает указатель и связанные с ним данные в memMap. Функция operator delete() «отменяет» эти операции, вызывая free() и уменьшая nptrs, но сначала она проверяет,

присутствует ли заданный указатель в карте памяти. Если указатель не найден, значит, вы пытаетесь освободить блок, либо отсутствующий в куче, либо уже освобожденный ранее и удаленный из кучи. Переменная activeFlag играет важную роль, потому что мы не хотим обрабатывать операции освобождения памяти, обусловленные завершением работы системы. При вызове MEM_OFF() в конце программы флаг activeFlag становится равным false, и последующие вызовы delete игнорируются (конечно, в обычной программе так делать нельзя, но мы занимаемся поиском утечки памяти в *вашем* коде, а не отладкой библиотеки). Простоты ради, вся работа версий операторов new и delete для массивов перепоручается их скалярным аналогам.

Ниже приведен простой тест, в котором используется система MemCheck:

```
//: C02:MemTest.cpp
//{L} MemCheck
// Test of MemCheck system
#include <iostream>
#include <vector>
#include <cstring>
#include "MemCheck.h"   // Должен включаться последним!
using namespace std;
class Foo {
  char* s;
public:
  Foo(const char*s ) {
    this->s = new char[strlen(s) + 1];
    strcpy(this->s, s);
  }
  ~Foo() {
    delete [] s;
  }
};

int main() {
  MEM_ON();
  cout << "hello\n";
  int* p = new int;
  delete p;
  int* q = new int[3];
  delete [] q;
  int* r;
  delete r;
  vector<int> v;
  v.push_back(1);
  Foo s("goodbye");
  MEM_OFF();
} ///:~
```

Этот пример доказывает, что система MemCheck может использоваться с потоками, стандартными контейнерами и классами, выделяющими память в конструкторах. Память по указателям p и q выделяется и освобождается без проблем, но r не является действительным указателем на память в куче, поэтому в выходных данных включается сообщение об ошибке освобождения неизвестного указателя:

```
hello
Allocated 4 bytes at address 0xa010778 (file: memtest.cpp, line: 25)
Deleted memory at address 0xa010778
Allocated 12 bytes at address 0xa010778 (file: memtest.cpp, line: 27)
Deleted memory at address 0xa010778
Attempt to delete unknown pointer: 0x1
Allocated 8 bytes at address 0xa0108c0 (file: memtest.cpp, line: 14)
```

```
Deleted memory at address 0xa0108c0
No user memory leaks!
```

Из-за вызова MEM_OFF() последующие вызовы функций operator delete() из объектов vector и ostream не обрабатываются. Теоретически еще возможны отдельные вызовы delete из-за перераспределений памяти, выполненных контейнерами.

Если вызвать TRACE_OFF() в начале программы, результат будет таким:

```
hello
Attempt to delete unknown pointer: 0x1
No user memory leaks!
```

Итоги

Тщательное предварительное планирование помогает решить многие проблемы, возникающие при программировании. Даже те программисты, которые не привыкли постоянно пользоваться макросом assert(), все равно применяют некие «воображаемые» аналоги утверждений в своих циклах и функциях. Макрос assert() упрощает поиск ошибок, а программы становятся более понятными. Однако следует помнить, что утверждения должны применяться только для проверки инвариантов, но не для обработки ошибок времени выполнения.

Ничто не вселяет в программиста такое спокойствие духа, как тщательно протестированная программа. Если раньше тестирование казалось вам мучительной и неприятной процедурой, воспользуйтесь средствами автоматизации, наподобие представленных в этой главе, и сделайте его частью своей повседневной работы. Это пойдет на пользу — как вам, так и пользователям ваших программ!

Упражнения

1. Напишите тестовую программу, использующую систему TestSuite, для стандартного класса vector. Программа должна проверять работу следующих функций целочисленного вектора: push_back() (присоединяет элемент в конец вектора), front() (возвращает первый элемент вектора), back() (возвращает последний элемент вектора), pop_back() (удаляет последний элемент, не возвращая его), at() (возвращает элемент с заданным индексом) и size() (возвращает количество элементов). Убедитесь в том, что для недопустимых индексов функция vector::at() генерирует исключение std::out_of_range.
2. Предположим, вы разрабатываете класс Rational для представления рациональных чисел (дробей). Объекты Rational всегда хранятся в нормализованном виде, а нулевой знаменатель является ошибкой. Примерный интерфейс класса Rational выглядит так:

```
//: C02:Rational.h {-xo}
#ifndef RATIONAL_H
#define RATIONAL_H
#include <iosfwd>

class Rational {
public:
    Rational(int numerator = 0, int denominator = 1);
    Rational operator-() const;
```

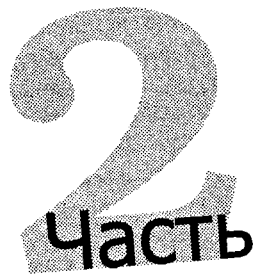
```
    friend Rational operator+(const Rational&,
                              const Rational&);
    friend Rational operator-(const Rational&,
                              const Rational&);
    friend Rational operator*(const Rational&,
                              const Rational&);
    friend Rational operator/(const Rational&,
                             const Rational&);
    friend ostream& operator<<(ostream&,
                               const Rational&);
    friend istream& operator>>(istream&, Rational&);
    Rational& operator+=(const Rational&);
    Rational& operator-=(const Rational&);
    Rational& operator*=(const Rational&);
    Rational& operator/=(const Rational&);
    friend bool operator<(const Rational&,
                          const Rational&);
    friend bool operator>(const Rational&,
                          const Rational&);
    friend bool operator<=(const Rational&,
                           const Rational&);
    friend bool operator>=(const Rational&,
                           const Rational&);
    friend bool operator==(const Rational&,
                           const Rational&);
    friend bool operator!=(const Rational&,
                           const Rational&);
};
#endif // RATIONAL_H ///:~
```

Напишите полную спецификацию этого класса с предусловиями, постусловиями и спецификациями исключений.

3. Напишите тест с использованием системы TestSuite, который бы досконально тестировал спецификации из предыдущего примера (в том числе и исключения).
4. Реализуйте класс Rational так, чтобы все тесты из предыдущего упражнения завершались успешно. Используйте утверждения только для инвариантов.
5. Приведенный ниже файл BuggedSearch.cpp содержит функцию бинарного поиска what в интервале [beg,end). Алгоритм содержит ряд ошибок. Отладьте функцию поиска при помощи средств трассировки, описанных в этой главе.

```
// C02:BuggedSearch.cpp {-xo}
//{L} ../TestSuite/Test
#include "../TestSuite/Test.h"
#include <cstdlib>
#include <ctime>
#include <cassert>
#include <fstream>
using namespace std;

// Функция содержит ошибки
int* binarySearch(int* beg, int* end, int what) {
  while(end - beg != 1) {
    if(*beg == what) return beg;
    int mid = (end - beg) / 2;
    if(what <= beg[mid]) end = beg + mid;
    else beg = beg + mid;
  }
```

```
  return 0;
}

class BinarySearchTest : public TestSuite::Test {
  enum { sz = 10 };
  int* data;
  int max; // Наибольшее число
  int current; // Текущее число, не содержащееся в массиве.
               // Используется в notContained()
  // Поиск следующего числа, не содержащегося в массиве
  int notContained() {
    while(data[current] + 1 == data[current + 1])
     current++;
    if(current >= sz) return max + 1;
    int retValue = data[current++] + 1;
    return retValue;
  }
  void setData() {
    data = new int[sz];
    assert(!max);
    // Набор входных данных. Пропускаем некоторые значения
    // как для четных, так и для нечетных индексов.
    for(int i = 0; i < sz;
        rand() % 2 == 0 ? max += 1 : max += 2)
      data[i++] = max;
  }
  void testInBound() {
    // Проверка для четных и нечетных позиций,
    // присутствующих и отсутствующих в массиве.
    for(int i = sz; --i >=0;)
      test_(binarySearch(data, data + sz, data[i]));
    for(int i = notContained(); i < max;
            i = notContained())
      test_(!binarySearch(data, data + sz, i));
  }
  void testOutBounds() {
    // Проверка нижних значений
    for(int i = data[0]; --i > data[0] - 100;)
      test_(!binarySearch(data, data + sz, i));
    // Проверка верхних значений
    for(int i = data[sz - 1];
        ++i < data[sz -1] + 100;)
      test_(!binarySearch(data, data + sz, i));
  }
public:
  BinarySearchTest() {
    max = current = 0;
  }
  void run() {
    srand(time(0));
    setData();
    testInBound();
    testOutBounds();
    delete [] data;
  }
};

int main() {
   srand(time(0));
   BinarySearchTest t;
   t.run();
   return t.report();
} ///:~
```



Стандартная библиотека C++

Стандартный язык C++ не только включает все стандартные библиотеки C (с небольшими дополнениями и изменениями, обеспечивающими безопасность типов), но и дополняет их собственными библиотеками. Эти библиотеки гораздо мощнее своих прототипов из стандартного языка C. При переходе на них достигается практически такой же выигрыш, как при переходе с C на C++.

В этой части книги мы подробно рассмотрим важнейшие компоненты стандартной библиотеки C++. Самым полным (но также наименее понятным) источником информации обо всех библиотеках является сам стандарт[1]. Мы стремились представить читателю по возможности более широкую подборку описаний и примеров, которые послужили бы отправной точкой для решения любых задач, требующих применения стандартных библиотек. Впрочем, некоторые темы и приемы на практике применяются редко и в книге не рассматриваются. Если вы не найдете их описаний в этих главах, обратитесь к специализированной литературе; эта книга не заменяет ее, а лишь дополняет. В частности, мы надеемся, что после знакомства с материалом этой части вам будет гораздо проще понять другие книги.

Учтите, что в главах этой части не приводится подробная документация по всем функциям и классам стандартной библиотеки C++. За полной документацией обращайтесь к другим источникам, прежде всего — «Справочнику по стандартным библиотекам C/C++» П. Дж. Плаугера (P. J. Plauger) по адресу http://www.dinkumware.com. Это превосходный сборник электронной документации по стандартным библиотекам в формате HTML; вы можете просматривать его в браузере при возникновении любых вопросов. Справочник можно изучать непосредственно в режиме подключения или приобрести его для локального просмотра. Он содержит документацию по библиотекам как C, так и C++. Электронная документация хороша не только своей доступностью, но и тем, что в ней удобно производить поиск.

В главе 3 (первой в этой части) представлен стандартный класс C++ string — мощный инструмент, упрощающий многие повседневные задачи обработки текста. Большинство операций с символьными строками, для которых в C приходилось писать несколько строк программного кода, выполняются в классе string вызовом одной функции.

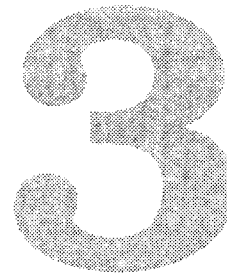
Глава 4 посвящена библиотеке потоков ввода-вывода. Классы этой библиотеки предназначены для ввода-вывода файлов, строк и операций с системной консолью.

Хотя глава 5 не имеет прямого отношения к библиотеке, она необходима для лучшего понимания двух следующих глав. В главе 6 рассматриваются общие алгоритмы стандартной библиотеки C++. Реализация на базе шаблонов позволяет применять эти алгоритмы к *последовательностям* объектов. Глава 7 посвящена стандартным контейнерам и их итераторам. Мы начинаем с алгоритмов, потому что для их объяснения достаточно массивов и контейнера vector (а с этим контейнером мы начали работать еще в начале первого тома). Кроме того, вполне естественно использовать с контейнерами стандартные алгоритмы, поэтому знакомство разумнее начать именно с алгоритмов.

---

[1] Самый известный справочник по стандартной библиотеке C++ — Джосьютис Н. C++ Стандартная библиотека. Для профессионалов. СПб.: Питер, 2003.

Строки



Обработка строк в символьных массивах отнимала массу времени у программистов С. При работе с символьным массивом программисту приходилось различать статические строки в кавычках и массивы, созданные в стеке и в куче, а также помнить, что в одних случаях передается char*, а в других приходится копировать целый массив.

С учетом того, что операции со строками так широко распространены, символьные массивы превращались в источник сплошных недоразумений и ошибок. Однако несмотря на это, создание собственных строковых классов в течение долгих лет оставалось распространенным упражнением для начинающих программистов C++. Класс string стандартной библиотеки C++ раз и навсегда решает проблему работы с символьными массивами, он правильно выделяет память даже при конструировании копий и присваивании. Вам просто не придется думать об этих мелочах.

В этой главе[1] рассматривается класс string стандартного языка C++. Сначала мы разберемся, что же представляют собой строки C++ и чем они отличаются от традиционных символьных массивов С. Вы узнаете, какие операции выполняются с объектами string, и как класс string обеспечивает применение различных кодировок и преобразований строковых данных.

Обработка текста относится к числу старейших задач программирования. Вполне понятно, что класс string в значительной степени основан на идеях и терминологии, традиционно использовавшейся в С и других языках. Это обстоятельство придаст вам уверенности во время изучения string. Какую бы идиому программирования вы ни выбрали, основные операции с классом string делятся на три категории:

- создание или модификация последовательности символов, хранящейся в string;
- проверка наличия или отсутствия элементов в string;
- преобразование между различными схемами представления символов в string.

---

[1] Часть материала этой главы была подготовлена Нэнси Николайсен (Nancy Nicolaisen).

Вы увидите, как каждая из этих операций выполняется средствами строковых объектов C++.

Что такое строка?

В языке C строка представляет собой массив символов, последним элементом которого всегда является двоичный ноль (часто называемый *нуль-терминатором*). Между строками C++ и их прототипами C существуют заметные различия. Первое и самое важное из них состоит в том, что строковые объекты C++ скрывают физическое представление содержащихся в них символов. Вам не придется беспокоиться о размерах массива или о нуль-терминаторах. Объект string также содержит служебную информацию о размере и местонахождении буфера данных. Говоря точнее, строковый объект C++ знает свой начальный адрес в памяти, свое содержимое, свою длину в символах, а также максимальную длину в символах, до которой он может увеличиться без увеличения внутреннего буфера данных. Строки C++ существенно снижают вероятность самых распространенных и опасных ошибок программирования C: выхода за границы массива, попытки обращения к массиву через неинициализированный или ошибочный указатель, появление «висячих» указателей после освобождения блока памяти, в котором ранее хранился массив.

Стандарт C++ не определяет конкретную структуру памяти для строкового класса. Предполагается, что эта архитектура является достаточно гибкой, чтобы по-разному реализовываться разработчиками компиляторов, но при этом гарантировать предсказуемое поведение для пользователей. В частности, в стандарте не определяются точные условия выделения памяти для хранения данных. Правила сформулированы таким образом, что они *допускают* реализацию на базе подсчета ссылок, но не делают ее обязательной. Впрочем, независимо от того, используется ли в реализации подсчет ссылок или нет, семантика должна сохраняться. Так, в языке C каждый символьный массив всегда занимает уникальный физический блок памяти. В C++ отдельные объекты string могут занимать или не занимать уникальные физические блоки памяти, но даже если хранение лишних копий данных предотвращается благодаря подсчету ссылок, с точки зрения программиста отдельные объекты должны работать так, словно каждый из них хранится в отдельном блоке. Пример:

```
//: C03:StringStorage.h
#ifndef STRINGSTORAGE_H
#define STRINGSTORAGE_H
#include <iostream>
#include <string>
#include "../TestSuite/Test.h"
using std::cout;
using std::endl;
using std::string;

class StringStorageTest : public TestSuite::Test {
public:
  void run() {
    string s1("12345");
    // Первая строка может быть скопирована во вторую
    // или копирование может имитироваться подсчетом ссылок.
```

```
    string s2 = s1;
    test_(s1 == s2);
    // В любом случае эта команда должна изменять ТОЛЬКО s1
    s1[0] = '6';
    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    test_(s1 != s2);
  }
};
#endif // STRINGSTORAGE_H ///:~

//: C03:StringStorage.cpp
//{L} ../TestSuite/Test
#include "StringStorage.h"

int main() {
  StringStorageTest t;
  t.run();
  return t.report();
} ///:~
```

Говорят, что в реализации, при которой уникальные копии создаются только при модификации строк, используется стратегия *копирования при записи*. Такое решение экономит время и память в тех случаях, когда строки только передаются по значению (а также в других ситуациях с доступом только для чтения).

Пользователю класса string должно быть безразлично, основана реализация библиотеки на подсчете ссылок или нет. К сожалению, это правило выполняется далеко не всегда. В многопоточных программах практически невозможно безопасно использовать реализацию с подсчетом ссылок[1].

Создание и инициализация строк C++

Создание и инициализация строк — вполне очевидные операции, обладающие достаточно гибкими возможностями. В приведенном ниже примере SmallString.cpp первая строка imBlank объявляется, но не содержит исходного значения. В отличие от символьных массивов C, которые до момента инициализации содержат случайный и бессмысленный набор битов, imBlank содержит полезную информацию. Этот объект string инициализируется «пустой строкой», он может правильно сообщить о своей нулевой длине и отсутствии элементов через функции класса.

Следующая строка heyMom инициализируется литералом "Where are my socks?". При такой форме инициализации конструктору string передается символьный массив, заключенный в кавычки. С другой стороны, standardReply инициализируется простым присваиванием. Последняя строка группы, useThisOneAgain, инициализируется существующим строковым объектом C++. Другими словами, этот пример демонстрирует следующие способы инициализации объектов string:

- создание пустого объекта string (инициализация объекта символьными данными откладывается на будущее);
- инициализация объекта string с передачей конструктору литерала — символьного массива, заключенного в кавычки;

[1] Многопоточное программирование будет рассматриваться в главе 10.

- инициализация объекта string с помощью знака равенства (=);
- использование объекта string для инициализации другого объекта.

```
//: C03:SmallString.cpp
#include <string>
using namespace std;

int main() {
  string imBlank;
  string heyMom("Where are my socks?");
  string standardReply = "Beamed into deep "
    "space on wide angle dispersion?";
  string useThisOneAgain(standardReply);
} ///:~
```

Таковы простейшие формы инициализации объектов string; их разновидности обладают большей гибкостью и лучше поддаются контролю. В частности, возможно:

- использование подмножества символьного массива C или строкового объекта C++;
- объединение нескольких источников инициализационных данных оператором +;
- выделение подстроки функцией substr() объекта string.

Следующая программа демонстрирует перечисленные возможности.

```
//: C03:SmallString2.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1
    ("What is the sound of one clam napping?");
  string s2
    ("Anything worth doing is worth overdoing.");
  string s3("I saw Elvis in a UFO");
  // Копирование первых 8 символов
  string s4(s1, 0, 8);
  cout << s4 << endl;
  // Копирование 6 символов из середины источника
  string s5(s2, 15, 6);
  cout << s5 << endl;
  // Копирование из середины в конец
  string s6(s3, 6, 15);
  cout << s6 << endl;
  // Комбинированное копирование
  string quoteMe = s4 + "that" +
  // substr() копирует 10 символов начиная с элемента 20
  s1.substr(20, 10) + s5 +
  // substr() копирует 100 символов или остаток до конца
  // строки, начиная с элемента 5.
  "with" + s3.substr(5, 100) +
  // Функция substr() также может копировать отдельные символы
  s1.substr(37, 1);
  cout << quoteMe << endl;
} ///:~
```

В первом аргументе функции substr() класса string передается начальная позиция, а во втором — длина подстроки в символах. У обоих аргументов имеются значения по умолчанию. Функция substr() с пустым списком аргументов возвращает копию всего объекта string; это удобный способ копирования строк C++.

Программа выводит следующий результат:

```
What is
doing
Elvis in a UFO
What is that one clam doing with Elvis in a UFO?
```

Обратите внимание на последнюю строку примера. C++ позволяет объединять разные способы инициализации string в одной команде, что очень удобно. Также стоит заметить, что последний инициализатор копирует *всего один символ* из исходного объекта string.

Другая, более изощренная, методика инициализации основана на применении строковых итераторов string::begin() и string::end(). Строка интерпретируется как объект-*контейнер*, в котором начало и конец последовательности символов обозначаются при помощи *итераторов* (мы уже встречались с контейнерами на примере векторов, а в главе 7 будут рассмотрены и другие их разновидности). В этом варианте конструктору string передаются два итератора, и он копирует символы из одного объекта string в другой:

```
//: C03:StringIterators.cpp
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

int main() {
  string source("xxx");
  string s(source.begin(), source.end());
  assert(s == source);
} ///:~
```

Операторы с итераторами не ограничиваются вызовами begin() и end(). Итераторы можно увеличивать и уменьшать, а также прибавлять к ним целочисленные смещения, чтобы извлечь подмножество символов из исходной строки.

Строки C++ *не могут* инициализироваться одиночными символами, ASCII-кодами или другими целочисленными значениями. Впрочем, строка может инициализироваться несколькими экземплярами одного символа:

```
//: C03:UhOh.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
  // Ошибка: инициализация одиночным символом недопустима
  //! string nothingDoing1('a');
  // Ошибка: инициализация целочисленными кодами недопустима
  //! string nothingDoing2(0x37);
  // Следующий вариант допустим:
  string okay(5, 'a');
  assert(okay == string("aaaaa"));
} ///:~
```

Первый аргумент определяет количество экземпляров второго аргумента в строке. Второй аргумент может быть только одиночным символом (char), но не символьным массивом.

Операции со строками

Каждому программисту с опытом работы на C хорошо знакомы функции записи, поиска, модификации и копирования массивов char. У функций стандартной библиотеки C имеются два недостатка. Во-первых, они неформально делятся на две категории: «простые» функции и те, которым при вызове необходимо передать количество символов, участвующих в выполняемой операции. Список библиотечных функций C для работы с символьными массивами поражает неопытного пользователя: перед ним оказывается длинный перечень загадочных и неудобопроизносимых имен функций. Типы и количество аргументов у этих функций более или менее выдержаны, но для их правильного использования необходимо тщательно следить за именами и передачей параметров.

Однако у функций стандартной библиотеки C есть второй недостаток: все они подразумевают, что символьный массив завершается нуль-терминатором. Если нуль-символ по недосмотру или ошибке будет пропущен или перезаписан, ничто не помешает функциям символьных массивов C выйти за пределы строки. Результаты обычно оказываются катастрофическими.

Строковые объекты C++ существенно превосходят свои прототипы C по удобству и безопасности. Количество имен функций класса string примерно соответствует количеству функций в библиотеке C, но механизм перегрузки существенно расширяет их функциональность. В сочетании с разумными правилами выбора имен и выбором аргументов по умолчанию, работать с классом string гораздо проще и удобнее, чем с функциями символьных массивов стандартной библиотеки C.

Присоединение, вставка и конкатенация строк

Одно из самых ценных и удобных свойств строк C++ состоит в том, что они автоматически растут по мере надобности, не требуя вмешательства со стороны программиста. Работа со строками не только становится более надежной, из нее почти полностью устраняются «нетворческие» операции — отслеживание границ памяти, в которой хранятся данные строк. Например, если при создании строковый объект был инициализирован 50 экземплярами символа "X", а позднее в нем были сохранены 50 экземпляров строки "Zowie", объект сам выделит достаточный блок памяти в соответствии с увеличившимся объемом данных. Но в полной мере это свойство проявляется в ситуациях, когда обрабатываемые в программе строки изменяются в размерах, но вы не можете оценить эти изменения количественно. Строковые функции append() и insert() автоматически перераспределяют память при увеличении строки:

```
//: C03:StrSize.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
```

```
  string bigNews("I saw Elvis in a UFO. ");
  cout << bigNews << endl;
  // Сколько данных фактически получено?
  cout << "Size = " << bigNews.size() << endl;
  // Сколько данных можно сохранить без перераспределения памяти?
  cout << "Capacity = "
    << bigNews.capacity() << endl;
  // Вставка строки в bigNews в позицию перед bigNews[1]
  bigNews.insert(1, " thought I");
  cout << bigNews << endl;
  cout << "Size = " << bigNews.size() << endl;
  cout << "Capacity = "
    << bigNews.capacity() << endl;
  // Резервирование достаточного объема памяти
  bigNews.reserve(500);
  // Присоединение в конец строки
  bigNews.append("I've been working too hard.");
  cout << bigNews << endl;
  cout << "Size = " << bigNews.size() << endl;
  cout << "Capacity = "
    << bigNews.capacity() << endl;
} ///:~
```

В одном из компиляторов был получен следующий результат:

```
I saw Elvis in a UFO.
Size = 22
Capacity = 31
I thought I saw Elvis in a UFO.
Size = 32
Capacity = 47
I thought I saw Elvis in a UFO. I've been working too hard.
Size = 59
Capacity = 511
```

Из приведенного примера видно, что хотя вы снимаете с себя большую часть ответственности за выделение памяти и управление ею в строках, класс string предоставляет в ваше распоряжение ряд средств для контроля за их размером. Обратите внимание, как легко был изменен размер блока памяти, выделенного под хранение символов строки. Функция size() возвращает текущее количество символов, она идентична функции length(). Функция capacity() возвращает размер текущего блока памяти, выделенного для хранения данных строки (то есть количество символов, которые можно сохранить в строке без необходимости выделения дополнительной памяти). Функция reserve() является средством оптимизации, выражающим ваше намерение зарезервировать определенный объем памяти для будущего использования; capacity() всегда возвращает значение, по крайней мере не меньшее того, которое было задано при последнем вызове reserve(). Функция resize() дополняет строку пробелами, если новый размер больше текущего, или усекает ее в противном случае (перегруженная версия resize() позволяет задать символ для дополнения строки).

Точный алгоритм выделения памяти функциями класса string зависит от реализации библиотеки. При тестировании предыдущего примера в одной из реализаций оказалось, что память выделялась с выравниванием по границе машинных слов, при этом один байт резервировался. Проектировщики класса string стремились к тому, чтобы строковые объекты C++ по возможности использовались вместе с символьными массивами C. Скорее всего, именно этот факт отразился в дан-

ных о емкости строк, выводимых в примере StrSize.cpp: резервирование одного байта позволяет легко вставить нуль-терминатор.

Замена символов в строках

Функция вставки символов insert() чрезвычайно удобна: вам не придется беспокоиться о том, чтобы вставляемые символы не вышли за пределы текущего блока памяти и не стерли символы, следующие за точкой вставки. Строка расширяется, и существующие символы вежливо подвигаются, уступая место новым. Впрочем, иногда такое поведение оказывается нежелательным. Если вы хотите, чтобы существующие символы были заменены новыми, воспользуйтесь функцией перезаписи replace(). Существует несколько перегруженных версий replace(), но простейшая форма получает три аргумента: начальную позицию в строке; количество символов, заменяемых в исходной строке; и строку замены (длина которой может не совпадать со вторым аргументом). Пример:

```
//: C03:StringReplace.cpp
// Простейший поиск с заменой в строках.
#include <cassert>
#include <string>
using namespace std;

int main() {
  string s("A piece of text");
  string tag("$tag$");
  s.insert(8, tag + ' ');
  assert(s == "A piece $tag$ of text");
  int start = s.find(tag);
  assert(start == 8);
  assert(tag.size() == 5);
  s.replace(start, tag.size(), "hello there");
  assert(s == "A piece hello there of text");
} ///:~
```

Строка tag сначала вставляется в s (обратите внимание: вставка производится *перед* заданной позицией, и после tag еще вставляется дополнительный пробел). Далее выполняются операции поиска и замены.

Прежде чем вызывать replace(), стоит проверить, удалось ли найти искомую подстроку. В предыдущем примере для замены использовался тип char*, но существует перегруженная версия replace() с аргументом типа string. Далее приводится более полный пример с заменой подстроки:

```
//: C03:Replace.cpp
#include <cassert>
#include <cstddef>  // Для size_t
#include <string>
using namespace std;

void replaceChars(string& modifyMe,
  const string& findMe, const string& newChars) {
  // Найти в modifyMe подстроку findMe
  // начиная с позиции 0.
  size_t i = modifyMe.find(findMe, 0);
  // Найдена ли заменяемая подстрока?
  if (i != string::npos)
    // Заменить найденную подстроку содержимым newChars
```

```
    modifyMe.replace(i, findMe.size(), newChars);
}

int main() {
  string bigNews =
   "I thought I saw Elvis in a UFO. "
   "I have been working too hard.";
  string replacement("wig");
  string findMe("UFO");
  // Найти и заменить в bigNews подстроку "UFO":
  replaceChars(bigNews, findMe, replacement);
  assert(bigNews == "I thought I saw Elvis in a "
         "wig. I have been working too hard.");
} ///:~
```

Если функция replace не находит искомую подстроку, она возвращает string::npos — статическую константу класса string, которая представляет несуществующую позицию символа[1].

В отличие от insert(), функция replace() не расширяет область данных строки при копировании в середину существующей последовательности символов. Тем не менее, она *расширит* область данных при необходимости, например, если в результате «замены» исходная строка выходит за пределы текущего блока. Пример:

```
//: C03:ReplaceAndGrow.cpp
#include <cassert>
#include <string>
using namespace std;

int main() {
  string bigNews("I have been working the grave.");
  string replacement("yard shift.");
  // Первый аргумент означает: "заменить символы
  // за концом существующей строки":
  bigNews.replace(bigNews.size() - 1,
    replacement.size(), replacement);
  assert(bigNews == "I have been working the "
         "graveyard shift.");
} ///:~
```

Вызов replace() начинает «замену» за концом существующего массива, что эквивалентно операции присоединения символов. В этом случае функция replace() соответствующим образом расширяет массив.

Возможно, вы наскоро просматриваете эту главу в поисках рецепта для выполнения относительно простой операции, такой как замена всех вхождений одного символа другим символом. Замена вроде бы относится к делу, но возня с поиском, группами символов, позициями и т. д. выглядят слишком сложно. Не позволяет ли класс string просто заменить один символ другим во всей строке?

Такую функцию легко написать на базе функций find() и replace():

```
//: C03:ReplaceAll.h
#ifndef REPLACEALL_H
#define REPLACEALL_H
#include <string>

std::string& replaceAll(string& context,
```

[1] Наибольшее значение, которое по умолчанию может быть представлено для типа size_type строкового распределителя памяти (std::size_t по умолчанию).

```
  const string& from, const string& to);
#endif // REPLACEALL_H ///:~

//: C03:ReplaceAll.cpp {O}
#include <cstddef>
#include "ReplaceAll.h"
using namespace std;

string& replaceAll(string& context, const string& from,
  const string& to) {
  size_t lookHere = 0;
  size_t foundHere;
  while ((foundHere = context.find(from, lookHere))
    != string::npos) {
    context.replace(foundHere, from.size(), to);
     lookHere = foundHere + to.size();
  }
  return context;
} ///:~
```

Версия find(), использованная в этой программе, получает во втором аргументе начальную позицию поиска, и возвращает string::npos. Позицию, хранящуюся в переменной lookHere, важно сместить за строку замены на тот случай, если from является подстрокой to. Следующая программа предназначена для тестирования функции replaceAll():

```
//: C03:ReplaceAllTest.cpp
//{L} ReplaceAll
#include <cassert>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
using namespace std;

int main() {
  string text = "a man, a plan, a canal, panama";
  replaceAll(text, "an", "XXX");
  assert(text == "a mXXX, a plXXX, a cXXXal, pXXXama");
} ///:~
```

Как видите, класс string сам по себе не решает все возможные задачи — во многих решениях приходится привлекать алгоритмы стандартной библиотеки[1], поскольку класс string может рассматриваться как контейнер STL (для чего используются итераторы, о которых говорилось выше). Все общие алгоритмы работают с «интервалами» элементов, хранящихся в контейнерах. Чаще всего используются интервалы «от начала до конца контейнера». Объект string интерпретируется как контейнер с элементами-символами; начало интервала определяется итератором string::begin(), а конец интервала — итератором string::end(). Следующий пример демонстрирует применение алгоритма replace() для замены всех вхождений символа «X» символом «Y»:

```
//: C03:StringCharReplace.cpp
#include <algorithm>
#include <cassert>
#include <string>
```

[1] Алгоритмы подробно рассматриваются в главе 6.

```
using namespace std;

int main() {
  string s("aaaXaaaXXaaXXXaXXXXaaa");
  replace(s.begin(), s.end(), 'X', 'Y');
  assert(s == "aaaYaaaYYaaYYYaYYYYaaa");
} ///:~
```

Обратите внимание: алгоритм replace() вызывается *не* как функция класса string(). Кроме того, в отличие от функций string::replace(), выполняющих только одну замену, алгоритм replace() заменяет *все вхождения* одного символа другим символом.

Алгоритм replace() работает только с отдельными объектами (в данном случае — с объектами char), он не может использоваться для замены символьных массивов или объектов string. Поскольку объект string ведет себя как контейнер STL, к нему могут применяться и другие алгоритмы. Это позволяет решать задачи, не решаемые напрямую функциями класса string.

Конкатенация с использованием перегруженных операторов

Многих программистов C при знакомстве с классом string языка C++ приятно удивляет простота объединения объектов string операторами + и +=. При использовании этих операторов синтаксис конкатенации строк напоминает суммирование числовых данных:

```
//: C03:AddStrings.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
  string s1("This ");
  string s2("That ");
  string s3("The other ");
  // Конкатенация строк оператором +
  s1 = s1 + s2;
  assert(s1 == "This That ");
  // Другой способ конкатенации строк
  s1 += s3;
  assert(s1 == "This That The other ");
  // В правой части может производиться индексирование строки
  s1 += s3 + s3[4] + "ooh lala";
  assert(s1 == "This That The other The other oooh lala");
} ///:~
```

Операторы + и += обеспечивают гибкие и удобные средства для объединения строковых данных. В правой части оператора может использоваться практически любой тип, интерпретируемый как один или несколько символов.

Поиск в строках

Функции группы find класса string предназначены для поиска символа или группы символов в заданной строке. Ниже перечислены функции группы find и область их применения.

`find()`

Ищет в строке заданный символ или группу символов. Возвращает начальную позицию первого найденного экземпляра или npos при отсутствии совпадений.

`find_first_of()`

Ищет в строке и возвращает позицию первого символа, совпадающего с *любым* символом из заданной группы. При отсутствии совпадений возвращается npos.

`find_last_of()`

Ищет в строке и возвращает позицию последнего символа, совпадающего с *любым* символом из заданной группы. При отсутствии совпадений возвращается npos.

`find_first_not_of()`

Ищет в строке и возвращает позицию первого элемента, не совпадающего *ни с одним* из символов заданной группы. При отсутствии совпадений возвращается npos.

`find_last_not_of()`

Ищет в строке и возвращает позицию последнего элемента, не совпадающего *ни с одним* из символов заданной группы. При отсутствии совпадений возвращается npos.

`rfind()`

Просматривает строку от конца к началу в поисках заданного символа или группы символов и возвращает начальную позицию совпадения. При отсутствии совпадений возвращается npos.

Простейший вариант функции find() ищет в строке один или несколько символов. Перегруженная версия find() получает параметр, определяющий искомый символ (или символы), и необязательный параметр, который указывает, где в строке должен начинаться поиск подстроки (по умолчанию поиск начинается с позиции 0). Вызывая find в цикле, вы можете легко перемещаться по строке и повторять поиск до обнаружения всех вхождений заданного символа или группы символов в строке.

Следующая программа использует *решето Эратосфена* для поиска простых чисел, меньших 50. Этот алгоритм начинает с числа 2, помечает все числа, кратные 2, как непростые, и повторяет процесс для следующего кандидата в простые числа. Конструктор SieveTest инициализирует sieveChars, для чего он задает исходный размер символьного массива и записывает значение 'P' в каждый из его элементов.

```
//: C03:Sieve.h
#ifndef SIEVE_H
#define SIEVE_H
#include <cmath>
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"
using std::size_t;
using std::sqrt;
using std::string;

class SieveTest : public TestSuite::Test {
```

```
  string sieveChars;
public:
  // Создание строки длиной 50 и заполнение ее символов
  // значениями 'P' (сокращение от Prime, то есть "простое число")
  SieveTest() : sieveChars(50, 'P') {}
  void run() {
    findPrimes();
    testPrimes();
  }
  bool isPrime(int p) {
    if (p == 0 || p == 1) return false;
    int root = int(sqrt(double(p)));
    for (int i = 2; i <= root; ++i)
      if (p % i == 0) return false;
    return true;
  }
  void findPrimes() {
    // По определению числа 0 и 1 не являются простыми.
    // Заменяем эти элементы символами "N".
    sieveChars.replace(0, 2, "NN");
    // Перебор массива:
    size_t sieveSize = sieveChars.size();
    int root = int(sqrt(double(sieveSize)));
    for (int i = 2; i <= root; ++i)
      // Поиск всех кратных чисел:
      for (size_t factor = 2; factor * i < sieveSize;
           ++factor)
        sieveChars[factor * i] = 'N';
  }
  void testPrimes() {
    size_t i = sieveChars.find('P');
    while (i != string::npos) {
      test_(isPrime(i++));
      i = sieveChars.find('P', i);
    }
    i = sieveChars.find_first_not_of('P');
    while (i != string::npos) {
      test_(!isPrime(i++));
      i = sieveChars.find_first_not_of('P', i);
    }
  }
};
#endif // SIEVE_H ///:~

//: C03:Sieve.cpp
//{L} ../TestSuite/Test
#include "../TestSuite/Test.h"
#include "Sieve.h"

int main() {
  SieveTest t;
  t.run();
  return t.report();
} ///:~
```

Функция find() перебирает содержимое string в прямом направлении; с ее помощью можно найти все вхождения символа или группы символов. Функция find_first_not_of() ищет символы или подстроки, не соответствующие заданному критерию.

В классе string не предусмотрены функции для изменения регистра символов, однако такие функции легко реализуются на базе стандартных библиотечных функций toupper() и tolower() языка C, изменяющих регистр отдельного символа. В следующем примере продемонстрирован поиск без учета регистра:

```
//: C03:Find.h
#ifndef FIND_H
#define FIND_H
#include <cctype>
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"
using std::size_t;
using std::string;
using std::tolower;
using std::toupper;

// Создание копии s в верхнем регистре
string upperCase(const string& s) {
  string upper(s);
  for(size_t i = 0; i < s.length(); ++i)
    upper[i] = toupper(upper[i]);
  return upper;
}

// Создание копии s в нижнем регистре
string lowerCase(const string& s) {
  string lower(s);
  for(size_t i = 0; i < s.length(); ++i)
    lower[i] = tolower(lower[i]);
  return lower;
}

class FindTest : public TestSuite::Test {
  string chooseOne;
public:
  FindTest() : chooseOne("Eenie, Meenie, Miney, Mo") {}
  void testUpper() {
    string upper = upperCase(chooseOne);
    const string LOWER = "abcdefghijklmnopqrstuvwxyz";
    test_(upper.find_first_of(LOWER) == string::npos);
  }
  void testLower() {
    string lower = lowerCase(chooseOne);
    const string UPPER = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    test_(lower.find_first_of(UPPER) == string::npos);
  }
  void testSearch() {
    // Поиск с учетом регистра
    size_t i = chooseOne.find("een");
    test_(i == 8);
    // Поиск в нижнем регистре:
    string test = lowerCase(chooseOne);
    i = test.find("een");
    test_(i == 0);
    i = test.find("een", ++i);
    test_(i == 8);
    i = test.find("een", ++i);
    test_(i == string::npos);
```

```
    // Поиск в верхнем регистре
    test = upperCase(chooseOne);
    i = test.find("EEN");
    test_(i == 0);
    i = test.find("EEN", ++i);
    test_(i == 8);
    i = test.find("EEN", ++i);
    test_(i == string::npos);
  }
  void run() {
    testUpper();
    testLower();
    testSearch();
  }
};
#endif // FIND_H ///:~

//: C03:Find.cpp
//{L} ../TestSuite/Test
#include "Find.h"
#include "../TestSuite/Test.h"

int main() {
  FindTest t;
  t.run();
  return t.report();
} ///:~
```

Функции upperCase() и lowerCase() работают по одному принципу: они создают копию своего аргумента типа string с измененным регистром символов. Программа Find.cpp не лучшим образом решает проблему изменения регистра, и мы вернемся к этой проблеме, когда будем рассматривать сравнения строковых объектов.

Поиск в обратном направлении

Если поиск в строке требуется выполнять от конца к началу (чтобы искать данные «от последнего вхождения к первому»), воспользуйтесь функцией rfind():

```
//: C03:Rparse.h
#ifndef RPARSE_H
#define RPARSE_H
#include <cstddef>
#include <string>
#include <vector>
#include "../TestSuite/Test.h"
using std::size_t;
using std::string;
using std::vector;

class RparseTest : public TestSuite::Test {
  // Вектор для хранения слов:
  vector<string> strings;
public:
  void parseForData() {
    // Символы ':' являются ограничителями
    string s("now.:sense:make:to:going:is:This");
    // Последний элемент строки:
    int last = s.size();
```

```
    // Начало текущего слова:
    size_t current = s.rfind(';');
    // Перебор строки в обратном направлении:
    while(current != string::npos){
      // Занесение слов в вектор.
      // Переменная current инкрементируется перед копированием,
      // чтобы предотвратить копирование ограничителя:
      ++current;
      strings.push_back(
        s.substr(current, last - current));
      // Пропустить найденный ограничитель
      // и установить last в конец следующего слова:
      current -= 2;
      last = current + 1;
      // Поиск следующего ограничителя
      current = s.rfind(';', current);
    }
    // Получение первого слова, не имеющего
    // префикса-ограничителя.
    strings.push_back(s.substr(0, last));
  }
  void testData() {
    // Тестирование в новом порядке:
    test_(strings[0] == "This");
    test_(strings[1] == "is");
    test_(strings[2] == "going");
    test_(strings[3] == "to");
    test_(strings[4] == "make");
    test_(strings[5] == "sense");
    test_(strings[6] == "now.");
    string sentence;
    for(int i = 0; i < strings.size() - 1; i++)
      sentence += strings[i] += " ";
    // Вручную занести последнее слово, чтобы избежать
    // сохранения лишнего пробела.
    sentence += strings[strings.size() - 1];
    test_(sentence == "This is going to make sense now.");
  }
  void run() {
    parseForData();
    testData();
  }
};
#endif // RPARSE_H ///:~

//: C03:Rparse.cpp
//{L} ../TestSuite/Test
#include "Rparse.h"

int main() {
  RparseTest t;
  t.run();
  return t.report();
} ///:~
```

Строковая функция rfind() перебирает строку в обратном направлении, ищет заданные лексемы и возвращает массив индексов совпадающих символов (или string::npos в случае неудачи).

Поиск первого/последнего символа из заданного подмножества

Функции find_first_of() и find_last_of() удобно использовать для создания небольшой утилиты, удаляющей пропуски с обоих концов строки. Обратите внимание: функция не изменяет оригинал, а возвращает новую строку:

```
//: C03:Trim.h
// Утилита для удаления пропусков с концов строки.
#ifndef TRIM_H
#define TRIM_H
#include <string>
#include <cstddef>

inline std::string trim(const std::string& s) {
  if(s.length() == 0)
    return s;
  int beg = s.find_first_not_of(" \a\b\f\n\r\t\v");
  int end = s.find_last_not_of(" \a\b\f\n\r\t\v");
  if(beg == std::string::npos) // Строка содержит только пропуски
    return "";
  return std::string(s, beg, end - beg + 1);
}
#endif // TRIM_H ///:~
```

Первый тест выясняет, не является ли строка пустой; в этом случае проверка не выполняется, и функция возвращает копию строки. Обратите внимание: после нахождения конечных точек конструктор string строит новую строку на основе старой по заданной начальной позиции и длине.

Такие общецелевые утилиты нуждаются в тщательном тестировании:

```
//: C03:TrimTest.h
#ifndef TRIMTEST_H
#define TRIMTEST_H
#include "Trim.h"
#include "../TestSuite/Test.h"

class TrimTest : public TestSuite::Test {
  enum {NTESTS = 11};
  static std::string s[NTESTS];
public:
  void testTrim() {
    test_(trim(s[0]) == "abcdefghijklmnop");
    test_(trim(s[1]) == "abcdefghijklmnop");
    test_(trim(s[2]) == "abcdefghijklmnop");
    test_(trim(s[3]) == "a");
    test_(trim(s[4]) == "ab");
    test_(trim(s[5]) == "abc");
    test_(trim(s[6]) == "a b c");
    test_(trim(s[7]) == "a b c");
    test_(trim(s[8]) == "a \t b \t c");
    test_(trim(s[9]) == "");
    test_(trim(s[10]) == "");
  }
  void run() {
    testTrim();
  }
};
```

```
#endif // TRIMTEST_H ///:~

//: C03:TrimTest.cpp {O}
#include "TrimTest.h"

// Инициализация статических данных
std::string TrimTest::s[TrimTest::NTESTS] = {
  " \t abcdefghijklmnop \t ",
  "abcdefghijklmnop \t ",
  " \t abcdefghijklmnop",
  "a", "ab", "abc", "a b c",
  " \t a b c \t ", " \t a \t b \t c \t ",
  "\t \n \r \v \f",
  "" // Также необходимо провести проверку пустой строки
}; ///:~

//: C03:TrimTestMain.cpp
//{L} ../TestSuite/Test TrimTest
#include "TrimTest.h"

int main() {
  TrimTest t;
  t.run();
  return t.report();
} ///:~
```

На примере массива strings видно, как символьные массивы автоматически преобразовываются в объекты string. В массив включены примеры для проверки факта удаления пробелов и символов табуляции с обоих концов строки, а также примеры, показывающие, что пробелы и табуляции в середине строки остаются на своих местах.

Удаление символов из строк

Функция erase() класса string легко и эффективно удаляет символы из строк. Функция получает два аргумента: начальную позицию удаления (по умолчанию 0) и количество удаляемых символов (по умолчанию string::npos). Если заданное количество символов больше количества оставшихся символов в строке, стираются все символы до конца (таким образом, вызов erase() без аргументов удаляет из строки все символы). Допустим, вы хотите удалить из файла HTML все теги со специальными символами; предполагается, что после удаления останется примерно тот же текст, который отображался в браузере, но только в виде простого текстового файла. В следующем примере для решения этой задачи используется функция erase():

```
//: C03:HTMLStripper.cpp {RunByHand}
//{L} ReplaceAll
// Фильтр для удаления тегов и маркеров HTML
#include <cassert>
#include <cmath>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <string>
#include "ReplaceAll.h"
#include "../require.h"
using namespace std;
```

```
string& stripHTMLTags(string& s) {
  static bool inTag = false;
  bool done = false;
  while (!done) {
    if (inTag) {
      // В предыдущей строке начался, но не закончился тег HTML.
      // Продолжаем поиск '>'.
      size_t rightPos = s.find('>');
      if (rightPos != string::npos) {
        inTag = false;
        s.erase(0, rightPos + 1);
      }
      else {
        done = true;
        s.erase();
      }
    }
    else {
      // Поиск начала тега:
      size_t leftPos = s.find('<');
      if (leftPos != string::npos) {
        // Проверить, закрывается ли тег в текущей строке
        size_t rightPos = s.find('>');
        if (rightPos == string::npos) {
          inTag = done = true;
          s.erase(leftPos);
        }
        else
          s.erase(leftPos, rightPos - leftPos + 1);
      }
      else
        done = true;
    }
  }
  // Удаление всех специальных символов HTML
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
  replaceAll(s, " ", " ");
  // И т. д.
  return s;
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  string s;
  while(getline(in, s))
    if (!stripHTMLTags(s).empty())
      cout << s << endl;
} ///:~
```

Программа удаляет даже теги HTML, занимающие несколько строк[1]. Для этого используется статический флаг inTag, которому присваивается true, если при об-

[1] Ради простоты приведенная версия не обрабатывает вложенные теги (например, комментарии).

наружении начального тега парный завершающий тег не был обнаружен в той же строке. В функции stripHTMLFlags() встречаются все формы функции erase()[1]. Используемая версия getline() представляет собой глобальную функцию, объявленную в заголовочном файле <string>; она удобна тем, что в аргументе string может храниться строка произвольной длины. Нам не приходится беспокоиться о размерах символьного массива, как при использовании функции istream::getline(). В программе задействована функция replaceAll(), упоминавшаяся ранее в этой главе. В следующей главе будет создано более элегантное решение с применением строковых потоков.

Сравнение строк

Сравнение строк принципиально отличается от сравнения чисел. Числа обладают постоянными значениями, смысл которых всегда и везде одинаков. Но для сравнения двух строк требуются *лексические сравнения*. Иначе говоря, когда вы проверяете символ и определяете, «больше» или «меньше» он, чем другой, вы в действительности сравниваете числовые представления этих символов в выбранной кодировке. Чаще всего используется кодировка ASCII, в которой печатные символы английского языка представляются десятичными числами в интервале от 32 до 127. В кодировке ASCII список символов начинается с пробела, далее следуют некоторые знаки препинания, а затем буквы верхнего и нижнего регистра. Таким образом, буквы в начале алфавита имеют меньшие ASCII-коды, чем буквы в конце алфавита. Учитывая это обстоятельство, проще запомнить, что фраза «s1 меньше s2 при лексическом сравнении» просто означает, что первый различающийся символ в строке s1 находится ближе к концу алфавита, чем символ в соответствующей позиции s2.

В C++ предусмотрено несколько способов сравнения строк, каждый из которых обладает своими достоинствами и недостатками. Проще всего использовать перегруженные внешние (то есть не функции класса) операторные функции operator==, operator!=, operator>, operator<, operator>= и operator<=.

```
//: C03:CompStr.h
#ifndef COMPSTR_H
#define COMPSTR_H
#include <string>
#include "../TestSuite/Test.h"
using std::string;

class CompStrTest : public TestSuite::Test {
public:
  void run() {
    // Сравниваемые строки
    string s1("This");
    string s2("That");
    test_(s1 == s1);
    test_(s1 != s2);
    test_(s1 > s2);
    test_(s1 >= s2);
```

[1] На первый взгляд кажется, что простые математические вычисления позволят исключить часть вызовов erase(), но так как в некоторых случаях один из операндов равен string::npos (наибольшее беззнаковое целое), произойдет целочисленное переполнение, которое нарушит работу алгоритма.

```
    test_(s1 >= s1);
    test_(s2 < s1);
    test_(s2 <= s1);
    test_(s1 <= s1);
  }
};
#endif // COMPSTR_H ///:~

//: C03:CompStr.cpp
//{L} ../TestSuite/Test
#include "CompStr.h"

int main() {
  CompStrTest t;
  t.run();
  return t.report();
} ///:~
```

Перегруженные операторы сравнения обычно применяются для сравнения как целых строк, так и их отдельных символов.

В следующем примере обратите внимание на гибкость типов аргументов в левой и правой частях операторов сравнения. Для повышения эффективности в классе string определены перегруженные операторы для прямых сравнений строковых объектов, литералов в кавычках и указателей на строки C; это позволяет обойтись без создания временных объектов string.

```
//: C03:Equivalence.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s2("That"), s1("This");
  // В левой части находится литерал в кавычках,
  // в правой части - объект string
  if("That" == s2)
    cout << "A match" << endl;
    // В левой части находится объект string,
    // а в правой - указатель на строку в стиле C,
    // завершенную нуль-терминатором.
    if(s1 != s2.c_str())
      cout << "No match" << endl;
} ///:~
```

Функция c_str() возвращает const char* — указатель на строку C, завершенную нуль-символом, которая эквивалентна текущему содержимому объекта string. Такая возможность задействуется при передаче объектов string стандартным функциям C, например функции atoi() или любым функциям, определенным в заголовочном файле <cstring>. Использование значения, возвращаемого c_str(), в качестве неконстантного аргумента функции является ошибкой.

Среди операторов сравнения строк отсутствуют операторы логического отрицания (!) и конъюнкции/дизъюнкции (также вы не найдете перегруженных версий поразрядных операторов C &, |, ^ и ~). Перегруженные внешние операторы сравнения для класса string ограничиваются подмножеством, имеющим четкое однозначное применение к отдельным символам или их группам.

Функция compare() класса string позволяет выполнять гораздо более изощренные и точные сравнения, чем набор внешних операторов. Она существует в нескольких перегруженных версиях для сравнения:

- двух полных строк;
- части одной строки с полной строкой;
- подмножеств двух строк.

В следующем примере сравниваются две полные строки.

```
//: C03:Compare.cpp
// Применение функций compare() и swap()
#include <cassert>
#include <string>
using namespace std;

int main() {
  string first("This");
  string second("That");
  assert(first.compare(first) == 0);
  assert(second.compare(second) == 0);
  // Какая из строк лексически больше?
  assert(first.compare(second) > 0);
  assert(second.compare(first) < 0);
  first.swap(second);
  assert(first.compare(second) < 0);
  assert(second.compare(first) > 0);
} ///:~
```

Функция swap(), использованная в примере, меняет местами содержимое своего объекта и аргумента. Чтобы сравнить подмножества символов в одной или обеих строках, добавьте аргументы, определяющие начальную позицию и количество сравниваемых символов. Например, можно воспользоваться следующей перегруженной версией compare():

```
//: C03:Compare2.cpp
// Использование перегруженной версии compare()
#include <cassert>
#include <string>
using namespace std;

int main() {
  string first("This is a day that will live in infamy");
  string second("I don't believe that this is what "
                "I signed up for");
  // Сравнение "his is" в обеих строках:
  assert(first.compare(1, 7, second, 22, 7) == 0);
  // Сравнение "his is a" с "his is w":
  assert(first.compare(1, 9, second, 22, 9) < 0);
} ///:~
```

Во всех примерах, встречавшихся ранее, при обращении к отдельным символам строк использовался синтаксис индексирования в стиле массивов C. Строки C++ также поддерживают альтернативный вариант: функцию at(). Если все проходит нормально, эти два механизма индексации приводят к одинаковым результатам:

```
//: C03:StringIndexing.cpp
#include <cassert>
```

```
#include <string>
using namespace std;

int main(){
  string s("1234");
  assert(s[1] == '2');
  assert(s.at(1) == '2');
} ///:~
```

И все же между оператором [ ] и функцией at() существует одно важное различие. При попытке обратиться к элементу по индексу, выходящему за границы массива, функция at() великодушно выдает исключение, тогда как обычный синтаксис [ ] приводит к непредсказуемым последствиям:

```
//: C03:BadStringIndexing.cpp
#include <exception>
#include <iostream>
#include <string>
using namespace std;

int main(){
  string s("1234");
  // Функция at() запускает исключение и спасает вас от беды:
  try {
    s.at(5);
  } catch(exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Добросовестные программисты не пишут неправильные индексы, но если вы захотите воспользоваться преимуществами автоматической проверки индексов, используйте функцию at() вместо оператора [ ], это позволит корректно продолжить работу после ссылок на несуществующие элементы. При запуске этой программы на одном из тестовых компиляторов был получен следующий результат:

```
invalid string position
```

Функция at() запускает объект класса out_of_range, производный (в конечном счете) от std::exception. Перехватывая этот объект в обработчике, можно предпринять необходимые меры, например вычислить заново неправильный индекс или расширить массив. Индексация с применением операторной функции string::operator[]() не обеспечивает такой защиты и является таким же рискованным делом, как обработка символьных массивов в C[1].

Строки и характеристики символов

Знакомясь с программой Find.cpp, приведенной ранее в этой главе, трудно удержаться от очевидного вопроса: почему сравнение без учета регистра символов не поддерживается в стандартном классе string? Ответ заставляет по-новому взглянуть на истинную природу строковых объектов C++.

Подумайте, а что, собственно, означает «регистр» символа? В иврите, фарси и японской письменности отсутствуют концепции верхнего и нижнего регистров,

[1] По соображениям безопасности комитет по стандартизации C++ рассматривает предложение, согласно которому функция string::operator[] переопределяется идентично string::at().

поэтому для этих языков такая концепция бессмысленна. Конечно, можно создать механизм пометки некоторых языков «только верхним» или «только нижним» регистром, это позволит создать обобщенное решение. Но в ряде языков, поддерживающих концепцию регистра, смысл некоторых символов с диакритическими знаками меняется при изменении регистра (например, седиль в испанском, циркумфлекс во французском или умляут в немецком языке). По этой причине любые схемы с регистровыми символами, претендующие на полноту и законченность, окажутся неимоверно сложными в использовании.

Хотя строки C++ обычно интерпретируются как классы, на самом деле это не совсем так. Тип string представляет собой специализацию более общего шаблона basic_string. Посмотрите, как выглядит объявление string в стандартном заголовочном файле C++[1]:

```
typedef basic_string<char> string;
```

Чтобы лучше понять природу класса string, стоит взглянуть на шаблон basic_string:

```
template<class charT, class traits = char_traits<charT>,
  class allocator = allocator<charT> >  class basic_string;
```

В главе 5 шаблоны будут рассматриваться более подробно (гораздо подробнее, чем в главе 16 первого тома). А пока просто обратите внимание, что тип string создается специализацией шаблона basic_string по типу char. Внутри объявления basic_string<> следующая строка сообщает, что поведение класса, созданного на базе шаблона basic_string, задается классом, созданным на базе шаблона char_traits<>:

```
class traits = char_traits<charT>,
```

Таким образом, на базе шаблона basic_string<> создаются строковые классы, которые помимо char могут работать с другими типами (например, с символами в расширенной кодировке). При этом шаблон char_traits<> определяет порядок сортировки в различных кодировках при помощи функций eq() (равно), ne() (не равно) и lt() (меньше). От этих функций зависит работа функций сравнения строк basic_string<>.

Теперь понятно, почему класс string не содержит функций, не учитывающих регистр символов: это не входит в его задачи. Чтобы изменить способ сравнения строк в классе string, следует предоставить другой шаблон char_traits<>, поскольку именно он определяет поведение отдельных функций сравнения символов.

На основе этой информации можно создать новую разновидность класса string, игнорирующую регистр символов при сравнениях. Сначала мы должны определить новый шаблон char_traits<>, производный от существующего шаблона, который бы игнорировал регистр символов. Затем следует переопределить только те функции, которые бы обеспечивали посимвольное сравнение без учета регистра (кроме трех функций лексического сравнения, о которых упоминалось ранее, также определяется новая реализация функций find() и compare() шаблона char_traits). Наконец, мы определяем новую специализацию на базе шаблона basic_string<>, но передаем во втором аргументе новый шаблон ichar_traits:

```
//: C03:ichar_traits.h
// Создание пользовательских классов характеристик символов
#ifndef ICHAR_TRAITS_H
```

[1] Ваша реализация может определять все три аргумента шаблона. Поскольку последние два аргумента имеют значения по умолчанию, это объявление будет эквивалентно приведенному в тексте.

```
#define ICHAR_TRAITS_H
#include <cassert>
#include <cctype>
#include <cmath>
#include <cstddef>
#include <ostream>
#include <string>

using std::allocator;
using std::basic_string;
using std::char_traits;
using std::ostream;
using std::size_t;
using std::string;
using std::toupper;
using std::tolower;

struct ichar_traits : char_traits<char> {
  // Изменяются только функции сравнения символов.
  static bool eq(char c1st, char c2nd) {
    return toupper(c1st) == toupper(c2nd);
  }
  static bool ne(char c1st, char c2nd) {
    return !eq(c1st, c2nd);
  }
  static bool lt(char c1st, char c2nd) {
    return toupper(c1st) < toupper(c2nd);
  }
  static int
  compare(const char* str1, const char* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(tolower(*str1) < tolower(*str2))
        return -1;
      else if(tolower(*str1) > tolower(*str2))
        return 1;
      assert(tolower(*str1) == tolower(*str2));
      str1++; str2++; // Compare the other chars
    }
    return 0;
  }
  static const char*
  find(const char* s1, size_t n, char c) {
    while(n-- > 0)
      if(toupper(*s1) == toupper(c))
        return s1;
      else
        ++s1;
    return 0;
  }
};

typedef basic_string<char, ichar_traits> istring;

inline ostream& operator<<(ostream& os, const istring& s) {
  return os << string(s.c_str(), s.length());
```

```
}
#endif // ICHAR_TRAITS_H ///:~
```

Мы создали определение типа istring, чтобы наш класс был во всех отношениях аналогичен обычному классу string, кроме одного — все сравнения осуществляются без учета регистра символов. Для удобства также предоставлена перегруженная версия операторной функции operator<<() для вывода строк. Пример:

```
//: C03:ICompare.cpp
#include <cassert>
#include <iostream>
#include "ichar_traits.h"
using namespace std;

int main() {
  // Буквы совпадают, отличается только регистр:
  istring first = "tHis";
  istring second = "ThIS";
  cout << first << endl;
  cout << second << endl;
  assert(first.compare(second) == 0);
  assert(first.find('h') == 1);
  assert(first.find('I') == 2);
  assert(first.find('x') == string::npos);
} ///:~
```

Впрочем, это сильно упрощенный, «ненастоящий» пример. Чтобы класс istring стал полностью эквивалентным string, необходимо определить другие функции, обеспечивающие его работу.

В заголовочном файле <string> также определяется строковый класс с расширенной кодировкой символов:

```
typedef basic_string<wchar_t> wstring;
```

Поддержка расширенных кодировок проявляется в расширенных потоках ввода-вывода (класс wostream вместо ostream, также определяемый в <iostream>) и заголовочном файле <cwctype>, расширенной версии <cctype>. Наряду со специализацией шаблона char_traits по типу wchar_t в стандартной библиотеке это позволяет создать версию ichar_traits для символов в расширенной кодировке:

```
//: C03:iwchar_traits.h {-g++}
// Создание класса характеристик для расширенных символов
#ifndef IWCHAR_TRAITS_H
#define IWCHAR_TRAITS_H
#include <cassert>
#include <cmath>
#include <cstddef>
#include <cwctype>
#include <ostream>
#include <string>

using std::allocator;
using std::basic_string;
using std::char_traits;
using std::towlower;
using std::towupper;
using std::wostream;
using std::wstring;

struct iwchar_traits : char_traits<wchar_t> {
```

```
  // Изменяются только функции сравнения символов.
  static bool eq(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) == towupper(c2nd);
  }
  static bool ne(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) != towupper(c2nd);
  }
  static bool lt(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) < towupper(c2nd);
  }
  static int compare(const wchar_t* str1,
    const wchar_t* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(towlower(*str1) < towlower(*str2))
        return -1;
      else if(towlower(*str1) > towlower(*str2))
        return 1;
      assert(towlower(*str1) == towlower(*str2));
      str1++; str2++; // Сравнение следующих wchar_t
    }
    return 0;
  }
  static const wchar_t* find(const wchar_t* s1,
    size_t n, wchar_t c) {
    while(n-- > 0)
      if(towupper(*s1) == towupper(c))
        return s1;
      else
        ++s1;
    return 0;
  }
};

typedef basic_string<wchar_t, iwchar_traits> iwstring;

inline wostream& operator<<(wostream& os,
  const iwstring& s) {
  return os << wstring(s.c_str(), s.length());
}
#endif // IWCHAR_TRAITS_H ///:~
```

Как видите, задача в основном сводится к простой подстановке символа «w» в некоторых местах исходной программы. Тестовая программа выглядит так:

```
//: C03:IWCompare.cpp {-g++}
#include <cassert>
#include <iostream>
#include "iwchar_traits.h"
using namespace std;

int main() {
  // Буквы совпадают, отличается только регистр:
  iwstring wfirst = L"tHis";
  iwstring wsecond = L"ThIS";
  wcout << wfirst << endl;
  wcout << wsecond << endl;
```

```
  assert(wfirst.compare(wsecond) == 0);
  assert(wfirst.find('h') == 1);
  assert(wfirst.find('I') == 2);
  assert(wfirst.find('x') == wstring::npos);
} ///:~
```

К сожалению, некоторые компиляторы до сих пор не обладают полноценной поддержкой расширенных кодировок.

Пример обработки строк

Если вы внимательно рассматривали примеры программ в книге, то наверняка обратили внимание на специальные маркеры в комментариях. Они используются программой Python для извлечения программного кода из файлов и построения make-файлов. Например, последовательность //: в начале строки обозначает первую строку программы. За этими символами следует информация с именем файла, его местонахождением и флагом, который указывает, что программу необходимо только откомпилировать (без построения исполняемого файла). Так, первая строка предыдущей программы содержит строку `C03:IWCompare.cpp`, означающую, что файл `IWCompare.cpp` извлекается из текстового файла в каталог `C03`.

Последняя строка фрагмента помечается последовательностью ///:~. Если в первой строке сразу же после двоеточия следует восклицательный знак, первая и последняя строки кода не выводятся в файл (для файлов, содержащих только данные).

Возможности программы Python не ограничиваются простым извлечением кода. Если за именем файла следует маркер {O}, то запись в make-файле настраивается на компиляцию исходного текста программы без его компоновки в исполняемый файл (в частности, так строится тестовая система в главе 2). Чтобы скомпоновать такой файл с другим исходным файлом, включите в исходный код исполняемой программы директиву {L}:

```
//{L} ../TestSuite/Test
```

В этом разделе будет представлена утилита, которая извлекает из текста книги весь код (вы можете самостоятельно просмотреть и откомпилировать примеры программ). Сохраните документ в текстовом формате[1] (допустим, в файле `TICV2.txt`) и введите в командной строке следующую команду:

```
C:> extractCode TICV2.txt /TheCode
```

Команда читает текстовый файл `TICV2.txt` и сохраняет весь программный код в подкаталогах каталога `/TheCode`. Дерево каталогов будет выглядеть так:

```
TheCode/
   C0B/
   C01/
   C02/
   C03/
   C04/
   C05/
```

[1] Учтите, что некоторые версии Microsoft Word при сохранении в текстовом формате заменяют апострофы расширенным символом ASCII, что приводит к ошибкам компиляции. Мы понятия не имеем, почему это происходит. Просто замените символы апострофами вручную.

```
C06/
C07/
C08/
C09/
C10/
C11/
TestSuite/
```

Файлы с исходными текстами примеров каждой главы находятся в соответствующем каталоге.

А вот и сама программа:

```
//: C03:ExtractCode.cpp
// Извлечение программного кода из текста
#include <cassert>
#include <cstddef>
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
// Унаследованный нестандартный заголовочный файл C для mkdir()
#if defined (__GNUC__) || defined(__MSWERKS__)
#include <sys/stat.h>
#elif defined(__BORLANDC__) || defined(_MSC_VER)
  || defined(__DMC__)
#include <direct.h>
#else
#error Compiler not supported
#endif

// Чтобы проверить, существует ли каталог, мы пытаемся
// открыть в нем новый файл для вывода.
bool exists(string fname) {
  size_t len = fname.length();
  if(fname[len-1] != '/' && fname[len-1] != '\\')
    fname.append("/");
  fname.append("000.tmp");
  ofstream outf(fname.c_str());
  bool existFlag = outf;
  if (outf) {
    outf.close();
    remove(fname.c_str());
  }
  return existFlag;
}

int main(int argc, char* argv[]) {
  // Проверяем, указано ли имя входного файла
  if(argc == 1) {
    cerr << "usage: extractCode file [dir]\n";
    exit(EXIT_FAILURE);
  }
  // Проверяем, существует ли входной файл
  ifstream inf(argv[1]);
  if(!inf) {
    cerr << "error opening file: " << argv[1] << endl;
    exit(EXIT_FAILURE);
```

```
}
// Проверяем наличие необязательного выходного каталога
string root("./");  // По умолчанию используется текущий каталог
if(argc == 3) {
  // Проверяем, существует ли выходной каталог
  root = argv[2];
  if(!exists(root)) {
    cerr << "no such directory: " << root << endl;
    exit(EXIT_FAILURE);
  }
  size_t rootLen = root.length();
  if(root[rootLen-1] != '/' && root[rootLen-1] != '\\')
    root.append("/");
}
// Построчное чтение входного файла
// с проверкой маркеров начала и конца программных блоков
string line;
bool inCode = false;
bool printDelims = true;
ofstream outf;
while (getline(inf, line)) {
  size_t findDelim = line.find("//" "/:~");
  if(findDelim != string::npos) {
    // Вывод последней строки и закрытие файла
    if (!inCode) {
      cerr << "Lines out of order\n";
      exit(EXIT_FAILURE);
    }
    assert(outf);
    if (printDelims)
      outf << line << endl;
    outf.close();
    inCode = false;
    printDelims = true;
  } else {
    findDelim = line.find("//" ":");
    if(findDelim == 0) {
      // Проверка директивы '!'
      if(line[3] == '!') {
        printDelims = false;
        ++findDelim;  // Чтобы пропустить '!' при следующем поиске
      }
      // Извлечение имени подкаталога (если оно есть)
      size_t startOfSubdir =
        line.find_first_not_of(" \t", findDelim+3);
      findDelim = line.find(':', startOfSubdir);
      if (findDelim == string::npos) {
        cerr << "missing filename information\n" << endl;
        exit(EXIT_FAILURE);
      }
      string subdir;
      if(findDelim > startOfSubdir)
        subdir = line.substr(startOfSubdir,
                             findDelim - startOfSubdir);
      // Извлечение обязательного имени файла
      size_t startOfFile = findDelim + 1;
      size_t endOfFile =
        line.find_first_of(" \t", startOfFile);
      if(endOfFile == startOfFile) {
```

```
          cerr << "missing filename\n";
          exit(EXIT_FAILURE);
        }
        // Все компоненты присутствуют; построение имени в fullPath
        string fullPath(root);
        if(subdir.length() > 0)
          fullPath.append(subdir).append("/");
        assert(fullPath[fullPath.length()-1] == '/');
        if (!exists(fullPath))
#ifdef __GNUC__
          mkdir(fullPath.c_str(), 0);  // Создание подкаталога
#else
          mkdir(fullPath.c_str());  // Создание подкаталога
#endif
        fullPath.append(line.substr(startOfFile,
                        endOfFile - startOfFile));
        outf.open(fullPath.c_str());
        if(!outf) {
          cerr << "error opening " << fullPath
               << " for output\n";
          exit(EXIT_FAILURE);
        }
        inCode = true;
        cout << "Processing " << fullPath << endl;
        if(printDelims)
          outf << line << endl;
      }
      else if(inCode) {
        assert(outf);
        outf << line << endl;  // Вывод строки программы
      }
    }
  }
  exit(EXIT_SUCCESS);
} ///:~
```

Обратите внимание на директивы условной компиляции. Функция mkdir(), создающая каталог в файловой системе, по стандарту POSIX[1] определяется в заголовочном файле <sys/stat.h>. К сожалению, многие компиляторы продолжают использовать другой заголовочный файл (<direct.h>). Сигнатуры mkdir() также различаются: в POSIX функция вызывается с двумя аргументами, а в старых версиях — только с одним. Из-за этого в программе позднее присутствуют дополнительные команды условной компиляции для выбора правильного вызова mkdir(). Обычно мы не задействуем условную компиляцию в своих примерах, но эта конкретная программа слишком полезна, и к ней стоит приложить небольшие дополнительные усилия.

Функция exists() в программе ExtractCode.cpp проверяет существование каталога. Для этого она пытается открыть временный файл в этом каталоге. Если попытка открытия завершается неудачей, считается, что каталог не существует. Чтобы удалить файл, мы передаем его имя в формате char* функции std::remove().

Основная программа проверяет аргументы командной строки, а затем читает входной файл по строкам, проверяя наличие специальных ограничителей. Логи-

[1] В стандарте IEEE POSIX (Portable Operating System Interface) обобщены многие низкоуровневые системные функции, используемые в системах семейства Unix.

ческий флаг inCode указывает, находится ли программа в процессе чтения программного блока (то есть вывода читаемых строк). Флаг printDelims равен true, если за открывающим маркером не следует восклицательный знак; в противном случае первая и последняя строки не выводятся. Важно начать проверку с конечного маркера, поскольку начальный маркер полностью входит в него, и поиск по начальному маркеру будет успешным для обоих случаев. Обнаружив закрывающий маркер, мы убеждаемся в том, что в настоящее время обрабатывается блок программного кода; если это не так, то при размещении маркеров в текстовом файле были допущены ошибки. Если флаг inCode равен true, все идет нормально, мы выводим последнюю строку (если ее нужно выводить) и закрываем файл. При обнаружении начального маркера программа выделяет из строки имена каталога и файла, после чего открывает файл. В приведенном примере используются следующие строковые функции: length(), append(), getline(), find() (две версии), find_first_not_of(), substr(), find_first_of(), c_str() и, конечно, operator<<().

Итоги

Объекты C++ string обладают огромными преимуществами перед своими прототипами из языка C. Прежде всего это связано с тем, что класс string позволяет работать со строками без применения указателей. При этом исчезает целый класс программных ошибок, обусловленных применением неинициализированных или неправильно инициализированных указателей.

Строки C++ динамически расширяют свой внутренний блок данных в соответствии с увеличением объема строковых данных, причем это не требует участия пользователя. Когда данные строки выходят за пределы изначально выделенной памяти, строковый объект вызывает функции для выделения и освобождения памяти в куче (динамическом пуле). Проверенная схема управления памятью предотвращает утечки и обычно работает гораздо эффективнее «самодельных» решений.

Функции класса string предоставляют достаточно полный инструментарий для создания строк, их модификации и поиска. Сравнения в стандартном классе string всегда выполняются с учетом регистра символов, но это ограничение можно обойти разными способами — копированием строковых данных в строки C, завершенные нуль-символами, временным преобразованием данных строковых объектов к общему регистру или созданием класса с переопределением стандартных характеристик символов, используемых при специализации шаблона basic_string.

Упражнения

1. Напишите и протестируйте функцию, которая переставляет символы строки в обратном порядке.
2. Палиндромом называется слово или фраза, которые одинаково читаются в обоих направлениях (например, «madam» или «wow»). Напишите программу, которая читает строковый аргумент из командной строки и при помощи функции из предыдущего упражнения проверяет, является ли эта строка палиндромом.

3. Измените программу из упражнения 2, чтобы она возвращала true даже в том случае, если симметричные символы различаются регистром. Например, слово «Civic» должно считаться палиндромом, хотя его первая буква принадлежит к верхнему регистру.
4. Измените программу из упражнения 3, чтобы она игнорировала пробелы и знаки препинания. Например, для строки «Able was I, ere I saw Elba» должно возвращаться значение true.
5. Рассмотрим следующие объявления строк:

```
string one("I walked down the canyon with the moving mountain bikers.");
string two("The bikers passed by me too close for comfort.");
string three("I went hiking instead.")
```

Используя только эти объявления и символы char (никаких строковых литералов или «волшебных чисел»), постройте последовательность предложений: I moved down the canyon with the mountain bikers. The mountain bikers passed by me too close for comfort. So I went hiking instead.

6. Напишите программу replace, которая получает из командной строки три аргумента: входной текстовый файл, заменяемую строку (назовем ее from) и строку замены (назовем ее to). Программа должна записать в стандартный выходной поток новый файл, в котором все вхождения from заменяются на to.
7. Повторите предыдущее упражнение так, чтобы заменялись все экземпляры from независимо от регистра символов.
8. Измените программу из упражнения 3, чтобы она читала имя файла из командной строки и выводила все слова-палиндромы из этого файла (без учета регистра). Не выводите дубликаты даже при различиях в регистре символов. Не пытайтесь искать палиндромы, размер которых превышает одно слово (в отличие от упражнения 4).
9. Измените программу HTMLStripper.cpp, чтобы при обнаружении тегов она выводила имя тега и содержимое файла между начальным и конечным тегами. Предполагается, что вложенные теги отсутствуют, а конечные теги всегда задаются явно (в формате </*mer*>).
10. Напишите программу, которая получает из командной строки три аргумента (имя файла и две строки) и выводит на консоль все логические строки файла, содержащие обе искомые строки, одну из них или ни одной в зависимости от того, какой режим выберет пользователь в начале программы. Во всех режимах, кроме последнего, выделите входные строки (или строку), отметив их начало и конец символами *.
11. Напишите программу, которая получает из командной строки два аргумента (имя файла и строку) и подсчитывает количество вхождений строки в файл даже в составе других слов. Перекрывающиеся вхождения игнорируются. Например, входная строка «ba» обнаруживается в слове «basketball» дважды, но входная строка «ana» обнаруживается в слове «banana» только один раз. Выведите на консоль количество вхождений и среднюю длину слов, в которых присутствуют совпадения (если строка входит в слово несколько раз, учитывается только одно вхождение).

12. Напишите программу, которая получает из командной строки имя файла и выводит информацию об использовании всех символов, включая знаки препинания и пробелы (символы с кодами от 0x21[33] до 0x7E[126], а также пробел). Иначе говоря, подсчитайте количество вхождений каждого символа в файл и выведите результаты отсортированными либо последовательно (пробел, затем !, ", # и т. д.), либо по возрастанию или убыванию количества вхождений в зависимости от пользовательского ввода в начале программы. Для пробела вместо символа ' ' выводится строка Space. Примерный результат выглядит так:

```
Format sequentially, ascending or descending
 (S/A/D): D
t:  526
r:  490
etc.
```

13. Используя функции find() и rfind(), напишите программу, которая получает из командной строки два аргумента (имя файла и строку) и выводит первое и последнее слова (и их индексы), не совпадающие с заданной строкой, а также индексы первого и последнего вхождений строки. Если ни одна из проверок не дает результатов, выведите «Not found».

14. Используя функции семейства find_first_of (но не ограничиваясь ими), напишите программу, которая удаляет из файла все неалфавитные символы, кроме точек и пробелов, а затем преобразует каждую первую букву, следующую после точки, к верхнему регистру.

15. Используя функции семейства find_first_of, напишите программу, которая получает из командной строки имя файла и преобразует все числа в этом файле к формату денежных величин. Игнорируйте десятичные точки после первой до обнаружения нечислового символа, а затем округлите до двух чисел в дробной части. Например, строка 12.399abc29.00.6a приводится к виду $12.40abc$29.01a.

16. Напишите программу, которая получает из командной строки два аргумента (имя файла и число) и переставляет буквы в словах. Для этого программа выбирает две буквы, меняет их местами и повторяет эту процедуру столько раз, сколько задано вторым аргументом. Например, если во втором аргументе передается 0, слова не изменяются; если второй аргумент равен 1, меняется местами одна пара букв; если второй аргумент равен 2, меняются местами две пары и т. д.

17. Напишите программу, которая получает из командной строки имя файла и выводит количество предложений в этом файле (определяемое по количеству точек), среднее количество символов в одном предложении и общее количество символов в файле.

18. Докажите, что функция at() действительно запускает исключение при недопустимом значении индекса, а оператор индексирования [ ] этого не делает.

Потоки ввода-вывода

В плане ввода-вывода язык C++ отнюдь не ограничивается простой инкапсуляцией в классах стандартного ввода-вывода.

Разве не замечательно было бы, если бы все операции с обычными участниками ввода-вывода — консолью, файлами и даже блоками памяти — выполнялись одинаково, чтобы программисту было достаточно запомнить только один интерфейс? Эта концепция заложена в основу *потоков ввода-вывода*. Они гораздо удобнее, безопаснее, а в отдельных случаях и эффективнее соответствующих функций стандартной библиотеки stdio языка C.

Начинающие программисты C++ обычно начинают свое знакомство со стандартной библиотекой именно с классов потоков ввода-вывода. В этой главе будет показано, в чем потоки ввода-вывода превосходят средства языка C, а также исследовано поведение файловых и строковых потоков в дополнение к стандартным консольным потокам.

Зачем нужны потоки?

Возникает вопрос — чем плоха старая библиотека C? Нельзя ли просто «завернуть» библиотеку C в классы и ограничиться этим? Иногда такого решения оказывается вполне достаточно. Допустим, вы хотите сделать так, чтобы файл, представленный указателем stdio FILE, всегда безопасно открывался и корректно закрывался, даже если пользователь забудет вызвать функцию close(). Попытка такого рода сделана в следующей программе:

```
//: C04:FileClass.h
// Объектный интерфейс для файлов stdio
#ifndef FILECLASS_H
#define FILECLASS_H
#include <cstdio>
#include <stdexcept>

class FileClass {
```

```
  std::FILE* f;
public:
  struct FileClassError : std::runtime_error {
    FileClassError(const char* msg)
      : std::runtime_error(msg) {}
  };
  FileClass(const char* fname, const char* mode = "r");
  ~FileClass();
  std::FILE* fp();
};
#endif // FILECLASS_H ///:~
```

При выполнении файловых операций ввода-вывода в C вы работаете с низкоуровневым указателем на структуру FILE. Представленный класс инкапсулирует этот указатель, гарантируя его правильную инициализацию и освобождение в конструкторе и деструкторе. Второй аргумент конструктора определяет режим файла; по умолчанию используется режим "r" (чтение).

Функция fp() возвращает указатель, используемый функциями файлового ввода-вывода. Определения функций класса FileClass выглядят так:

```
//: C04:FileClass.cpp {O}
// Реализация FileClass
#include "FileClass.h"
#include <cstdlib>
#include <cstdio>
using namespace std;

FileClass::FileClass(const char* fname, const char* mode) {
  if((f = fopen(fname, mode)) == 0)
    throw FileClassError("Error opening file");
}

FileClass::~FileClass() { fclose(f); }

FILE* FileClass::fp() { return f; } ///:~
```

Конструктор вызывает функцию fopen(), как при обычном открытии файла, но он также проверяет, что результат отличен от нуля (признак ошибки открытия файла). Если попытка открытия файла завершилась неудачей, класс запускает исключение.

Деструктор закрывает файл, а функция fp() возвращает закрытую переменную f. Ниже приводится простой пример использования класса FileClass:

```
//: C04:FileClassTest.cpp
//{L} FileClass
#include <cstdlib>
#include <iostream>
#include "FileClass.h"
using namespace std;

int main() {
  try {
    FileClass f("FileClassTest.cpp");
    const int BSIZE = 100;
    char buf[BSIZE];
    while(fgets(buf, BSIZE, f.fp()))
      fputs(buf, stdout);
  } catch(FileClass::FileClassError& e) {
```

```
    cout << e.what() << endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
} // Файл автоматически закрывается деструктором
///:~
```

Мы создаем объект FileClass и используем его в обычных вызовах функций ввода-вывода C, вызывая функцию fp(). Завершив работу с файлом, о нем можно просто забыть; файл будет закрыт деструктором в конце области видимости.

Хотя указатель на FILE хранится в закрытой переменной, назвать его полностью защищенным нельзя — функция fp() возвращает этот указатель. Поскольку весь эффект от применения класса сводится к гарантированной инициализации и закрытию файла, почему бы не сделать указатель открытым или не воспользоваться структурой вместо класса? Обратите внимание: хотя вы можете получить копию f с помощью функции fp(), присвоить указателю новое значения нельзя — он находится под полным контролем класса. Но сохранение указателя, возвращаемого функцией fp(), позволит прикладному программисту присваивать значения элементам структуры и даже закрыть ее, так что защита скорее обеспечивает действительность указателя на FILE, нежели целостность содержимого структуры.

Чтобы обеспечить полную защиту, необходимо предотвратить прямой доступ к указателю на FILE со стороны пользователя. Некоторые версии обычных функций файлового ввода-вывода должны быть оформлены в виде функций класса, чтобы все, что делается с файлом средствами C, также могло делаться при помощи класса C++:

```
//: C04:Fullwrap.h
// Полная инкапсуляция файлового ввода-вывода
#ifndef FULLWRAP_H
#define FULLWRAP_H
#include <cstddef>
#include <cstdio>
#undef getc
#undef putc
#undef ungetc
using std::size_t;
using std::fpos_t;

class File {
  std::FILE* f;
  std::FILE* F(); // Возвращает проверенный указатель на f
public:
  File(); // Создает объект, но не открывает файл
  File(const char* path, const char* mode = "r");
  ~File();
  int open(const char* path, const char* mode = "r");
  int reopen(const char* path, const char* mode);
  int getc();
  int ungetc(int c);
  int putc(int c);
  int puts(const char* s);
  char* gets(char* s, int n);
  int printf(const char* format, ...);
  size_t read(void* ptr, size_t size, size_t n);
  size_t write(const void* ptr, size_t size, size_t n);
  int eof();
```

```
  int close();
  int flush();
  int seek(long offset, int whence);
  int getpos(fpos_t* pos);
  int setpos(const fpos_t* pos);
  long tell();
  void rewind();
  void setbuf(char* buf);
  int setvbuf(char* buf, int type, size_t sz);
  int error();
  void clearErr();
};
#endif // FULLWRAP_H ///:~
```

Класс содержит почти все функции файлового ввода-вывода из файла <cstdio> (отсутствует функция vprintf(), реализующая функцию printf()).

Класс File содержит такой же конструктор, как и в предыдущем примере, а также конструктор по умолчанию. Конструктор по умолчанию играет важную роль в ситуациях, когда инициализация выполняется не в конструкторе, а позднее (например, при создании массива объектов File или использовании объекта File как члена другого класса).

Конструктор по умолчанию обнуляет закрытый указатель на FILE f. Но теперь перед любыми ссылками на f необходимо проверить значение указателя и убедиться в том, что оно отлично от нуля. Задача решается с помощью функции F(), также объявленной закрытой, потому что она должна использоваться только функциями нашего класса (в этой версии класса мы не собираемся предоставлять пользователю прямой доступ к базовой структуре FILE).

В целом решение получилось вполне приличным. Оно достаточно функционально. Можно без труда представить аналогичные классы для стандартного (консольного) ввода-вывода и чтения-записи данных в памяти вместо файла или консоли.

Однако появляется неожиданный камень преткновения — интерпретатор, используемый при обработке переменных списков аргументов. Он разбирает форматную строку во время выполнения программы и интерпретирует аргументы в переменном списке. Проблемы возникают по четырем причинам.

- Хотя мы используем малую часть функциональности интерпретатора, он все равно полностью загружается в исполняемый файл. Включая в программу команду printf("%c",'x');, вы получаете весь пакет вместе с компонентами вывода вещественных чисел и строк. Не существует стандартных средств для сокращения объема памяти, используемой программой.
- Поскольку интерпретация происходит во время выполнения программы, она приводит к неизбежным затратам ресурсов. И это досадно, потому что вся информация *присутствует* в форматной строке на стадии компиляции, но не обрабатывается до стадии выполнения. Если бы разбор аргументов в форматной строке можно было выполнить на стадии компиляции, стали бы возможными прямые вызовы функций, заметно превосходящие потенциально по скорости интерпретацию (хотя семейство функций printf() обычно достаточно хорошо оптимизируется).
- Поскольку форматная строка обрабатывается лишь во время выполнения, проверка ошибок на стадии компиляции невозможна. Вероятно, вы уже стал-

кивались с этой проблемой при диагностике ошибок, возникающих из-за неверного типа или количества аргументов в командах printf(). C++ старается обнаружить как можно больше ошибок на стадии компиляции, чтобы упростить работу программиста. Обидно отказываться от безопасности типов для библиотеки ввода-вывода, особенно если учесть частоту обращения к вводу-выводу.

- Основная проблема заключается в том, что в C++ семейство функций printf() плохо поддается расширению. Эти функции проектировались для работы с базовыми типами данных C (char, int, float, double, wchar_t, char*, wchar_t* и void*) и их разновидностями. Конечно, можно попытаться добавлять в каждый новый класс перегруженные функции printf() и scanf() (и их разновидности для файлов и строк), но не забывайте: перегруженные функции должны различаться по типам аргументов, а семейство printf() скрывает информацию о типах в форматной строке и переменном списке аргументов. Одной из целей проектирования C++ была простота добавления новых типов, поэтому такое ограничение неприемлемо.

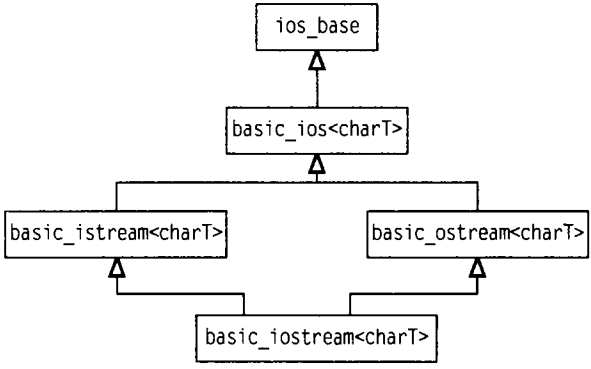
Потоки ввода-вывода

Все перечисленные проблемы очевидно показывают, что система ввода-вывода является одним из первоочередных кандидатов на включение в стандартную библиотеку классов C++. Едва ли не каждый программист начинает изучение нового языка с программы «Hello, world», а ввод-вывод требуется почти в любой программе, поэтому библиотека ввода-вывода C++ должна быть особенно удобной в использовании. Но существует другая, гораздо более серьезная проблема — библиотека ввода-вывода должна легко адаптироваться к введению новых классов. Значит, из этих ограничений следует, что архитектура библиотеки классов ввода-вывода должна быть действительно творческой. Помимо объяснения основных принципов ввода-вывода и форматирования в этой главе также представлены примеры использования этой действительно мощной библиотеки C++.

Операторы чтения и записи

Потоком данных (stream) называется объект, предназначенный для передачи и форматирования символов в фиксированном формате. Потоки данных делятся на *потоки ввода* (потомки класса istream), *потоки вывода* (потомки класса ostream) и *потоки ввода-вывода* (объекты, производные от iostream), поддерживающие оба класса операций. В библиотеку потоков ввода-вывода включены несколько специализаций этих классов: ifstream, ofstream и fstream для файлов и istringstream, ostringstream и stringstream для взаимодействия с стандартным классом string языка C++. Все перечисленные потоковые классы обладают почти одинаковым интерфейсом, поэтому операции с потоком данных практически не отличаются оттого, работаете ли вы с файлом, консолью, блоком памяти или строковым объектом. Единый интерфейс также хорошо подходит для добавления расширений, поддерживающих новые классы. Одни функции реализуют форматированный ввод-вывод, другие позволяют читать и записывать символы без форматирования.

Упомянутые потоковые классы в действительности представляют собой специализации шаблонов[1], по аналогии с тем, как стандартный класс string является специализацией шаблона basic_string. Базовые классы иерархии потоков ввода-вывода изображены на следующем рисунке:



В классе ios_base объявляется все, что относится ко всем потокам независимо от типа символов. В основном это объявления констант и функций для работы с ними (примеры встретятся вам в этой главе). Остальные классы представляют собой шаблоны, параметризованные по типу символов. Например, класс istream определяется следующим образом:

```
typedef basic_istream<char> istream;
```

Все упоминавшиеся классы также представляют собой аналогичные определения типов. Кроме того существуют определения типов для всех потоковых классов, использующих тип wchar_t (тип символов с расширенной кодировкой, о котором говорилось в главе 3) вместо char. Мы рассмотрим их в конце главы. Шаблон basic_ios определяет функции, общие для ввода и вывода, но эти функции зависят от базового типа символов (в книге они встречаются редко). Шаблон basic_istream определяет общие функции ввода, а basic_ostream делает то же самое для вывода. Классы файловых и строковых потоков определяют новые возможности для специализированных разновидностей потоков.

В библиотеке потоков ввода-вывода были перегружены два оператора, упрощающие работу с потоками данных: оператор чтения из потока >> и оператор записи в поток <<.

Оператор чтения из потока осуществляет разбор данных в соответствии с типом приемника. Для примера возьмем объект cin, потоковый аналог объекта stdin языка C, то есть перенаправляемый стандартный входной поток. Этот стандартный объект определяется при включении заголовочного файла <iostream>:

```
int i;
cin >> i;

float f;
cin >> f;

char c;
```

[1] См. главу 5.

```
cin >> c;

char buf[100];
cin >> buf;
```

Перегруженные версии оператора >> существуют для всех встроенных типов данных. Как будет показано далее, вы также можете перегружать его для своих типов.

Вывод переменных осуществляется их записью в объект cout (аналог стандартного выходного потока; также существует объект cerr для стандартного потока ошибок) с помощью оператора <<:

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

Такой синтаксис выглядит единообразно и не имеет особых преимуществ перед функцией printf(), несмотря на улучшенную проверку типов. К счастью, перегруженные операторы чтения и записи могут объединяться в более сложные выражения, которые гораздо удобнее читать (и записывать):

```
cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "c = " << c << endl;
cout << "buf = " << buf << endl;
```

Определение операторов >> и << для ваших классов сводится к простой перегрузке, которая должна делать следующее:

- первый параметр объявляется как неконстантная ссылка на поток данных (istream для ввода, ostream для вывода);
- операции выполняются записью-чтением соответствующих данных в поток или из него (по данным объекта);
- работа оператора завершается возвращением ссылки на поток.

Поток должен быть неконстантным, потому что обработка потоковых данных изменяет состояние потока. Возвращение ссылки на поток позволяет объединять потоковые операции в одной команде, как показано выше.

В качестве примера рассмотрим оператор для вывода представления объекта Date в формате ММ-ДД-ГГГГ:

```
ostream& operator<<(ostream& os, const Date& d) {
  char fillc = os.fill('0');
  os << setw(2) << d.getMonth() << '-'
     << setw(2) << d.getDay() << '-'
     << setw(4) << setfill(fillc) << d.getYear();
  return os;
}
```

Эта операторная функция не может быть членом класса Date, потому что левый операнд оператора << должен быть потоком вывода. Функция fill() класса ostream изменяет символ-заполнитель, используемый, если ширина поля, определяемая *манипулятором* setw(), превышает размер выводимых данных. Мы выбираем символ «0», чтобы месяцы до октября отображались с начальным нулем (например, 09 для сентября). Функция fill() также возвращает предыдущий заполнитель (пробел по умолчанию), чтобы позднее его можно было восстановить манипулятором setfill(). Манипуляторы будут подробно описаны далее.

С операторами чтения дело обстоит сложнее, потому что при чтении данных возможны ошибки. Сигналы об ошибках потоков подаются при помощи флага failbit, как показано в следующем примере:

```
istream& operator>>(istream& is, Date& d) {
  is >> d.month;
  char dash;
  is >> dash;
  if (dash != '-')
    is.setstate(ios::failbit);
  is >> d.day;
  is >> dash;
  if (dash != '-')
    is.setstate(ios::failbit);
  is >> d.year;
  return is;
}
```

Если установить для потока бит ошибки, все последующие операции с потоком игнорируются до тех пор, пока поток не будет восстановлен (об этом чуть позже). Именно поэтому оператор продолжает читать данные, не проверяя ios::failbit. Такая реализация допускает присутствие пропусков между числами и дефисами в строке даты (так как оператор >> по умолчанию игнорирует пропуски при чтении встроенных типов). Следующие строки являются допустимыми для данного оператора:

```
"08-10-2003"
"8-10-2003"
"08 - 10 - 2003"
```

А вот эти строки недопустимы:

```
"A-10-2003"    // Алфавитные символы запрещены
"08%10/2003"   // Разделителями могут быть только дефисы
```

Состояние потока более подробно рассматривается в разделе «Обработка потоковых ошибок» этой главы.

Типичное применение

Как показывает оператор чтения для класса Date, следует учитывать возможность ошибок при вводе. Если прочитанные данные не соответствуют типу переменной, весь процесс нарушается, и восстановить нормальную работу программы будет нелегко. Кроме того, форматированный ввод по умолчанию разделяется пропусками. Посмотрим, что произойдет, если собрать приведенные выше фрагменты в программу:

```
//: C04:Iosexamp.cpp
// Пример работы с потоками ввода-вывода
```

```
#include <iostream>
using namespace std;

int main() {
  int i;
  cin >> i;

  float f;
  cin >> f;

  char c;
  cin >> c;

  char buf[100];
  cin >> buf;

  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;

  cout << flush;
  cout << hex << "0x" << i << endl;
} ///:~
```

Попробуем передать этой программе такой набор входных данных:

```
12 1.4 c this is a test
```

Казалось бы, данные должны распределиться следующим образом:

```
12
1.4
c
this is a test
```

Однако тест дает несколько неожиданный результат:

```
i = 12
f = 1.4
c = c
buf = this
0xc
```

Переменной buf достается только первое слово, потому что функция ввода считает пробел после «this» признаком конца ввода. Кроме того, если непрерывная входная строка превысит объем блока памяти, выделенного для buf, произойдет переполнение буфера.

На практике обычно бывает проще получить от интерактивной программы строку в виде последовательности символов, сканировать ее и выполнить необходимые преобразования, после того как строка окажется в буфере. В этом случае вам не придется беспокоиться о том, что функция ввода «подавится» непредвиденными данными.

Другой важный фактор — сама концепция интерфейса командной строки. Она была вполне оправданна в прошлом, когда консоль была обычной комбинацией пишущей машинки и телевизора, но мир быстро меняется, и сейчас в нем преобладают графические пользовательские интерфейсы (Graphical User Interface, GUI). Найдется ли место консольному вводу-выводу в таком мире? Гораздо разумнее

ограничиться применением объекта cin для простейших примеров или тестов, и выбрать один из перечисленных ниже вариантов.

- Прочитать входные данные из файла — вскоре вы убедитесь, что потоки ввода-вывода чрезвычайно упрощают работу с файлами. Файловые потоки нормально работают в графических средах.
- Прочитать входные данные, не пытаясь преобразовывать их, как предлагалось ранее. Когда входные данные окажутся в надежном месте, где преобразование не сможет ничего испортить, их можно спокойно отсканировать.
- С выводом дело обстоит иначе. В графических средах объект cout работает не всегда, поэтому вывод приходится посылать в файл (средствами, идентичными выводу в cout) или использовать механизмы вывода графической среды. Впрочем, если объект cout нормально работает, для вывода данных смысл имеет задействовать его. В любом случае потоковые функции форматирования вывода приносят несомненную пользу.

Стоит упомянуть и другой распространенный прием, сокращающий время компиляции больших проектов. Подумайте, как бы вы объявили упоминавшиеся ранее потоковые операторы Date в заголовочном файле? Необходимо объявить только прототипы функций, поэтому включать весь заголовок <iostream> в файл Date.h не обязательно. В обычной ситуации вы бы использовали только объявление класса:

```
class ostream;
```

Это классический принцип отделения интерфейса от реализации, часто называемый *опережающим объявлением* (а ostream в этой точке программы рассматривается как *неполный тип*, потому что компилятор еще не «видел» определения класса).

Однако такое объявление не будет работать по двум причинам:

- потоковые классы определяются в пространстве имен std;
- потоковые классы представляют собой шаблоны.

Правильное объявление выглядит так:

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
    class basic_ostream;
  typedef basic_ostream<char> ostream;
}
```

Как видите, потоковые классы используют классы характеристик, упоминавшиеся в главе 3. Набирать этот текст для каждого потокового класса, на который вы хотите сослаться, было бы крайне утомительно, поэтому в стандарте определяется заголовочный файл <iosfwd>, который делает это за вас. Тогда заголовок Date принимает следующий вид:

```
// Date.h
#include <iosfwd>

class Date {
  friend std::ostream& operator<<(std::ostream&, const Date&);
  friend std::istream& operator>>(std::istream&, Date&);
  // И т. д.
```

Построчный ввод

Для получения одной строки входных данных у вас имеются три инструмента:

- функция класса get();
- функция класса getline();
- глобальная функция getline(), определенная в заголовочном файле <string>.

Первые две функции получают три аргумента:

- указатель на символьный буфер для хранения результата;
- размер буфера (для контроля переполнения);
- завершающий символ, по которому прекращается чтение данных.

По умолчанию завершающим символом является символ '\n' (конец строки); обычно именно он чаще всего используется на практике. Встречая завершающий символ во входных данных, обе функции записывают ноль в выходной буфер.

Тогда чем же они отличаются? Отличие тонкое, но важное: встречая завершитель, функция get() останавливается, но не извлекает его из входного потока. Таким образом, если немедленно вызвать get() с тем же завершителем, функция вернется, не прочитав ни одного символа (поэтому придется либо вызвать другую функцию, либо get() с другим завершителем). С другой стороны, функция getline() извлекает завершитель из входного потока, хотя и не сохраняет его в буфере.

Функция getline(), определенная в заголовочном файле <string>, весьма удобна. Она не принадлежит какому-либо классу, а является автономной функцией, объявленной в пространстве имен std. Ей передаются только два обязательных аргумента: входной поток и заполняемый объект string. Как и одноименная функция класса, она читает символы до обнаружения первого вхождения завершителя (по умолчанию '\n') и поглощает завершитель. Преимущество этой функции состоит в том, что данные читаются в объект string, поэтому вам не придется беспокоиться о размере буфера.

Как правило, при построчном чтении текстовых файлов используется одна из разновидностей функции getline().

Перегруженные версии get()

Функция get() также существует в трех перегруженных версиях. Первая версия вызывается без аргументов и возвращает следующий символ в виде int; вторая версия сохраняет символ в аргументе char по ссылке; третья версия сохраняет символ в буфере другого потока ввода-вывода. Последняя возможность будет рассмотрена в этой главе.

Неформатированный ввод

Если вы точно знаете, с какими данными работаете, и хотите напрямую записать байты в переменную, массив или структуру памяти, воспользуйтесь функцией неформатированного ввода-вывода read(). Первый аргумент этой функции содержит указатель на приемный адрес, а второй — количество читаемых байтов. Функция особенно удобна при загрузке данных, предварительно сохраненных в файле (например, парной функцией write() выходного потока — конечно, для того же компилятора). Примеры использования всех этих функций будут приведены далее.

Обработка потоковых ошибок

Как уже отмечалось, оператор чтения данных типа Date в некоторых ситуациях устанавливает флаг ошибочного состояния потока. Как пользователь узнает о таких ошибках? Либо вызывая специальные потоковые функции класса, либо (если его не интересует конкретный тип ошибки) просто проверяя поток в логическом контексте. Оба способа зависят от состояния флагов ошибочного состояния потока.

Состояние потока

В классе ios_base, производным от которого является ios[1], определены четыре флага для проверки состояния потока.

`badbit`

Фатальная (возможно, физическая) ошибка. Поток непригоден для дальнейшего использования.

`eofbit`

Конец входных данных (физическое завершение файлового потока или завершение консольного потока пользовательским вводом, например нажатием клавиш Ctrl+Z или Ctrl+D).

`failbit`

Операция ввода-вывода завершилась неудачей, вероятнее всего, из-за недопустимых данных (например, обнаружены буквы при чтении числа). Поток остается пригодным к использованию. Флаг failbit также устанавливается при обнаружении конца ввода.

`goodbit`

Все нормально, ошибок нет. Конец входных данных еще не обнаружен.

Вы всегда можете узнать о наступлении любого из этих условий при помощи функций потоковых классов; функция возвращает логический признак установки соответствующего флага. Функция good() потокового класса возвращает true, если не установлен ни один из трех флагов аномальных состояний. Функция eof() возвращает true при установленном флаге eofbit, что происходит при попытке чтения из потока без данных (обычно из файла). Поскольку конец ввода в C++ обнаруживается по преждевременному исчерпанию данных, также устанавливается флаг failbit — признак того, что «ожидаемые» данные не были успешно прочитаны. Функция fail() возвращает true при установке флага failbit *или* badbit, а bad() возвращает true только при установленном флаге badbit.

После установки флаги аномальных состояний потока остаются активными, что иногда оказывается нежелательным. Допустим, при чтении файла до обнаружения конца файла вы хотите вернуться к более ранней позиции. Простое перемещение указателя не приводит к автоматическому сбросу флагов eofbit и failbit. Флаги приходится сбрасывать вручную с помощью функции clear():

```
myStream.clear();  // Сбрасывает все биты аномальных состояний
```

Если после clear() немедленно вызвать функцию good(), она вернет true. Как было показано при описании оператора чтения Date, функция setstate() устанавли-

[1] Вместо ios_base::failbit для краткости обычно используется запись ios::failbit.

вает заданные флаги. На другие флаги ее вызов не влияет — ранее установленные флаги остаются установленными. Чтобы установить некоторые флаги с одновременным сбросом всех остальных, воспользуйтесь перегруженной версией clear(), получающей логическое выражение с указанием устанавливаемых битов:

```
myStream.clear(ios::failbit | ios::eofbit);
```

Как правило, проверка отдельных флагов аномального состояния потока выполняется относительно редко. Чаще пользователь просто желает убедиться в том, что все идет нормально. Например, читая файл от начала до конца, вы просто хотите узнать о достижении конца файла. В таких случаях можно воспользоваться функцией преобразования для типа void*, которая автоматически вызывается при включении объекта потока в логическое выражение. Чтение до конца входных данных в этой идиоме выполняется так:

```
int i;
while(myStream >> i)
   cout << i << endl;
```

Помните, что функция operator>>() возвращает ссылку на поток, переданный в аргументе, поэтому условие while проверяет объект потока в логическом выражении. Этот конкретный пример предполагает, что входной поток myStream содержит целые числа, разделенные пропусками. Функция ios_base::operator void*() просто вызывает good() для своего потока и возвращает результат[1]. Поскольку большинство потоковых операций возвращает свои потоки, эта идиома часто применяется на практике.

Потоки ввода-вывода и исключения

Потоки ввода-вывода появились в C++ гораздо раньше исключений, поэтому в прежние времена состояние потока проверялось только «вручную», то есть вызовом проверочных функций. Для сохранения совместимости именно эта возможность считается основной, но современные реализации потоков ввода-вывода могут сообщать об ошибках при помощи исключений. Функции класса потока exceptions() передается параметр, определяющий флаги состояния, для которых должно запускаться исключение. При каждом переходе в такое состояние поток запускает исключение типа std::ios_base::failure, производное от std::exception.

Хотя исключения могут генерироваться для всех четырех состояний потоков, разрешать все четыре исключения не всегда разумно. Как объяснялось в главе 1, исключения следует применять в действительно исключительных состояниях, однако состояние конца файла не только *не является* исключительным — его появление *ожидается*! По этой причине исключения обычно разрешаются только для ошибок, представленных флагом badbit:

```
myStream.exceptions(ios::badbit);
```

Запуск исключений разрешается для каждого потока по отдельности, так как функция exceptions() является функцией потоковых классов. Функция exceptions()

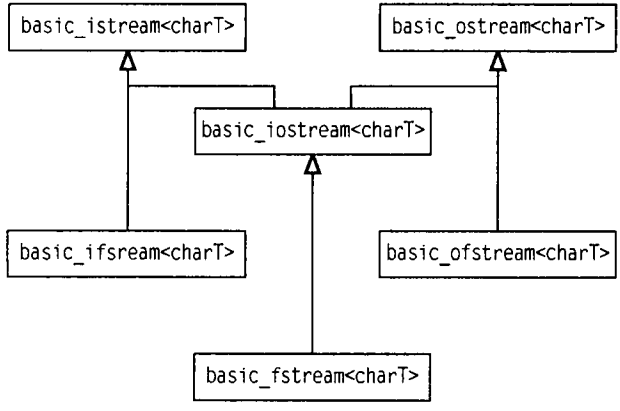
[1] Обычно вместо operator bool() используется operator void*(), потому что неявные преобразования bool в int могут приводить к неожиданностям, если поток задействован в контексте, допускающем преобразование к целому типу. Операторная функция operator void*() неявно вызывается только в логических выражениях.

возвращает битовую маску[1] (iostate — некоторого типа, преобразуемого к int; конкретный выбор зависит от компилятора), указывающую, какие состояния потоков будут порождать исключения. Если соответствующие флаги уже были установлены, исключение запускается немедленно. Конечно, чтобы использовать исключения с потоками, нужно подготовиться к их перехвату. Отсюда следует, что все операции с потоками должны выполняться внутри блока try с обработчиком ios::failure. Многие программисты считают, что это неудобно, и вручную проверяют состояние потока там, где предполагаются ошибки (в частности, предполагается, что функция bad() крайне редко возвращает true). Это другая причина, по которой механизм обработки исключений считается дополнительным, а не основным средством передачи информации об ошибках. Так или иначе, вы сами выбираете более удобный способ. По тем же причинам, по которым мы рекомендуем применять исключения для обработки ошибок в других контекстах, мы будем использовать их здесь.

Файловые потоки ввода-вывода

Работать с файлами при помощи потоков ввода-вывода гораздо проще и безопаснее, чем с применением библиотеки stdio языка C. Чтобы открыть файл, достаточно создать в программе объект — конструктор берет на себя всю работу. Закрывать файлы необязательно (хотя это можно сделать вызовом функции close()), поскольку деструктор автоматически закроет файл при выходе объекта из области видимости. Чтобы создать объект файла, по умолчанию предназначенного для чтения данных, создайте объект ifstream; объект файла для записи представляется объектом ofstream. Объекты fstream позволяют выполнять как чтение, так и запись.

Место классов файловых потоков в общей иерархии ввода-вывода показано на следующем рисунке:



Как и прежде, фактически используемые классы представляют собой специализации шаблонов для определенных типов. Например, класс ifstream, предназначенный для обработки файлов char, определяется следующим образом:

```
typedef basic_ifstream<char> ifstream;
```

[1] Целочисленный тип, предназначенный для хранения набора битовых флагов.

Пример обработки файлов

Следующий пример демонстрирует многие возможности, упоминавшиеся до настоящего момента. Обратите внимание на включение заголовка <fstream> с объявлениями классов файлового ввода-вывода. Хотя на многих платформах компилятор также автоматически включает заголовок <iostream>, он не обязан это делать. Чтобы программа была действительно переносимой, всегда включайте оба заголовка.

```
//: C04:Strfile.cpp
// Потоковый ввод-вывод на примере файлов
// Демонстрация различий между get() и getline()
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  const int sz = 100; // Размер буфера
  char buf[sz];
  {
    ifstream in("Strfile.cpp"); // Чтение
    assure(in, "Strfile.cpp"); // Проверка успешного открытия
    ofstream out("Strfile.out"); // Запись
    assure(out, "Strfile.out");
    int i = 1; // Line counter

    // Неудобный способ построчного ввода:
    while(in.get(buf, sz)) { // \n остается в прочитанных данных
      in.get(); // Отбросить следующий символ (\n)
      cout << buf << endl; // Необходимо добавить \n
      // Вывод в файл напоминает стандартный вывод:
      out << i++ << ": " << buf << endl;
    }
  } // Деструкторы закрывают файлы in и out

  ifstream in("Strfile.out");
  assure(in, "Strfile.out");
  // Более удобный способ построчного ввода:
  while(in.getline(buf, sz)) { // Удаляет \n
    char* cp = buf;
    while(*cp != ':')
      cp++;
    cp += 2; // Пропустить ": "
    cout << cp << endl; // Все равно необходимо добавить \n
  }
} ///:~
```

Сразу же за созданием объектов ifstream и ofstream следуют вызовы assure(), которые проверяют, что файл был успешно открыт. В этом случае объект, использованный в контексте логического выражения, возвращает логический признак успеха или неудачи.

Первый цикл while демонстрирует две формы функции get(). Первая форма читает символы в буфер и помещает в него нуль-символ либо по прочтении SZ-1 символов, либо при обнаружении завершителя, указанного в третьем аргументе ('\n' по умолчанию). Функция get() оставляет завершитель во входном потоке,

поэтому его приходится отдельно удалять вызовом in.get() без аргументов, когда один байт читается и возвращается в виде int. Также можно воспользоваться функцией класса ignore(), имеющей два аргумента. В первом аргументе передается количество игнорируемых символов (по умолчанию 1). Второй аргумент определяет символ, при обнаружении (и после извлечения) которого функция ignore() завершает работу. По умолчанию этот символ равен EOF.

Дальше идут две команды, внешне очень похожие: одна выводит данные в cout, а другая — в out. Такое сходство чрезвычайно удобно: вам не приходится беспокоиться о типе объекта, поскольку команды форматирования одинаково работают для всех объектов ostream. Первая команда дублирует строку в стандартном выходном потоке, а вторая записывает строку в новый файл с включением номера строки.

Чтобы продемонстрировать работу функции getline(), мы открываем только что созданный файл и удаляем из него номера строк. Но как гарантировать, что файл будет должным образом закрыт перед его открытием для чтения? Есть два способа. Можно заключить первую часть программы в фигурные скобки; объект out выходит из области видимости, для него вызывается деструктор, и файл автоматически закрывается. Именно этот вариант использован в нашем примере. Кроме того, можно вызвать функцию close() для обоих файлов; это даже позволит заново использовать объект in вызовом функции open().

Второй цикл while показывает, как функция getline() удаляет символы-завершители (третий аргумент, по умолчанию '\n') из входного потока. Хотя функция getline(), как и get(), заносит в буфер ноль, символ-завершитель в буфере не сохраняется.

В этом примере, как и в большинстве примеров этой главы, предполагается, что каждый вызов перегруженной версии getline() находит символ перевода строки. Если это условие нарушается, для потока будет установлен флаг конца данных, и вызов getline() вернет false; в результате программа потеряет последнюю строку входных данных.

Режимы открытия файла

Вы можете выбрать нужный режим открытия файлов, передавая конструктору нужное значение вместо аргументов по умолчанию. Далее перечислены флаги, управляющие режимом открытия файла.

`ios::in`

Открывает файл для чтения. Флаг используется при открытии ofstream для предотвращения усечения существующих файлов.

`ios::out`

Открывает файл для записи. При использовании для ofstream без флага ios::app, ios::ate или ios::in, подразумевается ios::trunc.

`ios::app`

Выходной файл открывается только для присоединения данных.

`ios::ate`

Открывает существующий файл (для чтения или записи) и устанавливает текущую позицию в конец файла.

`ios::trunc`

Усекает содержимое существующих файлов.

`ios::binary`

Открывает файл в *двоичном режиме*. По умолчанию используется *текстовый режим*.

Флаги объединяются поразрядным оператором |.

Флаг двоичного режима является переносимым, но влияет только на открытие файлов в некоторых системах, не входящих в семейство Unix, в частности, в клонах MS-DOS, в которых применяются особые правила хранения завершителей строк. Например, в системах MS-DOS в текстовом режиме (используемом по умолчанию) при каждой записи в поток символа перевода строки ('\n') файловая система на самом деле выводит два физических символа с ASCII-кодами 0x0D (CR, возврат курсора) и 0x0A (LF, перевод строки). Соответственно, при чтении такого файла в память в текстовом режиме каждое вхождение этой пары байтов заменяется символом '\n', который передается программе. Чтобы предотвратить эту замену, откройте файл в двоичном режиме. Двоичный режим не имеет никакого отношения к *возможности* записи произвольных байтов в файл — такая возможность присутствует всегда (посредством вызова write()). Тем не менее, при использовании функций read() и write() следует открывать файлы именно в двоичном режиме, потому что в параметрах этим функциям передается количество байтов, которое будет искажено присутствием дополнительных символов. Если вы собираетесь задействовать команды позиционирования, описанные далее в этой главе, файлы также следует открывать в двоичном режиме.

Объявление объекта fstream позволяет открыть файл для чтения и записи. При этом необходимо задать достаточное количество флагов режима, чтобы файловая система знала, будет ли файл использоваться для чтения и/или записи. Чтобы переключиться с вывода на ввод, необходимо либо записать текущее содержимое буфера, либо изменить текущую позицию в файле. Чтобы переключиться с ввода на вывод, измените текущую позицию. Если вы хотите создать файл при объявлении объекта fstream, укажите режим открытия ios::trunc при вызове конструктора.

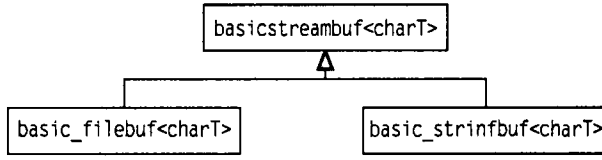
Буферизация

Одно из основных правил разработки гласит, что при создании нового класса необходимо по возможности скрыть подробности реализации от пользователя класса. Пользователь видит только то, что необходимо для его работы, а остальные члены класса объявляются закрытыми. Используя операторы << и >>, вы обычно не знаете (да вам и не нужно знать), с чем именно вы работаете: с консолью, файлом, блоком памяти, каким-то новым классом или устройством.

Но в какой-то момент возникает необходимость в том компоненте потока ввода-вывода, который непосредственно выдает или принимает байты. Чтобы предоставить единый интерфейс к этому компоненту, в стандартной библиотеке он абстрагируется в отдельный класс streambuf. Любой объект потока содержит указатель на некоторый объект streambuf (конкретный тип зависит от того, с чем работает поток: с консолью, файлом, памятью и т. д.). С объектом streambuf можно

обращаться напрямую, например, напрямую читать и записывать байты без их форматирования потоком. Для этой цели используются функции объекта streambuf.

Главное, что вам нужно знать на данный момент, — что каждый объект потока содержит указатель на объект streambuf, а объект streambuf содержит функции, которые могут вызываться в случае необходимости. Для файловых и строковых потоков определены специальные типы потоковых буферов, изображенные на следующей диаграмме:



Указатель на объект streambuf потока возвращается с помощью функции rdbuf() класса потока. Через полученный указатель можно вызывать любые функции streambuf. Но из всех операций, которые могут выполняться с указателем на streambuf, наибольший интерес представляет возможность присоединения указателя к другому потоку ввода-вывода оператором <<. В результате все символы из потока, расположенного справа от <<, передаются в поток, расположенный в левой части выражения. Если потребуется переместить все символы из одного потока в другой, вам не придется прибегать к утомительному (и чреватому ошибками) процессу посимвольного или построчного чтения. Такое решение гораздо элегантнее.

Следующая программа открывает файл и передает все его содержимое в стандартный выходной поток (по аналогии с предыдущим примером):

```
//: C04:Stype.cpp
// Направление файла в стандартный выходной поток
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Stype.cpp");
  assure(in, "Stype.cpp");
  cout << in.rdbuf(); // Вывод всего файла
} ///:~
```

Объект ifstream создается на базе файла с исходным кодом программы. Функция assert() сообщает о неудачной попытке открытия файла. Вся настоящая работа выполняется следующей командой, передающей все содержимое файла в cout:

```
cout << in.rdbuf();
```

Программа получается не только более компактной, но обычно и более эффективной по сравнению с побайтовым копированием символов.

Одна из форм функции get() записывает данные непосредственно в объект streambuf другого потока. Первым аргументом этой функции является ссылка на приемный объект streambuf, а вторым — символ-завершитель, прерывающий работу get() ('\n' по умолчанию). Так что можно рассмотреть еще один способ направления содержимого файла в стандартный выходной поток:

```
//: C04:Sbufget.cpp
// Копирование файла в стандартный выходной поток
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Sbufget.cpp");
  assure(in);
  streambuf& sb = *cout.rdbuf();
  while (!in.get(sb).eof()) {
    if (in.fail())          // Обнаружена пустая строка
      in.clear();
    cout << char(in.get()); // Обработка '\n'
  }
} ///:~
```

Функция rdbuf() возвращает указатель, поэтому для получения объекта результат ее вызова необходимо разыменовать. Копирование потоковых буферов не предусмотрено (у них отсутствует копирующий конструктор), поэтому мы определяем sb как *ссылку* на потоковый буфер cout. Вызовы fail() и clear() необходимы на случай, если во входном файле присутствует пустая строка (в нашем примере она есть). Когда эта конкретная перегруженная версия get() встречает два символа перевода строки подряд (признак пустой строки), она устанавливает флаг failbit для входного потока. Мы должны сбросить этот флаг функцией clear(), чтобы продолжить чтение из потока. Второй вызов get() извлекает и воспроизводит каждый завершитель строки (помните, что функция get(), в отличие от getline(), не извлекает из потока символ-завершитель).

Вряд ли вам придется часто использовать этот прием, но все равно полезно знать, что он существует.

Поиск в потоках ввода-вывода

У каждого потока имеется свое представление о том, в какой позиции будет прочитан (для istream) или записан (для ostream) «следующий» символ. Иногда эту «текущую» позицию требуется сместить, что можно сделать двумя способами. В первом способе задается абсолютная позиция в потоке (streampos). Второй способ работает по аналогии с библиотечной функцией fseek() языка C для файлов: смещение производится на заданное количество байтов от начала, конца или текущей позиции в файле.

Вариант со streampos требует предварительного вызова специальной функции: tellp() для ostream или tellg() для istream. Функция возвращает объект streampos, который передается при вызове функции seekp() для ostream или seekg() для istream. Заданная позиция становится текущей позицией потока.

Во втором варианте с относительным позиционированием используются перегруженные версии seekp() и seekg(). Первый аргумент содержит смещение в символах, положительное или отрицательное. Второй аргумент определяет базу смещения:

ios::beg

От начала потока.

`ios::cur`

От текущей позиции потока.

`ios::end`

От конца потока.

Следующий пример демонстрирует позиционирование в файле. Помните, что позиционирование в потоках не ограничивается файлами, как в библиотеке stdio языка C. C++ позволяет выполнять позиционирование в любых потоках ввода-вывода (хотя для стандартных потоков, таких как cin и cout, позиционирование запрещено):

```
//: C04:Seeking.cpp
// Позиционирование в потоках ввода-вывода
#include <cassert>
#include <cstddef>
#include <cstring>
#include <fstream>
#include "../require.h"
using namespace std;

int main() {
  const int STR_NUM = 5, STR_LEN = 30;
  char origData[STR_NUM][STR_LEN] = {
    "Hickory dickory dus. . .",
    "Are you tired of C++?",
    "Well, if you have,",
    "That's just too bad.",
    "There's plenty more for us!"
  };
  char readData[STR_NUM][STR_LEN] = {{ 0 }};
  ofstream out("Poem.bin", ios::out | ios::binary);
  assure(out, "Poem.bin");
  for(size_t i = 0; i < STR_NUM; i++)
    out.write(origData[i], STR_LEN);
  out.close();
  ifstream in("Poem.bin", ios::in | ios::binary);
  assure(in, "Poem.bin");
  in.read(readData[0], STR_LEN);
  assert(strcmp(readData[0], "Hickory dickory dus. . .")
    == 0);
  // Смещение на -STR_LEN байтов от конца файла
  in.seekg(-STR_LEN, ios::end);
  in.read(readData[1], STR_LEN);
  assert(strcmp(readData[1], "There's plenty more for us!")
    == 0);
  // Абсолютное позиционирование (по аналогии с operator[])
  in.seekg(3 * STR_LEN);
  in.read(readData[2], STR_LEN);
  assert(strcmp(readData[2], "That's just too bad.") == 0);
  // Смещение в обратном направлении от текущей позиции
  in.seekg(-STR_LEN * 2, ios::cur);
  in.read(readData[3], STR_LEN);
  assert(strcmp(readData[3], "Well, if you have,") == 0);
  // Позиционирование от начала файла
  in.seekg(1 * STR_LEN, ios::beg);
  in.read(readData[4], STR_LEN);
  assert(strcmp(readData[4], "Are you tired of C++?")
    == 0);
} ///:~
```

Программа записывает стихотворение в файл, используя двоичный поток вывода. Поскольку файл открывается заново через объект ifstream, для позиционирования вызывается функция seekg(). Как видите, позиционирование может выполняться от начала, от конца файла или от текущей позиции в файле. Естественно, для позиционирования от начала файла смещение должно быть положительным, а для позиционирования от конца файла — отрицательным.

Зная, что такое streambuf и как смещается текущая позиция, вы сможете разобраться в альтернативном способе создания объекта потока, обеспечивающего чтение и запись в файл (без использования fstream). Следующая программа создает объект ifstream с флагами, указывающими, что файл будет задействован для чтения и записи. Запись в ifstream невозможна, поэтому нам придется создать поток ostream с тем же потоковым буфером:

```
ifstream in("filename", ios::in | ios::out );
ostream out(in.rdbuf());
```

Что же произойдет при записи в один из этих объектов? Пример:

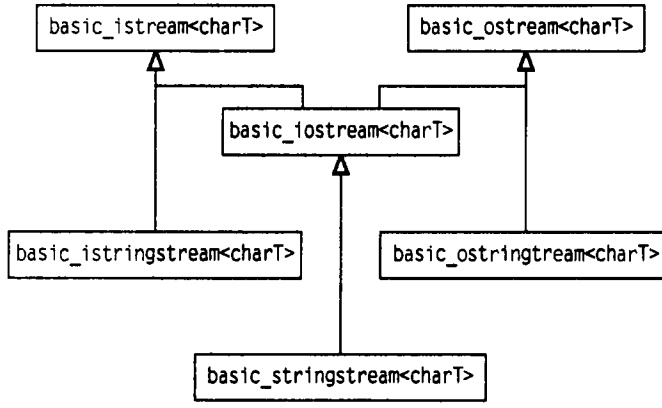
```
//: C04:Iofile.cpp
// Чтение и запись в файл.
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Iofile.cpp");
  assure(in, "Iofile.cpp");
  ofstream out("Iofile.out");
  assure(out, "Iofile.out");
  out << in.rdbuf(); // Копирование файла
  in.close();
  out.close();
  // Открытие для чтения и записи:
  ifstream in2("Iofile.out", ios::in | ios::out);
  assure(in2, "Iofile.out");
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();  // Вывод всего файла
  out2 << "Where does this end up?";
  out2.seekp(0, ios::beg);
  out2 << "And what about this?";
  in2.seekg(0, ios::beg);
  cout << in2.rdbuf();
} ///:~
```

В первых пяти строках исходный код программы копируется в файл с именем iofile.out, после чего файлы закрываются. Так мы получаем текстовый файл для безопасных экспериментов. Затем мы применяем упоминавшуюся методику для создания двух объектов, читающих и записывающих данные в один файл. В команде cout<<in2.rdbuf() «позиция чтения» инициализируется позицией начала файла. С другой стороны, «позиция записи» устанавливается в конец файла, чтобы строка "Where does this end up?" была присоединена к файлу. Если переместить «позицию записи» в начало файла функцией seekp(), выводимый текст будет записываться *поверх* существующего текста. Чтобы наглядно продемонстрировать эффект обеих операций вывода, мы перемещаем «позицию чтения» в начало файла функцией seekg() и выводим содержимое файла. Файл автоматически закрывается при выходе out2 из области видимости и вызове деструктора.

Строковые потоки

Строковые потоки работают не с файлами и не с консолью, а непосредственно с памятью. Они используют те же функции чтения и форматирования, что и cin с cout, для работы с байтами в памяти.

Имена классов строковых потоков образуются по тому же принципу, что и имена файловых потоков. Если вы хотите создать строковый поток для чтения символов, создайте объект istringstream. Если строковый поток предназначен для вывода символов, создайте объект ostringstream. Все объявления строковых потоков собраны в стандартном заголовочном файле <sstream>. Шаблоны классов образуют иерархию, изображенную на следующей диаграмме:



Строковые потоки ввода

Чтобы читать данные из строки с использованием потоковых операций, создайте объект istringstream, инициализированный строкой. Следующая программа демонстрирует работу с объектом istringstream:

```
//: C04:Istring.cpp
// Строковые потоки ввода
#include <cassert>
#include <cmath>  // Для fabs()
#include <iostream>
#include <limits> // Для epsilon()
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream s("47 1.414 This is a test");
  int i;
  double f;
  s >> i >> f; // Входные данные разделяются пропусками
  assert(i == 47);
  double relerr = (fabs(f) - 1.414) / 1.414;
  assert(relerr <= numeric_limits<double>::epsilon());
  string buf2;
  s >> buf2;
```

```
  assert(buf2 == "This");
  cout << s.rdbuf(); // " is a test"
} ///:~
```

Такой способ преобразования символьных строк в типизованные значения оказывается более гибким и универсальным, чем функции atof() и atoi() стандартной библиотеки C, хотя для одиночных преобразований последний вариант может оказаться более эффективным.

В выражении s>>i>>f первое число читается в переменную i, а второе — в переменную f. Данные отбираются не по критерию «первой группы символов, ограниченной пропусками», потому что все зависит от типа данных читаемой переменной. Например, для строки "1.414 47 This is a test" переменной i было бы присвоено значение 1, поскольку чтение остановилось бы на десятичной точке. Далее переменной f было бы присвоено значение 0.414. Такое деление хорошо подходит для разбиения вещественного числа на целую и дробную части, но в остальных ситуациях оно больше похоже на ошибку. Второй вызов assert() вычисляет относительное расхождение между прочитанными и ожидаемыми данными; всегда лучше использовать этот способ вместо прямого сравнения вещественных чисел. Константа, возвращаемая функцией epsilon(), определяемой в файле <limits>, представляет *машинный эпсилон* для чисел с двойной точностью, то есть максимальное допустимое отклонение при сравнениях типов double[1].

Вероятно, вы уже догадались, что переменной buf2 достается не остаток строки, а только следующее слово, ограниченное пропусками. В общем случае лучше использовать оператор >>, если вам известна точная последовательность данных во входном потоке, а преобразование осуществляется к типу, отличному от символьной строки. Впрочем, если потребуется прочитать сразу весь остаток строки и передать его в другой поток ввода-вывода, используйте функцию rdbuf(), как показано далее.

При тестировании оператора >> для типа Date, представленного в начале главы, использовался строковый поток ввода со следующей тестовой программой:

```
//: C04:DateIOTest.cpp
//{L} ../C02/Date
#include <iostream>
#include <sstream>
#include "../C02/Date.h"
using namespace std;

void testDate(const string& s) {
  istringstream os(s);
  Date d;
  os >> d;
  if (os)
    cout << d << endl;
  else
    cout << "input error with \"" << s << "\"\n";
}

int main() {
```

[1] За дополнительной информацией о допустимых отклонениях и вычислениях с плавающей запятой обращайтесь к статье «The Standard C Library, Part 3» в *C/C++ Users Journal*, март 1995 г., по адресу www.freshsources.com/1995006a.htm.

```
  testDate("08-10-2003");
  testDate("8-10-2003");
  testDate("08 - 10 - 2003");
  testDate("A-10-2003");
  testDate("08%10/2003");
} ///:~
```

Строковые литералы в main() передаются по ссылке функции testDate(), которая, в свою очередь, упаковывает их в объекты istringstream. На этих объектах проверяется работа операторов чтения, написанных нами для объектов Date. Функция testDate() также в первом приближении проверяет работу оператора записи данных operator<<().

