Классика computer science

Программирование на C++

Дж. Коплиен

Addison-Wesley

Питер

Advanced C++

Programming Styles and Idioms

James O. Coplien

Addison-Wesley

КЛАССИКА COMPUTER SCIENCE

Дж. Коплиен

ПРОГРАММИРОВАНИЕ НА C++

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2005

ББК 32.973-018.1
УДК 681.3.06
К65

Коплиен Дж.
К65 Программирование на C++. Классика CS. — СПб.: Питер, 2005. — 479 с.: ил.

ISBN 5-469-00189-X

Эта книга написана для программистов, уже владеющих языком C++ и желающих поднять свою квалификацию на новый уровень. Давая представление о стиле и идиоматике языка, книга знакомит читателя с теми нетривиальными знаниями, которые опытные программисты C++ получают на личном опыте. Она показывает, что C++ можно использовать и для разработки простых абстракций данных, и для полноценной реализации абстрактных типов данных, и для объектно-ориентированного программирования различных стилей. Кроме того, в ней исследуются идиомы, не поддерживаемые напрямую на базовом уровне C++, например функциональное и фреймовое программирование, а также расширенные методы уборки мусора.

ББК 32.973-018.1
УДК 681.3.06

ISBN 0201548550 (англ.)
ISBN 5-469-00189-X

Краткое содержание

Содержание

Посвящается
Сандре, Кристоферу,
Лорелее и Эндрю Майклу
с любовью

Предисловие

Эта книга написана для программистов, уже владеющих языком C ++ и желающих поднять свою квалификацию на новый уровень. Но для этого нужно сначала разобраться в том, как мы осваиваем новый язык (в данном случае — язык программирования), и в том, как использовать язык для эффективного решения задач программирования.

Изучение языка программирования

Далеко не все, что необходимо знать о продукте, написано в техническом руководстве. Незадолго до того, как в моей семье появился первый ребенок, один мой знакомый предупредил, что никакие книги и курсы для неопытных родителей не смогут полностью подготовить нас к воспитанию ребенка. Конечно, мы должны освоить минимальные, самые необходимые навыки, и все же самое интересное, трудное и полезное выходит за рамки базовых знаний. Ни одна книга или «руководство пользователя» не поможет понять, почему ваша трехлетняя дочь намазала зубной пастой голову своего младшего брата или зачем они оба положили свои носки в холодильник.

То же самое относится к языкам программирования. Синтаксис языка до определенной степени формирует наше восприятие, но простое описание синтаксиса в «руководстве пользователя» станет всего лишь отправной точкой. Структура наших программ (а следовательно, и тех систем, которые мы строим) в основном определяется *стилем* и *идиомами*, используемыми для выражения архитектурных концепций.

Стиль отличает истинное мастерство от простой удачи. Эффективный стиль воспитания ребенка, программирования и вообще всего на свете строится на основе личного опыта или опыта других. Программист, который умеет правильно связывать возможности языка программирования с потребностями приложения, пишет превосходные программы. Но чтобы выйти на этот уровень, необходимо от правил и механического запоминания перейти к идиомам и стилю, а в конечном счете — к концептуальным и структурным абстракциям.

Правила, идиомы и концепции определяют структуру тех систем, которые мы создаем, — они предоставляют в наше распоряжение модель для их разработки. Модель декомпозиции задачи и формирования системы является *парадигмой* —

образцом для разбиения предметной области на части, с которыми удобно работать. C++ является «мультипарадигменным» языком. Программисты C используют C++ как «улучшенный язык C», а сторонники объектно-ориентированного подхода ценят в C++ полиморфизм. Однако эффективное и элегантное решение задачи обычно *требует* разных подходов.

Изучение языка программирования имеет много общего с изучением естественного языка. Знание базового синтаксиса позволяет программисту писать простые процедуры и строить из них нетривиальные программы — подобно тому, как человек со словарным запасом в несколько тысяч иностранных слов способен написать нетривиальный рассказ. Но настоящее мастерство — совсем другое дело. Рассказ может быть нетривиальным, но от этого он не станет читаться «на одном дыхании», подчеркивая свободу владения языком его автора. Изучение синтаксиса и базовой семантики языка сродни 13-часовым курсам немецкого для начинающих: после прохождения таких курсов вы сможете заказать колбаски в ресторане, но для работы в Германии журналистом или поэтом их наверняка окажется недостаточно. Различие кроется в *идиоматике* языка. Например, в синтаксисе C нет ничего, что бы определяло следующую конструкцию как один из базовых «строительных блоков» программ:

```
while (*cp1++ = *cp2++);
```

Но если программист не знаком с этой конструкцией, его нельзя считать полноценно владеющим C.

В программировании, как и в естественных языках, пригодность и выразительность языковых конструкций базируется на важных идиомах. Хорошие идиомы упрощают работу прикладного программиста, подобно тому как идиомы любого языка обогащают наше общение. Программные идиомы являются «выражениями» семантики программирования, пригодными для многократного использования в том же смысле, в котором классы служат для многократного использования архитектурных решений и кода. Простые идиомы (вроде упомянутого цикла while) всего лишь обеспечивают удобную запись, но редко играют сколь-нибудь заметную роль в архитектуре программы. В книге основное внимание уделяется идиомам, влияющим на применение языка в общей структуре и реализации архитектурных решений. По сравнению с обычными «вспомогательными» идиомами их освоение требует больше времени и умственных усилий. Обычно такие идиомы относительно сложны и запутанны; в них приходится учитывать много тонкостей. Но после того как вы их освоите, они станут удобным и мощным инструментом программирования.

О книге

Предполагается, что читатель владеет базовым синтаксисом C ++. Давая представление о стиле и идиоматике языка, книга знакомит читателя с теми нетривиальными знаниями, которые опытные программисты C++ получают на личном

опыте. Она показывает, что C++ можно использовать и для разработки простых абстракций данных, и для полноценной реализации абстрактных типов данных, и для объектно-ориентированного программирования различных стилей. Кроме того, в ней исследуются идиомы, не поддерживаемые напрямую на базовом уровне C++, например функциональное и фреймовое программирование, а также расширенные методы уборки мусора.

Структура книги

Вместо «одномерного» подхода к описанию нетривиальных возможностей C++ с упорядочением по языковым средствам книга описывает абстракции с точки зрения возможностей C++, необходимых для их поддержки. Каждая глава книги посвящена отдельному семейству таких идиом, причем идиомы, описанные в начале книги, используются как основа для изложения материала последующих глав.

В главе 1 приводится исторический обзор идиом C++. Из нее читатель узнает, почему появились идиомы, а также познакомится с разными вариантами интерпретации идиом как внешних конструкций или как составляющих языка.

В главе 2 представлены главные «строительные блоки» языка C++ — классы и функции классов. Хотя в этой главе в основном излагается базовый материал, к некоторым идиомам и терминам мы вернемся в последующих главах. Среди прочего рассматриваются системы контроля типов компилятора, их связь с пользовательскими типами и классами с точки зрения архитектуры. Кроме того, рассматриваются идиомы, связанные с использованием ключевого слова const.

Глава 3 отдана идиомам, которые превращают классы в «полноценные» типы. В процессе эволюции C++ операции копирования и инициализации объектов постепенно автоматизировались, но для большинства нетривиальных классов программистам все равно приходится настраивать присваивание и конструктор по умолчанию. В этой главе описана общая схема такой настройки. Представленные идиомы называются *каноническими формами*; это означает, что они определяют принципы и стандарты, образующие базовую механику работы объектов. Помимо самой распространенной ортодоксальной канонической формы представлены идиомы реализации подсчета ссылок в новых и существующих классах — первые идиомы в книге, выходящие за рамки базового синтаксиса C++. Одна из разновидностей подсчета ссылок — подсчет указателей — несколько ослабляет связь C++ с аппаратурой (компьютером); вместо обычных указателей программист использует их более интеллектуальные объектные аналоги. В завершение главы читатель узнает, как отделить создание объекта от его инициализации. Для человека, хорошо знакомого с базовым языком C ++, такое разделение выглядит противоестественно, поскольку в C++ эти две операции тесно связаны. Необходимость их разделения возникает при проектировании драйверов устройств и систем с взаимозависимыми ресурсами.

Глава 4 посвящена наследованию, а в главе 5 наследование обогащается полиморфизмом в рамках знакомства с объектно-ориентированным программированием. Многие неопытные программисты C++ склонны злоупотреблять наследованием и применять его при любом удобном случае. Хотя наследование требуется в основном для поддержки объектной парадигмы, у него имеется другое, никак не связанное с объектной парадигмой применение — многократное использование кода. Рассмотрение наследования отдельно от полиморфизма поможет читателю разделять эти две концепции и предотвратит недоразумения, часто возникающие при попытке их одновременного изучения.

В главе 6 конструкции, стили и идиомы C++ рассматриваются с точки зрения архитектуры и проектирования. В этой главе мы попытаемся ответить на вопрос, что же *означают* классы на уровне приложения, который гораздо выше уровня синтаксиса. Правильное понимание связей между архитектурными абстракциями приложения с одной стороны и классами/объектами его реализации с другой способствует появлению надежных легко развивающихся систем. Другим ключевым фактором эволюции систем является расширение архитектуры за пределы конкретного приложения до предметной области в целом; важную часть главы составляют принципы доменного анализа. Также здесь приводятся рекомендации по применению наследования — эта тема часто вызывает трудности у неопытных программистов C++. Читатели, уже знакомые с объектно-ориентированным проектированием, оценят рекомендации по превращению результатов проектирования в код C++. Кроме того, в главе рассматривается инкапсуляция как альтернатива наследованию в отношении многократного использования и полиморфизма.

Глава 7 посвящена теме многократного использования кода и архитектурных решений. Мы разберем четыре механизма многократного использования, обращая особое внимание на достоинства и недостатки «скороспелого наследования». Представленные идиомы позволяют существенно сократить объем кода, сгенерированного параметризованными библиотеками типов с помощью шаблонов.

В оставшейся части книги рассматриваются нетривиальные идиомы программирования, выходящие далеко за рамки традиционного языка C++. В главе 8 представлены прототипы — специальные объекты, которым отводится роль классов C++ и которые позволяют решать некоторые стандартные задачи (такие как имитация виртуальных конструкторов). Помимо этого прототипы закладывают основу для освоения более совершенных приемов проектирования, рассматриваемых в следующих главах.

Глава 9, посвященная стилю символических языков программирования, отходит от концепций, которые обычно считаются базовыми для программирования на C++, включая сильную типизацию и непосредственное управление памятью. Идиомы этой главы ближе к стилю Smalltalk и Lisp, отходя от канонов традиционного программирования на C++. Конечно, можно сказать, что тот, кто хочет программировать в стиле Smalltalk, должен программировать на Smalltalk. Действительно, если вы хотите пользоваться всеми возможностями Smalltalk — пишите на

Smalltalk. И все же некоторая экзотичность идиом, представленных в этой главе, вовсе не означает, что они редко применяются на практике. Иногда нужно, чтобы небольшая часть системы могла опереться на гибкость и полиморфизм символических языков. В таких ситуациях приходится выходить за рамки философии C++, оставаясь в границах семантики и системы контроля типов C++. В главе 9 приводится упорядоченное описание таких идиом, чтобы их в случае необходимости не приходилось создавать заново.

В главе 9 также представлены идиомы, обеспечивающие частичное обновление системы на стадии выполнения. Реализации этих идиом заведомо зависят от многих факторов, связанных с целевой платформой, поэтому материал призван познакомить читателя с технологическим уровнем, на котором должны решаться вопросы оперативной дозагрузки. Представленный пример достаточно типичен, а с точки технологии он не слишком сложен, но и не тривиален. Для любых платформ, кроме рабочих станций Sun, он потребует серьезной переработки, а в некоторых средах он вообще работать не будет. Но от этой идиомы не зависит ни одна другая идиома в книге, поэтому читатель можно принять или отвергнуть ее, руководствуясь исключительно ее собственной ценностью. Глава 9 написана *не для того*, чтобы превратить C++ в Smalltalk; делать этого не стоит (да и не получится). Идиомы этой главы не столь безопасны по отношению к типам на стадии компиляции, а в общем случае менее эффективны, чем «традиционный» код C++; с другой стороны, они обладают большей гибкостью и автоматизируют управление памятью.

Глава 10 посвящена динамическому множественному наследованию. По поводу множественного наследования в C++ до сих пор идут споры, поэтому его динамическая разновидность была выделена в отдельную главу, чтобы не смешиваться с материалом других глав. Статическое множественное наследование, описанное в главе 5, тоже применяется на практике, но динамическое множественное наследование решает проблемы комбинаторного роста числа сочетаний классов. Этот подход приносит пользу во многих реальных программах, включая редакторы, системы автоматизации проектирования и управления базами данных.

В последней главе объекты рассматриваются с высокоуровневой системной точки зрения. Уровень абстракции поднимается от масштаба классов C++ до более крупных и обобщенных единиц программной архитектуры и организации. Мы рассмотрим ряд важных системных аспектов, таких как планирование, обработка исключений и распределенные вычисления. Также здесь приводятся некоторые рекомендации по поводу модульности и многократного использования, связанные с материалом глав 6 и 7. Попутно обсуждаются проблемы выбора структуры и сопровождения библиотек.

В приложении A основные концепции C++ сравниваются с аналогами из языка C. Несомненно, многие читатели уже знакомы с этим материалом или смогут найти подробности в других книгах. Сравнение языков включено в книгу по двум причинам. Во-первых, такой материал может послужить справочником на случай, если вам потребуется разобраться с какой-нибудь непростой конструкцией.

Во-вторых, поскольку стили программирования C и C++ рассматриваются с точки зрения проектирования, вы узнаете, как смешивать процедурный и объектно-ориентированный код. Это особенно важно для программистов C++, работающих с готовым кодом C.

Приводимые в книге примеры основаны на стандарте C++ версии 3.0. Они были протестированы под управлением системы 3 AT&T USL Language System на многих аппаратных платформах, а также в других средах C++. Многие примеры тестировались в GNU C++ версии 1.39.0 и Zortech C++ 2.0, хотя примеры, ориентированные на специфические возможности версии 3.0, еще ожидают выхода обновленных версий этих компиляторов. В некоторых примерах используются библиотеки классов общего назначения для работы с множествами, отображениями, списками и т. д. Многие разработчики предлагают эффективные версии таких библиотек, однако достаточно функциональные версии, предназначенные для учебных целей, можно написать «с нуля». Из приводимых примеров можно извлечь заготовки, а иногда и готовые реализации многих классов общего назначения.

Благодарности

Книга обязана своим существованием многим. Исходная идея принадлежит Крису Карсону (Chris Carson) из Bell Laboratories; именно он как никто иной заслуживает звания «крестного отца» этой книги. Автор благодарен ему за инициативу и поддержку, оказанную в начале проекта. Книга создавалась под присмотром Кейт Уоллман (Keith Wollman), изобретательного и великодушного редактора, а также Хелен Уайт (Helen Wythe), руководителя проекта. Лорен Миньяччи (Lorraine Mignacci) внесла значительный вклад в материал главы 6, а обсуждение с ней материала других глав принесло огромную пользу. Книга многим обязана группе добросовестных и дотошных рецензентов, делившихся со мной своими идеями, среди которых Тим Борн (Tim Born), Маргарет А. Эллис (Margaret A. Ellis), Билл Хопкинс (Bill Hopkins), Эндрю Кениг (Andrew Koenig), Стен Липман (Stan Lippman), Барбара Му (Barbara Moo), Бонни Прокопович (Bonnie Prokopowicz), Ларри Шутт (Larry Schutte) и Бьярн Страуструп (Bjarne Stroustrup). Алексис Лейтон (Alexis Layton), Джим Эдкок (Jim Adcock) и Майк Экройд (Mike Ackroyd) дали множество советов и предложений относительно того, как сделать книгу более содержательной, и автор глубоко благодарен им за это. Масса других усовершенствований появилась благодаря рецензиям Мирона Абрамовича (Miron Abramovich), Мартина Кэррола (Martin Carroll), Брайана Кернигана (Brian Kernighan), Эндрю Клейна (Andrew Klein), Дуга Макилроя (Doug McIlroy), Денниса Манкла (Dennis Mancl), Уоррена Монтгомери (Warren Montgomery), Тома Мюллера (Tom Mueller), Анил Пал (Anil Pal), Пегги Куинн (Peggy Quinn) и Бена Ровеньо (Ben Rovegno). Мэри Крэбб (Mary Crabb), Джин Оуэн (Jean Owen) и Крис Скассел (Chris Scussel) поделились своим опытом в области форматирования текста. Бретт Л. Шухерт (Brett L. Schuchert) и Стив Виноски (Steve Vinoski) постарались сообщить обо всех ошибках, допущенных в первых

материалах; их труд значительно повысил качество материала в последующих итерациях. Особая благодарность автора компании Corporate Desktop Services из Глен Эллин, штат Иллинойс, за ликвидацию ошибок в готовых гранках. Большое спасибо Джуди Марксхаузен (Judy Marxhausen) за консультации по специальным вопросам.

Автор благодарен своему начальству из AT&T за поддержку и за выделение времени и ресурсов для работы над книгой. Спасибо Полу Зислису (Paul Zislis) и Бену Ровеньо (Ben Rovegno) за поддержку на первых порах, а также Уоррену Монтгомери (Warren Montgomery), Джеку Вангу (Jack Wang) и Эрику Саммерумладшему (Eric Summer Jr.) за поддержку, идеи и терпение.

Студенты многих ведущихся автором курсов помогли ему в выборе и оценке материалов, которые позднее вошли в книгу. Автор особенно благодарен студентам курсов C++, которые он вел в AT&T Bell Laboratories в Напервилле, штат Иллинойс, и Колумбусе, штат Огайо, в 1989 году.

От издателя перевода

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.

Глава 1
Введение

Идиомы являются важной составляющей как естественных языков, так и языков программирования. Эта книга посвящена идиомам, делающим язык C++ более выразительным, и тому стилю программирования, который придает программе особенную структуру, или если хотите — «характер». Программировать на C++ можно и без идиом, однако именно стиль и идиомы в значительной степени определяют выразительность, эффективность и эстетическую ценность программы. Они порождены многолетним опытом применения языка, многократными решениями одних и тех же задач и обменом информацией.

Многие идиомы имеют довольно занятную, но поучительную историю. В данной главе две разновидности идиом C++ рассматриваются с точки зрения истории языка. К первой категории относятся идиомы, непосредственно повлиявшие на развитие C++, а ко второй — идиомы поддержки объектных конструкций, выходящих за пределы базовой модели C++.

1.1. C++ как развивающийся язык

Язык C++ появился в нужное время и в нужном месте. Его предок, «C с классами», родился в 1980 году (также ставшем поворотной точкой в истории Smalltalk), а его росту способствовало бурное распространение «объектного феномена» в начале 1980-х годов. Язык развивался по мере осмысления объектной парадигмы отраслью. К лету 1983 года язык «C с классами» прочно вошел в академический и исследовательский мир. Так начался диалог, который продолжает влиять на формирование языка и в наши дни.

Стремясь к максимальной гибкости и возможности развития, C++ эффективно использовал опыт собственных пользователей и достижения объектно-ориентированного программирования в других компьютерных технологиях. Языки программирования открывали новые горизонты в множественном наследовании, методах инкапсуляции и в других важных областях. Поскольку первые попытки стандартизации C++ начались только в 1989 году, ничто не мешало проектировщикам формировать язык на базе постоянно расширяющегося опыта объектно-ориентированного программирования.

Конкретным примером влияния опыта разработки на развитие языка служит защищенная (protected) область видимости. Ранее для того чтобы предоставить

производному классу доступ, нарушающий инкапсуляцию базового класса, обычно использовались дружественные (friend) отношения. А после применения ключевого слова friend вполне естественным стало включение в язык поддержки ключевого слова protected. Обратная связь с сообществом пользователей помогла усовершенствовать модель ограничения доступа в C++ и сделать ее одной из лучших для своего времени. Аналогичные события проложили дорогу поддержке множественного наследования и шаблонов, а также улучшению множества второстепенных деталей языка.

«Роль C++ в истории» в значительной мере обусловлена происхождением этого языка от C. Сегодня C является одним из самых распространенных языков программирования, у него почти нет конкурентов в широкой области системного программирования. Программы, написанные на C, отличаются эффективностью и тесной связью с оборудованием; то же самое должно быть характерно и для C++. Среди других улучшений «классического языка C» C++ обеспечивает проверку типов на стадии компиляции. Данное обстоятельство в сочетании с эффективностью ставит C++ на особое место среди распространенных объектно-ориентированных языков.

Однако эти два фактора, отразившихся на развитии языка, — развитие объектной парадигмы и влияние неизменно популярного языка C — иногда вступали в противоречие. Разработчики C++ никогда не стремились (и не стремятся) сделать его языком «на все случаи жизни», поэтому им приходилось искать компромисс между традициями C и новыми направлениями, прокладываемыми языками вроде Smalltalk и Flavors. В большинстве случаев предпочтение отдавалось зрелым или хорошо знакомым технологиям; хотя в культурном контексте C++ является порождением эпохи Smalltalk, его корни уходят к языкам Simula (1967), Algol 68 (1968) и C (1973).

У этих решений был один важный аспект: где скрывать сложность, связанную с реализацией нетривиальных возможностей? Ее размещение в компиляторе упрощает разработку для пользователя: компилятор может автоматизировать решение повседневных задач, связанных с управлением памятью, инициализацией, зачисткой, преобразованием типов, вводом-выводом и т. д. Один из принципов, определявших направление эволюции C++, гласил, что компилятор не следует упрощать, если это создает неудобства для пользователя. Но если компилятор *чрезмерно* перегружается интеллектом, возникает дилемма: либо пользователь ограничивается одной моделью некоторого аспекта программирования («диктаторский» язык), либо сам язык приходится расширять для выражения всего богатства альтернатив («анархический» язык).

1.2. Решение проблемы сложности при помощи идиом

Чтобы предоставить в распоряжение пользователей всю мощь высокоуровневых языков, но обойтись без жесткости, присущей «тираническим» языкам, для C++ были разработаны «стандартные блоки», которые могли использоваться

программистами для реализации собственных моделей вычислений. Функциональность, в значительной степени зависевшая от поддержки со стороны компилятора (например, преобразования типов), была интегрирована непосредственно в язык. С другой стороны, другие «языковые возможности», в том числе в основном модель управления памятью, ввод-вывод, а также отчасти инициализация данных объектов, в сложных приложениях могут контролироваться пользователем. Многие стандартные задачи программирования становятся *идиоматическими*; конструкции C++ используются для выражения функциональности, не входящей непосредственно в язык, и создают иллюзию того, что она является частью языка.

Хорошим примером служит идиома копирования объектов и управления памятью. Когда объект присваивается другому объекту или передается в параметре функции, компилятор автоматически генерирует код присваивания или инициализации полей нового объекта на основании полей исходного объекта по принципу «один в один». Для больших объектов такое решение может оказаться неэффективным, к тому же оно создает аномалии в поведении указателей, хранящихся в переменных класса: компилятор по умолчанию использует поверхностное копирование, тогда как в отдельных случаях требуется глубокое копирование. Обе проблемы решаются при помощи идиом подсчета ссылок, разделяющей класс на две части: первая (*манипулятор*) обеспечивает управление памятью, а вторая (*тело*) решает прикладные задачи. Класс-манипулятор использует предоставленные программистом версии кода присваивания, инициализации и зачистки объектов, которые автоматически применяются для управления памятью по мере необходимости. Эта идиома описана в главе 3.

Другой пример — ввод-вывод, который (как и в C) не является частью языка. Но на практике операторы `<<` и `>>` часто перегружаются для поддержки идиомы потоковых объектов ввода и вывода. Все классы поддерживают ввод-вывод по единой схеме. В данном случае сложность сосредоточена не в компиляторе и не в пользовательском приложении, а в библиотеке общего назначения. Чтобы получить доступ к библиотеке, программист директивой `#include` включает в программу файл, который перегружает операторы и создает впечатление, будто операторы сдвига изначально были спроектированы в C++ для ввода и вывода.

Ключ к успеху этих идиом — их прозрачность для конечного пользователя, то есть полная иллюзия, будто они являются частью самого языка. Такие идиомы позволяют сочетать всесторонность и мощь «анархических» языков с прозрачностью «диктаторских» языков. Пользователь не ограничивается единой моделью управления памятью, которая абсолютно универсальна, но в отдельных случаях неприемлема из-за малой эффективности; он может выбрать нужное решение среди целого спектра альтернатив. Все компромиссы между эффективностью, совместимостью с интерфейсом C и удобством использования находятся под контролем программиста. Более того: те аспекты, которые в C++ *не определяются*, обеспечивают гибкость, причем этой гибкости было бы трудно добиться в языках с более высокой интеграцией моделей управления памятью. Язык C++ богат не столько возможностями, сколько *мета-возможностями* — простыми

стандартными блоками, объединение которых создает новую целостную форму, превосходящую по своей мощи сумму отдельных составляющих.

Эти мета-возможности порождены не единым грандиозным замыслом, а практическим опытом, полученным сначала в образовательных и исследовательских кругах, а затем в ходе пробных разработок. C++ стал языком с «инструментарием» вспомогательных средств для определения семантики стандартных задач программирования. Некоторые языковые средства (скажем, конструкторы или перегруженные операторы) обладают самостоятельной ценностью, но являются низкоуровневыми конструкциями. Только идиомы, объединяющие конструкторы с перегрузкой операторов присваивания, обеспечивают настоящую мощь высокоуровневого языка. Многие идиомы, как и части языковых конструкций, были разработаны пользователями ранних версий C++. Конечным итогом эволюции языка C+ в свете объектно-ориентированных приложений, а также эволюции приложений в области использования новых возможностей C++, стало определение языка в спецификации [1]. Эта спецификация достаточно хорошо проработана, чтобы послужить основой для формальной стандартизации, а приведенные в ней идиомы используются настолько часто, что могут считаться общепринятыми для повседневного программирования на C++.

Случайные пользователи C++ и те, кто изучал C++ как «улучшенный язык C», могут пройти мимо этих идиом при самостоятельном освоении C++. Язык программирования не заставляет вас пользоваться идиомами; он лишь предоставляет основные конструкции, благодаря которым идиомы становятся доступными. В этом смысле идиомы нетривиальны, поскольку они выходят далеко за пределы того, что требует (или просто поощряет) язык. Главная цель настоящей книги — познакомить читателя с классическими идиомами C++ как с самостоятельными инструментами программирования. Идиомы занимают в книге центральное место, а языковые средства их поддержки описываются по мере необходимости (а не наоборот). Подобная структура поможет неопытным программистам C++ быстрее освоить идиоматические конструкции, а программисты, хорошо знакомые с синтаксисом, смогут перейти к более мощной семантике.

1.3. Объекты для 90-х

Упомянутые ранее классические идиомы C++ берут свое начало в происхождении от языка C. Другой класс идиом появился благодаря росту численности и квалификации сообщества пользователей C++ и не без влияния со стороны других языков программирования. Сегодня C++ сохраняет совместимость с C на культурном уровне, а объектная ориентация C++ сильна ровно настолько, чтобы не нарушить эту совместимость. C++ занимает важную нишу, внося объектно-ориентированные конструкции в традиционную культуру разработки C с ее компиляторами, операционными системами и системами управления реального времени. C++ обладает богатой поддержкой объектно-ориентированного программирования на уровне языка, хотя мы видели, что большая часть этой поддержки

обусловлена соглашениями, стилем и идиомами. Многие идиомы поддерживаются на уровне мета-возможностей языка.

По мере развития отрасли репутация «языка объектно-ориентированного программирования» все выше поднимала планку требований. Язык C++ рос и развивался, но сообщество пользователей развивалось еще быстрее. Пользователи обнаружили, что для новоизобретенных применений объектно-ориентированных технологий им нужны все более мощные конструкции. В качестве примера можно привести потребность в виртуальных конструкторах, для поддержки которых необходима степень полиморфизма, более характерная для языков со слабой типизацией, нежели для модели сильной типизации C++. *Слабая*, или *динамическая*, типизация имитируется в C ++ введением дополнительных уровней абстракции и идиоматических конструкций. Сложность большинства таких идиом инкапсулируется в классах, которые их используют, — с точки зрения пользователя приложения они ничем не отличаются от обычных классов, хотя и проявляют гибкость, характерную для языков программирования более высокого уровня.

Идиомы, разработанные для поддержки этих нетривиальных возможностей, менее «идиоматичны», нежели идиомы, происходящие от развития самого языка C++; это означает, что они поддерживаются меньшим количеством мета-возможностей. Данный факт в значительной степени обусловлен происхождением C++ от C. C++ — это прежде всего C и кое-что еще, поэтому C++ не может одновременно воплотить философии проектирования, характерные для C и Smalltalk. *Прямая* поддержка моделей типов Smalltalk или Flavors потребовала бы принципиальных изменений в философии, и как следствие — отхода от объектной парадигмы C++. При этом снова бы возникла дилемма между «диктаторскими» и «анархическими» языками. Опыт показал, что идиомы могут эффективно использоваться и без специальной поддержки со стороны языка, с инкапсуляцией подробностей реализации в закрытых членах классов.

Отсюда следует вторая цель книги: представление идиом C++, которые многими считаются необходимыми для логического завершения объектной модели C++ и набора его классических идиом. Однако не стоит полагать, что эти идиомы представляют интерес не только для тех, кто желает эмулировать конструкции Smalltalk на C++. Они предназначены для решения архитектурных и технических задач, типичных для любой программы C++ с сильной объектной ориентацией.

1.4. Проектирование и язык

С развитием языка развивалось также понимание связи между объектно-ориентированной архитектурой и объектно-ориентированным языком программирования в сообществе программистов. На ранней стадии объектная парадигма в основном использовалась как инструмент для получения доступа к абстракции, инкапсуляции и гибкости, выходящей за рамки процедурных языков. Позднее объекты (или точнее — *сущности*, их аналоги в предметной области проектирования) стали рассматриваться как архитектурные конструкции.

Язык C++ также прошел по этому эволюционному пути. Концепция класса C++ изначально проектировалась для отражения архитектурных конструкций, а многократное использование кода (посредством закрытого наследования) считалось дополнительным, но второстепенным обстоятельством. По мере накопления опыта модель выделения подтипов C++ также развивалась для отражения отношений между объектами и классами, полученными в результате объектно-ориентированного анализа и проектирования. Модель C++ с мощной проверкой типов на стадии компиляции C++ помогает защититься от распространенных ошибок при наследовании.

Проектирование не является основной темой книги, хотя оно и неоднократно упоминается в тексте. Но проектирование и программирование тесно связаны в объектной парадигме, и хотя книга вроде бы посвящена «программированию», мы не можем не обращать внимания на важность выбора архитектурных решений. В главе 6 рассматривается процесс проектирования с точки зрения C++; в главе 7 говорится о многократном использовании кода, что в большей мере относится к факторам проектирования; наконец, в главе 11 описаны системные проблемы, стоящие за «поиском объектов» и правильным применением наследования. Эти вопросы, как и синтаксис языка, семантика и идиомы, появились в результате многолетнего опыта практического программирования.

Язык C++ находится на пике популярности в плане объектно-ориентированного проектирования, и, как уже было отмечено, его успех в значительной степени обусловлен этим обстоятельством. К сожалению, выбирая объектно-ориентированные решения, программисты слишком часто забывают об альтернативах. Другие парадигмы не умерли и не утратили актуальности. Они по-прежнему могут применяться, если толковый проектировщик поймет, что они хорошо подходят для конкретной задачи. Многие идиомы C ++, представленные в следующих главах, практически не связаны с объектной парадигмой. Например, функторы (см. 5.6), методика компоновки подсистем (см. главу 11), параметрические конструкции (см. главу 7) остаются нетривиальными применениями C++ в контексте основных положений и духа языка.

Хочется надеяться, что описания языка C++, идиом и стиля программирования, приведенные в книге, повысят квалификацию читателя и помогут ему воспользоваться опытом других. Вы сможете задействовать готовые идиомы и элементы стиля в тех проектах, над которыми вы сейчас работаете, или тех, которые ждут вас в будущем. Как сказал Страуструп, хорошие навыки программирования и проектирования являются результатом личного вкуса, проницательности и опыта. Данная книга ни в коем случае не является высшим авторитетом в области применения C++ в больших системах. Экспериментируйте, исследуйте и учитесь на ошибках.

Литература

Ellis, Margaret A., Stroustrup B. «The Annotated C++ Reference Manual». Reading Mass: Addison-Wesley, 1990.

Глава 2
Абстракция и абстрактные типы данных

Типизованные языки программирования появились в 1950-х годах не как «наилучшие на сегодняшний день» инструменты качественного проектирования или структурирования систем, а лишь как средство генерирования *эффективного* объектного кода из исходного текста программ. Постепенно отношение к системам типизации стало меняться, и в них усмотрели самостоятельную ценность для проверки типов и типовой безопасности интерфейсов. В 1960-х годах типы развились до такой степени, что стали рассматриваться наравне с процедурами среди основных компонентов программ. Это привело к появлению новых более абстрактных синтаксических концепций и конструкций в языках программирования.

- *Абстрактные типы данных* (АТД) определяют интерфейс к абстракции данных без указания подробностей реализации. Абстрактный тип данных может иметь несколько реализаций для разных потребностей или для работы в разных средах.
- *Классы* реализуют абстрактные типы данных. Поддержка классов в языках программирования впервые появилась в языке Simula 67.
- *Улучшенные системы типизации* позволяют АТД занять свое законное место среди языковых абстракций. Перегрузка операторов и другие усовершенствования подняли мощь и выразительность АТД на новый уровень. Эти средства впервые появились в языках Bliss, Algol 68 и Simula, и были позднее усовершенствованы в Mesa и Clu.

В C++ встроенные типы (int, short, long, float, double, long double, char и указатели) обеспечивают проверку интерфейса на стадии компиляции и эффективность сгенерированного кода. Перечисленные абстракции существуют в любой среде C++, вы можете сразу задействовать их в качестве счетчиков и индексов или воспользоваться ими как строительными блоками для числового анализа, обработки символьной информации и т. д. Классы C++ позволяют объединить логически связанные данные с операторами и функциями и тем самым создать программную единицу, поддерживаемую на уровне языка. По своей мощи и удобству классы не уступают встроенным типам: они могут использоваться для объявления или динамического создания новых экземпляров, на них распространяется

механизм проверки типов. Интерфейс класса отделяется от его реализации и служит тем же целям, что и спецификация АТД.

Впрочем, это не означает, что экземпляры классов так же универсальны, как объекты встроенных типов int и double. Проектировщик класса может определить конструкции, интегрирующие новый класс в систему типов компилятора. Например, компилятор можно «научить» выполнять преобразование между встроенным типом double и экземпляром пользовательского класса Complex. Его также можно «научить» генерировать эффективный код создания, присваивания и копирования экземпляров класса. Ценой дополнительных усилий (большей частью по готовым «рецептам») класс преобразуется в пользовательский тип, который мы называем *конкретным типом данных*. В результате объекты класса объявляются, присваиваются и передаются в параметрах функций точно так же, как обычные встроенные типы C и C++.

Класс приносит пользу как абстракция из области проектирования и реализации даже без механизмов его преобразования в конкретный тип данных. Преобразование в конкретный тип данных проходит относительно легко, а дополнительные удобства конкретных типов данных обычно оправдывают затраченные усилия. Но и «простой» класс является хорошей отправной точкой, а приведенная в этой главе информация станет хорошим подспорьем в построении ваших собственных нетривиальных классов. Тема данной главы развивается в главе 3, посвященной преобразованию классов в конкретные типы данных.

В начале этой главы рассматривается концепция *класса* — вероятно, основополагающим механизмом абстракции в C++. Мы разберемся, как функции и данные объединяются в классе способом, вроде бы противоречащим традиционному подходу C. Далее описаны конструкции корректной инициализации и уничтожения переменных классов. Кроме того, в этой главе будут представлены другие конструкции и концепции C++, часто ассоциируемые с классами: подставляемые функции, константные функции и константные объекты классов, статические функции классов и указатели на функции классов.

2.1. Классы

Конструкция C++ class тесно связана с конструкцией C struct. C++ поддерживает объявления struct и сохраняет их семантику в языке C. Ключевое слово class обычно подчеркивает, что структура задействуется как пользовательский тип данных, а не как простой агрегат данных. А именно, при группировке функций и данных применяется ключевое слово class, а при группировке одних данных — ключевое слово struct. Функции, сгруппированные в классе, называются *функциями класса*; они «принадлежат» содержащему их классу. В C++ структуры (struct), как и классы (class), тоже могут содержать функции, но обычно они рассматриваются в традиционном смысле C как простые записи данных. Формально классы и структуры в C++ различаются только по уровню доступа, принятому по умолчанию. В листинге 2.1 показано сходство между структурами C и классами C++ при использовании в механизмах инкапсуляции данных.

Листинг 2.1. Аналогия между структурами C и классами C++

Код C:

```
struct {
   char name[NAME_SIZE];
   char id[ID_SIZE];
   short yearsExperience;
   int gender:1;
   unsigned char dependents:3;
   unsigned char exemptions:4;
} foo;

struct {
   char name[NAME_SIZE];
   char id[ID_SIZE];
   short yearsExperience;
   int gender:1;
   unsigned char dependents:3;
   unsigned char exemptions:4;
} bar;
```

Код C++:

```
struct {
   char name[NAME_SIZE];
   char id[ID_SIZE];
   short yearsExperience;
   int gender:1;
   unsigned char dependents:3;
   unsigned char exemptions:4;
} foo;

class {
public:
   char name[NAME_SIZE];
   char id[ID_SIZE];
   short yearsExperience;
   int gender:1;
   unsigned char dependents:3;
   unsigned char exemptions:4;
} bar;
```

В отличие от C, C++ вводит в программу новый тип для каждой помеченной (tagged) структуры или класса, словно после объявления в программе следует директива typedef:

Код C:

```
struct EmpLabel {
   char name[NAME_SIZE];
   char id[ID_SIZE];
   int gender:1;
   ...
} foo;

typedef struct EmpLabel Employee;

struct EmpLabel fred;
Employee lisa;
```

Код C++:

```
struct Employee {
   char name[NAME_SIZE];
   char id[ID_SIZE];
   int gender:1;
   ...
} foo;

struct Employee fred;
Employee lisa;
```

В C++ пометка и тип являются синонимами, тогда как в C они считаются разными именами.

Структуры C++ ведут себя как классы, все члены которых открыты по умолчанию, тогда как члены классов по умолчанию являются закрытыми и доступны только внутри самого класса. Уровень доступа изменяется при помощи ключевых слов private, public и protected. Одно из этих трех ключевых слов, за которым следует двоеточие, определяет уровень доступа всех последующих членов до конца класса или до следующего квалификатора. Уровень доступа по умолчанию действует до первого квалификатора.

- Открытые (public) члены объекта доступны для всех функций, для которых доступен класс данного объекта, а также сам объект по правилам области видимости.

- Защищенные (protected) члены объекта доступны только для функций класса данного объекта или его производных классов. Объект производного класса не может обращаться к защищенным полям объекта базового класса и даже объекта *собственного* базового класса. Функции производных классов могут обращаться к защищенным членам только базовой части объекта своего класса.
- Закрытые (private) члены объекта доступны только для функций класса этого объекта.

Инкапсуляцию, обеспечиваемую ключевыми словами private и protected, при желании можно нарушить при помощи механизма дружественных отношений (friend). Тема защиты подробно рассматривается в главе 3.

Хотя классы можно рассматривать как синтаксические расширения структур, мы обычно рассматриваем их как средство передачи информации об архитектуре программы. Структуры применяются для упаковки логически связанных данных в любой методике структуризации. Например, при функциональной декомпозиции (см. приложение Е) каждый уровень функций обладает собственными данными, и структуры могут использоваться для объединения взаимосвязанных данных (или для создания нескольких группировок взаимосвязанных данных) на определенном уровне. Структуры данных, сгенерированные при функциональной декомпозиции, чаще отражают не состояние абстракций в приложении, а скорее некие уловки, примененные в реализации.

Классы C++ обладают двумя возможностями, выходящими за пределы структур C: типизацией и абстракцией. Классы применяются для создания новых *пользовательских типов* в программах C — тех, которые мы называем абстрактными типами данных. Классы используются как компоненты при создании типов, поскольку наряду с представлением набора данных в них сохраняется информация о его поведении. Для примера рассмотрим класс комплексных чисел Complex:

```
class Complex {
public:
friend Complex& operator+(double, const Complex&);
       Complex& operator+(const Complex&) const;
friend Complex& operator-(double, const Complex&);
       Complex& operator-(const Complex&) const;
friend Complex& operator*(double, const Complex&);
       Complex& operator*(const Complex&) const;
friend Complex& operator/(double, const Complex&);
       Complex& operator/(const Complex&) const;
       double rpart() const;
       double ipart() const;
       Complex(double=0.0, double=0.0);
private:
       double realPart, imaginaryPart;
};
```

Здесь класс отражает не только логическую связь между переменными realPart и imaginaryPart, на что способна обычная структура, но и связь между представле-

нием комплексного числа и его поведением (то есть операциями). Конструкции языка C не позволяют хранить эти связи в удобном и интуитивно понятном виде, а лишь делают возможным их представление на уровне соглашений.

Классы помогают программисту использовать программные конструкции более высокого уровня, чем отдельно взятые функции или структуры. Эти конструкции служат *абстракциями*, а абстрагируемые сущности обычно тесно связаны со спецификой программируемого приложения. Применение классов выделяет отображение абстракций области приложения в абстракции области решения, на что простые структуры C не способны.

Хороший проектировщик может мыслить в терминах абстрактных типов данных и программировать в терминах структур и функций. В этом случае структура хорошей программы C++ соответствует структуре хорошей программы C, но как бы «выворачивается наизнанку». Это явление, называемое *объектной инверсией*, рассматривается в следующем разделе.

2.2. Объектная инверсия

Во многих программах имеются библиотеки и модули, построенные на основе некоторой структуры данных. Для примера возьмем простой стек. Данные стека хранятся в структуре, и с этой структурой связываются несколько процедур, работающих с объявленными в ней переменными.

В C++ все наоборот. Функции рассматриваются как *принадлежащие* структуре из-за тесной связи со структурой и ее данными. Вместо того чтобы передавать структуру Stack всем функциям, работающим с ней, мы сохраняем эти функции *внутри* структуры вместе с данными. Такие функции называются *функциями*, или *операторами*, *класса*. В других языках (и прежде всего в Smalltalk) они называются *методами*. В листинге 2.2 приведен пример стека, оформленный в этом стиле. (Хотя этот фрагмент работоспособен, он слишком упрощен и приводится только в учебных целях. Стек имеет фиксированный размер, его элементы однородны и относятся только к одному конкретному типу long.)

Листинг 2.2. Эквивалентные реализации стека на C и C++

Код C:

```
#define STACK_SIZE 10


struct Stack {
    long items[STACK_SIZE];
    int sp;
};

void Stack_initialize(s)
struct Stack *s;
{
```

Код C++:

```
const int STACK_SIZE = 10;

class Stack{
private:
    long items[STACK_SIZE];
    int sp;
public:
    void initialize();
    long top() const;
    long pop();
    void push(long);
```

продолжение

Листинг 2.2 (*продолжение*)

```
    s->sp = -1;
}

long Stack_top(s)
struct Stack *s;
{
    return s->items[s->sp];
}

long Stack_pop(s)
struct Stack *s;
{
    return s->items[s->sp--];
}

void Stack_push(s, i)
struct Stack *s; long i;
{
    s->items[++s->sp] = i;
}

int main()
{
    struct Stack q;
    long i;
    Stack_initialize(&q);
    Stack_push(&q,1);
    i = Stack_top(&q);
    Stack_pop(&q);
}
```

```
};

void Stack::initialize() {
    sp = -1;
}

long Stack::top() const {
    return items[sp];
}

long Stack::pop() {
    return items[sp--];
}

void Stack::push(long i) {
    items[++sp] = i;
}

int main()
{
    Stack q;

    q.initialize();
    q.push(1);
    long i = q.top();
    q.pop();
}
```

Обратите внимание: в C++ функции класса (в частности, функция push класса Stack, обозначаемая как Stack::push) не разыменовывают указатель на структуру и вообще обходятся без явных обращений к объекту класса, соответствующему структуре в реализации C. Это объясняется тем, что в C++ для классов создается новая область видимости, и функции могут рассматриваться как существующие *внутри* экземпляров данной структуры. Каждое объявление «переменной» класса Stack выделяет блок памяти (как для структуры). Например, экземпляр q в функции main программы C++ хранится в стеке времени выполнения. Этот экземпляр называется *объектом* класса Stack. Функции класса Stack применяются к объектам этого класса (например, q.initialize() или q.push(1)). При всех упоминаниях членов класса Stack в функциях этого класса используются данные того объекта, для которого была вызвана функция (в нашем примере — это объект q).

Функции класса обращаются к данным своего объекта через замаскированный параметр с именем this — указатель на объект, для которого была вызвана функция. Хотя параметр this передается незаметно для программиста, к нему можно обращаться в программе. Параметр this служит тем же целям, что и указатель на структуру Stack в функциях C; он используется так часто, что компилятор автоматически организует его передачу. Более того, компилятор автоматически присоединяет квалификатор this-> ко всем обращениям к членам класса в функциях этого класса. Например, внутри Stack::initialize выражение sp=-1 в действительности означает this->sp=-1. Явное разыменование this допустимо, но слишком длинно и обычно излишне. Следующие два фрагмента кода C++ эквивалентны:

```
void Stack::initialize() {
   sp = -1;
}
long Stack::top() const {
    return items[sp];
}
```

```
void Stack::initialize() {
   this->sp = -1;
}
long Stack::top() const {
    return this->items[this->sp];
}
```

Обратите внимание на обозначение const в функции Stack::top. Оно утверждает, что функция не изменяет того объекта, для которого она вызывается (то есть объекта, на который указывает параметр this). Конструкция const подробнее рассматривается в 2.9.

2.3. Конструкторы и деструкторы

При проектировании языка C++ были поставлены две важные цели:

- переменные всегда должны инициализироваться автоматически;
- классы должны контролировать операции с памятью для своих объектов.

Поддержка этих концепций на уровне языка позволяет авторам классов управлять выделением и освобождением памяти, чтобы избавить пользователей от этих хлопот. Для инициализации и уничтожения объектов определяются специальные функции, называемые соответственно *конструктором* и *деструктором*.

Компилятор особо выделяет функции, опознанные им как конструкторы и деструкторы по специальным именам. Имя конструктора совпадает с именем класса, которому принадлежит конструктор. Имя деструктора представляет собой имя класса с префиксом ~ (тильда). Хотя тильда обычно не отделяется от имени класса пробелами, это всего лишь условное соглашение, не обязательное по правилам языка.

При использовании C++ в «базовом» режиме, то есть без применения идиом, конструкторы и деструкторы не вызываются явно. Компилятор автоматически генерирует вызовы конструкторов и деструкторов для инициализации и уничтожения переменных по мере необходимости. Например, при входе в функцию с локальными объектными переменными автоматически вызываются конструкторы, а при возвращении из функции автоматически вызываются деструкторы.

Конструкторы и деструкторы также используют примитивы управления памятью new и delete для инициализации и уничтожения динамически созданных объектов.

Обратите внимание: конструкторы и деструкторы являются функциями класса, которые *не имеют* возвращаемого значения. Дело в том, что они практически никогда не вызываются напрямую, а компилятор генерирует их вызовы по мере необходимости. Казалось бы, для конструктора было бы естественно «возвращать» объект при завершении своей работы. Но конструктор с таким же успехом может вызываться как для инициализации существующего блока памяти, зарезервированного для объекта (например, глобального), так и для инициализации динамически созданных объектов (например, при использовании оператора new). Получение конструктором адреса объекта и его возврат производится компилятором незаметно для программиста. Впрочем, нестандартные языковые конструкции *позволяют* вызывать конструкторы и деструкторы для заданных объектов, что может быть полезно в особых ситуациях с нетривиальным управлением памятью, описанных в главе 3. И все же на практике чаще встречаются автоматически сгенерированные вызовы.

Если класс используется для простого объединения или абстрагирования данных, определять для него конструкторы или деструкторы не обязательно. Их *нужно* определять только для классов с динамической внутренней структурой, которые ведут себя как естественные типы языков программирования. Так, в приведенном ранее примере стека нет ни конструкторов, ни деструкторов; класс содержит функцию initialize, которая должна явно вызываться в программе для приведения объекта в нормальное состояние. Эту функцию можно заменить конструктором, чтобы класс Stack сам инициализировал свою внутреннюю структуру, а пользователи класса Stack избавились от лишних хлопот. Сначала мы изменяем объявление класса:

```
class Stack {
public:
    Stack();
    long top() const;
    long pop();
    void push(long);
private:
    long items[10];
    int sp;
};
```

А затем заменяем функцию initialize конструктором:

```
Stack::Stack() {
    sp = -1;
}
```

Теперь функцию initialize не нужно вызывать в программе main; конструктор вызывается *автоматически* при создании объекта Stack, как и при объявлении q.

Таким образом, вызов initialize просто исключается из main, и программа продолжает нормально работать.

Перейдем к более общей реализации стека, которая поближе познакомит нас с конструкторами, а также даст начальное представление о деструкторах (листинг 2.3).

Листинг 2.3. Объявление стека с конструкторами и деструктором

```
const int STACK_SIZE = 10;

class Stack{
public:
    Stack();              // Первый конструктор
    Stack(int);           // Второй конструктор
    ~Stack();             // Деструктор
    long top() const;
    long pop();
    void push(long);
private:
    long *items;
    int sp;
};
```

В исходной версии класс Stack имел фиксированный размер в десять элементов; мы хотим, чтобы в момент объявления можно было создать стек с большим или меньшим количеством элементов. Для этого в классе определяется конструктор с параметром, в котором передается размер стека. Конструктор динамически выделяет память оператором C++ new:

```
Stack::Stack(int size) {
    items = new long[size];
}
```

Так как конструктор динамически выделяет память, в программу нужно включить деструктор для освобождения памяти при уничтожении объекта. Для этого деструктор применяет оператор delete к указателю на динамически выделенный блок:

```
Stack::~Stack() {
    delete[] items;
}
```

Квадратные скобки в синтаксисе delete[] items сообщают компилятору о том, что items указывает на *вектор* типа long, а не на одиночный динамически созданный экземпляр long.

Обратите внимание: новая версия класса содержит два конструктора с одинаковыми именами! Имя Stack «наполняется» двумя смыслами — это называется *перегрузкой*. Наличие одноименных функций не создает никакой путаницы или двусмысленности; если по контексту должен вызываться конструктор с аргументом, будет вызван конструктор с аргументом. В противном случае вызывается другой конструктор, который создает стек со стандартным размером в 10 элементов. Компилятор автоматически выбирает нужную функцию (вскоре вы поймете, как

это делается). Реализация нового объявления класса Stack, приведенного в листинге 2.3, представлена в листинге 2.4.

Листинг 2.4. Реализация стека с конструкторами и деструктором

```
Stack::Stack() {
    items = new long[STACK_SIZE];
    sp = -1;
}

Stack::Stack(int size) {
    items = new long[size]; // Аналог типизованного вызова sbrk
                            // или malloc, но с вызовом конструктора,
                            // если он имеется
    sp = -1;
}

Stack::~Stack() {
    delete[] items;
}

long Stack::top() const {
    return items[sp];
}

long Stack::pop() {
    return items[sp--];
}

void Stack::push(long i) {
    items[++sp] = i;
}

int main()
{
    Stack q;        // Вызов Stack::Stack()
    Stack r(15);    // Вызов Stack::Stack(int)
    q.push(1);
    int i = q.top();
    q.pop();
}
```

Если класс *не содержит* конструктора, то инициализация объектов класса не гарантирована. Если класс без конструктора содержит другие объекты классов, имеющих конструкторы по умолчанию, то эти объекты будут правильно инициализированы, однако содержимое других полей остается неопределенным. Для примера рассмотрим код в листинге 2.5. Объекты класса B не содержат внутренних объектов классов, только экземпляры встроенных типов C (указатель на char и short). Но конструктор B берет на себя все хлопоты по инициализации полей своих объектов. Экземпляры C содержат только внутренний объект B и int; объекты класса D содержат только int. Программа main создает объекты классов

C и D, и нетрудно заметить, что поля примитивных типов классов C и D остаются неинициализированными. Если бы эти классы имели конструкторы, и если бы конструкторы определяли порядок инициализации этих полей, то инициализация была бы полной.

Листинг 2.5. Инициализация полей класса переменных типов

```
class B {
public:
    B() { p = 0; s = 0; }
    int f();
private:
    char *p;
    short s;
};

class C {
public:
    int g();  // Нет конструктора
private:
    int i;
    B b;
};

class D {
public:
    int h();
private:
    int j, k;
};

C gc;     // p и s инициализированы, переменная i обнулена

int main() {
    C c;  // p и s инициализированы, значение i не определено
    D d;  // j и k не инициализированы
    int l = c.g();
}
```

Как упоминалось ранее, класс C содержит переменную класса B. Так как класс B содержит конструктор, поля C::b будут инициализироваться при каждом создании объекта C. Конструктор B::B вызывается из конструктора по умолчанию (не имеющего параметров), автоматически сгенерированного компилятором для класса C. Компилятор обязан предоставить этот конструктор, чтобы у него было место для вызова C::b. Но код, сгенерированный компилятором, инициализирует только те члены, у которых существуют конструкторы, поэтому при отсутствии конструктора будут неявно инициализированы только объектные члены классов. Члены, объявленные с встроенным типом, *не инициализируются* автоматически. Такое поведение логически согласовано с отсутствием инициализации структур в C.

2.4. Подставляемые функции

Модификатор inline запрашивает подстановку функции, то есть ее расширение в точке вызова, чтобы предотвратить затраты на вызов. Компилятор пытается подставить код функции, объявленной с ключевым словом inline, перед ее использованием, если точки объявления и использования находятся в одном исходном потоке (то есть являются частью одного исходного файла или совокупности исходных файлов, включенных директивами #include). Подстановка иногда приводит к значительной экономии процессорного времени: так, один программист обнаружил, что переход на подставляемые функции в программном комплексе имитации оборудования ускорил работу в четыре раза!

В языке C в течение долгого времени подставляемые функции имитировались при помощи макросов препроцессора. В следующем примере макрос имитирует функцию вычисления модуля (абсолютного значения):

```
#define abs(x) (x < 0 ? (-x) : x)

int main() {
    int i, j;
    ...
    i = abs(j);
    ...
}
```

Подставляемые функции не только заменяют препроцессорные макросы, но и превосходят их, потому что они подчиняются тем же правилам типизации и области видимости, что и обычные функции. Сгенерированный ими код не уступает аналогичным макросам по эффективности, а очень часто и превосходит их (см. упражнения в конце главы). Иногда подставляемые функции упрощают процесс отладки. Например, некоторые среды программирования на C++ позволяют отключить подстановку функций при помощи флага компилятора, чтобы все функции компилировались в автономные фрагменты кода. А это означает, что отладчик может легко устанавливать в них точки прерывания (тогда как в макросах это обычно невозможно).

Чтобы объявить функцию подставляемой, вставьте в ее определение ключевое слово inline:

```
inline int abs(int i) {return i < 0? -i: i; }

class Stack {
    ...
    long pop();
};

inline long Stack::pop() {
    return items[sp--];
}
```

Если подставляемой требуется объявить функцию класса, достаточно включить ее определение в объявление; впрочем, делать это не рекомендуется, потому что при этом усложняется администрирование программы (см. раздел 2.11). Тем не менее, синтаксис «определения при объявлении» будет использоваться в книге для наглядности, без связи с подстановкой. Пример подставляемой функции класса:

```
class Stack {
  ...
  long pop() { return items[sp--]; } // Определение подставляемой функции
};
```

Определение подставляемой функции (и конечно, объявление функции с ключевым словом inline) должно располагаться в исходном тексте программы до первого вызова этой функции. Объявление функции как подставляемой после вызова является ошибкой. Рассмотрим следующий пример:

```
class C {
public:
    inline int a() { return 1; }
    int b()        { return 2; }  // Тоже подставляемая
    int c();
    int d();
    int e();
};

inline int C::d() {  // Ошибка: C::c вызывается до объявления
    return c() + 1;  // функции подставляемой (см. ниже)
{

inline int C::c() { return 0; }

int C::e() {
    return c() + 5;    // Функция C::c подставляется
}

int main() {
    C o;
    int i = o.d();     // C::d подставляется,
                       // а вложенный вызов C::c -- нет
}
```

Некоторые реализации C++ устанавливают собственные ограничения на подстановку. Например, компилятор может запретить подстановку для функций в циклах (хотя первая итерация может подставляться) или рекурсивных функций (хотя начальные итерации при частичной рекурсии тоже могут подставляться). В случае превышения некоторого порога сложности, определяемого компилятором, подстановка запрещается.

Подставляемые функции не являются панацеей в плане повышения быстродействия. За ускорение работы программы часто приходится расплачиваться

увеличением объема ее кода; коэффициент увеличения зависит от среды программирования. В данном случае действует правило 80/20: найдите 20 % кода, в котором ваша программа проводит 80 % времени, и преобразуйте в подстановку вызовы функций из внутренних циклов. Выделение кандидатов на подстановку в «горячих точках» программы является простым и эффективным способом повышения быстродействия.

2.5. Инициализация статических переменных

Классы C++ могут содержать статические переменные, которые являются концептуальными аналогами статических функций классов. Статические переменные объявляются в интерфейсе класса, обычно в заголовочных файлах (чаще всего с расширением .h, хотя это зависит от системы). Определения этих данных (возможно — с инициализаторами, хотя это и не обязательно) должны находиться не в заголовках, а в файлах с расширением .c (.C, .CPP и т. д., в зависимости от того, какое расширение используется в вашей системе для файлов с исходными текстами C++). Пример приведен в листинге 2.6.

Листинг 2.6. Инициализация статических переменных класса

```
Объявления в заголовочных файлах:
struct X {
    X(int, int);
    ...
};

struct s {
    static const int a;
    static X b;
    ...
    int f();
};

Определения в файле с расширением .c:
const int s::a = 7;
X s::b(1,2);
int s::f() { return a; }
```

Статические переменные классов уменьшают количество глобальных имен в программе. Кроме того, они наделяют объекты некоторыми полезными свойствами обычных переменных, а также позволяют закрепить за ними данные, существующие на уровне класса в целом, а не отдельного объекта, например, правила управления доступом. Статические переменные классов очень удобны для разработчиков библиотек, потому что они предотвращают загромождение глобального пространства имен, упрощают программирование библиотек, а также использование нескольких библиотек в программе.

Все эти свойства присущи как функциям, так и объектам, и наряду со статическими переменными классов существует аналогичная концепция статических функций классов, о которых речь пойдет в следующем разделе.

2.6. Статические функции классов

Первый опыт использования C++ показал, что *большинство* имен, локальных по отношению к библиотекам, составляли имена функций. Оказалось, что для имитации статических функций классов применялся непереносимый код вида:

```
((X*)0)->f();
```

Этот «фокус» не работал в некоторых реализациях динамической компоновки, а язык не обеспечивал никакой разумной семантики.

Статические функции классов C++ напоминают глобальные функции, область видимости которых определяется границами класса. По эффективности они не уступают вызовам глобальных функций и немного превосходят вызовы обычных функций классов:

```
class X {
public:
    // ...
    void foo() { fooCounter++; ... }
    static void printCounts() {
        printf("foo called %d times\n", fooCounter);
    }
private:
    static int fooCounter;
};

int X::fooCounter = 0;

int main() {
    printCounts();    // Ошибка (если не существует глобальной
                      // функции с таким именем)
    X::printCounts(); // Нормально
}
```

Статические функции классов обычно применяются для управления ресурсами, общими для всех объектов класса. Как правило, статические функции классов работают со статическими данными классов. Например, в переменных класса могут храниться счетчики вызова его функций; каждая функция класса ведет счет своих вызовов в собственной переменной static int. С другой стороны, одна или несколько статических функций используются для сброса, чтения или вывода значений счетчиков.

Другое возможное применение статических функций класса связано с группировкой функций, совокупность которых не должна рассматриваться как объект. Связи между такими функциями могут быть слабее тех, которые объединяют большинство

функций в объектах, и они могут не относиться к ресурсам уровня проектирования (см. главу 6). В некотором смысле такие конструкции моделируют концепцию пакета в языке Ada. Они будут подробно рассматриваться в главе 11.

2.7. Область видимости и константность

В языке C конструкция #define заменяет константные значения символическими именами. Применение именованных констант лучше отвечает духу C++. Обычно именованные константы могут применяться везде, где в C допускается использование директив #define. Константы C++ могут обладать областью видимости, поэтому константы, специфические для конкретного класса, не загромождают глобальное пространство имен — макросы таким свойством не обладают.

Константы, используемые локально по отношению к файлу, можно объявлять и определять в этом файле; они останутся невидимыми для всех остальных файлов той же программы:

```
const int MAX_STACK_SIZE = 1000;
const char *const paradigm = "The Object Paradigm";
```

Константы также могут совместно использоваться несколькими исходными файлами, для чего их следует объявить с ключевым словом extern. Такие объявления обычно размещаются в заголовке, включаемом во все файлы:

```
extern const double e;
```

При этом *единственное* определение значения помещается в удобный исходный файл (с расширением .c, .C, .cpp и т. д.):

```
#include <math.h>
extern const double e = exp(1.0);
```

Синтаксис константных статических членов классов усложняет объявление символических констант внутри класса. В частности, значение статических констант, являющихся членами классов, не может задаваться на стадии компиляции. А это означает, что их значения не могут использоваться для объявления размера массива:

```
static const int SIZE1 = 10;

class C {
    ...
    static const int SIZE2;
    char vec1[SIZE1];          // OK
    char vec2[SIZE2];          // Нельзя, значение SIZE2 неизвестно
};

const int C::SIZE2 = 10;
```

Чтобы преодолеть этот недостаток ключевого слова const, мы используем конструкцию, которая является его синонимом: перечисляемые типы (enum) содержат фиксированные целочисленные значения и могут справиться с задачами, недос-

тупными для типа const int. Перечисляемый тип в C++, как и в C, обычно определяется в виде набора символических целых констант, которые могут использоваться в командах switch и if, но конкретные целые значения которых для нас несущественны. В объявлении enum символическим именам можно присвоить конкретные значения, наделив их свойствами символических констант. В следующем фрагменте в классе Stack создается «константа» StackSize:

```
class Stack {
public:
    void push(int);
    int pop();
    int top() const;
private:
    enum { StackSize = 100 };    // Имитация константы
    int rep[StackSize], sp;
};

void Stack::push(int el) {
    if (sp >= StackSize) error ("stack overflow");
    else rep[sp++] = el;
}
```

2.8. Порядок инициализации глобальных объектов, констант и статических членов классов

C++ гарантирует, что переменные инициализируются в порядке их следования в исходном файле, но при этом не существует гарантированного порядка инициализации глобальных объектов в нескольких исходных файлах. Это относится ко всем глобальным объектным переменным, но особенно важно для статических переменных классов, потому что инициализация глобальных объектов может зависеть от действительности значений статических констант их классов.

Любые предположения относительно порядка инициализации могут приводить к трудноуловимым ошибкам. Для примера рассмотрим простой заголовочный файл Angle.h, содержащий следующие объявления:

```
#include <math.h>
#include <stdio.h>

extern const double pi;

class Angle {
public:
    Angle(double degrees) {
        r = degrees * pi / 180.0;
    }
    ...
```

```
    void print() {
        printf("radians = %f\n", r);
    }
private:
    double r;
};
```

Теперь рассмотрим исходный файл **pi1.c** с определением константы **pi**:

```
#include <Angle.h>
extern const double pi = 4.0 * atan(1.0);
```

Далее рассмотрим исходный файл **pi2.c** с кодом приложения:

```
#include <Angle.h>

Angle northeast = 45;

int main() {
    Angle northeast2 = 45;
    northeast.print();
    northeast2.print();
    return 0;
}
```

Если откомпилировать и запустить эту программу в системе автора, будет получен следующий результат:

```
radians = 0.000000
radians = 0.785398
```

Причина в том, что значение **pi** еще не было инициализировано к моменту вызова конструктора **northeast**.

Если вам действительно необходимо управлять порядком инициализации, воспользуйтесь методикой, представленной в разделе 3.7. И все же лучше по возможности избегать этих зависимостей, особенно в коде библиотек, с которыми будут работать другие пользователи.

2.9. Обеспечение константности функций классов

В версию 2.0 языка C++ были включены некоторые новые варианты применения модификатора **const**. В предыдущих версиях C++ значение константного объекта можно было изменить, вызвав для него функцию класса. Многие жаловались на проблемы, возникающие из-за этой ошибки.

Если вообще запретить вызов функций класса для константных объектов, последние станут бесполезными. Для сохранения полноты языка в версии 2.0 были представлены *константные функции классов*, которые могут вызываться для

константных объектов. Компилятор гарантирует, что константная функция не изменит того объекта, для которого она вызывается, если только программист не попытается сознательно обойти это ограничение (см. обсуждение «логической константности» в следующем разделе). Кроме того, гарантируется, что для константных объектов будут вызываться только константные функции. Пример приведен в листинге 2.7. Использование ключевого слова const в качестве суффикса при объявлении списка параметров функции соответствует его применению в качестве суффикса для символа *. Из-за раздельной компиляции мы не можем рассчитывать на то, что сам компилятор (без помощи со стороны пользователя) обнаружит вызов для константного объекта функции класса, модифицирующей свой объект.

Листинг 2.7. Константные функции класса

```
struct s {
    int a;
    f(int aa) { return a = aa; }
    g(int aa) const { return aa; }
    //  h(int aa) const { return a = aa; }
};

void g()
{
    s o1;
    const s o2;
    o1.a = 1;
    //  o2.a = 2;
    o1.f(3);
    //  o2.f(4);
    o1.g(5);
    o2.g(6);
}

/*
Если удалить признаки комментария (//),
компилятор выдаст следующие сообщения об ошибках:
"", line 5: error: assignment to member s::a of const struct s
"", line 13: error: assignment to member s::a of const s
"", line 15: warning: non const member function s::f() called
             for const object
*/
```

Логическая и физическая константность

Состояние объекта — то есть совокупность значений всех его данных — определяет работу функций класса. Состояние объекта имеет три составляющие:

- данные, отражающие состояние приложения (например, факт поступления некоторого сообщения или сигнала);

- данные, отдаленно относящиеся к состоянию приложения, но являющиеся артефактами реализации (например, Stack::sp);
- данные, используемые при отладке и администрировании и не связанные с семантикой приложения (например, счетчик вызовов некоторой функции).

Представленная классификация не идеальна. Например, такие данные, как счетчики ссылок, относятся как ко второй, так и к третьей категориям, а некоторые состояния ограничиваются первыми двумя категориями. Тем не менее, подобная классификация помогает подчеркнуть один важный аспект во взаимодействии функций класса с константными объектами.

Для примера рассмотрим класс бинарного дерева BinaryTree с функцией findNode, которая возвращает копию узла, найденного по некоторому критерию. Алгоритм может быть оптимизирован так, чтобы при слишком большом времени поиска узла (которое определяется по пороговому значению, зависящему от количества узлов в дереве) балансировка дерева производилась заново. Вроде бы логично объявить функцию findNode константной, но из-за возможной повторной балансировки такое объявление вызовет протест у компилятора: константные функции классов не могут модифицировать данные своих объектов.

Такие случаи встречаются относительно редко, и для них существует элементарный обходной путь. Фокус основан на обращении к локальному объекту через указатель, который является синонимом для this:

```
T BinaryTree::findNode(String key) const {
    ...
    if (needReBalance) {
        // Сюда следует вставить комментарий, объясняющий,
        // зачем нарушается константность объекта
        BinaryTree *This = (BinaryTree*)this;
        ...
        This->left = This->left->left;
        ...
    }
}
```

Другое возможное решение основано на создании *ссылки* на this, которая ведет себя как синоним, но с меньшими ограничениями:

```
T BinaryTree::findNode(String key) const {
    ..
    if (needReBalance) {
        typedef BinaryTree *BinaryTreePointer;
        const BinaryTreePointer &This = (BinaryTree*)this;
        ...
        This->left = This->left->left;
        ...
    }
}
```

Вторая форма подчеркивает, что This всего лишь является константным синонимом для this. Скорее всего, сгенерированный код получится таким же, как в первом случае. Если функция объявлена подставляемой, одно из этих решений может генерировать меньший объем кода (в зависимости от реализации компилятора).