Строковые потоки вывода

Строковые потоки вывода представлены объектами ostringstream. Объект содержит символьный буфер с динамически изменяемыми размерами, в котором хранятся все вставленные символы. Чтобы получить отформатированный результат в виде объекта string, вызовите функцию str(). Пример:

```
//: C04:Ostring.cpp
// Строковые потоки вывода
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
  cout << "type an int, a float and a string: ";
  int i;
  float f;
  cin >> i >> f;
  cin >> ws; // Игнорировать пропуски
  string stuff;
  getline(cin, stuff); // Получение остатка строки
  ostringstream os;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << "string = " << stuff << endl;
  string result = os.str();
  cout << result << endl;
} ///:~
```

В целом программа напоминает пример Istring.cpp, в котором из потока читались числа int и float. Ниже приводится примерный результат (ввод с клавиатуры выделен полужирным шрифтом):

```
type an int, a float and a string: 10 20.5 the end
integer = 10
float = 20.5
string = the end
```

При передаче байтов в ostringstream применяются те же средства, что и для других потоков вывода: оператор << и манипулятор endl. Функция str() при каждом вызове возвращает новый объект string, поэтому базовый объект stringbuf, принадлежащий строковому потоку, остается в прежнем состоянии.

В предыдущей главе была представлена программа HTMLStripper.cpp, удалявшая из текстового файла теги HTML и специальные символы. Тогда мы пообещали, что представим более элегантное решение с использованием строковых потоков:

```
//: C04:HTMLStripper2.cpp {RunByHand}
//{L} ../C03/ReplaceAll
// Фильтр для удаления тегов и маркеров HTML
#include <cstddef>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include "../C03/ReplaceAll.h"
#include "../require.h"
using namespace std;

string& stripHTMLTags(string& s) throw(runtime_error) {
  size_t leftPos;
  while ((leftPos = s.find('<')) != string::npos) {
    size_t rightPos = s.find('>', leftPos+1);
    if (rightPos == string::npos) {
      ostringstream msg;
      msg << "Incomplete HTML tag starting in position "
          << leftPos;
      throw runtime_error(msg.str());
    }
    s.erase(leftPos, rightPos - leftPos + 1);
  }
  // Удаление всех специальных символов HTML
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
  replaceAll(s, " ", " ");
  // И т. д.
  return s;
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper2 InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  // Прочитать весь файл в строку и удалить HTML
  ostringstream ss;
  ss << in.rdbuf();
  try {
    string s = ss.str();
    cout << stripHTMLTags(s) << endl;
    return EXIT_SUCCESS;
  }
  catch (runtime_error& x) {
    cout << x.what() << endl;
    return EXIT_FAILURE;
  }
} ///:~
```

В этой программе все содержимое файла читается в строку, для чего результат вызова rdbuf() файлового потока направляется в ostringstream. Далее остается лишь искать пары тегов HTML и удалять их, не беспокоясь о пересечении границ строк, как в предыдущем примере из главы 3.

Следующий пример показывает, как работать с двусторонним (то есть доступным для чтения и записи) строковым потоком:

```
//: C04:StringSeeking.cpp {-bor}{-dmc}
// Чтение и запись в строковый поток
#include <cassert>
#include <sstream>
#include <string>
using namespace std;

int main() {
  string text = "We will hook no fish";
  stringstream ss(text);
  ss.seekp(0, ios::end);
  ss << " before its time.";
  assert(ss.str() ==
    "We will hook no fish before its time.");
  // Замена "hook" на "ship"
  ss.seekg(8, ios::beg);
  string word;
  ss >> word;
  assert(word == "hook");
  ss.seekp(8, ios::beg);
  ss << "ship";
  // Замена "fish" на "code"
  ss.seekg(16, ios::beg);
  ss >> word;
  assert(word == "fish");
  ss.seekp(16, ios::beg);
  ss << "code";
  assert(ss.str() ==
    "We will ship no code before its time.");
  ss.str("A horse of a different color.");
  assert(ss.str() == "A horse of a different color.");
} ///:~
```

Как обычно, для перемещения позиции записи используется функция seekp(), а для перемещения позиции чтения — функция seekg(). Хотя из приведенного примера этого не видно, строковые потоки не так строги, как файловые потоки, и позволяют в любой момент переключиться с чтения на запись. Вам не приходится заново устанавливать текущие позиции записи или сбрасывать поток. В программе также продемонстрирована перегрузка функции str(), заменяющая базовый объект stringbuf потока новой строкой.

Форматирование в потоках вывода

Одной из главных целей при разработке потоков ввода-вывода была простота перемещения и/или форматирования символов. Несомненно, практическая ценность потоков ввода-вывода во многом определяется поддержкой большинства операций, реализуемых функциями printf() языка C. Как будет показано в этом разделе,

для потоков доступны все функции форматирования вывода, и это позволяет вам форматировать свой набор байтов так, как вы сочтете нужным.

При первом знакомстве с форматными функциями потоков ввода-вывода часто возникает путаница, так как во многих случаях форматированием можно управлять разными способами: при помощи функций класса и манипуляторов. Ситуация дополнительно осложняется тем, что обобщенные функции классов управляют форматированием при помощи флагов (например, флага выравнивания по левому или правому краю, флага использования прописных букв в шестнадцатеричной записи, флага обязательного включения десятичной точки в вещественные числа и т. д.). С другой стороны, чтение и запись символа-заполнителя, ширины поля и точности осуществляются отдельными функциями потоковых классов.

Чтобы прояснить эту запутанную ситуацию, мы сначала рассмотрим внутреннее форматирование данных в потоках и функции потоковых классов, изменяющие состояние этих данных. Манипуляторы будут рассматриваться отдельно.

Форматные флаги

Класс ios содержит переменные для хранения всех форматных данных, относящихся к потоку. Некоторые из этих данных принимают значения в допустимых интервалах (точность вещественных чисел, ширина поля и символ, используемый для заполнения пустых позиций при выводе; обычно это пробел). Остальные атрибуты форматирования определяются флагами, которые обычно объединяются для экономии места; в совокупности они называются *форматными флагами*. Значения форматных флагов можно получить функцией ios::flags() потокового класса. Функция вызывается без аргументов и возвращает объект типа fmtflags (обычно это синоним для типа long) с текущими форматными флагами. Другие функции изменяют состояние форматных флагов и возвращают их предыдущий набор:

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

Первая функция изменяет *все* флаги. Иногда именно это и нужно, но чаще требуется изменить состояние одного отдельного флага. Эта задача осуществляется тремя оставшимися функциями.

Функция setf() нередко вызывает недоразумения. Чтобы выбрать одну из ее перегруженных версий, нужно знать тип изменяемого флага. Существуют два типа флагов: одни устанавливаются или сбрасываются независимо, а вторые работают в сочетании с другими флагами. С независимыми флагами все просто: они устанавливаются функцией set(fmtflags) и сбрасываются функцией unset(fmtflags). Эти флаги перечислены ниже.

`ios::skipws`

Игнорирование пропусков (при вводе; используется по умолчанию).

`ios::showbase`

Вывод признака системы счисления (dec, oct или hex) при выводе целочисленных величин. Входные потоки также распознают префикс системы счисления при установленном флаге showbase.

`ios::showpoint`

Отображение десятичной точки и завершающих нулей в вещественных числах.

`ios::uppercase`

Вывод в верхнем регистре букв A–F в шестнадцатеричных числах и буквы E в экспоненциальной записи.

`ios::showpos`

Отображение знака плюс (+) для положительных величин.

`ios::unitbuf`

После каждой операции записи данные немедленно передаются в выходной поток.

Например, вывод знака плюс в положительных числах для cout включается вызовом функции cout.setf(ios::showpos). Чтобы отказаться от дальнейшего вывода знака плюс, вызовите функцию cout.unsetf(ios::showpos).

Флаг unitbuf устанавливает режим *немедленного вывода*, при котором каждая операция записи немедленно актуализируется в своем выходном потоке. Он особенно удобен при трассировке ошибок, чтобы при аварийном завершении программы данные все равно были записаны в файл журнала. Следующая программа демонстрирует немедленный вывод:

```
//: C04:Unitbuf.cpp
#include <cstdlib>  // Для abort()
#include <fstream>
using namespace std;

int main() {
  ofstream out("log.txt");
  out.setf(ios::unitbuf);
  out << "one\n";
  out << "two\n";
  abort();
} ///:~
```

Режим немедленного вывода должен быть включен до первой операции записи в поток. Когда мы закомментировали вызов setf(), один из компиляторов записал в файл log.txt только букву «o». Установка флага предотвратила потерю данных.

Немедленный вывод включен по умолчанию для стандартного потока вывода ошибок cerr. С другой стороны, отказ от промежуточной буферизации снижает эффективность вывода, поэтому при интенсивном использовании потока вывода не стоит включать режим немедленного вывода (если вас хоть сколько-нибудь беспокоят проблемы буферизации).

Форматные поля

Форматные флаги второго типа используются в группах. В любой момент времени может быть установлен только один флаг в группе, вроде кнопок в старых автомобильных радиоприемниках — если нажать одну кнопку, остальные кнопки автоматически сбрасываются. К сожалению, у форматных флагов автоматический сброс не работает, и вы должны следить за тем, какие флаги устанавливаются, чтобы по ошибке не вызвать функцию setf() с неверным набором флагов. Например, существуют флаги для трех систем счисления: шестнадцатеричной, десятичной

и восьмеричной. В совокупности эти флаги обозначаются ios::basefield. Если флаг ios::dec установлен и вы вызываете функцию setf(ios::hex), то флаг ios::hex устанавливается, однако флаг ios::dec при этом *не сбрасывается*, что приводит к непредсказуемым последствиям. Вместо этого следует вызвать вторую форму функции setf() вида setf(ios::hex,ios::basefield). Этот вызов сначала сбрасывает все флаги группы ios::basefield, *а затем* устанавливает ios::hex. Таким образом, данная форма setf() гарантирует, что при установке одного флага остальные флаги соответствующей группы будут сброшены. Манипулятор ios::hex делает это автоматически, поэтому вам не придется разбираться во внутренних подробностях реализации этого класса и вообще *знать*, что система счисления определяется набором двоичных флагов. Позднее вы убедитесь, что функциональность setf() всегда обеспечивается эквивалентными манипуляторами.

Ниже перечислены группы флагов и эффект от их установки.

Флаги группы ios::basefield:

`ios::dec`

Целочисленные значения форматируются в десятичной системе счисления (используется по умолчанию, префикс не нужен).

`ios::hex`

Целочисленные значения форматируются в шестнадцатеричной системе счисления.

`ios::oct`

Целочисленные значения форматируются в восьмеричной системе счисления.

Флаги группы ios::floatfield (если ни один флаг не установлен, ее количество значащих цифр определяется полем precision):

`ios::scientific`

Вещественные числа отображаются в научной (экспоненциальной) записи. Количество цифр в дробной части определяется полем precision.

`ios::fixed`

Вещественные числа отображаются в формате с фиксированной точностью. Количество цифр в дробной части определяется полем precision.

Флаги группы ios::adjustfield:

`ios::left`

Вещественные числа отображаются в научной (экспоненциальной) записи. Количество цифр в дробной части определяется полем precision.

`ios::right`

Вещественные числа отображаются в формате с фиксированной точностью. Количество цифр в дробной части определяется полем precision.

`ios::internal`

Общее количество значащих цифр определяется полем precision.

Переменные width, fill и precision

Значения внутренних переменных, управляющих шириной поля, символом-заполнителем и точностью вещественных чисел, читаются и задаются одноименными функциями потоковых классов.

`int ios::width()`

Возвращает текущую ширину поля (по умолчанию 0). Используется как при чтении, так и при записи в поток.

`int ios::width(int n)`

Задает ширину поля, возвращает предыдущее значение.

`int ios::fill()`

Возвращает текущий заполнитель (по умолчанию — пробел).

`int ios::fill(int n)`

Задает символ-заполнитель, возвращает предыдущее значение.

`int ios::precision()`

Возвращает текущую точность вещественных чисел (по умолчанию — 6).

`int ios::precision(int n)`

Задает точность вещественных чисел, возвращает предыдущее значение (см. ранее описание флагов группы ios::floatfield).

Атрибуты fill и precision вполне понятны, а width стоит пояснить особо. Если значение этого атрибута равно 0, то при записи значения в поток генерируется минимальное количество символов, необходимых для представления этого значения. Положительная ширина поля означает, что количество сгенерированных символов будет по крайней мере не меньше width; если оно содержит меньше символов, поле дополняется до необходимой ширины заполнителями fill. Тем не менее, выводимые значения никогда не усекаются, поэтому при попытке вывести 123 с шириной поля 2 все равно будет выведено число 123. Ширина поля определяет *минимальное* количество символов; задать максимальное количество символов невозможно.

Ширина также отличается от других атрибутов тем, что она обнуляется при каждой операции чтения или записи, на которую она может повлиять. В действительности она является не переменной состояния, а скорее неявным аргументом операторов << и >>. Если вы хотите, чтобы при работе с потоком использовалось постоянное значение ширины, вызывайте width() после каждой операции чтения или записи.

Пример форматирования в потоках вывода

Следующий пример, в котором задействованы все упоминавшиеся функции, поможет вам лучше понять, как они используются на практике:

```
//: C04:Format.cpp
// Функции форматирования
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;
#define D(A) T << #A << endl; A

int main() {
  ofstream T("format.out");
  assure(T);
  D(int i = 47;)
  D(float f = 2300114.414159;)
```

```
  const char* s = "Is there any more?";

  D(T.setf(ios::unitbuf);)
  D(T.setf(ios::showbase);)
  D(T.setf(ios::uppercase | ios::showpos);)
  D(T << i << endl;) // По умолчанию используется десятичная система (dec)
  D(T.setf(ios::hex, ios::basefield);)
  D(T << i << endl;)
  D(T.setf(ios::oct, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::showbase);)
  D(T.setf(ios::dec, ios::basefield);)
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.fill('0');)
  D(T << "fill char: " << T.fill() << endl;)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::right, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::internal, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T << i << endl;) // Без width(10)

  D(T.unsetf(ios::showpos);)
  D(T.setf(ios::showpoint);)
  D(T << "prec = " << T.precision() << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.unsetf(ios::uppercase);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.precision(20);)
  D(T << "prec = " << T.precision() << endl;)
  D(T << endl << f << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)

  D(T.width(10);)
  T << s << endl;
  D(T.width(40);)
  T << s << endl;
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.width(40);)
  T << s << endl;
} ///:~
```

В данном примере создается трассировочный файл для анализа работы программы. Макрос `D(a)` средствами препроцессора преобразует `a` в отображаемую строку, после чего `a` выполняется как команда. Вся информация передается в трассировочный файл `T`. Результат выглядит так:

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
```

```
T.setf(ios::showbase);
T.setf(ios::uppercase | ios::showpos);
T << i << endl;
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
0X2F
T.setf(ios::oct, ios::basefield);
T << i << endl;
057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
fill char: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "prec = " << T.precision() << endl;
prec = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300114E+06
T.unsetf(ios::uppercase);
T << endl << f << endl;

2.300114e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;

2300114.50000000000000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.30011450000000000000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000000000000000
T.width(10);
Is there any more?
T.width(40);
0000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
```

```
T.width(40):
Is there any more?0000000000000000000000
```

Анализ этих данных поможет разобраться в том, как работают функции форматирования потоковых классов.

Манипуляторы

Как видно из предыдущего примера, частые вызовы функций потоковых классов при форматировании несколько утомительны. Чтобы программы было проще читать и писать, были определены специальные *манипуляторы*, которые дублируют некоторые функции потоковых классов. Манипулятор — не более чем удобство, избавляющее вас от необходимости специально вызывать функцию.

Манипуляторы изменяют состояние потока вместо обработки данных (или наряду с ней). Например, при вставке манипулятора endl в команду вывода он не только выводит символ новой строки, но и *сбрасывает* поток (то есть выводит все символы, которые хранились во внутреннем буфере потока, но еще не были непосредственно выведены). Поток можно сбросить и другим способом:

```
cout << flush;
```

Это приводит к вызову функции flush() без вывода новых данных в поток:

```
cout.flush();
```

Другие манипуляторы позволяют быстро переключиться на восьмеричную (oct), десятичную (dec) или шестнадцатеричную (hex) систему счисления:

```
cout << hex << "0x" << i << endl;
```

В данном случае весь вывод будет осуществляться в шестнадцатеричной системе до тех пор, пока в выходной поток не будет вставлен манипулятор dec или oct.

Существует и манипулятор для чтения, «поглощающий» пропуски во входном потоке:

```
cin >> ws;
```

Манипуляторы без аргументов предоставляются заголовочным файлом <iostream>. К их числу относятся dec, oct и hex, делающие то же самое, что и, соответственно, setf(ios::dec,ios::basefield), setf(ios::oct,ios::basefield) и setf(ios::hex,ios::basefield), хотя и в более компактной записи. Кроме того, заголовок <iostream> содержит манипуляторы ws, endl и flush, а также дополнительные манипуляторы, перечисленные далее.

showbase/noshowbase

Вывод признака системы счисления (dec, oct или hex) при выводе целочисленных величин.

showpos/noshowpos

Отображение знака плюс (+) для положительных величин.

uppercase/nouppercase

Вывод в верхнем регистре букв A–F в шестнадцатеричных числах и буквы E в экспоненциальной записи.

showpoint/noshowpoint

Отображение десятичной точки и завершающих нулей в вещественных числах.

`skipws/noskipws`

Игнорирование пропусков (при вводе; используется по умолчанию).

`left`

Выравнивание по левому краю (дополнение справа).

`right`

Выравнивание по правому краю (дополнение слева).

`internal`

Заполнение между знаком (или признаком системы счисления) и значением.

`scientific/fixed`

Тип записи вещественных чисел (научная или с фиксированной точностью).

Манипуляторы с аргументами

Шесть стандартных манипуляторов, определяемых в заголовочном файле <iomanip>, получают аргументы. Краткое описание этих манипуляторов приводится ниже.

`setiosflags(fmtflags n)`

Эквивалент вызова setf(n). Продолжает действовать до следующего изменения флагов (например, вызовом ios::setf()).

`resetiosflags(fmtflags n)`

Сброс флагов, включенных в n. Продолжает действовать до следующего изменения флагов (например, вызовом ios::unsetf()).

`setbase(base n)`

Переход на систему счисления с основанием n, равным 8, 10 или 16 (при любом другом значении используется 0). Если n=0, вывод осуществляется в десятичной системе, но при вводе используются правила C: 10 интерпретируется как 10, 010 — как 8, а 0xf — как 16. С таким же успехом для вывода можно применять манипулятор dec, oct или hex.

`setfill(char n)`

Выбор заполнителя n, как при вызове ios::fill().

`setprecision(int n)`

Выбор точности n, как при вызове ios::precision().

`setw(int n)`

Выбор ширины поля n, как при вызове ios::width().

Если вам приходится выполнять многочисленные операции форматирования, переход от функций потоковых классов к манипуляторам сделает программу более четкой и понятной. Для примера рассмотрим программу из предыдущего раздела, переписанную с применением манипуляторов (макрос D() удален, чтобы программа проще читалась).

```
//: C04:Manips.cpp
// Программа format.cpp с манипуляторами
#include <fstream>
#include <iomanip>
#include <iostream>
```

```
using namespace std;

int main() {
  ofstream trc("trace.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = "Is there any more?";

  trc << setiosflags(ios::unitbuf
          | ios::showbase | ios::uppercase
          | ios::showpos);
  trc << i << endl;
  trc << hex << i << endl
      << oct << i << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << resetiosflags(ios::showbase)
      << dec << setfill('0');
  trc << "fill char: " << trc.fill() << endl;
  trc << setw(10) << i << endl;
  trc.setf(ios::right, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc.setf(ios::internal, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc << i << endl; // Без setw(10)

  trc << resetiosflags(ios::showpos)
      << setiosflags(ios::showpoint)
      << "prec = " << trc.precision() << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << resetiosflags(ios::uppercase) << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc << f << endl;
  trc << setprecision(20);
  trc << "prec = " << trc.precision() << endl;
  trc << f << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc << f << endl;

  trc << setw(10) << s << endl;
  trc << setw(40) << s << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << setw(40) << s << endl;
} ///:~
```

Группы отдельных команд были объединены в одну цепочечную операцию вывода. Обратите внимание на вызов setiosflags() с передачей поразрядной дизъюнкции флагов. То же самое можно было сделать функциями setf() и unsetf(), как в предыдущем примере.

При вызове setw() с потоком вывода выводимое выражение форматируется во временную строку и при необходимости дополняется текущим заполнителем до нужной ширины, для чего длина отформатированного результата сравнивается с аргументом setw(). Другими словами, setw() влияет на *итоговую строку* операции форматированного вывода. С другой стороны, применение setw() к потокам ввода имеет смысл лишь при чтении *строк*. В этом нетрудно убедиться на следующем примере:

```
//: C04:InputWidth.cpp
// Ограничения при использовании setw с потоками ввода
#include <cassert>
#include <cmath>
#include <iomanip>
#include <limits>
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream is("one 2.34 five");
  string temp;
  is >> setw(2) >> temp;
  assert(temp == "on");
  is >> setw(2) >> temp;
  assert(temp == "e");
  double x;
  is >> setw(2) >> x;
  double relerr = fabs(x - 2.34) / x;
  assert(relerr <= numeric_limits<double>::epsilon());
} ///:~
```

При чтении строк функция setw() будет успешно управлять количеством прочитанных символов... до определенного момента. При первом чтении будут получены два символа, а при втором — только один, хотя мы запрашивали два. Дело в том, что операторная функция operator>>() использует пропуски как ограничители ввода (хотя с этим можно бороться, сбросив флаг skipws). Но при чтении чисел невозможно ограничить количество читаемых символов при помощи setw(). В случае потоков ввода функция setw() работает только при чтении строк.

Создание манипуляторов

В некоторых ситуациях бывает удобно создать собственный манипулятор. Оказывается, это делается на удивление просто. Манипулятор без аргументов (аналог endl) представляет собой простую функцию, которая получает и возвращает ссылку на ostream. Объявление endl выглядит так:

```
ostream& endl(ostream&);
```

После этого в выражениях наподобие следующего endl интерпретируется как *адрес* этой функции:

```
cout << "howdy" << endl;
```

Компилятор спрашивает: «Существует ли подходящая функция, в аргументе которой передается адрес функции?» Для этого в <iostream> определены специальные функции, называемые *аппликаторами*. Аппликатор вызывает функцию, полученную в аргументе, и передает ей объект ostream в качестве аргумента. Чтобы создать собственный манипулятор, не нужно разбираться в принципах работы аппликаторов; достаточно знать, что они существуют. Упрощенный код аппликатора ostream выглядит так:

```
ostream& ostream::operator<<(ostream& (*pf)(ostream&)) {
  return pf(*this);
}
```

Реальное определение выглядит несколько сложнее, потому что в нем используются шаблоны, но для пояснения достаточно и этого. Когда в поток «выводится» функция, похожая на *pf (которая получает и возвращает ссылку на поток), вызывается функция-аппликатор. В свою очередь, она выполняет функцию, на которую ссылается pf. Аппликаторы для ios_base, basic_ios, basic_ostream и basic_istream определены в стандартной библиотеке C++.

Для пояснения сказанного рассмотрим тривиальный пример — манипулятор nl, эквивалентный занесению в поток символа перевода строки (то есть не выполняющий сброс потока, в отличие от endl):

```
//: C04:nl.cpp
// Создание манипулятора
#include <iostream>
using namespace std;

ostream& nl(ostream& os) {
  return os << '\n';
}

int main() {
  cout << "newlines" << nl << "between" << nl
       << "each" << nl << "word" << nl;
} ///:~
```

При направлении nl в поток вывода (например, в cout) происходит следующая последовательность вызовов:

```
cout.operator<<(nl) ® nl(cout)
```

Показанное ниже выражение внутри nl() вызывает операторную функцию ostream::operator(char):

```
os << '\n';
```

Поток, возвращаемый этой функций, в конечном счете возвращается манипулятором nl[1].

Эффекторы

Итак, манипуляторы без аргументов создаются легко. А если вам нужен манипулятор с аргументами? Просматривая заголовок <iomanip>, вы найдете тип smanip, возвращаемый манипуляторами с аргументами. Возникает предположение, что этот тип нужно использовать при определении собственных манипуляторов, но делать этого не стоит. Тип smanip зависит от реализации, а его использование нарушает переносимость программы. К счастью, манипуляторы с аргументами можно определять без применения специальных трюков. Для этой цели Джерри Шварц (Jerry Schwarz)[2] предложил методику, основанную на применении *эффекторов*. Эффектор представляет собой класс с конструктором и перегруженным оператором <<. Конструктор форматирует строку, представляющую нужную операцию, а оператор << направляет эту строку в поток. Здесь приводятся два эффектора: первый выводит усеченную символьную строку, а второй — число в двоичном виде.

---

[1] Перед включением nl в заголовочный файл объявите его подставляемой (inline) функцией.

[2] Разработчик библиотеки потоков ввода-вывода.

```
//: C04:Effector.cpp
// "Эффекторы" Джерри Шварца
#include <cassert>
#include <limits>  // Для max()
#include <sstream>
#include <string>
using namespace std;

// Вывод префикса:
class Fixw {
  string str;
public:
  Fixw(const string& s, int width) : str(s, 0, width) {}
  friend ostream& operator<<(ostream& os, const Fixw& fw) {
    return os << fw.str;
  }
};

// Вывод числа в двоичном виде:
typedef unsigned long ulong;

class Bin {
  ulong n;
public:
  Bin(ulong nn) { n = nn; }
  friend ostream& operator<<(ostream& os, const Bin& b) {
    const ulong ULMAX = numeric_limits<ulong>::max();
    ulong bit = ~(ULMAX >> 1); // Установлен старший бит
    while(bit) {
      os << (b.n & bit ? '1' : '0');
      bit >>= 1;
    }
    return os;
  }
};

int main() {
  string words =
    "Things that make us happy, make us wise";
  for(int i = words.size(); --i >= 0;) {
    ostringstream s;
    s << Fixw(words, i);
    assert(s.str() == words.substr(0, i));
  }
  ostringstream xs, ys;
  xs << Bin(0xCAFEBABEUL);
  assert(xs.str() ==
    "1100""1010""1111""1110""1011""1010""1011""1110");
  ys << Bin(0x76543210UL);
  assert(ys.str() ==
    "0111""0110""0101""0100""0011""0010""0001""0000");
} ///:~
```

Конструктор Fixw создает укороченную копию своего аргумента char*, а деструктор освобождает память, выделенную для этой копии. Перегруженный оператор << берет содержимое своего второго аргумента (объект Fixw), записывает его в свой первый аргумент (ostream) и возвращает ostream для последующего использования в цепочечном выражении. При использовании Fixw в выражениях следу-

ющего вида вызов конструктора Fixw создает *временный объект*, который затем передается оператору <<:

```
cout << Fixw(string, 1) << endl;
```

Результат получается тем же, что и при использовании манипулятора с аргументами. Временный объект Fixw существует до конца выполнения команды.

Работа эффектора Bin основана на том факте, что при сдвиге беззнакового числа вправо старшие биты заполняются нулями. Сначала мы с помощью конструкции numeric_limits<unsigned long>::max() (наибольшее значение типа unsigned long из стандартного заголовка <limits>) создаем маску со старшим установленным битом, а затем последовательно маскируем каждый бит выводимого числа (слева направо). Строковые литералы разделены в программе для удобства чтения; компилятор объединяет строковые компоненты в одну строку.

Традиционно считалось, что у этого решения есть недостаток: после создания класса Fixw для char* или Bin для unsigned long никто больше не сможет создать другой класс Fixw или Bin для своего типа. Однако с введением пространств имен эта проблема исчезла. Эффекторы и манипуляторы не эквивалентны, хотя они достаточно часто используются для решения одних и тех же задач. Если эффектор вас почему-либо не устраивает, вам придется разбираться во всех сложностях самостоятельного программирования манипуляторов.

Примеры использования потоков ввода-вывода

В этом разделе приводятся примеры, иллюстрирующие все то, о чем рассказывалось в данной главе. Существует немало программ для работы с байтовыми потоками; вероятно, наибольшей известностью пользуются потоковые редакторы, такие, как sed и awk в Unix, но текстовые редакторы тоже входят в эту категорию. И все же подобные программы обычно обладают определенными недостатками. Редакторы sed и awk относительно медлительны и способны обрабатывать строки лишь в одном направлении, а текстовые редакторы обычно требуют участия пользователя (или по крайней мере изучения специализированного макроязыка). Программы, написанные с использованием потоков ввода-вывода, лишены этих недостатков: они быстры, переносимы и гибки.

Сопровождение исходного кода библиотеки классов

Обычно создание класса рассматривается с позиций формирования библиотеки: вы создаете заголовочный файл Name.h с объявлением класса и файл Name.cpp, содержащий реализации функций этого класса. Эти файлы должны подчиняться некоторым требованиям: они кодируются в определенном стиле (так, приведенная далее программа соответствует общему формату программ в этой книге), а код заголовочного файла заключается в препроцессорную конструкцию, предотвращающую повторное включение классов (повторные включения сбивают с толку компилятор, который не знает, какую из версий выбрать; при наличии расхождений компилятор сдается и выдает сообщение об ошибке).

Следующий пример создает новую пару файлов (заголовочный файл и файл реализации) или изменяет существующую пару. Если файлы уже существуют,

программа проверяет их и, возможно, вносит изменения. Но если файлы не существуют, они создаются в заранее определенном формате.

```
//: C04:Cppcheck.cpp
// Создание заготовок файлов .h и .cpp с учетом требований стиля.
// Существующие файлы проверяются на соответствие.
#include <fstream>
#include <sstream>
#include <string>
#include <cstddef>
#include "../require.h"
using namespace std;

bool startsWith(const string& base, const string& key) {
  return base.compare(0, key.size(), key) == 0;
}
void cppCheck(string fileName) {
  enum bufs { BASE, HEADER, IMPLEMENT, HLINE1, GUARD1,
    GUARD2, GUARD3, CPPLINE1, INCLUDE, BUFNUM };
  string part[BUFNUM];
  part[BASE] = fileName;
  // Поиск вхождений '.' в строке:
  size_t loc = part[BASE].find('.');
  if(loc != string::npos)
    part[BASE].erase(loc); // Удаление расширения
  // Преобразование к верхнему регистру:
  for(size_t i = 0; i < part[BASE].size(); i++)
    part[BASE][i] = toupper(part[BASE][i]);
  // Создание имен файлов и обязательных строк:
  part[HEADER] = part[BASE] + ".h";
  part[IMPLEMENT] = part[BASE] + ".cpp";
  part[HLINE1] = "//" ": " + part[HEADER];
  part[GUARD1] = "#ifndef " + part[BASE] + "_H";
  part[GUARD2] = "#define " + part[BASE] + "_H";
  part[GUARD3] = "#endif // " + part[BASE] +"_H";
  part[CPPLINE1] = string("//") + ": " + part[IMPLEMENT];
  part[INCLUDE] = "#include \"" + part[HEADER] + "\"";
  // Сначала пытаемся открыть существующие файлы:
  ifstream existh(part[HEADER].c_str()),
           existcpp(part[IMPLEMENT].c_str());
  if(!existh) { // Файл не существует; создать его
    ofstream newheader(part[HEADER].c_str());
    assure(newheader, part[HEADER].c_str());
    newheader << part[HLINE1] << endl
      << part[GUARD1] << endl
      << part[GUARD2] << endl << endl
      << part[GUARD3] << endl;
  } else { // Файл существует; проверить его содержимое
    stringstream hfile; // Запись и чтение
    ostringstream newheader; // Запись
    hfile << existh.rdbuf();
    // Убедиться в том, что первые три строки
    // соответствуют стилевому стандарту:
    bool changed = false;
    string s;
    hfile.seekg(0);
    getline(hfile, s);
    bool lineUsed = false;
    // Вызов good() необходим для компилятора Microsoft (также см. далее)
```

```
  for (int line = HLINE1; hfile.good() && line <= GUARD2;
       ++line) {
    if(startsWith(s, part[line])) {
      newheader << s << endl;
      lineUsed = true;
      if (getline(hfile, s))
        lineUsed = false;
    } else {
      newheader << part[line] << endl;
      changed = true;
      lineUsed = false;
    }
  }
  // Копирование остатка файла
  if (!lineUsed)
    newheader << s << endl;
  newheader << hfile.rdbuf();
  // Проверка GUARD3
  string head = hfile.str();
  if(head.find(part[GUARD3]) == string::npos) {
    newheader << part[GUARD3] << endl;
    changed = true;
  }
  // При наличии изменений файл перезаписывается:
  if(changed) {
    existh.close();
    ofstream newH(part[HEADER].c_str());
    assure(newH, part[HEADER].c_str());
    newH << "//@//\n"  // Признак модификации
      << newheader.str();
  }
}
if(!existcpp) { // Создание файла реализации
  ofstream newcpp(part[IMPLEMENT].c_str());
  assure(newcpp, part[IMPLEMENT].c_str());
  newcpp << part[CPPLINE1] << endl
    << part[INCLUDE] << endl;
} else { // Файл существует; проверить его содержимое
  stringstream cppfile;
  ostringstream newcpp;
  cppfile << existcpp.rdbuf();
  // Убедиться в том, что первые две строки
  // соответствуют стилевому стандарту:
  bool changed = false;
  string s;
  cppfile.seekg(0);
  getline(cppfile, s);
  bool lineUsed = false;
  for (int line = CPPLINE1;
       cppfile.good() && line <= INCLUDE;
       ++line) {
    if(startsWith(s, part[line])) {
      newcpp << s << endl;
      lineUsed = true;
      if (getline(cppfile, s))
        lineUsed = false;
    } else {
      newcpp << part[line] << endl;
      changed = true;
```

```
          lineUsed = false;
        }
      }
      // Копирование остатка файла
      if (!lineUsed)
        newcpp << s << endl;
      newcpp << cppfile.rdbuf();
      // При наличии изменений файл перезаписывается:
      if(changed){
        existcpp.close();
        ofstream newCPP(part[IMPLEMENT].c_str());
        assure(newCPP, part[IMPLEMENT].c_str());
        newCPP << "//@//\n"  // Признак модификации
          << newcpp.str();
      }
    }
  }

  int main(int argc, char* argv[]) {
    if(argc > 1)
      cppCheck(argv[1]);
    else
      cppCheck("cppCheckTest.h");
  } ///:~
```

Для начала обратите внимание на вспомогательную функцию startsWith(), которая возвращает true, если первый строковый аргумент начинается со второго аргумента. Она используется при проверке стандартных комментариев и директив условного включения. Наличие массива строк part позволяет легко организовать перебор последовательности ожидаемых строк в исходном коде. Если исходный файл не существует, команды просто записываются в новый файл с заданным именем. Если файл существует, программа последовательно проверяет начальные строки и убеждается в присутствии обязательных строк. Если эти строки отсутствуют, они вставляются в файл. При этом необходимо особо позаботиться о том, чтобы не удалить существующие строки (см. использование логической переменной lineUsed). Обратите внимание: для существующего файла задействуется объект stringstream, и это позволяет нам сначала записать в него содержимое файла, а затем прочитать его и провести поиск.

В перечисляемый тип включены индексы компонентов, представленные следующими именами:

- BASE — имя базового файла (без расширения), записанное в верхнем регистре;
- HEADER — имя заголовочного файла;
- IMPLEMENT — имя файла реализации (cpp);
- HLINE1 — заготовка первой строки заголовочного файла;
- GUARD1, GUARD2 и GUARD3 — «сторожевые» строки заголовочного файла, предотвращающие повторное включение;
- CPPLINE1 — заготовка первой строки файла реализации (cpp);
- INCLUDE — строка файла реализации, в которой включается заголовочный файл.

При запуске без аргументов программа создает следующие два файла:

```
// CPPCHECKTEST.h
#ifndef CPPCHECKTEST_H
#define CPPCHECKTEST_H

#endif // CPPCHECKTEST_H

// CPPCHECKTEST.cpp
#include "CPPCHECKTEST.h"
```

Мы удалили двоеточие после // в первом комментарии, чтобы не «смущать» программу извлечения кода. Двоеточие будет присутствовать в коде, фактически сгенерированном программой cppCheck.

Попробуйте поэкспериментировать с этими файлами, удаляя из них отдельные строки и запуская программу заново. При каждом запуске обязательные строки будут снова появляться в файле. При модификации в файл вставляется первая строка //@//, единственное назначение которой — привлечь внимание читателя. Перед повторной обработкой файла эту строку необходимо удалить (в противном случае cppcheck сочтет, что в файле отсутствует первая строка с комментарием).

Обнаружение ошибок компиляции

Весь программный код, приводимый в книге, должен компилироваться без ошибок. Строки, намеренно порождающие ошибки компиляции, помечаются специальным комментарием //!. Следующая программа удаляет эти специальные комментарии, а в конец строки добавляется нумерованный комментарий. При запуске компилятора генерируются сообщения об ошибках, в которых будут присутствовать эти номера. Программа также включает модифицированную строку в специальный файл, что упрощает поиск строк, которые были откомпилированы без ошибок.

```
//: C04:Showerr.cpp
// Удаление комментариев из строк с ошибками компиляции
#include <cstddef>
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;

const string usage =
  "usage: showerr filename chapnum\n"
  "where filename is a C++ source file\n"
  "and chapnum is the chapter name it's in.\n"
  "Finds lines commented with //! and removes\n"
  "comment, appending //(#) where # is unique\n"
  "across all files, so you can determine\n"
  "if your compiler finds the error.\n"
  "showerr /r\n"
  "resets the unique counter.";

class Showerr {
  const int CHAP;
```

```
  const string MARKER, FNAME;
  // Файл со счетчиком номеров ошибок:
  const string ERRNUM;
  // Файл с ошибочными строками:
  const string ERRFILE;
  stringstream edited; // Измененный файл
  int counter;
public:
  Showerr(const string& f, const string& en,
    const string& ef, int c) : CHAP(c), MARKER("//!"),
    FNAME(f), ERRNUM(en), ERRFILE(ef), counter(0) {}
  void replaceErrors() {
    ifstream infile(FNAME.c_str());
    assure(infile, FNAME.c_str());
    ifstream count(ERRNUM.c_str());
    if(count) count >> counter;
    int linecount = 1;
    string buf;
    ofstream errlines(ERRFILE.c_str(), ios::app);
    assure(errlines, ERRFILE.c_str());
    while(getline(infile, buf)) {
      // Поиск маркера в начале строки:
      size_t pos = buf.find(MARKER);
      if(pos != string::npos) {
        // Удаление маркера:
        buf.erase(pos, MARKER.size() + 1);
        // Присоединение счетчика и информации об ошибке:
        ostringstream out;
        out << buf << " // (" << ++counter << ") "
            << "Chapter " << CHAP
            << " File: " << FNAME
            << " Line " << linecount << endl;
        edited << out.str();
        errlines << out.str(); // Запись в файл ошибок
      }
      else
        edited << buf << "\n"; // Простое копирование
      linecount++;
    }
  }
  void saveFiles() {
    ofstream outfile(FNAME.c_str()); // Перезапись
    assure(outfile, FNAME.c_str());
    outfile << edited.rdbuf();
    ofstream count(ERRNUM.c_str()); // Перезапись
    assure(count, ERRNUM.c_str());
    count << counter; // Сохранение нового счетчика
  }
};

int main(int argc, char* argv[]) {
  const string ERRCOUNT("../errnum.txt"),
    ERRFILE("../errlines.txt");
  requireMinArgs(argc, 1, usage.c_str());
  if(argv[1][0] == '/' || argv[1][0] == '-') {
    // Другие ключи:
    switch(argv[1][1]) {
      case 'r': case 'R':
```

```
        cout << "reset counter" << endl;
        remove(ERRCOUNT.c_str()); // Удаление файлов
        remove(ERRFILE.c_str());
        return 0;
      default:
        cerr << usage << endl;
        return 1;
    }
  }
  if (argc == 3) {
    Showerr s(argv[1], ERRCOUNT, ERRFILE, atoi(argv[2]));
    s.replaceErrors();
    s.saveFiles();
  }
} ///:~
```

Вы можете заменить маркер любым другим по своему усмотрению.

Программа построчно читает файлы и ищет в начале каждой строки маркер. При обнаружении маркера строка модифицируется и сохраняется в списке ошибок и в строковом потоке **edited**. После завершения обработки всего файла он закрывается (с выходом из области видимости), открывается заново для вывода, и в него направляется содержимое **edited**. Также обратите внимание на сохранение счетчика во внешнем файле. При следующем запуске программы нумерация будет продолжена.

Простая программа ведения журнала

В приведенном далее примере демонстрируется подход, который может пригодиться для сохранения наборов данных на диске для последующей обработки (например, температурного профиля океанской воды в разных точках). Данные хранятся в классе **DataPoint**:

```
//: C04:DataLogger.h
// Структура записи в журнал
#ifndef DATALOG_H
#define DATALOG_H
#include <ctime>
#include <iosfwd>
#include <string>
using std::ostream;

struct Coord {
  int deg, min, sec;
  Coord(int d = 0, int m = 0, int s = 0)
    : deg(d), min(m), sec(s) {}
  std::string toString() const;
};
ostream& operator<<(ostream&, const Coord&);

class DataPoint {
  std::time_t timestamp; // Время и дата
  Coord latitude, longitude;
  double depth, temperature;
public:
  DataPoint(std::time_t ts, const Coord& lat,
            const Coord& lon, double dep, double temp)
    : timestamp(ts), latitude(lat), longitude(lon),
```

```
      depth(dep), temperature(temp) {}
  DataPoint() : timestamp(0), depth(0), temperature(0) {}
  friend ostream& operator<<(ostream&, const DataPoint&);
};
#endif // DATALOG_H ///:~
```

Структура DataPoint состоит из метки времени, хранящейся в переменной типа time_t (определяется в <ctime>), двух координат (широты и долготы), а также значений глубины и температуры. Оператор << упрощает форматирование данных. Ниже приводится файл реализации:

```
//: C04:DataLogger.cpp {O}
// Реализация Datapoint
#include "DataLogger.h"
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

ostream& operator<<(ostream& os, const Coord& c) {
  return os << c.deg << '*' << c.min << '\''
            << c.sec << '"';
}

string Coord::toString() const {
  ostringstream os;
  os << *this;
  return os.str();
}

ostream& operator<<(ostream& os, const DataPoint& d) {
  os.setf(ios::fixed, ios::floatfield);
  char fillc = os.fill('0'); // Дополнение слева символами '0'
  tm* tdata = localtime(&d.timestamp);
  os << setw(2) << tdata->tm_mon + 1 << '\\'
     << setw(2) << tdata->tm_mday << '\\'
     << setw(2) << tdata->tm_year+1900 << ' '
     << setw(2) << tdata->tm_hour << ':'
     << setw(2) << tdata->tm_min << ':'
     << setw(2) << tdata->tm_sec;
  os.fill(' '); // Дополнение слева символами ' '
  streamsize prec = os.precision(4);
  os << " Lat:" << setw(9) << d.latitude.toString()
     << ", Long:" << setw(9) << d.longitude.toString()
     << ", depth:" << setw(9) << d.depth
     << ", temp:" << setw(9) << d.temperature;
  os.fill(fillc);
  os.precision(prec);
  return os;
} ///:~
```

Функция Coord::toString() необходима, поскольку оператор << типа DataPoint вызывает setw() перед выводом широты и долготы. Если бы вместо этого для Coord использовался оператор <<, то ширина поля относилась бы только к первой операции вывода (то есть Coord::deg), поскольку все изменения ширины немедленно отменяются после первого вывода. Вызов setf() обеспечивает вывод вещественных

чисел с фиксированной точностью, а функция precision() задает точность в четыре цифры в дробной части. Также обратите внимание на восстановление заполнителя и точности, действовавших перед вызовом оператора <<.

Чтобы получить данные метки времени, хранящиеся в DatePoint::timestamp, мы вызываем функцию std::localtime(), которая возвращает статический указатель на объект tm. Структура tm определяется следующим образом:

```
struct tm {
  int tm_sec;   // 0-59 секунд
  int tm_min;   // 0-59 минут
  int tm_hour;  // 0-23 часов
  int tm_mday;  // День месяца
  int tm_mon;   // 0-11 месяцев
  int tm_year;  // Прошло лет с 1900
  int tm_wday;  // Воскресенье == 0 и т. д.
  int tm_yday;  // 0-365, день года
  int tm_isdst; // Летнее время?
};
```

Генератор тестовых данных

Следующая программа создает два файла: первый файл содержит двоичные данные и создается функцией write(), а второй содержит данные ASCII и создается с использованием оператора << для класса DataPoint. Данные также можно вывести на экран, но удобнее просматривать их в виде файла.

```
//: C04:Datagen.cpp
// Генератор тестовых данных
//{L} DataLogger
#include <cstdlib>
#include <ctime>
#include <cstring>
#include <fstream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
  time_t timer;
  srand(time(&timer)) // Раскрутка генератора случайных чисел
  ofstream data("data.txt");
  assure(data, "data.txt");
  ofstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  for(int i = 0; i < 100; i++, timer += 55) {
    // От 0 до 199 метров:
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += 1.0 / fraction;
    double newtemp = 150 + rand()%200; // По шкале Кельвина
    fraction = rand() % 100 + 1;
    newtemp += 1.0 / fraction;
    const DataPoint d(timer, Coord(45,20,31),
      Coord(22,34,18), newdepth, newtemp);
    data << d << endl;
    bindata.write(reinterpret_cast<const char*>(&d), sizeof(d));
  }
} ///:~
```

Файл data.txt создается обычным способом как ASCII-файл, но для файла data.bin устанавливается флаг ios::binary, который сообщает конструктору, что файл должен создаваться как двоичный. Чтобы продемонстрировать форматирование данных в текстовом файле, мы приводим первую строку файла data.txt (строка не помещается на странице, поэтому она перенесена):

```
07\28\2003 12:54:40 Lat:45*20'31", Long:22*34'18", depth:
  16.0164, temp: 242.0122
```

Функция time() стандартной библиотеки C обновляет переменную time_t, на которую ссылается ее аргумент, текущим временем. На большинстве платформ последнее кодируется как количество секунд, прошедшее с 00:00:00 по Гринвичу 1 января 1970 года. Текущее время также хорошо подходит для раскрутки генератора случайных чисел функцией srand() стандартной библиотеки C, как это сделано в нашей программе.

После этого таймер увеличивается с приращением в 55 секунд, чтобы интервалы между контрольными точками выглядели менее тривиально.

Фиксированные значения широты и долготы обозначают набор измерений в одной географической точке. Глубина и температура генерируются с использованием функции rand() стандартной библиотеки C, возвращающей псевдослучайное число от нуля до RAND_MAX — константы, зависящей от платформы и определяемой в заголовке <cstdlib> (обычно это наибольшее целое число, представимое на данной платформе). Для приведения числа к нужному интервалу используется оператор вычисления остатка % и верхняя граница интервала. Полученные числа являются целыми; чтобы добавить к ним дробную часть, мы снова вызываем rand() и делим 1 на полученное число (с прибавлением 1, чтобы предотвратить деление на 0).

Фактически файл data.bin задействуется как хранилище данных программы, хотя он находится на диске, а не в оперативной памяти. Функция write() записывает данные на диск в двоичной форме. Первый аргумент содержит начальный адрес исходного блока, преобразованный к типу char*, поскольку именно такой тип write() ожидает для потоков с обычной (не расширенной) кодировкой. Второй аргумент содержит количество записываемых символов, которое в данном случае совпадает с размером объекта DataPoint (тоже потому, что мы используем обычную кодировку). Поскольку объект DataPoint не содержит указателей, проблем с его сохранением на диске не возникает. Для более сложных объектов приходится разрабатывать схему *сериализации* с сохранением данных, на которую ссылаются указатели, и созданием новых указателей при последующей загрузке объекта. (Сериализация в книге не рассматривается — во многих библиотеках классов реализована та или иная ее разновидность.)

Проверка и просмотр данных

Чтобы проверить правильность сохранения данных в двоичном формате, можно прочитать их в память функцией read() класса потоков ввода и сравнить с текстовым файлом, созданным ранее программой Datagen.cpp. В следующем примере отформатированный результат просто записывается в cout, но вы можете перенаправить результат в файл, а затем при помощи утилиты сравнения файлов проверить его идентичность с оригиналом:

```
//: C04:Datascan.cpp
//{L} DataLogger
```

```
#include <fstream>
#include <iostream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
  ifstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  DataPoint d;
  while (bindata.read(reinterpret_cast<char*>(&d), sizeof d))
    cout << d << endl;
} ///:~
```

Интернационализация

В наше время индустрия разработки программного обеспечения стала процветающей отраслью мирового уровня, что повышает спрос на предложения с поддержкой различных языков и культурных стандартов. Еще в конце 1980-х годов Комитет по стандартизации C включил в стандартный язык поддержку нестандартных схем форматирования данных в виде *локальных контекстов*. Локальный контекст представляет собой набор параметров, определяющих внешний вид некоторых выводимых значений, например дат или денежных величин. В 1990 году Комитет по стандартизации C утвердил дополнение к стандарту C, в котором определялись функции для работы с *расширенными символами* (тип wchar_t). Тем самым обеспечивалась поддержка других кодировок, помимо ASCII и ее типичных западноевропейских расширений. Хотя в спецификации не указан точный размер расширенных символов, на некоторых платформах они реализуются в виде 32-разрядных величин и могут содержать данные в кодировке, определенной консорциумом Юникод, а также коды многобайтовых азиатских кодировок. Интегрированная поддержка расширенных символов и локальных контекстов включена в библиотеку потоков ввода-вывода.

Расширенные потоки

Расширенным потоком называется потоковый класс, поддерживающий операции с символами в расширенных кодировках. До настоящего момента во всех примерах использовались *узкие потоки*, содержащие экземпляры типа char (не считая последнего примера с классом характеристик из главы 3). Поскольку потоковые операции практически не зависят от базового типа символов, они обычно инкапсулируются в виде шаблонов. Так, базовое поведение потоков ввода определяется шаблоном basic_istream:

```
template<class charT, class traits = char_traits<charT> >
class basic_istream {…};
```

Фактически все типы потоков ввода представляют собой специализации этого шаблона в виде следующих определений типов:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
```

```
typedef basic_istringstream<char> istringstream;
typedef basic_istringstream<wchar_t> wistringstream;
```

Остальные типы потоков определяются аналогичным образом.

В идеале этого было бы достаточно для создания потоков с различными типами символов, но на практике все не так просто. Дело в том, что функции обработки символов для типов char и wchar_t имеют разные имена. Например, для сравнения двух строк с обычной кодировкой используется функция strcmp(), а для строк с расширенной кодировкой — функция wcscmp(). Не забывайте, что поддержка расширенных кодировок появилась в языке C, в котором перегрузка функций отсутствует, поэтому пришлось использовать уникальные имена. Из-за этого обобщенный поток не мог просто вызвать strcmp() при вызове оператора сравнения. Понадобился некий механизм автоматического выбора правильной низкоуровневой функции.

Данная проблема была решена выделением различий в новую абстракцию. Операции с символами были абстрагированы в шаблон char_traits, у которого имеются стандартные специализации для char и wchar_t (см. главу 3). Итак, чтобы сравнить две строки, basic_string просто вызывает шаблон traits::compare() (вспомните: traits является вторым параметром шаблона), который в свою очередь вызывает strcmp() или wcscmp() в зависимости от того, какая именно специализация используется (причем все это происходит незаметно для basic_string).

Вам придется думать о char_traits только при вызове низкоуровневых функций обработки символов; в большинстве случаев различия в именах функций для вас несущественны. Впрочем, подумайте, не стоит ли определить операторы >> и << в виде шаблонов на тот случай, если кто-нибудь захочет их использовать с расширенными потоками.

Чтобы было понятнее, вспомните оператор для вывода Date, приведенный в начале этой главы. В исходном варианте его объявление выглядело так:

```
ostream& operator<<(ostream&, const Date&);
```

Такое объявление годится только для узких потоков. Обобщение производится преобразованием в шаблон на базе basic_ostream:

```
template<class charT, class traits>
std::basic_ostream<charT, traits>&
operator<<(std::basic_ostream<charT, traits>& os, const Date& d) {
  charT fillc = os.fill(os.widen('0'));
  charT dash = os.widen('-');
  os << setw(2) << d.month << dash
     << setw(2) << d.day << dash
     << setw(4) << d.year;
  os.fill(fillc);
  return os;
}
```

Также придется заменить char параметром шаблона charT в объявлении fillc, поскольку это может быть как char, так и wchar_t в зависимости от специализации шаблона.

При написании шаблона тип потока неизвестен, поэтому необходим механизм автоматического преобразования символьных литералов к правильному размеру для данного потока. Задача решается благодаря функции widen() потокового класса. Например, выражение widen('-') преобразует свой аргумент в L'-' для расширенных потоков и оставляет его без изменений для узких потоков. Также су-

ществует функция narrow(), которая при необходимости преобразует символ в char.

Функция widen() позволяет написать обобщенную версию представленного ранее манипулятора nl:

```
template<class charT, class traits>
basic_ostream<charT,traits>&
nl(basic_ostream<charT,traits>& os) {
  return os << charT(os.widen('\n'));
}
```

Локальный контекст

Вероятно, самым заметным различием в стандартах вывода числовых данных, принятых в разных странах, является выбор разделителя между целой и дробной частями вещественных чисел. В США таким разделителем является точка, а во многих странах мира — запятая. Было бы крайне неудобно вручную форматировать данные для разных национальных стандартов. Как и прежде, проблема решается созданием абстракции, которая нивелирует эти различия с точки зрения пользователя.

Такая абстракция называется *локальным контекстом*. С каждым потоком связывается объект локального контекста, параметрами которого поток руководствуется для отображения данных в конкретной культурной среде. Локальный контекст управляет несколькими категориями данных, отображение которых определяется национальными стандартами.

`collate`

Поддержка сравнения строк по разным схемам сортировки.

`ctype`

Абстракция классификации символов и средств преобразования, определяемых в файле **<cctype>**.

`monetary`

Поддержка разных форматов вывода денежных величин.

`numeric`

Поддержка разных форматов вывода вещественных чисел, включая разделитель целой и дробной части и разделитель групп разрядов.

`time`

Поддержка разных международных форматов вывода даты и времени.

`messages`

Поддержка контекстно-зависимых каталогов сообщений (например, сообщений об ошибках на разных языках).

Следующая программа демонстрирует простейшие операции с локальным контекстом:

```
//: C04:Locale.cpp {-g++}{-bor}{-edg} {RunByHand}
// Эффект от использования локальных контекстов
#include <iostream>
#include <locale>
using namespace std;

int main() {
```

```
  locale def;
  cout << def.name() << endl;
  locale current = cout.getloc();
  cout << current.name() << endl;
  float val = 1234.56;
  cout << val << endl;
  // Переключение на французский локальный контекст
  cout.imbue(locale("french"));
  current = cout.getloc();
  cout << current.name() << endl;
  cout << val << endl;

  cout << "Enter the literal 7890,12: ";
  cin.imbue(cout.getloc());
  cin >> val;
  cout << val << endl;
  cout.imbue(def);
  cout << val << endl;
} ///:~
```

А вот как выглядит результат:

```
C
C
1234.56
French_France.1252
1234,56
Enter the literal 7890,12: 7890,12
7890,12
7890.12
```

По умолчанию используется локальный контекст "C", хорошо знакомый программистам C и C++ (английский язык, американские стандарты). Все потоки изначально настраиваются на локальный контекст "C". Функция imbue() потокового класса переключает локальный контекст потока. Обратите внимание: в результатах выводится полное название «французского» локального контекста по стандарту IOS (то есть французский язык, используемый во Франции, в отличие от французского, используемого в другой стране). Этот пример показывает, что во французском локальном контексте целая часть вещественного числа отделяется от дробной части запятой, а не точкой. Чтобы ввод данных также осуществлялся по правилам этого локального контекста, следует переключить на него cin.

Каждая категория локального контекста делится на *фацеты* — классы, в которых инкапсулируется функциональность данной категории. Например, в категорию time входят фацеты time_put и time_get, содержащие функции соответственно для ввода и вывода даты/времени. Категория monetary содержит фацеты money_get, money_put и moneypunct (последний определяет знак денежной единицы). Следующая программа демонстрирует использование фацета moneypunct (фацет time требует нетривиальных операций с итераторами, выходящих за рамки темы этой главы).

```
//: C04:Facets.cpp {-bor}{-g++}{-mwcc}{-edg}
#include <iostream>
#include <locale>
#include <string>
using namespace std;

int main() {
```

```
  // Переключение на французский локальный контекст
  locale loc("french");
  cout.imbue(loc);
  string currency =
    use_facet<moneypunct<char> >(loc).curr_symbol();
  char point =
    use_facet<moneypunct<char> >(loc).decimal_point();
  cout << "I made " << currency << 12.34 << " today!"
       << endl;
} ///:~
```

В выходных данных отображаются французский символ денежной единицы и разделитель:

```
I made €12,34 today!
```

Вы также можете определять собственные фацеты для конструирования нестандартных локальных контекстов. Учтите, что применение локальных контекстов связано с существенными затратами. Некоторые разработчики библиотек выпускают «диалекты» стандартной библиотеки C++ для сред с ограниченными ресурсами[1].

Итоги

Данная глава достаточно подробно знакомит читателя с библиотекой потоков ввода-вывода. Вполне вероятно, этого материала будет достаточно для создания собственных программ, использующих потоки. Некоторые дополнительные возможности потоков требуются нечасто, и поэтому здесь не рассматриваются. Вы можете изучить их самостоятельно. Для этого достаточно просмотреть заголовочные файлы библиотеки потоков ввода-вывода, прочитать документацию компилятора или специализированные книги по этой теме.

Упражнения

1. Откройте файл, создав объект ifstream. Создайте объект ostringstream, загрузите в него все содержимое файла функцией rdbuf(). Извлеките копию строки из буфера и преобразуйте все символы к верхнему регистру стандартным макросом toupper() языка C, определенным в файле <cctype>. Запишите результат в новый файл.
2. Создайте программу, которая открывает файл (первый аргумент в командной строке) и ищет в нем любое слово из заданного набора (остальные аргументы командной строки). Читайте входные данные по строкам и записывайте пронумерованные строки, в которых будут найдены совпадения, в новый файл.
3. Напишите программу, которая включает уведомление об авторском праве в начало каждого файла с исходным кодом программы, переданного в аргументах командной строки программы.

[1] Один из примеров — библиотека Abridged фирмы Dinkumware (http://www.dinkumware.com). Из библиотеки исключена поддержка локальных контекстов, а поддержка исключений является необязательной.

4. При помощи своей любимой утилиты поиска (например, grep) найдите имена всех файлов, содержащих определенную последовательность символов. Перенаправьте вывод в файл. Напишите программу, которая по содержимому этого файла генерирует пакетный файл, запускающий текстовый редактор для каждого найденного файла.
5. Мы знаем, что манипулятор setw() позволяет задать минимальное количество вводимых символов. А если размер вводимых данных потребуется ограничить сверху? Напишите эффектор, который бы позволял задать максимальное количество вводимых символов. Пусть ваш эффектор также работает для вывода (выводимые поля должны при необходимости усекаться по ширине).
6. Докажите, что если при установленном флаге badbit или failbit активизировать потоковые исключения, то поток немедленно запустит исключение.
7. Строковые потоки упрощают преобразования текстовых данных, но за это приходится платить определенную цену. Напишите программу для проведения сравнительного хронометража atoi() и системы преобразований stringstream. Посмотрите, к каким затратам приводит использование системы stringstream.
8. Создайте структуру Person с информацией о человеке: имя, адрес и т. д. Объявите строковые поля как массивы фиксированного размера. Ключом каждой записи должен быть номер социального страхования. Реализуйте следующий класс Database:

```
class DataBase {
public:
  // Поиск записи на диске
  size_t query(size_t ssn);
  // Выборка структуры Person по номеру записи
  Person retrieve(size_t rn);
  // Сохранение записи на диске
  void add(const Person& p);
};
```

   Запишите примеры данных Person на диск (не держите их в памяти). Когда пользователь запросит запись, прочитайте ее с диска и передайте пользователю. В операциях ввода-вывода класса DataBase при обработке записей Person должны применяться функции read() и write().
9. Напишите оператор << для структуры Person, который бы отображал записи в формате, удобном для чтения. Продемонстрируйте его работу, сохранив данные в файле.
10. Допустим, база данных со структурами Person была потеряна, но у вас остался файл, записанный в предыдущем примере. Воссоздайте базу данных по содержимому файла. Не забудьте организовать проверку ошибок!
11. Запишите size_t(-1) (наибольшее беззнаковое целое на вашей платформе) в текстовый файл 1 000 000 раз. Сделайте то же самое для двоичного файла. Сравните размеры двух файлов и посмотрите, сколько места экономит вывод в двоичном формате (попробуйте предсказать результат заранее).

12. Определите максимальную точность вывода в вашей реализации библиотеки потоков, последовательно увеличивая значение аргумента precision() при выводе трансцендентного числа (например, sqrt(2.0)).
13. Напишите программу, которая читает вещественные числа из файла и выводит их сумму, среднее арифметическое, минимальное и максимальное значения.
14. Определите, какие данные выведет следующая программа, до ее выполнения:

```
//: C04:Exercise14.cpp
#include <fstream>
#include <iostream>
#include <sstream>
#include "../require.h"
using namespace std;

#define d(a) cout << #a " ==\t" << a << endl;

void tellPointers(fstream& s) {
  d(s.tellp());
  d(s.tellg());
  cout << endl;
}

void tellPointers(stringstream& s) {
  d(s.tellp());
  d(s.tellg());
  cout << endl;
}

int main() {
  fstream in("Exercise14.cpp");
  assure(in, "Exercise14.cpp");
  in.seekg(10);
  tellPointers(in);
  in.seekp(20);
  tellPointers(in);
  stringstream memStream("Here is a sentence.");
  memStream.seekg(10);
  tellPointers(memStream);
  memStream.seekp(5);
  tellPointers(memStream);
} ///:~
```

15. Предположим, у вас имеются строковые данные в файле следующего формата:

```
//: C04:Exercise15.txt
Australia
5E56,7667230284,Langler,Tyson,31.2147,0.00042117361
2B97,7586701,Oneill,Zeke,553.429,0.0074673053156065
4D75,7907252710,Nickerson,Kelly,761.612,0.010276276
9F2,6882945012,Hartenbach,Neil,47.9637,0.0006471644
Austria
480F,7187262472,Oneill,Dee,264.012,0.00356226040013
1B65,4754732628,Haney,Kim,7.33843,0.000099015948475
DA1,1954960784,Pascente,Lester,56.5452,0.0007629529
3F18,1839715659,Elsea,Chelsy,801.901,0.010819887645
```

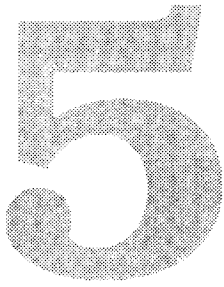
```
Belgium
BDF,5993489554,Oneill,Meredith,283.404,0.0038239127
5AC6,6612945602,Parisienne,Biff,557.74,0.0075254727
6AD,6477082,Pennington,Lizanne,31.0807,0.0004193544
4D0E,7861652688,Sisca,Francis,704.751,0.00950906238
Bahamas
37D8,6837424208,Parisienne,Samson,396.104,0.0053445
5E98,6384069,Willis,Pam,90.4257,0.00122009564059246
1462,1288616408,Stover,Hazal,583.939,0.007878970561
5FF3,8028775718,Stromstedt,Bunk,39.8712,0.000537974
1095,3737212,Stover,Denny,3.05387,0.000041205248883
7428,2019381883,Parisienne,Shane,363.272,0.00490155
```

В заголовках секций указаны географические регионы, а в строках каждой секции перечисляются торговые представители некой организации. Данные каждого представителя хранятся в виде полей, разделенных запятыми. Первое поле в строке содержит идентификатор представителя — к сожалению, записанный в шестнадцатеричном виде. Во втором поле хранится телефонный номер; обратите внимание: междугородные коды кое-где пропущены. Затем следуют поля фамилии и имени. В предпоследнем поле содержится общий объем продаж данного представителя, а в последнем — его доля в общем объеме продаж компании. Вы должны отформатировать данные в терминальном окне так, чтобы данные легко читались. Ниже приведен пример вывода:

```
                        Australia
                ---------------------------------

*Last Name*    *First Name*    *ID*    *Phone*         *Sales*    *Percent*

Langler        Tyson           24150   766-723-0284      31.24    4.21E-02
Oneill         Zeke            11159   XXX-758-6701     553.43    7.47E-01
```

и т. д.

Шаблоны



Шаблоны C++ представляют собой нечто гораздо большее, чем «контейнеры для T». Хотя первоначально шаблоны проектировались именно как основа для создания обобщенных контейнеров, безопасных по отношению к типам, в современном языке C++ шаблоны также используются для автоматического построения специализированного кода и оптимизации выполнения программ.

В этой главе мы с практической точки зрения рассмотрим возможности шаблонов в современном языке C++ (а также типичные ошибки, допускаемые программистами при их использовании).

Параметры шаблонов

Как отмечалось в первом томе, шаблоны делятся на две категории: шаблоны функций и шаблоны классов. Обе разновидности шаблонов полностью характеризуются своими параметрами. Параметры шаблонов могут представлять:

- типы (встроенные или пользовательские);
- константы времени компиляции (например, целые числа, указатели или ссылки на статические данные; часто называются *нетиповыми параметрами*);
- другие шаблоны.

Все шаблоны, упоминавшиеся в первом томе, относятся к первой, наиболее распространенной категории. Классическим примером простого контейнерного шаблона является стек (**Stack**). Для объекта **Stack** как контейнера неважно, объекты какого типа в нем хранятся; логика его работы не зависит от типа элементов. По этой причине тип элементов может быть параметризован:

```
template<class T>
class Stack {
  T* data;
  size_t count;
```

```
public:
  void push(const T& t);
  // И т. д.
};
```

Фактический тип, используемый конкретным экземпляром Stack, определяется аргументом, передаваемым в параметре T:

```
Stack<int> myStack;   // Контейнер Stack с элементами типа int
```

Компилятор создает версию Stack для типа int, для чего он подставляет int вместо T и генерирует соответствующий код. В данном случае будет сгенерирован специализированный класс с именем Stack<int>.

Нетиповые параметры шаблонов

Нетиповой параметр шаблона должен быть целочисленным значением, известным на стадии компиляции. Например, для создания стека фиксированного размера можно использовать нетиповой параметр, который будет определять размер базового массива:

```
template<class T, size_t N>
class Stack {
  T data[N];  // Фиксированная емкость N
  size_t count;
public:
  void push(const T& t);
  // И т. д.
};
```

При специализации шаблона в нетиповом параметре N необходимо передать константу, известную на стадии компиляции, например:

```
Stack<int, 100> myFixedStack;
```

Поскольку значение N известно на стадии компиляции, базовый массив (data) может храниться не в куче, а в стеке. Это позволяет повысить быстродействие программы, так как избавляет от издержек, связанных с динамическим выделением памяти. По аналогии с предыдущим примером, специализации будет присвоено имя Stack<int,100>. А это означает, что разные значения N порождают уникальные типы классов. Например, класс Stack<int,99> отличен от класса Stack<int,100>.

Из всех классов стандартной библиотеки C++ только шаблон bitset использует нетиповой параметр, определяющий количество битов в объекте (шаблон bitset подробно рассматривается в главе 7). В следующем примере генератор случайных чисел задействует объект bitset для отслеживания сгенерированных чисел, чтобы числа начинали повторяться только после полного перебора всего интервала. Обратите внимание на перегрузку оператора (), обеспечивающую привычный синтаксис вызова функции.

```
//: C05:Urand.h {-bor}
// Генератор уникальных случайных чисел
#ifndef URAND_H
#define URAND_H
#include <bitset>
#include <cstddef>
#include <cstdlib>
#include <ctime>
using std::size_t;
```

```
using std::bitset;

template<size_t UpperBound>
class Urand {
  bitset<UpperBound> used;
public:
  Urand() { srand(time(0)); } // Раскрутка
  size_t operator()(); // Функция-генератор
};

template<size_t UpperBound>
inline size_t Urand<UpperBound>::operator()() {
  if(used.count() == UpperBound)
    used.reset();  // Сброс (очистка объекта bitset)
  size_t newval;
  while(used[newval = rand() % UpperBound])
    ; // Пока не будет найдено уникальное значение
  used[newval] = true;
  return newval;
}
#endif // URAND_H ///:~
```

Программа Urand генерирует только случайные числа без повторений, поскольку объект bitset с именем used отслеживает все возможные числа в интервале (верхняя граница задается аргументом шаблона) и запоминает каждое число, устанавливая бит в соответствующей позиции. После того как все числа будут использованы, объект bitset сбрасывается, и все начинается сначала. По следующей простой тестовой программе можно понять, как работать с объектом Urand:

```
//: C05:UrandTest.cpp {-bor}
#include <iostream>
#include "Urand.h"
using namespace std;

int main() {
  Urand<10> u;
  for(int i = 0; i < 20; ++i)
    cout << u() << ' ';
} ///:~
```

Как будет показано в этой главе, нетиповые аргументы шаблонов также играют важную роль при оптимизации математических вычислений.

Аргументы шаблонов по умолчанию

Аргументы шаблонов могут иметь значения по умолчанию в шаблонах классов, но не в шаблонах функций. Как и обычные аргументы функций по умолчанию, они должны определяться только один раз, когда компилятор впервые встречает объявление или определение шаблона. Если одному из параметров шаблона был назначен аргумент по умолчанию, они также должны быть назначены всем аргументам, следующим за ним. Например, чтобы сделать шаблон стека фиксированного размера чуть более удобным, можно добавить определение аргумента по умолчанию:

```
template<class T, size_t N = 100> class Stack {
  T data[N];  // Фиксированная емкость N
  size_t count;
```

```
public:
  void push(const T& t);
  // И т. д.
};
```

Если теперь при объявлении шаблона опустить второй аргумент, то значение N по умолчанию будет равным 100.

Значения по умолчанию можно определить для всех аргументов, но тогда при объявлении экземпляра необходимо использовать пустые угловые скобки. Без них компилятор не поймет, что речь идет о шаблоне класса:

```
template<class T = int, size_t N = 100>  // Значения по умолчанию
class Stack {                            // у обоих параметров
  T data[N];  // Фиксированная емкость N
  size_t count;
public:
  void push(const T& t);
  // И т. д.
};

Stack<> myStack;  // То же, что Stack<int,100>
```

Аргументы по умолчанию широко используются в стандартной библиотеке C++. Например, шаблон класса vector объявляется следующим образом:

```
template <class T, class Allocator = allocator<T> >
class vector;
```

Обратите внимание на пробел между правыми угловыми скобками. Без него компилятор интерпретировал бы эти символы (>>) как оператор сдвига.

Объявление показывает, что шаблон vector получает два аргумента: тип хранящихся в нем объектов и тип используемого шаблоном vector распределителя памяти. Если второй аргумент не указан, применяется стандартный шаблон allocator, параметризованный по типу первого параметра шаблона. Данное объявление также показывает, что параметры шаблонов тоже могут указываться в последующих параметрах шаблонов, как параметр T в нашем примере.

Хотя аргументы по умолчанию не разрешается использовать в шаблонах функций, параметры шаблонов могут быть аргументами по умолчанию обычных функций. Следующий шаблон функции суммирует элементы последовательности:

```
//: C05:FuncDef.cpp
#include <iostream>
using namespace std;

template<class T> T sum(T* b, T* e, T init = T()) {
  while(b != e)
    init += *b++;
  return init;
}

int main() {
  int a[] = {1,2,3};
  cout << sum(a, a+sizeof a / sizeof a[0]) << endl; // 6
} ///:~
```

Третий аргумент sum() определяет начальное значение накапливаемой суммы. Поскольку при вызове он не указан, по умолчанию этот аргумент равен T(). В слу-

чае типа int и других встроенных типов это приводит к вызову псевдоконструктора и инициализации переменной нулями.

Шаблоны как параметры шаблонов

Третья разновидность параметров, передаваемых шаблонам, — другие шаблоны классов. Если вы собираетесь использовать в качестве параметра шаблон, то компилятор должен знать, что это именно шаблон. Следующий пример демонстрирует применение шаблонов в качестве параметров шаблонов:

```
//: C05:TempTemp.cpp
// Шаблоны как параметры шаблонов
#include <cstddef>
#include <iostream>
using namespace std;

template<class T>
class Array {       // Простой, динамически расширяемый массив
  enum {INIT = 10};
  T *data;
  size_t capacity;
  size_t count;
public:
  Array() {
    count = 0;
    data = new T[capacity = INIT];
  }
  ~Array() { delete [] data; }
  void push_back(const T& t) {
    if(count == capacity) {
      // Увеличение базового массива
      size_t newCap = 2*capacity;
      T* newData = new T[newCap];
      for (size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete data;
      data = newData;
      capacity = newCap;
    }
    data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T, template<class> class Seq>
class Container {
  Seq<T> seq;
public:
  void append(const T& t) {
    seq.push_back(t);
  }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
```

```
};

int main() {
  Container<int, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

Шаблон класса Array представляет собой тривиальный контейнер для хранения последовательности элементов. Шаблон Container получает два параметра: тип хранящихся в нем объектов и шаблон, предназначенный для хранения данных. Следующая строка в реализации класса Container требует, чтобы мы сообщили компилятору о том, что Seq является шаблоном:

```
Seq<T> seq;
```

Если не объявить Seq как шаблон, являющийся параметром шаблона, то компилятор пожалуется, что Seq не является шаблоном, поскольку мы задействуем его в нешаблонном контексте. В функции main() создается специализация Container, использующая шаблон Array для хранения int, так что Seq фактически является синонимом для Array.

Обратите внимание: задавать имя параметра Seq в объявлении Container в данном случае не нужно. Речь идет о строке

```
template<class T, template<class> class Seq>
```

Хотя с таким же успехом можно было написать следующую строку, параметр U нигде не используется:

```
template<class T, template<class U> class Seq>
```

Важно лишь то, что Seq — шаблон класса, получающий один параметр-тип. Можно провести аналогию с пропуском имен неиспользуемых параметров функций, как при перегрузке постфиксного оператора ++:

```
T operator++(int);
```

В данном случае тип int передает всю необходимую информацию, и имя переменной не требуется.

В следующей программе используется массив фиксированного размера, в шаблоне которого определен дополнительный параметр, задающий длину массива:

```
//: C05:TempTemp2.cpp
// Шаблон как параметр шаблона с несколькими параметрами
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N>
class Array {
  T data[N];
  size_t count;
public:
  Array() { count = 0; }
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
```

```
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T,size_t N,template<class,size_t> class Seq>
class Container {
  Seq<T,N> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  const size_t N = 10;
  Container<int, N, Array> container;
  theData.append(1);
  theData.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

И снова при объявлении Seq в объявлении Container имена параметров не обязательны, но переменная seq должна объявляться с двумя параметрами, чем и объясняется появление нетипового параметра N на верхнем уровне.

Использование аргументов по умолчанию совместно с шаблонами в качестве параметров шаблонов создает определенные проблемы. Когда компилятор анализирует внутренние параметры шаблонов, являющихся параметрами шаблонов, он не рассматривает аргументы по умолчанию, поэтому вам придется повторно перечислить значения по умолчанию для обеспечения точного совпадения. Следующий пример, в котором используется аргумент по умолчанию для шаблона Array с фиксированным размером, показывает, как решается проблема с этой странностью языка:

```
//: C05:TempTemp3.cpp {-bor}{-msc}
// Использование шаблонов как параметров шаблонов
// совместно с аргументами по умолчанию
#include <cstddef>
#include <iostream>
using namespace std;

template<class T, size_t N = 10>  // Аргумент по умолчанию
class Array {
  T data[N];
  size_t count;
public:
  Array() { count = 0; }
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
  }
```

```
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};

template<class T, template<class, size_t = 10> class Seq>
class Container {
  Seq<T> seq;  // Используется значение по умолчанию
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  Container<int, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
} ///:~
```

Значение по умолчанию 10 должно присутствовать в следующей строке:

```
template<class T, template<class, size_t = 10> class Seq>
```

Это значение требуется как в определении seq в Container, так и в определении контейнера в main(). Единственный способ использования значения, отличного от значения по умолчанию, был показан в примере TempTemp2.cpp. Это единственное исключение из правила, гласящего, что аргументы по умолчанию должны лишь один раз указываться в единице трансляции.

Поскольку стандартные последовательные контейнеры vector, list и deque (подробнее см. в главе 7) содержат аргумент-распределитель со значением по умолчанию, показанная методика пригодится, если вам когда-нибудь потребуется передать один из этих последовательных контейнеров в качестве параметра шаблона. В следующей программе контейнеры vector и list передаются двум экземплярам шаблона Container:

```
//: C05:TempTemp4.cpp {-bor}{-msc}
// Передача стандартных последовательных контейнеров
// в аргументах шаблонов
#include <iostream>
#include <list>
#include <memory>  // Объявление allocator<T>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
                  class Seq>
class Container {
  Seq<T> seq; // Неявное применение allocator<T>
public:
  void push_back(const T& t) { seq.push_back(t); }
  typename Seq<T>::iterator begin() { return seq.begin(); }
```

```
  typename Seq<T>::iterator end() { return seq.end(); }
};

int main() {
  // Передача контейнера vector
  Container<int, vector> vContainer;
  vContainer.push_back(1);
  vContainer.push_back(2);
  for(vector<int>::iterator p = vContainer.begin();
      p != vContainer.end(); ++p) {
    cout << *p << endl;
  }
  // Передача контейнера list
  Container<int, list> lContainer;
  lContainer.push_back(3);
  lContainer.push_back(4);
  for(list<int>::iterator p2 = lContainer.begin();
      p2 != lContainer.end(); ++p2) {
    cout << *p2 << endl;
  }
} ///:~
```

Мы присваиваем имя U первому параметру внутреннего шаблона Seq, поскольку распределители стандартных последовательных контейнеров должны быть параметризованы по тому же типу, что и объекты, хранящиеся в последовательном контейнере. Кроме того, так как об использовании параметра allocator по умолчанию уже известно, мы можем исключить его из последующих ссылок на Seq<T>, как в предыдущей версии. Однако для полного понимания этого примера необходимо подробнее объяснить семантику ключевого слова typename.

Ключевое слово typename

Рассмотрим следующий фрагмент:

```
//: C05:TypenamedID.cpp {-bor}
// typename в качестве префикса вложенных типов

template<class T> class X {
  // Без typename в программе произойдет ошибка:
  typename T::id i;
public:
  void f() { i.g(); }
};

class Y {
public:
  class id {
  public:
    void g() {}
  };
};

int main() {
  X<Y> xy;
  xy.f();
} ///:~
```

Определение шаблона предполагает, что в передаваемом ему классе T должен содержаться некий вложенный идентификатор с именем id. Однако id также может быть статической переменной T; в этом случае вы сможете напрямую выполнять операции с id, но не сможете «создать объект типа id». В приведенном примере идентификатор id интерпретируется как вложенный тип по отношению к T. В случае класса Y идентификатор id действительно является вложенным типом, но без ключевого слова typename компилятор не будет знать об этом при компиляции X.

Если компилятор, встречая идентификатор в шаблоне, может интерпретировать его как тип или как нечто иное, он всегда выбирает «нечто иное». Иначе говоря, компилятор будет считать, что идентификатор относится к объекту (включая переменные примитивных типов), перечисляемому типу или чему-нибудь в этом роде. Но он никогда не будет — а точнее, не сможет — интерпретировать его как тип.

Раз компилятор по умолчанию предполагает, что потенциально неоднозначное имя не является типом, необходимо использовать ключевое слово typename для вложенных типов (кроме списков инициализаторов в конструкторах, где это ключевое слово не только не нужно, но и недопустимо). Когда в предыдущем примере компилятор встречает конструкцию typename T::id, он знает (благодаря ключевому слову typename), что идентификатор id относится к вложенному типу, и что он может создать объект этого типа.

Короче говоря, если тип, на который вы ссылаетесь в коде шаблона, уточняется параметром-шаблоном, обязательно используйте префикс typename везде, кроме спецификаций базового класса или списка инициализаторов в той же области видимости.

Предыдущий пример поясняет, для чего нужно ключевое слово typename в программе TempTemp4.cpp. Без него компилятор считает, что выражение Seq<T>::iterator не является типом, но мы используем его для определения возвращаемых типов функций begin() и end().

Аналогичное применение typename продемонстрировано в следующем примере. Эта программа выводит содержимое любого стандартного последовательного контейнера C++:

```
//: C05:PrintSeq.cpp {-msc}{-mwcc}
// Функция вывода стандартных последовательных контейнеров C++
#include <iostream>
#include <list>
#include <memory>
#include <vector>
using namespace std;

template<class T, template<class U, class = allocator<U> >
                  class Seq>
void printSeq(Seq<T>& seq) {
  for (typename Seq<T>::iterator b = seq.begin();
       b != seq.end();)
    cout << *b++ << endl;
}

int main() {
  // Вывод содержимого вектора
  vector<int> v;
```

```
  v.push_back(1);
  v.push_back(2);
  printSeq(v);
  // Вывод содержимого списка
  list<int> lst;
  lst.push_back(3);
  lst.push_back(4);
  printSeq(lst);
} ///:~
```

Как и прежде, без ключевого слова typename компилятор интерпретирует iterator как статическую переменную Seq<T>, что является синтаксической ошибкой, поскольку в этом месте должен быть указан тип.

Не стоит полагать, будто ключевое слово typename создает новое имя типа. Оно всего лишь сообщает компилятору, что уточненный идентификатор должен интерпретироваться как тип. Следующая строка объявляет переменную It как относящуюся к типу Seq<T>::iterator:

```
typename Seq<T>::iterator It;
```

Если же вы хотите создать новый тип, воспользуйтесь ключевым словом typedef:

```
typedef typename Seq<T>::iterator It;
```

У ключевого слова typename существует и другое применение — оно может использоваться вместо ключевого слова class в списке аргументов шаблона при его определении:

```
//: C05:UsingTypename.cpp
// typename в списке аргументов шаблона.

template<typename T> class X { };

int main() {
  X<int> x;
} ///:~
```

Некоторые программисты считают, что это делает программу более наглядной.

Ключевое слово template

Итак, ключевое слово typename помогает компилятору понять в потенциально неоднозначной ситуации, что речь идет именно о типе. Аналогичные трудности возникают с лексемами, которые не являются идентификаторами, например, символами < и >. В одних случаях эти лексемы представляют знаки «больше» и «меньше», в других ограничивают списки параметров шаблонов. В качестве примера снова воспользуемся классом bitset:

```
//: C05:DotTemplate.cpp
// Конструкция .template
#include <bitset>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

template<class charT, size_t N>
basic_string<charT> bitsetToString(const bitset<N>& bs) {
  return bs. template to_string<charT, char_traits<charT>,
```

```
                                    allocator<charT> >();
}

int main() {
  bitset<10> bs;
  bs.set(1);
  bs.set(5);
  cout << bs << endl; // 0000100010
  string s = bitsetToString<char>(bs);
  cout << s << endl;  // 0000100010
} ///:~
```

Функция to_string класса bitset преобразует содержимое битового поля в строку. Для поддержки разных строковых классов функция to_string оформлена в виде шаблона по образцу шаблона basic_string (см. главу 3). Объявление to_string в bitset выглядит так:

```
template <class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> to_string() const;
```

Наша функция bitsetToString() позволяет выбрать нужный тип строкового представления bitset. Например, для получения расширенной строки применяется следующий вызов:

```
wstring s = bitsetToString<wchar_t>(bs);
```

Обратите внимание: шаблон basic_string использует аргументы по умолчанию, поэтому нам не нужно повторять аргументы char_traits и allocator в возвращаемом значении. К сожалению, функция bitset::to_string не имеет аргументов по умолчанию, и написать bitsetToString<char>(bs) гораздо удобнее, чем каждый раз вводить полную запись:

```
bs.template to_string<char, char_traits, allocator<char> >()
```

Команда return в bitsetToString() содержит ключевое слово template в неожиданном месте — после оператора точка (.), примененного к объекту bs типа bitset. Дело в том, что при разборе шаблона символ < после лексемы to_string будет восприниматься как операция «меньше», а не как начало списка аргументов шаблона. Ключевое слово template в этом контексте сообщает компилятору, что за ним следует имя шаблона, благодаря чему символ < интерпретируется правильно. То же относится и к операторам -> и ::, применяемым к шаблонам. Данный способ разрешения неоднозначностей, как и ключевое слово typename, может использоваться только в коде шаблонов[1].

Вложенные шаблоны

Шаблон функции bitset::to_string() является примером *вложенного шаблона*, то есть шаблона, объявленного в другом классе или шаблоне класса. Вложенные шаблоны позволяют использовать различные комбинации независимых аргументов шаблонов. Один из полезных примеров встречается в шаблоне класса complex стандартной библиотеки C++. Шаблон complex имеет типовой параметр для представления базового вещественного типа, предназначенного для хранения вещественной и мнимой частей комплексного числа. В следующем фрагменте кода стандартной биб-

[1] Комитет по стандартизации C++ собирается снять это ограничение, а некоторые компиляторы уже сейчас позволяют использовать эти конструкции в обычном (нешаблонном) коде.

лиотеки показан конструктор шаблонного класса complex, объявленный в виде вложенного шаблона:

```
template<typename T> class complex {
public:
  template<class X> complex(const complex<X>&);
```

В стандартную библиотеку включены готовые специализации стандартного шаблона complex по параметру T для типов float, double и long double. Вложенный шаблон-конструктор создает новый объект комплексного числа с другим базовым вещественным типом, как в следующем фрагменте:

```
complex<float> z(1,2);
complex<double> w(z);
```

В этом объявлении w параметр T шаблона complex замещается типом double, а параметр X — типом float. Вложенные шаблоны заметно упрощают реализацию подобных гибких преобразований.

Вложенный шаблон определяется внутри другого шаблона, и это обстоятельство должно быть отражено в префиксах при определении вложенного шаблона за пределами вмещающего класса. Например, если при реализации шаблона complex вложенный шаблон-конструктор определяется вне определения класса complex, это должно выглядеть так:

```
template<typename T>
template<typename X>
complex<T>::complex(const complex<X>& c) {/*Тело определения...*/}
```

Вложенные шаблоны функций также применяются в стандартной библиотеке для инициализации контейнеров. Предположим, имеется вектор с элементами int, и мы хотим использовать его для инициализации нового вектора с элементами double:

```
int data[5] = {1,2,3,4,5};
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());
```

Пока элементы v1 остаются совместимыми по присваиванию с элементами v2 (как double и int), можно не волноваться. В шаблоне класса vector определен следующий конструктор, оформленный в виде вложенного шаблона:

```
template <class InputIterator>
vector(InputIterator first, InputIterator last,
       const Allocator& = Allocator());
```

В приведенных выше объявлениях vector этот конструктор вызывается дважды. При инициализации v1 на базе массива int тип InputIterator соответствует int*. При инициализации v2 на базе v1 используется экземпляр конструктора, в котором InputIterator представляет vector<int>::iterator.

Вложенные шаблоны тоже могут быть классами (не только функциями). В следующем примере вложенный шаблон класса определяется за пределами вмещающего класса:

```
//: C05:MemberClass.cpp
// Вложенный шаблон класса
#include <iostream>
#include <typeinfo>
using namespace std;

template<class T> class Outer {
```

```
public:
  template<class R> class Inner {
  public:
    void f();
  };
};

template<class T> template <class R>
void Outer<T>::Inner<R>::f() {
  cout << "Outer == " << typeid(T).name() << endl;
  cout << "Inner == " << typeid(R).name() << endl;
  cout << "Full Inner == " << typeid(*this).name() << endl;
}

int main() {
  Outer<int>::Inner<bool> inner;
  inner.f();
} ///:~
```

Оператор typeid (см. главу 8) получает один аргумент и возвращает объект type_info; функция name() этого объекта возвращает строку, представляющую тип аргумента. Например, вызов typeid(int).name() возвращает строку "int" (впрочем, вид возвращаемой строки зависит от платформы). Оператор typeid может получить выражение и вернуть объект type_info, описывающий тип этого выражения. Скажем, для int i вызов typeid(i).name() вернет нечто вроде "int", а typeid(&i).name() — что-нибудь вроде "int *".

Результат выполнения приведенной выше программы выглядит примерно так:

```
Outer == int
Inner == bool
Full Inner == Outer<int>::Inner<bool>
```

Объявление переменной inner в основной программе создает специализации Inner<bool> и Outer<int>.

Вложенные шаблонные функции не могут объявляться виртуальными. Современная технология компиляторов предполагает возможность определения размера таблицы виртуальных функций класса в процессе компиляции класса. Возможность использования виртуальных вложенных шаблонных функций требовала бы, чтобы информация обо всех вызовах таких функций была доступна заранее. Такое требование неприемлемо, особенно если проект состоит из большого количества файлов.

Шаблоны функций

Подобно тому как шаблон класса описывает семейство классов, шаблон функции описывает семейство функций. Обе разновидности шаблонов создаются практически одинаково, но различаются по способу использования. При специализации шаблона класса всегда необходимо использовать угловые скобки и передавать все аргументы, для которых не определены значения по умолчанию. С другой стороны, в шаблонах функций аргументы часто можно опускать, а аргументы по умолчанию вообще запрещены. Рассмотрим типичную реализацию шаблона функции min() из заголовочного файла <algorithm>:

```
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
```

При вызове этого шаблона можно указать типы аргументов в угловых скобках, как это делается при использовании шаблонов классов:

```
int z = min<int>(i, j);
```

Такой синтаксис сообщает компилятору, что он должен генерировать специализацию шаблона min, в которой параметр T замещается типом int. По аналогии с именами классов, сгенерированным по шаблонам, можно считать, что компилятор генерирует функцию с именем min<int>().

Определение типа аргументов в шаблонах функций

Всегда можно явно указать специализацию шаблона функции, как в предыдущем примере. Но часто бывает удобнее поручить компилятору вычислить его на основании переданных аргументов:

```
int z = min (i, j);
```

Если i и j относятся к одному типу int, компилятор понимает, что вам нужна специализация min<int>(), и генерирует ее автоматически. Типы должны совпадать, потому что шаблон изначально определялся с одним типовым аргументом, который применялся к обоим параметрам функции. К аргументам функций, тип которых определяется параметром шаблона, не применяются стандартные преобразования. Например, если вы захотите вычислить минимальное значение из int и double, следующая попытка вызова min() закончится неудачей:

```
int z = min (x, j);  // Переменная x типа double
```

Поскольку x и j относятся к разным типам, компилятору не удается подобрать единый тип для параметра T в определении min(), и вызов не соответствует объявлению шаблона. Проблема решается явным преобразованием одного аргумента к типу другого аргумента или полным уточнением синтаксиса:

```
int z = min<double>(x,j);
```

Вы приказываете компилятору сгенерировать версию min() для double, после чего переменная j может быть преобразована в double по стандартным правилам (потому что тогда будет существовать функция min<double>(const double&,const double&)).

Возникает искушение определить min() с двумя аргументами, чтобы функция могла вызываться с независимыми типами аргументов:

```
template<typename T, typename U>
const T& min(const T& a, const U& b) {
  return (a < b) ? a : b;
}
```

Нередко такое решение оправданно, но в данном случае оно порождает проблемы. Функция min() возвращает значение, и невозможно сколь-нибудь обоснованно выбрать, какой тип из двух следует вернуть (T или U)?

Если тип возвращаемого значения шаблона функции является независимым параметром шаблона, он всегда должен явно задаваться при вызове, поскольку его невозможно вычислить на основании аргументов. Примером служит приведенный далее шаблон fromString:

```
//: C05:StringConv.h
// Шаблоны функций для преобразования в строку и обратно
#ifndef STRINGCONV_H
#define STRINGCONV_H
#include <string>
#include <sstream>

template<typename T>
T fromString(const std::string& s) {
  std::istringstream is(s);
  T t;
  is >> t;
  return t;
}

template<typename T>
std::string toString(const T& t) {
  std::ostringstream s;
  s << t;
  return s.str();
}
#endif // STRINGCONV_H ///:~
```

Эти шаблоны обеспечивают преобразование в std::string и обратно для любого типа, для которого определен потоковый оператор << или >> соответственно. Ниже приведена тестовая программа, в которой используется тип complex из стандартной библиотеки:

```
//: C05:StringConvTest.cpp
#include <complex>
#include <iostream>
#include "StringConv.h"
using namespace std;

int main() {
  int i = 1234;
  cout << "i == \"" << toString(i) << "\"" << endl;
  float x = 567.89;
  cout << "x == \"" << toString(x) << "\"" << endl;
  complex<float> c(1.0, 2.0);
  cout << "c == \"" << toString(c) << "\"" << endl;
  cout << endl;

  i = fromString<int>(string("1234"));
  cout << "i == " << i << endl;
  x = fromString<float>(string("567.89"));
  cout << "x == " << x << endl;
  c = fromString< complex<float> >(string("(1.0,2.0)"));
  cout << "c == " << c << endl;
} ///:~
```

Результат выполнения программы вполне предсказуем:

```
i == "1234"
x == "567.89"
c == "(1,2)"

i == 1234
x == 567.89
c == (1,2)
```

Обратите внимание: при каждой специализации fromString() в вызов включается параметр шаблона. Если у вас имеется шаблон функции, параметры которого определяют типы как параметров функции, так и ее возвращаемого значения, очень важно начать с объявления параметра для типа возвращаемого значения. В противном случае вам придется обязательно указывать типы параметров функции при перечислении параметров шаблона. Для примера рассмотрим шаблон функции, знакомый многим по книгам Страуструпа (Stroustrup):

```
//: C05:ImplicitCast.cpp
template<typename R, typename P>
R implicit_cast(const P& p) {
  return p;
}

int main() {
  int i = 1;
  float x = implicit_cast<float>(i);
  int j = implicit_cast<int>(x);
  // char* p = implicit_cast<char*>(i);
} ///:~
```

Если поменять местами R и P в списке параметров, находящемся во второй строке файла, программа не будет компилироваться, потому что тип возвращаемого значения останется неизвестным (первый параметр шаблона будет соответствовать типу параметра функции). Последняя (закомментированная) строка недопустима, поскольку стандартного преобразования int в char* не существует. Вызов implicit_cast выявляет естественные преобразования типов в вашей программе.

Если действовать достаточно осмотрительно, вы даже сможете вычислить размеры массивов. В следующем примере эти вычисления выполняются в шаблоне функции инициализации массива (init2):

```
//: C05:ArraySize.cpp
#include <cstddef>
using std::size_t;

template<size_t R, size_t C, typename T>
void init1(T a[R][C]) {
  for (size_t i = 0; i < R; ++i)
    for (size_t j = 0; j < C; ++j)
      a[i][j] = T();
}

template<size_t R, size_t C, class T>
void init2(T (&a)[R][C]) {  // Ссылочный параметр
  for (size_t i = 0; i < R; ++i)
    for (size_t j = 0; j < C; ++j)
      a[i][j] = T();
}

int main() {
  int a[10][20];
  init1<10,20>(a);  // Обязательно задаются размеры
  init2(a);         // Автоматическое определение размеров
} ///:~
```

Размеры массивов не являются частью типа параметра функции, если только этот параметр не передается в виде указателя или ссылки. Шаблон функции init2 объявляет a ссылкой на двумерный массив, поэтому его размеры R и C определяются автоматически. Таким образом, init2 является удобным средством инициализации двумерных массивов произвольного размера. В шаблоне init1 массив не передается по ссылке, поэтому размер приходится задавать явно (хотя параметр типа все равно может быть определен автоматически).

Перегрузка шаблонов функций

Одноименные шаблоны функций могут перегружаться так же, как и обычные функции. Обрабатывая вызов функции в программе, компилятор должен решить, шаблон или обычная функция «лучше всего» подходит для данного вызова.

Давайте добавим к уже рассматривавшемуся шаблону функции min() несколько обычных функций:

```
//: C05:MinTest.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
using std::cout;
using std::endl;

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}
double min(double x, double y) {
  return (x < y) ? x : y;
}

int main() {
  const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
  cout << min(1, 2) << endl;       // 1: 1 (шаблон)
  cout << min(1.0, 2.0) << endl;   // 2: 1 (double)
  cout << min(1, 2.0) << endl;     // 3: 1 (double)
  cout << min(s1, s2) << endl;     // 4: knights who (const char*)
  cout << min<>(s1, s2) << endl;   // 5: say "Ni-!"
                                   //    (шаблон)
} ///:~
```

Помимо шаблона функции в этой программе определяются две нешаблонные функции: версия min() для строк C и для double. Если бы шаблона не было, то для первой строки была бы вызвана double-версия min(), так как существует стандартное преобразование int в double. Однако шаблон может сгенерировать int-версию, которая обеспечивает лучшее совпадение; именно это и происходит в программе. Вызов во второй строке точно совпадает с double-версией, а в третьей строке снова вызывается та же функция, при этом 1 автоматически преобразуется в 1.0. В четвертой строке напрямую вызывается версия min() для const char*. В строке 5 мы заставляем компилятор задействовать шаблон, присоединяя пустую пару угловых скобок к имени функции. Компилятор генерирует специализацию для const char* и использует ее (в этом можно убедиться по неправильному ответу — компилятор

просто сравнивает адреса[1]!) Вероятно, вас интересует, почему мы указываем объявление using вместо директивы using namespace std? Потому что некоторые компиляторы автоматически включают заголовочные файлы, содержащие std::min(), и эта версия будет конфликтовать с именем нашей функции min().

Как отмечалось выше, перегрузка одноименных шаблонов допускается при условии, что компилятор сможет различить их. Например, можно объявить шаблон функции min() с тремя аргументами:

```
template<typename T>
const T& min(const T& a, const T& b, const T& c);
```

Специализации этого шаблона будут сгенерированы только для вызовов min() с тремя однотипными аргументами.

Получение адреса сгенерированного шаблона функции

В некоторых ситуациях требуется узнать адрес функции. Например, в программе может использоваться функция, в параметре которой передается указатель на другую функцию. Вполне возможно, что эта другая функция будет сгенерирована на базе шаблонной функции, поэтому необходим механизм получения таких адресов[2]:

```
//: C05:TemplateFunctionAddress.cpp {-mwcc}
// Получение адреса функции, сгенерированной из шаблона

template <typename T> void f(T*) {}

void h(void (*pf)(int*)) {}

template <typename T>
  void g(void (*pf)(T*)) {}

int main() {
  // Полная спецификация типа:
  h(&f<int>);
  // Вычисление типа:
  h(&f);
  // Полная спецификация типа:
  g<int>(&f<int>);
  // Автоматическое определение типа:
  g(&f<int>);
  // Неполная (но достаточная) спецификация типа
  g<int>(&f);
} ///:~
```

В этом примере стоит обратить внимание на ряд моментов. Во-первых, даже при использовании шаблонов сигнатуры должны совпадать. Функция h() получает указатель на функцию с аргументом int*, возвращающую void, и эта сигнатура должна быть сгенерирована шаблоном f(). Во-вторых, функция, получающая указатель на функцию, сама может быть шаблоном, как g().

Функция main() также показывает, как автоматически определяются типы. При первом вызове h() явно указано, что аргумент f() является шаблоном, но так как в соответствии с объявлением h() эта функция должна получать адрес функции

---

[1] Строго говоря, результат сравнения двух указателей, не принадлежащих одному массиву, не определен, но многие компиляторы на это не жалуются... Тем больше причин сделать все так, как положено.

[2] Спасибо Натану Майерсу (Nathan Myers) за предоставленный пример.

с параметром int*, компилятор мог бы и сам придти к этому выводу. В случае с g() возникает еще более интересная ситуация, в которой участвуют два шаблона. Компилятор не может определить тип без исходных данных, но если f() или g() передается int, остальное можно вычислить.

Другая нетривиальная проблема возникает при передаче в качестве параметров функций tolower и toupper, определенных в файле <cctype>. Например, эти функции могут использоваться в сочетании с алгоритмом transform (см. следующую главу) для преобразования строки к нижнему или верхнему регистру. Будьте осторожны: у этих функций имеется несколько определений. Наивное решение выглядит примерно так:

```
// Переменная s относится к типу std::string
transform(s.begin(), s.end(), s.begin(), tolower);
```

Алгоритм transform применяет свой четвертый параметр (tolower в данном примере) к каждому символу s и помещает результат обратно в s. Таким образом, все символы s заменяются своими эквивалентами в нижнем регистре. Но в том виде, в котором эта команда приведена выше, она может и не работать! Например, в следующем контексте произойдет неудача:

```
//: C05:FailedTransform.cpp {-xo}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s("LOWER");
  transform(s.begin(),s.end(),s.begin(),tolower);
  cout << s << endl;
} ///:~
```

Даже если компиляция пройдет успешно, это все равно ошибка. Дело в том, что в заголовке <iostream> также объявляются версии tolower() и toupper() с двумя аргументами:

```
template <class charT> charT toupper(charT c, const locale& loc);
template <class charT> charT tolower(charT c, const locale& loc);
```

Эти шаблоны функций получают второй аргумент типа locale. Компилятор не может определить, какую версию tolower() он должен использовать — с одним аргументом, определенную в <cctype>, или приведенную выше. Проблема (почти) решается преобразованием типа в вызове tranform (вспомните, что tolower() и toupper() работают с int вместо char):

```
transform(s.begin(),s.end(),s.begin() static_cast<int (*)(int)>(tolower));
```

Преобразование типа однозначно указывает, что в программе должна использоваться версия tolower() с одним аргументом. На некоторых компиляторах такое решение работает, но это без гарантии. Дело в том, что для функций, унаследованных из языка C, реализации библиотеки позволено задействовать «компоновку C» (то есть не включать в имена функций дополнительную информацию, характерную для имен функций C++[1]). В этом случае попытка преобразования завер-

[1] В частности, информацию о типах, закодированную в «украшенных» именах функций.

шается неудачей, поскольку transform как шаблон функции C++ предполагает, что четвертый аргумент использует компоновку C++, а преобразование типа не может изменять тип компоновки.

Проблема решается вызовами tolower() в однозначном контексте. Например, вы можете написать функцию strTolower() и поместить ее в отдельный файл без включения заголовка <iostream>:

```
//: C05:StrTolower.cpp {O} {-mwcc}
#include <algorithm>
#include <cctype>
#include <string>
using namespace std;

string strTolower(string s) {
  transform(s.begin(), s.end(), s.begin(), tolower);
  return s;
} ///:~
```

Файл не включает заголовочный файл <iostream>, поэтому компилятор в этом контексте не включает двухаргументную версию tolower()[1], и проблем не возникает. Далее новая функция используется обычным образом:

```
//: C05:Tolower.cpp {-mwcc}
//{L} StrTolower
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;
string strTolower(string);

int main() {
  string s("LOWER");
  cout << strTolower(s) << endl;
} ///:~
```

В другом решении создается функция-оболочка, которая явно вызывает правильную версию tolower():

```
//: C05:ToLower2.cpp {-mwcc}
#include <algorithm>
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

template<class charT>
charT strTolower(charT c) {
  return tolower(c);  // Вызов версии с одним аргументом
}

int main() {
  string s("LOWER");
  transform(s.begin(),s.end(),s.begin(),&strTolower<char>);
  cout << s << endl;
} ///:~
```

[1] Вообще говоря, компиляторы C++ могут вводить новые имена по своему усмотрению. К счастью, большинство компиляторов не объявляет неиспользуемые имена.

У этого решения имеется важное преимущество: оно работает и с узкими, и с расширенными потоками, поскольку базовый тип символов определяется параметром шаблона. Комитет по стандартизации C++ сейчас занимается такой модификацией языка, при которой бы работало самое первое решение (без преобразования типа), поэтому когда-нибудь об этих обходных решениях можно будет забыть.

Применение функции к последовательным контейнерам STL

Допустим, вы хотите вызвать некоторую функцию класса для всех объектов, хранящихся в последовательном контейнере STL (контейнеры STL рассматриваются в следующих главах, а пока воспользуемся знакомым контейнером vector). Поскольку vector может содержать объекты любого типа, вам понадобится функция, работающая с любой разновидностью vector:

```
//: C05:ApplySequence.h
// Применение функции к элементам последовательного контейнера STL

// Константная функция, 0 аргументов,
// произвольный тип возвращаемого значения:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)() const) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
    ((*it++)->*f)();
  }
}

// Константная функция, 1 аргумент,
// произвольный тип возвращаемого значения:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A) const, A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
    ((*it++)->*f)(a);
  }
}

// Константная функция, 2 аргумента,
// произвольный тип возвращаемого значения:
template<class Seq, class T, class R, class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2) const,
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
    ((*it++)->*f)(a1, a2);
  }
}

// Неконстантная функция, 0 аргументов,
// произвольный тип возвращаемого значения:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)()) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
```

```
    ((*it++)->*f)();
  }
}

// Неконстантная функция, 1 аргумент,
// произвольный тип возвращаемого значения:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A), A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
    ((*it++)->*f)(a);
  }
}

// Неконстантная функция, 2 аргумента,
// произвольный тип возвращаемого значения:
template<class Seq, class T, class R, class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2),
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
    ((*it++)->*f)(a1, a2);
  }
}

// И т. д. для наиболее вероятных аргументов ///:~
```

Шаблон функции apply() получает ссылку на класс контейнера и указатель на функцию класса объектов, содержащихся в контейнере. Шаблон перебирает элементы контейнера при помощи итератора и вызывает функцию для каждого объекта. Мы перегрузили функции по константности, поэтому шаблон может использоваться как с константными, так и с неконстантными функциями.

Обратите внимание: пример applySequence.h не включает заголовочные файлы STL (и вообще какие-либо заголовочные файлы), поэтому он не ограничивается контейнерами STL. С другой стороны, он делает определенные предположения, относящиеся к последовательным контейнерам STL (прежде всего по поводу имени и поведения iterator), а также считает, что элементы контейнера представляют собой указатели.

Наличие нескольких версий apply() еще раз показывает, как перегружаются шаблоны функций. Хотя эти шаблоны позволяют использовать любой тип возвращаемого значения (который на самом деле игнорируется и требуется лишь для обязательного соответствия типов), разные версии получают разное число аргументов, а поскольку это шаблон, аргументы могут относиться к произвольному типу. Единственное неудобство состоит в том, что не существует «супершаблона», который бы генерировал шаблоны за вас. Вы должны сами решить, сколько аргументов может понадобиться, и создать соответствующие определения.

Чтобы проверить, как работают перегруженные версии apply(), мы создадим класс Gromit[1] с функциями, получающими разное количество аргументов и возвращающих разные типы:

[1] Так зовут собаку, персонаж мультипликационных фильмов Ника Парка (Nick Park) о приключениях Уоллеса и Громита.

```
//: C05:Gromit.h
// Киберсобака. Содержит несколько функций
// с разным количеством аргументов.
#include <iostream>

class Gromit {
  int arf;
  int totalBarks;
public:
  Gromit(int arf = 1) : arf(arf + 1), totalBarks(0) {}
  void speak(int) {
    for(int i = 0; i < arf; i++) {
      std::cout << "arf! ";
      ++totalBarks;
    }
    std::cout << std::endl;
  }
  char eat(float) const {
    std::cout << "chomp!" << std::endl;
    return 'z';
  }
  int sleep(char, double) const {
    std::cout << "zzz..." << std::endl;
    return 0;
  }
  void sit() const {
    std::cout << " Sitting..." << std::endl;
  }
}; ///:~
```

Теперь мы можем воспользоваться шаблонными функциями apply() для применения функций класса Gromit к контейнеру vector<Gromit*>:

```
//: C05:ApplyGromit.cpp
// Тестирование ApplySequence.h
#include <cstddef>
#include <iostream>
#include <vector>
#include "ApplySequence.h"
#include "Gromit.h"
#include "../purge.h"
using namespace std;

int main() {
  vector<Gromit*> dogs;
  for(size_t i = 0; i < 5; i++)
    dogs.push_back(new Gromit(i));
  apply(dogs, &Gromit::speak, 1);
  apply(dogs, &Gromit::eat, 2.0f);
  apply(dogs, &Gromit::sleep, 'z', 3.0);
  apply(dogs, &Gromit::sit);
  purge(dogs);
} ///:~
```

Вспомогательная функция purge() вызывает delete для каждого элемента последовательного контейнера. Она определяется в главе 7 и используется во многих примерах книги.

Определение apply() довольно запутанно; вряд ли неопытный программист сможет разобраться в нем. Но зато применение apply() выглядит крайне просто и оче-

видно. Даже новичок сможет пользоваться им, зная, *что* оно должно делать, а не *как* оно это делает. К такому разделению функциональности вы должны стремиться в своих программных компонентах. Все сложные и малопонятные подробности остаются «за барьером», на стороне проектировщика. Пользователь имеет дело только с тем, что относится к решению его практических задач; он не видит реализации, не знает ее и никак от нее не зависит. В следующей главе будут представлены еще более гибкие средства применения функций к последовательным контейнерам.

Приоритеты шаблонов функций

Как уже упоминалось выше, компилятор отдает предпочтение перегрузке обычной функции min() перед использованием шаблона. Если уже имеется функция, соответствующая вызову, зачем генерировать другую? Однако при отсутствии обычных функций перегруженные шаблоны функций могут привести к неоднозначности. Чтобы вероятность этого была сведена к минимуму, для шаблонов функций определен приоритет, при котором выбирается *наиболее специализированный* шаблон. Шаблон функции считается более специализированным, чем другой шаблон, если каждый потенциально подходящий для него список аргументов также подходит и для другого шаблона, но не наоборот. Рассмотрим следующие объявления шаблонов функций, взятые из примеров стандарта C++:

```
template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);
```

Для первого шаблона подходит любой тип. Второй шаблон более специализирован, чем первый, потому что для него подходят только типы-указатели. Иначе говоря, набор вызовов, подходящих для второго шаблона, составляет подмножество набора вызовов для первого шаблона. Аналогичные отношения существуют между вторым и третьим объявлениями: третий шаблон может вызываться только для указателей на const, тогда как второй шаблон допускает любые типы указателей. Следующая программа показывает, как работают эти правила:

```
//: C05:PartialOrder.cpp
// Упорядочение шаблонов функций
#include <iostream>
using namespace std;

template<class T> void f(T) {
  cout << "T" << endl;
}

template<class T> void f(T*) {
  cout << "T*" << endl;
}

template<class T> void f(const T*) {
  cout << "const T*" << endl;
}

int main() {
  f(0);           // T
  int i = 0;
```

```
  f(&i);          // T*
  const int j = 0;
  f(&j);          // const T*
} ///:~
```

Бесспорно, вызов f(&i) соответствует первому шаблону, но поскольку второй шаблон более специализирован, вызывается именно он. Вызов третьего шаблона невозможен, поскольку указатель не является указателем на const. Вызов f(&j) подходит для всех трех шаблонов (например, во втором шаблоне T будет соответствовать const int), но третий шаблон снова выбирается как наиболее специализированный.

Если в наборе перегруженных шаблонов функций не существует «самого специализированного» шаблона, возникает неоднозначность, и компилятор сообщает об ошибке. Из-за этого система приоритетов иногда называется «неполной» — она позволяет решить лишь часть потенциальных проблем. Аналогичные правила существуют для шаблонов классов (см. раздел «Неполная специализация и приоритеты шаблонов классов» в этой главе).

Специализация шаблонов

Термин «специализация» в C++ имеет конкретный смысл, связанный с шаблонами. Определение шаблона по своей природе является *обобщением*, поскольку оно описывает семейство функций или классов со сходными характеристиками. При получении аргументов шаблон специализируется, то есть определяется уникальный экземпляр из множества возможных экземпляров, входящих в это семейство функций или классов. Шаблон функции min(), приведенный в начале главы, представляет обобщенную функцию определения минимума, поскольку типы его параметров не заданы. Но стоит задать значения параметров шаблона явно или косвенно (с вычислением типов на основании имеющихся данных), и компилятор сгенерирует специализированную версию шаблона (например, min<int>).

Явная специализация

Вы можете самостоятельно задать код конкретной специализации шаблона, если возникнет такая необходимость. Обычно явная специализация требуется для шаблонов классов, но мы начнем описание синтаксиса с шаблона функции min().

Вспомните, что в приведенном примере MinTest.cpp присутствовала следующая обычная функция:

```
const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}
```

Она была нужна для того, чтобы вызов min() сравнивал строки, а не адреса. Хотя в данном случае это не дало бы никаких преимуществ, мы также могли бы определить отдельную специализацию min() для const char*, как сделано в следующей программе:

```
//: C05:MinTest2.cpp
#include <cstring>
#include <iostream>
using std::strcmp;
```

```
using std::cout;
using std::endl;

template<class T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
// Переопределение специализации шаблона min
template<>
const char* const& min<const char*>(const char* const& a,
                                    const char* const& b) {
  return (strcmp(a, b) < 0) ? a : b;
}

int main() {
  const char *s2 = "say \"Ni-!\"", *s1 = "knights who";
  cout << min(s1, s2) << endl;
  cout << min<>(s1, s2) << endl;
} ///:~
```

Префикс **template<>** сообщает компилятору, что далее следует специализация шаблона. Тип специализации должен быть указан в угловых скобках сразу же после имени функции, как при обычном вызове. Обратите внимание на то, как *тщательно* в явной специализации параметр T заменяется на const char*. Всюду, где в исходном шаблоне присутствует const T, const модифицирует *весь* тип T. В результате получается константный указатель const char*. Следовательно, в специализации вместо const T должна использоваться запись const char* const. Когда компилятор встречает вызов min() с аргументами const char*, он создает экземпляр нашей версии min() для const char*, чтобы она могла быть вызвана в программе. Для вызова min() в приведенном примере требуется та же специализация min().

Переопределение специализаций обычно приносит больше пользы в шаблонах классов, чем в шаблонах функций. Однако если вы переопределяете полную специализацию для шаблона класса, возможно, вам придется реализовать все функции класса. Дело в том, что вы фактически предоставляете отдельный класс, а клиентский код может рассчитывать на то, что весь интерфейс будет реализован в полном объеме.

В стандартную библиотеку входит переопределенная специализация шаблона vector для хранения объектов типа bool. Предполагается, что **vector<bool>** позволит реализациям библиотеки сэкономить память за счет упаковки битов в целых числах[1].

Как было показано ранее, объявление основного шаблона класса vector выглядит так:

```
template <class T, class Allocator = allocator<T> >
class vector {...};
```

Объявление переопределенной специализации для объектов типа bool могло бы выглядеть так:

```
template<> class vector<bool, allocator<bool> > {...}
```

Это объявление мгновенно распознается как явная специализация благодаря префиксу **template<>** и тому обстоятельству, что параметры основного шаблона соответствуют списку аргументов после имени класса.

---

[1] Контейнер vector<bool> подробно рассматривается в главе 7.

На самом деле контейнер vector<bool> более гибок, но об этом речь пойдет в следующем разделе.

Неполная специализация и приоритеты шаблонов классов

Шаблоны классов могут специализироваться *частично*. Это означает, что хотя бы один из параметров в специализации остается «открытым». Так, vector<bool> определяет тип объекта (bool), но оставляет неопределенным тип распределителя памяти. Рассмотрим фактическое объявление vector<bool>:

```
template<class Allocator> class vector<bool, Allocator>;
```

Внешними признаками неполных специализаций являются непустой список параметров в фигурных скобках после ключевого слова template (оставшиеся параметры) и после класса (заданные аргументы). Такое определение vector<bool> позволяет выбрать нестандартный тип распределителя памяти даже при фиксированном типе элементов bool. Другими словами, специализация вообще и неполная специализация в особенности образуют некую разновидность «перегрузки» шаблонов классов.

Правила выбора шаблона для специализации напоминают неполные приоритеты шаблонов функций — выбирается «наиболее специализированный» шаблон. Строки внутри функций f() в следующем примере поясняют роль каждого определения шаблона:

```
//: C05:PartialOrder2.cpp
// Неполные приоритеты при выборе шаблона класса
#include <iostream>
using namespace std;

template<class T, class U> class C {
public:
  void f() { cout << "Primary Template\n"; }
};

template<class U> class C<int, U> {
public:
  void f() { cout << "T == int\n"; }
};

template<class T> class C<T, double> {
public:
  void f() { cout << "U == double\n"; }
};

template<class T, class U> class C<T*, U> {
public:
  void f() { cout << "T* used \n"; }
};

template<class T, class U> class C<T, U*> {
public:
  void f() { cout << "U* used\n"; }
};

template<class T, class U> class C<T*, U*> {
```

```
public:
  void f() { cout << "T* and U* used\n"; }
};

template<class T> class C<T, T> {
public:
  void f() { cout << "T == U\n"; }
};

int main() {
  C<float, int>().f();    // 1: Основной шаблон
  C<int, float>().f();    // 2: T == int
  C<float, double>().f(); // 3: U == double
  C<float, float>().f();  // 4: T == U
  C<float*, float>().f(); // 5: Используется T* [float]
  C<float, float*>().f(); // 6: Используется U* [float]
  C<float*, int*>().f();  // 7: Используются T* и U* [float,int]

  // Следующие вызовы неоднозначны:
//   8: C<int, int>().f();
//   9: C<double, double>().f();
//  10: C<float*, float*>().f();
//  11: C<int, int*>().f();
//  12: C<int*, int*>().f();
} ///:~
```

Как видите, частичная специализация также может производиться по тому, являются ли параметры шаблона указателями, и по их совпадению. При использовании специализации `T*`, как в строке 5, `T` представляет не передаваемый тип указателя верхнего уровня, а тот тип, на который этот указатель ссылается (в данном случае `float`). Спецификация `T*` может рассматриваться как условное обозначение для поиска типов указателей. Если передать в первом аргументе шаблона `int**`, то `T` будет соответствовать `int*`. Строка 8 неоднозначна, потому что наличие первого параметра типа `int` и совпадение двух типов параметров не зависят друг от друга — один аспект не обеспечивает большей специализации по сравнению с другим. Аналогичная логика применима к строкам 9–12.

Пример

На базе шаблона класса легко можно определять производные шаблоны классов. Допустим, если шаблон `vector` делает большую часть того, что вам нужно, но в вашей конкретной ситуации нужна поддержка автоматической сортировки, вы можете легко воспользоваться готовым кодом `vector`. В следующем примере создается шаблон, производный от `vector<T>`, и в него добавляется поддержка сортировки. Учтите, что создание классов, производных от класса `vector`, который не имеет виртуального деструктора, может оказаться рискованным, если в деструкторе должна выполняться зачистка.

```
//: C05:Sortable.h
// Специализация шаблона
#ifndef SORTABLE_H
#define SORTABLE_H
#include <cstring>
#include <cstddef>
#include <string>
```

```
#include <vector>
using std::size_t;

template<class T>
class Sortable : public std::vector<T> {
public:
  void sort();
};

template<class T>
void Sortable<T>::sort() { // Простая сортировка
  for(size_t i = this->size(); i > 0; i--)
    for(size_t j = 1; j < i; ++j)
      if(this->at(j-1) > this->at(j)) {
        T t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}

// Неполная специализация для указателей:
template<class T>
class Sortable<T*> : public std::vector<T*> {
public:
  void sort();
};

template<class T>
void Sortable<T*>::sort() {
  for(size_t i = this->size(); i > 0; i--)
    for(size_t j = 1; j < i; ++j)
      if(*this->at(j-1) > *this->at(j)) {
        T* t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}

// Полная специализация для char*
// (Объявлена подставляемой для удобства -
// обычно тело функции находится в отдельном файле,
//  а в заголовке остается только объявление)
template<>
inline void Sortable<char*>::sort() {
  for(size_t i = size(); i > 0; i--)
    for(size_t j = 1; j < i; ++j)
      if(std::strcmp(this->at(j-1), this->at(j)) > 0) {
        char* t = this->at(j-1);
        this->at(j-1) = this->at(j);
        this->at(j) = t;
      }
}
#endif // SORTABLE_H ///:~
```

Шаблон Sortable устанавливает ограничение для всех классов, для которых он специализируется (кроме одного): класс должен содержать оператор >. Полная специализация сортирует векторы типа char*, сравнивая строки C с нуль-завершителями при помощи функции strcmp(). Наличие уточнения this-> в данном слу-

чае обязательно[1] (этот вопрос рассматривается в разделе «Разрешение имен» этой главы, а также в объяснении к примеру PriorityQueue6.cpp в главе 7).

Здесь приведена тестовая программа для файла Sortable.h, в которой используется генератор случайных чисел Urand (см. выше):

```
//: C05:Sortable.cpp
//{bor} (из-за присутствия bitset в Urand.h)
// Тестирование специализированных шаблонов
#include <cstddef>
#include <iostream>
#include "Sortable.h"
#include "Urand.h"
using namespace std;

#define asz(a) (sizeof a / sizeof a[0])

char* words[] = { "is", "running", "big", "dog", "a", };
char* words2[] = { "this", "that", "theother", };

int main() {
  Sortable<int> is;
  Urand<47> rand;
  for(size_t i = 0; i < 15; ++i)
    is.push_back(rnd());
  for(size_t i = 0; i < is.size(); ++i)
    cout << is[i] << ' ';
  cout << endl;
  is.sort();
  for(size_t i = 0; i < is.size(); ++i)
    cout << is[i] << ' ';
  cout << endl;

  // Использование частичной специализации шаблона:
  Sortable<string*> ss;
  for(size_t i = 0; i < asz(words); ++i)
    ss.push_back(new string(words[i]));
  for(size_t i = 0; i < ss.size(); ++i)
    cout << *ss[i] << ' ';
  cout << endl;
  ss.sort();
  for(size_t i = 0; i < ss.size(); i++) {
    cout << *ss[i] << ' ';
    delete ss[i];
  }
  cout << endl;

  // Использование полной специализации char*:
  Sortable<char*> scp;
  for(size_t i = 0; i < asz(words2); ++i)
    scp.push_back(words2[i]);
  for(size_t i = 0; i < scp.size(); ++i)
    cout << scp[i] << ' ';
  cout << endl;
  scp.sort();
  for(size_t i = 0; i < scp.size(); ++i)
```

[1] Вместо this-> может использоваться любое допустимое уточнение, например, Sortable::at() или vector<T>::at(). Важно наличие уточнения, а не его конкретный вид.

```
    cout << scp[i] << ' ';
  cout << endl;
} ///:~
```

Во всех специализациях задействованы разные версии шаблона: Sortable<int> использует основной шаблон, а Sortable<string*> — неполную специализацию для указателей, Sortable<char*> — полную специализацию для char*. Без этой полной специализации у вас могло бы сложиться обманчивое впечатление, будто все работает правильно — массив words правильно сортируется в строку «a big dog is running», поскольку частичная специализация ограничивается сравнением первых символов массивов. Однако на примере массива words2 такая сортировка уже не работает.

Ограничение объема генерируемого кода

Каждый раз, когда в программе определяется специализация шаблона класса, компилятор генерирует код из определения класса для конкретной специализации вместе со всеми функциями, вызываемыми в программе. При этом генерируются только те функции класса, которые вызываются в программе. И это вполне логично, как показывает следующий пример:

```
//: C05:DelayedInstantiation.cpp
// Генерируется только код используемых функций шаблонов классов.

class X {
public:
  void f() {}
};

class Y {
public:
  void g() {}
};

template <typename T> class Z {
  T t;
public:
  void a() { t.f(); }
  void b() { t.g(); }
};

int main() {
  Z<X> zx;
  zx.a(); // Код Z<X>::b() не генерируется
  Z<Y> zy;
  zy.b(); // Код Z<Y>::a() не генерируется
} ///:~
```

Хотя шаблон Z вроде бы использует обе функции шаблона T (f() и g()), программа успешно компилируется, а это означает, что код Z<X>::a() генерируется только при явном вызове этой функции для zx (если бы при этом также генерировался код Z<X>::b(), то компилятор выдал бы сообщение об ошибке вызова несуществующей функции X::g()). Аналогично, вызов zy.b() не генерирует Z<Y>::a(). Следовательно, шаблон Z может использоваться для обоих классов (X и Y). Если бы все функции класса генерировались непосредственно при специализации, это бы существенно ограничило возможности применения многих шаблонов.

Допустим, у нас имеется шаблонный контейнер Stack, и в программе используются специализации int, int* и char*. В результате три версии кода Stack будут сгенерированы компилятором и скомпонованы с вашей программой. Как отмечалось ранее, шаблоны нужны прежде всего для того, чтобы избежать ручного дублирования кода; однако код все равно дублируется, просто компилятор генерирует его за вас. Основную часть реализации для указателей можно выделить в единый класс, используя комбинацию полной и частичной специализации. Идея заключается в том, чтобы определить полную специализацию для void*, а затем создать неполные специализации для остальных типов; это позволит применять общий код для разных типов. Следующий пример демонстрирует эту методику:

```
//: C05:Nobloat.h
// Совместное использование кода при хранении указателей в стеке
#ifndef NOBLOAT_H
#define NOBLOAT_H
#include <cassert>
#include <cstddef>
#include <cstring>

// Основной шаблон
template<class T>
class Stack {
  T* data;
  std::size_t count;
  std::size_t capacity;
  enum {INIT = 5};
public:
  Stack() {
    count = 0;
    capacity = INIT;
    data = new T[INIT];
  }
  void push(const T& t) {
    if (count == capacity) {
      // Выделение дополнительной памяти
      std::size_t newCapacity = 2*capacity;
      T* newData = new T[newCapacity];
      for (size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete [] data;
      data = newData;
      capacity = newCapacity;
    }
    assert(count < capacity);
    data[count++] = t;
  }
  void pop() {
    assert(count > 0);
    --count;
  }
  T top() const {
    assert(count > 0);
    return data[count-1];
  }
  std::size_t size() const { return count; }
};
```

```
// Полная специализация для void*
template<>
class Stack<void *> {
  void** data;
  std::size_t count;
  std::size_t capacity;
  enum {INIT = 5};
public:
  Stack() {
    count = 0;
    capacity = INIT;
    data = new void*[INIT];
  }
  void push(void* const & t) {
    if (count == capacity) {
      std::size_t newCapacity = 2*capacity;
      void** newData = new void*[newCapacity];
      std::memcpy(newData, data, count*sizeof(void*));
      delete [] data;
      data = newData;
      capacity = newCapacity;
    }
    assert(count < capacity);
    data[count++] = t;
  }
  void pop() {
    assert(count > 0);
    --count;
  }
  void* top() const {
    assert(count > 0);
    return data[count-1];
  }
  std::size_t size() const {return count;}
};

// Неполная специализация для других указателей
template<class T>
class Stack<T*> : private Stack<void *> {
  typedef Stack<void *> Base;
public:
  void push(T* const & t) {Base::push(t);}
  void pop() {Base::pop();}
  T* top() const {return static_cast<T*>(Base::top());}
  std::size_t size() {return Base::size();}
};
#endif // NOBLOAT_H ///:~
```

Этот простой стек автоматически расширяется по мере заполнения. Специализация для **void*** является полной, на что указывает префикс **template<>** (с пустым списком параметров шаблона). Как уже упоминалось, в специализации шаблона класса должны быть реализованы все функции класса. Экономия достигается для всех остальных указателей. Неполная специализация для прочих типов указателей создается закрытым наследованием от **Stack<void*>**, поскольку мы всего лишь задействуем **Stack<void*>** в реализации и не желаем предоставлять пользователю прямой доступ к его интерфейсу. Функции класса в каждой специализации ограничиваются простым перенаправлением вызова к соответствующим функциям

Stack<void*>. Таким образом, при каждой специализации шаблона для типа указателя, отличного от void*, размер сгенерированного кода составит лишь ничтожную часть того размера, который бы потребовался при использовании основного шаблона. Ниже приводится тестовая программа:

```
//: C05:NobloatTest.cpp
#include <iostream>
#include <string>
#include "Nobloat.h"
using namespace std;

template<class StackType>
void emptyTheStack(StackType& stk) {
  while (stk.size() > 0) {
    cout << stk.top() << endl;
    stk.pop();
  }
}
// Перегрузка для emptyTheStack (не специализация!)
template<class T>
void emptyTheStack(Stack<T*>& stk) {
  while (stk.size() > 0) {
    cout << *stk.top() << endl;
    stk.pop();
  }
}

int main() {
  Stack<int> s1;
  s1.push(1);
  s1.push(2);
  emptyTheStack(s1);
  Stack<int *> s2;
  int i = 3;
  int j = 4;
  s2.push(&i);
  s2.push(&j);
  emptyTheStack(s2);
} ///:~
```

Для удобства в программу включены два шаблона функции emptyTheStack. Поскольку шаблоны функций не поддерживают неполной специализации, мы предоставляем перегруженные шаблоны. Вторая версия emptyTheStack более специализирована по сравнению с первой, поэтому она выбирается при каждом использовании типов указателей. В программе создаются три специализации шаблона класса: Stack<int>, Stack<void*> и Stack<int*>. Специализация Stack<void*> создается косвенно, поскольку Stack<int*> является производной от нее. Программа, в которой задействованы специализации для разных типов указателей, обеспечивает существенную экономию по сравнению с простым использованием одного шаблона Stack.