Класс String дает обратный пример. Он имеет очевидную реализацию в виде указателя на динамически выделенный блок памяти, содержащий символьный вектор. Функция String::getChar может определяться таким образом, чтобы она удаляла из строки и возвращала последний символ. Функция класса не может изменять состояние самого объекта, то есть содержимое указателя должно оставаться неизменным. Однако *логическое* состояние объекта при этом изменяется, поэтому функцию getChar не следует объявлять константной, хотя компилятор и разрешит это сделать.

2.10. Указатели на функции классов

В некоторых ситуациях требуется изменить поведение функции на стадии выполнения программы. Для решения этой задачи можно воспользоваться указателями на функции классов. Рассмотрим класс, представляющий фильтр в электрической цепи из последовательно подключенных индуктивности, конденсатора и сопротивления[1]. Фильтр является особой разновидностью цепи, называемой RLC-цепью (где R, L и C — обозначения соответственно для сопротивления, индуктивности и конденсатора, принятые в электротехнике). На практике часто требуется определить переходную характеристику фильтра, то есть ток, получаемый при мгновенном изменении поданного напряжения. Мы хотим создать объект, моделирующий поведение фильтра. Конструктор класса должен получать параметры, задающие индуктивность, емкость, сопротивление, начальную силу тока и напряжение, и т. д. Функция класса вычисляет силу тока, проходящего через цепь, как функцию времени. Возможны три характерных режима поведения компонента: чрезмерное затухание, недостаточное затухание и критическое затухание. Режим работы фильтра определяется параметрами конструктора.

Каждый из трех режимов описывается формулой, задающей силу тока как функцию времени. Понимание этих формул не обязательно для данного примера, но на всякий случай приведем краткую сводку. Формула чрезмерного затухания:

$$i(t) = A_1 e^{s_1 t} + A_2 e^{s_2 t}.$$

Формула критического затухания:

$$i(t) = e^{-\alpha t}(A_1 t + A_2).$$

Наконец, для недостаточного затухания зависимость выглядит так:

$$i(t) = e^{-\alpha t}(B_1 \cos \omega_d t + B_2 \sin \omega_d t).$$

[1] Спасибо Дону Стейну (Don Stein) за некоторые идеи для этого раздела.

Здесь:

$$s_{1,2} = -\alpha \pm j\omega_d,$$

$$\alpha = \frac{R}{2L},$$

$$\omega_0 = \frac{1}{\sqrt{LC}},$$

$$\omega_d = \sqrt{\omega_0^2 - \alpha^2}.$$

Каждый, кто изучал физику, узнает эти уравнения (и сможет использовать код) для моделирования характеристик массы, закрепленной на пружине в вязкой среде. Аналогии существуют и в других областях.

Наш класс должен выглядеть так, словно он содержит всего одну функцию current, внутреннее поведение которой радикально изменяется в зависимости от контекста вызова.

В C++ можно объявить указатель на функцию класса и использовать полученную переменную для вызова функции (при условии, что указатель должным образом инициализирован). Объявление указателя на функцию класса должно задавать как тип класса, содержащего функцию, так и сигнатуру (интерфейс) самой функции; даже указатели на функции участвуют в системе проверки типов. Во время выполнения программы указателю на функцию класса можно присваивать адреса разных функций класса, однако все эти функции должны иметь одинаковые типы параметров и возвращаемого значения.

Ниже приводятся примеры объявлений функций и указателей для работы с ними:

Объявления функций	Совместимые объявления указателей
`int String::curColumn();`	
`int String::length();`	`int (String::*p1)();`
`int String::hash();`	
`char Stack::pop(int);`	`char (Stack::*p2)(int);`
`void Stack::push(char);`	`void (Stack::*p3)(char);`
`int PathName::error(int,` `  const char* ...);`	`int (PathName::*p4)(int,` `  const char* ...);`

Примеры инициализации этих указателей:

```
p1 = &String::length;
p2 = &Stack::pop;
p3 = &Stack::push;
p4 = &PathName::error;
```

А вот как эти функции вызываются:

```
int main() {
    String s;
    Stack t;
    PathName name, *namePointer = new PathName;
    p4 = PathName::error;

    int m = (s.*p1)();
    char c = (t.*p2)(2);
    (t.*p3)('a');
    (name.*p4)(1, "at line %d\n", __LINE__);
    (namePointer->*p4)(3, "another error (%d) in file %s",
        __LINE__, __FILE__);
    return 0;
}
```

Также допускается объявление указателей на переменные класса:

```
class Table {
public:
    sort();
    ...
};

class X {
public:
   table t1, t2;
   ...
};

int main() {
    Table X::*tablePointer = &X::t1;
    X a, *b = new X;
    (a.*tablePointer).sort();     // a.t1.sort()
    (b->*tablePointer).sort();    // b->t1.sort()
    tablePointer = &X::t2;
    (a.*tablePointer).sort();     // a.t2.sort()
    (b->*tablePointer).sort();    // b->t2.sort()
    return 0;
}
```

Как же может выглядеть модель электрической цепи? В листинге 2.8 приведен класс, определяющий реакцию фильтра. В параметрах конструктора класса передаются значения сопротивления (r), индуктивности (l) и емкости (c), а также исходная сила тока в цепи. Мы создаем объект этого класса, а затем вызываем функцию current с параметром time, чтобы узнать силу тока в любой момент времени.

Листинг 2.8. Класс, характеризующий отклик резонансной системы

```
#include <complex.h>

typedef double time;

class SeriesRLCStepResponse {
public:
    complex (SeriesRLCStepResponse::*current)(time t);
    SeriesRLCStepResponse(double r, double l,
        double c, double initialCurrent);
    double frequency() const { return 1.0 / sqrt(L * C); }
private:
    complex underDampedResponse(time t) {
        return exp(-alpha * t) * (b1 * cos(omegad * t) +
               b2 * sin(omegad * t));
    }
    complex overDampedResponse(time t) {
        return a1 * exp(s1 * t) + a2 * exp(s2 * t);
    }
    complex criticallyDampedResponse(time t) {
        return exp(-alpha * t) * (a1 * t + a2);
    }
    double R, L, C, currentT0, alpha;
    complex omegad, a1, b1, a2, b2, s1, s2;
};
```

Тем не менее «функция класса» current вообще не является функцией; это всего лишь указатель на функцию, значение которого задается конструктором. Конструктор выбирает функцию отклика на основании параметров фильтра и устанавливает указатель current на нужную функцию. Ниже приводится код конструктора с инициализацией указателя current:

```
SeriesRLCStepResponse::SeriesRLCStepResponse(
   double r, double l, double c, double initialCurrent) {
    R = r; L = l; C = c; currentT0 = initialCurrent;
    alpha = R / (L + L);
    omegad = sqrt(frequency()*frequency() - alpha*alpha);
    ... Вычисление a1, b1, a2, b2 и т. д. ...
    if (alpha < frequency()) {
        current =
          &SeriesRLCStepResponse::underDampedResponse;
    } else if (alpha > frequency()) {
        current =
          &SeriesRLCStepResponse::overDampedResponse;
    } else {
        current =
          &SeriesRLCStepResponse::criticallyDampedResponse;
    }
}
```

Остается рассмотреть простое приложение, в котором используется класс фильтра:

```
int main()
{
    double R, L, C, IO;
    cin >> R >> L >> C >> IO;
    SeriesRLCStepResponse aFilter(R, L, C, IO);
    for (time t = 1.0; t < 100; t += 1.0) {
        cout << (aFilter.*(aFilter.current))(t) << endl;
    }
    return 0;
}
```

Задачи, решаемые с помощью указателей на функции, часто имеют другое, более элегантное решение. Так, в приведенном примере можно воспользоваться специальными объектами, называемыми *функторами*, и заменить каждую функцию объектом. Эта методика основана на вызове виртуальных функций. Она будет подробно рассмотрена (на том же примере) в разделе 5.6.

2.11. Правила организации программного кода

В результате многолетнего опыта практического программирования на C++ были выработаны некоторые правила организации программного кода [1]. В большинстве своем эти правила достаточно очевидны и элементарны, но для порядка стоит повторить их еще раз.

Объявления класса обычно хранятся в заголовочных файлах. Заголовочный файл представляет собой исходный файл C++ с расширением .h. Имя файла определяется по очевидной схеме: так, для класса Stack файл будет называться Stack.h. Термин «объявление» подразумевает, что эта информация нужна *пользователю* класса, а не тому, кто занимается его реализацией. Исходный код реализации класса в основном хранится в исходных файлах C++, имеющих расширение .c (например, Stack.c).

Объявления класса упорядочиваются таким образом, что сначала перечисляются открытые (public) члены (обычно представляющие наибольший интерес для пользователей класса), за ними следуют защищенные (protected) члены, и в последнюю очередь перечисляются закрытые (private) члены. Подставляемые функции обычно выделяются из интерфейса и размещаются в заголовочном файле после объявления своего класса с использованием ключевого слова inline. Отделение тела подставляемых функций от объявления делает интерфейс класса менее громоздким. Оптимальное разделение достигается размещением определений подставляемых функций в отдельном заголовочном файле, который включается директивой #include из заголовочного файла класса или из кода приложения. Подобная организация также упрощает преобразование неподставляемых функций

в подставляемые или наоборот. Весь заголовочный файл заключается в директивы условной компиляции, предотвращающие повторное включение при вложенных директивах #include:

```
// Stack.h версия 1.4

#ifndef _STACK_H
#define _STACK_H 1
class Stack {
public:
    Stack();              // Первый конструктор
    Stack(int);           // Второй конструктор
    ~Stack();             // Деструктор
    long top() const;
    long pop();
    void push(long);
private:
    long *items;
    int sp;
};

inline void Stack::push(long i) {
    items[++sp] = i;
}
#endif _STACK_H
```

Упражнения

1. Перепишите пример стека с помощью механизма шаблонов C++, чтобы этот код мог использоваться для создания стеков с элементами произвольного типа.
2. Усовершенствуйте класс **Stack** так, чтобы следующий фрагмент создавал стек и читал его элементы из потока **stdin** (или любого другого заданного потока) до обнаружения признака конца файла:

   ```
   #include <stdio.h>
   ...
   Stack interactiveStack = stdin;
   ```

3. Функция UNIX **open** получает аргумент **const char***, определяющий имя открываемого файла. Разработайте примерную структуру заголовочных файлов и кода библиотеки, которая позволяла бы использовать любые из следующих конструкций:

   ```
   String group("/etc/group");
   FILE *f = fopen("input", "r");
   int fd1 = open("/etc/passwd", O_RDONLY);
   int fd2 = open(group, O_RDONLY);
   ```

```
int fd3 = open("outfile", O_CREAT, 0640);
int fd4 = open(fd2, O_NDELAY);
          // (Подсказка: fcntl(2), fcntl(5))
int fd5 = open(f, O_RDONLY);
```

4. Пакет Curses [2] — простая оконная система, часто используемая в редакторах и игровых программах UNIX. Найдите руководство по Curses для своей локальной установки и преобразуйте структуру окна (она может называться _win_st или **WINDOW**) в класс. Составьте объявления функций класса addch, getch, addstr, getstr, move, clear, erase и т. д.
5. Доработайте предыдущий пример и создайте работоспособный класс Window. Функции класса должны для своей работы вызывать функции Curses.
6. Используя макросы препроцессора, можно определить аналог подставляемой функции, который будет добавлять расширение .c к экземпляру абстрактного типа данных String:

```
// Оператор + определен как оператор конкатенации строк
#define addsuf(s) (s + ".c")
```

   - Определите подставляемую функцию C++, которая бы делала то же самое.
   - Напишите небольшую программу C++ с использованием обоих решений.
   - Включите в эту программу второй вызов макро-версии и определите, на сколько байтов возрастет объем программы.
   - Замените второй вызов макроса вызовом (вторым) подставляемой функции. Проверьте новый размер программы. Объясните разницу между приращением объема в двух случаях.
7. Может ли конструктор определить, был ли он вызван в результате выполнения оператора new или же для объекта, для которого уже была выделена память? Почему?
8. Напишите эффективный класс стека, максимальный размер которого не ограничивается фиксированным значением.

Литература

1. Kirslis Peter A., «A Style for Writing C++ Classes», Proceedings of the USENIX C++ Workshop, Santa Fe: USENIX Association Publishers (November 1987).
2. Strang, John. «Programming with Curses», Newton, Mass.: Reilly & Associates, 1986.

Глава 3
Конкретные типы данных

Конкретными типами данных называются типы, определяемые программистом. Их представление определяется на основе примитивных типов C, а также других абстрактных или конкретных типов данных, определенных вами. Для конкретных типов можно определять операторы, их имена и поведение. Кроме того, для конкретного типа можно переопределить смысл стандартных операторов C (таких, как +, –, / и *). Короче говоря, определенный вами тип становится таким же конкретным, как встроенные типы C `int`, `char`, `double` и т. д.

Конкретные типы данных следует отличать от реализаций классов, о которых рассказывалось в предыдущей главе. Конкретные типы работают по таким же предсказуемым правилам, что и встроенные типы C (скажем, `int`). Они создаются по специальному образцу, в котором члены классов конкретных типов данных передают информацию системе поддержки типов компилятора C++. Это позволяет компилятору генерировать эффективный и надежный код для абстракций произвольной сложности. Мы будем называть эту форму *ортодоксальной канонической формой класса*. Термин «каноническая» означает, что форма определяет систему правил, которые должны соблюдаться компилятором при генерировании кода, а «ортодоксальная» — что форма непосредственно поддерживается самим языком. Именно такой подход рекомендуется для решения практических задач на C++, потому что он дает программисту свободу действий и помогает избежать неожиданностей. В принципе реализация АТД возможна и без дополнительной «нагрузки» в виде канонической формы класса. Такой подход встречается в некоторых приложениях, однако каноническая форма не требует особого труда, и связанные с ней хлопоты обычно оправдываются (за исключением наиболее тривиальных случаев).

В начале настоящей главы мы познакомимся с ортодоксальной канонической формой. Затем будут описаны средства ограничения доступа к данным C++, обеспечивающие скрытие информации в классах. Далее рассматривается механизм перегрузки, необходимый для многих канонических форм и просто полезный для построения классов (например, для классов комплексных чисел). Пользователь также должен указать, как должно происходить автоматическое преобразование объектов существующих типов в объекты нового класса. В разделе 3.1 показано, как определяются такие преобразования.

В оставшейся части главы обсуждаются три важных аспекта управления динамической памятью, играющей важнейшую роль во многих объектно-ориентированных программах. Сначала мы рассмотрим идиомы, снижающие затраты на копирование объектов посредством подсчета ссылок. Далее описываются языковые средства, при помощи которых программист может управлять структурами данных и алгоритмами выделения/освобождения системной памяти. В завершение главы мы рассмотрим связь между созданием и инициализацией объектов, а также примеры разделения этих двух фаз.

3.1. Ортодоксальная каноническая форма класса

Ортодоксальная каноническая форма принадлежит к числу важнейших идиом C++ и лежит в основе едва ли не всего материала книги. Если вы будете следовать этому «рецепту» при определении своих классов, то переменная, созданная на базе такого класса, будет объявляться, присваиваться и передаваться в аргументах точно так же, как любая переменная C.

Допустим, вы хотите создать строковый класс String с поддержкой элементарных операций вычисления длины, конкатенации и т. д. Конечно, для этого нужно определить новый класс с функциями, выполняющими операции со строками, но если вы хотите, чтобы класс работал как полноценный тип данных, его интерфейс не может ограничиваться одними строковыми операциями. В соответствии с общепринятой схемой его интерфейс должен быть создан по образцу, представленному в листинге 3.1. Предполагается, что класс String будет предоставлять абстракцию данных, скрывающую второстепенные детали представления строк в C (то есть char*). Строка C «скрывается» в закрытых данных класса; только операторы класса и его функции работают с ней напрямую.

Листинг 3.1. Класс String, в котором используется идиома ортодоксальной канонической формы

```
class String {
public:
    // Открытый интерфейс класса String:

    // Переопределение "+" для обозначения конкатенации.
    // два случая:
    friend String operator+(const char*, const String&);
    String operator+(const String&) const;
    int length() const;       // Длина строки в символах
    // ...                    // Другие интересные операции

    // Стереотипные функции класса:

    String();                 // Конструктор по умолчанию
```

продолжение

Листинг 3.1 (*продолжение*)

```
    String(const String&);   // Конструктор для инициализации новой
                             // строки на основании существующей
    String& operator=(const String&); // Присваивание
    ~String();               // Деструктор

    /*
     * Эти операции характерны для поведения, обычно определяемого
     * пользователем для типа. Они хорошо подходят для класса String.
     */

    String(const char *);    // Инициализация по "строке С"
private:
    char *rep;               // Данные реализации и внутренние
                             // функции (здесь хранится
                             // внутренняя строка в формате С).

};
```

Давайте последовательно рассмотрим все функции класса и разберемся, как они должны быть реализованы.

`String()`

Конструктор, инициализирующий объекты String значениями по умолчанию при отсутствии контекста инициализации; обычно называется *конструктором по умолчанию*. Для класса String по умолчанию логичнее всего создавать пустую строку (если бы это был класс комплексного числа ComplexNumber, конструктор по умолчанию обнулял бы число, и т. д.). Реализация конструктора выглядит достаточно прямолинейно:

```
String::String() {
    rep = new char[1];
    rep[0] = '\0';
}
```

Конструктор по умолчанию автоматически вызывается компилятором при инициализации элементов вектора объектов. Например, он инициализирует каждый элемент следующего вектора:

```
String stringVec[10];    // Каждый элемент инициализируется
                         // конструктором String::String()
```

Если пользователь не определил для класса ни одного конструктора, то компилятор сгенерирует конструктор по умолчанию за программиста. Этот конструктор будет вызывать конструкторы по умолчанию для всех членов класса, которые сами по себе являются объектами (в классе String такие члены отсутствуют). Но учтите: если конструктор по умолчанию генерируется компилятором, то члены класса, не имеющие конструктора по умолчанию, останутся неинициализированными (см. раздел 2.3).

`String(const String&)`

Конструктор создает точную копию объекта String, созданного ранее. В параметре конструктора передается *ссылка* на String; в данном случае это означает, что функция получает исходный объект, с которым она работает (вместо указателя на объект или его копии).

```
String::String(const String& s) {
    // Зарезервировать место для '\0'
    rep = new char[s.length() + 1];
    // Копирование данных из старой строки в новую
    ::strcpy(rep, s.rep);
}
```

Запись ::strcpy обозначает вызов функции с заданным именем, принадлежащей внешней, то есть глобальной области видимости. В общем случае этот конструктор должен скопировать *все* данные существующего объекта в новый объект. Мы будем называть его *копирующим конструктором*, подразумевая, что он инициализирует новый объект копированием данных из объекта-прототипа. Возможно, термин выбран не совсем удачно, потому что копирующий конструктор часто ограничивается *логическим*, или *поверхностным*, копированием объекта (под поверхностным копированием понимается копирование объекта без физического копирования тех объектов, ссылки на которые содержатся в копируемом объекте, тогда как при глубоком копировании происходит рекурсивное копирование объектов, на которые ссылается копируемый объект). Копирующий конструктор не обязан выполнять *физическое* копирование; копия может быть логической, но с точки зрения внешнего пользователя она должна вести себя как физическая копия. Отказ от физического копирования используется в различных стратегиях управления памятью, описанных далее в этой главе. Наверное, правильнее было бы сказать, что этот конструктор создает «логические клоны» существующих объектов, но термин «копирующий конструктор» уже прижился.

Вызов этого конструктора генерируется компилятором C++ при передаче объекта String по значению в параметре функции, а иногда — и при возврате по значению.

`String& operator=(const String&)`

Оператор присваивания во многом похож на копирующий конструктор, хотя существуют и различия. Во-первых, он имеет возвращаемое значение, тогда как у конструктора его нет. Когда функция класса не является конструктором, она должна предоставить некоторое возвращаемое значение (если это значение обрабатывается вызывающей стороной). Оказывается, самым подходящим типом возвращаемого значения для оператора присваивания является *ссылка* на текущий тип, поскольку она предотвращает лишние затраты на создание промежуточного (временного) объекта (оператор присваивания рассматривается в приложении Г).

Во-вторых, раз содержимое объекта замещается новым содержимым (в этом и состоит суть операции присваивания), функция отвечает за уничтожение старого содержимого. В процессе создания нового значения оператор присваивания должен освободить всю память, ссылки на которую перестают существовать. Использование канонической формы *гарантирует*, что конструктор будет вызван перед оператором operator= (при условии, что все конструкторы выполнили свою работу и инициализировали rep). Следовательно, старое содержимое rep *заведомо* может быть уничтожено в процессе присваивания:

```
String& String::operator=(const String& s) {
    if (rep != s.rep) {  // Специальная проверка
                         // присваивания a=a
        // Уничтожение старого содержимого rep:
        // квадратные скобки сообщают компилятору,
        // что удаляется не отдельная величина,
        // а вектор.
        delete[] rep;

        // Как в String::String(const String&)
        int lengthOfOriginal = s.length() + 1;
        rep = new char[lengthOfOriginal];
        ::strcpy(rep, s.rep);
    }

    // Также нужно вернуть результат, что не было
    // сделано в конструкторе. Результат представляет
    // собой ссылку на объект.
    return *this;
}
```

Внутренняя переменная this используется компилятором для ссылок на «текущий» объект. Компилятор автоматически определяет эту переменную, и ее не нужно (да и нельзя) объявлять. Переменная this отличается от других переменных: ее имя принадлежит к числу ключевых слов C++ и не может использоваться как имя обычной переменной. Типом this является константный указатель на объект класса, содержащего текущую функцию. Никогда не пытайтесь присваивать переменной this новое значение.

Переопределение семантики присваивания является формой *перегрузки*: для класса String оператору присваивания назначается новый смысл, отличный от его смысла в других классах. По умолчанию в ходе выполнения операции присваивания для объектов классов последовательно замещаются значения переменных объекта в правой части соответствующими переменными объекта, указанного в левой части. Фактически мы «перехватываем» контроль над оператором присваивания у компилятора и определяем для него новый смысл. В разных контекстах оператору присваивания может назначаться разный смысл. За сохранение интуитивно понятного представления о смысле присваивания отвечает только сам программист; если вам захочется пойти

наперекор здравому смыслу, ни язык, ни компилятор не смогут вам в этом помешать!

Если вас интересует, чем плохо поразрядное копирование и почему даже попарное копирование членов (как в текущей версии C++, то есть версии 2.0 и выше) не всегда работает без помощи программиста, обращайтесь к приложению Г.

Объявление operator= требует вернуть ссылку на тип, в котором этот оператор определяется. Это означает, что выражение a=b может использоваться в левой части другой операции присваивания. Такие выражения называются *левосторонними значениями*, или *l-значениями*. Возврат ссылки создает нежелательный побочный эффект — становятся возможными выражения вида:

```
String a, b, c;
(a = b) = c;
```

Семантика подобных команд обычно неочевидна, и лучше защититься от ее применения. Альтернативная форма оператора присваивания может выглядеть так:

```
const String& String::operator=(const String& s) {
    ...
}
```

Первый модификатор const запрещает дальнейшую модификацию результата полученного выражения другими функциями класса, вследствие чего левостороннее присваивание становится недопустимым. У такого решения есть и оборотная сторона: недействительными также становятся потенциально полезные конструкции вида:

```
(a = b).put('c');
```

```
~String()
```

Деструктор должен освободить все ресурсы, захваченные объектом в процессе выполнения конструкторов или любых функций класса. Другие функции класса String могут сохранять информацию о других динамически выделенных ресурсах в переменных String, чтобы деструктор мог найти эти ресурсы и освободить их. В нашем примере используется только один динамически выделенный объект — строка C, на которую указывает rep. И снова благодаря канонической форме мы знаем, что деструктор будет вызван лишь после вызова конструктора, поэтому указатель rep заведомо ссылается на строку в блоке памяти, выделенном в динамическом пуле. Код деструктора предельно прост:

```
String::~String() {
    delete[] rep;
}
```

Помните, что деструкторы всегда вызываются без аргументов. Класс не может содержать более одного деструктора.

`String (const char*)`

Конструктор создает объект String на основании строки С. Такой способ конструирования является специфическим для типа String, но во многих типах определяются собственные нестандартные конструкторы для инициализации экземпляров по некоторому контексту, предоставленному программистом. Конструкторы могут иметь несколько аргументов. Класс может содержать несколько конструкторов с разными комбинациями аргументов; правильная комбинация автоматически выбирается компилятором на основании контекста. В нашем примере код этого конструктора прост:

```
String::String(const char *s) {
    int lengthOfOriginal = ::strlen(s) + 1;
    rep = new char[lengthOfOriginal];
    ::strcpy(rep, s);
}
```

Строго говоря, подобные специализированные конструкторы не являются частью ортодоксальной канонической формы; это обычный пользовательский конструктор, семантика которого соответствует приложению. Помимо инициализации при конструировании объекта String, он также является оператором преобразования, определяющим способ построения объекта String при передаче указателя на символьные данные вместо объекта String:

```
extern int hash(String);
...
hash("character literal");
    // Автоматически вызывается String::String(const char*)
    // (и вероятно, также String::String(const String&).
```

`length()`

Функция length также специфична для класса String, и она тоже легко программируется:

```
int String::length() const {
    return ::strlen( rep );
}
```

Формально функция length не является частью ортодоксальной канонической формы. Это обычная функция класса, семантика которой определяется спецификой приложения.

Приведенный пример класса String иллюстрирует идиому, то есть образец, которым надлежит руководствоваться при реализации любого класса. Для произвольного класса X эта форма характеризуется присутствием следующих элементов:

- конструктора по умолчанию (X::X());
- копирующего конструктора (X::X(const X&));
- оператора присваивания (X& operator=(const X&));
- деструктора (X::~X()).

ПРИМЕЧАНИЕ

В общем случае ортодоксальную каноническую форму НУЖНО использовать, если, во-первых, требуется обеспечить поддержку присваивания для объектов класса или передачу их по значению в параметрах функций, во-вторых, объект содержит указатели на объекты, для которых используется подсчет ссылок, или же деструктор класса вызывает оператор delete для переменной объекта.

Ортодоксальную каноническую форму МОЖНО использовать для всех нетривиальных классов, поскольку она обеспечивает единый стиль оформления классов и помогает справиться с растущей сложностью классов в процессе развития программы.

Ортодоксальная каноническая форма заложена в основу других идиом, приводимых в книге. Используйте эти идиомы, если это соответствует специфике вашего приложения.

Указатели широко используются в объектно-ориентированном программировании на C++, и большинство классов удовлетворяет перечисленным критериям. Единства ради, большинство нетривиальных классов следует оформлять в ортодоксальной канонической форме. «Тривиальным» классом в данном контексте считается класс, используемый как структура C, то есть ограничивающийся агрегированием данных на функциональном уровне (хотя он также может содержать вспомогательные функции для удобства записи). Если же функции класса требуются для управления доступом к данным класса, среди которых встречаются указатели, лучше применять ортодоксальную каноническую форму.

Некоторые отклонения от канонической формы позволяют изменять стандартные аспекты поведения классов. Так, объявляя конструктор по умолчанию закрытым, вы запрещаете создание экземпляров класса без передачи параметра, по которому можно выбрать конкретный конструктор. Для некоторых классов копирование не имеет смысла. Например, объект, переменные которого представляют микропроцессорные регистры ввода-вывода, отображаемые на память, не должен копироваться и не может перемещаться в памяти. Возможно, копирование объектов Window графической системы тоже не имеет смысла. Если бы копирование объектов Window было допустимо, то эти объекты могли бы передаваться функциям по значению, и при любом вызове функции с таким параметром на экране появлялось бы новое окно! Объявление оператора присваивания и копирующего конструктора подобных классов закрытыми предотвращает копирование их объектов.

Одну из категорий классов, часто применяемых в объектно-ориентированных архитектурах, составляют *контейнерные классы* — типы объектов, предназначенных для хранения наборов других объектов. К их числу относятся множества, списки, очереди, словари, мультимножества и стеки. Контейнерные классы должны использовать ортодоксальную каноническую форму или ее разновидность.

Ортодоксальная каноническая форма лежит в основе большинства конкретных типов данных. Другие канонические формы, описываемые далее, также строятся на ее основе.

3.2. Видимость и управление доступом

Класс может управлять доступностью его членов со стороны функций, не входящих в иерархию этого класса. Мы будем использовать термин «горизонтальное управление доступом» для обозначения управления доступом со стороны одноранговых классов и соседей. С другой стороны, термин «вертикальное управление доступом» определяет права доступа со стороны базовых и производных классов. Правильное горизонтальное управление доступом обеспечивает инкапсуляцию, а также гарантирует логическую согласованность и семантическую разумность использования класса его клиентами. Кроме того, оно помогает обеспечить абстракцию, столь важную для успешного применения пользовательского типа в сложной системе. В следующих главах книги, особенно в главах 5 и 6, мы подробно рассмотрим связь между управлением доступом, наследованием и объектно-ориентированным программированием.

Обратите внимание на ключевые слова private и public внутри класса String (см. листинг 3.1). Все члены класса, следующие за меткой public (вплоть до следующей метки), напрямую доступны для всех функций C++. Если некоторая функция располагает указателем на тип String, она сможет использовать его для вызова любых функций или обращения к любым переменным, находящимся в секции public.

Закрытые члены класса (те, что перечисляются после метки private), доступны только для функций этого класса. Прямое обращение к ним из-за пределов класса невозможно. У этого правила существует единственное исключение — механизм дружественных отношений, позволяющий другим функциям нарушать инкапсуляцию класса. Следующее объявление разрешает всем функциям класса George доступ к закрытым и защищенным членам того класса, в котором оно находится:

```
friend class George;
```

Например, такой фрагмент указывает, что класс George является «другом» с точки зрения класса Sue, хотя оно ничего не говорит о том, как Sue рассматривается с точки зрения George:

```
class Sue {
    friend class George;
public:
    ...
};
```

Классу George разрешено обходить инкапсуляцию Sue. Следующее объявление работает более избирательно:

```
friend Sally::peek(int);
```

То есть доступ к закрытым членам текущего класса предоставляется только перечисленным функциям класса Sally.

Закрытые функции класса часто используются для маскировки алгоритмов, задействованных во внутренней реализации класса и не входящих во внешний интерфейс.

Функции класса могут обращаться ко всем членам своего класса, закрытым и открытым. Защищенные члены базового класса доступны для членов производного класса только в том случае, если они образуют часть объекта производного класса и доступ к ним производится именно в этом контексте.

По умолчанию структура ведет себя как класс, все члены которого являются открытыми:

Код C:

```
struct Stack {
    ...
};
```

Код C++:

```
class Stack {
public:
    ..
};
```

Впрочем, для членов структуры также может устанавливаться защита:

Код структуры:

```
struct Stack {
    void push(int);
    int pop();
    int top() const;
private:
    int rep[100];
};
```

Код класса:

```
class Stack {
public:
    void push(int);
    int pop();
    int top() const;
private:
    int rep[100];
};
```

Хотя закрытые данные доступны для функций «своего» класса как для чтения, так и для записи, класс может ограничить доступ к закрытым данным со стороны внешних пользователей и разрешить им только чтение. Допустим, мы хотим сделать представление класса String доступным для использования в контексте языка C (листинг 3.2). Такой подход более эффективен, чем создание копии внутренних данных String для применения за пределами класса. С другой стороны, возможность записи сопряжена с большим риском, поскольку у внешнего пользователя появляется «черный ход» для модификации внутреннего состояния String.

Листинг 3.2. Предоставление доступа к внутренним данным класса

```
#include <stdlib.h>

class String {
public:
    String();               // Конструктор по умолчанию
    String(const String&); // Копирующий конструктор
    String& operator=(const String&); // Оператор присваивания
    ~String();              // Деструктор
    String(const char*);    // Создание объекта String
                            // на базе "строки C"
```

продолжение

Листинг 3.2 (*продолжение*)

```
    String operator+(const String&) const;
                          // Переопределение "+"
                          // для обозначения конкатенации.
    int length() const;   // Длина строки в символах
    const char *const C_rep() {
        return (const char *const)rep;
    }
private:
    char *rep;
};

int main() {
    String s;
    // ...
    printf("string is %s\n", s.C_rep());
}
```

Ниже приведен альтернативный вариант синтаксиса с применением ссылок:

```
class String {
public:
    ...
    String(const char *s): C_rep(rep){
        rep = new char[::strlen(s) + 1];
        ::strcpy(rep, s);
    }
    const char* &C_rep();
    ...
private:
    char *rep;
};
```

3.3. Перегрузка — переопределение семантики операторов и функций

Как было показано ранее, программист может взять на себя контроль над реализацией присваивания с сохранением его высокоуровневой семантики. То же самое можно сделать с большинством операторов C. Переопределение смысла оператора, называемое *перегрузкой оператора*, играет важную роль в работе пользовательских типов; мы определяем типы в контексте операций, выполняемых с ними. Возьмем операторы +, –, * и / языка C и поддерживаемые ими числовые типы. Конечно, для удобства и наглядности нам бы хотелось определить смысл этих операторов для операндов пользовательских классов Complex, Vector и Matrix. Одним из преимуществ C++ перед C является прозрачность пользовательских типов (таких, как Complex), которые могут применяться точно так

же, как встроенные объекты int или double. В языке C даже директива typedef на самом деле не создает нового типа, потому что она не влияет на работу операторов +, – и т. д. В C++ это становится возможным благодаря как конкретным типам данных, описанным в этой главе, так и перегрузке операторов.

У перегрузки существуют и другие потенциальные применения — пусть не столь очевидные, как для математических типов, но все равно полезные. Например, в наш классе String включен оператор + с семантикой конкатенации строк. Следовательно, код в левой части следующего листинга эквивалентен коду в правой части (при объявлениях в листинге 3.2 допустимы оба варианта):

```
int main() {
    String a, b("this is b");
    String c("!");
    a = b + c;
}
```

```
int main() {
    String a, b("this is b");
    String c("!");
    a = b.operator+(c);
}
```

Так как мы можем назначить оператору + любой смысл по своему усмотрению, для объектов String его можно определить для выполнения конкатенации:

```
String String::operator+(const String& s) const
{
    char *buf = new char[s.length() + length() + 1];
    ::strcpy(buf, rep);
    ::strcat(buf, s.rep);
    String retval( buf );
    delete[] buf;           // Освобождение временной памяти
    return retval;
}
```

Обратите внимание на то, как функция создает новый объект String «на ходу». Она вызывает String::String(const char*), чтобы новый объект String был создан на базе buf, после чего этот объект становится локальной переменной функции operator+. При возврате управления из operator+ вызывающей стороне передается *копия* этого объекта, созданная вызовом копирующего конструктора String::String(const String&).

В использовании перегрузки нужно знать меру. Скажем, перегрузка оператора [] для класса, который является аналогом вектора (см. далее), выглядит вполне логично, а перегрузка оператора = необходима для канонической формы. С другими применениями (такими как перегрузка математических операторов с нематематической семантикой) все не так однозначно. Они помогают проектировщику программы лучше выразить свои намерения, но с другой стороны, могут сбить с толку случайного читателя программы. При идиоматическом введении этих операторов в некоторый контекст (например, для строк оператор + обычно означает конкатенацию) их ограниченное применение приносит пользу и делает синтаксис более гибким. Однако некоторые программисты склонны злоупотреблять перегрузкой; подобные примеры можно найти в упражнениях в конце главы.

Пример перегрузки оператора индексирования

Перегружать можно даже такие внешне неожиданные операторы, как оператор индексирования []. Предположим, вы хотите использовать синтаксис индексирования для выборки отдельных символов из объекта String. Пусть объявление оператора в объявлении класса выглядит так:

```
class String {
public:
    ...
    char& operator[](int);
    ...
};
```

После объявления оператора программируется сама функция:

```
char& String::operator[](int index)
{
    return rep[index];
}
```

При желании в функции можно организовать проверку границ и убедиться в том, что переменная index больше нуля, но меньше length(). Если индекс имеет недопустимое значение, функция может выводить сообщение об ошибке и/или возвращать символ EOF. Такие конструкции позволяют включить код для отладки приложений, использующих объекты типа String, без модификации самого приложения. После того как программа будет отлажена, диагностический код удаляется из нее; еще лучше оставить его на месте и исключить его из компиляции директивами #ifdef на случай, если он понадобится в будущем.

Возвращаемое значение функции operator[] объявляется как *ссылка* на символ, что позволяет использовать его в левой части оператора присваивания. За дополнительной информацией о ссылках обращайтесь к приложению В.

Оператор [] не ограничивается целочисленными аргументами; он может получать аргументы разных типов или многократно перегружаться для нескольких разных типов. При наличии нескольких объявлений компилятор по контексту вызова выбирает нужную версию. Например, оператор индексирования можно перегрузить для типа const char*:

```
тип operator[](const char *) // Ассоциативный массив
```

Такой оператор может использоваться как функция контейнерного класса (скажем, списка или мультимножества) для выборки объекта по имени. Возможны и другие применения, скажем, этот оператор можно включить в класс String для поиска вхождений заданных подстрок.

Возьмем другой, более содержательный пример — перегрузку оператора [] с выбором реализации в зависимости от контекста: в качестве l-значения (то есть в левой части оператора присваивания) и при выборке значения с заданным индексом. В основу решения заложена небольшая хитрость: наша версия оператора [] создает временное возвращаемое значение фиктивного типа. При создании фик-

тивного объекта или его появлении в определенных контекстах (например, в левой части оператора присваивания или в контексте, в котором ожидается другой тип) управление передается его перегруженным операторам. Для примера возьмем абстракцию File из листинга 3.3. Если объект File индексируется в контексте l-значения, то заданный байт файла заменяется новым значением, а если нет — заданный байт читается из файла и возвращается как результат выражения.

Листинг 3.3. Перегрузка оператора [] с учетом контекста

```
class FileRef {
private:
    class File &f;
    char buf[1];
    unsigned long ix;
public:
    FileRef (File &ff, unsigned long i) : f (ff), ix (i) { }
    FileRef &operator=(char c);
    operator char ();
};

class File {
friend class FileRef;
public:
    File(const char *name) {
        fd = open(name, O_RDWR|O_CREAT, 0664);
    }
    ~File() { close(fd); }
    FileRef operator[] (unsigned long ix) {
        return FileRef(*this, ix);
    }
private:
    int fd;
};

FileRef& FileRef::operator=(char c) {
    lseek(f.fd, ix, 0); write(f.fd, &c, 1); return *this;
}

FileRef::operator char () {
    lseek(f.fd, ix, 0); read(f.fd, buf, 1); return buf[0];
}

int main() {
    File foo("foo");
    foo[5] = '5';
    foo[10] = '1';
    char c = foo[5];
    cout << "c = " << c << endl;
}
```

В результате вызова функции operator[] для объекта File возвращается результат «фиктивного» типа FileRef. Этот объект либо используется для присваивания в контексте l-значения, либо его значение задействуется напрямую. Вызов File::operator[] в контексте l-значения дает объект FileRef. Итак, если в левой части находится объект FileRef, а в правой — символ, вызывается FileRef::operator=, в котором и выполняется запись. Если объект File индексируется в любом контексте, предполагающем символьный результат, то оператор индексирования снова создает промежуточный объект FileRef, который затем преобразуется в char вызовом FileRef::operator char. Код чтения данных из файла помещается в оператор присваивания. Таким образом, если результат индексирования используется как приемник при присваивании, происходит запись, а если используется его значение, происходит чтение из файла.

Перегрузка операторов в классах и глобальная перегрузка

В C++ существуют две разновидности перегрузки: *перегрузка в классе*, продемонстрированная в предыдущем примере, и *глобальная перегрузка*, осуществляемая во внешней области видимости. Допустим, переменные a и b объявлены как объекты класса C. В классе C определен оператор C::operator+(C), поэтому этот пример является частным случаем перегрузки в классах, как в предыдущем примере, а a+b означает a.operator+(b). Также возможна *глобальная* перегрузка оператора +:

```
C operator+(C,C) {...}
```

Она тоже применима к выражению a+b, где a и b передаются соответственно в первом и втором параметрах функции. Из этих двух форм предпочтительной считается перегрузка в классе. Вторая форма требует применения дружественных отношений, а это отрицательно отражается на строгой эстетике автономного класса. Вторая форма может быть более удобной для адаптации классов, находящихся в библиотеках объектного кода, когда исходный текст невозможно изменить и перекомпилировать. Смешивать эти две формы в программе не рекомендуется. Если для некоторого оператора определены обе формы с одинаковыми типами формальных параметров, то использование оператора может создать двусмысленность, которая, скорее всего, окажется фатальной.

Тем не менее, глобальная перегрузка операторов обеспечивает симметрию, которая также обладает эстетической ценностью. Рассмотрим случай с глобальной перегрузкой функций operator+, дружественных для класса String:

```
class String {
friend String operator+(const char*, const String&);
friend String operator+(const String&, const char*);
public:
    ...
private:
    char *rep;
};

String operator+(const char* s, const String& S) {
```

```
    String retval;
    retval.rep = new char[::strlen(s) + S.length()];
    ::strcpy(retval.rep, s);
    ::strcpy(retval.rep, S.rep);
    return retval;
}

String operator+(const String& S, const char *s) {
    String retval;
    retval.rep = new char[::strlen(s) + S.length()];
    ::strcpy(retval.rep, S.rep);
    ::strcpy(retval.rep, s);
    return retval;
}
```

При использовании этих объявлений будут работать обе следующие операции «суммирования»:

```
String s1;
"abcd" + s1;
s1 + "efgh";
```

Без глобальной перегрузки эта задача не решается. Поскольку мы не можем получить доступ к «родному» строковому типу C (то есть к типу char*) и переопределить его операции, обеспечить симметрию простым определением операторов класса не удастся; потребуется решение с глобальными дружественными функциями. Если два класса (в отличие от примитивных типов вроде int, char и т. д.) по взаимному согласию должны получить доступ к закрытым членам друг друга, дружественные отношения можно установить между этими классами; глобальная дружественная функция обычно не нужна. Также может потребоваться дружественная функция преобразования для упрощения преобразований между двумя классами, если один из классов находится в библиотеке, для которой отсутствует исходный текст, а имеются только объявления интерфейса класса. Допустим, вы захотели изменить класс PathName так, чтобы оператор + мог использоваться для присоединения расширения в формате String. Если класс PathName находится в библиотеке, для которой отсутствуют исходные тексты, приходится задействовать глобальную перегрузку:

```
PathName operator+(const PathName &p, const String &s) {
    ...
}
```

3.4. Преобразование типа

При помощи дополнительных конструкторов и определений операторов можно предоставить компилятору информацию, достаточную для автоматического преобразования типов в зависимости от контекста. Предположим, нужно, чтобы в программе работала следующая конструкция:

```
#include <String.h>

extern "C" int strlen (const char *);

int main() {
    String s("1234");

    int j = ::strlen(s);      // То же, что j = s.length();
    ...
}
```

Чтобы такая конструкция работала, мы должны научить компилятор создавать «примитивные» типы на основании нашего типа. Задача решается при помощи особых спецификаций операторов. В данном примере компилятор должен автоматически преобразовывать String в char*, поэтому мы определяем соответствующий оператор в классе String:

```
class String {
public:
    operator const char*() const { return rep; }
    ...
};
```

Первый модификатор const означает, что память, на которую ссылается возвращаемое значение, не должна модифицироваться кодом за пределами класса, а второй — что эта функция не изменяет содержимого объекта String, для которого она вызывается.

Если теперь компилятор встречает объект String в любом контексте, в котором может использоваться тип const char*, он автоматически вызывает String::operator const char* для выполнения преобразования. Оператор просто возвращает указатель на внутренние данные, уже находящиеся в формате языка C. Если же потребуются дополнительные преобразования, весь служебный код следует разместить в теле оператора. Это иллюстрирует идиому *операторной функции класса.*

Возьмем другой пример: если объект String находится там, где должно находиться целое число, оператор преобразования интерпретирует его содержимое как представление целого числа в формате ASCII и возвращает результат. Возможная реализация может выглядеть так:

```
class String {
public:
    ...
    operator int();
    ...
};

String::operator int()
{
```

```
    int retval;
    if (sscanf(rep, "%d", &retval) != 1) return 0;
    else return retval;
}
```

Чтобы функция преобразования была вызвана в программе, объект String должен находиться в контексте, в котором он может интерпретироваться как целое число. Это может быть сделано как неявно (например, при передаче String в списке фактических параметров функции на том месте, где должен находиться параметр int), так и явным преобразованием типа:

```
#include "String.h"

int main()
{
    char buf[10];
    printf("enter number of bytes: ");
    scanf("%s\n", buf);
    String sbuf = buf;

    // Явное преобразование в int:
    printf("read in value %d\n", int(sbuf));

    char *thing = new char[100];
    // Явное преобразование к типу в const char *const:
    ::strcpy(thing, (const char *const)sbuf);

    // Неявное преобразование в int:
    int charsLeftInThing = 100 - sbuf;
    ...
    return 0;
}
```

ПРИМЕЧАНИЕ

Используйте эту идиому только для преобразования к встроенным типам C. Если потребуется преобразовать String в другой класс — скажем, в ParseTree, то преобразование должно выполняться классом ParseTree, а не классом String.

Для преобразования объектов String в ParseTree следует по возможности включить в класс ParseTree конструктор ParseTree(const String&). Класс String не может располагать информацией обо всех возможных классах; пусть об этом беспокоятся классы, которым *нужна* поддержка String! Чтобы класс ParseTree обладал свободным доступом к внутренним данным String (для повышения эффективности преобразования), возможно, вам между двумя классами придется установить дружественные отношения. Для простых классов вроде String эффективность преобразования не создает проблем, но для более сложных структур и типов данных она может быть весьма существенной.

Включение операторов преобразования в класс должно быть хорошо продумано с позиций проектирования. Слишком частое использование операторов преобразования повышает риск неоднозначных преобразований.

Впрочем, у любого общего правила могут быть свои исключения; делайте то, что имеет смысл. Давайте вернемся к примеру с преобразованием String в ParseTree. Если класс ParseTree входит в библиотеку, для которой доступны только объектный код и заголовочный файл с объявлением класса, внутренняя модификация класса ParseTree оказывается затрудненной или невозможной. Во-первых, имеющейся информации о внутреннем строении ParseTree может быть недостаточно для выражения преобразования в контексте закрытых членов класса, присутствующих в объявлении. А это наводит на мысль, что преобразование должно проводиться в контексте открытого интерфейса класса, чтобы оно с таким же успехом могло выполняться за пределами класса. Во-вторых, изменение интерфейса ParseTree без повторного построения кода библиотеки может привести к несовместимости «старого» и «нового» кода ParseTree (сгенерированного для двух разных интерфейсов). Наконец, внесение изменений за пределами класса в контексте открытого интерфейса упрощает последующий переход на новые версии библиотеки без модификации кода приложения. Проблема решается изменением класса String:

```
#include "ParseTree.h"

class String {
public:
    ...
    operator ParseTree() {...}
    ...
};
```

Мы уже встречались с примерами преобразования существующих типов в String; примером служит функция String::String(const char*), которая строит объект String на базе объекта const char*. Определяя дополнительные конструкторы, можно преобразовать в String любой другой тип. Возьмем класс Node, представляющий элемент бинарного дерева символьных строк:

```
class Node {
public:
    Node();
    Node(const Node&);
    Node& operator=(const Node&);
    Node *leftChild() const;  // Не изменяет объект
    Node *rightChild() const; // Не изменяет объект
    char *contents() const;   // Возвращает содержимое узла;
                              // объявляется с ключевым словом
                              // const, так как не изменяет объект.
private:
    ...
};
```

Конструктор, приведенный в листинге 3.4, сворачивает поддерево узла **Node** в форму с круглыми скобками. Также можно считать, что этот конструктор умеет строить **String** из **Node**. Для перебора узлов дерева в нем используется вспомогательная функция **nodeWalk**.

Листинг 3.4. Конструктор для преобразования Node в String

```
#include "Node.h"

class String {
public:
    ...
    String (Node);
    ...
private:
    ...
    String nodeWalk(const Node*);
    ...
};

String
String::nodeWalk(const Node *n) { // Инфиксный перебор узлов
    if (n == 0) return "";
    String retval = String("(") + nodeWalk(n->leftChild());
    retval = retval + " " + n->contents();
    retval = retval + " " + nodeWalk(n->rightChild()) + ")";
}

String::String(Node n)
{
    String temp = nodeWalk(&n);
    p = new char[temp.length() + 1];
    ::strcpy(p, temp.p);
}
```

3.5. Подсчет ссылок

В объектно-ориентированных программах, в отличие от процедурных, переменные чаще связываются с объектами, созданными в динамической куче, вместо статически объявленных объектов. Главным образом это связано с тем, что время жизни экземпляров объектов и их данных играет в объектной парадигме примерно такую же роль, как кадр стека в процедурном программировании. Большая часть данных существует в виде объектов, моменты создания и уничтожения которых должны соответствовать жизненному циклу моделируемых сущностей реального мира. С этой точки зрения повышается важность операций выделения и освобождения памяти. Так как программист может управлять тем, что происходит с объектом при его создании и уничтожении, он может оптимизировать эти операции для повышения быстродействия и эффективности использования памяти отдельными классами.

Среди механизмов такого рода наиболее распространены механизмы *общего представления* и *подсчета ссылок*. Эта идиома обобщается для любых классов C++, использующих копирование экземпляров, и оказывается особенно полезной для классов с динамическим выделением памяти. Возьмем класс String из предыдущего раздела: присваивание приводит к дублированию всей строки и фактическому удвоению затрат памяти только для копирования и хранения избыточной информации. Но класс можно наделить «интеллектом», чтобы вместо создания копий несколько переменных совместно использовали общую внутреннюю память. Здесь само представление является объектом, который управляет общими данными и ведет подсчет ссылок. Существует несколько способов реализации этой идиомы; мы рассмотрим три варианта. Первый имеет общий характер и адаптируется для большинства абстракций. Второй вариант показывает, как избежать неэффективного копирования памяти, присущего первому варианту. Последний вариант позволяет автоматизировать часть «черновой работы» по сопровождению таких абстракций. Кроме того, вы узнаете, как организовать подсчет ссылок в существующих классах.

Идиома класса-манипулятора

В первом варианте исходный класс String (см. листинг 3.1) инкапсулируется в классе, управляющем памятью. Мы будем называть этот класс *манипулятором*, поскольку он манипулирует другим классом, непосредственно решающим задачи приложения.

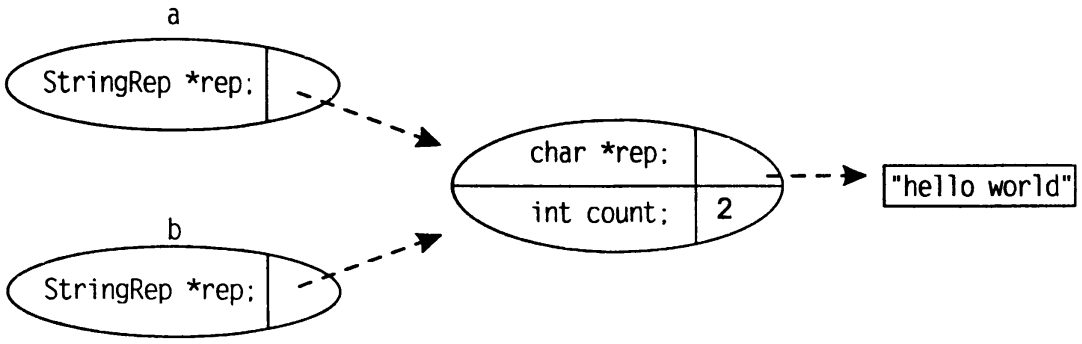


Рис. 3.1. Подсчет строк для класса String

Пользователю кажется, что все делает манипулятор, но на самом деле настоящую работу выполняет внутренний класс. Итак, мы должны сделать следующее.

1. Переименовать класс String в StringRep.
2. Включить в StringRep новое поле для подсчета ссылок.
3. Создать класс String, содержащий указатель на StringRep.
4. Перенаправить большинство операций String классу StringRep, обеспечив совместное использование представлений при выполнении присваивания.

На рис. 3.1 показана структура нового класса String в памяти. Объект String с именем a инициализируется константой "hello world", после чего значение a присваивается объекту b. Хотя переменным a и b сопоставлены имена разных объектов String, обе переменные ссылаются на общее представление, содержащее большую часть низкоуровневых данных и реализации строковых операций. В представлении хранится *счетчик ссылок* — целое число, которое указывает, сколько объектов String содержит указатель на эти данные. Когда счетчик ссылок уменьшается до нуля, объект можно уничтожать.

В листинге 3.5 приведено объявление нового класса StringRep. Его функции соответствуют функциям String, но они объявлены подставляемыми (inline) и стали более компактными. (Функции объявляются подставляемыми только в том случае, если это решение оправдано анализом быстродействия. В некоторых приложениях увеличение объема сгенерированного кода может оказаться неприемлемым.)

Листинг 3.5. Класс представления String

```
class StringRep {
friend String;
private:     // Теперь эти члены доступны только для String:
    StringRep()           { *(rep = new char[1]) = '\0'; }
    StringRep(const StringRep& s)  {
      ::strcpy(rep=new char[::strlen(s.rep)+1], s.rep);
    }
    StringRep& operator=(const StringRep& s) {
       if (rep != s.rep) {
         delete[] rep;
         ::strcpy(rep=new char[::strlen(s.rep)+1], s.rep);
       }
       return *this;
    }
    ~StringRep()          { delete[] rep; }
    StringRep(const char *s) {
       ::strcpy(rep=new char[::strlen(s)+1], s);
    }
    StringRep operator+(const StringRep&) const;
    int length() const    { return ::strlen(rep); }
private:
    char *rep;
    int count;
};

StringRep StringRep::operator+(const StringRep& s) const
{
    char *buf = new char[s.length() + length() + 1];
    ::strcpy(buf, rep);
    ::strcat(buf, s.rep);
    StringRep retval( buf );
    delete[] buf;         // Освобождение временной памяти
    return retval;
}
```

Также нам понадобится новый класс String, который использует StringRep для выполнения всех строковых операций, а сам выполняет основные операции управления памятью; этот класс приведен в листинге 3.6.

Листинг 3.6. Интерфейс String к классу StringRep

```
class String {
public:
    String() {
        rep = new StringRep; rep->count=1;
    }
    String(const String& s) {
        rep = s.rep; rep->count++;
    }
    String& operator=(const String& s) {
       s.rep->count++;
       if (--rep->count <= 0) delete rep;
       rep = s.rep;  return *this;
    }
    ~String()           {
        if (--rep->count <= 0) delete rep;
    }
    String(const char *s) {
       rep = new StringRep(s);
       rep->count = 1;
    }
    String operator+(const String& s) const {
       StringRep y = *rep + *s.rep;
       return String(y.rep);
    }
    int length() const {
        return rep->length();
    }
private:
    StringRep *rep;
};
```

Обратите внимание: «низкоуровневые» конструкторы (строящие объект String «на пустом месте», а не на базе другого объекта String) присваивают полю count класса представления значение 1. Операции, копирующие другую строку (String(const String&) и operator=(const String&)), заимствуют представление оригинала и увеличивают его счетчик ссылок. Любые операции, требующие модификации представления, должны уменьшить счетчик ссылок и освободить память, если счетчик достигнет нуля.

Одно из преимуществ такого решения состоит в том, что оно позволяет сосредоточить всю строковую специфику во внутреннем классе, тогда как внешний класс занимается в основном управлением памятью. В частности, это означает, что представленное решение хорошо подходит для реализации подсчета ссылок

в существующих классах. Однако оно сопряжено с серьезными проблемами эффективности; так, ради достижения выразительной простоты функция operator+ выполняет много лишней работы. В этой программе вызов operator+ приводит к трем операциям выделения памяти и созданию лишних временных объектов в стеке. О том, как этого избежать, рассказано в следующем разделе.

Мы будем называть класс String *манипулятором*, а класс StringRep — *телом*, с которым работают при помощи манипулятора. Использование двух (а возможно, и более) классов в ситуации, когда экземпляр одного класса управляет экземплярами другого класса, иллюстрирует идиому *манипулятор/тело*.

ПРИМЕЧАНИЕ

Идиома «манипулятор/тело» часто требуется для разложения сложных абстракций на меньшие классы, с которыми удобнее работать. В этой идиоме также может отражаться совместное использование одного ресурса несколькими классами, управляющими доступом к этому ресурсу. И все же чаще всего она применяется для подсчета ссылок (см. далее).

Самым распространенным частным случаем идиомы «манипулятор/тело» является подсчет ссылок, как в примере класса String. Если класс-тело содержит счетчик ссылок, значение которого изменяется классом-манипулятором, этот частный случай иллюстрирует идиому *подсчета ссылок*.

ПРИМЕЧАНИЕ

Подсчет ссылок требуется для классов, экземпляры которых часто копируются посредством присваивания или передачи параметров, особенно если копирование обходится дорого из-за большого объема или сложности объекта. Используйте эту идиому, если в программе могут существовать несколько логических копий объекта без копирования данных, например, для объекта экранной памяти, который может передаваться между функциями, или общих блоков памяти, совместно используемых несколькими процессорами, когда каждый пользователь должен получить собственную копию объекта для работы с ресурсом.

Экономное решение

Второе решение строится более или менее «на пустом месте» и отличается высокой эффективностью как по скорости работы, так и по затратам ресурсов. Класс String, с которым работает пользователь, строится по образцу из листинга 3.1. В его представление включается новое поле для счетчика ссылок (листинг 3.7), а при обращениях к нему используется дополнительный уровень абстракции. Счетчик и указатель на данные группируются в тривиальном классе с одним простым конструктором и одним простым деструктором.

Листинг 3.7. Более эффективная реализация подсчета ссылок для String

```
class StringRep {
friend String;
private:
    StringRep(const char *s) {
       ::strcpy(rep = new char[::strlen(s)+1], s);
       count = 1;
```

продолжение

Листинг 3.7. (*продолжение*)

```
    }
    ~StringRep() { delete[] rep; }
private:
    char *rep;
    int count;
};

class String {
public:
    String()                        { rep = new StringRep(""); }
    String(const String& s)         { rep = s.rep; rep->count++; }
    String& operator=(const String& s) {
        s.rep->count++;
        if (--rep->count <= 0) delete rep;
        rep = s.rep;
        return *this;
    }
    ~String() {
        if (--rep->count <= 0) delete rep;
    }
    String(const char *s)           { rep = new StringRep(s); }
    String operator+(const String&) const;
    int length() const              { return ::strlen(rep->rep); }
private:
    StringRep *rep;
};

String String::operator+(const String& s) const
{
    char *buf = new char[s.length() + length() + 1];
    ::strcpy(buf, rep->rep);
    ::strcat(buf, s.rep->rep);
    String retval( buf );
    delete[] buf;          // Освобождение временной памяти
    return retval;
}
```

В реализации можно предусмотреть «черный ход» для сохранения копии при вызове operator+(const String&) класса String. Мы создаем новый конструктор, в аргументе которого передается ссылка на указатель на символьные данные. Иначе говоря, указатель передается по ссылке, конструктор «отбирает» те данные, на которые он указывает, и обнуляет оригинал (листинг 3.8).

Листинг 3.8. Оптимизация конкатенации строк

```
#include <string.h>
class StringRep {
friend class String;
private: // Чтобы не загромождать глобальное пространство имен
    typedef char *Char_p;
```

```
    StringRep(StringRep::Char_p* const r) {
        rep = *r; *r = 0; count = 1;
    }
public:
    StringRep(const char *s)
      { ::strcpy(rep = new char[::strlen(s)+1], s); count = 1; }
    ~StringRep() { delete[] rep; }
private:
    char *rep;
    int count;
};

class String {
public:
    String()               { rep = new StringRep(""); }
    String(const String& s) { rep = s.rep; rep->count++; }
    String& operator=(const String& s) {
       s.rep->count++;  if (--rep->count <= 0) delete rep;
       rep = s.rep;
       return *this;
    }
    ~String() { if (--rep->count <= 0) delete rep; }
    String(const char *s)  { rep = new StringRep(s); }
    String operator+(const String&) const;
    int length() const     { return ::strlen(rep->rep); }
private:
    String(StringRep::Char_p* const r) {
        rep = new StringRep(r);  // Вызов нового конструктора
    }
    StringRep *rep;
};

String String::operator+(const String& s) const
{
    char *buf = new char[s.length() + length() + 1];
    ::strcpy(buf, rep->rep);
    ::strcat(buf, s.rep->rep);
    String retval( &buf );  // Вызов нового закрытого конструктора
    return retval;
}
```

В коде функции operator+(const String&) класса String используется специальная возможность, которую предоставляет конструктор StringRep(Char_p* const) класса StringRep, доступный благодаря дружественным отношениям между двумя классами. Две функции вступают в соглашение: String::operator+(const String&) выделяет память для хранения символов, указатель на нее передается конструктору String(const Char_p*), а тот в свою очередь передает его специальному конструктору

StringRep::StringRep(Char_p* const), сохраняющему указатель. Специальный конструктор StringRep обнуляет исходный указатель в функции operator+(buf), чтобы предотвратить возникновение висячих ссылок. Все эти действия позволяют избежать копирования строкового представления при построении объекта retval в String::operator+.

Подсчет указателей

Методика подсчета указателей слегка отличается от двух предыдущих решений. Данная идиома была выработана в результате исследований общих методов уборки мусора в C++, а в ее основу заложено создание указателей на реальные объекты. Каждый раз, когда в программе копируется указатель, вызывается его операторная функция operator= или копирующий конструктор, поэтому количество ссылок на объект отслеживается в одном объекте представления, совместно используемом несколькими указателями. При выходе указателя из области видимости вызывается его деструктор с соответствующим уменьшением счетчика ссылок. Когда счетчик уменьшается до нуля, блок памяти освобождается — либо на месте, либо позднее в ходе уборки мусора. Единственная хитрость заключается в перехвате вызовов оператора -> для объекта посредством перегрузки.

Перегруженный оператор -> отличается от других перегруженных операторов C++. Для всех остальных операторов значение, возвращаемое как результат операции, определяется реализацией. Но для оператора -> возвращаемое значение представляет собой промежуточный результат, к которому затем применяется базовая семантика ->, что и дает окончательный результат. Рассмотрим такой фрагмент:

```
class A {
public:
    B *operator->();
};
...
int main() {
    A a;
    ...a->b...
}
```

Этот фрагмент означает следующее.

1. Для объекта a вызывается функция A::operator->().
2. Возвращаемое значение сохраняется во временной переменной x типа B*.
3. Вычисление x->b дает окончательный результат.

Здесь b может быть заменено любым членом класса B (данными или функцией). Если класс B перегружает оператор ->, то описанный выше алгоритм из трех шагов выполняется заново.

Такой подход не только обладает преимуществами предыдущей методики подсчета ссылок, но и предоставляет дополнительные возможности. Во-первых,

подсчитываемым указателям присущи все достоинства конкретных типов данных: их можно передавать в параметрах, создавать, уничтожать, присваивать и т. д. Во-вторых, они обладают свойствами, обычно ассоциируемыми с указателями, скажем, адресацией объектов, динамически создаваемых в куче. Но при этом объекты автоматически создаются без вызова оператора new и автоматически уничтожаются без вызова оператора delete; другими словами, в этом отношении они ведут себя так же, как автоматические и статические экземпляры классов. Например, пусть класс String реализован как подсчитываемый указатель (см. далее), и объект String встроен в класс A:

```
class A {
public:
    A() : s("hello world") { }
private:
    String s;
};
```

Тогда каждый созданный объект класса A будет содержать подсчитываемый указатель s класса String, который вроде бы указывает на String с содержимым "hello world". Копии объекта класса A тоже вроде бы будут содержать указатель на тот же объект String, что и оригинал: копирование s выглядит как копирование указателя. Но при этом сохраняются все свойства настоящих объектов с подсчетом ссылок. Например, изменение значения приведет к отказу от использования общего представления. Другими словами подсчитываемые указатели ведут себя как указатели, у которых имеются конструкторы, поэтому при входе в область видимости они автоматически инициализируются и ассоциируются с вновь созданным объектом.

В листинге 3.9 приведена реализация класса StringRep для подсчета указателей. Она базируется на предыдущем примере, но конкатенация перемещена из String в StringRep ближе к той информации, с которой эта реализация работает. Кроме того, определен конструктор для создания StringRep на базе const char* и вспомогательная функция print.

Листинг 3.9. «Внутренняя реализация» подсчета указателей на String

```
class StringRep {
friend String;
private:
    StringRep()       { *(rep = new char[1]) = '\0'; }
    StringRep(const StringRep& s) {
        ::strcpy(rep=new char[::strlen(s.rep)+1], s.rep);
    }
    ~StringRep()      { delete[] rep; }
    StringRep(const char *s) {
        ::strcpy(rep=new char[::strlen(s)+1], s);
    }
    String operator+(const String&) const {
```

продолжение

Листинг 3.9 (*продолжение*)

```
        char *buf = new char[s->length() + length() + 1];
        ::strcpy(buf, rep);
        ::strcat(buf, s->rep);
        String retval( &buf );
        return retval;
    }
    int length() const   { return ::strlen(rep); }
    void print() const   { ::printf("%s\n", rep); }
private:
    StringRep(char** const r) {
        rep = *r; *r = 0; count = 1;
    }
    char *rep;
    int count;
};
```

В листинге 3.10 приведен класс String, из которого исключена операция length; любой вызов операции length для String перенаправляется StringRep вызовом функции operator->. То же относится к другим функциям, которые могут поддерживаться классом: strchr (поиск символа в строке), hash и т. д. Класс String по-прежнему выполняет большую часть работы по управлению памятью и подсчету ссылок в экземпляре StringRep. В листинге 3.9 приводится реализация оператора конкатенации StringRep::operator+. Она выражается главным образом на основе членов StringRep, что позволяет обойтись без лишнего уровня абстракции по сравнению с реализацией из листинга 3.8. Так как класс String перенаправляет вызовы функций operator+ классу StringRep, такой подход требует затрат на лишний вызов функции при каждом вызове operator+. Впрочем, обычно эти затраты ликвидируются за счет оптимизации с подстановкой функций.

Листинг 3.10. Открытый интерфейс класса String с подсчетом указателей

```
class String {
friend class StringRep;
public:
    String operator+(const String &s) const { return *p + s; }
    StringRep* operator->() const   { return p; }
    String() {
        (p = new StringRep())->count = 1;
    }
    String(const String& s)  { (p = s.p)->count++; }
    String(const char *s) {
        (p = new StringRep(s))->count = 1;
    }
    String operator=(const String& q) {
        // Реализация слегка изменена для разнообразия
        if (--p->count <= 0 && p != q.p) delete p;
```

```
        (p=q.p)->count++; return *this;
    }
    ~String()       { if (--p->count<=0) delete p; }
private:
    String(char** r) {
        p = new StringRep(r);
    }
    StringRep *p;
};
```

Теперь вместо того, чтобы использовать String как экземпляр, мы используем его как указатель (этот класс, единственное представление которого является указателем, можно рассматривать его как указатель с операторами). В сущности, теперь все ссылки на String осуществляются *только* через указатели. Несколько странно лишь то, что они не объявляются как указатели:

```
int main()
{
    String a("abcd"), b("efgh");
    printf("a is "); a->print();
    printf("b is "); b->print();
    printf("length of b is %d\n", b->length());
    printf("catenation is "); (a+b)->print();
    return 0;
}
```

Одно из преимуществ подсчета указателей состоит в том, что новые функции добавляются только в класс-тело. Например, большинство новых строковых операций может быть реализовано на уровне функций StringRep; класс-манипулятор String автоматически получает эти операции благодаря оператору ->.

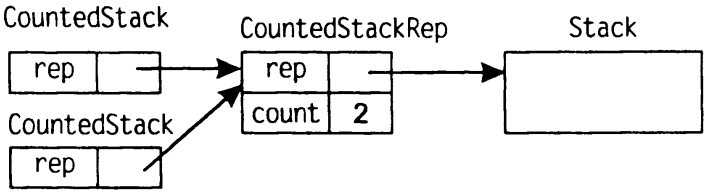
ПРИМЕЧАНИЕ

Используйте подсчет указателей, чтобы избавиться от хлопот с ручным освобождением динамически выделенной памяти. Подсчет указателей также может пригодиться при построении прототипов, когда приходится согласовывать изменения в сигнатуре тела с сигнатурой манипулятора. Подсчет указателей в основном определяется стилем и личным вкусом программиста, хотя этот механизм является важным компонентом идиом, представленных в главе 9.

Реализация подсчета ссылок в существующих классах

Реализация подсчета ссылок в существующих классах иногда существенно улучшает их быстродействие. На этом пути могут возникать различные препятствия, в том числе отсутствие доступа к исходным файлам, ограничения на размер или структуру объекта в памяти, которая будет нарушена добавлением поля счетчика, необходимость подсчета ссылок лишь в отдельных случаях и т. д. Для таких классов приходится вводить новый уровень с дополнительным классом-манипулятором.

Предположим, библиотека уже содержит класс Stack без подсчета ссылок, но специфика приложения требует применения стека с подсчетом ссылок. Мы создаем новую абстракцию стека с именем CountedStack на базе существующего класса Stack:



Исходная абстракция Stack остается без изменений, а для управления используются два класса. Класс CountedStack обеспечивает управление памятью, а класс CountedStackRep используется как простая структура данных для создания дополнительного уровня косвенных обращений:

```
class CountedStack {
public:
    CountedStack(): rep(new CountedStackRep) { }
    int pop() { return rep->rep->pop(); }
    void push(int i) { rep->rep->push(i); }
    CountedStack(const CountedStack &c): rep(c.rep) {
        rep->count++;
    }
    CountedStack(const Stack &c):
        rep(new CountedStackRep(new Stack(c))) { }
    operator Stack() { return *(rep->rep); }
    ~CountedStack() {
        if (--rep->count <= 0) delete rep;
    }
    CountedStack &operator=(const CountedStack &c) {
        c.rep->count++;
        if (--rep->count <= 0) delete rep;
        rep = c.rep; return *this;
    }
private:
    struct CountedStackRep {
        int count;
        Stack *rep;
        CountedStackRep(Stack *s = 0): count(1) {
            rep = s? s: new Stack;
        }
    } *rep;
};
```

Здесь нам пришлось проделать некоторую дополнительную работу, чтобы объекты CountedStack и объекты Stack стали взаимозаменяемыми.

Классы конвертов и синглетные письма

Ранее было показано, как разделить String на два класса для предотвращения дублирования памяти и затрат на копирование объектов. Внешний класс String назывался манипулятором, а класс StringRep — телом. Оба класса были видны пользователю и находились в глобальном пространстве имен. Все классы, содержащие указатели на динамически выделяемые ресурсы, должны реализовываться в виде пар «манипулятор/тело» или «конверт/письмо», если они должны использоваться как конкретные типы данных. В этом разделе будет показано, что во многих случаях классы тела и письма могут инкапсулироваться в соответствующих классах манипулятора и конверта для создания более устойчивых абстракций.

Взаимодействие String/StringRep является примером *синглетного класса письма*. Класс «конверта» String предназначен для хранения только одного вида «писем», а именно StringRep. В совокупности они образуют составной объект, который с точки зрения приложения рассматривается как единая сущность. Письмо *никогда* не рассматривается вне конверта. Следовательно, вместо того чтобы размещать экземпляр письма в экземпляре конверта, мы размещаем *класс* письма в *классе* конверта (листинг 3.11).

Листинг 3.11. Инкапсуляция синглетного письма в виде вложенного класса

```
class String {
public:
    String()                    { rep = new StringRep(""); }
    String(const String& s)     { rep = s.rep; rep->count++; }
    String& operator=(const String& s) {
       s.rep->count++;
       if (--rep->count == 0)
          delete rep;
       rep = s.rep;
       return *this;
    }
    ~String() {
        if (--rep->count <= 0) delete rep;
    }
    String(const char *s)       { rep = new StringRep(s); }
    String operator+(const String&) const;
    int length() const          { return ::strlen(rep->rep); }
private:
    class StringRep {
    public:
        StringRep(const char *s) {
           ::strcpy(rep = new char[::strlen(s)+1], s);
           count = 1;
        }
```

продолжение

Листинг 3.11 (*продолжение*)

```
        ~StringRep() { delete rep; }
        char *rep;
        int count;
    };
    StringRep *rep;
};
```

ПРИМЕЧАНИЕ

Синглетные классы писем используются в том случае, если пара «конверт/письмо» применяется для подсчета ссылок, или если интерфейс письма должен быть видимым только для конверта. Идиома приносит наибольшую пользу для устойчивых абстракций, поскольку все изменения в письме требуют перекомпиляции конверта и всех его клиентов чаще, нежели при раздельном сопровождении письма и конверта.

Кодирование функций класса в этом варианте не отличается от предыдущей версии String. Данное решение улучшает инкапсуляцию и уменьшает загромождение глобального пространства имен по сравнению с использованием двух глобальных классов. Оборотная сторона состоит в том, что изменения в интерфейсе или данных класса письма требуют перекомпиляции большего объема кода.

Конверт может содержать несколько вложенных классов писем. В главе 5 будет представлен простой класс, генерирующий атомарные лексемы. Класс GeneralAtom является конвертом по отношению к классам писем Punct, Name и Oper. В отличие от класса StringRep, который остается невидимым для пользователя, эти классы могут напрямую задействоваться приложением. Письма могут объявляться в открытом интерфейсе конверта GeneralAtom. При обращении к таким вложенным классам должен присутствовать уточняющий префикс (вида GeneralAtom::Punct). Слишком большое количество классов писем в классе конверта усложняет интерфейс последнего. Естественно, приложения, в которых классы писем являются производными от класса конверта (например, если классы Complex, BigInteger и DoublePrecisionFloat являются производными от класса Number), не могут использовать методику вложенных классов.

3.6. Операторы new и delete

C++ предоставляет в распоряжение программиста возможность управлять выделением памяти. Такое управление может играть важную роль для «фокусов с памятью», необходимых для эффективной работы объектов классов как конкретных типов данных на некоторой платформе. Кроме того, управление памятью позволяет контролировать выделение памяти на уровне классов согласованно с системой поддержки типов языка. По умолчанию программы, написанные в большинстве реализаций C++, в конечном счете применяют примитивы malloc и free языка C для управления свободной памятью в адресном пространстве

пользовательского процесса. Частичный или полный отказ от этих примитивов объясняется несколькими причинами.

- Может возникнуть необходимость в статическом распределении памяти, как во встроенных системах (скажем, в телекоммуникационной системе или системе управления полетом), или в выделении памяти из фиксированного пула.
- Быстродействие функций malloc и free оказывается недостаточным для целей пользователя. Специализированный алгоритм может превзойти стандартные примитивы по производительности, особенно при большом количестве мелких объектов.
- Среда может не поддерживать функции управления памятью, поэтому их приходится предоставлять программисту.

Чтобы выделить память из динамического пула, программист вызывает оператор new с указанием типа. Скажем, такая команда создает один объект типа int:

```
new int;
```

А следующая команда создает вектор из десяти объектов типа int:

```
new int[10];
```

При вызове new для типа компилятор обеспечивает возврат указателя на соответствующий тип; в нашем примере в обоих случаях это будет указатель int*. Показанная ниже команда выделяет память под вектор, размер которого был указан пользователем, и присваивает ia указатель на выделенный блок:

```
int *ia = new int [ size ];
```

Динамически созданный вектор целых чисел не инициализируется, как и динамически создаваемые экземпляры всех встроенных типов. Объекты классов, созданные оператором new, инициализируются своими конструкторами.

Чтобы вернуть память вектора объектов в динамический пул, следует выполнить команду

```
delete[] ia;
```

При этом передается значение ia, полученное ранее при вызове new. Память, на которую ссылается указатель, возвращается в систему. Квадратные скобки при уничтожении динамически выделенного вектора объектов указывать *желательно*, а при уничтожении вектора объектов, для которых должен быть вызван деструктор, — *обязательно*. Синглетный объект создается такой командой:

```
ia = new int;
```

Этот объект освобождается командой

```
delete ia;
```

Для начала рассмотрим ситуацию, в которой мы хотим полностью контролировать управление памятью, например, если вы хотите использовать более быстрый

алгоритм управления памятью, или если C++ работает в среде, не имеющей базовой поддержки (скажем, в автономной программе на внесистемном микропроцессоре). Обе задачи решаются переопределением операторов new и delete. Переопределенные версии этих операторов приведены в листинге 3.12. Используется простой алгоритм, в котором блоки памяти хранятся в едином списке и который удовлетворяет запросы выборкой первого блока достаточного размера. Блоки, находящиеся в списке, интерпретируются как экземпляры struct Head — специальной структуры данных для организации списка.

Листинг 3.12. Простой алгоритм распределения памяти для операторов new и delete

```
// stddef.h для использования size_t
#include <stddef.h>

struct Head {
   long length;
   struct Head *next;
};

static Head pool = { 0, 0 };

static Head *h = (Head *)HEAP_BASE_ADDRESS;

/* Быстрый, но глупый алгоритм выделения
   первого блока достаточного размера */

typedef char *Char_p;

// СЛЕДУЮЩАЯ СТРОКА ЗАВИСИТ ОТ ПЛАТФОРМЫ И КОМПИЛЯТОРА:
const long WRDSIZE = sizeof(void*);

void* operator new(size_t nbytes)
{
    /* Начинаем с поиска в списке */
    if (pool.next) {
        Head *prev = &pool;
        for (Head *cur = &pool; cur; cur = cur->next) {
            if (cur->length >= nbytes) {
                prev->next = cur->next;
                return (void*)(Char_p(cur) + sizeof(Head));
            } else prev = cur;
        }
    }
    // В списке нет подходящего блока.
    // запросить новый блок из кучи
    h = (Head*)(((int)((Char_p(h)+WRDSIZE-1))/WRDSIZE)*WRDSIZE);
    h->next = 0;
```

```
    h->length = nbytes;
    h += nbytes + sizeof(Head);
    return (void*)(Char_p(h) + sizeof(Head));
}

void operator delete(void *ptr)
{
    Head *p = (Head *)(Char_p(ptr) - sizeof(Head));
    p->next = pool.next;
    pool.next = p;
}
```

Теперь каждый раз, когда в функции C++ встретится вызов оператора new или delete, вместо операторов стандартной библиотеки будут вызываться перегруженные версии. Алгоритм управления памятью, задействованный перегруженными версиями операторов, может быть абсолютно произвольным; алгоритмы, показанные в листинге 3.12, используют блок кучи, который начинается с адреса HEAP_BASE_ADDRESS и распространяется до бесконечности.

Другой распространенный трюк основан на специализированном распределении памяти на уровне классов. Он имеет свои преимущества даже в операционных системах с собственными примитивами управления памятью, потому что пользователь может предоставить алгоритм, учитывающий особенности конкретного класса (в отличие от универсальных примитивов операционной системы). Для классов, часто создающих мелкие объекты (по эмпирической оценке — от 4 до 10 байт), применение перегруженных версий позволяет ускорить работу программы на порядок по сравнению со стандартными примитивами на базе функции malloc. Кроме того, механизмы специализированного распределения памяти помогают более экономно расходовать память, поскольку позволяют избегать присущих malloc затрат от 4 до 8 байт на каждый выделенный блок.

Для любого класса создаваемые объекты всегда имеют одинаковый размер. Иногда для объектов, создаваемых динамически оператором new, заранее выделяется память блоками, кратными размеру объекта класса. Каждый класс поддерживает собственный пул объектов, размер которых в точности соответствует запросам оператора new.

В листинге 3.13 объявляется класс String с собственными операторами выделения памяти, подменяющими семантику операторов new и delete глобального уровня. В остальном открытый интерфейс String остается таким же, как и без механизма специализированного управления памятью. Объединение (union) в реализации String позволяет рассматривать блок памяти либо как активный объект String (в этом случае содержимое объединения интерпретируется как указатель на представление строки), либо как блок памяти в списке свободных блоков (и тогда содержимое интерпретируется как ссылка в списке). Начало списка определяется закрытой статической переменной newlist.

Листинг 3.13. Объявление объекта String со специализированными операторами управления памятью

```
// Примитивная строка (не конкретный тип данных)
// для демонстрации операторов new и delete

class String {
public:
          String() { rep = new char[1]; *rep = '\0'; }
          String(const char *s) {
              rep = new char[::strlen(s)+1];
              ::strcpy(rep,s);
          }
          ~String() { delete[] rep; }
   void*  operator new(size_t);
   void   operator delete(void*);
          // ...    // Другие операции
private:
          static String *newlist;
          union {
               String *freepointer;
               char *rep;
          };
          int len;
};
```

В листинге 3.14 представлены операторы new и delete для класса String; они замещают стандартные примитивы управления памятью при построении объектов String. Следующее выражение выделяет память для объекта String, после чего вызывается конструктор String для его инициализации:

```
new String
```

В конструкторе память представления строки выделяется из кучи в виде символьного вектора; при этом вместо оператора new класса String используется системная версия. Приведенная здесь реализация ведет связанный список блоков памяти, размер которых соответствует размеру экземпляра String; уничтоженный объект String помещается в список, и запросы на создание новых экземпляров удовлетворяются удалением блоков из списка. Если список окажется исчерпанным, создается новый пул из 100 объектов String; как известно, однократный вызов malloc для большого блока обходится дешевле, чем многократный вызов для маленьких блоков.

Листинг 3.14. Операторы new и delete класса String

```
String *String::newlist = 0;

void String::operator delete(void *p) {
    String *s = (String *)p;
    s->freepointer = newlist;
    newlist = s;
}
```

```
void* String::operator new(size_t size) {
    if (size != sizeof(String)) {
        // Эта проверка при наследовании необходима
        // (см. главу 4), когда производный класс
        // не содержит оператора new.
        // String::operator delete (см. выше)
        // освободит увеличенный блок.
        return malloc(size);
    } else if (!newlist) {
        newlist = (String *)new char[100 * sizeof(String)];
        for (int i = 0; i < 99; i++) {
            newlist[i].freepointer = &(newlist[i+1]);
        }
        newlist[i].freepointer =0;
    }
    String *savenew = newlist;
    newlist = newlist->freepointer;
    return savenew;
}
```

Абстракция String скрывает все подробности управления памятью, причем объект String может использоваться программами точно так же, как и без специального механизма управления памятью. В следующем фрагменте компилятор интерпретирует вызов new String как запрос на выделение памяти оператором new класса String, если такой оператор существует, и стандартной системной версией ::operator new при его отсутствии:

```
int main() {
    String *p = new String("abcdef");
    delete p;
    return 0;
}
```

Слегка видоизмененная схема распределения памяти работает в приложениях, в которых функции operator new и operator delete используют статически выделенную память. Более того, программа даже упрощается, поскольку при исчерпании всех заранее выделенных блоков происходит ошибка (например, вызов системной функции восстановления) вместо попытки выделить дополнительные блоки и включить их в пул. Пример реализации такого рода приведен в листингах 3.15 и 3.16.

Листинг 3.15. Объявление класса String, использующего статический пул памяти

```
class String {
private:
          static String *newlist;
          union {
               String *freepointer;
               char *rep;
```

продолжение

Листинг 3.15 (*продолжение*)

```
        };
        int len;
public:
        enum { POOLSIZE = 1000 } ;
        String() { rep = new char[1]; *rep = '\0'; }
        String(const char *s) {
            rep = new char[::strlen(s)+1];
            ::strcpy(rep,s);
        }
        ~String() { delete[] rep; }
    void  *operator new(size_t);
    void  operator delete(void*);
        // ...    // Другие операции
};
```

Листинг 3.16. Операторы new и delete класса String со статическим пулом памяти

```
String* String::newlist = 0;
static unsigned char memPool[String::POOLSIZE * sizeof(String)];
static const unsigned char *memPoolEnd =
    memPool + String::POOLSIZE + sizeof(String);

void String::operator delete(void *p) {
    String *s = (String *)p;
    s->freepointer = newlist;
    newlist = s;
}

void* String::operator new(size_t size) {
    if (size != sizeof(String)) {
        // Эта проверка при наследовании необходима
        // (см. главу 4), когда производный класс
        // не содержит оператора new.
        return ::malloc(size);
    } else if (newlist == memPoolEnd) {
        StringOutOfMemory();
        return 0
    } else if (!newlist) {
        newlist = (String*)memPool;
        for (int i = 0; i < POOLSIZE-1; i++) {
            newlist[i].freepointer = &(newlist[i+1]);
        }
        newlist[i].freepointer = memPoolEnd;
    }
    String *savenew = newlist;
    newlist = newlist->freepointer;
    return savenew;
}
```

Векторы всех типов создаются глобальным оператором new. Даже если класс String содержит собственный оператор new, следующий вызов выделяет один блок памяти размером 25*sizeof(String) байт, причем этот блок выделяется функцией operator new:

```
String *sp = new String[25];
```

Дело в том, что объект выделяется как вектор, а векторы являются артефактами C, а не объектами классов C++.

3.7. Отделение инициализации от создания экземпляра

Иногда бывает важно отделить *создание экземпляра*, то есть процесс выделения основных ресурсов для объекта, от *инициализации* — настройки начального объекта. Такое разделение играет важную роль при инициализации (исходной или повторной) в автономных микропроцессорных системах. Кроме того, оно может пригодиться при инициализации объектов, содержащих взаимные ссылки.

В качестве примера первого случая рассмотрим систему управления реального времени, работающую на автономном микропроцессоре. Система инициализируется подсистемой инициализации, задающей точный порядок вызова конструкторов для глобальных объектов. Например, глобальные таблицы, ссылки на которые хранятся в постоянной памяти, должны инициализироваться на ранней стадии с фиксированными адресами, чтобы они могли использоваться зашитым в постоянной памяти кодом инициализации. Размер и местонахождение этих объектов в памяти фиксируется, но содержимое инициализируется на стадии выполнения программы. Если во время выполнения в системе произойдет исправимая ошибка, возможно, потребуется заново инициализировать испорченные объекты «на месте», оставив остальные объекты без изменений. Например, сбойное периферийное устройство может быть заменено без остановки процессора (важный фактор в непрерывно работающих системах), после чего потребуется заново выполнить конструктор периферийного устройства. Другие объекты продолжат обращаться к объекту периферийного устройства по исходному адресу.

Рассмотрим другой пример: программу-редактор с классами Editor и Window; каждый класс содержит члены, ссылающиеся на экземпляр другого класса. Для каждого объекта существование другого является условием его собственной инициализации, поэтому между объектами возникает циклическая зависимость. Проблема может быть решена разделением инициализации между конструктором и функцией класса и обработкой взаимных ссылок в функции после того, как оба объекта будут созданы и (частично) инициализированы. С другой стороны, это противоречит главной причине создания конструкторов — гарантии *автоматической* инициализации объектов, чтобы программисту не приходилось помнить о вызове функции инициализации. В нашем примере появляется конструктор, после выполнения которого объект остается инициализированным лишь частично.

Подобные «эрзац-решения» неуклюжи и неполноценны — инициализацию проще отделить от создания объекта средствами языка программирования. В этом разделе показано, как без применения доморощенных решений сначала создать экземпляры объектов обоих классов, а затем инициализировать оба объекта.

Такой подход выходит за пределы «стандартного» набора приемов программирования на C++ и потому требует определенной осторожности. В C++ связывание инициализации с созданием объекта избавляет пользователя от необходимости при создании объекта помнить о вызове специальной функции инициализации. С другой стороны, аналогичные проблемы характерны и для уничтожения объекта с освобождением ресурсов, поэтому этот подход следует использовать только в том случае, если того требуют обстоятельства. Пример будет представлен далее, но мы еще вернемся к этой теме в главе 9 при рассмотрении более серьезных идиом.

Вашему вниманию предлагается листинг 3.17. Мы перегружаем оператор new с дополнительным параметром — адресом в памяти, по которому должен размещаться объект. Тело оператора new просто возвращает свой аргумент. Новая версия оператора new вызывается только при вызове new с дополнительным параметром в синтаксисе вида

new (*адрес_объекта*) *тип* (*параметры...*)

Листинг 3.17. Явное создание и уничтожение экземпляров объектов

```
#include <stddef.h>

void *operator new(size_t, void *p) { return p; }

struct S {
    S();
    ~S();
};

char buf [sizeof(S)][100];
int bufIndex = 0;

void f() {
    S a;    // Автоматическое создание/инициализация объекта

    S *p = new S;          // Явное создание объекта
                           // Оператор new находит память
                           // Автоматическая инициализация
    delete p;              // Явное удаление
                           // Автоматическая зачистка

    S *ppp = new (buf) S;  // Явное создание объекта в buf
                           // Автоматическая инициализация
    ppp->S::~S();          // Явная зачистка

    // "a" автоматически деинициализируется и уничтожается
    // при возвращении из f()
}
```

Такой вызов противоречит семантике автоматического размещения объектов в памяти и сводится к простому вызову конструктора для объекта, местонахождение которого известно заранее. Аналогичным образом можно применить деструктор к объекту без освобождения памяти, для чего деструктор напрямую вызывается в программе:

```
адрес_объекта->тип::~тип();
```

Общий результат: сначала в программе происходит выделение памяти, а затем инициализация как отдельная выполняемая по отдельному запросу операция. Объект может размещаться где угодно, а вместо buf может передаваться произвольный указатель.

СОВЕТ

Будьте внимательны и никогда не вызывайте оператор delete для объектов, созданных подобным способом. Диспетчер памяти ошибочно решит, что часть локальной памяти на самом деле была взята из кучи. Скорее всего, возникнет хаос.

В листинге 3.18 представлен другой пример с единственным глобальным объектом foobar типа Foo. Память для объекта резервируется на стадии компиляции; это можно сделать так, что программа будет работать на примитивной микропроцессорной платформе, для которой не существует операционных систем с поддержкой динамического распределения памяти. А может быть, значения, передаваемые конструктору Foo при инициализации foobar, должны генерироваться в результате неких вычислений в main или в другой функции. Кроме того, в этом примере одна и та же память (foobar) в разное время может использоваться двумя независимыми объектами класса Foo.

Листинг 3.18. Явное создание и уничтожение объектов в заранее выделенной памяти

```
#include <iostream.h>
#include <stddef.h>

void *operator new(size_t, void *p) { return p; }

class Foo {
public:
    Foo() { rep = 1; }
    Foo(int i) { rep = i; }
    ~Foo() { rep = 0; }
    void print() { cout << rep << "\n"; }
private:
    int rep;
};

struct { int:0; };  // Машинно-зависимое выравнивание
char foobar[sizeof(Foo)];

Foo foefum;
```

продолжение

Листинг 3.18 (*продолжение*)

```
int main()
{
    foefum.print();
    (&foefum)->Foo::~Foo(); // Вызывает преждевременную зачистку
                            // глобального объекта
    foefum.print();         // Неопределенный результат!
    Foo *fooptr = new(foobar) Foo;
    fooptr->Foo::~Foo();
    fooptr = new(foobar) Foo(1); // Не связано с предыдущим
                            // созданием объекта
    fooptr->Foo::~Foo();    // Преждевременная зачистка
                            // НЕ ВЫЗЫВАЙТЕ delete для fooptr!
    fooptr = new Foo;
    delete fooptr;
}
```

Преждевременная зачистка объекта foefum в листинге 3.18 не должна выполняться, если среда вызывает деструкторы глобальных переменных при выходе из программы; в этом случае деструктор для foefum будет вызван дважды! Такое решение безопасно только при повторном создании foefum (командой new (&foefum) Foo), если платформа не вызывает деструкторы при выходе из программы, или если объект был размещен в памяти, первоначально не выделенной для объекта класса (как в случае с footpr).

Упражнения

1. Опишите различия между структурами, классами, абстрактными типами данных и конкретными типами данных.
2. Перечислите, сравните и выделите различия между известными вам механизмами абстракций в языке C++.
3. Измените класс String таким образом, чтобы он позволял извлекать подстроки из существующей строки, оставляя ее без изменений:

   ```
   String s = "abcdefg";
   ...
   String t = s (j, k);
   ```

   Здесь j определяет начало подстроки (1 соответствует первому символу). Если параметр j отрицателен, то значение задается *от конца* строки *в обратном направлении*, а последний символ задается значением –1 (обозначение конца строк нуль-символами используется на более низком уровне в языке C). Если параметр k отрицателен, то j интерпретируется как индекс конца строки, а подстрока отсчитывается справа налево.
4. Подумайте, как решить проблемы с выходом за пределы строки в предыдущем примере.

5. Конструктор X::X(const X&) является частью канонической формы класса. В частности, он используется для копирования параметров, переданных при вызове функций, в стек и создания «копий» переменных вызывающей стороны в кадре стека. Покажите, что произойдет с объектом класса X, у которого имеется конструктор X::X(X), но нет канонического конструктора X::X(const X&).
6. Напишите конструктор Node, который бы получал строку, созданную функцией String::String(Node), и строил по ней все дерево.
7. Напишите распределитель памяти, использующий только статически выделенную память, то есть память, местонахождение и размер которой фиксируется программой до запуска. Что делать, когда вся свободная память будет исчерпана?
8. Что произойдет, если оператор new выделит блок большего размера, чем требуется для объектов его класса? Сможете ли вы найти практические применения для такой возможности? А если будет выделен блок меньшего размера?
9. Усовершенствуйте функцию printf, написанную вами для упражнения к приложению A (или доработайте код printf в своей версии C), чтобы функция поддерживала формат %S и соответствующий аргумент String.
10. Покажите, что если все классы будут следовать канонической форме, представленной выше для String с оператором -> (см. листинг 3.10), то программе C++ никогда не придется задействовать указатели на объекты класса на уровне приложения (то есть вне классов, использующих функцию operator->).
11. Имеется следующий фрагмент:

```
class String {
public:
    ...
    String(char* s) {... }
    operator const char*() const { ... }
    ...
};

extern "C" {
    int open(char*, int ...);
}

int open(String, int ...);
```

Что именно произойдет при выполнении каждой из следующих команд? Почему? Являются ли они семантически идентичными? Почему? (Попробуйте включить в программу команды трассировки, чтобы разобраться в происходящем).

```
int i = open("abcd", O_RDONLY);
int j = open(String("abcd", O_RDONLY);
String abcd = "abcd";
int k = open(abcd, O_RDONLY);
```

Возможно, это упражнение поможет вам научиться избегать неоднозначностей в своих разработках. В общем случае перегрузка должна рассматриваться как *системный* фактор проектирования, чтобы ваши объявления не конфликтовали с объявлениями других программистов.

12. Возьмите достаточно большую программу C++ и измените ее таким образом, чтобы деструкторы классов обнуляли значение this (если ваш компилятор поддерживает такую возможность). Сравните быстродействие двух версий программы, исходной и модифицированной. Можете ли вы объяснить различия?
13. Предложите другой способ реализации функции strlen(String) с использованием решений из раздела 3.3 (вместо преобразования типа).
14. Начните с класса String, приведенного на с. 53. Включите перегруженную версию функции operator&(), возвращающую объект StringPointer. Объекты StrintPointer должны работать в программе всюду, где может использоваться объект String*. Завершите реализацию StringPointer, включив в нее перегруженные версии функций operator->() и operator*(). Внесите необходимые изменения в класс String. Где бы вы использовали эти два класса? Сравните это решение с тем, которое описано на с. 77.
15. Создайте новый класс Int с перегруженным оператором +, при котором следующие строки означали бы *бит* 25 OR *бит* 15 OR *бит* 14 OR *бит* 2, то есть 0200140004:

```
Int i = 0, j = 0;
j = i + 25 + 15 + 14 + 2;
```

Реализуйте аналогичную версию оператора – (минус) для *сброса* битов. Где можно использовать такой класс? Какие проблемы он создает? (Задача была предложена в качестве примера в [1]).

Литература

Lippman, Stanley B. «A C++ Primer», Reading, Mass.: Addison-Wesley, 1989.

Глава 4
Наследование

Наследование представляет собой языковой механизм, предназначенный для выражения особых отношений между классами. Из классов как абстракций языка программирования строятся иерархии, которые также являются абстракциями. Эти абстракции более высокого уровня закладывают основу для объектно-ориентированного программирования, о котором рассказывается в следующей главе.

Наследование обычно служит двум основным целям. Во-первых, наследование можно задействовать как механизм абстракции, средство организации классов в специализированные иерархии. Во-вторых, наследование обеспечивает возможность многократного использования кода и создания новых классов, сходных с уже существующими классами. В обоих случаях наследование можно рассматривать как механизм, при помощи которого класс делегирует (перепоручает) часть своих задач другому классу.

Отношение к наследованию как к механизму абстракции удобно с практической точки зрения. В языке C функции являются механизмом абстракции для алгоритмов, а структуры — основным механизмом абстракции данных. Классы C++ объединяют эти две концепции в абстрактный тип данных и его реализацию. При некотором усовершенствовании эти абстракции в C++ выглядят и ведут себя как полноценные типы, аналогичные встроенным типам языка, «входящим в поставку компилятора», что само по себе является мощным механизмом абстракции. Наследование позволяет идти еще дальше, предоставляя средства для объединения взаимосвязанных классов в обобщение высокого уровня, характеризующее их все сразу. Классы со сходным, сопоставимым поведением организуются в иерархию наследования. Класс, находящийся в вершине иерархии, рассматривается как абстракция для классов более низких уровней, избавляя программиста от всех подробностей его реализации.

В представлении о наследовании как о средстве построения новых абстракций на базе существующих, где один класс наследует данные и функции от других классов, язык C++ похож на большинство других объектно-ориентированных языков. Но в C++ наследование обычно определяет еще и совместимость типов. А именно, два типа могут быть настолько тесно связаны друг с другом, что объект одного типа может использоваться вместо объекта другого типа. Наследование применяется для выражения подобных связей, а проверка типов

компилятором позволяет организовать замену типов. Как показано в главе 6, открытое наследование класса B от класса A означает, что объекты класса B могут быть задействованы всюду, где допустимо использовать объекты класса A. Исходный класс, возглавляющий иерархию, называется *базовым классом*. В C++ базовым классом также называется непосредственный «предок» любого класса; в других языках программирования используется еще термин *суперкласс*. Новый класс, наследующий свойства родительского класса, то есть «потомок», называется *производным классом*; в других языках также встречается термин *субкласс*. В свою очередь, производный класс может быть базовым по отношению к другому классу. Иерархии наследования могут иметь произвольную глубину, хотя на практике обычно используется не более трех-четырех уровней.

Классы в иерархии наследования могут рассматриваться как разные формы абстракции, задаваемой базовым классом, а код, написанный для базового класса, будет работать с любой из этих форм. *Полиморфизм*, то есть способность интерпретировать разные формы класса как одну форму, — мощный механизм абстракции, скрывающий от программиста подробности реализации производных классов. Наследование позволяет установить связи между классами приложения, представляющими сходные ресурсы, поставщиков, клиентов и т. д. Такое применение наследования при проектировании упрощает управление большими системами.

Наследование также может рассматриваться как средство определения новых классов, последовательно уточняющих существующий класс. Если вам потребуется новый класс, который ведет себя как существующий класс со слегка измененной реализацией, наследование позволит выразить эту концепцию. Механизм наследования дает возможность определить новый класс на основе существующего класса, слегка изменить то, что требуется, и использовать готовый код остальной части класса, то есть наследование может рассматриваться как альтернатива условной компиляции с применением директив #if и #ifdef или принятию решений во время выполнения программы (см. раздел 7.9). Новый класс наследует поведение существующего класса и изменяет его, подменяя отдельные функции.

Таким образом, наследование может применяться для двух слегка различающихся целей.

- *Наследование служит для отражения семантических отношений между классами.* Иногда такие отношения возникают в результате эволюции, когда одни функции производного класса уточняют или специализируют поведение базового класса, а другие функции наследуются без изменений. Также наследование может отражать существенное сходство между двумя существующими абстракциями — настолько сильное, что объект производного класса может использоваться везде, где ожидается объект базового класса. При реализации подобных отношений обычно применяются:
  - открытое наследование;
  - виртуальные функции классов.

- *Наследование обеспечивает многократное использование кода.* Хотя производный класс может не обладать всеми свойствами базового класса, он может быть настолько близок к нему, что возможность многократного использования готового кода оправдает наследование. Хотя классы очереди и циклического списка не являются взаимозаменяемыми, они содержат достаточно большой объем общего кода, чтобы наследование упростило развитие этих двух абстракций. При реализации этих отношений обычно применяются:
  - закрытое наследование;
  - невиртуальные функции классов.

Впрочем, эти рекомендации не следует считать истиной в последней инстанции. Они позволяют лишь дать представление о том, что вас ждет впереди, и показать, как эта глава связана со следующим материалом. Выбор между открытым и закрытым вариантами наследования в большей степени определяется соображениями проектирования, нежели реализации, поэтому в главе 6 они будут рассматриваться отдельно. Рассмотрение виртуальных функций откладывается до знакомства с объектно-ориентированным программированием в главе 5. Из примеров этой главы класс PathName ориентируется на многократное использование кода, а в классах Imaginary и Telephone воплощено моделирование семантических связей между классами.

Обсуждение *канонической формы наследования* тоже придется отложить до главы 5. Дело в том, что разобраться во всех нюансах можно лишь при понимании концепций виртуальных функций и множественного наследования, а эти вопросы изучаются позже. Множественное наследование уместно рассматривать при описании объектно-ориентированного программирования. Но без виртуальных функций от множественного наследования особой пользы нет, поэтому разбирать здесь изощренные примеры с множественным наследованием было бы бессмысленно.

4.1. Простое наследование

Наше знакомство с наследованием начнется с класса Complex (листинг 4.1). Комплексные числа применяются в физике и прикладных дисциплинах как обобщение концепции рациональных, мнимых, вещественных и целых чисел. Код классов Rational, Imaginary и т. д. будет строиться последовательной модификацией класса Complex. Эти классы идеально подходят для демонстрации наследования.

Листинг 4.1. Простой класс Complex

```
class Complex {
friend Imaginary;
public:
    Complex(double r = 0, double i = 0): rpart(r), ipart(i) { }
    Complex(const Complex &c): rpart(c.rpart), ipart(c.ipart) { }
```

продолжение

Листинг 4.1 (*продолжение*)

```
    Complex& operator=(const Complex &c) {
        rpart = c.rpart; ipart = c.ipart; return *this;
    }
    Complex operator+(const Complex &c) const {
        return Complex(rpart + c.rpart, ipart + c.ipart);
    }
    friend Complex operator+(double d, const Complex &c) {
        return c + Complex(d);
    }
    friend Complex operator+(int i, const Complex &c) {
        return c + Complex(i);
    }
    Complex operator-(const Complex &c1) const {
        return Complex(rpart - c1.rpart, ipart - c1.ipart);
    }
    friend Complex operator-(double d, const Complex &c) {
        return -c + Complex(d);
    }
    friend Complex operator-(int i, const Complex &c) {
        return -c + Complex(i);
    }
    Complex operator*(const Complex &c) const {
        return Complex (rpart*c.rpart - ipart*c.ipart,
                    rpart*c.ipart + c.rpart*ipart);
    }
    friend Complex operator*(double d, const Complex& c) {
        return c * Complex(d);
    }
    friend Complex operator*(int i, const Complex& c) {
        return c * Complex(i);
    }
    friend ostream& operator<<(ostream &o, const Complex& c) {
      o << "(" << c.rpart << "," << c.ipart << ")"; return o;
    }
    Complex operator/(const Complex &c) const { return 0; }
    operator double() {
        return ::sqrt(rpart*rpart + ipart*ipart);
    }
    Complex operator-() const { return Complex(-rpart, -ipart); }
private:
    double rpart, ipart;
};
```

А вот как выглядит простой класс мнимых чисел Imaginary, *производный* от класса Complex:

```
class Complex {
friend class Imaginary;
```

```
public:
    // ...
};

class Imaginary: public Complex {
public:
    Imaginary(double i = 0): Complex(0, i) { }
    Imaginary(const Complex &c): Complex(0, c.ipart) { }
    Imaginary& operator=(const Complex &c) {
        rpart = 0; ipart = c.ipart; return *this;
    }
};
```

Обратите внимание: класс Imaginary объявлен дружественным по отношению к Complex. Классы этой иерархии образуют единый логический пакет. Группировка классов обеспечивает выигрыш в эффективности, а также упрощает запись. Как правило, классы не должны раскрывать свою внутреннюю структуру даже перед производными классами — не стоит рассматривать наследование как лицензию на нарушение инкапсуляции. С другой стороны, этот пример показывает, как производные классы могут влиять на архитектуру базового класса, и наоборот. Итеративная методика и прототипизация важны для оптимальной организации классов и наследования.

Зависимости между уровнями иерархии классов проясняются по мере разработки приложения. Рассмотрим следующий фрагмент:

```
int main() {
    Complex a(42.42);
    Imaginary b = 2;
    Complex c = a - b;
    // c = Complex (42.0)
    return 0;
}
```

Результат, возвращаемый функцией Complex::operator-, объявлен со статическим типом Complex, но возвращаемое значение в действительности представляет собой вещественное число. При этом мы сталкиваемся с двумя проблемами. Первая, специфическая для числовых типов C++, возникает из-за двойственности между встроенными типами вроде double и пользовательскими классами (такими, как Complex и Imaginary). Нам бы *хотелось*, чтобы тип double занял какое-то место в иерархии Complex. Особенно хочется объявить double производным от Complex (в конце концов, double является частным случаем Complex). Но поскольку double является встроенным типом, а не классом, его нельзя встроить в иерархию наследования, поэтому у нас нет простого способа сообщить компилятору, что double и Complex совместимы по отношению к типам. Фактически лучшее, что можно сделать — обеспечить автоматическое преобразование типа double в Complex (см. раздел 3.4), но это приведет к лишним затратам

во всех операциях с типом double. Альтернативное решение — заменить double полноценным классом:

```
class Double: public Complex {
    // Double является частным случаем Complex
public:
    Double(double r = 0): Complex(r) { }
    Double(const Complex &c) : Complex(c.rpart) { }
    Double& operator=(const Complex &c) {
        rpart = c.rpart; ipart = 0; return *this;
    }
    ... // Операции, оптимизированные для Double
};
```

Первая проблема решена, но есть другая, более общая: класс Complex содержит функцию, которая получает параметр Imaginary и возвращает результат Double. В общем случае классы, участвующие в наследовании, ничего не знают друг о друге. Исчерпывающее решение этой проблемы выходит за рамки темы настоящей главы, но ее решение средствами объектно-ориентированного программирования представлено в разделе 5.5.

Настройка операций класса Complex для семантики класса Imaginary

Класс Imaginary наследует большинство операций от родительского класса. Тем не менее, мы хотим гарантировать равенство нулю вещественной части в объектах Imaginary. Класс Imaginary *переопределяет* операции, которые могут привести к нарушению этого условия. Например, присваивание объекта Complex объекту Imaginary требует специальной обработки, поскольку Imaginary не содержит вещественной части. Проблема проще всего решается выдачей ошибки времени выполнения или обнулением вещественной составляющей при присваивании. Мы пойдем по второму пути. Преобразование выполняется с потерей информации (вещественной части), но это вполне согласуется с потерей дробной части при преобразовании типа double в тип int.

Среди остальных функций базового класса только функция Complex::Complex(double, double) может занести в поле rpart ненулевое значение, но эта функция не наследуется классом Imaginary. Конструкторы базового класса не наследуются производными классами.

Использование кода базового класса в производном классе

Наследование позволяет классу Imaginary использовать код математических операторов класса Complex. Мы создали новый тип, который выполняет большую часть своей работы, применяя функции существующего типа. Замена возможна

благодаря тому, что мнимые числа являются комплексными; просто это особая *разновидность* комплексных чисел, например, в следующем примере объект Imaginary суммируется с Complex, давая в сумме Complex:

```
int main() {
    Complex c = Complex(1,2);
    Imaginary k = -1;
    Complex sum = c+k;  // Разрешается: c.operator+(const Complex&)
    // sum = Complex(1,1)
    return 0;
}
```

Хотя функция Complex::operator+ получает параметр Complex, класс Imaginary является *частным случаем* класса Complex; компилятор знает об этом факте благодаря отношениям наследования между классами. Все, что класс Complex предоставляет в своем интерфейсе, присутствует и в интерфейсе Imaginary. Объявление открытого интерфейса C++ отражает эту связь и позволяет передавать объекты, объявленные с типом Imaginary, везде, где ожидаются объекты Complex. Таким образом, наследование определяет совместимость типов между сходными классами.

Изменение функций производного класса для повышения эффективности

Функции класса Complex, вызванные для объекта Imaginary, работают правильно, но выполняют лишнюю работу. Например, при вызове оператора double, возвращающего модуль комплексного числа в формате double, выполняется умножение двух пар чисел с возвратом квадратного корня из их суммы. Алгоритм работает для Imaginary, но его можно упростить и сделать более эффективным для мнимых чисел, поскольку их вещественная часть заведомо равна нулю. В класс Imaginary включается новая версия операторной функции operator double:

```
Imaginary::operator double() {
    return ipart;
}
```

Новая версия вызывается для переменных, объявленных с типом Imaginary, а исходная версия по-прежнему используется для объектов Complex:

```
int main() {
    Imaginary i = -1;
    Complex j = Complex(0, -1);
    double a, b;
    a = double(i);  // Imaginary::operator double
    b = double(j);  // Complex::operator double
    // В обоих случаях результат равен 1
    ...
}
```

В примерах, приводившихся до настоящего момента, для каждого базового класса существовал всего один производный класс. Конечно, наследование позволяет создать несколько классов на базе общего базового класса. В листинге 4.2 общие атрибуты нескольких разновидностей телефонов выделяются в базовый класс. Обратите внимание: объекты класса Telephone в программе не создаются, «абстрактные телефоны» существовать не могут. Класс Telephone присутствует только как заготовка, то есть шаблон для построения специализированных классов телефонов. Далее рассматривается примитивный механизм контроля над созданием объектов Telephone. Впрочем, класс Telephone умеет выполнять некоторые операции, общие для всех телефонов; в частности, он знает, где искать переменные с информацией о контексте вызова или другой системной информацией для определения состояния звонка. Более того, предполагается, что способ получения такой информации не зависит от конкретного типа телефона, то есть одни и те же механизмы задействуются как в «традиционных» телефонах, реализованных классом POTSPhone (POTS означает Plain Old Telephone Service — традиционная простая телефонная услуга), так и в «навороченных» телефонах класса ISDNPhone (ISDN означает Integrated Services Digital Network — цифровая сеть с комплексными услугами). А это означает, что мы можем использовать запись вида

```
POTSPhone phone1, phone2;
...
if (phone1.isTalking()) phone2.ring();
```

Класс POTSPhone *наследует* операции ring и isTalking от класса Telephone, словно операции были физически скопированы в производные классы. При этом многократное использование кода реализуется без его дублирования.

Листинг 4.2. Базовый класс, общий для двух производных классов

```
class Telephone {
public:
    void ring();
    Bool isOnHook();
    Bool isTalking();
    Bool isDialing();
    DigitString collectDigits();
    LineNumber extension();
    ~Telephone();
protected:
    LineNumber extensionData;
    Telephone();
};

// POTS - это традиционная простая телефонная услуга

class POTSPhone: public Telephone {
public:
```

```
    Bool runDiagnostics();
    POTSPhone();
    POTSPhone(POTSPhone&);
    ~POTSPhone();
private:
    // Служебная информация для подключения телефона
    Frame frameNumberVal;
    Rack rackNumberVal;
    Pair pairVal;
};

// ISDN - это цифровая сеть с комплексными услугами

class ISDNPhone: public Telephone {
public:
    ISDNPhone();
    ISDNPhone(ISDNPhone&);
    ~ISDNPhone();
    void sendBPacket();
    void sendDPacket();
private:
    Channel b1, b2, d;
};

class PrincessPhone: public POTSPhone {
    ...
};
```

4.2. Видимость и управление доступом

Итак, как было показано, наследование служит для объединения существующих типов и построения на их базе новых типов. Но если вспомнить о важности инкапсуляции и маскировки информации, мы также должны разобраться в том, как наследование влияет на доступ к членам классов — и напрямую связанных отношениями наследования, и классов, в которых один класс пользуется функциональностью, предоставляемой другим классом. Доступ к членам классов, связанных отношениями наследования, будет называться *вертикальным*; один из классов как бы находится выше, а другой — ниже в иерархии наследования. Под *горизонтальным доступом* понимается внешний доступ к членам классов «извне», со стороны классов, равноправных с точки зрения структуры программы. Наследование порождает целый ряд проблем в отношении доступа к членам классов, не характерных для простой схемы защиты, описанной в разделе 3.2. Например, доступен ли некоторый член класса для функций классов, производных от него? А для третьего класса, использующего функциональность объекта одного из производных классов?

Вертикальное управление доступом при наследовании

Класс может выбирать, какие из его членов должны быть доступны для производных классов. Производный класс не может обращаться к закрытым членам своего базового класса. Рассмотрим следующий фрагмент:

```
class A {
public:
    A();
    ~A();
    ...
private:
    int val;
};

class B: public A {
public:
    B();
    ~B();
    ...
    void func();     // Не может обращаться к A::val
};
```

Ни одна функция класса B не может обращаться к переменной val, хотя экземпляр val хранится в каждом объекте B. « Заградительный барьер», проходящий по границе вертикальной области видимости, полностью закрывает производным классам доступ к закрытым членам базового класса. Производные классы не имеют права нарушать инкапсуляцию базового класса, как и все остальные классы.

Спецификатор private делает переменную val недоступной для других классов, использующих классы A и B. Если бы переменная val была объявлена открытой, то она была бы доступна для функций класса B (и любого другого класса). Если мы хотим открыть доступ к некоторому члену класса A, но не делать его общедоступным, следует объявить его защищенным (protected):

```
class A {
public:
    A();
    ~A();
    ...
protected:
    int val;
};

class B: public A {
public:
    B();
    ~B();
```

```
    ...
    void func();  // Может обращаться к A::val
};

class C: public B {
public:
    C();
    ~C();
    ...
    void func2();  // Может обращаться к A::val
};
```

Здесь переменная val по-прежнему ведет себя так, словно она является закрытой для всех, *кроме* функций производных классов и их друзей.

Если потребуется ограничить доступ к переменной val так, чтобы она была доступна только для производного класса B, но не для C, можно воспользоваться конструкцией со спецификатором friend:

```
class A {
friend class B;
public:
    ...
private:        // Недоступно для всех, кроме членов и друзей A.
    int val;
};

class B: public A {
public:
    ...
    void func();   // Может обращаться к A::val
};

class C: public A {
public:
    C();
    ~C();
    ...
    void func3();  // Не может обращаться к A::val
};
```

Здесь класс B связан с A особыми отношениями, разрешающими доступ к закрытым полям базового класса из производного класса; у класса C таких прав нет. При объявлении защищенного члена базового класса все производные классы как бы становятся дружественными для базового класса. Все они имеют доступ к защищенным членам базового класса, поэтому базовый класс не приходится изменять каждый раз, когда потребуется предоставить доступ к его членам новому производному классу. Если бы мы использовали ключевое слово friend, при добавлении нового производного класса в базовый класс пришлось бы включать

новую секцию friend. С другой стороны, доступ к защищенным членам предоставляется всем производным классам без разбора, тогда как дружественные отношения можно устанавливать на уровне отдельных классов. Защищенные члены доступны для друзей и функций производных классов, но только при обращении через указатель, ссылку или объект производного класса. Этим и объясняется то, что класс Imaginary в листинге 4.1 был объявлен другом класса Complex вместо объявления переменных базового класса защищенными. В частности, оператор присваивания производного класса должен иметь доступ к членам переданного объекта базового класса:

```
Imaginary &Imaginary::operator=(const Complex &c) {
    rpart = 0;
    ipart = c.ipart;
    return *this;
}
```

Если бы переменные базового класса rpart и ipart были защищенными, то этот фрагмент вызвал бы ошибку компиляции.

Рассмотрим еще один пример — класс PathName, созданный на базе класса String из главы 3:

```
class String {
public:
    String()                    { rep = new StringRep(""); }
    String(const String& s)     { rep = s.rep; rep->count++; }
    String(const char *s)       { rep = new StringRep(s); }
    ~String() { if (--rep->count <= 0) delete rep; }
    String &operator=(const String &s) {
        s.rep->count++;
        if (--rep->count <= 0) delete rep;
        rep = s.rep;
        return *this;
    }
    String operator+(const String&) const; // Определяется
                                           // в другом месте
    operator const char* () const { return rep->rep; }
    String operator()(int,int) const;
          // Выделение подстрок, см. упражнения к главе 3
    int length() const          { return ::strlen(rep->rep); }
protected:
    StringRep *rep;
};
```

Класс PathName наследует свойства String:

```
class PathName: public String {
public:
    PathName(const String&);
    PathName(): baseNameRep(""), String() { }
```

```
    PathName(const PathName &p) : String(p),
        baseNameRep(p.baseNameRep) { /* Пусто */ }
    PathName &operator=(const PathName &p) {
        String::operator=(p); // Присваивание подобъекта
                              // базового класса
        baseNameRep = p.baseNameRep;
        return *this;
    }
    ~PathName()   { /* Пусто */ }
    String baseName()   { return baseNameRep; }
    String prefix();
    String suffix();
    String fullPathName() { return *this; }
    String dirName() {
        return (*this)(0, length() - baseName().length());
    }
    String changeBaseName(const String &newFile);
private:
    String baseNameRep;
};
```

Пользователь может вызвать любую открытую функцию класса String для объекта PathName, потому что класс PathName связан с классом String отношениями *открытого наследования*. Предположим, вы хотите запретить пользователю выполнение операций с подстроками для PathName, чтобы полное имя файла нельзя было изменять напрямую, а лишь посредством высокоуровневых функций, работающих на уровне компонентов пути вместо уровня отдельных символов (скажем, changeBaseName). Один из вариантов — организовать перехват обращений во время выполнения и выводить сообщение об ошибке:

```
class PathName: public String {
public:
    ...
    String operator()(int,int) {
        printf("Illegal PathName substring\n");
        return String("");
    }
    ...
};
```

Но в этом случае недопустимые обращения обнаруживаются лишь на стадии выполнения, тогда как подобные ошибки лучше перехватывать на стадии компиляции. Чтобы получить гарантированную защиту от вызова operator()(int,int) для объектов PathName, следует переопределить защиту оператора в классе PathName и сделать его недоступным. Получается, что функция operator()(int,int) может вызываться для объектов String, но не для объектов PathName. Но из этого следует, что объекты PathName не наследуют свойства String, или, по крайней мере, наследуют их не полностью.

Если нужно точно указать, какие компоненты интерфейса String должны быть доступны для кода, работающего с объектами класса PathName, следует применить закрытое наследование. Так, согласно следующему фрагменту только оператор преобразования const char* класса String «переносится» в интерфейс класса PathName:

```
class PathName: private String {
public:
    String::operator const char*;  // Из базового класса
    ... // Другие функции PathName
};
```

Такая конструкция называется *спецификатором доступа*. Она не создает новый оператор и не изменяет реализацию; просто оператор включается в открытый интерфейс PathName вместо того, чтобы скрываться из-за закрытого наследования. Другие операторы и функции класса — такие, как функция operator()(int,int) — остаются закрытыми по отношению к String и не публикуются в интерфейсе PathName. Если за пределами String или PathName встретится следующий фрагмент, компилятор выдаст сообщение о фатальной ошибке, в котором будет сказано, что оператор объявлен закрытым:

```
PathName p;
...p(1,3)...
```

Тем не менее операторная функция базового класса String::operator()(int,int) останется доступной для функций класса PathName.

Горизонтальное управление доступом при наследовании

В предыдущем разделе рассматривалось управление доступом к базовому классу из производного класса. Эти же конструкции, а также ряд других, позволяют управлять доступностью операций базового класса для пользователей.

Хотя базовый класс в какой-то степени контролирует доступность своих членов для производных классов, производный класс отчасти управляет тем, какая часть интерфейса базового класса будет доступна для клиентов производного класса. Эта возможность поддерживается двумя механизмами: разными режимами наследования, определяющими публикацию членов базового класса в интерфейсе производного класса, и использованием спецификаторов доступа на уровне отдельных членов.

Самая распространенная конструкция управления доступом к унаследованным членам носит название *открытого наследования*. Возьмем базовый класс A и производный класс B:

```
class A {     // Базовый класс (суперкласс)
public:
    ...
};
```

```
class B: public A {
public:
    ...
};
```

При открытом наследовании все открытые члены A доступны как члены класса B. Но при использовании следующей записи ни один член класса A не будет доступен как член класса B пользователям класса B:

```
class A {      // Базовый класс (суперкласс)
public:
    ...
};

class B: private A {
public:
    ...
};
```

Это называется *закрытым наследованием*. Впрочем, функции класса B сохраняют доступ к открытым и защищенным функциям класса A (см. далее).

Если класс закрыто наследует от другого класса, вы можете избирательно открыть доступ к некоторым частям открытого интерфейса базового класса через открытый интерфейс производного класса. Для этого применяется второй вариант управления доступом к унаследованным членам, а именно спецификаторы доступа. Раньше этот вариант уже применялся для классов String и PathName. Рассмотрим следующий класс:

```
class List {
public:
    virtual void *head();
    virtual void *tail();
    virtual int count();
    virtual long has(void*);
    virtual void insert(void*);
};
```

Рассмотрим еще один класс, закрыто наследующий от предыдущего:

```
class Set: private List {
public:
    void insert(void *m);
    List::count;       // Спецификаторы
    List::has;         // доступа
};
```

Класс множества Set использует функциональность класса списка List в своей реализации, но посредством закрытого наследования скрывает некоторые открытые функции своего базового класса (head и tail), потому что они не имеют смысла в операциях с множествами. Остальные члены наследуются без изменений,

кроме функции insert, семантика которой в множествах отличается от семантики в списках (список может содержать одинаковые элементы, а множество — нет).

Обратите внимание: поступить наоборот *нельзя*. Например, следующий фрагмент недопустим:

```
class Set: public List {
private:
    List::head;          // Недопустимо
    List::tail;          // Недопустимо
public:
    void insert(void *m);
};
```

Компилятор выдаст для него сообщение об ошибке. Открытое наследование Set от List предполагает, что интерфейс Set содержит все компоненты, присутствующие в интерфейсе List. А это означает, что объект Set должен уметь делать все, что делают объекты List. Система контроля типов C++ принимает объекты Set вместо объектов List, если они связаны открытым наследованием. Но при закрытом наследовании возможны аномалии вроде

```
void *listhead(List l) {
    return l.head();
}

int main() {
    Set s;
    void *p = listhead(s);
    return 0;
}
```

Допустим ли вызов l.head()? Если компилятор разрешит его, это может преподнести сюрприз программисту, указавшему, что функция head является закрытой для объектов Set, и приведет к нарушению инкапсуляции. А так как listhead может компилироваться отдельно от main(), компилятор в этом примере даже не сможет выдать диагностику на стадии компиляции. Чтобы застраховаться от подобной ситуации, компилятор запрещает саму возможность ее возникновения. Таким образом, доступ к символическому имени в производном классе с открытым наследованием не может быть ограничен более жестко, чем в базовом классе.

Все эти излишне подробные (на первый взгляд) пояснения имеют убедительную аналогию в объектно-ориентированном проектировании. Мы вернемся к этому вопросу в главе 6.

На практике почти всегда следует прибегать к открытому наследованию. Закрытое наследование обычно применяется для многократного использования кода базового класса в производном классе. Существует ряд других конструкций, которые рекомендуется задействовать вместо закрытого наследования (см. главу 6). К сожалению, в C++ наследование является закрытым по умолчанию (если

ключевое слово, определяющее тип наследования, не указано), поэтому обычно приходится специально делать его открытым.

Базовый класс не может *принудительно* сделать свои операции доступными через члены производного класса: это решение находится под контролем производных классов.

Производный класс может определить функцию с таким же именем, как у функции базового класса. С точки зрения внешнего пользователя эта новая функция замещает определение базового класса, даже если функции различаются по количеству/типу аргументов и возвращаемых значений. Пример приведен в листинге 4.3. Функции класса `B` могут ссылаться на функцию `f` класса `A` только с уточнением класса, а именно `A::f`.

Листинг 4.3. Наследование одноименных функций с разными сигнатурами

```
class A {                  // Базовый класс (суперкласс)
public:
    void f(int);
    void g(void*);
    //
};

class B: public A {        // Производный класс (субкласс)
public:
    void f(double);
    void g(int);
    // ...
};

void B::g(int k) {
    f(k);                  // B::f(double) с повышением
    A::f(k);               // A::f(int);
    this->A::f(k);         // A::f(int);
    ::f(k);                // Недопустимо
}

int main() {
    A *a;
    B *b;
    int i;
    // ...
    a->f(1);          // A::f(int)
    a->f(3.0);        // A::f(int) с понижением int(3.0)
    b->f(2);          // B::f(double) с повышением double(2)
    b->f(2.0);        // B::f(double)
    a->g(0);          // A::g(void*)
    b->g(0);          // B::g(int)
    b->g(&i);         // Ошибка: B::g скрывает A::g
}
```

Если программа обращается к объекту класса через ссылку или указатель на базовый класс, она сможет вызывать только функции базового класса. Но если функция вызывается напрямую для экземпляра производного класса с использованием синтаксиса *объект.функция*, то вместо функции базового класса вызывается одноименная функция производного класса.

Пользователь класса может явно подменить видимость функции и вызвать функцию определенного уровня. В листинге 4.4 приведена слегка измененная версия предыдущего примера с добавлением глобальной функции f(). Глобальная функция, по своим аргументам точно соответствующая вызову f(6), игнорируется в пользу функции B::f(double), замещающей глобальную функцию. Для вызова глобальной версии f используется оператор уточнения области видимости ::f(5).

Листинг 4.4. Вызов одноименных функций с разными областями видимости

```
void f(int j) { /* ... */ }

class A {                  // Базовый класс (суперкласс)
public:
    void f(int);
    // ..
};

class B: public A {        // Производный класс (субкласс)
public:
    void f(double);
    void g(int);
    // ...
};

void B::g(int k) {
    f(k);                  // B::f(double) с повышением
    A::f(k);               // A::f(int);
    this->A::f(k);         // A::f(int);
    ::f(k);                // ::f(int);
}

int main() {
    A *a;
    B *b;
    int i;
    // ...
    a->f(1);          // A::f(int)
    a->f(3.0);        // A::f(int) с понижением int(3.0)
    b->f(2);          // B::f(double) с повышением double(2)
    b->f(2.0);        // B::f(double)
    b->A::f(7.0);     // A::f(int) с понижением int(7.0)
    b->g(10);         // B::g(int)
}
```

Создание экземпляров путем наследования и управления доступом

В некоторых ситуациях базовый класс существует только как основа для построения производных классов. Например, приведенный ранее класс Telephone служит заготовкой для построения моделей «настоящих» телефонов со специализированным поведением: ISDNPhone — для цифровых, POTSPhone — для традиционных. В классе Telephone декларируются общие аспекты поведения всех телефонов, определяемые семантикой их функций классов. Этот класс также может содержать реализацию, общую для всех телефонов и не зависящую от разновидности (например, номер).

Используя средства ограничения доступа C++, класс может предоставить свой интерфейс как основу для интерфейса производных классов, одновременно предотвращая создание своих экземпляров. Рассмотрим следующий пример класса Telephone:

```
class Telephone {
public:
    ~Telephone();
    void ring();
    ...
protected:
    Telephone();
private:
    ...
};

class ISDNPhone: public Telephone {
    ...
};

class POTSPhone: public Telephone {
    ...
};
```

Обратите внимание: конструктор Telephone::Telephone() объявлен защищенным, а это предполагает, что он может быть «вызван» только в производном классе. Семантика подобного объявления такова, что объект класса Telephone не может быть объявлен в программе. Абстрактный телефон не может существовать как таковой; программа может объявлять и создавать экземпляры только конкретных, «настоящих» телефонов вроде ISDNPhone или POTSPhone. Свойства, общие для всех обобщенных «абстрактных» телефонов, являются частью «реального» телефона; конструктор базового класса неявно вызывается при создании экземпляра производного класса. Но экземпляр базового класса сам по себе создаваться не может.

В действительности подобная защита от создания объектов типа Telephone не идеальна: эти объекты могут создаваться функциями любого подкласса Telephone или функциями, дружественными для Telephone (если они существуют).

В разделе 5.4 описываются более мощные языковые конструкции, которые точнее и понятнее выражают намерение проектировщика создать абстрактный базовый класс.

4.3. Конструкторы и деструкторы

Конструкторы и деструкторы представляют собой специальные функции класса, которые автоматически инициализируют новые объекты и освобождают задействованные ресурсы в конце жизненного цикла объектов. Объект производного класса содержит переменные из нескольких разных классов; конструкторы и деструкторы этих классов вносят свой вклад в инициализацию и уничтожение объекта. В данном разделе рассматриваются конструкторы и деструкторы в контексте наследования, то есть порядок их выполнения и механизмы вызова параметризованных конструкторов базового класса.

Порядок выполнения конструкторов и деструкторов

При наследовании работа с объектом осуществляется функциями его класса, а также всех родительских классов, расположенных выше в иерархии. Таким образом, каждый объект содержит несколько конструкторов и деструкторов, а это означает, что логика инициализации и уничтожения объектов не сосредоточивается в одной функции, а имеет распределенную природу. Каждый конструктор, вызываемый для выполнения своей части инициализации, знает, какие конструкторы вызывались до него, а следовательно — какие аспекты состояния он может безопасно использовать.

Порядок выполнения конструкторов классов в иерархиях наследования четко определен. Компилятор автоматически организует выполнение конструкторов базовых классов и объектов, встроенных в создаваемый объект. Непосредственный код конструктора не завершает всей картины; по мере необходимости компилятор также генерирует собственный код для выделения памяти и вызова других конструкторов.

Динамическое выделение памяти производится при создании объекта с ключевым словом new. При этом вызывается либо операторная функция operator new этого класса (если она существует), либо глобальная операторная функция ::operator new. Без динамического создания объекта конструктор использует память, предоставленную конструируемой переменной (для автоматических и глобальных переменных), или выделенную заранее иным способом (например, конструктором производного класса).

Если конструктор принадлежит производному классу, он прежде всего должен вызвать конструктор своего базового класса. Это происходит автоматически, и программисту не приходится заботиться об этом; тем не менее программист может до определенной степени управлять вызовом конструктора (см. далее). Вызванный конструктор видит, что память уже выделена, и в свою очередь вызывает конструкторы своих базовых классов.

Далее конструктор инициализирует все переменные своего класса, являющиеся объектами классов, вызывая их конструкторы. Обычно такие объекты инициализируются своим конструктором по умолчанию (см. раздел 3.1), но вы можете повлиять на выбор при помощи списка инициализаторов, о котором речь пойдет далее.

После того как конструктор обеспечит инициализацию своего базового класса и внутренних объектов, выполняется тело конструктора, определенное программистом. В нем ранее инициализированным полям могут быть присвоены новые значения. Кроме того, тело конструктора должно инициализировать все переменные своего класса, относящиеся к встроенным типам (int, short, long, char и double), а также все указатели.

После завершения конструктора управление возвращается в точку его вызова. Конструктор базового класса возвращает управление конструктору производного класса, из которого он был вызван, и инициализация объекта производного класса продолжается с точки возврата. Последним выполняется код инициализации листового (то есть «самого производного») класса в иерархии наследования.

При компиляции конструктора компилятор обычно включает в него скрытый код для управления памятью, диспетчеризации функций класса, вызова конструкторов базовых классов, инициализации переменных класса, которые также являются объектами классов. Обычно пользователи просто игнорируют присутствие этого кода, но так как его структура зависит от установленного экземпляра и версии, не стоит полагаться на его конкретные особенности.

Деструкторы выполняются в порядке, противоположном порядку вызова конструкторов: сначала вызывается деструктор класса объекта, затем деструкторы нестатических внутренних объектов, в последнюю очередь — деструкторы базовых классов. Если класс операнда (листового класса в иерархии наследования) содержит (или наследует) оператор delete, то вместо стандартной системной версии будет вызвана специализированная версия. Освобождение памяти, занимаемой объектом-операндом, выполняется в последнюю очередь перед возвратом из деструктора.

Передача параметров конструкторам базовых классов и внутренних объектов

Вернемся к синтаксису конструктора Imaginary::Imaginary(double) на с. 100. Обратите внимание: этот конструктор имеет пустое тело и выполняется только ради неких побочных эффектов. Если бы пользователь написал следующее простое определение, компилятор организовал бы вызов конструктора базового класса по умолчанию Complex::Complex() в самом начале вызова Imaginary::Imaginary (перед присваиванием):

```
Imaginary::Imaginary(double d) {
    ipart = d;
}
```

Благодаря этому обстоятельству класс Imaginary знает, что класс Complex полностью инициализирован, и его данные могут использоваться для построения класса Imaginary.

Теоретически формальные параметры конструктора Imaginary могут влиять на работу конструкторов его базовых классов. У программиста имеется возможность приказать компилятору вызвать конкретный конструктор базового класса. Так, в примере на с. 100 программист явно указал, что перед вызовом Imaginary::Imaginary(double) должен быть вызван конструктор Complex(0,i).

Следует отметить, что конструктор Complex::Complex(double, double) (см. листинг 4.1) определяет инициализацию переменных rpart и ipart. Следующие два конструктора работают практически одинаково:

```
Imaginary::Imaginary(double a,
                     double b)
{
    rpart = a;
    ipart = b
}
```

```
Imaginary::Imaginary(double a,
                     double b)
   : rpart(a), ipart(b)
{
      /* Пусто */
}
```

Прямой вызов конструктора, как в правом варианте, позволяет избежать лишней операции копирования. Эффект становится более очевидным в примере класса PathName, где вместо простых машинных слов в новый объект копируется последовательность символов:

```
PathName::PathName()
{
    rep = String("");
    baseNameRep = String("");
}
```

```
PathName::PathName()
  : baseNameRep(""), String()
      { /* Пусто */ }
```

Кроме того, перед присваиванием какого-либо значения объект baseNameRep необходимо инициализировать; в левом примере конструктор String::String() вызывается для baseNameRep после инициализации базового класса String, но до выполнения команд конструктора PathName. Это означает, что переменная rep тоже инициализируется дважды — при неявном вызове конструктора базового класса и при явном вызове конструктора PathName.

В конструкторе PathName(const PathName&) встречается аналогичный синтаксис: конструктор базового класса String::String(const String&) вызывается с параметром p. Он выполняется до того, как будет выполнен какой-либо код конструктора PathName.

Другой пример передачи параметров конструктором при вызове конструктора базового класса встречается в объявлении класса Square в примере Shapes из приложения Б. Квадрат (Square) является частным случаем прямоугольника (Rect). Единственное различие между ними связано со способом их конструирования: квадрату достаточно одного параметра, определяющего длину стороны. Проблема решается в конструкторе Square простым вызовом конструктора Rect:

```
class Square: public Rect {
public:
    Square(Coordinate ctr, long x): Rect(ctr, x, x) { }
};
```

Новая абстракция, обладающая всеми возможностями и аспектами поведения Rect, создается всего в четырех строках программного кода.

4.4. Преобразование указателей на классы

Возможности наследования могут использоваться для получения архитектурных преимуществ, упрощающих проектирование и сопровождение больших систем. Мощь наследования во многом обусловлена способностью интерпретации объектов классов, входящих в иерархию наследования, как взаимозаменяемых абстракций. Любой объект производного класса можно интерпретировать как экземпляр класса, находящегося в корне иерархии. Данное свойство называется *полиморфизмом* (иначе — «многообразие форм»); разные формы объектов рассматриваются так, словно они принадлежат одному классу. Это означает, что код приложения может игнорировать подробности реализации конкретных типов, например телефонов POTSPhone и ISDNPhone, и обходиться с ними так, словно все они относятся к классу Telephone (а если в отдельных случаях потребуется использовать специфику конкретной разновидности телефона, код приложения может «спуститься» на нижний уровень абстракции и сделать все необходимое). В этом и в следующем разделах будет заложена основа для более элегантного и мощного подхода, представленного в главе 5.

Полиморфизм базируется на механизме преобразования указателей на объекты классов, связанных в иерархию наследования. Дело в том, что большинство программ C++ работает с объектами через указатели, поскольку объекты часто создаются в куче. Указатели C++ обладают двумя преимуществами, не присущими обычным переменным. Во-первых, указатель на базовый класс может ссылаться на объект любого класса, находящегося на более низком уровне иерархии. Во-вторых, функции класса, вызываемые через указатель, могут выбираться не во время компиляции, а во время выполнения программы (для чего они должны быть объявлены *виртуальными*). Эти два свойства имеют важнейшее значение для объектно-ориентированного программирования (см. главу 5). Если переменная содержит ссылку на объект класса, то функции класса, вызываемые через эту переменную, тоже выбираются полиморфно; тем не менее, на практике чаще применяются указатели. Прямой вызов функций для объекта в точечной записи (например, officePhone.ring) такой возможности не дает.