Разрешение имен

Встречая в программе идентификатор, компилятор должен сначала определить тип и область видимости (а в случае переменных — срок жизни) для сущности, пред-

ставляемой данным идентификатором. Использование шаблонов усложняет ситуацию. Когда компилятор впервые встречает шаблон, он еще не располагает полной информацией о нем, а значит, не может определить, правильно ли он используется. По этой причине компиляция шаблонов выполняется в два этапа.

Имена в шаблонах

В первой фазе компилятор разбирает определение шаблона, ищет очевидные синтаксические ошибки и разрешает все имена, которые может разрешить на этой стадии. К этой категории относятся имена, не зависящие от параметров шаблона и разрешаемые стандартным способом или, при необходимости, с помощью аргументов (см. далее). Имена, которые компилятор разрешить не может, называются *зависимыми именами*, поскольку они так или иначе зависят от параметров шаблона. Разрешение этих имен невозможно до специализации шаблона с фактически переданными аргументами. Следовательно, вторая фаза компиляции шаблона наступает при специализации. Здесь компилятор определяет, нужно ли использовать переопределенную специализацию вместо основного шаблона.

Но прежде чем переходить к примерам, нужно разобраться еще с двумя терминами. *Уточненными именами* называются имена, построенные по следующим схемам:

- *класс*::*имя*;
- *объект*.*имя*;
- *указатель_на_объект*->*имя*.

Примеры уточненных имен:

```
MyClass::f();
x.f();
p->f();
```

Уточненные имена неоднократно встречались на страницах книги, причем в последний раз при описании ключевого слова typename. Они называются уточненными, потому что целевое имя (как f) явно связывается с конкретным классом или пространством имен. По этой информации компилятор узнает, где искать объявления этих имен.

Другой важный термин — *поиск с учетом аргументов* (Argument-Dependent Lookup, ADL)[1]. Этот механизм изначально разрабатывался для упрощения вызова внешних функций (в том числе операторов), объявленных в пространствах имен. Рассмотрим пример:

```
#include <iostream>
#include <string>
//...
  std::string s("hello");
  std::cout << s << std::endl;
```

Как обычно в заголовочных файлах, в этом фрагменте отсутствует директива using namespace std. Без этой директивы все имена в пространстве имен std должны

[1] Так называемое *правило Кёнига*, по имени Эндрю Кёнига (Andrew Koenig), который первым предложил эту методику комитету по стандартизации C++. ADL применяется как при наличии шаблонов, так и без них.

объявляться с префиксом std::. Однако некоторые элементы из пространства имен std остались неуточненными. Видите, какие именно?

Не указано, какие операторные функции должны использоваться в программе. Мы хотим, чтобы происходило следующее:

```
std::operator<<(std::operator<<(std::cout,s),std::endl);
```

Но вводить эту конструкцию совершенно не хочется!

Исходная команда вывода работает именно так, как мы хотим, благодаря механизму ADL. Если в программе присутствует неуточненный вызов функции, объявление которой отсутствует в обычной области видимости, то поиск подходящих объявлений функций производится в пространствах имен каждого из ее аргументов. В исходной команде первой вызывается функция

```
operator<<(std::cout, s);
```

Поскольку такая функция отсутствует в области видимости исходного фрагмента, компилятор замечает, что первый аргумент функции (std::cout) принадлежит пространству имен std; он включает это пространство имен в список поиска уникальных функций, лучше всего соответствующих сигнатуре operator<<(std::ostream& std::string). Такое объявление находится в пространстве имен std через заголовок <string>.

Без ADL работать с пространствами имен было бы крайне неудобно. Обратите внимание: механизм ADL обычно анализирует *все* объявления с указанным именем из всех доступных пространств имен. При отсутствии единого оптимального совпадения возникает неоднозначность.

Чтобы запретить применение ADL, заключите имя функции в круглые скобки:

```
(f)(x, y);      // Подавление механизма ADL
```

Теперь рассмотрим следующую программу[1]:

```
//: C05:Lookup.cpp
// Правильно работает только в EDG и Metrowerks (со специальным ключом)
#include <iostream>
using std::cout;
using std::endl;

void f(double) { cout << "f(double)" << endl; }

template<class T> class X {
public:
  void g() { f(1); }
};

void f(int) { cout << "f(int)" << endl; }

int main() {
  X<int>().g();
} ///:~
```

Из всех имеющихся у нас компиляторов программа работала правильно только в интерфейсе Edison Design Group[2], хотя в некоторых компиляторах (например, Metrowerks) правильное поведение при поиске активизируется специальным

[1] Из презентации Херба Саттера (Herb Sutter).

[2] Этот интерфейс используется некоторыми компиляторами, в том числе Comeau C++.

ключом. Поскольку f является независимым именем, которое может быть разрешено на ранней стадии по контексту определения шаблона, когда в области видимости находится только f(double), результат должен выглядеть так:

```
f(double)
```

К сожалению, многие существующие программы зависят от этого нестандартного поведения с привязкой вызова f(1) из g() к более позднему определению f(int), поэтому производители компиляторов не хотят вносить изменения.

Рассмотрим более подробный пример[1]:

```
//: C05:Lookup2.cpp {-bor}{-g++}{-dmc}
// Microsoft: необходим ключ -Za (режим ANSI)
#include <algorithm>
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;

void g() { cout << "global g()" << endl; }

template <class T> class Y {
public:
  void g()
    { cout << "Y<" << typeid(T).name() << ">::g()" << endl; }
  void h() {
    cout << "Y<" << typeid(T).name() << ">::h()" << endl;
  }
  typedef int E;
};

typedef double E;

template<class T> void swap(T& t1, T& t2) {
  cout << "global swap" << endl;
  T temp = t1;
  t1 = t2;
  t2 = temp;
}

template<class T> class X : public Y<T> {
public:
  E f() {
    g();
    this->h();
    T t1 = T(), t2 = T(1);
    cout << t1 << endl;
    swap(t1, t2);
    std::swap(t1, t2);
    cout << typeid(E).name() << endl;
    return E(t2);
  }
};

int main() {
  X<int> x;
  cout << x.f() << endl;
} ///:~
```

[1] Также основанный на одном из примеров Херба Саттера.

Результат выполнения программы будет выглядеть так:

```
global g()
Y<int>::h()
0
global swap
double
1
```

Теперь посмотрим на объявления в X::f().

- E, тип возвращаемого значения X::f(), не является зависимым именем, поэтому его поиск производится при обработке шаблона. В результате обнаруживается определение типа, в соответствии с которым E определяется как double. На первый взгляд это выглядит странно, поскольку в нешаблонных классах сначала будет найдено определение E в базовом классе, но таковы правила (базовый класс Y является *зависимым базовым классом*, поэтому в момент определения шаблона поиск в нем производиться не может).
- Вызов g() тоже независим, поскольку он не содержит упоминаний T. Если бы функция g() имела параметры, относящиеся к типу класса, определенного в другом пространстве имен, то механизм ADL начал бы работать, так как в области видимости нет определения g() с параметрами. В итоге для вызова устанавливается соответствие с глобальным объявлением g().
- Вызов this->h() является уточненным; уточнение (this) соответствует текущему объекту, относящемуся к типу X, который, в свою очередь, зависит от имени Y<T> из-за наследования. В X не существует функции h(), поэтому поиск выполняется в области видимости базового класса X, то есть Y<T>. Данное имя является зависимым и потому рассматривается на стадии специализации при наличии надежной информации о Y<T> (включая все потенциальные специализации, которые могли быть написаны после определения X), поэтому в конечном счете вызывается Y<int>::h().
- Объявления t1 и t2 являются зависимыми.
- Вызов operator<<(cout,t1) зависим, потому что t1 относится к типу T. Поиск выполняется позднее, когда T соответствует int, а оператор << для int находится в std.
- Неуточненный вызов swap() зависим, потому что его аргументы относятся к типу T. В конечном счете это приводит к специализации глобальной функции swap(int&,int&).
- Уточненный вызов std::swap() *не является* зависимым из-за присутствия фиксированного пространства имен std. Компилятор знает, что нужное объявление нужно искать в std (чтобы уточненное имя считалось зависимым, в квалификаторе слева от парного двоеточия должен упоминаться параметр шаблона). Позднее шаблон функции std::swap() генерирует std::swap(int&,int&) на стадии специализации. Других зависимых имен в X<T>::f() нет.

Подведем итог: поиск зависимых имен осуществляется в момент специализации; исключение из этого правила составляют неуточненные зависимые имена, поиск которых начинается с определения шаблона. Поиск независимых имен в шаблонах производится рано, при обработке определения шаблона (при необхо-

димости следующий поиск выполняется в момент специализации, когда известны фактические типы аргументов).

Если вы действительно внимательно изучили этот пример (и даже поняли его смысл), в следующем разделе вас поджидает очередной сюрприз.

Шаблоны и дружественные функции

Объявление дружественной функции в классе открывает внешний доступ к закрытым и защищенным членам класса. Если имя дружественной функции уточнено, оно ищется в квалификаторе (пространстве имен или классе). Но если уточнение отсутствует, компилятор должен сделать предположение по поводу того, где находится определение дружественной функции, так как все идентификаторы должны быть однозначно причислены к некоторой области видимости. Компилятор предполагает, что функция будет определяться в ближайшем вмещающем пространстве имен (не классе!), содержащем класс, который предоставляет дружественный доступ. Следующий пример поясняет эту замысловатую формулировку:

```
//: C05:FriendScope.cpp
#include <iostream>
using namespace std;

class Friendly {
  int i;
public:
  Friendly(int theInt) { i = theInt; }
  friend void f(const Friendly&); // Необходимо глобальное определение
  void g() { f(*this); }
};

void h() {
  f(Friendly(1));  // Использует ADL
}

void f(const Friendly& fo) {  // Определение дружественной функции
  cout << fo.i << endl;
}

int main() {
  h();              // Выводит 1
  Friendly(2).g();  // Выводит 2
} ///:~
```

Объявление f() в классе Friendly не уточняется, поэтому компилятор предполагает, что когда-нибудь ему удастся связать это объявление с определением на уровне файла (область видимости пространства имен, содержащего Friendly в данном примере). Определение следует после определения функции h(). С другой стороны, с вызовом f() внутри h() дело обстоит совершенно иначе — он разрешается с использованием ADL. Поскольку аргументом f() внутри h() является объект Friendly, поиск объявления f() производится в классе Friendly и завершается успехом. Но если бы вместо этого вызов имел вид f(1) (что в общем-то логично, поскольку значение 1 может быть косвенно преобразовано во Friendly(1)), компилятор не узнал бы, где ему искать объявление f(). В этом случае внешний интерфейс EDG справедливо жалуется на то, что переменная f не определена.

Теперь предположим, что и Friendly, и f являются шаблонами, как в следующей программе:

```
//: C05:FriendScope2.cpp
#include <iostream>
using namespace std;

// Необходимые опережающие объявления
template<class T> class Friendly;
template<class T> void f(const Friendly<T>&);

template<class T> class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f<>(const Friendly<T>&);
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}

template<class T>
void f(const Friendly<T>& fo) {
  cout << fo.t << endl;
}

int main() {
  h();
  Friendly<int>(2).g();
} ///:~
```

Для начала обратите внимание на угловые скобки в объявлении f внутри Friendly. Они сообщают компилятору, что f является шаблоном. В противном случае компилятор искал бы обычную функцию с именем f и не нашел ее. Мы также могли бы указать параметр шаблона в угловых скобках (<T>), но он легко вычисляется по объявлению.

Опережающее объявление шаблона функции f перед определением класса необходимо, хотя оно и отсутствовало в предыдущем примере, где функция f не была шаблоном; дружественные шаблоны функций должны предварительно объявляться в соответствии со спецификацией языка. Для правильного объявления f необходимо объявление Friendly, поскольку f получает аргумент типа Friendly; этим объясняется опережающее объявление Friendly в начале. Вообще говоря, полное определение f можно было разместить сразу же после начального объявления Friendly, не отделяя определение от объявления, но мы решили оставить его в форме, больше соответствующей предыдущему примеру.

Остался еще один последний вариант использования дружественных функций в шаблонах: их полное определение внутри определения шаблона класса. Вот как будет выглядеть предыдущий пример в этом варианте:

```
//: C05:FriendScope3.cpp {-bor}
// Microsoft: необходим ключ -Za (режим ANSI)
#include <iostream>
using namespace std;
```

```
template<class T>
class Friendly {
  T t;
public:
  Friendly(const T& theT) : t(theT) {}
  friend void f(const Friendly<T>& fo) {
    cout << fo.t << endl;
}
  void g() { f(*this); }
};

void h() {
  f(Friendly<int>(1));
}

int main() {
  h();
  Friendly<int>(2).g();
} ///:~
```

Между этими двумя примерами существует важное различие: здесь f является не шаблоном, а обычной функцией (вспомните, что раньше угловые скобки сообщали компилятору, что f() является шаблоном). При каждой специализации шаблона класса Friendly создается новая обычная перегруженная функция, которая получает аргумент с типом текущей специализации Friendly. Это самый удобный способ определения дружественных функций для шаблонов.

Чтобы стало понятнее, давайте предположим, что мы хотим включить в шаблон класса внешние дружественные операторы. Следующий шаблон класса просто хранит обобщенное значение:

```
template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
};
```

Не разобравшись в сути предыдущих примеров этого раздела, новички часто удивляются, почему им не удается заставить работать простейший оператор <<. Если операторы не определяются внутри определения Box, для них необходимо предоставить опережающие объявления, как показано выше:

```
//: C05:Box1.cpp
// Определение операторов для шаблонов
#include <iostream>
using namespace std;

// Опережающие объявления
template<class T> class Box;

template<class T>
Box<T> operator+(const Box<T>&, const Box<T>&);

template<class T>
ostream& operator<<(ostream&, const Box<T>&);

template<class T> class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
```

```
  friend Box operator+<>(const Box<T>&, const Box<T>&);
  friend ostream& operator<< <>(ostream&, const Box<T>&);
};

template<class T>
Box<T> operator+(const Box<T>& b1, const Box<T>& b2) {
  return Box<T>(b1.t + b2.t);
}

template<class T>
ostream& operator<<(ostream& os, const Box<T>& b) {
  return os << '[' << b.t << ']';
}

int main() {
  Box<int> b1(1), b2(2);
  cout << b1 + b2 << endl;  // [3]
//  cout << b1 + 2 << endl; // Автоматические преобразования отсутствуют!
} ///:~
```

В этом примере определяются внешние операторы сложения и вывода в поток. В функции main() проявляется главный недостаток такого подхода: автоматические преобразования (как в выражении b1+2) становятся невозможными, потому что они не поддерживаются шаблонами. При внутреннем («нешаблонном») определении программа получается короче и надежнее:

```
//: C05:Box2.cpp
// Определение нешаблонных операторов
#include <iostream>
using namespace std;

template<class T>
class Box {
  T t;
public:
  Box(const T& theT) : t(theT) {}
  friend Box operator+(const Box<T>& b1, const Box<T>& b2) {
    return Box<T>(b1.t + b2.t);
  }
  friend ostream& operator<<(ostream& os, const Box<T>& b) {
    return os << '[' << b.t << ']';
  }
};

int main() {
  Box<int> b1(1), b2(2);
  cout << b1 + b2 << endl;  // [3]
  cout << b1 + 2 << endl;   // [3]
} ///:~
```

Поскольку операторы являются обычными функциями (перегружаемыми для каждой специализации Box — в данном случае для int), неявные преобразования действуют нормально, и выражение b1+2 допустимо.

Учтите, что один тип не может быть объявленным дружественным по отношению к Box и вообще какому-либо шаблону класса. Речь идет о типе T, а вернее, о том типе, по которому параметризован шаблон класса. Насколько нам известно, не существует сколько-нибудь обоснованных причин для такого запрета, но пока объявление friend class T считается недопустимым и вызывает ошибку компиляции.

Вы можете точно указать, какие специализации шаблона являются друзьями класса. В примерах предыдущего раздела дружественной была только специализация шаблона функции f с тем же типом, по которому специализировался шаблон Friendly. Например, дружественной для класса Friendly<int> была только специализация f<int>(const Friendly<int>&). Поэтому параметр шаблона Friendly использовался для специализации f в объявлении friend. При желании мы могли бы сделать конкретную фиксированную специализацию f дружественной для всех экземпляров Friendly:

```
// Внутри Friendly:
friend void f<>(const Friendly<double>&);
```

После замены T на double специализация f для double получает доступ ко всем закрытым и защищенным членам любой специализации Friendly. Как и прежде, специализация f<double>() создается лишь при явном вызове.

Аналогично, если объявить нешаблонную функцию без параметров, зависящих от T, эта функция становится дружественной для всех специализаций Friendly:

```
// Внутри Friendly:
friend vote g(int);  // g(int) является другом
                     // всех специализаций Friendly
```

Как всегда, неуточненная функция g(int) должна определяться на уровне файла (в пространстве имен, содержащем Friendly).

Также можно сделать все специализации f дружественными для всех специализаций Friendly при помощи так называемых *дружественных шаблонов*:

```
template<class T> class Friendly {
  template<class U> friend void f<>(const Friendly<U>&);
```

Поскольку аргумент в объявлении дружественного шаблона не зависит от T, в этих дружественных отношениях допускаются любые комбинации T и U. Дружественные шаблоны, как и вложенные, могут использоваться внутри нешаблонных классов.

Идиомы программирования с применением шаблонов

Язык является орудием мышления, поэтому новые языковые возможности обычно порождают новые приемы. В этом разделе мы рассмотрим ряд распространенных идиом, появившихся с момента включения шаблонов в язык C++.

Характеристики

Шаблоны характеристик, концепция которых была впервые предложена Натаном Майерсом (Nathan Myers), предназначаются для группировки объявлений, зависящих от типа. В сущности, характеристики позволяют «смешивать» некоторые типы и значения с контекстами, в которых они используются, без ущерба для удобочитаемости и простоты сопровождения программы.

Простейшим примером шаблона характеристик является класс numeric_limits, определяемый в файле <limits>. Определение основного шаблона выглядит так:

```
template<class T> class numeric_limits {
public:
  static const bool is_specialized = false;
  static T min() throw();
```

```
  static T max() throw();
  static const int digits = 0;
  static const int digits10 = 0;
  static const bool is_signed = false;
  static const bool is_integer = false;
  static const bool is_exact = false;
  static const int radix = 0;
  static T epsilon() throw();
  static T round_error() throw();
  static const int min_exponent = 0;
  static const int min_exponent10 = 0;
  static const int max_exponent = 0;
  static const int max_exponent10 = 0;
  static const bool has_infinity = false;
  static const bool has_quiet_NaN = false;
  static const bool has_signaling_NaN = false;
  static const float_denorm_style has_denorm = denorm_absent;
  static const bool has_denorm_loss = false;
  static T infinity() throw();
  static T quiet_NaN() throw();
  static T signaling_NaN() throw();
  static T denorm_min() throw();
  static const bool is_iec559 = false;
  static const bool is_bounded = false;
  static const bool is_modulo = false;
  static const bool traps = false;
  static const bool tinyness_before = false;
  static const float_round_style round_style = round_toward_zero;
};
```

В заголовке <limits> определяются специализации для всех основных числовых типов (переменная is_specialized равна true). Например, основание системы счисления для экспоненты в вещественном типе double может быть получено при помощи выражения numeric_limits<double>::radix. Наименьшее доступное целое число определяется выражением numeric_limits<int>::min(). Не все члены numeric_limits относятся ко всем основным типам (например, функция epsilon() имеет смысл только для вещественных типов).

Значения, которые всегда являются целыми, определяются в виде статических переменных numeric_limits. Те, которые могут оказаться нецелыми (например, минимальное значение float), реализуются в виде статических подставляемых функций. Такое различие объясняется тем, что C++ позволяет инициализировать в определении класса только *целочисленные* статические переменные.

В главе 3 было показано, как при помощи классов характеристик управлять средствами обработки символьных данных в строковых классах. Классы std::string и std::wstring являются специализациями шаблона std::basic_string, который определяется следующим образом:

```
template<class charT,
  class traits = char_traits<charT>,
  class allocator = allocator<charT> >
  class basic_string;
```

Параметр шаблона charT представляет базовый тип символов, обычно это char или wchar_t. Основной шаблон char_traits обычно остается пустым, а специализации для char и wchar_t предоставляются стандартной библиотекой. Далее приведена спецификация специализации char_traits<char> в соответствии со стандартом C++:

```
template<> struct char_traits<char> {
  typedef char char_type;
  typedef int int_type;
  typedef streamoff off_type;
  typedef streampos pos_type;
  typedef mbstate_t state_type;
  static void assign(char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1, const char_type& c2);
  static bool lt(const char_type& c1, const char_type& c2);
  static int compare(const char_type* s1,
                     const char_type* s2, size_t n);
  static size_t length(const char_type* s);
  static const char_type* find(const char_type* s, size_t n,
                               const char_type& a);
  static char_type* move(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* copy(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n,
                           char_type a);
  static int_type not_eof(const int_type& c);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1,
                          const int_type& c2);
  static int_type eof();
};
```

Эти функции используются шаблоном класса **basic_string** для выполнения операций с символами, общих для любых строк. Рассмотрим такое объявление переменной **string**:

```
std::string s;
```

На самом деле **s** объявляется следующим образом (благодаря аргументам по умолчанию в спецификации **basic_string**):

```
std::basic_string<char, std::char_traits<char>,
                  std::allocator<char> > s;
```

Отделение характеристик символов от шаблона **basic_string** позволяет предоставить нестандартный класс характеристик для замены **std::char_traits**. Следующий пример демонстрирует замечательную гибкость такого решения:

```
//: C05:BearCorner.h
#ifndef BEARCORNER_H
#define BEARCORNER_H
#include <iostream>
using std::ostream;

// Классы угощения (характеристики гостей):
class Milk {
public:
  friend ostream& operator<<(ostream& os, const Milk&) {
    return os << "Milk";
  }
};
class CondensedMilk {
public:
  friend ostream&
  operator<<(ostream& os, const CondensedMilk&) {
```

```
    return os << "CondensedMilk";
  }
};

class Honey {
public:
  friend ostream& operator<<(ostream& os, const Honey&) {
    return os << "Honey";
  }
};

class Cookies {
public:
  friend ostream& operator<<(ostream& os, const Cookies&) {
    return os << "Cookies";
  }
};

// Классы гостей:
class Bear {
public:
  friend ostream& operator<<(ostream& os, const Bear&) {
    return os << "Theodore";
  }
};

class Boy {
public:
  friend ostream& operator<<(ostream& os, const Boy&) {
    return os << "Patrick";
  }
};

// Основной шаблон характеристик (пустой - может использоваться
// для хранения базовых типов)
template<class Guest>
class GuestTraits;

// Специализации характеристик для типов гостей
template<> class GuestTraits<Bear> {
public:
  typedef CondensedMilk beverage_type;
  typedef Honey snack_type;
};

template<>
class GuestTraits<Boy> {
public:
  typedef Milk beverage_type;
  typedef Cookies snack_type;
};
#endif // BEARCORNER_H

//: C05:BearCorner.h
// Использование классов характеристик
#include <iostream>
#include "BearCorner.h"
using namespace std;

// Нестандартный класс характеристик
```

```
class MixedUpTraits {
public:
  typedef Milk beverage_type;
  typedef Honey snack_type;
};

// Шаблон гостя (использует класс характеристик)
template< class Guest, class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g), {}
  void entertain() {
    cout << "Entertaining " << theGuest
         << " serving " << bev
         << " and " << snack << endl;
  }
};

int main() {
  Boy cr;
  BearCorner<Boy> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear> pc2(pb);
  pc2.entertain();
  BearCorner<Bear, MixedUpTraits> pc3(pb);
  pc3.entertain();
} ///:~
```

В этой программе экземплярам классов «гостей» **Boy** и **Bear** назначаются классы «угощения» по вкусу. Для класса **Boy** это классы **Milk** и **Cookies**, и для класса **Bear** — **CondensedMilk** и **Honey**. «Гости» связываются с «угощением» через специализации основного (пустого) шаблона класса характеристик. Аргументы **BearCorner** по умолчанию обеспечивают стандартное «меню», однако такое поведение можно переопределить, как было сделано ранее с классом **MixedUpTraits**. Результат выполнения программы выглядит так:

```
Entertaining Patrick serving Milk and Cookies
Entertaining Theodore serving Condensed Milk and Cookies
Entertaining Theodore serving Milk and Honey
```

Характеристики обладают двумя важными преимуществами. Во-первых, связывание объектов с атрибутами или функциональностью делает программу более гибкой и упрощает ее расширение. Во-вторых, списки параметров шаблонов остаются небольшими и удобочитаемыми. Если бы с каждым «гостем» ассоциировалось 30 типов, было бы крайне неудобно указывать все 30 аргументов при каждом объявлении **BearCorner**. Выделение типов в отдельный класс характеристик существенно упрощает ситуацию.

Как было показано в главе 4, характеристики также применяются в реализациях потоков данных и локальных контекстов. Пример характеристик итераторов используется в заголовочном файле **PrintSequence.h** главы 6.

Политики

Просмотр специализации char_traits для wchar_t показывает, что она почти не отличается от аналогичной специализации для char:

```
template<>
  struct char_traits<wchar_t> {
  typedef wchar_t char_type;
  typedef wint_t int_type;
  typedef streamoff off_type;
  typedef wstreampos pos_type;
  typedef mbstate_t state_type;
  static void assign(char_type& c1, const char_type& c2);
  static bool eq(const char_type& c1, const char_type& c2);
  static bool lt(const char_type& c1, const char_type& c2);
  static int compare(const char_type* s1,
                     const char_type*  s2, size_t n);
  static size_t length(const char_type* s);
  static const char_type* find(const char_type* s, size_t n,
                               const char_type& a);
  static char_type* move(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* copy(char_type* s1,
                         const char_type* s2, size_t n);
  static char_type* assign(char_type* s, size_t n,
                           char_type a);
  static int_type not_eof(const int_type& c);
  static char_type to_char_type(const int_type& c);
  static int_type to_int_type(const char_type& c);
  static bool eq_int_type(const int_type& c1,
                          const int_type& c2);
  static int_type eof();
};
```

Две версии реально различаются лишь набором типов (char и int против wchar_t и wint_t), а функциональность остается той же[1]. Этим подчеркивается то обстоятельство, что классы характеристик действительно что-то *характеризуют*, а различия между взаимосвязанными классами характеристик обычно проявляются в типах, константах или фиксированных алгоритмах, которые используют параметры шаблонов, определяющие типы. Классы характеристик обычно сами оформляются в виде шаблонов, поскольку представленные в них типы и константы рассматриваются как атрибуты параметров основного шаблона (например, char и wchar_t).

В некоторых ситуациях бывает удобно связать *функциональность* с аргументами шаблонов, чтобы прикладной программист мог легко изменить поведение шаблона. Для примера рассмотрим новую версию BearCorner:

```
//: C05:BearCorner2.cpp
// Использование классов политик
#include <iostream>
#include "BearCorner.h"
using namespace std;

// Классы политик (должны содержать статическую функцию doAction())
```

[1] Тот факт, что char_traits<>::compare() может в одном случае вызывать strcmp(), а в другом — wcscmp(), в данном случае неважен; «функциональность» вызова compare() одинакова.

```
class Feed {
public:
  static const char* doAction() { return "Feeding"; }
};

class Stuff {
public:
  static const char* doAction() { return "Stuffing"; }
};

// Шаблон Guest (использует класс политик и класс характеристик)
template< class Guest, class Action,
          class traits = GuestTraits<Guest> >
class BearCorner {
  Guest theGuest;
  typedef typename traits::beverage_type beverage_type;
  typedef typename traits::snack_type snack_type;
  beverage_type bev;
  snack_type snack;
public:
  BearCorner(const Guest& g) : theGuest(g) {}
  void entertain() {
    cout << Action::doAction() << " " << theGuest
         << " with " << bev
         << " and " << snack << endl;
  }
};

int main() {
  Boy cr;
  BearCorner<Boy, Feed> pc1(cr);
  pc1.entertain();
  Bear pb;
  BearCorner<Bear, Stuff> pc2(pb);
  pc2.entertain();
} ///:~
```

Предполагается, что параметр шаблона **Action** в классе **BearCorner** содержит статическую функцию **doAction()**, задействованную в функции **BearCorner<>::entertain()**. Пользователь может выбирать между классами **Feed** и **Stuff**, каждый из которых содержит нужную функцию. Классы, инкапсулирующие функциональность подобным образом, называются *классами политик*. В нашем примере «политики» предоставляются через **Feed::doAction()** и **Stuff::doAction()**. Здесь классы политик являются обычными классами, но они могут оформляться в виде шаблонов и даже использовать наследование, что открывает дополнительные возможности.

Псевдорекурсия и подсчет объектов

Даже неопытный программист C++ знает, как отслеживать количество существующих объектов класса. Все, что для этого нужно, — добавить статические переменные и немного изменить логику конструктора и деструктора:

```
//: C05:CountedClass.cpp
// Подсчет объектов с использованием статических переменных
#include <iostream>
using namespace std;
```

```
class CountedClass {
  static int count;
public:
  CountedClass() { ++count; }
  CountedClass(const CountedClass&) { ++count; }
  ~CountedClass() { --count; }
  static int getCount() { return count; }
};

int CountedClass::count = 0;

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;   // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;   // 2
  { // Произвольная область видимости:
    CountedClass c(b);
    cout << CountedClass::getCount() << endl; // 3
    a = c;
    cout << CountedClass::getCount() << endl; // 3
  }
  cout << CountedClass::getCount() << endl;   // 2
} ///:~
```

Все конструкторы **CountedClass** увеличивают статическую переменную **count**, а деструкторы уменьшают ее. Статическая функция **getCount()** возвращает текущее количество объектов.

Было бы довольно утомительно вручную добавлять эти члены в каждый класс, в котором вы хотите организовать подсчет объектов. Стандартный объектно-ориентированный подход, применяемый, чтобы организовать многократное или совместное использование кода, — наследование. Однако наследование в данном случае решает лишь половину проблемы. Посмотрите, что происходит при включении логики подсчета объектов в базовый класс:

```
//: C05:CountedClass2.cpp
// Неправильная попытка подсчета объектов
#include <iostream>
using namespace std;

class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};

int Counted::count = 0;

class CountedClass : public Counted {};
class CountedClass2 : public Counted {};

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;    // 1
  CountedClass b;
```

```
  cout << CountedClass::getCount() << endl;    // 2
  CountedClass2 c;
  cout << CountedClass2::getCount() << endl;   // 3 (Ошибка)
} ///:~
```

Все классы, производные от Counted, используют одну статическую переменную, поэтому количество объектов отслеживается сразу *по всем* классам иерархии Counted. Нам же нужен способ автоматического построения *отдельного* базового класса для каждого производного класса. Задача решается при помощи занятной шаблонной конструкции, представленной ниже:

```
//: C05:CountedClass3.cpp
#include <iostream>
using namespace std;

template<class T>
class Counted {
  static int count;
public:
  Counted() { ++count; }
  Counted(const Counted<T>&) { ++count; }
  ~Counted() { --count; }
  static int getCount() { return count; }
};

template<class T> int Counted<T>::count = 0;

// Необычные объявления классов
class CountedClass : public Counted<CountedClass> {};
class CountedClass2 : public Counted<CountedClass2> {};

int main() {
  CountedClass a;
  cout << CountedClass::getCount() << endl;    // 1
  CountedClass b;
  cout << CountedClass::getCount() << endl;    // 2
  CountedClass2 c;
  cout << CountedClass2::getCount() << endl;   // 1 (!)
} ///:~
```

Каждый производный класс наследует от уникального базового класса, при определении которого в параметре шаблона указывается сам производный класс! На первый взгляд похоже на циклическое определение, и так бы оно и было, если бы какие-нибудь члены базовых классов использовали аргумент шаблона в вычислениях. Но так как ни одна переменная Counted не зависит от T, размер Counted (ноль!) известен на момент обработки шаблона. А значит, какой бы аргумент ни использовался для специализации Counted, размер все равно не изменится. Любое наследование от специализации Counted полностью определяется на момент обработки, и никакой рекурсии не возникает. Поскольку каждый базовый класс уникален, он обладает собственным набором статических данных, поэтому в вашем распоряжении оказывается удобная методика для организации подсчета объектов в абсолютно любом классе. Первое упоминание об этой любопытной идиоме встречается в статье Джима Коплин (Jim Coplien)[1].

[1] Статья «Curiously Recurring Template Patterns» в сборнике «C++ Gems» под редакцией Стена Липмана (Stan Lippman), SIGS, 1996 г.

Шаблонное метапрограммирование

В 1993 году в компиляторах постепенно стала реализовываться поддержка простых шаблонных конструкций, а пользователи начали определять обобщенные контейнеры и функции. Примерно в то же время, когда рассматривалась возможность включения STL в стандарт C++, члены комитета по стандартизации C++ обменивались друг с другом интересными и удивительными примерами вроде следующего[1]:

```
//: C05:Factorial.cpp
// Вычисление факториала на стадии компиляции!
#include <iostream>
using namespace std;
template<int n>
struct Factorial {
   enum {val = Factorial<n-1>::val * n};
};
template<>
struct Factorial<0> {
   enum {val = 1};
};
int main() {
   cout << Factorial<12>::val << endl; // 479001600
} ///:~
```

Программа выводит правильное значение 12!, и это нормально. Удивляет другое — результат вычисляется еще до запуска программы!

Пытаясь создать специализацию Factorial<12>, компилятор выясняет, что он также должен создать специализацию Factorial<11>, для которой нужна специализация Factorial<10> и т. д. Рекурсия заканчивается на специализации Factorial<1>, после чего вычисления проводятся в обратном направлении. В конечном счете переменная Factorial<12>::val замещается целочисленной константой 479 001 600, и компиляция завершается. Поскольку все вычисления выполняются компилятором, участвующие в них значения должны быть константами времени компиляции, отсюда и использование константы enum. При запуске программы остается лишь вывести эту константу и символ перевода строки. Чтобы убедиться в том, что специализация Factorial создает правильное значение времени компиляции, вы можете указать его как размерность массива:

```
double nums[Factorial<5>::val];
assert(sizeof nums == sizeof(double)*120);
```

Программирование на стадии компиляции

Итак, то, что задумывалось как удобный способ подстановки типовых параметров, превратилось в некое подобие механизма программирования на стадии компиляции. Такие программы, которые называются *шаблонными метапрограммами*, оказываются способными на многое. Шаблонное метапрограммирование обладает *полнотой по Тьюрингу*, поскольку оно поддерживает выбор (if-else) и циклы (посредством рекурсии). Следовательно, теоретически в метапрограммах можно выполнять любые вы-

[1] С технической точки зрения речь идет о константах времени компиляции, и по стандартным правилам их следовало бы записать прописными буквами. Мы оставили для имен строчные буквы, потому что они имитируют переменные.

числения[1]. Ранее приведенный пример с факториалом демонстрирует реализацию циклов: вы пишете рекурсивный шаблон и определяете критерий остановки посредством специализации. Следующий пример показывает, как аналогичным способом на стадии компиляции вычисляется последовательность чисел Фибоначчи:

```
//: C05:Fibonacci.cpp
#include <iostream>
using namespace std;

template<int n> struct Fib {
   enum {val = Fib<n-1>::val + Fib<n-2>::val};
};

template<> struct Fib<1> { enum {val = 1}; };

template<> struct Fib<0> { enum {val = 0}; };

int main() {
   cout << Fib<5>::val << endl;   // 6
   cout << Fib<20>::val << endl;  // 6765
} ///:~
```

На языке математики числа Фибоначчи определяются так:

$$f_n = \begin{cases} 0, n = 0, \\ 1, n = 1, \\ f_{n-2} + f_{n-2}, n > 1. \end{cases}$$

Первые два правила оформляются как специализации, а третье правило преобразуется в основной шаблон.

Циклы на стадии компиляции

Чтобы выполнить любой цикл в шаблонной метапрограмме, необходимо сначала сформулировать его в рекурсивном виде. Например, для возведения целого числа n в степень p в обычной программе применяется цикл вида:

```
int val = 1;
while(p--)
  val *= n;
```

Вместо этого цикла можно воспользоваться рекурсивной процедурой:

```
int power(int n, int p) {
  return (p == 0) ? 1 : n*power(n,p - 1);
```

Она легко формулируется в виде шаблонной метапрограммы:

```
//: C05:Power.cpp
#include <iostream>
using namespace std;

template<int N, int P> struct Power {
  enum {val = N * Power<N, P-1>::val};
};
```

[1] В 1966 году Бём (Bohm) и Якопини (Jacopini) доказали, что любой язык с поддержкой конструкций условного выбора и циклов, позволяющий использовать произвольное количество переменных, эквивалентен машине Тьюринга, которая способна представить любой алгоритм.

```
template<int N> struct Power<N, 0> {
  enum {val = 1};
};

int main() {
  cout << Power<2, 5>::val << endl;  // 32
} ///:~
```

Условие остановки должно задаваться неполной специализацией, поскольку значение N остается свободным параметром шаблона. Помните, что программа работает только для неотрицательных показателей степени.

Следующая метапрограмма, позаимствованная из книги Чарнецки (Charnecki) и Эйзенекера (Eisenecker)[1], интересна тем, что в ней используется шаблон в качестве параметра шаблона и имитируется передача функции как параметра другой функции, которая «перебирает» числа 0..n:

```
//: C05:Accumulate.cpp
// Передача "функции" как параметра на стадии компиляции
#include <iostream>
using namespace std;

// Накопление результатов F(0)..F(n)
template<int n, template<int> class F> struct Accumulate {
  enum {val = Accumulate<n-1, F>::val + F<n>::val};
};

// Критерий остановки (возвращает значение F(0))
template<template<int> class F> struct Accumulate<0, F> {
  enum {val = F<0>::val};
};

// Различные "функции":
template<int n> struct Identity {
  enum {val = n};
};

template<int n> struct Square {
  enum {val = n*n};
};

template<int n> struct Cube {
  enum {val = n*n*n};
};

int main() {
  cout << Accumulate<4, Identity>::val << endl; // 10
  cout << Accumulate<4, Square>::val << endl;   // 30
  cout << Accumulate<4, Cube>::val << endl;     // 100
} ///:~
```

Основной шаблон Accumulate вычисляет сумму $F(n) + F(n-1) \ldots F(0)$. Критерий остановки определяется неполной реализацией, которая «возвращает» $F(0)$. Параметр F сам по себе является шаблоном и работает как функция (по аналогии с предыдущими примерами этого раздела). Шаблоны Identity, Square и Cube вычисляют соответственно значение своего параметра, его квадрат и куб. Первая специализация Accumulate в main() вычисляет 4 + 3 + 2 + 1, поскольку шаблон Identity просто

[1] «Generative Programming: Methods, Tools, and Applications», Addison-Wesley, 2000, с. 417.

«возвращает» свой параметр. Вторая строка main() суммирует квадраты этих чисел (16 + 9 + 4 + 1 + 0), а третья — их третьи степени (64 + 27 + 8 + 1 + 0).

Развертывание цикла

Разработчики алгоритмов всегда стремились оптимизировать свои программы. Одной из классических оптимизаций в области математического программирования является развертывание цикла — методика, сокращающая издержки на выполнение циклов. Наиболее типичным примером развертывания цикла является матричное умножение. Следующая функция умножает матрицу на вектор (предполагается, что константы ROWS и COLS определены):

```
void mult(int a[rows][cols], int x[cols], int y[cols]) {
  for (int i = 0; i < rows; ++i) {
      y[i] = 0;
      for (int j = 0; j < cols; ++j)
        y[i] += a[i][j]*x[j];
  }
}
```

Если значение COLS четно, то издержки на увеличение и сравнение счетчика j можно сократить наполовину, «развертывая» вычисления во внутреннем цикле:

```
void mult(int a[rows][cols], int x[cols], int y[cols]) {
  for (int i = 0; i < rows; ++i) {
      y[i] = 0;
      for (int j = 0; j < cols; j += 2)
        y[i] += a[i][j]*x[j] + a[i][j+1]*x[j+1];
  }
}
```

В общем случае, если значение COLS кратно k, при каждой итерации внутреннего цикла можно выполнять k операций, что существенно повышает эффективность цикла. Конечно, экономия заметна только для очень больших массивов, но именно такие объемы данных обрабатываются современными математическими системами.

Подстановка функций тоже может рассматриваться как своего рода разновидность развертывания циклов. Рассмотрим следующий способ вычисления степеней целых чисел:

```
//: C05:Unroll.cpp
// Развертывание неявного цикла путем подстановок
#include <iostream>
using namespace std;

template<int n>
inline int power(int m) {
   return power<n-1>(m) * m;
}
template<>
inline int power<1>(int m) {
   return m;
}
template<>
inline int power<0>(int m) {
   return 1;
}
int main()
{
```

```
  int m = 4;
  cout << power<3>(m) << endl;
} ///:~
```

Формально компилятор должен генерировать три специализации шаблона power<>, по одной для каждого параметра 3, 2 и 1. Поскольку код каждой из этих функций может подставляться, фактически в main() будет вставлено единственное выражение m*m*m. Таким образом, простая специализация шаблона в сочетании с подстановкой позволяет полностью избавиться от издержек управления циклом[1]. Эта разновидность развертывания цикла ограничивается максимальной глубиной подстановки в вашем компиляторе.

Условный выбор на стадии компиляции

Чтобы имитировать условные конструкции на стадии компиляции, можно воспользоваться условным тернарным оператором в объявлении enum. В следующей программе эта методика применяется для вычисления на стадии компиляции большего из двух целых чисел:

```
//: C05:Max.cpp
#include <iostream>
using namespace std;

template<int n1, int n2> struct Max {
  enum {val = n1 > n2 ? n1 : n2};
};

int main() {
  cout << Max<10, 20>::val << endl;  // 20
} ///:~
```

Чтобы в зависимости от условия компилятор генерировал разный код, используйте специализации с параметрами true и false:

```
//: C05:Conditionals.cpp
// Применение условий стадии компиляции для выбора кода
#include <iostream>
using namespace std;

template<bool cond> struct Select {};

template<> struct Select<true> {
  static inline void statement1() {
    cout << "This is statement1 executing\n";
  }
public:
  static inline void f() { statement1(); }
};

template<> struct Select<false> {
  static inline void statement2() {
    cout << "This is statement2 executing\n";
  }
public:
  static inline void f() { statement2(); }
```

[1] Существует гораздо более эффективный способ вычисления степеней целых чисел, который называется «алгоритмом русского крестьянина».

```
};

template<bool cond> void execute() {
  Select<cond>::f();
}

int main() {
  execute<sizeof(int) == 4>();
} ///:~
```

Эта программа эквивалентна следующему выражению:

```
if(cond)
  statement1();
else
  statement2();
```

Отличие только в том, что условие cond проверяется на стадии компиляции, а компилятор создает специализации соответствующих версий execute<>() и Select<>. Функция Select<>::f() выполняется на стадии выполнения. Команда switch эмулируется аналогичным образом, только специализация производится по всем вариантам вместо двух (true и false).

Утверждения времени компиляции

В главе 2 отмечалась польза утверждений как части общей стратегии защитного программирования. Утверждение фактически представляет собой проверку логического выражения, за которой программа либо не делает ничего (если условие истинно), либо аварийно завершается с выдачей диагностических сообщений. Понятно, что нарушения утверждений должны выявляться как можно раньше. Если условие может быть проверено на стадии компиляции, лучше использовать утверждение времени компиляции. В следующем примере показан простой прием с отображением логического выражения на объявление массива:

```
//: C05:StaticAssert1.cpp {-xo}
// Простой механизм проверки условий времени компиляции

#define STATIC_ASSERT(x) \
  do { typedef int a[(x) ? 1 : -1]; } while (0)

int main() {
  STATIC_ASSERT(sizeof(int) <= sizeof(long)); // Проходит
  STATIC_ASSERT(sizeof(double) <= sizeof(int)); // Не проходит
} ///:~
```

Цикл do создает временную область видимости для определения массива, размер которого задается проверяемым условием. Определение массива с размером –1 недопустимо, поэтому при нарушении условия команда также не выполняется.

В предыдущем разделе было показано, как организовать проверку логических выражений на стадии компиляции. Однако для полноценной эмуляции проверки утверждений на стадии компиляции нужно решить еще одну задачу: вывести осмысленное сообщение об ошибке и завершить работу программы. Для прерывания работы компилятора достаточно простой ошибки компиляции; фокус заключается в том, чтобы вставить осмысленный текст в сообщение об ошибке. В следующем примере, предложенном Александреску (Alexandrescu), используется специализация шаблона, локальный класс и трюк с макросами:

```
//: C05:StaticAssert.cpp {-g++}
#include <iostream>
using namespace std;

// Шаблон и специализация
template<bool> struct StaticCheck {
  StaticCheck(...);
};

template<> struct StaticCheck<false>{};

// Макрос (генерирует локальный класс)
#define STATIC_CHECK(expr, msg) {              \
  class Error_##msg{};                         \
  sizeof((StaticCheck<expr>(Error_##msg()))); \
}

// Обнаружение сужающих преобразований
template<class To, class From> To safe_cast(From from) {
  STATIC_CHECK(sizeof(From) <= sizeof(To),
               NarrowingConversion);
  return reinterpret_cast<To>(from);
}

int main() {
  void* p = 0;
  int i = safe_cast<int>(p);
  cout << "int cast okay" << endl;
//!  char c = safe_cast<char>(p);
} ///:~
```

В этом примере определяется шаблон функции safe_cast<>() для обнаружения сужающих преобразований, когда тип исходного объекта больше целевого типа, получаемого в результате преобразования. Если тип целевого объекта оказывается меньше, пользователь во время компиляции получает сообщение о попытке сужающего преобразования. Обратите внимание на любопытную особенность шаблона StaticCheck: к специализации StaticCheck<true> можно преобразовать *все, что угодно* (благодаря многоточию в конструкторе), а к специализации StaticCheck<false> нельзя преобразовать *ничего*, потому что для этой специализации отсутствуют преобразования. Идея заключается в том, чтобы попытаться создать экземпляр нового класса и преобразовать его в объект StaticCheck<true> *во время компиляции*, если проверяемое условие истинно, или в объект StaticCheck<false>, если условие ложно. Поскольку оператор sizeof выполняет свою работу на стадии компиляции, он задействуется для попытки выполнения преобразования. Если условие ложно, компилятор сообщает, что он не знает, как преобразовать тип нового класса к StaticCheck<false> (дополнительные круглые скобки внутри вызова sizeof в STATIC_CHECK() нужны для того, чтобы компилятор не решил, будто мы пытаемся вызвать оператор sizeof для функции, что недопустимо). Чтобы сообщение об ошибке содержало полезную информацию, ключевой текст помещается в имя нового класса.

Впрочем, эту методику проще всего разобрать на конкретном примере. Возьмем следующую строку в функции main():

```
int i = safe_cast<int>(p);
```

В вызове safe_cast<int>(p) первая строка программы заменяется следующим расширением макроса (вспомните, что препроцессорный оператор ## выполняет

конкатенацию своих операндов, поэтому Error_##NarrowingConversion после обработки препроцессором превращается в Error_Narrowing_conversion):

```
{                                                    \
   class Error_NarrowingConversion{};                \
   sizeof(StaticCheck<sizeof(void*) <= sizeof(int)> \
          (Error_NarrowingConversion()));            \
}
```

Класс Error_Narrowing_conversion объявляется как *локальный*, поскольку в программе он больше нигде не используется. Оператор sizeof пытается определить размер экземпляра StaticCheck<true> (так как условие sizeof(void*)<=sizeof(int) истинно на всех платформах), неявно созданного на базе временного объекта, возвращаемого вызовом Error_NarrowingConversion(). Компилятор знает размер нового класса Error_Narrowing_conversion (так как он пуст), поэтому использование sizeof во время компиляции на внешнем уровне STATIC_CHECK() допустимо. Преобразование временного объекта Error_Narrowing_conversion в StaticCheck<true> проходит успешно, внешнее применение StaticCheck<true> тоже, и выполнение программы продолжается.

Теперь посмотрим, чтоб произойдет, если раскомментировать последнюю строку main():

```
char c = safe_cast<char>(p);
```

На этот раз макрос STATIC_CHECK() внутри safe_cast<char>(p) расширяется в следующий фрагмент:

```
{                                                    \
   class Error_NarrowingConversion{};                \
   sizeof(StaticCheck<sizeof(void*) <= sizeof(char)> \
          (Error_NarrowingConversion()));            \
}
```

Поскольку выражение sizeof(void*)<=sizeof(char) ложно, компилятор пытается преобразовать временный объект Error_NarrowingConversion в StaticCheck<false>:

```
sizeof(StaticCheck<false>(Error_NarrowingConversion()));
```

Попытка завершается неудачей, и компилятор прекращает свою работу с выдачей сообщения вида

```
Cannot cast from 'Error_NarrowingConversion' to
'StaticCheck<0>' in function
char safe_cast<char,void *>(void *)
```

Имя класса Error_NarrowingConversion представляет собой осмысленное сообщение, специально подготовленное программистом. В общем случае для проверки статических утверждений достаточно вызвать макрос STATIC_CHECK с проверяемым условием и осмысленным именем, описывающим ошибку.

Шаблоны выражений

Вероятно, самое мощное применение шаблонам было найдено в 1994 г. Тоддом Вельдхузеном (Todd Veldhuizen) и Дэвидом Вандерворде (Daveed Vandervoorde) независимо друг от друга[1]. Шаблоны выражений позволяют кардинально оптими-

[1] Стоит заметить, что помимо сохранения математической записи и оптимизации кода шаблоны выражений позволяют реализовывать в библиотеках C++ парадигмы и механизмы (такие как лямбда-выражения) других языков программирования. В качестве еще одного примера можно назвать фантастическую библиотеку классов Spirit (за информацией обращайтесь на сайт http://spirit.sourceforge.net/).

зировать некоторые виды вычислений и получить код, который по скорости как минимум не уступает вручную оптимизированному коду языка Фортран, но при этом сохраняет естественную математическую запись за счет перегрузки операторов. Хотя вряд ли вы будете использовать эту методику в повседневном программировании, она заложена в основу многих сложных, высокопроизводительных математических библиотек, написанных на C++[1].

Чтобы было понятно, зачем нужны шаблоны выражений, рассмотрим типичные операции линейной алгебры, такие, как сложение матриц или векторов[2]:

```
D = A + B + C;
```

В примитивной реализации этого выражения создаются временные объекты для `A + B` и `(A + B) + C`. Если переменные представляют матрицы или векторы очень больших размеров, затраты ресурсов на хранение временных объектов могут оказаться неприемлемыми. Шаблоны выражений позволяют использовать аналогичные выражения без временных объектов.

В следующей программе определяется класс `MyVector` для представления математических векторов произвольного размера. Длина вектора определяется нетиповым аргументом шаблона. Мы также определяем промежуточный класс `MyVectorSum` для хранения суммы объектов `MyVector`. Это позволяет выполнять отложенные вычисления, чтобы векторы-компоненты можно было добавлять по мере необходимости без создания временных объектов.

```
//: C05:MyVector.cpp
// Оптимизация посредством шаблонов с исключением временных объектов
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// Промежуточный класс для суммы векторов
template<class, size_t> class MyVectorSum;

template<class T, size_t N> class MyVector {
  T data[N];
public:
  MyVector<T,N>& operator=(const MyVector<T,N>& right) {
    for (size_t i = 0; i < N; ++i)
      data[i] = right.data[i];
    return *this;
  }
  MyVector<T,N>& operator=(const MyVectorSum<T,N>& right);
  const T& operator[](size_t i) const { return data[i]; }
  T& operator[](size_t i) { return data[i]; }
};

// Промежуточный класс содержит ссылки
// и использует отложенное суммирование.
template <class T, size_t N> class MyVectorSum {
  const MyVector<T,N>& left;
```

[1] А именно Blitz (http://www.oonumerics.org/blitz/), Matrix Template Library (http://www.osl.iu.edu/research/mtl) и POOMA (http://www.acl.lanl.gov/pooma/).

[2] Речь идет о векторах в математическом смысле, то есть одномерных числовых массивах фиксированной длины.

```
  const MyVector<T,N>& right;
public:
  MyVectorSum(const MyVector<T,N>& lhs,
              const MyVector<T,N>& rhs)
      : left(lhs), right(rhs) {}
  T operator[](size_t i) const {
    return left[i] + right[i];
  }
};
// Operator= для операций v3 = v1 + v2
template<class T, size_t N> MyVector<T,N>&
MyVector<T,N>::operator=(const MyVectorSum<T,N>& right) {
  for (size_t i = 0; i < N; ++i)
    data[i] = right[i];
  return *this;
}
// Operator+ просто сохраняет ссылки
template<class T, size_t N>
inline MyVectorSum<T,N>
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N>(left, right);
}

// Вспомогательные функции для тестовой программы
template<class T, size_t N> void init(MyVector<T,N>& v) {
  for (size_t i = 0; i < N; ++i)
    v[i] = rand() % 100;
}

template<class T, size_t N> void print(MyVector<T,N>& v) {
  for (size_t i = 0; i < N; ++i)
    cout << v[i] << ' ';
  cout << endl;
}

int main() {
  srand(time(0));
  MyVector<int, 5> v1;
  init(v1);
  print(v1);
  MyVector<int, 5> v2;
  init(v2);
  print(v2);
  MyVector<int, 5> v3;
  v3 = v1 + v2;
  print(v3);
  MyVector<int, 5> v4;
  // Пока не поддерживается:
//!  v4 = v1 + v2 + v3;
} ///:~
```

Класс MyVectorSum при создании ничего не вычисляет; он просто сохраняет ссылки на два суммируемых объекта. Вычисление производится только при обращении к компоненту суммы векторов (см. операторную функцию operator[]()). Перегрузка оператора присваивания MyVector с аргументом MyVectorSum предназначена для выражений вида

```
v1 = v2 + v3;  // Суммирование двух векторов
```

При вычислении v1 + v2 возвращается myVectorSum — компактный объект фиксированного размера, содержащий только две ссылки. Затем вызывается упоминавшийся выше оператор присваивания:

```
v3.operator=<int,5>(MyVectorSum<int,5>(v2, v3));
```

Тем самым каждому элементу v3 присваивается сумма соответствующих элементов v1 и v2, вычисляемая в реальном времени. Временные объекты MyVector при этом не создаются.

Однако приведенная программа не поддерживает выражения, содержащие более двух операндов:

```
v4 = v1 + v2 + v3;
```

Дело в том, что после первого суммирования делается следующая попытка

```
(v1 + v2) + v3;
```

Для нее требуется функция operator+() с аргументами MyVectorSum и аргументом MyVector. Можно определить несколько перегруженных версий для всех случаев, но лучше поручить работу шаблонам, как в следующей версии программы:

```
//: C05:MyVector2.cpp
// Вычисление сумм произвольной длины с использованием шаблонов выражений
#include <cstddef>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// Промежуточный класс для суммы векторов
template<class, size_t, class, class> class MyVectorSum;

template<class T, size_t N> class MyVector {
  T data[N];
public:
  MyVector<T,N>& operator=(const MyVector<T,N>& right) {
    for (size_t i = 0; i < N; ++i)
      data[i] = right.data[i];
    return *this;
  }
  template<class Left, class Right> MyVector<T,N>&
  operator=(const MyVectorSum<T,N,Left,Right>& right);
  const T& operator[](size_t i) const {
    return data[i];
  }
  T& operator[](size_t i) {
    return data[i];
  }
};
// Позволяет смешивать MyVector с MyVectorSum
template <class T, size_t N, class Left, class Right>
class MyVectorSum {
  const Left& left;
  const Right& right;
public:
  MyVectorSum(const Left& lhs, const Right& rhs)
      : left(lhs), right(rhs) {}
  T operator[](size_t i) const {
    return left[i] + right[i];
```

```
  }
};

template<class T, size_t N>
template<class Left, class Right>
MyVector<T,N>&
MyVector<T,N>::
operator=(const MyVectorSum<T,N,Left,Right>& right) {
  for (size_t i = 0; i < N; ++i)
    data[i] = right[i];
  return *this;
}
// Operator+ просто сохраняет ссылки
template<class T, size_t N>
inline MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
operator+(const MyVector<T,N>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N,MyVector<T,N>,MyVector<T,N> >
      (left,right);
}

template<class T, size_t N, class Left, class Right>
inline MyVectorSum<T, N, MyVectorSum<T,N,Left,Right>,
            MyVector<T,N> >
operator+(const MyVectorSum<T,N,Left,Right>& left,
          const MyVector<T,N>& right) {
  return MyVectorSum<T,N,MyVectorSum<T,N,Left,Right>,
                         MyVector<T,N> >
    (left, right);
}
// Вспомогательные функции для тестовой программы
template<class T, size_t N>
void init(MyVector<T,N>& v) {
  for (size_t i = 0; i < N; ++i)
    v[i] = rand() % 100;
}
template<class T, size_t N>
void print(MyVector<T,N>& v) {
  for (size_t i = 0; i < N; ++i)
    cout << v[i] << ' ';
  cout << endl;
}
int main() {
  srand(time(0));
  MyVector<int, 5> v1;
  init(v1);
  print(v1);
  MyVector<int, 5> v2;
  init(v2);
  print(v2);
  MyVector<int, 5> v3;
  v3 = v1 + v2;
  print(v3);
  // Теперь поддерживается:
  MyVector<int, 5> v4;
  v4 = v1 + v2 + v3;
  print(v4);
  MyVector<int, 5> v5;
  v5 = v1 + v2 + v3 + v4;
```

```
  print(v5);
} ///:~
```

Шаблон определяет типы операндов по аргументам Left и Right вместо того, чтобы фиксировать их заранее. Шаблон MyVectorSum получает эти два дополнительных параметра, что позволяет представить сумму произвольной комбинации пар MyVector и MyVectorSum.

Оператор присваивания теперь оформлен в виде шаблона функции класса. В результате любая пара <T,N> может комбинироваться с любой парой <Left,Right>, поэтому объекту MyVector может быть присвоен объект MyVectorSum, содержащий ссылки на любую возможную пару типов MyVector и MyVectorSum.

Как и в предыдущем примере, давайте проследим за ходом присваивания, начиная с выражения

```
v4 = v1 + v2 + v3;
```

Поскольку итоговые выражения получаются весьма громоздкими, в дальнейших пояснениях мы сокращаем MyVectorSum до MVS и не приводим аргументы шаблонов.

Сначала выполняется операция v1 + v2, для которой вызывается подставляемая функция operator+(). В свою очередь, она вставляет выражение MVS(v1,v2) в поток компиляции. Результат прибавляется к v3, что приводит к созданию временного объекта в соответствии с выражением MVS(MVS(v1,v2),v3). В окончательном виде вся команда выглядит так:

```
v4.operator+(MVS(MVS(v1, v2), v3));
```

Все эти действия выполняются компилятором. Это объясняет, почему за этой методикой закрепилось название «шаблоны выражений». Шаблон MyVectorSum представляет выражение (сумму в данном случае), а вложенные вызовы напоминают дерево разбора левоассоциативного выражения v1 + v2 + v3.

Модели компиляции шаблонов

Вероятно, вы заметили, что во всех наших примерах шаблонов в каждую единицу трансляции включаются полные определения шаблонов (например, мы размещаем их полностью в программах из одного файла или в заголовочных файлах в программах из нескольких файлов). Это противоречит общепринятой практике отделения определений обычных функций от их объявлений, когда объявления размещаются в заголовочных файлах, а реализация — в cpp-файлах.

Для традиционных функций такое разделение объясняется следующими причинами:

- присутствие тел функций, не являющихся подставляемыми, в заголовочных файлах приводит к повторному определению функций и ошибкам компоновки;
- скрытие реализации от клиента ослабляет привязку на стадии компиляции;
- фирмы-разработчики могут распространять заранее откомпилированный код (для конкретного компилятора) с заголовками, чтобы пользователи не видели реализацию функций;
- заголовочные файлы имеют меньшие размеры, что сокращает время компиляции.

Модель с включением

С другой стороны, шаблон представляет собой не фрагмент программного кода как таковой, а лишь инструкции для построения кода. Только специализации шаблонов содержат настоящий код. Если компилятор видит полное определение шаблона во время компиляции, а затем встречает специализацию этого шаблона в этой же единице трансляции, ему приходится учитывать тот факт, что идентичная специализация может присутствовать и в другой единице трансляции. Чаще всего компилятор генерирует код для всех специализаций во всех единицах трансляции, после чего компоновщик уничтожает дубликаты. Данное решение хорошо работает и для функций, которые объявлены как подставляемые, но подставляться не могут, и для таблиц виртуальных функций, чем отчасти объясняется его популярность. Впрочем, некоторые компиляторы предпочитают задействовать более сложные схемы, предотвращающие многократное генерирование одинаковых специализаций. Так или иначе, система трансляции C++ должна предотвратить ошибки, возникающие из-за появления одинаковых специализаций.

Однако у такого подхода имеются недостатки: весь исходный код шаблонов виден клиенту, и у разработчика библиотек практически нет возможности скрыть свою стратегию реализации. Кроме того, заголовочные файлы имеют гораздо большие размеры, чем при раздельной компиляции тел функций. Это может привести к серьезному замедлению компиляции по сравнению с традиционными моделями.

Для уменьшения размеров заголовочных файлов, необходимых для модели с полным включением, в C++ предусмотрены два альтернативных механизма организации кода (причем эти механизмы не являются взаимоисключающими). Вы можете вручную объявить все необходимые специализации, прибегнув к *явной специализации*, или воспользоваться *экспортированными шаблонами*.

Явная специализация

Вы можете прямо приказать компилятору сгенерировать любые указанные вами специализации шаблона. В этом случае для каждой специализации в программе может присутствовать только одна директива явной специализации; в противном случае возникают ошибки множественного определения, как для обычных функций с идентичными сигнатурами. Для демонстрации мы сначала (ошибочно) отделим объявление шаблона `min()`, приведенного в начале главы, от его определения, как это делается для обычных (не являющихся подставляемыми) функций. Следующий пример состоит из пяти файлов:

- `OurMin.h` — объявление шаблона функции `min()`;
- `OurMin.cpp` — определение шаблона функции `min()`;
- `UseMin1.cpp` — попытка использования специализации `min()` для `int`;
- `UseMin2.cpp` — попытка использования специализации `min()` для `double`;
- `MinMain.cpp` — вызовы `usemin1()` и `usemin2()`.

```
//: C05:OurMin.h
#ifndef OURMIN_H
#define OURMIN_H
// Объявление min()
template<typename T> const T& min(const T&, const T&);
```

```
#endif ///:~ OURMIN.Y ///:~

//: C05:OurMin.cpp
#include "OurMin.h"
//Определение min()
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}

//: C05:UseMin1.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin1() {
  std::cout << min(1.2) << std::endl;
} ///:~

//: C05:UseMin2.cpp {O}
#include <iostream>
#include "OurMin.h"
void usemin2() {
  std::cout << min(3.1,4.2) << std::endl;
} ///:~

//: C05:MinMain.cpp
//{L} UseMin1 UseMin2 MinInstances
void usemin1();
void usemin2();

int main() {
  usemin1();
  usemin2();
} ///:~
```

Если попытаться скомпоновать эту программу, компоновщик сообщит о неразрешенных внешних ссылках для min<int>() и min<double>(). Дело в том, что когда компилятор встречает вызовы специализаций min() в UseMin1 и UseMin2, он видит только объявление min(). А раз определение недоступно, компилятор считает, что оно находится в другой единице трансляции, и не генерирует специализации в этой точке. Естественно, компоновщик жалуется на то, что специализации не найдены.

Для решения проблемы мы создадим новый файл MinInstances.cpp, в котором явно объявляются необходимые специализации min():

```
//: C05:MinInstances.cpp {O}
#include "OurMin.cpp"

// Специализации для int и double
template const int& min<int>(const int&, const int&);
template const double& min<double>(const double&, const double&);
///:~
```

Чтобы вручную сгенерировать конкретную специализацию шаблона, перед объявлением специализации ставится ключевое слово template. Учтите, что в данном случае вместо OurMin.h необходимо включить файл OurMin.cpp, поскольку для построения специализаций компилятору необходимо определение шаблона. Впрочем, в других местах это делать не придется[1], поскольку необходимые уникальные специализации min() уже получены, и для остальных файлов объявлений доста-

[1] Как объяснялось ранее, эта директива может включаться в программу только один раз.

точно. Так как файл OurMin.cpp включается средствами препроцессора, также следует добавить защитные директивы:

```
//: C05:OurMin.cpp {O}
#ifndef OURMIN_CPP
#define OURMIN_CPP
#include "OurMin.h"

template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
}
#endif ///:~ OURMIN_CPP ///:~
```

Если теперь откомпилировать все файлы, уникальные специализации min() будут успешно найдены, и программа выведет правильный результат:

```
1
3.1
```

Явная специализация также может потребоваться для классов и статических переменных. При явном объявлении специализации класса автоматически генерируются все функции класса этой специализации, кроме тех, которые были явно сгенерированы ранее. Это важно, поскольку при использовании этого механизма становятся бесполезными многие шаблоны, а конкретно шаблоны, реализующие разную функциональность в зависимости от типов своих параметров. Неявная специализация здесь обладает преимуществами: при ней генерируются только вызываемые функции.

Механизм явной специализации предназначен для больших проектов, в которых он позволяет существенно сократить время компиляции. Впрочем, выбор между явной и неявной специализациями не зависит от используемой модели компиляции шаблонов. Явная специализация может применяться как в модели с включением, так и в модели с разделением (см. следующий раздел).

Модель с разделением

В модели компиляции шаблонов с разделением определения шаблонных функций или статических переменных классов отделяются от их объявлений и переносятся в другие единицы трансляции, как это обычно делается с обычными функциями и данными. Для этой цели используется *экспорт шаблонов*. После знакомства с двумя предыдущими разделами это может показаться странным. Зачем было вводить модель с включением, если можно было просто сохранить существующий порядок вещей? Причины имеют как исторический, так и технический характер.

Исторически модель с включением первой получила широкое коммерческое распространение. В наши дни все компиляторы C++ поддерживают модель с включением. Отчасти это объясняется тем, что нормальная спецификация модели с разделением появилась лишь на поздней стадии процесса стандартизации, а также тем, что модель с включением проще реализуется. Множество работоспособных программ было написано задолго до приведения семантики модели с разделением к окончательному виду.

С реализацией модели с разделением возникает столько трудностей, что летом 2003 года она поддерживалась только одним компилятором (внешним интерфейсом EDG). Более того, она продолжает требовать, чтобы исходный текст шаблона был доступен на стадии компиляции. Требование о наличии исходных

текстов планируется заменить некоторой формой промежуточного кода, что позволит распространять «заранее откомпилированные» шаблоны без исходных текстов. Из-за отмеченных ранее сложностей поиска (поиска зависимых имен в контексте определения шаблона) полное определение должно быть доступно в той или иной форме при компиляции программы, в которой этот шаблон специализируется.

Синтаксис отделения исходного кода определения шаблона от его объявления достаточно прост. Это делается при помощи ключевого слова **export**:

```
//: C05:OurMin2.h
// Объявление min как экспортированного шаблона
// (работает только в компиляторах на базе EDG)
#ifndef OURMIN2_H
#define OURMIN2_H
// Объявление min
export template<typename T> const T& min(const T&, const T&);
#endif ///:~ OURMIN2_H ///:~
```

Ключевое слово **export** по аналогии с **inline** или **virtual** должно присутствовать в потоке компиляции только один раз при первом упоминании экспортируемого шаблона. По этой причине повторять его в файле реализации не обязательно, хотя это рекомендуется:

```
// C05:OurMin2.cpp
// Определение экспортированного шаблона min
// (работает только в компиляторах на базе EDG)
#include "OurMin2.h"
export
template<typename T> const T& min(const T& a, const T& b) {
  return (a < b) ? a : b;
} ///:~
```

Остается лишь включить в файлы **UseMin** правильный заголовочный файл (**OurMin2.h**), а главная программа остается неизменной. Хотя на первый взгляд такой подход обеспечивает полноценное разделение, файл с определением шаблона (**OurMin2.cpp**) все равно должен поставляться пользователям (поскольку он должен обрабатываться для каждой специализации **min()**) до тех пор, пока не появится какая-либо форма представления промежуточного кода шаблона. Таким образом, хотя стандарт позволяет отделить объявление шаблона от определения, не все преимущества такого разделения доступны сегодня. Ключевое слово **export** сейчас поддерживается лишь одним семейством компиляторов (на базе внешнего интерфейса EDG), причем эти компиляторы сейчас не обеспечивают потенциальную возможность распространения определений шаблонов в откомпилированной форме.

Итоги

Возможности шаблонов далеко выходят за рамки простой параметризации по типам. В сочетании с автоматическим определением типов аргументов, переопределением специализаций и метапрограммированием шаблоны C++ превращаются в мощный механизм генерирования программного кода.

Среди недостатков шаблонов C++, о которых в этой главе не упоминалось, стоит упомянуть трудности с интерпретацией сообщений об ошибках компиляции;

многие компиляторы выдают очень длинные и абсолютно невразумительные сообщения. В последнее время сообщения об ошибках шаблонов в компиляторах C++ были несколько усовершенствованы, а Леор Золман (Leor Zolman) написал утилиту STLFilt, которая извлекает из этих сообщений полезную информацию и выкидывает все лишнее[1].

Еще одной важной идеей этой главы является то, что *шаблон подразумевает интерфейс*. Иначе говоря, хотя ключевое слово template означает: «Подойдет любой тип»; код определения шаблона требует поддержки некоторых операций и функций класса, то есть обеспечения определенного интерфейса. Так что на самом деле определение шаблона означает «Подойдет любой тип, поддерживающий данный интерфейс». Конечно, было бы хорошо, если бы компилятор мог просто сказать: «Тип, по которому параметризуется этот шаблон, не поддерживает этот интерфейс — ничего не выйдет». Шаблоны организуют своего рода «отложенную проверку типов», более гибкую по сравнению с традиционной объектно-ориентированной практикой порождения всех типов от некоторых базовых классов.

В главах 6 и 7 подробно рассматривается самое известное применение шаблонов — подмножество стандартной библиотеки C++, называемое стандартной библиотекой шаблонов (Standard Template Library, STL). В главах 9 и 10 также встречаются приемы использования шаблонов, не упомянутые в этой главе.

Упражнения

1. Напишите шаблон унарной функции, получающий один типовой параметр. Создайте полную специализацию для типа int. Также создайте нешаблонную перегрузку этой функции с одним параметром типа int. Вызовите в main() три разновидности функции.
2. Напишите шаблон класса, использующий класс vector для реализации стековой структуры данных.
3. Измените решение из предыдущего упражнения так, чтобы тип контейнера, используемого для реализации стека, определялся шаблоном, являющимся параметром шаблона.
4. В следующем фрагменте класс NonComparable не содержит операторной функции operator==(). Почему присутствие структуры HardLogic приводит к ошибке компиляции, а структуры SoftLogic — нет?

```
//: C05:Exercise4.cpp {-xo}
class Noncomparable {};

struct HardLogic {
  Noncomparable nc1, nc2;
  void compare() {
     return nc1 == nc2; // Ошибка компилятора
  }
};

template<class T> struct SoftLogic {
  Noncomparable nc1, nc2;
```

[1] См. http://www.bdsoft.com/tools/stlfilt.html.

```
  void noOp() {}
  void compare() {
    nc1 == nc2;
  }
};

int main() {
  SoftLogic<Noncomparable> l;
  l.noOp();
} ///:~
```

5. Напишите шаблон функции с одним типовым параметром T. Функция получает четыре аргумента: массив T, начальный индекс интервала, конечный индекс интервала и необязательное исходное значение. Функция возвращает сумму всех элементов массива в заданном интервале (с включением конечного индекса) и исходного значения. Исходное значение по умолчанию должно создаваться конструктором T по умолчанию.
6. Повторите предыдущее упражнение и создайте переопределенные специализации для int и double способом, описанным в этой главе.
7. Почему не компилируется следующая программа? В качестве подсказки попробуйте ответить, к чему получают доступ функции класса?

```
//: C05:Exercise7.cpp {-xo}
class Buddy {};

template<class T> class My {
  int i;
public:
  void play(My<Buddy>& s) {
    s.i = 3;
  }
};
int main() {
  My<int> h;
  My<Buddy> me, bud;
  h.play(bud);
  me.play(bud);
} ///:~
```

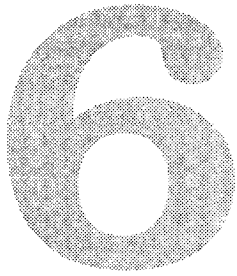
Почему не компилируется следующая программа?

```
//: C05:Exercise8.cpp {-xo}
template<class T> double pythag(T a, T b, T c) {
  return (-b + sqrt(double(b*b - 4*a*c))) / 2*a;
}

int main() {
  pythag(1, 2, 3);
  pythag(1.0, 2.0, 3.0);
  pythag(1, 2.0, 3.0);
  pythag<double>(1, 2.0, 3.0);
}
} ///:~
```

8. Напишите шаблон с нетиповыми параметрами следующих видов: int, указатель на int, указатель на статическую переменную класса типа int, указатель на статическую функцию класса.

9. Напишите шаблон класса с двумя типовыми параметрами. Определите неполную специализацию для первого параметра, а затем другую неполную специализацию второго параметра. Включите в каждую специализацию члены, отсутствующие в основном шаблоне.
10. Определите шаблон класса Bob с одним типовым параметром. Объявите Bob дружественным для всех специализаций шаблонного класса Friendly и для шаблонного класса Picky только при совпадении типов параметров Bob и Picky. Включите в Bob функции, демонстрирующие дружественные отношения между классами.

Обобщенные алгоритмы



Алгоритмы занимают центральное место в обработке данных. Универсальные алгоритмы, работающие с любыми наборами объектов, делают программу и проще, и безопаснее, а возможность влиять на работу алгоритма во время выполнения произвела настоящую революцию в программировании.

В подмножество стандартной библиотеки C++, известное под названием стандартной библиотеки шаблонов (Standard Template Library, STL), была изначально заложена концепция *обобщенных алгоритмов* — фрагментов кода для обработки последовательностей значений произвольного типа с обеспечением безопасности по отношению к типам. Это делалось для того, чтобы программист мог использовать готовые алгоритмы для решения практически любых задач, и ему не приходилось вручную программировать циклы каждый раз, когда потребуется обработать набор данных. Впрочем, чтобы овладеть всей мощью STL, вам придется изрядно потрудиться. К концу этой главы вы либо начнете относиться к алгоритмам как к привычному рабочему инструменту, либо решите, что разбираться в них слишком хлопотно. Большинство программистов обычно сначала сторонится алгоритмов, но с течением времени начинает все чаще ими пользоваться.

Первый взгляд

Среди прочего, обобщенные алгоритмы стандартной библиотеки формируют новую терминологию описания решений. После знакомства с алгоритмами у вас появится целый набор новых слов для описания выполняемых операций, относящихся к более высокому уровню абстракции. Никто не говорит: «перебираем элементы в цикле и последовательно присваиваем значения элементов первого интервала элементам другого интервала — вы просто говорите «интервальное копирование» (`copy()`), и все становится ясно. Собственно, программирование с первых дней своего существования решало именно эту задачу — создание высокоуровневых абстракций для выражения того, *что* вы делаете, и сокращение затрат времени на опреде-

ление того, *как* это делается. Проблема «как делать?» решается раз и навсегда и скрывается в коде алгоритма, готовом к многократному использованию.

Рассмотрим пример использования алгоритма copy:

```
//: C06:CopyInts.cpp
// Копирование последовательности int без явного определения цикла
#include <algorithm>
#include <cassert>
#include <cstddef>  // Для size_t
using namespace std;

int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  int b[SIZE];
  copy(a, a + SIZE, b);
  for (int i = 0; i < SIZE; ++i)
    assert(a[i] == b[i]);
} ///:~
```

Первые два параметра алгоритма copy() определяют *интервал* входной последовательности — в данном случае это массив a. Интервалы определяются парой указателей. Первый указатель ссылается на первый элемент последовательности, а второй — на элемент в позиции, следующей *за последним элементом* массива. Поначалу такой способ определения выглядит несколько странно, но это старая и довольно удобная идиома языка C. Например, разность этих двух указателей дает количество элементов в последовательности. Что еще важнее, в реализации copy второй указатель может использоваться как «барьер», позволяющий остановить перебор последовательности. Третий аргумент ссылается на начало выходной последовательности (массива b в нашем примере). Предполагается, что выходной массив имеет достаточный размер и сможет вместить копируемые элементы.

Если бы алгоритм copy() умел копировать только целые числа, особой пользы он бы не приносил. Однако этот алгоритм умеет копировать любые последовательности объектов. Например, в следующей программе копируются объекты типа string:

```
//: C06:CopyStrings.cpp
// Копирование строк
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <string>
using namespace std;

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  string b[SIZE];
  copy(a, a + SIZE, b);
  assert(equal(a, a + SIZE, b));
} ///:~
```

В этом примере также используется алгоритм equal(), который возвращает true только в том случае, если каждый элемент первого интервала равен соответствующему элементу второго интервала (проверка осуществляется оператором ==). В этом примере каждый интервал перебирается дважды: сначала для копирования, потом для сравнения, и при этом программа не содержит ни одного цикла!

Универсальность обобщенных алгоритмов объясняется тем, что они реализованы в виде шаблонов функций. Если вы думаете, что реализация copy() выглядит так, как показано ниже, то вы почти правы:

```
template<typename T> void copy(T* begin, T* end, T* dest) {
  while (begin != end)
    *dest++ = *begin++;
}
```

Мы говорим «почти», потому что copy() обрабатывает интервалы, границы которых определяются любыми объектами, похожими на указатели, например, итераторами. Благодаря такому подходу алгоритм copy() может использоваться для копирования вектора:

```
//: C06:CopyVector.cpp
// Копирование содержимого вектора
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <vector>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  vector<int> v1(a, a + SIZE);
  vector<int> v2(SIZE);
  copy(v1.begin(), v1.end(), v2.begin());
  assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

Первый вектор v1 инициализируется целыми числами из массива a. В определении вектора v2 используется другой конструктор vector, выделяющий память для SIZE элементов, инициализированных нулями (значение по умолчанию для int).

Как и в предыдущем примере с массивом, очень важно, чтобы вектор v2 имел достаточные размеры для размещения копии содержимого v1. Специальная библиотечная функция back_inserter() возвращает особый тип итератора, который *вставляет* элементы вместо их *перезаписи*. При этом контейнер автоматически выделяет новую память по мере надобности. Благодаря использованию функции back_inserter() в следующем примере нам не приходится заранее определять размер выходного вектора v2:

```
//: C06:InsertVector.cpp
// Присоединение элементов одного вектора к другому вектору
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <iterator>
#include <vector>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  vector<int> v1(a, a + SIZE);
  vector<int> v2;  // Вектор v2 пуст
  copy(v1.begin(), v1.end(), back_inserter(v2));
  assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

Функция back_inserter() определяется в заголовочном файле <iterator>. О том, как работают итераторы, будет подробно рассказано далее в этой главе.

Поскольку итераторы имеют много общего с указателями, алгоритмы стандартной библиотеки записываются так, чтобы в их аргументах могли передаваться как указатели, так и итераторы. Из-за этого реализация copy() больше похожа на следующий фрагмент:

```
template<typename Iterator>
void copy(Iterator begin, Iterator end, Iterator dest) {
  while (begin != end)
    *begin++ = *dest++;
}
```

Какой бы тип аргумента не использовался при вызове, алгоритм copy() предполагает, что он правильно реализует операторы * и ++. Если это условие не выполняется, происходит ошибка компиляции.

Предикаты

Еще одна распространенная задача — копирование точно определенного подмножества одного интервала (например, элементов, удовлетворяющих некоторому условию) в другой интервал. Для этого во многих алгоритмах предусмотрены альтернативные варианты вызова с возможностью передачи *предиката* (то есть обычной функции, возвращающей логическое значение по некоторому критерию). Допустим, вы хотите выделить в интервале целых чисел только те числа, которые меньше либо равны 15. Задача легко решается разновидностью алгоритма copy(), которая называется remove_copy_if():

```
//: C06:CopyInts2.cpp
// При копировании пропускаются числа, соответствующие предикату
#include <algorithm>
#include <cstddef>
#include <iostream>
using namespace std;

// Предикат определяется пользователем
bool gt15(int x) { return 15 < x; }

int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  int b[SIZE];
  int* endb = remove_copy_if(a, a+SIZE, b, gt15);
  int* beginb = b;
  while (beginb != endb)
    cout << *beginb++ << endl; // Выводит только 10
} ///:~
```

Шаблон функции remove_copy_if() получает стандартную пару указателей, определяющих границы интервала, за которыми следует предикат по выбору пользователя. Предикат передается в виде указателя на функцию[1]. Эта функция получает один аргумент, тип которого соответствует типу элементов интервала, и возвра-

[1] Или объекта, который может «вызываться» как функция. Об этом речь пойдет далее.

щает bool. В нашем примере функция gt15 возвращает true, если ее аргумент больше 15. Алгоритм remove_copy_if() применяет gt15() к каждому элементу исходного интервала и игнорирует элементы, для которых предикат возвращает true, при записи в приемный (выходной) интервал.

Следующая программа демонстрирует другую разновидность алгоритма copy:

```
//: C06:CopyStrings2.cpp
// Замена строк, соответствующих предикату
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

// Предикат
bool contains_e(const string& s) {
  return s.find('e') != string::npos;
}

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  string b[SIZE];
  string* endb = replace_copy_if(a, a + SIZE, b,
                    contains_e, string("kiss"));
  string* beginb = b;
  while (beginb != endb)
    cout << *beginb++ << endl;
} ///:~
```

При заполнении выходного интервала алгоритм replace_copy_if() заменяет элементы, не соответствующие заданному критерию, фиксированными значениями. Поскольку исходная строка «read» (единственный элемент, содержащий букву «e») заменяется словом «kiss» согласно последнему аргументу replace_copy_if(), результат выглядит так:

```
kiss
my
lips
```

Алгоритм replace_if() изменяет исходный интервал «на месте» вместо того, чтобы записывать его в отдельный выходной интервал:

```
//: C06:ReplaceStrings.cpp
// Замена строк "на месте"
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

bool contains_e(const string& s) {
  return s.find('e') != string::npos;
}

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  replace_if(a, a + SIZE, contains_e, string("kiss"));
```

```
  string* p = a;
  while (p != a + SIZE)
    cout << *p++ << endl;
} ///:~
```

Потоковые итераторы

Стандартная библиотека C++, как и любая нормальная библиотека, пытается предоставить удобные средства, позволяющие автоматизировать решения типичных задач. В начале главы мы уже упоминали о том, что циклические конструкции могут заменяться обобщенными алгоритмами. Тем не менее, до сих пор в наших примерах результаты выводились в цикле. Поскольку вывод является одной из самых типичных задач, логично предположить, что его тоже можно как-то автоматизировать.

На помощь приходят *потоковые итераторы*, которые в качестве исходного или приемного интервала используют потоки ввода-вывода. Например, чтобы задействовать потоковый итератор для исключения цикла вывода из программы CopyInt2.cpp, можно поступить так:

```
//: C06:CopyInts3.cpp
// Использование потокового итератора при выводе
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <iterator>
using namespace std;

bool gt15(int x) { return 15 < x; }

int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(cout, "\n"), gt15);
} ///:~
```

В этом примере приемный интервал b в третьем аргументе алгоритма remove_copy_if() заменяется *потоковым итератором вывода*. Он представляет собой специализацию шаблона класса ostream_iterator, объявленного в заголовочном файле <iterator>. Этот конкретный экземпляр ostream_iterator связывается со стандартным потоком вывода cout. Каждый раз, когда remove_copy_if() выдает в cout целое число из интервала a через этот итератор, последний записывает в cout целое число и строку-разделитель, переданную во втором аргументе (в нашем примере это символ перевода строки).

Так же просто данные можно записать в файл; для этого вместо cout передается файловый поток вывода:

```
//: C06:CopyIntsToFile.cpp
// Использование потокового итератора вывода с файлом
#include <algorithm>
#include <cstddef>
#include <fstream>
#include <iterator>
using namespace std;

bool gt15(int x) {
```

```
  return 15 < x;
}

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  ofstream outf("ints.out");
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(outf, "\n"), gt15);
} ///:~
```

Потоковый итератор ввода позволяет алгоритму прочитать исходные данные из потока ввода. Для этого конструктор и операторная функция operator++() читают следующий элемент из базового потока, а перегруженная операторная функция operator*() возвращает ранее прочитанное значение. Поскольку исходный интервал в алгоритмах задается двумя указателями, объект istream_iterator можно сконструировать двумя способами:

```
//: C06:CopyIntsFromFile.cpp
// Использование потокового итератора ввода
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
using namespace std;

bool gt15(int x) {
  return 15 < x;
}

int main() {
  ofstream inf("someInts.dat");
  inf << "1 3 47 5 84 9";
  inf.close();
  ifstream inf("someInts.dat");
  assure(inf, "someInts.dat");
  remove_copy_if(istream_iterator<int>(inf),
                 istream_iterator<int>(),
                 ostream_iterator<int>(cout, "\n"), gt15);
} ///:~
```

Первый аргумент replace_copy_if() связывает объект istream_iterator с файловым потоком ввода, содержащим данные типа int. Второй аргумент использует конструктор по умолчанию класса istream_iterator. Этот вызов конструирует istream_iterator() в особом состоянии, обозначающем конец файла. Таким образом, когда первый итератор достигает конца физического файла, он становится равным значению istream_iterator<int>(), и работа алгоритма корректно завершается. Обратите внимание: в этом примере мы полностью обошлись без явного объявления массива.

Сложность алгоритмов

Использование программной библиотеки основано на доверии. Пользователю приходится верить не только в то, что библиотека делает именно то, что нужно, но и в то, что функции работают с максимально возможной эффективностью. Лучше написать собственный цикл, чем использовать неэффективные алгоритмы.

Чтобы гарантировать качество реализаций библиотеки, стандарт C++ не только указывает, что должен делать алгоритм, но и с какой скоростью, а в отдельных случаях определяются даже допустимые затраты памяти. Любой алгоритм, не удовлетворяющий требованиям к эффективности, не соответствует стандарту. Мера эффективности алгоритмов называется *сложностью*.

Там, где это возможно, стандарт определяет примерное количество операций, выполняемых алгоритмом. Например, алгоритм count_if() возвращает количество элементов в интервале, удовлетворяющих заданному предикату. Следующий вызов count_if() для интервала целых чисел (вроде тех, что использовались в наших примерах) возвращает количество элементов, больших 15:

```
size_t n = count_if(a, a+SIZE, gt15);
```

Поскольку алгоритм count_if() должен просмотреть каждый элемент ровно один раз, стандарт указывает, что количество операций сравнения должно соответствовать количеству элементов в интервале. Аналогичная спецификация установлена для алгоритма copy().

Для других алгоритмов может быть установлено максимально допустимое количество операций. Так, алгоритм find() последовательно перебирает интервал до тех пор, пока не обнаружит элемент, равный третьему аргументу:

```
int* p = find(a, a + SIZE, 20);
```

Как только искомый элемент будет обнаружен, алгоритм прекращает поиск и возвращает указатель на первый найденный экземпляр. Если поиск оказывается безуспешным, алгоритм возвращает указатель, установленный в следующую позицию за концом интервала (a+SIZE в данном случае). Следовательно, количество сравнений, выполняемых find(), не превышает количества элементов в интервале.

Иногда количество необходимых операций не удается определить с подобной точностью. В таких случаях стандарт устанавливает *асимптотическую сложность* алгоритма, которая определяет, насколько рабочие характеристики алгоритма при очень больших интервалах соответствуют хорошо известным формулам. В качестве примера можно назвать алгоритм sort(), для которого стандарт устанавливает «...в среднем приблизительно n log n сравнений» (где n — количество элементов в интервале[1]).

Такая оценка сложности дает некоторое представление о затратах времени на работу алгоритма, и по крайней мере позволяет осмысленно сравнивать алгоритмы. Как будет показано в следующей главе, функция find() контейнера set обладает логарифмической сложностью, то есть затраты времени на поиск элемента в контейнере set при большом объеме данных пропорциональны количеству элементов. При больших значениях n это гораздо меньше количества элементов, поэтому для поиска в контейнере set вместо обобщенного алгоритма find() всегда следует применять функцию класса этого контейнера.

Объекты функций

В приводившихся примерах неоднократно встречалась функция gt15(). Как нетрудно заметить, польза от такой функции весьма невелика. Если порог сравнения бу-

[1] Или в математической записи — $O(n \log n)$. Это означает, что при больших значениях n количество сравнений растет прямо пропорционально функции $f(n) = n \log n$.

дет определяться другой величиной, придется определять функции gt20(), gt25() и т. д. Определять несколько одинаковых функций не только скучно, но и неразумно, потому что все необходимые значения известны на момент написания программы.

Более того, из этого следует, что для управления поиском не могут использоваться данные времени выполнения[1], а такое ограничение неприемлемо. Для решения проблемы необходим способ передачи информации предикатам во время выполнения программы. Например, было бы удобно определить функцию «больше», которая инициализируется произвольным значением. К сожалению, это значение не может передаваться при вызове, поскольку унарные предикаты (к числу которых относится наша функция gt15()) применяются к каждому элементу интервала по отдельности и получают только один параметр.

Как обычно, выход из положения основан на создании абстракции. Здесь нужна абстракция, которая бы работала, как функция, хранила информацию состояния и вызывалась с нужным количеством параметров. Такая абстракция называется *объектом функции*[2].

Объект функции представляет собой экземпляр класса с перегруженным оператором вызова функции (). Этот оператор позволяет использовать объект в традиционном синтаксисе вызова функции. Как и любой другой объект, объект функции инициализируется конструкторами. Ниже приведен объект функции, который может использоваться вместо gt15():

```
//: C06:GreaterThanN.cpp
#include <iostream>
using namespace std;

class gt_n {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) {
    return n > value;
  }
};

int main() {
  gt_n f(4);
  cout << f(3) << endl;  // Выводит 0 (для false)
  cout << f(5) << endl;  // Выводит 1 (для true)
} ///:~
```

Фиксированная величина для сравнения (4) передается при создании объекта функции. Затем компилятор интерпретирует выражение f(3) как вызов следующей функции:

```
f.operator()(3);
```

Команда возвращает значение выражения 3 > value, ложное при value = 4, как в нашем примере.

Поскольку такие сравнения применимы и к другим типам, помимо int, было бы логично определить gt_n() в виде шаблона класса. Впрочем, вам не придется делать

---

[1] Разве что с нежелательным применением глобальных переменных.

[2] Объекты функций также называются *функторами* (по названию математической концепции, обладающей сходным поведением).

это самостоятельно — в стандартной библиотеке это уже сделано за вас. Следующее описание объектов функций не только улучшит ваше представление о них, но и поможет лучше разобраться в том, как работают обобщенные алгоритмы.

Классификация объектов функций

В стандартной библиотеке C++ объекты функций классифицируются по количеству аргументов, передаваемых оператору () (ноль, один или два), и возвращаемому значению.

- *Генератор* — категория объектов функций, вызываемых без аргументов и возвращающих значение произвольного типа. Примером генератора может служить генератор случайных чисел. В стандартную библиотеку входит один генератор (функция rand(), объявленная в файле <cstdlib>), а также некоторые алгоритмы (например, generate_n()), применяющие генераторы к элементам интервалов.
- *Унарная функция* — категория объектов функций, вызываемых с одним аргументом произвольного типа и возвращающих значение, которое может относиться к другому типу (в том числе void).
- *Бинарная функция* — категория объектов функций, вызываемых с двумя аргументами любых двух (в том числе различных) типов и возвращающих значение произвольного типа (в том числе void).
- *Унарный предикат* — унарная функция, возвращающая bool.
- *Бинарный предикат* — бинарная функция, возвращающая bool.
- *Строгая квазиупорядоченность* — бинарный предикат, обеспечивающий расширенную интерпретацию понятия «равенство». Некоторые стандартные контейнеры считают два элемента равными (или эквивалентными), если ни один из них не меньше другого (по критерию <). Это существенно при сравнении вещественных чисел и объектов других типов, когда оператор == ненадежен или недоступен. Понятие строгой квазиупорядоченности также применяется при сортировке структур (записей данных) по подмножеству полей структуры. Две структуры с равными ключами не равны как объекты в целом, но они считаются равными в контексте критерия сравнения. Важность концепции строгой квазиупорядоченности станет очевидна в следующей главе.

Кроме того, некоторые алгоритмы подразумевают определенные предположения относительно того, какие операции поддерживаются для типов обрабатываемых объектов. Для обозначения этих предположений будут использоваться следующие термины:

- *с поддержкой* < — класс поддерживает оператор «меньше» <;
- *с поддержкой присваивания* — класс поддерживает оператор копирующего присваивания = для своего типа;
- *с поддержкой* == — класс поддерживает оператор проверки равенства == для своего типа[1].

[1] В сигнатурах будут использоваться обозначения LessThanComparable, Assignable и EqualityComparable. — *Примеч. перев.*

В данной главе эти термины будут использоваться для описания обобщенных алгоритмов стандартной библиотеки.

Автоматическое создание объектов функций

В заголовочном файле <functional> определяется целый ряд полезных объектов функций общего назначения. Сами по себе такие объекты достаточно просты, но они могут использоваться для построения более сложных объектов функций. Нередко вместо того, чтобы самостоятельно писать сложные предикаты, программист конструирует их из готовых «строительных блоков». Задача решается при помощи *адаптеров объектов функций*, которые получают простые объекты функций и комбинируют их с другими объектами функций в цепочке операций.

Чтобы пояснить сказанное, давайте с использованием только стандартных объектов функций сделаем то, что делалось раньше функцией gt15(). Стандартный бинарный объект функции greater возвращает true, если его первый аргумент меньше второго. Однако его не удастся напрямую применить к интервалам целых чисел при помощи алгоритма вроде remove_copy_if(), поскольку алгоритм remove_copy_if() должен получать *унарный* предикат. Проблема легко решается — мы конструируем унарный предикат, использующий greater для сравнения своего первого аргумента *с фиксированной величиной*. Значение второго параметра фиксируется равным 15 при помощи адаптера bind2nd, как показано ниже:

```
//: C06:CopyInts4.cpp
// Использование стандартного объекта функции и адаптера
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
#include <iterator>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(cout, "\n"),
                 bind2nd(greater<int>(), 15));
} ///:~
```

Программа выводит тот же результат, что и CopyInts3.cpp, но нам не потребовалось писать собственную функцию-предикат gt15(). Адаптер bind2nd() представляет собой шаблонную функцию, которая создает объект функции типа binder2nd. Этот объект просто сохраняет два аргумента, переданных bind2nd(); первый аргумент должен быть бинарной функцией или объектом функции (то есть чем угодно, что может вызываться с двумя аргументами). Функция operator() в binder2nd сама по себе является унарной, но она вызывает хранящуюся в ней бинарную функцию и передает ей параметр и фиксированное значение.

Рассмотрим конкретный пример. Допустим, экземпляр binder2nd, созданный bind2nd(), называется b. При создании b получает два параметра (greater<int> и 15) и сохраняет их. Для удобства обозначим экземпляр greater<int> именем g, а экземпляр потокового итератора вывода — o. Тогда приведенный выше вызов remove_copy_if() концептуально выглядит так:

```
remove_copy_if(a, a+SIZE, o, b(g, 15).operator());
```

В процессе перебора интервала алгоритм remove_copy_if() вызывает b для каждого элемента; в зависимости от результата вызова он решает, нужно ли игнорировать элемент при копировании в приемник. Если обозначить текущий элемент именем e, этот вызов внутри remove_copy_if() эквивалентен следующему:

```
if(b(e))
```

Однако оператор вызова функции binder2nd просто вызывает g(e,15), поэтому приведенная команда эквивалентна такой:

```
if(greater<int>(e,15))
```

А это и есть нужное сравнение. Также существует адаптер bind1st для создания объекта binder1st, фиксирующего *первый* аргумент бинарной функции.

Еще один пример: подсчет количества элементов интервала, отличных от 20. На этот раз будет использоваться алгоритм count_if(), упоминавшийся ранее. В библиотеке определен стандартный объект бинарной функции equal_to и адаптер not1(), который получает объект унарной функции и возвращает его логическое отрицание. Следующая программа делает то, что требуется:

```
//: C06:CountNotEqual.cpp
// Подсчет элементов, не равных 20
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
using namespace std;

int main() {
  int a[] = { 10, 20, 30 };
  const size_t SIZE = sizeof a / sizeof a[0];
  cout << count_if(a, a + SIZE,
                   not1(bind1st(equal_to<int>(), 20)));// 2
} ///:~
```

Алгоритм count_if() вызывает предикат, переданный в третьем аргументе (назовем его n), для каждого элемента своего интервала. Каждый раз, когда предикат возвращает true, алгоритм увеличивает свой внутренний счетчик. По аналогии с предыдущим примером обозначим текущий элемент символом e и рассмотрим команду

```
if (n(e))
```

В реализации count_if эта команда интерпретируется так:

```
if(!(bind1st(equal_to<int>, 20)(e))
```

В конечном счете выражение принимает следующий вид, так как адаптер not1() возвращает логическое отрицание своего аргумента (результата вызова унарной функции):

```
if(!equal_to<int>(20, e))
```

Первый аргумент equal_to равен 20, потому что вместо bind2nd() используется адаптер bind1st(). Проверка на равенство симметрична по отношению к своим аргументам; следовательно, в нашем примере можно было задействовать как bind1st(), так и bind2nd().

В табл. 6.1 перечислены шаблоны, генерирующие стандартные объекты функций (с указанием контекста применения).

Таблица 6.1. Шаблоны, генерирующие стандартные объекты функций

Имя	Тип	Результат
plus	Бинарная функция	arg1 + arg2
minus	Бинарная функция	arg1 – arg2
multiplies	Бинарная функция	arg1 * arg2
divides	Бинарная функция	arg1/arg2
modulus	Бинарная функция	arg1%arg2
negate	Унарная функция	–arg
equal_to	Бинарный предикат	arg1 == arg2
not_equal_to	Бинарный предикат	arg1 != arg2
greater	Бинарный предикат	arg1 > arg2
less	Бинарный предикат	arg1 < arg2
greater_equal	Бинарный предикат	arg1 >= arg2
less_equal	Бинарный предикат	arg1 <= arg2
logical_and	Бинарный предикат	arg1 && arg2
logical_or	Бинарный предикат	arg1 || arg2
logical_not	Унарный предикат	!arg1
unary_negate	Унарная логическая функция	!(унарный_предикат(arg1))
binary_negate	Бинарная логическая функция	!(бинарный_предикат(arg1, arg2)

Адаптируемые объекты функций

Стандартные адаптеры (такие как bind1st() и bind2nd()) делают определенные допущения относительно обрабатываемых объектов функций. Вернемся к выражению из последней строки программы CountNotEqual.cpp:

```
not1(bind1st(equal_to<int>(), 20))
```

Адаптер bind1st() создает унарный объект функции типа binder1st, который просто сохраняет экземпляр equal_to<int> и значение 20. Однако функция binder1st::operator() должна знать тип своего аргумента и возвращаемого значения, иначе она не может иметь нормального объявления. Как решается эта проблема? Предполагается, что каждый объект функции предоставляет вложенные определения для этих типов. Для унарных объектов функций используются имена типов argument_type и result_type, а для бинарных объектов функций — имена first_argument_type, second_argument_type и result_type. В этом нетрудно убедиться, просматривая реализацию bind1st() и binder1st в заголовке <functional>. Начнем с bind1st() в типичной реализации библиотеки:

```
template <class Op, class T>
binder1st<Op> bind1st(const Op& f, const T& val) {
  typedef typename Op::first_argument_type Arg1_t;
  return binder1st<Op>(f, Arg1_t(val));
}
```

Параметр шаблона Op, представляющий тип бинарной функции, адаптируемой адаптером bind1st(), должен содержать вложенное определение типа с именем first_argument_type (обратите также внимание на ключевое слово typename, которое, как отмечалось в главе 5, сообщает компилятору, что речь идет об имени вложенного *типа*). Теперь посмотрим, как binder1st использует имена типов Op при объявлении своего оператора вызова функции:

```
// Внутри реализации binder1st<Op>
typename Op::result_type
operator()(const typename Op::second_argument_type& x)
  const;
```

Объекты функций, классы которых предоставляют эти имена типов, называются *адаптируемыми объектами функций*.

Поскольку указанные имена должны поддерживаться всеми стандартными объектами функций, а также любыми объектами функций, которые должны использоваться с адаптерами, в заголовочный файл <functional> входят два шаблона, автоматически определяющие эти имена: unary_function и binary_function. Вы просто наследуете от этих классов при заполнении типов аргументов. Допустим, вы хотите сделать адаптируемым объект функции gt_n, определение которого приводилось выше. Для этого достаточно сделать следующее:

```
class gt_n : public unary_function<int, bool> {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) {
    return n > value;
  }
};
```

Как видно из определения unary_function, этот шаблон просто определяет нужные типы, взятые из своих параметров шаблонов:

```
template <class Arg, class Result> struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
};
```

Эти типы доступны через класс gt_n, поскольку класс открыто наследует от unary_function. Шаблон binary_function работает аналогично.

Другие примеры объектов функций

Следующий пример, FunctionObjects.cpp, содержит простые тесты для большинства стандартных шаблонов объектов функций. Он показывает, как использовать каждый шаблон и как он себя ведет. Для удобства в программе задействованы следующие генераторы:

```
//: C06:Generators.h
// Различные способы заполнения интервалов
#ifndef GENERATORS_H
#define GENERATORS_H
#include <cstring>
#include <set>
#include <cstdlib>

// Генератор, пропускающий числа:
class SkipGen {
  int i;
  int skp;
public:
  SkipGen(int start = 0, int skip = 1)
    : i(start), skp(skip) {}
  int operator()() {
```

```
    int r = i;
    i += skp;
    return r;
  }
};

// Генератор, выдающий уникальные случайные числа
// в интервале от 0 до mod:
class URandGen {
  std::set<int> used;
  int limit;
public:
  URandGen(int lim) : limit(lim) {}
  int operator()() {
    while(true) {
      int i = int(std::rand()) % limit;
      if(used.find(i) == used.end()) {
        used.insert(i);
        return i;
      }
    }
  }
};

// Генератор случайных символов:
class CharGen {
  static const char* source;
  static const int len;
public:
  char operator()() {
    return source[std::rand() % len];
  }
};

//: C06:Generators.cpp {O}
#include "Generators.h"
const char* CharGen::source = "ABCDEFGHIJK"
  "LMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
const int CharGen::len = strlen(source);
///:~
```

Эти функции-генераторы используются во многих примерах этой главы. Объект функции SkipGen возвращает следующее число математической прогрессии, разность которой хранится в переменной skp. Объект URandGen генерирует уникальное случайное число в заданном интервале (в нем используется контейнер set, описанный в следующей главе). Объект CharGen возвращает случайный алфавитный символ. Пример программы с генератором URandGen:

```
//: C06:FunctionObjects.cpp {-bor}
// Демонстрация некоторых стандартных объектов функций
// из стандартной библиотеки C++
//{L} Generators
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
```

```
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

template<typename Contain, typename UnaryFunc>
void testUnary(Contain& source, Contain& dest,
  UnaryFunc f) {
  transform(source.begin(), source.end(), dest.begin(), f);
}

template<typename Contain1, typename Contain2,
  typename BinaryFunc>
void testBinary(Contain1& src1, Contain1& src2,
  Contain2& dest, BinaryFunc f) {
  transform(src1.begin(), src1.end(),
    src2.begin(), dest.begin(), f);
}

// Макрос выполняет выражение, а затем выводит его
// в строковом виде при помощи print
#define T(EXPR) EXPR; print(r.begin(), r.end(), \
  "After " #EXPR);
// Для логических проверок:
#define B(EXPR) EXPR; print(br.begin(), br.end(), \
  "After " #EXPR);

// Генератор случайных логических величин:
struct BRand {
  bool operator()() { return rand() % 2 == 0; }
};

int main() {
  const int SZ = 10;
  const int MAX = 50;
  vector<int> x(SZ), y(SZ), r(SZ);
  // Генератор целых случайных чисел:
  URandGen urg(max);
  srand(time(0));  // Раскрутка генератора
  generate_n(x.begin(), SZ, urg);
  generate_n(y.begin(), SZ, urg);
  // Прибавление единицы предотвращает деление на ноль:
  transform(y.begin(), y.end(), y.begin(),
    bind2nd(plus<int>(), 1));
  // Гарантированное совпадение одной пары элементов:
  x[0] = y[0];
  print(x.begin(), x.end(), "x");
  print(y.begin(), y.end(), "y");
  // Выполнение операции с каждой парой элементов x и y
  // с сохранением результата в r:
  T(testBinary(x, y, r, plus<int>()));
  T(testBinary(x, y, r, minus<int>()));
  T(testBinary(x, y, r, multiplies<int>()));
  T(testBinary(x, y, r, divides<int>()));
  T(testBinary(x, y, r, modulus<int>()));
  T(testUnary(x, r, negate<int>()));
  vector<bool> br(SZ); // Для логических результатов
  B(testBinary(x, y, br, equal_to<int>()));
  B(testBinary(x, y, br, not_equal_to<int>()));
  B(testBinary(x, y, br, greater<int>()));
```

```
  B(testBinary(x, y, br, less<int>()));
  B(testBinary(x, y, br, greater_equal<int>()));
  B(testBinary(x, y, br, less_equal<int>()));
  B(testBinary(x, y, br, not2(greater_equal<int>())));
  B(testBinary(x,y,br,not2(less_equal<int>())));
  vector<bool> b1(SZ), b2(SZ);
  generate_n(b1.begin(), SZ, BRand());
  generate_n(b2.begin(), SZ, BRand());
  print(b1.begin(), b1.end(), "b1");
  print(b2.begin(), b2.end(), "b2");
  B(testBinary(b1, b2, br, logical_and<int>()));
  B(testBinary(b1, b2, br, logical_or<int>()));
  B(testUnary(b1, br, logical_not<int>()));
  B(testUnary(b1, br, not1(logical_not<int>())));
} ///:~
```

В этом примере используется удобный шаблон функции print(), предназначенный для вывода произвольного интервала с необязательным сообщением. Он определяется в заголовочном файле PrintSequences.h, описанном далее в этой главе.

Две шаблонные функции автоматизируют процесс тестирования различных шаблонов объектов функций. Две — потому что объекты функций могут быть унарными или бинарными. Функция testUnary() получает исходный вектор, приемный вектор и объект унарной функции, который применяется к исходному вектору для получения приемного вектора. В testBinary() два исходных вектора передаются бинарной функции для получения приемного вектора. В обоих случаях шаблонные функции просто вызывают алгоритм transform(), который применяет унарную функцию или объект функции (четвертый параметр) к каждому элементу интервала. Результат записывается в интервал, определяемый третьим параметром; в данном случае он совпадает с исходным.

Для каждого теста выводится строка с кратким описанием и результаты. Препроцессор помогает автоматизировать процесс вывода; макросы T() и B() получают выполняемое выражение. После вычисления интервал передается функции print(). При построении сообщения выражение «преобразуется в строку» средствами препроцессора. Это позволяет видеть код выражения, за которым следует полученный вектор.

Последний вспомогательный инструмент BRand представляет собой объект-генератор, выдающий случайные логические (bool) значения. Для этого он генерирует случайное число функцией rand() и проверяет, больше ли оно (RANDMAX + 1)/2. При равномерном распределении случайных чисел это условие выполняется в половине случаев.

В функции main() создаются три вектора типа int: x и y для исходных значений и r для результатов. Для инициализации x и y случайными значениями, не превышающими 50, используется генератор типа URandGen из Generators.h. Стандартный алгоритм generate_n() заполняет интервал, заданный первым аргументом, вызывая свой третий аргумент (который должен быть генератором) заданное количество раз (определяется вторым аргументом). Поскольку в одной из операций элементы x делятся на элементы y, необходимо позаботиться о том, чтобы в y не было нулевых элементов. Задача также решается при помощи алгоритма transform(); исходные данные берутся из y, увеличиваются на 1, а результат снова записывается в y. Объект функции для выполнения этой операции выглядит так:

```
bind2nd(plus<int>(), 1)
```

В этом выражении объект функции plus используется для увеличения первого аргумента на 1. Как и прежде, мы используем адаптер для превращения бинарной функции в унарную, чтобы ее можно было применить к целому интервалу одним вызовом transform().

При другой проверке в программе элементы двух векторов сравниваются на равенство. Чтобы результат был более содержательным, по крайней мере одна пара элементов заведомо должна совпадать; в нашем примере выбирается элемент 0.

После вывода двух векторов все объекты функций, выдающие числовые значения, тестируются макросом T(), а все объекты функций с логическими значениями тестируются макросом B(). Результат помещается в vector<bool>. При выводе этого вектора вместо элементов true выводятся единицы, а вместо false — нули. Ниже приводится результат выполнения программы FunctionObjects.cpp:

```
x:
4 8 18 36 22 6 29 19 25 47
y:
4 14 23 9 11 32 13 15 44 30
After testBinary(x, y, r, plus<int>()):
B 22 41 45 33 38 42 34 69 77
After testBinary(x, y, r, minus<int>()):
0 -6 -5 27 11 -26 16 4 -19 17
After testBinary(x, y, r, multiplies<int>()):
16 112 414 324 242 192 377 285 1100 1410
After testBinary(x, y, r, divides<int>()):
1 0 0 4 2 0 2 1 0 1
After testBinary(x, y, r, limit<int>()):
0 8 18 0 0 6 3 4 25 17
After testUnary(x, r, negate<int>()):
-4 -8 -18 -36 -22 -6 -29 -19 -25 -47
After testBinary(x, y, br, equal_to<int>()):
1 0 0 0 0 0 0 0 0 0
After testBinary(x, y, br, not_equal_to<int>()):
0 1 1 1 1 1 1 1 1 1
After testBinary(x, y, br, greater<int>()):
0 0 0 1 1 0 1 1 0 1
After testBinary(x, y, br, less<int>()):
0 1 1 0 0 1 0 0 1 0
After testBinary(x, y, br, greater_equal<int>()):
1 0 0 1 1 0 1 1 0 1
After testBinary(x, y, br, less_equal<int>()):
1 1 1 0 0 1 0 0 1 0
After testBinary(x, y, br, not2(greater_equal<int>())):
0 1 1 0 0 1 0 0 1 0
After testBinary(x,y,br,not2(less_equal<int>())):
0 0 0 1 1 0 1 1 0 1
b1:
0 1 1 0 0 0 1 0 1 1
b2:
0 1 1 0 0 0 1 0 1 1
After testBinary(b1, b2, br, logical_and<int>()):
0 1 1 0 0 0 1 0 1 1
After testBinary(b1, b2, br, logical_or<int>()):
0 1 1 0 0 0 1 0 1 1
After testUnary(b1, br, logical_not<int>()):
1 0 0 1 1 1 0 1 0 0
After testUnary(b1, br, not1(logical_not<int>())):
0 1 1 0 0 0 1 0 1 1
```

Если вы хотите, чтобы логические данные выводились в виде значений true и false вместо 1 и 0, вызовите cout.set(ios::boolalpha).

Адаптеры привязки не обязаны создавать унарный *предикат*; кроме того, они могут создавать любые унарные *функции* (то есть функции с типом возвращаемого значения, отличным от bool). Например, можно умножить каждый элемент вектора на 10, используя адаптер привязки в сочетании с алгоритмом transform():

```
//: C06:FBinder.cpp
// Адаптеры привязки не ограничиваются построением предикатов
//{L} Generators
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
#include "Generators.h"
using namespace std;

int main() {
  ostream_iterator<int> out(cout," ");
  vector<int> v(15);
  srand(time(0));  // Раскрутка генератора случайных чисел
  generate(v.begin(), v.end(), URandGen(20));
  copy(v.begin(), v.end(), out);
  transform(v.begin(), v.end(), v.begin(),
            bind2nd(multiplies<int>(), 10));
  copy(v.begin(), v.end(), out);
} ///:~
```

Третий аргумент transform() равен первому, поэтому итоговые элементы копируются обратно в исходный вектор. Объект функции, созданный адаптером bind2nd(), в данном примере генерирует результат типа int.

Адаптер привязки не позволяет зафиксировать объект функции, однако фиксируемый аргумент не обязан быть константой времени компиляции. Пример:

```
//: C06:BinderValue.cpp
// Определение фиксируемого аргумента на стадии выполнения
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <cstdlib>
using namespace std;

int boundedRand() { return rand() % 100; }

int main() {
  const int SZ = 20;
  int a[SZ], b[SZ] = {0};
  generate(a, a + SZ, boundedRand);
  int val = boundedRand();
  int* end = remove_copy_if(a, a + SZ, b,
                            bind2nd(greater<int>(), val));
  // Сортировка для упрощения просмотра:
  sort(a, a + SZ);
  sort(b, end);
```

```
  ostream_iterator<int> out(cout, " ");
  cout << "Original Sequence:\n";
  copy(a, a + SZ, out); cout << endl;
  cout << "Values less <= " << val << endl;
  copy(b, end, out); cout << endl;
} ///:~
```

Мы заполняем массив 20 случайными числами от 0 до 100, после чего пользователь вводит пороговое значение в командной строке. В вызове remove_copy_if() фиксируемый аргумент bind2nd() представляет собой случайное число из исходного интервала. Вот как выглядит результат одного из запусков:

```
4 12 15 17 19 21 26 30 47 48 56 58 60 63 71 79 82 90 92 95
Values less <= 41
4 12 15 17 19 21 26 30
```

Адаптация указателей на функции

Алгоритмам, получающим функции при вызове, может передаваться как указатель на обычную функцию, так и указатель на объект функции. В первом случае алгоритм использует традиционный механизм вызова, а во втором вызывается операторная функция operator() этого объекта. Так, в примере CopyInts2.cpp в качестве предиката для вызова remove_copy_if() передавалась функция gt15(). Кроме того, указатели на функции, возвращающие случайные числа, передавались алгоритмам generate() и generate_n().

Однако простые функции не могут использоваться с адаптерами объектов функций (такими, как bind2nd()), поскольку последние предполагают существование определений типов для аргументов и возвращаемого значения. Но вам не придется вручную преобразовывать функции в объекты функций — в стандартной библиотеке предусмотрены адаптеры для выполнения этой работы. Адаптер ptr_fun() получает указатель на функцию и преобразует его в объект функции. Он не рассчитан на работу с функциями без аргументов и может использоваться только с унарными и бинарными функциями.

В следующей программе адаптер ptr_fun() преобразует унарную функцию в объект функции:

```
//: C06:PtrFun1.cpp
// Использование ptr_fun() с унарной функцией
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int d[] = { 123, 94, 10, 314, 315 };
const int DSZ = sizeof d / sizeof *d;
bool isEven(int x) { return x % 2 == 0; }

int main() {
  vector<bool> vb;
  transform(d, d + DSZ, back_inserter(vb),
    not1(ptr_fun(isEven)));
  copy(vb.begin(), vb.end(),
```

```
    ostream_iterator<bool>(cout, " "));
  cout << endl;
  // Результат: 1 0 0 0 1
} ///:~
```

Мы не можем просто передать адаптеру **not1** функцию **isEven**, потому что адаптеру необходимо знать фактический тип аргумента и возвращаемого значения. Адаптер **ptr_fun()** определяет эти типы по аргументам шаблона. Определение унарной версии **ptr_fun()** выглядит примерно так:

```
template <class Arg, class Result>
pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*fptr)(Arg)) {
  return pointer_to_unary_function<Arg, Result>(fptr);
}
```

Как видите, эта версия **ptr_fun()** определяет типы аргумента и результата по **fptr** и использует их для инициализации объекта **pointer_to_unary_function**, в котором хранится **fptr**. Оператор вызова функции объекта **pointer_to_unary_function** просто вызывает **fptr** в последней строке своей реализации:

```
template <class Arg, class Result>
class pointer_to_unary_function
: public unary_function<Arg, Result> {
  Result (*fptr)(Arg); // Сохраняет указатель на функцию
public:
  pointer_to_unary_function(Result (*x)(Arg)) : fptr(x) {}
  Result operator()(Arg x) const { return fptr(x); }
};
```

Поскольку объект **pointer_to_unary_function** объявлен производным от **unary_function**, унаследованные определения типов становятся доступными для адаптера **not1**.

Также существует бинарная версия **ptr_fun()**, которая возвращает объект **pointer_to_binary_function** (производный от **binary_function**). Этот объект ведет себя так же, как его унарный аналог. В следующей программе бинарная версия **ptr_fun()** используется для возведения элементов числового интервала в степень. Кроме того, этот пример демонстрирует потенциальную проблему с передачей адаптеру **ptr_fun()** перегруженных функций.

```
//: C06:PtrFun2.cpp
// Использование ptr_fun() с бинарными функциями
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

double d[] = { 01.23, 91.370, 56.661,
  023.230, 19.959, 1.0, 3.14159 };
const int DSZ = sizeof d / sizeof *d;

int main() {
  vector<double> vd;
  transform(d, d + DSZ, back_inserter(vd),
    bind2nd(ptr_fun<double, double, double>(pow), 2.0));
```

```
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, " "));
  cout << endl;
} ///:~
```

Функция pow() перегружена в стандартном заголовке C++ **<cmath>** для всех вещественных типов данных:

```
float pow(float, int); // Эффективные версии с целым показателем степени
double pow(double, int);
long double pow(long double, int);
float pow(float, float);
double pow(double, double);
long double pow(long double, long double);
```

Компилятор обнаруживает несколько версий pow() и не знает, какую из них нужно выбрать. Нам придется помочь ему и воспользоваться явной специализацией шаблонов функций, о которой рассказывалось в предыдущей главе[1].

Еще больше трудностей возникает с преобразованием функции класса в объект функции, подходящий для работы с обобщенными алгоритмами. Рассмотрим простой пример: имеется классическая иерархия геометрических фигур. Требуется вызвать функцию draw() для каждого указателя на Shape, хранящегося в контейнере:

```
//: C06:MemFun1.cpp
// Вызов функций класса по указателям
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  virtual void draw() { cout << "Circle::Draw()" << endl; }
  ~Circle() { cout << "Circle::~Circle()" << endl; }
};

class Square : public Shape {
public:
  virtual void draw() { cout << "Square::Draw()" << endl; }
  ~Square() { cout << "Square::~Square()" << endl; }
};

int main() {
  vector<Shape*> vs;
  vs.push_back(new Circle);
```

[1] Существует дополнительная трудность, связанная с различиями в реализации библиотек. Если для функции pow() используется компоновка C (то есть ее имя не «украшается», в отличие от функций C++), то этот пример компилироваться не будет. Адаптеру ptr_fun необходим указатель на обычную перегружаемую функцию C++.

```
  vs.push_back(new Square);
  for_each(vs.begin(), vs.end(), mem_fun(&Shape::draw));
  purge(vs);
} ///:~
```

Алгоритм for_each() передает каждый элемент интервала объекту функции, определяемому третьим аргументом. Здесь нам нужно, чтобы объект функции представлял одну из функций класса, поэтому «аргумент» объекта функции становится указателем на объект для вызова функции. Для получения такого объекта используется шаблон mem_fun(), в аргументе которого передается указатель на функцию класса.

Функция mem_fun() создает объекты функций; их вызов производится через указатель на объект, функция которого вызывается, тогда как при использовании mem_fun_ref() функция вызывается непосредственно для объекта. Один набор перегруженных версий mem_fun() и mem_fun_ref() предназначен для функций с нулем и с одним аргументом, причем их количество увеличивается вдвое для константных и неконстантных функций класса. Тем не менее, шаблоны и перегрузка избавляют вас от всех сложностей выбора нужной версии — вам остается лишь запомнить, в каких случаях используется mem_fun(), а в каких — mem_fun_ref().

Допустим, имеется контейнер объектов (не указателей!), и вы хотите вызвать функцию класса с аргументом, при этом передаваемый аргумент берется из другого контейнера. Для решения этой задачи используется вторая перегруженная форма алгоритма transform():

```
//: C06:MemFun2.cpp
// Вызов функций класса по ссылке на объект
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

class Angle {
  int degrees;
public:
  Angle(int deg) : degrees(deg) {}
  int mul(int times) { return degrees *= times; }
};

int main() {
  vector<Angle> va;
  for(int i = 0; i < 50; i += 10)
    va.push_back(Angle(i));
  int x[] = { 1, 2, 3, 4, 5 };
  transform(va.begin(), va.end(), x,
    ostream_iterator<int>(cout, " "),
    mem_fun_ref(&Angle::mul));
  cout << endl;
  // Результат: 0 20 60 120 200
} ///:~
```

Так как в контейнере хранятся объекты, с указателем на функцию класса должен использоваться адаптер mem_fun_ref(). Данная версия transform() получает начальную и конечную точки первого интервала (с объектами); начальную точку второго интервала (с аргументами функции), итератор приемника, которым в данном случае является стандартный выходной поток, и объект функции, вызывае-

мый для каждого элемента в первом интервале. Объект функции создается из адаптера mem_fun_ref() и указателя на функцию. Учтите, что для шаблонов transform() и for_each() установлены некоторые ограничения; transform() требует, чтобы вызываемая функция возвращала значение, а у for_each() не существует версии с передачей двух аргументов вызываемой функции. Следовательно, при использовании шаблона transform() или for_each() вам не удастся вызвать функцию класса, которая возвращает void и получает дополнительный аргумент.

Адаптер mem_fun_ref() может использоваться с функциями классов стандартной библиотеки, если ваш компилятор не добавляет аргументы по умолчанию помимо обычных аргументов, указанных в стандарте[1]. Предположим, требуется найти в читаемом файле пустые строки. Возможно, ваш компилятор позволит задействовать функцию string::empty() так, как показано в следующем примере:

```
//: C06:FindBlanks.cpp
// Использование mem_fun_ref() с string::empty()
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <fstream>
#include <functional>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

typedef vector<string>::iterator LSI;

int main(int argc, char* argv[]) {
  char* fname = "FindBlanks.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  vector<string> vs;
  string s;
  while(getline(in, s))
    vs.push_back(s);
  vector<string> cpy = vs; // Для тестирования
  LSI lsi = find_if(vs.begin(), vs.end(),
     mem_fun_ref(&string::empty));
  while(lsi != vs.end()) {
    *lsi = "A BLANK LINE";
    lsi = find_if(vs.begin(), vs.end(),
      mem_fun_ref(&string::empty));
  }
  for(size_t i = 0; i < cpy.size(); i++)
    if(cpy[i].size() == 0)
      assert(vs[i] == "A BLANK LINE");
    else
      assert(vs[i] != "A BLANK LINE");
} ///:~
```

[1] Если компилятор определяет string::empty с аргументами по умолчанию (что разрешено), то выражение &string::empty определяет указатель на функцию класса с максимальным числом аргументов. Поскольку компилятор не сможет предоставить дополнительные значения по умолчанию, при попытке применить string::empty через mem_fun_ref компилятор выдаст сообщение об отсутствии аргументов.

В этом примере алгоритм find_if() ищет первую пустую строку в заданном интервале, используя адаптер mem_fun_ref() с функцией string::empty(). После открытия файла и загрузки его содержимого в вектор этот процесс повторяется для всех пустых строк в файле. Найденные пустые строки заменяются символьной последовательностью "A BLANK LINE". Все, что для этого нужно, — разыменовать итератор для выборки текущей строки.

Написание пользовательских адаптеров для объектов функций

Попробуем написать программу, которая бы преобразовывала строковые представления вещественных чисел в их фактические числовые значения. Начнем с генератора строк:

```
//: C06:NumStringGen.h
// Генератор случайных вещественных чисел в строковом формате.
#ifndef NUMSTRINGGEN_H
#define NUMSTRINGGEN_H
#include <string>
#include <cstdlib>
#include <ctime>

class NumStringGen {
  const int sz; // Количество цифр
public:
  NumStringGen(int ssz = 5) : sz(ssz) {}
  std::string operator()() {
    std::string digits("0123456789");
    const int ndigits = digits.size();
    std::string r(sz, ' ');
    // Первая цифра должна быть отлична от нуля
    r[0] = digits[std::rand() % *(ndigits - 1)] + 1;
    // Заполнение остальных цифр
    for(int i = 0; i < sz; i++)
      if(sz >= 3 && i == sz/2)
        r[i] = '.'; // Вставка десятичной точки
      else
        r[i] = digits[std::rand() % ndigits];
    return r;
  }
};
#endif // NUMSTRINGGEN_H ///:~
```

Размер строк указывается при создании объекта NumStringGen. Генератор случайных чисел выбирает цифры, а десятичная точка вставляется в середину.

В следующей программе генератор NumStringGen служит для заполнения контейнера vector<string>. Но чтобы использовать стандартную библиотечную функцию atof() языка C для преобразования строк в целые числа, объекты string сначала приходится преобразовывать в char*, поскольку автоматическое преобразование типа из string в char* не поддерживается. Алгоритм transform() в сочетании с mem_fun_ref() и string::c_str() преобразует все объекты string в char*, после чего результаты преобразуются в вещественные числа функцией atof.

```
//: C06:MemFun3.cpp
// Использование mem_fun()
```

```
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

int main() {
  const int SZ = 9;
  vector<string> vs(SZ);
  // Заполнение вектора случайными строками:
  srand(time(0));   // Раскрутка генератора случайных чисел
  generate(vs.begin(), vs.end(), NumStringGen());
  copy(vs.begin(), vs.end(),
    ostream_iterator<string>(cout, "\t"));
  cout << endl;
  const char* vcp[SZ];
  transform(vs.begin(), vs.end(), vcp,
    mem_fun_ref(&string::c_str));
  vector<double> vd;
  transform(vcp, vcp + SZ, back_inserter(vd),
    std::atof);
  cout.precision(4);
  cout.setf(ios::showpoint);
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

Программа выполняет две трансформации. В ходе одной объекты string преобразуются в строки C (массивы символов), а в ходе другой строки C преобразуются в числа функцией atof(). Было бы неплохо объединить эти две операции. В конце концов, если композиция функций существует в математике, почему бы ей не поддерживаться в C++?

В наиболее очевидном решении шаблон композиции получает две функции в аргументах и применяет их в нужном порядке:

```
//: C06:ComposeTry.cpp
// Первая попытка реализовать композицию функций
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <string>
using namespace std;

template<typename R, typename E, typename F1, typename F2>
class unary_composer {
   F1 f1;
   F2 f2;
public:
   unary_composer(F1 fone, F2 ftwo) : f1(fone), f2(ftwo) {}
   R operator()(E x) {
      return f1(f2(x));
```

```
  }
};
template<typename R, typename E, typename F1, typename F2>
unary_composer<R, E, F1, F2> compose(F1 f1, F2 f2) {
  return unary_composer<R, E, F1, F2>(f1, f2);
}
int main()
{
  double x =
    compose<double, const string&>(atof,
      mem_fun_ref(&string::c_str))("12.34");
  assert(x == 12.34);
} ///:~
```

В объекте unary_composer сохраняются указатели на функции atof и string::c_str, причем последняя функция применяется первой при вызове operator(). Адаптер compose() упрощает вызов, чтобы нам не приходилось явно передавать все четыре аргумента шаблона — F1 и F2 определяются на основании вызова.

Однако было бы гораздо удобнее, если бы мы могли вообще обойтись без передачи аргументов шаблона. Чтобы это стало возможным, следует обеспечить наличие необходимых определений типов для адаптируемых объектов функций. Иначе говоря, мы будем предполагать, что функции, входящие в композицию, являются адаптируемыми. Для этого функция atof() должна использоваться с ptr_fun(). А чтобы добиться максимальной гибкости, мы также сделаем адаптируемым шаблон unary_composer на случай, если он будет передан адаптеру функции. Следующая программа использует этот подход и легко решает исходную задачу:

```
//: C06:ComposeFinal.cpp
// Адаптируемый шаблон композиции
#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

template<typename F1, typename F2>
class unary_composer
  : public unary_function<typename F2::argument_type,
                          typename F1::result_type> {
  F1 f1;
  F2 f2;
public:
  unary_composer(F1 f1, F2 f2) : f1(f1), f2(f2) {}
  typename F1::result_type
  operator()(typename F2::argument_type x) {
    return f1(f2(x));
  }
};

template<typename F1, typename F2>
unary_composer<F1, F2> compose(F1 f1, F2 f2) {
  return unary_composer<F1, F2>(f1, f2);
```

```
}

int main() {
  const int SZ = 9;
  vector<string> vs(SZ);
  // Заполнение вектора случайными строками:
  generate(vs.begin(), vs.end(), NumStringGen());
  copy(vs.begin(), vs.end(),
    ostream_iterator<string>(cout, "\t"));
  cout << endl;
  vector<double> vd;
  transform(vs.begin(), vs.end(), back_inserter(vd),
    compose(ptr_fun(atof), mem_fun_ref(&string::c_str)));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

Как и прежде, ключевое слово **typename** сообщает компилятору, что указанный член класса является вложенным типом.

В некоторых реализациях[1] композиция объектов функций поддерживается как расширение. Вероятно, комитет по стандартизации C++ включит поддержку этих возможностей в следующую версию стандарта C++.

Каталог алгоритмов STL

Краткая сводка, приведенная в этом разделе, поможет вам найти нужный алгоритм. За полными описаниями алгоритмов STL, информацией об их сложности и т. д. лучше обращаться к другим источникам. Мы же всего лишь хотим быстро познакомить вас с алгоритмами.

Хотя во многих книгах при описании алгоритмов указывается полный синтаксис объявления шаблона, мы этого делать не будем. Предполагается, что вы уже достаточно хорошо разбираетесь в шаблонах, а смысл аргументов шаблона достаточно очевиден из объявлений функций. Необходимые типы итераторов описываются именами типов аргументов. Надеемся, в такой форме материал будет проще восприниматься, а полное описание при необходимости можно найти в заголовочном файле соответствующего шаблона.

Вся эта возня с итераторами нужна для того, чтобы алгоритмы работали с любыми типами контейнеров, удовлетворяющим требованиям стандартной библиотеки. До настоящего момента работа обобщенных алгоритмов демонстрировалась на примере массивов и векторов с последовательным хранением элементов, но в следующей главе будут представлены разнообразные структуры данных, не имеющие строгого порядка следования элементов. По этой причине одна из частичных классификаций алгоритмов основана на типе перебора.

Имена классов итераторов описывают возможности перебора, которые должны ими обеспечиваться. Не существует интерфейсных базовых классов, которые бы гарантировали выполнение этих операций — просто предполагается, что они присутствуют; в противном случае компилятор выдаст сообщение об ошибке. Далее кратко описаны основные категории итераторов.

[1] Например, в реализации STLPort на базе SGI STL, которая входит в Borland C++ Builder версии 6 и в компилятор Digital Mars.

- *Итераторы ввода* (InputIterator). Итератор ввода поддерживает только *чтение* элементов в прямом направлении (от начала к концу интервала) при помощи операторов ++ и *. Состояние итератора проверяется операторами == и !=.
- *Итераторы вывода* (OutputIterator). Итератор вывода поддерживает только *запись* элементов в прямом направлении при помощи операторов ++ и *. С другой стороны, состояние итераторов вывода не может проверяться операторами == и !=; предполагается, что элементы просто передаются в приемник, и проверка конечного условия не обязательна. Другими словами, считается, что контейнер, на который ссылается OutputIterator, может принять бесконечное число объектов. Это необходимо для того, чтобы тип OutputIterator мог использоваться с потоками ostream (через ostream_iterator), но на практике также часто применяются *итераторы вставки*, вроде возвращаемых функцией back_inserter().

Невозможно определить, ссылаются ли разные итераторы ввода или вывода на один и тот же интервал, поэтому такие итераторы не могут использоваться совместно. Просто представьте себе поддержку потоков istream и ostream на уровне итераторов, и смысл итераторов InputIterator и OutputIterator станет предельно ясен. Также стоит заметить, что InputIterator и OutputIterator относятся к категории обобщенных итераторов. Другими словами, когда вы видите InputIterator или OutputIterator в аргументах шаблонных алгоритмов STL, это означает, что вместо них можно подставить любой «более сложный» тип итератора.

- *Прямой итератор* (ForwardIterator). Через итератор InputIterator можно только читать, через итератор OutputIterator — только записывать, но ни один из них не позволяет одновременно читать и модифицировать интервал. *Прямые итераторы* снимают это ограничение; они по-прежнему позволяют перебирать элементы только в прямом направлении оператором ++, но при этом поддерживают как чтение, так и запись. Два прямых итератора, принадлежащих к одному интервалу, можно сравнить между собой. Так как прямые итераторы поддерживают чтение и запись, они могут использоваться вместо итераторов InputIterator и OutputIterator.
- *Двусторонний итератор* (BidirectionalIterator). Фактически двусторонний итератор представляет собой прямой итератор с возможностью перемещения в обратном направлении. Иначе говоря, BidirectionalIterator поддерживает все операции ForwardIterator, но при этом дополнительно поддерживает оператор --.
- *Итератор произвольного доступа* (RandomAccessIterator). Итераторы произвольного доступа поддерживают все операции обычных указателей: сложение и вычитание целочисленных смещений для перехода вперед и назад на несколько позиций (вместо одной), индексирование оператором [], вычитание одного итератора из другого, сравнение итераторов операторами <, > и т. д. Если вы программируете процедуру сортировки или что-нибудь в этом роде, вам не удастся создать эффективный алгоритм без итераторов произвольного доступа.

Типы параметров шаблонов обозначаются одним из перечисленных имен итераторов (иногда с добавлением суффикса 1 или 2, чтобы было проще различать аргументы). В списке также могут присутствовать другие аргументы, как правило — объекты функций.

При описании групп элементов, передаваемых операции, часто применяется математическое обозначение интервалов. Квадратная скобка означает, что граница входит в интервал, а круглая скобка — что граница исключается из интервала. Интервал определяется двумя итераторами: первый указывает на начальный элемент, а второй — в позицию «за концом интервала», то есть за последним элементом. Поскольку элемент «за последним» никогда не используется, такой интервал можно выразить в виде [first,last), где first — итератор, указывающий на начальный элемент, а last — итератор, установленный в позицию за последним элементом.

В большинстве книг и описаний алгоритмов STL алгоритмы классифицируются по их побочным эффектам: *неизменяющие алгоритмы* оставляют элементы интервала в прежнем состоянии, *изменяющие алгоритмы* модифицируют их, и т. д. Такие описания базируются в основном на базовом поведении или реализации алгоритма, то есть подаются с точки зрения разработчика. На практике подобные классификации особой пользы не приносят, поэтому мы будем разделять алгоритмы в соответствии с решаемыми задачами. Что именно вы хотите сделать: найти элемент или набор элементов, выполнить операцию с каждым элементом, подсчитать элементы, заменить элементы и т. д.? Это поможет вам быстрее найти нужный алгоритм.

Если над объявлениями функций не указан другой заголовок (например, <utility> или <numeric>), функция объявляется в <algorithm>. Кроме того, все алгоритмы принадлежат пространству имен std.

Вспомогательные инструменты для создания примеров

Начнем с создания вспомогательных инструментов, которые будут использоваться для демонстрации работы алгоритмов. В следующих примерах задействованы генераторы, определенные ранее в файле Generators.h, а также представленные далее шаблоны.

Вывод содержимого интервала является типичной задачей, поэтому мы определим шаблон функции для вывода произвольных интервалов независимо от типа элементов:

```
//: C06:PrintSequence.h
// Вывод содержимого любого интервала
#ifndef PRINTSEQUENCE_H
#define PRINTSEQUENCE_H
#include <algorithm>
#include <iostream>
#include <iterator>

template<typename Iter>
void print(Iter first, Iter last, const char* nm = "",
           const char* sep = "\n",
           std::ostream& os = std::cout) {
  if(nm != 0 && *nm != '\0')
    os << nm << ": " << sep;
  typedef typename
```

```
    std::iterator_traits<Iter>::value_type T;
  std::copy(first, last, std::ostream_iterator<T>(std::cout, sep));
  os << std::endl;
}
#endif // PRINTSEQUENCE_H ///:~
```

По умолчанию этот шаблон функции выводит данные в поток cout, разделяя их символами перевода строк, однако вы можете выбрать другой разделитель и передать его вместо аргумента по умолчанию. Также можно задать текст сообщения, выводимого перед основными данными. Функция print() использует алгоритм copy() для передачи объектов в cout через ostream_iterator, поэтому итератор ostream_iterator должен знать тип выводимого объекта, который мы определяем по члену value_type переданного итератора.

Шаблон std::iterator_traits позволяет шаблону функции print() обрабатывать интервалы, ограниченные любыми типами итераторов. В итераторах, возвращаемых стандартными контейнерами (такими, как vector), определяется вложенный тип value_type, представляющий тип элемента. Но при работе с массивами итераторы представляют собой обычные указатели, не имеющие вложенных типов. Чтобы обеспечить поддержку необходимых типов, связанных с итераторами в стандартной библиотеке, шаблон std::iterator_traits предоставляет следующую неполную специализацию для указателей:

```
template<class T>
  struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer'
    typedef T& reference;
};
```

В результате тип элементов, на которые ссылается указатель (а именно T), становится доступным через имя типа value_type.

Устойчивая и неустойчивая сортировка

Для некоторых алгоритмов STL, перемещающих элементы интервалов, различают *устойчивую* и *неустойчивую* сортировку. Устойчивая сортировка сохраняет исходный относительный порядок элементов, эквивалентных с точки зрения функции сравнения. Например, возьмем последовательность {c(1),b(1),c(2),a(1),b(2),a(2)}. Пусть элементы сравниваются по своим буквенным обозначениям, а цифры означают порядок их следования в исходном интервале. Если отсортировать эту последовательность с применением неустойчивой сортировки, порядок следования элементов с совпадающими буквами не гарантируется, поэтому в итоге может быть получен интервал вида {a(2),a(1),b(1),b(2),c(2),c(1)}. С другой стороны, при устойчивой сортировке вы получите {a(1),a(2),b(1),b(2),c(1),c(2)}. Алгоритм STL sort() использует разновидность алгоритма *быстрой сортировки* и поэтому является неустойчивым, однако в библиотеку также входит устойчивый алгоритм stable_sort()[1].

[1] В алгоритме stable_sort() применяется сортировка со слиянием, которая действительно сохраняет исходный порядок следования элементов, но в среднем работает несколько медленнее алгоритма быстрой сортировки.

Чтобы продемонстрировать устойчивую и неустойчивую сортировку алгоритмов, изменяющих порядок элементов последовательности, мы должны как-то отслеживать исходный порядок элементов. Приведенная ниже разновидность объекта string запоминает исходную относительную позицию данного объекта при помощи статической карты, связывающей строки с числовыми индексами. В каждом объекте NString присутствует поле thisOccurrence, обозначающее номер вхождения данного объекта.

```
//: C06:NString.h
// "Нумерованная строка", отслеживающая номер вхождения
// данного слова
#ifndef NSTRING_H
#define NSTRING_H
#include <algorithm>
#include <iostream>
#include <string>
#include <utility>
#include <vector>

typedef std::pair<std::string, int> psi;

// Сравнение только по первому элементу
bool operator==(const psi& l, const psi& r) {
  return l.first == r.first;
}

class NString {
  std::string s;
  int thisOccurrence;
  // Отслеживание количества вхождений:
  typedef std::vector<psi> vp;
  typedef vp::iterator vpit;
  static vp words;
  void addString(const std::string& x) {
    psi p(x, 0);
    vpit it = std::find(words.begin(), words.end(), p);
    if(it != words.end())
      thisOccurrence = ++it->second;
    else {
      thisOccurrence = 0;
      words.push_back(p);
    }
  }
public:
  NString() : thisOccurrence(0) {}
  NString(const std::string& x) : s(x) { addString(x); }
  NString(const char* x) : s(x) { addString(x); }
  // Автоматически сгенерированные оператор =
  // и копирующий конструктор здесь подойдут.
  friend std::ostream& operator<<(
    std::ostream& os, const NString& ns) {
    return os << ns.s << " [" << ns.thisOccurrence << "]";
  }
  // Оператор необходим для сортировки. Сравнение
  // производится только по строкам, без учета счетчиков вхождений:
  friend bool
  operator<(const NString& l, const NString& r) {
```

```
    return l.s < r.s;
  }
  friend
  bool operator==(const NString& l, const NString& r) {
    return l.s == r.s;
  }
  // Для сортировки с greater<NString>:
  friend bool
  operator>(const NString& l, const NString& r) {
    return l.s > r.s;
  }
  // Для прямого обращения к строке:
  operator const std::string&() const {return s;}
};

// Поскольку NString::vp является шаблоном, а мы используем
// модель с включением, шаблон должен определяться
// в этом заголовочном файле.
NString::vp NString::words;
#endif // NSTRING_H ///:~
```

Вообще говоря, для связывания строк со счетчиками вхождений было бы разумнее воспользоваться отображением (контейнер map). Однако эта разновидность контейнеров будет описана лишь в следующей главе, поэтому мы воспользуемся вектором, содержащим пары. В главе 7 будет приведено немало похожих примеров.

Для выполнения обычной сортировки по возрастанию необходима только операторная функция NString::operator<(). Чтобы сортировка могла выполняться в обратном порядке, мы также определяем функцию operator>(), вызываемую из шаблона greater.

Заполнение интервалов и генерирование значений

Алгоритмы, представленные в этом разделе, предназначены для автоматического заполнения интервалов. Функции fill и fill_n многократно включают в контейнер одно и то же значение. Функции generate и generate_n генерируют наборы значений, включаемых в контейнер, при помощи специальных функций-генераторов.

```
void fill(ForwardIterator first, ForwardIterator last, const T& value);
void fill_n(OutputIterator first, Size n, const T& value);
```

Алгоритм fill() присваивает значение value каждому элементу в интервале [first, last). Алгоритм fill_n() присваивает значение value n элементам интервала, начиная с first.

```
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
void generate_n(OutputIterator first, Size n, Generator gen);
```

Алгоритм generate() генерирует новые значения (вероятно, разные) для каждого элемента в интервале [first, last). Алгоритм generate_n() вызывает gen() n раз и присваивает результаты n элементам, начиная с first.

В следующем примере эти алгоритмы заполняют и генерируют новые значения в векторах. Кроме того, пример демонстрирует применение функции print():

```
//: C06:FillGenerateTest.cpp
// Демонстрация алгоритмов "fill" и "generate"
//{L} Generators
```

```
#include <vector>
#include <algorithm>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  vector<string> v1(5);
  fill(v1.begin(), v1.end(), "howdy");
  print(v1.begin(), v1.end(), "v1", " ");
  vector<string> v2;
  fill_n(back_inserter(v2), 7, "bye");
  print(v2.begin(), v2.end(), "v2");
  vector<int> v3(10);
  generate(v3.begin(), v3.end(), SkipGen(4,5));
  print(v3.begin(), v3.end(), "v3", " ");
  vector<int> v4;
  generate_n(back_inserter(v4),15, URandGen(30));
  print(v4.begin(), v4.end(), "v4", " ");
} ///:~
```

Программа создает контейнер vector<string> заранее заданного размера. Поскольку для всех объектов string в контейнере память уже выделена, алгоритм fill() может использовать свой оператор присваивания для замены каждого элемента вектора строкой "howdy". Кроме того, используемый по умолчанию разделитель (символ перевода строки) заменяется пробелом.

Для второго контейнера vector<string> v2 начальный размер не задан, поэтому нам приходится вставлять новые элементы при помощи функции back_inserter() (вместо заполнения существующих позиций).

Функции generate() и generate_n() очень похожи на функции заполнения, но вместо константы при вызове используется генератор. В нашем примере продемонстрированы два разных генератора.

Подсчет

Во всех классах контейнеров определена функция size(), которая возвращает текущее количество элементов. Возвращаемое значение size() относится к типу difference_type[1] типа итератора (обычно ptrdiff_t); в дальнейшем оно будет обозначаться IntegralValue. Следующие два алгоритма подсчитывают объекты, удовлетворяющие некоторому критерию.

```
IntegralValue count(InputIterator first, InputIterator
                    last, const EqualityComparable& value);
```

Алгоритм возвращает количество элементов в интервале [first,last), равных value (при сравнении используется оператор ==).

```
IntegralValue count_if(InputIterator first, InputIterator
                    last, Predicate pred);
```

Алгоритм возвращает количество элементов в интервале [first,last), для которых предикат pred возвращает true.

[1] Итераторы более подробно рассматриваются в следующей главе.

В этом примере контейнер vector<char> v заполняется случайными символами (с повторениями). Затем на базе вектора v инициализируется контейнер set<char>, который содержит только один экземпляр каждого символа, встречающегося в v. Программа подсчитывает, сколько раз каждый символ встречается в векторе, и выводит полученные данные:

```
//: C06:Counting.cpp
// Алгоритмы подсчета
//{L} Generators
#include <algorithm>
#include <functional>
#include <iterator>
#include <set>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  vector<char> v;
  generate_n(back_inserter(v), 50, CharGen());
  print(v.begin(), v.end(), "v", "");
  // Создание множества (set) на базе символьного вектора v:
  set<char> cs(v.begin(), v.end());
  typedef set<char>::iterator sci;
  for(sci it = cs.begin(); it != cs.end(); it++) {
    int n = count(v.begin(), v.end(), *it);
    cout << *it << ": " << n << ", ";
  }
  int lc = count_if(v.begin(), v.end(),
    bind2nd(greater<char>(), 'a'));
  cout << "\nLowercase letters: " << lc << endl;
  sort(v.begin(), v.end());
  print(v.begin(), v.end(), "sorted", "");
} ///:~
```

Работа алгоритма count_if() продемонстрирована на примере подсчета всех символов в нижнем регистре. Предикат создается с использованием стандартных шаблонов объектов функций bind2nd() и greater().

Копирование и перестановки

Алгоритмы группы копирования и перестановки предназначены для перемещения элементов.

```
OutputIterator copy(InputIterator first, InputIterator
                    last, OutputIterator destination);
```

Алгоритм копирует элементы из [first,last) в destination с использованием операции присваивания. После каждого присваивания destination инкрементируется. Итератор destination не может содержаться в исходном интервале. Так как при копировании задействована операция присваивания, алгоритм не позволяет напрямую вставлять элементы в пустой контейнер или добавлять их в конец контейнера. Вместо этого в качестве итератора destination приходится применять итератор вставки insert_iterator (обычно полученный функцией back_inserter() или inserter() для ассоциативных контейнеров).

```
BidirectionalIterator2 copy_backward(BidirectionalIterator1
  first, BidirectionalIterator1 last,
  BidirectionalIterator2 destinationEnd);
```

Алгоритм делает то же самое, что и алгоритм copy(), но копирование элементов производится в обратном порядке. Исходный интервал [first,last) копируется в приемник в обратном порядке, начиная с позиции destinationEnd-1. Далее итератор destinationEnd декрементируется после каждого присваивания. Приемный интервал должен содержать необходимое количество элементов (чтобы было возможно присваивание). Как и в случае с алгоритмом copy(), итератор destinationEnd не может содержаться в исходном интервале.

```
void reverse(BidirectionalIterator first, BidirectionalIterator last);
OutputIterator reverse_copy(BidirectionalIterator first,
  BidirectionalIterator last, OutputIterator destination);
```

Обе формы переставляют элементы интервала [first,last) в обратном порядке. Алгоритм reverse() осуществляет перестановку «на месте», а reverse_copy() оставляет исходный интервал без изменений, копирует переставленные элементы в destination и возвращает итератор, установленный в позицию за концом полученного интервала.

```
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2);
```

Алгоритм меняет местами содержимое двух интервалов одинакового размера попарной перестановкой элементов.

```
void rotate(ForwardIterator first, ForwardIterator middle,
  ForwardIterator last);
OutputIterator rotate_copy(ForwardIterator first,
  ForwardIterator middle, ForwardIterator last,
  OutputIterator destination);
```

Алгоритмы перемещают содержимое [first,middle) в конец интервала, а содержимое [middle,last) — в начало. Алгоритм rotate() осуществляет перестановку «на месте», а rotate_copy() оставляет исходный интервал без изменений, копирует измененную версию в destination и возвращает итератор, установленный в позицию за концом полученного интервала. Обратите внимание: в отличие от swap_ranges(), эти алгоритмы не требуют, чтобы размеры интервалов были одинаковыми.

```
bool next_permutation(BidirectionalIterator first,
  BidirectionalIterator last);
bool next_permutation(BidirectionalIterator first,
  BidirectionalIterator last, StrictWeakOrdering binary_pred);
bool prev_permutation(BidirectionalIterator first,
  BidirectionalIterator last);
bool prev_permutation(BidirectionalIterator first,
  BidirectionalIterator last, StrictWeakOrdering binary_pred);
```

Перестановкой называется уникальное упорядочение набора элементов. Если группа состоит из n уникальных элементов, в ней существует n! (n факториал) различных перестановок. На концептуальном уровне перестановки могут сортироваться по лексикографическому критерию, что позволяет использовать понятия «следующей» и «предыдущей» перестановок. Таким образом, независимо от текущего порядка следования элементов в последовательности перестановок всегда однозначно определяются «следующая» и «предыдущая» перестановки.

Алгоритмы next_permutation() и prev_permutation() пытаются построить следующую или предыдущую перестановку и в случае успеха возвращают true. Если элементы полностью отсортированы по возрастанию, а «следующих» перестановок не осталось, next_permutation() возвращает false. Если же элементы отсортированы по убыванию и не осталось «предыдущих» перестановок, то previous_permutation() возвращает false.

Версии с аргументом StrictWeakOrdering выполняют сравнения, используя бинарный предикат вместо оператора <.

```
void random_shuffle(RandomAccessIterator first,
  RandomAccessIterator last);
void random_shuffle(RandomAccessIterator first,
  RandomAccessIterator last, RandomNumberGenerator& rand);
```

Случайная перестановка элементов в интервале. Алгоритм обеспечивает равномерное распределение результатов только в том случае, если оно обеспечивается генератором случайных чисел. В первой форме задействован внутренний генератор случайных чисел, а второй форме передается пользовательская функция. Генератор должен возвращать значение в интервале [0,n) для некоторого положительного значения n.

```
BidirectionalIterator partition(BidirectionalIterator first,
  BidirectionalIterator last, Predicate pred);
BidirectionalIterator stable_partition(BidirectionalIterator first,
  BidirectionalIterator last, Predicate pred);
```

Элементы, удовлетворяющие предикату pred, перемещаются в начало интервала. Алгоритмы возвращают итератор, установленный в позицию за последним из перемещенных элементов (то есть фактически в позицию «конечного» итератора для исходного подинтервала элементов, удовлетворяющих предикату pred). Данная позиция часто называется «точкой разбиения».

Для алгоритма partition() элементы в полученных подинтервалах располагаются в неопределенном порядке, а stable_partition() сохраняет элементы до и после точки разбиения в том же относительном порядке, в котором они следовали в исходном интервале.

Следующая программа демонстрирует работу алгоритмов копирования и перестановки элементов.

```
//: C06:Manipulations.cpp
// Копирование и перестановки
//{L} Generators
// NString
#include <vector>
#include <string>
#include <algorithm>
#include "PrintSequence.h"
#include "NString.h"
#include "Generators.h"
using namespace std;

int main() {
  vector<int> v1(10);
  // Простой подсчет:
  generate(v1.begin(), v1.end(), SkipGen());
  print(v1.begin(), v1.end(), "v1", " ");
```

```
vector<int> v2(v1.size());
copy_backward(v1.begin(), v1.end(), v2.end());
print(v2.begin(), v2.end(), "copy_backward", " ");
reverse_copy(v1.begin(), v1.end(), v2.begin());
print(v2.begin(), v2.end(), "reverse_copy", " ");
reverse(v1.begin(), v1.end());
print(v1.begin(), v1.end(), "reverse", " ");
int half = v1.size() / 2;
// Интервалы должны иметь одинаковые размеры:
swap_ranges(v1.begin(), v1.begin() + half,
  v1.begin() + half);
print(v1.begin(), v1.end(), "swap_ranges", " ");
// Сгенерировать интервал заново:
generate(v1.begin(), v1.end(), SkipGen());
print(v1.begin(), v1.end(), "v1", " ");
int third = v1.size() / 3;
for(int i = 0; i < 10; i++) {
  rotate(v1.begin(), v1.begin() + third,
    v1.end());
  print(v1.begin(), v1.end(), "rotate", " ");
}
cout << "Second rotate example:" << endl;
char c[] = "aabbccddeeffgghhiijj";
const char CSZ = strlen(c);
for(int i = 0; i < 10; i++) {
  rotate(c, c + 2, c + CSZ);
  print(c, c + CSZ, "", "");
}
cout << "All n! permutations of abcd:" << endl;
int nf = 4 * 3 * 2 * 1;
char p[] = "abcd";
for(int i = 0; i < nf; i++) {
  next_permutation(p, p + 4);
  print(p, p + 4, "", "");
}
cout << "Using prev_permutation:" << endl;
for(int i = 0; i < nf; i++) {
  prev_permutation(p, p + 4);
  print(p, p + 4, "", "");
}
cout << "random_shuffling a word:" << endl;
string s("hello");
cout << s << endl;
for(int i = 0; i < 5; i++) {
  random_shuffle(s.begin(), s.end());
  cout << s << endl;
}
NString sa[] = { "a", "b", "c", "d", "a", "b",
  "c", "d", "a", "b", "c", "d", "a", "b", "c"};
const int sasz = sizeof sa / sizeof *sa;
vector<NString> ns(sa, sa + sasz);
print(ns.begin(), ns.end(), "ns", " ");
vector<NString>::iterator it =
  partition(ns.begin(), ns.end(),
    bind2nd(greater<NString>(), "b"));
cout << "Partition point: " << *it << endl;
print(ns.begin(), ns.end(), "", " ");
```

```
  // Повторная загрузка вектора:
  copy (sa, sa + sasz, ns.begin());
  it = stable_partition(ns.begin(), ns.end(),
    bind2nd(greater<NString>(), "b"));
  cout << "Stable partition" << endl;
  cout << "Partition point: " << *it << endl;
  print(ns.begin(), ns.end(), "", " ");
} ///:~
```

Чтобы увидеть, как работает эта программа, проще всего запустить ее (вероятно, с перенаправлением вывода в файл).

Сначала мы загружаем вектор vector<int> v1 простой возрастающей последовательностью и выводим его. Вы увидите, что алгоритм copy_backward() (который копирует данные в вектор v2, по размеру равный v1) приводит к тому же результату, что и обычное копирование алгоритмом copy(), — просто операция выполняется в обратном порядке[1].

Алгоритм reverse_copy() создает копию исходного интервала с элементами, переставленными в обратном порядке, а алгоритм reverse() выполняет перестановку на месте. Далее алгоритм swap_ranges() меняет местами верхнюю половину переставленного интервала с нижней.

После воссоздания первоначальной последовательности программа демонстрирует работу алгоритма rotate() на примере многократного сдвига первой трети v1. Во втором примере rotate() используются символы, а сдвиг выполняется на две позиции. Кстати, этот пример также демонстрирует гибкость алгоритмов STL и шаблона print() — они могут работать с массивами char с такой же легкостью, как с любыми другими данными.

Чтобы продемонстрировать работу алгоритмов next_permutation() и prev_permutation(), мы делаем все n! возможных перестановок для набора из четырех символов «abcd». Из результатов видно, что перестановки генерируются в строго определенном порядке, то есть процесс их перебора детерминирован.

При простейшей демонстрации алгоритма random_shuffle() мы применяем его к строке и смотрим, какие слова при этом получаются. У объектов string имеются функции begin() и end(), которые возвращают соответствующие итераторы, что позволяет легко использовать строки со многими алгоритмами STL. Также можно было воспользоваться массивом char.

Напоследок демонстрируются алгоритмы partition() и stable_partition() с массивом NString. Обратите внимание: в выражении агрегатной инициализации используются массивы char, но у NString имеется конструктор для char*, который автоматически вызывается в этом случае.

Из выходных данных программы видно, что алгоритм partition() размещает данные до и после точки разбиения правильно, но в непредсказуемом порядке, тогда как алгоритм stable_partition() сохраняет их исходный порядок.

Поиск и замена

Алгоритмы группы поиска и замены предназначены для поиска объектов (одного или нескольких) в интервале, определяемом первыми двумя итераторами.

---

[1] Различия между copy() и copy_backward() проявляются только при частичном перекрытии исходного и приемного интервалов. — *Примеч. перев.*

```
InputIterator find(InputIterator first, InputIterator last,
  const EqualityComparable& value);
```

Алгоритм ищет значение value в заданном интервале [first,last) и возвращает итератор, указывающий на первое вхождение. Если значение value не найдено, возвращается итератор last. Алгоритм выполняет *линейный поиск*; иначе говоря, он начинает с начала и последовательно просматривает все элементы, не делая никаких предположений относительно их сортировки. С другой стороны, алгоритм binary_search() (см. далее) работает только с сортированными интервалами, но зато гораздо быстрее.

```
InputIterator find_if(InputIterator first, InputIterator
  last, Predicate pred);
```

Алгоритм find_if(), как и find(), выполняет линейный поиск в интервале. Но вместо фиксированного значения он ищет элемент, для которого предикат pred возвращает true. При отсутствии таких элементов возвращается итератор last.

```
ForwardIterator adjacent_find(ForwardIterator first,
  ForwardIterator last);
ForwardIterator adjacent_find(ForwardIterator first,
  ForwardIterator last, BinaryPredicate binary_pred);
```

Как и find(), этот алгоритм выполняет линейный поиск в интервале, но вместо одного фиксированного значения он ищет два эквивалентных соседних элемента. Первая форма ищет два соседних элемента с одинаковыми значениями (сравнение осуществляется оператором ==). Вторая форма ищет два соседних элемента, для которых бинарный предикат binary_pred возвращает true. Если искомая пара успешно найдена, алгоритм возвращает итератор для первого из двух элементов; в противном случае возвращается итератор last.

```
ForwardIterator1 find_first_of(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2,
  ForwardIterator2 last2);
ForwardIterator1 find_first_of(ForwardIterator1 first1,
  ForwardIterator1 last1, ForwardIterator2 first2,
  ForwardIterator2 last2, BinaryPredicate binary_pred);
```

Как и find(), алгоритм find_first_of() выполняет линейный поиск в интервале. Обе формы ищут в первом интервале элемент, значение которого эквивалентно одному из элементов второго интервала. Первая форма сравнивает элементы при помощи оператора ==, а вторая — при помощи заданного предиката, аргументами которого являются текущий элемент первого интервала и элемент второго интервала.

```
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
  ForwardIterator2 first2, ForwardIterator2 last2);
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
  ForwardIterator2 first2, ForwardIterator2 last2,
  BinaryPredicate binary_pred);
```

Алгоритм проверяет, входит ли второй интервал в первый интервал (при том же порядке следования элементов), и если входит — возвращает итератор для начальной позиции первого вхождения второго интервала. Если подмножество найти не удается, возвращается last1. Первая форма сравнивает элементы оператором ==, а вторая форма вызывает для каждой пары сравниваемых объектов предикат binary_pred и проверяет, равен ли результат true.

```
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
  ForwardIterator2 first2, ForwardIterator2 last2);
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
  ForwardIterator2 first2, ForwardIterator2 last2,
  BinaryPredicate binary_pred);
```

По своим формам и аргументам этот алгоритм аналогичен search(): он тоже ищет вхождение второго интервала в виде подинтервала первого. Но если search() ищет первое вхождение подинтервала, то find_end() находит *последнее* вхождение и возвращает итератор, установленный на его первый элемент.

```
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
  Size count, const T& value);
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
  Size count, const T& value, BinaryPredicate binary_pred);
```

Алгоритм ищет в интервале [first,last) группу из count последовательных элементов, равных value (первая форма) или возвращающих true при передаче предикату binary_pred вместе с value (вторая форма). Если найти такую группу не удается, алгоритм возвращает last.

```
ForwardIterator min_element(ForwardIterator first,
  ForwardIterator last);
ForwardIterator min_element(ForwardIterator first,
  ForwardIterator last, BinaryPredicate binary_pred);
```

Алгоритм возвращает итератор, указывающий на первое вхождение «наименьшего» значения в интервале (как объясняется далее, таких вхождений может быть несколько). При неудачном поиске возвращается last. Первая версия выполняет сравнения оператором < и возвращает значение r, при котором условие *e<*r ложно для каждого элемента e в интервале [first,r). Вторая версия использует для сравнения бинарный предикат и возвращает значение r, при котором функция binary_pred(*e,*r) возвращает false для каждого элемента e в интервале [first,r).

```
ForwardIterator max_element(ForwardIterator first,
  ForwardIterator last);
ForwardIterator max_element(ForwardIterator first,
  ForwardIterator last, BinaryPredicate binary_pred);
```

Алгоритм возвращает итератор, указывающий на первое вхождение наибольшего значения в интервале (таких вхождений может быть несколько). При неудачном поиске возвращается last. Первая версия выполняет сравнения оператором < и возвращает значение r, при котором условие *r<*e ложно для каждого элемента e в интервале [first,r). Вторая версия использует для сравнения бинарный предикат и возвращает значение r, при котором функция binary_pred(*r,*e) возвращает false для каждого элемента e в интервале [first,r).

```
void replace(ForwardIterator first, ForwardIterator last,
  const T& old_value, const T& new_value);
void replace_if(ForwardIterator first, ForwardIterator last,
  Predicate pred, const T& new_value);
OutputIterator replace_copy(InputIterator first,
  InputIterator last, OutputIterator result, const T& old_value,
  const T& new_value);
OutputIterator replace_copy_if(InputIterator first,
  InputIterator last, OutputIterator result, Predicate pred,
  const T& new_value);
```

Все разновидности replace() перебирают интервал [first,last) в поиске значений, удовлетворяющих заданному критерию, и заменяют их значением new_value. Алгоритмы replace() и replace_copy() просто ищут фиксированное значение old_value; алгоритмы replace_if() и replace_copy_if() ищут значения, удовлетворяющие предикату pred. Версии с суффиксом copy не изменяют исходный интервал, а создают модифицированную копию в result (после каждого присваивания result инкрементируется).

Для удобства просмотра результатов в данном примере используются векторы с элементами типа int. Продемонстрированы не все версии каждого алгоритма (наиболее очевидные опущены).

```
//: C06:SearchReplace.cpp
// Алгоритмы поиска и замены в STL
#include <algorithm>
#include <functional>
#include <vector>
#include "PrintSequence.h"
using namespace std;

struct PlusOne {
  bool operator()(int i, int j) { return j == i + 1; }
};

class MulMoreThan {
  int value;
public:
  MulMoreThan(int val) : value(val) {}
  bool operator()(int v, int m) { return v * m > value; }
};

int main() {
  int a[] = { 1, 2, 3, 4, 5, 6, 6, 7, 7, 7,
    8, 8, 8, 8, 11, 11, 11, 11, 11 };
  const int ASZ = sizeof a / sizeof *a;
  vector<int> v(a, a + ASZ);
  print(v.begin(), v.end(), "v", " ");
  vector<int>::iterator it = find(v.begin(), v.end(), 4);
  cout << "find: " << *it << endl;
  it = find_if(v.begin(), v.end(),
    bind2nd(greater<int>(), 8));
  cout << "find_if: " << *it << endl;
  it = adjacent_find(v.begin(), v.end());
  while(it != v.end()) {
    cout << "adjacent_find: " << *it
      << ", " << *(it + 1) << endl;
    it = adjacent_find(it + 1, v.end());
  }
  it = adjacent_find(v.begin(), v.end(), PlusOne());
  while(it != v.end()) {
    cout << "adjacent_find PlusOne: " << *it
      << ", " << *(it + 1) << endl;
    it = adjacent_find(it + 1, v.end(), PlusOne());
  }
  int b[] = { 8, 11 };
  const int BSZ = sizeof b / sizeof *b;
  print(b, b + BSZ, "b", " ");
  it = find_first_of(v.begin(), v.end(), b, b + BSZ);
```

```
  print(it, it + BSZ, "find_first_of", " ");
  it = find_first_of(v.begin(), v.end(),
    b, b + BSZ, PlusOne());
  print(it,it + BSZ,"find_first_of PlusOne"," ");
  it = search(v.begin(), v.end(), b, b + BSZ);
  print(it, it + BSZ, "search", " ");
  int c[] = { 5, 6, 7 };
  const int CSZ = sizeof c / sizeof *c;
  print(c, c + CSZ, "c", " ");
  it = search(v.begin(), v.end(), c, c + CSZ, PlusOne());
  print(it, it + CSZ,"search PlusOne", " ");
  int d[] = { 11, 11, 11 };
  const int DSZ = sizeof d / sizeof *d;
  print(d, d + DSZ, "d", " ");
  it = find_end(v.begin(), v.end(), d, d + DSZ);
  print(it, v.end(),"find_end", " ");
  int e[] = { 9, 9 };
  print(e, e + 2, "e", " ");
  it = find_end(v.begin(), v.end(), e, e + 2, PlusOne());
  print(it, v.end(),"find_end PlusOne"," ");
  it = search_n(v.begin(), v.end(), 3, 7);
  print(it, it + 3, "search_n 3, 7", " ");
  it = search_n(v.begin(), v.end(),
    6, 15, MulMoreThan(100));
  print(it, it + 6,
    "search_n 6, 15, MulMoreThan(100)", " ");
  cout << "min_element: " <<
    *min_element(v.begin(), v.end()) << endl;
  cout << "max_element: " <<
    *max_element(v.begin(), v.end()) << endl;
  vector<int> v2;
  replace_copy(v.begin(), v.end(),
    back_inserter(v2), 8, 47);
  print(v2.begin(), v2.end(), "replace_copy 8 -> 47", " ");
  replace_if(v.begin(), v.end(),
    bind2nd(greater_equal<int>(), 7), -1);
  print(v.begin(), v.end(), "replace_if >= 7 -> -1", " ");
} ///:~
```

В начале примера определяются два предиката. Бинарный предикат PlusOne возвращает true, если второй аргумент на 1 больше второго аргумента, а MulMoreThan возвращает true, если первый аргумент равен произведению на второй аргумент значения, хранящегося в объекте.

Функция main() создает массив a и передает его конструктору vector<int> v. Этот вектор используется для выполнения операций поиска и замены. Обратите внимание на присутствие дубликатов — они обнаруживаются некоторыми алгоритмами поиска и замены.

Первый тест демонстрирует алгоритм find() на примере поиска значения 4 в v. Возвращаемое значение представляет собой итератор, указывающий на первое вхождение 4 или на конец входного интервала (v.end()), если искомое значение не найдено.

Алгоритм find_if() использует предикат для проверки элементов. В нашем примере предикат строится «на месте» при помощи стандартного объекта функции greater<int> (то есть «проверяет, что первый аргумент типа int больше второго аргумента») и адаптера bind2nd(), который фиксирует второй аргумент равным 8. Таким образом, предикат возвращает true, если значение v больше 8.

Поскольку вектор v содержит несколько смежных пар одинаковых объектов, тест adjacent_find() спроектирован так, чтобы найти все дубликаты. Поиск начинается от начала интервала и продолжается в цикле до тех пор, пока итератор it не достигнет конца исходного интервала (когда других совпадений заведомо не остается). Цикл выводит информацию о каждом найденном совпадении, после чего снова вызывает adjacent_find() с первым аргументом it + 1. Такая реализация позволяет найти две пары в последовательности из трех элементов.

При взгляде на цикл while кажется, что его можно записать изящнее:

```
while(it != v.end()) {
  cout << "adjacent_find: " << *it++
    << ", " << *it++ << endl;
  it = adjacent_find(it, v.end());
}
```

Именно это мы попытались сделать в первом варианте программы. Однако нам не удалось получить ожидаемый результат ни на одном компиляторе! Дело в отсутствии гарантированного порядка выполнения инкрементов в этом выражении.

В следующем тесте алгоритм adjacent_find() используется в сочетании с предикатом PlusOne. Тест обнаруживает все пары, в которых следующий элемент на 1 больше предыдущего. Как и в описанном тесте, цикл while находит все вхождения.

Алгоритму find_first_of() передается второй интервал с объектами, которые ищутся в первом интервале. В нашем примере это массив b. Поскольку первый и второй интервалы find_first_of() определяются разными аргументами шаблонов, они могут относиться к контейнерам разного типа, как в данном случае. Также в программе тестируется вторая форма find_first_of() с использованием предиката PlusOne.

Алгоритм search() находит в первом интервале точную копию содержимого второго интервала с тем же порядком следования элементов. Вторая форма search() использует предикат, который обычно определяет некоторую форму проверки эквивалентности, но существуют и другие интересные возможности — в нашей программе предикат PlusOne позволяет найти интервал {4,5,6}.

Тест find_end() находит *последнее* вхождение всей последовательности {11,11,11}. Чтобы доказать, что найдено действительно последнее вхождение, мы выводим оставшиеся элементы v, начиная с it.

Первый тест search_n() ищет первые 3 экземпляра значения 7, находит и выводит их. Во второй версии search_n() предикат обычно проверяет эквивалентность двух элементов. Однако мы пошли нестандартным путем и воспользовались объектом функции, который умножает значение в интервале на 15 и проверяет, окажется ли результат больше 100. Другими словами, тест с search_n() означает: «Найти 6 последовательных значений, которые при умножении на 15 дают число больше 100». На практике такие решения встречаются не так часто, но возможно, оно все же пригодится вам, когда вы в очередной раз столкнетесь с нетипичной задачей поиска.

Алгоритмы min_element() и max_element() просты и понятны, хотя на первый взгляд разыменование функции символом * выглядит странно. На самом деле мы разыменовываем возвращаемый итератор, чтобы получить значение для вывода.

Тестирование алгоритмов замены начинается с алгоритма replace_copy(), не изменяющего исходный вектор. Мы заменяем все вхождения 8 значением 47. Обратите внимание на использование функции back_inserter() с пустым вектором v2.

Для демонстрации алгоритма replace_if() мы создаем на базе стандартного шаблона greater_equal с адаптером bind2nd() объект функции для замены всех элементов, больших либо равных 7, значением –1.

Сравнение интервалов

На первый взгляд может показаться, что по функциональности алгоритмы группы сравнения интервалов напоминают алгоритмом search(). Однако search() сообщает, где второй интервал находится внутри первого, а equal() и lexicographical_compare() — в каком отношении между собой находятся два интервала. Алгоритм mismatch() указывает, где начинается расхождение в содержимом двух интервалов, но для этого интервалы должны иметь одинаковую длину.

```
bool equal(InputIterator first1, InputIterator last1,
  InputIterator first2);
bool equal(InputIterator first1, InputIterator last1,
  InputIterator first2, BinaryPredicate binary_pred);
```

В обоих алгоритмах первый интервал задается в канонической форме [first1,last1). Второй интервал начинается с first2, но парного конечного итератора last2 не существует, потому что его длина определяется длиной первого интервала. Функция equal() возвращает true, если оба интервала точно совпадают (содержат одни и те же элементы, следующие в одинаковом порядке). В первой форме элементы сравниваются оператором ==, а во второй эквивалентность определяется бинарным предикатом binary_pred.

```
bool lexicographical_compare(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2);
bool lexicographical_compare(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, BinaryPredicate binary_pred);
```

Алгоритм проверяет, меньше ли первый интервал, чем второй по лексикографическому критерию (если интервал 1 меньше интервала 2, алгоритм возвращает true, а если нет — возвращается false). *Лексикографическое сравнение* производится по тому же принципу, по которому мы ищем слова в словаре, то есть последовательным сравнением элементов. Если первые элементы различаются, то их отношение определяет результат сравнения. Но если элементы равны, алгоритм переходит к следующей паре элементов и рассматривает их. Это продолжается до тех пор, пока не будет найдено расхождение. На этой стадии алгоритм сравнивает различающиеся элементы, и если элемент из первого интервала меньше элемента из второго, алгоритм lexicographical_compare() возвращает true; в противном случае — false.

Если интервалы имеют разную длину, отсутствующий элемент одного интервала считается «предшествующим» существующему элементу второго интервала, то есть «abc» предшествует «abcd». Если алгоритм достигает конца одного из интервалов, не обнаружив расхождений, более короткий интервал считается меньшим. Если первый интервал имеет меньшую длину, то результат равен true; в противном случае он равен false.

В первой форме алгоритма элементы сравниваются оператором <, а во второй эквивалентность определяется бинарным предикатом binary_pred.

```
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2);
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  BinaryPredicate binary_pred);
```

Как и в случае с equal(), оба интервала должны иметь одинаковую длину, поэтому для второго интервала достаточно одного итератора, а его длина определяется длиной первого интервала. Если алгоритм equal() просто сообщает, равны ли два интервала, то mismatch() позволяет узнать, где именно начинаются расхождения. Передаваемая информация состоит из двух значений: различающихся элементов первого и второго интервалов. Эти два итератора упаковываются в объект pair и возвращаются как единое целое. При отсутствии расхождений алгоритм возвращает last1 в сочетании с конечным итератором второго интервала. Шаблон pair представляет собой структуру из двух элементов, обозначенных именами first и second, а его определение находится в заголовочном файле <utility>.

Как и в случае с equal(), в первой форме алгоритма элементы сравниваются оператором ==, а во второй эквивалентность определяется бинарным предикатом binary_pred.

Стандартный класс C++ обладает свойствами контейнера (он содержит функции begin() и end(), возвращающие объекты типа string::iterator), поэтому его удобно использовать для создания символьных интервалов при тестировании алгоритмов сравнения STL. Однако следует помнить, что класс string поддерживает достаточно полный набор собственных операций. Прежде чем задействовать алгоритмы STL для выполнения операций, стоит внимательнее изучить класс string.

```
//: C06:Comparison.cpp
// Алгоритмы сравнения интервалов STL
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include "PrintSequence.h"
using namespace std;

int main() {
  // Класс string содержит удобные средства для создания
  // символьных интервалов, но обычно строковые операции
  // лучше выполнять функциями этого класса:
  string s1("This is a test");
  string s2("This is a Test");
  cout << "s1: " << s1 << endl << "s2: " << s2 << endl;
  cout << "compare s1 & s1: "
    << equal(s1.begin(), s1.end(), s1.begin()) << endl;
  cout << "compare s1 & s2: "
    << equal(s1.begin(), s1.end(), s2.begin()) << endl;
  cout << "lexicographical_compare s1 & s1: " <<
    lexicographical_compare(s1.begin(), s1.end(),
      s1.begin(), s1.end()) <<  endl;
  cout << "lexicographical_compare s1 & s2: " <<
    lexicographical_compare(s1.begin(), s1.end(),
      s2.begin(), s2.end()) << endl;
  cout << "lexicographical_compare s2 & s1: " <<
    lexicographical_compare(s2.begin(), s2.end(),
```

```
      s1.begin(), s1.end()) << endl;
  cout << "lexicographical_compare shortened "
    "s1 & full-length s2: " << endl;
  string s3(s1);
  while(s3.length() != 0) {
    bool result = lexicographical_compare(
      s3.begin(), s3.end(), s2.begin(),s2.end());
    cout << s3 << endl << s2 << ", result = "
      << result << endl;
    if(result == true) break;
    s3 = s3.substr(0, s3.length() - 1);
  }
  pair<string::iterator, string::iterator> p =
    mismatch(s1.begin(), s1.end(), s2.begin());
  print(p.first, s1.end(), "p.first", "");
  print(p.second, s2.end(), "p.second","");
} ///:~
```

Строки s1 и s2 различаются только прописной буквой «T» в слове «Test» строки s2. Как и следовало ожидать, при проверке равенства s1 и s1 возвращается значение true, а s1 и s2 не равны из-за отличий в регистре буквы «T».

Чтобы понять результаты тестов lexicographical_compare(), следует помнить две вещи: во-первых, сравнение производится посимвольно, во-вторых, на нашей платформе прописные буквы «предшествуют» строчным. В первом тесте s1 сравнивается с s1. Так как строки точно совпадают, ни одна из них *не меньше* другой по лексикографическому критерию (что проверяется алгоритмом), поэтому результат равен false. Фактически второй тест проверяет, предшествует ли s1 строке s2. Когда алгоритм добирается до буквы «t» в слове «test», он обнаруживает, что строчная «t» в s1 «больше» прописной «t» в s2, так что результат снова равен false. Но если проверить, предшествует ли s2 строке s1, мы получим true.

Чтобы читатель лучше понял смысл лексикографического сравнения, следующий тест в этом примере снова сравнивает s1 с s2 (раньше при этой проверке был получен результат false). Но в новом варианте проверки с конца строки s1, предварительно скопированной в s3, последовательно отсекается по одному символу до тех пор, пока алгоритм не вернет true. Как только из s3 (копии s1) удаляется буква «T», результат меняется. Так как строка s3 короче s2, в лексикографическом отношении она предшествует ей.

В последнем тесте используется алгоритм mismatch(). Чтобы сохранить возвращаемое значение, мы создаем объект pair p с типами итераторов из первого и второго интервалов (в нашем примере оба итератора относятся к типу string::iterator). При выводе результатов несовпадающий элемент в первом интервале определяется итератором p.first, а во втором — итератором p.second. В обоих случаях выводится содержимое интервала от несовпадающего элемента до конца, чтобы было понятно, на какой элемент ссылается итератор.

Удаление элементов

Универсальность STL накладывает свои ограничения на концепцию удаления. Поскольку элементы могут «удаляться» только через итераторы, а итератор может ссылаться на массив, вектор, список и т. д., было бы небезопасно и неразумно пытаться уничтожать удаляемые элементы и изменять размер исходного интервала [first,last) (к примеру, размер массива вообще не меняется). По этой причине

алгоритмы «удаления» в STL всего лишь переставляют элементы интервала так, чтобы «удаленные» элементы находились в конце интервала, а «оставшиеся» — в начале (и в том же порядке, что прежде, за исключением удаленных элементов, то есть операция является *устойчивой*). Алгоритм возвращает итератор для нового «последнего» элемента, обозначающий конец подынтервала «оставшихся» элементов и начало подынтервала «удаленных» элементов. Другими словами, если алгоритм удаления возвращает итератор new_last, то интервал [first,new_last) не содержит ни одного удаленного элемента, а интервал [new_last,last) состоит только из удаленных элементов.

Если вы работаете с элементами интервала при помощи других алгоритмов STL, можно просто использовать итератор new_last как новый конечный итератор. С другой стороны, если вы применяете контейнер c, поддерживающий изменение размеров (то есть не массив), и хотите исключить «логически удаленные» элементы из контейнера, воспользуйтесь алгоритмом erase():

```
c.erase(remove(c.begin(), c.end(), value), c.end());
```

Также можно воспользоваться функцией resize(), поддерживаемой всеми стандартными контейнерами (о ней будет рассказано в следующей главе).

Алгоритм remove() возвращает итератор new_last, а erase() уничтожает все удаленные элементы из контейнера c.

К итераторам в интервале [new_last,last) может применяться операция разыменования, но значения элементов не определены и не должны использоваться в программе.

```
ForwardIterator remove(ForwardIterator first,
  ForwardIterator last, const T& value);
ForwardIterator remove_if(ForwardIterator first,
  ForwardIterator last, Predicate pred);
OutputIterator remove_copy(InputIterator first,
  InputIterator last, OutputIterator result, const T& value);
OutputIterator remove_copy_if(InputIterator first,
  InputIterator last, OutputIterator result, Predicate pred);
```

Все формы удаляющих алгоритмов перебирают интервал [first,last), находят элементы, соответствующие критерию удаления, и копируют оставшиеся элементы поверх удаленных (логическое удаление). При этом сохраняется исходный относительный порядок следования неудаленных элементов. Алгоритмы возвращают итератор, установленный в следующую позицию за концом интервала, содержащего неудаленные элементы. Значение, на которое ссылается этот итератор, не определено.

Версии этих алгоритмов с суффиксом «if» передают каждый элемент предикату pred и по полученному значению определяют, должен ли этот элемент остаться в контейнере (если pred() возвращает true, элемент удаляется). Версии с суффиксом «copy» не изменяют исходный интервал; они копируют неудаленные элементы в интервал, начинающийся с result, и возвращают конечный итератор для этого нового интервала.

```
ForwardIterator unique(ForwardIterator first,
  ForwardIterator last);
ForwardIterator unique(ForwardIterator first,
  ForwardIterator last, BinaryPredicate binary_pred);
OutputIterator unique_copy(InputIterator first,
  InputIterator last, OutputIterator result);
```

```
OutputIterator unique_copy(InputIterator first,
  InputIterator last, OutputIterator result,
  BinaryPredicate binary_pred);
```

Версии с префиксом «unique» перебирают интервал [first,last), находят в нем смежные эквивалентные значения и «удаляют» дубликаты, копируя следующие элементы поверх них. При этом сохраняется исходный порядок следования неудаленных элементов. Возвращаемое значение представляет собой конечный итератор для интервала, полученного после удаления смежных дубликатов.

Поскольку эти алгоритмы удаляют только смежные дубликаты, перед их вызовом обычно вызывается алгоритм sort(), обеспечивающий гарантированное удаление всех дубликатов.

Для каждого значения итератора i в исходном интервале версии с бинарным предикатом binary_pred вызывают

```
binary_pred(*i, *(i-1));
```

Если результат равен true, *i считается дубликатом.

Версии с суффиксом «copy» не изменяют исходный интервал; они копируют неудаленные элементы в интервал, начинающийся с result, и возвращают конечный итератор для этого нового интервала.

Следующая программа наглядно демонстрирует работу алгоритмов семейств remove и unique:

```
//: C06:Removing.cpp
// Алгоритмы удаления
//{L} Generators
#include <algorithm>
#include <cctype>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

struct IsUpper {
  bool operator()(char c) { return isupper(c); }
};

int main() {
  string v;
  v.resize(25);
  generate(v.begin(), v.end(), CharGen());
  print(v.begin(), v.end(), "v original", "");
  // Создание набора символов на базе v:
  string us(v.begin(), v.end());
  sort(us.begin(), us.end());
  string::iterator it = us.begin(), cit = v.end(),
    uend = unique(us.begin(), us.end());
  // Перебор и удаление всего содержимого:
  while(it != uend) {
    cit = remove(v.begin(), cit, *it);
    print(v.begin(), v.end(), "Complete v", "");
    print(v.begin(), cit, "Pseudo v ", " ");
    cout << "Removed element:\t" << *it
         << "\nPseudo Last Element:\t"
         << *cit << endl << endl;
```

```
    it++;
  }
  generate(v.begin(), v.end(), CharGen());
  print(v.begin(), v.end(), "v", "");
  cit = remove_if(v.begin(), v.end(), IsUpper());
  print(v.begin(), cit, "v after remove_if IsUpper", " ");
  // Копирующие версии remove и remove_if не показаны.
  sort(v.begin(), cit);
  print(v.begin(), cit, "sorted", " ");
  string v2;
  v2.resize(cit - v.begin());
  unique_copy(v.begin(), cit, v2.begin());
  print(v2.begin(), v2.end(), "unique_copy", " ");
  // Работает так же:
  cit = unique(v.begin(), cit, equal_to<char>());
  print(v.begin(), cit, "unique equal_to<char>", " ");
} ///:~
```

Строка v представляет собой контейнер, заполненный случайно сгенерированными символами. Символы последовательно удаляются вызовом remove, но при этом каждый раз выводится полная строка v, чтобы вы могли проследить за состоянием всего интервала после текущей конечной точки (cit).

Для демонстрации алгоритма remove_if() мы вызываем стандартную библиотечную функцию isupper() языка C (заголовочный файл <cctype>) внутри объекта функции IsUpper, переданного в качестве предиката remove_if(). Предикат возвращает true для символов верхнего регистра, поэтому в итоге в интервале остаются только символы нижнего регистра. При вызове print() передается новый конец интервала, поэтому выводятся только оставшиеся элементы. Копирующие версии remove() и remove_if() не приводятся, так как они принципиально не отличаются от некопирующих версий, и отдельные примеры для них не нужны.

Перед тестированием алгоритмов unique мы сортируем полученный интервал строчных букв (эти алгоритмы могут применяться и к несортированным интервалам, но, вероятно, это не приведет к ожидаемому результату). Сначала unique_copy() заносит уникальные элементы в новый вектор (способ сравнения элементов определяется по умолчанию), а затем использует форму unique() с передачей предиката. Предикатом является стандартный объект функции equal_to(), который дает тот же результат, что и при сравнении по умолчанию.

Сортировка и операции с отсортированными интервалами

Значительная часть алгоритмов STL работает только с отсортированными интервалами. Библиотека STL содержит разные алгоритмы сортировки, и выбор зависит от того, какой должна быть сортировка: устойчивой, неустойчивой или неполной. Как ни странно, версия с копированием предусмотрена только для неполной сортировки. Если вы используете другой тип сортировки, но при этом хотите работать с копией, вам придется самостоятельно скопировать интервал перед сортировкой.

После сортировки с интервалом можно выполнять разнообразные операции, от простого поиска элементов или их групп до слияния с другим отсортированным интервалом, а также выполнять с элементами операции из теории множеств.

Каждый алгоритм, связанный с сортировкой или операциями с сортированными интервалами, существует в двух версиях. Первая версия использует для определения относительного порядка элементов a и b оператор сравнения < самого объекта, а вторая — операторную функцию operator()(a,b) дополнительного бинарного предиката (объект StrictWeakOrdering). Никаких других различий не существует, поэтому данное обстоятельство не будет особо оговариваться в описании каждого алгоритма.

Сортировка

Алгоритмам сортировки должны передаваться интервалы, ограниченные итераторами произвольного доступа (например, векторы или деки). Контейнер list содержит встроенную функцию sort(), поскольку он поддерживает только двусторонние итераторы.

```
void sort(RandomAccessIterator first, RandomAccessIterator last);
void sort(RandomAccessIterator first, RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

Алгоритм сортирует интервал [first,last) по возрастанию. Первая форма определяет порядок следования элементов при помощи оператора <, а вторая передает элементы заданному предикату.

```
void stable_sort(RandomAccessIterator first,
  RandomAccessIterator last);
void stable_sort(RandomAccessIterator first,
  RandomAccessIterator last, StrictWeakOrdering binary_pred);
```

Алгоритм сортирует интервал [first,last) по возрастанию с сохранением исходного порядка следования эквивалентных элементов (это важно, если элементы могут быть эквивалентными, но не идентичными).

```
void partial_sort(RandomAccessIterator first,
  RandomAccessIterator middle, RandomAccessIterator last);
void partial_sort(RandomAccessIterator first,
  RandomAccessIterator middle, RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

Алгоритм сортирует элементы интервала [first,last), которые могут войти в интервал [first,middle). Остальные элементы оказываются в интервале [middle,last) без гарантированного порядка следования.

```
RandomAccessIterator partial_sort_copy(InputIterator first,
  InputIterator last, RandomAccessIterator result_first,
  RandomAccessIterator result_last);
RandomAccessIterator partial_sort_copy(InputIterator first,
  InputIterator last, RandomAccessIterator result_first,
  RandomAccessIterator result_last, StrictWeakOrdering binary_pred);
```

Алгоритм сортирует элементы интервала [first,last), которые могут войти в интервал [result_first, result_last), и копирует эти элементы в [result_first, result_last). Если интервал [first,last) меньше [result_first, result_last), используется меньшее количество элементов.

```
void nth_element(RandomAccessIterator first,
  RandomAccessIterator nth, RandomAccessIterator last);
void nth_element(RandomAccessIterator first,
  RandomAccessIterator nth, RandomAccessIterator last,
  StrictWeakOrdering binary_pred);
```

Алгоритм nth_element(), как и partial_sort(), частично упорядочивает интервал элементов. Тем не менее, результат получается гораздо «менее упорядочен-

ным». Алгоритм nt_element() гарантирует лишь то, что выбранная *позиция* является точкой разбиения, то есть все элементы в интервале [first,nth) удовлетворяют бинарному предикату (как обычно, по умолчанию используется оператор <), а для всех элементов в интервале [nth,last) это условие не выполняется. Тем не менее, ни в одном из подынтервалов элементы не располагаются в определенном порядке, тогда как алгоритм partial_sort() сортирует первый интервал.

Если такого (очень слабого упорядочения) достаточно, например, при вычислении медиан, процентилей и т. д., этот алгоритм удобнее, так как он работает быстрее алгоритма partial_sort().

Поиск элементов в отсортированных интервалах

Отдельная группа алгоритмов предназначена для поиска элементов в отсортированных интервалах. В следующих описаниях всегда присутствуют две формы. В первой форме при сортировке используется внутренний оператор <, а во второй — объект функции сравнения. При поиске должен применяться тот же способ сравнения, что и при сортировке; в противном случае результат работы алгоритма не определен. Попытка применения этих алгоритмов к несортированным интервалам также приводит к неопределенным результатам.

```
bool binary_search(ForwardIterator first, ForwardIterator last,
  const T& value);
bool binary_search(ForwardIterator first, ForwardIterator last,
  const T& value, StrictWeakOrdering binary_pred);
```

Алгоритм сообщает, присутствует ли заданное значение в сортированном интервале [first,last).

```
ForwardIterator lower_bound(ForwardIterator first,
  ForwardIterator last, const T& value);
ForwardIterator lower_bound(ForwardIterator first,
  ForwardIterator last, const T& value, StrictWeakOrdering binary_pred);
```

Алгоритм возвращает итератор, установленный в позицию первого вхождения value в отсортированный интервал [first,last). Если значение value отсутствует в интервале, возвращается last.

```
ForwardIterator upper_bound(ForwardIterator first,
  ForwardIterator last, const T& value);
ForwardIterator upper_bound(ForwardIterator first,
  ForwardIterator last, const T& value, StrictWeakOrdering binary_pred);
```

Алгоритм возвращает итератор, установленный в позицию за последним вхождением value в отсортированный интервал [first,last). Если значение value отсутствует в интервале, возвращается last.

```
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
  const T& value);
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
  const T& value, StrictWeakOrdering binary_pred);
```

Алгоритм фактически объединяет lower_bound() с upper_bound() и возвращает объект pair с итераторами для позиций первого вхождения и на одну позицию за последним вхождением value в отсортированный интервал [first,last). Если

значение value отсутствует в интервале, оба итератора указывают на ту позицию, в которой оно должно было бы находиться.

Возможно, вас удивит, что алгоритмы бинарного поиска получают прямой итератор вместо итератора произвольного доступа (обычно все объяснения бинарного поиска основаны на индексации). Помните, что итератор произвольного доступа является частным случаем прямого итератора и всегда может использоваться вместо него. Если итератор, переданный одному из этих алгоритмов, действительно поддерживает прямой доступ, имеет место эффективный поиск с логарифмической сложностью, а если нет, — линейный поиск[1].

Следующая программа преобразует каждое слово входных данных в объект NString и включает его в vector<NString>. Затем полученный вектор используется для демонстрации различных алгоритмов сортировки и поиска.

```
//: C06:SortedSearchTest.cpp
// Тестирование поиска в отсортированных интервалах
// NString
#include <algorithm>
#include <cassert>
#include <ctime>
#include <cstdlib>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <iterator>
#include <vector>
#include "NString.h"
#include "PrintSequence.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  typedef vector<NString>::iterator sit;
  char* fname = "test.txt";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  srand(time(0));
  cout.setf(ios::boolalpha);
  vector<NString> original;
  copy(istream_iterator<string>(in),
    istream_iterator<string>(), back_inserter(original));
  require(original.size() >= 4, "Must have four elements");
  vector<NString> v(original.begin(), original.end()),
    w(original.size() / 2);
  sort(v.begin(), v.end());
  print(v.begin(), v.end(), "sort");
  v = original;
  stable_sort(v.begin(), v.end());
  print(v.begin(), v.end(), "stable_sort");
  v = original;
  sit it = v.begin(), it2;
```

[1] Алгоритм может определить тип переданного итератора по его тегу (см. следующую главу).

Слияние отсортированных интервалов

Как и прежде, первая форма каждого алгоритма предполагает, что при сортировке интервала используется внутренний оператор <. Вторая форма применяется в том случае, если сортировка выполнялась специальным объектом функции. При вызове алгоритмов должен быть задействован тот же способ сравнения, что и при сортировке; в противном случае результат выполнения алгоритма не определен. Попытка применить эти алгоритмы к несортированным интервалам также приводит к неопределенным результатам.

```
Outputlterator mergednputlteratorl firstl. InputIteratorl lastl,
  InputIterator2 first2. InputIterator2 last2, Outputlterator result):
Outputlterator mergednputlteratorl firstl. Inputlteratorl lastl.
  InputIterator2 first2, InputIterator2 last2.
  Outputlterator result. StrictWeakOrdering binary_pred):
```

Алгоритм копирует элементы из [firstl,lastl) в result так, что полученный интервал сортируется по возрастанию. После выполнения операции сохраняется относительный порядок следования элементов.

```
void inplace_merge(BidirectionalIterator first.
  Bidirectional Iterator middle. Bidirectional Iterator last):
void inplace_merge(BidirectionalIterator first.
  Bidirectional Iterator middle. Bidirectional Iterator last.
  StrictWeakOrdering binary_pred);
```

Предполагается, что [first,middie) и [middle,last) являются отсортированными последовательностями, образующими один интервал. Алгоритм производит слияние последовательностей таким образом, что полученный интервал [first,last) содержит отсортированную совокупность элементов исходных последовательностей.

Слияние интервалов удобно демонстрировать на примере контейнеров с элементами int. Следующий пример также показывает, что алгоритмы (и наш шаблон print) работают не только с традиционными контейнерами, но и с массивами:

```
//: C06:MergeTest.cpp
// Слияние отсортированных интервалов
//{L} Generators
linclude <algorithm>
linclude "PrintSequence.h"
linclude "Generators.h"
using namespace std:

int mainO {
  const int SZ = 15:
  int a[SZ*2] = {0}:
  // Оба подынтервала принадлежат одному массиву:
  generated, a + SZ. SkipGen(0. 2)):
  a[3] = 4:
  a[4] = 4:
  generated + SZ.a + SZ*2,SkipGend, 3)):
  print(a, a + SZ."rangel", " "):
  print(a + SZ, a + SZ*2, "range2". " "):
  int b[SZ*2] = {0}: // Инициализация всего массива нулями
  merge(a. a + SZ, a+ SZ. a + SZ*2, b):
  print(b. b + SZ*2."merge", " ");
  // Обнуление b
  for(int i = 0: i < SZ*2: i++)
```

```
    b[i] = 0;
  inplace_merge(a, a + SZ, a + SZ*2);
  print(a, a + SZ*2, "inplace_merge", " ");
  int* end = set_union(a, a + SZ, a + SZ, a + SZ*2, b);
  print(b, end, "set_union", " ");
} ///:~
```

В функции main() вместо двух раздельных массивов мы создаем два интервала, расположенных вплотную друг к другу в массиве a (это удобно для алгоритма inplace_merge). Первый вызов merge() заносит результат в другой массив b. Для сравнения также вызывается алгоритм set_union(), который обладает сходной сигнатурой и аналогичным поведением, но с одним отличием: он удаляет дубликаты из второго набора. Наконец, алгоритм inplace_merge() объединяет обе части a.

Теоретико-множественные операции с отсортированными интервалами

С отсортированными интервалами также можно выполнять математические операции из теории множеств.

```
bool includes(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, InputIterator2 last2);
bool includes(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, InputIterator2 last2,
  StrictWeakOrdering binary_pred);
```

Алгоритм возвращает true, если интервал [first2,last2) является подмножеством интервала [first1,last1). Ни один из интервалов не обязан содержать только уникальные элементы, но если интервал [first2,last2) содержит n экземпляров некоторого значения, то интервал [first1,last1) тоже должен содержать минимум n экземпляров этого значения.

```
OutputIterator set_union(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result);
OutputIterator set_union(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result,
  StrictWeakOrdering binary_pred);
```

Алгоритм создает теоретико-множественное объединение двух отсортированных интервалов в интервале result и возвращает конечный итератор полученного интервала. Ни один из исходных интервалов не обязан содержать только уникальные элементы, но если некоторое значение многократно встречается в обоих исходных интервалах, то в объединении будет присутствовать наибольшее из этих двух количеств.

```
OutputIterator set_intersection(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result);
OutputIterator set_intersection(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result,
  StrictWeakOrdering binary_pred);
```

Алгоритм строит в интервале result пересечение двух исходных интервалов и возвращает конечный итератор полученного интервала. Ни один из исходных интервалов не обязан содержать только уникальные элементы, но если не-

которое значение многократно встречается в обоих исходных интервалах, то в объединении будет присутствовать наименьшее из этих двух количеств.

```
OutputIterator set_difference(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result);
OutputIterator set_difference(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result,
  StrictWeakOrdering binary_pred);
```

Алгоритм строит в интервале **result** теоретико-множественную разность двух исходных интервалов и возвращает конечный итератор полученного интервала. В приемный интервал включаются только элементы, присутствующие в интервале [**first1**,**last1**), но отсутствующие в интервале [**first2**,**last2**). Ни один из исходных интервалов не обязан содержать только уникальные элементы, но если некоторое значение многократно встречается в обоих исходных интервалах (**n** в интервале 1 и **m** в интервале 2), то разность будет содержать **max(n-m,0)** экземпляров этого значения.

```
OutputIterator set_symmetric_difference(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result);
OutputIterator set_symmetric_difference(InputIterator1 first1,
  InputIterator1 last1, InputIterator2 first2,
  InputIterator2 last2, OutputIterator result,
  StrictWeakOrdering binary_pred);
```

Алгоритм заносит в интервал **result**:

- все элементы интервала 1, отсутствующие в интервале 2;
- все элементы интервала 2, отсутствующие в интервале 1.

Ни один из исходных интервалов не обязан содержать только уникальные элементы, но если некоторое значение многократно встречается в обоих исходных интервалах (**n** в интервале 1 и **m** в интервале 2), то разность будет содержать **abs(n-m)** экземпляров этого значения. Возвращаемое значение представляет собой конечный итератор полученного интервала.

Теоретико-множественные операции проще всего продемонстрировать на примере вектора с символьными элементами. Программа генерирует случайные символы и сортирует их с сохранением повторяющихся значений. Это сделано для того, чтобы вы видели, как выполняются различные операции при наличии дубликатов в исходных интервалах.

```
//: C06:SetOperations.cpp
// Теоретико-множественные операции с отсортированными интервалами
//{L} Generators
#include <vector>
#include <algorithm>
#include "PrintSequence.h"
#include "Generators.h"
using namespace std;

int main() {
  const int SZ = 30;
  char v[SZ + 1], v2[SZ + 1];
```

```
  CharGen g;
  generate(v, v + SZ, g);
  generate(v2, v2 + SZ, g);
  sort(v, v + SZ);
  sort(v2, v2 + SZ);
  print(v, v + SZ, "v", "");
  print(v2, v2 + SZ, "v2", "");
  bool b = includes(v, v + SZ, v + SZ/2, v + SZ);
  cout.setf(ios::boolalpha);
  cout << "includes: " << b << endl;
  char v3[SZ*2 + 1], *end;
  end = set_union(v, v + SZ, v2, v2 + SZ, v3);
  print(v3, end, "set_union", "");
  end = set_intersection(v, v + SZ,
    v2, v2 + SZ, v3);
  print(v3, end, "set_intersection", "");
  end = set_difference(v, v + SZ, v2, v2 + SZ, v3);
  print(v3, end, "set_difference", "");
  end = set_symmetric_difference(v, v + SZ,
    v2, v2 + SZ, v3);
  print(v3, end, "set_symmetric_difference","");
} ///:~
```

После заполнения, сортировки и вывода векторов v и v2 алгоритм includes() проверяет, содержится ли вторая половина вектора v во всем интервале v. Естественно, это условие заведомо выполняется, поэтому проверка всегда дает положительный результат. Просмотрите результаты вызова алгоритмов set_union(), set_intersection(), set_difference() и set_symmetric_difference() и убедитесь в том, что они работают так, как положено.

Операции с кучей

Кучей называется структура данных, напоминающая массив и используемая при реализации «приоритетной очереди» — интервала с возможностью выборки элементов по приоритету, определяемому некоторой функцией сравнения. В стандартную библиотеку включен набор операций, позволяющих интерпретировать интервал как «кучу», которая всегда обеспечивает эффективную выборку элемента с наибольшим приоритетом без полного упорядочения всего интервала.

Как и в случае с операциями сортировки, каждый алгоритм существует в двух версиях. Первая версия предполагает, что при сравнении используется внутренний оператор <. Вторая форма проверяет условие a<b при помощи операторной функции operator()(a,b) объекта StrictWeakOrdering.

```
void make_heap(RandomAccessIterator first,
  RandomAccessIterator last);
void make_heap(RandomAccessIterator first,
  RandomAccessIterator last, StrictWeakOrdering binary_pred);
```

Алгоритм преобразует произвольный интервал в кучу.

```
void push_heap(RandomAccessIterator first,
  RandomAccessIterator last);
void push_heap(RandomAccessIterator first,
  RandomAccessIterator last, StrictWeakOrdering binary_pred);
```

Алгоритм включает элемент *(last-1) в кучу, определяемую интервалом [first,last-1). Иначе говоря, последний элемент помещается в правильную позицию кучи.

```
void pop_heap(RandomAccessIterator first,
  RandomAccessIterator last);
void pop_heap(RandomAccessIterator first,
  RandomAccessIterator last, StrictWeakOrdering binary_pred);
```

Алгоритм помещает наибольший элемент (который перед выполнением этой операции находится в позиции *first по самому определению кучи) в позицию (last-1) *и* переупорядочивает остальные элементы так, чтобы они сохраняли правильный порядок следования. Если ограничиться простой выборкой с позиции *first, то следующий элемент не будет следующим по величине элементом кучи. По этой причине необходимо использовать алгоритм pop_heap(), чтобы в куче сохранялся правильный порядок следования элементов приоритетной очереди.

```
void sort_heap(RandomAccessIterator first,
  RandomAccessIterator last);
void sort_heap(RandomAccessIterator first,
  RandomAccessIterator last, StrictWeakOrdering binary_pred);
```

Алгоритм может рассматриваться как обратный по отношению к make_heap(): он берет интервал, элементы которого размещены в порядке кучи, и переупорядочивает его в обычном порядке сортировки, так что интервал перестает быть кучей. Это означает, что после вызова sort_heap() для интервала уже нельзя вызывать функции push_heap() и pop_heap() (вернее, можно, но ничего разумного они уже не сделают). Итоговая сортировка не является устойчивой, то есть относительный порядок следования элементов не сохраняется.

Применение операции к каждому элементу интервала

Алгоритмы следующей группы перебирают все элементы интервала и выполняют с каждым элементом некоторую операцию. Они различаются по принципу использования результата этой операции: алгоритм for_each() игнорирует возвращаемые значения, а transform() помещает их в приемный интервал (который может совпадать с исходным интервалом).

```
UnaryFunction for_each(InputIterator first,
  InputIterator last, UnaryFunction f);
```

Алгоритм применяет объект функции f к каждому элементу интервала [first,last). Возвращаемое значение каждого отдельного применения f игнорируется. Если f является обычным указателем на функцию, возвращаемое значение обычно интереса не представляет; но если f является объектом с внутренним состоянием, он может накапливать результаты применения f к элементам интервала. Окончательным возвращаемым значением алгоритма for_each() является f.

```
OutputIterator transform(InputIterator first,
  InputIterator last, OutputIterator result, UnaryFunction f);
OutputIterator transform(InputIterator1 first,
  InputIterator1 last, InputIterator2 first2,
  OutputIterator result, BinaryFunction f);
```

Алгоритм transform(), как и for_each(), применяет объект функции f к каждому элементу интервала [first,last). Но вместо того чтобы игнорировать результаты вызовов функции, transform() копирует их (оператором =) в *result и инкрементирует result после каждого копирования. Интервал, на который ссылается ите-

ратор result, должен содержать достаточное количество элементов. Если вы не уверены в этом, используйте итератор вставки и создавайте новые элементы вместо присваивания.

Первая форма transform() просто вызывает f(*first), где first последовательно перебирает элементы исходного интервала. Вторая форма вызывает f(*first1,*first2) (длина второго интервала определяется длиной первого интервала). Возвращаемое значение в обоих случаях представляет собой конечный итератор полученного интервала.

Применение какой-либо операции ко всем элементам является одной из самых распространенных задач при работе с контейнерами, поэтому описанные в этом разделе алгоритмы достаточно важны и заслуживают нескольких примеров.

Начнем с алгоритма for_each(). Он перебирает содержимое интервала, последовательно берет каждый элемент и передает его объекту функции, переданному при вызове. В принципе for_each() делает то, что вы могли бы запрограммировать вручную; определение for_each() в заголовочном файле компилятора выглядит примерно так:

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f) {
    while (first != last)
      f(*first++);
    return f;
}
```

Следующий пример демонстрирует некоторые нестандартные возможности применения этого алгоритма. Для начала нам потребуется класс, который отслеживает созданные объекты и сообщает об их уничтожении:

```
//: C06:Counted.h
// Класс, отслеживающий свои объекты
#ifndef COUNTED_H
#define COUNTED_H
#include <vector>
#include <iostream>

class Counted {
  static int count;
  char* ident;
public:
  Counted(char* id) : ident(id) { count++; }
  ~Counted() {
    std::cout << ident << " count = "
      << --count << std::endl;
  }
};

class CountedVector :  public std::vector<Counted*> {
public:
  CountedVector(char* id) {
    for(int i = 0; i < 5; i++)
      push_back(new Counted(id));
  }
};
#endif // COUNTED_H ///:~

//: C06:Counter.cpp {O}
#include "Counted.h"
```

```
int Counted::count = 0;
///:~
```

Класс Counted содержит статический счетчик созданных объектов и выводит сообщения об их уничтожении[1]. Каждый объект Counted снабжается идентификатором char*, чтобы за ним было удобнее следить.

Класс CountedVector объявлен производным от vector<Counted*>. В конструкторе он создает несколько объектов Counted и передает каждому из них заданный идентификатор char*. Класс CountedVector существенно упрощает тестирование, о чем наглядно свидетельствует следующий пример:

```
//: C06:ForEach.cpp
// Использование алгоритма STL for_each()
//{L} Counted
#include <algorithm>
#include <iostream>
#include "Counted.h"
using namespace std;

// Объект функции:
template<class T>
class DeleteT {
public:
  void operator()(T* x) { delete x; }
};

// Шаблонная функция:
template <class T> void wipe(T* x) { delete x; }

int main() {
  CountedVector B("two");
  for_each(B.begin(),B.end(),DeleteT<Counted>());
  CountedVector C("three");
  for_each(C.begin(), C.end(), wipe<Counted>);
} ///:~
```

Поскольку вызов delete для всех указателей в контейнере требуется достаточно часто, почему бы не создать для этой цели алгоритм? За основу можно взять transform(). Преимущество transform() перед for_each() заключается в том, что transform() присваивает результат вызова объекта функции элементу приемного интервала, который может совпадать с исходным интервалом. В последнем случае выполняется преобразование исходного интервала, так как каждый элемент заменяется новым значением. В данном случае этот подход особенно уместен, потому что после вызова delete для указателя будет разумно присвоить ему безопасное нулевое значение. Алгоритм transform() позволяет легко решить эту задачу:

```
//: C06:Transform.cpp
// Использование алгоритма STL transform()
//{L} Counted
#include <iostream>
#include <vector>
#include <algorithm>
#include "Counted.h"
using namespace std;
```

[1] Копирующий конструктор и оператор присваивания отсутствуют в классе, поскольку они не задействованы в нашем примере.

```
template<class T> T* deleteP(T* x) { delete x; return 0; }

template<class T> struct Deleter {
  T* operator()(T* x) { delete x; return 0; }
};

int main() {
  CountedVector cv("one");
  transform(cv.begin(), cv.end(), cv.begin(),
    deleteP<Counted>);
  CountedVector cv2("two");
  transform(cv2.begin(), cv2.end(), cv2.begin(),
    Deleter<Counted>());
} ///:~
```

В этой программе продемонстрированы оба подхода: использование шаблонной функции и шаблонного объекта функции. После вызова transform() вектор содержит пять нулевых указателей, что позволяет защититься от повторных вызовов delete.

Однако вам не удастся вызвать delete для каждого указателя в контейнере без инкапсуляции вызова в функции или объекте. Иначе говоря, речь идет о следующей конструкции:

```
for_each(a.begin(), a.end(), ptr_fun(operator delete));
```

Дело в том, что операторная функция operator delete() получает void*, но итератор не является указателем. Впрочем, даже если бы вам удалось заставить компилироваться программу, в итоге получилась бы последовательность вызовов функции освобождения памяти. В отличие от вызова delete для каждого указателя в a, деструкторы вызываться не будут. Обычно это не то, что требуется, поэтому вызовы delete необходимо инкапсулировать.

В предыдущем примере с алгоритмом for_each() возвращаемое значение алгоритма игнорировалось. Этим возвращаемым значением является функция, переданная for_each(). Для обычного указателя на функцию возвращаемое значение особой пользы не приносит, но объект функции может накапливать информацию об объектах, перебираемых в процессе работы for_each(), в своей внутренней переменной.

Для примера рассмотрим простую модель складского учета. Каждый объект Inventory содержит название товара (в нашем примере он представляется одиночным символом), количество единиц на складе и цену одной единицы:

```
//: C06:Inventory.h
#ifndef INVENTORY_H
#define INVENTORY_H
#include <iostream>
#include <cstdlib>
using std::srand;

class Inventory {
  char item;
  int quantity;
  int value;
public:
  Inventory(char it, int quant, int val)
```

```
    : item(it), quantity(quant), value(val) {}
  // Сгенерированный оператор присваивания
  // и копирующий конструктор подходят.
  char getItem() const { return item; }
  int getQuantity() const { return quantity; }
  void setQuantity(int q) { quantity = q; }
  int getValue() const { return value; }
  void setValue(int val) { value = val; }
  friend std::ostream& operator<<(
    std::ostream& os, const Inventory& inv) {
    return os << inv.item << ": "
      << "quantity " << inv.quantity
      << ", value " << inv.value;
  }
};

// Генератор:
struct InvenGen {
    Inventory operator()() {
    static char c = 'a';
    int q = rand() % 100;
    int v = rand() % 500;
    return Inventory(c++, q, v);
  }
};
#endif // INVENTORY_H ///:~
```

Функции класса возвращают название товара, а также возвращают и задают количество и цену. Оператор << выводит объект Inventory в поток ostream. Генератор создает объекты с последовательно сгенерированными названиями, случайными количествами и ценами.

Чтобы подсчитать общее количество единиц товара и общую цену, мы создаем для for_each() объект функции, в переменных которого будет накапливаться нужная информация:

```
//: C06:CalcInventory.cpp
// Пример использования for_each()
#include <algorithm>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

// Вычисление сводной информации:
class InvAccum {
  int quantity;
  int value;
public:
  InvAccum() : quantity(0), value(0) {}
  void operator()(const Inventory& inv) {
    quantity += inv.getQuantity();
    value += inv.getQuantity() * inv.getValue();
  }
  friend ostream&
  operator<<(ostream& os, const InvAccum& ia) {
    return os << "total quantity: "
      << ia.quantity
```

```
      << ", total value: " << ia.value;
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0)); // Randomize
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi.begin(), vi.end(), "vi");
  InvAccum ia = for_each(vi.begin(),vi.end(), InvAccum());
  cout << ia << endl;
} ///:~
```

Операторная функция operator() класса InvAccum вызывается с одним аргументом, как того требует алгоритм for_each(). По мере перебора элементов интервала алгоритм for_each() берет каждый элемент и передает его операторной функции InvAccum::operator(), где производятся вычисления и сохраняется результат. В конце перебора for_each() возвращает объект InvAccum, и содержимое этого объекта выводится на печать.

Многие операции с объектами Inventory могут выполняться алгоритмом for_each(). Например, for_each() позволяет легко поднять все цены на 10 %. Однако стоит обратить внимание, что объекты Inventory не позволяют изменить значение item – программист, разработавший Inventory, решил, что так будет безопаснее. Да и кому может понадобиться изменять название товара? Но вдруг коммерческий отдел решил, что для придания товару «нового, улучшенного вида» нужно преобразовать все названия к верхнему регистру. Специалисты провели исследования, которые показали, что смена названия приведет к росту продаж (надо же коммерческому отделу *хоть что-то* делать...). В этой ситуации алгоритм for_each() не работает, но зато поможет алгоритм transform():

```
//: C06:TransformNames.cpp
// Пример использования transform()
#include <algorithm>
#include <cctype>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct NewImproved {
  Inventory operator()(const Inventory& inv) {
    return Inventory(toupper(inv.getItem()),
      inv.getQuantity(), inv.getValue());
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0));   // Раскрутка генератора случайных чисел
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi.begin(), vi.end(), "vi");
  transform(vi.begin(), vi.end(), vi.begin(),
    NewImproved());
  print(vi.begin(), vi.end(), "vi");
} ///:~
```

Обратите внимание: приемный интервал совпадает с исходным интервалом, то есть преобразование выполняется «на месте».

Теперь предположим, что отдел сбыта захотел сгенерировать специальные ценники с разными скидками по каждой позиции. Исходный список должен остаться без изменений, но при этом нужно сгенерировать несколько специальных списков. Отдел сбыта предоставил список скидок для каждого нового списка. Задача решается при помощи второй формы алгоритма transform():

```
//: C06:SpecialList.cpp
// Пример использования второй формы transform()
#include <algorithm>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct Discounter {
  Inventory operator()(const Inventory& inv,
    float discount) {
    return Inventory(inv.getItem(), inv.getQuantity(),
      int(inv.getValue() * (1 - discount)));
  }
};

struct DiscGen {
  float operator()() {
    float r = float(rand() % 10);
    return r / 100.0;
  }
};

int main() {
  vector<Inventory> vi;
  srand(time(0));  // Раскрутка генератора случайных чисел
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi.begin(), vi.end(), "vi");
  vector<float> disc;
  generate_n(back_inserter(disc), 15, DiscGen());
  print(disc.begin(), disc.end(), "Discounts:");
  vector<Inventory> discounted;
  transform(vi.begin(),vi.end(), disc.begin(),
    back_inserter(discounted), Discounter());
  print(discounted.begin(), discounted.end(), "discounted");
} ///:~
```

Объект функции Discounter строит для заданного объекта Inventory и процента скидки новый объект Inventory со сниженной ценой. Объект функции DiscGen просто генерирует случайные величины скидок от 1 до 10 %, используемые в нашем тесте. В функции main() создаются два вектора: для объектов Inventory и для скидок. Эти объекты передаются алгоритму transform() вместе с объектом Discounter, а алгоритм transform() заполняет новый вектор vector<Inventory> с именем discounted.

Числовые алгоритмы

Числовые алгоритмы определяются в заголовочном файле <numeric>, поскольку они используются в основном для выполнения математических вычислений.

```
T accumulate(InputIterator first, InputIterator last, T result);
T accumulate(InputIterator first, InputIterator last, T result,
  BinaryFunction f);
```

Первая форма реализует обобщенное суммирование. Для каждого элемента [first,last), на который ссылается итератор i, выполняется операция result = result + *i, где result относится к типу T. Вторая форма имеет более общий характер: она применяет функцию f(result,*i) к каждому элементу *i от начала до конца интервала.

Обратите внимание на сходство между вторыми формами алгоритмов transform() и accumulate().

```
T inner_product(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, T init);
T inner_product(InputIterator1 first1, InputIterator1 last1,
  InputIterator2 first2, T init, BinaryFunction1 op1,
  BinaryFunction2 op2);
```

Алгоритм вычисляет обобщенное скалярное произведение двух интервалов [first1,last1) и [first2,first2+(last1-first1)). Возвращаемое значение определяется умножением элементов первого интервала на «параллельные» элементы второго интервала и прибавлением результата к накапливаемой сумме. Таким образом, для двух интервалов {1,1,2,2} и {1,2,3,4} скалярное произведение равно

```
(1*1) + (1*2) + (2*3) + (2*4) = 17
```

Аргумент init содержит начальное значение накапливаемой суммы. Обычно оно равно нулю, но вы можете задать любое другое значение. Начальное значение особенно важно при пустом первом интервале, потому что оно становится возвращаемым значением. Второй интервал должен содержать как минимум не меньше элементов, чем первый.

Вторая форма просто вызывает пару функций для переданных интервалов. Функция op1 используется вместо сложения, а op2 заменяет умножение. Таким образом, вторая версия inner_product() в приведенном примере выполняет следующие операции:

```
init = op1(init, op2(1,1));
init = op1(init, op2(1,2));
init = op1(init, op2(2,3));
init = op1(init, op2(2,4));
```

Происходящее напоминает работу алгоритма transform(), но с выполнением двух операций вместо одной.

```
OutputIterator partial_sum(InputIterator first,
  InputIterator last, OutputIterator result);
OutputIterator partial_sum(InputIterator first,
  InputIterator last, OutputIterator result, BinaryFunction op);
```

Алгоритм вычисляет обобщенную частичную сумму и создает новый интервал, начинающийся с result. Значение n-го элемента этого интервала равно сумме всех элементов интервала [first,last) от 1 до n. Например, для исходной последовательности {1,1,2,2,3} генерируется последовательность {1,1 + 1,1 + 1 + + 2,1 + 1 + 2 + 2,1 + 1 + 2 + 2 + 3}, то есть {1,2,4,6,9}.

Во второй форме алгоритма вместо оператора + используется бинарная функция op. Она получает накопленное значение и объединяет его с текущим элементом. Например, если задействовать объект multiplies<int>() с предшествую-

щим интервалом, вы получите интервал {1,1,2,4,12}. Обратите внимание: первый элемент в выходном интервале всегда равен первому элементу исходного интервала.

Возвращаемое значение представляет собой конечный итератор выходного интервала [result,result+(last-first)).

```
OutputIterator adjacent_difference(InputIterator first,
  InputIterator last, OutputIterator result);
OutputIterator adjacent_difference(InputIterator first,
  InputIterator last, OutputIterator result, BinaryFunction op);
```

Алгоритм вычисляет разность между соседними элементами в интервале [first,last). Это означает, что каждый элемент нового интервала равен разности текущего и предыдущего элементов исходного интервала (первый элемент остается без изменений). Например, для исходного интервала {1,1,2,2,3} будет получен интервал {1,1 - 1,2 - 1,2 - 2,3 - 2}, то есть {1,0,1,0,1}.

Во второй форме алгоритма вместо оператора – используется бинарная функция op. Например, если задействовать объект multiplies<int>() с предшествующим интервалом, вы получите интервал {1,1,2,4,6}. Обратите внимание: первый элемент в выходном интервале всегда равен первому элементу исходного интервала.

Возвращаемое значение представляет собой конечный итератор выходного интервала [result,result+(last-first)).

Следующая программа тестирует все алгоритмы <numeric> в обеих формах на примере целочисленных массивов. В тестах второй формы (с передачей функции) всегда используется объект функции, эквивалентный операции в первой форме, поэтому и результаты всегда одинаковы. Такой подход поможет вам лучше понять, какие операции при этом выполняются, и как заменить их собственными операциями.

```
//: C06:NumericTest.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
#include <numeric>
#include "PrintSequence.h"
using namespace std;

int main() {
  int a[] = { 1, 1, 2, 2, 3, 5, 7, 9, 11, 13 };
  const int ASZ = sizeof a / sizeof a[0];
  print(a, a + ASZ, "a", " ");
  int r = accumulate(a, a + ASZ, 0);
  cout << "accumulate 1: " << r << endl;
  // С тем же результатом:
  r = accumulate(a, a + ASZ, 0, plus<int>());
  cout << "accumulate 2: " << r << endl;
  int b[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2 };
  print(b, b + sizeof b / sizeof b[0], "b", " ");
  r = inner_product(a, a + ASZ, b, 0);
  cout << "inner_product 1: " << r << endl;
  // С тем же результатом:
  r = inner_product(a, a + ASZ, b, 0,
    plus<int>(), multiplies<int>());
```

```
  cout << "inner_product 2: " << r << endl;
  int* it = partial_sum(a, a + ASZ, b);
  print(b, it, "partial_sum 1", " ");
  // С тем же результатом:
  it = partial_sum(a, a + ASZ, b, plus<int>());
  print(b, it, "partial_sum 2", " ");
  it = adjacent_difference(a, a + ASZ, b);
  print(b, it, "adjacent_difference 1"," ");
  // С тем же результатом:
  it = adjacent_difference(a, a + ASZ, b, minus<int>());
  print(b, it, "adjacent_difference 2"," ");
} ///:~
```

Обратите внимание: алгоритмы inner_product() и partial_sum() возвращают конечный итератор полученного интервала, который используется во втором аргументе функции print().

Поскольку вторая форма каждого алгоритма позволяет задать собственный объект функции, только первые формы являются «чисто числовыми». Алгоритм inner_product() позволяет выполнять операции, не являющиеся числовыми в общепринятом смысле.

Вспомогательные алгоритмы

Ниже описаны некоторые базовые алгоритмы, используемые в работе других алгоритмов. Трудно сказать, пригодятся ли они вам в работе.

```
// Заголовочный файл <utility>
template<class T1, class T2> struct pair;
template<class T1, class T2> pair<T1,T2>
  make_pair(const T1&, const T2&);
```

Эти шаблоны упоминались и использовались ранее в этой главе. Объект pair просто упаковывает два объекта (которые могут относиться к разным типам) в один объект. Чаще всего он требуется тогда, когда функция должна вернуть более одного объекта, но у него есть и другие применения, например, в контейнерах или при передаче двух объектов в одном аргументе функции. Компоненты пары определяются записью p.first и p.second, где p — объект pair. Например, функция equal_range(), описанная в этой главе, возвращает свой результат в виде пары итераторов. Объекты pair можно напрямую сохранять в отображениях (map) и мультиотображениях (multimap); для этих контейнеров pair является типом значения элементов (value_type).

Для создания вспомогательных объектов pair вместо явного конструирования обычно применяется шаблонная функция make_pair(). Эта функция автоматически определяет типы полученных аргументов, избавляя вас от лишнего ввода (а заодно повышая надежность программы).

```
// Заголовочный файл <iterator>
difference_type distance(InputIterator first, InputIterator last);
```

Возвращает расстояние между first и last в элементах. Точнее, этот алгоритм возвращает целое количество инкрементов итератора first, после которых тот станет равным last. Разыменование итераторов при этом не производится.

```
// Заголовочный файл <iterator>
void advance(InputIterator& i, Distance n);
```

Перемещает итератор i вперед на величину n (двусторонние итераторы также могут смещаться назад при отрицательных значениях n). Алгоритм учитывает фактическую категорию итератора и выбирает наиболее эффективный способ перемещения. Например, итераторы произвольного доступа увеличиваются прибавлением целочисленного смещения (i += n), а двусторонние итераторы последовательно инкрементируются n раз.

```
// Заголовочный файл <iterator>
back_insert_iterator<Container>
  back_inserter(Container& x);
front_insert_iterator<Container>
  front_inserter(Container& x);
insert_iterator<Container>
  inserter(Container& x, Iterator i);
```

Эти функции применяются для создания итераторов, которые при использовании оператора = вставляют новые элементы в контейнер (вместо того чтобы изменять значения существующих элементов). Для разных типов итераторов требуются разные операции вставки: для back_insert_iterator — push_back(), для front_insert_iterator — push_front(), для insert_iterator — insert() (поэтому последняя функция может вызываться с ассоциативными контейнерами, а первые две — с последовательными). Более подробная информация о контейнерах и вставке элементов приводится в следующей главе.

```
const LessThanComparable& min(const LessThanComparable& a,
  const LessThanComparable& b);
const T& min(const T& a, const T& b, BinaryPredicate binary_pred);
```

Возвращает меньший из двух аргументов или первый аргумент, если аргументы равны. Первая версия выполняет сравнение оператором <, а вторая передает оба аргумента бинарному предикату binary_pred.

```
const LessThanComparable& max(const LessThanComparable& a,
  const LessThanComparable& b);
const T& max(const T& a, const T& b, BinaryPredicate binary_pred);
```

Работает аналогично min(), но возвращает больший из двух аргументов.

```
void swap(Assignable& a, Assignable& b);
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Меняет местами a и b, используя присваивание. Учтите, что во всех классах контейнеров определены специализированные версии swap(), которые обычно превосходят обобщенную версию по эффективности.

Функция iter_swap() меняет местами значения, на которые ссылаются аргументы.

Создание пользовательских алгоритмов

Освоившись с алгоритмами STL, вы можете заняться созданием собственных обобщенных алгоритмов. Если эти алгоритмы будут соответствовать правилам, установленным для всех алгоритмов STL, с ними смогут легко работать программисты, знакомые с STL, и ваши алгоритмы станут своего рода расширением инструментария STL.

Проще всего начать с заголовочного файла <algorithm>, найти что-нибудь похожее на то, что вам нужно, и построить свой код по готовому образцу[1] (практически во всех реализациях STL код шаблонов определяется непосредственно в заголовочных файлах).

Если внимательнее изучить список алгоритмов стандартной библиотеки C++, можно заметить очевидное упущение: в нем отсутствует алгоритм copy_if(). Хотя желаемого эффекта можно добиться при помощи алгоритма remove_copy_if(), это не так удобно, поскольку условие приходится инвертировать (вспомните: алгоритм remove_copy_if() копирует только те элементы, которые *не соответствуют* его предикату).

Напрашивается очевидное решение — инвертировать предикат адаптером not1 перед тем, как передавать его алгоритму remove_copy_if():

```
// Pred -- исходное условие
replace_copy_if(begin, end, not1(pred));
```

Выглядит вполне разумно, но... Если мы хотим, чтобы наше решение поддерживало предикаты, которые являются указателями на обычные функции, становится понятно, почему оно не работает — адаптер not1 ожидает получить адаптируемый объект функции. Остается лишь написать алгоритм copy_if() «с нуля». Изучение других алгоритмов копирования показывает, что на концептуальном уровне следует использовать разные итераторы для ввода и вывода, поэтому решение может выглядеть так:

```
//: C06:copy_if.h
// Написание собственных алгоритмов в стиле STL
#ifndef COPY_IF_H
#define COPY_IF_H

template<typename ForwardIter,
  typename OutputIter, typename UnaryPred>
OutputIter copy_if(ForwardIter begin, ForwardIter end,
  OutputIter dest, UnaryPred f) {
  while(begin != end) {
    if(f(*begin))
      *dest++ = *begin;
    ++begin;
  }
  return dest;
}
#endif // COPY_IF_H ///:~
```

Обратите внимание: инкремент begin не может интегрироваться в выражение копирования.

Итоги

Эта глава дает читателю практическое представление об алгоритмах стандартной библиотеки шаблонов. Иначе говоря, вы должны знать алгоритмы STL и разбираться в них достаточно хорошо, чтобы регулярно их использовать (или хотя бы учитывать возможность их применения, чтобы при необходимости вернуться к этой

[1] Конечно, без нарушения авторских прав.

главе и найти подходящее решение). Мощь библиотеки STL обусловлена не только относительной полнотой ее инструментария, но и тем, что она формирует концептуальный подход к решению задач и созданию дополнительных инструментов.

Хотя в этой главе приводятся примеры пользовательских расширений STL, мы не стали подробно излагать весь теоретический материал, необходимый для полного понимания многочисленных тонкостей STL (хотя это позволило бы вам создавать программы гораздо совершеннее тех, которые рассматривались в книге). Отчасти данное упущение объясняется нехваткой места, но это не главное. Основная причина все же заключается в том, что полное изложение материала выходит за рамки темы настоящей главы — мы лишь стремились дать практическое представление об STL, которое бы пригодилось вам в повседневной работе.

Существует немало книг, полностью посвященных STL. Мы особенно рекомендуем вам книгу Скотта Мейерса «Эффективное использование STL. Библиотека программиста», вышедшую в издательстве «Питер» в 2003 году.

Упражнения

1. Напишите генератор, который бы возвращал текущее значение clock() (из заголовка <ctime>). Создайте контейнер list<clock_t> и заполните его своим генератором, используя алгоритм generate_n(). Удалите все дубликаты из списка и выведите его в cout при помощи алгоритма copy().
2. Используя алгоритмы transform() и toupper() (из <cctype>), напишите вызов функции, который бы преобразовывал все символы строки к верхнему регистру.
3. Создайте объект функции Sum, который бы накапливал сумму всех элементов в интервале при использовании с алгоритмом for_each().
4. Напишите генератор анаграмм, который получает слово в командной строке и строит все возможные перестановки букв.
5. Напишите программу, которая получает предложение в командной строке и строит все возможные перестановки (сами слова при этом остаются без изменений, меняется лишь их порядок).
6. Создайте иерархию классов, состоящую из базового класса B и производного класса D. Определите в B виртуальную функцию void f(), которая выводит сообщение о вызове f() класса B. Переопределите эту функцию в D так, чтобы она выводила другое сообщение. Создайте вектор vector<B*>, заполните его объектами B и D, и при помощи алгоритма for_each() вызовите f() для каждого объекта в векторе.
7. Измените программу FunctionObjects.cpp так, чтобы вместо int в ней использовался тип float.
8. Измените программу FunctionObjects.cpp и оформите основной код тестирования в виде шаблона, чтобы вы могли выбрать тестируемый тип (большую часть кода main() придется выделить в отдельную шаблонную функцию).
9. Напишите программу, которая получает целое число в аргументе командной строки и находит все его множители.

10. Напишите программу, которая получает из командной строки имя тестового файла. Откройте файл и прочитайте его по словам (используйте оператор >>). Сохраните слова в vector<string>. Преобразуйте все слова к нижнему регистру, отсортируйте их, удалите дубликаты и выведите результат.
11. Напишите программу для поиска всех слов, присутствующих одновременно в двух входных файлах (используйте алгоритм set_intersection()). Затем измените ее так, чтобы она выводила все слова, присутствующие только в одном из файлов (используйте алгоритм set_symmetric_difference()).
12. Напишите программу, которая строит таблицу факториалов до числа, заданного в командной строке (включительно). Для этого напишите генератор, заполняющий вектор vector<int>, а затем воспользуйтесь алгоритмом partial_sum() со стандартным объектом функции.
13. Измените программу CalcInventory.cpp для поиска всех объектов, количество которых меньше заданного порога, переданного в командной строке. Используйте алгоритмы copy_if() и bind2nd() для построения набора всех значений, меньших заданного.
14. Сгенерируйте 100 чисел функцией UrandGen() (размер неважен). Определите, какие числа в этом интервале дают одинаковый остаток при делении на 23. Выберите случайное число и определите, входит ли оно в этот интервал; для этого разделите каждый элемент интервала на это число и проверьте, равен ли результат 1 (вместо простого поиска алгоритмом find()).
15. Заполните вектор vector<double> числами, представляющими углы в радианах. Используя механизм композиции объектов функций, вычислите синусы всех элементов вектора (см. файл <cmath>).
16. Оцените быстродействие вашего компьютера. Вызовите функцию srand(time(0)), создайте массив случайных чисел. Снова вызовите функцию srand(time(0)) и сгенерируйте такое же количество случайных чисел во втором массиве. При помощи алгоритма equal() проверьте, совпадает ли содержимое массивов (если ваш компьютер достаточно быстр, time(0) вернет одинаковые значения при обоих вызовах). Если массивы не совпадают, отсортируйте их и найдите различия алгоритмом mismatch(). Если их содержимое идентично, увеличьте длину массивов и попробуйте снова.
17. Напишите алгоритм transform_if() в стиле STL. Возьмите за образец первую форму transform(), которая преобразует только объекты, удовлетворяющие унарному предикату. Объекты, не удовлетворяющие предикату, исключаются из результата. Алгоритм должен возвращать новый конечный итератор.
18. Создайте алгоритм в стиле STL, который бы представлял собой перегруженную версию алгоритма for_each(). Алгоритм должен работать по образцу второй формы transform(): он должен получать два входных интервала и передавать объекты второго интервала бинарной функции, применяющей их к объектам первого интервала.
19. Создайте шаблон класса Matrix на базе вектора vector<vector<T>>. Определите дружественную операторную функцию ostream& operator<<(ostream&,const

Matrix&) для вывода матрицы. Создайте следующие бинарные операторные функции, по возможности используя объекты функций STL: operator+(const Matrix&,const Matrix&) для сложения матриц, operator*(const Matrix&,const vector<int>&) для умножения матрицы на вектор, operator*(const Matrix&,const Matrix&) для умножения матриц (если вы забыли, как выполняются эти операции, посмотрите в учебнике). Протестируйте шаблон Matrix для типов int и float.

20. Постройте шифр из символов

```
"~`!@#$%^&*()_-+={[]|\:;"'<,>.?/".
```

Входной файл, имя которого передается в командной строке, используется как словарь. Не обращайте внимания на неалфавитные символы и на регистр слов в файле. Каждому слову должна соответствовать уникальная перестановка символов шифра:

```
"=')/%[}]|{*@?!"`,:>&^-~_:$+.#(<\"   apple
"|]\~>#.+%(/-_[`':;={*"$^!&?),@<"    carrot
"@=~['].\/<-`>#*)^%+.":&?!_{:|$}("   Carrot
```

Проследите за тем, чтобы в шифре не было одинаковых кодов или слов. Для сортировки шифров используйте алгоритм lexicographical_compare(). Зашифруйте исходный файл словаря, затем расшифруйте его и убедитесь, что вы получаете тот же текст.

21. Имеется список имен:

```
Jon Brittle
Jane Brittle
Mike Brittle
Sharon Brittle
George Jensen
Evelyn Jensen
```

Постройте все возможные комбинации разнополых пар.

22. Используя данные предыдущего упражнения, найдите все возможные комбинации разнополых пар, при которых Jon Brittle и Jane Brittle будут находиться в одной паре.

23. Транспортная компания хочет узнать среднюю продолжительность пересечения континента. Проблема состоит в том, что некоторые путешественники совершали пересадки, и дорога заняла у них гораздо больше времени, чем обычно. Используя приведенный ниже генератор, заполните вектор со статистическими данными. Удалите из вектора все аномальные маршруты при помощи алгоритма remove_if(). Вычислите среднее арифметическое элементов вектора и определите, сколько времени в среднем занимает путь.

```
int travelTime() {
  // Аномальный маршрут
  if(rand() % 10 == 0)
    return rand() % 100;
  // Обычный маршрут
  return rand() % 10 + 10;
}
```

24. Определите, насколько алгоритм binary_search() превосходит по быстродействию алгоритм find() при поиске в *отсортированных* интервалах.

25. В армию набираются на сборы военнослужащие запаса. Решено в первую очередь призвать тех, кто уходил на службу в 1997 году, от младших к старшим. Сгенерируйте вектор со случайным списком людей (определите в классе переменные, содержащие возраст и год призыва). Произведите разбиение вектора так, чтобы все призванные в 1997 году были сгруппированы в начале списка от младших к старшим. Остальная часть списка должна быть отсортирована по возрасту.
26. Создайте класс Town с информацией о городе: население, высота над уровнем моря, погодные условия (перечисляемый тип {RAINY,SNOWY,CLOUDY,CLEAR}). Создайте класс, генерирующий объекты Town. Сгенерируйте названия городов (пусть даже это будут бессмысленные последовательности символов) или найдите подходящий список в Интернете. Проследите за тем, чтобы названия были записаны в нижнем регистре, а список не содержал повторяющихся названий. Создайте генератор, который будет случайным образом генерировать погодные условия, численность населения в интервале [100, 1000000) и высоту над уровнем моря (в футах) в интервале [0,8000). Заполните вектор объектами Town, запишите его в новый файл с именем Towns.txt.
27. В результате демографического взрыва население во всех городах увеличилось на 10 %. Обновите данные городов алгоритмом transform() и запишите данные обратно в файл.
28. Найдите города с наибольшим и наименьшим населением. Для этого определите в классе Town оператор <. Также реализуйте функцию, которая бы возвращала true, если первый параметр меньше второго. Используйте ее в качестве предиката выбранного алгоритма.
29. Найдите все города, расположенные на высоте 2500–3500 футов над уровнем моря включительно. Реализуйте в классе Town необходимые операторы сравнения.
30. Упорядочьте список городов по высоте над уровнем моря так, чтобы в нем не было дубликатов (то есть двух городов, различающихся по высоте менее чем на 100 футов). Отсортируйте список по возрастанию минимум двумя разными способами, используя объекты <functional>. Сделайте то же самое для сортировки по убыванию. Реализуйте необходимые операторы сравнения для объектов Town.
31. Сгенерируйте произвольное количество случайных чисел в массиве. Найдите наибольший элемент массива при помощи алгоритма max_element(), поменяйте его местами с последним элементом массива. Найдите следующее по величине число и поместите его в предпоследнюю позицию. Продолжайте до тех пор, пока не будут перемещены все элементы. После завершения работы алгоритма вы получите отсортированный массив (это называется «сортировка методом выбора»).
32. Напишите программу, которая читает номера телефонов из файла (также содержащего имена и другие сведения) и заменяет префикс 222 на 863. Старые номера также следует сохранить. Номера хранятся в следующем формате:

```
222 8945
756 3920
222 8432
```

и т. д.

33. Напишите программу, которая находит в списке всех абонентов с заданной фамилией при помощи интервальных алгоритмов lower_bound, upper_bound и equal_range. Отсортируйте список по фамилии (первичный ключ) и имени (вторичный ключ). Предполагается, что имена и номера телефонов читаются из файла следующего формата:

```
Jonn Doe        345 9483
Nick Bonham     349 2930
Jane Doe        283 2819
```

34. Имеется файл с сокращенными и полными названиями американских штатов:

```
ALABAMA
AL
AK
ALASKA
ARIZONA
AZ
ARKANSAS
AR
CA
CALIFORNIA
CO
COLORADO
```

и т. д.

Выделите все сокращения и запишите их в отдельный файл (учтите, что тип данных не определяется номером строки; полное и сокращенное название могут следовать в любом порядке).

35. Создайте класс Employee, представляющий работника, с двумя переменными: hours (количество отработанных часов) и hourlyPay (почасовая оплата). В классе также определяется функция calcSalary(), возвращающая суммарный заработок данного работника. Сгенерируйте случайные значения hours и hourlyPay для произвольного количества работников, сохраните эти данные в контейнере vector<Employee*>. Вычислите суммарные расходы компании на оплату труда за данный период.

36. Проведите сравнительный хронометраж алгоритмов sort(), partial_sort() и nth_element(). Оправдывает ли экономия времени применение более слабой сортировки, если вам не требуется большего?

Обобщенные контейнеры

Контейнерные классы предназначены для решения целой категории задач, связанных с многократным использованием программного кода. Контейнеры применяются в качестве компонентов, заметно упрощая разработку объектно-ориентированных программ.

Контейнерный класс описывает объект, предназначенный для хранения других объектов. Контейнеры играют настолько важную роль, что в ранних объектно-ориентированных языках они считались фундаментальными структурами данных. Например, в Smalltalk язык с точки зрения программиста представляет собой совокупность транслятора и библиотеки классов, а набор контейнерных классов является важнейшей частью этой библиотеки. Вполне естественно, что разработчики компиляторов C++ также прилагают к своим продуктам библиотеку контейнерных классов. Кстати, весьма удобный класс vector в простейшей форме был представлен в начале первого тома книги.

Первые библиотеки контейнерных классов, как и многие другие ранние библиотеки C++, были основаны на *объектно-базированной иерархии* Smalltalk. Для Smalltalk такое решение работало хорошо, но в C++ оно оказалось громоздким и неудобным. Требовался другой подход.

Контейнеры C++ реализуются в виде шаблонов. Контейнеры стандартной библиотеки C++ предоставляют широкий ассортимент структур данных, хорошо работают со стандартными алгоритмами и подходят для решения многих типовых задач из области программирования.

Контейнеры и итераторы

Если вы не знаете, сколько объектов потребуется для решения той или иной задачи и как долго они будут существовать, вам не удастся заранее выяснить, как хранить эти объекты. Как узнать, сколько памяти для них выделить? Ответ на этот вопрос становится известным лишь на стадии выполнения.

Большинство задач в объектно-ориентированных архитектурах решается просто — определением нового типа объекта. Для нашей задачи этот новый тип объекта должен содержать другие объекты или указатели на них. В C++ этот объект, обычно называемый контейнером (в других языках также используется термин «коллекция»), автоматически расширяется по мере необходимости для хранения всех содержащихся в нем объектов. Вам не нужно заранее знать, сколько объектов будет помещено в контейнер, достаточно создать объект контейнерного класса, а об остальном он позаботится сам.

К счастью, хорошие объектно-ориентированные языки поставляются со своей библиотекой контейнеров. В C++ это стандартная библиотека шаблонов (STL). Разработчики некоторых библиотек сочли, что на все случаи жизни достаточно одного обобщенного контейнера; разработчики других (и особенно это касается библиотек C++) предусмотрели разные типы контейнеров для разных целей: вектор (vector) обеспечивает эффективный доступ к любому элементу, связанный список (list) — эффективную вставку в произвольной позиции, и т. д. Программист выбирает ту разновидность контейнера, которая лучше всего отвечает его требованиям.

Во всех контейнерах предусмотрены средства для записи и удаления элементов. С записью элементов все достаточно очевидно — задача решается при помощи функции с именем push, add или что-нибудь в этом роде. С выборкой элементов дело обстоит сложнее; для контейнеров, являющихся аналогами массивов (например, для векторов) выборка может производиться оператором индексирования или функцией, но во многих ситуациях это не имеет смысла. Кроме того, одиночная выборка устанавливает слишком жесткие ограничения. Что, если потребуется сравнить или обработать группу элементов контейнера?

Гибкий доступ к элементам обеспечивается при помощи *итераторов* — специальных объектов, предназначенных для выборки элементов контейнера. Оформление итератора в виде класса означает дополнительный уровень абстракции, позволяющий отделить детали реализации контейнера от реализации доступа к элементам контейнера. При обращении через итератор контейнер рассматривается как последовательность элементов. Итератор позволяет перебирать элементы, не беспокоясь о базовой структуре контейнера, будь то вектор, связанный список, множество или что-нибудь еще. Это дает возможность легко изменять базовую структуру данных без нарушения кода, перебирающего элементы контейнера. Отделение средств перебора от логики контейнера позволяет использовать несколько итераторов одновременно.

С точки зрения архитектуры все, что вам действительно нужно, — это интервал, с которым можно выполнять необходимые операции. Если бы существовала одна разновидность интервалов, подходящая для любой ситуации, создавать разные типы было бы незачем. Разные контейнеры необходимы по двум причинам. Во-первых, контейнеры предоставляют разные интерфейсы и обладают разным поведением. Так, стек по своему поведению и интерфейсу отличается от очереди, которая, естественно, отличается от списка. Одна разновидность контейнера может лучше подойти для решения вашей конкретной задачи, или предоставляемая ей абстракция может лучше выражать ваши намерения при проектировании. Контейнеры различаются по эффективности выполнения некоторых операций. Для примера сравним вектор со списком. Оба являются простыми последовательными

контейнерами, обладающими практически одинаковыми интерфейсами и внешним поведением, но некоторые операции выполняются с принципиально разной сложностью. Например, произвольный доступ к элементам вектора требует постоянной сложности, то есть занимает одинаковое время независимо от элемента. Тогда как в связанных списках обращение к элементу требует перемещения по списку и обходится дороже для элементов, расположенных дальше от начала списка. Однако вставка элемента в середину списка выполняется быстрее, чем в середину вектора. Эффективность этих и других операций зависит от базовой структуры данных контейнера. На стадии проектирования можно начать со списка, а в процессе оптимизации быстродействия переключиться на вектор, или наоборот. Благодаря итераторам код перебора изолируется от изменений в реализации базовой структуры данных.

Помните, что контейнер представляет собой простое хранилище для объектов. Если это хранилище отвечает всем вашим потребностям, наверное, не так уж важно, *как* оно реализовано. В программной среде с изначальными затратами, обусловленными другими факторами, различия в эффективности вектора и связанного списка могут оказаться несущественными. Возможно, вам удастся обойтись одним типом контейнера. Можно даже представить себе «идеальную» абстракцию контейнера, которая автоматически выбирает свою базовую реализацию в зависимости от контекста использования[1].

Эта глава, как и предыдущая, не содержит полной документации по всем функциям всех контейнеров STL. Хотя в книге приводятся описания функций, используемых в примерах, за полной информацией следует обращаться к другим источникам. Мы рекомендуем электронную документацию реализаций STL от Dinkumware, Silicon Graphics и STLPort[2].

Первое знакомство

В следующем примере используется множество (шаблон класса set) — контейнер, построенный по образцу классических математических множеств, которые не могут содержать одинаковые значения. В данном примере множество предназначено для работы со значениями типа int:

```
//: C07:Intset.cpp
// Простой пример использования множеств STL
#include <cassert>
#include <set>
using namespace std;

int main() {
  set<int> intset;
  for(int i = 0; i < 25; i++)
     for(int j = 0; j < 10; j++)
        // Попытка вставить дубликат:
        intset.insert(j);
  assert(intset.size() == 10);
} ///:~
```

[1] Частный случай эталона Состояние, описанного в главе 10.

[2] См. http://www.dinkumware.com, http://ww.sgi.com/tech/stl и http://ww.stlport.org.

Функция insert() пытается вставить элемент в контейнер; если элемент с таким значением уже присутствует, попытка игнорируется. Чаще всего операции с множествами ограничиваются вставкой и проверкой наличия элемента в множестве. Кроме того, можно вычислить объединение, пересечение или разность двух множеств, а также проверить, является ли одно множество подмножеством другого. В нашем примере значения 0–9 вставляются в множество 25 раз, но принимаются только 10 уникальных значений.

Теперь попробуем приспособить программу Intset.cpp для вывода списка слов, использованных в документе. Решение оказывается на удивление простым:

```
//: C07:WordSet.cpp
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

void wordSet(char* fileName) {
  ifstream source(fileName);
  assure(source, fileName);
  string word;
  set<string> words;
  while(source >> word)
    words.insert(word);
  copy(words.begin(), words.end(),
    ostream_iterator<string>(cout, "\n"));
  cout << "Number of unique words:"
    << words.size() << endl;
}

int main(int argc, char* argv[]) {
  if(argc > 1)
    wordSet(argv[1]);
  else
    wordSet("WordSet.cpp");
} ///:~
```

Единственное принципиальное различие состоит в том, что в новой версии программы множество содержит строки вместо целых чисел. Слова читаются из файла, но остальные операции сходны с теми, которые использовались в программе Intset.cpp. В выходных данных отсутствуют дубликаты, а вследствие особенностей реализации множеств слова автоматически сортируются.

Множество относится к числу *ассоциативных контейнеров* — одной из трех категорий контейнеров стандартной библиотеки C++. Эти категории и входящие в них контейнеры перечислены ниже:

- последовательные контейнеры — vector, list, deque;
- контейнерные адаптеры — queue, stack, priority_queue;
- ассоциативные контейнеры — set, map, multiset, multimap.

Категории контейнеров представляют разные модели, используемые для разных целей. Последовательные контейнеры, обладающие линейной организацией элементов, составляют наиболее важный тип контейнеров. Иногда последователь-

ность элементов требуется наделить особыми свойствами, именно для этой цели предназначены контейнерные адаптеры — они моделируют такие абстрактные структуры данных, как очередь и стек. В ассоциативных контейнерах данные упорядочиваются по ключу, что обеспечивает быструю выборку данных.

Все контейнеры стандартной библиотеки содержат *копии* сохраняемых объектов и расширяются по мере необходимости, поэтому для объектов должны быть доступны копирующий конструктор и оператор присваивания. Контейнеры различаются прежде всего по особенностям хранения объектов в памяти и набору операций, доступных для пользователя.

Вектор, как вы уже знаете, представляет собой линейную последовательность с произвольным доступом к элементам. Тем не менее, вставка элемента в середину контейнера, хранящегося в памяти в виде непрерывного блока, обходится достаточно дорого; это относится и к векторам, и к массивам. Дек (deque, сокращение от «double-ended queue», то есть «двусторонняя очередь») также обеспечивает произвольный доступ, почти не уступающий векторам по скорости, но он гораздо быстрее работает при выделении новой памяти. Кроме того, дек обеспечивает быструю вставку новых элементов в начало и конец последовательности. Список (list) является двусвязным, поэтому перебор элементов в нем происходит медленно, но зато быстро выполняется вставка элемента в произвольной позиции. Таким образом, список, дек и вектор обладают сходной базовой функциональностью (все эти контейнеры содержат линейные последовательности элементов), но различаются по сложности выполнения операций. Если вы только осваиваете программирование с применением контейнеров, выберите один тип контейнера и пользуйтесь им, экспериментируя с остальными типами только на стадии оптимизации программы.

Для большинства задач, которые вам предстоит решать, простых последовательных контейнеров (векторов, деков и списков) будет вполне достаточно. Все три разновидности содержат функцию push_back(), используемую для вставки новых элементов в конец последовательности (у деков и списков также имеется функция push_front(), предназначенная для вставки элементов в начало последовательности).

Но как производится выборка элементов в последовательном контейнере? При работе с деками и векторами можно воспользоваться оператором индексирования [ ], но со списками этот способ не работает. Для обращения к элементам всех трех разновидностей последовательных контейнеров можно применять итераторы. Каждый контейнер предоставляет соответствующий тип итератора для обращения к своим элементам.

Хотя объекты хранятся в контейнерах по значению (то есть в контейнере сохраняется копия всего объекта), в некоторых ситуациях бывает удобнее хранить указатели на объекты. Такой подход позволяет заменять объекты в пределах иерархии и учитывать полиморфное поведение классов. Рассмотрим классический пример с геометрическими фигурами, для которых определен общий набор операций, а также операции, специфические для каждого конкретного класса. В следующей программе вектор STL содержит указатели на разные виды объектов Shape, созданных в куче:

```
//: C07:Stlshape.cpp
// Работа с классами геометрических фигур в STL
#include <vector>
#include <iostream>
using namespace std;
```

```
class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};

class Circle : public Shape {
public:
  void draw() { cout << "Circle::draw\n"; }
  ~Circle() { cout << "~Circle\n"; }
};

class Triangle : public Shape {
public:
  void draw() { cout << "Triangle::draw\n"; }
  ~Triangle() { cout << "~Triangle\n"; }
};

class Square : public Shape {
public:
  void draw() { cout << "Square::draw\n"; }
  ~Square() { cout << "~Square\n"; }
};

typedef std::vector<Shape*> Container;
typedef Container::iterator Iter;

int main() {
  Container shapes;
  shapes.push_back(new Circle);
  shapes.push_back(new Square);
  shapes.push_back(new Triangle);
  for(Iter i = shapes.begin();
      i != shapes.end(); i++)
    (*i)->draw();
  // ... Завершение работы с фигурами:
  for(Iter j = shapes.begin();
      j != shapes.end(); j++)
    delete *j;
} ///:~
```

Определения классов Shape, Circle, Square и Triangle должны выглядеть вполне знакомо. Абстрактный базовый класс Shape (о чем говорит спецификатор =0) определяет интерфейс для всех разновидностей фигур. Производные классы переопределяют виртуальную функцию draw() для выполнения операции, соответствующей данному типу. Теперь мы хотим создать несколько разнотипных объектов Shape и сохранить их в контейнере STL. Рассмотрим вспомогательное определение типа

```
typedef std::vector<Shape*> Container;
```

Это определение создает синоним для вектора с элементами Shape*, а следующее определение использует его и создает другой синоним для vector<Shape*>::iterator:

```
typedef Container::iterator Iter;
```

Обратите внимание: чтобы получить правильный тип итератора, определенный в виде вложенного класса, необходимо указать имя типа контейнера. Хотя в STL существуют разные виды итераторов (прямые, двусторонние, итераторы произ-

вольного доступа и т. д.), все они обладают одинаковым базовым интерфейсом. Итератор можно перевести к следующему элементу оператором ++, разыменовать его для получения объекта, на который итератор указывает в данный момент, и проверить, не вышел ли он за границу интервала. Именно эти операции выполняются с итераторами в 90 % случаев. После создания контейнер заполняется указателями на различные подклассы Shape. Помните, что при включении указателей на Circle, Square или Rectangle в контейнер указателей на Shape происходит повышающее преобразование, так как контейнер ничего не знает об этих специализированных типах, а работает лишь с Shape*. Как только указатель включается в контейнер, он теряет свою «индивидуальность» и становится анонимным указателем Shape*. Происходит именно то, что нужно: мы смешиваем разные типы указателей и предоставляем механизму полиморфного вызова разбираться, что к чему.

В первом цикле for мы создаем итератор и устанавливаем его в начало интервала, вызывая функцию begin() класса контейнера. Во всех контейнерных классах определены функции begin() и end(), которые возвращают соответственно начальный и конечный[1] итераторы. Чтобы узнать, не завершен ли перебор элементов контейнера, следует проверить, *не равен ли* итератор конечному итератору endl(). Для этого нельзя использовать операторы < и <=; работают только проверки != и ==, поэтому на практике часто встречаются циклы вида

```
for(Iter i = shapes.begin(); i != shapes.end(); i++)
```

Это означает «перебрать все элементы контейнера от начала до конца».

Как получить значение элемента, на который ссылается итератор? Конечно, разыменовать его перегруженным оператором *. Полученное значение соответствует типу элементов контейнера. В нашем примере контейнер содержит указатели Shape*, поэтому *i вернет указатель Shape*. Функция класса Shape по указателю вызывается оператором ->; значит, следующая строка вызывает функцию draw() для указателя Shape*, на который в данный момент ссылается итератор:

```
(*i)->draw();
```

Круглые скобки в левой части выглядят некрасиво, но они нужны для получения желаемого приоритета операторов.

При уничтожении и в других ситуациях, связанных с исключением указателей из контейнера, контейнеры STL *не вызывают* оператор delete для содержащихся в них указателей. Допустим, вы создали объект в куче оператором new и сохранили указатель на него в контейнере. Контейнер не знает, хранится этот указатель в другом контейнере или нет; он не знает даже того, относится или нет этот указатель к памяти в куче. Как всегда, программист несет всю ответственность за освобождение памяти, выделенной из кучи. В последних строках программы происходит зачистка: мы перебираем все указатели в контейнере и вызываем для каждого из них оператор delete. Учтите, что указатель auto_ptr не может использоваться для этой цели, поэтому подходящие умные указатели придется искать за пределами стандартной библиотеки C++[2].

---

[1] Стоит напомнить, что конечный итератор ссылается не на последний элемент, а на позицию *за* последним элементом. — *Примеч. перев.*

[2] Ситуация скоро изменится, так как следующая версия стандарта должна специфицировать дополнительные типы умных указателей. Если вы захотите познакомиться с ними заранее, найдите классы умных указателей на сайте www.boost.org.

Вы можете изменить тип контейнера, используемого программой. Например, чтобы перейти от вектора к списку, достаточно изменения двух строк. Вместо включения заголовка **<vector>** включается заголовок **<list>**, а первое определение типа приводится к виду

```
typedef std::list<Shape*> Container;
```

Весь остальной код остается без изменений. Такая гибкость стала возможной благодаря интерфейсу, установленному не механизмом наследования (в STL наследование почти не используется), а правилами, которыми руководствовались разработчики STL специально для того, чтобы сделать возможной подобную замену. Теперь с вектора можно легко переключиться на список или любой другой контейнер с идентичным интерфейсом (как синтаксически, так и семантически) и посмотреть, какой вариант лучше подходит для ваших целей.

Хранение строк в контейнере

В предыдущем примере функция main() должна была завершаться перебором всего списка и вызовом оператора delete для указателей на Shape:

```
for(Iter j = shapes.begin(); j != shapes.end(); j++)
    delete *j;
```

Контейнеры STL гарантируют, что для каждого содержащегося в них *объекта* будет вызван деструктор при уничтожении контейнера. Однако у указателей нет деструкторов, поэтому для них оператор delete приходится вызывать специально.

Так мы приходим к тому, что на первый взгляд кажется просчетом разработчиков STL: ни в одном контейнере STL не предусмотрена возможность автоматического вызова оператора delete для содержащихся в контейнере указателей, поэтому оператор delete приходится вызывать вручную. Неужели разработчики решили, что контейнеры указателей не представляют интереса для программиста? Конечно, нет.

Автоматический вызов оператора delete для указателя создает проблемы из-за возможности *множественной принадлежности* элементов. Если контейнер содержит указатель на объект, вполне возможно, что этот указатель также хранится в другом контейнере. Указатель на объект Aluminium, хранящийся в списке указателей на Trash, также может храниться в списке указателей на Aluminium. Какой из двух списков в этом случае должен отвечать за уничтожение объекта, или, другими словами, какому из списков «принадлежит» этот объект?

Проблема полностью исчезает, если вместо указателей в списке хранятся объекты. В этом случае вполне очевидно, что при уничтожении списка также должны уничтожаться все хранящиеся в нем объекты. Здесь STL проявляет себя с лучшей стороны; чтобы убедиться в этом, достаточно создать контейнер с объектами типа string. В следующем примере входные строки сохраняются в виде строковых объектов в векторе **vector<string>**:

```
//: C07:StringVector.cpp
// Вектор строк
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
```

```
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "StringVector.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  vector<string> strings;
  string line;
  while(getline(in, line))
    strings.push_back(line);
  // Операции со строками...
  int i = 1;
  vector<string>::iterator w;
  for(w = strings.begin();
      w != strings.end(); w++) {
    ostringstream ss;
    ss << i++;
    *w = ss.str() + ": " + *w;
  }
  // Вывод содержимого контейнера:
  copy(strings.begin(), strings.end(),
    ostream_iterator<string>(cout, "\n"));
  // Поскольку строки не являются указателями,
  // объекты string уничтожаются автоматически!
} ///:~
```

Программа сначала создает вектор vector<string> с именем strings, читает строки из файла в объекты string и сохраняет их в векторе:

```
while(getline(in, line))
  strings.push_back(line);
```

В данном примере строки, прочитанные из файла, нумеруются. Класс stringstream обеспечивает удобное преобразование int в строку, представляющую данное число.

Перегрузка оператора + упрощает сборку объектов string из отдельных компонентов. Существует и другая удобная возможность: разыменование итератора w дает строку, которая может использоваться *как в правой, так и в левой* части команды присваивания:

```
*w = ss.str() + ": " + *w;
```

Присваивание элементам контейнера через разыменованный итератор выглядит несколько неожиданно, но является следствием тщательно продуманной архитектуры STL.

Так как вектор vector<string> содержит объекты, а не указатели, стоит обратить внимание на два обстоятельства. Во-первых, как объяснялось ранее, вам не придется явно уничтожать объекты string. Даже если сохранить адреса объектов string в виде указателей в других контейнерах, понятно, что контейнер с объектами string является «главным», и право владения объектами принадлежит именно ему.

Во-вторых, в программе объекты создаются динамически, но при этом нигде не вызываются ни оператор new, ни оператор delete! Все эти операции выполняются контейнером vector, который сохраняет *копии* переданных ему объектов. В результате программа становится более компактной и понятной.