В общем случае указатель на объект некоторого класса C может использоваться для хранения адреса объекта любого из его производных классов — прямого потомка, «внука» или любого производного класса *n*-го поколения. Обратное возможно *не всегда*. Например, указатель Telephone* может ссылаться на объект Telephone, ISDNPhone и даже POTSPhone. Класс POTSPhone может рассматриваться как частный случай класса Telephone, потому что он обладает всеми возможностями последнего. Но обратная интерпретация недопустима, поскольку не все аспекты поведения POTSPhone поддерживаются классом Telephone; они не являются общими для всех телефонов. Ошибка может быть обнаружена на стадии компиляции.

Механизм явного преобразования типов *позволяет* преобразовать указатель на базовый класс в указатель на производный класс. Применяя этот механизм, вы сообщаете компилятору, что у вас имеется информация о типе объекта, которой не обладает компилятор. Но подобные претензии часто опасны, особенно если они основаны на анализе разрозненных фрагментов кода.

Если функции базового класса в равной степени применимы к объектам любых производных классов, то указатель на базовый класс можно использовать для вызова функций классов любых объектов иерархии, даже если конкретный тип объекта неизвестен на стадии компиляции. Это удобно во многих ситуациях, например, функции, не входящие в иерархию Telephone, могут применять общие операции «телефонных» классов к объектам телефонов, не зная, с какой конкретной разновидностью телефона они работают в данный момент. В следующем примере функция ringPhone получает при вызове вектор разнородных указателей на Telephone:

```
void ringPhones(Telephone *phoneArray[])
{
    for (Telephone *p = phoneArray; p; p++) {
        p->ring();
    }
}

int main()
{
    Telephone *phoneArray[10];
    int i = 0;
    phoneArray[i++] = new POTSPhone("5384");
    phoneArray[i++] = new POTSPhone("5010");
    phoneArray[i++] = new ISDNPhone("0");
    phoneArray[i++] = new POTSPhone("5383");
    phoneArray[i++] = 0;
    ringPhones(phoneArray);
    ...
}
```

Такое решение работает: для каждого объекта в массиве phoneArray будет вызвана функция Telephone::ring. С другой стороны, это решение не позволяет задействовать дополнительные возможности данных или функций, специфических для производных классов, потому что ringPhones знает лишь то, что известно о базовом классе Telephone на стадии компиляции. Для наглядности добавим в иерархию класс OperatorPhone со специализированной операцией ring, а затем выполним слегка измененную программу main:

```
class OperatorPhone: public ISDNPhone {
public:
    OperatorPhone();
```

```
    OperatorPhone(OperatorPhone&);
    ~OperatorPhone();
    void ring(); // Специальная операция ring
                 // для класса OperatorPhone
};

int main()
{
    Telephone *phoneArray[10];
    int i = 0;
    phoneArray[i++] = new POTSPhone("5384");
    phoneArray[i++] = new POTSPhone("5010");

    // Имеет специализированную функцию ring:
    phoneArray[i++] = new OperatorPhone("0");

    phoneArray[i++] = new POTSPhone("5383");
    phoneArray[i++] = 0;
    ...
}
```

Так как функции ringPhones известно лишь то, что элементы вектора указывают на объекты Telephone, она вызывает для каждого объекта функцию Telephone::ring. Хотя мы хотели, чтобы для объекта OperatorPhone вызывалась специальная функция ring, но у компилятора для этого было недостаточно информации. В главе 5 будет показано, как компилятор может автоматически передавать информацию о контексте вызова посредством виртуальных функций. Далее рассматривается другое («неполноценное») решение, основанное на включении в каждый объект *поля селектора типа*.

4.5. Селектор типа

Селектором типа называется переменная базового класса, значение которой определяет фактический тип содержащего ее объекта. Механизм *виртуальных функций* (см. главу 5) считается более предпочтительным решением, но селекторы типов задействованы в некоторых нетривиальных идиомах. Мы рассмотрим их здесь как для использования в будущем, так и для логического моделирования механизма, автоматизируемого при помощи виртуальных функций.

В листинге 4.5 приведена очередная версия класса Telephone с полем селектора типа, идентифицирующим разные объекты телефонов. Селектор присутствует в объектах всех разновидностей телефонов, поэтому мы всегда можем обратиться к объекту, на который указывает Telephone*, и узнать его тип. Обратите внимание: конструкторы всех классов, производных от Telephone, заносят информацию о себе в новое поле.

Листинг 4.5. Иерархия классов Telephone с селектором типа

```
class Telephone {
public:
    enum PhoneType { POTS, ISDN, OPERATOR, OTHER }
      phoneType()
        { return phoneTypeVal; }
    void ring();
    Bool isOnHook();
    Bool isTalking();
    Bool isDialing();
    DigitString collectDigits();
    LineNumber extension();  // Не подменяется
    ~Telephone();
protected:
    LineNumber extensionData;
    PhoneType phoneTypeVal;
    Telephone();
};

class POTSPhone: public Telephone {
public:
    Bool runDiagnostics();
    POTSPhone(): phoneTypeVal(POTS) ... {
        ...
    }
    POTSPhone(POTSPhone &p): phoneTypeVal(POTS) ... {
        ...
    }
    ~POTSPhone();
private:
    Frame frameNumberVal;
    Rack rackNumberVal;
    Pair pairVal;
};

class ISDNPhone: public Telephone {
public:
    ISDNPhone(): phoneTypeVal(ISDN) ...   { ... }
    ISDNPhone(ISDNPhone &p): phoneTypeVal(ISDN) ... {
        ...
    }
    ~ISDNPhone();
    void sendBPacket();  // Отправка пакета по каналу B
    void sendDPacket();  // Отправка пакета по каналу D
private:
    Channel b1, b2, d;
};
```

Теперь функцию ringPhones можно записать так:

```
void ringPhones(Telephone *phoneArray[])
{
    for (Telephone *p = phoneArray; p; p++) {
        switch (p->PhoneType()) {
        case Telephone::POTS:
            ((POTSPhone *)p)->ring(); break;
        case Telephone::ISDN:
            ((ISDNPhone *)p)->ring(); break;
        case Telephone::OPERATOR:
            ((OperatorPhone *)p)->ring(); break;
        case Telephone::OTHER:
        default:
            error(...);
        }
    }
}
```

После этого для любой фактической разновидности телефона будет вызвана правильная операция ring. В функцию даже можно встроить некое подобие интеллекта: в первых двух случаях достаточно вызова p->ring(), поскольку ни POTSPhone, ни в ISDNPhone не имеют специализированной операции ring. Но и в этом виде функция успешно работает; более того, даже если в класс POTSPhone или ISDNPhone будет включена своя операция ring, код все равно останется работоспособным.

В этом варианте функция ringPhones выбирает для объекта функцию на основании поля, содержимое которого проверяется на стадии выполнения. Но компилятор «знает» класс объекта при его создании и может сгенерировать код, связывающий информацию о типе с каждым объектом. Если вызвать для такого «расширенного» объекта функцию ring, компилятор на основании информации о типе объекта сможет автоматически выбрать вызываемую функцию ring. На этом принципе работают виртуальные функции C++ (эта тема главы 5). Обычно виртуальные функции считаются более предпочтительным решением, нежели селекторы типов. В сообществе C++ не принято пользоваться селекторами типов, поэтому предыдущий фрагмент приводится только для демонстрационных целей.

Селекторы типов используются в идиомах, которые будут представлены позднее, в том числе при имитации мультиметодов (выборе функции на стадии выполнения в зависимости от типа нескольких параметров) в разделе 9.7. Другой пример имеется среди примеров функторов в разделе 5.6. Идиома расширяемого поля типа довольно подробно описана в [1].

Рассмотрим основные недостатки решения с селектором типа. При выборе операций на уровне исходного текста, как в команде switch в ringPhones, развитие программы быстро становится утомительным и неудобным. Возможно, при включении операции ring в класс POTSPhone вам даже не придется переписывать программу, но в большинстве реализаций C++ ее необходимо перекомпилировать. Также представьте, к чему приведет создание нового класса телефона: вам

придется изменять функцию ringPhones, а также *все аналогичные* функции. В перечисляемый тип базового класса включается новый элемент, а разработчик нового класса должен помнить о необходимости инициализировать поле типа во всех конструкторах. Весь код, связанный с изменениями в программе, придется перекомпилировать и протестировать заново.

Упражнения

1. Охарактеризуйте возможные применения закрытого наследования.
2. Предложите способы имитации абстракции данных и наследования в C.
3. Рассмотрите возможные применения идиомы «манипулятор/тело» для управления памятью и обеспечения необходимой гибкости на стадии выполнения (то есть ситуации, в которых основной класс использует внутренний объект вспомогательного класса, содержащий большую часть подробностей реализации). Иногда это делается для того, чтобы изолировать приложение от изменений в реализации, например, от изменений в структуре данных. Общая функциональность объекта разделяется между манипулятором и телом по аналогии с разделением функциональности между базовым и производным классами при наследовании.

   Наследование может быть реализовано двумя способами: от внешнего класса манипулятора или от внутреннего класса тела; в обоих случаях используется один и тот же конверт. Проанализируйте оба варианта, их достоинства и недостатки. Существуют ли другие варианты?
4. Откомпилируйте следующую программу:

```
#include <iostream.h>

class ZooAnimal {
protected:
    int zooLoc;
public:
    int cnt;
};

class Bear:public ZooAnimal {
public:
    int find(ZooAnimal*);
};

int
Bear::find( ZooAnimal *pz ) {
    if (cnt) cout << zooLoc;
    if (pz->cnt) cout << pz->zooLoc;
}
```

Что происходит при компиляции? Какой из этого можно сделать вывод относительно модели защиты C++ применительно к объектам и классам? Можно ли обойти эту защиту? И если можно, то как? (См. [2].)

5. Определите классы для системы управления лифтами, в которой одновременно работают несколько лифтов. Какие операции должны быть определены для классов этой системы? Все ли классы вашей системы представляют «общие сущности»? Почему?
6. Проанализируйте (и откомпилируйте) следующую программу:

```
class A {
public:
    A(int i = 0) { a = i; }
    A& operator=(A& x) { a = x.a; return *this; }
protected:
    int a;
};

class B: public A {
public:
    B(int i = 0): A(i) { b = i; }
    B& operator=(B& x) { b = x.b; return *this; }
private:
    int b;
};

int main() {
    B b1, b2(2);
    b1 = b2;
}
```

Какое значение будет присвоено переменной `b1.a`? Какой из этого нужно сделать вывод о реализации присваивания в условиях наследования? Как решить проблему в данном случае?

Литература

1. Stroustrup, B. «The C++ Programming Language», 2nd ed. Reading, Mass.: Addison-Wesley, 1991, ch.13.
2. Lippman, Stanley B. «A C++ Primer», Reading, Mass.: Addison-Wesley, 1989.

Глава 5
Объектно-ориентированное программирование

Объектно-ориентированным называется стиль программирования, отталкивающийся от инкапсуляции и абстрактных типов данных. В языке программирования и среде разработки абстрактные типы данных представлены *системой контроля типов*. Эта система выполняет проверку типов, следя за тем, чтобы для выражения a+b, в котором оба слагаемых объявлены с типом double, была сгенерирована машинная команда суммирования с двойной точностью. Языки с системами контроля типов изначально создавались для генерирования кода, более эффективного по сравнению с кодом нетипизованных программ, но «безопасность типов» тоже была достаточно важным фактором.

Большинство систем контроля типов реализуется в компиляторе и сопровождающих утилитах. Все, о чем уже рассказывалось, относится к стадии компиляции и не проявляется напрямую во время работы программы. Однако компилятор может выполнить преобразование типа или оптимизацию, из-за которой некоторые аспекты системы контроля типов проявятся во время выполнения. Рассмотрим простую последовательность команд:

```
int i;
double d;
...
d = i;
```

Компилятор должен сгенерировать код преобразования целого числа в вещественное с двойной точностью *во время выполнения*. Сходство между типами int и double делает такое преобразование возможным и даже естественным; до определенной степени эти типы «совместимы». В языках вроде C все типы определяются заранее, компилятор знает все операции, применимые к этим типам, может интерпретировать их в текущем контексте и генерировать код выполнения операций или преобразования между типами.

Но в C++ ситуация из-за классов осложняется. Программист может определять новые типы (например, конкретные типы данных, о которых рассказывалось в главе 3) и отношения между ними — такие как отношения наследования (см. главу 4). C++ позволяет конструировать системы, основанные на иерархи-

ческом принципе, — абстракции, естественной для человеческого разума. Эти иерархии и типы выходят за пределы того, что известно компилятору, поэтому, чтобы обеспечить нужную степень «совместимости», естественную для сходных классов, пользователь должен предоставить компилятору дополнительную информацию.

Количество иерархий классов в большой системе должно быть разумным, даже если этого нельзя сказать о количестве отдельных классов. Чтобы в полной мере использовать мощь иерархии, программист должен иметь возможность эффективно работать на верхних уровнях иерархии, не опускаясь к производным классам. Этот принцип распространяется как на проектирование, так и на реализацию. Программа, написанная для высокоуровневой абстракции SignedQuantity (величина со знаком), на стадии выполнения может работать со значениями классов Complex (комплексное число) или InfinitePrecision (число с неограниченной точностью), если два последних класса входят в иерархию SignedQuantity. Если программист использует следующую запись, то операция сложения для соответствующего класса должна выбираться на основании контекста во время выполнения:

```
void afunction(SignedQuantity &c, SignedQuantity &d) {
    ... c+d ...
}
```

Такой подход выходит за рамки системы типов стадии компиляции, но ради эффективности и безопасности типов средства времени выполнения «принимают эстафету» у средств времени компиляции. *В этом заключается сущность объектно-ориентированного программирования.* Появляется новая система контроля типов, основанная на инкапсуляции абстрактных типов данных, организованных в иерархии, с поддержкой типов на стадии выполнения.

C++ удерживает равновесие между системой контроля типов времени компиляции, обеспечивающей эффективность кода, и системой времени выполнения, обеспечивающей гибкость и «совместимость» программных компонентов. Язык предоставляет основные средства для объектно-ориентированного программирования, но сохраняет многие конструкции для «культурной» совместимости с C.

Объектно-ориентированное программирование в каком-то смысле представляет собой «трюк», основанный на косвенных обращениях; хорошие программисты пользовались этим трюком в течение многих лет. Особого внимания заслуживают два обстоятельства: важность поддержки этой методики на уровне языка и еще более важная роль тех принципов проектирования, которые стоят за ними.

Во-первых, C++ выводит эту методику из области трюков и делает ее полноценной составляющей языка, благодаря чему упрощается чтение, написание и сопровождение программ. Конечно, такое развитие можно только приветствовать, но для эффективного применения подобных приемов программист должен хорошо владеть несколькими каноническими формами и языковыми идиомами, выходящими за пределы базового синтаксиса. C++ предоставляет дополнительный логический уровень при вызове функций; программисты, использующие это

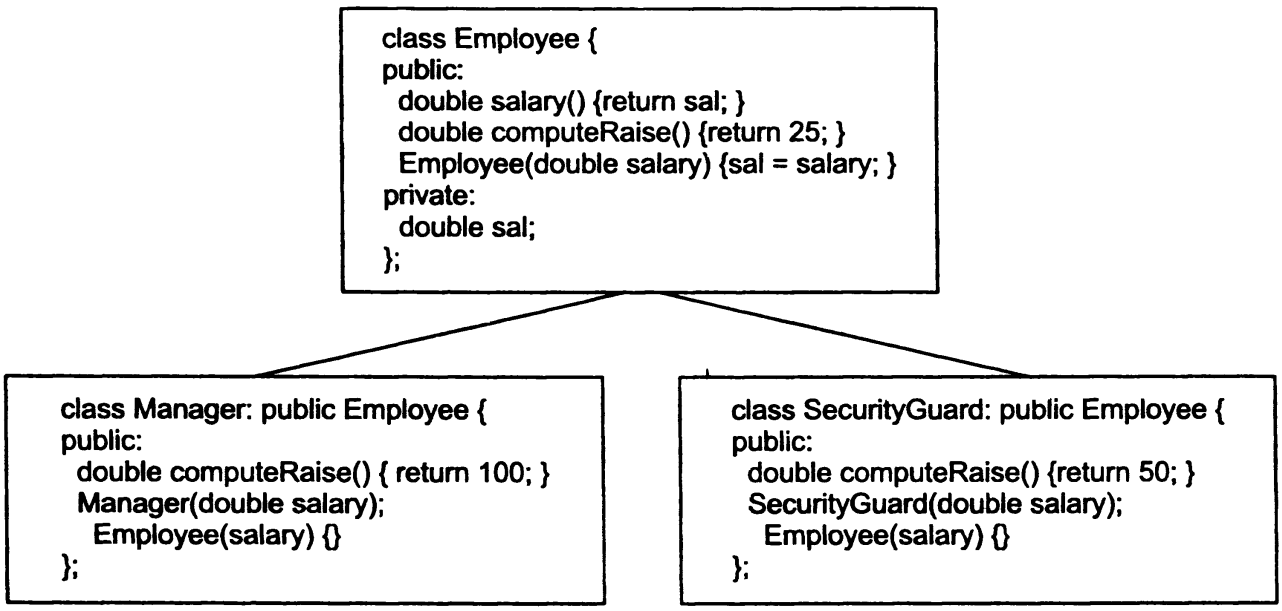
обстоятельство для повышения гибкости или для применения общих абстракций в другом измерении, также нуждаются в дополнительных идиомах. Такие идиомы описаны в этой главе. Они занимают одно из важнейших мест в книге.

Во-вторых, важны не конкретные приемы, а принципы проектирования, которые в них воплощаются. Главным условием написания хорошей программы является не столько оптимальное применение синтаксиса, сколько квалифицированные глубокие мыслительные процессы ее создателей. В этом контексте данная глава закладывает основу для приемов проектирования, рассматриваемых в главе 6, где описаны центральные концепции всей книги. Но и программист, и проектировщик должны помнить, что приемы проектирования наиболее эффективно применяются в контексте описываемых далее идиом.

5.1. Идентификация типов на стадии выполнения и виртуальные функции

Механизм наследования, описанный в предыдущей главе, позволяет программисту по-новому организовать программу, ориентированную на многократное использование кода. Код, общий для нескольких классов, выделяется в базовый класс, а производные классы расширяют поведение базового класса для создания более специализированных абстракций. Переменная, объявленная как указатель на базовый класс, может применяться для обращения к объекту любого производного класса и вызова общих (принадлежащих базовому классу) функций этого объекта. Таким образом становится возможным *полиморфизм*, когда один указатель годится для работы с объектами нескольких разных форм. Программист объявляет указатель, который может ссылаться на любой объект из набора классов в иерархии наследования. Через этот указатель программа работает одинаково со всеми объектами, не зная их фактического класса. Каждый объект интерпретируется в контексте своего обобщенного *типа*, базового класса иерархии.

Наследование позволяет применять общий набор функций к объектам, классы которых являются «соседями» и «дальними родственниками» в дереве наследования. Но если для поддержки полиморфизма используется только наследование, то он распространяется исключительно на функции с *общей реализацией*. Если производный класс содержит собственную версию реализации функции базового класса, то одно лишь наследование не приведет к автоматическому выбору версии производного класса при вызове через переменную, объявленную с типом базового класса. В соответствии с заданным типом указатель знает только о базовом классе и понятия не имеет не то что о структуре, но и о самом факте существования производных классов. Для примера рассмотрим иерархию с базовым классом Employee, изображенную на рис. 5.1. Если программа «понимает» общий класс Employee, но не знает о существовании производных классов SecurityGuard и Manager, она использует *стандартную* версию функции computeRaise для всех объектов в иерархии Employee. У нее просто не хватает информации для более разумного поведения.




```
Manager *boss1 = new Manager(2000);        // можно
double boss1Salary = boss1 -> salary();    // результат 2000
Employee*boss2 = new Manager(2300);        // можно
double boss2Salary = boss2 -> salary();    // возвращает 2300

double boss1Raise = boss1 -> computeRaise();    // результат 100
double boss2Raise = boss2 -> computeRaise();    // результат 25
```

Если тип указателя точно соответствует типу объекта, возвращается правильный результат. Но если для ссылок на объект производного класса используется указатель на обобщенный базовый класс, вызываются только функции базового класса; другие одноименные функции в классе объекта невидимы для указателя.

Рис. 5.1. Ограниченный полиморфизм с использованием наследования

Базовый класс должен иметь выразительный интерфейс, который бы характеризовал *поведение* представляемой им абстракции в терминах имен, типов параметров и возвращаемых типов его функций. Открытые функции базового класса либо становятся частью интерфейса производного класса, либо замещаются в производном классе более подходящей функцией с тем же логическим поведением. Все аспекты поведения базового класса распространяются на объекты производных классов, поэтому базовый класс определяет поведение классов, производных от него.

Мы использовали схему наследования в главе 4 при определении разных версий функции ring для разных телефонов. Производный класс может *подменить* реализацию функции, заданную в базовом классе, иначе говоря, он предоставляет собственную реализацию функции, входящей в открытый интерфейс базового класса. Замена функции базового класса одноименной функцией производного класса позволяет оптимизировать реализацию с сохранением основополагающего смысла указанной функции. Применение функции ring к телефонам разных классов всегда имеет одинаковый смысл, хотя подробности реализации меняются в зависимости от типа телефона.

Взаимозаменяемость этих объектов должна быть как можно более прозрачной на уровне компилятора. Компилятор знает, что некоторая функция входит в интерфейс базового класса, но мы хотим, чтобы компилятор автоматически вызывал версию этой функции из соответствующего производного класса — того, к которому относился объект при создании. Выбор функции должен определяться *классом объекта*, а не объявлением указателя, используемого для ссылки на него. В результате программист может считать объекты *всех* производных классов эквивалентными в контексте базового класса, а в программе произойдет именно то, что требовалось: для объектов Manager будет вызываться своя версия функции computeRaise, хотя на стадии компиляции при вызове этой функции будет сгенерирован код, рассчитанный на обобщенный класс Employee.

Этот механизм называется *идентификацией операций на стадии выполнения*. В сочетании с наследованием он реализует вид полиморфизма, предоставляющий в распоряжение программиста и проектировщика гибкий инструментарий для создания высокоуровневых программных компонентов. Стиль программирования, в котором этот механизм логически последовательно применяется к конкретным типам данных, называется *объектно-ориентированным программированием*; эта методика является наиболее прямолинейным способом поддержки объектно-ориентированного проектирования в C++.

Идентификация операций на стадии выполнения применима только к указателям и ссылкам на объекты классов. Почему? Потому что только переменные, объявленные как указатели и ссылки, могут работать как с объектами своего класса, так и с объектами других производных классов. В частности, это можно объяснить тем, что информация о типе теряется только при использовании указателя: чтобы напрямую вызвать функцию класса для экземпляра, вы должны располагать объявлением класса объекта, и привязка вызова будет осуществляться на стадии компиляции. Но если указатель на объект находится далеко от точки создания объекта, пользователю не обязательно передавать весь «лишний груз» подробностей типа в любую часть программы, имеющую дело с объектом.

Давайте вернемся к примеру класса Telephone из главы 4 и реализуем его *правильно*, используя объектную парадигму. На самом деле изменений будет относительно немного. Для удобства оригинал приведен в листинге 5.1.

Листинг 5.1. Иерархия с классом Telephone

```
class Telephone {
public:
    void ring();
    Bool isOnHook(), isTalking(), isDialing();
    DigitString collectDigits();
    LineNumber extension();
    ~Telephone();
protected:
    Telephone();
    LineNumber extensionData;
};
```

```
class POTSPhone: public Telephone {
public:
    Bool runDiagnostics();
    POTSPhone();
    POTSPhone(POTSPhone&);
    ~POTSPhone();
private:
    Frame frameNumberVal;
    Rack rackNumberVal;
    Pair pairVal;
};

class ISDNPhone: public Telephone {
public:
    ISDNPhone();
    ISDNPhone(ISDNPhone&);
    ~ISDNPhone();
    void sendBPacket(), sendDPacket();
private:
    Channel b1, b2, d;
};

class OperatorPhone: public ISDNPhone {
public:
    OperatorPhone();
    OperatorPhone(OperatorPhone&);
    ~OperatorPhone();
    void ring();
};

class PrincessPhone: public POTSPhone {
    ...
};
```

Обратите внимание: используются версии классов без встроенного селектора типа, представленного в главе 4. Теперь он стал лишним — компилятор автоматически обеспечивает идентификацию типов. Некоторые функции базового класса объявлены виртуальными; производные классы могут содержать специализированные версии этих функций. При вызове виртуальных функций компилятор генерирует специальный код, чтобы нужная версия функции выбиралась во время выполнения программы в зависимости от типа объекта. Каждый раз, когда в программе создается объект класса, содержащего виртуальные функции, компилятор сохраняет в нем «поле типа». (Детали реализации зависят от компилятора, но на концептуальном уровне все сводится к применению селектора типа, упоминавшегося в разделе 4.5. Поле используется компилятором, и язык не предоставляет программисту средств для работы с его содержимым.) Более того, компилятор работает с этим полем более эффективно, чем при ручном кодировании, и с меньшими последствиями при модификации. Например, включение нового класса в программу с реализованным вручную полем типа потребует

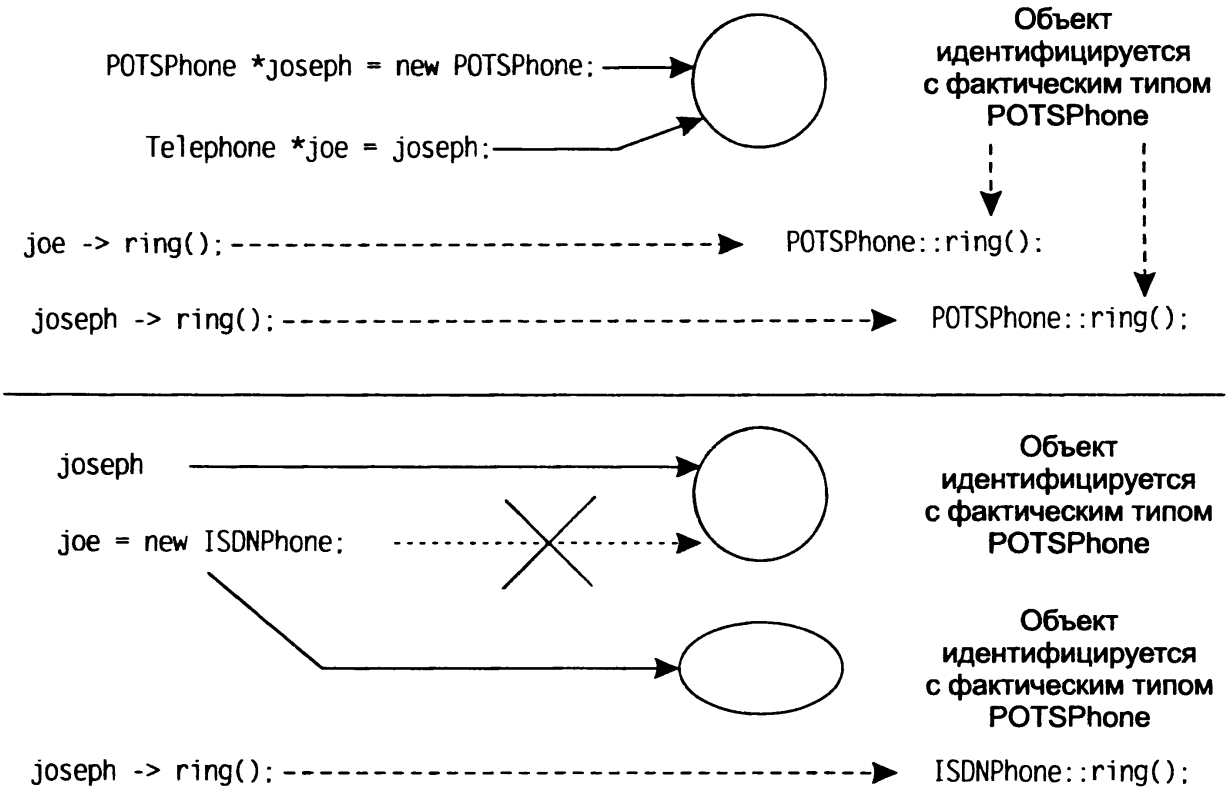
глобального редактирования и перекомпиляции. При наличии виртуальных функций в большинстве сред программирования C++ достаточно перекомпилировать исходный код, относящийся непосредственно к новому классу. *Вручную вставлять поля идентификации типов в классы не рекомендуется*, поскольку это затрудняет дальнейшее развитие программы.

Превращение класса Telephone в объектно-ориентированный начинается с объявления виртуальных функций. Объявляя функции базового класса виртуальными, мы сообщаем компилятору, что их поведение и реализация определяются фактическим классом объекта, для которого эти функции вызываются. Естественно, базовый класс не может предвидеть потребностей всех производных классов, которые могут быть созданы на протяжении жизненного цикла программы. Объявление виртуальных функций позволяет предоставить новые реализации этих функций в классах, производных от того базового класса, в котором функции объявлены виртуальными. Таким образом, виртуальная функция может существовать в множестве разных версий — в крайнем случае каждый класс в иерархии наследования данного базового класса получит собственную версию функции.

В базовом классе обычно определяется тело виртуальной функции, которое по умолчанию наследуется производным классом, если последний не подменяет его реализацию. Если производный класс не переопределяет функцию, то он ведет себя так, словно функция базового класса является частью его интерфейса. Если же функция подменяется в одном или нескольких производных классах, то версия производного класса замещает версию базового класса. Виртуальные функции отличаются от остальных функций тем, что версия производного класса доминирует *всегда*, даже если объект производного класса используется в контексте, в котором ожидается объект базового класса.

При использовании виртуальных функций компилятор организует необходимую поддержку на стадии выполнения, чтобы при вызове функции, объявленной виртуальной в базовом классе, вызывалась функция соответствующего производного класса. Операция должна соответствовать типу объекта, к которому она применяется, а *не* типу указателя, использованного для вызова (рис. 5.2). Вот как выглядит базовый класс с объявлениями виртуальных функций:

```
class Telephone {
public:
    virtual void ring();
    virtual Bool isOnHook();
    virtual Bool isTalking();
    virtual Bool isDialing();
    virtual DigitString collectDigits();
    virtual ~Telephone();
    LineNumber extension();
protected:
    LineNumber extensionData;
    Telephone();
};
```



Вызываемая версия виртуальной функции определяется фактическим типом объекта, а не типом указателя

Рис. 5.2. Идентификация виртуальных функций на стадии выполнения

Все изменения в базовом классе сводятся к добавлению ключевых слов virtual. Ключевое слово virtual в объявлении класса должно находиться в *объявлении*, а не в *определении* функции.

В архитектуру базового класса могут входить функции, которые программист не планирует подменять в производном классе. Например, функции isOnHook, isTalking и isDialing задуманы как обычные функции базового класса без поддержки производных классов. Они не являются «временными заполнителями», а предоставляют стандартное поведение, подходящее для всех производных классов. Тем не менее, опыт показывает, что в процессе эволюции системы функции базового класса часто приходится подменять в производных классах, поэтому для страховки лучше сразу объявить их виртуальными. Даже если их особые свойства никогда не будут востребованы, лишние затраты на стадии выполнения пренебрежимо малы — как правило, при объявлении виртуальной функции затраты на ее вызов возрастают на 15 %. Если вызовы функций составляют 10 % от общего времени выполнения программы, массовое объявление виртуальных функций снижает быстродействие на 2 %. Впрочем, реализацию с виртуальными функциями необходимо сравнить с альтернативными реализациями. Скорее всего, в ситуации, когда в программах C++ реализация выбирается посредством механизма виртуальных функций, в программе на C будут использоваться команды switch и if. Некоторые приложения (например, интерпретатор Scheme, написанный на C++ [1]), показали, что виртуальные функции работают *быстрее* конструкций switch/case.

Теперь мы можем переписать приложение, используя новые средства. Напомним, как выглядела прежняя версия функции ringPhones:

```
void ringPhones(Telephone *phoneArray[])
{
    for (Telephone *p = phoneArray: p; p++) {
        switch (p->PhoneType()) {
        case Telephone::POTS:
            ((POTSPhone *)p)->ring(); break;
        case Telephone::ISDN:
            ((ISDNPhone *)p)->ring(); break;
        case Telephone::OPERATOR:
            ((OperatorPhone *)p)->ring(); break;
        case Telephone::OTHER:
        default:
            error(...);
        }
    }
}
```

Новая версия гораздо проще:

```
void ringPhones(Telephone *phoneArray[])
{
    for (Telephone *p = phoneArray: p; p++) p->ring();
}
```

Если указатель p ссылается на объект OperatorPhone, то конструкция p->ring() вызовет функцию OperatorPhone::ring(). Если он ссылается на любой другой объект в иерархии Telephone, то вместо нее будет вызвана функция Telephone::ring(), поскольку в других классах сохраняется реализация этой функции по умолчанию.

Однако этот механизм работает только в том случае, если функция объявлена виртуальной в базовом классе. Рассмотрим следующий фрагмент:

```
Telephone *digitalPhone = new ISDNPhone;
...
digitalPhone->sendPacket();   // Неверно, функция не найдена
```

Этот пример не работает, поскольку функция sendPacket не объявлена в базовом классе Telephone. Даже если объявить ее виртуальной в своем классе, это не поможет (хотя и обеспечит виртуальное поведение в классах, производных от ISDNPhone). Чтобы операции идентифицировались на стадии выполнения, функция должна быть виртуальной. Функция класса может стать виртуальной одним из двух способов. Во-первых, функция считается виртуальной, если она была объявлена таковой в классе, использованном для объявления указателя, через который вызывается операция. Во-вторых, функция класса также считается виртуальной, если выше в иерархии наследования присутствует одноименная функция с такими же типами параметров и возвращаемого значения (рис. 5.3). Виртуальность функции наследуется производными классами до самого дна иерархии наследования.

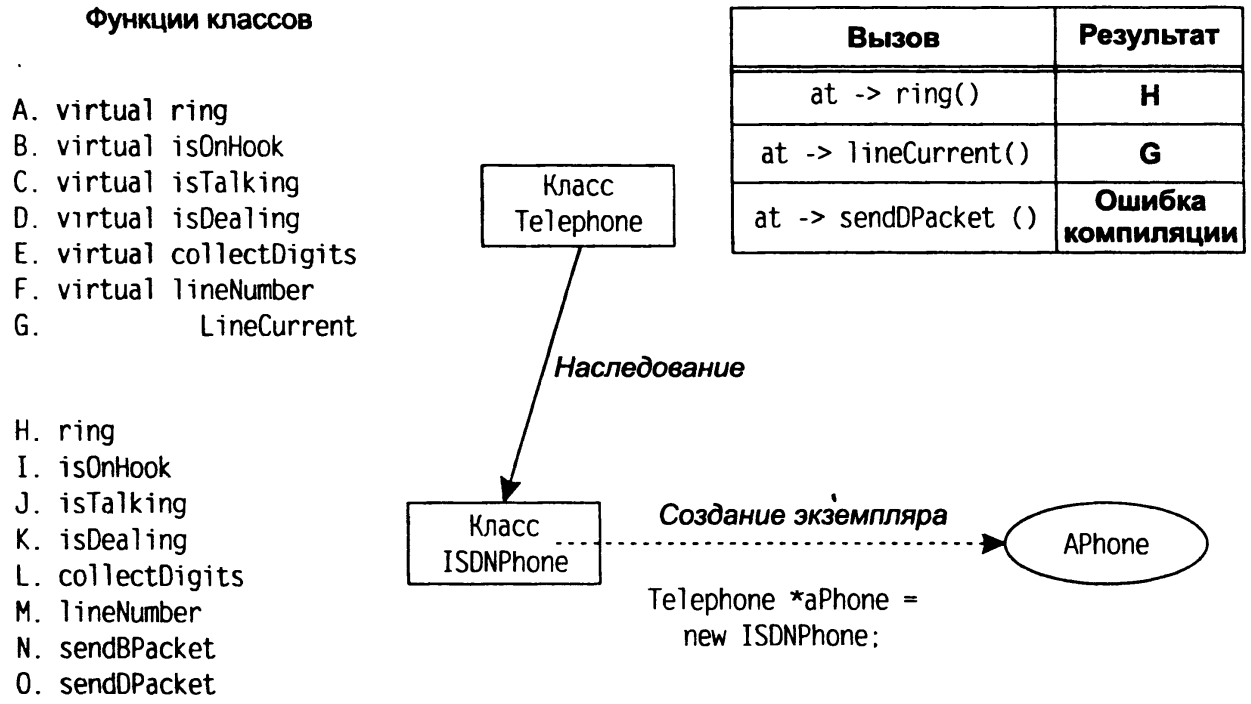


Рис. 5.3. Наследование «виртуальности» функций

Ключевое слово virtual можно рассматривать и иначе: так, словно оно объявляет не функцию класса, а *имя* функции класса. Объявления обычных (не виртуальных) функций создают функции, тело которых связывается с именем на стадии компиляции или компоновки. К виртуальным функциям это не относится. Связывание имени виртуальной функций с телом производится при ее вызове на стадии выполнения. В этом смысле виртуальные функции аналогичны селекторам методов в языке SmallTalk, а тела функций — самим методам.

5.2. Взаимодействие деструкторов и виртуальные деструкторы

Обратите внимание: деструктор Telephone::~Telephone тоже объявлен виртуальным. На первый взгляд это выглядит несколько странно, потому что обычно мы не собираемся *вызывать* деструктор в своей программе. Но на самом деле очень важно, чтобы деструктор был объявлен виртуальным. Рассмотрим следующий пример:

```
class Telephone {
    ...
    ~Telephone();
    ...
};

Telephone *digitalPhone = new ISDNPhone;
...
delete digitalPhone;
```

Что происходит? Вызванный оператор delete не знает, что объект относится к типу ISDNPhone, а в классе Telephone ничто не указывает, что деструктор тоже должен

выбираться на стадии выполнения. Следовательно, в программе будет вызван деструктор Telephone. А это означает, что ресурсы, которые были выделены конструктором ISDNPhone и должны освободиться деструктором ISDNPhone, освобождены *не будут* и превратятся в «мусор».

Если объявить деструктор базового класса Telephone виртуальным, то на стадии выполнения программа выберет деструктор в соответствии с фактическим типом объекта:

```
class Telephone {
    ...
    virtual ~Telephone();
    ...
};

Telephone *digitalPhone = new ISDNPhone;
...
delete digitalPhone; // Работает, вызывается ISDNPhone::~ISDNPhone()
```

В результате будет вызван собственный деструктор объекта, правильно освобождающий ресурсы. Естественно, в соответствии с обычным поведением деструкторов C++ сразу же после этого вызывается деструктор базового класса Telephone::~Telephone (см. раздел 4.3).

5.3. Виртуальные функции и видимость[1]

Объявление функции виртуальной может фактически изменить ее область видимости. Это может вести к изменениям в семантике программы, которые необходимо учитывать при построении иерархий наследования.

Прежде всего стоит заметить, что сигнатура любой функции, участвующей в идентификации операций на стадии выполнения, *обязана* точно соответствовать сигнатуре виртуальной функции своего базового класса. Сигнатура функции представляет собой совокупность имени, упорядоченной спецификации типов параметров и типа возвращаемого значения (в C ++ тип возвращаемого значения формально не считается частью сигнатуры при разрешении вызовов перегруженных функций, но все равно используется при идентификации). Изменять тип возвращаемого значения виртуальной функции запрещено:

```
class Number {
public:
    virtual float add(const Number&);
    Number(double);
    ...
};

class BigNumber: public Number {
```

[1] Перед чтением этого раздела следует вспомнить материал разделов 3.2 и 4.2.

```
public:
    virtual double add(const Number&);  // Недопустимо
    BigNumber(double);
    ...
};
```

Такая замена недопустима, потому что тип выражения, полученного при вызове функции **add** для объекта, неизвестен во время компиляции, и компилятор не сможет сгенерировать код обработки возвращаемого значения:

```
void someFunction(Number &num) {
    Number a = 10, *num;
    ... num->add(a)....     // double или float?
};
```

Механизм виртуальных функций также «ломается» при изменении списка параметров функции в производных классах. Вообще говоря, такое изменение возможно, но в результате появляются две разные функции, не связанные друг с другом: одна функция *скрывает* другую функцию с таким же именем на более высоком уровне иерархии наследования. Пример приведен в листинге 5.2 — функция **BigNumber::add(double)** полностью скрывает одноименную функцию базового класса, и последняя становится недоступной для пользователей базового класса. То же самое произошло бы, если бы функция **BigNumber::add** была объявлена виртуальной.

Листинг 5.2. Изменение сигнатуры нарушает наследование виртуальности

```
class Number {          // Базовый класс
public:
    virtual void add(int);
    // ...
};

class BigNumber : public Number {    // Производный класс
public:
    void add(double);
    // ..
};

int main() {
    Number *a;
    BigNumber *b, bo;
    // ...
    a->add(1);              // Number::add(int)
    a->add(3.0);            // Number::add(int)
                            //  с понижением int(3.0)
    b->add(2);              // BigNumber::add(double)
                            //  с повышением double(2)
    b->add(2.0);            // BigNumber::add(double)
```

продолжение

Листинг 5.2 (*продолжение*)

```
    b->Number::add(7.0);    // Number::add(int)
                            //  с понижением int(7.0)
    bo.add(8);              // BigNumber::add(double)
                            //  с повышением double(8)
    bo.add(9.0);            // BigNumber::add(double)
    bo.Number::add(9);      // Number::add(int)
    bo.Number::add(10.0);   // Number::add(int)
                            //  с понижением int(10.0)
    return 0;
}
```

Между видимостью и виртуальными функциями могут быть другие неочевидные связи. Рассмотрим следующий пример:

```
#include <iostream.h>

class A { public: virtual void f(int); };
void A::f(int) { cout << "A::f\n"; }
class B: public A { public: void f(double); };
void B::f(double) { cout << "B::f\n"; }
class C: public B { public: virtual void f(int); };
void C::f(int) { cout << "C::f\n"; }

int main()
{
    A *a;
    B *b;
    C *c = new C;
    c->f(2);              // C::f
    c->f(2.0);            // C::f с понижением
    b = c;
    b->f(2);              // B::f с повышением
    b->f(2.0);            // B::f
    a = c;
    a->f(2);              // C::f
    a->f(2.0);            // C::f с понижением
    return 0;
}
```

В корневом базовом классе A определена виртуальная функция. В промежуточном классе B определяется невиртуальная функция с другой сигнатурой, а в листовом классе C виртуальная функция определяется заново. Заслуживает внимания тот факт, что виртуальная функция, находящаяся высоко в иерархии, может скрываться функциями в середине иерархии. На ситуацию можно взглянуть и так: объект класса C «несет» в себе функцию C::f(int), но видимость функции C::f(int) изменяется в зависимости от типа указателя, используемого для адресации объекта. Такое представление противоречит нашей интуиции, которая вроде бы

подсказывает, что при объявлении виртуальной функции для объекта всегда вызывается одна и та же функция независимо от того, как вы на этот объект ссылаетесь. Многие современные компиляторы C++ предупреждают о том, что перегруженная функция замещает одноименную виртуальную функцию, объявленную выше в иерархии наследования.

5.4. Чисто виртуальные функции и абстрактные базовые классы

Вернемся к примеру класса Telephone из главы 4, в котором создание экземпляров Telephone предотвращалось объявлением защищенного конструктора (с. 115). Предполагалось, что экземпляры могут создаваться только на базе производных классов: POTSPhone, VideoPhone, ISDNPhone и т. д. С абстрактным классом Telephone реально ничего нельзя *сделать*! Почему? Потому что мы не можем написать код функции ring для выдачи звонка на абстрактном телефоне. На разных телефонах эта операция выполняется по-разному. На традиционных телефонах подается напряжение; цифровые ISDN-телефоны принимают сообщение на выдачу звонка и подсветку индикатора, и т. д.

В C++ имеется синтаксис для сохранения семантики функций класса (таких, как ring) в объявлениях этих функций. Прежде всего такая функция должна быть виртуальной; мы хотим, чтобы программы, в которой используются разные типы объектов Telephone, могла вызывать для них функцию ring, не заботясь о подробностях ее реализации для конкретного типа. А поскольку для такой функции заведомо нельзя написать обобщенную, стандартную реализацию, вместо тела функция ассоциируется с нулевым указателем:

```
class Telephone {
public:
    virtual void ring() = 0;   // Чисто виртуальная функция
    Bool isOnHook();
    Bool isTalking();
    Bool isDialing();
    virtual DigitString collectDigits();
    LineNumber extension();
    virtual ~Telephone();
    Telephone();               // Снова объявляется открытым
    ...
};
```

Такие функции называются *чисто виртуальными*. Объявление чисто виртуальной функции всего лишь несет информацию о том, что абстрактный телефон способен звонить, хотя мы не можем (на обобщенном уровне) сказать, как именно выполняется эта операция. Термин «чистый» в данном случае означает, что функция не имеет тела, определен лишь интерфейс.

Теперь программа не сможет создать объект класса Telephone просто потому, что компилятор не знает, что делать с таким объектом при вызове для него операции ring. Такие классы, определяющие некоторый аспект поведения без определения его реализации, называются *абстрактными базовыми классами*, или *неполными классами*. Они накладывают на производные классы обязательство определить реализации чисто виртуальных функций. В сущности, абстрактный базовый класс определяет тип (абстрактный тип данных) без определения реализации. Если же все функции класса являются чисто виртуальными, то абстрактный базовый класс задает спецификацию абстрактного типа данных (АТД), и не более того.

Чисто виртуальные функции делают нечто большее, чем скрытый конструктор из главы 4 (см. с. 115). Во-первых, они объявляют, что их класс не может использоваться для создания экземпляров, а следовательно — не может использоваться в качестве типа формального параметра или возвращаемого значения функции. Скрытый конструктор тоже обладал этим свойством, но с одним исключением: объект базового конструктора мог создаваться в функциях производного класса. Во-вторых, чисто виртуальные функции *должны* подменяться в производном классе, чтобы программа могла создавать экземпляры этого класса.

Для чисто виртуальной функции *можно* определить тело. Для вызова чисто виртуальной функции с телом используется оператор уточнения области видимости (::):

```
#include <iostream.h>

class Base {
public:
    virtual void pure() = 0;
};

class Derived: public Base {
public:
    void pure() { cout << "Derived::pure" << endl; }
    void foo() { Base::pure(); }
};

void Base::pure() { cout << "Base::pure" << endl; }

int main() {
    Derived d;
    d.Base::pure();
    d.foo();
    d.pure();
    Base *b = &d;
    b->pure();
    return 0;
}
```

При запуске программа выводит следующий результат:

```
Base::pure
Base::pure
Derived::pure
Derived::pure
```

Если удалить тело Base::pure, произойдет ошибка компоновки.

Данная возможность может пригодиться в примере класса Window (окно). Допустим, класс Window настаивает на том, чтобы производные классы предоставляли собственную версию функции очистки окна clear, но при этом может предоставлять базовую функциональность функции clear, общую для всех типов окон. Например, тело Window::clear может устанавливать курсор в центр экрана, когда оно вызывается в качестве завершающего действия в функциях clear производных классов.

5.5. Классы конвертов и писем

Идиома «манипулятор/тело» позволяет применять некоторые приемы проектирования, обладающие большей гибкостью и быстродействием, а также меньшими последствиями при внесении изменений по сравнению с ортодоксальной канонической формой проектирования (см. главу 3). В этой идиоме задействована пара классов, используемых как единое целое: внешний класс (манипулятор), который является видимой частью, и внутренний класс (тело), в котором спрятаны подробности реализации. Говорят, что манипулятор *передает* запросы телу, находящемуся внутри него. В совокупности эти два класса образуют *составной объект*. В этом разделе представлено расширение идиомы «манипулятор/тело» — идиома «конверт/письмо», обеспечивающая дополнительную гибкость и качество инкапсуляции по сравнению с идиомой «манипулятор/тело».

Идиома «манипулятор/тело» добавляет новый уровень косвенных обращений во взаимодействия объектов. Повышение гибкости обусловлено в основном этой косвенностью, то есть привязкой на стадии выполнения. Хотя на самом деле объектов несколько, с точки зрения пользователя существует один объект (манипулятор), управляющий всей работой составного объекта. Такое разделение может потребоваться для того, чтобы изолировать «прослойку» управления памятью от «настоящей семантики» внутреннего объекта при нетривиальном управлении памятью, как в примере класса String (см. раздел 3.5). Кроме того, оно задействовано в механизмах уборки мусора и в схемах замены классов/объектов во время выполнения, описанных в главе 9.

Идиома «конверт/письмо» является частным случаем идиомы «манипулятор/тело» для ситуаций, когда манипулятор и тело обладают общим поведением, но тело специализирует или оптимизирует поведение манипулятора. При подобном использовании класс манипулятора называется *конвертом*, а класс тела — *письмом*. Отношения между этими классами во многом напоминают отношения между

базовым классом (эта роль отводится конверту) и производным классом (письмо), но с существенно большей гибкостью на стадии выполнения по сравнению с наследованием. Класс письма является производным от класса конверта, но его экземпляры также содержатся в экземплярах конверта.

Общности «конверт/письмо» обычно более замкнуты, чем классы в иерархии наследования. Обращение к большинству классов писем может осуществляться только через конверт. Это в определенной степени справедливо для классов в иерархии наследования, за исключением того, что вызовы конструкторов производных классов могут распределяться по всей программе. Вызовы конструкторов классов писем чаще локализуются в коде класса конверта.

ПРИМЕЧАНИЕ

В некоторых ситуациях необходима особая гибкость, недостижимая при связывании на стадии компиляции. Например, программисту может потребоваться контроль над функциями, традиционно относимыми к стадии компиляции, скажем, ассоциациям «объект/класс». Идиома «конверт/письмо» способна обеспечить более высокий уровень полиморфизма и идентификации типов на стадии выполнения, чем простое наследование с виртуальными функциями, а это способствует упрощению и обобщению взаимодействия пользователя с пакетом взаимосвязанных классов. Кроме того, снижается потребность в редактировании и перекомпиляции программы при внесении изменений.

Существуют две точки зрения на отношения между конвертом и классом. Во-первых, можно представить, что конверт делегирует часть своей функциональности работающему с ним объекту письма. Во-вторых, письмо может рассматриваться как «смысловая нагрузка», которую логично инкапсулировать в конверте. Вторая модель хорошо подходит для обеспечения инкапсуляции и имеет много общего с примерами из раздела 3.5.

В следующем разделе конверты и письма рассматриваются более подробно, поскольку на этой основе строятся более мощные идиомы объектно-ориентированного программирования, нежели те, которые поддерживаются на уровне «монолитных» классов C++. Сначала будет обоснована потребность в связывании на стадии выполнения объекта со свойствами, обычно ассоциируемыми с классами. Затем мы перейдем к частному случаю этой идиомы и посмотрим, как лучше создавать объекты, не зная их точного класса. Далее это решение будет расширено: вы научитесь создавать объекты переменного размера без дополнительного уровня косвенных обращений. Специальный вспомогательный синтаксис избавит вас от дублирования кода писем и конвертов. Раздел завершается усовершенствованным вариантом реализации писем и конвертов с использованием вложенных классов C++.

Классы конвертов и делегированный полиморфизм

В чистых объектно-ориентированных языках вроде Smalltalk переменные связываются с объектами на стадии выполнения. Связывая переменную с объектом, вы словно наклеиваете на последний временный «ярлык». Когда в таком языке

выполняется присваивание, с объекта снимается один ярлык, а на его место наклеивается другой. Переменные в таких языках несут минимальный объем информации о типе, а компилятор практически не следит за совместимостью типов. Конечно, у самих объектов имеются операции, функции и атрибуты, которые в совокупности могут рассматриваться как информация о типе, и это позволяет выполнять проверку типов на стадии выполнения. К недостаткам языков, обладающих этим свойством, следует отнести неприятные сюрпризы с типами, возникающие во время выполнения. С другой стороны, высокая степень гибкости, присущая таким языкам, приносит пользу при построении прототипов и доработке программы; она же закладывает основу для более совершенных механизмов управления памятью. Модель типов C++ не обладает такой гибкостью по двум причинам: из-за раннего связывания символических имен с адресами и из-за ранней проверки совместимости типов.

В C и C++ переменная является синонимом адреса или смещения, а не «меткой» объекта. Присваивание не сводится к простой смене ярлыков; на место старого содержимого объекта записывается новое содержимое. Чтобы компенсировать это обстоятельство, мы вручную добавляем промежуточный логический уровень, обращаясь к объектам через указатели. Хотя имена переменных по-прежнему связаны с блоками памяти, память определяется «простым» указателем, поэтому ассоциация между именем и «настоящим» объектом легко изменяется заменой указателя. Впрочем, программирование с применением указателей порождает свои неудобства. Оно усложняет управление памятью; возникает опасность появления «висячих указателей». Обращение к любым объектам через указатели затрудняет использование некоторых языковых конструкций, особенно перегрузку операторов:

```
class Number {
public:
    Number();
    virtual Number *operator*(const Number&);
    ...
};

class Complex: public Number {
public:
    Complex(double,double);
    Number *operator*(const Number&);
    ...
};

class Integer: public Number {
public:
    Integer(int);
    Number *operator*(const Number&);
    ...
};
```

```
Number *numberVector[5], *numberVector2[5], *productVector[5];
...
numberVector[0] = new Complex(0,10);
numberVector[1] = new Integer(0);
numberVector[2] = 0;
numberVector2[0] = new Integer(10);
numberVector2[1] = new Complex(10,10);
for (int i = 0; numberVector[i] && numberVector2[i]; i++) {
    // Уродливый (но необходимый) синтаксис:
    productVector[i] = *numberVector[i] * *numberVector2[i];
}
```

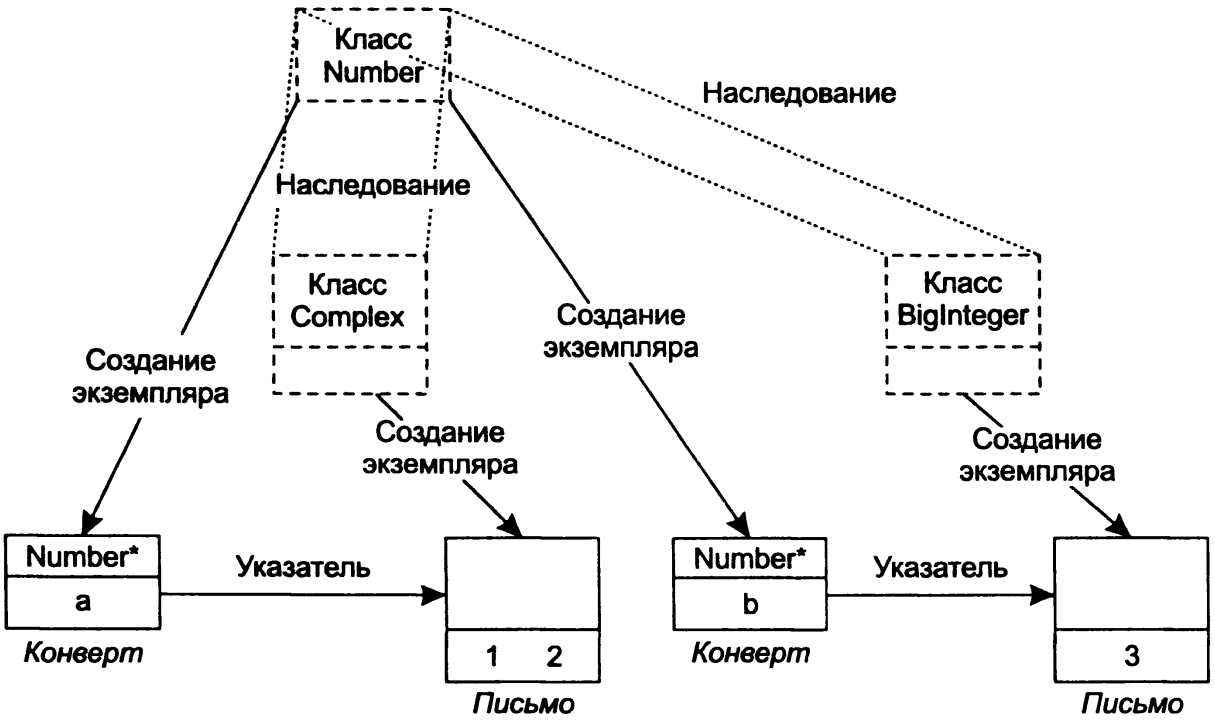
Существует и другая сложность: компилятор C++ следит за тем, чтобы объект мог предоставить функции, вызванные для переменной, через которую вы обращаетесь к объекту. Для этой цели компилятор использует *систему контроля типов*. «Система контроля типов» в этом контексте довольно слабо связана с концептуальными абстракциями приложения, которые могут рассматриваться как формальные типы, отражаемые программой в своей структуре классов (см. раздел 6.1), Системы контроля типов способствуют предотвращению ошибок; в частности, система контроля типов C++ выявляет несоответствия типов на стадии компиляции. Эта система весьма консервативна в своей интерпретации структуры классов, и как следствие — в своем представлении об отношениях между архитектурными абстракциями: она находит все реальные несоответствия типов предметной области, но при этом может заблокировать некоторые допустимые операции.

Нежесткое связывание переменных с объектами и мощные системы контроля типов в символических языках избавляют программиста от необходимости специально объявлять о совместимости между типами. На практике часто требуется работать с группой взаимозаменяемых классов и интерпретировать все объекты этих классов так, словно они относятся к одному классу. Виртуальные функции предоставляют такую возможность для указателей и ссылок. Однако указатели плохо защищены от злоупотреблений (их называют «командами GOTO в мире данных»), к тому же они не интегрируются с перегруженными операторами. В приведенном примере класса Number указатели позволяют интерпретировать любую числовую величину как экземпляр Number, хотя за это приходится расплачиваться неудобным синтаксисом. На самом деле мы хотим, чтобы каждый объект автоматически применял свое умение выполнять различные операции:

```
int main() {
    Number c(1.0, -2.1);
    Number r(5.0);
    Number product = c * r;
    r = 0;
    return 0;
}
```

Но в этом случае возникает проблема — объекты должны идентифицировать свой фактический класс на стадии выполнения. Для переменной c — это тип Complex, для переменной r — DoublePrecisionFloat, а для переменной product — тот тип, который должен быть получен в результате операции (теоретически первые два случая могут быть разрешены особо проницательным компилятором на стадии компиляции, но третий тип определяется только во время выполнения). Но объекты разных классов занимают разный объем памяти, поэтому смена класса объекта приведет к изменению его размера, а эта величина известна на стадии компиляции. Связывание объектов с виртуальными функциями производится в процессе инициализации объекта и остается неизменным; данное обстоятельство тоже препятствует изменению типа объекта. Еще одна проблема заключается в том, что после идентификации класса объект может «передумать» и сменить свою принадлежность. Например, исходное определение product предполагает, что этот объект представляет комплексное число, но после обнуления сохранять его принадлежность к категории комплексных чисел бессмысленно.

Итак, мы хотим создать механизм установления связи между объектом и его классом на стадии выполнения. Более того, связывание переменных с классом их объектов не должно быть жестким — иначе говоря, в процессе выполнения программы нужно разрешить переменным «менять свои типы». Набор таких классов для нашего примера с числами представлен на рис. 5.4, а также в листингах 5.3 и 5.4.




```
Number a = Number(1, 2);
Number b = 3;
```

Рис. 5.4. Полиморфные числа

Листинг 5.3. Часть реализации класса Number

```
struct BaseConstructor { BaseConstructor(int=0) { } };

class Number {
public:
    Number() { rep = new RealNumber(0.0); }
    Number(double d) { rep = new RealNumber(d); }
    Number(double rpart, double ipart) {
        rep = new Complex(rpart, ipart);
    }
    Number operator=(Number & n) {
        n.rep->referenceCount++;
        if (--rep->referenceCount == 0) delete rep;
        rep = n.rep;
        return *this;
    }
    Number(Number & n) {
        n.rep->referenceCount++;
        rep = n.rep;
    }
    virtual Number operator+(const Number &n) {
        return rep->operator+(n);
    }
    virtual Number complexAdd(Number &n) {
        return rep->complexAdd(n);
    }
    virtual ostream& operator<<(ostream &s) {
        return rep->operator<<(s);
    }
    virtual Number realAdd(const Number &n) {
        return rep->realAdd(n);
    }
    void redefine(Number *n) {
        if (--rep->referenceCount == 0) delete rep;
        rep = n;
    }
    ...
protected:
    Number(BaseConstructor) { referenceCount = 1; }
private:
    Number *rep;
    short referenceCount;
};
```

Листинг 5.4. Часть реализации класса Complex

```
class Complex: public Number {
public:
    Complex(double d, double e): Number(BaseConstructor()) {
```

```
        rpart = d; ipart = e;
        referenceCount = 1;
    }
    Number operator+(Number &n) { return n.complexAdd(*this); }
    ...
    Number realAdd(Number &n) {
        Number retval;
        Complex *c1 = new Complex(*this);
        RealNumber *c2 = (RealNumber*)&n;
        c1->rpart += c2->r;
        retval.redefine(c1);
        return retval;
    }
    Number complexAdd(Number &n) {
        ...
    }
private:
    double rpart, ipart;
};
```

Для сохранения разумной семантики сигнатура всех классов нашей иерархии должна быть сходна с сигнатурой класса Number. В C++ это требование выражается определением этих классов как производных от абстрактного базового класса Number. На базе Number строится иерархия числовых типов. Объекты таких классов упаковываются в «конверт», сохраняющий «идентичность» (адрес, если хотите) величины, пока ее значение — а возможно, и тип — изменяется с применением к ней различных операций. «Конверт» обладает такой же сигнатурой, что и содержащиеся в нем объекты, поэтому один класс Number играет сразу две роли: абстрактного базового класса для классов «писем» и «конверта», скрывающего подробности реализации от пользователя. Модель изображена на рис. 5.4, а пример кода приведен в листинге 5.3.

В соответствии с идиомой подсчета ссылок (см. раздел 3.5) основная логика управления памятью для объектов Number сосредоточена в классе конверта, а семантика приложения размещается в классе письма (см. листинги 5.3 и 5.4). Конструкторы строят разновидность объекта письма для заданных параметров. В операторе присваивания и копирующем конструкторе также используется идиома подсчета ссылок; в переменной класса referenceCount хранится счетчик ссылок на общее представление. Класс Number перенаправляет операции письму, хранящемуся «внутри» него, как показано в реализации функции operator+ и других операторных функций в листинге 5.3.

Класс Number (см. листинг 5.3) служит одновременно классом объекта «универсального числа», который виден пользователю, и базовым классом для классов писем; некоторые из его функций отражают особый сервис, предоставляемый производным классам. Специальный конструктор Number::Number(BaseConstructor) инициализирует экземпляры Number, образующие подобъект базового класса в классах писем. Он замещает конструктор по умолчанию Number::Number для

предотвращения бесконечной рекурсии при создании нового экземпляра класса, производного от Number.

Функции complexAdd, realAdd и redefine являются специфическими функциями приложения и предоставляют необходимый сервис производным классам. Тем не менее, эти функции не объявляются защищенными, потому что они вызываются между экземплярами. Объект Number должен быть готов принять экземпляр любой «разновидности» Number в качестве операнда своей операции. Рассмотрим следующий код:

```
Number aComplex(1.0, 2.0);
Number aReal(3.0);
Number result = aComplex + aReal;
```

Когда операторная функция operator+ объекта aComplex вызывается с параметром aReal, она перенаправляет запрос своему объекту письма. Тот вызывает операторную функцию operator+ класса Complex с операндом aComplex и параметром aReal. В свою очередь, Complex::operator+ вызывает операцию complexAdd для своего параметра, передавая ей *this (значение aComplex). Управление передается функции complexAdd класса Number, которая просто перенаправляет вызов функции complexAdd класса RealNumber. Располагая дружественным доступом к обоим объектам, функция complexAdd класса RealNumber генерирует результат соответствующего типа и возвращает его.

Теперь вы примерно представляете, как этот механизм функционирует. О том, как такие полиморфные объекты появляются на свет, рассказано в следующем разделе.

Имитация виртуальных конструкторов

Один из важных принципов объектно-ориентированного проектирования (см. главу 6) гласит, что каждый класс должен уметь выполнять свои операции. Мощь объектно-ориентированного программирования в значительной мере обусловлена полиморфизмом, создаваемым сочетанием наследования и виртуальных функций. Наследование и виртуальные функции позволяют осуществлять все взаимодействие пользователя с объектом исключительно через интерфейс, определенный в базовом классе. Компилятор обеспечивает «волшебный» механизм перенаправления вызовов соответствующим функциям производных классов на стадии выполнения. Конструкции C++ отделяют пользователя семейства классов от служебной информации о количестве и природе производных классов, а также от подробностей реализации этих классов.

Впрочем, в C++ имеется одно исключение, нарушающее этот принцип, который считается основополагающим в «чистых» объектно-ориентированных языках. Для примера возьмем класс Number с производными классами BigNumber и Complex, где многие функции производных классов также присутствуют в виде виртуальных функций в базовом классе. Пользователь, располагающий ссылкой или указателем на Number, может связать его с объектом класса Number, BigNumber или

Complex, и спокойно работать с любым из этих объектов, используя виртуальные функции Number. Пользователь не обязан знать строение иерархии Number; он одинаково работает со всеми экземплярами в контексте свойств, унаследованных этими экземплярами от Number.

Но как появился указатель, ссылающийся на объект производного класса? Где был создан этот объект, и как его создатель указал, какой из производных классов требуется? Пользователь не мог создать объект конструкцией new BigNumber или new Complex; это было бы нарушением принципа, в соответствии с которым пользователю неизвестны детали иерархии производных классов. Объект не мог быть получен от третьей стороны, поскольку третья сторона знает о производных классах Number ничуть не больше, чем пользователь. Следовательно, указатель должен быть получен внутри самой комбинации классов Number/BigNumber/Complex. Можно представить, что конструктор Number создал объект BigNumber, потому что переданный параметр превысил заданный порог, или же объект Complex был создан внутри Number при вычислении квадратного корня для отрицательной величины.

Итак, возникает предположение, что при использовании конструкции new Number конструктор Number создает объект BigNumber или Complex, делая так, что возвращаемое значение оператора new ссылается на один из этих объектов. Но такого быть не может! Посмотрите:

```
Number::Number() {
    if (некоторое условие)
        this = new BigNumber;  // Недопустимо
}
```

Присваивать новое значение this в C++ запрещено.

Для решения можно воспользоваться дополнительной абстракцией, которая выполняет функции агента для взаимодействия с семейством Number/BigNumber/Complex по поручению клиента, создающего объект. На роль такого агента подходит объект класса Number, используемого в качестве класса конверта. Каждый экземпляр Number указывает на объект письма, созданный на базе одного из его производных классов, например, BigNumber или Complex. Указатель объявляется с типом Number*:

```
class Number {
private:
    Number *rep;
    ...
```

Вызовы функций Number перенаправляются объекту, на который ссылается закрытое поле rep:

```
public:
    Number operator*(const Number &n) {
        return rep->operator*(n);
    }
    ...
```

Объявления функций класса письма в точности соответствуют объявлениям их базового класса Number. Закрытое поле rep инициализируется конструктором конверта, который конструирует BigNumber или Complex в зависимости от некоторого условия и помещает указатель на полученный объект в закрытое поле rep.

Для примера возьмем класс Number, для которого было бы естественно определить операцию направления в поток вывода (тип ostream распространенной библиотеки C++ iostreams). Но при этом нам также хотелось бы иметь возможность инициализации Number на базе istream (потока ввода) — то есть наделить Number возможностью прочитать себя из потока и создать новый объект в процессе чтения. Следовательно, в классе должен присутствовать конструктор следующего вида:

```
class Number {
public:
    ...
    Number(istream& s) {
        char buf[10];
        s >> buf;
        switch( numParse(buf) ) {
        case COMPLEX:
            rep = new Complex(buf);
            break;
        case FLOAT:
            rep = new DoublePrecisionFloat(buf);
            break;
        ...
        }
    }
private:
    enum NumType {COMPLEX, FLOAT, BIGINT};
    NumType numParse(char *);      // Определение типа числа
    Number *rep;
};
```

(Конечно, в реальном приложении значение rep будет присваиваться в теле numParse, чтобы избежать дублирования информации обо всех числовых классах между этой функцией и функцией Number::Number(istream&).)

Теперь все выглядит так, будто класс создает разные типы объектов, свойства которых определяются контекстом, переданным конструктору (в данном случае — потоком байтов из файла или другого источника).