Наследование от контейнеров STL

Мгновенное создание последовательности элементов выглядит просто потрясающе. Сразу становится ясно, сколько времени вы раньше тратили попусту на решение этой задачи. Например, многие утилиты читают файл в память, изменяют его и записывают обратно на диск. Теперь с таким же успехом можно упаковать функциональность программы StringVector.cpp в класс для последующего использования.

Но при этом возникает вопрос: нужно ли создать вложенный объект типа vector или воспользоваться наследованием? Общие рекомендации объектно-ориентированного проектирования обычно отдают предпочтение композиции (то есть созданию вложенных объектов) перед наследованием, однако стандартные алгоритмы предполагают реализацию конкретного интерфейса, поэтому наследование часто бывает необходимым.

```
//: C07:FileEditor.h
// Редактирование файла в памяти
#ifndef FILEEDITOR_H
#define FILEEDITOR_H
#include <iostream>
#include <string>
#include <vector>

class FileEditor :
  public std::vector<std::string> {
public:
  void open(const char* filename);
  FileEditor(const char* filename) { open(filename); }
  FileEditor() {};
  void write(std::ostream& out = std::cout);
};
#endif // FILEEDITOR_H ///:~
```

Конструктор открывает файл и читает его в объект FileEditor, а функция write() выводит вектор строк в любой поток ostream. Обратите внимание на определение аргумента по умолчанию в функции write().

Реализация выглядит достаточно просто:

```
//: C07:FileEditor.cpp {O}
#include "FileEditor.h"
#include <fstream>
#include "../require.h"
using namespace std;

void FileEditor::open(const char* filename) {
  ifstream in(filename);
  assure(in, filename);
  string line;
  while(getline(in, line))
    push_back(line);
}

// Также можно воспользоваться алгоритмом copy():
void FileEditor::write(ostream& out) {
  for(iterator w = begin();  w != end(); w++)
    out << *w << endl;
} ///:~
```

В этой программе функции StringVector.cpp просто упакованы в новом классе. Довольно часто эволюция классов происходит именно так — сначала программист пишет программу для решения конкретной задачи, а затем обнаруживает в ней типовые функциональные возможности, которые можно преобразовать в класс.

Теперь мы можем переписать программу нумерации строк с использованием класса FileEditor:

```
//: C07:FEditTest.cpp
//{L} FileEditor
// Использование класса FileEditor
#include <sstream>
#include "FileEditor.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  FileEditor file;
  if(argc > 1) {
    file.open(argv[1]);
  } else {
    file.open("FEditTest.cpp");
  }
  // Операции со строками...
  int i = 1;
  FileEditor::iterator w = file.begin();
  while(w != file.end()) {
    ostringstream ss;
    ss << i++;
    *w = ss.str() + ": " + *w;
    ++w;
  }
  // Вывод в cout:
  file.write();
} ///:~
```

Теперь чтение файла выполняется в конструкторе (или в функции open()):

```
FileEditor file(argv[1]);
```

А вывод происходит в одной строке (при этом выходные данные по умолчанию направляются в cout):

```
file.write();
```

Основной код программы связан с главной задачей, то есть с модификацией файла в памяти.

Классификация итераторов

Итератор представляет собой абстрактную модель. Он работает с различными типами контейнеров, не зная базовой структуры данных этих контейнеров. Итераторы поддерживаются большинством контейнеров[1], поэтому типы итераторов для заданного контейнера определяются записью

[1] Контейнерные адаптеры (стек, очередь и приоритетная очередь) не поддерживают итераторы, поскольку с точки зрения пользователя они не ведут себя как последовательность элементов.

```
<тип_контейнера>::iterator
<тип_контейнера>::const_iterator
```

У каждого контейнера имеется функция begin(), которая возвращает итератор, обозначающий начало элементов контейнера, и функция end(), возвращающая конечный итератор, установленный в позицию *за последним элементом* контейнера. Если контейнер объявлен константным, то функции begin() и end() возвращают константные итераторы, запрещающие модификацию элементов, на которые они ссылаются (поскольку соответствующие операторы также объявлены константными).

Все итераторы поддерживают перемещение к следующему элементу (оператор ++) и сравнения == и !=. Следовательно, перемещение итератора it вперед до последнего элемента производится примерно так:

```
while(it != pastEnd) {
  // Какие-то операции
  ++it;
}
```

Здесь pastEnd — конечный итератор, возвращенный функцией end() класса контейнера.

Выборка элемента, на который в данный момент ссылается итератор, производится оператором разыменования *. Существуют две формы вызова функции объекта, хранящегося в контейнере:

```
(*it).f()
it->f();
```

Здесь it — итератор, используемый для перебора элементов, а f() — функция класса объекта, хранящегося в контейнере.

Руководствуясь этой информацией, можно создать шаблон, работающий с контейнером любого типа. В следующей программе шаблон функции apply() вызывает для каждого элемента в контейнере функцию класса, указатель на которую передается в аргументе:

```
//: C07:Apply.cpp
// Простой перебор элементов
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

template<class Cont, class PtrMemFun>
void apply(Cont& c, PtrMemFun f) {
  typename Cont::iterator it = c.begin();
  while(it != c.end()) {
    ((*it).*f)(); // Альтернативная форма
    it++;
  }
}

class Z {
  int i;
public:
  Z(int ii) : i(ii) {}
  void g() { i++; }
  friend ostream&
```

```
  operator<<(ostream& os, const Z& z) {
    return os << z.i;
  }
};

int main() {
  ostream_iterator<Z> out(cout, " ");
  vector<Z> vz;
  for(int i = 0; i < 10; i++)
    vz.push_back(Z(i));
  copy(vz.begin(), vz.end(), out);
  cout << endl;
  apply(vz, &Z::g);
  copy(vz.begin(), vz.end(), out);
} ///:~
```

В данном случае мы не можем использовать операторную функцию operator->, потому что команда будет иметь вид

```
(it->*f)();
```

Компилятор попытается использовать операторную функцию operator->*, не поддерживаемую классами итераторов[1].

Как было показано в предыдущей главе, для выполнения некоторой операции с каждым элементом контейнера гораздо удобнее воспользоваться алгоритмом for_each() или transform().

Итераторы в обратимых контейнерах

Контейнеры могут обладать свойством *обратимости*; это означает, что они позволяют создавать как итераторы, перемещающиеся от начала к концу, так и итераторы, перемещающиеся от конца к началу. Все стандартные контейнеры поддерживают двусторонний перебор.

У обратимых контейнеров определены функции rbegin() (для получения итератора reverse_iterator, установленного на последний элемент) и rend() (для получения итератора reverse_iterator, установленного в позицию «перед началом контейнера»). Если контейнер объявлен константным, функции rbegin() и rend() возвращают константные итераторы.

В следующем примере используется класс vector, но он также работает со всеми контейнерами, поддерживающими итераторы:

```
//: C07:Reversible.cpp
// Использование обратимых контейнеров
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Reversible.cpp");
  assure(in, "Reversible.cpp");
```

[1] Такое решение работает только в тех реализациях вектора, которые в качестве типа итератора используют *указатель* (T*), например, в STLPort.

```
  string line;
  vector<string> lines;
  while(getline(in, line))
    lines.push_back(line);
  for(vector<string>::reverse_iterator r = lines.rbegin();
      r != lines.rend(); r++)
    cout << *r << endl;
} ///:~
```

Синтаксис перебора в обратном направлении ничем не отличается от синтаксиса прямого перебора с применением обычных итераторов.

Категории итераторов

Итераторы стандартной библиотеки C++ делятся на категории, описывающие их возможности. Обычно перечисление этих категорий начинается с простейших.

Итераторы ввода — только чтение, один проход

Итераторы ввода существуют всего в двух стандартных реализациях: istream_iterator и istreambuf_iterator для чтения из потока istream. Как нетрудно догадаться, итератор ввода может быть разыменован только один раз для каждого элемента, подобно тому, как каждая часть входного потока может быть прочитана только один раз. Итераторы ввода перемещаются только в прямом направлении (от начала к концу); конечный итератор определяется специальным конструктором. Короче говоря, итератор ввода поддерживает разыменование (только один раз для каждого значения) и перемещение вперед.

Итераторы вывода — только запись, один проход

Итератор вывода похож на итератор ввода, но он предназначен для записи, а не для чтения. Итераторы вывода существуют в стандартных реализациях ostream_iterator и ostreambuf_iterator для записи в поток ostream, а также реже используемой реализации raw_storage_iterator. Они также могут разыменовываться только один раз для каждого записанного значения и поддерживают перемещение только от начала к концу контейнера. Для итераторов вывода не существует понятия «конечного значения», после которого дальнейшее перемещение невозможно. Итак, итераторы вывода поддерживают разыменование (только один раз для каждого значения) и перемещение вперед.

Прямые итераторы — множественные операции чтения-записи

Прямой итератор объединяет всю функциональность итераторов ввода и вывода, а также возможность многократного разыменования, что позволяет многократно читать и записывать элементы. Как подсказывает название, перемещение возможно только в прямом направлении. В стандартной библиотеке не существует стандартных итераторов, которые бы относились к этой категории.

Двусторонние итераторы — оператор --

Двусторонний итератор обладает всеми возможностями прямого итератора, а также может перемещаться назад на одну позицию оператором --. Итераторы, возвращаемые контейнером list, являются двусторонними.

Итераторы произвольного доступа — аналоги указателей

Наконец, итераторы произвольного доступа обладают всеми возможностями двусторонних итераторов, а также всеми основными возможностями указателей (указатель является частным случаем итератора произвольного доступа), кроме одной: у них отсутствует «нуль-итератор», аналог нуль-указателя. С итераторами произвольного доступа выполняются такие же операции, как с указателями: индексация оператором [ ], прибавление целочисленного смещения (положительного или отрицательного) для перехода сразу на несколько позиций, сравнение итераторов операторами отношения.

О важности классификации

Но стоит ли задумываться о том, к какой категории относится тот или иной итератор? При выполнении простейших операций с контейнерами (например, при ручном кодировании всех операций, выполняемых с объектами контейнеров) категория итератора обычно несущественна — итератор либо работает, либо не работает. Однако в некоторых ситуациях категории итераторов действительно важны.

- При использовании нетривиальных разновидностей итераторов (см. далее) и при создании собственных итераторов.
- При использовании алгоритмов STL (эта тема рассматривалась в предыдущей главе). Каждый алгоритм предъявляет определенные требования к своим итераторам. Информация о категории итератора играет еще более важную роль при создании пользовательских шаблонов алгоритмов, поскольку категория итератора, обязательная для вашего алгоритма, определяет его универсальность. Если алгоритм обходится самой примитивной категорией итераторов (ввода или вывода), он будет работать с *любыми* итераторами (примером может служить алгоритм copy()).

Категория итератора определяется иерархией специальных классов — итераторных тегов. Имена классов соответствуют категориям итераторов, а иерархия наследования отражает логические связи между ними:

```
// Итератор ввода
struct input_iterator_tag {};
// Итератор вывода
struct output_iterator_tag {};
// Прямой итератор
struct forward_iterator_tag: public input_iterator_tag {};
// Двусторонний итератор
struct bidirectional_iterator_tag : public
  forward_iterator_tag {};
// Итератор произвольного доступа
struct random_access_iterator_tag : public
  bidirectional_iterator_tag {};
```

Класс forward_iterator_tag наследует только от input_iterator_tag, но не от output_iterator_tag, потому что алгоритмы, использующие прямые итераторы, должны поддерживать концепцию конечных итераторов, а алгоритмы с итераторами вывода предполагают, что итератор всегда можно разыменовать оператором *. По этой причине очень важно, чтобы конечный итератор никогда не передавался алгоритму, получающему итератор вывода.

В целях эффективности некоторые алгоритмы предоставляют разные реализации для разных типов итераторов. Информацию о типе итератора они получают из итераторного тега, определяемого итератором. С некоторыми классами итераторных тегов мы познакомимся позже, когда мы займемся определением пользовательских итераторов.

Стандартные итераторы

В STL входит набор заранее определенных итераторов. В частности, мы уже встречались с объектами reverse_iterator, которые создаются функциями rbegin() и rend() всех основных контейнерных классов.

Итераторы вставки необходимы из-за того, что некоторые алгоритмы STL (например, copy()) заносят объекты в приемный контейнер оператором присваивания =. Проблемы возникают, если алгоритм используется для *заполнения* контейнера (вместо замены объектов, уже присутствующих в контейнере), то есть когда память для элементов еще не выделена. Итераторы вставки изменяют реализацию оператора = так, чтобы вместо присваивания вызывалась функция push или insert соответствующего контейнера. Тем самым обеспечивается выделение памяти для новых элементов. Конструкторы basic_insert_iterator и front_insert_iterator получают при вызове объект одного из базовых классов последовательных контейнеров (vector, deque или list) и создают итератор, который для присваивания вызывает соответственно функции push_back() или push_front(). Вспомогательные функции back_inserter() и front_inserter() создают те же объекты итераторов вставки с конца и с начала контейнера, но вводятся быстрее. Поскольку функция push_back() поддерживается всеми базовыми последовательными контейнерами, вероятно, вы будете довольно часто использовать в своей работе функцию back_inserter().

Итератор insert_iterator вставляет элементы в середину последовательности. Он тоже переопределяет смысл оператора =, но вместо функций категории push он автоматически вызывает функцию insert(). При вызове этой функции класса должен передаваться итератор, установленный в позицию *перед* позицией вставки, поэтому у объекта insert_iterator должны быть два аргумента: контейнер и итератор. Вспомогательная функция inserter() создает этот же объект.

Следующий пример демонстрирует применение различных категорий итераторов.

```
//: C07:Inserters.cpp
// Различные типы итераторов вставки
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <iterator>
using namespace std;

int a[] = { 1, 3, 5, 7, 11, 13, 17, 19, 23 };

template<class Cont>
void frontInsertion(Cont& ci) {
  copy(a, a + sizeof(a)/sizeof(Cont::value_type),
    front_inserter(ci));
  copy(ci.begin(), ci.end(),
```

```
    ostream_iterator<typename Cont::value_type>(
    cout, " "));
  cout << endl;
}

template<class Cont>
void backInsertion(Cont& ci) {
  copy(a, a + sizeof(a)/sizeof(Cont::value_type),
    back_inserter(ci));
  copy(ci.begin(), ci.end(),
    ostream_iterator<typename Cont::value_type>(
    cout, " "));
  cout << endl;
}

template<class Cont>
void midInsertion(Cont& ci) {
  typename Cont::iterator it = ci.begin();
  ++it; ++it; ++it;
  copy(a, a + sizeof(a)/(sizeof(Cont::value_type) * 2),
    inserter(ci, it));
  copy(ci.begin(), ci.end(),
    ostream_iterator<typename Cont::value_type>(
    cout, " "));
  cout << endl;
}

int main() {
  deque<int> di;
  list<int>  li;
  vector<int> vi;
  // front_inserter() не может использоваться с вектором
  frontInsertion(di);
  frontInsertion(li);
  di.clear();
  li.clear();
  backInsertion(vi);
  backInsertion(di);
  backInsertion(li);
  midInsertion(vi);
  midInsertion(di);
  midInsertion(li);
} ///:~
```

Так как шаблон vector не поддерживает функцию push_front(), он не может создать объект front_insert_iterator. С другой стороны, вектор поддерживает два других типа вставки (хотя как будет показано позднее, операция insert() для векторов работает неэффективно). Обратите внимание на использование вложенного типа Cont::value_type вместо жесткого кодирования типа int.

Снова о потоковых итераторах

Потоковые итераторы ostream_iterator (итератор вывода) и istream_iterator (итератор ввода) уже упоминались при рассмотрении алгоритма copy() в главе 6. Не забывайте, что для потоков вывода не определено понятие «конца потока», поскольку в них всегда можно вывести новые элементы. Однако поток ввода рано или поздно завершается (например, при достижении конца файла), и это состояние нужно как-то представить. У объектов istream_iterator имеются два конструктора:

первый получает istream и создает итератор, через который можно читать данные, а второй (конструктор по умолчанию) создает объект конечного итератора. В следующей программе этот объект называется end:

```
//: C07:StreamIt.cpp
// Итераторы потоков ввода и вывода
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("StreamIt.cpp");
  assure(in, "StreamIt.cpp");
  istream_iterator<string> begin(in), end;
  ostream_iterator<string> out(cout, "\n");
  vector<string> vs;
  copy(begin, end, back_inserter(vs));
  copy(vs.begin(), vs.end(), out);
  *out++ = vs[0];
  *out++ = "That's all, folks!";
} ///:~
```

Когда в потоке in кончаются входные данные (в данном случае — при достижении конца файла), копирование завершается.

Так как итератор out относится к классу output_iterator<string>, разыменованному итератору можно просто присвоить любой объект string оператором =. Присвоенная строка помещается в выходной поток, как в двух присваиваниях out в нашем примере. При определении out во втором аргументе передается символ перевода строки, поэтому команда присваивания также включает в поток перевод строки.

Хотя создать объекты istream_iterator<char> и ostream_iterator<char> возможно, фактически они будут выполнять *синтаксический разбор* входных данных. В частности, произойдет автоматическое исключение пропусков (пробелов, символов табуляции и перевода строки), а это нежелательно, если вы хотите работать с точным представлением потока istream. Вместо них следует использовать итераторы istreambuf_iterator и ostreambuf_iterator, спроектированные специально для работы с отдельными символами[1]. Хотя они реализованы в виде шаблонов, предполагается, что в аргументах передаются только типы char и wchar_t[2]. В следующем примере поведение потоковых итераторов сравнивается с итераторами streambuf:

```
//: C07:StreambufIterator.cpp
// istreambuf_iterator и ostreambuf_iterator
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
```

[1] Точнее говоря, эти итераторы создавались для абстрагирования потоков ввода-вывода от фацетов локальных контекстов, чтобы фацеты могли работать с любыми последовательностями символов, не только с потоками. Локальные контексты обеспечивают удобное форматирование потоков ввода-вывода по национальным стандартам (например, представление денежных величин).

[2] Для остальных типов аргументов необходимо предоставить специализацию шаблона char_traits.

```
using namespace std;

int main() {
  ifstream in("StreambufIterator.cpp");
  assure(in, "StreambufIterator.cpp");
  // Точное представление потока:
  istreambuf_iterator<char> isb(in), end;
  ostreambuf_iterator<char> osb(cout);
  while(isb != end)
    *osb++ = *isb++; // Копирование 'in' в cout
  cout << endl;
  ifstream in2("StreambufIterator.cpp");
  // Удаление пропусков:
  istream_iterator<char> is(in2), end2;
  ostream_iterator<char> os(cout);
  while(is != end2)
    *os++ = *is++;
  cout << endl;
} ///:~
```

Правила синтаксического разбора для потоковых итераторов определяет оператор istream::operator>>. Если вы самостоятельно обрабатываете символьные данные, вряд ли вас это устроит — не так уж часто требуется удалять из входного потока символов все пропуски. При работе с символами вместо обычных потоковых итераторов почти всегда используются буферные итераторы streambuf. Вдобавок оператор istream::operator>> существенно замедляет выполнение всех операций вывода, поэтому он применяется только при высокоуровневых операциях (например, выборке чисел из входного потока[1]).

Работа с блоками памяти

Итератор raw_storage_iterator определяется в заголовке <memory> и относится к категории итераторов вывода. Он позволяет алгоритмам сохранять результаты своей работы в неинициализированной памяти. Интерфейс этого итератора достаточно прост: конструктору передается итератор вывода, указывающий на физический блок памяти (как правило — указатель), а оператор = записывает объект в указанный блок. Параметры шаблона определяют тип итератора вывода, ссылающегося на блок памяти, и тип сохраняемого объекта. В следующем примере создаются объекты Noisy, которые выводят сообщения о своем конструировании, присваивании и уничтожении (определение класса Noisy будет приведено позднее):

```
//: C07:RawStorageIterator.cpp {-bor}
// Использование raw_storage_iterator
//{L} Noisy
#include <iostream>
#include <iterator>
#include <algorithm>
#include "Noisy.h"
using namespace std;

int main() {
  const int quantity = 10;
  // Выделение памяти и приведение к нужному типу:
```

[1] Спасибо Натану Майерсу (Nathan Myers) за это объяснение.

```
  Noisy* np =
    reinterpret_cast<Noisy*>(
      new char[quantity * sizeof(Noisy)]);
  raw_storage_iterator<Noisy*, Noisy> rsi(np);
  for(int i = 0; i < quantity; i++)
    *rsi++ = Noisy(); // Сохранение объектов в памяти
  cout << endl;
  copy(np, np + quantity,
    ostream_iterator<Noisy>(cout, " "));
  cout << endl;
  // Явный вызов деструктора:
  for(int j = 0; j < quantity; j++)
    (&np[j])->~Noisy();
  // Освобождение памяти:
  delete reinterpret_cast<char*>(np);
} ///:~
```

Шаблон raw_storage_iterator требует, чтобы выделенная память соответствовала типу создаваемых объектов, поэтому указатель на созданный массив char преобразуется в указатель Noisy*. Оператор присваивания сохраняет объекты в выделенной памяти с использованием копирующего конструктора. Обратите внимание на необходимость явного вызова деструктора для освобождения выделенной памяти; это также позволяет удалять объекты по одному в процессе операций с контейнером. Выражение delete np все равно было бы недопустимым, поскольку статический тип указателя в выражении с оператором delete должен совпадать с типом, которому присваивалось значение в выражении с оператором new.

Основные последовательные контейнеры

Последовательные контейнеры (вектор, список, дек) хранят объекты в порядке их занесения в контейнер. Однако они различаются по эффективности операций, и если вы собираетесь выполнять с контейнером множество однотипных операций, выберите ту разновидность контейнера, в которой эти операции выполняются эффективнее всего. До сих пор во всех приводимых примерах мы использовали контейнер vector; и в самом деле, на практике векторы применяются чаще других разновидностей контейнеров. Но постепенно вы начнете осваивать нетривиальные возможности контейнеров, и чтобы принять правильное решение, вам нужно будет больше знать об особенностях реализации и поведения основных разновидностей контейнеров.

Базовые операции в последовательных контейнерах

В следующем примере продемонстрированы операции, поддерживаемые всеми основными последовательными контейнерами: векторами, деками и списками.

```
//: C07:BasicSequenceOperations.cpp
// Операции, доступные для всех основных
// последовательных контейнеров.
#include <deque>
#include <iostream>
#include <list>
#include <vector>
using namespace std;
```

```
template<typename Container>
void print(Container& c, char* title = "") {
  cout << title << ':' << endl;
  if(c.empty()) {
    cout << "(empty)" << endl;
    return;
  }
  typename Container::iterator it;
  for(it = c.begin(); it != c.end(); it++)
    cout << *it << " ";
  cout << endl;
  cout << "size() " << c.size()
    << " max_size() "<< c.max_size()
    << " front() " << c.front()
    << " back() " << c.back() << endl;
}

template<typename ContainerOfInt>
void basicOps(char* s) {
  cout << "------- " << s << " -------" << endl;
  typedef ContainerOfInt Ci;
  Ci c;
  print(c, "c after default constructor");
  Ci c2(10, 1); // 10 элементов со значением 1
  print(c2, "c2 after constructor(10,1)");
  int ia[] = { 1, 3, 5, 7, 9 };
  const int iasz = sizeof(ia)/sizeof(*ia);
  // Инициализация с начальным и конечным итераторами:
  Ci c3(ia, ia + iasz);
  print(c3, "c3 after constructor(iter,iter)");
  Ci c4(c2); // Копирующий конструктор
  print(c4, "c4 after copy-constructor(c2)");
  c = c2; // Оператор присваивания
  print(c, "c after operator=c2");
  c.assign(10, 2); // 10 элементов со значением 2
  print(c, "c after assign(10, 2)");
  // Присваивание с начальным и конечным итераторами:
  c.assign(ia, ia + iasz);
  print(c, "c after assign(iter, iter)");
  cout << "c using reverse iterators:" << endl;
  typename Ci::reverse_iterator rit = c.rbegin();
  while(rit != c.rend())
    cout << *rit++ << " ";
  cout << endl;
  c.resize(4);
  print(c, "c after resize(4)");
  c.push_back(47);
  print(c, "c after push_back(47)");
  c.pop_back();
  print(c, "c after pop_back()");
  typename Ci::iterator it = c.begin();
  ++it; ++it;
  c.insert(it, 74);
  print(c, "c after insert(it, 74)");
  it = c.begin();
  ++it;
  c.insert(it, 3, 96);
  print(c, "c after insert(it, 3, 96)");
  it = c.begin();
```

```
  ++it;
  c.insert(it, c3.begin(), c3.end());
  print(c, "c after insert("
    "it, c3.begin(), c3.end())");
  it = c.begin();
  ++it;
  c.erase(it);
  print(c, "c after erase(it)");
  typename Ci::iterator it2 = it = c.begin();
  ++it;
  ++it2; ++it2; ++it2; ++it2; ++it2;
  c.erase(it, it2);
  print(c, "c after erase(it, it2)");
  c.swap(c2);
  print(c, "c after swap(c2)");
  c.clear();
  print(c, "c after clear()");
}

int main() {
  basicOps<vector<int> >("vector");
  basicOps<deque<int> >("deque");
  basicOps<list<int> >("list");
} ///:~
```

Первый шаблон функции print() выводит основную информацию, которую можно получить у любого последовательного контейнера: наличие элементов в контейнере, его текущий размер, максимально возможный размер контейнера, начальный и конечный элементы. Также из листинга видно, что каждый контейнер содержит функции begin() и end(), возвращающие итераторы.

Функция basicOps() тестирует все остальное (и в свою очередь, вызывает print()), включая разнообразные конструкторы (конструктор по умолчанию, копирующий конструктор), функции получения начального и конечного итераторов. Также тестируется оператор присваивания = и две разновидности функции assign класса контейнера. Одна функция получает количество элементов и исходное значение, а вторая — начальный и конечный итераторы.

Все основные последовательные контейнеры являются обратимыми, о чем свидетельствует использование функций rbegin() и rend(). Вы можете изменить размеры последовательного контейнера или удалить из него все элементы функцией clear(). При изменении размеров контейнера функцией resize() новые элементы инициализируются конструктором по умолчанию для типа элементов или нулями, если они относятся к встроенным типам.

Используя итератор для обозначения позиции вставки в любом последовательном контейнере, можно при помощи функции insert() вставить один элемент, несколько элементов с одинаковыми значениями или группу элементов из другого контейнера, заданную начальным и конечным итераторами.

Чтобы уничтожить один элемент из середины контейнера функцией erase(), воспользуйтесь итератором; для уничтожения группы элементов требуется пара итераторов. Поскольку список (list) поддерживает только двусторонние итераторы, все перемещения итераторов должны выполняться на уровне инкрементов и декрементов (для векторов и деков, поддерживающих итераторы произвольного доступа, итераторы можно перемещать сразу на несколько позиций операторами + и –).

Функции push_front() и pop_front() в отличие от списков и деков векторами не поддерживаются. С другой стороны, функции push_back() и pop_back() работают со всеми тремя типами контейнеров.

Функция swap() класса контейнера может вызвать недоразумения, поскольку в стандартной библиотеке присутствует алгоритм swap(), меняющий местами значения двух однотипных объектов. Функция swap() класса контейнера меняет местами содержимое двух *контейнеров* с однотипными объектами. Операция выполняется эффективно за счет пересылки внутреннего содержимого контейнеров, состоящего в основном из указателей. Алгоритм swap() обычно использует присваивание (дорогостоящая операция для целых контейнеров), но механизм специализации заставляет его вызывать функцию swap() для стандартных контейнеров. Также существует алгоритм iter_swap, который с помощью итераторов меняет местами два элемента одного контейнера.

В следующих разделах описаны отличительные особенности всех типов последовательных контейнеров.

Вектор

Шаблон класса vector напоминает усовершенствованный массив, и это сходство не случайное. Он также поддерживает индексацию в стиле массивов, но при этом расширяется динамически. Векторы настолько удобны, что в простейшем виде этот шаблон был представлен в начале первого тома и регулярно использовался в предыдущих примерах. В данном разделе приводится дополнительная информация о векторах.

Для достижения максимальной эффективности индексирования и перебора элементы вектора хранятся в одном непрерывном блоке. Данное обстоятельство очень важно для понимания базовых особенностей вектора. Из него следует, что индексирование и перебор выполняются с молниеносной быстротой — практически так же быстро, как в массиве объектов. Но с другой стороны, вставка новых объектов в любой позиции, кроме конечной, выполняется крайне медленно. Кроме того, когда в векторе кончается заранее выделенная свободная память, ему приходится выделять новый (увеличенный) блок памяти и копировать в него все элементы. Такой подход ведет к ряду неприятных побочных эффектов.

Издержки на выделение дополнительной памяти

При создании вектор выделяет блок памяти, как бы предполагая, сколько объектов вы собираетесь хранить в нем. Пока количество объектов не превышает размеров изначально выделенного блока, все операции выполняются быстро (если вы *знаете*, сколько объектов будет храниться в векторе, можно заранее зарезервировать необходимую память функцией reserve()). Но если произойдет переполнение, вектор выполняет следующие операции.

1. Выделение нового блока памяти большего размера.
2. Копирование всех объектов из прежнего блока памяти в новый (с использованием копирующего конструктора).
3. Уничтожение всех прежних объектов (с вызовом деструктора для каждого объекта).
4. Освобождение прежнего блока памяти.

Если вектор переполняется достаточно часто, а содержащиеся в нем объекты достаточно сложны, конструирование копий и уничтожение обходятся достаточно дорого. По этой причине векторы (и контейнеры STL вообще) ориентированы на хранение значений, копирование которых обходится малыми затратами. К этой категории относятся указатели.

Давайте посмотрим, что происходит при заполнении вектора. В этом нам поможет упоминавшийся ранее класс Noisy. Этот класс выводит информацию о создании и уничтожении объектов, о присваивании и конструировании копий:

```
//: C07:Noisy.h
// Класс для отслеживания основных операций с обьектами
#ifndef NOISY_H
#define NOISY_H
#include <iostream>
using endl;
using cout;
using ostream;

class Noisy {
  static long create, assign, copycons, destroy;
  long id;
public:
  Noisy() : id(create++) {
    cout << "d[" << id << "]" << endl;
  }
  Noisy(const Noisy& rv) : id(rv.id) {
    cout << "c[" << id << "]" << endl;
    ++copycons;
  }
  Noisy& operator=(const Noisy& rv) {
    cout << "(" << id << ")=[" << rv.id << "]" << endl;
    id = rv.id;
    ++assign;
    return *this;
  }
  friend bool operator<(const Noisy& lv, const Noisy& rv) {
    return lv.id < rv.id;
  }
  friend bool operator==(const Noisy& lv, const Noisy& rv) {
    return lv.id == rv.id;
  }
  ~Noisy() {
    cout << "~[" << id << "]" << endl;
    ++destroy;
  }
  friend ostream& operator<<(ostream& os, const Noisy& n) {
    return os << n.id;
  }
  friend class NoisyReport;
};

struct NoisyGen {
  Noisy operator()() { return Noisy(); }
};

// Синглетный класс. Статистика автоматически выводится
// при завершении программы:
class NoisyReport {
  static NoisyReport nr;
  NoisyReport() {} // Закрытый конструктор
```

```
  NoisyReport & operator=(NoisyReport &); // Запрещено
  NoisyReport (const NoisyReport&);       // Запрещено
public:
  ~NoisyReport() {
    cout << "\n-------------------\n"
      << "Noisy creations: " << Noisy::create
      << "\nCopy-Constructions: " << Noisy::copycons
      << "\nAssignments: " << Noisy::assign
      << "\nDestructions: " << Noisy::destroy << endl;
  }
};
#endif // NOISY_H ///:~

//: C07:Noisy.cpp{O}
#include "Noisy.h"
long Noisy::create = 0, Noisy::assign = 0,
  Noisy::copycons = 0, Noisy::destroy = 0;
NoisyReport NoisyReport::nr;
///:~
```

Каждый объект Noisy обладает собственным идентификатором, а все операции создания, присваивания, конструирования копий и уничтожения отслеживаются в статических переменных. Идентификатор id инициализируется на основании счетчика create в конструкторе по умолчанию; копирующий конструктор и оператор присваивания берут свои значения id из копируемого/присваиваемого объекта. В операторной функции operator= присваивание производится уже инициализированному объекту, поэтому прежнее значение id выводится перед его заменой на id присваиваемого объекта.

Для поддержки операций, автоматически выполняемых некоторыми контейнерами (таких, как сортировка и поиск), в Noisy необходимо определить операторы < и ==. Эти операторы просто сравнивают значения id. Оператор вывода в поток ostream написан по стандартному образцу и просто выводит значение id.

Тип NoisyGen представляет объекты функции (на что указывает оператор ()) для создания объектов Noisy.

Объект NoisyReport реализован как Синглет[1], потому что при завершении программы должен выводиться только один экземпляр отчета. Он имеет закрытый конструктор, предотвращающий создание дополнительных объектов NoisyReport, запрещает присваивание и конструирование копий, а также содержит один статический экземпляр NoisyReport с именем nr.

Все исполняемые команды находятся в деструкторе, который вызывается в фазе вызова статических деструкторов при завершении программы. Деструктор выводит статистику, накопленную в статических переменных Noisy.

Следующая программа использует файл Noisy.h для демонстрации переполнения вектора:

```
//: C07:VectorOverflow.cpp {-bor}
// Демонстрация конструирования копий и уничтожения объектов
// при перераспределении памяти вектора
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include "Noisy.h"
using namespace std;
```

[1] Синглет, один из хорошо известных эталонов проектирования, подробно рассматривается в главе 10.

```
int main(int argc, char* argv[]) {
  int size = 1000;
  if(argc >= 2) size = atoi(argv[1]);
  vector<Noisy> vn;
  Noisy n;
  for(int i = 0; i < size; i++)
    vn.push_back(n);
  cout << "\n cleaning up " << endl;
} ///:~
```

Вы можете использовать значение по умолчанию 1000 или передать собственный размер массива в командной строке.

При запуске программы выводится информация об одном вызове конструктора по умолчанию (для n), затем о множестве вызовов копирующего конструктора, далее о вызовах деструкторов, новых вызовах копирующих конструкторов и т. д. Когда в линейном массиве, выделенном вектором, кончается свободная память, вектор должен выделить новый блок памяти большего размера (чтобы сохранить непрерывное размещение объектов, являющееся важным свойством вектора) и перенести туда все текущее содержимое. Но для этого необходимо сначала скопировать, а затем уничтожить старые объекты. Как нетрудно представить, при большом количестве сложных объектов эти затраты быстро становятся неприемлемыми.

У проблемы есть два решения. В первом (оптимальном) случае необходимо заранее знать, сколько объектов будет создано. Тогда вы заранее резервируете необходимую память функцией reserve(); все лишнее копирование и уничтожение ликвидируется, и все операции работают очень быстро (особенно произвольный доступ к объектам, реализуемый оператором [ ]). Обратите внимание на отличие функции reserve() от конструктора vector с целочисленным первым аргументом; последний инициализирует заданное количество элементов, используя конструктор по умолчанию для типа элементов.

Однако в общем случае количество объектов заранее неизвестно. Если перераспределения памяти существенно замедляют работу с вектором, попробуйте перейти на другой последовательный контейнер. Можно воспользоваться списком, но как вскоре будет показано, дек обеспечивает быструю вставку с любого конца последовательности и не требует копирования/уничтожения объектов при перераспределении памяти. Дек также поддерживает индексацию элементов оператором [ ], но по скорости она уступает оператору [ ] вектора. Следовательно, если вы создаете все объекты в одной части программы, а затем организуете произвольный доступ к ним из другой части, в принципе можно попробовать заполнить дек, а затем создать вектор на базе дека и использовать его для быстрой индексации. Впрочем, не стоит прибегать к этому приему в повседневной работе — просто помните о такой возможности (преждевременные оптимизации вообще нежелательны).

Однако у перераспределения памяти есть и другие более серьезные последствия. Поскольку вектор хранит свои объекты в удобном, компактном массиве, используемые им итераторы могут представлять собой простые указатели. И это хорошо — именно эта особенность обеспечивает векторам самую быструю выборку и обработку данных среди последовательных контейнеров. Но давайте посмотрим, что произойдет при добавлении одного дополнительного объекта, после которого вектор выделяет новую память и перемещает данные в другое место. Указатель итератора указывает «в никуда»!

```
//: C07:VectorCoreDump.cpp
// Появление недействительных итераторов
```

```
#include <iterator>
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> vi(10, 0);
  ostream_iterator<int> out(cout, " ");
  vector<int>::iterator i = vi.begin();
  *i = 47;
  copy(vi.begin(), vi.end(), out);
  cout << endl;
  // Выполняем принудительное перераспределение памяти
  // (также можно было бы добавить нужное количество объектов):
  vi.resize(vi.capacity() + 1);
  // Теперь i ссылается на недействительную память:
  *i = 48;  // Нарушение доступа
  copy(vi.begin(), vi.end(), out); // vi[0] не изменяется
} ///:~
```

Данный пример иллюстрирует концепцию *недействительности итераторов.* Некоторые операции приводят к изменениям в базовой структуре данных контейнера, поэтому итераторы, действительные до выполнения операции, могут оказаться недействительными после нее. Если ваша программа «зависает» по каким-то загадочным причинам, посмотрите, не используются ли старые итераторы после включения новых объектов в вектор. После добавления элементов следует получить новый итератор или производить выборку элементов оператором [ ]. А если связать этот факт с потенциальными затратами на добавление новых объектов в вектор, становится ясно, что самый безопасный вариант использования вектора — его однократное заполнение заранее известным количеством объектов и последующая работа с ним без добавления новых объектов. В частности, именно так применялись векторы в приводившихся ранее примерах. В стандартной библиотеке C++ документированы операции с контейнерами, в результате которых итераторы становятся недействительными.

Стоит заметить, что выбор вектора в качестве «основного» контейнера для приводившихся примеров не всегда оптимален. В этом проявляется основополагающий принцип выбора контейнеров и структур данных вообще — оптимальный выбор зависит от того, как будет использоваться контейнер. До сих пор мы постоянно применяли вектор лишь из-за его сходства с массивом, благодаря чему он выглядит знакомо и легко осваивается. Но в дальнейшем при выборе контейнера также необходимо будет учитывать ряд других факторов.

Вставка и удаление элементов

Вектор работает наиболее эффективно, если выполняются два условия.

1. Программист заранее выделяет нужный объем памяти функцией reserve(), чтобы предотвратить дальнейшие перераспределения памяти.
2. Все операции добавления и удаления элементов выполняются только в конце вектора.

Вообще говоря, вы можете вставлять и уничтожать элементы в середине вектора. Но следующая программа показывает, почему этого делать не стоит:

```
//: C07:VectorInsertAndErase.cpp {-bor}
// Уничтожение элемента вектора
```

```
//{L} Noisy
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include "Noisy.h"
using namespace std;

int main() {
  vector<Noisy> v;
  v.reserve(11);
  cout << "11 spaces have been reserved" << endl;
  generate_n(back_inserter(v), 10, NoisyGen());
  ostream_iterator<Noisy> out(cout, " ");
  cout << endl;
  copy(v.begin(), v.end(), out);
  cout << "Inserting an element:" << endl;
  vector<Noisy>::iterator it =
    v.begin() + v.size() / 2; // Middle
  v.insert(it, Noisy());
  cout << endl;
  copy(v.begin(), v.end(), out);
  cout << "\nErasing an element:" << endl;
  // Старое значение it использовать нельзя:
  it = v.begin() + v.size() / 2;
  v.erase(it);
  cout << endl;
  copy(v.begin(), v.end(), out);
  cout << endl;
} ///:~
```

При запуске программы становится видно, что вызов reserve() всего лишь резервирует память — конструкторы элементов при этом не вызываются. Вызов generate_n() требует основательной работы: при каждом вызове NoisyGen::operator() происходит конструирование объекта, конструирование копии (в векторе) и уничтожение временного объекта. Но если объект вставляется в середину вектора, для сохранения линейности массива вектор должен сдвинуть все следующие элементы на одну позицию. Свободного места в векторе достаточно, поэтому смещение выполняется оператором присваивания (хотя если бы аргумент reserve() был равен 10 вместо 11, пришлось бы выделять дополнительную память). При уничтожении элемента функцией erase() оператор присваивания снова используется для смещения всех последующих элементов на одну позицию назад, чтобы занять освободившееся место (обратите внимание: для этого необходимо, чтобы оператор присваивания правильно уничтожал левосторонний объект). После сдвига элементов остается удалить последний элемент массива.

Дек

Дек (контейнер deque) представляет последовательность элементов, оптимизированную для добавления и уничтожения элементов с обоих концов контейнера. Кроме того, он обеспечивает достаточно быстрый произвольный доступ — в деках, как и в векторах, определен оператор индексирования [ ]. Однако на деки не распространяется основное ограничение векторов — необходимость хранения всех элементов в непрерывном блоке памяти. Вместо этого в типичной реализации дека используется цепочка блоков, каждый из которых содержит непрерывную после-

довательность элементов (все блоки и порядок их следования отслеживаются контейнером в непрерывной структуре данных). При такой базовой структуре затраты на добавление или удаление элементов с любого конца дека относительно невелики. Вдобавок деку, в отличие от вектора, не приходится копировать и уничтожать объекты при выделении нового блока, поэтому при добавлении заранее неизвестного количества элементов с любого конца контейнера дек работает гораздо эффективнее вектора. Таким образом, вектор оптимален лишь в том случае, если вы хорошо знаете, сколько объектов будет храниться в контейнере. Многие из приведенных ранее программ, использовавших вектор и функцию push_back(), более эффективно работали бы с деком. Интерфейс дека незначительно отличается от интерфейса вектора (например, у дека есть функции push_front() и pop_front(), которых нет у вектора), поэтому переход с вектора на дек в программе осуществляется тривиально. Рассмотрим пример StringVector.cpp; чтобы перевести его на использование дека, достаточно повсюду заменить слово «vector» словом «deque». Следующая программа является модификацией программы StringVector.cpp. Добавлены операции с деком, параллельные операциям с вектором, и сравнительный хронометраж:

```
//: C07:StringDeque.cpp
// Модификация StringVector.cpp
#include <cstddef>
#include <ctime>
#include <deque>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "StringDeque.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  vector<string> vstrings;
  deque<string> dstrings;
  string line;
  // Загрузка данных в вектор:
  clock_t ticks = clock();
  while(getline(in, line))
    vstrings.push_back(line);
  ticks = clock() - ticks;
  cout << "Read into vector: " << ticks << endl;
  // То же для дека:
  ifstream in2(fname);
  assure(in2, fname);
  ticks = clock();
  while(getline(in2, line))
    dstrings.push_back(line);
  ticks = clock() - ticks;
  cout << "Read into deque: " << ticks << endl;
  // Теперь сравниваем индексирование:
```

```
  ticks = clock();
  for(size_t i = 0; i < vstrings.size(); i++) {
    ostringstream ss;
    ss << i;
    vstrings[i] = ss.str() + ": " + vstrings[i];
  }
  ticks = clock() - ticks;
  cout << "Indexing vector: " << ticks << endl;
  ticks = clock();
  for(size_t j = 0; j < dstrings.size(); j++) {
    ostringstream ss;
    ss << j;
    dstrings[j] = ss.str() + ": " + dstrings[j];
  }
  ticks = clock() - ticks;
  cout << "Indexing deque: " << ticks << endl;
  // Сравнение перебора
  ofstream tmp1("tmp1.tmp"), tmp2("tmp2.tmp");
  ticks = clock();
  copy(vstrings.begin(), vstrings.end(),
    ostream_iterator<string>(tmp1, "\n"));
  ticks = clock() - ticks;
  cout << "Iterating vector: " << ticks << endl;
  ticks = clock();
  copy(dstrings.begin(), dstrings.end(),
    ostream_iterator<string>(tmp2, "\n"));
  ticks = clock() - ticks;
  cout << "Iterating deque: " << ticks << endl;
} ///:~
```

После всего, что говорилось о неэффективности включения элементов в вектор из-за перераспределения памяти, можно было бы ожидать радикальных различий между результатами двух тестов. Тем не менее, для текстового файла размером 1,7 Мбайт программа, созданная одним из компиляторов, выдала следующие результаты (не в секундах, а в тактах, зависящих от платформы и компилятора):

```
Read into vector: 8350
Read into deque: 7690
Indexing vector: 2360
Indexing deque: 2480
Iterating vector: 2470
Iterating deque: 2410
```

Результаты для другого компилятора и платформы выглядели примерно также. Выходит, все не так уж плохо? Отсюда можно сделать несколько важных выводов.

- Мы (программисты и авторы) не всегда точно представляем, какие именно компоненты наших программ работают недостаточно эффективно.
- Эффективность определяется несколькими факторами. В нашем примере затраты на чтение строк и их преобразование к типу string могут существенно превосходить дополнительные затраты, связанные с неоптимальным выбором контейнера.
- Вероятно, класс string достаточно хорошо спроектирован в плане эффективности.

Впрочем, это не означает, что вам следует использовать вектор вместо дека, если в конец контейнера добавляется неизвестное количество объектов. Напротив, лучше задействовать дек — если вы добиваетесь от своей программы максимальной эффективности. Но вы также должны знать, что проблемы с эффективностью не всегда лежат на поверхности, а «узкие места» программы можно выявить только тщательным тестированием. Позднее в этой главе будет представлено более «чистое» сравнение вектора, дека и списка по быстродействию.

Преобразования контейнеров

Иногда в разных точках программы нужно обеспечить поведение или показатели эффективности, соответствующие разным типам контейнеров. Например, при включении объектов в контейнер требуется эффективность дека, а при их индексировании — эффективность вектора. У всех основных последовательных контейнеров (вектора, дека и списка) имеется конструктор, которому при вызове передаются два итератора (указывающие на начало и конец интервала, элементы которого образуют новый объект), и функция assign() для загрузки данных в существующий контейнер. Это позволяет легко перемещать объекты из одного последовательного контейнера в другой.

В следующем примере объекты загружаются в дек, который затем преобразуется в вектор:

```
//: C07:DequeConversion.cpp {-bor}
// Загрузка данных в дек с последующим преобразованием в вектор
//{L} Noisy
#include <algorithm>
#include <cstdlib>
#include <deque>
#include <iostream>
#include <iterator>
#include <vector>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
  int size = 25;
  if(argc >= 2) size = atoi(argv[1]);
  deque<Noisy> d;
  generate_n(back_inserter(d), size, NoisyGen());
  cout << "\n Converting to a vector(1)" << endl;
  vector<Noisy> v1(d.begin(), d.end());
  cout << "\n Converting to a vector(2)" << endl;
  vector<Noisy> v2;
  v2.reserve(d.size());
  v2.assign(d.begin(), d.end());
  cout << "\n Cleanup" << endl;
} ///:~
```

Размеры контейнера могут быть разными, это ничего не изменит — объекты просто создаются в новых векторах копирующим конструктором. Интересно другое: независимо от количества элементов память для v1 выделяется только один раз. Может показаться, что для v1, как и для v2, следовало бы заранее выделить память, чтобы предотвратить нежелательные многократные ее перераспределения. Но в действительности это не нужно, поскольку использованный при создании v1 конструктор заранее определяет необходимый объем памяти.

Издержки на перераспределение памяти

Интересно посмотреть, что происходит с деком при переполнении выделенной памяти, и сравнить с программой VectorOverflow.cpp:

```
//: C07:DequeOverflow.cpp {-bor}
// При включении большого количества элементов
// с конца контейнера дек работает гораздо
// эффективнее вектора, поскольку он не требует
// копирования и уничтожения элементов.
#include <cstdlib>
#include <deque>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
  int size = 1000;
  if(argc >= 2) size = atoi(argv[1]);
  deque<Noisy> dn;
  Noisy n;
  for(int i = 0; i < size; i++)
    dn.push_back(n);
  cout << "\n cleaning up " << endl;
} ///:~
```

На этот раз перед появлением завершающей надписи «cleaning up» сообщения о вызове деструкторов практически не выводятся. Поскольку память для дека выделяется несколькими блоками (вместо одного непрерывного блока, как для вектора), деку не приходится перемещать данные между блоками, а следовательно, отпадает необходимость в лишнем конструировании копий и уничтожении элементов. По той же причине дек обеспечивает эффективную вставку элементов *в начало* контейнера, поскольку при нехватке памяти он просто выделяет новый блок для начальных элементов (разве что может потребоваться заново выделить память для индексного блока, связывающего блоки данных). С другой стороны, вставка в середину дека выполняется не так тривиально, как для векторов (хотя и с меньшими затратами).

Благодаря разумной схеме управления памятью в деках включение новых элементов в начало или конец контейнера не аннулирует существующие итераторы, как в случае векторов (см. пример VectorCoreDump.cpp). Если вы будете ограничиваться теми операциями, которые выполняются деком с наибольшей эффективностью (вставка и удаление элементов с обоих концов, достаточно быстрый перебор и индексирование), дек вас не подведет.

Проверка границ при произвольном доступе

В векторах и деках определены две функции произвольного доступа: оператор индексирования [ ] и функция at(), которая проверяет границы индексируемого контейнера и запускает исключение при их нарушении. Вызов at() обходится дороже простого индексирования:

```
//: C07:IndexingVsAt.cpp
// Сравнение функции at() с оператором []
#include <ctime>
#include <deque>
#include <iostream>
```

```
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  long count = 1000;
  int sz = 1000;
  if(argc >= 2) count = atoi(argv[1]);
  if(argc >= 3) sz = atoi(argv[2]);
  vector<int> vi(sz);
  clock_t ticks = clock();
  for(int i1 = 0; i1 < count; i1++)
    for(int j = 0; j < sz; j++)
      vi[j];
  cout << "vector[] " << clock() - ticks << endl;
  ticks = clock();
  for(int i2 = 0; i2 < count; i2++)
    for(int j = 0; j < sz; j++)
      vi.at(j);
  cout << "vector::at() " << clock()-ticks <<endl;
  deque<int> di(sz);
  ticks = clock();
  for(int i3 = 0; i3 < count; i3++)
    for(int j = 0; j < sz; j++)
      di[j];
  cout << "deque[] " << clock() - ticks << endl;
  ticks = clock();
  for(int i4 = 0; i4 < count; i4++)
    for(int j = 0; j < sz; j++)
      di.at(j);
  cout << "deque::at() " << clock()-ticks <<endl;
  // Поведение at() при нарушении границ контейнера:
  try {
    di.at(vi.size() + 1);
  } catch(...) {
    cerr << "Exception thrown" << endl;
  }
} ///:~
```

Как было показано в главе 1, в разных системах неперехваченные исключения обрабатываются по-разному. И все же при использовании функции at() вы хотя бы узнаете о возникших проблемах, тогда при использовании оператора [ ] можно остаться в неведении.

Список

Контейнер list реализуется в виде двусвязного списка. Такая реализация обеспечивает быструю вставку и удаление элементов *в любой позиции*, тогда как у векторов и деков эта операция обходится гораздо дороже. С другой стороны, произвольный доступ к элементам в списке выполняется настолько медленно, что у него даже нет оператора [ ]. Списки лучше всего использовать при последовательном переборе элементов от начала к концу контейнера (или наоборот) вместо случайного выбора элементов из середины. Впрочем, даже последовательный перебор в списках может работать медленнее, чем в векторах, но при относительно небольшом объеме перебора это не станет «узким местом» программы.

В каждом элементе списка, помимо данных самого объекта, должны храниться указатели на следующий и предыдущий элементы. Таким образом, список оптимально подходит для хранения больших объектов, часто вставляемых в середину списка и удаляемых из середины.

Списки лучше не использовать при большом объеме операций перебора или поиска объектов, потому что затраты времени на переход от начала списка[1] к нужному объекту пропорциональны расстоянию этого объекта от начала списка.

Объекты в списке никогда не перемещаются в памяти после создания. «Перемещение» элемента сводится к простому изменению ссылок и не требует копирования/присваивания самих объектов. Следовательно, добавление новых элементов в список не приводит к аннулированию итераторов, в отличие от контейнера vector. В следующем примере используется список объектов Noisy:

```
//: C07:ListStability.cpp {-bor}
// Элементы списков не перемещаются в памяти
//{L} Noisy
#include "Noisy.h"
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

int main() {
  list<Noisy> l;
  ostream_iterator<Noisy> out(cout, " ");
  generate_n(back_inserter(l), 25, NoisyGen());
  cout << "\n Printing the list:" << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Reversing the list:" << endl;
  l.reverse();
  copy(l.begin(), l.end(), out);
  cout << "\n Sorting the list:" << endl;
  l.sort();
  copy(l.begin(), l.end(), out);
  cout << "\n Swapping two elements:" << endl;
  list<Noisy>::iterator it1, it2;
  it1 = it2 = l.begin();
  ++it2;
  swap(*it1, *it2);
  cout << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Using generic reverse(): " << endl;
  reverse(l.begin(), l.end());
  cout << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Cleanup" << endl;
} ///:~
```

Даже такие внешне трудоемкие операции, как сортировка и обращение[2] списка, не требуют копирования объектов — вместо этого достаточно изменить соответствующие ссылки. Однако следует учесть, что функции sort() и reverse() являются функциями класса list, поэтому они знают о внутреннем устройстве контейнера

[1] А это единственная отправная точка, если ранее не был получен итератор, находящийся ближе к нужной позиции.

[2] То есть перестановка элементов в обратном порядке. — *Примеч. перев.*

и умеют переставлять элементы без копирования. С другой стороны, обобщенный алгоритм swap() ничего не знает об особенностях list, поэтому он меняет два элемента местами через присваивание. В общем случае вместо обобщенных алгоритмов рекомендуется задействовать эквивалентную функцию класса (если она существует). В частности, обобщенные алгоритмы sort() и reverse() следует использовать только с массивами, векторами и деками.

Если вы работаете с большими сложными объектами, попробуйте начать со списка, особенно если операции конструирования, уничтожения, конструирования копий и присваивания обходятся достаточно дорого, а вы собираетесь часто сортировать объекты или иным образом изменять порядок их следования.

Специальные операции со списками

Контейнер list содержит специальные функции, которые с максимальной эффективностью используют особенности его структуры. Функции reverse() и sort() уже упоминались выше. Следующий пример демонстрирует еще несколько таких функций:

```
//: C07:ListSpecialFunctions.cpp
//{L} Noisy
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
#include "Noisy.h"
#include "PrintContainer.h"
using namespace std;

int main() {
  typedef list<Noisy> LN;
  LN l1, l2, l3, l4;
  generate_n(back_inserter(l1), 6, NoisyGen());
  generate_n(back_inserter(l2), 6, NoisyGen());
  generate_n(back_inserter(l3), 6, NoisyGen());
  generate_n(back_inserter(l4), 6, NoisyGen());
  print(l1, "l1"); print(l2, "l2");
  print(l3, "l3"); print(l4, "l4");
  LN::iterator it1 = l1.begin();
  ++it1; ++it1; ++it1;
  l1.splice(it1, l2);
  print(l1, "l1 after splice(it1, l2)");
  print(l2, "l2 after splice(it1, l2)");
  LN::iterator it2 = l3.begin();
  ++it2; ++it2; ++it2;
  l1.splice(it1, l3, it2);
  print(l1, "l1 after splice(it1, l3, it2)");
  LN::iterator it3 = l4.begin(), it4 = l4.end();
  ++it3; --it4;
  l1.splice(it1, l4, it3, it4);
  print(l1, "l1 after splice(it1,l4,it3,it4)");
  Noisy n;
  LN l5(3, n);
  generate_n(back_inserter(l5), 4, NoisyGen());
  l5.push_back(n);
  print(l5, "l5 before remove()");
  l5.remove(l5.front());
  print(l5, "l5 after remove()");
  l1.sort(); l5.sort();
```

```
  l5.merge(l1);
  print(l5, "l5 after l5.merge(l1)");
  cout << "\n Cleanup" << endl;
} ///:~
```

После заполнения четырех списков объектами Noisy производятся три операции врезки. Сначала весь список l2 вставляется в список l1 в позиции итератора it1, причем список l2 остается пустым — врезка приводит к удалению элементов из исходного списка. Вторая операция врезки вставляет элементы списка l3, начиная с it2, в список l1, начиная с позиции it1. Третья врезка начинается с позиции it1 и использует элементы списка l4 в интервале от it3 до it4. Излишнее на первый взгляд упоминание списка-источника объясняется тем, что стирание элементов из источника является частью операции их пересылки в приемник.

Результаты тестирования функции remove() показывают, что удаление всех элементов с заданным значением не требует предварительной сортировки списка.

Наконец, слияние списков функцией merge() работает разумно только в том случае, если списки были отсортированы. Тогда вы получаете отсортированный список, содержащий все элементы обоих списков (при этом источник остается пустым, то есть его элементы *перемещаются* в приемник).

Функция unique() удаляет все дубликаты из предварительно отсортированного списка:

```
//: C07:UniqueList.cpp
// Тестирование функции unique() контейнера list
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

int a[] = { 1, 3, 1, 4, 1, 5, 1, 6, 1 };
const int asz = sizeof a / sizeof *a;

int main() {
  // Для вывода:
  ostream_iterator<int> out(cout, " ");
  list<int> li(a, a + asz);
  li.unique();
  // Ошибка! Дубликаты остаются в контейнере:
  copy(li.begin(), li.end(), out);
  cout << endl;
  // Список необходимо предварительно отсортировать:
  li.sort();
  copy(li.begin(), li.end(), out);
  cout << endl;
  // Теперь функция unique() работает правильно:
  li.unique();
  copy(li.begin(), li.end(), out);
  cout << endl;
} ///:~
```

Использованный в этом примере конструктор list получает начальный и конечный итераторы из другого контейнера и копирует все элементы из заданного интервала. В данном случае «контейнером» является простой массив, а «итераторами» — указатели на элементы этого массива, но благодаря архитектуре STL конструктор list работает с массивами так же легко, как с любыми другими контейнерами.

Функция unique() удаляет только *смежные* дубликаты, поэтому перед ее вызовом обычно необходимо отсортировать элементы. Исключение составляют ситуации, когда требуется удалить смежные дубликаты в соответствии с текущей сортировкой.

У шаблона list имеются еще четыре функции, которые здесь не показаны:

- функция remove_if() получает предикат для отбора удаляемых объектов;
- функция unique() получает бинарный предикат для проверки уникальности;
- функция merge() получает дополнительный аргумент, выполняющий сравнение;
- функция sort() получает дополнительный аргумент, выполняющий сравнение.

Сравнение списка с множеством

Как видно из предыдущего примера, для получения отсортированной последовательности элементов без дубликатов можно воспользоваться множеством (контейнер set). Интересно сравнить быстродействие этих двух контейнеров:

```
//: C07:ListVsSet.cpp
// Сравнение списка и множества по быстродействию
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iteraror>
#include <list>
#include <set>
#include "PrintContainer.h"
using namespace std;

class Obj {
  int a[20]; // Чтобы занять больше памяти
  int val;
public:
  Obj() : val(rand() % 500) {}
  friend bool
  operator<(const Obj& a, const Obj& b) {
    return a.val < b.val;
  }
  friend bool
  operator==(const Obj& a, const Obj& b) {
    return a.val == b.val;
  }
  friend ostream&
  operator<<(ostream& os, const Obj& a) {
    return os << a.val;
  }
};

struct ObjGen {
  Obj operator()() { return Obj(); }
};
```

```
int main() {
  const int SZ = 5000;
  srand(time(0));
  list<Obj> lo;
  clock_t ticks = clock();
  generate_n(back_inserter(lo), SZ, ObjGen());
  lo.sort();
  lo.unique();
  cout << "list:" << clock() - ticks << endl;
  set<Obj> so;
  ticks = clock();
  generate_n(inserter(so, so.begin()),
    SZ, ObjGen());
  cout << "set:" << clock() - ticks << endl;
  print(lo);
  print(so);
} ///:~
```

При запуске программы выясняется, что множество работает гораздо быстрее списка. Впрочем, в этом нет ничего удивительного, ведь множества предназначены для хранения отсортированного набора уникальных элементов!

В этом примере использован заголовок PrintContainer.h с шаблоном функции, выводящим содержимое любого последовательного контейнера в выходной поток. Определение PrintContainer.h выглядит так:

```
//: C07:PrintContainer.h
// Вывод последовательного контейнера
#ifndef PRINT_CONTAINER_H
#define PRINT_CONTAINER_H
#include "../C06/PrintSequence.h"

template<class Cont>
void print(Cont& c, const char* nm = "",
           const char* sep = "\n",
           std::ostream& os = std::cout) {
  print(c.begin(), c.end(), nm, sep, os);
}
#endif ///:~
```

Определяемый здесь шаблон print() просто вызывает шаблон функции print(), которая определялась в заголовке PrintSequence.h (см. главу 6).

Перестановка интервалов

Мы уже упоминали функцию swap() контейнерных классов, которая меняет местами содержимое двух контейнеров (но только однотипных). Функция swap() знает внутреннее строение конкретного контейнера, что и позволяет ей работать с максимальной эффективностью:

```
//: C07:Swapping.cpp {-bor}
// Функция swap() поддерживается всеми основными
// последовательными контейнерами
//{L} Noisy.h
#include <algorithm>
#include <deque>
#include <iostream>
```

```
#include <iterator>
#include <list>
#include <vector>
#include "Noisy.h"
#include "PrintConteiner.h"
using namespace std;
ostream_iterator<Noisy> out(cout, " ");

template<class Cont> void testSwap(char* cname) {
  Cont c1, c2;
  generate_n(back_inserter(c1), 10, NoisyGen());
  generate_n(back_inserter(c2), 5, NoisyGen());
  cout << "\n" << cname << ":" << endl;
  print(c1, "c1"); print(c2, "c2");
  cout << "\n Swapping the " << cname << ":" << endl;
  c1.swap(c2);
  print(c1, "c1"); print(c2, "c2");
}
int main() {
  testSwap<vector<Noisy> >("vector");
  testSwap<deque<Noisy> >("deque");
  testSwap<list<Noisy> >("list");
} ///:~
```

Запустив эту программу, вы убедитесь, что любая разновидность последовательных контейнеров поддерживает перестановку без операций копирования и присваивания, даже если контейнеры содержат разное количество элементов. Фактически меняются местами не элементы, а внутренние представления двух объектов.

В STL существует также алгоритм с именем swap(). Когда этот алгоритм применяется к двум контейнерам одного типа, для повышения быстродействия он задействует функцию swap() соответствующего контейнера. По этой причине алгоритм sort() очень быстро работает с контейнерами, элементами которых являются другие контейнеры, — оказывается, эта задача решалась еще на стадии проектирования STL.

Множество

В множествах (контейнер set) каждый элемент хранится только в одном экземпляре. Множество автоматически сортирует свои элементы (точнее, сортировка не является частью концептуального определения множества, но для ускорения поиска элементы множества STL хранятся в виде сбалансированного дерева, что обеспечивает их сортировку при переборе). Примеры использования множеств уже встречались в двух первых программах этой главы.

Допустим, вы хотите построить алфавитный указатель для книги. Для начала нужно создать список всех используемых слов, но каждое слово должно входить в перечень только один раз, при этом слова должны быть отсортированы. Множество идеально подходит для подобных целей. Применив этот контейнер, вы решите задачу с минимальными усилиями. Впрочем, заодно необходимо решить проблему со знаками препинания и другими неалфавитными символами — удалив их из текста, мы получим нормальные слова. Для этого можно воспользоваться функ-

циями isalpha() и isspace() стандартной библиотеки C, заменив все ненужные символы пробелами. После замены мы сможем легко извлечь слова из каждой прочитанной строки:

```
//: C07:WordList.cpp
// Вывод списка слов, встречающихся в документе
#include <algorithm>
#include <cctype>
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;

char replaceJunk(char c) {
   // В тексте остаются только алфавитные символы,
   // пробелы (в качестве разделителей) и символы '
   return (isalpha(c) || c == '\'') ? c : ' ';
}
int main(int argc, char* argv[]) {
  char* fname = "WordList.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  set<string> wordlist;
  string line;
  while(getline(in, line)) {
    transform(line.begin(), line.end(), line.begin(), replaceJunk);
    istringstream is(line);
    string word;
    while (is >> word)
      wordlist.insert(word);
    }
  // Вывод результатов:
  copy(wordlist.begin(), wordlist.end(),
       ostream_iterator<string>(cout, "\n"));
} ///:~
```

Вызов transform() заменяет все посторонние символы пробелами. Множество не только игнорирует повторяющиеся слова, но и сравнивает свои элементы объектом функции less<string> (второй аргумент по умолчанию шаблона set). В свою очередь, этот объект функции использует операторную функцию string::operator<(), поэтому слова следуют в алфавитном порядке.

Если вы всего лишь хотите получить отсортированную последовательность, не обязательно использовать множество. То же самое можно сделать при помощи алгоритма sort() (и множества других алгоритмов STL) с другими контейнерами STL. И все же вполне вероятно, что множество позволит решить эту задачу быстрее. Множество особенно хорошо приспособлено для поиска, поскольку его функция find() работает с логарифмической сложностью и заметно превосходит по скорости обобщенный алгоритм find(). Как вы помните, обобщенный алгоритм find() перебирает весь интервал до тех пор, пока не найдет искомый элемент, по-

этому в худшем случае он работает со сложностью N, а в среднем — со сложностью $N/2$. Впрочем, для заранее отсортированных последовательных контейнеров поиск может выполняться функцией equal_range() с логарифмической сложностью.

В следующем примере список слов строится при помощи итератора istreambuf_iterator. Программа перемещает из входного потока в объект string те символы, для которых функция isalpha() стандартной библиотеки C возвращает true:

```
//: C07:WordList2.cpp
// istreambuf_iterator и итераторы вставки
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "WordList2.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  istreambuf_iterator<char> p(in), end;
  set<string> wordlist;
  while (p != end) {
    string word;
    insert_iterator<string>
      ii(word, word.begin());
    // Поиск первого алфавитного символа:
    while (p != end && !isalpha (*p))
      ++p;
    // Копирование до первого неалфавитного символа:
    while (p != end && !isalpha (*p))
      ++*ii = *p++;
    if (word.size() != 0)
      wordlist.insert(word);
  }
  // Вывод результатов:
  copy(wordlist.begin(), wordlist.end(),
    ostream_iterator<string>(cout, "\n"));
} ///:~
```

Данный пример был предложен Натаном Майерсом (Nathan Myers), разработчиком итератора istreambuf_iterator и его «родственников». Итератор посимвольно читает информацию из потока. Хотя аргумент шаблона istreambuf_iterator наводит на мысль, что вместо char можно было бы извлечь, допустим, int, на самом деле это не так. Аргумент должен относиться к символьному типу — обычному (char) или расширенному.

После открытия файла итератор istreambuf_iterator с именем p присоединяется к потоку для чтения символов. Полученные слова сохраняются в множестве set<string> с именем wordlist.

Цикл while читает слова до тех пор, пока не обнаружит конец входного потока. Для проверки используется конструктор по умолчанию istreambuf_iterator, возвра-

щающий конечный итератор end. Следовательно, если вы хотите проверить, не достиг ли итератор конца потока, можно воспользоваться простым условием p != end.

Вторая разновидность итераторов, встречающихся в программе, — итератор вставки insert_iterator, с которым мы уже знакомы. Этот итератор вставляет новые объекты в контейнер. В данном случае «контейнером» является объект string с именем word, который с точки зрения insert_iterator вполне подходит на роль контейнера. Конструктор insert_iterator получает контейнер и итератор, установленный в начальную позицию вставки. Также можно было воспользоваться итератором back_insert_iterator, требующим, чтобы контейнер поддерживал функцию push_back() (объект string ее поддерживает).

После завершения подготовки программа начинает поиск первого алфавитного символа и увеличивает start до тех пор, пока такой символ не будет найден. Далее символы копируются из одного итератора в другой до тех пор, пока не встретится неалфавитный символ. Каждое непустое значение word включается в wordlist.

Выделение лексем из потока

В рассмотренных примерах используются разные способы выделения слов (в более общем виде — лексем) из потока, ни один из которых не обладает достаточной гибкостью. Поскольку итераторы занимают центральное место в работе контейнеров и алгоритмов STL, наиболее гибкое решение также должно использовать итератор. Класс TokenIterator представляет собой итератор-оболочку для любого другого итератора, выдающего символьные данные. Поскольку TokenIterator несомненно принадлежит к числу итераторов ввода (самой примитивной категории итераторов), он может поставлять входные данные любому алгоритму STL. Такой итератор не только полезен сам по себе, но и является хорошим примером, на котором можно учиться разрабатывать собственные итераторы[1].

Класс TokenIterator обладает двумя «степенями свободы». Во-первых, он позволяет выбрать тип итератора, поставляющего символьные входные данные. Во-вторых, вместо простого определения символов-ограничителей TokenIterator использует предикат — объект функции, оператор () которого получает char и решает, принадлежит ли он текущей лексеме. Хотя в двух приводимых примерах принадлежность символов к лексемам определяется статическим критерием, вы можете легко написать собственный объект функции, изменяющий свое состояние в процессе чтения символов.

Следующий заголовочный файл содержит два базовых предиката Isalpha и Delimiters, а также шаблон TokenIterator:

```
//: C07:TokenIterator.h
#ifndef TOKENITERATOR_H
#define TOKENITERATOR_H
#include <algorithm>
#include <cctype>
#include <functional>
#include <iterator>
#include <string>

struct Isalpha : std::unary_function<char, bool> {
```

[1] Этот пример также был предложен Натаном Майерсом (Nathan Myers).

```
  bool operator()(char c) { return std::isalpha(c); }
};

class Delimiters : std::unary_function<char, bool> {
  std::string exclude;
public:
  Delimiters() {}
  Delimiters(const std::string& excl) : exclude(excl) {}
  bool operator()(char c) {
    return exclude.find(c) == std::string::npos;
  }
};

template <class InputIter, class Pred = Isalpha>
class TokenIterator : public std::iterator<
    std::input_iterator_tag,std::string, std::ptrdiff_t> {
  InputIter first;
  InputIter last;
  std::string word;
  Pred predicate;
public:
  TokenIterator(InputIter begin, InputIter end,
    Pred pred = Pred())
    : first(begin), last(end), predicate(pred) {
      ++*this;
  }
  TokenIterator() {} // Конечный итератор
  // Префиксный инкремент:
  TokenIterator& operator++() {
    word.resize(0);
    first = std::find_if(first, last, predicate);
    while (first != last && predicate(*first))
      word += *first++;
    return *this;
  }
  // Постфиксный инкремент
  class CaptureState {
    std::string word;
  public:
    CaptureState(const std::string& w) : word(w) {}
    std::string operator*() { return word; }
  };
  CaptureState operator++(int) {
    CaptureState d(word);
    ++*this;
    return d;
  }
  // Разыменование (выборка значения):
  std::string operator*() const { return word; }
  std::string* operator->() const { return &(operator*()); }
  // Сравнение итераторов:
  bool operator==(const TokenIterator&) {
    return word.size() == 0 && first == last;
  }
  bool operator!=(const TokenIterator& rv) {
    return !(*this == rv);
  }
};
#endif // TOKENITERATOR_H ///:~
```

Класс TokenIterator объявляется производным от шаблона std::iterator. Может показаться, что он использует какие-то функциональные возможности шаблона std::iterator, но на самом деле это всего лишь способ «пометить» итератор, то есть передать контейнеру информацию о том, что он может делать. В аргументе шаблона, определяющем категорию итератора, передается значение input_iterator_tag — оно сообщает, что TokenIterator обладает минимальными возможностями итератора ввода и не может использоваться с алгоритмами, для которых нужны более совершенные итераторы. В остальном шаблон std::iterator не делает ничего, помимо определения нескольких полезных типов. Всю полезную функциональность нам придется реализовать самостоятельно.

Класс TokenIterator выглядит несколько странно: первому конструктору наряду с предикатом должны передаваться итераторы (начальный и конечный). Но как отмечалось выше, TokenIterator — всего лишь оболочка; он не узнает о достижении конца входных данных, поэтому присутствие конечного итератора в первом конструкторе необходимо. Наличие второго конструктора (конструктора по умолчанию) объясняется тем, что в алгоритмах STL, а также в пользовательских алгоритмах вызов конструктора без аргументов означает конец перебора. Но вся информация, по которой определяется достижение итератором TokenIterator конца входных данных, собрана в первом конструкторе. Таким образом, второй конструктор создает объект TokenIterator, который просто занимает положенное место в алгоритмах.

Вся основная функциональность класса сосредоточена в операторе ++. Он уничтожает текущее значение word вызовом string::resize(), а затем ищет первый символ, удовлетворяющий предикату (то есть начало новой лексемы), алгоритмом find_if(). Полученный итератор присваивается переменной first, в результате чего first перемещается в начало новой лексемы. Затем до тех пор, пока не будет достигнут конец входных данных, удовлетворяющие предикату входные символы копируются в word. В завершение своей работы префиксный инкремент возвращает объект TokenIterator. Чтобы обратиться к новой лексеме, достаточно разыменовать этот итератор.

В случае с постфиксным инкрементом необходим промежуточный объект CaptureState, в котором сохраняется старое значение для его последующего возвращения.

Для получения текущего значения используется обычный оператор разыменования *. Далее остается лишь определить операторные функции operator== и operator!= для проверки достижения конца входных данных. Как видно из листинга, аргумент функции operator== игнорируется — итератор просто проверяет, достигнут ли его внутренний итератор last. Обратите внимание на то, как operator!= определяется через operator==.

Качественный тест TokenIterator должен включать различные источники входных символов, в том числе streambuf_iterator, char* и deque<char>::iterator. В конце программы решается исходная задача со списком слов:

```
//: C07:TokenIteratorTest.cpp {-g++}
#include <fstream>
#include <iostream>
#include <vector>
#include <deque>
#include <set>
#include "TokenIterator.h"
```

```
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "TokenIteratorTest.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  ostream_iterator<string> out(cout, "\n");
  typedef istreambuf_iterator<char> IsbIt;
  IsbIt begin(in), isbEnd;
  Delimiters
    delimiters(" \t\n~;()\"<>:{}[]+-=&*#../\\");
  TokenIterator<IsbIt, Delimiters>
    wordIter(begin, isbEnd, delimiters),
    end;
  vector<string> wordlist;
  copy(wordIter, end, back_inserter(wordlist));
  // Вывод результата:
  copy(wordlist.begin(), wordlist.end(), out);
  *out++ = "----------------------------------";
  // Получение данных из символьного массива:
  char* cp =
    "typedef std::istreambuf_iterator<char> It";
  TokenIterator<char*, Delimiters>
    charIter(cp, cp + strlen(cp), delimiters),
    end2;
  vector<string> wordlist2;
  copy(charIter, end2, back_inserter(wordlist2));
  copy(wordlist2.begin(), wordlist2.end(), out);
  *out++ = "----------------------------------";
  // Получение данных из deque<char>:
  ifstream in2("TokenIteratorTest.cpp");
  deque<char> dc;
  copy(IsbIt(in2), IsbIt(), back_inserter(dc));
  TokenIterator<deque<char>::iterator,Delimiters>
    dcIter(dc.begin(), dc.end(), delimiters),
    end3;
  vector<string> wordlist3;
  copy(dcIter, end3, back_inserter(wordlist3));
  copy(wordlist3.begin(), wordlist3.end(), out);
  *out++ = "----------------------------------";
  // Повторение примера Wordlist.cpp:
  ifstream in3("TokenIteratorTest.cpp");
  TokenIterator<IsbIt, Delimiters>
    wordIter2((IsbIt(in3)), isbEnd, delimiters);
  set<string> wordlist4;
  while(wordIter2 != end)
    wordlist4.insert(*wordIter2++);
  copy(wordlist4.begin(), wordlist4.end(), out);
} ///:~
```

В тесте **istreambuf_iterator** создаются два итератора: один присоединяется к объекту **istream**, а другой создается конструктором по умолчанию и отмечает конец интервала. Оба итератора используются для создания объекта **TokenIterator**, разбирающего входной поток на лексемы; конструктор по умолчанию создает «фальшивый» объект **TokenIterator**, изображающий конечный итератор (в действительности он просто занимает положенное место и игнорируется). Объект **TokenIterator** создает

объекты string, вставляемые в контейнер с элементами string — во всех случаях, кроме последнего, используется вектор vector<string> (также можно было объединить результаты в string). В остальном итератор TokenIterator работает так же, как любой другой.

При определении двустороннего итератора (а следовательно, и итератора произвольного доступа) вы можете «бесплатно» получить обратные итераторы при помощи адаптера std::reverse_iterator. Если в программе уже определен итератор для контейнера с двусторонним перебором, вы можете получить обратный итератор на базе прямого итератора. Для этого в класс контейнера включается фрагмент вида

```
// Предполагается, что "iterator" -- вложенный тип итератора
typedef std::reverse_iterator<iterator> reverse_iterator;
reverse_iterator rbegin() {return reverse_iterator(end());
reverse_iterator rend()   {return reverse_iterator(begin());
```

Адаптер std::reverse_iterator выполняет всю работу за вас. Например, при разыменовании обратного итератора оператором * он автоматически уменьшает временную копию своего прямого итератора, чтобы вернуть правильный элемент, потому что обратные итераторы логически установлены на одну позицию за тем элементом, на который они ссылаются.

Стек

Контейнер stack, наряду с queue и priority_queue, относится к категории *контейнерных адаптеров*. Это означает, что он адаптирует один из базовых последовательных контейнеров для хранения своих данных. Впрочем, этот термин лишь вносит некоторую путаницу — тот факт, что эти контейнеры называются «адаптерами», имеет значение разве что с точки зрения создания библиотеки. При практическом применении важно не то, как называется контейнер, а то, что он решает вашу задачу. Возможно, в отдельных случаях было бы полезно знать о возможности выбора альтернативной реализации или построения адаптера на базе существующего объекта контейнера, но обычно адаптеры не наделяются соответствующей функциональностью. И если в других книгах тот или иной контейнер постоянно именуется адаптером, мы будем напоминать об этом обстоятельстве лишь тогда, когда оно приносит реальную пользу.

В следующем примере продемонстрированы три варианта реализации stack<string>. В первом (стандартном) варианте используется дек, а в двух других — вектор и список:

```
//: C07:Stack1.cpp
// Работа со стеком STL
#include <fstream>
#include <iostream>
#include <list>
#include <stack>
#include <string>
#include <vector>
using namespace std;

// Переставьте комментарии, чтобы использовать другую версию стека.
typedef stack<string> Stack1; // По умолчанию: deque<string>
```

```
// typedef stack<string, vector<string> > Stack2;
// typedef stack<string, list<string> > Stack3;

int main() {
  ifstream in("Stack1.cpp");
  Stack1 textlines; // Попробуйте использовать другие версии.
  // Чтение файла и сохранение строк в стеке:
  string line;
  while(getline(in, line))
    textlines.push(line + "\n");
  // Вывод строк и их извлечение из стека:
  while(!textlines.empty()) {
    cout << textlines.top();
    textlines.pop();
  }
} ///:~
```

Если раньше вы работали с другими классами стеков, функции top() и pop() могут показаться противоестественными. Вместо верхнего элемента стека, как можно было бы ожидать, функция pop() возвращает void. Если вас интересует значение верхнего элемента, получите ссылку на него функцией top(). Такой вариант более эффективен, поскольку традиционная функция pop() возвращает значение вместо ссылки, а для этого необходим вызов копирующего конструктора. Что еще важнее, такое решение *безопасно по отношению к исключениям* (эта тема обсуждалась в главе 1). Если бы функция pop() одновременно изменяла состояние стека и пыталась вернуть верхний элемент, исключение в копирующем конструкторе элемента могло бы привести к его потере. При использовании шаблона stack (или приоритетной очереди priority_queue, о которой речь пойдет далее) можно эффективно ссылаться на элемент top() сколько угодно раз, а затем удалить верхний элемент из стека вызовом pop(). Наверное, если бы разработчики использовали вместо «pop» какое-нибудь другое имя, недоразумений было бы меньше.

Шаблон stack обладает простым интерфейсом — в сущности, все его функции уже были перечислены. Поскольку все обращения к элементам ограничиваются вершиной, стек не поддерживает итераторы для перебора элементов. Также отсутствуют изощренные формы инициализации, но при необходимости можно использовать средства того контейнера, на базе которого реализован стек. Допустим, имеется функция, рассчитанная на интерфейс стека, но в остальной части программы объекты должны храниться в виде списка. Следующая программа сохраняет каждую строку файла вместе с информацией о количестве начальных пробелов в этой строке (например, это может стать отправной точкой для переформатирования исходного кода программы).

```
//: C07:Stack2.cpp
// Преобразование списка в стек
#include <iostream>
#include <fstream>
#include <stack>
#include <list>
#include <string>
#include <cstddef>
using namespace std;

// Ориентируется на стек:
template<class Stk>
```

```
void stackOut(Stk& s, ostream& os = cout) {
  while(!s.empty()) {
    os << s.top() << "\n";
    s.pop();
  }
}

class Line {
  string line; // Без начальных пробелов
  size_t lspaces; // Количество начальных пробелов
public:
  Line(string s) : line(s) {
    lspaces = line.find_first_not_of(' ');
    if(lspaces == string::npos)
      lspaces = 0;
    line = line.substr(lspaces);
  }
  friend ostream& operator<<(ostream& os, const Line& l) {
    for(size_t i = 0; i < l.lspaces; i++)
      os << ' ';
    return os << l.line;
  }
  // Прочие функции...
};

int main() {
  ifstream in("Stack2.cpp");
  list<Line> lines;
  // Чтение файла и сохранение строк в списке:
  string s;
  while(getline(in, s))
    lines.push_front(s);
  // Преобразование списка в стек для вывода:
  stack<Line, list<Line> > stk(lines);
  stackOut(stk);
} ///:~
```

Функция, ориентированная на интерфейс stack, просто отправляет каждый объект top() в ostream, а затем удаляет его вызовом pop(). Класс Line определяет количество начальных пробелов и сохраняет содержимое строки *без них*. Оператор вывода в ostream снова вставляет начальные пробелы, чтобы строка выводилась в исходном виде, однако количество пробелов легко изменяется модификацией lspaces (соответствующая функция здесь не приведена). В функции main() содержимое входного файла читается в list<Line>, после чего каждая строка в списке копируется в стек, выводимый в stackOut().

Перебор элементов стека невозможен; тем самым подчеркивается тот факт, что операции со стеком должны выполняться при создании стека. Функциональность стека можно имитировать при помощи вектора и его функций back(), push_back() и pop_back(), при этом становятся доступными все дополнительные возможности вектора. Программу Stack1.cpp можно записать в следующем виде:

```
//: C07:Stack3.cpp
// Эмуляция стека на базе вектора: измененная версия Stack1.cpp
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
```

```
using namespace std;

int main() {
  ifstream in("Stack3.cpp");
  vector<string> textlines;
  string line;
  while(getline(in, line))
    textlines.push_back(line + "\n");
  while(!textlines.empty()) {
    cout << textlines.back();
    textlines.pop_back();
  }
} ///:~
```

Программа выводит те же результаты, что и Stack1.cpp, но теперь в ней могут выполняться векторные операции. Список также поддерживает включение элементов с начала контейнера, но обычно этот вариант уступает по эффективности функции push_back() вектора (кроме того, включение элементов в начало контейнера более эффективно выполняется в деках, чем в списках).

Очередь

Очередь (контейнер queue) представляет собой дек с ограниченными возможностями — элементы добавляются только с одного конца контейнера, а извлекаются только с другого конца. С функциональной точки зрения очередь всегда можно заменить деком, и тогда в вашем распоряжении также появятся все дополнительные возможности дека. Очередь используется вместо дека только в одной ситуации — когда вы хотите подчеркнуть, что контейнер ведет себя именно как очередь.

Класс queue, как и stack, принадлежит к категории контейнерных адаптеров, то есть строится на базе других последовательных контейнеров. Как нетрудно предположить, идеальная реализация очереди создается на базе дека, поэтому по умолчанию queue использует аргумент шаблона deque. Необходимость в выборе другой реализации встречается редко.

Очереди часто требуются при моделировании систем, в которых отдельные элементы ожидают обслуживания со стороны других элементов. Классическим примером такого рода является «задача кассира». Клиенты приходят в банк, становятся в очередь и обслуживаются несколькими кассирами. Поскольку клиенты появляются со случайными интервалами, а продолжительность их обслуживания неизвестна заранее, предсказать длину очереди в конкретный момент времени невозможно. Тем не менее, можно смоделировать ситуацию и посмотреть, что получится.

В реалистичной модели каждый клиент и кассир должны быть представлены отдельными программными потоками. К сожалению, в стандарте C++ поддержка многопоточных приложений не предусмотрена. С другой стороны, небольшие изменения в программе позволяют имитировать многопоточность на приемлемом уровне[1].

В многопоточной программе одновременно работают несколько программных потоков, совместно использующих общее адресное пространство. На практике ко-

[1] Мы вернемся к теме многопоточности в главе 11.

личество процессоров обычно меньше количества программных потоков (а в большинстве систем установлен только один процессор). Чтобы создать иллюзию того, что каждый программный поток работает на отдельном процессоре, *механизм квантования* периодически прерывает текущий программный поток и передает управление другому. Модель с автоматическим прерыванием и запуском программных потоков называется моделью с *вытесняющей многопоточностью*; она избавляет программиста от необходимости самостоятельно обеспечивать передачу управления между программными потоками.

Существует и другая модель: каждый поток добровольно уступает процессор планировщику, который находит другой поток и передает ему управление. В нашем примере квантование имитируется на уровне классов системы. Программные потоки будут представляться кассирами (клиенты ведут себя пассивно). В классе кассира присутствует функция run(), работающая в цикле. Она выполняется в течение определенного количества «квантов», а затем просто возвращает управление. Несмотря на свои скромные размеры, следующая программа на удивление прилично имитирует многопоточность:

```
//: C07:BankTeller.cpp {RunByHand}
// Моделирование банковского обслуживания
// на базе очереди и имитации многопоточности
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <list>
#include <queue>
using namespace std;

class Customer {
  int serviceTime;
public:
  Customer() : serviceTime(0) {}
  Customer(int tm) : serviceTime(tm) {}
  int getTime() { return serviceTime; }
  void setTime(int newtime) { serviceTime = newtime; }
  friend ostream&
  operator<<(ostream& os, const Customer& c) {
    return os << '[' << c.serviceTime << ']';
  }
};

class Teller {
  queue<Customer>& customers;
  Customer current;
  enum { SLICE = 5 };
  int ttime; // Остаток времени в кванте
  bool busy; // Кассир обслуживает клиента?
public:
  Teller(queue<Customer>& cq)
    : customers(cq), ttime(0), busy(false) {}
  Teller& operator=(const Teller& rv) {
    customers = rv.customers;
    current = rv.current;
    ttime = rv.ttime;
    busy = rv.busy;
```

```
    return *this;
  }
  bool isBusy() { return busy; }
  void run(bool recursion = false) {
    if(!recursion)
      ttime = SLICE;
    int servtime = current.getTime();
    if(servtime > ttime) {
      servtime -= ttime;
      current.setTime(servtime);
      busy = true; // Продолжать обслуживание текущего клиента
      return;
    }
    if(servtime < ttime) {
      ttime -= servtime;
      if(!customers.empty()) {
        current = customers.front();
        customers.pop(); // Удаление из очереди
        busy = true;
        run(true); // Рекурсия
      }
      return;
    }
    if(servtime == ttime) {
      // Клиент обслужен:
      current = Customer(0);
      busy = false;
      return; // Завершение текущего кванта
    }
  }
};

// Наследование для обращения к защищенной реализации:
class CustomerQ : public queue<Customer> {
public:
  friend ostream&
  operator<<(ostream& os, const CustomerQ& cd) {
    copy(cd.c.begin(), cd.c.end(),
      ostream_iterator<Customer>(os, ""));
    return os;
  }
};

int main() {
  CustomerQ customers;
  list<Teller> tellers;
  typedef list<Teller>::iterator TellIt;
  tellers.push_back(Teller(customers));
  srand(time(0)); // Раскрутка генератора случайных чисел
  clock_t ticks = clock();
  // Запуск имитации минимум на 5 секунд:
  while(clock() < ticks + 5 * CLOCKS_PER_SEC) {
    // Заполнение очереди случайным количеством клиентов
    // со случайным временем обслуживания:
    for(int i = 0; i < rand() % 5; i++)
      customers.push(Customer(rand() % 15 + 1));
    cout << '{' << tellers.size() << '}'
      << customers << endl;
    // Обслуживание клиентов:
```

```
    for(TellIt i = tellers.begin();
      i != tellers.end(); i++)
      (*i).run();
    cout << '{' << tellers.size() << '}'
      << customers << endl;
    // Если очередь слишком длинна, создаем нового кассира:
    if(customers.size() / tellers.size() > 2)
      tellers.push_back(Teller(customers));
    // Если очередь слишком коротка, убираем кассира:
    if(tellers.size() > 1 &&
      customers.size() / tellers.size() < 2)
      for(TellIt i = tellers.begin();
        i != tellers.end(); i++)
        if(!(*i).isBusy()) {
          tellers.erase(i);
          break; // Выход из цикла
        }
  }
} ///:~
```

Каждому клиенту назначается определенное время обслуживания — количество квантов времени, которое должно быть потрачено кассиром на данного клиента. Время обслуживания определяется для каждого клиента случайным образом. Также мы не знаем, сколько клиентов будет поступать в каждый интервал, и эта величина также должна определяться случайным образом.

Объекты Customer хранятся в очереди queue<Customer>. Каждый объект Teller содержит ссылку на эту очередь. Завершив обслуживание текущего объекта Customer, объект-кассир Teller берет из очереди новый объект клиента Customer и начинает «обслуживание», то есть уменьшает время обслуживания Customer в течение каждого выделенного кванта. Вся логика обслуживания сосредоточена в функции run(). Фактически эта функция представляет собой команду if с выбором одного их трех вариантов: если время, необходимое для обслуживания клиента, меньше времени, оставшегося в текущем кванте кассира, равно или больше. Если после завершения работы с клиентом у кассира еще остается свободное время, он берет нового клиента и рекурсивно вызывает run().

Очередь, как и стек, не обладает функциональностью других базовых последовательных контейнеров. В частности, невозможно получить итератор для перебора элементов очереди. Тем не менее, последовательный контейнер, на базе которого реализована очередь, хранится в queue в виде защищенной переменной класса. В соответствии со стандартом C++ этой переменной присваивается имя c, а это означает, что для доступа к базовой реализации вы можете породить класс, производный от queue. Класс CustomerQ именно это и делает с единственной целью: определить операторную функцию ofstream operator<< для перебора очереди и вывода ее элементов.

Все управление имитацией производится в цикле while функции main(). Цикл использует процессорные такты (см. <ctime>) для проверки того, что имитация продолжается минимум 5 секунд. В начале каждой итерации цикл генерирует случайное количество клиентов со случайными временами обслуживания. Программа выводит количество кассиров и содержимое очереди, чтобы вы могли оценить текущее состояние системы. После отработки каждого кассира программа снова выводит сведения о состоянии системы. В этот момент происходит автоматическая регулировка системы, для чего количество клиентов сравнивается с количе-

ством кассиров. Если очередь стала слишком длинной, в системе создается новый кассир, а при недостаточной длине очереди кассир удаляется. Попробуйте поэкспериментировать с разными схемами подключения и удаления кассиров для достижения оптимального результата. Если в программе изменяется только эта часть, схему изменения числа кассиров стоит выделить в отдельный объект.

Мы вернемся к этому примеру при рассмотрении многопоточности в главе 11.

Приоритетная очередь

При занесении объекта в приоритетную очередь (контейнер priority_queue) функцией push() позиция нового объекта определяется функцией сравнения или объектом функции (по умолчанию используется шаблон less, но вы также можете предоставить собственный критерий). Приоритетная очередь гарантирует, что верхний элемент очереди, возвращаемый функцией top(), обладает наибольшим значением (приоритетом). Когда все необходимые операции с элементом будут выполнены, верхний элемент выталкивается функцией pop(), а его место занимает следующий элемент. Таким образом, приоритетная очередь почти не отличается по интерфейсу от стека, но работает несколько иначе.

Приоритетная очередь также относится к категории контейнерных адаптеров, работающих на базе последовательных контейнеров. По умолчанию в реализации priority_queue используется вектор.

Ниже приведен тривиальный пример приоритетной очереди для типа int:

```
//: C07:PriorityQueue1.cpp
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <queue>
using namespace std;

int main() {
  priority_queue<int> pqi;
  srand(time(0)); // Раскрутка генератора случайных чисел
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

Мы заносим в приоритетную очередь 100 случайных чисел от 0 до 24. При запуске этой программы становится видно, что наибольшие значения находятся на первых местах, и приоритетная очередь может содержать дубликаты. Следующий пример показывает, как изменить порядок следования элементов при помощи пользовательской функции или объекта функции. На этот раз наибольший приоритет назначается числам с минимальными значениями:

```
//: C07:PriorityQueue2.cpp
// Изменение приоритета
#include <cstdlib>
#include <ctime>
#include <functional>
```

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
  priority_queue<int, vector<int>, greater<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

Более интересный пример — список задач, в котором каждый объект содержит строку с описанием задачи, а также два приоритета (первичный и вторичный):

```
//: C07:PriorityQueue3.cpp
// Нетривиальный пример приоритетной очереди
#include <iostream>
#include <queue>
#include <string>
using namespace std;

class ToDoItem {
  char primary;
  int secondary;
  string item;
public:
  ToDoItem(string td, char pri ='A', int sec =1)
    : item(td), primary(pri), secondary(sec) {}
  friend bool operator<(
    const ToDoItem& x, const ToDoItem& y) {
    if(x.primary > y.primary)
      return true;
    if(x.primary == y.primary)
      if(x.secondary > y.secondary)
        return true;
    return false;
  }
  friend ostream&
  operator<<(ostream& os, const ToDoItem& td) {
    return os << td.primary << td.secondary
      << ": " << td.item;
  }
};

int main() {
  priority_queue<ToDoItem> toDoList;
  toDoList.push(ToDoItem("Empty trash", 'C', 4));
  toDoList.push(ToDoItem("Feed dog", 'A', 2));
  toDoList.push(ToDoItem("Feed bird", 'B', 7));
  toDoList.push(ToDoItem("Mow lawn", 'C', 3));
  toDoList.push(ToDoItem("Water lawn", 'A', 1));
  toDoList.push(ToDoItem("Feed cat", 'B', 1));
  while(!toDoList.empty()) {
    cout << toDoList.top() << endl;
    toDoList.pop();
  }
} ///:~
```

Чтобы оператор < для вывода ToDoItem работал с шаблоном less<>, он должен быть оформлен в виде внешней функции (а не функции класса). Все остальное происходит автоматически. Результат выполнения программы выглядит так:

```
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
```

Приоритетная очередь не поддерживает перебор элементов, однако ее поведение можно имитировать при помощи вектора (и получить полный доступ к функциональности вектора). Попробуйте разобраться в реализации приоритетной очереди, использующей функции make_heap(), push_heap() и pop_heap(). Эти функции являются «сердцем» приоритетной очереди. В сущности, можно сказать, что куча *является* приоритетной очередью, а priority_queue — всего лишь ее интерфейсная оболочка. Имитация получается достаточно прямолинейной, хотя на первый взгляд может показаться, что существует более короткий путь. Поскольку базовый контейнер priority_queue хранится в защищенной переменной класса (которой в соответствии со стандартом C++ назначается идентификатор c), можно объявить производный класс и получить доступ к базовой реализации:

```
//: C07:PriorityQueue4.cpp
// Получение доступа к базовой реализации
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

class PQI : public priority_queue<int> {
public:
  vector<int>& impl() { return c; }
};

int main() {
  PQI pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  copy(pqi.impl().begin(), pqi.impl().end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

При запуске этой программы выясняется, что элементы вектора не упорядочены по убыванию. Другими словами, порядок их следования отличен от того, который будет получен последовательными вызовами функции pop() для приоритетной очереди. Похоже, если вы хотите создать вектор, имитирующий приоритетную очередь, придется делать это вручную — примерно так:

```
//: C07:PriorityQueue5.cpp
// Построение собственной приоритетной очереди
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template<class T, class Compare>
class PQV : public vector<T> {
  Compare comp;
public:
  PQV(Compare cmp = Compare()) : comp(cmp) {
    make_heap(this->begin(), this->end(), comp);
  }
  const T& top() { return this->front(); }
  void push(const T& x) {
    this->push_back(x);
    push_heap(this->begin(), this->end(), comp);
  }
  void pop() {
    pop_heap(this->begin(), this->end(), comp);
    pop_back();
  }
};

int main() {
  PQV<int, less<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  copy(pqi.begin(), pqi.end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

Однако и эта программа ведет себя так же, как предыдущая! На самом деле в базовом векторе хранится не отсортированная последовательность элементов, а структура данных, называемая *кучей*. Она представляет дерево приоритетной очереди (хранящееся в виде линейной векторной структуры), но при последовательном переборе элементов кучи вы не получите ожидаемого порядка следования элементов приоритетной очереди. Вообще говоря, нужного результата можно добиться простым вызовом функции sort_heap(), но это решение работает только один раз; после вызова куча превращается в отсортированный список. Чтобы снова использовать его как кучу, необходимо снова вызвать функцию make_heap(). Мы можем инкапсулировать этот вызов в пользовательской реализации очереди:

```
//: C07:PriorityQueue6.cpp
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
```

```
#include <iterator>
#include <queue>
using namespace std;

template<class T, class Compare>
class PQV : public vector<T> {
  Compare comp;
  bool sorted;
  void assureHeap() {
    if(sorted) {
      // Обратное преобразование списка в кучу:
      make_heap(this->begin(), this->end(), comp);
      sorted = false;
    }
  }
public:
  PQV(Compare cmp = Compare()) : comp(cmp) {
    make_heap(this->begin(), this->end(), comp);
    sorted = false;
  }
  const T& top() {
    assureHeap();
    return this->front();
  }
  void push(const T& x) {
    assureHeap();
    this->push_back(x);  // Занести элемент в конец
    // Внести изменения в кучу:
    push_heap(this->begin(), this->end(), comp);
  }
  void pop() {
    assureHeap();
    // Перемещение верхнего элемента в последнюю позицию:
    pop_heap(this->begin(), this->end(), comp);
    // Удаление элемента:
    pop_back();
  }
  void sort() {
    if(!sorted) {
      sort_heap(this->begin(), this->end(), comp);
      reverse(this->begin(), this->end());
      sorted = true;
    }
  }
};

int main() {
  PQV<int, less<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++) {
    pqi.push(rand() % 25);
    copy(pqi.begin(), pqi.end(),
      ostream_iterator<int>(cout, " "));
    cout << "\n-----\n";
  }
  pqi.sort();
  copy(pqi.begin(), pqi.end(),
    ostream_iterator<int>(cout, " "));
  cout << "\n-----\n";
```

```
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

Если флаг sorted равен true, значит, вектор не оформлен в виде кучи, а содержит отсортированную последовательность. Функция assureHeap() гарантирует, что он будет возвращен к нужной форме перед выполнением каких-либо операций, специфических для кучи. Теперь первый цикл for в функции main() выводит содержимое кучи по мере ее построения.

В двух предыдущих программах нам пришлось использовать вроде бы лишний префикс this->. Хотя некоторые компиляторы позволяют обойтись без него, по стандарту C++ он необходим. Обратите внимание: класс PQV является производным от vector<T>, поэтому имена функций begin() и end(), унаследованных от vector<T>, принадлежат к числу зависимых[1]. Компилятор не может разрешать имена из зависимых базовых классов (в данном случае vector) в определении шаблона, потому что для данной специализации может быть использована переопределенная версия шаблона, которая не содержит соответствующих членов. Специальные правила оформления имен предотвращают ситуацию, при которой в одних случаях вызывается функция базового класса, а в других — функция из внешней области видимости (например, глобальной). Компилятор не знает, что вызов begin() является зависимым, поэтому мы должны сообщить ему об этом при помощи уточнения this->[2]. Так компилятор узнает, что begin() находится в области видимости PQV, и поэтому нужно ожидать полной специализации PQV. Если бы уточняющий префикс отсутствовал, то компилятор попытался бы использовать для имен begin и end раннее разрешение на стадии определения шаблона (такая попытка завершилась бы неудачей, потому что во внешних лексических областях видимости такие имена не объявлены). В нашей программе компилятор ожидает до момента создания специализации pqi, а затем находит правильные специализации begin() и end() в классе vector<int>.

Единственный недостаток этого решения состоит в том, что пользователь должен помнить о необходимости вызова sort() перед просмотром сортированной последовательности (хотя теоретически можно переопределить все функции, создающие итераторы, так чтобы они гарантировали сортировку). Другое решение — задействовать приоритетную очередь, которая вектором формально не является, и строить вектор по запросу пользователя:

```
//: C07:PriorityQueue7.cpp
// Приоритетная очередь, создающая
// отсортированный вектор по запросу
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
#include <vector>
using namespace std;

template<class T, class Compare> class PQV {
```

[1] То есть зависят от параметра шаблона. См. раздел «Разрешение имен» в главе 5.

[2] Как объяснялось в главе 5, подойдет любое допустимое уточнение, например, PQV::.

```
  vector<T> v;
  Compare comp;
public:
  // Вызывать make_heap() не нужно;
  // контейнер не содержит элементов
  PQV(Compare cmp = Compare()) : comp(cmp) {}
  void push(const T& x) {
    // Занести элемент в конец:
    v.push_back(x);
    // Внести изменения в кучу:
    push_heap(v.begin(), v.end(), comp);
  }
  void pop() {
    // Перемещение верхнего элемента в последнюю позицию:
    pop_heap(v.begin(), v.end(), comp);
    // Удаление элемента:
    v.pop_back();
  }
  const T& top() { return v.front(); }
  bool empty() const { return v.empty(); }
  int size() const { return v.size(); }
  typedef vector<T> TVec;
  TVec vector() {
    TVec r(v.begin(), v.end());
    // Остается отсортировать существующую кучу
    sort_heap(r.begin(), r.end(), comp);
    // Приведение к порядку приоритетной очереди:
    reverse(r.begin(), r.end());
    return r;
  }
};

int main() {
  PQV<int, less<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  const vector<int>& v = pqi.vector();
  copy(v.begin(), v.end(),
    ostream_iterator<int>(cout, " "));
  cout << "\n----------\n";
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

Шаблон класса **PQV** построен по тому же образцу, что и шаблон **priority_queue** библиотеки STL. Но в него включена дополнительная функция **getVector()** для создания нового вектора, являющегося копией содержимого **PQV** (то есть базовой кучи). Далее копия сортируется (при этом вектор **PQV** остается неизменным), а ее элементы переставляются в обратном порядке, чтобы порядок перебора в новом векторе соответствовал порядку извлечения элементов из приоритетной очереди.

Совмещение этой методики с наследованием от **priority_queue** (см. пример **PriorityQueue4.cpp**) позволяет получить более компактный код:

```
//: C07:PriorityQueue8.cpp
// Более компактная версия PriorityQueue7.cpp
```

```
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
using namespace std;

template<class T> class PQV : public priority_queue<T> {
public:
  typedef vector<T> TVec;
  TVec vector() {
    TVec r(c.begin(), c.end());
    // Базовая куча хранится в переменной c
    sort_heap(r.begin(), r.end(), comp);
    // Приведение к порядку приоритетной очереди:
    reverse(r.begin(), r.end());
    return r;
  }
};

int main() {
  PQV<int> pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  const vector<int>& v = pqi.vector();
  copy(v.begin(), v.end(),
    ostream_iterator<int>(cout, " "));
  cout << "\n----------\n";
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

Благодаря своей компактности это решение оказывается самым простым и удобным. Вдобавок оно гарантирует, что пользователь всегда получит доступ к отсортированному вектору. Теоретически существует только одна проблема: функция **getVector()** возвращает **vector<T>** по значению, что может привести к большим затратам ресурсов при сложных типах параметра T.

Битовые поля

Принято считать, что язык C «близок к оборудованию», поэтому многих программистов раздражает отсутствие в нем собственного варианта представления двоичных чисел. Есть десятичные числа, есть шестнадцатеричные (которые можно терпеть только из-за удобной группировки битов в уме), но работать в восьмеричной системе? Фу. Ни в одной спецификации микросхемы регистры не описываются ни в восьмеричной, ни в шестнадцатеричной системах — только в двоичной. Однако C не позволяет использовать запись вида **0b0101101**, абсолютно естественную для языка, «близкого к оборудованию».

Хотя в C++ естественная двоичная запись так и не появилась, ситуация несколько улучшилась с появлением классов **bitset** и **vector<bool>**. Оба класса пред-

назначены для выполнения операций с группами двоичных флагов[1]. Основных различий между этими типами два.

- Битовое поле (bitset) содержит фиксированное количество битов. Количество битов задается в аргументе шаблона bitset. Класс vector<bool>, как и обычный вектор, расширяется динамически и позволяет хранить любое количество элементов типа bool.
- Шаблон bitset проектировался специально для достижения максимального быстродействия при операциях с битами и не принадлежит к числу «обычных» контейнеров STL. Из-за этого он не поддерживает итераторы. Размер поля в битах, передаваемый в параметре шаблона, известен на стадии компиляции, благодаря чему базовый целочисленный массив может храниться в стеке времени выполнения. С другой стороны, контейнер vector<bool> является специализацией класса vector и поддерживает все операции обычного вектора — просто эта специализация рассчитана на эффективное хранение данных типа bool.

Тривиальное преобразование между типами bitset и vector<bool> невозможно. Отсюда можно сделать вывод, что эти два контейнера предназначены для разных целей. Более того, ни один из них не является традиционным «контейнером STL». Шаблон bitset обладает собственным интерфейсом для работы с отдельными битами и не имеет ничего общего с рассмотренными контейнерами STL. Специализация вектора vector<bool> похожа на контейнер STL, но ведет себя несколько иначе. Эти отличия будут рассматриваться далее.

Контейнер bitset<n>

В аргументе шаблона bitset передается беззнаковое целое число — количество битов в поле. Следовательно, bitset<10> и bitset<20> являются разными типами, поэтому между ними не могут выполняться операции сравнения, присваивания и т. д.

Битовые поля обеспечивают выполнение основных поразрядных операций в наиболее эффективной форме. Однако во внутренней реализации bitset биты логически упаковываются в массив с целочисленными элементами (обычно unsigned long с минимальной длиной 32 бит). Единственное преобразование bitset в число определено только для unsigned long (оно выполняется функцией to_ulong()).

Следующий пример демонстрирует практически все возможности битовых полей (отсутствующие операции несущественны или тривиальны). Справа от выходных данных каждой операции выводится ее краткое описание, а все биты выровнены так, чтобы их было удобно сравнивать с исходными значениями. Если вы еще недостаточно хорошо разбираетесь в поразрядных операциях, эта программа должна помочь.

```
//: C07:BitSet.cpp {-bor}
// Работа с классом bitset
#include <bitset>
#include <climits>
```

[1] Чак Эллисон (Chuck Allison) спроектировал и разработал исходные эталонные реализации bitset и bitstring (предшественника vector<bool>) во время своей активной работы в комитете по стандартизации C++ в середине 1990-х годов.

```
#include <cstdlib>
#include <ctime>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;

const int SZ = 32;
typedef bitset<SZ> BS;

template<int bits> bitset<bits> randBitset() {
  bitset<bits> r(rand());
  for(int i = 0; i < bits/16 - 1; i++) {
    r <<= 16;
    // Объединение операцией "OR" с младшими 16 битами
    // нового значения:
    r |= bitset<bits>(rand());
  }
  return r;
}

int main() {
  srand(time(0));
  cout << "sizeof(bitset<16>) = "
    << sizeof(bitset<16>) << endl;
  cout << "sizeof(bitset<32>) = "
    << sizeof(bitset<32>) << endl;
  cout << "sizeof(bitset<48>) = "
    << sizeof(bitset<48>) << endl;
  cout << "sizeof(bitset<64>) = "
    << sizeof(bitset<64>) << endl;
  cout << "sizeof(bitset<65>) = "
    << sizeof(bitset<65>) << endl;
  BS a(randBitset<SZ>()), b(randBitset<SZ>());
  // Преобразование битового поля в число:
  unsigned long ul = a.to_ulong();
  cout << a << endl;
  // Преобразование строки в битовое поле:
  string cbits("111011010110111");
  cout << "as a string = " << cbits <<endl;
  cout << BS(cbits) << " [BS(cbits)]" << endl;
  cout << BS(cbits, 2)
    << " [BS(cbits, 2)]" << endl;
  cout << BS(cbits, 2, 11)
    << " [BS(cbits, 2, 11)]" << endl;
  cout << a << " [a]" << endl;
  cout << b << " [b]" << endl;
  // Поразрядная операция AND:
  cout << (a & b) << " [a & b]" << endl;
  cout << (BS(a) &= b) << " [a &= b]" << endl;
  // Поразрядная операция OR:
  cout << (a | b) << " [a | b]" << endl;
  cout << (BS(a) |= b) << " [a |= b]" << endl;
  // Поразрядная операция исключающего OR:
  cout << (a ^ b) << " [a ^ b]" << endl;
  cout << (BS(a) ^= b) << " [a ^= b]" << endl;
  cout << a << " [a]" << endl; // Для сравнения
  // Логический сдвиг влево (с заполнением нулями):
  cout << (BS(a) <<= SZ/2)
    << " [a <<= (SZ/2)]" << endl;
```

```
  cout << (a << SZ/2) << endl;
  cout << a << " [a]" << endl; // Для сравнения
  // Логический сдвиг вправо (с заполнением нулями):
  cout << (BS(a) >>= SZ/2)
    << " [a >>= (SZ/2)]" << endl;
  cout << (a >> SZ/2) << endl;
  cout << a << " [a]" << endl; // Для сравнения
  cout << BS(a).set() << " [a.set()]" << endl;
  for(int i = 0; i < SZ; i++)
    if(!a.test(i)) {
      cout << BS(a).set(i)
        << " [a.set(" << i <<")]" << endl;
      break; // Только одна операция
    }
  cout << BS(a).reset() << " [a.reset()]"<< endl;
  for(int j = 0; j < SZ; j++)
    if(a.test(j)) {
      cout << BS(a).reset(j)
        << " [a.reset(" << j <<")]" << endl;
      break; // Только одна операция
    }
  cout << BS(a).flip() << " [a.flip()]" << endl;
  cout << ~a << " [~a]" << endl;
  cout << a << " [a]" << endl; // Для сравнения
  cout << BS(a).flip(1) << " [a.flip(1)]"<< endl;
  BS c;
  cout << c << " [c]" << endl;
  cout << "c.count() = " << c.count() << endl;
  cout << "c.any() = "
    << (c.any() ? "true" : "false") << endl;
  cout << "c.none() = "
    << (c.none() ? "true" : "false") << endl;
  c[1].flip(); c[2].flip();
  cout << c << " [c]" << endl;
  cout << "c.count() = " << c.count() << endl;
  cout << "c.any() = "
    << (c.any() ? "true" : "false") << endl;
  cout << "c.none() = "
    << (c.none() ? "true" : "false") << endl;
  // Индексация:
  c.reset();
  for(int k = 0; k < c.size(); k++)
    if(k % 2 == 0)
      c[k].flip();
  cout << c << " [c]" << endl;
  c.reset();
  // Присваивание логических значений:
  for(int ii = 0; ii < c.size(); ii++)
    c[ii] = (rand() % 100) < 25;
  cout << c << " [c]" << endl;
  // Проверка логических значений:
  if(c[1])
    cout << "c[1] == true";
  else
    cout << "c[1] == false" << endl;
} ///:~
```

Чтобы результаты не были слишком однообразными, функция **randBitset()** генерирует случайные битовые поля. В этой функции оператор **<<=** сдвигает очеред-

ные 16 случайных битов влево вплоть до заполнения битового поля (параметризованного по длине). Сгенерированное число и очередные 16 бит объединяются оператором |=.

Функция main() сначала демонстрирует дискретный характер выделения памяти для битовых полей. Если поле содержит менее 32 бит, то sizeof возвращает 4 (4 байт = 32 бит), размер типа long в большинстве реализаций. Если поле содержит от 32 до 64 бит, для их хранения требуются два числа типа long, если больше 64 — 3 числа, и так далее. Следовательно, наиболее эффективное использование памяти достигается в том случае, если количество битов кратно размеру long. С другой стороны, битовые поля не требуют лишних затрат на хранение служебной информации — все выглядит так, словно вы вручную реализовали работу с набором битов через массив long.

Несмотря на отсутствие других преобразований из bitset, помимо to_ulong(), имеется потоковый итератор вывода, который выдает строку из единиц и нулей. Длина строки может достигать длины битового поля.

Примитивы для задания двоичных значений по-прежнему отсутствуют, но зато bitset поддерживает почти такой же удобный формат — объекты string из единиц и нулей, в которых младший (наименее значимый) бит расположен справа. Три конструктора, продемонстрированных выше, получают соответственно всю строку, подстроку, начинающуюся со второго символа, и подстроку из символов 2–11. Содержимое bitset можно вывести в ostream оператором <<, и вы получите желаемое двоичное представление из 0 и 1. Также возможно чтение двоичных данных из istream оператором >> (не показано).

Битовые поля поддерживают всего три внешних оператора: конъюнкции (&), дизъюнкции (|) и исключающей дизъюнкции (^). Каждый из этих операторов создает новое битовое поле для своего возвращаемого значения. Все операторы класса работают в более эффективных формах &=, |= и т. д., не создающих временные объекты. С другой стороны, эти формы изменяют значение bitset (объект a в большинстве тестов данного примера). Чтобы предотвратить нежелательную модификацию объекта, мы создаем временный левосторонний объект, конструируя копию a; этим объясняется присутствие формы BS(a). Программа выводит результаты каждого теста и периодически повторяет вывод a, чтобы вам было удобнее изучать результаты.

В остальном пример не требует комментариев. Если что-то останется непонятным, обращайтесь за подробностями к описанию компилятора или другой документации, упоминавшейся ранее.

Контейнер vector<bool>

Контейнер vector<bool> является специализацией шаблона vector. Обычная переменная типа bool занимает минимум 1 байт, но если она может находиться только в двух состояниях, в идеальной реализации vector<bool> каждое значение представляется всего одним битом. Поскольку в типичной реализации библиотеки биты упаковываются в целочисленные массивы, итератор приходится определять специальным образом; он не может быть простым указателем на bool.

Битовые операции vector<bool> существенно ограничены по сравнению с bitset. К основным операциям вектора добавляется единственная функция flip(), инвер-

тирующая все биты. В отличие от **bitset**, контейнер **vector<bool>** не поддерживает функции установки-сброса отдельных битов **set()** и **reset()**. Оператор индексирования [ ] возвращает объект типа **vector<bool>::reference**, который также поддерживает функцию **flip()**, для инвертирования этого отдельного бита.

```
//: C07:VectorOfBool.cpp
// Специализация vector<bool>
#include <bitset>
#include <cstddef>
#include <iostream>
#include <iterator>
#include <sstream>
#include <vector>
using namespace std;

int main() {
  vector<bool> vb(10, true);
  vector<bool>::iterator it;
  for(it = vb.begin(); it != vb.end(); it++)
    cout << *it;
  cout << endl;
  vb.push_back(false);
  ostream_iterator<bool> out(cout, "");
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  bool ab[] = { true, false, false, true, true,
    true, true, false, false, true };
  // Существует похожий конструктор:
  vb.assign(ab, ab + sizeof(ab)/sizeof(bool));
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  vb.flip(); // Инвертирование всех битов
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  for(size_t i = 0; i < vb.size(); i++)
    vb[i] = 0; // (эквивалент "false")
  vb[4] = true;
  vb[5] = 1;
  vb[7].flip(); // Инвертирование одного бита
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  // Преобразование в bitset:
  ostringstream os;
  copy(vb.begin(), vb.end(),
    ostream_iterator<bool>(os, ""));
  bitset<10> bs(os.str());
  cout << "Bitset:\n" << bs << endl;
} ///:~
```

Последний фрагмент этого примера преобразует **vector<bool>** в **bitset** через промежуточную строку из единиц и нулей. Чтобы такое преобразование стало возможно, размер **bitset** должен быть известен на стадии компиляции, поэтому такое преобразование на практике применяется относительно редко.

Специализация **vector<bool>** в определенном смысле является «неполноценным» контейнером STL — она не дает некоторых гарантий, предоставляемых другими контейнерами. Например, для других контейнеров выполняются следующие отношения:

```
// Если c - контейнер STL, отличный от vector<bool>:
T& r = c.front();
T* p = &*c.begin();
```

Для всех остальных контейнеров функция front() возвращает левосторонний объект (нечто, для чего можно получить неконстантную ссылку), а функция begin() возвращает объект, который можно разыменовать, а потом получить его адрес. Для vector<bool> и то, и другое невозможно, потому что отдельные биты адресоваться не могут. Оба контейнера (vector<bool> и bitset) используют промежуточный класс (вложенный класс reference, упоминавшийся ранее) для чтения и записи отдельных битов.

Ассоциативные контейнеры

Множество (set), отображение (map), мультимножество (multiset) и мультиотображение (multimap) относятся к категории *ассоциативных контейнеров*, потому что их элементы представляют собой ассоциированные пары «ключ–значение». Точнее говоря, ключи ассоциируются со значениями в отображениях и мультиотображениях, однако множество можно рассматривать как отображение, содержащее только ключи без значений (его даже можно реализовать подобным образом). Сказанное относится и к связи между мультимножествами и мультиотображениями. Из-за этого структурного сходства множества и мультимножества также причислены к категории ассоциативных контейнеров.

Важнейшие операции с ассоциативными контейнерами — занесение новых элементов, а для множеств — проверка наличия элементов в контейнере. При работе с отображением нужно сначала проверить, присутствует ли ключ в контейнере, и если присутствует, получить значение, ассоциированное с этим ключом. Впрочем, это лишь фундаментальный принцип. Основные операции с ассоциативными контейнерами продемонстрированы в следующем примере:

```
//: C07:AssociativeBasics.cpp {-bor}
// Основные операции с множествами и отображениями
#include <cstddef>
#include <iostream>
#include <iterator>
#include <map>
#include <set>
#include "Noisy.h"
using namespace std;

int main() {
  Noisy na[7];
  // Добавление элементов в конструкторе :
  set<Noisy> ns(na, na + sizeof na/sizeof(Noisy));
  // Обычная вставка:
  Noisy n;
  ns.insert(n);
  cout << endl;
  // Проверка наличия элемента:
  cout << "ns.count(n)= " << ns.count(n) << endl;
  if(ns.find(n) != ns.end())
    cout << "n(" << n << ") found in ns" << endl;
  // Вывод элементов:
```

```
  copy(ns.begin(), ns.end(),
    ostream_iterator<Noisy>(cout, " "));
  cout << endl;
  cout << "\n-----------\n";
  map<int, Noisy> nm;
  for(int i = 0; i < 10; i++)
    nm[i]; // Пары создаются автоматически
  cout << "\n-----------\n";
  for(size_t j = 0; j < nm.size(); j++)
    cout << "nm[" << j <<"] = " << nm[j] << endl;
  cout << "\n-----------\n";
  nm[10] = n;
  cout << "\n-----------\n";
  nm.insert(make_pair(47, n));
  cout << "\n-----------\n";
  cout << "\n nm.count(10)= "
    << nm.count(10) << endl;
  cout << "nm.count(11)= "
    << nm.count(11) << endl;
  map<int, Noisy>::iterator it = nm.find(6);
  if(it != nm.end())
    cout << "value:" << (*it).second
      << " found in nm at location 6" << endl;
  for(it = nm.begin(); it != nm.end(); it++)
    cout << (*it).first << ":"
      << (*it).second << ", ";
  cout << "\n-----------\n";
} ///:~
```

При создании объекта ns типа set<Noisy> используются два итератора для массива объектов Noisy. Но у множеств также имеется конструктор по умолчанию и копирующий конструктор, причем конструктору можно передать объект, обеспечивающий альтернативную схему сравнения элементов. У множеств и отображений имеется функция insert(), предназначенная для занесения элементов в контейнер, а проверка принадлежности объекта к ассоциативному контейнеру может осуществляться двумя способами. Функция count() сообщает, сколько значений с заданным ключом присутствует в контейнере (для отображений и множеств она возвращает 0 или 1, но мультиотображения и мультимножества могут содержать несколько элементов с одинаковыми ключами). Функция find() возвращает итератор, установленный в позицию первого (а для отображений и множеств — *единственного*) вхождения заданного ключа. Если ключ отсутствует в контейнере, возвращается конечный итератор. Функции count() и find() поддерживаются всеми ассоциативными контейнерами, и это вполне логично. У ассоциативных контейнеров также имеются функции lower_bound(), upper_bound() и equal_range(); как будет показано далее, они имеют смысл только для мультимножеств и мультиотображений. Не пытайтесь изобрести какое-нибудь полезное применение этих функций для множеств и отображений — они предполагают существование интервалов одинаковых ключей, которые невозможны в этих контейнерах.

Разработка оператора индексирования [ ] всегда связана с выбором. Поскольку этот оператор работает так же, как при индексировании массивов, программисты обычно не проверяют индекс перед использованием. Но что произойдет, если индекс выходит за границы? Для массива можно запустить исключение, но для отображений «нарушение границ» может означать, что вы хотите создать новый

элемент с указанным ключом. Во всяком случае, контейнер map библиотеки STL воспринимает происходящее именно так. Первый цикл for после создания объекта map<int,Noisy> nm вроде бы «ищет объекты» оператором [ ], но на самом деле он создает новые объекты Noisy! При обращении к отсутствующему элементу отображения оператором [ ] контейнер автоматически создает новую пару «ключ–значение» (для значения используется конструктор по умолчанию). Следовательно, если вы хотите действительно проверить наличие элемента без его автоматического создания, следует задействовать функцию count() (простая проверка) или find() (получение итератора).

Вывод значений контейнера с помощью оператора [ ] в цикле for порождает сразу несколько проблем. Прежде всего ключи должны иметь целочисленные значения (это условие выполняется в нашем примере). Вторая (и более серьезная проблема) состоит в том, что если ключи не образуют непрерывную последовательность, при переборе от нуля до размера контейнера будут автоматически созданы пары для отсутствующих ключей, а более высокие значения ключей окажутся упущенными. Наконец, если взглянуть на трассировку цикла for, вы увидите, что при выводе выполняется *очень много* операций. На первый взгляд совершенно непонятно, почему простой перебор сопровождается таким количеством вызовов конструкторов и деструкторов. Ситуация проясняется только при просмотре кода операторной функции operator[ ] в шаблоне map, который выглядит примерно так:

```
mapped_type& operator[] (const key_type& k) {
  value_type tmp(k,T());
  return (*((insert(tmp)).first)).second;
}
```

Функция map::insert() получает пару «ключ–значение». Если в отображении уже имеется элемент с заданным ключом, ничего не происходит, иначе создается элемент. В любом случае функция возвращает новую пару из итератора вставленного элемента (то есть пары «ключ–значение») и логического флага (true, если элемент был вставлен в отображение). Стоит напомнить, что запись map::value_type в действительности представляет собой простое определение типа для шаблона std::pair:

```
typedef pair<const Key, T> value_type;
```

Шаблон std::pair уже встречался нам раньше. Как видно из определения, он предназначен для простого хранения значений двух независимых типов:

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair();
  pair(const T1& x, const T2& y)
  . : first(x), second(y) {}
  // Параметризованный копирующий конструктор:
  template<class U, class V>
   pair(const pair<U, V> &p);
};
```

Шаблон pair очень удобен. Он особенно часто применяется, когда функция должна вернуть два объекта (команда return может получать только один объект).

Для создания пар даже существует специальная сокращенная запись make_pair(), использованная в программе AssociativeBasics.cpp.

Итак, тип map::value_type представляет собой пару ключ и значение (то есть один элемент отображения). Но компоненты pair хранятся по значению, то есть для занесения объектов в pair необходим вызов копирующего конструктора. Таким образом, создание объекта tmp в операторной функции map::operator[ ] потребует по крайней мере вызова копирующего конструктора и деструктора для каждого компонента pair. В нашем случае это терпимо, поскольку ключ является целым числом. Но если вы хотите увидеть, что может произойти при вызове операторной функции map::operator[ ], попробуйте запустить следующую программу:

```
//: C07:NoisyMap.cpp
// Отображение Noisy в Noisy
//{L} Noisy
#include <map>
#include "Noisy.h"
using namespace std;

int main() {
  map<Noisy, Noisy> mnn;
  Noisy n1, n2;
  cout << "\n--------\n";
  mnn[n1] = n2;
  cout << "\n--------\n";
  cout << mnn[n1] << endl;
  cout << "\n--------\n";
} ///:~
```

Пример показывает, что вставка и поиск порождают множество лишних объектов, и все это происходит из-за создания объекта tmp. Если вернуться к функции map::operator[ ], вы увидите, что вторая строка вызывает функцию insert() и передает ей объект tmp. Возвращаемое значение функции insert() представляет собой пару, в которой переменная first содержит итератор для вставленной пары «ключ–значение», а переменная second — логический флаг, указывающий, была ли выполнена вставка. Оператор [ ] берет переменную first (итератор), разыменовывает ее для получения объекта pair и возвращает значение переменной second полученной пары (значение элемента).

Итак, к достоинствам отображений следует отнести автоматическое создание несуществующих элементов, а к недостаткам — многочисленные лишние операции создания и уничтожения объектов при *любом* использовании операторной функции map::operator[ ]. Впрочем, пример AssociativeBasics.cpp также показывает, как избежать лишних затрат — достаточно обойтись без оператора [ ] там, где это возможно (функция insert() работает более эффективно). В множествах хранятся отдельные объекты, а в отображениях — пары «ключ–значение», поэтому в аргументе insert() должен передаваться объект pair. Для его создания удобно воспользоваться функцией make_pair(), как это сделано в нашем примере.

При поиске объектов в отображении можно воспользоваться функцией count(), чтобы узнать о присутствии или отсутствии ключа, или функцией find() для получения итератора; разыменование этого итератора дает пару «ключ–значение». Чтобы получить доступ к ее компонентам, выберите переменные first и second. Если запустить программу AssociativeBasics.cpp, вы увидите, что вариант с итератором

обходится без создания и уничтожения дополнительных объектов. С другой стороны, он труднее пишется и читается.

Заполнение ассоциативных контейнеров данными

Вы уже видели, как удобны шаблоны fill(), fill_n(), generate() и generate_n(), из заголовка <algorithm> для заполнения данными последовательных контейнеров (векторов, списков и деков). Однако эти алгоритмы присваивают значения элементам последовательных контейнеров оператором =, тогда как добавление элементов в ассоциативные контейнеры производится соответствующими функциями insert(). Следовательно, используемый по умолчанию механизм присваивания порождает проблемы при попытке применить эти алгоритмы к ассоциативным контейнерам.

Первое очевидное решение — продублировать алгоритмы fill/generate и создать новые алгоритмы, предназначенные для ассоциативных контейнеров. Впрочем, дублируются только алгоритмы fill_n() и generate_n() (алгоритмы fill() и generate() копируют интервалы, а для ассоциативных контейнеров это бессмысленно), но задача решается просто — за образец можно взять заголовочный файл <algorithm>:

```
//: C07:assocGen.h
// Аналоги fill_n() и generate_n() для
// ассоциативных контейнеров.
#ifndef ASSOCGEN_H
#define ASSOCGEN_H

template<class Assoc, class Count, class T>
void assocFill_n(Assoc& a, Count n, const T& val) {
  while(n-- > 0)
    a.insert(val);
}

template<class Assoc, class Count, class Gen>
void assocGen_n(Assoc& a, Count n, Gen g) {
  while(n-- > 0)
    a.insert(g());
}
#endif // ASSOCGEN_H ///:~
```

Вместо итераторов мы передаем сам класс контейнера (разумеется, по ссылке).

Из этого примера следуют два важных урока. Первый урок: если алгоритм не делает того, что вам нужно, скопируйте наиболее близкий по смыслу образец и внесите исправления. Заголовочные файлы STL находятся под рукой, так что большая часть работы за вас уже сделана.

Второй урок более специфичен: если хорошенько подумать, обычно удается решить задачу средствами STL, не изобретая ничего нового. Например, наша задача может быть решена при помощи итератора вставки (создаваемого вызовом inserter()), который включает элементы в контейнер вызовом insert() вместо operator=. Это решение *не является* разновидностью итератора front_insert_iterator или back_insert_iterator, потому что эти итераторы используют функции push_front() и push_back() соответственно. Каждый итератор вставки характеризуется функцией, требуемой для вставки, и в нашем примере нужна именно функция insert(). Следующий пример демонстрирует заполнение данными отображений и множеств, хотя он будет работать и с мультиотображениями и мультимножествами. Начнем

с определения пары шаблонных генераторов (на первый взгляд это явный перебор, но такие шаблоны часто удобны, поэтому их определения лучше вынести в заголовочный файл).

```
//: C07:SimpleGenerators.h
// Обобщенные генераторы (включая генератор пар).
#include <iostream>
#include <utility>

// Генератор, увеличивающий свое текущее значение:
template<typename T> class IncrGen {
  T i;
public:
  IncrGen(T ii) : i (ii) {}
  T operator()() { return i++; }
};

// Генератор для создания объектов STL pair<>:
template<typename T1, typename T2>
class PairGen {
  T1 i;
  T2 j;
public:
  PairGen(T1 ii, T2 jj) : i(ii), j(jj) {}
  std::pair<T1,T2> operator()() {
    return std::pair<T1,T2>(i++, j++);
  }
};

namespace std {
// Обобщенный глобальный оператор <<
// для вывода произвольных объектов STL pair<>:
template<typename F, typename S> ostream&
operator<<(ostream& os, const pair<F,S>& p) {
  return os << p.first << "\t"
    << p.second << endl;
}
} ///:~
```

Оба генератора предполагают, что тип T поддерживает инкремент, и создают новые значения по исходным данным при помощи оператора ++. Шаблон **PairGen** возвращает объект pair библиотеки STL, который напрямую включается в отображение или мультиотображение функцией insert().

Последняя функция представляет собой обобщенную версию оператора << для ostream. Наш оператор позволяет вывести любой объект pair при условии, что каждый элемент пары поддерживает потоковый оператор << (из-за некоторых особенностей разрешения имен, о которых рассказывалось в главе 5, он должен находиться в пространстве std; мы вернемся к этой теме при описании программы Thesaurus.cpp позднее в этой главе). Как видно из следующего примера, это позволяет использовать алгоритм copy() для вывода элементов отображения:

```
//: C07:AssocInserter.cpp
// Итератор вставки делает возможным
// использование алгоритмов fill_n() и generate_n()
// с ассоциативными контейнерами.
#include <iterator>
#include <iostream>
```

```
#include <algorithm>
#include <set>
#include <map>
#include "SimpleGenerators.h"
using namespace std;

int main() {
  set<int> s;
  fill_n(inserter(s, s.begin()), 10, 47);
  generate_n(inserter(s, s.begin()), 10,
    IncrGen<int>(12));
  copy(s.begin(), s.end(),
    ostream_iterator<int>(cout, "\n"));
  map<int, int> m;
  fill_n(inserter(m, m.begin()), 10, make_pair(90,120));
  generate_n(inserter(m, m.begin()), 10,
    PairGen<int, int>(3, 9));
  copy(m.begin(), m.end(),
    ostream_iterator<pair<int,int> >(cout,"\n"));
} ///:~
```

Передача итератора во втором аргументе inserter помогает оптимизировать процесс вставки — итератор рекомендует, с какой позиции следует начинать поиск (вместо того, чтобы всегда начинать его с корня базового дерева). Однако итератор вставки может использоваться с разными типами контейнеров, и для неассоциативных контейнеров итератор обязателен.

Обратите внимание на создание итератора ostream_iterator для вывода объекта pair — без определения операторной функции operator<< он бы не сработал. Поскольку ostream_iterator является шаблоном, он автоматически специализируется для типа pair<int,int>.

Отображения

В обычных массивах элемент определяется целочисленным индексом, задающим его позицию в последовательности однотипных элементов. Отображение (контейнер map) является *ассоциативным массивом*; это означает, что он позволяет ассоциировать один объект с другим (по аналогии с тем, как индекс связывается с элементом в массиве). Но вместо простого индексирования выборка в отображениях осуществляется по объекту-ключу! В следующем примере подсчитывается число вхождений слов в текстовом файле. Таким образом, «индекс» (объект string) представляет слово, а искомое значение — объект со счетчиком числа вхождений.

В обычных контейнерах вроде векторов или списков элемент представляет собой самостоятельный объект данных. Но в отображениях один элемент состоит из двух компонентов: *ключа* (по которому осуществляется поиск в форме *отображение*[*ключ*]) и *значения*, ассоциированного с ключом. Чтобы перебрать все содержимое отображение и вывести каждую пару «ключ/значение», следует использовать итератор, разыменование которого дает пару (объект pair) с ключом и значением. Доступ к компонентам пары осуществляется через переменные first и second.

Аналогичный принцип упаковки двух элементов также используется при вставке элементов в отображения, но в этом случае объект pair создается в контексте конкретного отображения и называется value_type. Таким образом, в одном из

вариантов вставки нового элемента вы создаете объект value_type, присваиваете компонентам пары нужные объекты и вызываете функцию insert() контейнера map. Однако в следующем примере задействован другой способ, основанный на уже упоминавшейся особенности контейнера map: если ключ, переданный при поиске объекта оператору [ ], отсутствует в контейнере, то оператор [ ] автоматически вставляет пару «ключ–значение», создавая объект значения конструктором по умолчанию. С учетом сказанного рассмотрим следующую реализацию программы с подсчетом слов:

```
//: C07:WordCount.cpp
// Подсчет вхождений слов в файле
// с использованием отображения.
#include <iostream>
#include <fstream>
#include <map>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  typedef map<string, int> WordMap;
  typedef WordMap::iterator WMIter;
  const char* fname = "WordCount.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  WordMap wordmap;
  string word;
  while(in >> word)
    wordmap[word]++;
  for(WMIter w = wordmap.begin(); w != wordmap.end(); w++)
    cout << w->first << ": " << w->second << endl;
} ///:~
```

Заодно этот пример демонстрирует мощь *нулевой инициализации*. Обратим внимание на следующую строку программы:

```
wordmap[word]++;
```

Выражение увеличивает значение int, ассоциированное с word. При отсутствии ключа word в отображении автоматически создается новая пара, в которой значение инициализируется нулем, возвращаемым при вызове псевдоконструктора int().

Вывод всего списка требует перебора с применением итератора (чтобы использовать для контейнера map сокращенную запись с алгоритмом copy(), придется написать операторную функцию operator<< для объекта pair). Как уже отмечалось, разыменование этого итератора дает объект pair, в котором переменная first содержит ключ, а переменная second — значение.

Чтобы получить счетчик вхождений для конкретного слова, можно воспользоваться оператором индексирования:

```
cout << "the: " << wordmap["the"] << endl;
```

В этой записи проявляется одно из важнейших преимуществ отображений — наглядность синтаксиса; операции с ассоциативным массивом понятны читателю программы на уровне здравого смысла (заметьте: если слово «the» отсутствует в wordmap, для него автоматически создается новый элемент!)

Мультиотображения и дубликаты ключей

Мультиотображением (multimap) называется отображение, которое может содержать повторяющиеся ключи. На первый взгляд идея выглядит несколько странно, но на практике она встречается на удивление часто. Например, телефонная книга может содержать несколько записей с одинаковыми именами.

Допустим, вы наблюдаете за природой и ведете журнал с информацией о том, где и когда вам встретился тот или иной вид животных. При этом животные одного вида могут водиться в разных местах и в разное время. Следовательно, если в качестве ключа выбран вид животного, для хранения данных наблюдений нам потребуется мультиотображение. Программа выглядит примерно так:

```
//: C07:WildLifeMonitor.cpp
#include <algorithm>
#include <cstdlib>
#include <cstddef>
#include <ctime>
#include <iostream>
#include <iterator>
#include <map>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

class DataPoint {
  int x, y; // Координаты
  time_t time; // Время наблюдения
public:
  DataPoint() : x(0), y(0), time(0) {}
  DataPoint(int xx, int yy, time_t tm) :
    x(xx), y(yy), time(tm) {}
  // Сгенерированный оператор = и копирующий конструктор годятся
  int getX() const { return x; }
  int getY() const { return y; }
  const time_t* getTime() const { return &time; }
};

string animal[] = {
  "chipmunk", "beaver", "marmot", "weasel",
  "squirrel", "ptarmigan", "bear", "eagle",
  "hawk", "vole", "deer", "otter", "hummingbird",
};
const int ASZ = sizeof animal/sizeof *animal;
vector<string> animals(animal, animal + ASZ);

// Информация о наблюдении в объекте Sighting,
// содержимое которого можно направить в ostream:
typedef pair<string, DataPoint> Sighting;

ostream&
operator<<(ostream& os, const Sighting& s) {
  return os << s.first << " sighted at x= " <<
    s.second.getX() << ", y= " << s.second.getY()
    << ", time = " << ctime(s.second.getTime());
}

// Генератор объектов Sighting:
```

```
class SightingGen {
  vector<string>& animals;
  enum { D = 100 };
public:
  SightingGen(vector<string>& an) : animals(an) {}
  Sighting operator()() {
    Sighting result;
    int select = rand() % animals.size();
    result.first = animals[select];
    result.second = DataPoint(
      rand() % D, rand() % D, time(0));
    return result;
  }
};

// Функция выводит меню с названиями животных,
// предлагает пользователю выбрать один из пунктов
// и возвращает индекс:
int menu() {
  cout << "select an animal or 'q' to quit: ";
  for(int i = 0; i < animals.size(); i++)
    cout <<'['<< i <<']'<< animals[i] << ' ';
  cout << endl;
  string reply;
  cin >> reply;
  if(reply.at(0) == 'q') return 0;
  istringstream r(reply);
  int i;
  r >> i; // Преобразование в int
  i %= animals.size();
  return i;
}

int main() {
  typedef multimap<string, DataPoint> DataMap;
  typedef DataMap::iterator DMIter;
  DataMap sightings;
  srand(time(0)); // Раскрутка генератора случайных чисел
  generate_n(inserter(sightings, sightings.begin()),
    50, SightingGen(animals));
  // Вывод всех элементов:
  copy(sightings.begin(), sightings.end(),
    ostream_iterator<Sighting>(cout, ""));
  // Вывод данных по выбранному типу животных:
  for(int count = 1; count < 10; count++) {
    // Выбор типа через меню:
    // int i = menu();
    // Случайный выбор (для автоматизации тестирования):
    int i = rand() % animals.size();
    // Итераторы "range" определяют начальную и конечную позиции
    // интервала с искомым ключом:
    pair<DMIter, DMIter> range =
      sightings.equal_range(animals[i]);
    copy(range.first, range.second,
      ostream_iterator<Sighting>(cout, ""));
  }
} ///:~
```

Вся информация о наблюдении инкапсулируется в классе DataPoint. Этот класс достаточно прост, чтобы мы могли положиться на сгенерированный оператор при-

сваивания и конструктор по умолчанию. Для сохранения времени наблюдения используются стандартные библиотечные функции C.

При инициализации массива animal объектов string автоматически используется конструктор char*, что существенно упрощает процедуру инициализации. Так как названия животных удобнее хранить в векторе, мы вычисляем размер массива и инициализируем vector<string> конструктором vector с двумя итераторами.

Пары «ключ–значение», составляющие объекты Sighting, состоят из объекта string (название животного) и объекта DataPoint (место и время наблюдения). Эти два типа объединяются стандартным шаблоном pair, и для него создается определение типа Sighting. Затем для типа Sighting определяется операторная функция ostream& operator<<, чтобы мы могли перебрать элементы отображения/мультиотображения с элементами Sighting и вывести их.

Класс SightingGen генерирует тестовые данные со случайным временем и местом наблюдения. Он использует оператор (), необходимый для объекта функции, но также для него определен конструктор с сохранением ссылки на вектор vector<string> с названиями животных.

В мультиотображении DataMap хранятся пары string/DataPoint (то есть объекты Sighting). Мы заполняем DataMap 50 объектами Sighting при помощи алгоритма generate_n() и выводим его содержимое (существование оператора << для типа Sighting позволяет создать итератор ostream_iterator). Далее программа предлагает пользователю выбрать животное, для которого выводится информация о наблюдениях. Если нажать клавишу Q, программа завершается, а при выборе номера вызывается функция equal_range(). Она возвращает два итератора, определяющих начало и конец набора элементов с искомым значением ключа. Так как функция может возвращать только один объект, equal_range() возвращает свой результат в виде объекта pair. Полученные итераторы передаются алгоритму copy() для вывода информации обо всех наблюдениях животных заданного вида.

Мультимножества

Мы уже рассматривали множество (контейнер set), в котором каждое значение может храниться только в одном экземпляре. Мультимножество (контейнер multiset) может содержать несколько одинаковых значений. На первый взгляд это противоречит самой идее множества, к которому можно обратиться с вопросом: «Присутствует ли *такой-то* элемент во множестве?» Если *таких* элементов может быть несколько, как понимать вопрос?

Если немного подумать, становится ясно, что хранение нескольких *абсолютно одинаковых* объектов действительно не имеет особого смысла (исключение составляет разве что подсчет экземпляров объектов, но, как было показано в этой главе, подобная задача имеет другое, более изящное решение). Таким образом, у каждого дубликата должна быть некая особенность, которая отличает его от других дубликатов; как правило, это разная информация состояния, не используемая при вычислении ключа в процессе сравнения. Итак, с точки зрения операции сравнения объекты выглядят одинаково, но в действительности они обладают разным внутренним состоянием.

Как и любой контейнер STL с упорядочением элементов, шаблон multiset по умолчанию определяет порядок следования элементов при помощи объекта функ-

ции less. При этом используется оператор < класса элементов, но вы всегда можете предоставить собственный критерий сравнения.

Рассмотрим простой класс с двумя переменными; одна переменная задействована в сравнении, а другая — нет:

```
//: C07:MultiSet1.cpp
// Мультимножества
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <set>
using namespace std;

class X {
  char c; // Используется при сравнении
  int i; // Не используется при сравнении
  // Конструктор по умолчанию и оператор = не нужны
  X();
  X& operator=(const X&);
  // Обычно нужен копирующий конструктор (но в данном случае
  // подойдет и сгенерированная версия)
public:
  X(char cc, int ii) : c(cc), i(ii) {}
  // Обратите внимание: оператор == может отсутствовать
  friend bool operator<(const X& x, const X& y) {
    return x.c < y.c;
  }
  friend ostream& operator<<(ostream& os, X x) {
    return os << x.c << ":" << x.i;
  }
};

class Xgen {
  static int i;
  // Количество возможных символов:
  enum { span = 6 };
public:
  X operator()() {
    char c = 'A' + rand() % span;
    return X(c, i++);
  }
};

int Xgen::i = 0;

typedef multiset<X> Xmset;
typedef Xmset::const_iterator Xmit;

int main() {
  Xmset mset;
  // Заполнение объектами X:
  srand(time(0));   // Раскрутка генератора случайных чисел
  generate_n(inserter(mset, mset.begin()),
    25, Xgen());
  // Инициализация обычного множества на базе mset:
  set<X> unique(mset.begin(), mset.end());
  copy(unique.begin(), unique.end(),
```

```
    ostream_iterator<X>(cout, " "));
  cout << "\n----\n";
  // Перебор уникальных значений:
  for(set<X>::iterator i = unique.begin();
      i != unique.end(); i++) {
    pair<Xmit, Xmit> p = mset.equal_range(*i);
    copy(p.first, p.second,
      ostream_iterator<X>(cout, " "));
    cout << endl;
  }
} ///:~
```

В классе X объекты сравниваются по значению поля char c. Сравнение выполняется оператором <; для мультимножества этого достаточно, потому что в нашем примере по умолчанию используется критерий less. Класс Xgen генерирует случайные объекты X, при этом символы, по которым производится сравнение, лежат в интервале от «A» до «E». Функция main() создает контейнер multiset<X> и заполняет его 25 объектами X, используя Xgen. Контейнер заведомо содержит одинаковые ключи. Чтобы получить набор уникальных значений, мы создаем на базе мультимножества обычное множество set<X> (при этом задействован итератор с двумя конструкторами). Программа выводит эти значения, а затем для каждого из них вызывается функция equal_range() (в мультимножествах она делает то же, что в мультиотображениях: ищет элементы с одинаковыми ключами). В завершение программа выводит все наборы с одинаковыми ключами.

Рассмотрим другой пример — более элегантную реализацию программы WordCount.cpp на базе мультимножества:

```
//: C07:MultiSetWordCount.cpp
// Подсчет вхождений слов в файле
// с использованием мультимножества.
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "MultiSetWordCount.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  multiset<string> wordmset;
  string word;
  while(in >> word)
    wordmset.insert(word);
  typedef multiset<string>::iterator MSit;
  MSit it = wordmset.begin();
  while(it != wordmset.end()) {
    pair<MSit, MSit> p = wordmset.equal_range(*it);
    int count = distance(p.first, p.second);
    cout << *it << ": " << count << endl;
    it = p.second; // Переход к следующему слову
  }
} ///:~
```

Подготовительная фаза работы функции main() не отличается от таковой в программе WordCount.cpp, но затем каждое слово просто вставляется в контейнер multiset<string>. Мы создаем итератор и инициализируем его началом мультимножества; разыменование этого итератора дает текущее слово. Функция equal_range() (не обобщенный алгоритм!) возвращает начальный и конечный итераторы для текущего слова, а алгоритм distance() (определяемый в заголовке <iterator>) подсчитывает количество элементов в этом интервале. Итератор it перемещается в конец интервала, то есть к следующему слову. Если вы недостаточно хорошо разбираетесь в специфике мультимножеств, эта программа может показаться более сложной. С другой стороны, ее компактность и отсутствие вспомогательных классов вроде Count являются несомненными достоинствами.

Насколько удачен выбор названия «мультимножество»? Может, стоило назвать контейнер как-нибудь иначе? В некоторых библиотеках контейнеров присутствует контейнер bag, в котором могут храниться любые объекты, в том числе и одинаковые. Он близок к мультимножествам STL, но для контейнера bag не задан способ упорядочения элементов. Мультимножество, требующее, чтобы все дубликаты находились в смежных позициях, на концептуальном уровне устанавливает еще более жесткие ограничения, чем множество. В реализации множества может применяться хеширование, в результате чего элементы не будут располагаться в порядке сортировки. Наконец, если вам потребуется просто хранить набор объектов без специальных критериев упорядочения, обычно бывает удобнее воспользоваться вектором, деком или списком.

Объединение контейнеров STL

Тезаурус содержит все слова, связанные с некоторым словом. Ключом здесь является одно слово, а значением — список слов. Для реализации подобного механизма хранения мультиконтейнеры (multimap и multiset) не подходят. Чтобы добиться нужного результата, нужно создать комбинацию контейнеров; к счастью, в STL это делается просто. Конкретное средство для решения задачи — отображение, в котором строки ассоциируются с векторами — подводит нас к мощной общей концепции объединения контейнеров.

```
//: C07:Thesaurus.cpp
// Отображение со значениями-векторами
#include <map>
#include <vector>
#include <string>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <ctime>
#include <cstdlib>
using namespace std;

typedef map<string, vector<string> > Thesaurus;
typedef pair<string, vector<string> > TEntry;
typedef Thesaurus::iterator TIter;

// Для правильного разрешения имен:
namespace std {
```

```
ostream& operator<<(ostream& os,const TEntry& t){
  os << t.first << ": ";
  copy(t.second.begin(), t.second.end(),
    ostream_iterator<string>(os, " "));
  return os;
}
}

// Генератор тестовых данных для тезауруса:
class ThesaurusGen {
  static const string letters;
  static int count;
public:
  int maxSize() { return letters.size(); }
  TEntry operator()() {
    TEntry result;
    if(count >= maxSize()) count = 0;
    result.first = letters[count++];
    int entries = (rand() % 5) + 2;
    for(int i = 0; i < entries; i++) {
      int choice = rand() % maxSize();
      char cbuf[2] = { 0 };
      cbuf[0] = letters[choice];
      result.second.push_back(cbuf);
    }
    return result;
  }
};

int ThesaurusGen::count = 0;
const string ThesaurusGen::letters("ABCDEFGHIJKL"
  "MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz");

// Запрос искомого "слова":
string menu(Thesaurus& thesaurus) {
  while(true) {
    cout << "Select a \"word\", 0 to quit: ";
    for(TIter it = thesaurus.begin();
      it != thesaurus.end(); it++)
      cout << (*it).first << ' ';
    cout << endl;
    string reply;
    cin >> reply;
    if(reply.at(0) == '0') exit(0); // Выход
    if(thesaurus.find(reply) == thesaurus.end())
      continue; // Слово отсутствует в списке, повторить попытку
    return reply;
  }
}

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  Thesaurus thesaurus;
  // Заполнение тезауруса 10 объектами:
  generate_n(
    inserter(thesaurus, thesaurus.begin()),
    10, ThesaurusGen());
  // Вывод всего содержимого:
  copy(thesaurus.begin(), thesaurus.end(),
```

```
    ostream_iterator<TEntry>(cout, "\n"));
  // Построение списка ключей:
  string keys[10];
  int i = 0;
  for(TIter it = thesaurus.begin();
    it != thesaurus.end(); it++)
    keys[i++] = (*it).first;
  for(int count = 0; count < 10; count++) {
    // Ввод с консоли:
    // string reply = menu(thesaurus);
    // Случайные данные:
    string reply = keys[rand() % 10];
    vector<string>& v = thesaurus[reply];
    copy(v.begin(), v.end(),
      ostream_iterator<string>(cout, " "));
    cout << endl;
  }
} ///:~
```

Контейнер Thesaurus связывает объект string (слово) с объектом vector<string> (синонимы). Класс TEntry представляет одну запись тезауруса. Создание операторной функции ostream& operator<< для TEntry позволяет легко выводить отдельные записи тезауруса (а также вывести все его содержимое алгоритмом copy()). Обратите внимание на очень странное размещение потокового итератора вставки: мы включаем его в пространство имен std[1]! При первом вызове copy() в main() шаблон ostream_iterator использует функцию operator<<. Когда компилятор создает необходимую специализацию ostream_iterator, по правилам поиска с учетом аргументов (ADL) он проводит поиск только в пространстве std, в котором объявлены все аргументы copy(). Если бы итератор объявлялся в глобальном пространстве имен (для чего достаточно удалить окружающий блок namespace), он не был бы найден. Размещение итератора в std позволит механизму ADL найти его.

Класс ThesaurusGen создает «слова» (в нашем примере это просто буквы) и «синонимы» (другие случайно сгенерированные буквы) для заполнения тезауруса. Количество синонимов выбирается случайно, но не может быть меньше двух. Буквы выбираются индексированием статической строки, которая является частью ThesaurusGen.

Функция main() создает контейнер, заполняет его 10 объектами и выводит алгоритмом copy(). Функция menu() предлагает пользователю выбрать искомое «слово» (то есть ввести соответствующую букву). Функция find() проверяет, присутствует ли заданный элемент в отображении (вспомните: мы не используем оператор [ ], потому что он автоматически создает новый элемент при отсутствии совпадения!). Если элемент присутствует, мы получаем вектор vector<string> оператором [ ] и выводим его содержимое. Строка reply генерируется случайным образом для автоматизации тестирования.

Поскольку шаблоны упрощают выражение многих сложных концепций, можно пойти еще дальше и создать отображение векторов, элементами которых являются отображения, и т. д. В таких комбинациях могут использоваться любые контейнеры STL.

---

[1] Строго говоря, пользователи не должны расширять стандартное пространство имен, но это самый простой способ решения всех проблем разрешения имен. К тому же он поддерживается всеми известными компиляторами.

Освобождение контейнеров указателей

В программе Stlshape.cpp уничтожение контейнера не приводило к автоматическому освобождению указателей. Конечно, нам хотелось бы упростить эту процедуру, чтобы ее не приходилось каждый раз программировать заново. Следующая шаблонная функция освобождает указатели в любом последовательном контейнере. Учтите, что в архиве примеров она находится в корневом каталоге (для упрощения доступа):

```
//: :purge.h
// Освобождение указателей в последовательных контейнерах STLcontainer
#ifndef PURGE_H
#define PURGE_H
#include <algorithm>

template<class Seq> void purge(Seq& c) {
  typename Seq::iterator i;
  for(i = c.begin(); i != c.end(); ++i) {
    delete *i;
    *i = 0;
  }
}

// Версия с итераторами:
template<class InpIt>
void purge(InpIt begin, InpIt end) {
  while(begin != end) {
    delete *begin;
    *begin = 0;
    ++begin;
  }
}
#endif // PURGE_H ///:~
```

В первой версии purge() ключевое слово typename абсолютно необходимо. Перед нами та самая ситуация, для которой предназначалось это ключевое слово: Seq — аргумент шаблона, а iterator — нечто вложенное в этот шаблон. Что же тогда должна обозначать запись Seq::iterator? Ключевое слово typename указывает, что она обозначает тип и ничего больше.

Хотя первая версия purge() ориентирована на контейнеры в стиле STL, вторая версия (с итераторами) работает с любыми интервалами, в том числе с массивами.

Далее приведен новый вариант Stlshape.cpp с использованием функции purge():

```
//: C07:Stlshape2.cpp
// Stlshape.cpp с использованием функции purge()
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};

class Circle : public Shape {
```

```
public:
  void draw() { cout << "Circle::draw\n"; }
  ~Circle() { cout << "~Circle\n"; }
};

class Triangle : public Shape {
public:
  void draw() { cout << "Triangle::draw\n"; }
  ~Triangle() { cout << "~Triangle\n"; }
};

class Square : public Shape {
public:
  void draw() { cout << "Square::draw\n"; }
  ~Square() { cout << "~Square\n"; }
};

int main() {
  typedef std::vector<Shape*> Container;
  typedef Container::iterator Iter;
  Container shapes;
  shapes.push_back(new Circle);
  shapes.push_back(new Square);
  shapes.push_back(new Triangle);
  for(Iter i = shapes.begin();
      i != shapes.end(); i++)
    (*i)->draw();
  purge(shapes);
} ///:~
```

При использовании функции purge() необходимо тщательно следить за принадлежностью объектов. Если указатель на объект хранится в двух контейнерах, нужно позаботиться о том, чтобы он не был освобожден дважды; кроме того, нельзя уничтожать объект из первого контейнера до того, как второй контейнер закончит работу с ним. С повторным освобождением указателей в одном контейнере проблем не будет, потому что purge() обнуляет указатель после его освобождения, а вызов delete для нулевого указателя является безопасной операцией.

Создание пользовательских контейнеров

Взяв STL за основу, вы можете создавать собственные контейнеры. Если вы соблюдаете все требования по поддержке итераторов, то новый контейнер будет работать так же, как стандартные контейнеры STL.

Для примера возьмем кольцевую структуру данных, то есть циклический последовательный контейнер. При достижении последнего элемента перебор просто возвращается к началу. Реализация кольца на базе списка может выглядеть так:

```
//: C07:Ring.cpp
// Построение кольцевой структуры данных на базе STL
#include <iostream>
#include <iterator>
#include <list>
#include <string>
using namespace std;

template<class T> class Ring {
  list<T> lst;
```

```
public:
  // Объявление класса необходимо для того, чтобы
  // следующее объявление 'friend' видело итератор 'iterator'
  // вместо std::iterator:
  class iterator;
  friend class iterator;
  class iterator : public std::iterator<
    std::bidirectional_iterator_tag,T,ptrdiff_t>{
    typename list<T>::iterator it;
    list<T>* r;
  public:
    iterator(list<T>& lst,
      const typename list<T>::iterator& i)
      : r(&lst), it(i) {}
    bool operator==(const iterator& x) const {
      return it == x.it;
    }
    bool operator!=(const iterator& x) const {
      return !(*this == x);
    }
    typename list<T>::reference operator*() const {
      return *it;
    }
    iterator& operator++() {
      ++it;
      if(it == r->end())
        it = r->begin();
      return *this;
    }
    iterator operator++(int) {
      iterator tmp = *this;
      ++*this;
      return tmp;
    }
    iterator& operator--() {
      if(it == r->begin())
        it = r->end();
      --it;
      return *this;
    }
    iterator operator--(int) {
      iterator tmp = *this;
      --*this;
      return tmp;
    }
    iterator insert(const T& x){
      return iterator(*r, r->insert(it, x));
    }
    iterator erase() {
      return iterator(*r, r->erase(it));
    }
  };
  void push_back(const T& x) { lst.push_back(x); }
  iterator begin() { return iterator(lst, lst.begin()); }
  int size() { return lst.size(); }
};

int main() {
  Ring<string> rs;
```

```
  rs.push_back("one");
  rs.push_back("two");
  rs.push_back("three");
  rs.push_back("four");
  rs.push_back("five");
  Ring<string>::iterator it = rs.begin();
  ++it; ++it;
  it.insert("six");
  it = rs.begin();
  // Двукратный перебор элементов кольца:
  for(int i = 0; i < rs.size() * 2; i++)
    cout << *it++ << endl;
} ///:~
```

Как видите, основной код сосредоточен в классе итератора. Итератор Ring должен уметь возвращаться к началу контейнера, поэтому в нем хранится ссылка на список своего «родительского» объекта Ring. По этой ссылке итератор узнает, добрался ли он до конца контейнера и как вернуться к началу.

Интерфейс Ring весьма ограничен. В частности, в нем отсутствует функция end(), поскольку при достижении конца происходит переход к началу. А это означает, что Ring не может использоваться с многочисленными алгоритмами STL, для которых необходим конечный итератор (добавление этой возможности оказывается делом нетривиальным). А если это ограничение вас огорчает, вспомните стек, очередь и приоритетную очередь — эти контейнеры вообще не используют итераторов!

Расширения STL

Хотя контейнеры STL обладают практически всеми функциональными возможностями, которые вам могут понадобиться, их набор все же нельзя считать абсолютно универсальным. Например, в стандартных реализациях множеств и отображений используются деревья. Такие реализации работают достаточно быстро, но в какой-то конкретной ситуации может потребоваться еще более высокая скорость. Участники комитета по стандартизации C++ сошлись на том, что в стандартный язык C++ следует включить поддержку хешированных реализаций set и map, но времени на проработку этих компонентов уже не оставалось, и они так и остались нестандартизованными[1].

К счастью, существует целый ряд свободно распространяемых альтернативных решений. Одно из преимуществ STL состоит в том, что эта библиотека устанавливает базовую модель для создания «STL-подобных» классов, и тот, кто уже знаком с STL, сможет легко разобраться в других решениях, основанных на этой модели.

Реализация библиотеки STL от Silicon Graphics[2], которая называется SGI, принадлежит к числу самых мощных альтернативных реализаций. При желании ею можно заменить реализацию STL вашего компилятора. Кроме того, SGI содержит ряд дополнительных контейнеров, включая хешированное множество (hash_set), хешированное мультимножество (hash_multiset), хешированное отображение

[1] Скорее всего, они появятся в следующей версии стандарта C++.

[2] См. http://www.sgi.com/tech/stl.

(hash_map), хешированное мультиотображение (hash_multimap), односвязный список (slist) и rope (разновидность класса string, оптимизированная для работы с очень большими строками и для быстрых операций конкатенации и выделения подстрок).

Давайте сравним относительное быстродействие реализации map на базе дерева и хешированной реализации hash_map из библиотеки SGI. Для простоты ограничимся отображением int/int:

```
//: C07:MapVsHashMap.cpp
// Заголовочный файл hash_не входит в стандартную
// реализацию STL для C++. Это расширение,
// доступное только в SGI STL
// (включается в поставку dmc).
//{-bor}{-msc}{-g++}{-mwcc}
#include <hash_map>
#include <iostream>
#include <map>
#include <ctime>
using namespace std;

int main(){
  hash_map<int, int> hm;
  map<int, int> m;
  clock_t ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m.insert(make_pair(j,j));
  cout << "map insertions: " << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm.insert(make_pair(j,j));
  cout << "hash_map insertions: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m[j];
  cout << "map::operator[] lookups: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm[j];
  cout << "hash_map::operator[] lookups: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m.find(j);
  cout << "map::find() lookups: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm.find(j);
  cout << "hash_map::find() lookups: "
    << clock() - ticks << endl;
} ///:~
```

Согласно результатам хронометража, hash_map во всех операциях превосходит map по скорости примерно в 4 раза (причем, как и ожидалось, функция find() при поиске в обоих типах отображений работает чуть быстрее оператора [ ]). Если профайлер показывает, что операции с отображением являются «узким местом» вашего приложения, подумайте о переходе на hash_map.

Другие контейнеры

В стандартную библиотеку входят два контейнера, которые не являются «контейнерами STL»: bitset и valarray[1]. Иначе говоря, эти контейнеры не удовлетворяют всем требованиям контейнеров STL. Контейнер bitset, описанный ранее в этой главе, пакует биты в целые числа и не поддерживает прямой адресации элементов. Шаблон valarray представляет собой аналог контейнера vector, оптимизированный для эффективных математических вычислений. Ни один из этих контейнеров не поддерживает итераторов. Хотя valarray может специализироваться для нечисловых типов, этот шаблон содержит математические функции, ориентированные на работу с числовыми данными: sin, cos, tan и т. д.

Определим вспомогательный шаблон для вывода элементов valarray:

```
//: C07:PrintValarray.h
#ifndef PRINTVALARRAY_H
#define PRINTVALARRAY_H
#include <valarray>
#include <iostream>
#include <cstddef>

template<class T>
void print(const char* lbl, const valarray<T>& a) {
  std::cout << lbl << ": ";
  for(size_t i = 0; i < a.size(); ++i)
    std::cout << a[i] << ' ';
  std::cout << std::endl;
}
#endif // PRINTVALARRAY_H ///:~
```

Большинство функций и операторов valarray ориентировано на работу не с отдельными элементами, а с массивом valarray в целом, как показывает следующий пример:

```
//: C07:Valarray1.cpp {-bor}
// Базовые возможности valarray
#include "PrintValarray.h"
using namespace std;

double f(double x) { return 2.0 * x - 1.0; }

int main() {
  double n[] = {1.0, 2.0, 3.0, 4.0};
  valarray<double> v(n, sizeof n / sizeof n[0]);
  print("v", v);
  valarray<double> sh(v.shift(1));
  print("shift 1", sh);
```

[1] Как уже отмечалось, специализация vector<bool> тоже в определенной степени не является контейнером STL.

```
  valarray<double> acc(v + sh);
  print("sum", acc);
  valarray<double> trig(sin(v) + cos(acc));
  print("trig", trig);
  valarray<double> p(pow(v, 3.0));
  print("3rd power", p);
  valarray<double> app(v.apply(f));
  print("f(v)", app);
  valarray<bool> eq(v == app);
  print("v == app?", eq);
  double x = v.min();
  double y = v.max();
  double z = v.sum();
  cout << "x = " << x << ", y = " << y
    << ", z = " << z  << endl;
} ///:~
```

В классе valarray определен конструктор, которому передается массив базового типа и количество элементов массива, требуемых для инициализации valarray. Функция shift() сдвигает все элементы valarray на одну позицию влево (или вправо при отрицательном значении аргумента) и заполняет освободившиеся места значением по умолчанию для базового типа (0 в данном случае). Также имеется функция cshift() для выполнения циклического сдвига. Все математические операторы и функции перегружены для работы с объектами valarrray, а бинарные операторы требуют, чтобы аргументы valarray имели одинаковые базовые тип и размер. Функция apply() по аналогии с алгоритмом transform() применяет функцию к каждому элементу, но результаты собираются в итоговый объект valarray. Операторы сравнения возвращают объекты valarray<bool> соответствующего размера с результатами поэлементных сравнений (см. eq в предыдущем примере). Большинство операций возвращает новый объект valarray, но некоторые операции (такие как min(), max() и sum()) по очевидным причинам возвращают скалярную величину.

Однако самая необычная операция с valarray — выделение подмножеств элементов, причем не только для получения информации, но и для ее обновления. Подмножество элементов valarray называется *срезом*. Срезы используются некоторыми операторами. Следующий пример демонстрирует работу со срезами:

```
//: C07:Valarray2.cpp {-bor}{-dmc}
// Срезы и маски
#include "PrintValarray.h"
using namespace std;

int main() {
  int data[] = {1,2,3,4,5,6,7,8,9,10,11,12};
  valarray<int> v(data, 12);
  valarray<int> r1(v[slice(0, 4, 3)]);
  print("slice(0,4,3)", r1);
  // Выделение элементов по условию
  valarray<int> r2(v[v > 6]);
  print("elements > 6", r2);
  // Возведение в квадрат первого столбца
  v[slice(0, 4, 3)] *= valarray<int>(v[slice(0, 4, 3)]);
  print("after squaring first column", v);
  // Восстановление исходных значений
  int idx[] = {1,4,7,10};
  valarray<int> save(idx, 4);
  v[slice(0, 4, 3)] = save;
```

```
    print("v restored", v);
    // Выделение двухмерного подмножества: {{1, 3, 5}, {7, 9, 11}}
    valarray<size_t> siz(2);
    siz[0] = 2;
    siz[1] = 3;
    valarray<size_t> gap(2);
    gap[0] = 6;
    gap[1] = 2;
    valarray<int> r3(v[gslice(0, siz, gap)]);
    print("2-d slice", r3);
    // Выделение подмножества по логической маске (элементы bool)
    valarray<bool> mask(false, 5);
    mask[1] = mask[2] = mask[4] = true;
    valarray<int> r4(v[mask]);
    print("v[mask]", r4);
    // Выделение подмножества по индексной маске (элементы size_t)
    size_t idx2[] = {2,2,3,6};
    valarray<size_t> mask2(idx2, 4);
    valarray<int> r5(v[mask2]);
    print("v[mask2]", r5);
    // Использование индексной маски при присваивании
    valarray<char> text("now is the time", 15);
    valarray<char> caps("NITT", 4);
    valarray<size_t> idx3(4);
    idx3[0] = 0;
    idx3[1] = 4;
    idx3[2] = 7;
    idx3[3] = 11;
    text[idx3] = caps;
    print("capitalized", text);
} ///:~
```

Конструктор slice получает три аргумента: начальный индекс, количество выделяемых элементов и «шаг» (расстояние между нужными элементами). Срезы могут использоваться для индексации существующих объектов valarray; при этом возвращается новый объект valarray с выделенными элементами. Объект valarray с элементами bool (такой, как полученный при выделении элементов по условию v > 6 в нашем примере) может потребоваться для индексирования других объектов valarray. При этом из целевого массива выделяются элементы, соответствующие истинным элементам маски. Как видно из приведенного примера, срезы и маски также могут применяться в качестве индексов слева от оператора присваивания. Обобщенные срезы gslice (от «generalized slice») в целом аналогичны срезам, однако количество элементов и шаги индексирования для них задаются массивами, что позволяет интерпретировать valarray как многомерный массив. В предыдущем примере из v выделяется подмассив 2×3, в котором индексы по одному измерению находятся в v на расстоянии 6 элементов, а по другому — на расстоянии двух элементов. Таким образом, фактически извлекается матрица:

```
1  3   5
7  9  11
```

Результат выполнения программы выглядит так:

```
slice(0,4,3): 1 4 7 10
elements > 6: 7 8 9 10
after squaring v: 1 2 3 16 5 6 49 8 9 100 11 12
v restored: 1 2 3 4 5 6 7 8 9 10 11 12
2-d slice: 1 3 5 7 9 11
```

```
v[mask]: 2 3 5
v[mask2]: 3 3 4 7
capitalized: N o w  I s  T h e  T i m e
```

Одним из примеров практического применения срезов является умножение матриц. Посмотрим, как должна выглядеть функция для умножения двух целочисленных матриц при использовании обычных массивов:

```
void matmult(const int a[][MAXCOLS], size_t m, size_t n,
             const int b[][MAXCOLS], size_t p, size_t q,
             int result[][MAXCOLS);
```

Функция умножает матрицу a с размерами m×n на матрицу b с размерами p×q; предполагается, что n и p равны. Как видите, без применения объекта valarray или его аналога приходится фиксировать максимальную вторую размерность в каждой матрице. Кроме того, такой подход затрудняет возвращение массива-результата по значению, поэтому вызывающая сторона обычно передает его в дополнительном аргументе.

Объекты valarray не только позволяют передавать матрицы произвольного размера, но и упрощают обработку матриц любого типа и возврат результата по значению. Вот как это делается:

```
//: C07:MatrixMultiply.cpp
// Умножение матриц с использованием объектов valarray
#include <cassert>
#include <cstddef>
#include <cmath>
#include <iostream>
#include <iomanip>
#include <valarray>
using namespace std;

// Вывод объекта valarray в виде матрицы
template<class T>
void printMatrix(const valarray<T>& a, size_t n) {
  size_t siz = n*n;
  assert(siz <= a.size());
  for(size_t i = 0; i < siz; ++i) {
    cout << setw(5) << a[i];
    cout << ((i+1)%n ? ' ' : '\n');
  }
  cout << endl;
}

// Умножение совместимых матриц в формате valarray
template<class T>
valarray<T>
matmult(const valarray<T>& a, size_t arows, size_t acols,
        const valarray<T>& b, size_t brows, size_t bcols) {
  assert(acols == brows);
  valarray<T> result(arows * bcols);
  for(size_t i = 0; i < arows; ++i)
    for(size_t j = 0; j < bcols; ++j) {
      // Внутреннее произведение строки a[i] и столбца b[j]
      valarray<T> row = a[slice(acols*i, acols, 1)];
      valarray<T> col = b[slice(j, brows, bcols)];
      result[i*bcols + j] = (row * col).sum();
    }
```

```
  return result;
}

int main() {
  const int n = 3;
  int adata[n*n] = {1.0,-1.2,2.2,-3.3,4.0};
  int bdata[n*n] = {3.4,-1.1,-3.0,-1.1,2};
  valarray<int> a(adata, n*n);
  valarray<int> b(bdata, n*n);
  valarray<int> c(matmult(a, n, n, b, n, n));
  printMatrix(c, n);
} ///:~
```

Каждый элемент итоговой матрицы c равен внутреннему произведению соответствующей строки a на столбец b. Используя срезы, мы можем извлечь эти строки и столбцы в объекты valarray и выполнить умножение с максимальной компактностью, применив глобальный оператор * и функцию sum(). Размеры итогового объекта valarray определяются на стадии выполнения; вам не придется беспокоиться о статическом ограничении размеров массива. Правда, линейные смещения для позиции [i][j] придется вычислять самостоятельно (см. ранее i*bcols + j), но свобода выбора размеров и типов того стоит.

Итоги

В этой главе мы не стремились к сколько-нибудь углубленному описанию контейнеров STL. Ограниченный объем не позволяет привести все подробности, но теперь полученных читателем знаний хватит для самостоятельного поиска информации в других ресурсах. Надеемся, что эта глава дает представление о потенциальных возможностях STL и показывает, насколько быстрее и эффективнее решаются многие задачи при хорошем понимании и умелом применении STL.

Упражнения

1. Создайте контейнер set<char>, откройте файл (имя которого передается в командной строке), прочитайте его содержимое по одному символу и занесите каждый символ в множество. Выведите результат и проанализируйте порядок следования элементов. Имеются ли в алфавите буквы, не встречающиеся в этом конкретном файле?
2. Создайте три контейнера с объектами Noisy: вектор, дек и список. Отсортируйте их. Напишите шаблон функции, которая получает вектор и дек, сортирует их и определяет время сортировки. Напишите специализированную шаблонную функцию, которая бы делала то же самое для списка (вместо обобщенного алгоритма в ней должна использоваться функция sort() класса списка). Сравните скорость сортировки для разных типов контейнеров.
3. Напишите программу для сравнения скорости сортировки списков с использованием алгоритмов list::sort() и std::sort() (то есть алгоритмической версии sort()).

4. Напишите генератор, который выдает случайные целые числа от 0 до 20 включительно, и заполните контейнер multiset<int>. Подсчитайте количество вхождений каждого значения по образцу программы MultiSetWordCount.cpp.
5. Измените программу StlShape.cpp так, чтобы вместо вектора в ней использовался дек.
6. Измените программу Reversible.cpp так, чтобы вместо вектора она работала с деками и списками.
7. Создайте контейнер stack<int> и заполните его последовательностью чисел Фибоначчи (длина последовательности передается в командной строке). Напишите цикл, который при каждой итерации получает два последние элемента стека и заносит в стек новый элемент.
8. Отсортируйте случайную последовательность чисел, используя только три стека: source, sorted и losers. Исходная последовательность находится в стеке source. Занесите число с вершины source в стек sorted. Продолжайте извлекать числа из стека source, сравнивая их с вершиной стека sorted. Меньшее из двух чисел извлекается из своего стека и отправляется в стек losers. Когда стек source останется пустым, повторите процесс, используя стек losers вместо source и стек source вместо losers. Работа алгоритма должна завершиться тогда, когда все числа будут помещены в стек sorted.
9. Откройте текстовый файл, имя которого передается в командной строке. Прочитайте содержимое файла по словам. Используя контейнер multiset<string>, подсчитайте количество вхождений для каждого слова.
10. Измените программу WordCount.cpp так, чтобы вставка элементов в отображение осуществлялась функцией insert() вместо оператора [ ].
11. Создайте класс с операторными функциями operator< и ostream& operator<<. В переменной класса должен храниться приоритет. Напишите генератор, создающий объекты класса со случайными приоритетами. Заполните с его помощью контейнер priority_queue, после чего извлеките элементы и убедитесь в том, что они хранятся в правильном порядке.
12. Перепишите программу Ring.cpp так, чтобы в ее базовой реализации вместо списка использовался дек.
13. Измените программу Ring.cpp так, чтобы ее базовая реализация выбиралась при помощи аргумента шаблона (пусть аргументом по умолчанию является list).
14. Создайте класс итератора BitBucket, который просто поглощает все передаваемые данные, никуда не записывая их.
15. Запрограммируйте разновидность игры «виселица». Создайте класс с переменными char и bool (логический признак, указывающий, угадывалась ли данная буква). Случайным образом выберите слово из файла и прочитайте его в вектор нового типа. В цикле запрашивайте символы у пользователя; после каждого введенного символа отображайте слово с угаданными символами (символы, которые еще не были угаданы, заменяются подчеркиваниями). Выберите исходное значение счетчика и уменьшайте его с каж-

дым вводимым символом. Если пользователь сможет угадать все слово раньше, чем счетчик упадет до нуля, он выиграл.

16. Откройте файл и прочитайте его в один объект string. Преобразуйте строку в stringstream. Организуйте чтение лексем из stringstream в list<string> с использованием итератора TokenIterator.
17. Сравните эффективность контейнера stack в зависимости от выбора базовой реализации (вектор, дек или список).
18. Создайте шаблон для реализации односвязного списка SList. Определите конструктор по умолчанию, а также функции begin(), end() (с использованием соответствующего вложенного итератора), insert(), erase() и деструктор.
19. Сгенерируйте последовательность случайных целых чисел, сохраните ее в массиве int. Инициализируйте valarray<int> содержимым массива. Вычислите сумму, минимальное и максимальное значения, среднее арифметическое и медиану целых чисел средствами контейнера valarray.
20. Создайте два объекта valarray<int> с 12 и 20 случайными числами int. Первый объект valarray интерпретируется как матрица int 3 × 4, а второй — как матрица int 4 × 5. Перемножьте объекты по правилам умножения матриц. Результат должен храниться в объекте valarray<int> с 15 элементами, представляющем матрицу 3 × 5. Умножение строк первой матрицы на столбцы второй должно производиться с использованием срезов. Выведите результат в виде прямоугольной матрицы.

Часть

Специальные возможности

Настоящий профессионализм проявляется во внимании к деталям. В этой части книги мы рассмотрим некоторые нетривиальные возможности C++, а также приемы программирования из арсенала истинных профессионалов C++.

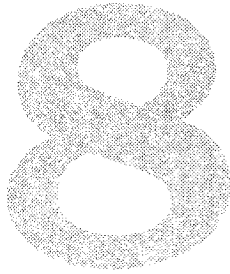
В некоторых ситуациях приходится отходить от канонов объектно-ориентированной разработки и определять тип объекта на стадии выполнения. Как правило, эту задачу лучше поручать виртуальным функциям, но при написании специальных программ (например, отладчиков, средств просмотра баз данных или классов) все равно требуется получать информацию о типе во время выполнения. В подобных ситуациях используется механизм RTTI. Этой теме посвящена глава 8.

Множественное наследование подвергалось критике в течение многих лет, а в некоторых языках оно вообще не поддерживается. Тем не менее, при правильном применении оно становится мощным средством для создания элегантного и эффективного кода. За прошедшее время появилось немало стандартных приемов программирования с использованием множественного наследования; мы рассмотрим их в главе 9.

Вероятно, одним из самых заметных новшеств в области программирования с момента появления объектно-ориентированных технологий стали *паттерны* — типовые решения целого круга сходных задач, связанных с разработкой программ. Паттерны применяются во многих ситуациях и могут быть реализованы в любом языке. В главе 10 описываются некоторые паттерны проектирования и способы их реализации в C++.

В главе 11 рассматриваются преимущества и трудности многопоточного программирования. Многопоточная модель не упоминается в текущей версии стандарта C++, хотя она поддерживается большинством операционных систем. На примере переносимой, свободно распространяемой библиотеки многопоточного программирования мы покажем преимущества многопоточной модели для разработки высокофункциональных приложений, быстро реагирующих на действия пользователя.

RTTI



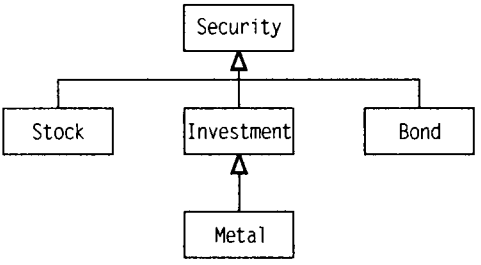
Механизм RTTI (RunTime Type Identification — идентификация типов в процессе исполнения) позволяет узнать динамический тип объекта в том случае, если у вас имеется только указатель или ссылка на базовый тип.

RTTI можно рассматривать как «вторичную» возможность C++; сугубо практическое средство для выхода из сложных нетипичных ситуаций. Обычно программист намеренно игнорирует конкретный тип объекта и позволяет механизму виртуальных функций реализовать правильное поведение для этого типа. Тем не менее, в отдельных случаях бывает полезно знать фактический (то есть «наиболее производный») тип объекта, для которого имеется только указатель на базовый тип. Наличие такой информации позволяет более эффективно выполнять особые операции или избавиться от неудобных ограничений, обусловленных интерфейсом базового класса. Виртуальные функции, предназначенные для получения динамической информации о типе, присутствуют в большинстве библиотек классов. Когда в C++ был добавлен механизм обработки исключений, для его реализации требовалась информация о динамическом типе объектов, а предоставить доступ к этой информации было несложно. В этой главе объясняется, для чего нужен механизм RTTI и как им правильно пользоваться.

Преобразования типов на стадии выполнения

Один из способов определения фактического типа объекта по ссылке или указателю на базовый класс основан на динамическом преобразовании типа. Поскольку на диаграммах наследования базовые классы обычно изображаются над производными классами, такие преобразования называются *понижающими*.

Рассмотрим следующую иерархию классов:



В следующей программе класс **Investment** поддерживает дополнительную операцию, которая не поддерживается другими классами, поэтому на стадии выполнения важно знать, ссылается ли на объект **Investment** указатель на **Security**, или нет. Чтобы реализовать проверку понижающих преобразований, в каждом классе хранится целочисленный идентификатор, отличающий его от других классов иерархии.

```
//: C08:CheckedCast.cpp
// Проверка преобразований типа на стадии выполнения
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Security {
protected:
  enum {BASEID = 0};
public:
  virtual ~Security() {}
  virtual bool isA(int id) { return (id == BASEID); }
};

class Stock : public Security {
  typedef Security Super;
protected:
  enum {OFFSET = 1, TYPEID = BASEID + OFFSET};
public:
  bool isA(int id) {
    return id == TYPEID || Super::isA(id);
  }
  static Stock* dynacast(Security* s) {
    return (s->isA(TYPEID)) ? static_cast<Stock*>(s) : 0;
  }
};

class Bond : public Security {
  typedef Security Super;
protected:
  enum {OFFSET = 2, TYPEID = BASEID + OFFSET};
public:
  bool isA(int id) {
    return id == TYPEID || Super::isA(id);
  }
  static Bond* dynacast(Security* s) {
    return (s->isA(TYPEID)) ? static_cast<Bond*>(s) : 0;
  }
```

```
};

class Investment : public Security {
  typedef Security Super;
protected:
  enum {OFFSET = 3, TYPEID = BASEID + OFFSET};
public:
  bool isA(int id) {
    return id == BASEID || Super::isA(id);
  }
  static Investment* dynacast(Security* s) {
    return (s->isA(TYPEID)) ?
      static_cast<Investment*>(s) : 0;
  }
  void special() {
    cout << "special Investment function\n";
  }
};

class Metal : public Investment {
  typedef Investment Super;
protected:
  enum {OFFSET = 4, TYPEID = BASEID + OFFSET};
public:
  bool isA(int id) {
    return id == BASEID || Super::isA(id);
  }
  static Metal* dynacast(Security* s) {
    return (s->isA(TYPEID)) ? static_cast<Metal*>(s) : 0;
  }
};

int main() {
  vector<Security*> portfolio;
  portfolio.push_back(new Metal);
  portfolio.push_back(new Investment);
  portfolio.push_back(new Bond);
  portfolio.push_back(new Stock);
  for (vector<Security*>::iterator it = portfolio.begin();
       it != portfolio.end(); ++it) {
    Investment* cm = Investment::dynacast(*it);
    if(cm)
      cm->special();
    else
      cout << "not a Investment" << endl;
  }
  cout << "cast from intermediate pointer:\n";
  Security* sp = new Metal;
  Investment* cp = Investment::dynacast(sp);
  if(cp) cout << "  it's an Investment\n";
  Metal* mp = Metal::dynacast(sp);
  if(mp) cout << "  it's a Metal too!\n";
  purge(portfolio);
} ///:~
```

Полиморфная функция isA() проверяет совместимость своего объекта с аргументом типа (id). Это означает, что id совпадает с typeID самого объекта либо одного из его предков (отсюда и вызов Super::isA()). Функция dynacast(), статическая во

всех классах, проверяет допустимость преобразования, вызывая функцию isA() для своего аргумента-указателя. Если isA() возвращает true, значит, преобразование допустимо, и функция возвращает соответствующим образом преобразованный указатель. В противном случае возвращается нулевой указатель. Он сообщает вызывающей стороне, что преобразование недопустимо, то есть исходный указатель ссылается на объект, несовместимый с нужным типом (и не преобразуемый к нему). Все эти манипуляции нужны для проверки опосредованных преобразований (например, от указателя на Security, ссылающегося на объект Metal, к указателю на Investment из предыдущего примера[1]).

В большинстве программ без понижающего преобразования можно обойтись. Более того, использовать его не рекомендуется, потому что основная часть проблем в объектно-ориентированных приложениях решается при помощи обычного полиморфизма. Тем не менее, возможность проверки допустимости преобразования к производному типу важна для служебных программ, таких как отладчики, средства просмотра классов и баз данных. В C++ такие проверяемые преобразования выполняются оператором dynamic_cast. Следующая программа представляет собой модификацию предыдущего примера с использованием оператора dynamic_cast:

```
//: C08:Security.h
#ifndef SECURITY.H
#define SECURITY.H
#include <iostream>

class Security {
public:
  virtual ~Security(){}
};

class Stock : public Security {};
class Bond : public Security {};

class Investment : public Security {
public:
  void special() {
    std::cout << "special Investment function" << std::endl;
  }
};
class Metal : public Investment {};
#endif // SECURITY_H ///:~

//: C08:CheckedCast2.cpp
// Использование RTTI и dynamic_cast
#include <vector>
#include "../purge.h"
#include "Security.h"
using namespace std;

int main() {
  vector<Security*> portfolio;
  portfolio.push_back(new Metal);
  portfolio.push_back(new Investment);
  portfolio.push_back(new Bond);
```

[1] В компиляторах Microsoft поддержку RTTI приходится специально активизировать; по умолчанию она отключена. Для этой цели используется ключ командной строки /GR.

```
  portfolio.push_back(new Stock);
  for (vector<Security*>::iterator it = portfolio.begin();
       it != portfolio.end(); ++it) {
    Investment* cm = dynamic_cast<Investment*>(*it);
    if(cm)
      cm->special();
    else
      cout << "not a Investment" << endl;
  }
  cout << "cast from intermediate pointer:" << endl;
  Security* sp = new Metal;
  Investment* cp = dynamic_cast<Investment*>(sp);
  if(cp) cout << "  it's an Investment" << endl;
  Metal* mp = dynamic_cast<Metal*>(sp);
  if(mp) cout << "  it's a Metal too!" << endl;
  purge(portfolio);
} ///:~
```

Эта программа гораздо короче, потому что большая часть кода исходного примера содержала избыточные проверки преобразований. Целевой тип dynamic_cast указывается в угловых скобках, как и для остальных новых операторов преобразований типов C++ (static_cast и т. д.), а преобразуемый объект передается в виде операнда. Чтобы понижающие преобразования были безопасными, оператор dynamic_cast требует, чтобы используемые типы были *полиморфными*[1], то есть содержали как минимум одну виртуальную функцию. К счастью, в базовом классе Security имеется виртуальный деструктор, поэтому нам не придется вставлять лишнюю функцию. Так как оператор dynamic_cast работает на стадии выполнения, используя таблицу виртуальных функций, он обычно требует больших издержек, чем остальные новые операторы преобразования типов.

Оператор dynamic_cast также может применяться к ссылкам вместо указателей, но так как «нулевых ссылок» не существует, информация о неудачном преобразовании должна передаваться другим способом. Этот «другой способ» основан на перехвате исключения bad_cast:

```
//: C08:CatchBadCast.cpp
#include <typeinfo>
#include "Security.h"
using namespace std;

int main() {
  Metal m;
  Security& s = m;
  try {
    Investment& c = dynamic_cast<Investment&>(s);
    cout << "It's an Investment" << endl;
  } catch (bad_cast&) {
    cout << "s is not an Investment type" << endl;
  }
  try {
    Bond& c = dynamic_cast<Bond&>(s);
    cout << "It's an Bond" << endl;
  } catch (bad_cast&) {
    cout << "It's not a Bond type" << endl;
  }
} ///:~
```

[1] Компилятор обычно вставляет указатель на таблицу RTTI класса в таблицу виртуальных функций.

Класс bad_cast определяется в заголовке <typeinfo>. Как и большая часть стандартной библиотеки, он принадлежит к пространству имен std.

Оператор typeid

Другой способ получения динамической информации об объекте основан на применении оператора typeid. Этот оператор возвращает объект класса type_info с информацией о типе объекта, к которому он был применен. Для полиморфных типов возвращается информация о «наиболее производном» (*динамическом*) типе, а в остальных случаях — статическая информация о типе. В частности, оператор typeid может использоваться для получения динамического имени объекта в формате const char*, как показано в следующем примере:

```
//: C08:TypeInfo.cpp
// Применение оператора typeid
#include <iostream>
#include <typeinfo>
using namespace std;

struct PolyBase {virtual ~PolyBase(){}};
struct PolyDer : PolyBase { PolyDer() {} };
struct NonPolyBase {};
struct NonPolyDer : NonPolyBase {NonPolyDer(int){} };

int main() {
  // Для полиморфных типов
  const PolyDer pd;
  const PolyBase* ppb = &pd;
  cout << typeid(ppb).name() << endl;
  cout << typeid(*ppb).name() << endl;
  cout << boolalpha << (typeid(*ppb) == typeid(pd))
    << endl;
  cout << (typeid(PolyDer) == typeid(const PolyDer))
    << endl;
  // Для неполиморфных типов
  const NonPolyDer npd(1);
  const NonPolyBase* nppb = &npd;
  cout << typeid(nppb).name() << endl;
  cout << typeid(*nppb).name() << endl;
  cout << (typeid(*nppb) == typeid(npd))
    << endl;
  // Для встроенных типов
  int i;
  cout << typeid(i).name() << endl;
} ///:~
```

Результат выполнения программы для одного конкретного компилятора выглядит так:

```
struct PolyBase const *
struct PolyDer
true
true
struct NonPolyBase const *
struct NonPolyBase
false
int
```

Первая строка просто воспроизводит статический тип ppb. Чтобы механизм RTTI вступил в силу, следует разыменовать указатель, как это сделано во второй строке. Обратите внимание: RTTI игнорирует квалификаторы const и volatile верхнего уровня. Для неполиморфных типов возвращается статический тип (то есть тип самого указателя). Как видите, RTTI также работает с встроенными типами.

Оказывается, результат выполнения оператора typeid невозможно сохранить в объекте type_info, так как конструкторы для него недоступны, а присваивание запрещено. Информация должна использоваться так, как показано в примере. Кроме того, конкретный вид строки, возвращаемой функцией type_info::name(), зависит от компилятора. Например, для класса с именем C некоторые компиляторы возвращают строку class C вместо C. Применение оператора typeid к выражению, разыменовывающему нулевой указатель, приводит к запуску исключения bad_typeid (также определяемому в <typeinfo>).

Следующий пример показывает, что для вложенных классов вызов type_info::name() возвращает полностью уточненное имя:

```
//: C08:RTTIandNesting.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class One {
  class Nested {};
  Nested* n;
public:
  One() : n(new Nested) {}
  ~One() { delete n; }
  Nested* nested() { return n; }
};

int main() {
  One o;
  cout << typeid(*o.nested()).name() << endl;
} ///:~
```

Поскольку Nested является вложенным типом по отношению к классу One, в результате выводится строка One::Nested.

Можно выяснить, предшествует ли объект type_info другому объекту type_info в некотором порядке, определяемом реализацией. Для этой цели используется функция before(type_info&), возвращающая true или false. Так, следующая строка проверяет, предшествует ли тип me типу you в текущем порядке:

```
if(typeid(me).before(typeid(you))) // ...
```

Такая возможность может пригодиться при использовании объектов type_info в качестве ключей.

Преобразование к промежуточным типам

Как было показано в предыдущей программе с иерархией классов Security, оператор dynamic_cast способен обнаруживать как фактический тип, так и промежуточные типы в многоуровневых иерархиях. Пример:

```
//: C08:IntermediateCast.cpp
#include <cassert>
```

```
#include <typeinfo>
using namespace std;

class B1 {
public:
  virtual ~B1() {}
};

class B2 {
public:
  virtual ~B2() {}
};

class MI : public B1, public B2 {};
class Mi2 : public MI {};

int main() {
  B2* b2 = new Mi2;
  Mi2* mi2 = dynamic_cast<Mi2*>(b2);
  MI* mi = dynamic_cast<MI*>(b2);
  B1* b1 = dynamic_cast<B1*>(b2);
  assert(typeid(b2) != typeid(Mi2*));
  assert(typeid(b2) == typeid(B2*));
  delete b2;
} ///:~
```

В этом примере возникают дополнительные трудности, связанные с множественным наследованием (эта тема рассматривается далее в настоящей главе, а также в главе 9). Если создать объект **Mi2** и преобразовать его к корневому типу (в данном случае выбирается один из двух возможных корней), то приведение к любому из производных уровней **MI** или **Mi2** через **dynamic_cast** оказывается успешным.

Возможно даже преобразование от одного корня к другому:

```
B1* b1 = dynamic_cast<B1*>(b2);
```

Такое преобразование проходит успешно, потому что **B2** в действительности ссылается на объект **Mi2**, который содержит подобъект типа **B1**.

Преобразование к промежуточным типам открывает интересное различие между операторами **dynamic_cast** и **typeid**. Оператор **typeid** всегда выдает указатель на статический объект **type_info**, описывающий динамический тип объекта. Таким образом, он не предоставляет информации о промежуточных уровнях. В следующем (истинном) выражении оператор **typeid**, в отличие от **dynamic_cast**, не воспринимает **b2** как указатель на производный тип:

```
typeid(b2) != typeid(Mi2*)
```

Тип **b2** просто соответствует фактическому типу указателя:

```
typeid(b2) != typeid(B2*)
```

Указатели на void

RTTI работает только с полноценными типами. Иначе говоря, при использовании оператора **typeid** должна быть доступна вся информация о классе. В частности, RTTI не работает с указателями на **void**:

```
//: C08:VoidRTTI.cpp
// RTTI и указатели на void
```

```
//!#include <iostream>
#include <typeinfo>
using namespace std;

class Stimpy {
public:
  virtual void happy() {}
  virtual void joy() {}
  virtual ~Stimpy() {}
};

int main() {
  void* v = new Stimpy;
  // Ошибка:
//!  Stimpy* s = dynamic_cast<Stimpy*>(v);
  // Ошибка:
//!  cout << typeid(*v).name() << endl;
} ///:~
```

Синтаксис void* фактически означает «информация о типе отсутствует»[1].

RTTI и шаблоны

RTTI нормально работает с шаблонами классов, поскольку они просто генерируют новые классы. В частности, RTTI позволяет легко определить имя текущего класса. Следующая программа выводит сообщения о вызове конструкторов и деструкторов:

```
//: C08:ConstructorOrder.cpp
// Порядок вызовов конструкторов
#include <iostream>
#include <typeinfo>
using namespace std;

template<int id> class Announce {
public:
  Announce() {
    cout << typeid(*this).name()
         << " constructor" << endl;
  }
  ~Announce() {
    cout << typeid(*this).name()
         << " destructor" << endl;
  }
};

class X : public Announce<0> {
  Announce<1> m1;
  Announce<2> m2;
public:
  X() { cout << "X::X()" << endl; }
  ~X() { cout << "X::~X()" << endl; }
};

int main() { X x; } ///:~
```

[1] Вызов dynamic_cast<void*> всегда дает адрес всего объекта (не одного из его подобъектов). Эта тема более подробно рассматривается в следующей главе.

В данном случае шаблон параметризуется по константам int, но подошли бы и типовые аргументы. Внутри конструктора и деструктора мы средствами RTTI определяем имя класса для вывода. В классе X используется наследование и композиция, чтобы порядок вызова конструкторов и деструкторов был нетривиальным. Результат выглядит так:

```
Announce<0> constructor
Announce<1> constructor
Announce<2> constructor
X::X()
X::~X()
Announce<2> destructor
Announce<1> destructor
Announce<0> destructor
```

Естественно, конкретный вывод зависит от того, как ваш компилятор представляет результат вызова name().

Множественное наследование

Механизм RTTI должен правильно работать со всеми сложными аспектами множественного наследования, включая виртуальные базовые классы (эта тема подробно рассматривается в следующей главе; возможно, вам стоит вернуться к этому разделу после прочтения главы 9).

```
//: C08:RTTIandMultipleInheritance.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class BB {
public:
  virtual void f() {}
  virtual ~BB() {}
};

class B1 : virtual public BB {};
class B2 : virtual public BB {};
class MI : public B1, public B2 {};

int main() {
  BB* bbp = new MI; // Upcast
  // Правильное определение имени:
  cout << typeid(*bbp).name() << endl;
  // Dynamic_cast работает правильно:
  MI* mip = dynamic_cast<MI*>(bbp);
  // Принудительное преобразование типа
  // в старом стиле невозможно:
//! MI* mip2 = (MI*)bbp; // Ошибка компиляции
} ///:~
```

Оператор typeid правильно определяет имя для фактического объекта даже через указатель на виртуальный базовый класс. Оператор dynamic_cast тоже работает правильно. Тем не менее, компилятор не позволяет выполнить принудительное преобразование типа старым способом:

```
MI* mip2 = (MI*)bbp; // Ошибка компиляции
```

Компилятор знает, что такое преобразование всегда является нежелательным, и поэтому требует использовать оператор dynamic_cast.

Области применения RTTI

Новички часто злоупотребляют возможностью получения информации типа по анонимному полиморфному указателю, потому что обычно они понимают логику работы RTTI раньше, чем логику работы механизма виртуальных функций. Многим программистам с опытом процедурного программирования нелегко избавиться от привычки делить программу на команды switch. RTTI позволяет реализовать такую логику, однако при этом теряются важные преимущества полиморфизма в разработке и сопровождении программ. Язык C++ проектировался с расчетом на то, что программист будет использовать виртуальные функции и прибегнет к RTTI только в случае необходимости.

Тем не менее, предполагаемое использование виртуальных функций требует наличия определения базового класса, поскольку на некоторой стадии расширения программы может оказаться, что базовый класс не содержит необходимых виртуальных функций. Если базовый класс взят из библиотеки или по другим причинам неподконтролен вам, единственным решением остается RTTI. Вы определяете новый тип и включаете в него дополнительные функции, а затем в другой точке программы выявляете этот конкретный тип и вызываете функцию. Полиморфизм и расширяемость программы при этом не страдают, поскольку добавление нового типа не требует массового перехода на логику switch. Тем не менее при включении в основную программу новых фрагментов, опирающихся на новые возможности, необходимо организовать идентификацию нового типа.

Расширение базового класса может привести к тому, что ради удобства одного конкретного класса во все остальные классы, производные от общей базы, придется включать какую-нибудь бессмысленную заглушку для чисто виртуальной функции. В результате интерфейс становится менее понятным и раздражает тех, кто должен переопределять чисто виртуальные функции при наследовании от этого базового класса.

Наконец, RTTI иногда помогает решить проблемы эффективности. Если программа использует полиморфизм так, как положено, но какой-нибудь объект реагирует на обобщенный код крайне неэффективно, можно идентифицировать этот тип средствами RTTI и написать специализированный код для повышения эффективности.

Пример

Следующий пример, демонстрирующий практическое применение RTTI, имитирует систему переработки мусора. Различные виды «мусора» попадают в общий контейнер, а затем сортируются в соответствии со своими динамическими типами.

```
//: C08:Trash.h
// Описания разных видов мусора
#ifndef TRASH_H
#define TRASH_H
#include <iostream>

class Trash {
```

```
  float _weight;
public:
  Trash(float wt) : _weight(wt) {}
  virtual float value() const = 0;
  float weight() const { return _weight; }
  virtual ~Trash() {
    std::cout << "~Trash()" << std::endl;
  }
};

class Aluminum : public Trash {
  static float val;
public:
  Aluminum(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(float newval) {
    val = newval;
  }
};

class Paper : public Trash {
  static float val;
public:
  Paper(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(float newval) {
    val = newval;
  }
};

class Glass : public Trash {
  static float val;
public:
  Glass(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(float newval) {
    val = newval;
  }
};
#endif // TRASH_H ///:~
```

Статические значения, определяющие стоимость единицы «вторсырья», определяются в файле реализации:

```
//: C08:Trash.cpp {O}
// Переработка мусора
#include "Trash.h"

float Aluminum::val = 1.67;
float Paper::val = 0.10;
float Glass::val = 0.23;
///:~
```

Шаблон sumValue() перебирает содержимое контейнера, вычисляет суммарную стоимость по разным видам мусора и отображает результаты:

```
//: C08:Recycle.cpp
// {L} Trash
// Переработка мусора
#include <cstdlib>
```

```
#include <ctime>
#include <iostream>
#include <typeinfo>
#include <vector>
#include "Trash.h"
#include "../purge.h"
using namespace std;

// Вычисление суммарной стоимости одного вида мусора:
template<class Container>
void sumValue(Container& bin, ostream& os) {
  typename Container::iterator tally = bin.begin();
  float val = 0;
  while(tally != bin.end()) {
    val += (*tally)->weight() * (*tally)->value();
    os << "weight of " << typeid(**tally).name()
       << " = " << (*tally)->weight() << endl;
    ++tally;
  }
  os << "Total value = " << val << endl;
}

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  vector<Trash*> bin;
  // Заполнение контейнера объектами Trash:
  for(int i = 0; i < 30; i++)
    switch(rand() % 3) {
      case 0 :
        bin.push_back(new Aluminum((rand() % 1000)/10.0));
        break;
      case 1 :
        bin.push_back(new Paper((rand() % 1000)/10.0));
        break;
      case 2 :
        bin.push_back(new Glass((rand() % 1000)/10.0));
        break;
    }
  // Внимание: в специализированных "мусорных баках"
  // хранятся фактические типы, а не базовый тип:
  vector<Glass*> glassBin;
  vector<Paper*> paperBin;
  vector<Aluminum*> alumBin;
  vector<Trash*>::iterator sorter = bin.begin();
  // Сортировка мусора:
  while(sorter != bin.end()) {
    Aluminum* ap =
      dynamic_cast<Aluminum*>(*sorter);
    Paper* pp =
      dynamic_cast<Paper*>(*sorter);
    Glass* gp =
      dynamic_cast<Glass*>(*sorter);
    if(ap) alumBin.push_back(ap);
    else if(pp) paperBin.push_back(pp);
    else if(gp) glassBin.push_back(gp);
    ++sorter;
  }
  sumValue(alumBin, cout);
  sumValue(paperBin, cout);
```

```
  sumValue(glassBin, cout);
  sumValue(bin, cout);
  purge(bin);
} ///:~
```

Сначала несортированный мусор бросается в один общий бак, так что специализированная информация типа «теряется». Но позднее для правильной сортировки необходимо получить фактическую информацию о типе, поэтому мы используем RTTI.

Данное решение можно улучшить за счет отображения, связывающего указатели на объекты type_info с вектором указателей на Trash. Для отображения необходим предикат упорядочения, поэтому мы предоставляем такой предикат с именем TInfoLess, вызывающий функцию type_info::before(). В процессе вставки в отображение указатели на Trash автоматически ассоциируются с их ключами type_info. Обратите внимание на изменение определения sumValue() в новом варианте решения:

```
//: C08:Recycle2.cpp
//{L} Trash
// Переработка мусора с использованием отображения
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <map>
#include <typeinfo>
#include <utility>
#include <vector>
#include "Trash.h"
#include "../purge.h"
using namespace std;

// Критерий сравнения указателей на type_info
struct TInfoLess {
  bool operator()(const type_info* t1, const type_info* t2)
  const { return t1->before(*t2); }
};

typedef map<const type_info*, vector<Trash*>, TInfoLess>
  TrashMap;

// Вычисление суммарной стоимости одного вида мусора:
void sumValue(const TrashMap::value_type& p, ostream& os) {
  vector<Trash*>::const_iterator tally = p.second.begin();
  float val = 0;
  while(tally != p.second.end()) {
    val += (*tally)->weight() * (*tally)->value();
    os << "weight of "
       << p.first->name()  // type_info::name()
       << " = " << (*tally)->weight() << endl;
    ++tally;
  }
  os << "Total value = " << val << endl;
}

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  TrashMap bin;
```

```
  // Заполнение контейнера объектами Trash:
  for(int i = 0; i < 30; i++) {
    Trash* tp;
    switch(rand() % 3) {
      case 0 :
        tp = new Aluminum((rand() % 1000)/10.0);
        break;
      case 1 :
        tp = new Paper((rand() % 1000)/10.0);
        break;
      case 2 :
        tp = new Glass((rand() % 1000)/10.0);
        break;
    }
    bin[&typeid(*tp)].push_back(tp);
  }
  // Вывод отсортированных результатов
  for(TrashMap::iterator p = bin.begin();
      p != bin.end(); ++p) {
    sumValue(*p, cout);
    purge(p->second);
  }
} ///:~
```

Теперь sumValue() вызывает функцию type_info::name() напрямую, потому что объект type_info стал доступен как первый компонент пары TrashMap::value_type. Тем самым предотвращается лишний вызов оператора typeid для получения имени типа Trash, необходимый в предыдущей версии программы.

Реализация и издержки RTTI

Обычно реализация RTTI базируется на включении дополнительного указателя в таблицу виртуальных функций класса. Этот указатель ссылается на структуру type_info для этого конкретного типа. Выражение typeid() реализуется достаточно просто: по указателю на таблицу виртуальных функций производится выборка указателя на type_info, через который осуществляется доступ к структуре type_info. Поскольку операция сводится к разыменованию двух указателей, она выполняется с постоянной сложностью.

Выражение dynamic_cast<*приемник**>(*указатель_на_источник*) в большинстве случаев также реализуется достаточно прямолинейно: сначала производится выборка информации RTTI для типов указателей на источник и приемник. Затем библиотечная функция проверяет, относится ли тип указателя на источник к типу *приемник** или к типу его базового класса. Возвращаемый ею указатель может быть модифицирован при множественном наследовании, если базовый тип не является первым базовым типом производного класса. Множественное наследование порождает и другие трудности из-за возможности многократного вхождения базовых типов в иерархию и использования виртуальных базовых классов.

Поскольку библиотечная функция в реализации dynamic_cast должна проверить список базовых классов, затраты на вызов dynamic_cast могут превышать затраты на вызов typeid (с другой стороны, вы получаете другую информацию, которая может оказаться критически важной для вашего решения), а на идентификацию базового класса может уйти больше времени, чем на идентификацию производного

класса. Кроме того, dynamic_cast позволяет сравнивать любую пару типов, не ограничиваясь сравнением типов, принадлежащих к одной иерархии. Это также усложняет работу библиотечной функции, используемой оператором dynamic_cast.

Итоги

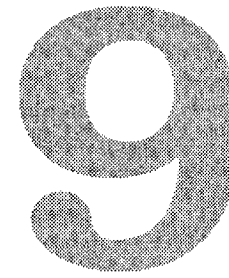
Хотя в общем случае указатели повышаются до базового класса с последующим использованием интерфейса базового класса (через виртуальные функции), иногда наличие информации о динамическом типе объекта, на который ссылается указатель на базовый класс, позволило бы повысить эффективность программы. Именно эту возможность вам предоставляет RTTI. Чаще всего ею злоупотребляют программисты, которые не понимают принципов работы виртуальных функций и применяют RTTI для ручного кодирования проверки типов. C++ предоставляет в ваше распоряжение мощный инструментарий и защиту от нарушений целостности типов, но если вы сознательно хотите неправильно или нетривиально задействовать какой-нибудь аспект языка, ничто не помешает вам в этом. Что ж, мы быстрее всего учимся на собственных ошибках.

Упражнения

1. Создайте класс Base с виртуальным деструктором и производный от него класс Derived. Создайте вектор указателей на Base, которые случайным образом распределяются между объектами Base и Derived. Используя содержимое этого вектора, заполните второй вектор всеми указателями на Derived. Сравните время выполнения с операторами typeid и dynamic_cast и посмотрите, какой вариант работает быстрее.
2. Измените класс C16:AutoCounter.h из первого тома так, чтобы он стал действительно полезным средством отладки. Он должен использоваться в качестве вложенного члена каждого класса, который вы планируете трассировать. Оформите AutoCounter в виде шаблона, параметризованного по имени внешнего класса. Во всех сообщениях об ошибках выводите имя класса средствами RTTI.
3. Используйте RTTI для вывода отладочной информации (полного имени специализированного шаблона) функцией typeid(). Специализируйте шаблон для разных типов и проанализируйте результаты.
4. Измените иерархию Instrument из главы 14 первого тома, но для начала скопируйте файл Wind5.cpp в другое место. Включите в класс Wind виртуальную функцию clearSpitValve() и переопределите ее для всех классов, производных от Wind. Создайте вектор указателей на Instrument; заполните его различными типами объектов Instrument, созданными оператором new. Далее при переборе элементов средствами RTTI выявляйте объекты класса Wind или производных от него. Вызовите для этих объектов функцию clearSpitValve(). Обратите внимание, что присутствие функции clearSpitValve() в базовом классе привело бы к раздражающему и ненужному усложнению интерфейса.

5. Включите в базовый класс из предыдущего упражнения функцию prepareInstrument(), которая бы вызывала соответствующую функцию (например, clearSpitValve() для духовых инструментов). Обратите внимание: функция prepareInstrument() уместно смотрится в базовом классе и позволяет обойтись без RTTI, как это сделано в предыдущем примере.
6. Создайте вектор с указателями на 10 случайно выбранных объектов Shape (например, Square и Circle). Функция draw() переопределяется в каждом классе и печатает размеры выводимого объекта (например, радиус окружности, там где это уместно). Напишите функцию main(), которая сначала рисует все объекты Square контейнера, отсортированные по длине периметра, а затем — все объекты Circle, отсортированные по радиусу.
7. Создайте большой вектор с указателями на случайные объекты Shape. Включите в Shape невиртуальную функцию draw(), которая бы средствами RTTI определяла динамический тип каждого объекта и исполняла соответствующий код «вывода» объекта в команде выбора. Затем перепишите иерархию «как положено», с использованием виртуальных функций. Сравните объем кода и время выполнения обоих решений.
8. Создайте иерархию классов Pet, включающую производные классы Dog, Cat и Horse. Также создайте иерархию Food с классами Beef, Fish и Oats. Класс Dog содержит функцию eat(), получающую параметр Beef; аналогично, функция Cat::eat() получает объект Fish, а объекты Oats передаются Horse::eat(). Создайте вектор указателей на случайные объекты Pet, переберите его элементы и вызовите для каждого функцию eat() с передачей правильного типа объекта Food.
9. Создайте глобальную функцию drawQuad(), получающую ссылку на объект Shape. Если параметр Shape представляет прямоугольную фигуру (то есть относится к фактическому типу Square или Rectangle), эта функция вызывает для него функцию draw(), а если нет — выводит предупреждающее сообщение. Переберите вектор указателей на объекты Shape и вызовите drawQuad() для каждого объекта. В векторе должны находиться указатели на Square, Rectangle, Circle и Triangle.
10. Отсортируйте вектор случайных объектов Shape по имени класса. Используйте функцию type_info::before() в качестве функции сравнения при сортировке.

Множественное наследование



Основной принцип множественного наследования формулируется достаточно просто: новый тип создается наследованием от нескольких базовых классов. Синтаксис выглядит именно так, как следует ожидать, и если иерархии классов достаточно просты, то и множественное наследование не порождает особых проблем.

Тем не менее, при множественном наследовании могут встречаться различные неоднозначности и странные ситуации, которые будут рассматриваться в этой главе. Но для начала стоит поближе познакомиться с этим предметом.

История

До появления C++ самым успешным объектно-ориентированным языком был Smalltalk. Он изначально проектировался как объектно-ориентированный язык. Его нередко называют *чистым* языком, тогда как C++ из-за поддержки разных парадигм программирования, помимо собственно объектно-ориентированной парадигмы, характеризуется как *гибридный*. Разработчики Smalltalk решили, что все классы должны быть производными от общего базового класса `Object` (так называемая *объектно-базированная иерархия*[1]). В Smalltalk невозможно создать новый класс, который бы не являлся производным от одного из существующих классов; следовательно, прежде чем браться за создание новых классов, программисту приходилось изучать библиотеку классов. Таким образом, иерархия классов Smalltalk представляет собой единое монолитное дерево.

Классы Smalltalk обычно обладают рядом общих характеристик, и уж по крайней мере обладают *некоторыми* общими характеристиками (унаследованными от класса `Object`), поэтому потребность в наследовании от нескольких базовых классов возникает нечасто. С другой стороны, C++ позволяет создать сколько угодно

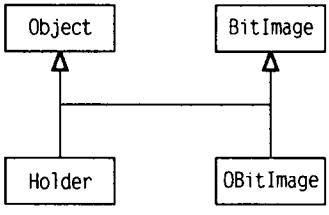
[1] Также используется в Java и других объектно-ориентированных языках.

независимых иерархий наследования. Следовательно, ради логической полноты язык должен допускать объединение классов из разных иерархий — так возникает идея множественного наследовании.

Впрочем, в течение долгого времени было непонятно, действительно ли нельзя обойтись без множественного наследования. Его необходимость в C++ вызывала (и продолжает вызывать) жаркие дискуссии. Поддержка множественного наследования была впервые реализована в 1989 году в AT&T cfront 2.0 и стала первым существенным изменением в языке с момента выхода версии 1.0[1]. С того времени в стандарт C++ был включен ряд других возможностей (прежде всего шаблоны), которые изменили классический подход к программированию, и множественное наследование отошло на второй план. Сейчас многие рассматривают его как «второстепенную» языковую возможность, редко применяемую в повседневной работе.

Один из самых веских доводов в пользу множественного наследования связан с контейнерами. Предположим, вы хотите создать контейнер, в котором можно хранить произвольные объекты. Первое очевидное решение — использовать тип `void*` для представления типов, хранящихся в контейнере. В Smalltalk эта задача решается просто: вы создаете контейнер для хранения типа `Object`, базового типа в иерархии Smalltalk. Поскольку все типы Smalltalk в конечном счете являются производными от `Object`, в контейнере с элементами `Object` можно хранить все, что угодно.

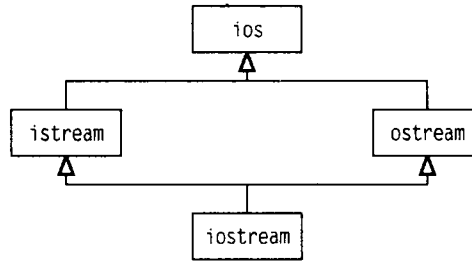
Но вернемся к C++. Предположим, разработчик A создает объектно-базированную иерархию с полезным набором контейнеров. В эту иерархию входит и нужный вам контейнер `Holder`. Далее в иерархии классов разработчика B обнаруживается другой нужный класс, скажем, класс `BitImage`, предназначенный для хранения графики. Создать контейнер `Holder` с элементами `BitImage` можно только одним способом: объявить новый класс производным от `Object`, чтобы он мог храниться как в `Holder`, так и в `BitImage`:



Это весьма убедительный аргумент в пользу множественного наследования. На базе этой модели были построены многие библиотеки классов. Тем не менее, как было показано в главе 5, поддержка шаблонов изменила подход к построению контейнеров, поэтому этот аргумент отчасти утратил свою значимость.

Применение множественного наследования также может быть оправданно, если оно делает общую архитектуру системы более гибкой или практичной (хотя бы внешне). Примером служит исходная архитектура библиотеки потоков ввода-вывода (которая, кстати, сохранилась и в современной шаблонной архитектуре, описанной в главе 4):

[1] Приведенные номера соответствуют внутренней нумерации версий, принятой в AT&T.



Классы istream и ostream сами по себе являются полезными, но они также могут быть объединены посредством множественного наследования в класс, сочетающий их характеристики и поведение. Класс ios предоставляет общую функциональность всех потоковых классов, поэтому в данном случае множественное наследование является механизмом логического структурирования программы.

Но какими бы соображениями не объяснялось применение множественного наследования, с ним обычно связано слишком много проблем.

Наследование интерфейса

Одно из применений множественного наследования, не вызывающее никаких возражений, связано с *наследованием интерфейса*. В C++ все наследование является *наследованием реализации*, поскольку все аспекты базового класса, интерфейс и реализация становятся частью производного класса. Унаследовать только часть класса (скажем, интерфейс) невозможно. Как объясняется в главе 14 первого тома, защищенное и закрытое наследование позволяет ограничить доступ к членам, унаследованным от базового класса, со стороны клиентов объекта производного класса, но на самом производном классе это не сказывается; он все равно содержит все данные базового класса и может обращаться ко всем незакрытым членам базового класса.

С другой стороны, наследование интерфейса только добавляет *объявления* функций в интерфейс производного класса. Такая возможность не поддерживается в C++ напрямую. Стандартная методика имитации наследования интерфейса основана на наследовании от *интерфейсного класса*, то есть класса, содержащего только объявления (но не данные или определения функций). Все объявления интерфейсного класса, кроме деструктора, должны быть чисто виртуальными функциями. Пример:

```
//: C09:Interfaces.cpp
// Множественное наследование интерфейса
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Printable {
public:
  virtual ~Printable() {}
  virtual void print(ostream&) const = 0;
};

class Intable {
```

```
public:
  virtual ~Intable() {}
  virtual int toInt() const = 0;
};

class Stringable {
public:
  virtual ~Stringable() {}
  virtual string toString() const = 0;
};

class Able : public Printable,
             public Intable,
             public Stringable {
  int myData;
public:
  Able(int x) { myData = x; }
  void print(ostream& os) const {
    os << myData;
  }
  int toInt() const {
    return myData;
  }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
};

void testPrintable(const Printable& p) {
  p.print(cout);
  cout << endl;
}
void testIntable(const Intable& n) {
  int i = n.toInt() + 1;
  cout << i << endl;
}
void testStringable(const Stringable& s) {
  string buf = s.toString() + "th";
  cout << buf << endl;
}

int main() {
  Able a(7);
  testPrintable(a);
  testIntable(a);
  testStringable(a);
} ///:~
```

Класс Able «реализует» интерфейсы Printable, Intable и Stringable, то есть предоставляет реализации для функций, объявленных в этих классах. Так как Able наследует от всех трех классов, объекты Able воплощают множественные связи типа «является частным случаем». Например, объект A может использоваться как объект Printable, потому что его класс Able открыто наследует от Printable и предоставляет реализацию print(). Тестовым функциям не нужно знать фактический тип их параметра; достаточно того, что тип переданного объекта может заменить тип их параметра.

Как обычно, решение с применением шаблонов получается более компактным:

```
//: C09:Interfaces2.cpp
// Неявное наследование интерфейса с применением шаблонов
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Able {
  int myData;
public:
  Able(int x) { myData = x; }
  void print(ostream& os) const { os << myData; }
  int toInt() const { return myData; }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
};

template<class Printable>
void testPrintable(const Printable& p) {
  p.print(cout);
  cout << endl;
}

template<class Intable>
void testIntable(const Intable& n) {
  cout << n.toInt() + 1 << endl;
}

template<class Stringable>
void testStringable(const Stringable& s) {
  cout << s.toString() + "th" << endl;
}

int main() {
  Able a(7);
  testPrintable(a);
  testIntable(a);
  testStringable(a);
} ///:~
```

Имена Printable, Intable и Stringable стали обычными параметрами шаблонов, которые предполагают существование операций, обозначенных в их контекстах. Иначе говоря, тестовые функции могут получать аргументы любого типа, предоставляющего определение функции класса с правильной сигнатурой и типом возвращаемого значения; наследование от общего базового класса перестает быть обязательным. Некоторым программистам больше нравится первый вариант, поскольку механизм наследования гарантирует реализацию предполагаемого интерфейса. Другие довольствуются тем, что если требуемые операции не поддерживаются типом аргументов шаблона, ошибка все равно будет выявлена на стадии компиляции. С технической точки зрения второй вариант реализует более «слабую» проверку типов, чем первый, но для программиста (и программы) эффект

одинаков. Это одна из форм слабой типизации, приемлемая для большинства современных программистов C++.

Наследование реализации

Как уже упоминалось, в C++ поддерживается только *реализация наследования*, то есть все аспекты базового класса всегда наследуются *полностью*. Иногда это удобно, потому что программисту не приходится беспокоиться о самостоятельной реализации всех аспектов производного класса, как в рассмотренном примере с наследованием интерфейса. Одно из стандартных применений множественного наследования — так называемые *подключаемые классы*, то есть классы, предназначенные для наделения других классов новыми возможностями через механизм наследования. Самостоятельные экземпляры подключаемых классов создаваться не должны.

Допустим, мы являемся клиентами класса, обеспечивающего доступ к базе данных. В нашем распоряжении имеется только заголовочный файл — здесь принципиально отсутствие доступа к исходным кодам реализации. Для примера рассмотрим следующую реализацию класса Database:

```
//: C09:Database.h
// Прототип ресурсного класса
#ifndef DATABASE_H
#define DATABASE_H
#include <iostream>
#include <stdexcept>
#include <string>

struct DatabaseError : runtime_error {
  DatabaseError(const string& msg)
    : std::runtime_error(msg) {}
};

class Database {
  std::string dbid;
public:
  Database(const string& dbStr) : dbid(dbStr) {}
  virtual ~Database(){}
  void open() throw(DatabaseError) {
    std::cout << "connected to " << dbid << std::endl;
  }
  void close() {
    std::cout << dbid << " closed" << std::endl;
  }
  //Прочие функции базы данных...
};
#endif // DATABASE_H ///:-
```

Непосредственные операции с базой данных (сохранение, выборка и т. д.) не приводятся, для нас они несущественны. При конструировании объекта класса задается строка подключения. Пользователь подключается к базе данных функцией Database::open() и отключается от нее функцией Database():

```
//: C09:UseDatabase.cpp
#include "Database.h"

int main() {
  Database db("MyDatabase");
```

```
  db.open();
  // Вызовы других функций...
  db.close();
}
/* Выходные данные:
connected to MyDatabase
MyDatabase closed
*/ ///:~
```

В типичной архитектуре «клиент-сервер» клиент работает с несколькими объектами, имеющими общее подключением к базе. База данных со временем должна быть закрыта, но только после того, как все операции с ней будут закончены. Обычно такая функциональность инкапсулируется в классе со счетчиком клиентских объектов, использующих подключение; когда счетчик падает до нуля, подключение автоматически закрывается. Чтобы добавить подсчет ссылок в класс Database, мы средствами множественного наследования «смешиваем» класс Database с классом Countable и создаем новый класс DBConnection. Подключаемый класс Countable выглядит так:

```
//: C09:Countable.h
// Подключаемый класс
#ifndef COUNTABLE_H
#define COUNTABLE_H
#include <cassert>

class Countable {
  long count;
protected:
  Countable() { count = 0; }
  virtual ~Countable() { assert(count == 0); }
public:
  long attach() { return ++count; }
  long detach() {
    return (--count > 0) ? count : (delete this, 0);
  }
  long refCount() const { return count; }
};
#endif // COUNTABLE_H ///:~
```

Очевидно, что этот класс не является самостоятельным, поскольку его конструктор объявлен защищенным (protected); он может использоваться только из дружественных или производных классов. Важно, что деструктор объявлен виртуальным — он вызывается только из команды delete this в detach(), а мы хотим, чтобы производные объекты правильно уничтожались[1].

Класс DBConnection наследует как от Database, так и от Countable, и предоставляет статическую функцию create() для инициализации подобъекта Countable. Перед вами пример использования паттерна Фабричный метод, описанного в следующей главе:

```
//: C09:DBConnection.h
// Использование подключаемого класса
#ifndef DBCONNECTION_H
#define DBCONNECTION_H
#include <cassert>
```

[1] И что еще важнее, мы не хотим непредсказуемых последствий. Отсутствие виртуального деструктора в базовом классе является ошибкой.

```
#include <string>
#include "Countable.h"
#include "Database.h"
using std::string;

class DBConnection : public Database, public Countable {
  DBConnection(const DBConnection&); // Запрет копирования
  DBConnection& operator=(const DBConnection&);
protected:
  DBConnection(const string& dbStr) throw(DatabaseError)
  : Database(dbStr) { open(); }
  ~DBConnection() { close(); }
public:
  static DBConnection*
  create(const string& dbStr) throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
    assert(con->refCount() == 1);
    return con;
  }
// Другие нужные функции...
};
#endif ///:~ DBCONNECTION_H ///:~
```

Мы получаем подключение к базе данных с подсчетом ссылок без модификации класса **Database**. Новая система с подсчетом ссылок гарантирует, что связь с базой не окажется преждевременно прерванной. При открытии и закрытии подключения (соответственно конструктором и деструктором **DBConnection**) используется идиома получения ресурсов при инициализации (RAII), упоминавшаяся в главе 1. Это упрощает работу с классом **DBConnection**:

```
//: C09:UseDatabase2.cpp
// Тестирование подсчета ссылок
#include <cassert>
#include "DBConnection.h"

class DBClient {
  DBConnection* db;
public:
  DBClient(DBConnection* dbCon) {
    db = dbCon;
    db->attach();
  }
  ~DBClient() {
    db->detach();
  }
  // Другие запросы к базе данных с использованием db...
};

int main() {
  DBConnection* db = DBConnection::create("MyDatabase");
  assert(db->refCount() == 1);
  DBClient c1(db);
  assert(db->refCount() == 2);
  DBClient c2(db);
  assert(db->refCount() == 3);
  // Используем базу данных, а затем освобождаем ресурс,
  // захваченный при вызове create.
  db->detach();
```

```
  assert(db->refCount() == 2);
} ///:~
```

Функция DBConnection::create() вызывает attach(), поэтому после завершения работы с объектом необходимо освободить захваченное подключение явным вызовом detach(). Обратите внимание: класс DBClient также управляет соединением при помощи RAII. При завершении программы деструкторы двух объектов DBClient уменьшают счетчик ссылок (вызовом функции detach(), унаследованной DBConnection от Countable). Подключение к базе данных закрывается (из-за виртуального деструктора Countable) при падении счетчика до нуля после уничтожения объекта c1.

Подключение функциональности через наследование чаще осуществляется с применением шаблонов, чтобы пользователь мог на стадии компиляции выбрать нужную разновидность. Это позволяет задействовать разные механизмы подсчета ссылок без повторного определения DBConnection. Вот как это делается:

```
//: C09:DBConnection2.h
// Параметризованное подключение функциональности
#ifndef DBCONNECTION_H
#define DBCONNECTION_H
#include "Database.h"
#include <cassert>
#include <string>
using std::string;

template<class Counter>
class DBConnection : public Database, public Counter {
  DBConnection(const DBConnection&); // Запрет копирования
  DBConnection& operator=(const DBConnection&);
protected:
  DBConnection(const string& dbStr) throw(DatabaseError)
  : Database(dbStr) { open(); }
  ~DBConnection() { close(); }
public:
  static DBConnection* create(const string& dbStr)
  throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
    assert(con->refCount() == 1);
    return con;
  }
// Другие нужные функции...
};
#endif // DBCONNECTION2_H ///:~
```

Единственное изменение — появление шаблонного префикса в определении класса (и переименование Countable в Counter для ясности). Класс доступа к базе данных тоже можно было бы оформить в виде параметра шаблона (если бы у нас было несколько классов доступа, из которых выбирался бы нужный вариант), но на этот раз класс получился вполне самостоятельным. В следующем примере исходная реализация Countable передается в качестве аргумента шаблона, но с таким же успехом можно было бы использовать любой тип, реализующий нужный интерфейс (attach(), detach() и т. д.):

```
//: C09:UseDatabase3.cpp
// Подключение функциональности через шаблон
#include <cassert>
#include "Countable.h"
```

```
#include "DBConnection2.h"

class DBClient {
  DBConnection<Countable>* db;
public:
  DBClient(DBConnection<Countable>* dbCon) {
    db = dbCon;
    db->attach();
  }
  ~DBClient() { db->detach(); }
};

int main() {
  DBConnection<Countable>* db =
    DBConnection<Countable>::create("MyDatabase");
  assert(db->refCount() == 1);
  DBClient c1(db);
  assert(db->refCount() == 2);
  DBClient c2(db);
  assert(db->refCount() == 3);
  db->detach();
  assert(db->refCount() == 2);
} ///:~
```

Общий паттерн для нескольких подключаемых классов выглядит так:

```
template<class Mixin1, class Mixin2, ..., class MixinK>
class Subject : public Mixin1,
                public Mixin2,
                ...
                public MixinK {...};
```

Дублирование подобъектов

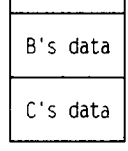
При наследовании в производный класс включаются копии всех переменных базового класса. Следующая программа демонстрирует возможное размещение нескольких базовых подобъектов в памяти[1]:

```
//: C09:Offset.cpp
// Размещение подобъектов в памяти
// при множественном наследовании
#include <iostream>
using namespace std;

class A { int x; };
class B { int y; };
class C : public A, public B { int z; };

int main() {
  cout << "sizeof(A) == " << sizeof(A) << endl;
  cout << "sizeof(B) == " << sizeof(B) << endl;
  cout << "sizeof(C) == " << sizeof(C) << endl;
  C c;
  cout << "&c == " << &c << endl;
```

[1] Конкретный вид результатов зависит от компилятора.

```
  A* ap = &c;
  B* bp = &c;
  cout << "ap == " << static_cast<void*>(ap) << endl;
  cout << "bp == " << static_cast<void*>(bp) << endl;
  C* cp = static_cast<C*>(bp);
  cout << "cp == " << static_cast<void*>(cp) << endl;
  cout << "bp == cp? " << boolalpha << (bp == cp) << endl;
  cp = 0;
  bp = cp;
  cout << bp << endl;
}
/* Результат:
sizeof(A) == 4
sizeof(B) == 4
sizeof(C) == 12
&c == 1245052
ap == 1245052
bp == 1245056
cp == 1245052
bp == cp? true
0
*/ ///:~
```

Как видите, подобъект B объекта c находится на расстоянии 4 байт от начала всего объекта. Можно предположить следующую структуру памяти:



Объект c начинается с подобъекта A, затем следует подобъект B и, наконец, — данные самого типа C. Поскольку C является частным случаем A и B, объекты этого класса могут быть преобразованы к любому из базовых типов. При повышении до типа A полученный указатель ссылается на подобъект A, начало которого совпадает с началом объекта C, поэтому адрес ap совпадает со значением &c. Но при повышении до типа B полученный указатель должен указывать на фактическое начало подобъекта B, так как класс B ничего не знает о классе C (не говоря уже о классе A). Другими словами, объект, на который ссылается bp, должен быть способен вести себя как автономный объект B (за исключением механизма полиморфного вызова).

Что же происходит при обратном преобразовании bp в C*? Поскольку исходный объект все-таки относился к типу C, местонахождение подобъекта B известно, поэтому указатель снова переводится на исходный адрес объекта производного класса. Если бы указатель bp изначально ссылался на автономный объект B вместо объекта C, такое преобразование было бы недопустимым[1]. Более того, в сравнении bp == sp указатель cp неявно преобразуется к B*, поскольку это един-

[1] Хотя компилятор бы не обнаружил ошибку. Впрочем, проблема решается при помощи оператора dynamic_cast — за подробностями обращайтесь к предыдущей главе.

ственный способ сделать сравнение осмысленным (то есть повышающее преобразование разрешено всегда), отсюда и истинный результат. Таким образом, при преобразовании типов подобъектов и полных типов применяется соответствующее смещение.

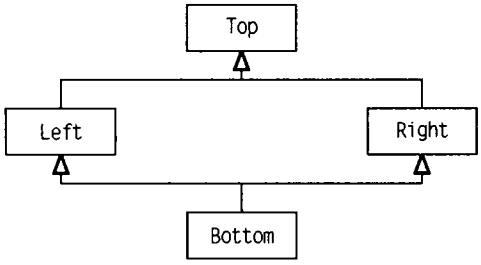
Естественно, необходимо организовать специальную проверку нулевых указателей, поскольку автоматическое вычитание смещения при преобразовании подобъекта B даст неверный результат, если указатель был равен нулю. По этой причине при преобразовании к B* или из него компилятор генерирует код, который сначала проверяет, не равен ли указатель нулю. Если указатель отличен от нуля, смещение применяется, а если нет, указатель так и остается нулевым.

Теперь предположим, что базовых классов несколько и у них в свою очередь имеется общий базовый класс. Как показывает следующий пример, в этом случае производный объект будет содержать несколько копий базового объекта верхнего уровня:

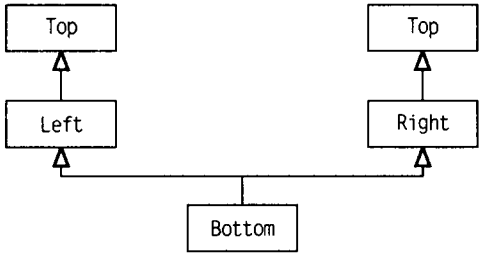
```
//: C09:Duplicate.cpp
// Дубликаты подобъектов
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
};

class Left : public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};

class Right : public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Left(i, k), Right(j, k) { w = m; }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl; // 20
} ///:~
```

Поскольку размер b равен 20 байт[1], полный объект Bottom содержит пять целых чисел. Типичная иерархия наследования в ситуациях такого рода выглядит так:

[1] Точнее, 5*sizeof(int). Компиляторы могут использовать дополнительное выравнивание, поэтому размер объекта по крайней мере не меньше суммы частей, хотя может превышать ее.



Перед вами так называемое «ромбовидное наследование», которое лучше было бы изобразить в виде



Неудобство этого подхода отражается в конструкторе класса Bottom из предыдущего примера. Пользователь думает, что для конструирования необходимы всего четыре числа, но какие аргументы должны передаваться в двух параметрах Left и Right? Хотя такая архитектура не является принципиально ошибочной, обычно это не то, что нужно вашему приложению. Кроме того, проблемы возникают и при попытке преобразовать указатель на Bottom в указатель на Top. Как было показано ранее, преобразование может потребовать дополнительной настройки адреса в зависимости от смещения подобъекта внутри полного объекта, но в данном случае приходится выбирать между *двумя* подобъектами Top. Компилятор не знает, какой именно подобъект он должен выбрать, поэтому такое повышающее преобразование запрещается как неоднозначное. Аналогичные рассуждения объясняют, почему объект Bottom не сможет вызвать функцию, определенную только в Top. Для такой функции Top::f() вызов b.f() потребует ссылки на подобъект Top в контексте исполнения, а таких объектов два.

Виртуальные базовые классы

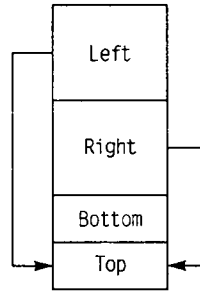
В подобных ситуациях помогло бы *настоящее* ромбовидное наследование, при котором один объект Top совместно используется обоими подобъектами, Left и Right, в полном объекте Bottom, как показано на первой диаграмме. Чтобы добиться этой цели, следует объявить Top *виртуальным базовым классом* для Left и Right:

```
//: C09:VirtualBase.cpp
// Совместное использование подобъектов
// через виртуальный базовый класс base
#include <iostream>
```

```
using namespace std;

class Top {
protected:
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top(){}
  friend ostream&
  operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
protected:
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};

class Right : virtual public Top {
protected:
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    return os << b.x << ',' << b.y << ',' << b.z
      << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl;
  cout << b << endl;
  cout << static_cast<void*>(&b) << endl;
  Top* p = static_cast<Top*>(&b);
  cout << *p << endl;
  cout << static_cast<void*>(p) << endl;
  cout << dynamic_cast<void*>(p) << endl;
} ///:~
```

Все виртуальные базы заданного типа относятся к одному объекту независимо от его местонахождения в иерархии[1]. Таким образом, экземпляр объекта Bottom имеет следующую структуру:

[1] Мы используем общепринятый термин «иерархия», хотя графическое представление отношений множественного наследования правильнее было бы называть *ориентированным ациклическим графом*, или *решеткой*.



Каждый из подобъектов Left и Right содержит указатель (или его концептуальный эквивалент) на общий подобъект Top, а все обращения к этому подобъекту из функций Left и Right осуществляются через эти указатели[1]. Теперь повышение Bottom до Top становится однозначным, потому что существует только один объект Top, к которому может осуществляться преобразование.

Результат выполнения этой программы выглядит так:

```
36
1,2,3,4
1245032
1
1245060
1245032
```

Выведенные адреса наводят на мысль, что в этой конкретной реализации подобъект Top хранится в конце полного объекта (хотя на самом деле неважно, где именно он находится). Преобразование к void* через dynamic_cast всегда дает адрес полного объекта.

Хотя с технической точки зрения это незаконно[2], при удалении виртуального деструктора (и dynamic_cast, чтобы программа компилировалась) размер Bottom уменьшается до 24 байт. Похоже, достигается экономия в размере трех указателей. Почему?

Не стоит воспринимать эти числа слишком буквально. В других компиляторах добавление виртуального конструктора увеличивает размер объекта всего на четыре байта. Наверное, эти секреты могут раскрыть только сами разработчики компиляторов. А мы лишь можем сказать, что при множественном наследовании производный объект должен вести себя так, словно он содержит несколько таблиц указателей VPTR, по одному для каждого из его непосредственных базовых классов, содержащих виртуальные функции. Ни больше, ни меньше. Разработчики компиляторов могут использовать любые оптимизации по своему усмотрению, но поведение должно оставаться одним и тем же.

В этой программе самое странное впечатление производит инициализатор Top в конструкторе Bottom. Обычно нам не приходится беспокоиться об инициализации других подобъектов, кроме непосредственных базовых классов, поскольку эти

---

[1] Присутствие этих указателей объясняет, почему размер b заметно превышает размер четырех целых чисел. Этот факт (отчасти) объясняется затратами, связанными с использованием виртуальных базовых классов. Кроме того, необходимо учитывать затраты на хранение VPTR, обусловленные наличием виртуального деструктора.

[2] Базовые классы должны иметь виртуальные деструкторы, но многие компиляторы позволяют проводить подобные эксперименты.

классы сами обеспечивают инициализацию своих баз. Тем не менее, от Bottom к Top ведут разные пути, поэтому надежда на передачу необходимых инициализационных данных промежуточными классами Left и Right создает неоднозначность: кто именно должен отвечать за инициализацию? По этой причине виртуальная база должна инициализироваться *последним производным классом* в иерархии. А как насчет выражений в конструкторах Left и Right, которые тоже инициализируют Top? Конечно, они необходимы для создания самостоятельных объектов Left и Right, но при создании объекта Bottom эти выражения должны *игнорироваться* (отсюда нули в их инициализаторах в конструкторе Bottom — при выполнении конструкторов Left и Right в контексте объекта Bottom любые значения в этих позициях игнорируются). Компилятор позаботится обо всем за вас, но вы должны понимать, кто и за что отвечает. Всегда следите за тем, чтобы все *конкретные* (не абстрактные) классы в иерархиях множественного наследования обеспечивали должную инициализацию всех своих виртуальных базовых классов.

Эти правила относятся не только к инициализации, но и ко всем операциям, распространяющимся на иерархию классов. Рассмотрим оператор записи в поток из предыдущего фрагмента. Данные были объявлены защищенными, чтобы можно было «сжульничать» и обратиться к унаследованным данным из операторной функции operator<<(ostream&,const Bottom&). Обычно бывает разумнее поручить вывод каждого подобъекта соответствующему классу и предоставить производному классу вызывать функции базовых классов по мере необходимости. Но что произойдет, если мы попытаемся использовать этот подход с оператором <<, как показано в следующем фрагменте?

```
//: C09:VirtualBase2.cpp
// Пример того, как НЕ СЛЕДУЕТ
// реализовывать оператор <<
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
  friend ostream&
  operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
  friend ostream&
  operator<<(ostream& os, const Left& l) {
    return os << static_cast<const Top&>(l) << ',' << l.y;
  }
};

class Right : virtual public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
```

```
  friend ostream&
  operator<<(ostream& os, const Right& r) {
    return os << static_cast<const Top&>(r) << ',' << r.z;
  }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    return os << static_cast<const Left&>(b)
      << ',' << static_cast<const Right&>(b)
      << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << b << endl;  // 1,2,1,3,4
} ///:~
```

Нельзя просто передать ответственность наверх, как обычно, потому что оператор вывода каждого из классов Left и Right вызывает оператор вывода Top, а это ведет к дублированию данных. Взамен необходимо вручную имитировать то, что компилятор делает при инициализации. В одном из решений в классах определяются специальные функции, которые знают о существовании виртуального базового класса и игнорируют его при выводе (так что решение задачи остается на долю последнего производного класса):

```
//: C09:VirtualBase3.cpp
// Правильная реализация оператора <<
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
  friend ostream&
  operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
  int y;
protected:
  void specialPrint(ostream& os) const {
    // Выводятся только данные Left
    os << ','<< y;
  }
public:
  Left(int m, int n) : Top(m) { y = n; }
  friend ostream&
  operator<<(ostream& os, const Left& l) {
```

```
    return os << static_cast<const Top&>(l) << ',' << l.y;
  }
};

class Right : virtual public Top {
  int z;
protected:
  void specialPrint(ostream& os) const {
    // Выводятся только данные Right
    os << ','<< z;
  }
public:
  Right(int m, int n) : Top(m) { z = n; }
  friend ostream&
  operator<<(ostream& os, const Right& r) {
    return os << static_cast<const Top&>(r) << ',' << r.z;
  }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    os << static_cast<const Top&>(b);
    b.Left::specialPrint(os);
    b.Right::specialPrint(os);
    return os << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << b << endl;  // 1,2,3,4
} ///:~
```

Функции specialPrint() объявлены защищенными, потому что они будут вызываться только из Bottom. Они выводят лишь собственные данные, игнорируя данные подобъекта Top, потому что вызов этих функций находится под контролем оператора << класса Bottom. Последний должен знать о существовании виртуального базового класса, по аналогии с конструктором Bottom. Аналогичные рассуждения применимы к операторам присваивания в иерархиях с виртуальным базовым классом, а также к любым функциям (как функциям классов, так и внешним), выполняющим совместную работу во всех классах иерархии.

После обсуждения виртуальных базовых классов можно показать «настоящую» процедуру инициализации объектов. Поскольку виртуальные классы порождают совместно используемые подобъекты, логично, что эти объекты должны быть доступны до того, как они начнут использоваться. Следовательно, инициализация подобъектов должна выполняться в следующем порядке (с рекурсией).

1. Инициализация всех подобъектов виртуальных базовых классов в порядке «сверху вниз, слева направо» по отношению к их расположению в определениях классов.
2. Инициализация подобъектов невиртуальных базовых классов в обычном порядке.

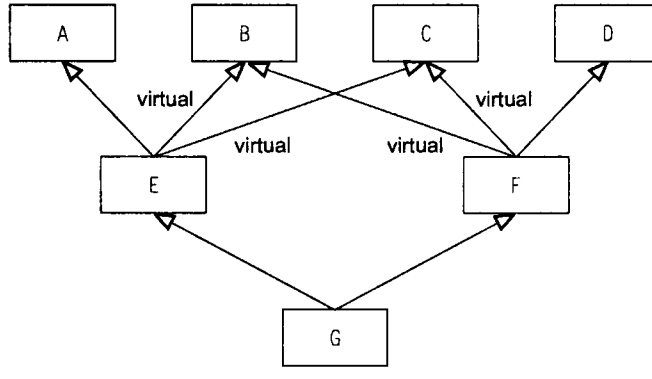
3. Инициализация вложенных объектов в порядке объявления.
4. Выполнение конструктора для всего объекта.

Следующая программа показывает, как это происходит:

```
//: C09:VirtInit.cpp
// Порядок инициализации объектов при наличии
// виртуальных базовых классов
#include <iostream>
#include <string>
using namespace std;

class M {
public:
  M(const string& s) {
    cout << "M " << s << endl;
  }
};

class A{
  M m;
public:
  A(const string& s) : m("in A") {
     cout << "A " << s << endl;
  }
  virtual ~A() {}
};

class B
{
  M m;
public:
  B(const string& s) : m("in B")  {
    cout << "B " << s << endl;
  }
  virtual ~B() {}
};

class C
{
  M m;
public:
  C(const string& s) : m("in C")  {
    cout << "C " << s << endl;
  }
  virtual ~C() {}
};

class D
{
  M m;
public:
  D(const string& s) : m("in D") {
    cout << "D " << s << endl;
  }
  virtual ~D() {}
};

class E : public A, virtual public B, virtual public C
{
  M m;
```

```
public:
  E(const string& s)
  : A("from E"), B("from E"), C("from E"), m("in E") {
    cout << "E " << s << endl;
  }
};

class F : virtual public B, virtual public C, public D
{
  M m;
public:
  F(const string& s)
  : B("from F"), C("from F"), D("from F"), m("in F") {
    cout << "F " << s << endl;
  }
};

class G : public E, public F
{
  M m;
public:
  G(const string& s)
  : B("from G"), C("from G"), E("from G"),
    F("from G"), m("in G") {
    cout << "G " << s << endl;
  }
};

int main() {
  G g("from main");
} ///:~
```

Отношения между классами из этого примера представлены на следующей диаграмме:



Каждый класс содержит вложенный объект типа M. Отметьте, что виртуальными являются лишь четыре наследования: E от B и C, а также F от B и C. Результат выполнения программы выглядит так:

```
M in B
B from G
M in C
C from G
M in A
A from E
```

```
M in E
E from G
M in D
D from F
M in F
F from G
M in G
G from main
```

Инициализация g требует предварительной инициализации подобъектов E и F, но сначала должны быть инициализированы подобъекты B и C, поскольку они относятся к виртуальным базовым классам и инициализируются по списку инициализации G, последнего производного класса в иерархии. Класс B не имеет базовых классов, поэтому по правилу 3 инициализируется его вложенный объект m, после чего конструктор выводит сообщение B from G; то же происходит с подобъектом C в E. Подобъект E требует инициализации подобъектов A, B и C. Поскольку подобъекты B и C уже были инициализированы, следующим инициализируется подобъект A подобъекта E, а затем — сам подобъект E. Далее сценарий повторяется для подобъекта F объекта g, но без дублирования инициализации виртуальных базовых классов.

Проблемы разрешения имен

Неоднозначности, рассмотренные нами для подобъектов, относятся к любым именам, в том числе к именам функций. Если класс имеет несколько непосредственных базовых классов с одноименными функциями, то при вызове одной из этих функций компилятор не будет знать, какую функцию выбрать. Рассмотрим пример подобной ошибки:

```
// C09:AmbiguousName.cpp {-xo}

class Top {
public:
  virtual ~Top() {}
};

class Left : virtual public Top {
public:
  void f() {}
};

class Right : virtual public Top {
public:
  void f() {}
};

class Bottom : public Left, public Right {};

int main() {
  Bottom b;
  b.f();     // Ошибка
} ///:~
```

Класс Bottom унаследовал две одноименные функции (сигнатура несущественна, поскольку разрешение имен происходит до разрешения перегрузки) и не может различить их. Стандартный способ ликвидации таких неоднозначностей основан на уточнении вызова функции именем базового класса:

```
//: C09:BreakTie.cpp

class Top {
public:
   virtual ~Top() {}
};

class Left : virtual public Top {
public:
   void f() {}
};

class Right : virtual public Top {
public:
   void f() {}
};

class Bottom : public Left, public Right {
public:
  using Left::f;
};

int main() {
   Bottom b;
   b.f();     // Вызов Left::f()
} ///:~
```

Имя **Left::f** теперь находится в области видимости **Bottom**, поэтому имя **Right::f** вообще не рассматривается. Если потребуется ввести дополнительные возможности, выходящие за пределы **Left::f()**, вы реализуете функцию **Bottom::f()**, которая вызывает **Left::f()**.

Между одноименными функциями в разных ветвях иерархии часто возникают конфликты. В следующей иерархии такой проблемы нет:

```
//: C09:Dominance.cpp

class Top {
public:
   virtual ~Top() {}
   virtual void f() {}
};

class Left : virtual public Top {
public:
   void f(){}
};

class Right : virtual public Top {};

class Bottom : public Left, public Right {};

int main() {
   Bottom b;
   b.f(); // Вызывает Left::f()
} ///:~
```

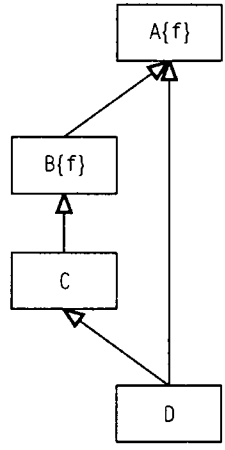
В этой иерархии функции **Right::f()** нет. Для выражения **b.f()** будет выбрана функция **Left::f()**. Почему? Давайте представим, что класс **Right** не существует, то есть мы имеем дело с одиночным наследованием **Top <= Left <= Bottom**. Конечно, в соответствии со стандартными правилами области видимости для выраже-

ния b.f() будет вызвана функция Left.f(): производный класс рассматривается как находящийся во вложенной области видимости базового класса. В общем случае имя A::f *доминирует* над именем B::f, если A наследует от B, напрямую или косвенно, или другими словами, если A является «более производным» в иерархии, чем B[1]. Выбирая между двумя одноименными функциями, компилятор выбирает доминирующее имя. Если такового не окажется, возникает неоднозначность.

Следующая программа демонстрирует принцип доминирования:

```
//: C09:Dominance2.cpp
#include <iostream>
using namespace std;

class A {
public:
   virtual ~A() {}
   virtual void f() {cout << "A::f\n";}
};

class B : virtual public A {
public:
   void f() {cout << "B::f\n";}
};

class C : public B {};
class D : public C, virtual public A {};

int main()
{
   B* p = new D;
   p->f();  // calls B::f()
   delete p;
} ///:~
```

Иерархия классов этого примера выглядит так:



[1] Обратите внимание: виртуальное наследование играет важную роль в этом примере. Если бы класс Top не был виртуальным базовым классом, то в производный класс содержал бы несколько подобъектов Top, и неоднозначность осталась бы. Доминирование при множественном наследовании действует только при наличии виртуальных базовых классов.

Класс A в этом случае является (непосредственно) базовым по отношению к B, поэтому имя B::f доминирует над A::f.

Отказ от множественного наследования

Принимая решение о применении множественного наследования, следует задать себе по крайней мере два вопроса.

- Должен ли новый тип предоставлять открытые интерфейсы обоих классов? (Подумайте, нельзя ли вложить один класс в другой, чтобы в новом классе проявлялась лишь часть его интерфейса.)
- Потребуется ли выполнять повышающее преобразование к обоим базовым классам (или к большему количеству базовых классов)?

Если хотя бы на один вопрос вы ответите «нет», то без множественного наследования можно — и, скорее всего, нужно — обойтись.

Обращайте особое внимание на ситуации, в которых один класс должен проходить повышающее преобразование только как аргумент функции. В этом случае класс можно оформить как вложенный и включить в новый класс функцию автоматического преобразования типа для получения ссылки на вложенный объект. Всюду, где объект нового класса передается в аргументе функции, рассчитывающей получить вложенный объект, используется функция преобразования типа[1]. Тем не менее, преобразование типа не годится для обычного полиморфного выбора функции; для этого необходимо наследование. Правила хорошего стиля проектирования гласят, что композиции следует отдавать предпочтение перед наследованием.

Расширение интерфейса

Одно из лучших применений множественного наследования связано с использованием кода, находящегося вне вашего контроля. Предположим, вам досталась библиотека с заголовочным файлом и откомпилированными функциями, но без исходных текстов. Библиотека представляет собой иерархию классов с виртуальными функциями и содержит глобальные функции, получающие указатели на базовый класс библиотеки; иначе говоря, библиотечные объекты используются полиморфно. Теперь допустим, вы строите на базе этой библиотеки свое приложение и пишете собственный код с полиморфным применением базового класса.

Позднее, на стадии разработки или сопровождения проекта, вдруг выясняется, что интерфейс базового класса, предоставляемый разработчиком, вас не устраивает: какая-нибудь его невиртуальная функция должна быть виртуальной, или же в нем отсутствует виртуальная функция, необходимая для решения вашей задачи. Возможно, вам поможет множественное наследование.

Рассмотрим примерный заголовочный файл библиотеки:

```
//: C09:Vendor.h
// Заголовок класса, предоставленный разработчиком.
```

[1] Джерри Шварц (Jerry Schwarz), автор библиотеки потоков ввода-вывода, неоднократно говорил, что если бы ему пришлось проектировать библиотеку заново, то он бы убрал множественное наследование из ее архитектуры и заменил его различными потоковыми буферами и операторами преобразования.

```
// В вашам распоряжении имеется только он
// и откомпилированный файл Vendor.obj.
#ifndef VENDOR_H
#define VENDOR_H

class Vendor {
public:
  virtual void v() const;
  void f() const;  // Может, функцию нужно сделать виртуальной...
  ~Vendor();    // Деструктор не виртуален!
};

class Vendor1 : public Vendor {
public:
  void v() const;
  void f() const;
  ~Vendor1();
};

void A(const Vendor&);
void B(const Vendor&);
// И т. д.
#endif // VENDOR_H ///:~
```

Конечно, библиотека в действительности содержит больше производных классов и обладает более обширным интерфейсом. Обратите внимание на функции A() и B(), которые получают ссылку на базовый класс и интерпретируют ее полиморфно. Вот как выглядит файл реализации библиотеки:

```
//: C09:Vendor.cpp {O}
// Этот код откомпилирован и недоступен
// для пользователя библиотеки.
#include "Vendor.h"
#include <iostream>
using namespace std;

void Vendor::v() const { cout << "Vendor::v()" << endl; }

void Vendor::f() const { cout << "Vendor::f()" << endl; }

Vendor::~Vendor() { cout << "~Vendor()" << endl; }

void Vendor1::v() const { cout << "Vendor1::v()" << endl; }

void Vendor1::f() const { cout << "Vendor1::f()" << endl; }

Vendor1::~Vendor1() { cout << "~Vendor1()" << endl; }

void A(const Vendor& V) {
  // ...
  V.v();
  V.f();
  //..
}
void B(const Vendor& V) {
  // ...
  V.v();
  V.f();
  //..
} ///:~
```

Но в вашем проекте исходные тексты недоступны. Вместо этого вы получаете откомпилированный файл Vendor.obj или Vendor.lib (или с другим расширением, принятым в вашей системе).

При использовании библиотеки возникают проблемы. Во-первых, деструктор базового класса не является виртуальным[1]. Во-вторых, функция f() тоже не является виртуальной; видимо, разработчик библиотеки решил, что это не обязательно. В-третьих, в интерфейсе базового класса отсутствует функция, абсолютно необходимая для решения вашей задачи. Также предположим, что вы уже написали довольно-таки объемистый код с использованием существующего интерфейса (не говоря уже о неподконтрольных вам функциях A() и B()), и изменять его не хочется.

Чтобы выйти из положения, создайте собственный интерфейс класса и новый набор производных классов от вашего интерфейса и существующих классов:

```
//: C09:Paste.cpp
//{L} Vendor
// Решение проблемы с использованием
// множественного наследования
#include <iostream>
#include "Vendor.h"
using namespace std;

class MyBase { // Исправление интерфейса Vendor
public:
  virtual void v() const = 0;
  virtual void f() const = 0;
  // Новая интерфейсная функция:
  virtual void g() const = 0;
  virtual ~MyBase() { cout << "~MyBase()\n"; }
};

class Paste1 : public MyBase, public Vendor1 {
public:
  void v() const {
    cout << "Paste1::v()" << endl;
    Vendor1::v();
  }
  void f() const {
    cout << "Paste1::f()" << endl;
    Vendor1::f();
  }
  void g() const {
    cout << "Paste1::g()" << endl;
  }
  ~Paste1() { cout << "~Paste1()" << endl; }
};

int main() {
  Paste1& p1p = *new Paste1;
  MyBase& mp = p1p; // Повышающее преобразование
  cout << "calling f()\n";
  mp.f();  // Правильное поведение
  cout << "calling g()\n";
  mp.g(); // Новое поведение
  cout << "calling A(p1p)\n";
```

[1] Мы встречали такое в коммерческих библиотеках C++, по крайней мере, в ранних.