Рассмотренная идиома наделяет классы той гибкостью, которой виртуальные функции наделяют объекты. Эта идиома известна под названием *виртуального конструктора*. Виртуальные конструкторы являются шагом вперед по направлению к имитации «полноценных» классов в C++; при этом язык C++ используется так, словно он является моноиерархической системой (то есть состоящей исключительно из объектов без типов, которые являются не классами, а «чем-то иным»). Идиома виртуального конструктора подробно проанализирована (и расширена) в главе 8.

ПРИМЕЧАНИЕ

Идиома виртуального конструктора используется в тех ситуациях, когда тип объекта должен определяться по контексту конструирования. В качестве примеров можно привести построение объекта окна правильного типа и размера в зависимости от типа и размера экрана, используемого программой, создание объекта в интерактивном режиме или по неформатированным данным из файла.

Следующий пример позаимствован из архитектуры класса Atom, используемого для представления атомов (неделимых лексем) при разборе текста. Класс Atom выполняет большую часть диспетчерских операций в простой системе разбора. Вот как выглядит определение базового класса, все функции которого объявлены виртуальными:

```
class Atom {
public:
    Atom() {}
public:    // Общие функции всех производных классов
    virtual ~Atom() {}
    virtual long value()     { return 0; }
    virtual String name()    { return String("error"); }
    virtual operator char()  { return 0; }
    virtual Atom *copy() = 0;
};
```

Обратите внимание: функция Atom::copy является *чисто виртуальной*; за ее объявлением следует спецификация = 0. Классы, производные непосредственно от Atom и не предоставившие тело Atom::copy, наследуют эту функцию как чисто виртуальную. А это означает, что функция copy *должна* быть определена во всех классах, непосредственно производных от Atom, чтобы программа могла создавать экземпляры этих классов; это необходимо для правильной работы идиомы. Отсутствие тела у функции Atom::copy делает невозможным создание объекта Atom. Только производный класс, содержащий или унаследовавший функцию copy, может воплощаться в экземплярах.

Теперь давайте посмотрим на классы писем, выполняющие большую часть «настоящей работы». Эти классы автономны; чтобы понять логику их работы, не нужно разбираться в подробностях устройства класса конверта. В листинге 5.5 приведен класс числовых атомов NumericAtom. Конструктор NumericAtom::NumericAtom(const String&) выделяет целое число из своего параметра String, «поглощая» в процессе работы символы строки. За ним следует простой копирующий конструктор. Функция value возвращает целое значение.

Листинг 5.5. Класс лексического атома NumericAtom

```
class NumericAtom: public Atom {
public:
    NumericAtom(): sum(0) { }
    NumericAtom(String &s) {
        sum = 0;
```

продолжение

Листинг 5.5 (*продолжение*)

```
        for (int i = 0; s[i] >= '0' && s[i] <= '9'; i++) {
            sum = (sum*10) + s[i] - '0';
        }
        s = s(i, s.length()-i);
    }
    NumericAtom(const NumericAtom &n) { sum = n.value(); }
    NumericAtom() { sum = 0; }
    long value() const { return sum; }
    Atom *copy() const {
        NumericAtom *retval = new NumericAtom;
        retval->sum = sum; return retval;
    }
private:
    long sum;
};
```

Поскольку класс NumericAtom предназначен для представления числовой величины, он переопределяет операцию value класса Atom, но сохраняет за функцией преобразования символов operator char стандартное поведение (функция возвращает нуль-символ). Следовательно, пользователь этих классов по контексту должен знать, когда осмыслены вызовы таких функций, как operator char или value; определять их чисто виртуальными не следует, поскольку нет смысла переопределять их во многих производных классах. Функция copy создает физическую копию объекта и возвращает Atom*. Так в нашем распоряжении появляется универсальная операция, дублирующая любой объект Atom. Благодаря функции copy отпадает необходимость в «поле типа», о котором речь пойдет далее.

Классы писем Name, Punct и Oper реализованы аналогично классу NumericAtom (листинг 5.6).

Листинг 5.6. Классы Name, Punct и Oper

```
class Name   public Atom {
public:
    Name(): n("") { }
    Name(String& s) {
        for (int i=0; s[i] >= 'a' && s[i] <= 'z'; i++) {
            n = n + s(i,1);
        }
        s = s(i, s.length()-i);
    }
    Name(const Name& m) { n = m.name(); }
    ~Name() {}
    String name() const { return n; }
    Atom *copy() { Name *retval = new Name;
                   retval->n = n; return retval; }
private:
    String n;
};
```

```
class Punct : public Atom {
public:
    Punct(): c('\0')       { }
    Punct(String& s)       { c = s[0]; s = s(1,s.length()-1); }
    Punct(const Punct& p) { c = char(p); }
    operator char() const { return c; }
    ~Punct() {}
    Atom *copy() { Punct *retval = new Punct;
                   retval->c = c;  return retval; }
private:
    char c;
};

class Oper : public Atom {
public:
    Oper(): c('\0') { }
    Oper(String& s) { c = s[0]; s = s(1,s.length()-1); }
    Oper(Oper& o)   { c = char(o); }
    ~Oper() {}
    operator char() const { return c; }
    Atom *copy() { Oper *retval = new Oper;
                   retval->c = c; return retval; }
private:
    char c;
};
```

Класс конверта **GeneralAtom** предназначен для управления созданием всех объектов производных классов **Atom**:

```
class GeneralAtom : public Atom {
public:
    GeneralAtom(String&);
    GeneralAtom(const GeneralAtom& a) { realAtom=a.copy(); }
    ~GeneralAtom()     { delete realAtom; }
    Atom *transform()  { Atom *retval=realAtom; realAtom=0;
                         return retval; }
public:   // Объединение сигнатуры всех классов иерархии Atom
          // Необходимо,чтобы класс GeneralAtom мог
          // использоваться так, словно он является
          // экземпляром одного из своих производных
          // классов (то есть без использования transform()).
    long value()    { return realAtom->value(); }
    String name()   { return realAtom->name(); }
    operator char() { return realAtom->operator char(); }
private:
    Atom *realAtom;
};
```

Конверт GeneralAtom — единственный класс, с которым пользователь работает напрямую. Объекты остальных классов рассматриваются как письма, хранящиеся в объектах класса конверта. Класс GeneralAtom может получать в параметре конструктора GeneralAtom::GeneralAtom(String&) символьную строку; он анализирует строку, создает и сохраняет указатель на объект соответствующего класса, производного от Atom:

```
GeneralAtom::GeneralAtom(String &s) {
    if (!s.length()) realAtom = 0;
    else switch(s[0]) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
       realAtom = new NumericAtom(s); break;
    case ',': case ':': case ';': case '.':
       realAtom = new Punct(s); break;
    case '*': case '/': case '+': case '-':
       realAtom = new Oper(s); break;
    default:
       if (s[0] >= 'a' && s[0] <= 'z') {
           realAtom = new Name(s);
       } else
           realAtom = 0;
    }
}
```

Простой пример использования этого конструктора:

```
Atom *lexAtom(String& s) {
    GeneralAtom g(s);
    return g.transform();   // Для повышения эффективности
}
void parse(String& s) {
    for (Atom *a = lexAtom(s); a = lexAtom(s)) {

    }
}
```

Экземпляр класса конверта может использоваться двумя способами: как заготовка (то есть временный объект для построения конкретного объекта в обобщенном контексте) или как реальный объект, который может передаваться в программе и «жить собственной жизнью». Обратите внимание: при построении объекта GeneralAtom итоговый указатель на объект, полученный от функции operator new, может быть объявлен с типом Atom*; такой указатель может указывать на объект GeneralAtom, потому что класс GeneralAtom является производным от Atom и поддерживает все аспекты его поведения. Наследование как конверта, так и писем от одного базового класса является важной частью этой идиомы; оно гарантирует их принадлежность к одной архитектурной концепции. Но вместо указателя на объект GeneralAtom с таким же успехом можно было воспользоваться указателем на сгенерированный объект письма класса Oper, Punct, NumericAtom и т. д.

В обоих случаях программа будет работать одинаково; в последнем случае конверт является только заготовкой для построения объекта письма. В таких ситуациях конверт не обязан абсолютно точно выдерживать поведение производных классов Atom, а его сигнатура — отражать все сигнатуры всех представляемых типов.

Объекты классов конвертов должны быть временными по своей природе, *если только* их сигнатура не является объединением сигнатур всех классов писем. В нашем примере за счет введения лишнего логического уровня косвенных обращений конверт может использоваться почти везде, где могут использоваться «настоящие» классы. Этот уровень можно исключить, заменив содержимое существующего указателя Atom*, указывающего на GeneralAtom, результатом вызова операции transform для этого указателя; после этого конверт может быть освобожден.

Если класс конверта действительно содержит все операции, которые когда-либо потребуются для классов писем, то функция transform будет вызываться только для повышения быстродействия за счет исключения лишнего логического уровня. Те, кто готов смириться с присутствием этого уровня, могут свободно задействовать экземпляры класса конверта в виде автоматических переменных или глобальных инициализированных объектов. Ключевое преимущество идиомы «конверт/письмо» состоит в том, что она полностью скрывает динамический характер создания объекта от пользователя, сохраняя предполагаемый интерфейс объекта. Если такая прозрачность не нужна, достаточно скрыть логику инициализации в глобальной функции или открытой статической функции класса.

И еще одно замечание: когда в программе вызывается конструктор следующего вида, он должен получить копию объекта, переданного ему в параметре:

```
GeneralAtom::GeneralAtom(const GeneralAtom&)
```

Для этого можно было бы включить поле типа во все классы, производные от Atom, чтобы объект GeneralAtom использовал это поле для вызова соответствующего конструктора (с необходимым преобразованием). В нашем примере эта проблема решалась применением виртуальной функции copy. При добавлении нового производного класса в Atom или GeneralAtom требуется изменить только сам конструктор GeneralAtom(String&).

Другой подход к виртуальным конструкторам

В этом разделе представлено другое выражение идиомы «виртуального конструктора» для создания объектов, тип и размер которых неизвестен при входе в конструктор. Вместо применения идиомы «конверт/письмо» с дополнительным уровнем косвенных обращений в этом решении тип объекта модифицируется на месте. Данное решение строится на базе схемы управления памятью методом близнецов и называется *идиомой виртуального конструктора для метода близнецов*. В качестве примера рассмотрим класс, который читает поток байтов из входного канала и преобразует его в объект типа, определяемого по содержимому входных сообщений.

Этот механизм также может использоваться для создания объектов переменного размера, то есть объектов, увеличивающих свой размер для резервирования памяти под вектор, находящийся в конце объекта. Данное решение выглядит более эстетично, чем решение, представленное в [2].

ПРИМЕЧАНИЕ

Идиома виртуального конструктора для метода близнецов лучше всего подходит для ситуаций, в которых размер объекта неизвестен заранее, например, если его невозможно определить на основании класса объекта (как при работе с сообщениями переменной длины). Она также полезна, когда затраты на дополнительный уровень косвенных обращений оказываются неприемлемыми, или использование указателей почему-либо невозможно (объекты в общей памяти, стратегии аудита с буферами, конструируемыми из готовых блоков, и т. д.).

Идиома виртуального конструктора для метода близнецов редко используется для конкретных типов данных и выходит за рамки того, что считается стандартной практикой в реализациях C++ (хотя следующий пример не выдвигает никаких особых предположений относительно специфики среды C++). Эта идиома особенно полезна при работе с динамичными системными ресурсами, как в описанном далее примере с буферами сообщений.

Допустим, имеется базовый класс LAPD (Link Access Protocol for Data — протокол доступа к каналу для данных), используемый в сетевом приложении. LAPD является подмножеством распространенного протокола HDLC (High-Level Data Link Control — высокоуровневое управление каналом), используемого на уровне 2 (канальный уровень) стека протоколов модели ISO (International Standards Organization — международная организация по стандартизации). На базе класса LAPD создается производный класс, характеризующий следующий уровень протокола. Предположим, существует несколько разных типов сообщений, соответствующих требованиям протокола X.25 уровня 3 (сетевой уровень), построенного на базе протокола LAPD уровня 2. Разные типы сообщений представляют разные коммуникационные функции: оповещение, подключение, настройку параметров, подтверждение и т. д.

Центральное место в этом решении занимает перегруженный оператор new, который позволяет конструктору базового класса заместить свой объект объектом производного класса. Когда конструктору базового класса при вызове передается поток символов из канала данных, нужный тип объекта сообщения строится в динамической памяти.

В следующей схеме эта задача решается посредством выборочной перегрузки функции operator new. Сначала глобально перегруженный оператор new обеспечивает вызов конструктором LAPD одного из конструкторов производных классов с указанием того, что объект должен строиться по адресу объекта базового класса. Класс LAPD содержит свой оператор new, который строит объект в заранее выделенном буфере (размер буфера заведомо достаточен для хранения наибольшего сообщения):

```
// Следующий оператор позволяет объекту в иерархии LAPD
// заменить себя экземпляром одного из его производных классов.
inline void *operator new(size_t, LAPD *l) { return l; }
```

```
unsigned char bigBuf[4096], *bigBufPtr = bigBuf;

class LAPD {
public:
    void *operator new(size_t) { return (void*) bigBufPtr; }
    LAPD(const char *buf) {
        ...
        (void) new(this) Setup(buf);
        ..
    }
    LAPD() {  /* Пусто */ }
};

class Setup: public LAPD {
    ...
};
```

Следующая конструкция вызывает функцию **LAPD::operator new** для выделения памяти под объект перед вызовом конструктора:

```
LAPD *msg = new LAPD(buf);
```

Память для объекта берется из буфера, на который указывает **bigBufPtr**. Затем вызывается конструктор **LAPD**, при этом значение **this** указывает на инициализированную память **bigBufPtr**. Допустим, анализируя содержимое **buf**, конструктор **LAPD** решает, что было передано сообщение **Setup**, и вызывает глобально перегруженный оператор **new** для выделения памяти под объект производного класса. Глобально перегруженный оператор **new** просто возвращает адрес, переданный в параметре, после чего вызывается конструктор **Setup**. Таким образом, значение **this** в конструкторе **LAPD** переходит в значение **this** в конструкторе **Setup**.

Основная проблема состоит в том, что память должна быть выделена перед вызовом конструктора производного класса, однако компилятор заранее не знает, сколько именно памяти понадобится. Задача решается выделением памяти под все сообщения в нижней части большого пула памяти и смещением адреса нижнего края пула за границу данных:

```
inline int round(int a, int b) { return ((a+b-1)/b)*b; }

class Setup: public LAPD {
friend LAPD;
private:
    SetupPacketBody body;
    Setup(const char *m): LAPD() {
        ...
        bigBufPtr += rounf(sizeof(Setup), 4);
    }
};
```

Такой трюк успешно работает, но создает проблемы фрагментации памяти.

Чтобы избавиться от фрагментации, мы воспользуемся разновидностью алгоритма управления памятью методом близнецов. Этот алгоритм получил известность после выхода книги [3], хотя ее автор утверждает, что алгоритм был опубликован Гарри Марковицем (Harry Markowitz) и Кеннетом Ноултоном (Kenneth Knowlton) в начале 1960-х годов. Приведенный ниже код базируется на материале [3].

В алгоритме близнецов на запрос блока возвращается блок, размер которого равен следующей степени двойки. Неиспользованные блоки связываются в список. Если рядом со свободным блоком находится смежный блок, освобожденный ранее, эти два блока объединяются; тем самым эффект фрагментации памяти ослабляется. Интересное свойство алгоритма близнецов состоит в том, что по адресу и размеру освободившегося блока можно вычислить адрес прилегающего блока; это упрощает проверку возможности слияния блоков.

Вместо того чтобы искать блок оптимального размера, в представленном решении алгоритм близнецов применяется для поиска «наихудшего соответствия» — запрос на выделение буфера сообщение удовлетворяется наибольшим из доступных блоков. Однако после идентификации такого блока алгоритм выделения памяти приостанавливается и дает возможность выполниться «виртуальному конструктору». После завершения разбора сообщения алгоритм знает размер объекта, и управление возвращается распределителю памяти, который усекает большой блок до ближайшего блока размером 2^k, вмещающего объект.

После освобождения блок пользуется всеми преимуществами метода близнецов по слиянию смежных блоков в один блок большего размера. Фрагментация памяти существенно снижается (предполагается, что пул памяти гораздо больше размера наибольшего сообщения), а гибкость «виртуальных конструкторов» сохраняется в полной мере.

В листинге 5.7 представлен интерфейс базового класса LAPD. В нем приведены интерфейсы операторов new и delete, а также конструктор с параметром const char*, используемый для построения сообщений. Также предусмотрен конструктор по умолчанию, который будет вызываться из конструкторов производных классов. Он должен определяться как пустая операция, так как вызывается для ранее инициализированного объекта. Также стоит заметить, что конструкторы этого базового класса вызываются дважды (LAPD(const char*) — в направлении пользователя, LAPD() — из производного класса), поэтому всем членам LAPD, инициализированным первым конструктором, будут возвращены значения по умолчанию в результате вызова второго. Другими словами, этот базовый класс не должен содержать внутренние экземпляры объектов.

Листинг 5.7. Класс LAPD

```
class LAPD {
    friend class LAPDMemoryManager;
  public:
    virtual int size() { return 0; }
    void *operator new(size_t);
    void operator delete(void *);
```

```
  LAPD(const char *const);
protected:
  LAPD() { /* Пусто */ }
private:
  // Как и в алгоритме близнецов, каждый узел может
  // представлять либо свободный, либо выделенный блок.
  // Выделенные блоки не требуют дополнительных ресурсов,
  // необходимых для ведения связанных списков
union {
  struct {
    unsigned char flag;
    struct {
      unsigned int sapi:6;
      unsigned int commandResponse:1;
      unsigned int zero:1;
      unsigned int tei:7;
      unsigned int ext:1;
    } address;
    unsigned char control;
  } header;
  struct {
    LAPD *linkf, *linkb;
    unsigned short size;
  } minfo;
};
// И выделенные, и свободные блоки должны содержать
// это поле, необходимое для уборки мусора.
unsigned char tag;  // Метка, назначаемая алгоритмом близнецов
int performCRCCheck() { /* ... */ }
};
```

Большая часть внутренних данных класса предназначена для хранения полей заголовка LAPD и информации динамического управления памятью. Поскольку многие поля управления памятью требуются только после того, как объект возвращается в список свободных блоков, для организации совместного доступа к их памяти внутри экземпляра приложения используется объединение (union). Поле tag должно присутствовать в таких объектах как во время их активности в приложении, так и во время нахождения в списке свободных блоков; по этой метке уборщик мусора метода близнецов определяет, был ли выделен соответствующий блок памяти.

В листинге 5.8 показан класс, управляющий буферами памяти сообщений. Алгоритм близнецов поддерживает собственные структуры данных, скрытые от приложения, и эти структуры должны существовать до того, как приложение начнет выделять память для буферов сообщений. По этой причине реализация управления памятью была выделена в отдельный класс (вместо ее распространения через классы сообщений). Один глобальный экземпляр LAPDMemoryManager с именем manager обеспечивает управление памятью для всех сообщений LAPD.

Листинг 5.8. Класс управления памятью сообщений

```
inline int round(int a, int b) { return ((a+b-1)/b)*b; }

// Этот класс также можно было оформить вложенным
// по отношению к LAPD. Он используется для
// конструирования синглетного объекта, управляющего
// памятью сообщений. Он выделен в самостоятельный
// класс в основном для инициализации структур данных
// метода близнецов.

class LAPDMemoryManager {
friend LAPD;
public:
    LAPDMemoryManager() {
    LAPD * largestBlock();
    void allocateResizeBlock(int);
    void deallocateBlock(LAPD, int);

private:
    enum { MessageBufSize=4096, Log2MessageBufSize=12 };
    unsigned char availBuf[
   (1+Log2MessageBufSize)*round(sizeof(LAPD),4)];
    // Avail содержит объекты, служащие заголовками списков
    // для освобождения объектов разных размеров 2**k
    LAPD *avail;
    int savej;              // Сохранение контекста между вызовами
                            // largestBlock и allocateResizeBlock
    unsigned char buf[MessageBufSize];
    LAPD *buddy(int k, LAPD *l) {
        char *cp = (char *) l;
        return (LAPD*)(long(cp) ^ (1<<(k+1)));
    }
};

static LAPDMemoryManager manager;  // Синглетный объект
```

В интерфейс класса управления памятью входит конструктор по умолчанию, который просто инициализирует структуры данных алгоритма близнецов:

```
LAPDMemoryManager::LAPDMemoryManager() {
    // Инициализация структур данных согласно [3]
    LAPD* buf = (LAPD*)this->buf;
    avail = (LAPD*)availBuf;
    avail[Log2MessageBufSize].minfo.linkf = 0;
    avail[Log2MessageBufSize].minfo.linkb = 0;
    buf[0].minfo.linkf = buf[0].minfo.linkb =
      &avail[Log2MessageBufSize];
    buf[0].tag = 1;
    buf[0].minfo.size = Log2MessageBufSize;
```

```
    for (int k = 0; k < Log2MessageBufSize; k++) {
        avail[k].minfo.linkf = avail[k].minfo.linkb =
          &avail[k];
    }
}
```

Алгоритм выделения памяти разбит на две функции и видоизменен таким образом, чтобы объект мог увеличиться в размерах после первой фазы выделения памяти. Функция largestBlock возвращает указатель на наибольший свободный блок в пуле:

```
LAPD *
LAPDMemoryManager::largestBlock() {
    for (int k = Log2MessageBufSize; k >= 0; --k) {
        if (avail[k].minfo.linkf != &avail[k]) {
            savej = k;
            return avail[k].minfo.linkf;
        }
    }
    return 0;
}
```

Закрытый вектор avail содержит объекты, служащие заголовками списков свободных объектов разных размеров (равных степеням двойки). Стандартный алгоритм близнецов просматривает этот вектор в поисках блока минимального размера, не меньшего заданного, а затем последовательно делит этот блок пополам, чтобы обеспечить наилучшее совпадение с запрашиваемым объемом. Поскольку класс LAPD не знает заранее, сколько памяти может понадобиться, измененная версия алгоритма возвращает наибольший свободный блок. Сокращение его размеров будет согласовываться между диспетчером памяти и производными классами LAPD.

Функция deallocateBlock возвращает блок в пул и объединяет его со смежными свободными блоками, если таковые найдутся. Ее реализация полностью соответствует описанной в [3]. Функция allocateResizeBlock содержит вторую часть реализации алгоритма близнецов; она тоже просто следует представленному в [3] алгоритму (листинг 5.9).

Листинг 5.9. Функции выделения и освобождения памяти для класса LAPD

```
void
LAPDMemoryManager::allocateResizeBlock(int sizeNeeded) {
    // В основном соответствует описанию из [3]
    int k, j = savej;
    // Округление в большую сторону до ближайшего 2**n
    for (int i = 1; i < Log2MessageBufSize; i++) {
        k = 1 << i;
        if (k > sizeNeeded) break;
    }
```

продолжение

Листинг 5.9 (*продолжение*)

```
    LAPD *l = avail[j].minfo.linkf;
    avail[j].minfo.linkf = l->minfo.linkf;
    l->minfo.linkf->minfo.linkb = &avail[j];
    for(l->tag = 0; j - k; ) {
        --j;
        LAPD *p = (LAPD*)(((char *)l) + (1 << j));
        p->tag = 1;
        p->minfo.size = j;
        p->minfo.linkf = &avail[j];
        p->minfo.linkb = &avail[j];
        avail[j].minfo.linkf = p;
        avail[j].minfo.linkb = p;
    }
}
void
LAPDMemoryManager::deallocateBlock(LAPD *l, int k) {
    for(;;) {
        LAPD *p = buddy(k,l);
        if (k==Log2MessageBufSize || p->tag == 0 ||
          p->minfo.size != k) {
            break;
        }
        p->minfo.linkb->minfo.linkf = p->minfo.linkf;
        p->minfo.linkf->minfo.linkb = p->minfo.linkb;
        ++k;
        if (p < l) l = p;
    }
    l->tag = 1;
    l->minfo.linkf = avail[k].minfo.linkf;
    avail[k].minfo.linkb = l;
    l->minfo.size = k;
    l->minfo.linkb = &avail[k];
    avail[k].minfo.linkf = l;
}
```

В закрытую реализацию LAPDMemoryManager входит переменная savej, используемая для хранения значения индекса avail между двумя фазами работы алгоритма выделения памяти. По индексу определяется размер и местонахождение текущего блока. Мы также находим спецификацию размера буфера и значение его двоичного логарифма, используемое алгоритмом выделения памяти. Сам буфер buf инкапсулируется в объекте диспетчера памяти. Вектор заголовков availBuf объявлен как анонимный байтовый вектор, но мы обращаемся к нему через указатель avail так, словно он является вектором объектов LAPD. Картину завершает функция buddy, предназначенная для идентификации смежных блоков, которые могут быть объединены. Переносимость кода buddy не идеальна; указатель char* преобразуется в тип long для выполнения операции «исключающего ИЛИ», задейство-

ванной в работе алгоритма близнецов. Возможно, программу придется подкорректировать для платформ, на которых длина указателя превышает длину long.

Итак, фундамент управления памятью успешно заложен; программа может находиться в библиотеке, скрытой от пользователя (за одним исключением, о котором речь пойдет далее). Прикладной программист создает класс сообщения, объявляя его производным от LAPD, как это сделано для классов Setup и Conn в листинге 5.10. Программная реализация тривиальна, если не считать вызова allocateResizeBlock в конструкторе, запускающего вторую фазу работы алгоритма близнецов для подгонки размера блока под фактический размер сообщения.

Листинг 5.10. Классы сообщений Setup и Connect протоколов уровня 3

```
class Setup: public LAPD {
friend LAPD;
private:
    struct SetupPacketBody {
        unsigned char rep[4096];
    };
    SetupPacketBody body;
    int size() { return sizeof(Setup); }
    Setup(const char *m): LAPD() {
        manager.allocateResizeBlock( size() );
        ::memcpy(&body, m+sizeof(header), sizeof(body));
        // ...
    }
};

class Conn: public LAPD {
friend LAPD;
private:
    struct ConnPacketBody {
        unsigned char rep[4096];
    };
    ConnPacketBody body;
    int size() { return sizeof(Conn); }
    Conn(const char *m): LAPD() {
        manager.allocateResizeBlock( size() );
        ::memcpy(&body, m+sizeof(header), sizeof(body));
    }
};
```

Обратите внимание: при проектировании классов сообщений мы различаем *тело* сообщения (протокол уровня 3 модели ISO) и абстракцию, соответствующую тем же данным «в процессе передачи» с битами заголовка и заключительной части и т. д. Тело называется SetupPacketBody, а абстракция — просто Setup. В листинге 5.10 показана аналогичная конфигурация для класса, обрабатывающего сообщения Conn (запросы на подключение).

Теперь разберемся, как строятся сообщения. Прикладная программа читает поток байтов из входного канала данных и передает его в параметре конструктора LAPD (см. листинг 5.10). Конструктор LAPD::LAPD(const char*) должен определить тип сообщения, чтобы вызвать подходящий конструктор производного класса. Вызов может выглядеть так:

```
LAPD *message = new LAPD(bytepointer);
```

Эта конструкция вызовет оператор new для выделения памяти под объект. Вызываемый оператор, специально определенный для класса LAPD, запрашивает адрес наибольшего свободного блока у диспетчера памяти и возвращает полученное значение:

```
void* LAPD::operator new(size_t /* не используется */) {
    return manager.largestBlock();
    // Завершается в конструкторе производного класса
}
```

После того как выделение памяти завершится, конструктор LAPD может заняться определением типа сообщения и вызовом соответствующего конструктора производного класса. Для наглядности давайте предположим, что тип сообщения определяется четвертым байтом сообщения (листинг 5.11). После идентификации типа сообщения конструктор LAPD напрямую вызывает конструктор соответствующего производного класса. Следующий синтаксис обеспечивает вызов перегруженного определения new:

```
(void) new(this) производный_класс (arg)
```

Теперь само перегруженное определение new:

```
// Оператор позволяет объекту иерархии LAPD
// заменить себя объектом одного из своих
// производных классов
inline void *operator new(size_t, LAPD* l) { return l; }
```

Другими словами, производный класс вынужден разместить свой экземпляр по тому же адресу, по которому находится экземпляр базового класса, то есть просто заменить существующий экземпляр. Помните, что это происходит на нижней границе наибольшего свободного блока, известного распределителю памяти.

Листинг 5.11. Тело конструктора объектов сообщений

```
LAPD::LAPD(const char *bits) {
    ::memcpy(&header, bits, sizeof(header));
    performCRCCheck();

    // Определение типа сообщений по четвертому октету
    // (байту) содержимого сообщения
    switch((bits + sizeof(header))[3]) {
    case SETUP:
```

```
        (void) ::new(this) Setup(bits); break;
    case CONN:
        (void) ::new(this) Conn(bits); break;
    case ALERT:
        (void) ::new(this) Alert(bits); break;
    ...
    default:
        error("invalid message"); break;
    }
}
```

В случае дальнейшего уточнения типов сообщений класс, производный от LAPD, может перепоручить более детализированную обработку сообщений одному из своих производных классов. Рано или поздно уточнение закончится, и конструктор производного класса вызовет функцию allocateResizeBlock диспетчера памяти для окончательного утверждения размера блока и реорганизации списков свободных блоков.

Освобождение памяти происходит еще проще. Каждое сообщение освобождает свою память оператором LAPD::operator delete, который в свою очередь вызывает функцию deallocateBlock диспетчера памяти для обработки блока по алгоритму близнецов:

```
void
LAPD::operator delete(void *l) {
    manager.deallocateBlock((LAPD*)l, ((LAPD*)l)->size());
}
```

Объекты переменного размера

Предыдущий пример можно расширить, чтобы создавать объекты переменного размера в заданном классе. Например, такая возможность пригодится при обработке сообщений, тело которых может иметь разные размеры.

Решение, первоначально предложенное в [2], использоваться не может, поскольку присваивание значения this было запрещено в C++ версии 2.0 и последующих версиях. Кстати, одной из причин для включения оператора new был поиск более общего решения этой задачи. Класс может позаботиться о том, чтобы все его экземпляры имели заданный размер:

```
class LAPD {
public:
    // Все объекты LAPD имеют размер 128 байт
    void *operator new(size_t) { return ::new[128]; }
    ...
};
```

Тем не менее, такое решение имеет ряд недостатков. Во-первых, оператор new не имеет доступа к параметрам конструктора, поэтому размер объекта может передаваться только через глобальные данные. Во-вторых, конструктор выполняется

слишком поздно, чтобы как-либо повлиять на размер объекта; оператор new завершает свою работу до выполнения тела конструктора. В-третьих, не существует очевидных способов регулировки размеров объекта в других классах, помимо базового.

Все перечисленные проблемы решаются применением алгоритма близнецов. Как и в случае с исходным классом LAPD, определение размера объекта откладывается до последних моментов его инициализации. Диспетчер памяти получает информацию о том, сколько *всего* байтов должно быть выделено для объекта, поэтому объект может вырасти на произвольную величину сверх размера, диктуемого его классом. В листинге 5.12 класс Conn содержит ранее вычисленное значение bodySize, содержащее размер тела сообщения. На основании этого размера можно вычислить общий размер объекта. Вызовом memcpy в конструкторе тело сообщения копируется в буфер bodyStorage, который ведет себя как вектор переменного размера. После подготовки объекта распределителю памяти отдается приказ зафиксировать объект на размере size(), в который входит и память для тела сообщения.

Листинг 5.12. Класс Conn с динамическим изменением размера

```
class Conn: public LAPD {
friend LAPD;
private:
    struct ConnPacketBody {
        unsigned char rep[4096];
    } *body;
    int bodySize;
    int size() { return sizeof(Conn) + bodySize; }
    Conn(const char *m): LAPD() {
        // ...
        ::memcpy(bodyStorage, m+sizeof(header), bodySize);
        body = (ConnPacketBody*) bodyStorage;
        manager.allocateResizeBlock( size() );
    }
    char bodyStorage[1];
};
```

Учтите, что классы, использующие эту методику, требуют большей дисциплины, чем конкретные типы данных. Такие классы не создают обычные объекты, которые могут создаваться в любом контексте. Их экземпляры могут размещаться только в заранее выделенном блоке памяти, адрес которого передается конструктору.

Делегирование и классы конвертов

Динамическая имитация наследования на стадии выполнения может производиться с применением *делегирования*. Делегирование открывает перед экземплярами объектов те же возможности, что и наследование перед классами — то есть

возможности совместного использования кода и автоматического применения кода одной абстракции для обработки запросов, обращенных к другой абстракции. Делегирование основано на связи между функцией класса одного объекта и контекста, предоставленного другим объектом. В частности, делегирование поддерживается языками программирования Actors [4] и self [5].

В C++ отношения наследования фиксируются на стадии компиляции. Прямая поддержка делегирования в языке отсутствует, поскольку это привело бы к существенному повышению затрат ресурсов на работу системы контроля типов, но многие аспекты делегирования имитируются перенаправлением вызовов функций одного объекта другому объекту. Перенаправление может осуществляться на уровне классов, как и при наследовании. Возможно, вам даже захочется использовать механизм перенаправления вызовов для имитации наследования, чтобы реализовать преимущества системы выделения подтипов в сочетании с гибкостью привязки символьного уровня на стадии выполнения. Тем не менее, реализация такого решения получается весьма громоздкой:

```
class A {
public:
    void a();               // Функции,
    void b();               // специфические
    void c();               // для класса A
    A(char *r) {
        cp = new char[::strlen(r)+1]; ::strcpy(cp,r);
    }
private:
    char *cp;
};

class B {
public:
    void a()            { delegate->a(); } // Некрасиво
    void b()            { delegate->b(); } // Некрасиво
    void c()            { delegate->c(); } // Некрасиво
    void d();           // Функция, специфическая для класса B
    B(char *r = "")     { delegate = new A(r); }
private:
    A *delegate;
};
```

Если бы мы применили наследование, унаследованные функции автоматически перешли бы в унаследованный класс. Дублирование интерфейса A в B усложняет как исходное кодирование, так и доработку программы: все изменения программы должны быть согласованы между классами.

Многие проблемы, возникающие из-за дублирования интерфейса, в C ++ решаются перегрузкой оператора -> и полным контролем программиста над семантикой разыменования указателей на внешний объект. Методика реализации сходна

с той, которая использовалась при подсчете указателей (см. с. 78). Такое решение позволяет одному объекту прозрачно «перенаправлять» операции другому объекту:

```
class A {
public:
    void a();              // Функции,
    void b();              // специфические
    void c();              // для класса A
    A(char *r) {
        cp = new char[::strlen(r)+1]; ::strcpy(cp,r);
    }
private:
    char *cp;
};

class B {
public:
    // Явное перенаправление вызовов a, b  и c становится лишним
    void d();         // Функция, специфическая для класса B
    A *operator->()  { return delegate; }
    B(char *c = "")  { delegate = new A(c); }
private:
    A *delegate;
};
```

Изменение указателя **delegate** позволяет в любой момент перевести делегирование вызовов на другой объект.

Теперь в класс A можно свободно добавлять новые функции, не беспокоясь о координации изменений с классом B. Также обратите внимание на следующие обстоятельства.

- Изменение класса A не требует перекомпиляции кода B (за исключением конструктора).
- Возможно, вы не сразу привыкнете к синтаксису объявления и использования таких объектов:

```
int main() {
    B aBInstance("hello world\n");
    aBInstance.d(); // Точечная запись (используется редко)
    aBInstance->a();
    // Оператор delete не вызывается для aBInstance:
    // это не указатель
    return 0;
}
```

Пользователь класса обязан помнить, перенаправлялся вызов функции или нет, и задействовать соответствующий вариант синтаксиса. Если это порождает слишком много неудобств, вам придется отказаться от перегрузки функции

operator-> и использовать перенаправление на уровне отдельных функций, как в приведенном ранее «некрасивом» примере. Конечно, при перенаправлении всех вызовов синтаксис вызова функций класса становится единообразным.

Если бы могли перегрузить функцию operator., многие проблемы исчезли бы сами собой, но точка (.) — один из немногих операторов C++, которые не могут перегружаться.

- Для вызова «настоящих» функций внутреннего класса вместо «точечной записи» необходимо использовать оператор ->. Точечная запись требуется только для функций, которые никогда не делегируются, но определяются во внешнем классе.
- Операторы (например, математические), перегружаемые в классе A, также должны определяться в B:

```
int A::operator+(const A &a) const {
    int q;
    ...
    return q;
}

int B::operator+(const A &a) const {
    return delegate->operator+(a);
}
```

Другой пример приведен в листинге 5.13. Здесь класс SeaPlane делегирует свою функциональность либо Boat, либо Plane в зависимости от текущего «режима».

Листинг 5.13. Класс SeaPlane с переменным делегированием

```
class Vehicle {
public:
    virtual int mpg() { return 0; }
    virtual void takeOff() { hostVehicle->takeOff(); }
    virtual void swim(){ hostVehicle->swim(); }
    Vehicle(Vehicle *host) { hostVehicle = host; }
private:
    Vehicle *hostVehicle;
};

class Plane: public Vehicle {
public:
    int mpg() { return 10; }
    Plane(Vehicle *v) : Vehicle(v) { }
};

class Boat: public Vehicle {
public:
    int mpg() { return 3; }
```

продолжение

Листинг 5.13 (*продолжение*)

```
    Boat(Vehicle *v) : Vehicle(v) { }
};

class SeaPlane: public Vehicle {
public:
    SeaPlane(): Vehicle(0) {
        currentMode = boat = new Boat(this);
        plane = new Plane(this);
    }
    ~SeaPlane() { delete boat; delete plane; }
    void takeOff() { currentMode = plane; }
    void swim() { currentMode = boat; }
    Vehicle *operator->() { return currentMode; }
private:
    Vehicle *boat, *plane, *currentMode;
};

int main() {
    SeaPlane seaPlane;
    cout << seaPlane->mpg() << endl;
    seaPlane->takeOff();    cout << seaPlane->mpg() << endl;
    seaPlane->swim();    cout << seaPlane->mpg() << endl;
}
```

ПРИМЕЧАНИЕ

Используйте эту идиому для часто изменяющихся парных классов «конверт/письмо», чтобы избежать лишних обновлений интерфейсов обоих классов (конверта и письма). Кроме того, данная идиома заложена в основу многих нетривиальных идиом, таких как символические идиомы из главы 9.

Итераторы и курсоры

Контейнерные классы (множества, списки, деревья и т. д.) часто применяются в программах C++. Они могут совместно использоваться несколькими классами; например список ресурсов в программе-имитаторе может применяться классом-создателем, классом-потребителем и классом, управляющим ходом имитации. Каждому из этих классов может потребоваться последовательно перебрать все элементы списка, по разным причинам: создателю — для изменения состояния элементов, ранее поставленных в очередь; потребителю — для поиска элемента с некоторым приоритетом или другой характеристикой; управляющему классу — для вывода содержимого очереди по запросу пользователя.

Все эти операции перебора являются *неразрушающими*, то есть сами по себе они не отражаются на содержимом списка. При поочередном выполнении таких операций объектами разных классов принципиально то, что внутренние данные списка не меняются. Тем не менее, данные, необходимые для последовательного перебора всех элементов, являются закрытыми для класса списка. Чтобы разные

объекты (создатель списка, потребитель и управляющий класс) могли перебирать элементы независимо друг от друга, каждый из них должен хранить собственную копию данных «текущей позиции» списка. Эти данные хранятся в специальном объекте, называемом *итератором*, или *курсором*, и позволяющем своему владельцу последовательно перебирать элементы составной структуры данных.

Итераторы составляют особый подвид классов-манипуляторов. В отличие от манипуляторов, упоминавшихся ранее в этой главе, итераторы являются вспомогательными классами, которые не заменяют, а лишь дополняют интерфейс классатела. Итераторы обычно объявляются друзьями (friend) своих классов-тел.

Рассмотрим класс Queue, подробно описанный в конце этой главы. После компиляции и запуска программа выводит следующий результат:

```
1 2 3
leaving: 1 2 3 4 5
6 7 8 9 10 11 12
leaving: 6 7
8 9 10 11 12
leaving: 8 9
```

Итератор Queue также может рассматриваться как «курсор» для последовательного перебора Queue. Каждый курсор, созданный для объекта Queue, работает независимо от других курсоров, и его применение ни при каких условиях не отражается на базовом содержимом Queue.

Программа должна работать даже в том случае, если содержимое Queue изменяется между фазами работы итератора. Класс QueueIterator не запоминает текущее состояние Queue между вызовами своих функций.

В нашем примере итератор инициализируется константной ссылкой на Queue, чтобы упростить проверку существования Queue перед началом перебора.

5.6. Функторы

Как было отмечено в разделе 2.10, указатели на функции классов могут использоваться для работы с разными версиями функции через общий интерфейс. Тем не менее, синтаксис вызова функции, «параметризованной» подобным способом, выглядит довольно неуклюже:

```
...afilter.*(afilter.current)(t)...
```

В его реализации обычно задействованы малопонятные конструкции C++. Наследование с виртуальными функциями позволяет упростить вызов.

Вспомните пример с электрической цепью, состоящей из индуктора, конденсатора и резистора (см. раздел 2.10). В зависимости от параметров этих компонентов цепь может реагировать на поданный импульс одним из трех способов. Для каждого из трех вариантов реакции использовалась отдельная функция.

Давайте попробуем пойти по другому пути: каждый вариант реакции будет запрограммирован в отдельном *классе*, а не в отдельной функции. В данном случае мы имеем дело с нетривиальной архитектурной конструкцией, называемой *функтором*. Функтор представляет собой функцию, которая ведет себя как объект (или объект, который ведет себя как функция). Функторы исполняют роль функций, но при этом создаются, передаются в параметрах и т. д. как объекты. Реализация функторов в C++ предельно проста: функтор оформляется в виде класса, содержащего всего одну функцию. Объект функтора может использоваться для определения каждой из трех возможных реакций цепи.

ПРИМЕЧАНИЕ

Используйте функторы везде, где были бы уместны указатели на функции (и особенно указатели на функции классов).

Что же дает применение функторов? Во-первых, наследование позволяет выделить общий код в базовый класс. Подвыражения, входящие в формулы, могут быть достаточно сложными, что оправдывает их оформление в виде самостоятельных функций. В нашем конкретном примере подвыражения (такие, как ω0) входят сразу в несколько формул и четко соответствуют конструкциям предметной области. Более того, в распоряжении программиста появляется языковая конструкция для объединения логически связанных функций: все функторы, взаимозаменяемые в заданном контексте, наследуют от общего базового класса.

Во-вторых, применение виртуальных функций с наследованием обеспечивает ту же гибкость на стадии выполнения, которая достигается применением указателей на функции. Вместо того чтобы использовать указатель на функцию для обращения к «настоящей» функции, мы заменяем его соответствующим объектом функции. Функция вызывается так, словно она находится в базовом классе, а выбор нужной функции на стадии выполнения обеспечивается механизмом виртуальных функций.

При реализации данной задачи возникает одно затруднение. Мы легко спроектируем три класса, соответствующих чрезмерному, недостаточному и критическому затуханию, и базовый класс с общей функциональностью. Остается понять, как *выбрать* создаваемый функтор из трех вариантов. При наличии указателя на функцию все данные находились «под рукой» в конструкторе, и мы могли использовать их для выбора функции. С переходом на функторы само поведение характеризуется созданным объектом; к моменту выполнения конструктора уже поздно принимать решения относительно того, какой объект создавать. На помощь приходит идиома «конверт/письмо» и модель виртуальных конструкторов, обеспечивающая дополнительный логический уровень, имитирующий создание объектов с динамической типизацией.

В листинге 5.14 приведены классы функторов для трех альтернативных режимов реакции цепи. Класс `SeriesRLCStepResponse` является базовым классом для всех классов писем, а также классом конверта. В качестве базового класса для писем он содержит виртуальную функцию `current`, переопределяемую для каждого отдельного случая в производных классах. Базовый класс имеет защищенные поля общих для всех производных классов свойств.

Листинг 5.14. Базовый класс функтора и классы трех режимов имитации

```
#include <complex.h>
typedef double time;

class SeriesRLCStepResponse {
public:
    virtual complex current(time t) {
        return object->current(t);
    };
    double frequency() const { return 1.0 / sqrt (L * C); }
    SeriesRLCStepResponse(double r, double l,
        double c, double initialCurrent, short isenvelope=1);
protected:
    double R, L, C, currentT0;
    double alpha;
    complex omegad, a1, b1, a2, b2, s1, s2;
private:
    SeriesRLCStepResponse *object;
};

class UnderDampedResponse: public SeriesRLCStepResponse {
public:
    UnderDampedResponse(double r, double l, double c, double i) :
    SeriesRLCStepResponse(r,l,c,i,0) { }
    complex current(time t) {
        return exp(-alpha * t) * (b1 * cos(omegad * t) +
          b2 * sin(omegad * t));
    }
};

class OverDampedResponse: public SeriesRLCStepResponse {
public:
    OverDampedResponse(double r, double l, double c, double i) :
        SeriesRLCStepResponse(r,l,c,i,0) { }
    complex current(time t) {
        return a1 * exp(s1 * t) + a2 * exp(s2 * t);
    }
};

class CriticallyDampedResponse: public SeriesRLCStepResponse {
public:
    CriticallyDampedResponse(double r, double l, double c, double i) :
    SeriesRLCStepResponse(r,l,c,i,0) { }
    complex criticallyDampedResponse(time t) {
        return exp(-alpha * t) * (a1 * t + a2);
    }
};
```

Будучи классом-конвертом, класс SeriesRLCStepResponse содержит указатель object со ссылкой на письмо, в котором обеспечивается специфическая обработка для конкретных режимов реакции цепи. Поле объявлено закрытым, поэтому оно не экспортируется в производных классах. Конверт перенаправляет вызовы current одноименной функции письма — виртуальной функции, поведение которой выбирается на стадии выполнения.

В листинге 5.14 приведены классы функторов для всех трех альтернативных режимов реакции цепи. Для имеющейся модели цепи создается объект одного из этих трех классов. Каждый класс содержит две открытые функции: конструктор, необходимый для сбора параметров, и функцию current, возвращающую текущее значение как функцию времени. Функция current объявлена виртуальной в базовом классе SeriesRLCSteeepResponse.

Вернемся к конструктору базового класса, в котором происходит самое интересное (листинг 5.15). На основании переданных аргументов конструктор выбирает режим реакции цепи, а затем связывает поле object базового класса с вновь созданным объектом соответствующего класса, производного от SeriesRLCResponse.

Листинг 5.15. Конструктор, создающий объект реакции цепи

```
SeriesRLCStepResponse::SeriesRLCStepResponse(
    double r, double l, double c, double initialCurrent,
    short isenvelope) {
    R = r; L = l; C = c; currentT0 = initialCurrent;
    alpha = R / (L + L);
    // Вычисление a1, b1, a2, b2 и т. д.
    if (isenvelope) {
        if (alpha < frequency()) {
            object = new UnderDampedResponse(
              r, l, c, initialCurrent);
        } else if (alpha > frequency()) {
            object = new OverDampedResponse(
              r, l, c, initialCurrent);
        } else {
            object = new CriticallyDampedResponse(
              r, l, c, initialCurrent);
        }
    } else {
        omegad = sqrt(1.0/(L*C) - (alpha * alpha));
    }
}
```

Стоит заметить, что конструктор может функционировать в одном из двух режимов: он вызывается либо для того, чтобы объект работал как конверт, либо для создания подобъекта базового класса в производном классе. Если параметр isenvelope равен 1, конструктор работает в режиме создания самостоятельного объекта. Такой объект анализирует параметры цепи и создает объект письма, моделирующий ее поведение. Если параметр isenvelope равен 0, то конструктор

считает, что он вызывается из производного класса, и в этом случае он ограничивается вычислением ω_d. Для удобства пользователей класса параметр isenvelope по умолчанию равен 1; производные классы должны подменять это значение среди параметров, передаваемых конструктору базовому классу.

Функция main проста и понятна:

```
int main() {
    double R, L, C, IO;
    cin >> R >> L >> C >> IO;
    SeriesRLCStepResponse afilter(R, L, C, IO);
    for (time t = 1.0; t < 100; t += 1.0) {
        cout << afilter.current(t) << endl;
    }
    return 0;
}
```

Эта функция просто создает объект SeriesRLCStepResponse с подходящими параметрами, а затем многократно вызывает функцию current для разных значений t. С точки зрения пользователя работать с этим классом ничуть не сложнее, чем с исходным примером (см. раздел 2.10). Тем не менее, такая реализация понятнее и обладает более эстетичной архитектурой. Многие традиционные применения указателей на функции могут быть реализованы с помощью виртуальных функций C++, причем программа от этого становится проще, а ее отладка и сопровождение упрощаются.

Функциональное и аппликативное программирование

Предыдущий пример дает хорошее представление на концептуальном уровне, но как бы нам ни хотелось, функции все равно не похожи на объекты. Механизм перегрузки C++ позволяет приблизить функции к объектам на синтаксическом уровне, придавая идиоме функторов логическую законченность.

Помимо самой функции current, мы предоставляем доступ к ней с использованием оператора вызова функции ():

```
class SeriesRLCStepResponse {
public:
    virtual complex operator () (time t) {
      return current(t);
    }
    virtual complex current(time t) {
      return object->current(t);
    }
    double frequency() const { return 1.0 / (L * C); }
    SeriesRLCStepResponse(double, double,
      double, double, short = 1);
    ...
};
```

Этот фильтр можно использовать так, словно он является функцией, определенной для объекта:

```
SeriesRLCStepResponse afilter(R, L, C, IO);
for (time t = 1.0; t < 100; t += 1.0) {
    cout << afilter(t) << endl;
}
```

На первый взгляд этот механизм кажется простым синтаксическим удобством, но на самом деле он имеет ряд мощных практических применений. Вспомните о парадигме *аппликативного программирования*, в котором основной абстракцией («значением») являются функции, а основной операцией — применение функции к чему-либо, обычно к результату вызова другой функции. В результате применения функции может быть получен объект данных или даже другая функция!

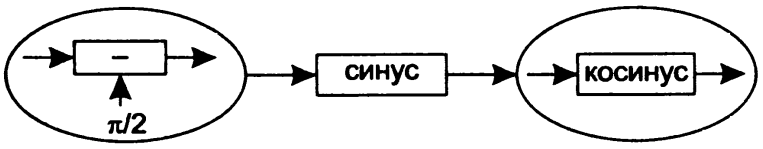
В этом разделе мы рассмотрим применение функторов при моделировании электронных фильтров. Фильтр представляет собой электронное устройство, пропускающее сигналы в заданном диапазоне и блокирующее прохождение сигналов других частот. Например, в телевизоре имеются *полосовые фильтры*, которые пропускают сигналы определенных каналов на экран и подавляют все остальные. На вход антенны можно установить узкополосный фильтр для избирательного «гашения» частоты расположенного поблизости радиопередатчика, вызывающего помехи на телевизионном изображении. Мы смоделируем некоторые аспекты поведения таких фильтров при помощи функторов. Каждый объект (функтор) инициализируется параметрами фильтра, определяющими его специфические выходные характеристики. Функтор получает входные данные, обрабатывает их и генерирует результат.

Чтобы имитировать поведение фильтра в функторе, удобно рассматривать его как математическую абстракцию. К электронному фильтру можно относиться как к реализации математической функции, определяющей непрерывное отображение входных данных на выходные. Если подать на фильтр входной сигнал, фильтр трансформирует его и выдает результат, как любая математическая функция. Мы представим эту функцию в виде «материальной абстракции» — объекта, подобно тому, как реальный электронный фильтр является материальной сущностью.

Величина напряжения тоже моделируется при помощи функций. По условиям нашей задачи напряжение является постоянным по амплитуде и по частоте, хотя в действительности оно, скорее всего, более динамично.

В парадигме аппликативного программирования функция *применяется* к входным данным для получения результата. Применение фильтра к напряжению дает некий результат. Но каким должен быть результат применения одной функции к другой, или в нашем случае — применением одного фильтра к другому? Возьмем алгебраическое применение функции «синус» к функции «вычита-

ние π/2» независимо от числовых значений. Результатом является не конкретная величина, а функция «косинус»:



Если сигналы являются функциями, к ним можно применять другие функции, результатом чего также будут функции. В конечном счете этот принцип распространяется и на фильтры. В результате применения фильтра к напряжению также будет получено выходное напряжение с определенной амплитудой и частотой.

В листинге 5.16 приведен общий аппликативный класс, предназначенный как для представления напряжения, так и являющийся базовым классом для классов электронных фильтров. Он экспортирует тип Type, который используется классами фильтров для определения типа управляющего фильтра и последующего создания нового экземпляра класса фильтра на основе двух классов (своего и класса управляющего фильтра).

Листинг 5.16. Аппликативный класс для представления обобщенного электронного сигнала

```
typedef double Frequency;
typedef double Voltage;

class Value {
public:
    enum Type { T_LPF, T_HPF, T_BPF, T_Notch, T_Data };
    virtual Type type() { return T_Data; }
    virtual Frequency fl() { return omegal; }
    virtual void print() {
        cout << "Value " << volts << " volts at frequency " <<
            omegal << endl;
    }
    virtual Value *evaluate(Value* = 0) {
        return this;
    }
    Value(Voltage v=0, Frequency wl=0): volts(v), omegal(wl) {
    }
private:
    Frequency omegal;
    Voltage volts;
};

Value Zero(0);

Value *zero = &Zero;
```

Фильтр публикует свой тип при помощи функции type (далее объясняется, почему вместо виртуальной функции использовано поле типа). С любым сигналом и любым фильтром ассоциируется минимум одна частота; для сигналов — это характеристическая частота, а для фильтров — точка отсечения. Функция print генерирует выводимые данные. Наконец, при вызове функции evaluate сигнал или фильтр возвращает свое выходное напряжение, если входной сигнал уже подан.

В листинге 5.17 приведен код абстракции Filter, подменяющей все функции базового класса Value. Некоторые функции (evaluate, operator() и print) класс Filter объявляет чисто виртуальными; для таких функций производные классы должны предоставить собственные версии. В класс Filter включены также функция f2, используемая фильтрами с двумя частотами отсечения вместо одной, и закрытая переменная omega2 для хранения ее величины. Кроме того, появилась чисто виртуальная перегрузка операторной функции operator()(Value*); именно эта функция позволяет «вызывать» объект как функцию. При необходимости в класс можно включить несколько перегруженных вариантов функции operator() с разными списками параметров.

Листинг 5.17. Абстракция обобщенного электронного фильтра

```
class Filter: public Value {
public:
    Filter(Frequency w1, Frequency w2 = 0):
        Value(0, w1), omega2(w2), cachedInput(0) {
    }
    virtual Value *evaluate(Value* = 0) = 0;
    virtual Value *operator()(Value*) = 0;
    virtual void print() = 0;
    Frequency f2() { return omega2; }
    Type type() {
        if (cachedInput) return cachedInput->type();
        else return baseType;
    }
protected:
    Type myType, baseType;
    Value *cachedInput;
private:
    Frequency omega2;
};
```

Следующий новый класс, производный от Filter, моделирует *фильтр нижних частот*:

```
class LPF: public Filter {
public:
    Value *evaluate(Value* input = 0) {
        Value *f = cachedInput->evaluate(input);
        if (f->f1() < f1()) return f;
        else return zero;
```

```
    }
    Value *operator()(Value*);
    void print() {
        if (cachedInput) cout << "COMPOUND" << endl;
        else cout << "LPF(" << f1() << ")" << endl;
    }
    LPF(Frequency w1): Filter(w1), baseType(T_LPF) { }
};
```

Функция print и конструктор не требуют комментариев. Функция evaluate участвует в передаче запроса на получение выходного напряжения фильтра (или сигнала), присоединенного к входному сигналу. Запрос передается до тех пор, пока в цепочке не будет обнаружен объект сигнала. Частота этого сигнала проверяется на вхождение в полосу пропускания фильтра, после чего возвращается соответствующее значение.

Определение функции operator()(Value*) этого класса выглядит поинтереснее:

```
Value*
LPF::operator()(Value* f) {
    switch (f->type()) {
    case T_Data:
         myType = T_Data;
         cachedInput = f;
         return evaluate();
    case T_LPF:
         if (f->f1() > f1()) return this;
         else return f;
    case T_HPF:
         if (f->f1() > f1()) return new Notch(f1(), f->f1());
         else return new BPF(f->f1(), f1());
    case T_BPF:
         if (((Filter*)f)->f2() < f1()) return f;
         else return new BPF(f->f1(), f1());
    case T_Notch:
         cachedInput = f;
         return this;
    }
}
```

Параметр определяет входное «значение» (сигнал или фильтр), к которому применяется этот фильтр, а возвращаемое значение — сигнал или фильтр в зависимости от ввода. Оператор вызова функции обеспечивает классификацию входных данных для фильтра: если для них можно узнать напряжение (тип T_Data), то оператор просто проверяет, лежит ли частота сигнала выше точки отсечения, и возвращает значение сигнала без изменений. В противном случае функция пытается вернуть новый тип фильтра, характеризующий объединенное поведение самого себя и своих входных данных; то же самое происходило с функцией

«синус», которая возвращала функцию «косинус» при подаче на вход подходящей функции сдвига. Например, любая комбинация фильтров низких и высоких частот создает полосовой фильтр, который может оказаться вырожденным, если совокупность исходных фильтров не пропускает никаких частот. Если при таких условиях трудно вернуть один примитивный тип фильтра, фильтр просто возвращает самого себя, объединенного с другим фильтром. Полученный составной объект будет передаваться в нашей модели так, словно он является единым объектом. Если ввод позднее связать с объектом Value, то последующие вызовы evaluate вернут объект Value вместо составного объекта (ответом будет «значение»-сигнал, а не функция).

Обратите внимание: функция LPF::operator()(Value*) содержит команду switch по типу одного из своих параметров, хотя в начале главы использовать подобные конструкции не рекомендовалось. В данной ситуации применение команды switch оправдывается тем, что она предотвращает размножение открытых виртуальных функций. Реализация функции operator()(Value*) с применением механизма виртуальных функций выглядела бы примерно так:

```
Value *
Filter::operator()(Value *f) {
    return f->combineWithLPFoutPut(this);
}
```

(Кстати, этот код следовало бы переместить в класс Filter, потому что он является общим для всех классов фильтров.) Таким образом, LPF перепоручает задачу объединения двух функторов другому фильтру, участвующему в комбинации. Для каждой секции case где-то была бы создана новая виртуальная функция класса — так, секция case T_LPF породила бы следующие функции:

```
LPF::combineWithLPFoutPut
BPF::combineWithLPFoutPut
HPF::combineWithLPFoutPut
Value::combineWithLPFoutPut
```

Общий объем кода этих функций примерно равен объему кода в секциях case. При равенстве прочих факторов решение с командой switch обладает преимуществом: оно содержит меньше глобальных функций, чем решение с виртуальными функциями. В обоих случаях добавление новых типов фильтров требует примерно одинаковых хлопот.

При использовании механизма виртуальных функций функции выбираются на основании класса одного из операндов. При более общем подходе выбор функции должен зависеть от типов *двух* объектов, участвующих в операции. Этот механизм, называемый *механизмом мультиметодов*, не поддерживается в C++ напрямую. Мы еще вернемся к рассмотрению мультиметодов в разделе 9.7.

Остается лишь определить классы для оставшихся типов фильтров (реализация класса полосового фильтра BPF приведена в листинге 5.18) и использовать их для создания и тестирования системы фильтров.

Листинг 5.18. Реализация полосового фильтра

```
Value *
BPF::evaluate(Value *input = 0) {
    Value *f = cachedInput->evaluate(input);
    if (f->f1() > f1() && f->f1() < f2()) return f;
    else return zero;
}

Value *
BPF::operator()(Value* f) {
    switch(f->type()) {
    case T_LPF:
        if (f->f1() > f2()) return this;
        else return new BPF(f1(), f->f1());
    case T_HPF:
        if (f->f1() < f1()) return this;
        else return new BPF(f->f1(), f2());
    case T_BPF:
        Frequency lowfreq = f->f1();
        Frequency highfreq = ((Filter*)f)->f2();
        if (f1() > lowfreq) lowfreq = f1();
        if (f2() < highfreq) highfreq = f2();
        return new BPF(lowfreq, highfreq);
    case T_Notch:
        cachedInput = f;
        return this;
    case T_Data:
        myType = T_Data;
        cachedInput = f;
        return evaluate();
    }
}
```

В листинге 5.19 приведена простая программа для тестирования классов фильтров. Мы создаем три фильтра для низких частот, высоких частот и полосы частот. Фильтр низких частот пропускает сигналы с частотой ниже 8000 Гц; фильтр высоких частот пропускает частоты выше 1100 Гц, а полосовой фильтр — частоты в диапазоне от 1000 до 10000 Гц. Мы применяем полосовой фильтр к фильтру высоких частот (строка 7) и выводим результат (строка 8), получая

```
BpF(1100, 10000)
```

Листинг 5.19. Простая программа для тестирования аппликативных фильтров

```
1  int main() {
2      Value *v = new Value(100,1260); // Напряжение
                                       // и частота сигнала
3      BPF bpf(1000, 10000);    // Полосовой фильтр
```

продолжение

Листинг 5.19 (*продолжение*)

```
 4      HPF hpf(1100);           // Фильтр высоких частот
 5      LPF lpf(8000);           // Фильтр низких частот
 6      Filter *a;               // Указатель на фильтр
 7      a = (Filter*)bpf(&hpf);  // Применение полосового
                                 // фильтра к фильтру высоких
 8      a->print();              // частот: результат?
 9      (*a)(v)->print();        // Применение к сигналу
                                 // и вывод
10      a = (Filter*)(*a)(&lpf); // Применение к фильтру низких
11      a->print();              // частот: результат?
12      a = (Filter*)(*a)(v);    // Подача сигнала на
                                 // вход комбинации
13      a->print();              // с последующим выводом
14      lpf(&hpf)->print();      // Объединение фильтров низких
15      return 0;                // и высоких частот
16  }
```

Таким образом, в результате применения полосового фильтра к фильтру высоких частот создается единый объект-фильтр, пропускающий частоты в диапазоне от 1100 до 10 000 Гц. Поведение этого фильтра идентично поведению комбинации исходного полосового фильтра и фильтра высоких частот. В строке 9 этот фильтр применяется к сигналу. Выведенный результат выглядит так:

```
Value 100 volts at frequency 1260
```

Частота сигнала (1260) входит в полосу пропускания полосового фильтра (1100–10 000). В строке 10 этот фильтр применяется к фильтру низких частот, а в строке 11 выводится результат 11:

```
BPF(1100.8000)
```

Фильтр применяется к прежнему входному сигналу (строка 12), который и на этот раз попадает в полосу пропускания:

```
Value 100 volts at frequency 1260
```

Наконец, мы объединяем фильтры низких и высоких частот (строка 14) и выводим результат как полосовой фильтр:

```
BPF(1100.8000)
```

В альтернативном синтаксисе, более отвечающем духу аппликативного программирования, мы определяем семейство операторов apply. В первом параметре им передается указатель на объект функции, а затем следуют параметры самого функтора:

```
Value *apply(Value *object, Value *arg) {
    return (*object)(arg); }
Value *apply(Value *object, Value *arg1, Value *arg2) {
    return (*object)(arg1, arg2); }
```

Для удобства мы также определяем вспомогательный функтор printer:

```
class Printer: public Filter {
public:
    Type type() { return T_Printer; }
    void print() { }
    Value *evaluate(Value*) {
        // Ошибка
    }
    Value *operator()(Value* v) {
        v->print();
        return 0;
    }
} printerObject;
Printer *printer = &printerObject;
```

Теперь мы можем применить фильтр низких частот к фильтру высоких частот, затем применить полученный фильтр к входным данным, взять результат и усилить его втрое:

```
LPF *lpf = new LPF(w1);
HPF *hpf = new HPF(w2);
AMPLIFIER *amplifier = new Amplifier;
Value *inputValue, *three = new Value(3);
apply(
    printer,
    apply(
        amplifier,
        apply(
            apply(lpf, hpf);
            inputValue
        ),
        three
    )
);
```

С точки зрения семантики такая запись эквивалентна следующей:

```
amplifier((*((*lpf)(hpf)))(inputValue), three)->print();
```

5.7. Множественное наследование

В некоторых ситуациях одиночного наследования оказывается недостаточно, а новый класс очевидно должен наследовать свойства двух и более родительских классов. Если базовые классы известны на стадии компиляции, механизм *множественного наследования* C++ позволяет объединить поведение нескольких базовых классов в одном производном классе. Множественное наследование — мощный инструмент с богатой семантикой, способный напрямую определять

сложные структуры. Однако его следует осмотрительно применять в ситуациях, в которых одиночное наследование (или отсутствие наследования) ведет к утрате важных архитектурных связей.

Динамическое множественное наследование, впервые представленное в виде «примесей» в языке Flavors, также является полезной конструкцией для «настройки» поведения типа на стадии выполнения. Пользователю иногда требуется объединить поведение некоторых базовых классов, чтобы определить производный класс для конкретных целей, а затем создать объект этого класса в приложении. В C++ производные классы вместе со всеми возможными комбинациями базовых классов должны объявляться на стадии компиляции; программа выбирает подходящий класс на стадии выполнения и создает объект для приложения. Это может привести к стремительному росту числа производных классов. В языке Flavors такие типы строятся более динамично; чтобы изменить поведение базового класса, программа «примешивает» к нему аспекты поведения других «облегченных» базовых классов. Имитация динамического множественного наследования для C++ представлена в главе 10, а в этой главе рассматривается статическое множественное наследование.

Пример абстракции окна

В литературе множественное наследование часто рассматривается на примере классов поддержки терминального окна/экрана. Допустим, у нас имеется класс Window, используемый текстовым редактором для отображения содержимого файла на терминале:

```
class Window {
public:
    Window();
    virtual ~Window();
    virtual void addch(char) = 0;
    virtual void addstr(string) = 0;
    virtual void clear() = 0;
    virtual void scroll() = 0;
    ...
private:
    Color curColor;
};
```

Редактор должен поддерживать разные типы терминалов. Существуют две категории классов, производных от Window, для двух основных технологий многооконного интерфейса.

Первая категория CursesWindow реализует класс Window с использованием пакета Curses [6]:

```
#include <curses.h>

class CursesWindow : public Window {
```

```
public:
    void addch(char);
    void addstr(string);
    void scroll();
    void clear();
    ...
private:
    WINDOW *cursesWindow;
};
```

Вторая категория **XWindow** ориентирована на графическую систему X Window [7]:

```
#include <X11/X.h>

class XWindow : public Window {
public:
    void addch(char);
    void addstr(const string);
    void clear();
    void scroll();
    ...
protected:
    Display *disp;
    Window window;
    XWindowAttributes windowAttributes;
    GC gc;
    Cursor theCursor;
    unsigned long whitePixel, blackPixel;
```

Функции класса **Window** в основном объявляются виртуальными, что указывает на их фактическую реализацию в производных классах. Идея состоит в том, чтобы создать экземпляр класса CursesWindow или **XWindow**, который затем используется редактором. Также на C++ могут быть написаны другие программы, опирающиеся на переносимый интерфейс Window.

Хотя мы стремимся создать обобщенный класс **Window** для его последующего использования во многих программах, также желательно иметь специализированный класс **Window** для потребностей редактора. Класс **EditWindow** представляет собой класс **Window** с дополнительными атрибутами, присущими редактору, такими, как номера первой и последней строк на экране:

```
class EditWindow : public Window {
public:
    int topLine()      { return topLineVal; }
    int bottomLine()   { return bottomLineVal; }
    String fileName()  { return fileNameVal; }
    ...
    void scroll();     // Использует topLineVal и bottomLineVal
                       // помимо выполнения операций с Window
```

```
private:
    int topLineVal, bottomLineVal;
    PathName fileNameVal;
    Editor *editorVal;
protected:
    Cursor theCursor;  // Специальный курсор для редактирования
};
```

Класс EditWindow должен быть классом, производным от Window, но при этом он также должен вести себя как класс CursesWindow или XWindow (в зависимости от типа окна). Возникает вопрос — как организовать иерархию наследования?

Ответ: нужно создать класс, наследующий свойства как многооконного интерфейса, так и архитектуры редактора. В результате будет получен новый класс окна, обладающий свойствами обоих предков:

```
class EditWnd_Curses:
    public EditWindow, public CursesWindow {
};

EditWnd_Curses aNewWindow;
```

В принципе этого достаточно: новый производный класс наследует свойства обоих родителей. Тем не менее существуют две проблемы.

- Некоторые операции нового класса неоднозначны. Например, что должен означать вызов aNewWindow.scroll()? Следует ли выполнить операции scroll обоих родителей? И в каком порядке?
- Каждый из родителей содержит подобъект Window. Сколько экземпляров этого подобъекта должен содержать класс EditWnd_Curses — один или два?

В том, что касается первой проблемы, C++ отличается от большинства объектно-ориентированных языков программирования — он не делает никаких предположений по поводу того, как разрешать такие неоднозначности. Язык просто переадресует эту проблему программисту и предлагает решить ее при помощи существующих языковых конструкций. Тем самым предотвращается усложнение языка из-за дополнительного синтаксиса. В объектно-ориентированных языках программирования, интегрированных в Lisp, для разрешения подобных неоднозначностей существуют концепции *оболочек*, а также пред- и постусловных конструкций. Решение C++ сохраняет всю потенциальную мощь конструкций, существующих в Lisp-подобных языках. Что касается второй проблемы, в C++ поддерживаются *виртуальные* базовые классы, эмулирующие семантику CLOS (Common Lisp Object System — обобщенная объектная система Lisp) в отношении единого экземпляра класса с несколькими вхождениями в ориентированный ациклический граф наследования.

Учтите, что эти неоднозначности не являются артефактами языка программирования, они отражают реально существующие расхождения в структуре приложения.

Разрешение подобных конфликтов имен производится в процессе проектирования, хотя C++ позволяет выразить принятые решения в программе.

Язык C++ определен так, что допускается *объявление* конфликтующих операций; только их неоднозначное использование приводит к фатальной ошибке компиляции. Следовательно, хотя об этой проблеме желательно помнить на стадии проектирования, компилятор выявит все конфликты в готовой программе. В следующем разделе подробно описана методика разрешения таких конфликтов, а также обсуждаются вопросы дублирования данных в классах, наследующих от разных предков.

Неоднозначность при вызове функций класса

Имена функций разных базовых классов могут конфликтовать друг с другом в производном классе, но пользователь может ликвидировать подобные конфликты на программном уровне. Например, функция CursesWindow::clear конфликтует с функцией Window::clear; поскольку последняя ничего не делает (а всего лишь занимает место), ее можно проигнорировать и передать вызов CursesWindow. Однако функции scroll классов EditWindow и CursesWindow также вступают в конфликт в EditWnd_Curses. На этот раз должны быть вызваны обе функции, причем порядок вызова может оказаться существенным. Производный класс EditWnd_Curses устраняет неоднозначность, предоставляя собственную функцию scroll, подменяющую функции scroll базового класса. Функция scroll производного класса вызывает функции базовых классов в нужном порядке:

```
class EditWnd_Curses: public EditWindow, public CursesWindow {
public:
    void clear() { CursesWindow::clear(); }
    void scroll() {
        EditWindow::scroll();
        CursesWindow::scroll();
    }
    ...
};

EditWnd_Curses aNewWindow;
```

Вызовы функции scroll экземпляров EditWnd_Curses перестали быть неоднозначными; операция определяется в самом классе как последовательность вызовов функций scroll базовых классов данного объекта. Производный класс не обязан вызывать прототипы операций из базового класса — функция может содержать любой код по вашему усмотрению, но выполнение всех потенциально неоднозначных операций в заданном порядке принадлежит к числу основных идиом объектно-ориентированных языков. В C++ эта идиома не имеет поддержки на языковом уровне.

Неоднозначность в данных

Неоднозначность в данных решается практически так же, как неоднозначность в именах функций. Рассмотрим следующий пример:

```
class EditWindow: public virtual Window {
public:
    ...
protected:
    Cursor theCursor;
};

class XWindow: public virtual Window {
public:
    ...
protected:
    Cursor theCursor;
};

class EditWnd_X: public EditWindow, public XWindow { };
```

Класс **EditWnd_X** содержит две переменные с именем **theCursor** — это переменные **EditWindow::theCursor** и **XWindow::theCursor**. Поэтому из-за неоднозначности недопустимы дальнейшие обращения вида:

```
EditWnd_X::refresh() {
    ...
    // Ошибка: EditWindow::theCursor или XWindow::theCursor?
    ...this->theCursor...
    ...
}
```

Для разрешения подобных конфликтов имен требуется уточнение:

```
this->EditWindow::theCursor; // Курсор подобъекта EditWindow
                             // объекта EditWnd
this->XWindow::theCursor;    // Курсор подобъекта XWindow
                             // объекта EditWnd
```

Виртуальные базовые классы

Вторая проблема множественного наследования связана с многократным вхождением одного класса в число предков другого класса. Она решается объявлением некоторых базовых классов *виртуальными*. Это означает, что хотя класс **Window** несколько раз входит в ориентированный ациклический граф наследования, его данные войдут в **EditWnd_Curses** только однажды:

```
class Window {
    ...
};
```

```
class EditWindow : public virtual Window {
    ...
};

class CursesWindow : public virtual Window {
    ...
};

class XWindow : public virtual Window {
    ...
};

class EditWnd_Curses :
    public EditWindow,
    public CursesWindow
{
    ...
};
```

По умолчанию производный класс содержит столько экземпляров данных базового класса, сколько раз этот базовый класс выступает в роли предка по разным линиям наследования. Объявление класса производным виртуально означает, что он готов ограничиться только одним экземпляром данных базового класса, если этот базовый класс многократно является его виртуальным предком. Если все пути от базового класса к производному в графе наследования проходят через виртуальное наследование, то производный класс будет содержать только одну копию данных. Если будет использоваться комбинация виртуального и невиртуального вариантов наследования, то для каждого невиртуального наследования будет создана отдельная копия данных, плюс одна общая копия для всех виртуальных вариантов наследования.

Учтите, что классы, имеющие общий виртуальный базовый класс и входящие в один ориентированный ациклический граф множественного наследования, могут нарушить инкапсуляцию друг друга. И это вполне естественно, поскольку эти классы выразили свою готовность совместно использовать данные базового класса посредством виртуального наследования. Свертка данных базовых классов полностью находится под контролем программиста — в отличие от других языков программирования, она не задействуется по умолчанию и не является обязательной.

Предотвращение лишних вызовов функций виртуальных базовых классов

Итак, мы выяснили, как исключить лишние копии подобъекта базового класса Window из экземпляра CursesWindow или XWindow. Почему бы не задать аналогичные вопросы относительно функций класса? Давайте еще раз посмотрим, как решалась проблема с неоднозначностью функций класса:

```
void EditWnd_Curses::scroll() {
    EditWindow::scroll();   CursesWindow::scroll();
}
```

Оказывается, нечто похожее относится к функции refresh и другим операциям с окнами. Возможно, вызов одноименных функций всех родителей в разумном порядке действительно имеет смысл. Но ведь функции EditWindow::scroll и CursesWindow::scroll в свою очередь обращаются к *своему* базовому классу Window, предлагая ему принять участие в операции:

```
void EditWindow::scroll() {
    ...   // Действия, специфические для EditWindow
    Window::scroll();
}

void CursesWindow::scroll() {
    ...   // Действия, специфические для CursesWindow
    Window::scroll();
}
```

Таким образом, при вызове функции scroll для объекта класса EditWnd_Curses функция Window::scroll вызывается дважды. Возможно, это окажется для вас неожиданностью. Если для функции очистки окна clear повторный вызов всего лишь приводит к небольшому снижению эффективности, то для функции прокрутки scroll повторный вызов может вызвать ошибку. Хотелось бы получить возможность управлять последовательностью вызовов Window::scroll при выполнении кода производных классов CursesWindow и EditWindow.

Для этой цели существует специальное условное обозначение. Код, относящийся к классу EditWindow, выделяется в отдельную функцию _scroll:

Старый вариант:

```
class EditWindow:
            public Window {
public:
    void scroll() {
        ... // Локальная логика
        Window::scroll();
    }
    ...
};
```

Новый вариант:

```
class EditWindow:
          public Window {
public:
    void scroll() {
        _scroll();
        Window::scroll();
    }
protected:
    void _scroll() {
        ...  // Локальный код
    }
    ...
};
```

То же самое делается с классом CursesWindow. Теперь производный класс может точнее указать, когда и в каком порядке должны выполняться операции его базовых классов:

```
class EditWnd_Curses:  public EditWindow,
                       public CursesWindow {
```

```
public:
    void scroll() {
        // Следующие четыре функции могут следовать
        // в любом порядке на усмотрение программиста.
        EditWindow::_scroll();
        CursesWindow::_scroll();
        _scroll();      // Не обязательно
        Window::_scroll();
    }
protected:
    void _scroll() {
        ...   // Вся дополнительная работа, которая должна
              // быть выполнена производным классом
    }
    ...
};
```

Функция EditWnd_Curses::scroll указывает, что сначала функция базового класса EditWindow должна выполнить свою часть работы, после чего управление передается функции CursesWindow. Далее выполняется собственная логика прокрутки класса EditWnd_Curses (если она имеется), которая по аналогии была выделена в функцию EditWnd::_scroll. Наконец, функция базового класса Window::_scroll связывает все выполненные операции воедино. При желании программист мог задать любой другой порядок выполнения операций.

Введение вспомогательной функции _scroll называется *идиомой неоднозначных функций при множественном наследовании.*

Виртуальные функции

В иерархиях с множественным наследованием виртуальные функции так же полезны и удобны, как в иерархиях с одиночным наследованием. Их поведение является логическим продолжением ситуации с одиночным наследованием: вызываемая функция определяется фактическим типом самого объекта, но не типом указателя на объект.

Для примера возьмем описанные выше классы Window. Переменная, объявленная с типом Window*, может указывать на экземпляр любого класса в иерархии. Указатель требуется для вызова функции класса; если функция является виртуальной, то ее реализация выбирается в соответствии с типом объекта окна, на которое ссылается данный указатель. Предположим, функция scroll объявлена виртуальной в базовом классе:

```
class Window {
public:
    ...
    virtual void scroll() = 0;
    ...
};
```

Вызовы виртуальной функции scroll автоматически переадресуются нужному объекту во время выполнения:

```
int main() {
    EditWnd_X anXEditWindow;
    EditWnd_Curses aCursesEditWindow;
    Window *windowPointer = &anXEditWindow;
    windowPointer->scroll(); // Вызывает EditWnd_X::scroll
    windowPointer = &aCursesEditWindow;
    windowPointer->scroll(); // Вызывает EditWnd_Curses::scroll
    return 0;
}
```

Такое поведение соответствует правилам поведения виртуальных функций при одиночном наследовании.

Преобразование указателей на объекты при множественном наследовании

Рассмотрим еще один пример с уже знакомыми классами EditWindow, XWindow и EditWnd_X:

```
EditWnd_X *pe = new EditWnd_X;
XWindow *px;
...
px = (XWindow*)pe;     // Теперь px содержит другой физический
                       // адрес, отличный от pe
px = pe;               // Тот же результат
pe = 0;                // pe присваивается целочисленный ноль
// pe = px;            // Ошибка, необходимо преобразование
pe = (EditWnd_X*)px;   // Восстанавливается исходное значение pe
```

Изначально в программе создается объект EditWnd_X, на который ссылается указатель pe. Мы преобразуем pe в указатель на XWindow и копируем значение в px. Поскольку в C++ указатели на объект базового класса могут ссылаться на объект любого класса, порожденного от него открытым наследованием, такое преобразование допустимо и имеет интуитивно понятную семантику на уровне языка C++. Тем не менее, то что при этом происходит «за кулисами», не столь очевидно и заслуживает отдельных пояснений.

При множественном наследовании представление различных «частей» объекта начинается с разных адресов памяти. Если переменной px присваивается значение pe, указатель px должен ссылаться на «подобъект XWindow» исходного объекта EditWnd_X, иначе разыменование членов станет невозможным. Данное обстоятельство дает интересные побочные эффекты. Возьмем следующий фрагмент:

```
void an_X_function(void *p1, void *p2) {
    if (p1 == p2) printf("p1 == p2\n");
    else printf("p1 != p2\n");
}
```

```
EditWnd_X *pe = new EditWnd_X;
XWindow *px = 0;
px = pe;
an_X_function(pe, px);
```

Проверка равенства в an_X_function завершится неудачей в большинстве реализаций C++. В общем случае нельзя гарантировать, что преобразование указателя от одного класса к другому, а затем к третьему (в данном случае void*) даст тот же результат, что и прямое преобразование указателя от первого к третьему типу. Можно провести аналогию с преобразованием float (скажем, 1,2) в integer, а затем в double; результат отличается от того, который был бы получен при непосредственном преобразовании float в double.

Особое место занимают нулевые указатели. C++ гарантирует, что даже после преобразования указателя к другому типу его сравнение с нулевым указателем всегда будет выполняться правильно:

```
EditWnd_X *pe = 0;
XWindow *px = 0;
if (px == 0) ...   // Условие истинно
px = pe;
if (px == 0) ...   // Условие истинно
```

5.8. Каноническая форма наследования

При проектировании программ необходимо учитывать их возможную эволюцию и структурировать их таким образом, чтобы они успешно переживали изменения в требованиях и применяемых технологиях. Системы условных обозначений, используемых при наследовании, упрощают понимание программы и обеспечивают правильность работы полиморфных объектов.

В листинге 5.20 представлена каноническая форма наследования в C ++. Подумайте о том, чтобы любое наследование в вашей программе всегда было виртуальным. Доводы в пользу такого решения связаны с семантикой множественного наследования C++ и количеством копий подобъектов Base в любом объекте, происходящем от Base по нескольким независимым линиям наследования.

Листинг 5.20. Каноническая форма наследования

```
class Base {
public:
    // Класс Base в ортодоксальной канонической форме
    Base();
    Base& operator=(const Base&);
    virtual ~Base();
    Base(const Base&);
private:
    ...
};
```

продолжение

Листинг 5.20 (*продолжение*)

```
class Derived: [virtual] [public|private] Base {
public:
    Derived();
    Derived operator=(const Derived&);
    ~Derived();
    Derived(const Derived&);
    void foobar() {
        ...
        _foobar();
        ...
        Base::foobar();
        ...
    }
protected:
    void _foobar() {...}
};
```

Возьмем базовый класс **Window**, имеющий две разновидности производных классов: одни отражают различия в технологии многооконного интерфейса, другие — различия в специфике приложения. С одной стороны, окна Microsoft Windows должны отличаться от окон X Window System; с точки зрения приложения эти окна ведут себя одинаково, но имеют разную реализацию. С другой стороны, требуется, чтобы разные категории приложений (например, редакторы и текстовые процессоры) тоже использовали специализированные разновидности окон. Иерархия наследования может выглядеть примерно так:

```
class Window {
    ...
private:
    short windowID;
};

class XWindow: public Window {
    ...
};

class MSWindow: public Window {
    ...
};

class WPWindow: public Window {
    ...
};

class EditWindow: public Window {
    ...
};
```

Допустим, мы определяем производный класс

```
class X_WP_Window: public XWindow. public WPWindow {
};
```

Сколько полей Window::windowID в него войдет? Правильный ответ — два. Анализ задачи может навести на мысль, что именно такой ответ нам нужен. Например, если в программе имеется объект WPWindowManager, отслеживающий окна текстовых процессоров, и другой объект XWindowManager, отслеживающий окна X Window, то каждый из них может иметь *собственное* представление об идентификаторах объектов окон. А это означает, что каждый объект должен содержать поле идентификатора для каждой категории, к которой он принадлежит, — в нашем случае для каждой разновидности управляющих объектов. Нечто похожее должно быть сделано и в том случае, если класс Window содержит конструкции для создания связанных списков:

```
class Window {
public:
    ...
    void insert(Window*);
    ...
private:
    Window *previous. *next;
    ...
};
```

Если каждый домен (например, домен приложений и домен технологий реализации) захочет поддерживать свое собственное представление о принадлежности объекта Window к связанному списку, прибегать к виртуальному наследованию нельзя. К сожалению, это решение принимается на первом уровне наследования (одиночном!), а *не* при множественном наследовании. Тщательный анализ и предусмотрительность помогут избежать переработки интерфейса в процессе развития программы. Возможно, ваши предположения не всегда будут идеальными, но, по крайней мере, постарайтесь продумать эти вопросы заранее.

Следующие классы из короткого банковского приложения, приведенного в конце главы, являются хорошим примером уместного виртуального наследования. От базового класса Account создаются производные классы CheckingAccount и SavingsAccount:

```
class Account {
public:
    virtual void print_statement() = 0;
    ...
protected:
    // Общие низкоуровневые функции доступа к файлам.
    // используемые производными классами
    void file_read(const char *);
    void file_write(const char *);
```

```
private:
    AccountNumber accountNumber;
};

class CheckingAccount: public virtual Account {
    ...
};

class SavingsAccount: public virtual Account {
    ...
};
```

А теперь рассмотрим новый производный класс:

```
class CheckingSavingsAccount: public CheckingAccount,
                              public SavingsAccount {
    ...
};
```

Виртуальное наследование используется по следующим причинам:

- каждый объект банковского счета обладает собственными идентификационными данными (номером счета), а мы не хотим, чтобы класс CheckingSavingsAccount имел два отдельных номера счета;
- наследование требуется для объединения аспектов поведения, а не для объединения состояний (и конечно, не для дублирования общих аспектов состояния объектов).

Стоит напомнить, что программист должен проявить предусмотрительность на первом уровне наследования. Если отложить эти решения до непосредственного обращения к виртуальному наследованию, возможно, вам придется вносить изменения в архитектуру приложения (и наверняка перекомпилировать большую часть кода).

В приведенном примере канонической формы наследования мы специально постарались отделить вызовы функций производного класса от вызовов одноименных функций базового класса. Согласно общепринятым «правилам», части функции, связанные исключительно с локальной спецификой производного класса, выделяются в секцию protected. Последовательность вызовов этих частей и их взаимодействие с базовым классом определяются «настоящей» реализацией функции в производном классе. Эти действия также выполняются для поддержки множественного наследования, как упоминалось в разделе 5.7. Они необходимы только в том случае, если функция производного класса вызывает одноименную функцию базового класса, а в будущем возможно виртуальное наследование.

Применение данной идиомы ограничивается двумя обстоятельствами. Во-первых, если вам точно известно, что множественного наследования не будет, то виртуальное наследование от базового класса становится лишним. Во-вторых, виртуальное наследование может отразиться на эффективности, а в некоторых ситуациях приходится жертвовать универсальностью ради экономии нескольких микросекунд.

Ниже приводятся дополнительные рекомендации по созданию иерархий наследования.

- При открытом наследовании функции базового класса в основном должны объявляться виртуальными. Тем самым вы указываете, что производный класс является специализацией базового класса. Объявление виртуальной функции гарантирует, что реализация из производного класса заменит реализацию базового класса даже при использовании объекта производного класса в контексте, предполагающем объект базового класса. Например, если класс SunWindow является производным от Window, было бы желательно, чтобы объекты SunWindow могли применяться везде, где применяются объекты Window. Чтобы компилятор в подобных ситуациях вызывал функции SunWindow, они должны быть объявлены виртуальными — компилятор выберет нужную реализацию во время выполнения.
- При закрытом наследовании функции базового класса в основном *не должны* объявляться виртуальными (эта тема подробно рассматривается в разделе 7.5).
- Если вы собираетесь преобразовывать указатели на базовый класс в указатели на производный класс, виртуальное наследование исключается. Впрочем, необходимость в подобных преобразованиях возникает крайне редко: обычно она означает, что не виртуальная функция должна быть заменена виртуальной, или же в классы иерархии должна быть добавлена виртуальная функция.
- Обращайтесь к виртуальному наследованию, если вас не особенно беспокоит снижение быстродействия, или разрабатываемый класс будет использоваться в иерархиях с множественным наследованием. При этом необходимо учитывать некоторые моменты.
  - Проанализируйте данные состояния в базовом классе. Если какие-либо данные должны дублироваться в каждом экземпляре (особенно при множественном наследовании), от виртуального наследования придется отказаться.
  - Если одни данные класса отвечают этому критерию, а другие — нет, подумайте о переработке архитектуры приложения. Правильно ли отражает множественное наследование то, что вы собираетесь сделать?
  - Согласны ли вы с возможным нарушением инкапсуляции базового класса? Виртуальное наследование открывает нетривиальные возможности для нарушения инкапсуляции.

Упражнения

1. Опишите совместное использование наследования и полиморфизма как механизма абстракции.
2. Виртуальные функции могут рассматриваться как средство поддержки объектно-ориентированного программирования на базе архитектурной модели классов как аналогов абстрактных типов данных. Полиморфные функции, в аргументах которых могут передаваться любые типы (при условии соблюдения сигнатуры), поддерживают объектно-ориентированное программирование

на базе процедурной архитектурной модели. В книгах часто упоминается пример полиморфной функции sort, способной сортировать вектор произвольных объектов. Можно ли реализовать полиморфные функции на C++? Как? Сравните подход C++ с подходом, принятым в других объектно-ориентированных языках программирования.

3. *Шаблоны* (см. главу 7) позволяют создавать полиморфные функции на C++. Они на уровне компилятора обеспечивают поддержку функций, способных работать с произвольными множествами типов. В каких ситуациях вместо наследования и виртуальных функций лучше применять шаблоны? Каковы преимущества шаблонов перед виртуальными функциями? Уступают ли шаблоны виртуальным функциям по своим возможностям? Почему?
4. Воспользуйтесь отладчиком для отслеживания вызовов функций в следующей программе, использующей описанные ранее в этой главе структуры Atom:

```
Atom *lexAtom(String& s) {
    GeneralAtom g(s);
    return g.transform();
}

void parse(String& s) {
    for (Atom *a = lexAtom(s); a; a = lexAtom(s)) {
        ...
    }
}
```

5. Переработайте реализацию обработки сообщений LAPD (см. раздел 5.5) так, чтобы не требовалось копировать буфер (передавайте указатели конструкторам классов, соответствующим разным типам сообщений и разным уровням протоколов).
6. Реализуйте класс Amplifiier из раздела 5.6.
7. Спроектируйте и реализуйте набор классов для выполнения операций символической алгебры на C++ с применением идиомы функторов. Попробуйте добавить дифференциатор и интегратор, выполняющие операции символического исчисления над формулами с несвязанными переменными. При наличии всех граничных условий и параметров классы должны использовать метод Ньютона, метод Рунге-Кутта и т. д.

Пример итератора очереди

```
class Queue {
friend class QueueIterator;
public:
    void *leave() {
        if (head == tail) retval = 0;
        else {
```

```
            if (--head < 0) head = max - 1;
            retval = rep[head];
        }
        return retval;
    }
    void *first() { return rep[head]; }
    void enter(void *v) {
        if ((head + 1) % max == tail) { /* ошибка -- переполнение */ }
        if (--tail < 0) tail = max -1;
        rep[tail] = v;
    }
    int empty() { return head == tail; }
    Queue() { rep = new void*[max]; head = tail = 0; }
private:
    enum Constants { max = 10 };
    void **rep;
    int head, tail;
};

class QueueIterator {
public:
    QueueIterator(const Queue& q): queue(q), current(q.head) { }
    int next(void* &i) {
        if (current == queue.tail) {
            return 0;
        } else {
            if (--current < 0) current = Queue::max - 1;
            i = queue.rep[current];
            return 1;
        }
    }
    void reset() { current = queue.head; }
private:
    const Queue &queue;
    int current;
};

int main() {
    int one = 1, two = 2, three = 3, four = 4,
        five = 5, six = 6, seven = 7, eight = 8,
        nine = 9, ten = 10, eleven = 11, twelve = 12;
    void *i;
    Queue a;
    a.enter(&one);
    a.enter(&two);
    a.enter(&three);
    QueueIterator qi = a;
    while (qi.next(i)) printf("%d ", (int)(*((int*)i)));
    printf("\n");
```

```
    a.enter(&four);
    a.enter(&five);
    a.enter(&six);
    printf("leaving: "); i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    a.enter(&seven);
    a.enter(&eight);
    a.enter(&nine);
    a.enter(&ten);
    a.enter(&eleven);
    a.enter(&twelve);
    QueueIterator qj = a;
    while (qj.next(i)) printf("%d ", (int)(*((int*)i)));
    printf("\nleaving: ");
    i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    qj.reset();
    while (qj.next(i)) printf("%d ", (int)(*((int*)i)));
    printf("\nleaving: ");
    i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    i = a.leave();
    printf("%d ", (int)(*((int*)i)));
    return 0;
}
```

Простые классы счетов для банковского приложения

```
#include <fstream.h>

class Account {
public:
    Account();
    Account(double penalty, double minimum,
      const char *name = "Account");
    ~Account();
```

```
    virtual void deposit(double) = 0;
    virtual void withdraw(double) = 0;
    virtual void print_statement() = 0;
protected:
    void file_write(const char*);
    void file_read(char*);
protected:
    void _deposit(double);
    void _withdraw(double);
    void _print_statement();
private:
    double penalty;
    double minimum;
    double balance;
    char* name;
    ifstream fpr;
    ofstream fpw;
};

class InterestAccount : virtual public Account {
protected:
    InterestAccount(double penalty,
      double interest, double minimum,
      char* s = "InterestAccount");
    ~InterestAccount();
public:
    void deposit(double);
    void withdraw(double);
    void print_statement();
    double calculate_interest();
protected:
    void _deposit(double);
    void _withdraw(double);
    void _print_statement();
private:
    double interest;
};

class SavingsAccount : public InterestAccount {
public:
    SavingsAccount(double penalty,
      double interest, double minimum,
      char* s = "SavingsAccount");
    ~SavingsAccount();
private:
};

class CheckingAccount : virtual public Account {
protected:
```

```
    CheckingAccount(double penalty,
      double minimum, const char* s = "CheckingAccount");
    void deposit(double);
    void withdraw(double);
    void print_statement();
protected:
    void _deposit(double);
    void _withdraw(double);
    void _print_statement();
private:
};

class InterestCheckingAccount : public CheckingAccount,
   public InterestAccount {
public:
    InterestCheckingAccount(double penalty,
      double interest, double minimum,
      char* s = "InterestCheckingAccount");
    void deposit(double);
    void withdraw(double);
    void print_statement();
protected:
    void _deposit(double);
    void _withdraw(double);
    void _print_statement();
};
```

Литература

1. Russo, V. F., and S. M. Kaplan. «A C++ Interpreter for Scheme», «Proceedings of the C++ Workshop». Denver: USENIX Association Publishers (October 1988).
2. Stroustrup, B. «The C++ Programming Language», 1st ed. Reading, Mass.: Addison-Wesley, 1986, p. 165.
3. Knuth, Donald E. «Fundamental Algorithms», Vol. 1, «The Art of Computer Programming», 2nd ed., Reading, Mass.: Addison-Wesley, 1973, pp. 460-461.
4. Agha, G.A. «Actors: A Model of Concurrent Computation in Distributed Systems» Cambridge: MIT Press, 1986.
5. Ungar, David, and Randall B. Smith. «Self: The Power of Simplicity», SIGPLAN Notices 22,12 (December 1987).
6. Strang, John. «Programming with Curses», Newton, Mass.: Reilly and Associates, 1986.
7. Mansfield, Niall. «The X Window System: A User's Guide». Reading, Mass.: Addison-Wesley, 1991.

Глава 6
Объектно-ориентированное проектирование

Ничто не влияет на качество программного продукта так, как архитектура и проектирование. Это хорошо известный принцип, действующий не только в области программирования: вот уже несколько десятилетий он проявляется в схемотехнике, и в течение тысячелетий — в строительстве. Именно на ранней стадии, в процессе проектирования, закладывается долгосрочная «несущая» структура системы, включая средства исправления дефектов и будущего расширения системы. Проектирование также в значительной мере определяет эстетику системы, ее элегантность и красоту. Хорошо спроектированная система благополучно переживет неблагоприятные внешние воздействия, а ее эволюция пройдет гладко и естественно. Плохо спроектированная система начнет разваливаться уже на ранней стадии разработки, а может быть, просто рухнет под собственным весом.

Первые главы книги были посвящены синтаксису и семантике наследования и виртуальных функций C++ — основам объектно-ориентированного *программирования*. Пришло время сделать шаг назад и взглянуть на происходящее с точки зрения *проектирования*, а может быть — с еще более удаленных архитектурных позиций. Чтобы программа была понятной и была хорошо приспособлена к развитию, ее необходимо строить на прочном фундаменте хорошего проекта. А чтобы программа C++ была хорошо спроектирована, программист должен хорошо понимать, из чего можно получить хорошие классы, объекты и функции, а из чего нельзя.

Объектная парадигма хорошо приспособлена к структурированию широкого спектра сложных систем. Многие сложные программы характеризуются абстракциями, легко преобразуемыми в объекты и классы: ресурсы, события и «сущности» предметной области. Другие методики (такие, как функциональная декомпозиция или анализ потока данных) прекрасно работают в некоторых специфических областях, но не обладают такой универсальностью. Объектно-ориентированный анализ становится неплохой отправной точкой во многих системах, а другие методики могут дополнить объектно-ориентированные аспекты реализации.

Объектно-ориентированные методы конструирования систем выходят за рамки технических вопросов определения структуры — они предоставляют средства для взаимодействия с клиентами и отражения их ожиданий в окончательной версии продукта. Несоответствие ожиданий действительности привело к краху многие проекты, как технологические, так и социальные.

Объектная парадигма, как и любая другая, является совокупностью множества архитектурных правил и принципов. Правила ограничивают процессы проектирования и программирования, но именно ограничения предотвращают возможные нарушения в процессе проектирования и долгосрочного сопровождения. Прежде всего эти ограничения нацелены на создание абстракций и последующего управления ими. Как и при любом архитектурном подходе, построение абстракций помогает справиться со сложностью приложения. Методики абстрагирования, применяемые в объектной парадигме, ориентируются на терминологию, ресурсы и абстракции *предметной области*; во многих других методиках проектирования абстракции ориентированы на *область решения*, или *реализации*. К счастью, объектная парадигма предоставляет технологию реализации (объектно-ориентированное программирование), позволяющую напрямую отразить структуру этих абстракций в реализации. Логически завершенная связь анализа приложения с проектированием и реализацией помогает отразить в реализации наиболее фундаментальные, стабильные и неизменные с точки зрения предметной области вещи (в той степени, в которой можно говорить о стабильности и неизменности чего-либо в программировании или в реальном мире). Параллелизм между стабильными факторами описания задачи и структурой решения является краеугольным камнем эволюции системы в объектной парадигме. Данный принцип, называемый *принципом изоморфной структуры*, занимает центральное место в способности классов и объектов инкапсулировать изменения.

Методики проектирования, описанные в настоящей главе, неформальны; это можно сказать о большинстве методик, применяемых на практике. Лучшим «инструментом», обеспечивающим их работу, является человеческий разум.

В этой главе программы рассматриваются с точки зрения формулировки абстракций стадии проектирования; такой подход демонстрирует читателю, как язык C++ напрямую отражает эти абстракции в программном коде. Кроме того, в этой главе представлены некоторые «неписаные правила» программирования на C++, которые помогают создавать гибкие, эффективные и элегантные программы. Глава начинается с уточнения определений классов, объектов, типов и их отношений друг с другом, после чего мы переходим непосредственно к процессу проектирования. В оставшейся части главы рассматриваются правила проектирования при построении абстракций классов и структуризации отношений между ними. Заложив основу объектно-ориентированного проектирования в этой главе, мы перейдем к более сложным идиомам в последующих главах.

6.1. Типы и классы

Классы часто характеризуются как средства создания новых типов и их включения в язык программирования. Во многих (хотя и не во всех) языках программирования существует концепция *типа*, связывающего интуитивно понятный набор свойств с языковыми переменными. Например, мы знаем, что с целыми числами можно выполнять операции сложения, вычитания, умножения и деления; со строковыми переменными — операции конкатенации, поиска подстрок и т. д.

Давайте взглянем на этот предмет с исторических позиций. Концепция *типа* первоначально создавалась как формальная модель, основанная на свойствах теории множеств. Таким образом, тип «целое число» означал множество всех целых чисел; он определял допустимые операции с членами множества, а также свойства (например, замкнутость), которыми обладало это множество. Одно множество (называемое *подмножеством*) может полностью содержаться внутри другого. Каждое из этих множеств определяет некоторый тип. Тип, определяемый меньшим множеством, называется *субтипом* по отношению к типу большего множества. Например, множество целых чисел является субтипом для множества вещественных чисел, потому что последнее включает все элементы множества целых чисел.

В синтаксисе первых языков программирования делалась попытка моделировать математические выражения. Как известно, один из ранних языков программирования (FORTRAN) ставил своей целью прямой перевод математических формул в программный код (FORmula TRANslation). Функции языка FORTRAN должны были моделировать математические функции, а его типы должны были стать моделями математических типов. *Целью было моделирование абстракций пространства задачи непосредственно на языке программирования.* Большинство ранних программ относились к областям инженерных и математических расчетов, поэтому сущности и типы этих областей отражались в переменных и типах языков того времени. Тем не менее, языкам программирования приходилось считаться с реальностью в отношении как представления, так и поведения типов; например, множества не могли быть действительно бесконечными; на практике их размер зависел от размера машинного слова.

Со временем «типы» превратились в удобные обозначения, утратившие большую часть своего теоретического наследования. Например, многие языки (включая C, C++ и версии FORTRAN до FORTRAN66) позволяют выполнять математические операции с символами. С точки зрения семантики приложения такая возможность выглядит сомнительно, но она приносит огромную пользу в практическом программировании. Большая часть функций в большинстве программ не ограничивается строгим моделированием математических функций (неоднозначное отображение области в интервал), поэтому формальные корни типов были утрачены.

Концепция *класса* тоже достаточно стара. Она восходит как минимум к книге «Principia Mathematica», опубликованной в начале XX века [1]. В этой работе классы описываются как «всего лишь символические или языковые удобства, а не настоящие объекты в отличие от своих представителей». Приведенная формулировка в целом сохранила свой смысл до наших дней, причем сейчас она отчасти перекрывается с пониманием термина «тип».

В C++ программист может создавать собственные классы и использовать их как типы в понимании традиционных языков программирования. Для любых двух типов/классов можно определить преобразование, имеющее смысл для моделируемой области. Хотя преобразование `double` в пользовательский тип `Complex` выглядит логично, другие преобразования (например, `List`, `Set` или `Apple` в `Orange`)

смысла не имеют. Соответствующие решения должны приниматься на стадии проектирования, а их поддержка на уровне языков программирования ограничена. Лучше всего браться за них с точки зрения архитектуры, используя принципы проектирования, обеспечивающие адекватную семантику и хорошую эстетику.

Итак, имеются типы, относящиеся к предметной области, и классы, выражающие соответствующие архитектурные решения. В хорошей объектно-ориентированной архитектуре *сущности* предметной области отображаются на объекты, а типы — на классы области решения (рис. 6.1). Сущности предметной области — это те понятия, с которыми мы имеем дело в приложениях (транзакции, денежные суммы, счета и кассиры в банковской системе; телефоны, звонки и коммутаторы в телекоммуникационной системе, и т. д.). Отдельные сущности обобщаются в *типы*, чтобы мы могли говорить о транзакциях, счетах и телефонах *вообще*, а не о конкретном счете Джо в Центральном банке или домашнем телефоне Джейн. Эти концепции предметной области отображаются на программные абстракции в области решения: типы превращаются в *классы*, а сущности превращаются в *объекты*.

Рис. 6.1. Отношения между предметной областью и областью решения

Хотя данная модель лучше всего подходит для C++, она не является единственно возможной. Концепции сущностей и типов могут быть реализованы в виде объектов, если класс сам по себе является объектом, как в языке Smalltalk или self [2]. Одни и те же конструкции языка программирования могут задействовать-

ся для отражения сущностей и типов предметной области; при использовании для представления типов такие объекты называются *прототипами*. Прототипы служат моделями для создания новых объектов либо посредством простого копирования, либо выполнением функции класса, которая создает и возвращает новый экземпляр. В главе 8 описана идиома C++, позволяющая моделировать концепцию прототипов объектно-ориентированного программирования в C++.

Помимо связей, изображенных на рис. 6.1, существуют и другие отношения между типами предметной области, которые должны быть отражены в области решения. Например, если один тип является специализацией другого типа (подобно тому, как DigitalPressureMeter и AnalogPressureMeter являются специализациями более абстрактного типа PressureMeter), этот факт отражается наследованием в области решения. Если класс DigitalPressureMeter открыто наследует от PressureMeter, он автоматически проявляет свойства PressureMeter, отражая отношения между двумя абстракциями в предметной области. С другой стороны, объект AnalogPressureMeter может содержать внутренний объект Spring, по аналогии с реальным миром. Как правило, подобные связи находят простое отражение в области решения, а изоморфизм между предметной областью и областью решения является одной из главных составляющих мощи объектно-ориентированного проектирования. Параллелизм этих двух областей способствует инкапсуляции изменений и упрощает эволюцию программы. Структура предметной области отражается в структуре классов. Общая структура реального мира изменяется реже, чем вторичные аспекты поведения его компонентов (отражаемые в изменениях функций классов) или их реализация (которая может отражаться во внутренних структурах данных или алгоритмах класса).

Отношения специализации в предметной области отражаются в C ++ посредством открытого наследования. Открытое наследование, инкапсуляция данных и виртуальные функции являются основными инструментами объектно-ориентированного программирования. Объявление класса AnalogPressureMeter открыто производным от PressureMeter предполагает, что класс AnalogPressureMeter будет проявлять все поведение PressureMeter в своем открытом интерфейсе. Программный код должен выглядеть примерно так:

```
class AnalogPressureMeter: public PressureMeter {
public:
    ...
    // С точки зрения пользователя объекта AnalogPressureMeter
    // все открытые аспекты PressureMeter работают
    // практически так же, как если бы они были объявлены здесь.
    ...
private:
    ...
};
```

Однако по умолчанию в C++ используется *закрытое наследование*, означающее, что функции базового класса доступны только *внутри* производного класса

и скрыты от его клиентов. Закрытое наследование является не столько приемом проектирования, сколько средством реализации, ориентированным на многократное использование кода. В этом качестве оно будет рассмотрено отдельно в 7.5. Различия между открытым и закрытым наследованием раскрыты также в начале главы 4.

Объекты реального мира связаны более сложными отношениями, которые не всегда выражаются в виде специализации и включения. Появилась даже специальная терминология для описания таких отношений. В этих терминах нет ничего особенно священного, мистического или даже сколько-нибудь научного, но они справляются со своей задачей — передавать информацию о связях между объектами. В соответствии с традициями теории информационного моделирования, мы будем использовать термин «связи IS-A» для обозначения специализации; так, DigitalPressureMeter является (is a) частным случаем PressureMeter. Термин «связи HAS-A» обозначает включение: ключевое класс AnalogPressureMeter содержит (has a) объект Spring. Но как насчет других отношений — например, сходства? Как обозначить родство между AnalogPressureMeter и AnalogVoltMeter? Какими отношениями связываются классы «начальник», «секретарь» и «копировальный аппарат»? В духе объектно-ориентированного проектирования мы хотим отразить эти отношения в структуре реализации. Их правильное отражение в объектно-ориентированном языке программирования потребует дополнительного планирования и поддержки со стороны языковых идиом, чтобы созданная программная структура могла нормально эволюционировать с течением времени. В этой главе будут представлены примеры таких идиом.

Объектно-ориентированные языки программирования открывают новые возможности для связей «IS-A» в области задачи. Механизм наследования помогает упорядочить абстракции в соответствии с отношениями «IS-A», упрощая многократное использование общих аспектов в родственных классах. Но существуют и другие возможности многократного использования кода, которые никак не связаны с отношениями «IS-A» и лишь слабо отражают свойства предметной области. C++ также обладает хорошими выразительными средствами для программирования оптимизаций области решения и многократного использования кода. Настоящая глава поможет проектировщикам лучше понять эти две разновидности отношений. Правильный баланс приложения между проблематикой области решения и предметной области является важным фактором, имеющим чрезвычайно важное значение для удобства сопровождения системы и эстетики ее архитектуры.

Разрыв связей между формальными концепциями предметной области и конструкциями области решения порождает негибкие реализации, в которых трудно разобраться. Как будет показано далее, проблема становится особенно серьезной, если классы связываются друг с другом отношениями «IS-A». Но прежде чем переходить к рассмотрению этой темы, стоит уделить еще немного внимания основам объектно-ориентированного проектирования.

6.2. Основные операции объектно-ориентированного проектирования

Хорошее понимание предметной области является ключом к успеху любого проектирования, основанного на любых методиках. Конечно, это относится и к проектированию, основанному на объектно-ориентированных принципах, но объектно-ориентированные методы упрощают эту задачу, потому что модули, из которых строится система, определяются понятиями *пользователя* о предметной области (телефоны, счета и т. д.), а не теми понятиями автора реализации (базы данных, дисковые накопители и т. д.). Автор реализации может использовать алгоритмы и инструменты, принятые в его сфере деятельности, но такие инструменты всегда работают в контексте, понятном для пользователя.

Очень важно, чтобы проектировщик объектно-ориентированной системы (да и любой системы вообще) поддерживал рабочие контакты с конечным пользователем и проверял свои предположения относительно цели и особенностей работы системы. В этом ему могут помочь прототипизация и итеративный характер разработки. Модульность, обеспечиваемая наличием объектов, позволяет разработчику перебрать несколько возможных архитектур, прежде чем остановиться на окончательном варианте. Когда разработчик хорошо понимает все основные абстракции с точки зрения потребностей пользователя, он начинает строить реализацию при помощи своего инструментария.

В результате процесс разработки существенно отличается от традиционных методик вроде нисходящего или структурного проектирования. Различия весьма важны, и умение проводить разработку в итеративном цикле с точки зрения клиента потребует от проектировщика радикальных изменений в процессе мышления.

С учетом сказанного в объектно-ориентированном проектировании можно выделить конкретные операции.

- *Определение сущностей в предметной области.* Сущностями могут быть ресурсы, события или любые другие четко определенные области специализации или знаний. Сущности часто соответствуют именам существительным в постановке задачи, предоставленной заказчиком приложения. Например, если компетентный пользователь пакета управления полетом полагает, что на модели самолета будет присутствовать руль, включите в архитектуру программы сущность `Rudder`.

  На этом уровне проектирования неважно, будет ли сущность обобщенной «классовой» сущностью или конкретным экземпляром. Различия между этими двумя уровнями абстракции выделяются в процессе преобразования сущностей в структуры реализации.

  Не существует твердых правил относительно того, какой набор сущностей окажется «лучшим», как не существует единственно верного варианта разбиения

системы на сущности. И все же существует ряд эмпирических критериев, которые служат «лакмусовой бумагой» для проверки качества абстракции.

- Легко ли идентифицировать и присвоить имена аспектам поведения сущности? Если нет — возможно, сущность лучше заменить функцией или другой абстракцией.
- Связаны ли аспекты поведения сущности друг с другом и с самой сущностью на интуитивно понятном уровне?
- Удачно ли выбран размер абстракции, не является ли она слишком большой или слишком мелкой? (Некоторые проектировщики считают, что интерфейс недопустимо велик, если его не удается описать на одной стороне карточки размером 3 × 5 дюйма [3]).
- Заранее определенные аппаратные компоненты и программы, взаимодействующие с ними, обычно хорошо подходят для оформления в виде сущностей.
- Если вы не можете предложить две хорошие существенно различающиеся реализации сущности, возможно, необходима более широкая абстракция.

В разделе 6.5 рассматриваются другие правила идентификации полезных сущностей.

- *Определение поведения сущностей.* Каждая сущность предоставляет некоторые услуги приложению — либо посредством выполняемых ею операций, либо других сущностей, содержащихся в ней. Эти услуги называются *обязательствами* сущности в отношении приложения. Все необходимые услуги и обязательства должны быть идентифицированы и связаны с правильными сущностями.

  «Хорошее» поведение отличается четкостью, логической полнотой и лаконичностью. Если рассматривать аспекты поведения как функции классов, которыми они должны стать, они редко требуют более одного или двух параметров. Возможно, самым важным фактором объектно-ориентированного проектирования является выбор аспектов поведения, достаточно общих для потребностей широкого круга систем в заданной предметной области, без создания абстракций, слишком крупных для понимания или недостаточно четко определенных для эффективной реализации. (За дополнительной информацией об идентификации «хорошего» поведения обращайтесь к разделу 6.5).
- *Определение отношений между сущностями, особенно отношений «IS-A».* Если сущность является частным случаем другой, более общей сущности, обязательно выделите эту связь; возможно, вам удастся ее использовать на стадии реализации. В этом вам помогут *транзакционные диаграммы*, описанные в главе 11. Другие важные отношения — например, вхождение одной сущности в другую — более подробно описываются далее в этой главе.
- *Построение начальной иерархии C++ на базе сущностей.* Сущности, выделенные на начальном этапе, являются хорошими кандидатами на роль классов и объектов C++. Конкретные сущности будут представлены объектами в программе; обобщения сущностей (типы) преобразуются в классы. Имена классов должны отражать семантику их аналогов в предметной области.

В реализации отношения между субтипами чаще всего воплощаются в иерархиях наследования, в которых большинство функций объявляется виртуальными. Иногда вместо наследования более уместно применить механизм *перенаправления*, описанный в этой главе.

Некоторые сущности предметной области отображаются не на классы или объекты реализации, а на абстракции из других парадигм. Например, для моделирования некоторых абстракций, выделенных на стадии анализа предметной области, могут хорошо подойти традиционные технологии баз данных.

- *Реализация*. В этой фазе окончательно формируются структуры данных классов, в классы добавляются новые закрытые функции, а также программируются сами функции классов. Иногда в ней происходит модульное тестирование функций или классов.
- *Оптимизация*. Некоторые функции объявляются подставляемыми (inline) для повышения эффективности (если вызов функции или копирование аргументов обходятся слишком дорого). Возможно, вам удастся обнаружить сходство между некоторыми реализациями, которое не проявилось в фазе анализа (то есть при выполнении первых трех операций). В этой фазе также задействуется закрытое наследование, обеспечивающее многократное использование кода.
- *Тестирование*. Тестирование методом черного ящика производится так же, как в любых других системах.

Как видите, представленные операции не пронумерованы — это не последовательные *шаги*, которые постепенно приведут вас от анализа к реализации. Иногда вам придется возвращаться к уже выполненным операциям и даже выполнять некоторые операции параллельно, пока итеративные расхождения не сведутся к минимуму, а система не придет к стабильному состоянию. Ключ к успешной реализации — итеративный перебор, исправление ошибок и повторная реализация аспектов, нуждающихся в улучшении.

При объектно-ориентированной разработке границы между этапами традиционных схем разработки теряют четкость. Предварительная реализация (построение прототипа) может сделать фазу проектирования более глубокой и содержательной; с другой стороны, некоторые решения из области проектирования могут потребовать хотя бы частичной реализации. Хотя анализ и проектирование по-прежнему можно рассматривать как задачи, преследующие разные цели, в объектной парадигме эти две операции имеют сходную структуру, и их разделение не имеет особого смысла.

Также обратите внимание на тот факт, что методика в большей степени ориентируется на результаты проектирования, нежели на взаимодействия между компонентами. Конечно, взаимодействия объектов очень важны; ведь именно они в конечном счете решают поставленную задачу. Однако архитектурная целостность системы обусловлена структурой ее интерфейсов, а структуры, сформированные объектными интерфейсами, обычно хорошо инкапсулируют изменения и обладают потенциалом для развития.

Логическое разбиение, полученное в результате объектно-ориентированного анализа системы, значительно отличается от структуры, выработанной традиционными методами функциональной декомпозиции, ориентированной на взаимодействия между компонентами системы. В методике функциональной декомпозиции функции являются артефактами архитектурных решений, принимаемых на более высоких уровнях: они создаются на основании анализа взаимодействий между высокоуровневыми компонентами. Решения, принятые на столь низком уровне, часто связываются с предметной областью неочевидными или излишне сложными связями. Если уделять основное внимание программным абстракциям, и лишь затем заниматься взаимодействиями, архитектура программы сможет нормально развиваться с течением времени (а также станет более простой и понятной).

Можно привести следующую аналогию с объектно-ориентированным проектированием: ваш начальник выбирает вас и еще нескольких людей для совместной работы над проблемой. Начальник (проектировщик системы) выбирает вас за ваш личный опыт, то есть за ту пользу, которую вы принесете группе. Начальник ставит перед группой задачу и предоставляет ей действовать так, как она считает нужным; он не указывает, как именно должны взаимодействовать между собой участники группы. Все они представляют автономные объекты, взаимодействующие друг с другом для решения общей задачи.

6.3. Объектно-ориентированный и доменный анализ

Основные операции объектно-ориентированного проектирования анализируют проблему и разлагают ее на абстракции, инкапсулирующие информацию о различных аспектах проблемы. Проектировщик должен выйти за рамки конкретного приложения и расширить абстракции на всю предметную область. В результате такого расширенного подхода к операциям проектирования формируется новая методика, называемая *доменным анализом*, или *анализом предметной области*. Как всегда, термины «проектирование» и «анализ» намеренно трактуются неформально и не имеют специального смысла, выходящего за рамки бытового уровня.

Причины увеличения объема проектирования

Допустим, нам поручено написать программу автопилота для самолета «Sopwith 123». Мы можем без особого труда выбрать хорошие абстракции — рули, элероны, двигатель и т. д. — и объединить их для управления взлетом, полетом и приземлением самолета.

Но вместо того, чтобы проектировать программу только для «Sopwith 123», мы постараемся расширить абстракции так, словно они предназначены не только для этой конкретной модели, а для гипотетического абстрактного самолета. «Абст-

рактный самолет» представляет характеристики интересующей нас *предметной области*. Такой подход выбран по трем причинам.

- Увеличение объема проектирования упростит переход на новые абстракции, сходные с уже имеющимися. Так, компания, заказавшая программу управления полетом для «Sopwith 123», может заказать аналогичные программы для других самолетов. А нам было бы удобно использовать готовые разработки в новых проектах.
- Увеличение объема проектирования содействует эволюции системы. Если для «Sopwith 123» будет выпущен новый двигатель, новая гидравлика или новый компас, нам было бы лучше ограничиться минимальными изменениями в программе для адаптации к новым требованиям. Если абстракции проектирования будут как можно меньше зависеть от информации о текущей модели двигателя, гидравлике, компасе и т. д., то переход на новые детали в меньшей степени отразится на программе. Чтобы упростить эту эволюцию, следует описывать абстракции в терминах их взаимодействий с другими абстракциями, а не их внутренней реализации. Если компас взаимодействует с оборудованием управления полетом посредством трех электронных сигналов, он будет характеризоваться тремя соответствующими свойствами. Скорее всего, новый компас тоже будет подключаться через те же три контакта, а если повезет, наша программа продолжит работу без изменений.
- Увеличение объема проектирования способствует наиболее полному соответствию системы предполагаемым целям ее разработки. В индустрии программных средств уже никто не удивляется тому, что готовые системы не отвечают ожиданиям заказчиков. Расширяя масштабы абстракций системы, мы помогаем ей «соответствовать более широкому кругу требований». Если окончательный вариант системы не устроит заказчика по некоторым параметрам, обобщенную систему будет проще адаптировать, чем систему, оптимизированную для единственной конкретной цели.

Способы расширения абстракций

Как же расширить абстракции, чтобы придать им большую гибкость в свете возможных изменений? Один из способов — применение «толстых» интерфейсов для предоставления всех возможных видов сервиса. Например, должна ли структура нашей системы предполагать, что на самолете установлен один мотор, или же моторов может быть несколько? Мы можем добавить немного «излишеств» в исходную архитектуру и выбрать в качестве абстракции для центральной силовой установки вектор двигателей. Архитектура «Sopwith 123» будет соответствовать вырожденному случаю с вектором из одного элемента. Применение вектора потребует лишних затрат и слегка снизит эффективность для «Sopwith 123», но с другой стороны, этот вектор значительно упростит адаптацию программы для многомоторного самолета. Если класс содержит аспекты поведения, добавленные по соображениям универсальности, говорят, что он обладает *толстым интерфейсом*.

Толстые интерфейсы принадлежат к числу классических механизмов обобщения. Процедурные библиотеки линейной алгебры часто содержат функции с десятками параметров, многие из которых обычно остаются неиспользованными. Такие абстракции труднее освоить и понять, но это один из немногих «нормальных» методов, используемых с процедурными абстракциями без нарушения общности.

В объектно-ориентированном проектировании лучше пойти в обратном направлении: *исключить* аспекты поведения из абстракции, чтобы сделать ее более общей. Анализируя двигатель «Sopwith 123», мы найдем десятки разных рычагов, которыми можно управлять. Но вместо того чтобы включать каждый рычаг в интерфейс абстракции двигателя, проще либо свернуть несколько рычагов в более абстрактную концепцию, либо скрыть их. Скажем, если двигатель содержит два карбюратора (устройства, управляющие подачей топлива и воздуха в двигатель), их можно объединить в одно абстрактное карбюраторное устройство. Применение *транзакций* (см. главу 11) как средства проектирования помогает выявить подобное сходство. Если двигатель «Sopwith 123» отличается от других двигателей наличием клапана сброса давления для ручного запуска, данная особенность не будет отражена в обобщенной абстракции двигателя. Таким образом, вместо «толстых классов» используются «тонкие классы», обобщенный характер которых связан с тем, что они *не содержат* лишней информации о сущности или типе.

Балансировка архитектуры

До каких же пределов следует расширять масштабы архитектуры? Этот вопрос можно задать как для отдельных абстракций, так и для предметной области, определяющей строение их интерфейсов. Если рассматривать «Sopwith 123» в качестве целевой системы, должны ли мы обобщить ее:

- для всех моделей «Sopwith?
- для всех бипланов?
- для всех самолетов с одним пропеллером?
- для всех самолетов с пропеллерами?
- для всех самолетов?
- для всех летающих объектов, оснащенных двигателями?
- для всех машин?
- для всех управляющих систем?
- для всех программных проектов C++?

С какого-то момента обобщение порождает больше проблем, чем решает. Как распознать этот момент? Это зависит от вашего личного опыта и характера предметной области, для которой ведется проектирование.

Результаты хорошей балансировки архитектуры

Мы расширяем масштабы архитектуры, чтобы упростить многократное использование... но чего именно? Основные усилия направлены на многократное использование программного кода. Но еще важнее многократное использование структуры решения, его архитектуры и результатов проектирования. Если нам удастся заново задействовать часть архитектуры, тем самым будет обеспечено и многократное использование большей части кода. Если некоторые абстракции не укладываются в рамки существующих реализаций, иерархия классов предоставляет основу для построения новой абстракции. Конечно, эта схема работает лишь в достаточно обобщенных архитектурах; новое приложение должно соответствовать предметной области, выбранной для предыдущей разработки.

Многоразовые архитектуры могут оформляться в виде *библиотек* (см. главу 11). Мы вернемся к проблеме многократного использования в главе 7.

6.4. Отношения между объектами и классами

Вспомните фазы идентификации отношений между сущностями и их воплощения в реализации. Здесь мы снова подробно остановимся на отношениях между абстракциями в объектно-ориентированных архитектурах. Эта тема настолько важна, что мы будем постоянно возвращаться к ней в оставшейся части главы.

В предыдущем разделе было показано, что отношения между сущностями могут быть напрямую преобразованы в иерархии классов и объектов C++. На рис. 6.2 и 6.3 изображены сущности (работники организации), связанные некоторыми отношениями. Допустим, нужно спроектировать программную систему, отражающую эти абстракции. Анализ отношений между работниками определяет отношения между классами, используемыми для их представления в нашей программе.

По крайней мере еще одна отрасль вычислительной техники интересуется отношениями между сущностями реального мира — речь идет об искусственном интеллекте. Одним из инструментов искусственного интеллекта являются *семантические сети*, у которых существует собственная терминология для связывания сущностей друг с другом [4]. Мы будем пользоваться этой терминологией, поскольку она хорошо подходит для описания программных иерархий, отражающих модели реального мира.

Отношения «IS-A»

Отношения «IS-A» определяют связи специализации между типами или классами. В простом житейском смысле вице-президент *является* (is a) работником фирмы: обобщенный вице-президент обладает всеми характеристиками обобщенного работника, но обратное неверно. Также можно сказать, что вице-президент является *частным случаем* работника, но не каждый работник является вице-президентом...

Или что множество работников включает множество вице-президентов. Конечно, секретарь тоже *является* работником, как и начальник отдела, и т. д.

Обычно отношения «IS-A» рассматриваются только в контексте классов (то есть не в контексте экземпляров). Данный вид связей лучше подходит для описания обобщенных характеристик, а не конкретных свойств. Например, мы обычно говорим, что кресло, на котором вы сидите, является сущностью одного из специализированных типов кресел, но не «креслом вообще». Отношения «IS-A» применяются для классификации: (анонимное) красное вращающееся кресло с подлокотниками связано с типом Chair (кресло) отношением «IS-A».


```
class Employee {
public:
  Employee(Name);
   ...
};

class Secretary: public Employee {
public:
  Secretary(Name);
  ...
};

class VicePresident: public Employee {
private:
  class AdminAsst: public Employee {
  public:
     ...
  }AAobj;
public:
  VicePresident(Name me, Name Asst,
    Name sec1, Name sec2):
      AsstObj(Asst),
      SEC1obj(sec1),
      SEC2obj(sec2),
      Employee(me) {}
  ...
private:
  Secretary SEC1obj, SEC2obj;
};

class DeptHead: public Employee {
public:
  DeptHead(Name me, Name secy):
      SECobj(secy),
      Employee(me) {}
  ...
private:
  Secretary SECobj;
};
```

Рис. 6.2. Отношения между типами

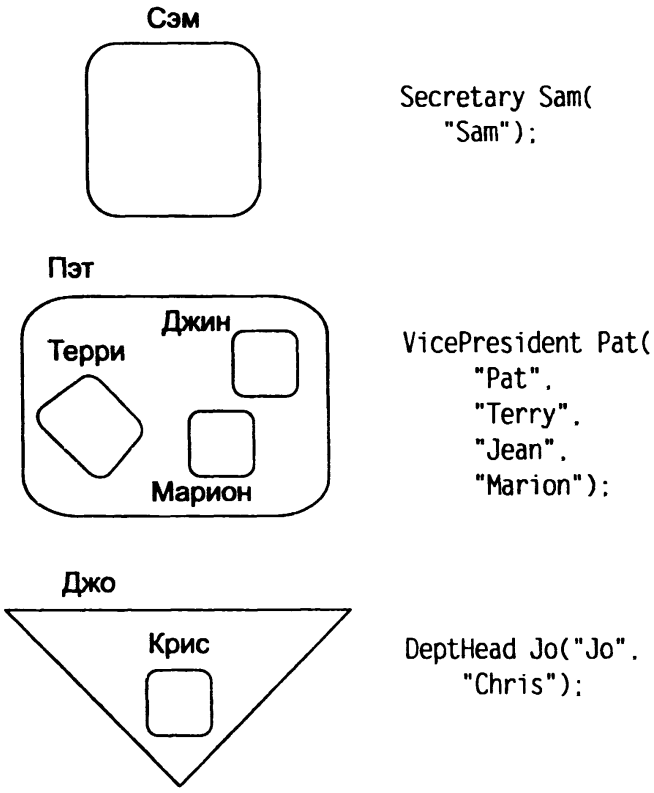


Рис. 6.3. Отношения между сущностями

В C++ отношения «IS-A» представляются открытым наследованием. Каждый из следующих классов объявляется открыто производным от класса Employee:

```
VicePresident
DeptHead
Secretary
AdminAsst
```

Это означает, что любые открытые (public) характеристики класса Employee также являются характеристиками любого из производных классов.

Отношения «HAS-A»

Отношения «HAS-A» определяют связи включения. Их синонимами являются выражения «является составной частью» или «использует в реализации». В отличие от связей «IS-A», выражающих исключительно отношения между классами, связи «HAS-A» могут определять отношения между двумя классами, между классом и объектом или между двумя объектами.

Начнем с уровня классов, а конкретно — с класса VicePresident (вице-президент). Предположим, вице-президенты договорились создать в компании должность *помощника по административной работе*, занимающегося исключительно обслуживанием вице-президентов. Помощники по административной работе подчиняются только вице-президентам (в целях нашего примера будем считать, что они взаимодействуют только с начальниками, секретарями и администраторами из других компаний). Поскольку это относится ко всем вице-президентам, отношение

существует на уровне класса. Класс VicePresident полностью распоряжается классом AdminAsst (помощник по административной работе), то есть фактически *содержит* (has a) его. Объявление второго класса размещается внутри объявления первого, отражая его *концептуальную* принадлежность.

Хотя должность секретаря придумана не вице-президентами, вполне естественно, что вице-президенты тоже пользуются услугами секретарей. На рис. 6.2 видно, что у начальников отделов тоже есть секретари, и это придумали не вице-президенты (а у помощников по административной работе могут быть свои секретари). Мы говорим, что *экземпляр* VicePresident или DeptHead (начальник отдела) связан отношениями «HAS-A» с *экземпляром* Secretary (секретарь). Отношения принадлежности существуют не на уровне *класса*, а на уровне *экземпляра*. Класс Secretary выражает самостоятельную концепцию; на концептуальном уровне он не «принадлежит» никакому другому классу. Отдельные экземпляры Secretary связываются со своими начальниками и объявляются как переменные классов, производных от Manager.

Отношение «HAS-A» на уровне экземпляра может быть реализовано как в форме полного включения объекта B в объект A, так и хранения указателя на B в объекте A независимо от того, создается ли объект B объектом A. Например, в иерархии на рис. 6.2 класс DeptHead может содержать только указатель на объект Secretary. Тем не менее, состояние объекта Secretary все равно рассматривается как часть состояния объекта DeptHead. Отношение «HAS-A» между DeptHead и Secretary проявляется через указатель так же, как если бы объект Secretary был встроен в DeptHead. Но при использовании указателя отношение может быть причислено к категории «HAS-A» только в том случае, если указатель на этот экземпляр не хранится в других объектах (то есть если ссылка на данный экземпляр Secretary хранится только в одном объекте DeptHead).

А если класс C++ содержит ссылку? Ссылка представляет собой псевдоним для обращения к другому существующему объекту. Если имеется ссылка на объект того же класса, то отношение «HAS-A» существует независимо от ссылки, и факт ее присутствия этой связи не изменяет. В противном случае объявление ссылки означает, что объект может не находиться в исключительной принадлежности класса, содержащего объявление, и что доступ к нему может осуществляться по другому имени. В этом случае отношение «HAS-A» может и не существовать.

Справедливо ли утверждение, что объект класса AdminAsst (например, Terry) принадлежит объекту класса VicePresident (например, Pat)? Конечно, это так — в дополнение к отношению «концептуальной принадлежности», существующей между этими классами на рис. 6.2. На практике редко встречаются отношения «HAS-A», существующие только на уровне классов и не опускающиеся на уровень объектов (тем не менее, один такой пример будет далее рассмотрен — заметите ли вы его?)

Давайте немного расширим пример и введем новый класс «обобщенного начальника» Manager (листинг 6.1). Как нетрудно заметить, отношения «IS-A» существуют между классами VicePresident и Manager, а также между DeptHead и Manager (DeptHead, как и VicePresident, является частным случаем Manager). Какое место в этой иерархии занимает концепция секретаря? Можно сказать, что *на концептуальном уровне* секретарь «принадлежит» начальнику, поэтому объявление Secretary сле-

дует разместить внутри Manager. Но если у разных начальников разное количество секретарей, было бы неправильно хранить экземпляр Secretary внутри экземпляра Manager — вместо этого следует объявить один или несколько экземпляров Secretary в каждом специализированном классе, представляющем конкретную разновидность начальника.

Листинг 6.1. Часть программной реализации корпоративной структуры

```
class Name { public: Name(const char *); /* ... */ };

class Employee {
public:
    virtual char *name();
    Employee(Name);
private:
    // ...
};

class Manager: public Employee {
public:
    Manager(Name n): Employee(n) { }
protected:
    class Secretary: public Employee {
    public:
        Secretary(Name n) : Employee(n) { }
        char *name();
    };
};

class VicePresident: public Manager {
public:
    VicePresident(Name me, Name Asst, Name sec1, Name sec2):
        Manager(me), SEC1obj(sec2), SEC2obj(sec2),
        AsstObj(Asst) { }
private:
    Secretary SEC1obj, SEC2obj;
    class AdminAsst: public Employee {
        // ...
    } AsstObj;
};

class DeptHead: public Manager {
public:
    DeptHead(Name me, Name secy): Manager(me), SECobj(secy) { }
private:
    Secretary SECobj;
};

Manager::Secretary Sam("Sam");
VicePresident Pat("Pat", "Terry", "Jean", "Marion");
DeptHead Jo("Jo", "Chris");
```

Отношения «USES-A»

Отношение «USES-A» возникает в том случае, если функция класса или функция, дружественная по отношению к некоторому классу, получает в параметре экземпляр другого класса, который она *использует* (uses a). Также отношение «USES-A» существует, если логика функции класса опирается на услуги другого класса. Например, можно говорить об отношениях «USES-A» между вице-президентом и начальником отдела; первый «использует» последнего для выполнения некоторых поручений. Данное отношение относится к классу «USES-A», а не «HAS-A», потому что обращение к начальнику отдела со стороны вице-президента не подразумевает, что объект DeptHead включается в состояние экземпляра VicePresident. Объект DeptHead не рассматривается как «строительный блок» для реализации VicePresident; тем не менее, эти два класса взаимодействуют через связь «USES-A».

Отношения «CREATES-A»

В контексте упомянутых канонических форм отношение «CREATES-A» выглядит необычно: оно связывает объект с классом. Экземпляр одного класса в процессе выполнения одной из своих функций обращается к другому классу с запросом на создание (creates a) экземпляра; во втором классе вызывается оператор new с последующим выполнением конструктора. Отношения этого вида похожи на «USES-A», но они формируются между объектом и классом (а не двумя объектами).

В главе 8 мы подробно рассмотрим прототипы и идиомы, расширяющие возможности отношений «CREATES-A». Используя идиому отношения «CREATES-A» (которая традиционно *не присуща* C++), мы убедимся в том, что это отношение связывает два объекта, один из которых вызывает функцию другого без учета классов обоих объектов.

Контекстное изучение отношений между объектами и классами

Четыре рассмотренных вида отношений отражают логические связи между сущностями и типами и особенности их преобразования в классы и объекты во время реализации. В зависимости от прикладной области, языка и используемых идиом, другие отношения могут занимать более важные места в лексиконе разработчика проекта.

Представленные концепции отношений не являются формальными; читателю не стоит придавать им слишком большого значения. Они должны интерпретироваться в своем контексте и разумно использоваться как средство передачи информации, а не как строго определенные правила проектирования. Например, для описания взаимодействий между Manager и Secretary в предыдущем примере вместо связи «HAS-A» с таким же успехом можно использовать связи «USES-A»,

особенно если объекты Secretary более или менее самостоятельны и помимо своих начальников взаимодействуют с другими объектами. А если взять за основу известное утверждение, будто *в действительности* работой организации управляют секретари, получится совсем другая структура.

Графическое представление отношений между объектами и классами

В [5] предложена специальная система обозначений для описания структуры объектно-ориентированных систем. Эта система состоит из четырех компонентов:

- *диаграммы классов* отражают структуру классов, их отношения друг с другом и общие отношения между их экземплярами;
- *объектные диаграммы* отражают взаимодействие между объектами, которые содержат ссылки, полностью включают друг друга или передаются в качестве параметров;
- *диаграммы состояния* представляют объекты как конечные автоматы и документируют переходы между состояниями в результате вызовов функций классов;
- *временные диаграммы* отражают последовательность событий разных объектов.

На практике чаще всего применяются диаграммы классов и объектные диаграммы, представляющие основные отношения между классами и объектами в графическом виде. У обеих разновидностей диаграмм имеются собственные условные обозначения. На рис. 6.4 приведены условные обозначения для диаграмм классов, а на рис. 6.5 — для объектных диаграмм.