```
  A(plp): // Старое поведение
  cout << "calling B(plp)\n";
  B(plp):  // Старое поведение
  cout << "delete mp\n";
  // Удаление ссылки на объект в куче:
  delete &mp; // Правильное поведение
} ///:~
```

В классе MyBase (*не использующем* множественное наследование) и функция f(), и деструктор объявлены виртуальными, а в интерфейс добавлена новая виртуальная функция g(). Теперь мы должны заново создать каждый из производных классов исходной библиотеки, подключая к нему новый интерфейс средствами множественного наследования. Функциям Paste1::v() и Paste1::f() достаточно вызвать свои прототипы из базового класса. Но теперь при следующем повышающем преобразовании MyBase, как в main(), все вызовы через mp, включая вызов delete, становятся полиморфными:

```
MyBase* mp = plp; // Повышающее преобразование
```

Кроме того, новая интерфейсная функция g() тоже может вызываться через mp. Результат выполнения программы выглядит так:

```
calling f()
Paste1::f()
Vendor1::f()
calling g()
Paste1::g()
calling A(plp)
Paste1::v()
Vendor1::v()
Vendor::f()
calling B(plp)
Paste1::v()
Vendor1::v()
Vendor::f()
delete mp
~Paste1()
~Vendor1()
~Vendor()
~MyBase()
```

Исходные библиотечные функции A() и B() работают так же, как раньше (предполагается, что новая версия v() вызывает свой прототип из базового класса). Деструктор стал виртуальным и работает так, как положено.

Хотя данная методика не отличается изяществом, она часто применяется на практике. В частности, она демонстрирует одно из предусловий множественного наследования: необходимость повышающего преобразования к обоим базовым классам.

Итоги

Одна из причин, по которым в C++ поддерживается множественное наследование, состоит в том, что C++ является гибридным языком и не может использовать единую монолитную иерархию классов, как Smalltalk или Java. Вместо этого C++ позволяет создавать множество самостоятельных деревьев наследования, поэтому иногда требуется объединить интерфейсы двух и более деревьев в новый класс.

Если иерархия не содержит «ромбов», множественное наследование обходится без особых сложностей (хотя вам все равно приходится решать проблему идентичных сигнатур функций в базовых классах). При появлении ромбовидных структур желательно избавиться от дублирования подобъектов за счет введения виртуальных базовых классов. Впрочем, это не только запутывает программу, но и усложняет базовую реализацию и снижает ее эффективность.

Множественное наследование называют «командой goto 90-х годов[1]». Сравнение весьма удачное: множественного наследования, как и команды goto, при нормальном программировании лучше избегать, но в отдельных случаях оно оказывается очень полезным. Оно принадлежит к числу «второстепенных», но крайне нетривиальных возможностей C++, предназначенных для решения проблем, возникающих в особых ситуациях. Если вы часто пользуетесь множественным наследованием, лишний раз проанализируйте свою мотивацию. Спросите себя: «Нужно ли мне выполнять повышающее преобразование ко всем базовым классам?» И если не нужно, вы основательно упростите себе жизнь, используя вложенные экземпляры всех классов, к которым повышающее преобразование выполнять *не нужно*.

Упражнения

1. Создайте базовый класс X с единственным конструктором, получающим аргумент int, и функцией f(), которая вызывается без аргументов и возвращает void. Теперь определите от X производные классы Y и Z, каждый из которых содержит конструктор с одним аргументом int. Создайте класс A, производный от Y и Z. Создайте объект класса A и вызовите f() для этого объекта. Решите проблему посредством уточнения.
2. Начните с результата упражнения 1. Создайте указатель на X с именем px и присвойте ему адрес созданного ранее объекта типа A. Решите проблему, используя виртуальный базовый класс. Теперь исправьте X так, чтобы вам не приходилось вызывать конструктор X внутри A.
3. Начните с результата упражнения 2. Удалите уточнение вызова f() и посмотрите, удастся ли вызвать f() через px. Посредством трассировки выясните, какая функция вызывается. Решите проблему так, чтобы в иерархии классов вызывалась правильная функция.
4. Создайте интерфейсный класс Animal с объявлением функции makeNoise(). Создайте интерфейсный класс SuperHero с объявлением функции savePersonFromFire(). Включите в оба интерфейсных класса объявление функции move() (не забудьте объявить методы интерфейсных классов чисто виртуальными). Определите три разных класса: SuperlativeMan, Amoeba и TarantulaWoman. SuperlativeMan реализует интерфейс SuperHero, тогда как Amoeba и TarantulaWoman реализуют интерфейсы Animal и SuperHero. Определите две глобальные функции animalSound(Animal*) и saveFromFire(SuperHero*). Вызовите в обеих функциях все доступные методы интерфейсов.

[1] Выражение принадлежит Заку Урлокеру (Zack Urlocker).

5. Повторите предыдущее упражнение, но используйте для реализации интерфейсов шаблоны вместо наследования, как в примере Interfaces2.cpp.
6. Определите подключаемые классы, реализующие дополнительные возможности SuperHero (например, StopTrain, BendSteel, ClimbBuilding и т. д.). Переделайте упражнение 4 так, чтобы производные классы SuperHero наследовали от этих подключаемых классов и вызывали их функции.
7. Повторите предыдущее упражнение, преобразовав подключаемые классы в параметры шаблонов. Продемонстрируйте вызов этих функций.
8. Исключив интерфейс Animal из упражнения 4, переопределите класс Amoeba так, чтобы он реализовывал только SuperHero. Определите класс SuperlativeAmoeba, наследующий от SuperlativeMan и Amoeba. Попробуйте передать объект Amoeba функции saveFromFire(). Что нужно сделать, чтобы это стало возможным? Как виртуальное наследование влияет на размеры объектов?
9. В продолжение предыдущего упражнения добавьте целочисленную переменную strengthFactor в класс SuperHero из упражнения 4, а также конструктор для инициализации этой переменной. Еще для ее инициализации включите конструкторы в три производных класса. Что необходимо сделать по-другому в SuperlativeAmoeba?
10. В продолжение предыдущего примера включите функцию eatFood() в SuperlativeMan и Amoeba (но не в SuperlativeAmoeba!), чтобы две версии eatFood() получали разные типы объектов, а их сигнатуры различались. Что нужно сделать в SuperlativeAmoeba для вызова любой из функций eatFood? Почему?
11. Определите для SuperlativeAmoeba операторы << и =.
12. Удалите SuperlativeAmoeba из иерархии и измените класс Amoeba так, чтобы он был производным как от класса SuperlativeMan (который по-прежнему наследует от SuperHero), так и от класса SuperHero. Реализуйте виртуальную функцию workout() в SuperHero и SuperlativeMan() (с идентичными сигнатурами) и вызовите ее для объекта Amoeba. Какая функция будет вызвана?
13. Переопределите класс SuperlativeAmoeba так, чтобы его интерпретация в качестве SuperlativeMan или Amoeba осуществлялась посредством композиции вместо наследования. Обеспечьте выполнение неявного повышения при помощи операторов преобразования. Сравните полученное решение с решением на базе наследования.
14. Допустим, вы получили предварительно откомпилированный класс Person (в вашем распоряжении только заголовочный файл и откомпилированный объектный файл). Пусть SuperHero содержит невиртуальную функцию work(). Сделайте так, чтобы класс SuperHero использовал версию work() класса Person. Для этого определите класс производным от Person и используйте реализацию Person::work(), но сделайте функцию SuperHero::work() виртуальной.
15. Определите подключаемый класс для регистрации ошибок ErrorLog, работающий на основе подсчета ссылок. Класс содержит статический файло-

вый поток, в который отправляются сообщения. Поток открывается, если счетчик ссылок положителен, и закрывается, когда счетчик снова падает до нуля (данные всегда дописываются в конец файла). Организуйте отправку сообщений в статический поток со стороны объектов нескольких классов. Проследите за открытием и закрытием потока, включив в ErrorLog необходимые команды трассировки.

16. Измените программу BreakTie.cpp, добавив в нее класс VeryBottom, наследующий (не виртуально!) от Bottom. Класс VeryBottom почти не отличается от Bottom, кроме одного: в объявлении функции f слово «Left» заменяется словом «Right». Измените функцию main() так, чтобы вместо объекта Bottom в ней создавался экземпляр VeryBottom. Какая версия f() будет вызываться?

Паттерны проектирования 10

...Опишите задачу, которая снова и снова встречается в вашей работе. А затем изложите суть ее решения этой задачи так, чтобы его можно было использовать миллион раз, но все решения были бы разными.

Кристофер Александер (Christopher Alexander)

В этой главе представлен важный, но еще не получивший широкого распространения подход к проектированию программ, основанный на использовании паттернов.

Вероятно, самым важным недавним достижением в области объектно-ориентированного проектирования является методология *паттернов проектирования* (от англ. *pattern*). Впервые она была представлена в книге «Паттерны проектирования»[1] (Гамма, Хелм, Джонсон и Влиссидес[2], изд-во «Питер», 2003 год), демонстрирующей 23 типовых решения для определенных классов задач. В этой главе рассматриваются основные принципы использования паттернов, которые поясняются примерами. Надеемся, это пробудит в вас интерес к дальнейшему изучению паттернов — важнейшего, почти необходимого инструмента объектно-ориентированного программирования.

Концепция паттернов

Поначалу паттерны можно было рассматривать как особенно изящный и хорошо продуманный способ решения определенного класса задач. Все выглядит так, будто целая группа аналитиков изучила все потенциальные трудности и выработала наиболее общее, гибкое решение для данного типа задач. Возможно, вам уже приходилось видеть и решать эти задачи раньше, но вряд ли ваше решение обладало той полнотой, которая воплощена в паттерне. Более того, паттерны не зависят от конкретных применений и могут реализовываться разными способами.

Несмотря на название «паттерны проектирования», в действительности паттерны не ограничиваются областью проектирования. Похоже, паттерны отходят

---

[1] Примеры в книге написаны на C++. К сожалению, в них используется устаревший диалект языка (предшествовавший стандартному языку C++), в котором не поддерживаются некоторые современные механизмы (в частности, контейнеры STL).

[2] В народе именуется книгой «банды четырех» (Gang of Four, GoF), далее — БЧ.

от традиционных представлений об анализе, проектировании и реализации. Паттерны воплощают некую идею, поэтому они часто встречаются в фазах анализа и высокоуровневого проектирования. Но так как паттерны обычно имеют прямую реализацию в виде программного кода, иногда они проявляются лишь на фазе низкоуровневого проектирования или реализации (а нередко даже сама необходимость применения того или иного паттерна становится очевидной лишь в этих фазах).

Основной принцип паттерна — введение новых уровней абстракции — также может рассматриваться как основной принцип проектирования программ вообще. Любая абстракция означает исключение частных, второстепенных деталей, а это обычно делается для того, чтобы отделить переменные составляющие проблемы от постоянных составляющих. Иначе говоря, если некая часть программы с большой вероятностью будет подвержена изменениям, эти изменения не должны приводить к распространению вместе с вашей программой побочных эффектов. Если вам удастся решить эту задачу, то упростится не только чтение и понимание, но и сопровождение программы — что со временем неизбежно приведет к снижению затрат.

Основные трудности при разработке элегантной и удобной в сопровождении архитектуры обычно возникают при идентификации так называемого «вектора изменений» (в данном случае термин «вектор» понимается в классическом смысле как направленный отрезок, а не как разновидность контейнера). Для этого нужно выявить важнейший фактор перемен в вашей системе. Выявление вектора изменений дает опорную точку для построения дальнейшей архитектуры.

Итак, паттерны предназначены прежде всего для *инкапсуляции изменений*. Если рассматривать их подобным образом, некоторые паттерны уже встречались вам в этой книге. Например, наследование тоже может рассматриваться как паттерн (пусть даже реализованный на уровне компилятора). Оно выражает различия в поведении (переменная составляющая) объектов, обладающих одинаковым интерфейсом (постоянная составляющая). Композиция также может рассматриваться как паттерн, потому что вы можете изменять (динамически или статически) объекты, участвующие в реализации класса. Впрочем, обычно возможности, напрямую поддерживаемые языком программирования, не принято относить к паттернам проектирования.

Нам также неоднократно встречался другой паттерн, упоминаемый в БЧ: *итератор*. Итераторы занимают центральное место в архитектуре STL. Они скрывают от пользователя конкретную реализацию контейнера в процессе перебора и последовательной выборки элементов. Итераторы позволяют написать обобщенный код, выполняющий операции со всеми элементами интервала без учета специфики контейнера, в котором этот интервал находится. Такой обобщенный код может использоваться с любым контейнером, поддерживающим итераторы.

Одним из самых фундаментальных принципов в БЧ является не паттерн, а аксиома, сформулированная в главе 1: «Отдавайте предпочтение композиции объектов перед наследованием классов». Хорошее понимание наследования и полиморфизма требует столь значительных усилий, что программисты склонны преувеличивать значение этих методик. Мы видели много излишне усложненных архитектур (включая и наши собственные разработки), возникающих в результате «тяги к наследованию». В частности, тяга к использованию наследования везде, где это возможно, породило много архитектур с множественным наследованием.

Один из канонов экстремального программирования (XP) гласит: «Выберите самое простое решение, которое может работать». Иногда композиция радикально упрощает архитектуры, в которых кажется уместным наследование, а полученная архитектура становится более гибкой; вы убедитесь в этом, изучая некоторые из представленных паттернов. Итак, продумывая новую архитектуру, спросите себя: «Так ли здесь необходимо наследование, что оно мне дает? Не упростится ли решение при использовании композиции?»

Классификация паттернов

В БЧ описаны 23 паттерна, разделенных на три категории (в соответствии с выбором переменной составляющей).

- *Паттерны создания объектов.* Паттерны этой категории часто связаны с инкапсуляцией технических подробностей создания объекта, чтобы код не приходилось изменять при добавлении новых типов объектов. Из этой категории будут представлены паттерны Синглет, Фабрика и Строитель.
- *Структурные паттерны.* Паттерны этой категории относятся к взаимодействиям между объектами, чтобы при изменениях в системе не приходилось модифицировать связи между объектами. Применение структурных паттернов часто обусловлено ограничениями проекта. Примеры — Посредник и Адаптер.
- *Поведенческие паттерны.* Объекты, выполняющие некоторые типы действий внутри программы. В них инкапсулируются такие процессы, как интерпретация языка, выполнение запроса, перебор элементов (как при использовании итераторов) или реализация алгоритма. В этой главе будут представлены поведенческие паттерны Команда, Шаблонный метод, Состояние, Стратегия, Цепочка ответственности, Наблюдатель, Множественная диспетчеризация и Посетитель.

Каждому из 23-х паттернов в БЧ посвящен специальный раздел с одним или несколькими примерами. Большинство примеров написано на C++, но иногда также встречается Smalltalk. Мы не будем повторять все подробности, приводимые в БЧ — эта книга заслуживает того, чтобы прочитать ее отдельно. Описания и примеры этой главы всего лишь дают начальное представление об паттернах, чтобы вы поняли, что такое паттерны и почему они так важны.

На издании книги БЧ работа над паттернами не закончилась. С момента публикации появились новые паттерны и новый усовершенствованный процесс определения паттернов проектирования[1]. Последнее тоже важно, потому что идентификация новых паттернов и их должное описание связаны с определенными трудностями. Например, в популярной литературе даже нет единого мнения относительного того, что же такое паттерн. Паттерны нетривиальны, и обычно они не имеют прямого представления во встроенных возможностях языков программирования. Например, конструкторы и деструкторы можно назвать «паттернами гарантированной инициализации и зачистки»; это важные и абсолютно необходимые конструкции, однако они принадлежат к числу стандартных языковых средств и недостаточно содержательны для того, чтобы считаться паттернами.

[1] За последней информацией обращайтесь по адресу http://hillside.net/patterns.

Другой характерный пример — различные формы агрегирования. Агрегирование, то есть применение объектов для построения других объектов, является одним из краеугольных камней объектно-ориентированного программирования. Тем не менее эту концепцию иногда ошибочно классифицируют как паттерн. Такие ошибки весьма прискорбны — они искажают представления об паттернах проектирования и создают впечатление, будто любая концепция, удивившая вас при первом знакомстве, является паттерном.

Еще один неудачный пример встречается в языке Java: разработчики спецификации JavaBeans решили назвать простую схему выбора имен get/set паттерном проектирования (например, метод getInfo() возвращает свойство Info, а setInfo() изменяет его). Но речь идет всего лишь о распространенной схеме формирования имен, которая ни в коем случае не является паттерном.

Упрощение идиом

Прежде чем браться за более сложные вопросы, стоит рассмотреть некоторые базовые приемы, которые делают программу более простой и прямолинейной.

Посыльный

Самым тривиальным из этих приемов является прием, воплощенный в паттерне *Посыльный* (Messenger)[1]; вместо того, чтобы передавать разнородную информацию по частям, вы упаковываете ее в объекте. Так, в следующем примере применения Посыльного существенно упрощает код функции translate():

```
//: C10:MessengerDemo.cpp
#include <iostream>
#include <string>
using namespace std;

class Point { // Посыльный
public:
  int x, y, z;
  Point(int xi, int yi, int zi) : x(xi), y(yi), z(zi) {}
  Point(const Point& p) : x(p.x), y(p.y), z(p.z) {}
  Point& operator=(const Point& rhs) {
    x = rhs.x;
    y = rhs.y;
    z = rhs.z;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const Point& p) {
    return os << "x=" << p.x << " y =" << p.y
              << " z=" << p.z;
  }
};

class Vector { // Математический вектор
public:
```

[1] Название предложено Биллом Веннерсом (Bill Venners); в других местах также встречаются другие названия.

```
  int magnitude, direction;
  Vector(int m, int d) : magnitude(m), direction(d) {}
};

class Space {
public:
  static Point translate(Point p, vector v) {
    // Копирующий конструктор предотвращает
    // модификацию оригинала.
    // Фиктивные вычисления:
    p.x += v.magnitude + v.direction;
    p.y += v.magnitude + v.direction;
    p.z += v.magnitude + v.direction;
    return p;
  }
};

int main() {
  Point p1(1, 2, 3);
  Point p2 = Space::translate(p1, Vector(11, 47));
  cout << "p1: " << p1 << " p2: " << p2 << endl;
} ///:~
```

Пример намеренно упрощен, чтобы не отвлекать читателя от содержательной стороны дела.

Так как единственной задачей Посыльного является передача данных, ради упрощения доступа эти данные объявлены открытыми. Возможно, у вас найдутся причины для того, чтобы ограничить доступ к данным.

Накопитель

Накопитель (Collecting parameter) является «старшим братом» Посыльного. Он предназначен для сохранения информации из функции, которой он передается при вызове. Как правило, Накопитель поочередно передается нескольким функциям; он напоминает пчелу, которая перелетает с цветка на цветок и собирает пыльцу.

Контейнеры особенно хорошо подходят на роль Накопителей, поскольку они изначально обладают способностью к динамическому включению объектов:

```
//: C10:CollectingParameterDemo.cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class CollectingParameter : public vector<string> {};

class Filler {
public:
  void f(CollectingParameter& cp) {
    cp.push_back("accumulating");
  }
  void g(CollectingParameter& cp) {
    cp.push_back("items");
  }
  void f(CollectingParameter& cp) {
    cp.push_back("as we go");
  }
```

```
};

int main() {
  Filler filler;
  CollectingParameter cp;
  filler.f(cp);
  filler.g(cp);
  filler.h(cp);
  vector<string>::iterator it = cp.begin();
  while(it != cp.end())
    cout << *it++ << " ";
  cout << endl;
} ///:~
```

Накопитель должен обладать средствами сохранения или вставки новых значений (кстати, Посыльный по определению может использоваться в качестве Накопителя). Принципиально здесь то, что Накопитель поочередно передается нескольким функциям и модифицируется ими.

Синглет и его разновидности

Вероятно, простейшим паттерном в книге БЧ является *Синглет*, гарантирующий существование единственного экземпляра класса. Следующая программа показывает, как реализовать Синглет на C++:

```
//: C10:SingletonPattern.cpp
#include <iostream>
using namespace std;

class Singleton {
  static Singleton s;
  int i;
  Singleton(int x) : i(x) {}
  Singleton& operator=(Singleton&);  // Запрещено
  Singleton(const Singleton&);       // Запрещено
public:
  static Singleton& instance() { return s; }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

Singleton Singleton::s(47);

int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

Ключевой принцип создания Синглета состоит в том, что прикладной программист никак не может повлиять на жизненный цикл объекта. Для этого следует объявить все конструкторы закрытыми и предотвратить любые попытки компилятора генерировать конструктор. Обратите внимание: копирующий конструктор и оператор присваивания (для которых реализация специально не определяется, пото-

му что они никогда не вызываются) объявляются закрытыми. Тем самым предотвращаются любые попытки создания копий.

Вы также должны выбрать способ создания объекта. В нашем случае объект создается статически, но возможно и другое решение: подождать, пока прикладной программист запросит объект, и создать его по требованию. Такая методика называется *отложенной инициализацией* и имеет смысл только в том случае, если создание объекта обходится дорого и объект используется не всегда.

Если вместо ссылки вернуть указатель, пользователь может случайно вызвать для него оператор delete, поэтому приведенная ранее реализация считается наиболее безопасной (проблема также решается объявлением деструктора закрытым или защищенным). В любом случае данные объекта должны быть закрытыми.

Доступ к данным осуществляется через открытые функции класса. В нашем примере функция instance() возвращает ссылку на объект Singleton. Остальные функции (getValue() и setValue()) образуют привычный интерфейс доступа.

В принципе эта методика не ограничивается лишь созданием одного объекта — она также позволяет создать ограниченный пул объектов, но тогда вы столкнетесь с проблемой организации совместного использования объектов в пуле. В этом случае можно создать решение с входным и выходным контролем общих объектов.

Любой вложенный статический объект внутри класса выражает концепцию Синглета: он существует в одном и только одном экземпляре. В каком-то смысле язык обеспечивает прямую поддержку «синглетности», которую мы регулярно применяем. Но при использовании статических объектов — как членов класса, так и внешних — возникает одна проблема, связанная с порядком инициализации (эта тема рассматривалась в первом томе). Если один статический объект зависит от другого, очень важно, чтобы эти объекты инициализировались в правильном порядке.

В первом томе было показано, как управлять порядком инициализации, определяя статический объект в функции. При этом инициализация объекта откладывается до первого вызова функции. Если функция возвращает ссылку на статический объект, фактически получается тот же Синглет, но без хлопот с инициализацией статических объектов. Допустим, вы хотите создать файл журнала при первом вызове функции, возвращающей ссылку на этот файл. Задача решается следующим заголовочным файлом:

```
//: C10:LogFile.h
#ifndef LOGFILE_H
#define LOGFILE_H
#include <fstream>
std::ofstream& logfile();
#endif // LOGFILE_H ///:~
```

Реализация *не должна быть подставляемой*, потому что это означало бы, что вся функция вместе с определением статического объекта дублируется во всех единицах трансляции, в которые она включается, а это привело бы к нарушению правила единственного определения C++[1]. Все попытки управлять порядком инициализации наверняка кончатся провалом (и скорее всего, самым неочевидным и с трудноуловимыми причинами). Итак, реализация должна быть отдельной:

[1] Стандарт C++ гласит: «Единица трансляции не должна содержать более одного определения любой переменной, функции, класса, перечисляемого типа или шаблона... Каждая программа содержит ровно одно определение каждой неподставляемой функции или объекта, используемого в этой программе».

```
//: C10:LogFile.cpp {O}
#include "LogFile.h"
std::ofstream& logfile() {
  static std::ofstream log("Logfile.log");
  return log;
} ///:~
```

Теперь инициализация объекта log откладывается до первого вызова logfile(). Допустим, вы создали функцию:

```
//: C10:UseLog1.h
#ifndef USELOG1_H
#define USELOG1_H
void f();
#endif // USELOG1_H ///:~
```

В реализации этой функции имеется вызов logfile():

```
//: C10:UseLog1.cpp {O}
#include "UseLog1.h"
#include "LogFile.h"
void f() {
  logfile() << __FILE__ << std::endl;
} ///:~
```

Если повторно вызвать функцию logfile() в другом файле, как показано ниже, то объект log создается лишь после первого вызова f():

```
//: C10:UseLog2.cpp
//{L} LogFile UseLog1
#include "UseLog1.h"
#include "LogFile.h"
using namespace std;
void g() {
  logfile() << __FILE__ << endl;
}

int main() {
  f();
  g();
} ///:~
```

Создание статического объекта внутри функции класса легко объединяется с синглетным классом. Ниже приводится новая версия программы SingletonPattern.cpp с применением этой методики[1]:

```
//: C10:SingletonPattern2.cpp
// Синглет Мейерса
#include <iostream>
using namespace std;

class Singleton {
  int i;
  Singleton(int x) : i(x) { }
  void operator=(Singleton&);
  Singleton(const Singleton&);
public:
  static Singleton& instance() {
    static Singleton s(47);
    return s;
```

[1] Также называемой «Синглетом Мейерса» по имени автора Скотта Мейерса (Scott Meyers).

```
  }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

Особенно интересная ситуация возникает тогда, когда два Синглета зависят друг от друга:

```
//: C10:FunctionStaticSingleton.cpp

class Singleton1 {
  Singleton1() {}
public:
  static Singleton1& ref() {
    static Singleton1 single;
    return single;
  }
};

class Singleton2 {
  Singleton1& s1;
  Singleton2(Singleton1& s) : s1(s) {}
public:
  static Singleton2& ref() {
    static Singleton2 single(Singleton1::ref());
    return single;
  }
  Singleton1& f() { return s1; }
};

int main() {
  Singleton1& s1 = Singleton2::ref().f();
} ///:~
```

Вызов Singleton2::ref() приводит к созданию синглетного объекта Singleton2. В процессе создания вызывается функция Singleton1::ref(), а это требует создания синглетного объекта Singleton1. Такая методика не зависит от порядка компоновки или загрузки, поэтому программист гораздо лучше контролирует процесс инициализации, что уменьшает количество проблем.

В другой разновидности данного паттерна «синглетность» объекта отделяется от его реализации. Для решения этой задачи используется механизм псевдорекурсии, упоминавшийся в главе 5:

```
//: C10:CuriousSingleton.cpp
// Отделение класса от его "синглетности" (почти)
#include <iostream>
using namespace std;

template<class T>
class Singleton {
  Singleton(const Singleton&);
```

```
  Singleton& operator=(const Singleton&);
protected:
  Singleton() {}
  virtual ~Singleton() {}
public:
  static T& instance() {
    static T theInstance;
    return theInstance;
  }
};

// Пример класса, преобразуемого в Синглет
class MyClass : public Singleton<MyClass> {
  int x;
protected:
  friend class Singleton<MyClass>;
  MyClass(){ x = 0; }
public:
  void setValue(int n) { x = n; }
  int getValue() const { return x; }
};

int main() {
  MyClass& m = MyClass::instance();
  cout << m.getValue() << endl;
  m.setValue(1);
  cout << m.getValue() << endl;
} ///:~
```

Чтобы сделать класс MyClass синглетным, мы решили следующие задачи.

1. Объявили его конструктор закрытым или защищенным.
2. Объявили класс Singleton<MyClass> дружественным.
3. Объявили класс MyClass производным от Singleton<MyClass>.

Рекурсия на шаге 3 кажется невозможной, но как объяснялось в главе 5, она работает, поскольку в шаблоне Singleton существует только статическая зависимость от аргумента. Иначе говоря, компилятор может генерировать код Singleton<MyClass>, потому что он не зависит от размера MyClass. Информация о размере MyClass становится необходимой только позднее, при первом вызове функции Singleton<MyClass>::instance(), но к этому времени класс MyClass уже откомпилирован, а его размер известен.

Интересно, насколько нетривиальным может оказаться даже такой простой паттерн, как Синглет — а ведь мы даже не начинали рассматривать вопросы безопасности программных потоков. И последнее. Синглет следует применять осмотрительно. Действительно синглетные объекты встречаются редко, а задействовать Синглет для замены глобальной переменной — последнее дело.

Команда

Со структурной точки зрения паттерн *Команда* (Command) очень прост, однако он помогает ослабить привязку компонентов программы, а следовательно, делает программу более понятной.

В своей книге «Advanced C++: Programming Styles and Idioms» (Addison Wesley, 1992) Джим Коплин (Jim Coplien) вводит термин *функтор* — объект, предназначенный только для инкапсуляции функции (поскольку этот термин также встречается в математике, мы остановимся на более однозначном термине *объект функции*). Этот объект нужен, чтобы отделить выбор вызываемой функции от места ее вызова.

В книге БЧ объект функции упоминается, но не используется. С другой стороны, концепция объектов функций повторяется в нескольких рассматриваемых паттернах.

Команда представляет собой объект функции в его изначальном смысле — это функция, оформленная в виде объекта. Инкапсулируя функцию в объекте, можно передать ее другим функциям или объектам в качестве параметра, приказывая им выполнить конкретную операцию в процессе обработки запроса. Можно сказать, что Команда является частным случаем Посыльного, в котором передаются не данные, а операции.

```
//: C10:CommandPattern.cpp
#include <iostream>
#include <vector>
using namespace std;

class Command {
public:
  virtual void execute() = 0;
};

class Hello : public Command {
public:
  void execute() { cout << "Hello "; }
};

class World : public Command {
public:
  void execute() { cout << "World! "; }
};

class IAm : public Command {
public:
  void execute() { cout << "I'm the command pattern!"; }
};

// Объект для хранения набора команд:
class Macro {
  vector<Command*> commands;
public:
  void add(Command* c) { commands.push_back(c); }
  void run() {
    vector<Command*>::iterator it = commands.begin();
    while(it != commands.end())
      (*it++)->execute();
  }
};

int main() {
  Macro macro;
  macro.add(new Hello);
```

```
    macro.add(new World);
    macro.add(new IAm);
    macro.run();
  } ///:~
```

Команда предназначена прежде всего для передачи нужного действия функции или объекту. В приведенном примере это позволяет создать очередь операций для последующего выполнения. Иначе говоря, поведение программы формируется динамически — обычно для этого приходится писать новый код, но в данном случае задача решается простой интерпретацией сценария, хотя при выполнении очень сложных операций лучше воспользоваться паттерном Интерпретатор (Interpreter).

В БЧ сказано: «Команды представляют собой объектно-ориентированную замену для функций обратного вызова». Однако мы считаем, что в концепции функций обратного вызова слово «обратный» играет очень важную роль — в будущем управление *возвращается* стороне, создавшей функцию обратного вызова. С другой стороны, объект Команды просто создается и передается некоторой функции или объекту, и в дальнейшем вы с ним никак не связаны.

Стандартный пример применения Команды — реализация «отката» в приложениях. Каждый раз, когда пользователь выполняет какую-нибудь операцию, в очередь отката заносится соответствующий объект Команды. Выполнение этого объекта возвращает состояние программы на один шаг назад.

Команда и смягчение привязки при обработке событий

Как вы узнаете из следующей главы, одна из причин для реализации концепции *многопоточности* заключается в том, что она упрощает *событийное программирование*[1], когда поведение программы определяется непредсказуемо возникающими событиями. Например, если пользователь нажимает кнопку Quit во время выполнения продолжительной операции, он рассчитывает, что программа достаточно быстро отреагирует на его действие.

В пользу многопоточности говорит и то, что многопоточность смягчает жесткую привязку между фрагментами программы. Другими словами, если вы запустите отдельный программный поток для отслеживания событий на кнопке Quit, то «обычным» операциям ничего не нужно знать об этой кнопке или других выполняемых операциях.

Проблемы с жесткой привязкой также можно решить при помощи паттерна Команда. Любая «обычная» операция должна периодически вызывать функцию проверки состояния событий, в то же время при использовании паттерна Команда о проверяемых условиях ей знать ничего не нужно; тем самым операция перестает жестко привязываться к коду обработки событий:

```
//: C10:MulticastCommand.cpp {RunByHand}
// Смягчение привязки к коду обработки событий
// с применением паттерна Команда.
#include <iostream>
#include <vector>
#include <string>
#include <ctime>
```

[1] Также встречается термин «программирование, управляемое событиями». — *Примеч. перев.*

```
#include <cstdlib>
using namespace std;

// Иерархия классов для выполнения задач:
class Task {
public:
  virtual void operation() = 0;
};

class TaskRunner {
  static vector<Task*> tasks;
  TaskRunner() {} // Синглетный класс
  TaskRunner& operator=(TaskRunner&); // Запрет
  TaskRunner(const TaskRunner&); // Запрет
  static TaskRunner tr;
public:
  static void add(Task& t) { tasks.push_back(&t); }
  static void run() {
    vector<Task*>::iterator it = tasks.begin();
    while(it != tasks.end())
      (*it++)->operation();
  }
};

TaskRunner TaskRunner::tr;
vector<Task*> TaskRunner::tasks;

class EventSimulator {
  clock_t creation;
  clock_t delay;
public:
  EventSimulator() : creation(clock()) {
    delay = CLOCKS_PER_SEC/4 * (rand() % 20 + 1);
    cout << "delay = " << delay << endl;
  }
  bool fired() {
    return clock() > creation + delay;
  }
};

// Источник асинхронных событий:
class Button {
  bool pressed;
  string id;
  EventSimulator e; // Для демонстрации
public:
  Button(string name) : pressed(false), id(name) {}
  void press() { pressed = true; }
  bool isPressed() {
    if(e.fired()) press(); // Имитация события
    return pressed;
  }
  friend ostream&
  operator<<(ostream& os, const Button& b) {
    return os << b.id;
  }
};

// Объект Команды
```

```
class CheckButton : public Task {
  Button& button;
  bool handled;
public:
  CheckButton(Button & b) : button(b), handled(false) {}
  void operation() {
    if(button.isPressed() && !handled) {
      cout << button << " pressed" << endl;
      handled = true;
    }
  }
};

// Процедуры, выполняющие основные вычисления. Должны
// периодически прерываться для проверки состояния
// кнопок или других событий:
void procedure1() {
  // Выполнение операций procedure1.
  // ...
  TaskRunner::run(); // Проверка всех событий
}

void procedure2() {
  // Выполнение операций procedure2.
  // ...
  TaskRunner::run(); // Проверка всех событий
}

void procedure3() {
  // Выполнение операций procedure3.
  // ...
  TaskRunner::run(); // Проверка всех событий
}

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  Button b1("Button 1"), b2("Button 2"), b3("Button 3");
  CheckButton cb1(b1), cb2(b2), cb3(b3);
  TaskRunner::add(cb1);
  TaskRunner::add(cb2);
  TaskRunner::add(cb3);
  cout << "Control-C to exit" << endl;
  while(true) {
    procedure1();
    procedure2();
    procedure3();
  }
} ///:~
```

В этом примере Команды, представленные объектами классом Task, выполняются синглетным объектом TaskRunner. Объект EventSimulator имитирует случайные внешние события: при периодическом вызове fired() результат через случайный промежуток времени изменяется с false на true. Объекты EventSimulator требуются объектам Button для имитации действий пользователя, происходящих с непредсказуемыми интервалами. Класс CheckButton представляет собой реализацию класса Task, которая периодически проверяется всем «обычным» кодом програм-

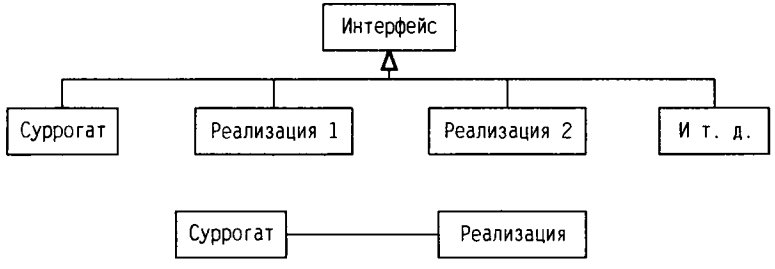
мы — соответствующие вызовы находятся в конце процедур procedure1, procedure2 и procedure3.

Конечно, при реализации этого решения приходится немного потрудиться. Но как будет показано в главе 11, для предотвращения проблем, возникающих при многопоточном программировании, необходимо действовать *крайне* продуманно и внимательно, поэтому более простое решение может оказаться предпочтительным. Также можно организовать очень простую многопоточную схему, переместив вызовы TaskRunner::run() в отдельный программный поток таймера. В этом случае полностью устраняется всякая привязка «обычных операций» к коду обработки событий.

Суррогатные объекты

Паттерны Посредник (Proxy) и Состояние (State) основаны на применении *суррогатных классов*. Ваша программа взаимодействует с суррогатным классом, за которым скрывается настоящий класс, выполняющий реальную работу. При вызове функции суррогатный класс просто переадресует вызов соответствующей функции класса реализации. Эти два паттерна имеют так много общего, что со структурной точки зрения Посредник может считаться частным случаем Состояния. Возможно, вам даже захочется объединить их в новый паттерн Суррогат, но эти паттерны различаются по своему *предназначению*, то есть смысловому наполнению. Многие ошибочно полагают, что если два паттерна имеют сходную структуру, то и различать их не нужно. Чтобы понять, когда применяется тот или иной паттерн, всегда следует разобраться в его предназначении.

Основная идея проста: суррогатный класс объявляется производным от базового класса. Вместе с ним от этого же базового класса порождаются другие классы, предоставляющие фактическую реализацию:



При создании суррогатного объекта задается реализация, которой будут передаваться вызовы функций.

Со структурных позиций отличия между Посредником и Состоянием просты: Посредник имеет только одну реализацию, тогда как у Состояния их может быть несколько. Соответственно различаются и области применения этих паттернов (по БЧ): Посредник обеспечивает контроль за доступом к реализации, а Состояние позволяет динамически выбирать реализацию. Впрочем, если понимать смысл выражения «управление доступом к реализации» более широко, различия между этими паттернами стираются.

Посредник

Реализация Посредника для приведенной выше диаграммы выглядит так:

```
//: C10:ProxyDemo.cpp
// Простая демонстрация паттерна Посредник
#include <iostream>
using namespace std;

class ProxyBase {
public:
  virtual void f() = 0;
  virtual void g() = 0;
  virtual void h() = 0;
  virtual ~ProxyBase() {}
};

class Implementation : public ProxyBase {
public:
  void f() { cout << "Implementation.f()" << endl; }
  void g() { cout << "Implementation.g()" << endl; }
  void h() { cout << "Implementation.h()" << endl; }
};

class Proxy : public ProxyBase {
  ProxyBase* implementation;
public:
  Proxy() { implementation = new Implementation(); }
  ~Proxy() { delete implementation; }
  // Передача вызовов реализации:
  void f() { implementation->f(); }
  void g() { implementation->g(); }
  void h() { implementation->h(); }
};

int main()  {
  Proxy p;
  p.f();
  p.g();
  p.h();
} ///:~
```

В некоторых случаях класс Implementation даже не обязан иметь одинаковый интерфейс с классом Proxy — если Proxy умеет взаимодействовать с классом Implementation и передавать ему вызовы функции, основной принцип успешно работает (обратите внимание: данное утверждение противоречит определению Посредника по БЧ). Тем не менее, наличие общего интерфейса позволяет произвести прямую замену исходного объекта объектом Посредника в клиентском коде: клиентский код написан для взаимодействия с исходным объектом, и его не придется изменять для работы с Посредником (вероятно, это один из важнейших аспектов применения Посредника). Кроме того, наличие общего интерфейса гарантирует, что Implementation содержит все функции, вызываемые Proxy.

Различия между Посредником и Состоянием определяются кругом задач, решаемых при помощи этих паттернов. Основные области применениями Посредника перечислены далее (по БЧ).

- *Удаленный посредник.* Посредник представляет объект, находящийся в другом адресном пространстве (в реализации используются удаленные объектные технологии).
- *Виртуальный посредник.* Обеспечивает отложенную инициализацию, чтобы дорогостоящие операции создания объектов выполнялись по мере необходимости.
- *Защитный посредник.* Применяется в тех случаях, когда вы не хотите предоставлять прикладному программисту полный доступ к объекту реализации.
- *Умные ссылки.* Выполнение дополнительных действий при обращении к объекту реализации. Примером служит *подсчет ссылок*, то есть отслеживание количества ссылок на объект (используется в реализации идиомы *копирования при записи*). Более простой пример — подсчет вызовов определенной функции.

Состояние

Объект, созданный с применением паттерна Состояние, словно меняет свой класс в процессе работы. Если многие или все функции класса содержат условный код, стоит подумать о применении этого паттерна. Как и в случае с Посредником, интерфейсный объект использует служебный объект реализации для выполнения своих функций. Однако паттерн Состояние позволяет переключаться на другие реализации в течение жизненного цикла интерфейсного объекта, а, следовательно, — изменять поведение объекта при одних и тех же вызовах функций. Такой подход позволяет улучшить структуру кода в тех случаях, когда вы проверяете в каждой функции несколько условий и по результатам проверки решаете, что нужно делать в этой функции. Возьмем классический сюжет о царевне-лягушке: имеется объект (существо), которое по-разному ведет себя в зависимости от текущего состояния. Очевидно, действие можно выбрать в зависимости от состояния логического флага:

```
//: C10:KissingPrincess.cpp
#include <iostream>
using namespace std;

class Creature {
  bool isFrog;
public:
  Creature() : isFrog(true) {}
  void greet() {
    if(isFrog)
      cout << "Ribbet!" << endl;
    else
      cout << "Darling!" << endl;
  }
  void kiss() { isFrog = false; }
};

int main() {
  Creature creature;
  creature.greet();
  creature.kiss();
  creature.greet();
} ///:~
```

Из-за необходимости проверять флаг isFrog в функции greet() и во всех остальных функциях программа становится громоздкой, особенно если позднее потребуется добавить в систему дополнительные состояния. Делегирование операций объекту с переменным состоянием упрощает программу:

```
//: C10:KissingPrincess2.cpp
// Паттерн Состояние.
#include <iostream>
#include <string>
using namespace std;

class Creature {
  class State {
  public:
    virtual string response() = 0;
  };
  class Frog : public State {
  public:
    string response() { return "Ribbet!"; }
  };
  class Prince : public State {
  public:
    string response() { return "Darling!"; }
  };
  State* state;
public:
  Creature() : state(new Frog()) {}
  void greet() {
    cout << state->response() << endl;
  }
  void kiss() {
    delete state;
    state = new Prince();
  }
};

int main() {
  Creature creature;
  creature.greet();
  creature.kiss();
  creature.greet();
} ///:~
```

Объявлять классы реализации вложенными или закрытыми не обязательно, но если это можно сделать, программа становится более стройной.

Обратите внимание: любые изменения в классах State автоматически распространяются в программе; вам уже не придется редактировать все классы, чтобы эти изменения вступили в силу.

Адаптер

Адаптер (Adapter) получает один тип и предоставляет интерфейс к другому типу. Предположим, вы получили две библиотеки или два фрагмента кода, которые используют одинаковые концепции, но выражают их по-разному (то есть обладают разным интерфейсом). Адаптация этих форм выражения друг к другу позволяет быстро придти к готовому решению.

Допустим, имеется класс, генерирующий числа Фибоначчи:

```
//: C10:FibonacciGenerator.h
#ifndef FIBONACCIGENERATOR_H
#define FIBONACCIGENERATOR_H

class FibonacciGenerator {
  int n;
  int val[2];
public:
  FibonacciGenerator() : n(0) { val[0] = val[1] = 0; }
  int operator()() {
    int result = n > 2 ? val[0] + val[1] : n > 0 ? 1 : 0;
    ++n;
    val[0] = val[1];
    val[1] = result;
    return result;
  }
  int count() { return n; }
};
#endif // FIBONACCIGENERATOR_H ///:~
```

Как и всякий генератор, этот класс вызывается оператором ():

```
//: C10:FibonacciGeneratorTest.cpp
#include <iostream>
#include "FibonacciGenerator.h"
using namespace std;

int main() {
  FibonacciGenerator f;
  for(int i =0; i < 20; i++)
    cout << f.count() << ": " << f() << endl;
} ///:~
```

Требуется использовать этот генератор совместно с числовыми алгоритмами STL. К сожалению, алгоритмы STL работают только с итераторами; возникает несоответствие интерфейсов. Проблема решается Адаптером, который получает объект FibonacciGenerator и создает итератор, применяемый алгоритмами STL. Поскольку для работы числовых алгоритмов достаточно итератора ввода, Адаптер получается достаточно прямолинейным (во всяком случае, для программы, генерирующей итераторы STL):

```
//: C10:FibonacciAdapter.cpp
// Адаптация к существующему интерфейсу.
#include <iostream>
#include <numeric>
#include "FibonacciGenerator.h"
#include "../C06/PrintSequence.h"
using namespace std;

class FibonacciAdapter { // Получение итератора
  FibonacciGenerator f;
  int length;
public:
  FibonacciAdapter(int size) : length(size) {}
  class iterator;
  friend class iterator;
  class iterator : public std::iterator<
    std::input_iterator_tag, FibonacciAdapter, ptrdiff_t> {
```

```
    FibonacciAdapter& ap;
  public:
    typedef int value_type;
    iterator(FibonacciAdapter& a) : ap(a) {}
    bool operator==(const iterator&) const {
      return ap.f.count() == ap.length;
    }
    bool operator!=(const iterator& x) const {
      return !(*this == x);
    }
    int operator*() const { return ap.f(); }
    iterator& operator++() { return *this; }
    iterator operator++(int) { return *this; }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this); }
};

int main() {
  const int SZ = 20;
  FibonacciAdapter a1(SZ);
  cout << "accumulate: "
    << accumulate(a1.begin(), a1.end(), 0) << endl;
  FibonacciAdapter a2(SZ), a3(SZ);
  cout << "inner product: "
    << inner_product(a2.begin(), a2.end(), a3.begin(), 0)
    << endl;
  FibonacciAdapter a4(SZ);
  int r1[SZ] = {0};
  int* end = partial_sum(a4.begin(), a4.end(), r1);
  print(r1, end, "partial_sum", " ");
  FibonacciAdapter a5(SZ);
  int r2[SZ] = {0};
  end = adjacent_difference(a5.begin(), a5.end(), r2);
  print(r2, end, "adjacent_difference", " ");
} ///:~
```

Чтобы инициализировать FibonacciAdapter, достаточно задать длину последовательности чисел Фибоначчи. При создании iterator мы сохраняем ссылку на вмещающий объект FibonacciAdapter для последующих обращений к FibonacciGenerator и length. Обратите внимание: при проверке равенства правостороннее значение игнорируется, поскольку существенно лишь то, достиг итератор конца последовательности или нет. Кроме того, operator++() не изменяет итератор; состояние FibonacciAdapter изменяется только одной операцией — вызовом функции operator() для FibonacciGenerator. Нам удается обойтись этой крайне упрощенной версией итератора, потому что для итераторов ввода устанавливаются крайне жесткие ограничения; в частности, каждое значение в последовательности может быть прочитано только один раз.

В функции main() Адаптер FibonacciAdapter используется для тестирования четырех числовых алгоритмов STL.

Шаблонный метод

Прикладная библиотека позволяет создать новый класс, производный от одного или нескольких готовых классов. Новое приложение заново использует большую часть кода существующих классов, а переопределение одной или нескольких функ-

ций адаптирует приложение для конкретной ситуации. Шаблонный метод (Template method) принадлежит к числу важнейших концепций в прикладных библиотеках; обычно он остается скрытым от пользователя и управляет работой приложения посредством вызова различных функций базового класса (часть из которых была переопределена программистом при создании приложения).

Важнейшая особенность Шаблонного метода заключается в том, что он определяется в базовом классе (иногда в виде закрытой функции) и не может изменяться. Он вызывает другие функции базовых классов для достижения своих целей, но прикладной программист не всегда может вызывать его напрямую, как показывает следующий пример:

```
//: C10:TemplateMethod.cpp
// Простая демонстрация Шаблонного метода.
#include <iostream>
using namespace std;

class ApplicationFramework {
protected:
  virtual void customize1() = 0;
  virtual void customize2() = 0;
public:
  void templateMethod() {
    for(int i = 0; i < 5; i++) {
      customize1();
      customize2();
    }
  }
};

// Создание нового "приложения":
class MyApp : public ApplicationFramework {
protected:
  void customize1() { cout << "Hello "; }
  void customize2() { cout << "World!" << endl; }
};

int main() {
  MyApp app;
  app.templateMethod();
} ///:~
```

Шаблонный метод — «двигатель», обеспечивающий выполнение приложения. В графических приложениях таким «двигателем» обычно является основной цикл обработки событий. Прикладной программист просто предоставляет определения customize1() и customize2(), и приложение готово к работе.

Стратегия

Шаблонный метод содержит «постоянный код», а переопределяемые функции содержат «переменный код». Тем не менее, механизм наследования фиксирует эти изменения на стадии компиляции. Следуя принципу «композиция предпочтительнее наследования», можно воспользоваться композицией для отделения изменяющегося кода от постоянного; так мы приходим к идее паттерна *Стратегия* (Strategy). Очевидным преимуществом такого подхода является возможность под-

ключения переменного кода во время выполнения. Наряду со Стратегией добавляется паттерн Контекст (Context), то есть суррогатный класс, управляющий выбором и использованием конкретного объекта стратегии — совсем как Состояние!

Термин «стратегия» означает лишь то, что у проблемы имеется несколько решений. Допустим, вы забыли, как зовут встреченного вами знакомого. Из неловкого положения можно выйти несколькими способами:

```
//: C10:Strategy.cpp
// Паттерн Стратегия.
#include <iostream>
using namespace std;

class NameStrategy {
public:
  virtual void greet() = 0;
};

class SayHi : public NameStrategy {
public:
  void greet() {
    cout << "Hi! How's it going?" << endl;
  }
};

class Ignore : public NameStrategy {
public:
  void greet() {
    cout << "(Pretend I don't see you)" << endl;
  }
};

class Admission : public NameStrategy {
public:
  void greet() {
    cout << "I'm sorry. I forgot your name." << endl;
  }
};

// Контекст управляет выбором стратегии:
class Context {
  NameStrategy& strategy;
public:
  Context(NameStrategy& strat) : strategy(strat) {}
  void greet() { strategy.greet(); }
};

int main() {
  SayHi sayhi;
  Ignore ignore;
  Admission admission;
  Context c1(sayhi), c2(ignore), c3(admission);
  c1.greet();
  c2.greet();
  c3.greet();
} ///:~
```

Метод **Context::greet()** обычно имеет более сложную структуру; как и шаблонный метод, он содержит неизменяемый код. Но функция main() наглядно показы-

вает, что выбор стратегии может осуществляться на стадии выполнения программы. Можно пойти еще дальше, объединить этот паттерн с Состоянием и изменять Стратегию в течение жизненного цикла объекта Context.

Цепочка ответственности

Паттерн *Цепочка ответственности* (Chain of responsibility) может рассматриваться как «динамическое обобщение рекурсии» с использованием объектов Стратегии. Программа выдает вызов, и каждая стратегия в связанном списке пытается обслужить этот вызов. Процесс кончается тогда, когда одна из стратегий окажется успешной, или с завершением цепочки. При рекурсии одна функция многократно вызывает сама себя до тех пор, пока не будет достигнуто условие выхода; в Цепочке ответственности функция вызывает себя, что (в результате смещения по цепочке) приводит к вызову другой реализации этой функции, и т. д. вплоть до достижения выходного условия. Выходным условием является либо успешность одной из Стратегий, либо достижение конца цепочки. В последнем случае обычно возвращается объект по умолчанию; впрочем, не всегда, и тогда нужно предусмотреть другие возможности для проверки успеха или неудачи.

Вместо того чтобы обслуживать запрос одной функцией, мы даем возможность обслужить запрос всем функциям цепочки. Такая процедура имеет нечто общее с экспертной системой. Поскольку цепочка фактически представляет собой список, она может создаваться динамически, так что ее можно рассматривать как обобщенную динамически создаваемую команду switch.

В книге БЧ тема построения Цепочки ответственности в виде связанного списка обсуждается достаточно подробно. Однако с точки зрения паттерна способ построения цепочки уходит на второй план; он относится к подробностям реализации. Поскольку книга БЧ была написана до того, как контейнеры STL стали доступными в большинстве компиляторов C++, вероятно, такое внимание к техническим деталям объясняется следующими причинами: во-первых, в языке отсутствовали связанные списки, и реализация паттерна требовала создания их собственной версии; во-вторых, академическая наука относится к структурам данных как к одной из основополагающих дисциплин, и авторам БЧ просто не пришло в голову, что структуры данных могут быть включены в стандартный инструментарий языка программирования. Мы считаем, что самостоятельная реализация Цепочки ответственности в виде списка (по БЧ — связанного списка) ничего не добавляет к решению, и задача, как показано далее, также легко решается с применением контейнеров STL.

В следующем примере Цепочка ответственности автоматически находит решение, используя механизм автоматического рекурсивного перебора всех Стратегий:

```
//: C10:ChainOfReponsibility.cpp
// Подход пятилетнего ребенка.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

enum Answer { NO, YES };

class GimmeStrategy {
```

```
public:
  virtual Answer canIHave() = 0;
  virtual ~GimmeStrategy() {}
};

class AskMom : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Mooom? Can I have this?" << endl;
    return NO;
  }
};

class AskDad : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Dad, I really need this!" << endl;
    return NO;
  }
};

class AskGrandpa : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Grandpa, is it my birthday yet?" << endl;
    return NO;
  }
};

class AskGrandma : public GimmeStrategy {
public:
  Answer canIHave() {
    cout << "Grandma, I really love you!" << endl;
    return YES;
  }
};

class Gimme : public GimmeStrategy {
  vector<GimmeStrategy*> chain;
public:
  Gimme() {
    chain.push_back(new AskMom());
    chain.push_back(new AskDad());
    chain.push_back(new AskGrandpa());
    chain.push_back(new AskGrandma());
  }
  Answer canIHave() {
    vector<GimmeStrategy*>::iterator it = chain.begin();
    while(it != chain.end())
      if((*it++)->canIHave() == YES)
        return YES;
    // Все попытки оказались безуспешными...
    cout << "Whiiiiinnne!" << endl;
    return NO;
  }
  ~Gimme() { purge(chain); }
};

int main() {
```

```
  Gimme chain;
  chain.canIHave();
} ///:~
```

Обратите внимание: «контекстный» класс Gimme и все классы Стратегий являются производными от общего базового класса GimmeStrategy.

Прочитав раздел с описанием Цепочки ответственности в книге БЧ, вы обнаружите, что его структура заметно отличается от приведенной ранее, потому что центральное место в нем занимает самостоятельная реализация связанного списка. Но если понять, что сущность Цепочки ответственности — перебор решений до обнаружения подходящего, становится ясно, что конкретная реализация механизма перебора в этом паттерне не важна.

Фабрика

Если вы планируете добавлять в систему новые типы, разумнее всего воспользоваться поддержкой полиморфизма и создать общий интерфейс для новых типов. Таким образом, код систем отделяется от информации о конкретных типах, и добавление новых типов не нарушит работоспособности готового кода... на первый взгляд. Поначалу кажется, что изменения в программе ограничиваются объявлением нового производного типа, но это не совсем так. Объект нового типа все равно нужно создать, а в точке создания требуется задать используемый конструктор. Следовательно, если код создания объектов распределен по приложению, возникает та же проблема, что и при добавлении новых типов — вам придется искать в программе все точки, в которых учитывается конкретный тип объекта. В данном случае речь идет о *создании* типа, а не его *использовании* (которое обеспечивается автоматически средствами полиморфизма), но последствия остаются прежними: добавление нового типа порождает проблемы.

Одно из возможных решений — потребовать, чтобы все объекты создавались через единую *Фабрику* (Factory), вместо распределения кода создания объектов по системе. Если каждый раз, когда программе потребуется создать новый объект, она будет пользоваться услугами Фабрики, то для добавления нового типа объекта достаточно модифицировать Фабрику. Такая архитектура является разновидностью паттерна, часто называемого *Фабричным методом* (Factory method). Объекты создаются во всех объектно-ориентированных программах, а расширение программ с добавлением новых типов встречается достаточно часто, поэтому Фабрики могут стать одним из самых полезных паттернов проектирования.

В качестве примера рассмотрим классическую иерархию Shape. Один из способов реализации Фабрики основан на определении статической функции в базовом классе:

```
//: C10:ShapeFactory1.cpp
#include <iostream>
#include <stdexcept>
#include <cstddef>
#include <string>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
```

```
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  static Shape* factory(const string& type)
    throw(BadShapeCreation);
};

class Circle : public Shape {
  Circle() {} // Закрытый конструктор
  friend class Shape;
public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square() {}
  friend class Shape;
public:
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  ~Square() { cout << "Square::~Square" << endl; }
};

Shape* Shape::factory(const string& type)
  throw(Shape::BadShapeCreation) {
  if(type == "Circle") return new Circle;
  if(type == "Square") return new Square;
  throw BadShapeCreation(type);
}

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };

int main() {
  vector<Shape*> shapes;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
      shapes.push_back(Shape::factory(sl[i]));
  } catch(Shape::BadShapeCreation e) {
    cout << e.what() << endl;
    purge(shapes);
    return EXIT_FAILURE;
  }
  for(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    shapes[i]->erase();
  }
  purge(shapes);
} ///:~
```

Функция factory() получает аргумент, по которому определяется тип создаваемого объекта Shape. В нашем примере аргумент относится к типу string, но он может

содержать произвольный набор данных. При добавлении в систему нового типа Shape все изменения вносятся только в функции factory() (предполагается, что данные для инициализаций объектов поступают извне; жестко закодированный массив показан только для примера).

Чтобы объекты создавались только в factory(), конструкторы конкретных подтипов Shape объявлены закрытыми, а тип Shape — дружественным (friend), чтобы функция factory() имела доступ ко всем конструкторам (дружественной можно объявить только функцию Shape::factory(), но, похоже, объявление дружественным всего базового класса никаких проблем не создаст). У этой архитектуры есть еще одна важная особенность: базовый класс Shape должен располагать полной информацией обо всех производных классах, что считается нежелательным в объектно-ориентированных архитектурах. Если базовый класс приходится обновлять с включением каждого нового тина, расширение библиотеки начинает порождать слишком много проблем. Чтобы избежать этой нежелательной циклической зависимости, следует воспользоваться полиморфными фабриками, описанными в следующем разделе.

Полиморфные фабрики

Статическая функция factory() из предыдущего примера позволяла сосредоточить все операции создания объектов в одном месте (единственном, в котором вносились изменения в программу). Бесспорно, это вполне разумное решение, обеспечивающее хорошую инкапсуляцию процесса создания объектов. Тем не менее, по БЧ главная цель Фабричного метода — возможность определения разных типов фабрик, производных от базовой фабрики. В сущности, Фабричный метод является особой разновидностью полиморфной фабрики. Далее приводится пример ShapeFactory2.cpp, в котором Фабричные методы размещаются в отдельном классе в виде виртуальных функций:

```
//: C10:ShapeFactory2.cpp
// Полиморфные Фабричные методы.
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <stdexcept>
#include <cstddef>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class ShapeFactory {
  virtual Shape* create() = 0;
  static map<string, ShapeFactory*> factories;
public:
  virtual ~ShapeFactory() {}
```

```
  friend class ShapeFactoryInitializer;
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  static Shape*
  createShape(const string& id) throw(BadShapeCreation) {
    if(factories.find(id) != factories.end())
      return factories[id]->create();
    else
      throw BadShapeCreation(id);
  }
};

// Определение статического объекта:
map<string, ShapeFactory*> ShapeFactory::factories;

class Circle : public Shape {
  Circle() {} // Закрытый конструктор
  friend class ShapeFactoryInitializer;
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Circle; }
    friend class ShapeFactoryInitializer;
  };
public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square() {}
  friend class ShapeFactoryInitializer;
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Square; }
    friend class ShapeFactoryInitializer;
  };
public:
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  ~Square() { cout << "Square::~Square" << endl; }
};

// Синглет для инициализации ShapeFactory:
class ShapeFactoryInitializer {
  static ShapeFactoryInitializer si;
  ShapeFactoryInitializer() {
    ShapeFactory::factories["Circle"]= new Circle::Factory;
    ShapeFactory::factories["Square"]= new Square::Factory;
  }
```

```
  ~ShapeFactoryInitializer() {
    map<string, ShapeFactory*>::iterator it =
      ShapeFactory::factories.begin();
    while(it != ShapeFactory::factories.end())
      delete it++->second;
  }
};

// Определение статического объекта:
ShapeFactoryInitializer ShapeFactoryInitializer::si;

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };

int main() {
  vector<Shape*> shapes;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
      shapes.push_back(ShapeFactory::createShape(sl[i]));
  } catch(ShapeFactory::BadShapeCreation e) {
    cout << e.what() << endl;
    return EXIT_FAILURE;
  }
  for(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    shapes[i]->erase();
  }
  purge(shapes);
} ///:~
```

В новой версии Фабричный метод находится в отдельном классе ShapeFactory в виде виртуальной функции create(). Функция объявлена закрытой, то есть она не может вызываться напрямую, но может переопределяться. Подклассы Shape должны создать собственные подклассы ShapeFactory и переопределить функцию create() для создания объекта своего типа. Вследствие своей закрытости фабрики доступны только из главного Фабричного метода. Следовательно, чтобы создать объект, весь клиентский код должен действовать через Фабричный метод.

Непосредственное создание объектов осуществляется вызовом ShapeFactory::createShape() — статической функции, которая путем отображения в ShapeFactory находит соответствующий объект фабрики по переданному идентификатору. Фабрика непосредственно создает объект, хотя при желании можно представить более сложную ситуацию, в которой объект-фабрика возвращается и используется вызывающей стороной для создания объекта более изощренным способом. Впрочем, чаще сложности полиморфного Фабричного метода отказываются излишними, и решение со статической функцией в базовом классе (см. программу ShapeFactory1.cpp) работает нормально.

Обратите внимание: объект ShapeFactory необходимо инициализировать и заполнить отображение объектами фабрик; это происходит в Синглете ShapeFactoryInitializer. Итак, чтобы добавить новый тип в эту архитектуру, вы должны определить тип, создать фабрику и модифицировать Синглет ShapeFactoryInitializer, чтобы экземпляр фабрики сохранялся в отображении. Все эти сложности снова наводят на мысль, что если вам не требуется создавать самостоятельные объекты фабрик, лучше воспользоваться статическим Фабричным методом.

Абстрактные фабрики

Паттерн *Абстрактная фабрика* (Abstract factory) напоминает фабрики, упоминавшиеся ранее, но с несколькими Фабричными методами. Каждый Фабричный метод создает отдельную разновидность объектов. Создавая объект-фабрику, вы решаете, как будут использоваться все объекты, создаваемые этой фабрикой. Пример в книге БЧ обеспечивает переносимость программы между разными графическими пользовательскими интерфейсами (GUI): вы создаете объект-фабрику, соответствующий текущему графическому интерфейсу, а затем при запросе меню, кнопки, ползунка и т. д. фабрика автоматически создает соответствующую версию указанного элемента. Таким образом, все последствия от перехода на другую версию графической среды изолируются в одном месте.

Или другой пример: допустим, вы создаете обобщенную игровую среду для разных типов игр. Вот как может выглядеть такая среда при использовании Абстрактной фабрики:

```
//: C10:AbstractFactory.cpp
// Игровая среда.
#include <iostream>
using namespace std;

class Obstacle {
public:
  virtual void action() = 0;
};

class Player {
public:
  virtual void interactWith(Obstacle*) = 0;
};

class Kitty: public Player {
  virtual void interactWith(Obstacle* ob) {
    cout << "Kitty has encountered a ";
    ob->action();
  }
};

class KungFuGuy: public Player {
  virtual void interactWith(Obstacle* ob) {
    cout << "KungFuGuy now battles against a ";
    ob->action();
  }
};

class Puzzle: public Obstacle {
public:
  void action() { cout << "Puzzle" << endl; }
};

class NastyWeapon: public Obstacle {
public:
  void action() { cout << "NastyWeapon" << endl; }
};

// Абстрактная фабрика:
class GameElementFactory {
```

```
public:
  virtual Player* makePlayer() = 0;
  virtual Obstacle* makeObstacle() = 0;
};

// Конкретные фабрики:
class KittiesAndPuzzles : public GameElementFactory {
public:
  virtual Player* makePlayer() { return new Kitty; }
  virtual Obstacle* makeObstacle() { return new Puzzle; }
};

class KillAndDismember : public GameElementFactory {
public:
  virtual Player* makePlayer() { return new KungFuGuy; }
  virtual Obstacle* makeObstacle() {
    return new NastyWeapon;
  }
};

class GameEnvironment {
  GameElementFactory* gef;
  Player* p;
  Obstacle* ob;
public:
  GameEnvironment(GameElementFactory* factory)
  : gef(factory), p(factory->makePlayer()),
    ob(factory->makeObstacle()) {}
  void play() { p->interactWith(ob); }
  ~GameEnvironment() {
    delete p;
    delete ob;
    delete gef;
  }
};

int main() {
  GameEnvironment
    g1(new KittiesAndPuzzles),
    g2(new KillAndDismember);
  g1.play();
  g2.play();
}
/* Вывод:
Kitty has encountered a Puzzle
KungFuGuy now battles against a NastyWeapon */ ///:~
```

В этой среде объекты Player взаимодействуют с объектами Obstacle, но типы этих объектов зависят от игры. Вы определяете тип игры, выбирая определенный объект GameElementFactory, после чего класс GameEnvironment управляет исходным состоянием и ходом игры. В нашем примере этот класс не рассчитан на наследование, хотя, возможно, это имело бы смысл.

Приведенный пример также демонстрирует методику *двойной диспетчеризации*, которая будет рассматриваться далее.

Виртуальные конструкторы

Одна из главных целей использования фабрик — такая организация кода, при которой бы вам не приходилось выбирать конкретный тип конструктора при созда-

нии объекта. Другими словами, фабрике можно сказать: «Я не знаю точно, какой объект мне нужен, но вот тебе информация. Создай соответствующий тип».

Однако при вызове конструктора механизм виртуального вызова не работает (происходит раннее связывание). Иногда это бывает неудобно. Например, в программе Shape было бы вполне логично выполнить всю подготовку в конструкторе Shape, а затем вывести фигуру вызовом draw(). Функция draw() должна быть виртуальной функцией, то есть сообщением классу Shape, приказывающим выполнить рисование конкретного типа фигуры, будь то Circle, Square или Line. Однако с конструктором такой вариант не проходит, потому что при вызове в конструкторах виртуальные функции заменяются «локальными» телами функций.

Если вы хотите, чтобы виртуальная функция вызывалась из конструктора и делала то, что требуется, придется использовать методику *имитации виртуального конструктора*. Возникает крайне запутанная ситуация. Вспомните: виртуальные функции создаются для того, чтобы вы могли отправить сообщение объекту и дать ему возможность выбрать правильное действие. Но конструктор предназначен для построения объектов, поэтому вызов виртуального конструктора означает следующее: «Я не знаю, к какому типу объекта ты относишься, но все равно построй мне правильный тип». В обычном конструкторе компилятор должен знать адрес таблицы VTABLE, связываемый с VPTR, а виртуальный конструктор (даже если бы он существовал) сделать этого не смог бы, потому что он не располагает всей информацией о типе на стадии компиляции. Невозможность создания виртуальных конструкторов оправданна, потому что это единственная функция, которая должна знать о типе объекта абсолютно все.

И все же встречаются ситуации, когда в программе нужно сделать нечто, приближенное к поведению виртуального конструктора.

Так, в примере с классом Shape нам хотелось бы передать конструктору Shape уточняющую информацию в списке аргументов и поручить создать конкретную разновидность объекта Shape (Circle или Square) без нашего дальнейшего вмешательства. Вместо этого обычно приходится явно вызывать конструктор Circle или Square.

У Джеймса Коплина (James O. Coplien)[1] решение этой задачи называется «конвертом». Под «конвертом» понимается базовый класс, содержащий указатель на объект, также относящийся к типу базового класса. Конструктор «конверта» определяет (при вызове на стадии выполнения, а не на стадии компиляции, когда обычно производится проверка типов), какой тип нужно создать, создает объект конкретного типа (в куче) и устанавливает на него свой указатель. Далее все вызовы функций производятся базовым классом через указатель. В действительности это всего лишь разновидность паттерна Состояние, где базовый класс работает как суррогатный для производного класса, а производный класс обеспечивает изменения в поведении.

```
//: C10:VirtualConstructor.cpp
#include <iostream>
#include <string>
#include <stdexcept>
#include <stdexcept>
#include <cstddef>
```

[1] «Advanced C++ Programming Styles and Idioms», Addison Wesley, 1992.

```
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
  Shape* s;
  // Запрет конструирования копий и присваивания
  Shape(Shape&);
  Shape operator=(Shape&);
protected:
  Shape() { s = 0; }
public:
  virtual void draw() { s->draw(); }
  virtual void erase() { s->erase(); }
  virtual void test() { s->test(); }
  virtual ~Shape() {
    cout << "~Shape" << endl;
    if(s) {
      cout << "Making virtual call: ";
      s->erase(); // Виртуальный вызов
    }
    cout << "delete s: ";
    delete s; // Полиморфное удаление
    // (выражение delete 0 допустимо, это пустая операция)
  }
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
    : logic_error("Cannot create type " + type) {}
  };
  Shape(string type) throw(BadShapeCreation);
};

class Circle : public Shape {
  Circle(Circle&);
  Circle operator=(Circle&);
  Circle() {} // Private constructor
  friend class Shape;
public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  void test() { draw(); }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};

class Square : public Shape {
  Square(Square&);
  Square operator=(Square&);
  Square() {}
  friend class Shape;
public:
  void draw() { cout << "Square::draw" << endl; }
  void erase() { cout << "Square::erase" << endl; }
  void test() { draw(); }
  ~Square() { cout << "Square::~Square" << endl; }
};

Shape::Shape(string type) throw(Shape::BadShapeCreation) {
  if(type == "Circle")
```

```
    s = new Circle;
  else if(type == "Square")
    s = new Square;
  else throw BadShapeCreation(type);
  draw();  // Виртуальный вызов в конструкторе
}

char* sl[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square" };

int main() {
  vector<Shape*> shapes;
  cout << "virtual constructor calls:" << endl;
  try {
    for(size_t i = 0; i < sizeof sl / sizeof sl[0]; i++)
      shapes.push_back(new Shape(sl[i]));
  } catch(Shape::BadShapeCreation e) {
    cout << e.what() << endl;
    purge(shapes);
    return EXIT_FAILURE;
  }
  for(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    cout << "test" << endl;
    shapes[i]->test();
    cout << "end test" << endl;
    shapes[i]->erase();
  }
  Shape c("Circle"); // Создание в стеке
  cout << "destructor calls:" << endl;
  purge(shapes);
} ///:~
```

Базовый класс Shape хранит указатель на объект типа Shape в своей единственной переменной (при моделировании «виртуального конструктора» следует особенно внимательно следить за тем, чтобы этот указатель всегда инициализировался существующим объектом). Фактически базовый класс является посредником, поскольку он единственный, что видит и с чем взаимодействует клиентский код.

При каждом порождении нового производного типа от Shape необходимо включить процедуру создания этого типа в одном месте, внутри виртуального конструктора в базовом классе Shape. Сделать это несложно, но при этом возникает нежелательная зависимость между классом Shape и всеми классами, производными от него.

В этом примере информация о создаваемом типе передается виртуальному конструктору в виде строки с именем типа. Тем не менее, в вашей схеме может использоваться другой способ, например в парсере виртуальному конструктору может передаваться результат работы лексического сканера, и на основании этой информации парсер решает, какую лексему следует создать.

Виртуальный конструктор Shape(type) может определяться лишь после объявления всех производных классов. Конструктор по умолчанию может быть определен в классе Shape, но его следует объявить защищенным, чтобы предотвратить создание временных объектов Shape. Конструктор по умолчанию вызывается только конструкторами объектов производных классов. Он должен определяться явно, потому что компилятор автоматически генерирует конструктор по умолчанию

только *при отсутствии* определенных конструкторов. Так как мы должны определить Shape(type), также иридется определить Shape().

Конструктор по умолчанию в этой схеме должен решать по крайней мере одну важную задачу — он должен обнулить указатель s. На первый взгляд это кажется странным, но вспомните, что конструктор по умолчанию вызывается как часть конструирования *фактического объекта* — в терминологии Коплина «письма», а не «конверта». Но «письмо» является производным от «конверта», поэтому оно также наследует переменную s. В «конверте» указатель s играет важную роль, потому что он ссылается на реальный объект, но в «письме» он просто является лишним грузом. Но даже лишний груз необходимо инициализировать, и если s не обнуляется вызовом конструктора по умолчанию для «письма», это кончится плохо (как вы вскоре увидите).

Виртуальный конструктор получает в своем аргументе информацию, полностью определяющую тип объекта. Но обратите внимание на то, что эта информация не читается и не обрабатывается до стадии выполнения, тогда как компилятор в обычном случае должен знать точный тип на стадии компиляции (еще одна причина, позволяющая говорить об успешной имитации виртуальных конструкторов).

Виртуальный конструктор использует свой аргумент для выбора реального конструируемого объекта («письма»), указатель на который затем присваивается переменной «конверта». В этой точке конструирование «письма» уже завершено, поэтому любые виртуальные вызовы будут должным образом перенаправлены.

В качестве примера рассмотрим вызов draw() в виртуальном конструкторе. Если трассировать этот вызов (вручную или с помощью отладчика), вы увидите, что он начинается с функции draw() базового класса Shape. Эта функция вызывает draw() для хранящегося в «конверте» указателя s на «письмо». Все типы, производные от Shape, обладают одинаковым интерфейсом, поэтому виртуальный вызов будет правильно выполнен, хотя он вроде бы находится в конструкторе (в действительности конструктор «письма» уже был завершен). Пока виртуальные вызовы в базовом классе просто вызывают идентичные виртуальные функции через указатель на «письмо», система будет работать правильно.

Чтобы разобраться в происходящем, рассмотрим код main(). При заполнении vector shapes используются вызовы «виртуального конструктора» Shape. Обычно в подобных ситуациях конструктор вызывается для фактического типа, и в объекте устанавливается VPTR для этого типа. Но здесь во всех случаях будет использоваться VPTR для Shape, а не для специализированных классов Circle, Square и Triangle.

В цикле for, где для каждого объекта Shape вызываются функции draw() и erase(), вызов виртуальной функции через VPTR направляется соответствующему типу. Тем не менее, во всех случаях это будет тип Shape. Может возникнуть вопрос: зачем же функции draw() и erase() объявлены виртуальными? Ответ на него проясняется на следующем шаге: версия draw() базового класса через указатель s на «письмо» вызывает виртуальную функцию draw() для «письма». На этот раз вызов направляется фактическому типу объекта, а не базовому классу Shape. Таким образом, за применение виртуальных деструкторов приходится расплачиваться дополнительным уровнем косвенности при каждом вызове виртуальной функции.

Чтобы создать любую переопределяемую функцию (такую как draw(), erase() или test()), вы должны перенаправить все вызовы через указатель s в реализации базового класса, как было показано ранее. Дело в том, что вызов функции «конвер-

та» будет интерпретирован как относящийся к типу Shape, а не к типу, производному от Shape. Только при перенаправлении вызова через s заработает механизм виртуального вызова. Пример показывает, что в функции main() все работает правильно, даже при вызове функций в конструкторах и деструкторах.

Работа деструкторов

Уничтожение объектов в этой схеме тоже происходит нетривиально. Чтобы разобраться, давайте мысленно проследим за тем, что происходит при вызове оператора delete для указателя на объект Shape (а конкретно Square), созданный в куче (ситуация с объектом, созданным в стеке, проще). Вызов delete через полиморфный интерфейс осуществляется при вызове purge().

В shapes хранятся указатели на базовый класс Shape, поэтому компилятор направляет вызов через Shape. В обычной ситуации такой вызов был бы виртуальным, что привело бы к вызову деструктора Square. Но в имитации виртуального конструктора компилятор создает объекты общего типа Shape, хотя конструктор инициализирует указатель конкретным подтипом Shape. Механизм виртуального вызова используется, но VPTR в объекте Shape указывает на таблицу виртуальных функций Shape, а не Square. В результате будет вызван деструктор Shape, который вызывает delete для указателя s в «письме», фактически ссылающегося на объект Square. Вызов также является виртуальным, но на этот раз он разрешается в деструктор Square.

C++ на уровне компилятора гарантирует вызов всех деструкторов в иерархии. Сначала вызывается деструктор Square, за которым по порядку вызываются все промежуточные деструкторы вплоть до деструктора базового класса. В деструкторе базового класса содержится команда delete s. Первоначальный вызов деструктора относился к указателю s «конверта», но теперь он относится к указателю s «письма», который присутствует только потому, что «письмо» наследует от «конверта», а не потому, что он содержит полезную информацию. Следовательно, *этот* вызов delete делать ничего не должен.

Проблема решается обнулением указателя s в «письме». Теперь при вызове деструктора базового класса для «письма» выполняется команда delete 0, которая по определению ничего не делает. Конструктор по умолчанию объявлен защищенным, поэтому он будет вызван *только* в процессе конструирования «письма»; это единственная ситуация, в которой s присваивается ноль.

Решение получается интересным, но достаточно сложным. На практике для маскировки конструирования чаще применяются обычные Фабричные методы вместо сложных схем «виртуальных конструкторов».

Строитель

Основной целью *Строителя* (Builder), который относится к категории паттернов создания объектов, как и только что рассмотренные Фабрики, является отделение конструирование объекта от его «представления». Это означает, что процесс конструирования всегда остается одним и тем же, но полученный объект обладает несколькими разными представлениями. По БЧ главное различие между Строителем и Абстрактной фабрикой состоит в том, что Строитель создает объект за несколь-

ко этапов, поэтому распределенный во времени характер процесса создания играет важную роль. Кроме того, «руководитель» получает серию «деталей», которые он передает Строителю, и каждая деталь используется для выполнения одного из этапов процесса сборки.

В следующем примере моделируется велосипед, состоящий из различных частей в зависимости от его типа (горный, дорожный или гоночный). С каждым типом велосипеда ассоциируется свой класс Строителя, и каждый Строитель реализует интерфейс, заданный в абстрактном классе **BicycleBuilder**. Отдельный класс **BicycleTechnician** представляет объект-«руководитель», описанный в книге БЧ. Этот объект использует конкретный объект **BicycleBuilder** для конструирования объекта **Bicycle**.

```
//: C10:Bicycle.h
// Определения классов для сборки велосипедов;
// демонстрация паттерна Строитель.
#ifndef BICYCLE_H
#define BICYCLE_H
#include <iostream>
#include <string>
#include <vector>
#include <cstddef>
#include "../purge.h"
using std::size_t;

class BicyclePart {
public:
  enum BPart { FRAME, WHEEL, SEAT, DERAILLEUR,
    HANDLEBAR, SPROCKET, RACK, SHOCK, NPARTS };
private:
  BPart id;
  static std::string names[NPARTS];
public:
  BicyclePart(BPart bp) { id = bp; }
  friend std::ostream&
  operator<<(std::ostream& os, const BicyclePart& bp) {
    return os << bp.names[bp.id];
  }
};

class Bicycle {
  std::vector<BicyclePart*> parts;
public:
  ~Bicycle() { purge(parts); }
  void addPart(BicyclePart* bp) { parts.push_back(bp); }
  friend std::ostream&
  operator<<(std::ostream& os, const Bicycle& b) {
    os << "{ ";
    for(size_t i = 0; i < b.parts.size(); ++i)
      os << *b.parts[i] << ' ';
    return os << '}';
  }
};

class BicycleBuilder {
protected:
  Bicycle* product;
public:
```

```
  BicycleBuilder() { product = 0; }
  void createProduct() { product = new Bicycle; }
  virtual void buildFrame() = 0;
  virtual void buildWheel() = 0;
  virtual void buildSeat() = 0;
  virtual void buildDerailleur() = 0;
  virtual void buildHandlebar() = 0;
  virtual void buildSprocket() = 0;
  virtual void buildRack() = 0;
  virtual void buildShock() = 0;
  virtual std::string getBikeName() const = 0;
  Bicycle* getProduct() {
    Bicycle* temp = product;
    product = 0;  // Сброс product
    return temp;
  }
};

class MountainBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const { return "MountainBike";}
};

class TouringBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const { return "TouringBike"; }
};

class RacingBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDerailleur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const { return "RacingBike"; }
};

class BicycleTechnician {
  BicycleBuilder* builder;
```

```
public:
  BicycleTechnician() { builder = 0; }
  void setBuilder(BicycleBuilder* b) { builder = b; }
  void construct();
};
#endif // BICYCLE_H ///:~
```

Класс Bicycle содержит вектор указателей на объекты BicyclePart, которые представляют части, используемые при сборке велосипедов. Чтобы начать сборку велосипеда, BicycleTechnician («руководитель» в этом примере) вызывает BicycleBuilder::createproduct() для производного объекта BicycleBuilder. Функция BicycleTechnician::construct() вызывает все функции интерфейса BicycleBuilder (поскольку она не знает, с каким конкретным типом строителя имеет дело). Конкретные классы строителей опускают (через пустые тела функций) те действия, которые не относятся к типу собираемого велосипеда, как видно из следующего файла реализации:

```
//: C10:Bicycle.cpp {O} {-mwcc}
#include "Bicycle.h"
#include <cassert>
#include <cstddef>
using namespace std;

std::string BicyclePart::names[NPARTS] = {
  "Frame", "Wheel", "Seat", "Derailleur",
  "Handlebar", "Sprocket", "Rack", "Shock" };

// Реализация MountainBikeBuilder
void MountainBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void MountainBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void MountainBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void MountainBikeBuilder::buildDerailleur() {
  product->addPart(
    new BicyclePart(BicyclePart::DERAILLEUR));
}
void MountainBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void MountainBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void MountainBikeBuilder::buildRack() {}
void MountainBikeBuilder::buildShock() {
  product->addPart(new BicyclePart(BicyclePart::SHOCK));
}

// Реализация TouringBikeBuilder
void TouringBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void TouringBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
```

```
}
void TouringBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void TouringBikeBuilder::buildDerailleur() {
  product->addPart(
    new BicyclePart(BicyclePart::DERAILLEUR));
}
void TouringBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void TouringBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void TouringBikeBuilder::buildRack() {
  product->addPart(new BicyclePart(BicyclePart::RACK));
}
void TouringBikeBuilder::buildShock() {}

// Реализация RacingBikeBuilder
void RacingBikeBuilder::buildFrame() {
  product->addPart(new BicyclePart(BicyclePart::FRAME));
}
void RacingBikeBuilder::buildWheel() {
  product->addPart(new BicyclePart(BicyclePart::WHEEL));
}
void RacingBikeBuilder::buildSeat() {
  product->addPart(new BicyclePart(BicyclePart::SEAT));
}
void RacingBikeBuilder::buildDerailleur() {}
void RacingBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void RacingBikeBuilder::buildSprocket() {
  product->addPart(new BicyclePart(BicyclePart::SPROCKET));
}
void RacingBikeBuilder::buildRack() {}
void RacingBikeBuilder::buildShock() {}

// Реализация BicycleTechnician
void BicycleTechnician::construct() {
  assert(builder);
  builder->createProduct();
  builder->buildFrame();
  builder->buildWheel();
  builder->buildSeat();
  builder->buildDerailleur();
  builder->buildHandlebar();
  builder->buildSprocket();
  builder->buildRack();
  builder->buildShock();
} ///:~
```

Оператор << класса Bicycle вызывает соответствующий оператор << для каждого объекта BicyclePart; этот объект выводит имя своего типа, чтобы вы видели, из каких частей состоит объект Bicycle. Пример:

```
//: C10:BuildBicycles.cpp
//{L} Bicycle
// Паттерн Строитель.
#include <cstddef>
#include <iostream>
#include <map>
#include <vector>
#include "Bicycle.h"
#include "../purge.h"
using namespace std;

// Сборка велосипеда с использованием конкретного строителя.
Bicycle* buildMeABike(
  BicycleTechnician& t, BicycleBuilder* builder) {
  t.setBuilder(builder);
  t.construct();
  Bicycle* b = builder->getProduct();
  cout << "Built a " << builder->getBikeName() << endl;
  return b;
}

int main() {
  // Заказ на велосипеды
  map <string, size_t> order;
  order["mountain"] = 2;
  order["touring"] = 1;
  order["racing"] = 3;

  // Сборка велосипедов
  vector<Bicycle*> bikes;
  BicycleBuilder* m = new MountainBikeBuilder;
  BicycleBuilder* t = new TouringBikeBuilder;
  BicycleBuilder* r = new RacingBikeBuilder;
  BicycleTechnician tech;
  map<string, size_t>::iterator it = order.begin();
  while(it != order.end()) {
    BicycleBuilder* builder;
    if(it->first == "mountain")
      builder = m;
    else if(it->first == "touring")
      builder = t;
    else if(it->first == "racing")
      builder = r;
    for(size_t i = 0; i < it->second; ++i)
      bikes.push_back(buildMeABike(tech, builder));
    ++it;
  }
  delete m;
  delete t;
  delete r;

  // Вывод информации о велосипедах
  for(size_t i = 0; i < bikes.size(); ++i)
    cout << "Bicycle: " << *bikes[i] << endl;
  purge(bikes);
}

/* Вывод:
Built a MountainBike
```

```
Built a MountainBike
Built a RacingBike
Built a RacingBike
Built a RacingBike
Built a TouringBike
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Shock }
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Shock }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: {
  Frame Wheel Seat Derailleur Handlebar Sprocket Rack }
*/ ///:~
```

Главным достоинством этого паттерна является то, что он отделяет алгоритм сборки окончательного продукта от частей и позволяет использовать разные алгоритмы для разных продуктов за счет выбора разных реализаций общего интерфейса.

Наблюдатель

Паттерн *Наблюдатель* (Observer) решает довольно распространенную задачу: что, если изменение состояния одного объекта должно приводить к автоматическому обновлению нескольких других объектов? В частности, эта задача встречается в паре «модель-представление» архитектуры Smalltalk модель-представление-контроллер (Model-View-Controller, MVC) и почти эквивалентной архитектуре документ-представление. Допустим, имеются некоторые данные («документ») и два представления: графическое и текстовое. Изменение данных должно привести к обновлению представлений; это и делает Наблюдатель.

В следующей реализации Наблюдателя задействованы два типа объектов. Класс `Observable` хранит список всех объектов, желающих получать оповещения об изменениях. Он вызывает для каждого наблюдателя в списке функцию `notifyObservers()`, которая является частью базового класса `Observable`.

В этом паттерне имеются две переменные составляющие: количество наблюдающих объектов и способ обновления. Таким образом, паттерн позволяет изменить обе эти составляющие без модификации остального кода.

Паттерн Наблюдатель можно реализовать разными способами. Код, приведенный далее, задает общую структуру, на базе которой вы можете построить собственное решение. Для начала объект-наблюдатель описывается следующим интерфейсом:

```
//: C10:Observer.h
// Интерфейс Observer.
#ifndef OBSERVER_H
#define OBSERVER_H

class Observable;
class Argument {};

class Observer {
public:
```

```
  // Вызывается наблюдаемым объектом при каждом его изменении:
  virtual void update(Observable* o, Argument* arg) = 0;
  virtual ~Observer() {}
};
#endif // OBSERVER_H ///:~
```

Так как Observer в этом решении взаимодействует с Observable, начинать нужно с объявления Observable. Пустой класс Argument действует только как базовый класс для любого типа аргумента, передаваемого при обновлении. При желании дополнительный аргумент можно передавать просто в виде void*. В обоих случаях вам придется выполнить понижающее преобразование.

Тип Observer представляет «интерфейсный» класс, содержащий единственную функцию update(). Эта функция вызывается наблюдаемым объектом, когда он решает, что пришло время обновить все наблюдающие объекты. Наличие аргументов не обязательно; вызов функции update() без аргументов все равно соответствует паттерну. Тем не менее, форма с аргументом более универсальна — вместе с оповещением можно передать объект, ставший причиной обновления (поскольку Observer может регистрироваться с несколькими наблюдаемыми объектами), и любую полезную дополнительную информацию. Без аргумента объекту Observer приходится самому выяснять, какой объект инициировал обновление, и собирать всю необходимую информацию.

«Наблюдаемый объект» относится к типу Observable:

```
//: C10:Observable.h
// Класс Observable.
#ifndef OBSERVABLE_H
#define OBSERVABLE_H
#include <set>
#include "Observer.h"

class Observable {
  bool changed;
  std::set<Observer*> observers;
protected:
  virtual void setChanged() { changed = true; }
  virtual void clearChanged() { changed = false; }
public:
  virtual void addObserver(Observer& o) {
    observers.insert(&o);
  }
  virtual void deleteObserver(Observer& o) {
    observers.erase(&o);
  }
  virtual void deleteObservers() {
    observers.clear();
  }
  virtual int countObservers() {
    return observers.size();
  }
  virtual bool hasChanged() { return changed; }
  // Если объект изменился, оповестить всех наблюдателей:
  virtual void notifyObservers(Argument* arg = 0) {
    if(!hasChanged()) return;
    clearChanged(); // Снять признак "измененности"
    std::set<Observer*>::iterator it;
    for(it = observers.begin();it != observers.end(); it++)
```

```
      (*it)->update(this, arg);
  }
  virtual ~Observable() {}
};
#endif // OBSERVABLE_H ///:~
```

И снова архитектура получается более сложной, чем необходимо. Если Observable может регистрировать объекты Observer и обновлять их, конкретный набор функций значения не имеет. Тем не менее, эта архитектура ориентирована на многократное использование (за образец была взята архитектура стандартной библиотеки Java)[1].

Объект Observable содержит флаг, указывающий, что объект был изменен. В более простой архитектуре такой флаг отсутствует; если с объектом что-нибудь происходит, об этом оповещаются все наблюдатели. Но стоит заметить, что функции, управляющие состоянием флага, объявлены защищенными, поэтому решить, что именно следует считать изменением, может только производный класс, но не конечный пользователь производного класса Observer.

Коллекция объектов Observer хранится в множестве set<Observer*> для предотвращения возможных дубликатов. Класс предоставляет стандартные операции множеств insert(), erase(), clear() и size() для произвольного добавления и удаления наблюдателей; тем самым обеспечивается необходимая гибкость на стадии выполнения.

Большая часть работы происходит в вызове notifyObservers(). Если флаг changed() не был установлен, вызов не делает ничего. В противном случае он сначала сбрасывает флаг changed, чтобы предотвратить напрасную трату времени при повторных вызовах notifyObservers(). Это делается перед оповещением наблюдателей на тот случай, если вызовы update() приведут к изменениям в объекте Observable. Затем функция перебирает элементы множества и вызывает функцию update() каждого объекта Observer.

На первый взгляд может показаться, что для управления обновлениями можно воспользоваться обычным объектом Observable. Но такое решение работать не будет; чтобы добиться какого-нибудь эффекта, вы *должны* создать класс, производный от Observable, и где-то в коде производного класса вызвать функцию setChanged(). Эта функция устанавливает признак изменения, чтобы вызов notifyObservers() действительно приводил к оповещению наблюдателей. А *когда именно* вызывать setChanged(), зависит от логики программы.

Но здесь возникает дилемма. Интерес для вашей программы могут представлять несколько аспектов наблюдаемых объектов. Например, при отслеживании элемента графического интерфейса (скажем, кнопки) такими аспектами могут быть щелчок мышью на кнопке, наведение указателя мыши на кнопку, изменение цвета кнопки (по какой-либо причине). Нам бы хотелось оповещать об этих событиях разных наблюдателей, каждый из которых реагирует на свой тип события.

Проблема в том, что в подобных ситуациях обычно хочется воспользоваться множественным наследованием: «Сначала унаследуем от Observable, чтобы обработать щелчки, потом... унаследуем от Observable, чтобы обработать перемещение указателя и, наконец, ... хм, ничего не выйдет».

---

[1] Отличие состоит в том, что java.util.Observable.notifyOvservers() не вызывает функцию clearChanged() после оповещения наблюдателей.

Идиома внутреннего класса

Перед нами ситуация, в которой мы (фактически) должны выполнить повышающее преобразование к нескольким типам, но в данном случае требуется предоставить несколько *разных* реализаций одного базового типа. Решение, позаимствованное нами из Java, развивает концепцию вложенных классов C++. В Java имеется встроенная поддержка *внутренних классов*, которые напоминают вложенные классы C++, но обладают доступом к нестатическим данным вмещающего класса; для этого неявно используется указатель this объекта класса, в котором они были созданы[1].

Чтобы реализовать идиому внутреннего класса в C++, необходимо получить и использовать указатель на вмещающий объект. Пример:

```
//: C10:InnerClassIdiom.cpp
// Идиома "внутреннего класса".
#include <iostream>
#include <string>
using namespace std;

class Poingable {
public:
  virtual void poing() = 0;
};

void callPoing(Poingable& p) {
  p.poing();
}

class Bingable {
public:
  virtual void bing() = 0;
};

void callBing(Bingable& b) {
  b.bing();
}

class Outer {
  string name;
  // Определение первого внутреннего класса:
  class Inner1;
  friend class Outer::Inner1;
  class Inner1 : public Poingable {
    Outer* parent;
  public:
    Inner1(Outer* p) : parent(p) {}
    void poing() {
      cout << "poing called for "
        << parent->name << endl;
      // Обращение к данным объекта внешнего класса
    }
  } inner1;
  // Определение второго внутреннего класса:
  class Inner2;
```

[1] Внутренние классы также отчасти напоминают *замыкания*, сохраняющие контекст вызова функции для его последующего воспроизведения.

```
  friend class Outer::Inner2;
  class Inner2 : public Bingable {
    Outer* parent;
  public:
    Inner2(Outer* p) : parent(p) {}
    void bing() {
      cout << "bing called for "
        << parent->name << endl;
    }
  } inner2;
public:
  Outer(const string& nm)
  : name(nm), inner1(this), inner2(this) {}
  // Возвращение ссылки на интерфейсы,
  // реализованные внутренними классами:
  operator Poingable&() { return inner1; }
  operator Bingable&() { return inner2; }
};

int main() {
  Outer x("Ping Pong");
  // Выглядит как повышающее преобразование
  // к разным базовым типам!
  callPoing(x);
  callBing(x);
} ///:~
```

Этот пример (предназначенный для демонстрации простейшего синтаксиса идиомы; пример реального использования будет приведен далее) начинается с интерфейсов Poingable и Bingable, каждый из которых содержит всего одну функцию. Обслуживание, предоставляемое функциями callPoing() и callBing(), требует, чтобы получаемый ими объект реализовывал соответственно интерфейсы Poingable и Bingable, но для достижения максимальной гибкости другие требования к объекту не предъявляются. Обратите внимание на отсутствие виртуальных деструкторов в обоих интерфейсах — это сделано для того, чтобы уничтожение объектов никогда не выполнялось через интерфейс.

Конструктор Outer содержит закрытые данные (name). Мы хотим, чтобы он предоставлял интерфейсы Poingable и Bingable, чтобы его можно было использовать с функциями callPoing() и callBing() (в этой ситуации *можно* прибегнуть к множественному наследованию, но для простоты мы обходимся без него). Чтобы предоставить объект Poingable без объявления Outer производным от Poingable, мы задействуем идиому внутреннего класса. Сначала объявление class Inner указывает на то, что где-то существует вложенный класс с указанным именем. Это позволяет объявить класс дружественным в следующей строке. Наконец, после предоставления вложенному классу доступа ко всем закрытым элементам Outer можно переходить к определению класса. Обратите внимание: в классе хранится указатель на создавший его объект Outer, и этот указатель должен инициализироваться в конструкторе. В завершение реализуется функция poing() из класса Poingable. Затем процесс повторяется для второго внутреннего класса, реализующего Bingable. Каждый внутренний класс создается в одном закрытом экземпляре, инициализируемом в конструкторе Outer. Создание вложенных объектов и возвращение ссылок на них решает проблемы с жизненным циклом объектов.

Следует помнить, что оба внутренних класса определяются закрытыми, и клиентский код не получает доступа к деталям реализации, потому что функции

доступа operator Poingable&() и operator Bingable&() возвращают только ссылку на интерфейс, получаемый в результате повышающего преобразования, но не на объект, его реализующий. Более того, из-за закрытости внутренних классов клиентский код даже не может выполнить понижающее преобразование к классам реализации; таким образом, интерфейс полностью изолируется от реализации.

Мы также определили функции автоматического преобразования типов operator Poingable&() и operator Bingable&(). Как видно из функции main(), это позволяет использовать конструкции, выглядящие так, словно Outer порождается множественным наследованием от Poingable и Bingable. Различие состоит в том, что «преобразования» являются односторонними: вы можете добиться эффекта повышающего преобразования к Poingable и Bingable, но не сможете выполнить обратное понижающее преобразование к Outer. В следующем примере продемонстрирован более типичный подход: доступ к объектам внутренних классов осуществляется с помощью обычных функций вместо автоматических функций преобразования типа.

Пример

Вооружившись заголовочными файлами Observer и Observable, а также идиомой внутреннего класса, мы переходим к рассмотрению паттерна Наблюдатель:

```
//: C10:ObservedFlower.cpp
// Паттерн Наблюдатель.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include "Observable.h"
using namespace std;

class Flower {
  bool isOpen;
public:
  Flower() : isOpen(false),
    openNotifier(this), closeNotifier(this) {}
  void open() { // Цветок открывает лепестки
    isOpen = true;
    openNotifier.notifyObservers();
    closeNotifier.open();
  }
  void close() { // Цветок закрывает лепестки
    isOpen = false;
    closeNotifier.notifyObservers();
    openNotifier.close();
  }
  // Применение идиомы "внутреннего класса"
  class OpenNotifier;
  friend class Flower::OpenNotifier;
  class OpenNotifier : public Observable {
    Flower* parent;
    bool alreadyOpen;
  public:
    OpenNotifier(Flower* f) : parent(f),
      alreadyOpen(false) {}
    void notifyObservers(Argument* arg = 0) {
      if(parent->isOpen && !alreadyOpen) {
```

```
        setChanged();
        Observable::notifyObservers();
        alreadyOpen = true;
      }
    }
    void close() { alreadyOpen = false; }
  } openNotifier;
  class CloseNotifier;
  friend class Flower::CloseNotifier;
  class CloseNotifier : public Observable {
    Flower* parent;
    bool alreadyClosed;
  public:
    CloseNotifier(Flower* f) : parent(f),
      alreadyClosed(false) {}
    void notifyObservers(Argument* arg = 0) {
      if(!parent->isOpen && !alreadyClosed) {
        setChanged();
        Observable::notifyObservers();
        alreadyClosed = true;
      }
    }
    void open() { alreadyClosed = false; }
  } closeNotifier;
};

class Bee {
  string name;
  // "Внутренний класс" для наблюдения за открытием:
  class OpenObserver;
  friend class Bee::OpenObserver;
  class OpenObserver : public Observer {
    Bee* parent;
  public:
    OpenObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s breakfast time!" << endl;
    }
  } openObsrv;
  // Другой "внутренний класс" для наблюдения за закрытием:
  class CloseObserver;
  friend class Bee::CloseObserver;
  class CloseObserver : public Observer {
    Bee* parent;
  public:
    CloseObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s bed time!" << endl;
    }
  } closeObsrv;
public:
  Bee(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};

class Hummingbird {
```

```
    string name;
    class OpenObserver;
    friend class Hummingbird::OpenObserver;
    class OpenObserver : public Observer {
      Hummingbird* parent;
    public:
      OpenObserver(Hummingbird* h) : parent(h) {}
      void update(Observable*, Argument *) {
        cout << "Hummingbird " << parent->name
          << "'s breakfast time!" << endl;
      }
    } openObsrv;
    class CloseObserver;
    friend class Hummingbird::CloseObserver;
    class CloseObserver : public Observer {
      Hummingbird* parent;
    public:
      CloseObserver(Hummingbird* h) : parent(h) {}
      void update(Observable*, Argument *) {
        cout << "Hummingbird " << parent->name
          << "'s bed time!" << endl;
      }
    } closeObsrv;
  public:
    Hummingbird(string nm) : name(nm),
      openObsrv(this), closeObsrv(this) {}
    Observer& openObserver() { return openObsrv; }
    Observer& closeObserver() { return closeObsrv;}
  };

  int main() {
    Flower f;
    Bee ba("A"), bb("B");
    Hummingbird ha("A"), hb("B");
    f.openNotifier.addObserver(ha.openObserver());
    f.openNotifier.addObserver(hb.openObserver());
    f.openNotifier.addObserver(ba.openObserver());
    f.openNotifier.addObserver(bb.openObserver());
    f.closeNotifier.addObserver(ha.closeObserver());
    f.closeNotifier.addObserver(hb.closeObserver());
    f.closeNotifier.addObserver(ba.closeObserver());
    f.closeNotifier.addObserver(bb.closeObserver());
    // Объект hb перестает наблюдать за открытием:
    f.openNotifier.deleteObserver(hb.openObserver());
    // Происходят изменения, интересующие наблюдателей:
    f.open();
    f.open(); // Цветок уже открыт, состояние не изменилось.
    // Объект A перестает наблюдать за закрытием:
    f.closeNotifier.deleteObserver(
      ba.closeObserver());
    f.close();
    f.close(); // Цветок уже закрыт, состояние не изменилось.
    f.openNotifier.deleteObservers();
    f.open();
    f.close();
  } ///:~
```

События, интересующие наблюдателей — открытие и закрытие цветка (Flower). Благодаря использованию идиомы внутреннего класса оба события наблюдаются

независимо друг от друга. Классы OpenNotifier и CloeNotifier являются производными от Observable, поэтому они обладают доступом к setChanged() и могут передаваться при любых вызовах, требующих объекта Observable. Обратите внимание: потомки Observable объявлены открытыми, что вроде бы противоречит идиоме внутреннего класса. Это объясняется тем, что некоторые из функций этих классов должны быть доступны для прикладного программиста. Нигде не сказано, что внутренний класс обязан быть закрытым; в InnerClassIdiom.cpp мы просто следуем рекомендации «объявлять закрытым все, что возможно». В принципе, можно объявить классы закрытыми и предоставить опосредованный доступ к функциям через Flower, но толку от этого немного.

Идиома внутреннего класса также хорошо подходит для определения нескольких типов наблюдателей в Bee и Hummingbird, поскольку оба класса независимо наблюдают за «открытием» и «закрытием» Flower. Идиома внутреннего класса позволяет получить большую часть достоинств наследования (в частности, возможность обращения к закрытым данным внешнего класса).

В функции main() также проявляется одно из главных преимуществ паттерна Наблюдатель: возможность изменения поведения программы на стадии выполнения посредством динамической регистрации (и ее отмены) объектов Observer в Observable. Гибкость достигается за счет существенного возрастания объема кода. Подобные компромиссы (усложнение программы в одном месте за счет увеличения гибкости и/или снижения сложности в другом месте) вообще характерны для паттернов.

Если изучить предыдущий пример, вы увидите, что OpenNotifier и CloseNotifier используют базовый интерфейс Observable. А это означает, что они могли бы быть производными от совершенно иных классов Observer; единственная связь между Observer и Flower — это интерфейс Observer.

Другой вариант реализации наблюдений с высокой избирательностью основан на пометке событий при помощи каких-нибудь маркеров (пустых классов, строк или перечисляемых типов, обозначающих разные типы наблюдаемых событий). Такой подход реализуется на базе агрегирования, а не наследования, и различия между ними главным образом определяются различными соотношениями эффективности по времени и затратам памяти. С точки зрения клиента эти различия несущественны.

Множественная диспетчеризация и паттерн Посетитель

Организация взаимодействия между несколькими типами часто является причиной хаоса в программе. Для примера возьмем систему разбора и исполнения математических выражений. Пусть нужно использовать запись вида Number + Number, Number * Number и т. д., где Number — базовый класс семейства числовых объектов. Но если в выражении a + b точный тип a и b неизвестен, как организовать их взаимодействие?

Нужно начать с того, о чем вы, вероятно, даже не задумывались: C++ ограничивается *одинарной диспетчеризацией*. Другими словами, при выполнении операции более чем с одним объектом, тип которого неизвестен, C++ позволяет задейство-

вать механизм динамической привязки только по одному из этих типов. Это не решит описанную проблему, поэтому в конечном счете вам придется идентифицировать типы вручную и фактически строить собственную модель динамической привязки.

В основу решения заложена методика, называемая *множественной диспетчеризацией*[1]. В нашем примере решение принимается по двум составляющим, что называется *двойной диспетчеризацией*. Помните, что полиморфизм работает только при вызове виртуальных функций, поэтому если вы хотите организовать множественную диспетчеризацию, в механизм необходимо включить вызов виртуальной функции для определения каждого неизвестного типа. Таким образом, при организации взаимодействия между двумя разными иерархиями типов в каждой иерархии должен присутствовать виртуальный вызов. Обычно используется конфигурация, при которой один вызов функции класса генерирует несколько вызовов виртуальных функций и таким образом определяет сразу несколько типов. В следующем примере вызываются виртуальные функции compete() и eval(), причем обе функции относятся к одному типу (что не является обязательным требованием при множественной диспетчеризации):

```
//: C10:PaperScissorsRock.cpp
// Множественная диспетчеризация.
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "../purge.h"
using namespace std;

class Paper;
class Scissors;
class Rock;

enum Outcome { WIN, LOSE, DRAW };

ostream& operator<<(ostream& os, const Outcome out) {
  switch(out) {
    default:
    case WIN: return os << "win";
    case LOSE: return os << "lose";
    case DRAW: return os << "draw";
  }
}

class Item {
public:
  virtual Outcome compete(const Item*) = 0;
  virtual Outcome eval(const Paper*) const = 0;
  virtual Outcome eval(const Scissors*) const= 0;
  virtual Outcome eval(const Rock*) const = 0;
  virtual ostream& print(ostream& os) const = 0;
  virtual ~Item() {}
  friend ostream& operator<<(ostream& os, const Item* it) {
```

[1] Рассматривается в книге БЧ при описании эталона Посетитель, о котором речь далее в этом разделе.

```
    return it->print(os);
  }
};

class Paper : public Item {
public:
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return DRAW; }
  Outcome eval(const Scissors*) const { return WIN; }
  Outcome eval(const Rock*) const { return LOSE; }
  ostream& print(ostream& os) const {
    return os << "Paper    ";
  }
};

class Scissors : public Item {
public:
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return LOSE; }
  Outcome eval(const Scissors*) const { return DRAW; }
  Outcome eval(const Rock*) const { return WIN; }
  ostream& print(ostream& os) const {
    return os << "Scissors";
  }
};

class Rock : public Item {
public:
  Outcome compete(const Item* it) { return it->eval(this);}
  Outcome eval(const Paper*) const { return WIN; }
  Outcome eval(const Scissors*) const { return LOSE; }
  Outcome eval(const Rock*) const { return DRAW; }
  ostream& print(ostream& os) const {
    return os << "Rock     ";
  }
};

struct ItemGen {
  Item* operator()() {
    switch(rand() % 3) {
      default:
      case 0: return new Scissors;
      case 1: return new Paper;
      case 2: return new Rock;
    }
  }
};

struct Compete {
  Outcome operator()(Item* a, Item* b) {
    cout << a << "\t" << b << "\t";
    return a->compete(b);
  }
};

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  const int sz = 20;
  vector<Item*> v(sz*2);
```

```
  generate(v.begin(), v.end(), ItemGen());
  transform(v.begin(), v.begin() + sz,
    v.begin() + sz,
    ostream_iterator<Outcome>(cout, "\n"),
    Compete());
  purge(v);
} ///:~
```

Перечисляемый тип Outcome классифицирует возможные результаты вызова compete(), а оператор << упрощает вывод конкретных значений Outcome.

Item — базовый класс для типов с множественной диспетчеризацией. Операторная функция Compete::operator() получает два объекта Item* (точный тип обоих объектов неизвестен) и начинает процесс двойной диспетчеризации вызовом виртуальной функции Item::compete(). Механизм виртуального вызова определяет тип a, поэтому управление передается функции compete() конкретного типа a. Функция compete() выполняет вторую диспетчеризацию, вызывая eval() для оставшегося типа. Передача текущего объекта (this) в аргументе eval() порождает вызов перегруженной функции eval() с сохранением информации о типе первой диспетчеризации. При завершении второй диспетчеризации точные типы обоих объектов Item становятся известными.

В функции main() алгоритм generate() библиотеки STL заполняет вектор v, а затем transform() применяет операторную функцию Compete::operator() к двум интервалам. Данная версия transform() получает начальную и конечную точки первого интервала (с левосторонними объектами Item, используемыми при двойной диспетчеризации); начальную точку второго интервала с правосторонними объектами Item; итератор приемника, которым в данном случае является стандартный выходной поток; и объект функции (временный объект типа Compete), вызываемый для каждого объекта.

Чтобы организовать множественную диспетчеризацию, придется потрудиться, но примите во внимание преимущества элегантного синтаксиса вызова — вместо написания громоздкого кода, определяющего типы одного или нескольких объектов при вызове, вы просто говорите: «Эй, вы! К каким бы типам вы ни относились, взаимодействуйте друг с другом!» Впрочем, прежде чем решаться на множественную диспетчеризацию, убедитесь, что такая элегантность действительно важна для вашей программы.

Учтите, что множественная диспетчеризация фактически сводится к поиску по таблице. В нашем случае поиск осуществляется применением виртуальных функций, но вместо этого также можно воспользоваться буквальным поиском. При относительно большом количестве диспетчеризаций (а также если вы склонны вносить в свою программу дополнения и изменения) поиск по таблице может оказаться более удачным решением.

Назначением паттерна *Посетитель* (Visitor) — последнего, и возможно, самого сложного паттерна БЧ — является отделение операций иерархии классов от самой иерархии. Формулировка выглядит довольно странно — в конце концов, объектно-ориентированное программирование в основном используется для объединения данных и операций в объекты, а также для применения полиморфизма, автоматически выбирающего нужную операцию в зависимости от конкретного типа объекта.

В паттерне Посетитель операции выделяются из иерархии классов в отдельную внешнюю иерархию. «Основная» иерархия содержит функцию visit(), прини-

мающую любой объект из иерархии операций. В результате вы получаете две иерархии классов вместо одной. Кроме того, «основная» иерархия становится очень непрочной — добавление нового класса требует внесения изменений во всей второй иерархии. В книге БЧ отмечается, что по этой причине основная иерархия «должна изменяться как можно реже». Такое требование значительно ограничивает свободу действий и сокращает практическую полезность этого паттерна.

Но справедливости ради предположим, что у вас имеется фиксированная основная иерархия классов; возможно, вы получили ее от другого разработчика и не можете вносить в нее изменения. Если бы у вас был исходный код библиотеки, в базовый класс можно было добавить новые виртуальные функции, но по каким-то причинам это невозможно. В другом, более правдоподобном сценарии добавление новых функций создает неудобства, уродливо выглядит или по иным причинам затрудняет сопровождение. БЧ возражает: «распределение всех этих операций по узловым классам усложняет понимание, сопровождение и модификацию системы» — но как вы вскоре убедитесь, паттерн Посетитель может породить еще больше проблем. Другой аргумент БЧ — интерфейс основной иерархии нежелательно захламлять слишком большим количеством операций (с другой стороны, если интерфейс становится излишне «тяжеловесным», возникает вопрос, не перегружен ли объект лишней функциональностью).

Впрочем, создатели библиотеки предвидели, что вы захотите добавить в иерархию новые операции, поэтому они включили функцию visit().

Итак, возникает следующая дилемма: вы хотите добавить в базовый класс новые функции, но по каким-то причинам не можете изменять его. Как обойти это препятствие?

Паттерн Посетитель строится по рассмотренной в начале этого разделе схеме с двойной диспетчеризацией. Он позволяет эффективно расширять интерфейс основного типа посредством создания отдельной иерархии классов типа Visitor для «виртуализации» операций, выполняемых с основным типом. Объекты основного типа просто «принимают» посетителя, а затем вызывают функции посетителя через механизм динамического связывания. Проще говоря, вы создаете посетителя, передаете его основной иерархии и получаете эффект виртуальной функции. Простой пример:

```
//: C10:BeeAndFlowers.cpp
// Паттерн Посетитель.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <ctime>
#include <cstdlib>
#include "../purge.h"
using namespace std;

class Gladiolus;
class Renuculus;
class Chrysanthemum;

class Visitor {
public:
  virtual void visit(Gladiolus* f) = 0;
  virtual void visit(Renuculus* f) = 0;
```

```
  virtual void visit(Chrysanthemum* f) = 0;
  virtual ~Visitor() {}
};

class Flower {
public:
  virtual void accept(Visitor&) = 0;
  virtual ~Flower() {}
};

class Gladiolus : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

class Renuculus : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

class Chrysanthemum : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

// Новая возможность - получение строки:
class StringVal : public Visitor {
  string s;
public:
  operator const string&() { return s; }
  virtual void visit(Gladiolus*) {
    s = "Gladiolus";
  }
  virtual void visit(Renuculus*) {
    s = "Renuculus";
  }
  virtual void visit(Chrysanthemum*) {
    s = "Chrysanthemum";
  }
};

// Новая возможность - выполнение операций "Bee":
class Bee : public Visitor {
public:
  virtual void visit(Gladiolus*) {
    cout << "Bee and Gladiolus" << endl;
  }
  virtual void visit(Renuculus*) {
    cout << "Bee and Renuculus" << endl;
  }
  virtual void visit(Chrysanthemum*) {
    cout << "Bee and Chrysanthemum" << endl;
  }
```

```
};

struct FlowerGen {
  Flower* operator()() {
    switch(rand() % 3) {
      default:
      case 0: return new Gladiolus;
      case 1: return new Renuculus;
      case 2: return new Chrysanthemum;
    }
  }
};

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  vector<Flower*> v(10);
  generate(v.begin(), v.end(), FlowerGen());
  vector<Flower*>::iterator it;
  // Выглядит почти так же, как если бы в класс Flower
  // была добавлена виртуальная функция для получения
  // строкового представления:
  StringVal sval;
  for(it = v.begin(); it != v.end(); it++) {
    (*it)->accept(sval);
    cout << string(sval) << endl;
  }
  // Выполнение операций "Bee" со всеми объектами Flower:
  Bee bee;
  for(it = v.begin(); it != v.end(); it++)
    (*it)->accept(bee);
  purge(v);
} ///:~
```

Класс Flower возглавляет основную иерархию, и каждый из его подтипов умеет «принимать» Посетителей (Visitor) функцией accept(). Иерархия flower() не содержит других операций, помимо accept(), поэтому вся функциональность иерархии Flower содержится в иерархии Visitor. Обратите внимание: классы Visitor должны обладать информацией о конкретных разновидностях Flower, и при добавлении нового типа Flower приходится перерабатывать всю иерархию Visitor.

Функция accept() в подклассах Flower начинает двойную диспетчеризацию. В ходе первой диспетчеризации определяется точный тип Flower, в ходе второй — точный тип Visitor. Зная оба типа, можно выполнять операции, соответствующие обоим типам.

Скорее всего, вам не придется использовать Посетителя — мотивация для его применения необычна, а жесткие ограничения сводят его ценность «на нет». Примеры в книге БЧ неубедительны; сначала авторы рассматривают компилятор (написанием компиляторов занимается не так уж много людей, и вряд ли в них будет использоваться этот паттерн), а потом извиняются за остальные примеры и говорят, что не стали бы применять Посетителя в подобных случаях. Чтобы отказаться от обычной объектно-ориентированной структуры в пользу Посетителя, понадобятся куда более убедительные доводы — какие преимущества вы реально получите в обмен на радикальное усложнение и ограниченность? Почему нельзя просто добавить новые виртуальные функции в базовый класс, когда выяснится, что они вам нужны? А если уж действительно нужно включить новые функции

в существующую иерархию, которая не может изменяться, почему бы сначала не попробовать множественное наследование? (Хотя даже в этом случае шансы на «спасение» существующей иерархии невелики.) Также учтите, что для использования Посетителя существующая иерархия должна изначально содержать функцию visit(), поскольку ее последующее добавление означало бы возможность модификации иерархии (но тогда можно было бы просто добавить нужные виртуальные функции). Нет, Посетитель должен быть частью архитектуры с самого начала, а для его применения нужны основания более веские, чем те, что приводятся БЧ[1].

Мы представляем Посетителя лишь потому, что сталкивались с его неуместным применением — подобно неуместному применению множественного наследования и множества других решений. Если вы используете Посетителя, спросите себя, зачем вам это нужно. Вы *действительно* не можете добавить новые виртуальные функции в базовый класс? Вы *действительно* хотите лишиться возможности добавлять новые типы в основную иерархию?

Итоги

Паттерны проектирования, как и любые абстракции, призваны упростить вашу жизнь. Обычно в любой системе что-нибудь меняется, будь то код во время жизненного цикла проекта или объекты во время исполнения программы. Выясните, что именно меняется; возможно, паттерн поможет вам инкапсулировать изменения и тем самым взять их под свой контроль.

Некоторые программисты увлекаются определенной архитектурой и сами себе создают проблемы, применяя ее только потому, что они хорошо умеют это делать. Как ни странно, соблюсти принцип экстремального программирования «Выберите самое простое решение, которое может работать» оказывается довольно сложно. И все же выбор простейшего решения не только ускоряет реализацию, но и упрощает сопровождение. А если самое простое решение не подходит, то вы узнаете об этом гораздо скорее, чем если бы вы попытались реализовать нечто сложное и потом выяснили, что не работает *именно оно*.

Упражнения

1. Создайте разновидность программы SingletonPattern.cpp, в которой все функции являются статическими. Нужна ли в этом случае функция instance()?
2. Начиная с программы SingletonPattern.cpp создайте класс, предназначенный для подключения к системе в целях сохранения и выборки данных из конфигурационного файла.
3. Взяв за отправную точку программу SingletonPattern.cpp, создайте класс, управляющий фиксированным количеством своих объектов. Допустим, объекты представляют подключения к базе данных, и условия лицензии ограничивают количество одновременных подключений.

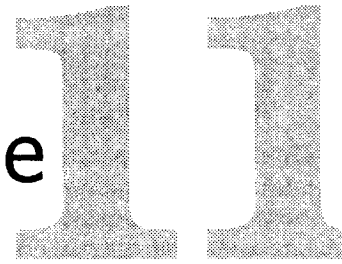
[1] Вероятно, Посетитель был включен в книгу БЧ просто как демонстрация интеллектуальной мощи авторов. На семинаре один из авторов сказал, что «Посетитель — его любимый эталон».

4. Измените программу KissingPrincess2.cpp и добавьте в систему еще одно состояние, чтобы каждый «поцелуй» переводил объект в следующее состояние.
5. Найдите класс C16:TStack.h из первого тома книги. Создайте для него Адаптер, чтобы к элементам TStack можно было применять алгоритм for_each() библиотеки STL. Создайте класс TStack с элементами string, заполните его строками и используйте алгоритм for_each() для подсчета букв во всех строках TStack.
6. Создайте заготовку программы (с помощью паттерна Шаблонный метод), которая получает в командной строке список имен файлов. Каждый файл, кроме последнего, открывается для чтения, а последний файл открывается для записи. Программа обрабатывает все входные файлы некоторой операцией и направляет вывод в последний файл. Воспользуйтесь наследованием и создайте два отдельных приложения для следующих операций:
   - преобразование всех букв в каждом файле к верхнему регистру;
   - поиск в файлах слов, содержащихся в первом файле.
7. Измените решение для упражнения 6 так, чтобы вместо паттерна Шаблонный метод использовался паттерн Стратегия.
8. Измените пример Strategy.cpp и включите в него поддержку Состояния, чтобы Стратегия могла изменяться на протяжении жизненного цикла объекта Context.
9. Измените пример Strategy.cpp так, чтобы в нем использовался паттерн Цепочка ответственности.
10. Добавьте класс Triangle в пример ShapeFactory1.cpp.
11. Добавьте класс Triangle в пример ShapeFactory2.cpp.
12. Добавьте новый тип GameEnvironment с именем GnomesAndFairies в программу AbstractFactory.cpp.
13. Измените пример ShapeFactory2.cpp так, чтобы он использовал Абстрактную фабрику для создания разных наборов фигур (например, фабрика одного типа создает «толстые фигуры», фабрика другого типа — «тонкие фигуры», но каждый объект фабрики способен создавать все фигуры: круги, квадраты, треугольники и т. д.).
14. Измените пример VirtualConstructor.cpp так, чтобы вместо команд if-else в Shape::Shape(string type) использовалось отображение.
15. Разбейте текстовый файл на входной поток слов (не усложняйте задачу и разбивайте входной поток по пропускам). Создайте два Строителя: один заносит слова в множество, а другой создает отображение со словами и количеством вхождений этих слов.
16. Создайте минимальную архитектуру Observer/Observable в двух классах, без базовых классов, без дополнительных аргументов в Observer.h и функций классов в Observable.h. Просто включите в два класса абсолютный минимум возможностей, затем продемонстрируйте работу своей архитектуры,

создав один объект Observable с несколькими объектами Observer. Объект Observable должен обновлять объекты Observer.

17. Измените пример InnerClassIdiom2.cpp так, чтобы вместо идиомы внутреннего класса в Outer использовалось множественное наследование.
18. Измените пример PaperScissorsRock.cpp так, чтобы вместо двойной диспетчеризации в нем был реализован поиск по таблице. Проще всего это делается отображением отображений, в котором ключом является вызов typeid(obj).name(). Поиск выполняется конструкцией map[typeid(obj1).name()][typeid(obj2).name()]. Обратите внимание, насколько упрощается изменение конфигурации системы. В каких случаях это решение предпочтительнее жесткого кодирования динамической диспетчеризации? Сможете ли вы создать систему, которая бы обладала простотой синтаксиса динамической диспетчеризации, но использовала поиск по таблице?
19. Создайте модель деловой среды с тремя типами обитателей (Inhabitant): Dwarf (инженеры), Elf (специалисты по маркетингу) и Troll (руководство). Создайте класс Project, который создает экземпляры разных обитателей и заставляет их взаимодействовать друг с другом (функция interact()) путем множественной диспетчеризации.
20. Измените предыдущий пример, чтобы взаимодействия стали более конкретными. Каждый объект Inhabitant может случайным образом получить оружие (Weapon) вызовом getWeapon: Dwarf использует Jargon и Play, Elf — InventFeature и SellImaginaryProduct, а Troll — Edict и Schedule. Решите, какое оружие «выигрывает» и «проигрывает» в каждом взаимодействии (по аналогии с PaperScissorsRock.cpp). Включите в Project функцию battle(), которая получает два объекта Inhabitant и заставляет их сражаться друг с другом. Далее определите в Project функцию meeting(), которая создает группы объектов Dwarf, Elf и Troll и заставляет их сражаться друг с другом до тех пор, пока в каждой группе не останется по одному участнику.
21. Добавьте в пример BeeAndFlowers.cpp нового посетителя Hummingbird.
22. Добавьте в пример BeeAndFlowers.cpp тип Sunflower. Обратите внимание на то, что нужно изменить для интеграции нового типа в систему.
23. Измените пример BeeAndFlowers.cpp так, чтобы он не *использовал* паттерн Посетитель, а возвращался к обычной иерархии классов. Преобразуйте Bee в параметр-Накопитель.

Многопоточное программирование



Объекты позволяют разделить программу на независимые компоненты. Но во многих случаях программы также делятся на отдельные независимо работающие подзадачи.

При использовании механизма *многопоточности* каждая из этих независимых подзадач работает в отдельном *программном потоке*. Программа пишется так, словно каждый программный поток монопольно распоряжается процессором. Процессорное время распределяется операционной системой, но обычно программисту беспокоиться об этом не нужно, что упрощает многопоточное программирование.

Процессом называется самостоятельная программа, выполняемая в отдельном адресном пространстве. В *многозадачных* операционных системах параллельно выполняются сразу несколько процессов (программ), для чего система периодически передает процессор от задачи к задаче. *Программный поток* представляет собой отдельную логическую подзадачу *внутри* процесса. Таким образом, один процесс может содержать несколько параллельно работающих программных потоков. Так как программные потоки работают внутри одного процесса, они совместно используют память и другие ресурсы. Основные трудности с написанием многопоточных программ возникают из-за необходимости делить эти ресурсы между разными потоками.

Многопоточность находит много возможных практических применений, но чаще всего она требуется тогда, когда некоторая часть вашей программы привязывается к некоторому событию или ресурсу. Чтобы не мешать работе остальных частей программы, вы создаете для этого события или ресурса отдельный программный поток и позволяете ему работать независимо от основной программы.

Многопоточное, или параллельное, программирование открывает перед программистом совершенно новый мир. Переход на него сродни изучению нового языка программирования или, по крайней мере, новых языковых концепций. С появлением многопоточной поддержки в большинстве микрокомпьютерных операционных систем многопоточные расширения стали появляться в языках программирования и библиотеках. Многопоточное программирование всегда:

- необычно и требует коренных изменений в подходе к программированию;
- работает на основе одних и тех же принципов во всех языках (если вы поняли, как работают программные потоки, считайте, что вы освоили новый «объединенный» язык программирования).

Понимание параллельного программирования требует примерно таких же усилий, как понимание полиморфизма. Общие принципы можно представить без особого труда, но чтобы действительно разобраться в этой теме, потребуется основательно потрудиться. В этой главе мы постараемся заложить надежный фундамент в области параллельного программирования, чтобы вы поняли основные концепции и смогли писать собственные многопоточные программы. Только не впадайте в излишнюю самоуверенность — если вам потребуется написать что-нибудь нетривиальное, обратитесь к специализированной литературе по этой теме.

Мотивация

Одно из самых очевидных применений многопоточности — повышение скорости реакции пользовательского интерфейса. Допустим, у вас имеется программа, в работе которой интенсивно используется процессор; это приводит к тому, что программа перестает реагировать на действия пользователя. Нужно, чтобы программа продолжала свою работу, но в то же время периодически возвращала управление пользовательскому интерфейсу для обработки действий пользователя. Если на форме имеется кнопка закрытия, было бы нежелательно опрашивать ее состояние во всех фрагментах программы, которые потенциально могли бы выполняться в момент опроса (сопровождение таких программ вызывает массу проблем). С другой стороны, кнопка все-таки должна реагировать на нажатие, словно вы периодически проверяете ее состояние.

Обычная функция не способна продолжать свою работу, и в то же время возвращать управление другим частям программы. На первый взгляд задача кажется неразрешимой — процессор вроде бы должен заниматься несколькими делами одновременно. Но именно такую иллюзию создает механизм многопоточности (хотя в многопроцессорных системах это нечто большее, чем иллюзия).

Многопоточность также может использоваться для оптимизации ввода-вывода. Например, программа может делать что-нибудь полезное в процессе ожидания данных в порте ввода-вывода. Без применения многопоточности существует единственное разумное решение — периодический опрос порта. Однако этот вариант получается громоздким и вызывает затруднения.

В многопроцессорной системе программные потоки могут распределяться между разными процессами, что приводит к заметному повышению быстродействия. Такая ситуация часто наблюдается на мощных многопроцессорных веб-серверах, которые распределяют многочисленные пользовательские запросы между процессорами в программах и создают для запросов отдельные программные потоки.

Многопоточная программа на однопроцессорном компьютере в любой момент времени все равно решает только одну задачу, поэтому теоретически ту же программу можно написать без использования потоков. Тем не менее, многопоточность обладает важными структурными преимуществами: она кардинально упро-

щает программу. Некоторые классы задач (например, имитация в видеоиграх) плохо решаются без поддержки многопоточности.

Многопоточная модель — не более чем вспомогательная программная конструкция, упрощающая переключения между несколькими операциями в одной программе: система периодически вмешивается в ход событий и предоставляет каждому потоку часть процессорного времени[1]. Каждый поток считает, что он единолично распоряжается процессором, хотя в действительности процессорное время распределяется между всеми потоками. Тем не менее, одно из основных преимуществ многопоточности заключается в том, что программист абстрагируется от этого уровня, поэтому программе не нужно знать, выполняется она на одном процессоре или на нескольких[2]. Таким образом, многопоточность открывает путь к повышению вычислительной мощности, прозрачному для программиста: если программа работает слишком медленно, ее можно легко ускорить, установив на компьютер дополнительные процессоры. Многозадачность и многопоточность обычно обеспечивают наиболее эффективное использование многопроцессорных систем.

Многопоточность сопряжена с определенными затратами ресурсов, однако общий выигрыш благодаря простоте архитектуры, балансировке ресурсов и удобстве для пользователя обычно оказывается гораздо большим. Как правило, программные потоки помогают ослабить жесткую привязку компонентов архитектуры; в противном случае отдельным частям вашей программы придется специально следить за выполнением задач, которые за счет программных потоков обычно решаются автоматически.

Параллелизм в C++

Когда комитет по стандартизации C++ работал над первой версией стандарта C++, механизмы параллельных вычислений не вошли в нее. Во-первых, их не было в C; во-вторых, в то время существовало несколько альтернативных подходов к их реализации. Разработчики стандарта посчитали, что ограничивать программиста только одним из них было бы неправильно.

Однако на практике все обернулось еще хуже. Чтобы написать многопоточную программу, приходилось искать и изучать новую библиотеку, учитывая все ее специфические особенности. Привязка к возможностям конкретного разработчика тоже не добавляла уверенности. Не было никаких гарантий, что такая библиотека будет работать на разных компиляторах и на разных платформах. Кроме того, поскольку параллелизм не был частью стандартного языка, было трудно найти программистов C++, которые бы разбирались в многопоточном программировании.

Другим фактором, повлиявшим на отношение к многопоточности в C++, стал язык Java, в котором поддержка многопоточности интегрирована в базовый син-

[1] Эта формулировка относится к системам с квантованием по времени (например, Windows). В Solaris используется модель многопоточности на базе очереди: если управление не будет передано потоку с более высоким приоритетом, текущий программный поток выполняется до тех пор, пока сам не отдаст процессор.

[2] При условии, что она специально спроектирована для многопроцессорных систем. В противном случае код, нормально работающий в системе с квантованием времени, иногда перестает работать в многопроцессорной системе. Дело в том, что дополнительные процессоры могут выявить проблемы, которые остаются скрытыми в однопроцессорных системах.

таксис. Хотя многопоточность по-прежнему остается достаточно сложной темой, программисты Java начинают осваивать и применять ее на относительно ранней стадии.

Комитет по стандартизации C++ рассматривает вопрос о включении поддержки многопоточности в следующую версию стандарта C++, но на момент написания книги остается неясным, как все это будет выглядеть. В качестве основы при подготовке этой главы была выбрана библиотека ZThreads, причем помощь авторам оказывал разработчик библиотеки, Эрик Крэхен (Eric Crahen) из IBM. Библиотека ZThreads хорошо спроектирована и свободно распространяется с открытыми текстами по адресу http://zthread.sourceforge.net.

Для демонстрации основных принципов многопоточности мы будем использовать лишь часть возможностей ZThreads. На самом деле библиотека обладает гораздо более совершенной поддержкой многопоточности, чем показано в книге. Самостоятельное изучение позволит вам лучше разобраться в ее возможностях.

Установка библиотеки ZThreads

Пожалуйста, учтите, что библиотека ZThreads является независимым проектом, и к ее разработке авторы книги отношения не имеют; мы просто используем эту библиотеку и не предоставляем технической поддержки для ее установки. За информацией об установке и ошибках обращайтесь на сайт разработчика.

Библиотека ZThreads распространяется на уровне исходных текстов. После ее загрузки (версии 2.3 и выше) с сайта необходимо откомпилировать библиотеку и подготовить проект к использованию библиотеки.

В большинстве разновидностей Unix (Linux, SunOS, Cygwin и т. д.) компиляцию ZThreads рекомендуется производить при помощи конфигурационного сценария. После распаковки файлов утилитой tar выполните в основном каталоге архива ZThreads команду

```
./configure && make install
```

Команда компилирует и устанавливает копию библиотеки в каталог /usr/local. При использовании этого сценария можно задать множество параметров, в том числе определяющих расположение файлов. Чтобы получить дополнительную информацию, выполните команду

```
./configure -help
```

Код ZThreads структурирован таким образом, чтобы упростить компиляцию для других платформ и компиляторов (таких, как Borland, Microsoft и Metrowerks). Для этого следует создать новый проект и включить все cxx-файлы из каталога src архива ZThreads в список компилируемых файлов. Также не забудьте включить каталог include архива в список путей к заголовочным файлам вашего проекта. Подробности зависят от конкретного компилятора, поэтому для использования этой возможности вы должны достаточно хорошо разбираться в своем инструментарии.

После успешной компиляции следующим шагом является создание проекта, использующего откомпилированную библиотеку. Сначала необходимо сообщить компилятору местонахождение заголовочных файлов, чтобы ваши команды #include работали правильно. Обычно для этого в проект включается ключ вида

```
-I/путь/include
```

Если вы задействовали сценарий configure, то путь определяется выбранным префиксом (по умолчанию /usr/local/). А при использовании файлов проектов из каталога build он просто соответствует пути к основному каталогу архива ZThreads.

Далее в проект необходимо включить ключ, который сообщает компоновщику, где ему искать библиотеку. При использовании конфигурационного сценария этот ключ выглядит так:

```
-L/путь/lib -lZThread
```

При использовании файлов проектов он принимает вид

```
-L/путь/Debug Zthread.lib
```

Как и при определении местонахождения заголовочных файлов, если вы задействовали сценарий configure, путь определяется выбранным префиксом (по умолчанию /usr/local/). А в случае готовых файлов проектов он соответствует пути к основному каталогу архива ZThreads.

Если вы используете Linux или Cygwin (www.cygwin.com) в среде Windows, возможно, изменять пути к заголовочным или библиотечным файлам не понадобится; процесс установки часто решает эту задачу за вас.

Вероятно, в системе Linux вам придется включить следующую строку в .bashrc, чтобы система времени выполнения могла находить общую библиотеку LibZThread-x.x.so.0 при выполнении программ, приводимых в этой главе (предполагается, что вы использовали стандартный процесс установки, а общая библиотека находится в каталоге /usr/local/lib; в противном случае соответствующим образом измените путь):

```
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
```

Определение задач

Каждый программный поток решет некоторую задачу; следовательно, нам понадобятся средства для описания подобных задач. Класс Runnable определяет общий интерфейс для выполнения произвольной задачи. Ниже приведено ядро класса Runnable библиотеки ZThread, находящееся в файле Runnable.h каталога include после установки библиотеки ZThread:

```
class Runnable {
public:
  virtual void run() = 0;
  virtual void ~Runnable() {}
};
```

Определение Runnable в виде абстрактного базового класса позволяет легко объединять его с другими классами.

Чтобы определить задачу, просто объявите ее класс производным от класса Runnable и переопределите функцию run() так, чтобы она выполняла нужные действия.

Например, следующая задача LiftOff осуществляет обратный отсчет до нуля:

```
//: C11:LiftOff.h
// Демонстрация интерфейса Runnable.
#ifndef LIFTOFF_H
#define LIFTOFF_H
#include "zthread/Runnable.h"
```

```
#include <iostream>

class LiftOff : public ZThread::Runnable {
  int countDown;
  int id;
public:
  LiftOff(int count, int ident = 0) :
    countDown(count), id(ident) {}
  ~LiftOff() {
    std::cout << id << " completed" << std::endl;
  }
  void run() {
    while(countDown--)
      std::cout << id << ":" << countDown << std::endl;
    std::cout << "Liftoff!" << std::endl;
  }
};
#endif // LIFTOFF_H ///:~
```

Переменная id идентифицирует разные экземпляры одной задачи. Если программа запускается в единственном экземпляре, передайте значение по умолчанию в параметре ident. Деструктор помогает убедиться в том, что объект задачи был должным образом уничтожен.

В следующем примере функция run() объекта задачи запускается не в отдельном потоке, а напрямую вызывается в main():

```
//: C11:NoThread.cpp
#include "LiftOff.h"

int main() {
  LiftOff launch(10);
  launch.run();
} ///:~
```

Классы, производные от Runnable, должны содержать функцию run(), но эта функция не делает ничего особенного — она не обладает никакими особыми многопоточными свойствами.

Многопоточное выполнение программы обеспечивается классом Thread.

Программные потоки

Чтобы запустить объект Runnable в отдельном программном потоке, создайте объект Thread и передайте его конструктору указатель на Runnable. Тем самым обеспечивается инициализация программного потока и дальнейшее выполнение в нем функции run() класса Runnable под управлением операционной системы. Следующий пример, в котором Thread управляет работой LiftOff, показывает, как организуется запуск задачи в контексте программного потока:

```
//: C11:BasicThreads.cpp
// Простейшее использование класса Thread
//{L} ZThread
#include <iostream>
#include "LiftOff.h"
#include "zthread/Thread.h"
using namespace ZThread;
```

```
using namespace std;

int main() {
  try {
    Thread t(new LiftOff(10));
    cout << "Waiting for LiftOff" << endl;
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Класс Synchronization_Exception входит в библиотеку ZThreads и является базовым классом для всех исключений ZThreads. Это исключение запускается при возникновении ошибок запуска или использования потока.