Рис. 6.4. Условные обозначения отношений между классами

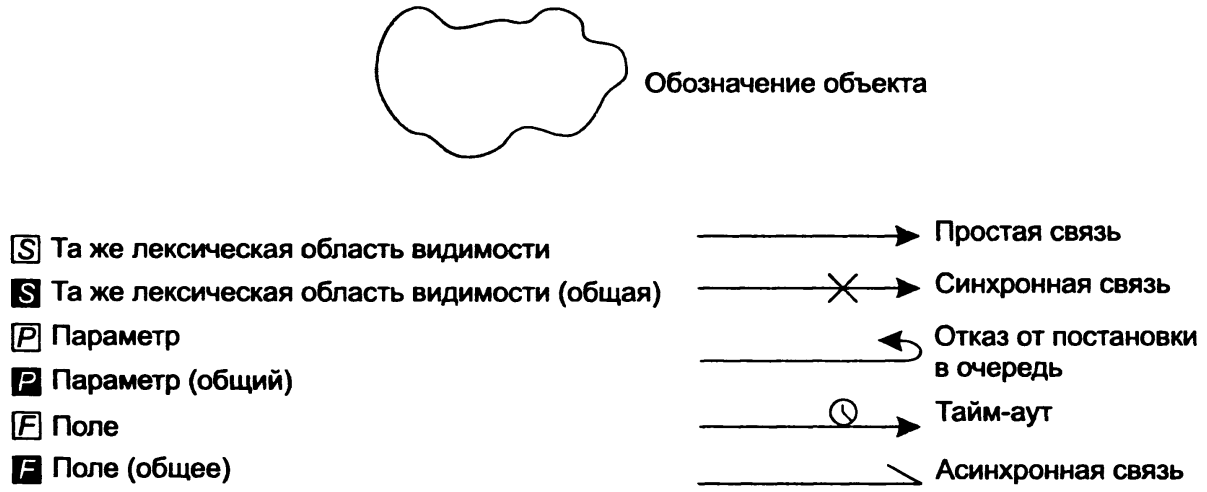


Рис. 6.5. Условные обозначения отношений между объектами

На рис. 6.6 представлена диаграмма классов для описанного ранее примера Employee. На диаграмме отражены отношения наследования (классы Employee и Manager являются базовыми) и некоторые отношения, характеризуемые общностью на уровне классов. Так, у каждого начальника отдела (DeptHead) имеется секретарь (Secretary), а у вице-президентов (VicePresident) сразу два секретаря; каждый объект класса Secretary имеет доступ к одному копировальному аппарату (CopyMachine). Обратите внимание: на этой диаграмме *не отражен* факт объявления класса Secretary внутри Manager.

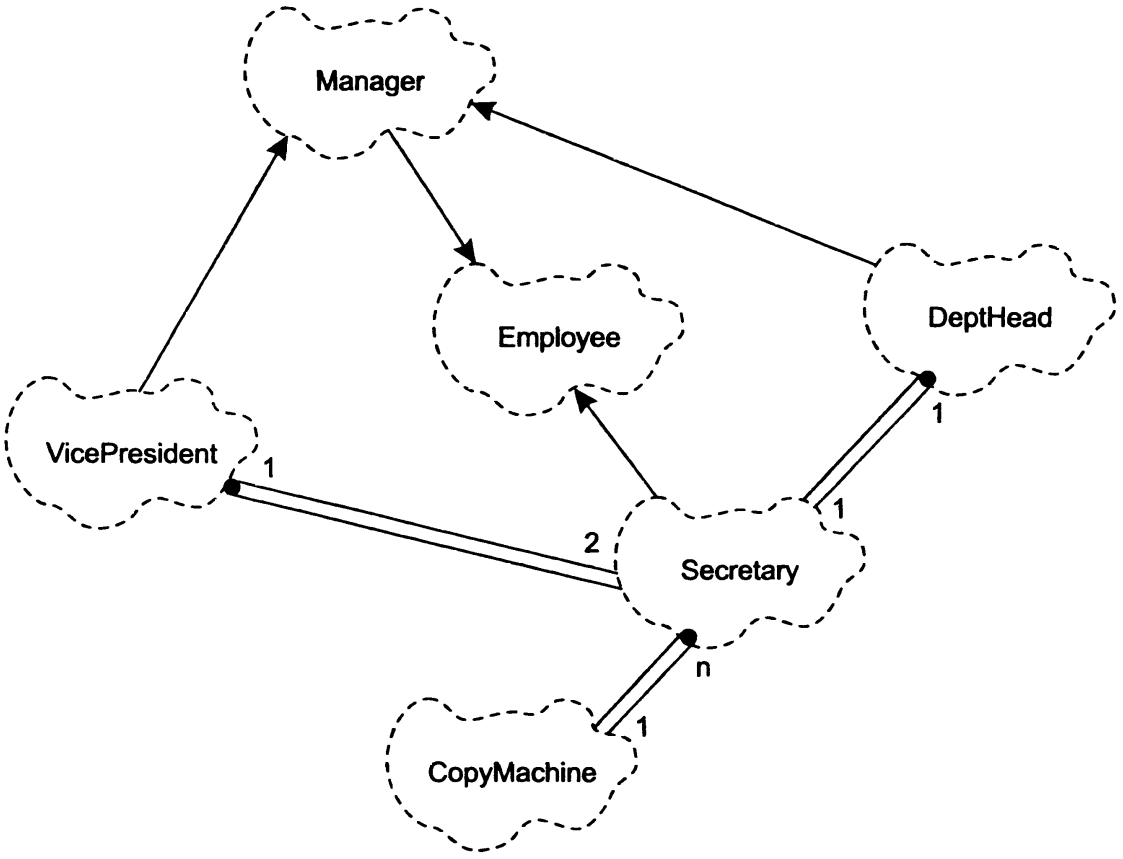


Рис. 6.6. Диаграмма классов для примера Employee

Соответствующая объектная диаграмма изображена на рис. 6.7.

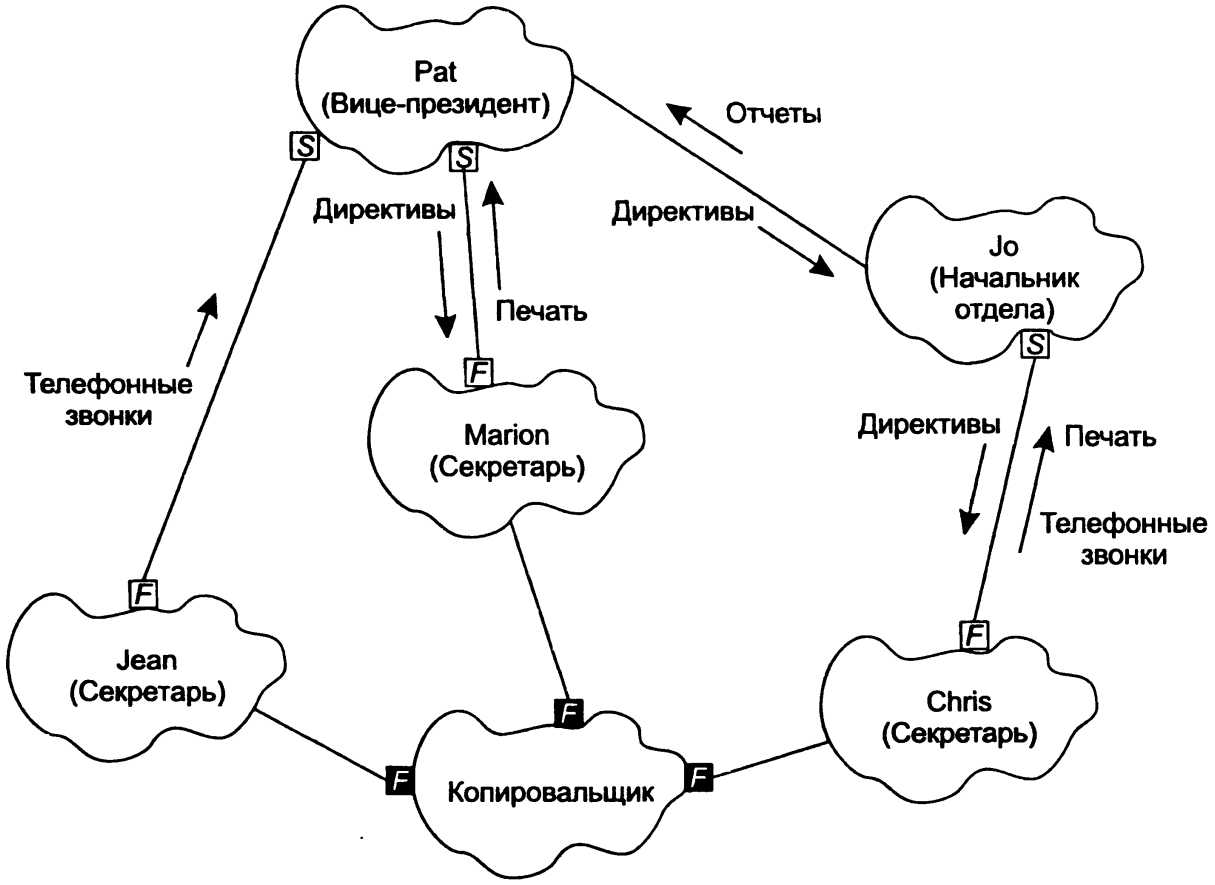
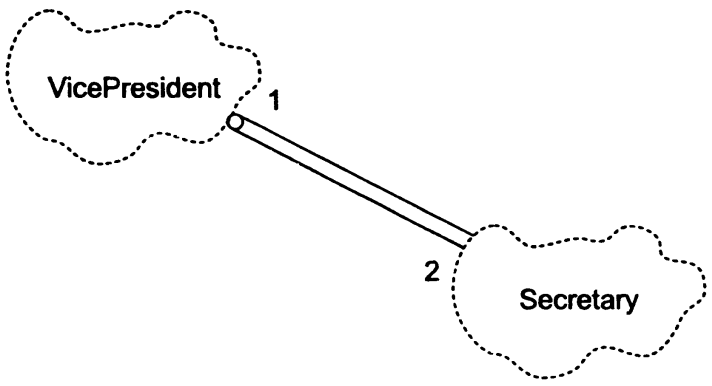


Рис. 6.7. Объектная диаграмма для примера Employee

Из диаграммы видно, что каждый объект, относящийся к классу Secretary, использует объект Copier совместно с другими; Copier является общим полем всех объектов класса Secretary. На объектной диаграмме не показаны экземпляры классов Manager и Employee, поскольку эти классы не имеют самостоятельных экземпляров. Секретарь Jean направляет телефонные звонки вице-президенту Pat; если некто, желающий поговорить с Pat, вместо этого говорит с Jean, возможно, диаграмму классов стоит изменить и обозначить на ней связь между VicePresident и Secretary иначе:



За полным описанием подобной системы обозначений обращайтесь к [5].

6.5. Субтипы, наследование и перенаправление

Многое из того, что можно сказать о наследовании, справедливо и по отношению к типам — пары «тип/субтип» часто реализуются в виде пар «базовый/производный класс». Но в некоторых случаях это соответствие нарушается. В этом разделе рассматриваются типичные ошибки, порождающие нежелательный разлад между семантикой типов в предметной области и структурой наследования или делегирования в области решения. В первых двух подразделах (посвященных ошибкам, связанным с потерей субтипов и типами-омонимами) рассматриваются отношения между типами и классами и ловушки, которые возникают на этой почве. Затем мы выясним, как наследование и делегирование связаны с инкапсуляцией, и как они могут привести к нарушению абстракции данных. Далее рассматриваются вопросы, относящиеся к «логической четкости» и инкапсуляции функций классов. В конце раздела описываются некоторые специальные идиомы поддержки множественного наследования.

Наследование ради наследования — ошибка потери субтипов

«Безумие передается по наследству — оно проявляется в потомках».

Хороший инженер проектирует с учетом реализации. В процессе идентификации сущностей предметной области (или типов) в голову приходят естественные стратегии реализации, включая структуры наследования, которые можно видоизменить для конкретного приложения. Но такая специфическая «настройка» иногда нарушает архитектурные связи между классами и разрушает структуру системы. В этом нам поможет убедиться пример.

Вернемся к приложению с обработкой чисел, для которого мы создали класс `Complex` (см. с. 99). У него было несколько производных классов, в том числе классы `Double` и `BigInteger`, а также класс `Imaginary`, который займет центральное место в следующем обсуждении. Класс `Complex` обеспечивает эффективное выполнение ряда операций: алгебраических функций, вычисления модуля, присваивания, возможно, вывода и т. д. Класс `Imaginary` ведет себя как `Complex`, но не имеет вещественной части.

Из предметной области (математики) очевидно, что множество объектов `Imaginary` является подмножеством множества объектов `Complex`. Разумно предположить, что тип `Imaginary` должен быть субтипом `Complex`. Хотя все чисто мнимые числа являются *частным случаем* комплексных чисел, не все комплексные числа являются чисто мнимыми.

Отношение субтипизации в предметной области является признаком открытого наследования одного класса от другого в реализации C++. Можно предположить, что программа будет содержать следующий фрагмент:

```
class Complex {
public:
    ...
};

class Imaginary : public Complex {
public:
    ...
};
```

Такая запись выражает отношение «Imaginary IS-A Complex», то есть Imaginary является субтипом Complex, а множество объектов Imaginary является подмножеством объектов Complex. Наследование не обеспечивает *принудительного* применения этой семантики; вся ответственность за поддержание совместимой семантики между классами иерархии возложена на программиста. Как правило, задача решается сохранением совместимой семантики между функциями классов с одинаковой сигнатурой. Оставшаяся часть главы содержит немало рекомендаций относительно того, как обеспечить эту семантику.

Если программист будет следовать этим рекомендациям, компилятор сможет предоставить мощную защиту, вспомогательные средства и средства оптимизации через свою систему типов. Представление компилятора о совместимости типов тесно связано с архитектурной субтипизацией; отношения подмножеств между абстракциями данных соответствуют классам и существующим между ними отношениям специализации. Именно эта связь позволяет языку программирования выражать архитектурные отношения и до определенной степени обеспечивать соблюдение правил проектирования.

Данные Imaginary находятся в базовом классе Complex — так, чтобы вычислить модуль экземпляра производного класса, мы обращаемся к данным базового класса. Производный класс также содержит «лишний груз»: поле rpart заведомо равно 0.

Учитывать реализацию в процессе проектирования — правильная привычка, если соображения реализации не выходят на первое место. В нашем примере они могут навести на мысль, что затраты на хранение лишнего слова в каждом объекте Imaginary неприемлемы. Ниже приведена альтернативная реализация класса Imaginary, лучше соответствующая этому представлению:

```
class Imaginary {       // Не является производным от Complex
public:
    Imaginary(double i = 0): ipart(i) { }
    Imaginary(const Complex &c): ipart(c.ipart) { }
    Imaginary& operator=(const Complex &c) {
        ipart = c.ipart; return *this;
    }
    Imaginary operator+(const Imaginary &c) const {
        return Imaginary(ipart + c.ipart);
    }
```

```
    Imaginary operator-(const Imaginary &c) const {
        return Imaginary(ipart + c.ipart);
    }
    Imaginary operator*(const Imaginary &c) { ... }
    Imaginary operator/(const Imaginary &c) const { ... }
    operator double() { return ipart; }
    operator Complex() { return Complex(0, ipart); }
    Imaginary operator-() const { return Imaginary(-ipart); }
    ...
private:
    double ipart;
};
```

В этой реализации стоит обратить внимание на два обстоятельства. Во-первых, алгебраические операции она выполняет быстрее своего предшественника. Во-вторых (что более важно), класс Imaginary перестал быть производным от Complex. Вместо отношения «IS-A» с Complex он использует отношение «USES-A» с Double.

Что же плохого в том, что для одних и тех же абстракций оптимизации области решения порождают две разные архитектурные структуры? Очевидно, две разные версии обладают одинаковой семантикой. Но в оптимизированной реализации теряется нечто очень важное: она перестает отражать связь между классами Complex и Imaginary в предметной области. Класс Imaginary в приведенной реализации продолжает отражать некоторые свойства Complex и до определенной степени остается совместимым с ним; эта задача решается функциями:

```
Imaginary::operator Complex
Imaginary::Imaginary(const Complex&)
```

Однако из-за потери наследования мы уже не сможем использовать объекты Imaginary вместо Complex там, где это имеет смысл. Например, оператор преобразования не поможет в ситуации, когда вместо Complex* передается Imaginary*.

Разные варианты реализации обладают своими достоинствами и недостатками, порой весьма нетривиальными. Приемлемое решение часто достигается только посредством компромисса. Возможно, в данном примере следовало бы объявить оба класса Complex и Imaginary производными от абстрактного базового класса Number без вынесения общих данных в базовый класс.

Тем не менее, в большинстве случаев злоупотребления наследованием происходят от недостатка опыта, легко обнаруживаются и имеют очевидные решения. Один программист, новичок в объектно-ориентированном программировании, объявил класс фильтра нижних частот Filter производным от классов Resistor, Capacitor и Inductor. Однако фильтр *не является* частным случаем резистора, конденсатора или индуктора, а множество объектов Inductor не содержит множество объектов Filter. «Правильное» решение состоит в том, чтобы включить объект каждого компонента в объект Filter. По аналогии с тем, как фильтр содержит конденсатор, индуктор и резистор в предметной области, программная абстракция Filter должна содержать соответствующие вложенные объекты.

А вот еще один плохой пример:

```
class Shape: public Point, public Color {    // Плохо
public:
    virtual void draw();
    virtual void rotate(const Angle&);
};
```

Данное объявление предполагает, что геометрическая фигура (Shape) является частным случаем цвета (Color), что неверно; фигура также не является и частным случаем точки (Point). Верно другое — геометрические фигуры *содержат* центральную точку и *обладают* цветом (отношение «HAS-A»). Конечно, реализация должна выглядеть так:

```
class Shape {
public:
    Shape(const Point&);
    virtual void draw();
    virtual void rotate(const Angle&);
private:
    Point center;        // Center и color - атрибуты,
    Color color;         // или свойства, класса Shape
};
```

Напоследок рассмотрим классы String и PathName, представленные в разделе 4.2. Если существующая библиотека содержит класс String, возникает искушение объявить класс PathName производным от него. Открытое наследование позволит передавать объекты PathName там, где ожидается объект String. Например, существующая хеш-функция для String может быть использована для хеширования PathName. Но открытое наследование приведет к тому, что вызовы некоторых функций PathName нарушат логическую целостность объекта. Конструктор PathName и операции класса могут предотвратить вхождение недопустимых символов во внутреннее представление. Но если PathName унаследует функцию operator[](int) от класса String, то следующий фрагмент нарушит ограничения, которые PathName пытается соблюдать в своих операциях:

```
int main() {
    PathName batFile = "AUTOEXEC.BAT";
    batFile[0] = '*';  // Запрещенный символ в имени файла
    ...
}
```

Как было показано в разделе 4.2, такие проблемы решаются за счет закрытого наследования.

Сомнительные иерархии также возникают при опоре на *изменчивость* как основу для наследования, то есть когда один класс строится на базе другого через отношение «LIKE-A» вместо отношения «IS-A». Близким родственником изменчивости является обратное наследование, имеющее своих сторонников как инструмент многократного использования. Например, графический пакет может

содержать класс Shape с производным от него классом Circle. Если потребуется нарисовать эллипс, мы обобщаем Circle посредством наследования и подменяем его операции для получения класса Ellipse. Оба подхода достаточно опасны и будут рассмотрены далее.

Случайное наследование — омонимы в мире типов

До сих пор в нашем рассказе основное внимание уделялось зависимости структуры системы от семантического *поведения* абстракций предметной области и представляющих их классов. Поведение класса должно определять логически целостную абстракцию, а классы в иерархии наследования группируются по предоставляемому ими сервису. Группировка по поведению сходна с группировкой классов по внешнему виду, но не эквивалентна ей. Различия между этими двумя парадигмами (моделями структурирования) состоит в том, что первая требует глубокого понимания семантики классов, а вторая может базироваться на простом лексическом сравнении имен. Но существует третий способ группировки классов по внутренней структуре представления. Он напоминает второй способ, но вместо интерфейса класса анализируется его внутренняя структура. Класс Line в графической программе содержит два объекта Point; это же относится к классу Rectangle. Из этого можно сделать вывод (ошибочный), что один из этих классов является специализацией другого.

При проектировании интерфейсов класса или группировке классов посредством наследования очень важно учитывать свойства поведения. В [6] предлагает следующий критерий организации классов в иерархиях наследования:

> ...если для каждого объекта o_1 типа S существует объект o_2 типа T такой, что для всех программ P, определенных в контексте T, поведение P не изменяется при замене o_1 на o_2, то S является субтипом T.

Например, некоторая функция (из интересующих нас программ P), написанная для работы с параметрами (o_2) типа Complex (тип T), также должна быть способна принимать объекты (o_1) типа Imaginary (тип S). В практическом смысле все функции T могут безопасно применяться к объектам S, поскольку поведение всех программ P остается неизменным при замене объектов S (o_1) объектами T (o_2). И это выглядит разумно, поскольку тип Imaginary может выполнить любую операцию, поддерживаемую типом Complex, хотя обратное неверно: функция, написанная для Imaginary, в общем случае не работает с типом Complex.

Этот принцип, называемый *принципом подстановки Лисков*, играет важную роль в объектно-ориентированном проектировании. Мы еще неоднократно вернемся к нему на страницах книги.

Некоторые проектировщики выступают за применение наследования, не отвечающее принципу подстановки. Иногда это рекламируется как субтипизация, определяемая больше реализацией, нежели архитектурой. Но такие применения порождают двусмысленности, являющиеся аналогом омонимов в языке програм-

мирования. В некоторых случаях подобные злоупотребления возникают из-за попыток применения объектной парадигмы «наоборот», например, объявления класса Ellipse производным от Circle и его «приукрашиванием», обеспечивающим возможность обобщения. Поскольку такие случаи очевидно сомнительны по принципу подстановки Лисков, здесь они не рассматриваются. Большинство других злоупотреблений относится к категории изменчивости, то есть попытки «наследования в сторону» с применением отношений «LIKE-A» вместо «IS-A».

Поучительный пример

Допустим, имеется готовый класс List, который служит контейнером для упорядоченных наборов объектов. Класс List не делает никаких предположений относительно содержащихся в нем объектов (то есть может быть реализован как список void*). Интерфейс List выглядит примерно так:

```
class List {
public:
    void* head();       // Возвращает начало списка
    void* tail();       // Возвращает конец списка
    int count();        // Возвращает количество элементов в списке
    Bool has(void*);    // Проверяет наличие элемента в списке
    void insert(void*); // Включает новый элемент в список
};
```

Пусть потребовалось создать новый класс множества Set со следующим интерфейсом:

```
class Set {
public:
    int count();        // Возвращает количество элементов в множестве
    Bool has(void*);    // Проверяет наличие элемента в множестве
    void insert(void*); // Включает новый элемент в множество
```

Гордясь собственной наблюдательностью, мы замечаем сходство между этими классами. Возможно, у нас даже имеются средства, позволяющие взять List за основу для реализации Set. Итак, мы пытаемся воспользоваться наследованием:

```
class Set: public List {
public:
    void insert(void* m) {
        if (!has(m)) List::insert(m);
    }
private:
    // Следующие две строки являются недопустимыми конструкциями C++
    List::head;    // Начало множества не определено
    List::tail;    // Конец множества не определен
};
```

Итак, чтобы реализовать семантику множеств, запрещающую вставку нескольких экземпляров одного значения, мы должны переопределить функцию insert(void*).

Операции count() и has(void*) наследуются без изменений. Операции head() и tail() становятся недоступными — они не определены для множеств, хотя и имеют смысл для списков.

Однако при этом возникает серьезная проблема. Применяя наследование, мы утверждаем, что множество является частным случаем списка («IS-A»). Другими словами, предполагается, что объекты List везде могут быть заменены объектами Set. Но это просто неверно, как неверно и обратное. Для примера возьмем их поведение при повторном включении одинаковых значений; для класса Set операция insert будет пустой, а для List — нет. Более того, класс Set не поддерживает семантики операций head() и tail(). C++ приходит на помощь в последнем случае: приведенные выше объявления Set, в которых List::head и List::tail объявлены закрытыми, определены в языке как недопустимые. Если бы такие операции были разрешены, их семантика была бы сомнительной — особенно для виртуальных функций.

Итак, типы List и Set почти являются *типами-омонимами*: они обладают сходными сигнатурами при существенно различающейся семантике (если бы тип List не содержал операций head() и tail(), омонимия была бы полной). Именно из-за подобных проблем Лисков разработала свои рекомендации по сигнатурам абстракций в иерархиях наследования, приведенные в начале раздела.

Рассмотрим другой пример, позаимствованный из [7]. Если у нас уже имеется готовый класс CircularList, и мы хотим создать класс Queue, можно попробовать сделать следующее:

```
class CircularList {
public:
    int empty();
        CircularList();
    void push(int);    // Занесение с начала
    int pop();
    void enter(int);   // Занесение с конца
private:
    cell *rear;
};

class Queue: public CircularList {
public:
        Queue() { }
    void enterq(int x)    { enter(x); }
    int leaveq()          { return pop(); }
};
```

Класс Queue автоматически получает операции push(int) и pop(); это побочный эффект наследования, не имеющий смысла для Queue. Корни этой проблемы уходят в механизм открытого наследования. Существует хорошее эмпирическое правило: открытое наследование должно применяться только для отношений субтипизации; если наследование применяется для удобства или по соображениям

многократного использования кода, следует применять закрытое наследование. Закрытое наследование (и лучшие альтернативы) рассматривается в главе 7 как средство многократного использования кода.

Похожие (и наверное, еще худшие) проблемы возникают тогда, когда наследование применяется для многократного использования *представления* одного типа в другом. Например, класс Set объявляется производным от List, потому что внутренние данные Set очень похожи на List: просто операции класса интерпретируют их несколько иначе. Естественно, такой подход не имеет отношения к типам и подтипам, и его следует избегать. Мы вернемся к этой теме в главе 7.

Решения без побочного наследования

В ситуациях, когда наследование кажется хорошим решением, часто удается обойтись более простыми средствами. Например, вспомним пример с классами List и Set. Пусть неудачная попытка вас не смущает; возможность использования некоторых операций List для реализации Set существует *реально*, просто открытое наследование было неправильным способом ее реализации. Вот как выглядит другой вариант:

```
class List {
public:
    void* head();       // Возвращает начало списка
    void* tail();       // Возвращает конец списка
    int count();        // Возвращает количество элементов в списке
    Bool has(void*);    // Проверяет наличие элемента в списке
    void insert(void*); // Включает новый элемент в список
};

class Set {
public:
    int count()          { return list.count(); }
    Bool has(void* m)    { return list.has(m); }
    void insert(void* m) { if (!has(m)) list.insert(m); }
private:
    List list;
};
```

Три операции Set вручную перенаправляются внутреннему объекту List. При этом нам не нужно принимать специальные меры, чтобы запретить вызов операций head() или tail() [6].

В этом варианте функциональность класса List заново используется для поддержки семантики класса Set. Хотя между List и Set существует определенное сходство, мы отказываемся от попыток установить тесные связи между их типами. Если взглянуть на происходящее несколько иначе, такая реализация указывает на архитектуру, которая ничего не говорит о связи между Set и List. Она отдаленно напоминает методику *делегирования*, в основу которой заложены не классы, а экземпляры (см. с. 166). C++ поддерживает *перенаправление* операций от одного класса к другому, которое может считаться слабой формой делегирования. Механизм делегирования описан в разделе 5.5.

Наследование с расширением и подменой

Правильное применение наследования, обеспечивающее реализацию субтипизации по принципу подстановки Лисков, достигается путем *наследования с расширением*. Интуиция подсказывает, что добавление новых функций в класс сокращает количество характеризуемых им объектов. Например, наделение простого телефона функции ускоренного вызова создает новую абстракцию телефона с бóльшим количеством функций, чем у оригинала: множество телефонов с ускоренным вызовом является подмножеством всех телефонов. Поскольку существующие операции остаются без изменений, объект производного класса по-прежнему всегда может использоваться вместо объекта базового класса — новые операции производного класса при этом просто игнорируются. Но это означает, что программа, работающая с такими объектами через интерфейс базового класса, не сможет использоваться новыми функциями, добавленными на более низких уровнях. Иначе говоря, различия между SimplePhone и SpeedCallingPhone заставляют программиста работать с каждым классом в его специфическом контексте. Функция, объявленная с параметром SimplePhone, примет объект SpeedCallingPhone, но не сможет задействовать возможности ускоренного вызова.

За счет наследования с расширением мы можем добавлять новые функции в производный класс, но не можем работать с ними через интерфейс базового класса. Данная модель наследования устанавливает жесткие ограничения и не позволяет организовать диспетчеризацию новых функций на стадии выполнения — одну из важнейших составляющих мощи объектной парадигмы. Если программисту приходится работать с каждым классом в иерархии наследования в его специфическом контексте, вся сила абстракции наследования теряется. Программист должен иметь возможность работать с большими наборами классов, интерпретируя их так, словно они являются экземплярами одного класса, являющегося их общим предком.

Другая крайность — использование абстрактного базового класса, определяющего только сигнатуру и полностью лишенного прикладной семантики. Отсутствие семантики гарантирует отсутствие семантических конфликтов с производными классами. Для примера рассмотрим «универсальный базовый класс» с операциями at, atPut, more, moreNow и getNext, которые работают с объектом через поля или рассматривают его как составную структуру данных с последовательным хранением элементов. Операция at может получать строку или значение перечисляемого типа — ключ, однозначно определяющий возвращаемое значение (ассоциативная выборка). Функция atPut может получать ключ и значение; ключ определяет поле, которому присваивается заданное значение (ассоциативное присваивание). Кроме того, некоторые операции atPut могут иметь побочные эффекты — при передаче специальных ключей инициируется выполнение внутренних функций класса. Рассмотрим простой пример ассоциативного массива:

```
class AssociativeArray {
public:
    void atPut(void *element, String key);
    void *at(const String&);
    ...
};
```

```
int main() {
    AssociativeArray a;
    a.atPut((void*)233, "AssociativeArray example");
    a.atPut((void*)230, "CircularList example");
    int circlistPage = (int)a.at("CircularList example");
                       // Должно быть 230
    a.atPut((void*)10, "!size");   // Изменение размера массива
    a.atPut((void*)0, "!print");   // Вывод содержимого массива
    int size = (int)a.at("!size"); // Получение размера массива
    ...
}
```

Ассоциативный массив используется для хранения номеров страниц, на которых находятся соответствующие разделы книги. Выборка осуществляется по ключу. Но наряду с обычными ключами существуют специальные ключи с префиксом !, используемые функциями at и atPut ради побочных эффектов: вывода, получения и изменения размера массива и других служебных операций.

Подобный стиль, часто называемый *каркасным* программированием, характерен для многих контекстов символических языков. Он приводит к появлению базовых классов, представляющих «толстый интерфейс», в котором объединяются функции множества несвязанных производных классов. Мы вернемся к этому стилю программирования в главе 8.

Подобные интерфейсы обладают почти неограниченной гибкостью; с другой стороны, они не способны выразить намерения проектировщика или обеспечить их соблюдение. Кроме того, их реализация будет недостаточно эффективной, потому что каждая операция требует поиска строки. Нам нужны полноценные классы, которые бы передавали общую семантику интерфейса, не ориентируясь на конкретную реализацию. Интерфейсы такого рода характеризуются базовыми классами, которые служат «заготовками» для реализаций, определяемых позднее в производных классах. Базовый класс, возглавляющий иерархию, может не использоваться для создания объектов, а лишь определять интерфейс к объектам производных классов на стадии компиляции. Базовый класс характеризует общее поведение всех классов, находящихся в иерархии под ним. Например, мы можем объявить базовый класс Window с операциями move(Point), addChar(char), clear(), deleteLine() и т. д., а также ряд производных классов: CursesWindow, XWindow, MSWindow и SunViewWindow. Самостоятельные экземпляры Window никогда не создаются в программах; они просто не умеют ничего делать! Базовые классы, обладающие таким свойством, называются *абстрактными базовыми классами*. Абстрактные базовые классы C++ подробно описаны в разделе 5.4. Иерархия Window выглядит примерно так:

```
class Window {
public:
    virtual void    addChar(char)         = 0;
    virtual void    clear()               = 0;
    virtual void    deleteLine()          = 0;
};
```

```
class CursesWindow: public Window {
public:
    void    addChar(char c)  { /* Подходящий алгоритм */ }
    void    clear()          { .. }
    void    deleteLine()     { ... }
};

class XWindow: public Window {
public:
    void    addChar(char c)  { /* Подходящий алгоритм */ }
    void    clear()          { .. }
    void    deleteLine()     { ... }
};

// И т. д.
```

Все функции класса **Window** объявлены чисто виртуальными, и класс не содержит данных. Такие классы называются *чисто абстрактными базовыми классами* — они характеризуют абстрактный тип данных, но ничего не говорят о его реализации.

Замена функций базового класса в производном классе называется *наследованием с подменой*. Подмена функций базового класса в производном классе должно сохранять их семантику. Допустим, вам потребовалось написать виртуальную функцию с учетом контекста, в котором производится вход в функцию и выход из нее [8]. Таким контекстом является состояние текущего объекта (возможно, дополненное некоторой внешней информацией о состоянии). При входе виртуальная функция базового класса должна предполагать о контексте не меньше, а при возврате — не больше, чем ее аналог в производном классе. Входной критерий основан на изменении контекста при вызове функции базового класса для объекта производного класса, то есть в случае, когда вызов обрабатывается функцией производного класса. Так как функция производного класса должна быть способна обработать любые запросы, обрабатываемые версией базового класса, она не может выдвигать более широкие предположения, чем версия базового класса. Для выходного критерия справедливо обратное утверждение: функция производного класса должна делать, по крайней мере, не меньше того, что ожидается от версии базового класса, хотя может делать и больше.

Если эти критерии выполняются, функция производного класса называется *семантически совместимой* с функцией базового класса. Они очевидны в тех предметных областях, в которых отношения между типами имеют формальную основу, например, в иерархиях математических типов (вещественные, рациональные, неотрицательные рациональные, кардинальные и т. д.). Установление подобных критериев в областях графики, экономических систем, телекоммуникаций и большинстве других областей, с которыми связана основная часть нашей работы, потребует несколько больших усилий.

Обратите внимание: чисто виртуальные функции абстрактных базовых классов заведомо удовлетворяют этим критериям. Все они знают о контексте вызова

лишь то, что им передается заданное количество параметров некоторых типов. Эти функции не содержат кода, а следовательно не меняют свой контекст, а вся информация о выходном контексте сводится к типу возвращаемого значения.

В распространенной ситуации, когда виртуальная функция «нечистого» базового класса подменяется в производном классе, эти две функции должны обладать, по крайней мере, синтаксическими различиями — если бы их не было, производный класс мог бы использовать версию базового класса. Тем не менее, эти две функции должны быть семантически совместимыми (см. выше). Сохранение имени функции не гарантирует совместимости семантики. Проектировщик должен проследить за тем, чтобы функция производного класса обладала по отношению к производному классу такой же семантикой, как функция базового класса — по отношению к базовому классу.

Итак, в наследовании существуют две крайности. Первая — абстрактные базовые классы, когда сигнатуры всех классов в иерархии совпадают, но новые производные классы слегка изменяют семантику операций своих родителей. Другая крайность — наследование с расширением, при котором существующая реализация остается неизменной, а изменения интерфейса сохраняют полную совместимость снизу вверх. К сожалению, на практике большинство ситуаций применения наследования не относится ни к одному из этих крайних случаев. Абстрактные базовые классы, содержащие только виртуальные функции и не содержащие данных, встречаются редко. Хорошие абстрактные базовые классы проектируются для отражения общих аспектов поведения своих производных классов. Стремление к многократному использованию кода приводит к тому, что общие функции и данные реализации выделяются в базовый класс. Они образуют «поведение по умолчанию», которое по необходимости подменяется в производных классах. Если позаботиться о сохранении семантики функций (поведения) при переходе от базового класса к производным, желаемая архитектурная абстракция иерархий наследования будет обеспечена.

С другой стороны, наследование с расширением тоже создает свои проблемы. Если анализ приложения показывает, что некоторая функция принадлежит только производному классу, но не входит в его базовый класс, часть архитектурной абстракции утрачивается. Для примера возьмем библиотеку Shape с замкнутыми и незамкнутыми фигурами (рис. 6.8). Если мы собираемся добавить функцию fill для закраски внутренней области фигуры, куда ее следует включать? Если в класс Shape, то следует ли объявить ее чисто виртуальной функцией или наделить каким-нибудь стандартным поведением? Объявление функции чисто виртуальной приведет к тому, что она будет определена для линии (Line) и других незамкнутых фигур, для которых она не имеет смысла. Определить для функции Shape::fill некое стандартное поведение? Но поведение, стандартное для окружности (Circle), скорее всего не имеет смысла для класса Line и его потомков, поэтому такой вариант тоже не годится. С семантической точки зрения правильнее всего было бы определить fill чисто виртуальной функций в классе замкнутых фигур (ClosedShape), но это означает, что приложение, которому потребуется закрашивать фигуры, наряду с интерфейсом Shape должно видеть интерфейс ClosedShape.

А это, в свою очередь, означает один лишний класс, о котором придется помнить программисту, одну лишнюю страницу документации и один лишний интерфейс, изменение которого способно нарушить работу приложения. Наконец, если бы классы замкнутых и разомкнутых фигур (ClosedShape и OpenShape) не входили в исходную иерархию наследования (что вполне вероятно, если функция fill не учитывалась в первоначальной архитектуре), найти семантически обоснованное место для fill становится еще сложнее. Обычно в подобных ситуациях программисту приходится слегка лукавить насчет семантики и определять функцию fill в классе Shape как виртуальную функцию с пустым телом. Другой, более строгий вариант — запускать исключение из вызова Shape::fill.

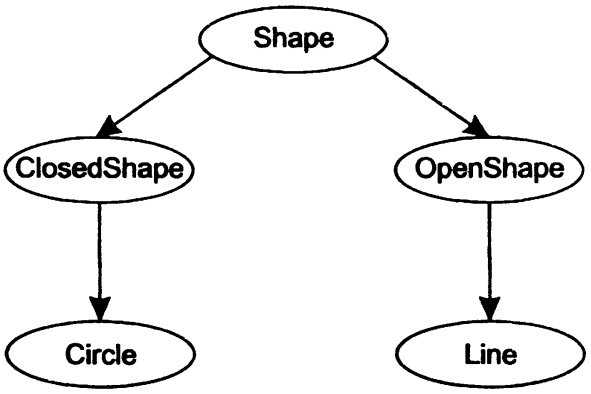


Рис. 6.8. Еще одна библиотека с геометрическими фигурами

Большинство реальных случаев лежит где-то посередине между «полюсами» наследования с чистым расширением и абстрактными базовыми классами, поэтому программисты часто ошибаются с выбором отношений субтипизации. Можно сказать, что неудача в приведенном выше примере, в котором класс Set определялся производным от List с подменой функции insert(void*), произошла именно по этой причине. Такое «примесное» наследование часто встречается в программах и создает массу сложностей с сопровождением и эволюцией, причем программист даже не понимает настоящей причины своих бед. Проблема описана в литературе и известна на практике, но программирование, основанное на наследовании вместо субтипизации, рождает ее снова и снова. Положение усугубляется полиморфизмом.

Учтите, что проблемы возникают только при наличии виртуальных функций и полиморфном использовании взаимосвязанных классов. Предположим, классы Set и List в виде, представленном на с. 229 (наследование с подменой), рассматриваются как совершенно самостоятельные классы. Другими словами, объекты List всегда интерпретируются как List, а объекты Set всегда интерпретируются как Set, и программа никогда не делает попыток заменить один объект другим. В этом случае подмена не создает нежелательных последствий; оно превращается в артефакт решения, не связанный с архитектурой приложения. Вопрос субтипизации между этими двумя абстракциями (и обусловленная ею возможность использования одной абстракции вместо другой) никогда не возникает в области приложения. Но если функции List объявлены виртуальными и их виртуальность используется в программе, свойства подмены оказываются задействованными,

что может привести к проблемам. Глубинная связь между наследованием (отражающим сходство реализаций или отношения субтипизации) и полиморфизмом (конструкцией области решения, требуемой для поддержки субтипизации в предметной области) делает этот вопрос настоящей проблемой.

Давайте снова предположим, что классы List и Set рассматриваются как отдельные объекты. Решение на с. 231 дает пример объединения классов Set и List без наследования. Вместо этого объект Set *перенаправляет* запросы своему внутреннему объекту List. Два взаимодействующих типа, один из которых перенаправляет часть своих операций другому типу, редко наводят на мысль о применении субтипизации. Впрочем, у этого правила имеются исключения, особенно в нетривиальных идиомах C++; они будут рассмотрены позднее. Но в общем случае перенаправление способно обеспечить полиморфизм (или видимость полиморфизма) без ненужного наследования. К недостаткам перенаправления следует отнести то, что по сравнению с наследованием оно обеспечивает более низкую безопасность типов на стадии компиляции и хуже отражает логику проектирования.

Наследование с сокращением

Некоторые среды объектно-ориентированного программирования позволяют *сокращать* производные классы, то есть исключать из них свойства базовых классов. C++ не поддерживает сокращения при открытом наследовании — такая поддержка на стадии компиляции противоречит возможности преобразования объекта производного класса в объект базового класса во время выполнения. Следующий фрагмент кода поясняет сказанное:

```
class Base {
public:
    void function1();
    void function2();
};

class Derived: public Base {
private:
    Base::function1;      // Сокращение (запрещено в C++)
public:
    void function3();
};

void f(Base *baseArg) {
    baseArg->function1();  // Сюрприз: сокращение игнорируется
}

int main() {
    Derived *d = new Derived;
    f(d);
    return 0;
}
```

Если бы сокращение было возможно, компилятору пришлось бы провести абсолютно полный анализ потока данных и убедиться в том, что исключенные свойства объекта нигде и никогда не потребуются. Проблема становится еще более серьезной при использовании виртуальных функций с раздельной компиляцией. Один модуль, обладающий доступом к объявлению производного класса, создает объект производного класса. Этот объект передается через указатель на базовый класс (допустимое преобразование) функции другого модуля, которая *не имеет* доступа к объявлению производного класса и ничего не знает о сокращении. Второй модуль сможет вызывать функции базового класса для объектов, из которых эти функции были исключены! Подумайте, что произойдет в приведенном фрагменте, если функции function1 и function2 объявить виртуальными, а функцию f компилировать в отдельном исходном файле, в котором видно только объявление Base.

Поддержка сокращения на стадии выполнения привела бы к чрезмерному усложнению языка. С каждым объектом класса пришлось бы передавать дополнительные данные, указывающие, какие функции класса следует замаскировать как «сокращенные».

Данные базового класса можно сделать недоступными для производного класса и его клиентов, но их не удастся ликвидировать (скажем, для экономии памяти в объектах производного класса). А что касается виртуальных функций, вызов функции, прошедший проверку типов во время компиляции, заведомо вызовет *что-то* на стадии выполнения — вместо того, чтобы выдать сообщение «я вас не понял», как в языках типа Smalltalk. Если производный класс не содержит функций с соответствующим именем, будет использована функция базового класса с подходящими именем и сигнатурой.

С неполиморфным наследованием дело обстоит несколько иначе. Как было показано ранее, сокращение на стадии компиляции посредством изменения видимости запрещено по правилам языка. Например, следующий фрагмент невозможен:

```
class List {
public:
    void* head();
    void* tail();
    int count();
    long has(void*);
    void insert(void*);
};

class Set: public List {
public:
    void insert(void* m);
private:
    List::count;        // Ошибка
    List::has;          // Ошибка
};
```

Однако в случае закрытого наследования (фактически исключающего *все* открытые операции базового класса) доступ к операциям базового класса может предоставляться по отдельности для каждой операции:

```
class Set: private List {
public:
    void insert(void* m);
    List::count;
    List::has;
};
```

Наследование с сокращением нередко является признаком неправильной структуры наследования, например, обратного направления связей или перекосов в отношениях «родитель/потомок». Вообще говоря, сокращение порождает массу проблем, и его не рекомендуется применять как методику проектирования (даже разовые применения нежелательны).

Потребность в функциях классов с «истинной» семантикой

В области объектно-ориентированных проектирования и программирования операции с объектами соответствуют некоторой семантике реального мира. Это утверждение более сильное и глубокое, чем простое заявление о том, что работа с объектами должна вестись через хорошо определенные интерфейсы. Сохранение семантической тождественности является залогом успешного развития проекта. Инкапсуляция прежде всего определяет не столько местонахождение данных или видимость операций, сколько организацию их поведения.

Для пояснения сказанного рассмотрим пару примеров. Эти примеры закладывают основу для анализа связей между инкапсуляцией и наследованием в последующих разделах.

Пример 1. Операции get и set

Программисты часто воспринимают рекомендации объектно-ориентированного проектирования в форме «открытые данные вредны; данные всегда необходимо инкапсулировать, а доступ к ним должен производиться только через операции get и set». Один из недостатков такого подхода — его выражение в контексте внутренних данных, относящихся исключительно к области решения. Другая проблема состоит в том, что этот подход ничего не говорит о целостном поведении объекта или представляемого им типа предметной области. Если функция ничего не говорит об объекте в целом или сигнатуре его поведения, она начинает выглядеть неестественно. Если она никак не связана с предметной областью, напрашивается предположение, что она создавалась на базе области решения. Последнее обстоятельство не так уж страшно, но наличие операций, не связанных с предметной областью, затрудняет понимание логики класса и его эволюцию.

Имена функций должны быть четкими и семантически насыщенными; они должны иметь понятный смысл или интерпретацию в контексте своего класса. Присутствие функций get и set часто свидетельствует о том, что архитектура приложения нуждается в пересмотре. Что делает объект, обращающийся к внутренним данным, и за что он отвечает? Чего добивается клиент, изменяющий состояние внутренних данных? Подобные вопросы помогают связать функции класса с предметной областью и отделить то, что должен знать класс, от тех аспектов, которые видны его клиентам. Функции класса, предназначенные исключительно для доступа к внутренним данным, не обеспечивают инкапсуляцию за счет соблюдения правил видимости C++: наоборот, они работают по правилам видимости C++ для нарушения инкапсуляции.

Пример 2. Открытые данные

Конечно, наличие открытых данных у класса нарушает инкапсуляцию. Присутствие внутреннего объекта B в открытом интерфейсе другого объекта A загромождает интерфейс A поведением B. Маловероятно, чтобы поведение B выражалось для его клиентов в контексте A: работа с B должна осуществляться функциями класса A. В этом случае интерфейс A остается однородным и «гладким», а пользователю не приходится спускаться на один уровень вглубь для достижения нужного результата:

Неправильно:

```
class Window {
public:
    Cursor cur;
    ...
};


int main() {
    Window win;
    win.cur.move(10,5);
    ...
}
```

Правильно:

```
class Window {
public:
    void MoveCursor(int,int);
    ...
private:
    Cursor cur;
};
int main() {
    Window win;
    win.moveCursor(10,5);
    ...
}
```

Вызов win.moveCursor лучше выражает намерения программиста, чем win.cur.move, потому что он понимается на одном уровне абстракции вместо двух.

При наследовании пример 2 становится еще интереснее. В [9] указывается, что в большинстве объектно-ориентированных языков производные классы могут обращаться к внутренним данным базовых классов, которые в остальных случаях недоступны извне. К C++ это не относится; таким образом, в C++ дело обстоит не так плохо, как в общем случае. Тем не менее, пользователи C++ могут реализовать описанную в [9] популярную семантику, присвоив полям данных атрибут protected. О том, как это связано с нарушением инкапсуляции, рассказывается в следующем разделе.

Наследование и независимость классов

Объектно-ориентированное проектирование базируется на абстракции данных. Абстракция данных как методика обобщения обычно тесно связана с *инкапсуляцией*, то есть *механизмом сокрытия информации*. Хотя абстракция данных означает попытку скрыть от пользователя неприятные подробности реализации, при желании пользователь может получить доступ к «внутренностям класса». С другой стороны, инкапсуляция означает яростную защиту от любых попыток преодолеть интерфейс и ознакомиться с внутренним строением класса. Абстракция до некоторой степени относится к предметной области, а инкапсуляция (опять же до некоторой степени) — к области решения.

Нельзя сказать, что абстракция данных и сокрытие информации необходимы или достаточны для объектно-ориентированного проектирования, но они являются важными компонентами хорошего объектно-ориентированного стиля и «по-хорошему» независимы от объектно-ориентированного программирования. Кроме того, на них интересно взглянуть в свете непрекращающихся споров о том, нарушает ли наследование принципы абстракции данных.

Сокрытие данных обладает тремя свойствами, которые обязательно должны сохраняться и которые служат «лакмусовой бумажкой» для выявления нарушений инкапсуляции или сокрытия информации. Здесь они формулируются в контексте изменений программы, поскольку их практическая ценность проявляется именно при эволюции программы. Под «изменениями» здесь подразумевается нечто большее, чем традиционные модели модификации внутреннего представления или функций абстрактного типа данных, хотя они составляют очевидное и наиболее распространенное направление эволюции программ. В изменениях также следует учитывать фактор наследования, когда новые классы строятся посредством модификации поведения существующих классов.

- Приводит ли изменение в одном классе к необходимости пересмотра архитектурных решений в других классах, или классы достаточно автономны?
- Возможно ли, чтобы некоторые операции с классом (например, внутренняя модификация, внешнее применение наследования или внешнее применение перенаправления) могли изменить семантику или поведение класса? (На этот критерий можно взглянуть иначе: спросите себя, какие компоненты системы придется тестировать заново при внесении изменений.)
- К каким затратам (например, времени на перекомпиляцию) приведет изменение класса? Пропорциональны ли эти затраты размеру класса; размеру тех классов, с которыми он взаимодействует; размеру программы, в которой этот класс используется; размеру всего проекта, содержащего класс?

Здесь мы будем рассматривать абстракцию и сокрытие данных только с точки зрения наследования — а именно с точки зрения того, когда наследование создает особые проблемы или открывает особые возможности.

C++ позволяет управлять доступностью членов базового класса (данных или функций) для производных классов. Следовательно, реализацией семантики

сокрытия данных и инкапсуляции занимается программист C++. Стандартное открытое наследование обладает вполне разумной семантикой обеспечения инкапсуляции.

- Закрытые члены базовых классов недоступны (во всяком случае, напрямую) для производных классов.
- Открытые члены базовых классов доступны для производных классов не в большей и не в меньшей степени, чем для любого другого кода.
- Члены, объявленные открытыми в базовом классе, становятся частью сигнатуры (открытого интерфейса) производного класса.

Но при этом остается одна серьезная проблема. Хотя языковая поддержка абстракции данных в C++ ограничивает семантическую видимость изменений закрытых данных, она по-прежнему серьезно отражается на перекомпиляции. В большинстве сред программирования на C++ приходится учитывать последствия.

- Изменение представления базового класса требует перекомпиляции кода всех производных классов.
- Включение виртуальных функций в базовый класс требует аналогичной перекомпиляции.
- Возможно, при включении невиртуальных функций в базовый класс перекомпилировать производные классы не придется. Но если новая функция замещает другую одноименную функцию (в *своем* базовом классе), то придется перекомпилировать весь код, в котором вызывается функция с этим именем.

В C++ существует и другая лазейка для нарушения инкапсуляции. Предположим, класс **Shape** объявляет свое поле **center** защищенным — иначе говоря, поле **center** доступно для «своих» (производных классов»), но должно быть недоступным для «чужих»:

```
class Shape {
public:
    ...
    virtual void draw() = 0;
    virtual void move(Point) = 0;
protected:
    Point center;
};
```

Класс **Ellipse** может наследовать от **Shape** и получить доступ к полю **center**. В данном случае налицо сознательное нарушение инкапсуляции, но придавать ему слишком большое значение было бы глупо, особенно если класс **Shape** является абстрактным базовым классом:

```
class Ellipse: public Shape {
public:
    ...
```

```
    virtual draw();
};
Ellipse anEllipse;
```

Теперь программа может работать с anEllipse, но язык не позволит напрямую манипулировать с center. Тем не менее, существует обходной маневр:

```
class Cheat: public Ellipse {
public:
    void cheat(Point p) { center = p; }
};
Cheat anEllipse;
```

Наследование от Ellipse позволяет Cheat нарушить абстракцию не только Ellipse, но и Shape. Это противоречит здравому смыслу и нарушает принцип «наименьшей неожиданности»: добавление класса Cheat вроде бы никак не связано с классом Shape, но это изменение привело к нарушению инкапсуляции Shape. Следующий пример еще коварнее:

```
Point center;   // Центр всего изображения

class Circle: public Ellipse {
public:
    void incidentalCenterReference() {
        ... center ...
    }
};

Circle aCircle;
```

Допустим, создатель Circle собирался обращаться к center для определения значения поля Shape::center объекта Circle. Если в результате изменений класса Shape поле center будет удалено, переименовано или объявлено закрытым, то обращения окажутся связанными с глобальной переменной center; программа будет компилироваться без ошибок, но с неправильной семантикой. Наследование нарушает абстракцию, и это происходит только потому, что автор Shape предоставил защищенный статус некоторым переменным класса. Это сознательный компромисс, часто применяемый для оптимизации быстродействия, эффективного расходования памяти или многократного использования кода. Мы вернемся к теме защищенных полей и многократного использования в главе 7.

В общем случае, хотя программист может обеспечить лингвистическую независимость классов C++ при помощи конструкций ограничения доступа, семантика языка не позволяет избежать широкого распространения последствий внесения изменений. Для небольших программ, в которых полная перекомпиляция приемлема, это не проблема. В более крупных проектах, где перекомпиляция в идеальном случае должна ограничиваться измененным кодом, применяются специальные идиомы, снижающие последствия изменений. Такие идиомы описаны в главах 5 и 8.

6.6. Практические рекомендации

Иерархии наследования, в которых функции производных классов подменяют виртуальные функции базовых классов, в значительной степени определяют мощь объектных технологий, но пользоваться ими нужно с осторожностью. Ниже перечислены ситуации, в которых рекомендуется применять наследование.

- Наличие формальных или интуитивно очевидных отношений субтипизации между базовым и производным классами, когда подмена приводит к сужению подмножества объектов базового класса, представленных производным классом. Хорошим примером служит наследование Circle от Ellipse с подменой операции вращения rotate(Angle). Впрочем, даже этот пример не идеален, потому что с точки зрения типов операция rotate(Angle), определенная для эллипсов, прекрасно работает с окружностями, и подменять ее незачем. Ничего не поделаешь — этот мир вообще не идеален.
- Совместимость семантики базового класса с семантикой производного класса. При входе виртуальная функция базового класса должна предполагать о контексте не меньше, а при возврате — не больше, чем ее аналог в производном классе. Чисто виртуальные функции абстрактных базовых классов заведомо удовлетворяют этому критерию.
- Программы, в которых фактически выполняется наследование с расширением, а объявление функции для удобства включено в интерфейс базового класса. Субклассы дополняются собственными специфическими операциями. При условии безопасности и семантической осмысленности реализации этих операций в классах-соседях (классах, имеющих общий базовый класс), включая пустую реализацию, такие операции можно включать в виде виртуальных функций в общий базовый класс с пустым телом. Например, класс Shape с производными классами ClosedShape и OpenShape может содержать операцию fill. Базовый класс реализует ее в виде пустой операции, и эта семантика наследуется незамкнутыми фигурами, для которых заполнение не имеет смысла. Операция подменяется только в классах, производных от ClosedShape. Говорят, что такие базовые классы обладают *толстым интерфейсом*; злоупотреблять этой методикой не рекомендуется.
- *Отсутствие* отношений «LIKE-A» между базовым и производным классом; классы должны быть связаны отношением «IS-A». Если классы связаны отношением «LIKE-A», найдите или создайте базовый класс, который бы они могли использовать в качестве общего родителя.

По поводу сокрытия данных в C++ тоже можно сформулировать некоторые рекомендации.

- C++ *требует*, чтобы предположения, сделанные на стадии проектирования, пересматривались в свете изменений класса только при изменении символических характеристик, экспортируемых классом. Иначе говоря, тип характеризуется своей сигнатурой (открытым интерфейсом), и компилятор C++ следит за согласованностью сигнатур между определением класса и его использованием.

- Даже при неизменности интерфейса трудно сказать, не приведут ли семантические изменения класса к нарушению работы классов, взаимодействующих с ним. Впрочем, в этой области обычно хватает простой интуиции программиста, а в большинстве грамотно спроектированных классов включение резервируется для структур данных и алгоритмов, доступ к которым должен быть ограничен.
- Старайтесь обеспечить инкапсуляцию базовых классов, избегая использования защищенных данных. С точки зрения производных классов защищенные данные слишком легко превращаются в открытые. Тем не менее, всегда отдавайте предпочтение защищенным данным перед дружественными и открытыми данными там, где это уместно.
- В общем случае изменение реализации класса требует перекомпиляции кода классов, производных от измененного или иначе зависящих от его интерфейса. В системах, поддерживающих поэтапную загрузку кода C++ в функционирующих программах (например, во встроенных системах, работающих в непрерывном режиме), любые изменения в реализации класса с большой вероятностью потребует замены или обновления всех существующих объектов этих классов.

Другое полезное правило: в общем случае функции классов, размещаемые в библиотеках, следует объявлять виртуальными, даже если в исходном варианте приложения производные классы отсутствуют. Это обеспечит поддержку наследования позднее, если будут обнаружены отношения субтипизации между новым классом и библиотечными классами. Чтобы классы C++ могли пользоваться всеми преимуществами объектно-ориентированного проектирования, их функции должны быть виртуальными.

По аналогии можно предположить, что все наследования следует объявлять виртуальными на тот случай, если в будущем вдруг потребуется перейти на множественное наследование. Тем не менее, такие решения обычно предоставляются проектировщику новых производных классов, и их не всегда удается предусмотреть при проектировании родителей. В каком-то смысле виртуальность родительских классов должна выбираться по усмотрению их потомков; тем не менее, C++ заставляет принимать это решение на более высоких уровнях иерархии наследования. Возможно, в будущих версиях языка ситуация изменится.

Упражнения

1. Какие из отношений «IS-A», «HAS-A», «USES-A» и т. д. обладают свойствами:
   - транзитивности;
   - ассоциативности;
   - коммутативности.
2. Рассмотрим следующий пример:

```
class CopyMachine {
public:
```

```
    ...
};

class Secretary: public Employee {
public:
    Secretary(const char *n): Employee(n) { }
private:
    static CopyMachine copyMachine;
};

Secretary Connie("Connie"), Terri("Terri");
```

- Какими отношениями связаны классы **Secretary** и **CopyMachine**? А объекты **Connie**, **Terri** и **CopyMachine**?
- Если класс **Manager** связан отношением «HAS-A» с классом **Secretary**, а класс **Secretary** — отношением «HAS-A» с объектом **CopyMachine**, то в каком отношении находятся классы **Manager** и **CopyMachine**?

3. Реализуйте пример **Account**, приведенный в этой главе.
4. Реализуйте ассоциативный массив, содержащий функции **at** и **atPut**, а также функции ортодоксальной канонической формы.
5. Разработайте новую реализацию ассоциативного массива, которая бы использовала в качестве ключей значения перечисляемого типа. Сравните быстродействие двух ассоциативных массивов.

Литература

1. Whitehead, A. N., and Russell, B. A. W. «Principia Mathematica». Cambridge, Mass.: Cambridge University Press, 1910.
2. Ungar, David, and Randall B. Smith. «Self: The Power of Simplicity». SIGPLAN Notices 22,12 (December 1987).
3. Beck, Kent, and Ward Cunningham. «A Laboratory for Teaching Object-Oriented Thinking». SIGPLAN Notices 24,10 (October 1989).
4. Brachman, Ronald J. «What IS-A and Isn't: An Analysis of Taxonomic Links in Semantic Networks». IEEE Computer 16,10 (October 1983).
5. Booch, Grady. «Object-Oriented Design with Applications». Redwood City, Calif.: Benjamin/Cummings, 1991.
6. Liskov, Barbara. «Data Abstraction and Hierarchy». SIGPLAN Notices 23,5 (May 1988).
7. Sethi, R. «Programming Languages». Reading, Mass.: Addison-Wesley, 1989.
8. Meyer, Bertrand. «Object-Oriented Software Construction». Englewood Cliffs, N.J.: Prentice Hall, 1988, 257.
9. Snyder, Alan. «Encapsulation and Inheritance in Object-Oriented Programming Languages». SIGPLAN Notices 21,11 (November 1986).

Глава 7
Многократное использование программ и объекты

Под возможностью многократного использования понимается нечто, помогающее предотвратить повторное выполнение уже выполненной работы. Мы можем заново задействовать старые идеи, архитектурные решения и основные системные интерфейсы, исходные коды программ и объектов. В настоящей главе основное внимание уделяется созданию программ, пригодных для многократного использования в контексте объектной парадигмы вообще и C++ в частности.

Многократное использование программ — обширная тема, и ее полное рассмотрение выходит за рамки темы книги. Хотя объектно-ориентированные проектирование и программирование способствуют многократному использованию, следует подчеркнуть, что многократное использование относится к области разработки; это отдельная область знаний с собственной методологией и правилами проектирования. Не следует полагать, будто многократное использование достигается автоматически в результате качественного проектирования или реализации, а то и просто по счастливой случайности. На самом деле многократное использование продумывается заранее и обеспечивается только применением соответствующих механизмов проектирования, документирования, управления и распространения. Не путайте многократное использование с *вторичным использованием* — практикой поиска и реконструирования готового кода, подходящего для выполнения некоторой задачи. Также многократное использование кода следует отличать от *консервации* программного кода, то есть сохранения значительной части программного кода в нескольких версиях системы. Вторичное использование и консервация тоже приносят пользу, но не в таком объеме, как хорошо написанная программа, рассчитанная на многократное использование.

В этой главе основное внимание уделяется языковым конструкциям и идиомам многократного использования C++ с дополнительным акцентом на проектирование. Тема многократного использования затрагивается и в других частях книги. Так, раздел 6.3 посвящен анализу предметной области — важному аспекту проектирования, рассчитанного на многократное использование. В главе 11 многократное использование программ рассматривается с системной точки зрения. У многократного использования есть и другие аспекты, здесь не рассматриваемые, например, документирование, организация архивов многократно используемого

кода и средства просмотра содержимого таких архивов. За более подробной информацией о социологических и управленческих аспектах разработки, рассчитанной на многократное использование, обращайтесь к [1].

Многократное использование кода относится к области решения. Одним из средств многократного использования является наследование — производный класс получает доступ к функциональности базового класса, и ее не приходится программировать заново. В области объектно-ориентированных технологий многократное использование показало неплохой выигрыш в производительности.

Объектно-ориентированное проектирование иногда путают с проектированием, рассчитанным на многократное использование в объектной парадигме. Казалось бы, если деление на классы способствует многократному использованию, то лучшей объектно-ориентированной архитектурой будет та, в которой наиболее эффективно применяется наследование. Также можно подумать, что соблюдение естественных отношений между типами, обусловленных предметной областью, автоматически сведет к минимуму дублирование кода. Оба предположения неверны, но самое опасное — не понять различий между ними.

Чтобы обнаружить возможности многократного использования, следует провести поиск сходных сущностей. В программах это сходство часто отражается посредством наследования. Но необходимо помнить, что наследование является концепцией области решения, применяемой как для многократного использования, так и для отражения отношений субтипизации. Чтобы выбрать способ отражения сходства между сущностями в программе, необходимо понять суть отношений между объектами: связаны ли они отношениями «IS-A» или «LIKE-A»? Если речь идет об отношениях «IS-A», следует ориентироваться на многократное использование *архитектуры*, а не *кода*. В случае отношений «LIKE-A» существует возможность многократного использования кода. Но при этом необходимо проявить осторожность и не пытаться организовать многократное использование посредством открытого наследования. В разделе 6.5 мы уже пытались реализовать многократное использование за счет наследования класса Set от класса List. До некоторой степени это получилось, но при попытке применения объекта производного класса в контексте базового класса все развалилось. «Конструктивная совместимость» объектов занимает центральное место в объектно-ориентированном проектировании, а ее связь с многократным использованием чисто случайна. Обычно решение о применении открытого наследования должно приниматься на базе соображений, изложенных в главе 6.

Рассмотрим класс Set, сходный (like a) с классом List. Как в C++ выразить многократное использование кода одного класса другим классом? Мы выяснили, что класс Set «похож» на класс List; оба класса обладают свойствами контейнеров, а их сигнатуры имеют много общего. Также выяснилось, что класс Set не является частным случаем List, или наоборот. Важно заметить, что хотя сходство между реализациями двух классов может указывать на возможность многократного использования, это еще не означает, что один класс должен быть производным от другого. Два класса могут быть связаны так, словно существует третья абстракция, обладающая общими свойствами двух других. Множество (Set) является

частным случаем коллекции (Collection), и список (List) тоже является частным случаем коллекции, поэтому оба класса можно объявить производными от класса Collection, содержащего код и данные, общие для Set и List.

Существует и другой способ совместного использования кода: включение экземпляра одного класса в другой класс и перенаправление вызовов функций внешнего класса внутреннему классу. Этот способ позволяет реализовать класс Set с классом List без наследования. Иерархии *объектов* (в отличие от иерархий классов) способны обеспечить многократное использование абстракций, даже не связанных отношениями «LIKE-A». Подобные случаи иногда называются *иерархиями реализации*.

В этой главе мы изучим связь между субтипизацией и наследованием в контексте многократного использования. Сначала будут проанализированы слабости отношений «субтип/производный класс» с точки зрения создания универсальных библиотек, рассчитанных на многократное использование. Затем мы разберемся, что же именно может применяться многократно: архитектура, код или память. Все перечисленные направления достаточно перспективны, но одни перспективнее других; кроме того, одни хорошо сочетаются с объектно-ориентированным проектированием и C++, другие — нет. Многократное использование архитектурных решений является важнейшей составляющей успешного многократного использования в объектной парадигме. Многократное использование кода и памяти тоже приносит пользу, поэтому мы поговорим и об этих направлениях. В частности, будут рассмотрены шаблоны (вероятно, наиболее перспективный механизм многократного использования кода C++) и оптимизации, разработанные специально для шаблонов. Также будут затронуты темы отношений между перенаправлением, наследованием и многократным использованием, инкапсуляции и абстрактных типов данных. Глава завершается некоторыми практическими рекомендациями и обобщениями.

7.1. Об ограниченности аналогий

Если наследование применяется для отражения отношений «IS-A» между классами, многократное использование кода приносит лишь дополнительную (хотя и немаловажную!) пользу. Возможности многократного использования в иерархиях классов чаще обусловлены соображениями проектирования, нежели стремлением к многократному использованию как таковому — иначе говоря, эти возможности связаны с субтипизацией предметной области, о которой рассказывалось в главе 6. Но поскольку объектно-ориентированные методы программирования тесно связаны со структурами объектно-ориентированного проектирования, возможности многократного использования кода также обусловлены иерархиями субтипов. Хотя класс Imaginary объявлялся производным от Complex по архитектурным соображениям, в итоге класс Imaginary задействовал большую часть кода класса Complex, что можно рассматривать как положительный побочный эффект. В этом смысле объектно-ориентированное проектирование является ценным

средством многократного использования, а многократное использование свойств базового класса в производном классе — ценным средством проектирования. Тем не менее, многократным использованием часто оправдывают странные применения наследования, не имеющие ничего общего с субтипизацией. Хотя такой подход может принести сиюминутную выгоду, он ухудшает инкапсуляцию изменений в долгосрочной перспективе.

Планирование с расчетом на многократное использование в объектной парадигме должно начинаться с анализа предметной области на стадии проектирования. Классы должны проектироваться как модули многократного использования, но при этом адекватно отображаться на предметную область. Между архитектурой классов, хорошо подходящих для конкретного приложения, и архитектурой, рассчитанной на многократное использование вообще, возникают определенные трения. Одним из основных свойств объектной парадигмы является изоморфизм областей задачи и решения. Идентифицируя сущности в процессе системного анализа, мы обычно по своему усмотрению создаем для них аналоги в приложении. В результате такой архитектурной ориентации создаются классы, хорошо подходящие для конкретных программ или систем, но не для многократного использования вообще. Слишком буквальное восприятие изоморфизма областей задачи/решения ухудшает возможности многократного использования программ. Эффективное многократное использование требует расширения абстракции на всю предметную область (см. раздел 6.3).

Для примера возьмем классы, задействованные в архитектуре программы-редактора. Одним из таких классов может быть File. Сигнатура File формулируется на основании того, что этот класс должен знать для работы с Window, Editor, Keyboard, Session и другими классами. Один из подходов основан на «очеловечивании» классов; проектировщики берут на себя роль классов и разыгрывают различные сценарии взаимодействия или пытаются сформулировать видение своих потребностей в контексте приложения.

Архитектурные решения, принятые в процессе формирования класса (особенно редактора), могут ухудшить потенциал его многократного использования в нескольких отношениях. Первая категория проблем относится к восприятию класса File проектировщиком: для разных приложений File может обозначать несколько различающиеся понятия. Один проектировщик выбирает для File семантику «файл UNIX», а в другом, более общем представлении File определяется как абстрактный базовый класс с производным классом UnixFile. Обобщенное представление существенно расширяет потенциал многократного использования класса File, который может постепенно уточняться в производных классах вроде VSAMFile.