Конструктору Thread передается только указатель на объект Runnable. При создании объекта Thread выполняется вся необходимая инициализация программного потока, после чего вызывается функция run() класса Runnable для запуска задачи. Хотя фактически конструктор Thread запускает долго выполняемую функцию, он быстро возвращает управление. Таким образом, вызванная в нашем примере функция LiftOff::run() продолжает работать, но поскольку она выполняется в другом программном потоке, программный поток main() продолжает выполнять другие операции (причем эта возможность не ограничивается потоком main() — из любого программного потока можно запустить другой программный поток). В этом нетрудно убедиться, запустив программу. Хотя main() передает управление функции LiftOff::run(), сообщение «Waiting for LiftOff» появляется до завершения обратного отсчета. Получается, что программа выполняет две функции сразу — LiftOff::run() и main().

В программу можно легко добавить новые программные потоки для решения новых подзадач. В следующем примере обратный отсчет ведется сразу в нескольких потоках:

```
//: C11:MoreBasicThreads.cpp
// Добавление новых программных потоков
//{L} ZThread
#include <iostream>
#include "LiftOff.h"
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

int main() {
  const int sz = 5;
  try {
    for(int i = 0; i < sz; i++)
      Thread t(new LiftOff(10, i));
    cout << "Waiting for LiftOff" << endl;
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Второй аргумент конструктора LiftOff предназначен для идентификации задач. Запустив программу, вы увидите, что система параллельно выполняет сразу несколько задач, последовательно передавая управление разным программным потокам. Переключение автоматически осуществляется планировщиком программ-

ных потоков. В многопроцессорных системах планировщик потоков распределяет потоки между процессорами.

Цикл for поначалу выглядит немного странно — переменная t создается локально в цикле, а затем немедленно выходит из области видимости и уничтожается. Создается впечатление, что программный поток тоже немедленно уничтожается, но выходные данные программы показывают, что на самом деле потоки продолжают работать. При создании объекта Thread программный поток регистрируется в системе, что обеспечивает продолжение его существования. Хотя стековый объект Thread теряется, сам поток продолжает жить вплоть до завершения задачи. С позиций C++ такое поведение выглядит противоестественно, но концепция программных потоков отходит от нормы: запуск потока создает отдельную «производственную» линию, которая продолжает работать после выхода из функции.

Ускорение реакции пользовательского интерфейса

Как уже отмечалось, многопоточная модель часто применяется для повышения скорости реакции пользовательского интерфейса. *Графические* интерфейсы в этой книге не рассматриваются, и ниже приводится простой пример с консольным интерфейсом.

Следующая программа читает строки из файла и выводит их на консоль, делая секундную паузу после вывода каждой строки (механизм организации задержки рассматривается далее в этой главе). Во время паузы программа не реагирует на ввод пользователя, ее интерфейс блокируется:

```
//: C11:UnresponsiveUI.cpp
// Блокировка пользовательского интерфейса
// в однопоточном приложении.
//{L} ZThread
#include <iostream>
#include <fstream>
#include <string>
#include "zthread/Thread.h"
using namespace std;
using namespace ZThread;

int main() {
  cout << "Press <Enter> to quit:" << endl;
  ifstream file("UnresponsiveUI.cpp");
  string line;
  while(getline(file, line)) {
    cout << line << endl;
    Thread::sleep(1000); // Время в миллисекундах
  }
  // Чтение ввода с консоли
  cin.get();
  cout << "Shutting down..." << endl;
} ///:~
```

Чтобы программа перестала блокироваться, задачу вывода файла можно выполнять в отдельном программном потоке. В этом случае главный поток отслеживает действия пользователя и реагирует на них:

```
//: C11:ResponsiveUI.cpp
// Многопоточная модель для ускорения реакции
// пользовательского интерфейса.
```

```
//{L} ZThread
#include <iostream>
#include <fstream>
#include <string>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

class DisplayTask : public Runnable {
  ifstream in;
  string line;
  bool quitFlag;
public:
  DisplayTask(const string& file) : quitFlag(false) {
    in.open(file.c_str());
  }
  ~DisplayTask() { in.close(); }
  void run() {
    while(getline(in, line) && !quitFlag) {
      cout << line << endl;
      Thread::sleep(1000);
    }
  }
  void quit() { quitFlag = true; }
};

int main() {
  try {
    cout << "Press <Enter> to quit:" << endl;
    DisplayTask* dt = new DisplayTask("ResponsiveUI.cpp");
    Thread t(dt);
    cin.get();
    dt->quit();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
  cout << "Shutting down..." << endl;
} ///:~
```

Теперь главный программный поток main() немедленно реагирует на нажатие клавиши Enter и вызывает функцию quit() для DisplayTask.

Данный пример также демонстрирует организацию взаимодействия между задачами — задача в программном потоке main() должна приказать задаче DisplayTask завершиться. Поскольку у нас имеется указатель на DisplayTask, кажется, что для завершения задачи достаточно вызвать оператор delete для этого указателя, однако такая программа будет работать ненадежно. Дело в том, что при уничтожении задача может выполнять какую-нибудь важную операцию, и ее прерывание приведет к нестабильному состоянию. В нашем примере задача сама решает, в какой момент она может безопасно завершиться. Чтобы оповестить задачу о необходимости завершиться, проще всего установить логический флаг. Достигнув точки стабильности, задача проверяет флаг и выполняет всю необходимую зачистку перед возвратом управления из run(). Выход из run() сообщает классу Thread о завершении задачи.

Наша программа достаточно проста, и никаких проблем с ней быть не должно. Тем не менее, в ней присутствуют мелкие недостатки, связанные с межпроцессными взаимодействиями. Эта важная тема будет рассматриваться далее в этой главе.

Исполнители

Специальные объекты библиотеки ZThreads, называемые *исполнителями*, позволяют упростить многопоточное программирование. Исполнители образуют дополнительный уровень абстракции между клиентом и задачей. При их использовании задачу запускает не сам клиент, а промежуточный объект.

Чтобы продемонстрировать эту возможность, мы воспользуемся классом Executor вместо явного создания объектов Thread в примере MoreBasicThreads.cpp. Объект LiftOff умеет решать конкретную задачу; по аналогии с паттерном Команда он предоставляет в распоряжение пользователя единственную функцию. Объект-исполнитель знает, как создать контекст для выполнения объектов Runnable. В следующем примере объект ThreadedExecutor создает отдельный программный поток для каждой задачи:

```
//: c11:ThreadedExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/ThreadedExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Иногда для создания и управления всеми программными потоками в системе достаточно одного объекта Executor. Многопоточный код в любом случае должен размещаться в блоке try, потому что при возникновении каких-либо проблем функция execute() объекта-исполнителя может запустить исключение Synchronization_Exception. Это относится ко всем функциям, изменяющим состояние объектов синхронизации, включая запуск потоков, захват мутексов, ожидание по условию и т. д. (см. далее в этой главе).

При завершении всех задач объектом-исполнителем программа немедленно завершается.

В предыдущем примере объект ThreadedExecutor создает программный поток для каждой запускаемой задачи, однако вы легко можете изменить режим выполнения задач, заменив ThreadedExecutor другим типом исполнителя. Для наших целей класс ThreadedExecutor вполне подходит, но в коммерческом коде его применение может привести к чрезмерным затратам на создание многочисленных программных потоков. В таких случаях лучше использовать класс PoolExecutor с ограниченным пулом программных потоков для параллельного выполнения задач:

```
//: C11:PoolExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/PoolExecutor.h"
#include "LiftOff.h"
```

```
using namespace ZThread;
using namespace std;

int main() {
  try {
    // В аргументе конструктора передается
    // минимальное количество программных потоков:
    PoolExecutor executor(5);
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

При использовании класса PoolExecutor дорогостоящие операции создания программных потоков выполняются заранее и только один раз, после чего программные потоки по мере возможности задействуются заново. Тем самым экономится время, потому что вам не приходится заново «платить» за создание потока для каждой отдельной задачи. Кроме того, в системах, управляемых событиями, события, для обработки которых необходимы потоки, создаются простой выборкой из пула. Ограниченное количество объектов Thread предотвращает чрезмерное потребление системных ресурсов. Таким образом, хотя в книге мы будем использовать объекты ThreadedExecutor, рассмотрите возможность применения класса PoolExecutor в окончательном варианте кода.

Класс ConcurrentExecutor ведет себя, как PoolExecutor с вырожденным пулом из одного программного потока. Он хорошо подходит для решения любых задач с длительным жизненным циклом, например, задачи прослушивания входящих подключений через сокеты. Кроме того, он удобен для выполнения коротких задач в отдельных потоках, например, мелких задач обновления локального или удаленного журнала, задач диспетчеризации событий.

Если ConcurrentExecutor получает сразу несколько задач, то следующая задача запускается только после завершения предыдущей, поскольку все эти задачи используют один программный поток. Таким образом, класс ConcurrentExecutor организует последовательное выполнение переданных ему задач:

```
//: C11:ConcurrentExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/ConcurrentExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    ConcurrentExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Класс SynchronousExecutor, как и ConcurrentExecutor, используется в ситуациях, когда из нескольких задач должна выполняться только одна. Но в отличие от

ConcurrentExecutor класс SynchronousExecutor не создает собственных программных потоков и не управляет ими. Он задействует поток, предоставивший задачу, а все его функции сводятся к синхронизации. Если n программных потоков поставляют задачи класса SynchronousExecutor, никакие две из этих задач не будут работать одновременно. Очередная задача отрабатывает до завершения, и только после этого стартует следующая задача в очереди.

Допустим, в нескольких потоках выполняются задачи, использующие файловую систему, но для сохранения переносимости программы вы не хотите блокировать файлы функцией flock() или другими специфическими функциями операционной системы. Одно из возможных решений — запустить эти задачи объектом SynchronousExecutor, чтобы в любой момент времени обрабатывалась только одна из них. В этом случае вам не придется заниматься синхронизацией доступа, а файловая система останется в сохранности. Конечно, правильнее было бы синхронизировать доступ к ресурсу (эта тема рассматривается далее), но объект SynchronousExecutor позволяет избавиться от лишних хлопот в черновом варианте программы.

```
//: C11:SynchronousExecutor.cpp
//{L} ZThread
#include <iostream>
#include "zthread/SynchronousExecutor.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

int main() {
  try {
    SynchronousExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new LiftOff(10, i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

При запуске программы вы увидите, что задачи выполняются в порядке предоставления, причем каждая задача завершается перед запуском следующей задачи. Однако мы не видим, чтобы в программе создавались новые потоки: для всех задач используется программный поток main(), поскольку в этом примере именно этот поток поставляет все задачи. Так как класс SynchronousExecutor в основном предназначен для построения прототипов, не стоит задействовать его в окончательной версии кода.

Передача управления

Если вы считаете, что очередная итерация цикла в функции run() (большинство функций run() содержат продолжительные циклы) выполнила достаточный объем работы, подскажите планировщику потоков, что процессор можно временно уступить другому программному потоку. Эта рекомендация (а это именно рекомендация — нет гарантий, что ваша реализация к ней прислушается) принимает форму функции yield().

Следующая измененная версия примера с задачей LiftOff уступает управление после каждой итерации:

```
//: C11:YieldingTask.cpp
// Вызов yield() определяет рекомендуемую
// точку переключения потоков
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class YieldingTask : public Runnable {
  int countDown;
  int id;
public:
  YieldingTask(int ident = 0) : countDown(5), id(ident) {}
  ~YieldingTask() {
    cout << id << " completed" << endl;
  }
  friend ostream&
  operator<<(ostream& os, const YieldingTask& yt) {
    return os << "#" << yt.id << ": " << yt.countDown;
  }
  void run() {
    while(true) {
      cout << *this << endl;
      if(--countDown == 0) return;
      Thread::yield();
    }
  }
};

int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new YieldingTask(i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Как видите, функция run() объекта задачи представляет собой бесконечный цикл. При вызове yield() вывод программы становится более равномерным, чем без него. Попробуйте закомментировать вызов Thread::yield() и проанализируйте результат. Впрочем, на практике функция yield() применяется крайне редко, и не стоит рассчитывать, что она сможет существенно повысить эффективность работы вашего приложения.

Приостановка

Другая операция, часто встречающаяся при управлении программными потоками, — их приостановка на заданное число миллисекунд функцией sleep(). Если в предыдущем примере заменить вызов yield() вызовом sleep(), программа принимает следующий вид:

```
//: C11:SleepingTask.cpp
// Приостановка программного потока функцией sleep().
```

```
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class SleepingTask : public Runnable {
  int countDown;
  int id;
public:
  SleepingTask(int ident = 0) : countDown(5), id(ident) {}
  ~SleepingTask() {
    cout << id << " completed" << endl;
  }
  friend ostream&
  operator<<(ostream& os, const SleepingTask& st) {
    return os << "#" << st.id << ": " << st.countDown;
  }
  void run() {
    while(true) {
      try {
        cout << *this << endl;
        if(--countDown == 0) return;
        Thread::sleep(100);
      } catch(Interrupted_Exception& e) {
        cerr << e.what() << endl;
      }
    }
  }
};

int main() {
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < 5; i++)
      executor.execute(new SleepingTask(i));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Функция Thread::sleep() может запустить исключение Interrupted_Exception (прерывания рассматриваются далее); это исключение перехватывается в run(). Но если задача создается и выполняется в блоке try функции main(), перехватывающем объекты Synchronization_Exception (базовый класс для всех исключений ZThreads), нельзя ли просто игнорировать исключение в run(), предположив, что оно перейдет в обработчик main()? Такое решение не работает, потому что исключения не возвращаются в main() через программные потоки. Следовательно, все возникающие исключения должны обрабатываться локально внутри задачи.

В общем случае порядок выполнения потоков непредсказуем, а это означает, что функция sleep() не может применяться для управления порядком их выполнения. Она просто приостанавливает выполнение потока на некоторое время. Гарантируется только то, что пауза продлится не менее 100 миллисекунд (в нашем примере), но может продолжаться и больше, потому что планировщик потоков должен еще добраться до приостановленного потока после истечения интервала. Для

управления порядком выполнения программных потоков лучше всего использовать средства синхронизации (см. далее) или в некоторых случаях вообще отказаться от многопоточности и написать собственные процедуры, передающие управление друг другу в заданном порядке.

Приоритеты

Приоритет программного потока определяет его относительную важность с точки зрения планировщика. Хотя порядок выполнения потоков не определен, планировщик *склонен* отдавать предпочтение ожидающим потокам с более высоким приоритетом. Тем не менее, это не означает, что потоки с более низкими приоритетами вообще не будут выполняться (в противном случае назначение высоких приоритетов приводило бы к возникновению блокировок). Просто планировщик реже передает управление низкоприоритетным потокам.

Здесь приведена измененная версия программы MoreBasicThreads.cpp, демонстрирующая работу системы приоритетов. Значения приоритетов регулируются функцией setPriority() класса Thread:

```
//: C11:SimplePriorities.cpp
// Приоритеты программных потоков.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

const double pi = 3.1415926538979323846;
const double e = 2.7182818284590452354;

class SimplePriorities : public Runnable {
  int countDown;
  volatile double d; // Без оптимизации
  int id;
public:
  SimplePriorities(int ident = 0): countDown(5),id(ident){}
  ~SimplePriorities() throw() {
    cout << id << " completed" << endl;
  }
  friend ostream&
  operator<<(ostream& os, const SimplePriorities& sp) {
    return os << "#" << sp.id << " priority: "
      << Thread().getPriority()
      << " count: "<< sp.countDown;
  }
  void run() {
    while(true) {
      // Дорогостоящая операция, которая может прерываться:
      for(int i = 1; i < 100000; i++)
        d = d + (M_PI + M_E) / (double)i;
      cout << *this << endl;
      if(--countDown == 0) return;
    }
  }
};

int main() {
```

```
  try {
    Thread high(new SimplePriorities);
    high.setPriority(High);
    for(int i = 0; i < 5; i++) {
      Thread low(new SimplePriorities(i));
      low.setPriority(Low);
    }
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:-
```

Оператор << переопределяется для вывода основных параметров задачи: идентификатора, приоритета и значения countDown.

Как видите, приоритет потока high устанавливается на максимальном уровне, а приоритеты всех остальных потоков находятся на минимальном уровне. Класс Executor в этом примере не используется, поскольку для задания приоритетов необходим прямой доступ к программным потокам.

Внутри функции SimplePriorities::run() 100 000 раз выполняются относительно дорогостоящие вещественные вычисления с суммированием и делением типа double. Переменная d объявлена подвижной (volatile), чтобы компилятор не пытался применять оптимизацию. Без этих вычислений эффект от назначения приоритетов остался бы незамеченным (попробуйте закомментировать цикл for с вещественными вычислениями). А так очевидно, что планировщик отдает большее предпочтение потоку high (по крайней мере, в системе Windows). Так как вычисления происходят достаточно долго, планировщик успевает вмешаться и переключить потоки с учетом приоритета, отдавая предпочтение потоку high.

Функция getPriority() возвращает приоритет существующего потока, а функция setPriority() позволяет сменить его в любой момент (а не только перед запуском потока, как в примере SimplePriorities.cpp).

Система приоритетов в значительной степени зависит от конкретной операционной системы. Например, Windows 2000 поддерживает семь уровней приоритета, а в системе Solaris фирмы Sun предусмотрено аж 2^{31} уровней. Существует только один переносимый вариант системы приоритетов очень большой гранулярности, например, Low, Medium и High, как в библиотеке ZThread.

Совместное использование ограниченных ресурсов

Однопоточную программу можно рассматривать как некую сущность, которая перемещается в пространстве задачи и в любой момент времени выполняет только одну операцию. В таких ситуациях можно не думать о проблемах, возникающих при одновременном обращении к ресурсу со стороны двух и более сущностей (когда двое людей пытаются одновременно припарковать машины в одном месте, пройти в одну дверь или просто поговорить).

Но в многопоточных приложениях нам уже приходится учитывать возможность того, что два и более программных потока попытаются одновременно использовать общий ресурс. Такие проблемы делятся на две категории. Во-первых, необходимые ресурсы могут не существовать. В C++ программист полностью контроли-

рует жизненный цикл своих объектов; ничто не мешает ему создать программный поток, который попытается использовать уже уничтоженные объекты.

Во-вторых, одновременное обращение к общему ресурсу может породить конфликт между потоками. Если не позаботиться о предотвращении таких конфликтов, два потока могут одновременно изменить состояние одного банковского счета, вывести данные на один принтер, отрегулировать состояние одного клапана и т. д.

В этом разделе рассматривается проблема исчезновения объектов во время их использования и проблема конфликта при обращениях к общим ресурсам. Вы познакомитесь со средствами решения этих проблем.

Гарантия существования объектов

Управление памятью и ресурсами занимает особое место в C++. При написании любых программ на C++ программист выбирает между созданием объектов в стеке и в куче (с помощью оператора new). В однопоточной программе жизненный цикл объектов легко отслеживается, и попытки использования ранее уничтоженных объектов встречаются крайне редко.

В примерах, приводимых в этой главе, объекты Runnable создаются в куче оператором new. Но обратите внимание: ни один из этих объектов не уничтожается *явно*. Тем не менее из выходных данных видно, что библиотека отслеживает каждую задачу и в конечном счете удаляет ее (об этом свидетельствует вызов деструкторов для объектов задач). Это происходит при выходе из Runnable::run() — возврат из run() означает, что задача прекращает свое существование.

Однако попытка возложить ответственность за уничтожение задачи на программный поток порождает проблемы. Поток не может знать, потребуется ли другому потоку обратиться к этому объекту Runnable, поэтому объект Runnable может быть уничтожен преждевременно. Для решения этой проблемы в библиотеке ZThreads организуется автоматический подсчет ссылок на задачи. Задача продолжает существовать до тех пор, пока счетчик ссылок на нее не упадет до нуля; в этот момент задача удаляется. Отсюда следует, что объекты задач всегда должны удаляться динамически, поэтому они не могут создаваться в стеке. Вместо этого задачи всегда создаются оператором new, как во всех примерах этой главы.

Нередко также приходится заботиться о том, чтобы другие объекты продолжали существовать до тех пор, пока они могут использоваться задачами. В противном случае эти объекты могут выйти из области видимости до завершения задач. Если это произойдет, попытки обращения к несуществующим объектам приведут к программным сбоям. Рассмотрим простой пример:

```
//: C11:Incrementer.cpp
// Уничтожение объектов до завершения программных потоков
// может вызвать серьезные проблемы.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class Count {
  enum { SZ = 100 };
```

```
  int n[SZ];
public:
  void increment() {
    for(int i = 0; i < SZ; i++)
      n[i]++;
  }
};

class Incrementer : public Runnable {
  Count* count;
public:
  Incrementer(Count* c) : count(c) {}
  void run() {
    for(int n = 100; n > 0; n--) {
      Thread::sleep(250);
      count->increment();
    }
  }
};

int main() {
  cout << "This will cause a segmentation fault!" << endl;
  Count count;
  try {
    Thread t0(new Incrementer(&count));
    Thread t1(new Incrementer(&count));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Класс Count на первый взгляд кажется лишним, но если использовать вместо массива простую переменную int, то компилятор может разместить ее в регистре, и эта память останется доступной после выхода объекта Count из области видимости (пусть это и незаконно с технической точки зрения). Это затруднило бы обнаружение недействительных обращений к памяти. Конкретный результат зависит от компилятора и операционной системы; попробуйте заменить n простой переменной типа int и посмотрите, что произойдет. В любом случае, если Count содержит массив int, компилятор вынужден разместить данные в стеке, а не в регистре.

Incrementer — простая задача, использующая объект Count. Внутри main() задачи Incrementer выполняются достаточно долго для того, чтобы объект Count вышел из области видимости, и задачи попытались обратиться к несуществующему объекту. При этом происходит сбой программы.

Чтобы уладить проблему, необходимо позаботиться о том, чтобы любые объекты продолжали существовать в течение всего времени использования этих объектов задачами (если бы объекты не требовались разным задачам, их можно было бы включить непосредственно в класс задачи и тем самым связать жизненный цикл объекта с жизненным циклом задачи). Поскольку в нашем случае жизненный цикл объекта не должен определяться статической областью видимости, мы размещаем объект в куче. А чтобы объект не был уничтожен, пока он требуется другим объектам, следует применить механизм подсчета ссылок.

Подсчет ссылок был достаточно подробно рассмотрен в первом томе книги; впрочем, он упоминался и на страницах этого тома. В библиотеку ZThread входит шаблон CountedPtr, который автоматически организует подсчет ссылок и вызывает

оператор delete для объекта при обнулении счетчика. Вот как выглядит предыдущий пример в случае использования шаблона CountedPtr:

```
//: C11:ReferenceCounting.cpp
// CountedPtr предотвращает преждевременное
// уничтожение объекта.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class Count {
  enum { SZ = 100 };
  int n[SZ];
public:
  void increment() {
    for(int i = 0; i < SZ; i++)
      n[i]++;
  }
};

class Incrementer : public Runnable {
  CountedPtr<Count> count;
public:
  Incrementer(const CountedPtr<Count>& c ) : count(c) {}
  void run() {
    for(int n = 100; n > 0; n--) {
      Thread::sleep(250);
      count->increment();
    }
  }
};

int main() {
  CountedPtr<Count> count(new Count);
  try {
    Thread t0(new Incrementer(count));
    Thread t1(new Incrementer(count));
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

В новой версии класс Incrementer содержит объект CountedPtr, управляющий использованием объекта Count. Внутри функции main() объекты CountedPtr передаются двум объектам Incrementer по значению, поэтому для них вызывается копирующий конструктор с увеличением счетчика ссылок. До тех пор пока задачи продолжают работать, счетчик ссылок остается ненулевым, и объект Count, находящийся под управлением CountedPtr, не уничтожается. Только после завершения всех задач, использующих Count, объект CountedPtr вызовет (автоматически) оператор delete для объекта Count.

Практически во всех случаях, когда объекты используются несколькими задачами, следует поставить эти объекты под контроль шаблона CountedPtr и предотвратить проблемы, связанные с жизненным циклом этих объектов.

Конфликты доступа к ресурсам

В следующем примере одна задача генерирует четные числа, а другие задачи эти числа поглощают. Единственная функция потоков-потребителей — проверка действительности (то есть четности) этих чисел.

Начнем с определения класса потока-потребителя EvenChecker, поскольку он будет нужен нам во всех последующих примерах. Чтобы отделить EvenChecker от разных типов генераторов, с которыми мы будем экспериментировать, создадим интерфейс Generator с минимально необходимыми функциями, о которых должен знать EvenChecker. Интерфейс содержит функцию nextValue() для получения следующего числа и функцию остановки генератора:

```
//: C11:EvenChecker.h
#ifndef EVENCHECKER_H
#define EVENCHECKER_H
#include <iostream>
#include "zthread/CountedPtr.h"
#include "zthread/Thread.h"
#include "zthread/Cancelable.h"
#include "zthread/ThreadedExecutor.h"

class Generator : public ZThread::Cancelable {
  bool canceled;
public:
  Generator() : canceled(false) {}
  virtual int nextValue() = 0;
  void cancel() { canceled = true; }
  bool isCanceled() { return canceled; }
};

class EvenChecker : public ZThread::Runnable {
  ZThread::CountedPtr<Generator> generator;
  int id;
public:
  EvenChecker(ZThread::CountedPtr<Generator>& g, int ident)
  : generator(g), id(ident) {}
  ~EvenChecker() {
    std::cout << "~EvenChecker " << id << std::endl;
  }
  void run() {
    while(!generator->isCanceled()) {
      int val = generator->nextValue();
      if(val % 2 != 0) {
        std::cout << val << " not even!" << std::endl;
        generator->cancel(); // Завершение всех задач EvenChecker
      }
    }
  }
  // Тестирование генератора произвольного типа:
  template<typename GenType> static void test(int n = 10) {
    std::cout << "Press Control-C to exit" << std::endl;
    try {
      ZThread::ThreadedExecutor executor;
      ZThread::CountedPtr<Generator> gp(new GenType);
      for(int i = 0; i < n; i++)
        executor.execute(new EvenChecker(gp, i));
    } catch(ZThread::Synchronization_Exception& e) {
```

```
      std::cerr << e.what() << std::endl;
    }
  }
};
#endif // EVENCHECKER_H ///:~
```

Класс Generator знакомит читателя с абстрактным классом Cancelable, входящим в библиотеку ZThread. Этот класс предоставляет единый интерфейс для изменения состояния объекта функцией cancel(), а также для проверки ее вызова в прошлом функцией isCanceled(). В нашем примере используется простое решение с логическим флагом — аналогом quitFlag из представленного ранее примера ResponsiveUI.cpp. Обратите внимание: в этом примере класс, реализующий Cancelable, не реализует Runnable. Вместо этого все задачи EvenChecker, зависящие от объекта Cancelable (Generator), проверяют, не были ли они отменены, как показано в run(). Таким образом, задачи, совместно использующие общий ресурс (задачу Generator, реализующую Cancelable), следят за тем, когда ресурс подаст сигнал на завершение. Тем самым устраняется так называемая *ситуация гонок*, когда две или более задачи независимо реагируют на некоторое условие, что приводит к конфликтам или нарушению логической целостности результатов. Вы должны тщательно продумать механизм защиты от всех возможных сбоев в многопоточной системе. Например, задача не может зависеть от другой задачи, потому что порядок завершения задач не гарантирован. Организуя зависимость задач от объектов (для которых реализован подсчет ссылок с использованием CounterPtr), мы устраняем потенциальную опасность гонок.

В следующих разделах мы познакомимся с более общими механизмами управления завершением потоков, поддерживаемыми библиотекой ZThread.

Так как Generator может совместно использоваться несколькими объектами EvenChecker, шаблон CountedPtr применяется для подсчета объектов Generator.

Последняя функция EvenChecker — статический шаблон функции класса, который готовит и выполняет проверку произвольного типа Generator. Для этого он создает экземпляр генератора в CounterPtr и запускает несколько экземпляров EvenChecker, задействующих этот генератор. Если при использовании Generator происходит сбой, функция test() сообщает об этом и возвращает управление; в противном случае генератор необходимо завершить нажатием клавиш Ctrl+C.

Задачи EvenChecker постоянно читают и проверяют значения, полученные от генератора. Обратите внимание: если выражение generator->isCanceled() истинно, функция run() возвращает управление, тем самым сообщая Executor в EvenChecker::test() о завершении задачи. Любая задача EvenChecker может вызвать cancel() для своего генератора, что приведет к корректному завершению всех остальных экземпляров EvenChecker, использующих этот генератор.

Класс EvenGenerator устроен очень просто — функция nextValue() выдает следующее четное значение:

```
//: C11:EvenGenerator.cpp
// Межпотоковые коллизии.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class EvenGenerator : public Generator {
```

```
  unsigned int currentEvenValue;  // Использование unsigned предотвращает
public:                           // потерю значимости.
  EvenGenerator() { currentEvenValue = 0; }
  ~EvenGenerator() { cout << "~EvenGenerator" << endl; }
  int nextValue() {
    ++currentEvenValue; // Опасный момент!
    ++currentEvenValue;
    return currentEvenValue;
  }
};

int main() {
  EvenChecker::test<EvenGenerator>();
} ///:~
```

Существует опасность того, что проверяющий программный поток вызовет nextValue() после первого, но до второго инкремента currentEvenValue (в точке, помеченной комментарием «Опасный момент!»). В результате будет получено неправильное (нечетное) значение. Чтобы доказать, что такое возможно, EvenChecker::test() создает группу объектов EvenChecker, которые постоянно читают выходные данные EvenGenerator и проверяют четность всех сгенерированных значений. Если среди них попадутся нечетные значения, программа выдает сообщение об ошибке и завершается.

Вполне возможно, что проблема проявится лишь при очень большом количестве циклов EvenGenerator; все зависит от особенностей операционной системы и других деталей реализации. Чтобы это произошло гораздо скорее, попробуйте включить вызов yield() между первым и вторым инкрементом. Так или иначе, рано или поздно ошибка *должна* произойти, потому что программные потоки EvenChecker могут обращаться к данным EvenGenerator, находящимся в «неправильном» состоянии.

Управление доступом

В предыдущем примере продемонстрирована фундаментальная проблема, возникающая при использовании программных потоков: программист не знает, в какой момент выполняется тот или иной поток. Представьте, что вы сидите за столом и готовитесь нацепить на вилку последний кусок колбасы, лежащий на тарелке. Но когда вы уже потянулись за ним, колбаса вдруг исчезает (потому что ваш программный поток был приостановлен, а в это время явился другой едок и все съел). Такие проблемы приходится решать при написании многопоточных программ.

Иногда вас совершенно не интересует, могут ли другие потоки обращаться к ресурсу во время его использования (например, при параллельном чтении данных). Но в большинстве случаев это существенно, и чтобы многопоточные приложения нормально работали, необходимы какие-то средства предотвращения одновременного доступа к ресурсу (хотя бы в критические моменты).

Предотвращение подобных коллизий основано на простой блокировке ресурса на время его использования. Первый поток, обращающийся к ресурсу, блокирует его, после чего другие потоки уже не могут обратиться к ресурсу до снятия блокировки. Когда ресурс освобождается, он блокируется другими потоком и т. д.

Итак, нам необходим механизм предотвращения обращений к памяти со стороны других задач в то время, пока эта память находится в «неподходящем» состоя-

нии. Другими словами, этот механизм должен *исключать* обращения к памяти в то время, пока она используется другой задачей. Концепция *взаимного исключения* занимает центральное место во всех многопоточных системах, а ее название обычно сокращается до *мутекса*[1]. В библиотеке ZThread объявления, относящиеся к механизму мутексов, находятся в заголовочном файле Mutex.h.

Чтобы решить проблему в представленной ранее программе, необходимо выделить *критические секции*, в которых должен действовать механизм взаимного исключения. Далее программный поток *захватывает* мутекс перед входом в критическую секцию и *освобождает* его в конце критической секции. В любой момент мутекс может быть захвачен только одним программным потоком, так что в результате достигается взаимное исключение:

```
//: C11:MutexEvenGenerator.cpp
// Предотвращение коллизий с применением мутексов.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Mutex.h"
using namespace ZThread;
using namespace std;

class MutexEvenGenerator : public Generator {
  int currentEvenValue;
  Mutex lock;
public:
  MutexEvenGenerator() { currentEvenValue = 0; }
  ~MutexEvenGenerator() {
    cout << "~MutexEvenGenerator" << endl;
  }
  int nextValue() {
    lock.acquire();
    ++currentEvenValue;
    Thread::yield();
    ++currentEvenValue;
    int rval = currentEvenValue;
    lock.release();
    return rval;
  }
};

int main() {
  EvenChecker::test<MutexEvenGenerator>();
} ///:~
```

Класс MutexEvenGenerator содержит мутекс (Mutex) с именем lock. Внутри функции nextValue() создается критическая секция, для чего используются функции acquire() и release(). Кроме того, между двумя инкрементами вставлен вызов yield(), чтобы повысить вероятность переключения контекста при нечетном значении currentEvenValue. Но поскольку мутекс ограничивает доступ к критической секции только одним потоком, сбоев не будет. Вызов yield() всего лишь ускоряет выявление ошибки, если бы она была теоретически возможна.

[1] От английского mutex. — *Примеч. ред.*

Обратите внимание: функция nextValue() должна сохранить возвращаемое значение внутри критической секции, потому что возврат управления из критической секции приведет к сохранению блокировки и невозможности ее повторного получения (обычно это приводит к ситуации *взаимной блокировки*, описанной в конце этой главы).

Первый программный поток, входящий в nextValue(), устанавливает блокировку. Все остальные потоки, пытающиеся получить доступ в критическую секцию, приостанавливаются до момента снятия блокировки первым потоком. Когда это произойдет, планировщик выбирает другой поток, ожидающий освобождения ресурса. Таким образом, секция программы, защищенная мутексом, в любой момент доступна только для одного потока.

Упрощенное программирование

Если в программе организуется обработка исключений, программирование мутексов резко усложняется. Чтобы мутексы всегда освобождались, необходимо проследить за тем, чтобы каждый возможный путь обработки исключения включал вызов release(). Кроме того, любая функция с несколькими путями возврата также должна позаботиться о вызове release() во всех соответствующих точках.

Все эти проблемы легко решаются благодаря наличию у стековых (автоматических) объектов деструктора, который всегда вызывается независимо от того, как именно происходит выход из области видимости функции. В библиотеке ZThread эта возможность реализована в форме шаблона Guard. Шаблон Guard создает объекты, называемые *стражами*, которые при конструировании захватывают объект Lockable вызовом acquire(), а при уничтожении освобождают его вызовом release(). Объекты Guard, созданные в локальном стеке, уничтожаются автоматически независимо от способа выхода из функции и всегда снимают блокировку с объекта Lockable. Здесь приводится реализация предыдущего примера с применением стражей:

```
//: C11:GuardedEvenGenerator.cpp
// Упрощенное программирование мутексов
// с применением шаблона Guard.
//{L} ZThread
#include <iostream>
#include "EvenChecker.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;
using namespace std;

class GuardedEvenGenerator : public Generator {
  int currentEvenValue;
  Mutex lock;
public:
  GuardedEvenGenerator() { currentEvenValue = 0; }
  ~GuardedEvenGenerator() {
    cout << "~GuardedEvenGenerator" << endl;
  }
  int nextValue() {
    Guard<Mutex> g(lock);
    ++currentEvenValue;
```

```
    Thread::yield();
    ++currentEvenValue;
    return currentEvenValue;
  }
};

int main() {
  EvenChecker::test<GuardedEvenGenerator>();
} ///:~
```

Обратите внимание: сохранять возвращаемое значение во временной переменной внутри nextValue() теперь не нужно. Обычно применение стражей сокращает объем программного кода и значительно снижает вероятность ошибок со стороны пользователей.

У шаблона Guard есть и другая интересная особенность: он позволяет безопасно манипулировать другими объектами-стражами. Например, второй объект Guard может использоваться для временного снятия блокировки:

```
//: C11:TemporaryUnlocking.cpp
// Временное снятие блокировки
// с использованием другого стража.
//{L} ZThread
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;

class TemporaryUnlocking {
  Mutex lock;
public:
  void f() {
    Guard<Mutex> g(lock);
    // Установление блокировки
    // ...
    {
      Guard<Mutex, UnlockedScope> h(g);
      // Снятие блокировки
      // ...
      // Установление блокировки
    }
    // ...
    // Снятие блокировки
  }
};

int main() {
  TemporaryUnlocking t;
  t.f();
} ///:~
```

Объекты Guard также могут использоваться, чтобы попытаться установить блокировку в течение заданного промежутка времени:

```
//: C11:TimedLocking.cpp
// Попытки установления блокировки
// в течение заданного интервала.
//{L} ZThread
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
```

```
#include "zthread/Guard.h"
using namespace ZThread;

class TimedLocking {
  Mutex lock;
public:
  void f() {
    Guard<Mutex, TimedLockedScope<500> > g(lock);
    // ...
  }
};

int main() {
  TimedLocking t;
  t.f();
} ///:~
```

Если в данном примере не удастся установить блокировку в течение 500 мс, запускается исключение Timeout_Exception.

Синхронизация целых классов

Библиотека ZThread содержит шаблон GuardedClass, предназначенный для автоматического создания синхронизированных оболочек для целых классов. Это подразумевает автоматическую защиту каждой функции класса:

```
//: C11:SynchronizedClass.cpp
//{L} ZThread
#include "zthread/GuardedClass.h"
using namespace ZThread;

class MyClass {
public:
  void func1() {}
  void func2() {}
};

int main() {
  MyClass a;
  a.func1(); // Не синхронизируется
  a.func2(); // Не синхронизируется
  GuardedClass<MyClass> b(new MyClass);
  // Синхронизированные вызовы, в любой момент времени
  // доступ разрешен только одному программному потоку:
  b->func1();
  b->func2();
} ///:~
```

Объект a не синхронизирован, поэтому функции func1() и func2() могут вызываться в любое время любым количеством потоков. Объект b защищен шаблоном GuardedClass, а, следовательно, все его функции автоматически синхронизируются, и в любой момент времени для каждого объекта может выполняться только одна функция.

Шаблон GuardedClass устанавливает блокировку на уровне класса, что может отразиться на быстродействии[1]. Если класс содержит несколько несвязанных функ-

[1] Причем весьма существенно. Обычно бывает достаточно защитить лишь небольшую часть функции. Установка защиты в точке входа в функцию часто делает критическую секцию длиннее, чем требуется.

ций, возможно, будет лучше организовать внутреннюю синхронизацию функций с разными блокировками. Но если вам приходится поступать подобным образом, это означает, что класс содержит группы данных, слабо связанные друг с другом. Подумайте, не стоит ли разбить такой класс на два.

Защита всех функций класса мутексами не гарантирует безопасности класса в плане работы с программными потоками. Чтобы обеспечить такую безопасность, необходимо тщательно продумать все потенциальные проблемы многопоточности.

Локальная память программных потоков

Второй способ решения проблем, обусловленных конфликтами доступа к общим ресурсам, заключается в отказе от совместного доступа как такового. В этом случае для одной переменной в каждом потоке, использующем объект, выделяется собственная копия памяти. Скажем, если пять программных потоков задействуют объект с переменной x, для x выделяется пять разных блоков памяти. К счастью, все операции создания и управления локальной памятью программных потоков в библиотеке ZThread осуществляются автоматически шаблоном ThreadLocal, как показано в примере:

```
//: C11:ThreadLocalVariables.cpp
// Автоматическое выделение локальной памяти
// для каждого программного потока.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/Cancelable.h"
#include "zthread/ThreadLocal.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class ThreadLocalVariables : public Cancelable {
  ThreadLocal<int> value;
  bool canceled;
  Mutex lock;
public:
  ThreadLocalVariables() : canceled(false) {
    value.set(0);
  }
  void increment() { value.set(value.get() + 1); }
  int get() { return value.get(); }
  void cancel() {
    Guard<Mutex> g(lock);
    canceled = true;
  }
  bool isCanceled() {
    Guard<Mutex> g(lock);
    return canceled;
  }
};

class Accessor : public Runnable {
  int id;
```

```
  CountedPtr<ThreadLocalVariables> tlv;
public:
  Accessor(CountedPtr<ThreadLocalVariables>& tl, int idn)
  : id(idn), tlv(tl) {}
  void run() {
    while(!tlv->isCanceled()) {
      tlv->increment();
      cout << *this << endl;
    }
  }
  friend ostream&
    operator<<(ostream& os, Accessor& a) {
    return os << "#" << a.id << ": " << a.tlv->get();
  }
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CountedPtr<ThreadLocalVariables>
      tlv(new ThreadLocalVariables);
    const int SZ = 5;
    ThreadedExecutor executor;
    for(int i = 0; i < SZ; i++)
      executor.execute(new Accessor(tlv, i));
    cin.get();
    tlv->cancel(); // Завершение всех задач Accessor
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

При создании объекта ThreadLocal посредством специализации шаблона для обращения к содержимому объекта могут использоваться только функции get() и set(). Функция get() возвращает копию объекта, связанного с программным потоком, а функция set() подставляет свой аргумент в объект, ассоциированный с данным потоком, и возвращает прежний объект. Примеры применения этих функций встречаются в функциях increment() и get() класса ThreadLocalVariables.

Так как объект tlv совместно используется несколькими объектами Accessor, он реализует интерфейс Cancelable, чтобы объектам Accessor можно было сообщить о завершении работы системы.

Запуск этой программы наглядно показывает, что каждому программному потоку выделяется отдельный блок памяти для хранения данных.

Завершение задач

В предыдущих примерах для завершения задач использовались флаги или интерфейс Cancelable. Обычно этого бывает достаточно, но в некоторых ситуациях нужно, чтобы задачи завершались быстрее. В этом разделе будут рассматриваться особенности и потенциальные проблемы, возникающие при таком завершении.

Начнем с программы, которая не только демонстрирует проблемы с ускоренным завершением, но и является дополнительным примером совместного использования ресурсов. Но прежде чем переходить к этому примеру, мы должны решить проблему коллизий в потоках ввода-вывода.

Предотвращение коллизий в потоках ввода-вывода

Возможно, вы заметили, что в предыдущих примерах выводимые данные иногда искажались. Система потоков ввода-вывода C++ проектировалась без учета многопоточности, поэтому ничто не мешало результатам одного потока смешаться с результатами другого. А это означает, что при написании приложений необходимо синхронизировать операции ввода-вывода.

Чтобы решить проблему, нужно сначала сгенерировать весь выходной пакет, а затем решить, когда отправить его на консоль. В одном из простых решений информация записывается в ostringstream, а затем один объект с мутексом синхронизирует вывод между всеми программными потоками:

```
//: C11:Display.h
// Предотвращение коллизий при выводе
#ifndef DISPLAY_H
#define DISPLAY_H
#include <iostream>
#include <sstream>
#include "zthread/Mutex.h"
#include "zthread/Guard.h"

class Display {          // Один объект совместно используется
  ZThread::Mutex iolock; // всеми потоками ввода-вывода.
public:
  void output(std::ostringstream& os) {
    ZThread::Guard<ZThread::Mutex> g(iolock);
    std::cout << os.str();
  }
};
#endif // DISPLAY_H ///:~
```

В этом варианте стандартные операторные функции operator<<() определены заранее, а объект строится в памяти с применением знакомых потоковых операторов. Когда задача хочет вывести сообщение, она создает временный объект ostringstream и использует его для построения нужного выходного сообщения. При вызове output() мутекс предотвращает запись в объект Display из нескольких программных потоков (как будет показано далее, в программе должен существовать только один объект Display).

Этот пример всего лишь демонстрирует базовый принцип, но при желании его можно усовершенствовать. Например, чтобы обеспечить выполнение требования о наличии только одного объекта Display в программе, можно преобразовать его в Синглет (в библиотеку ZThread входит шаблон Singleton, предназначенный для реализации Синглетов).

Подсчет посетителей

В следующей программе требуется узнать, сколько посетителей ежедневно заходит в парк. На каждом проходе установлена «вертушка» или похожий механизм. После увеличения отдельного счетчика вращений увеличивается общий счетчик, представляющий суммарное количество посетителей парка.

```
//: C11:OrnamentalGarden.cpp
//{L} ZThread
#include <vector>
```

```
#include <cstdlib>
#include <ctime>
#include "Display.h"
#include "zthread/Thread.h"
#include "zthread/FastMutex.h"
#include "zthread/Guard.h"
#include "zthread/ThreadedExecutor.h"
#include "zthread/CountedPtr.h"
using namespace ZThread;
using namespace std;

class Count : public Cancelable {
  FastMutex lock;
  int count;
  bool paused, canceled;
public:
  Count() : count (0), paused(false), canceled(false) {}
  int increment() {
    // Если закомментировать следующую строку, подсчет перестает работать:
    Guard<FastMutex> g(lock);
    int temp = count ;
    if(rand() % 2 == 0) // Передача управления в половине случаев
      Thread::yield();
    return (count  = ++temp);
  }
  int value() {
    Guard<FastMutex> g(lock);
    return count;
  }
  void cancel() {
    Guard<FastMutex> g(lock);
    canceled = true;
  }
  bool isCanceled() {
    Guard<FastMutex> g(lock);
    return canceled;
  }
  bool pause() {
    Guard<FastMutex> g(lock);
    paused = true;
  }
  bool isPaused() {
    Guard<FastMutex> g(lock);
    return paused;
  }
};

class Entrance : public Runnable {
  CountedPtr<Count> count;
  CountedPtr<Display> display;
  int number;
  int id;
  bool waitingForCancel;
public:
  Entrance(CountedPtr<Count>& cnt,
    CountedPtr<Display>& disp, int idn)
  : count(cnt), display(disp), id(idn), number(0),
    waitingForCancel(false) {}
  void run() {
```

```
    while(!count->isPaused()) {
      ++number;
      {
        ostringstream os;
        os << *this << " Total: "
           << count->increment() << endl;
        display->output(os);
      }
      Thread::sleep(100);
    }
    waitingForCancel = true;
    while(!count->isCanceled()) // Задержка...
      Thread::sleep(100);
    ostringstream os;
    os << "Terminating " << *this << endl;
    display->output(os);
  }
  int getValue() {
    while(count->isPaused() && !waitingForCancel)
      Thread::sleep(100);
    return number;
  }
  friend ostream&
  operator<<(ostream& os, const Entrance& e) {
    return os << "Entrance " << e.id << ": " << e.number;
  }
};

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  cout << "Press <ENTER> to quit" << endl;
  CountedPtr<Count> count(new Count);
  vector<Entrance*> v;
  CountedPtr<Display> display(new Display);
  const int SZ = 5;
  try {
    ThreadedExecutor executor;
    for(int i = 0; i < SZ; i++) {
      Entrance* task = new Entrance(count, display, i);
      executor.execute(task);
      // Сохранение указателя на задачу:
      v.push_back(task);
    }
    cin.get(); // Ждем, пока пользователь нажмет <Enter>
    count->pause(); // Остановка подсчета
    int sum = 0;
    vector<Entrance*>::iterator it = v.begin();
    while(it != v.end()) {
      sum += (*it)->getValue();
      ++it;
    }
    ostringstream os;
    os << "Total: " << count->value() << endl
       << "Sum of Entrances: " << sum << endl;
    display->output(os);
    count->cancel(); // Завершение программных потоков
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

В классе Count хранится суммарное количество посетителей парка. Единый объект Count, определенный в main() под именем count, хранится в классе Entrance под управлением CountedPtr, а, следовательно, совместно используется всеми объектами Entrance. В данном примере вместо обычного объекта Mutex задействован объект FastMutex с именем lock, потому что объект FastMutex опирается на поддержку мутексов операционной системы и дает более интересные результаты.

Объект Guard используется с lock в функции increment() для синхронизации доступа к count. Функция вызывает rand(), чтобы примерно в половине случаев между выборкой count в temp и сохранением temp в count включался вызов yield(). Если закомментировать определение объекта Guard, работа программы быстро нарушится, потому что сразу несколько потоков будут обращаться к переменной count и модифицировать ее одновременно.

Класс Entrance также содержит локальную переменную number с количеством посетителей, прошедших через данный вход. Это позволяет проверить объект count и убедиться в правильности количества зарегистрированных посетителей. Функция Entrance::run() просто увеличивает number с count и делает паузу продолжительностью в 100 миллисекунд.

В функции main() вектор vector<Entrance*> заполняется указателями на все созданные объекты Entrance. После нажатия пользователем клавиши Enter программа перебирает все объекты Entrance и суммирует их данные.

Программа тщательно следит за тем, чтобы завершение выполнялось корректно. Отчасти это делается для того, чтобы показать, какая осторожность необходима при завершении многопоточных программ, отчасти — чтобы продемонстрировать функцию interrupt(), о которой речь пойдет далее.

Все взаимодействие между объектами Entrance происходит в одном объекте Count. Когда пользователь нажимает клавишу Enter, функция main() отправляет count сообщение pause(). Так как все задачи Entrance::run() следят за приостановкой count, все объекты Entrance переходят в состояние waitingforCancel; в этом состоянии отсчет не производится, но объекты продолжают существовать. Это очень важно, поскольку функция main() должна обеспечить безопасный перебор объектов в векторе vector<Entrance*>. Обратите внимание: поскольку существует небольшая вероятность того, что перебор закончится раньше, чем Entrance завершит подсчет и перейдет в состояние waitingForCancel, функция getValue() в цикле вызывает функцию sleep() до тех пор, пока объект не перейдет в состояние waitingForCancel (это одна из форм так называемого *активного ожидания*, которое считается нежелательным; более предпочтительный подход с функцией wait() приведен далее). После того как main() завершит перебор вектора vector<Entrance*>, объекту count посылается сообщение cancel(), а все объекты Entrance снова ожидают изменения состояния. В этот момент они выводят сообщение о завершении и выходят из run(), что приводит к уничтожению всех задач механизмом многопоточности.

Запустите программу, и вы увидите, как по мере прохождения посетителей обновляются счетчики каждого входа, а также счетчик общего количества посетителей. Если закомментировать объект Guard в Count::increment(), вы заметите, что общее количество посетителей отличается от ожидаемого. Пока доступ к Counter синхронизируется при помощи мутекса, все работает нормально. Учтите, что Count::increment() нарочно увеличивает вероятность ошибки при помощи объекта temp и функции yield(). В реальных многопоточных программах вероят-

ность ошибки может оказаться статистически малой, поэтому у вас легко может создаться иллюзия, что все работает нормально. Как и в приведенном примере, могут существовать скрытые проблемы, с которыми вы просто еще не столкнулись, поэтому при написании многопоточного кода необходимо действовать предельно осторожно.

При возврате значения count функция Count::value() использует объект Guard для синхронизации. Это довольно любопытная подробность, хотя *вероятно*, что данный код будет работать в большинстве систем и компиляторов и без синхронизации. Дело в том, что в общем случае простые операции вроде возврата int являются *атомарными*; это означает, что они выполняются одной командой микропроцессора, которая не будет прерываться (механизм многопоточности не может прервать работу потока в середине выполнения команды микропроцессора). Иначе говоря, атомарные операции не прерываются механизмом многопоточности и не нуждаются в защите стражей[1]. Более того, если бы мы переместили выборку count в temp, удалили yield() и просто увеличивали count напрямую, скорее всего, блокировка вообще не понадобилась бы, поскольку операция инкремента тоже *обычно* является атомарной[2].

К сожалению, стандарт C++ не гарантирует атомарности всех этих операций. Хотя такие операции, как возврат и инкремент int, почти наверняка являются атомарными на большинстве компьютеров, утверждать это с полной уверенностью нельзя. А раз гарантий нет, значит, нужно предполагать самое худшее. Иногда программисты проверяют атомарность операций на конкретном компьютере (обычно на основании анализа ассемблерного кода) и пишут код, основанный на этих предположениях. Такой подход всегда рискован, и применять его не рекомендуется. О полученных результатах слишком легко забыть. Следующий программист может ошибочно предположить, что программа легко переносится на другой компьютер, и сойдет с ума, пытаясь выявить нерегулярные ошибки, вызванные многопоточными коллизиями.

Итак, хотя удаление стража из Count::value() вроде бы не отражается на работе программы, такой подход небезопасен. На некоторых компьютерах он может вызвать аномалии в поведении программ.

Завершение при блокировке

Функция Entrance::run() в предыдущем примере включает вызов sleep() в основном цикле. Мы знаем, что функция sleep() со временем даст программе продолжить работу, и задача достигнет начала цикла, где она сможет закончить проверять

---

[1] Это упрощенная формулировка. Иногда даже внешне безопасные атомарные операции оказываются небезопасными, поэтому вы должны очень хорошо подумать, принимая решение об отказе от синхронизации. Отказ от синхронизации часто является признаком поспешной оптимизации — того, что причиняет массу хлопот без сколько-нибудь заметного выигрыша (а то и вовсе без выигрыша).

[2] Атомарность операций — не единственный фактор. В многопроцессорных системах видимость играет гораздо более важную роль, чем в однопроцессорных. Изменения, внесенные одним потоком, даже атомарные в смысле непрерывности, могут остаться невидимыми для других потоков (например, храниться в локальном кэше процессора). Поэтому разные программные потоки будут по-разному представлять себе текущее состояние приложения. Механизм синхронизации обеспечивает распространение изменений, внесенных одним программным потоком, и их видимость во всем приложении, тогда как без синхронизации нельзя сказать, когда информация об изменениях дойдет до других потоков.

состояние isPaused(). Однако функция sleep() создает одну из ситуаций, в которых выполнение программного потока *блокируется*, а иногда возникает необходимость завершить заблокированную задачу.

Состояния потоков

Программный поток может находиться в одном из четырех состояний.

- *Создание.* Программный поток пребывает в этом состоянии в течение очень короткого времени. При этом потоку выделяются все необходимые системные ресурсы и проводится его инициализация, после чего ему может выделяться процессорное время. После завершения создания планировщик переводит поток в выполняемое или заблокированное состояние.
- *Готовность к выполнению.* Программный поток *может* выполняться, когда механизм квантования выделит для него процессорное время. Таким образом, поток, готовый к выполнению, может выполняться или не выполняться в конкретный момент времени, но ничто не мешает его выполнению, если планировщик сможет это устроить. Другими словами, поток, готовый к выполнению, не мертв и не заблокирован, даже если он в данный момент не выполняется.
- *Блокировка.* Программный поток мог бы выполняться, но что-то ему мешает (например, он ожидает завершения ввода-вывода). Пока поток находится в заблокированном состоянии, планировщик просто игнорирует его и не выделяет процессорное время. До перехода в состояние готовности поток не выполняет никаких операций.
- *Смерть.* Мертвый поток игнорируется планировщиком и не получает процессорного времени. Его задача завершена, и поток перестает быть готовым к выполнению. Стандартный способ смерти потока — выход из функции run().

Блокировка

Заблокированный поток не может продолжать работу. Ниже перечислены причины блокировки потоков.

- Поток был приостановлен вызовом sleep(*миллисекунды*); в этом случае он не будет выполняться в течение указанного времени.
- Выполнение потока было приостановлено вызовом wait(). Поток снова будет готов к выполнению лишь после получения сообщения signal() или broadcast(). Эти сообщения будут рассмотрены далее.
- Поток ожидает завершения операции ввода-вывода.
- Поток пытается войти в программную секцию, защищенную мутексом, но этот мутекс уже захвачен другим потоком.

Сейчас нас интересует одна проблема: иногда бывает нужно завершить поток, находящийся в заблокированном состоянии. Если нельзя подождать, пока поток возобновит работу и доберется до точки, в которой он может проверить состояние и завершиться по собственной инициативе, нужно принудительно вывести поток из состояния блокировки.

Прерывание

Как и следовало ожидать, выйти из середины функции Runnable::run() гораздо хлопотнее, чем дождаться момента, когда функция проверяет условие isCanceled() (или другого момента, когда программист посчитает возможным выйти из функции). В частности, при прерывании заблокированной задачи может возникнуть необходимость в уничтожении объектов и освобождении ресурсов. По этой причине выход из середины функции run() больше всего напоминает запуск исключения. В библиотеке ZThreads для аварийных прерываний такого рода тоже применяются исключения (причем в этом случае исключения часто запускаются для управления логикой программы, что обычно категорически не рекомендуется[1]). Чтобы при завершении задачи подобным образом происходил возврат к заведомо допустимому состоянию, необходимо тщательно проанализировать выполняемые ветви в вашей программе и выполнить всю необходимую зачистку в секции catch. Мы рассмотрим эти проблемы в данном разделе.

Для завершения заблокированных потоков в библиотеку ZThread включена функция Thread::interrupt(), устанавливающая *статус прерывания* для потока. Поток с установленным статусом прерывания запускает исключение Interrupted_Exception, если он уже заблокирован или попытается выполнить операцию блокировки. Статус прерывания сбрасывается при запуске исключения или при вызове задачей функции Thread::interrupted(). Как видите, функция Thread::interrupted() предоставляет второй способ выхода из цикла run() без запуска исключения.

Следующий пример демонстрирует простейшее применение функции interrupt():

```
//: C11:Interrupting.cpp
// Прерывание заблокированного потока
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

class Blocked : public Runnable {
public:
  void run() {
    try {
      Thread::sleep(1000);
      cout << "Waiting for get() in run():";
      cin.get();
    } catch(Interrupted_Exception&) {
      cout << "Caught Interrupted_Exception" << endl;
      // Выход из задачи
    }
  }
};

int main(int argc, char* argv[]) {
  try {
    Thread t(new Blocked);
```

[1] Тем не менее, исключения в ZThreads никогда не доставляются асинхронно, поэтому не существует опасности прерывания на середине выполнения команды или вызова функции. А если для захвата мутексов использовался шаблон Guard, при запуске исключения происходит автоматическое освобождение мутексов.

```
    if(argc > 1)
      Thread::sleep(1100);
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Как видно из листинга, помимо вывода в cout функция run() содержит две точки, в которых может произойти блокировка: вызов Thread::sleep(1000) и вызов cin.get(). Передавая программе любой аргумент командной строки, вы приказываете main() приостановиться на достаточно долгое время, чтобы задача завершила отработку sleep() и вызвала cin.get()[1]. Если программа запущена без аргумента, вызов sleep() в main() игнорируется. В нашем случае вызов interrupt() случится во время приостановки задачи, и вы увидите, что это приводит к запуску исключения Interrupted_Exception. Если передать программе аргумент командной строки, вы обнаружите, что задача не может быть прервана, если она была заблокирована при вводе-выводе. Прерывание работает для всех блокирующих операций, *кроме* ввода-вывода[2].

Это обстоятельство вызывает легкое замешательство при создании потоков, реализующих операции ввода-вывода, поскольку означает, что ввод-вывод теоретически может привести к блокировке многопоточных программ. Стоит напомнить, что эта проблема возникла из-за того, что язык C++ проектировался без учета многопоточности; наоборот, считается, что многопоточности как бы вообще не существует. Библиотека iostream не рассчитана на работу в многопоточных приложениях. Если в новый стандарт C++ будет включена поддержка многопоточности, вероятно, библиотеку iostream придется перерабатывать.

Блокировка по мутексу

При попытке вызвать функцию, мутекс которой уже захвачен, вызывающая задача приостанавливается до того момента, когда мутекс станет доступным. Следующая программа проверяет, допустимо ли прерывание блокировок этого вида:

```
//: C11:Interrupting2.cpp
// Прерывание программного потока, заблокированного
// в ожидании объекта синхронизации.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
using namespace ZThread;
using namespace std;

class BlockedMutex {
  Mutex lock;
public:
  BlockedMutex() {
    lock.acquire();
```

[1] Вообще говоря, sleep() гарантирует только минимальную, а не точно заданную задержку, поэтому возможно (хотя и маловероятно), что sleep(1100) активизируется раньше sleep(1000).

[2] В стандарте C++ не сказано, что прерывания не могут происходить во время операций ввода-вывода. Тем не менее в большинстве реализаций такая возможность не поддерживается.

```
  }
  void f() {
    Guard<Mutex> g(lock);
    // Никогда не будет доступен
  }
};

class Blocked2 : public Runnable {
  BlockedMutex blocked;
public:
  void run() {
    try {
      cout << "Waiting for f() in BlockedMutex" << endl;
      blocked.f();
    } catch(Interrupted_Exception& e) {
      cerr << e.what() << endl;
      // Завершение задачи
    }
  }
};

int main(int argc, char* argv[]) {
  try {
    Thread t(new Blocked2);
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Класс BlockedMutex содержит конструктор, который захватывает мутекс объекта и не освобождает его. Из-за этого попытка вызова f() всегда блокируется, поскольку захватить мутекс никогда не удастся. В Blocked2 функция run() будет остановлена при вызове blocked.f(). Если запустить программу, вы увидите, что, в отличие от предыдущего примера, interrupt() позволяет прервать вызов, заблокированный по ожиданию мутекса[1].

Проверка прерывания

При вызове interrupt() для программного потока прерывание происходит либо если задача входит в блокирующую операцию, либо если она уже заблокирована (как отмечалось ранее, исключение составляют операции ввода-вывода, которые не прерываются).

Но что, если написанный вами код может включать или не включать блокировку в зависимости от условий своего выполнения? Если бы выход всегда производился только по запуску исключения в блокирующем вызове, в некоторых случаях выход из цикла run() стал бы невозможным. Следовательно, если задача завершается вызовом interrupt(), необходимо предусмотреть *вторую* возможность выхода на тот случай, если в цикле run() не будет сделано ни одного блокирующего вызова.

Такую возможность предоставляет *статус прерывания*, устанавливаемый вызовом interrupt(). Для проверки статуса прерывания используется функция interrupted(). Она не только сообщает, вызывалась ли ранее функция interrupt(), но и сбрасывает статус прерывания. Сброс статуса прерывания гарантирует, что библиотека не

[1] Учтите, что вызов t.interrupt() может произойти раньше вызова blocked.f() (хотя это и маловероятно).

оповестит о прерывании задачи дважды. Вы будете оповещены либо одним вызовом Interrupted_Exception, либо одной успешной проверкой Thread::interrupted(). Если вам потребуется снова проверить, была ли прервана задача, сохраните результат, полученный при вызове Thread::interrupted().

В следующем примере показан типичный способ обработки обоих вариантов (с блокировкой и без) выхода из цикла в функции run() при установленном статусе прерывания:

```
//: C11:Interrupting3.cpp {RunByHand}
// Основная идиома прерывания задач.
//{L} ZThread
#include <iostream>
#include "zthread/Thread.h"
using namespace ZThread;
using namespace std;

const double PI = 3.14159265358979323846;
const double E = 2.7182818284590452354;

class NeedsCleanup {
  int id;
public:
  NeedsCleanup(int ident) : id(ident) {
    cout << "NeedsCleanup " << id << endl;
  }
  ~NeedsCleanup() {
    cout << "~NeedsCleanup " << id << endl;
  }
};

class Blocked3 : public Runnable {
  volatile double d;
public:
  Blocked3() : d(0.0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        point1:
        NeedsCleanup n1(1);
        cout << "Sleeping" << endl;
        Thread::sleep(1000);
        point2:
        NeedsCleanup n2(2);
        cout << "Calculating" << endl;
        // Продолжительная неблокирующая операция:
        for(int i = 1; i < 100000; i++)
          d = d + (PI + E) / (double)i;
      }
      cout << "Exiting via while() test" << endl;
    } catch(Interrupted_Exception&) {
      cout << "Exiting via Interrupted_Exception" << endl;
    }
  }
};

int main(int argc, char* argv[]) {
  if(argc != 2) {
    cerr << "usage: " << argv[0]
```

```
      << " delay-in-milliseconds" << endl;
    exit(1);
  }
  int delay = atoi(argv[1]);
  try {
    Thread t(new Blocked3);
    Thread::sleep(delay);
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Класс NeedsCleanup подчеркивает необходимость освобождения ресурсов при выходе из цикла через исключение. Обратите внимание на то, что в Blocked3::run() не определяются указатели, потому что по соображениям безопасности исключений все ресурсы должны инкапсулироваться в стековых объектах, чтобы обработчик исключений мог автоматически освободить их вызовом деструктора.

В командной строке программе передается аргумент, определяющий время задержки в миллисекундах перед вызовом interrupt(). Используя разные задержки, можно выйти из Blocked3::run() в разных точках цикла: в блокирующем вызове sleep() и в неблокирующих математических вычислениях. Вы увидите, что при вызове interrupt() после метки point2 (во время неблокирующей операции) сначала завершается цикл, затем уничтожаются локальные объекты и, наконец, происходит выход из цикла по условию while. Но если interrupt() вызывается между point1 и point2 (после команды while, но перед или во время блокирующей операции sleep()), выход из задачи происходит через Interrupted_Exception. В этом случае уничтожаются только те стековые объекты, которые были созданы до точки запуска исключения, и вам предоставляется возможность выполнить всю прочую зачистку в секции catch.

Класс, спроектированный для обработки interrupt(), должен обеспечить сохранение целостности своего состояния. Обычно это означает, что все захваты ресурсов должны инкапсулироваться в стековых объектах, чтобы деструкторы вызывались независимо от способа выхода из run(). При правильной реализации код выглядит достаточно элегантно. Вы можете создавать компоненты, которые полностью инкапсулируют свои механизмы синхронизации, но при этом реагируют на внешнее воздействие (через interrupt()) без включения специальных функций в интерфейс объекта.

Кооперация между программными потоками

Как было показано ранее, при параллельном запуске нескольких задач в разных программных потоках можно предотвратить нежелательный доступ задач к чужим ресурсам, для чего работа задач синхронизируется при помощи мутексов. Другими словами, если две задачи оспаривают друг у друга общий ресурс (обычно память), можно воспользоваться мутексом, чтобы в любой момент времени ресурс был доступен лишь для одной задачи.

Разобравшись с этим вопросом, можно переходить к вопросу кооперации потоков, то есть их совместной работе над некоторой проблемой. Поскольку проблема изменяется, нужно сделать так, чтобы потоки не просто не мешали друг другу,

а взаимодействовали с учетом общих интересов, поскольку решение некоторых частей задачи становится возможным лишь после решения всех предшествующих частей. Можно провести аналогию с планированием строительства: сначала нужно вырыть яму под фундамент будущего дома, но закладка стальных конструкций и построение бетонных форм могут производиться параллельно, причем заливка фундамента возможна лишь после решения обеих задач. Готовый фундамент должен быть на месте перед возведением каркаса здания, и т. д. Некоторые из этих этапов могут выполняться параллельно, но выполнение других становится возможным лишь после завершения всех предшествующих этапов.

Ключевая роль в кооперации задач отводится согласованию их действий. Согласование осуществляется уже знакомыми средствами: те же мутексы в данном случае гарантируют, что на сигнал ответит только одна задача. Тем самым устраняются все возможные ситуации гонок. В дополнение к мутексам добавляются средства приостановки задачи до изменения некоторого внешнего состояния («завершение строительства фундамента»), которое указывает, что задача может выполняться дальше. В настоящем разделе будут рассмотрены некоторые аспекты согласования задач, возникающие при этом проблемы и их решения.

Функции wait() и signal()

В библиотеку ZThreads входит класс Condition, упрощающий операции с мутексами и приостановку задач. Задача приостанавливается вызовом wait() для объекта Condition. Когда в программе изменяется некоторое условие, позволяющее задаче продолжить работу, вы оповещаете об этом либо отдельную задачу вызовом функции signal(), либо все задачи, приостановленные по данному объекту Condition, вызовом функции broadcast().

Функция wait() существует в двух формах. Первой форме передается интервал времени в миллисекундах, который имеет тот же смысл, что и при вызове sleep(): «приостановка в течение заданного промежутка времени». Вторая форма вызывается без аргументов и чаще используется на практике. Обе формы wait() освобождают мутекс, управляемый объектом Condition, и приостанавливают программный поток до вызова signal() или broadcast() для этого объекта. Первая форма также может завершиться по тайм-ауту (если заданный промежуток времени истечет до получения сигнала от функции signal() или broadcast()).

Освобождение мутекса функцией wait() означает, что мутекс может быть захвачен другим потоком. Иначе говоря, при вызове wait() вы говорите: «Я сделал все, что мог, и перехожу в ожидание; пусть поработают другие синхронизированные операции».

Обычно функция wait() используется для ожидания некоторого условия, не контролируемого текущей функцией (нередко это условие изменяется другим программным потоком). Но «крутить» холостой цикл с проверкой условия внутри программного потока было бы нежелательно; подобное активное ожидание обычно приводит лишь к напрасной трате процессорного времени. Поэтому функция wait() приостанавливает программный поток до изменения внешнего условия, а при вызове signal() или broadcast() (означающем, что в программе произошло нечто важное) программный поток «просыпается» и проверяет изменения. Таким образом, функция wait() обеспечивает механизм синхронизации действий между программными потоками.

Рассмотрим простой пример. Программа WaxOMatic.cpp содержит два процесса: первый (WaxOn) наносит на автомобиль защитное покрытие, а второй (WaxOff) его полирует. Второй процесс не может начать свою работу до завершения первого, а первый процесс должен дождаться завершения второго, чтобы нанести следующий слой покрытия. Оба процесса используют объект Car, содержащий объект Condition, который требуется для приостановки потока внутри waitForWaxing() и waitForBuffing():

```
//: C11:WaxOMatic.cpp {RunByHand}
// Простейшая кооперация программных потоков.
//{L} ZThread
#include <iostream>
#include <string>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

class Car {
  Mutex lock;
  Condition condition;
  bool waxOn;
public:
  Car() : condition(lock), waxOn(false) {}
  void waxed() {
    Guard<Mutex> g(lock);
    waxOn = true; // Готовность к полировке
    condition.signal();
  }
  void buffed() {
    Guard<Mutex> g(lock);
    waxOn = false; // Готовность к нанесению следующего слоя
    condition.signal();
  }
  void waitForWaxing() {
    Guard<Mutex> g(lock);
    while(waxOn == false)
      condition.wait();
  }
  void waitForBuffing() {
    Guard<Mutex> g(lock);
    while(waxOn == true)
      condition.wait();
  }
};

class WaxOn : public Runnable {
  CountedPtr<Car> car;
public:
  WaxOn(CountedPtr<Car>& c) : car(c) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        cout << "Wax On!" << endl;
        Thread::sleep(200);
```

```
        car->waxed();
        car->waitForBuffing();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Ending Wax On process" << endl;
  }
};

class WaxOff : public Runnable {
  CountedPtr<Car> car;
public:
  WaxOff(CountedPtr<Car>& c) : car(c) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        car->waitForWaxing();
        cout << "Wax Off!" << endl;
        Thread::sleep(200);
        car->buffed();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Ending Wax Off process" << endl;
  }
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CountedPtr<Car> car(new Car);
    ThreadedExecutor executor;
    executor.execute(new WaxOff(car));
    executor.execute(new WaxOn(car));
    cin.get();
    executor.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

В конструкторе Car мутекс инкапсулируется в объекте Condition и в дальнейшем используется для управления взаимодействием между задачами. Тем не менее, объект Condition не содержит информации о состоянии процесса, поэтому в класс необходимо включить дополнительные переменные. В нашем случае в Car добавляется флаг bool waxOn, обозначающий текущее состояние процесса (нанесение покрытия или полировка).

Функция waitForWaxing() проверяет флаг waxOn. Если флаг равен false, вызывающий поток приостанавливается вызовом wait() для объекта Condition. Важно, чтобы это происходило в защищенной секции, для которой программный поток установил блокировку (в нашем примере — созданием объекта Guard). При вызове wait() программный поток приостанавливается, а блокировка *снимается*. Снятие блокировки абсолютно необходимо, поскольку для безопасного изменения состояния объекта (например, для перевода waxOn в состояние true, без которого поток вообще не сможет продолжить работу) блокировка должна стать доступной для ее установления другой задачей. Когда в нашем примере другой поток вызывает waxed(), сообщая о завершении своей части работы, для перевода waxOn

в состояние true функция waxed() должна захватить мутекс. Затем waxed() вызывает для объекта Condition функцию signal(), что приводит к возобновлению потока, приостановленного вызовом wait(). Хотя функция signal() может вызываться в защищенной секции кода, как в нашем примере, это не является обязательным требованием[1].

Чтобы программный поток возобновил свою работу после вызова wait(), он должен сначала заново захватить мутекс, освобожденный при входе в wait(). Поток будет находиться в приостановленном состоянии до тех пор, пока мутекс не станет доступным.

Вызов wait() помещен в цикл while, проверяющий интересующее нас условие. Это важно по двум причинам[2].

- Не исключено, что при получении потоком сигнала от функции signal() изменилось другое условие, не связанное с вызовом wait(). В этом случае программный поток должен быть снова приостановлен до изменения нужного условия.
- К моменту выхода потока из wait() может оказаться, что другая задача сделала нечто такое, из-за чего текущий поток не может или не хочет продолжить работу. В этом случае он также должен быть повторно приостановлен вызовом wait().

Эти две причины должны учитываться при любом вызове wait(), поэтому вызовы wait() всегда следует заключать в циклы while с проверкой интересующего вас условия.

Функция waxOn::run() представляет первую стадию технологического процесса, поэтому она выполняет свою операцию (вызов sleep() имитирует время, необходимое для нанесения покрытия). Затем она сообщает объекту car о завершении своей стадии и вызывает функцию waitForBuffing(), которая приостанавливает поток функцией wait() до того момента, когда процесс WaxOff вызовет buffed() для объекта car, изменит состояние и оповестит об этом. С другой стороны, waxOff::run() немедленно переходит к вызову waitForWaxing(), а следовательно, приостанавливается до момента, когда WaxOn выполнит свою задачу и вызовет waxed(). Если запустить эту программу, вы увидите, что процесс повторяется по мере передачи управления между двумя потоками. При нажатии клавиши Enter функция interrupt() останавливает оба потока: когда interrupt() вызывается для объекта Executor, последний вызывает interrupt() для всех потоков, находящихся под его управлением.

---

[1] В отличие от языка Java, где для вызова notify() (аналога signal() в Java) необходимо предварительно установить блокировку. Хотя программные потоки стандарта Posix, взятые за основу при проектировании ZThread, не требуют установления блокировки для вызова signal() или broadcast(), часто это рекомендуется.

[2] На некоторых платформах существует третий способ выхода из wait(), называемый *спонтанной активизацией*. Фактически это означает, что программный поток может преждевременно выйти из состояния блокировки (во время ожидания условия или семафора) без получения signal() или broadcast(). Поток просто активизируется «сам по себе». Механизм спонтанной активизации существует из-за того, что на некоторых платформах программные потоки Posix или их эквиваленты реализуются далеко не так прямолинейно, как хотелось бы. Наличие спонтанной активизации упрощает построение библиотек (например, pthreads). В библиотеке ZThreads спонтанная активизация не поддерживается, поскольку библиотека решает все проблемы и скрывает их от пользователя.

Отношения поставщик-потребитель

В многопоточных программах часто моделируются отношения поставщик-потребитель, при которых одна задача создает объекты, а остальные задачи эти объекты потребляют. В подобных ситуациях необходимо позаботиться о том, чтобы (среди прочего) задачи-потребители не пропустили ни один из произведенных объектов.

Чтобы вы лучше поняли суть проблемы, рассмотрим автомат с тремя состояниями. В первом состоянии автомат подогревает тост, во втором намазывает его маслом, в третьем кладет джем на тост с маслом.

```
//: C11:ToastOMatic.cpp {RunByHand}
// Проблемы кооперации программных потоков.
//{L} ZThread
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

// Нанесение джема на тост с маслом:
class Jammer : public Runnable {
  Mutex lock;
  Condition butteredToastReady;
  bool gotButteredToast;
  int jammed;
public:
  Jammer() : butteredToastReady(lock) {
    gotButteredToast = false;
    jammed = 0;
  }
  void moreButteredToastReady() {
    Guard<Mutex> g(lock);
    gotButteredToast = true;
    butteredToastReady.signal();
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        {
          Guard<Mutex> g(lock);
          while(!gotButteredToast)
            butteredToastReady.wait();
          ++jammed;
        }
        cout << "Putting jam on toast " << jammed << endl;
        {
          Guard<Mutex> g(lock);
          gotButteredToast = false;
        }
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Jammer off" << endl;
```

```
  }
};

// Намазывание масла на тост:
class Butterer : public Runnable {
  Mutex lock;
  Condition toastReady;
  CountedPtr<Jammer> jammer;
  bool gotToast;
  int buttered;
public:
  Butterer(CountedPtr<Jammer>& j)
  : toastReady(lock), jammer(j) {
    gotToast = false;
    buttered = 0;
  }
  void moreToastReady() {
    Guard<Mutex> g(lock);
    gotToast = true;
    toastReady.signal();
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        {
          Guard<Mutex> g(lock);
          while(!gotToast)
            toastReady.wait();
          ++buttered;
        }
        cout << "Buttering toast " << buttered << endl;
        jammer->moreButteredToastReady();
        {
          Guard<Mutex> g(lock);
          gotToast = false;
        }
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Butterer off" << endl;
  }
};

class Toaster : public Runnable {
  CountedPtr<Butterer> butterer;
  int toasted;
public:
  Toaster(CountedPtr<Butterer>& b) : butterer(b) {
    toasted = 0;
  }
  void run() {
    try {
      while(!Thread::interrupted()) {
        Thread::sleep(rand()/(RAND_MAX/5)*100);
        // ...
        // Изготовление нового тоста
        // ...
        cout << "New toast " << ++toasted << endl;
        butterer->moreToastReady();
      }
```

```
    } catch(Interrupted_Exception&) { /* Выход */ }
    cout << "Toaster off" << endl;
  }
};

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  try {
    cout << "Press <Return> to quit" << endl;
    CountedPtr<Jammer> jammer(new Jammer);
    CountedPtr<Butterer> butterer(new Butterer(jammer));
    ThreadedExecutor executor;
    executor.execute(new Toaster(butterer));
    executor.execute(butterer);
    executor.execute(jammer);
    cin.get();
    executor.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Для упрощения опережающих ссылок мы определяем классы в порядке, обратном очередности их работы.

Каждый из классов Jammer и Butterer содержит мутекс, объект Condition и некоторую внутреннюю информацию состояния, изменение которой указывает, что процесс должен приостановить или возобновить свою работу (классу Toaster все это не нужно, потому что он поставляет объекты и ничего ждать не обязан). Две функции run() выполняют операцию, устанавливают флаг состояния, а затем приостанавливают задачу функцией wait(). Функции moreToastReady() и moreButteredToastReady() изменяют соответствующие флаги состояния, указывая, что ситуация изменилась и процесс должен рассмотреть возможность возобновления, а затем вызывают signal() для активизации программного потока.

Этот пример отличается от предыдущего тем, что в нем (по крайней мере, на концептуальном уровне) производятся объекты («тосты»). Частота создания объектов подвержена случайным отклонениям, чтобы в работе программы участвовал фактор неопределенности. Но при запуске выясняется, что автомат не работает, потому что многие тосты так и остаются в своем исходном виде: не намазываются ни маслом, ни джемом.

Решение проблем многопоточности с помощью очередей

Многие проблемы многопоточности связаны с необходимостью последовательного выполнения задач. Программа ToastOMatic.cpp должна не только последовательно обрабатывать создаваемые тосты, но и работать с одним тостом, не беспокоясь о том, что следующий тост тем временем упадет на пол. Нередко проблему удается решить использованием очереди с синхронизированным доступом к элементам:

```
//: C11:TQueue.h
#ifndef TQUEUE_H
#define TQUEUE_H
#include <deque>
#include "zthread/Thread.h"
#include "zthread/Condition.h"
```

```
#include "zthread/Mutex.h"
#include "zthread/Guard.h"

template<class T> class TQueue {
  ZThread::Mutex lock;
  ZThread::Condition cond;
  std::deque<T> data;
public:
  TQueue() : cond(lock) {}
  void put(T item) {
    ZThread::Guard<ZThread::Mutex> g(lock);
    data.push_back(item);
    cond.signal();
  }
  T get() {
    ZThread::Guard<ZThread::Mutex> g(lock);
    while(data.empty())
      cond.wait();
    T returnVal = data.front();
    data.pop_front();
    return returnVal;
  }
};
#endif // TQUEUE_H ///:~
```

Шаблон строится на базе контейнера deque из стандартной библиотеки C++ и дополняет его двумя новыми возможностями.

- Синхронизация гарантирует, что два программных потока не будут одновременно добавлять объекты в дек.
- Вызовы wait() и signal() обеспечивают автоматическую приостановку потребителей при пустой очереди и их активизацию при появлении новых элементов.

Этот относительно небольшой фрагмент кода решает на удивление много проблем[1].

В следующей простой тестовой программе организуется последовательная обработка объектов LiftOff. Потребителем является объект LiftOffRunner, который извлекает объекты LiftOff из контейнера TQueue и запускает их напрямую (то есть использует собственный программный поток прямым вызовом run() вместо того, чтобы порождать новый поток для каждой задачи).

```
//: C11:TestTQueue.cpp {RunByHand}
//{L} ZThread
#include <string>
#include <iostream>
#include "TQueue.h"
#include "zthread/Thread.h"
#include "LiftOff.h"
using namespace ZThread;
using namespace std;

class LiftOffRunner : public Runnable {
```

[1] Учтите, что если потребители по каким-то причинам прекратят свою работу, то поставщик будет выдавать новые объекты до исчерпания свободной памяти в системе. Чтобы этого не произошло, можно включить в программу ограничение на максимальное количество элементов и организовать блокировку поставщиков при переполнении очереди.

```
  TQueue<LiftOff*> rockets;
public:
  void add(LiftOff* lo) { rockets.put(lo); }
  void run() {
    try {
      while(!Thread::interrupted()) {
        LiftOff* rocket = rockets.get();
        rocket->run();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Exiting LiftOffRunner" << endl;
  }
};

int main() {
  try {
    LiftOffRunner* lor = new LiftOffRunner;
    Thread t(lor);
    for(int i = 0; i < 5; i++)
      lor->add(new LiftOff(10, i));
    cin.get();
    lor->add(new LiftOff(10, 99));
    cin.get();
    t.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Задачи помещаются в контейнер TQueue в функции main() и извлекаются из него в LiftOffRunner. Стоит отметить, что LiftOffRunner может игнорировать проблемы синхронизации, так как они решаются в TQueue.

Чтобы решить проблему в программе ToastOMatic.cpp, можно организовать хранение тостов в контейнере TQueue между процессами. А для этого нам понадобятся «полноценные» объекты тостов, способные хранить и выводить свое состояние:

```
//: C11:ToastOMaticMarkII.cpp {RunByHand}
// Решение проблем с использованием TQueue.
//{L} ZThread
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
#include "zthread/ThreadedExecutor.h"
#include "TQueue.h"
using namespace ZThread;
using namespace std;

class Toast {
  enum Status { DRY, BUTTERED, JAMMED };
  Status status;
  int id;
public:
  Toast(int idn) : status(DRY), id(idn) {}
  #ifdef __DMC__ // Ошибочно требует наличия конструктора по умолчанию
```

```
  Toast() { assert(0); } // Никогда не должен вызываться
  #endif
  void butter() { status = BUTTERED; }
  void jam() { status = JAMMED; }
  string getStatus() const {
    switch(status) {
      case DRY: return "dry";
      case BUTTERED: return "buttered";
      case JAMMED: return "jammed";
      default: return "error";
    }
  }
  int getId() { return id; }
  friend ostream& operator<<(ostream& os, const Toast& t) {
    return os << "Toast " << t.id << ": " << t.getStatus();
  }
};

typedef CountedPtr< TQueue<Toast> > ToastQueue;

class Toaster : public Runnable {
  ToastQueue toastQueue;
  int count;
public:
  Toaster(ToastQueue& tq) : toastQueue(tq), count(0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        int delay = rand()/(RAND_MAX/5)*100;
        Thread::sleep(delay);
        // Изготовление тоста
        Toast t(count++);
        cout << t << endl;
        // Постановка в очередь
        toastQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Toaster off" << endl;
  }
};

// Намазывание масла на тост:
class Butterer : public Runnable {
  ToastQueue dryQueue, butteredQueue;
public:
  Butterer(ToastQueue& dry, ToastQueue& buttered)
  : dryQueue(dry), butteredQueue(buttered) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Блокировка до появления следующего тоста:
        Toast t = dryQueue->get();
        t.butter();
        cout << t << endl;
        butteredQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Butterer off" << endl;
  }
```

```
};

// Нанесение джема на тост с маслом:
class Jammer : public Runnable {
  ToastQueue butteredQueue, finishedQueue;
public:
  Jammer(ToastQueue& buttered, ToastQueue& finished)
  : butteredQueue(buttered), finishedQueue(finished) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Блокировка до появления следующего тоста:
        Toast t = butteredQueue->get();
        t.jam();
        cout << t << endl;
        finishedQueue->put(t);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Jammer off" << endl;
  }
};

// Потребление тоста:
class Eater : public Runnable {
  ToastQueue finishedQueue;
  int counter;
public:
  Eater(ToastQueue& finished)
  : finishedQueue(finished), counter(0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Блокировка до появления следующего тоста:
        Toast t = finishedQueue->get();
        // Убеждаемся в том, что тосты следуют по порядку
        // и что все они намазаны джемом:
        if(t.getId() != counter++ ||
           t.getStatus() != "jammed") {
          cout << ">>>> Error: " << t << endl;
          exit(1);
        } else
          cout << "Chomp! " << t << endl;
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Eater off" << endl;
  }
};

int main() {
  srand(time(0)); // Раскрутка генератора случайных чисел
  try {
    ToastQueue dryQueue(new TQueue<Toast>),
               butteredQueue(new TQueue<Toast>),
               finishedQueue(new TQueue<Toast>);
    cout << "Press <Return> to quit" << endl;
    ThreadedExecutor executor;
    executor.execute(new Toaster(dryQueue));
    executor.execute(new Butterer(dryQueue,butteredQueue));
    executor.execute(
```

```
      new Jammer(butteredQueue, finishedQueue));
    executor.execute(new Eater(finishedQueue));
    cin.get();
    executor.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

В этом решении немедленно бросаются в глаза два обстоятельства: во-первых, применение контейнера TQueue кардинально сокращает объем и сложность кода в каждом классе Runnable, поскольку все операции по защите и взаимодействию потоков, а также вызовы wait() и signal() теперь выполняются в TQueue. В новой версии классы Runnable не содержат объекты Mutex и Condition. Во-вторых, в программе исчезла связь между классами, потому что каждый класс взаимодействует только со своим контейнером TQueue. Порядок определения классов теперь не имеет значения. Сокращение объема кода и смягчение привязки всегда приветствуются. Из этого можно сделать вывод, что применение контейнера TQueue дает положительный эффект, как и в большинстве подобных задач.

Функция broadcast()

Функция signal() активизирует один программный поток, находящийся в ожидании объекта Condition. Тем не менее, одного объекта могут дожидаться сразу несколько потоков, и в этом случае для их активизации лучше воспользоваться функцией broadcast() вместо signal().

В следующей программе объединены многие концепции, с которыми мы познакомились в этой главе. Рассмотрим гипотетическую автоматизированную линию сборки автомобилей. Объект Car строится в несколько этапов. В примере мы ограничимся одним этапом: когда на готовой раме закрепляются двигатель, трансмиссия и колеса. Объекты Car передаются из одного места в другое через контейнер CarQueue, который является разновидностью контейнера TQueue. Устройство управления (Director) берет очередной автомобиль (пока в виде голой рамы) из входной очереди CarQueue и помещает его на монтажный стенд (Cradle), где выполняются все операции. В этот момент Director сообщает (при помощи broadcast()) всем ожидающим роботам, что машина находится на месте и готова к монтажу. Три типа роботов начинают работу и отправляют Cradle сообщения о завершении своих задач. Director ожидает, когда все задачи будут завершены, после чего направляет Car в выходную очередь CarQueue для передачи на следующий этап сборки. Потребителем выходной очереди является объект Reporter, который просто выводит содержимое Car, чтобы мы могли убедиться в правильном завершении операций.

```
//: C11:CarBuilder.cpp {RunByHand}
// Использование функции broadcast().
//{L} ZThread
#include <iostream>
#include <string>
#include "zthread/Thread.h"
#include "zthread/Mutex.h"
#include "zthread/Guard.h"
#include "zthread/Condition.h"
```

```
#include "zthread/ThreadedExecutor.h"
#include "TQueue.h"
using namespace ZThread;
using namespace std;

class Car {
  int id;
  bool engine, driveTrain, wheels;
public:
  Car(int idn) : id(idn), engine(false),
  driveTrain(false), wheels(false) {}
  // Пустой объект Car:
  Car() : id(-1), engine(false),
  driveTrain(false), wheels(false) {}
  // Функции не синхронизируются -- предполагается,
  // что операции bool атомарны.
  int getId() { return id; }
  void addEngine() { engine = true; }
  bool engineInstalled() { return engine; }
  void addDriveTrain() { driveTrain = true; }
  bool driveTrainInstalled() { return driveTrain; }
  void addWheels() { wheels = true; }
  bool wheelsInstalled() { return wheels; }
  friend ostream& operator<<(ostream& os, const Car& c) {
    return os << "Car " << c.id << " ["
      << " engine: " << c.engine
      << " driveTrain: " << c.driveTrain
      << " wheels: " << c.wheels << " ]";
  }
};

typedef CountedPtr< TQueue<Car> > CarQueue;

class ChassisBuilder : public Runnable {
  CarQueue carQueue;
  int counter;
public:
  ChassisBuilder(CarQueue& cq) : carQueue(cq),counter(0) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        Thread::sleep(1000);
        // Создание рамы:
        Car c(counter++);
        cout << c << endl;
        // Постановка рамы в очередь
        carQueue->put(c);
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "ChassisBuilder off" << endl;
  }
};

class Cradle {
  Car c; // Машина, собираемая в данный момент
  bool occupied;
  Mutex workLock, readyLock;
  Condition workCondition, readyCondition;
  bool engineBotHired, wheelBotHired, driveTrainBotHired;
```

```
public:
  Cradle()
  : workCondition(workLock), readyCondition(readyLock) {
    occupied = false;
    engineBotHired = true;
    wheelBotHired = true;
    driveTrainBotHired = true;
  }
  void insertCar(Car chassis) {
    c = chassis;
    occupied = true;
  }
  Car getCar() { // Выборка производится только один раз
    if(!occupied) {
      cerr << "No Car in Cradle for getCar()" << endl;
      return Car(); // "Пустой" объект Car
    }
    occupied = false;
    return c;
  }
  // Обращение к машине на монтажном стенде:
  Car* operator->() { return &c; }
  // Для предложения роботами своих услуг:
  void offerEngineBotServices() {
    Guard<Mutex> g(workLock);
    while(engineBotHired)
      workCondition.wait();
    engineBotHired = true; // Принять задание
  }
  void offerWheelBotServices() {
    Guard<Mutex> g(workLock);
    while(wheelBotHired)
      workCondition.wait();
    wheelBotHired = true; // Принять задание
  }
  void offerDriveTrainBotServices() {
    Guard<Mutex> g(workLock);
    while(driveTrainBotHired)
      workCondition.wait();
    driveTrainBotHired = true; // Принять задание
  }
  // Оповещение ожидающих роботов о наличии работы:
  void startWork() {
    Guard<Mutex> g(workLock);
    engineBotHired = false;
    wheelBotHired = false;
    driveTrainBotHired = false;
    workCondition.broadcast();
  }
  // Каждый робот сообщает о выполнении своей задачи:
  void taskFinished() {
    Guard<Mutex> g(readyLock);
    readyCondition.signal();
  }
  // Director ждет, пока будут завершены все задачи:
  void waitUntilWorkFinished() {
    Guard<Mutex> g(readyLock);
    while(!(c.engineInstalled() && c.driveTrainInstalled()
            && c.wheelsInstalled()))
```

```
      readyCondition.wait();
    }
  };

  typedef CountedPtr<Cradle> CradlePtr;

  class Director : public Runnable {
    CarQueue chassisQueue, finishingQueue;
    CradlePtr cradle;
  public:
    Director(CarQueue& cq, CarQueue& fq, CradlePtr cr)
    : chassisQueue(cq), finishingQueue(fq), cradle(cr) {}
    void run() {
      try {
        while(!Thread::interrupted()) {
          // Блокировка до появления рамы:
          cradle->insertCar(chassisQueue->get());
          // Оповещение роботов о наличии работы:
          cradle->startWork();
          // Ожидание завершения работы
          cradle->waitUntilWorkFinished();
          // Включение машины в очередь для продолжения работы
          finishingQueue->put(cradle->getCar());
        }
      } catch(Interrupted_Exception&) { /* Выход */ }
      cout << "Director off" << endl;
    }
  };

  class EngineRobot : public Runnable {
    CradlePtr cradle;
  public:
    EngineRobot(CradlePtr cr) : cradle(cr) {}
    void run() {
      try {
        while(!Thread::interrupted()) {
          // Блокировка до предложения/принятия задачи:
          cradle->offerEngineBotServices();
          cout << "Installing engine" << endl;
          (*cradle)->addEngine();
          cradle->taskFinished();
        }
      } catch(Interrupted_Exception&) { /* Exit */ }
      cout << "EngineRobot off" << endl;
    }
  };

  class DriveTrainRobot : public Runnable {
    CradlePtr cradle;
  public:
    DriveTrainRobot(CradlePtr cr) : cradle(cr) {}
    void run() {
      try {
        while(!Thread::interrupted()) {
          // Блокировка до предложения/принятия задачи:
          cradle->offerDriveTrainBotServices();
          cout << "Installing DriveTrain" << endl;
          (*cradle)->addDriveTrain();
          cradle->taskFinished();
```

```
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "DriveTrainRobot off" << endl;
  }
};

class WheelRobot : public Runnable {
  CradlePtr cradle;
public:
  WheelRobot(CradlePtr cr) : cradle(cr) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        // Блокировка до предложения/принятия задачи:
        cradle->offerWheelBotServices();
        cout << "Installing Wheels" << endl;
        (*cradle)->addWheels();
        cradle->taskFinished();
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "WheelRobot off" << endl;
  }
};

class Reporter : public Runnable {
  CarQueue carQueue;
public:
  Reporter(CarQueue& cq) : carQueue(cq) {}
  void run() {
    try {
      while(!Thread::interrupted()) {
        cout << carQueue->get() << endl;
      }
    } catch(Interrupted_Exception&) { /* Exit */ }
    cout << "Reporter off" << endl;
  }
};

int main() {
  cout << "Press <Enter> to quit" << endl;
  try {
    CarQueue chassisQueue(new TQueue<Car>),
             finishingQueue(new TQueue<Car>);
    CradlePtr cradle(new Cradle);
    ThreadedExecutor assemblyLine;
    assemblyLine.execute(new EngineRobot(cradle));
    assemblyLine.execute(new DriveTrainRobot(cradle));
    assemblyLine.execute(new WheelRobot(cradle));
    assemblyLine.execute(
      new Director(chassisQueue, finishingQueue, cradle));
    assemblyLine.execute(new Reporter(finishingQueue));
    // Запускаем систему, начиная производство рам:
    assemblyLine.execute(new ChassisBuilder(chassisQueue));
    cin.get();
    assemblyLine.interrupt();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

В классе Car имеет место одно упрощение: предполагается, что операции bool являются атомарными. Как уже отмечалось, это предположение достаточно безопасно, и все же оно должно быть тщательно продумано. Каждый объект Car начинает свое существование в виде простой рамы. Несколько роботов прикрепляют к нему разные части, вызывая соответствующие функции add после завершения своей работы.

Класс ChassisBuilder просто создает новый объект Car каждую секунду и помещает его в очередь chassisQueue. Объект Director управляет процессом сборки: он извлекает очередной объект Car из chassisQueue, помещает его на монтажный стенд (Cradle), приказывает всем роботам начать работу функцией startWork() и приостанавливается вызовом waitUntilWorkFinished(). После завершения сборки Director извлекает объект Car из монтажного стенда и помещает его в очередь finishingQueue.

Объект Cradle занимает центральное место в системе сигналов. Объекты Mutex и Condition управляют как работой роботов, так и передачей информации о завершении всех операций. Робот некоторого типа предлагает свои услуги объекту Cradle, вызывая соответствующую функцию offer. Далее программный поток робота приостанавливается до момента, когда объект Director вызовет startWork(). При этом сбрасываются флаги занятости, а вызов broadcast() приказывает всем роботам явиться для получения задания. Хотя эта система позволяет любому количеству роботов предложить свои услуги, она приводит к приостановке программного потока каждого из роботов. Можно представить более совершенную систему, в которой роботы регистрируются в разных объектах Cradle без приостановки, а в свободное время находятся в общем пуле и ожидают первый объект Cradle, у которого появится работа.

Завершив свою задачу (с изменением состояния Car), робот вызывает функцию taskFinished(), которая сигнализирует об условии readyCondition; именно это условие ожидается объектом Director в функции waitUntilWorkFinished(). При каждой активизации управляющего программного потока проверяется состояние объекта Car, и если он еще не закончен, программный поток снова приостанавливается.

Когда Director помещает объект Car в Cradle, с этим объектом Car можно выполнять операции при помощи оператора ->. Для предотвращения повторной выборки одного объекта используется флаг occupied; если он равен false, программа выдает сообщение об ошибке (в библиотеке ZThreads исключения не распространяются между программными потоками).

Функция main() создает все необходимые объекты и инициализирует задачи. Задача ChassisBuilder запускается последней (впрочем, благодаря контейнеру TQueue она с таким же успехом могла бы запускаться первой). Программа выполняет все рекомендации, относящиеся к жизненному циклу объектов и задач, поэтому ее завершение проходит безопасно.

Взаимная блокировка

Так как программные потоки могут блокироваться, а с объектами могут связываться мутексы, не позволяющие потокам обращаться к объекту до освобождения мутекса, один поток может быть приостановлен в ожидании другого потока, который в свою очередь ожидает третьего потока и т. д., пока цепочка не вернется к самому первому объекту. Возникает замкнутый цикл из потоков, каждый из кото-

рых ожидает другого потока и не может сдвинуться с места. Такая ситуация называется *взаимной блокировкой*.

Если взаимная блокировка возникает сразу же после запуска программы, вы тут же узнаете о возникшей проблеме и сможете исправить ошибку. Настоящие проблемы начинаются, если программа на первый взгляд работает нормально, но в ней скрывается потенциальная возможность взаимной блокировки. В этом случае вы и не подозреваете о возможности взаимной блокировки, пока она неожиданно не проявится на компьютере клиента (причем, скорее всего, вам не удастся легко воспроизвести проблему). Можно сделать вывод, что предотвращение взаимной блокировки посредством тщательного проектирования является важным аспектом разработки многопоточных программ.

Рассмотрим классический пример взаимной блокировки, предложенный Эдгаром Дейкстрой (Edsger Dijkstra). Это задача об *обедающих философах*. В базовом варианте задачи речь шла о пяти философах, но в нашем случае философов может быть сколько угодно. Философы тратят время на размышления и на еду. Когда философ думает, он не нуждается ни в каких общих ресурсах, хотя количество столовых принадлежностей ограниченно. В исходной постановке задачи такими принадлежностями были вилки. У Дейкстры философы ели спагетти из блюда в середине стола двумя вилками, но логика подсказывает, что вилки лучше заменить палочками для еды. Разумеется, для еды нужны две палочки.

В задаче кроется одна сложность: из-за своей бедности у философов хватает денег только на пять палочек. Палочки разложены на столе между философами. Когда философ хочет есть, он берет палочки, лежащие слева и справа от него. Если какая-либо из палочек уже используется кем-то другим, нашему философу придется подождать, пока она освободится.

```
//: C11:DiningPhilosophers.h
// Классы для задачи об обедающих философах.
#ifndef DININGPHILOSOPHERS_H
#define DININGPHILOSOPHERS_H
#include <string>
#include <iostream>
#include <cstdlib>
#include "zthread/Condition.h"
#include "zthread/Guard.h"
#include "zthread/Mutex.h"
#include "zthread/Thread.h"
#include "Display.h"

class Chopstick {
  ZThread::Mutex lock;
  ZThread::Condition notTaken;
  bool taken;
public:
  Chopstick() : notTaken(lock), taken(false) {}
  void take() {
    ZThread::Guard<ZThread::Mutex> g(lock);
    while(taken)
      notTaken.wait();
    taken = true;
  }
  void drop() {
    ZThread::Guard<ZThread::Mutex> g(lock);
```

```
    taken = false;
    notTaken.signal();
  }
};

class Philosopher : public ZThread::Runnable {
  Chopstick& left;
  Chopstick& right;
  int id;
  int ponderFactor;
  ZThread::CountedPtr<Display> display;
  int randSleepTime() {
    if(ponderFactor == 0) return 0;
    return rand()/(RAND_MAX/ponderFactor) * 250;
  }
  void output(std::string s) {
    std::ostringstream os;
    os << *this << " " << s << std::endl;
    display->output(os);
  }
public:
  Philosopher(Chopstick& l, Chopstick& r,
  ZThread::CountedPtr<Display>& disp, int ident,int ponder)
  : left(l), right(r), id(ident), ponderFactor(ponder),
    display(disp) {}
  virtual void run() {
    try {
      while(!ZThread::Thread::interrupted()) {
        output("thinking");
        ZThread::Thread::sleep(randSleepTime());
        // Философ голоден
        output("grabbing right");
        right.take();
        output("grabbing left");
        left.take();
        output("eating");
        ZThread::Thread::sleep(randSleepTime());
        right.drop();
        left.drop();
      }
    } catch(ZThread::Synchronization_Exception& e) {
      output(e.what());
    }
  }
  friend std::ostream&
  operator<<(std::ostream& os, const Philosopher& p) {
    return os << "Philosopher " << p.id;
  }
};
#endif // DININGPHILOSOPHERS_H ///:~
```

Два философа (объекты Philosopher) не могут взять (take()) палочку (объект Chopstick) одновременно, поскольку функция take() синхронизируется при помощи мутекса. Кроме того, если палочка уже находится у кого-то другого, философ ожидает ее освобождения вызовом функции drop() (причем этот вызов тоже должен синхронизироваться для предотвращения ситуации гонок и обеспечения видимости изменений в многопроцессорных системах).

Каждый объект Philosopher содержит ссылки на два объекта Chopstick (левая и правая палочки). Через эти ссылки он пытается взять палочки. Объект Philosopher одну часть времени думает, а другую — ест, и это обстоятельство отражено в функции main(). Нетрудно заметить, что если философы будут слишком мало думать, то при попытках поесть у них возникнет конкуренция за палочки, и взаимная блокировка случится гораздо быстрее. Чтобы вам было удобнее экспериментировать, переменная ponderFactor определяет отношение между затратами времени на размышления и еду. Малые значения ponderFactor повышают вероятность взаимной блокировки.

В функции Philosopher::run() все объекты Philosopher непрерывно думают и едят. Объект Philosopher думает в течение случайного промежутка времени, после чего пытается функцией take() взять правую (right) и левую (left) палочки. Далее он ест в течение случайного промежутка времени, и весь процесс повторяется заново. Вывод на консоль синхронизируется, о чем рассказывалось ранее в этой главе.

Задача обедающих философов доказывает, что даже правильно работающая (на первый взгляд) программа может быть подвержена взаимным блокировкам. Чтобы вы могли убедиться в этом, аргумент командной строки изменяет долю времени, затрачиваемую философом на размышления. Если философов много или они тратят большую часть времени на раздумья, возможно, вы никогда не столкнетесь с взаимной блокировкой, хотя ее теоретическая возможность остается. Если аргумент командной строки равен нулю, взаимная блокировка обычно возникает довольно быстро[1]:

```
//: C11:DeadlockingDiningPhilosophers.cpp {RunByHand}
// Обедающие философы и взаимная блокировка.
//{L} ZThread
#include <ctime>
#include "DiningPhilosophers.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

int main(int argc, char* argv[]) {
  srand(time(0)); // Раскрутка генератора случайных чисел
  int ponder = argc > 1 ? atoi(argv[1]) : 5;
  cout << "Press <ENTER> to quit" << endl;
  enum { SZ = 5 };
  try {
    CountedPtr<Display> d(new Display);
    ThreadedExecutor executor;
    Chopstick c[SZ];
    for(int i = 0; i < SZ; i++) {
      executor.execute(
        new Philosopher(c[i], c[(i+1) % SZ], d, i,ponder));
    }
    cin.get();
    executor.interrupt();
    executor.wait();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

[1] На момент написания книги поддержка многопоточности в проекте Cygwin (www.cygwin.com) изменялась и совершенствовалась, но нам так и не удалось наблюдать взаимную блокировку в текущей версии Cygwin. С другой стороны, в системе Linux программа довольно быстро выходила на взаимную блокировку.

Объектам Chopstick не нужны внутренние идентификаторы, они идентифицируются своей позицией в массиве c. Каждому объекту Philosopher при конструировании назначается ссылка на левый и правый объекты Chopstick; это те самые столовые принадлежности, без которых философ не может есть. Каждый объект Philosopher, кроме последнего, инициализируется размещением между следующей парой объектов Chopstick. Последнему объекту Philosopher в «правой» ссылке назначается нулевой объект Chopstick, и круг замыкается (последний философ сидит рядом с первым, а нулевая палочка лежит между ними). При таком расположении в какой-то момент может возникнуть ситуация, когда все философы попытаются начать есть и будут ждать, пока их сосед положит палочку. В программе происходит взаимная блокировка.

Если программные потоки (философы) тратят на другие задачи (на размышления) больше времени, чем на еду, вероятность одновременного доступа к общим ресурсам (палочкам) существенно снижается. При ненулевом значении ponderFactor вы даже можете убедить себя, что программа не содержит ошибок, хотя на самом деле это не так.

Чтобы решить проблему, вы должны понять, что взаимная блокировка возникает при одновременном выполнении четырех условий.

- Взаимное исключение. По крайней мере, один ресурс, требующийся потокам, должен исключать одновременный доступ. В нашем примере каждая палочка одновременно может использоваться только одним философом.
- По крайней мере, один процесс должен удерживать ресурс и ожидать получения другого ресурса, захваченного другим процессом. Иначе говоря, чтобы возникла блокировка, философ должен держать одну палочку и ожидать освобождения другой.
- Ресурс не может быть принудительно отнят у процесса. Освобождение ресурсов происходит только естественным образом. Философы — народ вежливый, они не отбирают палочки у своих коллег.
- Возможно циклическое ожидание, когда процесс ожидает освобождения ресурса, удерживаемого другим процессом, который в свою очередь ожидает освобождения ресурса, удерживаемого третьим процессом и т. д., а n-й процесс ожидает доступности ресурса, удерживаемого первым процессом, то есть круг замыкается. В примере DeadlockingDiningPhilosophers.cpp циклическое ожидание возникает из-за того, что каждый философ сначала пытается взять правую, а потом — левую палочку.

Так как взаимная блокировка возникает лишь при соблюдении всех условий, для ее избежания достаточно предотвратить хотя бы одно условие. В нашей программе взаимные блокировки проще всего предотвращаются нарушением четвертого условия. Это условие возникает из-за того, что каждый философ пытается брать палочки в строго определенной последовательности: сначала правую, потом левую. Из-за этого возможна ситуация, когда каждый философ держит правую палочку и ждет освобождения левой, создавая ситуацию циклического ожидания. Если инициализировать последнего философа так, чтобы он сначала брал левую палочку, и только потом правую, он не помешает соседу справа взять палочку, находящуюся слева от последнего. В этом случае циклическое ожидание исключает-

ся. Впрочем, это всего лишь одно из возможных решений; вы также можете избавиться от взаимной блокировки, нарушая другие условия (за подробностями обращайтесь к специализированной литературе по многопоточному программированию):

```
//: C11:FixedDiningPhilosophers.cpp {RunByHand}
// Обедающие философы без взаимной блокировки.
//{L} ZThread
#include <ctime>
#include "DiningPhilosophers.h"
#include "zthread/ThreadedExecutor.h"
using namespace ZThread;
using namespace std;

int main(int argc, char* argv[]) {
  srand(time(0)); // Раскрутка генератора случайных чисел
  int ponder = argc > 1 ? atoi(argv[1]) : 5;
  cout << "Press <ENTER> to quit" << endl;
  enum { SZ = 5 };
  try {
    CountedPtr<Display> d(new Display);
    ThreadedExecutor executor;
    Chopstick c[SZ];
    for(int i = 0; i < SZ; i++) {
      if(i < (SZ-1))
        executor.execute(new Philosopher(c[i], c[i + 1], d, i, ponder));
      else
        executor.execute(
          new Philosopher(c[0], c[i], d, i, ponder));
    }
    cin.get();
    executor.interrupt();
    executor.wait();
  } catch(Synchronization_Exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Позаботившись о том, чтобы последний философ брал и клал левую палочку раньше правой, мы устраняем взаимную блокировку.

На уровне языка не существует средств борьбы с взаимными блокировками; вы сами должны позаботиться об этом за счет тщательного проектирования своей программы. Впрочем, эти слова вряд ли успокоят программиста, пытающегося отладить программу, «пораженную» взаимной блокировкой.

Итоги

В этой главе мы постарались изложить основы параллелизма применительно к многопоточному программированию.

- В контексте программы могут выполняться несколько независимых задач.
- Программист должен учесть все проблемы, которые могут возникнуть при завершении этих задач, в частности проблему уничтожения объектов (задач) до того, как другие задачи завершили работу с ними.
- Задачи могут конфликтовать друг с другом за общие ресурсы. Основным средством предотвращения подобных конфликтов являются мутексы.

- При недостаточно внимательном проектировании может возникнуть ситуация взаимной блокировки.

Впрочем, существует множество других аспектов многопоточного программирования и инструментов, используемых для решения многопоточных задач. Библиотека ZThreads содержит целый ряд таких вспомогательных средств, в том числе *семафоры* и особые разновидности очередей вроде тех, которые были представлены в этой главе. Самостоятельное изучение библиотеки и других информационных ресурсов по многопоточному программированию даст вам более глубокие познания в этой области.

Вы должны хорошо понимать, когда задействовать многопоточность, а когда лучше обойтись без нее. Ниже перечислены основные причины для использования многопоточности.

- Управление несколькими задачами, параллельное выполнение которых позволит более эффективно расходовать ресурсы компьютера, а также прозрачно распределять задачи между несколькими процессорами.
- Улучшение структуры программы.
- Удобство пользователя.

Классическим примером распределения нагрузки является использование процессора во время ожидания операций ввода-вывода. Классический пример удобства для пользователей — опрос кнопки Stop во время длинных операций загрузки.

Дополнительным преимуществом программных потоков считается то, что они обеспечивают «облегченное» переключение контекста (порядка 100 машинных команд) по сравнению с «тяжелым» переключением контекста на уровне процессов (тысячи машинных команд). Поскольку все программные потоки внутри процесса используют общее пространство памяти, при облегченном переключении контекста изменяется только точка выполнения и локальные переменные. Переключение на уровне процесса требует переключения всего пространства памяти.

Ниже перечислены основные недостатки многопоточности.

- Замедление работы программы при ожидании общих ресурсов.
- Дополнительные затраты процессорного времени на управление потоками.
- Неудачные архитектурные решения оборачиваются неоправданной сложностью программы.
- Опасность аномальных ситуаций вроде гонок, взаимных блокировок и активных тупиков.
- Расхождения между платформами. При разработке программ для этой главы (на языке Java) были обнаружены ситуации гонок, которые быстро возникали на одних компьютерах, но отсутствовали на других. Примеры на C++ вели себя по-разному (хотя обычно приемлемо) в разных операционных системах. Даже если написанная вами программа нормально работает на одном компьютере, при ее распространении возможны неприятные сюрпризы.

Основные трудности в многопоточных программах возникают из-за того, что ресурсы (например, память объекта) могут использоваться сразу несколькими

потоками, и вам приходится следить за тем, чтобы эти потоки не пытались читать и изменять ресурс одновременно. Это требует тщательно продуманных средств синхронизации, потому что в противном случае в программе может незаметно возникнуть ситуация взаимной блокировки.

Многопоточное программирование до определенной степени является искусством. Язык C++ проектировался так, чтобы программист мог создать столько объектов, сколько потребуется для решения его задачи, по крайней мере, теоретически (хотя вряд ли было бы разумно создавать миллионы объектов для инженерного расчета методом конечных элементов). Но количество программных потоков обычно ограничивается практическими соображениями, потому что с превышением некоторого числового порога программные потоки выходят из-под контроля. Определить это критическое число нелегко, оно часто зависит от операционной системы и потоковой библиотеки; в одних случаях может быть меньше сотни потоков, в других — больше тысячи. Для решения большинства задач обычно хватает десятка программных потоков, поэтому это не создает особых ограничений, но в более общей архитектуре ситуация может быть иной.

Какой бы простой ни казалась реализация многопоточности в некотором языке или библиотеке, всегда будьте начеку. Всегда найдется что-нибудь, чего вы не учли, и это «что-нибудь» нанесет коварный удар в тот момент, когда вы меньше всего этого ожидаете (для примера вспомните задачу с обедающими философами: ее можно настроить так, что взаимная блокировка будет возникать очень редко, и у вас создастся впечатление, что программа работает нормально). По этому поводу хорошо высказался Гвидо ван Россум (Guido van Rossum), создатель языка программирования Python:

> *В любом многопоточном проекте большинство ошибок возникает именно из-за многопоточности. Причем от языка программирования ничего не зависит — это какое-то глубокое и еще не понятое нами свойство программных потоков.*

Упражнения

1. Создайте класс, производный от `Runnable`, и переопределите функцию `run()`. Функция должна выводить сообщение, а затем вызывать `sleep()`. Повторите три раза, а затем верните управление из `run()`. Включите в конструктор вывод сообщения о создании объекта; другое сообщение должно выводиться при завершении задачи. Создайте несколько потоковых объектов этого типа, запустите их и проанализируйте происходящее.
2. Измените пример `BasicThreads.cpp` так, чтобы потоки `LiftOff` запускали другие потоки `LiftOff`.
3. Измените пример `ResponsiveUI.cpp` и полностью исключите из него любую возможность гонок (предположите, что операции `bool` не являются атомарными).
4. В программе `Incrementer.cpp` измените класс `Count` так, чтобы вместо массива `int` в нем использовалась одна переменная `int`. Объясните результат.

5. В примере EvenChecker.h исправьте потенциальную ошибку в классе Generator (предположите, что операции bool не являются атомарными).
6. Измените пример EvenGenerator.cpp так, чтобы вместо флагов завершения в нем использовалась функция interrupt().
7. В примере MutexEvenGenerator.cpp измените код MutexEvenGenerator::nexValue() так, чтобы команда возврата предшествовала команде release(). Объясните происходящее.
8. Измените пример ResponsiveUI.cpp так, чтобы вместо флага quitFlag в нем использовалась функция interrupt().
9. Изучите документацию по объекту Singleton в библиотеке ZThreads. Измените пример OrnamentalGarden.cpp так, чтобы объект Display контролировался объектом Singleton для предотвращения возможности создания нескольких экземпляров Display.
10. В примере OrnamentalGarden.cpp измените функцию Count::increment() так, чтобы она выполняла прямой инкремент count (то есть просто count++). Затем удалите стража и посмотрите, приведет ли это к сбою. Является ли такой вариант надежным и безопасным?
11. Измените программу OrnamentalGarden.cpp, чтобы она использовала вызов interrupt() вместо механизма pause(). Убедитесь в том, что в вашем решении не происходит преждевременного уничтожения объектов.
12. Измените пример WaxOMatic.cpp и добавьте новые экземпляры класса Process, чтобы покрытие наносилось и полировалось не в один, а в три слоя.
13. Создайте два подкласса Runnable. В одном функция run() после запуска должна вызывать wait(), в другом функция run() должна сохранять ссылку на первый объект Runnable и после истечения заданного количества секунд вызывать signal() для первого потока, чтобы первый поток мог вывести сообщение.
14. Напишите пример с наличием активного ожидания. Один поток приостанавливается на некоторое время, а затем устанавливает флаг. Второй поток отслеживает этот флаг в цикле while (активное ожидание) и, когда флаг устанавливается, сбрасывает его и сообщает об изменении на консоль. Зафиксируйте, сколько лишнего времени проводит программа внутри цикла активного ожидания, и создайте вторую версию, которая использует функцию wait() вместо активного ожидания. Дополнительно можете измерить время, затрачиваемое процессором в обоих случаях, при помощи программы-профайлера.
15. Измените файл TQueue.h и ограничьте максимальное допустимое количество элементов. При достижении заданного порога дальнейшая запись блокируется до тех пор, пока количество элементов не уменьшится. Напишите программу для тестирования новых возможностей.
16. Измените пример ToastOMaticMarkII.cpp так, чтобы готовить сэндвичи с арахисовым маслом и желе (половинки сэндвича должны изготавливаться на двух отдельных линиях). Готовые сэндвичи заносятся в выходную очередь

TQueue. Для вывода результатов воспользуйтесь объектом Reporter, как это сделано в примере CarBuilder.cpp.

17. Перепишите пример C07:BankTeller.cpp так, чтобы вместо имитации в нем присутствовала «нормальная» многопоточность.
18. Измените пример CarBuilder.cpp, присвойте идентификаторы роботам и добавьте дополнительные экземпляры разных типов роботов. Проверьте, все ли роботы используются при сборке.
19. Измените пример CarBuilder.cpp и добавьте в процесс сборки новый этап с установкой системы выхлопа, кузова и крыльев. Как и на первом этапе, эти процессы могут выполняться роботами одновременно.
20. Измените пример CarBuilder.cpp и обеспечьте синхронизацию доступа ко всем переменным bool внутри Car. Из-за невозможности копирования мутексов в программу придется внести существенные изменения.
21. Используя подход, продемонстрированный в примере CarBuilder.cpp, смоделируйте процесс строительства дома.
22. Создайте класс Timer с двумя режимами работы. В первом таймер должен сработать только один раз, во втором — срабатывать с регулярными промежутками. Воспользуйтесь этим классом в примере C10:MulticastCommand.cpp и переместите вызовы TaskRunner::run() из процедур в таймер.
23. Измените оба примера с обедающими философами так, чтобы в командной строке можно было задать не только временные параметры, но и количество философов. Поэкспериментируйте с разными значениями и объясните результаты.
24. Измените пример DiningPhilosophers.cpp так, чтобы каждый философ просто брал свободные палочки (завершив еду, философ бросает палочки в общую корзину; когда философ хочет есть, он берет из корзины две свободные палочки). Устраняется ли при этом возможность взаимной блокировки? Можно ли снова ввести взаимную блокировку простым сокращением количества палочек?

Список терминов

Ниже приведен список основных терминов, используемых в книге.

Термин	Аббревиатура	Перевод
Abstract factory		Абстрактная фабрика
Activation Record Instance	ARI	Экземпляр активационной записи
Adaptable function object		Адаптируемый объект функции
Adapter		Адаптер
Adaptor		Адаптер
Aggregation		Агрегирование
Allocator		Распределитель памяти
Applicator		Аппликатор
Argument-Dependent Lookup	ADL	Поиск с учетом аргументов
Assertion		Утверждение
Associative container		Ассоциативный контейнер
Asymptotic complexity		Асимптотическая сложность
Atomic operation		Атомарная операция
Bidirectional iterator		Двусторонний итератор
Binder		Адаптер привязки
Builder		Строитель
Busy wait		Активное ожидание
Call chain		Цепочка вызовов
Callback function		Функция обратного вызова
Cast		Преобразование типа
Chain of responsibility		Цепочка ответственности
Class invariant		Инвариант класса
Class template		Шаблон класса
Cleaning up		Зачистка
Closure		Замыкание
Collecting parameter		Накопитель
Command		Команда
Concurrency		Параллельность
Container		Контейнер
Container adaptor		Контейнерный адаптер
Container class		Контейнерный класс
Context		Контекст
Copy-on-write		Копирование при записи
Critical section		Критическая секция
Deadlock		Взаимная блокировка

Термин	Аббревиатура	Перевод
Defensive programming		Защитное программирование
Dependent base class		Зависимый базовый класс
Dependent name		Зависимое имя
Deque		Дек
Design pattern		Паттерн проектирования
Dispatching		Диспетчеризация
Double dispatching		Двойная диспетчеризация
Double linked list		Двусвязный список
Downcast		Понижающее преобразование типа
Dynamic chain		Динамическая цепочка
Dynamic type		Динамический тип
Effector		Эффектор
Event-driven programming		Событийное программирование
Exception handler		Обработчик исключений
Exception handling		Обработка исключений
Exception neutral		Нейтральная (библиотека) по отношению к исключениям
Exception safety		Безопасность исключений
Executor		Исполнитель
Expression template		Шаблон выражения
eXtreme Programming	XP	Экстремальное программирование
Facet		Фацет
Factory		Фабрика
Factory method		Фабричный метод
Format flag		Форматный флаг
Forward declaration		Опережающее объявление
Forward iterator		Прямой итератор
Friend template		Дружественный шаблон
Function inlining		Подстановка функций
Function object		Объект функции
Function template		Шаблон функции
Functor		Функтор
Gang of Four	GoF, или БЧ	Банда четырех
Generalization		Обобщение
Generic algorithm		Обобщенный алгоритм
Generic container		Обобщенный контейнер
Generic function		Обобщенная функция
Graphical User Interface	GUI	Графический пользовательский интерфейс
Guard		Страж
Heap		Куча
Inheritance		Наследование
Initialization		Инициализация
Inline function		Подставляемая функция
Inlining		Подстановка
Inner class		Внутренний класс
Input iterator		Итератор ввода
Input stream		Поток ввода
Input/Output	I/O	Ввод-вывод
Insertion iterator		Итератор вставки
Instantiation		Специализация

Термин	Аббревиатура	Перевод
Interpreter		Интерпретатор
Interrupted status		Статус прерывания
Invariant		Инвариант
Iostream		Поток ввода-вывода
Iterator		Итератор
Iterator invalidation		Недействительность итераторов
Iterator tag		Итераторный тег
Lazy initialization		Отложенная инициализация
Linked list		Связанный список
List		Список
Livelock		Активный тупик
Locale		Локальный контекст
Manipulator		Манипулятор
Map		Отображение
Member template		Вложенный шаблон
Messenger		Посыльный
Mixin class		Подключаемый класс
Model-View-Controller	MVC	Модель-представление-контроллер
Multimap		Мультиотображение
Multiple dispatching		Множественная диспетчеризация
Multiple membership		Множественная принадлежность
Multiset		Мультимножество
Multitasking		Многозадачность
Multithreading		Многопоточность
Mutual exclusion	Mutex	Взаимное исключение, или мутекс
Narrow stream		Узкий поток
Non-type parameter		Нетиповой параметр
Object-based hierarchy		Объектно-базированная иерархия
Observer		Наблюдатель
Operator function		Операторная функция
Output iterator		Итератор вывода
Output stream		Поток вывода
Parser		Парсер
Pattern		Паттерн
Placement syntax		Синтаксис размещения
Pointer		Указатель
Policy class		Класс политик
Polymorphic type		Полиморфный тип
Portable Operating System Interface	POSIX	Интерфейс переносимых операционных систем
Predicate		Предикат
Preemptive multithreading		Вытесняющая многопоточность
Priority		Приоритет
Priority queue		Приоритетная очередь
Private function		Закрытая функция
Process		Процесс
Proxy		Посредник
Public function		Открытая функция
Qualified name		Уточненное имя
Qualifier		Квалификатор
Queue		Очередь
Race		Гонки

Термин	Аббревиатура	Перевод
Race condition		Ситуация гонок
Random-access iterator		Итератор произвольного доступа
Reference		Ссылка
Reference counting		Подсчет ссылок
Resource Acquisition Is Initialization	RAII	Получение ресурсов при инициализации
Reversible container		Обратимый контейнер
RunTime Type Identification	RTTI	Идентификация типов в процессе исполнения
Semaphore		Семафор
Sequence		Последовательность
Sequence container		Последовательный контейнер
Serialization		Сериализация
Set		Множество
Shadow stack		Теневой стек
Single dispatching		Одинарная диспетчеризация
Singleton		Синглет
Slice		Срез
Smart pointer		Умный указатель
Smart reference		Умная ссылка
Spurious wakeup		Спонтанная активизация
Stable reordering		Устойчивая сортировка
Stack		Стек
Stack frame		Кадр стека
Stack unwinding		Раскрутка стека
Standard Template Library	STL	Стандартная библиотека шаблонов
State		Состояние
Strategy		Стратегия
Stream		Поток данных
Strict weak ordering		Строгая квазиупорядоченность
Surrogate class		Суррогатный класс
Template		Шаблон
Template function		Шаблонная функция
Template metaprogramm		Шаблонная метапрограмма
Template method		Шаблонный метод
Thread		Программный поток
Trait		Характеристика
Traits template		Шаблон характеристик
Turing complete		Полнота по Тьюрингу
Unit buffering		Немедленный вывод
Unstable reordering		Неустойчивая сортировка
Vector		Вектор
Virtual base class		Виртуальный базовый класс
Virtual function		Виртуальная функция
Visitor		Посетитель
Weak ordering		Квазиупорядоченность
Weak typing		Слабая типизация
Wide stream		Расширенный поток
Wrap		Оболочка
Zero-initialization		Нулевая инициализация

Алфавитный указатель

N

O

P

Q

R

S

T

U

V

W

Y

Z

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Ю

Я

Философия C++

Практическое программирование

«Это единственная книга, которую вам обязательно следует иметь, если вы всерьез занимаетесь разработкой на C++.»

Ричард Хейл Шоу, выпускающий редактор PC Magazine

Используйте C++ в полную силу!

- Изучение приемов практического программирования.
- Типичные проблемы при программировании на C++.
- Построение надежных программ.
- Применение объектов, композиции и полиморфизма.
- Доступное введение в многопоточное программирование, поддержка которого должна быть включена в следующую версию стандарта C++.
- Простые методы модульного тестирования и отладки.
- Подробное описание средств стандартной библиотеки C++: строки, потоки ввода/вывода, алгоритмы и контейнеры STL.
- Современное применение шаблонов, включая метапрограммирование.
- Объяснение сложностей множественного наследования.
- Практическое использование RTTI.
- Обработка исключений.
- Соответствие официальному стандарту C++ ISO.
- Представление возможностей, которые планируется включить в следующую версию стандарта C++.

PEARSON Prentice Hall

ПИТЕР®
WWW.PITER.COM

Посетите наш web-магазин:
www.piter.com

ISBN 5-469-00043-5
9 785469 000433