Другая категория проблем относится к тому, что же именно должен делать класс File. Для редактора File должен поддерживать чтение и запись. Современный редактор может реализовать в классе File поддержку контрольных точек, чтобы обеспечить восстановление от системных сбоев во время сеансов редактирования. Но для расширения возможностей многократного использования класс File

должен обладать средствами проверки логической целостности, сжатия и шифрования, которые могут отсутствовать в версии редактора. Кроме того, версия File для редактора может содержать дополнительные операции, лишние для более общих приложений, например, механизм контрольных точек.

На это можно возразить, что проектировщик должен выявить все основные операции для всех основных типов файлов. Если он не может этого сделать — значит, плохо справляется со своей работой. Но, во-первых, спроектировать абсолютно универсальный интерфейс для работы с файлами в принципе невозможно; количество видов информации, хранящейся в файлах, и достаточно примитивных операций, выполняемых с ними, слишком велико. На минуту попробуйте представить, что обязан поддерживать объект файла UNIX, чтобы использоваться редактором, компилятором (как исходные, так и объектные файлы) и операционной системой (I-узлы, операции с каталогами и т. д.). Во-вторых, «правильная» архитектура не может быть слишком «толстой» (то есть не может содержать слишком много кода, не относящегося к приложению). Таким образом, полная универсальность может оказаться недостижимой.

Некоторые из перечисленных проблем решаются переводом многократного использования на более высокий уровень абстракции и реализацией этой абстракции в контексте ее применения. Например, определение типа содержимого File (ASCII, EBCDIC или FIELDATA) можно отложить до момента выполнения фактических операций с объектом File, вместо его кодирования в поведении самого класса File. Одним из средств достижения этой цели является *параметрический полиморфизм*, поддерживаемый в C++ в виде шаблонов. Он расширяет возможности многократного использования и в большей степени ориентирует разработчика на проблематику проектирования, нежели на проблематику реализации. Чем теснее такие решения связываются с архитектурой, тем более гибким и мощным получится результат.

Другой способ повышения уровня абстракции основан на абстрактных базовых классах. В класс включаются чисто виртуальные функции для описания абстракции, предназначенной для многократного использования; эти функции реализуются пользователем класса в том контексте, в котором должна применяться эта абстракция. Абстрактные базовые классы и построение заготовок для многократного использования рассматриваются в главе 11.

7.2. Многократное использование архитектуры

В наши дни хорошо известно, что для многократного использования на «правильном» уровне следует сосредоточиться на проектировании, а не на реализации. Интеграция таксономий типов предметной области с проектированием, рассчитанным на многократное использование, приводит к созданию классов

с тщательно проработанной семантикой, поскольку в конечном счете многократно используется именно семантика, а не реализация. В [2] так описана связь между многократным использованием и объектно-ориентированным проектированием:

> Приспособляемость (и потенциал многократного использования) программных компонентов зависят от объема выполненного анализа предметной области и степени параметризации модуля. Кроме того, формы параметризации, поддерживаемые языком программирования, не всегда обеспечивают нужную степень или форму приспособляемости. Настраиваемые параметры языка Ada поддерживают многократное использование, но не могут заменить наследования. Впрочем, одних классов тоже недостаточно. Хотя некоторые языковые средства способствуют разработке программ, пригодных для многократного использования, решить эту проблему одними лишь языковыми средствами не удастся.

Анализ предметной области, а не конкретного приложения, позволяет создавать классы, пригодные для широкого круга приложений. Качественный анализ предметной области является сложной и трудоемкой задачей; может оказаться, что такой анализ может быть проведен лишь для некоторого подмножества классов. Особое внимание следует уделить обобщению классов, возглавляющих иерархии наследования, прежде всего абстрактных базовых классов. Производные классы могут быть более специализированными для конкретных приложений. Классы, выбираемые в качестве обобщенных абстракций для многократного использования, должны быть тщательно документированы и помещены в специальное хранилище: если программисты не смогут найти класс, то его многократное использование окажется невозможным.

Многократное использование архитектуры может означать в этом контексте одно из двух: либо многократное использование четко определенных пакетов, либо многократное использование *сигнатуры* класса (то есть совокупности аспектов его поведения, формирующей семантическую единицу для многократного использования). Вернемся к классу File, вроде бы подходящему для многократного использования. В операционной системе UNIX для файлов поддерживаются операции создания, удаления, последовательного чтения и последовательной записи. Также возможны операции произвольного чтения, проверки логической целостности на случай сбоя системы, переименования, сжатия и т. д. Вероятно, редактору понадобится небольшое подмножество этих операций: скорее всего, ему не потребуются операции проверки целостности, сжатия, произвольного доступа и переименования. Сохранение всего «балласта» только ради многократного использования усложняет программу и увеличивает ее объем. Строя иерархию класса File в виде набора классов с разными степенями детализации, вы предоставляете пользователю свободу выбора. Выбирая нужный уровень иерархии, программа находит абстракцию, подходящую для ее целей, и не обременяется лишним грузом. Прогнозирование применения абстракций типа File в разных приложениях и создание соответствующих уровней функциональности является

одним из полезных факторов проектирования. Освоение такого подхода требует времени и навыков ретроспективного анализа.

Многократное использование архитектурных решений сокращает фазу проектирования системы, которая занимает (или, по крайней мере, должна занимать) бóльшую, нежели кодирование, часть жизненного цикла программы. Если вам удалось воспользоваться готовой архитектурой, возможно, также удастся задействовать части готовой реализации. А может, будет лучше разработать реализацию заново (полностью или частично), чтобы она лучше подходила для потребностей приложения, создав «заглушки» для ненужных частей. Например, графический пакет может быть по-разному реализован для разных платформ в зависимости от набора аппаратных графических примитивов. Графические приложения смогут без изменений работать на разных платформах, при этом одна и та же архитектура будет многократно использоваться с разными реализациями. Впрочем, этот пример относится скорее не к многократному использованию, а к адаптации одного приложения для разных платформ. При многократном использовании более важную роль играют те аспекты, которые обеспечивают сохранение работоспособности между приложениями.

Затраты времени на поддержку многократного использования новых классов в будущих проектах могут перевесить экономию от многократного использования архитектурных решений. Проектировщику приходится тратить время на анализ предметной области, документирование и публикацию новых абстракций. Общая экономия времени достигается только в долгосрочной перспективе.

7.3. Четыре механизма многократного использования кода

Иногда нам удается разработать хорошую архитектуру, а возможности для многократного использования, открывшиеся благодаря грамотному применению наследования, обусловленного субтипизацией, с лихвой окупают затраченные усилия. Но в контексте многократного использования наследование обычно применяется как особая разновидность директив #include, то есть как механизм автоматического включения избранных частей одного класса в другой. Главное преимущество этого механизма состоит в том, что он работает на уровне объектного кода и избавляет от необходимости копировать исходные тексты.

Проблема многократного использования программного кода относится к области решения. Как организовать многократное использование кода без нарушения иерархии классов, хорошо интегрированных по отношению к предметной области? В случае перенаправления строится новый объект для взаимодействия с существующим объектом, причем система рассматривает совокупность двух объектов как один объект. Перенаправление особенно часто применяется к классам, связанных тесными отношениями «LIKE-A», но не дотягивающих до отношений «IS-A». Снова вспомните пример с классами Set и List (см. раздел 6.5). Перенаправление

естественно сочетается с абстракцией данных; класс Set может пользоваться услугами встроенного в него объекта List. Другой пример — класс Menu, использующий код Window.

Единицей многократного использования архитектуры является интерфейс класса. Но какой должна быть единица многократного использования кода? Многие полагают, что ее границы тоже должны совпадать с границами классов, а классы как «естественные» абстракции также должны быть многократно используемыми блоками в модели наследования. О наследовании речь пойдет позже, а пока следует сказать, что перенаправление обеспечивает более точную детализацию многократного использования на уровне отдельных функций класса.

Помимо перенаправления механизмом многократного использования кода является делегирование, которое не поддерживается напрямую в C++, но встречается в Actors, self и ряде других языков (см. также с. 166). При делегировании вызов функции для одного объекта автоматически перенаправляется другому объекту; аналогичным образом перенаправляются обращения к переменным класса. Два взаимодействующих экземпляра воспринимаются извне как один. Если один объект делегирует вызов функции другому, то функция второго объекта выполняется в контексте пространства имен первого объекта. Делегирование характерно для языков, в которых классы не поддерживаются: для таких языков оно играет ту же роль, что наследование для классических языков. В C++ для выполнения одного объекта в контексте другого применяется наследование (производный класс способен тесно взаимодействовать с членами базового класса), но в C++ не существует аналога «классового» наследования на уровне экземпляров. Если бы такой аналог существовал, язык имел бы прямую поддержку делегирования.

Делегирование обладает большей гибкостью на стадии выполнения, чем наследование. В C++ для имитации делегирования применяется перенаправление: объект одного класса передает большую часть своей работы объекту другого класса. Такой подход обеспечивает гибкость делегирования, хотя при нем теряется общность контекста, присутствующая в языках с «полноценным» делегированием. Чтобы класс Menu мог многократно использовать класс Window, достаточно включить в него объект Window. Многие функции Menu (move, refresh) напрямую реализуются в контексте функций Window. В простых случаях это решение хорошо работает в C++ и считается традиционным механизмом поддержки многократного использования без нарушения абстракции. Но если фактический класс потомка Window неизвестен во время компиляции (то есть если мы собираемся задействовать класс, производный от Window, размер и конкретные свойства которого могут изменяться), начинаются проблемы. Обычно они решаются сохранением указателя на нужный объект (например, включением *указателя* на Window в объект Menu) вместо экземпляра.

Третий механизм многократного использования кода — закрытое наследование. При закрытом наследовании производный класс применяет код базового класса в своей реализации без включения интерфейса базового класса в свой интерфейс.

С семантической точки зрения не существует принципиальных различий между закрытым наследованием и включением экземпляра в виде переменной другого класса. Закрытое наследование может потребоваться как условная запись, свидетельствующая о том, что проектировщик стремится именно к многократному использованию (в отличие от включения объекта только как характеристики внутреннего состояния). Пример:

```
class Resistor { ... };
class Inductor { ... };
class Capacitor { ... };
// Фильтр нижних частот использует (многократно) готовые
// абстракции Resistor, Inductor и Capacitor
// в своем внутреннем устройстве; этот факт
// отражается посредством закрытого наследования.
class LowPassFilter: Resistor, Inductor, Capacitor {
public:
    LowPassFilter(Resistor r, Inductor i, Capacitor c):
        Resistor(r), Inductor(l), Capacitor(c) { }
    ...
private:
    // Закрытые данные не отражают стремления к
    // многократному использованию, а лишь характеризуют
    // состояние объектов класса.
    Frequency w0;
    Quality q;
    double High3dbPoint, Low3dbPoint;
};
```

Впрочем, такое решение обладает жесткими ограничениями. Следующий фрагмент недопустим, потому что класс не может быть дважды указан в качестве базового:

```
class Arm { ... };
class Leg { ... };
// Недопустимо:
class Robot: Arm, Arm, Leg, Leg { // Наследование для многократного
private:                          // использования
    Direction direction;          // Переменные состояния
    Velocity  velocity;
public:
    Robot();
};
```

Вероятно, последний механизм многократного использования — *параметризация* (параметрический полиморфизм) — обладает наибольшим потенциалом из всех перечисленных. В следующем разделе описывается поддержка параметризации в C++, а также рассматриваются программные идиомы для сокращения избыточного кода, генерируемого для параметризованных типов C++.

7.4. Параметризованные типы, или шаблоны

Шаблон (параметризованный тип) C++ определяет семейство типов и функций. Шаблоны являются альтернативой для плоских иерархий наследования, создаваемых с целью многократного использования кода. Допустим, мы хотим применить алгоритмы реализации Stack к стекам с элементами любых типов (int, Window и т. д.). Если решать эту задачу путем наследования, весь общий код следует поместить в базовый класс и создать отдельный производный класс для каждого создаваемого типа (intStack, WindowStack и т. д.). Но так как код Stack работает с элементами стека, было бы естественно выразить эти алгоритмы в контексте типов этих элементов. А следовательно, большая часть кода не будет общей для всех типов Stack, а должна генерироваться для каждого класса отдельно. Класс Stack даже не сможет быть абстрактным базовым классом, потому что объявления функций pop и top зависят от типа стека.

Другое возможное решение — реализация класса Stack для нетипизированных элементов void*:

```
class Stack {
private:
    struct Cell {
        Cell *next;
        void *rep;
        Cell(Cell *c, void *r) next(c), rep(r) { }
    } *rep;
public:
    void *pop() {
        void *ret = rep->rep;
        Cell *c = rep;
        rep = rep->next;
        delete c;
        return ret; }
    void *top() { return rep->rep; }
    void push(void *v) { rep = new Cell(rep, v); }
    int empty() { return rep != 0; }
    Stack() { rep = 0; }
};
```

Но подобная реализация Stack создает синтаксические неудобства — поскольку в ней хранятся только указатели void*, код приложения содержит множество лишних символов & и преобразований типов. Кроме того, теряется большая часть преимуществ проверки типов. Еще более серьезные проблемы возникают из-за ослабления типизации. Допустим, в класс List добавляется функция sort, реализованная с применением указателя void*. Функции List::sort не удастся сколько-нибудь удобно сравнить два элемента, указатели на которые хранятся в списке; она не располагает информацией о том, на какие типы они ссылаются.

Высокоуровневые объектно-ориентированные языки вроде Smalltalk и CLOS решают эту проблему по-своему: большинство решений, связанных с типами, принимается на стадии выполнения. Их полиморфизм в меньшей степени зависит от иерархии наследования, чем полиморфизм C++. К достоинствам такого подхода следует отнести ослабление связи между классами, а к недостаткам — возможность ошибок отсутствия метода на стадии выполнения.

В C++ описанные проблемы решаются без потерь и в плане безопасности типов, и в плане эффективности выполнения. Все, что требуется сделать — параметризовать класс Stack с другим типом. Реализация стека, приведенная в листинге 7.1, представляет собой шаблон, на основе которого генерируются специализированные разновидности стека.

Листинг 7.1. Шаблон Stack

```
template <class T> class Stack;

template <class T> class Cell {
friend class Stack<T>;
private:
    Cell *next;
    T *rep;
    Cell(T *r, Cell<T> *c): rep(r), next(c) { }
};

template <class T> class Stack {
public:
    T *pop();
    T *top() { return rep->rep; }
    void push(T *v) { rep = new Cell<T>(v, rep); }
    int empty() { return rep == 0; }
    Stack() { rep = 0; }
private:
    Cell<T> *rep;
};

template <class T> T *Stack<T>::pop() {
    T *ret = rep->rep;
    Cell<T> *c = rep;
    rep = rep->next;
    delete c;
    return ret;
}
```

Шаблон определяется для некоторого обобщенного типа T, который, как предполагается, обладает всеми свойствами, обязательными для элементов Stack. Когда программа требует создать объект Stack с элементами конкретного типа, компилятор генерирует его и подставляет характеристики фактического типа на место

параметрического типа T в теле шаблона. Объект Stack с элементами int создается на базе шаблона Stack с передачей параметра int:

```
Stack<int> anIntegerStack;
```

Теперь в программе можно использовать конструкции вида:

```
Stack<Window> windowManagerStack;

int main() {
    Stack<int> anIntStack;
    anIntStack.push(5);
    int i = anIntStack.pop();
    Window w(0,50,4,4,4);
    windowManagerStack.push(w);
    ...
}
```

При применении шаблонов нужно придерживаться определенных правил.

- Параметризованный класс не может вкладываться в другой параметризованный класс.
- Статические переменные параметризованного класса принадлежат конкретной специализации шаблона.

Параметризация применима не только к классам, но и к функциям. Самым распространенным примером является функция sort, способная отсортировать вектор произвольного типа (листинг 7.2).

Листинг 7.2. Параметризованная версия функции sort

```
template <class S>
void sort(S elements[], const int nelements) {
    int flip = 0, sz = nelements - 1;
    do {
        for (int j = 0, flip = 0; j < sz; j++) {
            if (elements[j] < elements[j+1]) {
                S t = elements[j+1];
                elements[j+1] = elements[j];
                elements[j] = t;
                flip++;
            }
        }
    } while (flip);
}

int main() {
    Complex cvec[12];
    for (int i = 0; i < 12; i++) cin >> cvec[i];
    sort(cvec, 12);    // calls sort(Complex[], const int)
    for (i = 0; i < 12; i++) cout << cvec[i] << endl;
    return 0;
}
```

Шаблоны занимают центральное место среди механизмов многократного использования кода в C++. Чаще всего они применяются при построении библиотек. Тем не менее, необходимо хорошо понимать, что параметризация в C++ базируется на многократном использовании *исходных текстов*, а не объектного кода. С другой стороны, большая часть функциональности базового класса может быть задействована производным классом на уровне объектного кода, что отражается на общей компактности кода и последствиях от внесения изменений.

Если достаточно большая часть операций шаблона не зависит от аргумента-типа, то общий код может быть преобразован в контекст void* и использован как основа для других типов, выполняющих преобразование по мере необходимости. Рассмотрим обобщенный шаблон List, на базе которого требуется создать несколько специализированных списковых классов. Если шаблон List содержит функцию sort, то скорее всего, для каждой специализации List будет генерироваться код соответствующей версии функции sort независимо от того, вызывается она в программе или нет. Допустим, мы не собираемся сортировать списки, поэтому сохранять «балласт» в виде нескольких копий sort было бы нежелательно. В этом случае можно поступить так:

```
class WindowList: private List<void*> {
public:
    Window *get() { return List<void*>::get(); }
    void add(Window *p) { List<void*>::add(p); }
    // Функция sort остается закрытой
};

class AddressList: private List<void*> {
public:
    Address *get() { return (Address*)List<void*>::get(); }
    void add(Address *p) { List<void*>::add(p); }
    // Функция sort остается закрытой
};
```

Реализация довольно уныла, но, по крайней мере, она не требует переделки функций get и add для каждой новой абстракции, а количество копий List<T>::sort уменьшается до одной.

Описанная методика почти ничего не дает, если большинство функций шаблона (или те функции, которые мы намерены использовать) зависит от параметра-типа — как функция sort для сравнения двух элементов списка. Параметризация на уровне исходных текстов также создает интересные проблемы для библиотечных классов, параметризуемых *другими* библиотечными классами. Предположим, класс Set был реализован с применением класса List (листинг 7.3). Теперь для создания нового типа на базе шаблона Set необходимы исходные тексты как класса Set, так и класса List. Если List в свою очередь зависит от другого параметризованного типа, понадобится его исходный текст, и т. д.

Листинг 7.3. Параметризованный класс Set, реализованный с помощью класса List

```
template <class ElementType> class Set {
public:
    void add(const ElementType& e) {
        if (!rep.element(e)) rep.add(e);
    }
    ElementType get() {
        return rep.get;
    }
    void remove(const ElementType&);
    int exists(const ElementType& e) const {
        return rep.element(e);
    }
    Set<ElementType> Union(const Set<ElementType>&) const;
    Set<ElementType> Intersection(const Set<ElementType>&)
        const;
    int size() const { return rep.size(); }
    void sort() { rep.sort(); }
private:
    List<ElementType> rep;
};
```

Применение классов List в Set должно быть полностью прозрачным для пользователя. Требование исходных текстов класса List, не входящего в открытый интерфейс класса Set, противоречит здравому смыслу. Данное обстоятельство преподносит сюрпризы при сопровождении, о нем приходится особо упоминать в пользовательской документации. Похоже, это нарушает принцип сокрытия информации.

Другая проблема состоит в том, что изрядная часть кода List дублируется в каждой сгенерированной специализации Set. Различия между специализациями List могут быть небольшими, но код класса List, не нужный классу Set, все равно генерируется. Данное обстоятельство затрудняет упаковку программы: оно означает, что исходные тексты Set не могут продаваться без исходных текстов List. Скорее всего, в этом конкретном примере они в любом случае должны продаваться вместе. Но в общем случае нельзя заставлять клиента приобретать исходные тексты обоих типов только потому, что один тип использует другой тип в своей реализации.

Чтобы в полной мере использовать возможности параметризации без затрат на дублирование кода в разных версиях List, следует применять *идиому косвенного шаблона.*

В листинге 7.4 представлен файл List.h с определением простого параметризованного класса List, требуемого для реализации нового класса Set. Вспомогательный класс ListItem представляет элемент, включаемый в список, и указатель на начало связанного списка. Во внутреннем представлении класса List используется односвязный список элементов.

Листинг 7.4. Файл List.h

```
template <class T> class List;

template <class T> class ListItem {
friend class List<T>;
private:
    ListItem<T> *next;
    const T item;
    ListItem(const ListItem<T> *n, const T &i):
        item(i), next((ListItem<T>*)n) { }
};

template <class T> class List {
public:
    void sort();           // Сортировка списка "на месте"
    void put(const T& t) { head = new ListItem<T>(head, t); }
    T get() {              // Выборка из начала списка
        T retval = head->item;
        ListItem<T> *temp = head;
        head = head->next;
        delete temp;
        return retval;
    }
    int element(const T&) const; // Проверка принадлежности
    ...                          // Другие операции
    int size() const;            // Количество элементов
    List(): head(0); { }         // Конструктор по умолчанию
private:
    ListItem<T> *head;           // Начало списка
};
```

В отдельном файле **Set.h** объявляется шаблон **Set**, использующий класс **List** в своей реализации. Чтобы избежать специализации класса **List** для каждой параметризации **Set**, мы добавляем дополнительный уровень сокрытия информации в двух местах. Во-первых, каждая конкретная разновидность **Set** является производной от класса **SetBase**. Интерфейс **SetBase** содержит все операции, которые, как ожидает объект **Set**, поддерживаются его *элементами*:

```
class SetBase {
friend ErsatzListElement;
private:
    virtual int compareIt(const void*, const void*)
        const = 0;
    virtual int compareeq(const void*, const void*)
        const = 0;
};
```

Операции объявлены виртуальными и отделены от типа объектов, содержащихся в множестве: все взаимодействие с внешним миром осуществляется через тип

void*. В конечном счете эти операции определяются в «настоящем» классе, созданном специализацией шаблона Set, в контексте типа элементов Set.

Вторым в ход идет другой фокус — инкапсуляция элементов списка в другом классе ErsatzListElement, который обращается к ним только через указатели void*:

```
class ErsatzListElement {
    // Класс не параметризуется: это означает,
    // что одна специализация шаблона List обслуживает
    // все специализации Set.
public:
    void *rep;
    int operator< (const ErsatzListElement& l) const {
        return theSet->compareLt(this,&l);
    }
    int operator==(const ErsatzListElement& l) const {
        return theSet->compareEq(this,&l);
    }
    ErsatzListElement(const void* v = 0): rep(v) { }
    ErsatzListElement(const SetBase *s, void *v=0)
        : theSet(s), rep(v) { }
private:
    const SetBase *theSet;  // Требуется только для вызова
};                          // виртуальных функций сравнения
                            // двух объектов и т. д.
```

Экземпляры этого класса-оболочки поддерживаются классом List, необходимым для реализации Set. Это означает, что *все* специализации List, создаваемые для Set, будут относиться к одному типу List<ErsatzListElement>, а следовательно, один тип списка будет совместно использоваться всеми специализациями Set.

В завершение рассмотрим сам шаблон Set (листинг 7.5). Он параметризуется по типу содержащихся в нем объектов, но не создает новых типов, кроме List <ErsatzListElement>. Шаблон содержит закрытые функции для сравнения двух объектов класса T и операции, которые должны быть определены для T по настоянию Set. Эти «заготовки» функций перенаправляют свои запросы T только после преобразования параметров void* к типу T. Такое преобразование безопасно: в границах параметризованного типа можно гарантировать, что void* *всегда* указывает на объект типа T. И конечно, весь код — заготовки и все прочее — генерируется автоматически компилятором на базе шаблона благодаря поддержке параметризованных типов.

Листинг 7.5. Гибкая и эффективная версия Set

```
template <class T> class Set: private SetBase {
    // Закрытое наследование для многократного использования кода
public:
    void add(T t2) {
        if (!exists(t2)) {
            ErsatzListElement t(this, new T(t2)); rep.put(t);
```

```
        }
    }
    T get() {
        ErsatzListElement l=rep.get(); return *((T*)(l.rep));
    }
    void remove(const T&);
    int exists(const T& e) {
        ErsatzListElement t(this, (T*)&e);
        return rep.element(t);
    }
    Set<T> Union(const Set<T>&) const;
    Set<T> Intersection(const Set<T>&) const;
    int size() const { return rep.size(); }
    void sort() { rep.sort(); }
    Set();
private:
    List<ErsatzListElement> rep;
    int compareIt(const void* v1, const void* v2) const {
        const T* t1 = (const T*) v1;
        const T* t2 = (const T*) v2;
        return *t1<*t2;
    }
    int compareeq(const void* v1, const void* v2) const {
        const T* t1 = (const T*) v1;
        const T* t2 = (const T*) v2;
        return *t1==*t2;
    }
};

template <class T> Set<T>::Set() { }
```

На этом завершается рассмотрение класса **Set**. Далее приведено простое приложение, использующее параметризованный тип **Set**, который в свою очередь использует параметризованный тип **List**. Хотя в программе встречаются две разновидности **Set**, для поддержки обеих создается только одна специализация **List**:

```
int main() {
    Set<int> foo;
    Set<double> bar;
    foo.add(1);
    foo.add(2);
    cout << foo.get() << "\n";
    cout << foo.get() << "\n";
    bar.add(3.0);
    bar.add(4.0);
    cout << bar.get() << "\n";
    cout << bar.get() << "\n";
}
```

Теперь абстракция Set приобрела самостоятельность; ее можно сопровождать (а также продавать) как отдельный продукт. Вместе с исходными текстами Set достаточно поставлять объектный код абстракций List и List<ErsatzListElement>.

ПРИМЕЧАНИЕ

Применяйте эту идиому в том случае, если ваша программа интенсивно работает с шаблонами, и особенно — если один из параметров-типов параметризованного класса создает специализацию другого параметризованного класса для внутреннего использования.

Обеспечение однократной специализации List может выполняться разными способами в зависимости от установки C++. Как правило, для этого определения функций List помещаются в отдельный файл с расширением .c.

7.5. Закрытое наследование и многократное использование

В 6.5 был приведен пример ошибочного наследования, в котором класс Queue объявлялся производным от CircularList. Это вело к тому, что в сигнатуру Queue вошли лишние операции, унаследованные от базового класса. Такой результат объяснялся открытым наследованием — иначе говоря, ключевое слово public заявляло: «я, производный класс Queue, умею выполнять все операции, поддерживаемые базовым классом, и везде, где ожидается объект CircularList, можно использовать объект моего класса». Но для Queue такие претензии слишком завышены.

Чтобы решить возникшую проблему, достаточно понять, что Queue всего лишь опирается на функциональность CircularList, не претендуя на полную взаимозаменяемость с этим классом. Многократное использование не должно нарушать абстракцию ни CircularList, ни Queue. Существуют, по крайней мере, три возможных пути.

- Включить объект CircularList в каждый объект Queue (иерархия реализации).
- Организовать делегирование некоторых операций Queue соответствующему объекту CircularList.
- Использовать закрытое наследование (как рекомендуется в [3]).

Первое решение, за которое выступает Лисков, не нарушает абстракции данных; кроме того, эта модель многократного использования наиболее интуитивно понятна. Название решения — *иерархия реализации* — означает, что в иерархических отношениях находятся *экземпляры* многократно используемых классов, а уточнение позволяет отличить ее от иерархии классов при наследовании. Иерархия реализации представляет собой простое *включение* объекта одного класса в другой класс. В отличие от наследования, применение иерархии реализации в целях многократного использования не предполагает отношений субтипизации; оно просто утверждает, что мы хотим задействовать функциональность CircularList,

не претендуя на взаимозаменяемость с этим объектом. Пример применения иерархии реализации:

```
class CircularList {
public:
    int empty();
    CircularList();
    void push(int);    // Занесение с начала
    int pop();
    void enter(int);   // Занесение с конца
private:
    cell *rear;
};

class Queue {
public:
    Queue() { }
    void enterq(int x)    { l.enter(x); }
    int leaveq()          { return l.pop(); }
    long empty() { return l.empty(); }
private:
    CircularList l;
};
```

Перед нами разновидность перенаправления, подчеркивающая, что, во-первых, **Queue** содержит («HAS-A») список, во-вторых, отношения «IS-A» не действуют, в-третьих, объект **CircularList** не используется совместно с другими объектами (как при обращении к нему через указатель). Инкапсуляция **CircularList** при этом не нарушается.

Второе решение — делегирование — возможно, но на C++ оно выглядит неестественно из-за отсутствия поддержки на языковом уровне (см. с. 254).

Третье решение — закрытое наследование — может использоваться так:

```
class Queue: private CircularList {
public:
        Queue() { }
    void enterq(int x)    { enter(x); }
    int  leaveq()         { return pop(); }
        CircularList::empty;
};
```

С точки зрения эффективности закрытое наследование и иерархия реализации более или менее эквивалентны. У каждого подхода есть свои преимущества. Если производный класс может напрямую применять функции базового класса, при наследовании программисту приходится писать меньший объем кода, чем при включении.

Производный класс может подменить виртуальные функции, вызываемые функциями закрытого базового класса, поэтому для двух классов, взаимодействующих

с целью многократного использования кода, закрытое наследование подходит лучше включения. Базовый класс определяет общую структуру и характеризует обязанности производного класса в виде чисто виртуальных функций. Базовый класс зависит от реализации этих чисто виртуальных функций в производном классе, поскольку производный класс может вызывать функции базового класса. Хорошим примером служит библиотека TaxForm в главе 11.

Однако многократное использование кода посредством наследования обладает также недостатками. Клиентский класс может обращаться к защищенным членам закрытого базового класса, нарушая его инкапсуляцию; при включении инкапсуляция сохраняется. Кроме того, наследование может внушить программисту ложное предположение об отношениях субтипизации.

Также возможно гибридное решение, объединяющее закрытое наследование с перенаправлением:

```
class Queue: private CircularList {
public:
    Queue() { }
    void enterq(int x) { enter(x); }
    int leaveq()       { return pop(); }
    int empty()        { return CircularList::empty(); }
};
```

Некоторым это решение кажется более «стилистически чистым», чем другие варианты, однако оно требует дополнительных затрат на вызов функции **empty** (если только тело **Queue::empty** не расширяется посредством подстановки). Используйте эту конструкцию вместо решения со спецификаторами доступа, если имеются перегруженные функции, и программист желает предоставить разный уровень доступа к одноименным функциям базового класса.

Большинство компиляторов C++ сообщают о типичных ошибках, связанных с применением закрытого наследования для многократного использования (вместо субтипизации). Это может быть важно с точки зрения архитектуры. Вернемся к предыдущему примеру с **CircularList**, дополненному виртуальными функциями:

```
class CircularList {
public:
    virtual int empty();
              CircularList();
    virtual void push(int);    // Занесение с начала
    virtual int pop();
    virtual void enter(int);   // Занесение с конца
private:
    cell *rear;
};

class Queue: private CircularList {
public:
        Queue() { }
```

```
    void enterq(int x) { enter(x); }
    int leaveq()       { return pop(); }
        CircularList::empty();
};

int main() {
    CircularList *alist = new Queue;  // Ошибка компиляции
    return 0;
}
```

Если бы это было возможно, то объекты Queue могли бы находиться в любом контексте, в котором ожидается CircularList. Тем не менее, компилятор выдает ошибку при попытке создания экземпляра — он знает, что операции CircularList не могут перепоручаться Queue.

Эта проблема возникает только при использовании виртуальных функций в контексте указателей или ссылок на объекты классов.

7.6. Многократное использование памяти

Иногда требуется заимствовать поведение существующего класса другими классами или объектами, но без затрат на хранение лишних копий этого класса. Допустим, несколько объектов Window желают совместно использовать один объект Keyboard для различных функций, связанных с вводом. Вместо того чтобы встраивать отдельный объект Keyboard в каждый объект Window, все объекты Window могут перенаправлять свои операции одному общему классу Keyboard. В результате каждый объект Window наделяется поведением Keyboard, но все объекты совместно используют общую память. Для отражения факта совместного использования можно применить идиому «манипулятор/тело» (см. главу 5): тело применяется совместно, а манипуляторы выступают в роли «посыльных» или «суррогатов» этого объекта. В некоторых случаях оптимальный способ совместного использования памяти между всеми объектами класса основан на статических переменных класса.

Стремление к совместному использованию памяти также может быть обусловлено более глубокими архитектурными соображениями, например, размещением объекта в блоке памяти, общем для нескольких процессоров. В C++ это делается напрямую, *при условии, что класс объекта не содержит указателей.* Переменные-указатели класса необходимо либо удалить, либо организовать для них специальную обработку. Указатели иногда также встречаются в замаскированном виде: они могут быть задействованы в реализациях ссылочных переменных класса и виртуальных функций. Содержимое указателя должно правильно интерпретироваться в каждом контексте, в котором он используется; обычно это условие выполняется только в том случае, если указатель ссылается на другой объект в общей памяти. Общие указатели создают меньше проблем, если каждый процессор (или процесс) отображает сегмент общей памяти на

один и тот же логический адрес в своем адресном пространстве, но во многих операционных системах считается нежелательным, чтобы работа программы зависела от таких отображений.

Исключив из общих данных указатели и выделив всю семантику указателей и виртуальных функций в класс-конверт, можно организовать нормальную работу с общей памятью в приложениях C++. Классы, спроектированные в расчете на относительные смещения вместо указателей и не содержащие виртуальных функций, тоже могут использоваться в сегментах общей памяти.

7.7. Многократное использование интерфейса

Гибко спроектированная система может содержать несколько разных реализаций одной архитектуры. Каждая реализация работает с тем, что считается одним и тем же типом, хотя внутренние алгоритмы изменяются в зависимости от контекста. Рассмотрим ряд примеров.

- Удаленный вызов процедур: функции класса-«заместителя» перенаправляют запросы по сети удаленному классу, который выполняет фактическую работу. Удаленный и локальный классы обладают одинаковыми сигнатурами, но их реализации имеют мало общего: локальный класс только передает задания удаленному классу. На высоком уровне семантика их операций идентична.

  В сетевых приложениях класс может существовать в нескольких разновидностях.

  - Реализация для *локального* выполнения работы (то есть класс локального сервера). Например, если объект файлового сервера знает, что диск подключен к локальному контейнеру, он выполняет свою работу напрямую.
  - Реализация для *перенаправления* запросов удаленному компьютеру. Если локальный объект файлового сервера знает, что диск подключен к другому компьютеру, он перенаправляет запрос по сети на другой компьютер. Там этот запрос обрабатывается классом удаленного сервера.
  - Класс *удаленного* сервера. На логическом уровне класс удаленного сервера схож с классом локального сервера (и может содержать внедренный объект локального сервера), но свои запросы он получает по сети. Впрочем, его нельзя рассматривать как полноценную разновидность двух первых классов, поскольку он напрямую взаимодействует с сетевыми протоколами, а не с пользователем.
- Рыночные зависимости. Может потребоваться, чтобы класс имел разные реализации для разных контекстов. Например, в программе может быть один класс `Console` с разными реализациями для английского и для арабского языков.

- Аппаратные зависимости. Класс Stack может быть написан в расчете на непосредственную аппаратную поддержку стека на одном компьютере, и на программную эмуляцию для других платформ. Если класс находится в библиотеке, то обе реализации могут иметь идентичный интерфейс класса и использоваться эквивалентно. То же относится и к другим аппаратно-зависимым примитивам (например, операциям с вещественными числами).
- Контроль. Класс может существовать в двух версиях: тестовой, предназначенной для наблюдения за ходом выполнения, и основной, оптимизированной для максимального быстродействия без отладочных фрагментов.
- Оптимизация. Допустим, существует некий конкретный тип данных, не поддерживающий подсчет ссылок; может потребоваться его аналог с подсчетом ссылок для компонентов системы, критичных по затратам памяти и скорости выполнения. Пример (реализованный на базе простой абстракции данных, но который также может быть реализован с применением наследования) приведен в разделе 3.5 (см. с. 81).

У всех приведенных примеров имеется одна общая черта: разновидности классов слабо связаны с сущностями приложения, они в большей степени соответствуют специфике реализации. Тем не менее наследование является удобным механизмом реализации, особенно при наличии абстрактных базовых классов. Мы многократно используем одну архитектуру и отражаем ее в нескольких разных реализациях.

Такие производные классы называются *вариантами*. Имена производных классов несущественны, а ссылки на них локализованы. Обращения к объектам этих классов производятся через указатели и ссылки с использованием виртуальных функций. Клиентский код пишется в контексте абстрактного базового класса.

Другое возможное решение — обработка альтернатив *внутри* родительского класса (вместо создания на его базе производных классов). Такое решение может рассматриваться как форма инкапсулированного многократного использования; в этом контексте базовый класс называется *прототипом* (к сожалению, этим термином также обозначается другая, хотя и похожая идиома, описанная в главе 8). Получая запрос на выполнение некоторой операции, класс-прототип может перенаправить запрос одному из своих объектов, который обрабатывает его с помощью виртуальных функций. Преимущество прототипов перед наследованием состоит в инкапсуляции имен реализаций. Например, упоминавшийся ранее класс Stack может специализироваться одним из двух способов: в одном содержать объект, реализующий стек с применением особых аппаратных средств процессора, а в другом обрабатывать структуры данных обычным образом.

Прототипы могут быть реализованы по тем же принципам, что и делегированные полиморфные классы (см. раздел 5.5). Иначе говоря, внутренний объект не обязан физически находиться внутри байтового представления внешнего объекта; его логическая принадлежность может моделироваться хранением во внешнем объекте указателя на вариант с нужной семантикой.

7.8. Многократное использование, наследование и перенаправление

Проектировщики и программисты должны найти компромисс между логической полнотой и эффективностью многократного использования с одной стороны, и инкапсуляцией архитектуры независимых классов с другой. Проектировщики, работающие в более гибкой среде, чем код C++, могут различными способами распределения функциональности и данных выдерживать баланс между возможностями многократного использования и чистотой архитектуры. После того как архитектура воплощена в программном коде, языковые ограничения создают противоречие между самодостаточностью класса и возможностью других классов использовать его код.

Связь между двумя классами может определить способ реализации многократного использования: закрытое наследование или внедрение экземпляра одного класса в другой. Наследование приводит к частичному нарушению инкапсуляции базового класса, поскольку производный класс получает доступ к защищенным членам. Попытки организовать совместное использование символических имен без закрытого наследования (например, с применением друзей) оставляет альтернативу стопроцентной видимости одного класса для другого класса. Совместное использование символических имен без применения дружественных отношений и наследования приводит к разрастанию открытого интерфейса класса. В [4] написано:

> Самый серьезный недостаток этого решения [имеется в виду применение подставляемых функций для эффективного обращения к переменным базового класса] состоит в том, что оно требует взаимодействия с проектировщиком класса-предка (а также проектировщиками всех промежуточных предков), поскольку доступ к унаследованной переменной-экземпляру возможен лишь при наличии соответствующих операций. Если вам (как проектировщику класса) потребуется доступ к унаследованной переменной-экземпляру, а соответствующие операции не определены, следует обсудить с проектировщиком класса-предка возможность включения поддержки этих операций. (Среда программирования может дать возможность обойти это ограничение, например, позволить вам определить эти операции временно. Но если вы намерены серьезно использовать чужой код, взаимодействие с проектировщиком абсолютно необходимо.)

Здесь важно то, что речь идет не о простом «заимствовании» кода одного класса другим классом, а о распространении классом информации о возможности многократного использования среди заинтересованных сторон. Взаимодействие между «поставщиком» и «пользователем» кода, потенциально рассчитанного на многократное использование, отвечает интересам всех сторон для сохранения хорошей архитектуры с одновременным снижением объема кода и дублирования усилий.

7.9. Архитектурные альтернативы для многократного использования исходных текстов

Существует множество способов управления многократным использованием исходных текстов или адаптации существующего кода для данного приложения. Наследование является лишь одним из средств. В зависимости от ситуации может применяться как наследование, так и более традиционные методы. Далее охарактеризованы плюсы и минусы разных решений.

Простейшим среди всех решений является *условный выбор на стадии выполнения*. Под этим хитроумным термином скрывается простой набор команд if и других условных конструкций. Проверка логического флага и выбор одной из двух ветвей программы хорошо подходит для небольших ветвей, часто применяемых, но незначительных с архитектурной точки зрения (то есть не обладающих существенным интерфейсом на уровне системной архитектуры). Организационной единицей в этом случае является команда С. Например, решение об активизации отладочного кода может приниматься в зависимости от состояния глобального флага.

Другое решение — *условная компиляция* — хорошо знакомо программистам С под видом директив #ifdef и других директив этого семейства. Условная компиляция хорошо подходит для изолированных небольших фрагментов кода, и только в тех ситуациях, когда решение о присутствии кода в программе может быть принято на стадии компиляции. Это решение приносит максимальную пользу тогда, когда альтернативы связаны со значительными различиями в семантике или в синтаксисе, например, при выборе длины слова в регистрах ввода-вывода в зависимости от типа целевого процессора. Организационной единицей в данном случае можно считать функцию, хотя несколько функций могут быть упакованы в один исходный файл для удобства сопровождения и настройки. Условная компиляция также может потребоваться для активизации отладочного кода — с некоторой потерей гибкости, но с меньшим количеством лишних команд в окончательной версии программы (по сравнению с условным выбором).

Экстремальное (хотя и достаточно распространенное) решение заключается в получении *отдельных копий исходного текста*. Программист копирует исходный текст существующего класса, вносит исправления и добивается его работоспособности. После этого две копии продолжают самостоятельное существование. Чтобы два класса могли сосуществовать в одной программе, один класс необходимо переименовать, иначе компилятор не сможет их различить. Даже после внесения изменений классы могут иметь много общего: интерфейс класса в целом сохраняется, а внутренние структуры данных и даже весь набор функций класса могут вообще не измениться. Однако такой подход имеет существенные недостатки: если автор оригинала найдет и исправит в исходном тексте какие-нибудь ошибки, в копии эти ошибки останутся. Даже если вы будете оповещены

о них, вам придется вносить изменения вручную. Это приведет к напрасной трате времени, особенно если автор изменил те функции, к которым вы не прикасались. Конечно, модификация создает риск внесения новых ошибок. Другой недостаток — излишнее разрастание программного кода. А если кто-то захочет «многократно использовать» вашу программу в той же системе, ситуация только усугубится.

Как отмечалось выше, если вам понадобится новый класс, по функциональности сходный с существующим классом, но отличающийся от него реализацией, задачу лучше решать путем наследования. Наследование позволяет создать новый класс на базе существующего класса, изменить то, что требуется изменить, и *совместно использовать* оставшуюся часть класса.

Впрочем, у наследования как механизма многократного использования тоже есть недостатки. Допустим, имеется группа программ, построенная на базе объектов, а для адаптации поведения базовых классов к конкретным условиям (разнообразному составу оборудования и операционным системам) применено наследование. Продукт должен работать на новой платформе с минимальными изменениями в коде.

Для примера возьмем сеансовый интерфейс, приведенный в листинге 7.6. Класс SesInterface представляет функциональность клавиатуры и окна в некотором интерактивном приложении. Он содержит функцию SesInterface::saveTty(SGTTY*), задающую характеристики устройства для данного интерфейса. Функция вызывает примитивы той платформы, на которой она работает. Вызываемые примитивы также изменяются в зависимости от операционной системы и платформы; в данном случае расхождения проявляются на уровне исходного текста. Так, для USG-систем вызывается функция ioctl с параметром TCGETA; для систем Sun предусмотрена особая обработка, а все прочие системы используют функцию gtty.

Листинг 7.6. Программа, предназначенная для разных платформ (добавляемый код выделен курсивом)

```
int SesInterface::saveTty(SGTTY *tty) {
#ifndef USG
#       if sun
                return ioctl(2,TIOCGETP,tty);
#       else
                return gtty(2,tty);
#       endif
#else
        return ioctl(2, TCGETA, tty);
#endif
}
```

Другое возможное решение основано на структуре наследования, изображенной на рис. 7.1. Добавление нового кода всего лишь сводится к включению нового класса в соответствующую позицию дерева наследования; новый класс наследует все атрибуты своих родителей, кроме одной изменяемой функции.

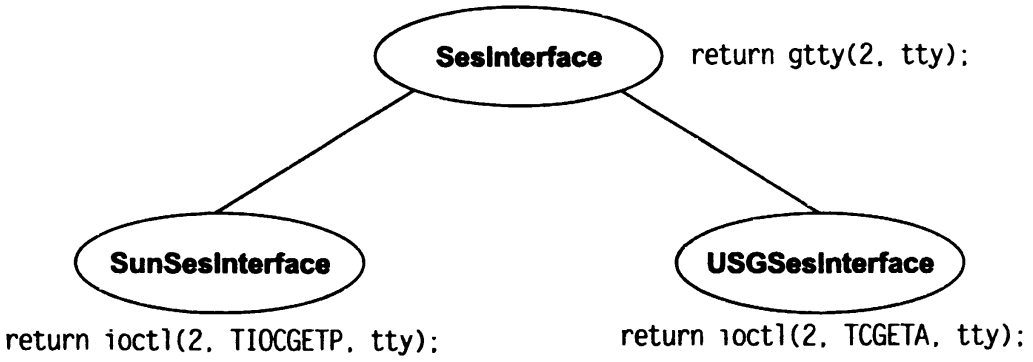


Рис. 7.1. Наследование как альтернатива условной проверке

При таком подходе модификация на уровне исходных текстов отсутствует. Может показаться, что его преимуществом является возможность выбора (соответствующих файлов с объектным кодом) на стадии компоновки вместо стадии компиляции. Тем не менее, здесь кроется ловушка. Допустим, где-то в системе скрывается код вида:

```
void User::code() {
    SesInterface *si = new SesInterface(keyboard, window);
    ...
}
```

Невнимательный проектировщик может не осознать, что при добавлении производного класса изменения должны быть внесены во весь код, содержащий ссылки на команды создания экземпляров базового класса. Без замены базового класса производным при создании экземпляров изменения будут проигнорированы. Так, мы можем взять приведенный выше код из User и заменить его следующим:

```
void User::code() {
#ifndef USG
#       if sun
           si = new SunSesInterface(keyboard, window);
#       else
           si = new SesInterface(keyboard, window);
#       endif
#else
           si = new USGSesInterface(keyboard, window);
#endif
    ...
}
```

Однако так мы снова возвращаемся к модификации исходных текстов, чего мы пытались избежать с помощью производных классов. Проблема решается созданием трех новых классов, производных от User:

```
class USGUser: public User {
public:
    void code() {
        si = new USGSesInterface(keyboard, window);
        ...
```

```
    }
};

class NonUSGUser: public USGUser {
public:
    void code() {
        si = new SesInterface(keyboard, window);
        ...
    }
};

class SunUser: public NonUSGUser {
public:
    void code() {
        si = new SesInterface(keyboard, window);
        ..
    }
};
```

Но тогда нам придется просмотреть весь код, в котором создаются объекты User, и сделать то же самое с классами, содержащими этот код. Таким образом, изменения лавинообразно распространяются по системе.

Как правило, в подобных случаях проще всего продолжать модификацию на уровне исходных текстов. Альтернативное решение — включение в класс дополнительного уровня абстракции (особенно для упрощения дальнейшей эволюции программы в области инициализации объектов класса). Эта идиома рассматривалась в главе 4.

7.10. Общие рекомендации относительно многократного использования кода

Следующие рекомендации, основанные на материале этой главы, помогут вам принять решение о применении наследования как механизма многократного использования кода C++.

- Анализ предметной области играет важную роль для многократного использования классов. Проанализируйте предметную область в поисках «критических точек» параметризации, чтобы обеспечить максимально широкую применимость класса с минимальными усилиями по ее достижению. Классы не должны быть рассчитаны «на все случаи жизни» — от этого они становятся слишком громоздкими, что затрудняет их понимание и использование (в том числе многократное). С другой стороны, классы не должны быть рассчитаны на одно конкретное приложение.
- Вероятно, многократное использование архитектуры, определяемое интерфейсом класса, обладает максимальным потенциалом для повышения производительности работы в больших проектах. Многократное использование интер-

фейса может означать, что вам придется удалять или переписывать части реализации для каждого конкретного случая, поэтому возможностей для многократного использования кода может быть не так уж много. Тем не менее, ничто не мешает многократно использовать код, для которого это возможно, и обеспечить доработку общих частей.

- Настоящая мощь объектной парадигмы в области многократного использования кроется в абстрактных базовых классах. Определяя иерархию классов, представляющих совокупность более или менее разнородных типов в предметной области, вы создаете высокоуровневую абстракцию, которая может применяться для обобщения в широком спектре приложений. Возможности многократного использования абстрактных классов обусловлены потенциальной широтой области их применения по сравнению с одиночной реализацией абстрактных типов данных. Впрочем, это не означает, что абстрактные базовые классы способны в одиночку решить проблему многократного использования; тщательный анализ предметной области и проработка потребностей приложения по-прежнему играют важную роль.
- Если наследование требуется только для многократного использования, применяйте закрытое наследование. Открытое наследование должно отражать отношения субтипизации, но не возможности многократного использования.
- При проектировании с расчетом на многократное использование приходится искать компромисс между созданием множества мелких классов или относительно небольшого числа крупных классов. Слабая детализация, подсказанная конкретным приложением, упрощает применение абстракции в этом приложении, но может усложнить его в других приложениях той же предметной области. Детализированные классы меньше зависят от контекста, что расширяет возможности их многократного использования. В традициях иерархий классов Smalltalk максимальный потенциал многократного использования достигается при наибольшей детализации классов. Тем не менее, сильная детализация полезна лишь при наличии мощных средств просмотра кода в среде программирования. Кроме того, очень детализированные классы в условиях многократного использования обычно не играют главной роли в конкретных приложениях; им отводится вспомогательная роль «массовки».
- Применение аналогов делегирования для имитации поддержки типов на стадии выполнения позволяет обойти многие ограничения компилятора, мешающие многократному использованию. Идиомы, заложенные в основу таких моделей, описаны в главах 6 и 8.

Упражнения

1. Обсудите следующую цитату: «Многократное использование — для тех, кто не хочет думать».
2. Реализуйте пример Stack из раздела 3.5 (см. с. 82) в виде варианта.

3. Составьте список многократно используемых классов, которые должны входить в стандартную библиотеку *любой* системы программирования на C++. Какой процент этих классов должны составлять шаблоны?
4. Проанализируйте зависимости между классами в предыдущем примере (к числу таких зависимостей относятся параметризация одного класса по другому классу, объявление одного класса производным от другого или любой другой способ использования кода одного класса другим классом). Насколько глубока структура зависимости? Хорошо это (для многократного использования) или плохо (по эффективности работы прикладного программиста)?
5. Какие обобщенные функции, не связанные с библиотекой классов из двух предыдущих упражнений, должны входить в каждую систему программирования на C++? Сколько из них должно оформляться в виде шаблонов? Сколько из этих функций потребует других функций библиотеки?
6. Напишите шаблон класса `CountedClassObject<T>`, создающий абстракцию класса с подсчетом ссылок для любого класса `T`, не поддерживающего подсчет ссылок (возьмите за образец класс `CountedStack` из раздела 3.5).

Литература

1. Tracz, Will. «Software Reuse: Emerging Technology». IEEE Computer Society Press, 1988.
2. Tracz, Will. «Software Reuse Myths». Software Engineering Notes (ACM SIGSOFT) 13,1 (January 1988) 17–20.
3. Sethi, Ravi. «Programming Languages», Boston, Mass.: Addison-Wesley, 1989, 238.
4. Snyder, Allan. «Encapsulation and Inheritance in Object-Oriented Programming Languages», SIGPLAN Notices 21,11 (November 1986).

Глава 8
Прототипы

Система контроля типов C++ в основном зависит от конструкций времени компиляции. Статическая проверка типов способствует раннему выявлению ошибок интерфейса и повышает эффективность кода по сравнению с проверкой типов на стадии выполнения (или полном отказе от нее). Тем не менее, именно поддержка типов на стадии выполнении в значительной мере определяет мощь и гибкость объектно-ориентированных языков. Программа, использующая виртуальные функции, частично откладывает проверку типов до стадии выполнения, что обеспечивает взаимозаменяемость объектов разных классов. За эту гибкость приходится расплачиваться некоторой неопределенностью в работе программы до ее непосредственного выполнения. Если вызвать функцию ring для класса, входящего в иерархию Telephone, вы не знаете, *что именно* произойдет, известно лишь то, что во время выполнения будет выбрана некоторая *разумная* операция.

Другие объектно-ориентированные языки (в частности, Smalltalk и семейство объектно-ориентированных языков, интегрированных с Lisp) идут на дополнительные жертвы в проверке типов на стадии компиляции в интересах гибкости во время выполнения. Обычно это приводит к снижению эффективности кода и возможным сюрпризам на стадии выполнения (если у объекта требуют выполнить операцию, которую он не поддерживает). Преимуществами такого подхода является более универсальная гибкость, обеспечиваемая полиморфизмом. Программы C++ с сильной объектной ориентацией обычно нуждаются в расширенной проверке типов на стадии выполнения, не обеспечиваемой одними виртуальными функциями.

Допустим, вы хотите загрузить объект Number из файла на диске. Объект может быть записан на диск в формате класса Complex, BigInteger, Imaginary или любого другого класса, производного от Number, однако программист, работающий с числами на уровне класса Number, этого не знает (а возможно, и знать не хочет). Требуется, чтобы класс Number «воссоздал» экземпляр самого себя по образу на диске. В процессе построения объект должен быть наделен характеристиками фактического типа (Complex, Imaginary и т. д.).

В C++ существуют специальные идиомы, обеспечивающие подобную гибкость. К их числу относится идиома *прототипов*, которой посвящена эта глава. Идиома прототипов также делает возможной эволюцию характеристик типов в процессе выполнения. Например, если объект Number меняет значение с комплексной

величины $(5 - 3i)$ на 2, напрашивается предположение, что его тип должен измениться с Complex на Integer. Сохраняя комплексную природу, объект будет использовать сложные комплексные алгоритмы для получения результатов, которые могут быть получены более простым способом.

В этой главе показано, как большая часть функциональности классов, присущей стадии компиляции, воспроизводится на стадии выполнения при помощи специальных объектов, называемых прототипами. Эти объекты обеспечивают гибкую систему типизации, возможности которой выходят за рамки того, что непосредственно поддерживается в C++. Тем не менее, для более полного понимания идиомы прототипов потребуется небольшой экскурс в область языков программирования.

Один из критериев нечеткой классификации объектно-ориентированных языков программирования основан на том, являются ли в них классы *базовыми* и *однозначными* конструкциями. C++ принадлежит к числу таких языков — классы занимают основополагающее место в языковой модели объектов и их поведения; невозможно говорить об объектах, не упоминая при этом классы. Языки, у которых классы занимают центральное место в объектной модели, называются *классическими*.

В C++ классы и объекты не тождественны. Классы ведут себя как типы (они реализуют абстрактные типы данных), а объекты представляют собой экземпляры, созданные на базе этих (или встроенных) типов посредством объявлений или оператора new. Классы фиксируются на стадии компиляции, а объекты вообще не существуют до стадии выполнения. Новые объекты создаются на базе класса или встроенного типа; невозможно создать новый объект на основе только другого объекта. Программа, написанная на C++, содержит набор классов, обычно организованных в иерархию (в соответствии с отношениями «базовый/производный класс»), и отдельный набор объектов, которые также могут рассматриваться как элементы другой иерархии — иерархии объектов (организованной в соответствии с тем, как одни объекты создают другие). Под *однозначностью* классов понимается то, что класс не является «чем-то еще»: функцией, объектом или командой. Языки, в которых различаются концепции классов и объектов, называются *языками с двойной иерархией*. C++ также входит в эту категорию.

Некоторые языки (такие, как self [1]) относятся к категории языков с единой иерархией. Вместо использования классов как «заготовок» для построения объектов в таких языках некоторые объекты назначаются *прототипами*; другие объекты со сходными свойствами копируются с прототипов по принципу «поле в поле» (этот процесс называется *клонированием*). Объект-клон может проходить дальнейшую модификацию и настройку посредством изменения свойств, обычно ассоциируемых с классами (например, изменение поведения функций) или с экземплярами (изменение значений переменных экземпляра). Объекты-прототипы занимают в языках с единой иерархией место, отведенное классам и типам в языках с двойной иерархией. Гибкость механизма прототипов приносит

пользу при развитии программы, при итеративной разработке, при обновлении систем, работающих в непрерывном режиме, а также в некоторых стилях программирования, для которых важна поддержка типов на стадии выполнения (например, некоторых приложений из области искусственного интеллекта).

В языках с единой иерархией поддерживаются механизмы создания исходных объектов-прототипов «с нуля» — то есть для создания исходного пустого объекта (или находящегося в состоянии по умолчанию) с последующим добавлением в него полей и функций. В C++ такая возможность отсутствует, или, по крайней мере, она не поддерживается во время выполнения. Следовательно, прототипы должны инициализироваться на базе классов, зафиксированных на стадии компиляции. Один класс может использоваться для создания нескольких объектов-прототипов, каждый из которых инициализируется отдельным набором данных для модификации его характеристик. Например, для разбора восьмеричных или десятичных данных может создаваться свой экземпляр класса. Любой экземпляр может использоваться как прототип, а любой прототип — как экземпляр, хотя по общепринятым правилам в программе обычно создается один специализированный прототип, играющий роль «типа» для набора объектов со сходной структурой.

Для читателей, знакомых с принципами функционирования баз данных, можно провести аналогию между конструкциями языка программирования и концепциями базы данных. Классы рассматриваются как компоненты схемы базы данных, то есть определения свойств, отношений, структуры записей и т. д. Схема указывает, каким образом новые записи создаются и заносятся в базу данных, определяя их общий формат и структуру. Схема является аналогом класса, а записи — аналогом объектов. Наличие схемы позволяет легко создать новую запись и включить ее в базу данных; точно так же при необходимости создается новый объект на базе класса. Изменение содержимого записи или объекта не отражается в схеме; так обеспечивается высокая степень гибкости при работе с информацией.

Но что, если изменится сама схема? В мире баз данных это называется *эволюцией схемы*. В разных системах она реализована с разной степенью гибкости. Если схема должна часто изменяться, желательно, чтобы по своей гибкости эволюция схемы не уступала модификации записей, то есть чтобы новые поля добавлялись в отношения с такой же легкостью, с какой в базе данных создаются новые записи. В мире объектно-ориентированного программирования иногда требуется, чтобы изменения в иерархии классов проходили так же гибко, как манипуляции содержимым объектов.

Такая гибкость является важной составляющей объектной парадигмы, как было показано на примере класса Number; она имеет далеко идущие последствия для разработки больших систем. Допустим, поведение класса должно вырабатываться постепенно в течение жизненного цикла программы. Данная особенность особенно ценна для систем, которые нельзя остановить, перекомпилировать и перезапустить при изменении функциональности со временем. К этой

категории относятся большие встроенные системы: телекоммуникационные системы, финансовые базы данных и т. д. Брокерская фирма может использовать в своей программе объектно-ориентированную архитектуру, в которой каждый тип счета представлен классом. Конечно, было бы неприемлемо останавливать все финансовые операции ради изменения поведения существующего типа счета (например, функции, ограничивающей количество одновременно приобретаемых акций).

Гибкость, о которой идет речь, сводится к выбору между принятием решений на стадии компиляции и на стадии выполнения. В C++ семантика класса интерпретируется на стадии компиляции: любые изменения характеристик класса вызывают необходимость перекомпиляции. С другой стороны, объекты обладают гибкостью стадии выполнения. Именно «компиляционная» природа классов (а также принципиальные различия между классами и объектами) затрудняет эволюцию программ. В одном из возможных решений проблемы классы наделяются характеристиками объектов; другими словами, в роли класса может оказаться другой объект, предназначенный исключительно для представления общих аспектов «потомков» класса. Прототипы удовлетворяют эту потребность.

В настоящей главе описана идиома моделирования прототипов в реализации C++. Она основана на использовании единой иерархии, объекты которой играют роль, традиционно отводимую классам; в свою очередь, классы становятся концептуальными аналогами метаклассов (как в языке Smalltalk).

8.1. Пример с прототипами класса Employee

Предположим, мы строим систему, моделирующую работу некой организации. В эту иерархию входят классы `Employee`, `Manager`, `VicePresident` и т. д. Поскольку определения ролей могут изменяться со временем, разумно выбрать подход, основанный на применении экземпляров.

В крупных проектах бывает полезно определять все классы как производные от единого общего базового класса. В частности, универсальный базовый класс может использоваться для отладки и анализа быстродействия. Кроме того, модель с единым общим базовым классом упрощает реализацию модели прототипов: все классы в системе объявляются производными от общего базового класса `Class`. В листинге 8.1 приведено объявление класса `Employee`, наследующего от `Class`. Объявление выглядит вполне обычно, если не считать дополнительных функций `make`, списки параметров которых совпадают со списками параметров конструкторов. Также обратите внимание на тот факт, что конструкторы выведены из открытого интерфейса класса (в нашем примере конструкторы находятся в секции `protected` для вызова в классах, производных от `Employee`). Функции `make` заменяют конструкторы при использовании класса в идиоме прототипа. Еще одно отличие от «обычного» класса C++ — дополнительный конструктор, в параметре которого передается объект `Exemplar`; его смысл поясняется далее.

Листинг 8.1. Класс Employee для идиомы прототипа

```
class Employee: public Class {
public:
    Employee(Exemplar);
    Employee *make();
    Employee *make(const char *name, EmployeeId id);
    long printPaycheck();
    void logTimeWorked(Hours);
protected:
    Employee();
    Employee(const char *name, EmployeeId id);
private:
    Dollars salary;
    Days vacationAllotted, vacationUsed;
    String name;
    EmployeeId id;
};

extern Employee *employee;
```

Итак, у нас появился класс; но как уже отмечалось, главная цель этого класса — служить «типом для типов», а его экземпляр использовать как фабрику для создания новых экземпляров. Следовательно, мы должны создать сам экземпляр. Прототипы могут статически определяться как объекты, существующие в течение жизненного цикла программы и обладающие файловой статической видимостью, а доступ к ним может осуществляться через глобальный указатель на класс, находящийся на более высоком уровне иерархии наследования:

```
static Employee employeeExemplar(Exemplar());
extern Employee *employee = &employeeExemplar;
```

Кроме того, можно создать анонимный прототип в куче и обращаться к нему через глобальный указатель:

```
extern Employee *employee = new Employee(Exemplar());
```

Эти определения могут находиться как в отдельном исходном файле, так и в одном файле с подробностями реализации класса прототипа. Объявления переменных объектов-прототипов могут публиковаться в глобальном заголовочном файле.

При инициализации объекта-прототипа объект Exemplar передается в качестве параметра. Тем самым выбирается конструктор, предназначенный для построения одного объекта на базе шаблона класса. Класс Exemplar не содержит информации и не имеет собственной семантики; он нужен только для устранения неоднозначности при выборе конструктора (перегруженного) для создания прототипа. Таким образом, объект Exemplar может делать все, что угодно; от него требуется лишь корректное поведение и наличие конструктора:

```
class Exemplar {
public:
    Exemplar()  {  /* Пусто */ }
};
```

Функция make объекта-прототипа вызывается для создания нового объекта. Таким образом, она берет на себя роль конструктора. В C++ конструкторы и деструкторы отличаются от «обычных» функций тем, что в действительности являются операциями самого *класса*, а не *объектов* этого класса. Функция make похожа на остальные функции класса, не считая, что в представленной схеме она выполняет запрос на создание нового объекта на базе прототипа. В обычном случае make возвращает указатель на копию прототипа. Как уже отмечалось, все новые объекты создаются «клонированием» экземпляра. Функция make может быть перегружена, а ее аргументы использоваться для параметризации содержимого полученного объекта («клонирование с мутациями»). Каждый вариант make может создавать новые экземпляры простым вызовом конструктора с аналогичной сигнатурой, обычно входящих в закрытый или защищенный интерфейс класса). В нашем примере новый объект Employee создается вызовом

```
Class *smith = employee->make("Smith","9120784393");
```

Уничтожение объекта соответствует правилам ортодоксальной канонической формы с участием деструктора класса, использованного для построения прототипа; деструктор этого класса должен быть объявлен виртуальным.

Объединим все изложенные концепции в простую абстракцию класса Employee. В листинге 8.2 приведено объявление самого класса, включающее конструктор с параметром Exemplar для создания объекта-прототипа. Кроме того, в определение интерфейса Employee входит версия make по умолчанию, а также версия make для создания нового экземпляра Employee по имени и идентификатору работника. Приведенный заголовочный файл также содержит объявление переменной объекта-прототипа employee.

Листинг 8.2. Файл Employee.h с объявлением прототипа Employee

```
#include "Class.h"
#include "Hours.h"
#include <String.h>

typedef long EmployeeId;

class Employee: public Class {
public:
    Employee(Exemplar /* Не используется */) { }

    // Конструкторы заменяются функциями make()
    Class *make() { return new Employee; }
    Class *make(const char *name, EmployeeId id) {
        return new Employee(name, id);
    }
    long printPaycheck();
    void logTimeWorked(Hours);
private:
```

```
    // Объявление конструкторов закрытыми запрещает
    // создание обычных экземпляров этого класса.
    Employee(): salary(0), vacationAllotted(0),
        vacationUsed(0), name(""), id(0) { }
    Employee(const char *emp_name, EmployeeId emp_id):
        salary(0), vacationAllotted(0), vacationUsed(0)
        {
            name = emp_name;  id = emp_id;
        }
    Dollars salary;
    Days vacationAllotted, vacationUsed;
    String name;
    EmployeeId id;
};

// Переменная используется как глобальный манипулятор
// для работы с объектом-прототипом Employee
extern Class *employee;
```

Чтобы отделение интерфейса прототипа от его реализации принесло максимальную пользу, пользовательский код не должен зависеть от объявления самого класса Employee. Это означает, что протокол взаимодействия клиентов с Employee должен определяться в контексте более общей абстракции — такой, как Class. Обратите внимание: в листинге 8.2 функции make возвращают указатели на объекты Class вместо объектов Employee, как в листинге 8.1. Дублируя операции Employee в виртуальных функциях Class, мы ограждаем пользователя от изменений в структуре или модификации закрытых функций самого класса Employee:

```
class Class {
public:
    virtual Class *make() = 0;
    virtual Class *make(const char*, EmployeeId) = 0;
    virtual long printPaycheck() = 0;
    virtual void logTimeWorked(Hours) = 0;
        ... // Чисто виртуальные функции
            // для других производных классов
};
```

Объявление манипулятора для прототипа employee организуется отдельно от объявления самого класса Employee, возможно, с дублированием в каждом клиентском исходном файле. При этом сохраняется полный доступ к функциям Employee. Чтобы обеспечить такую возможность для всех прототипов, класс Class должен объявить полную совокупность всех функций прототипов, что может создать проблемы с сопровождением и нежелательным распространением последствий изменений. Вопросы баланса между вынесением объявлений функций на верхний уровень иерархии и организацией доступа на нижних уровнях рассматриваются далее.

Файл с расширением .c содержит только сам прототип (объявленный с файловой статической видимостью, чтобы он оставался невидимым за пределами файла) и видимый извне «манипулятор» прототипа, объявленный с типом Class*:

```
#include "Employee.h"

// Сам объект-прототип: предок, создающий
// все остальные объекты Employee.
static Employee employeeExemplar(Exemplar());

// Здесь инициализируется глобальный
// манипулятор Employee
Class *employee = &employeeExemplar;
```

Альтернативная форма выглядит так:

```
#include "Employee.h"

Class *employee = new Employee(Exemplar());
```

Функции с таким же успехом могут определяться в этом файле, если они не были определены как подставляемые в заголовочном файле.

Использование прототипов в программе несколько отличается от использования реализации той же абстракции на базе «чистых» классов (листинг 8.3). Вместо вызова new для заданного имени класса операция make выполняется с прототипом, фактический тип которого неизвестен; программа обращается к нему через указатель Class*.

Листинг 8.3. Сравнение ортодоксальной канонической формы с прототипом

Код с классом:

```
#include "Employee.h"
int main() {
    Employee *ted =
        new Employee(
            "ted", 2833763108
        );
    ...
    ted->logTimeWorked(8);
    ted->printPaycheck();
    delete ted;
}
```

Код с прототипом:

```
#include "Employee.h"
int main() {
    Class *ted =
        employee->make(
            "ted", 2823763108
        );
    ...
    ted->logTimeWorked(8);
    ted->printPaycheck();
    delete ted;
}
```

Если потребуется более серьезная проверка типов, указатель на прототип можно объявить и использовать в виде Employee*, как показано в листинге 8.1. За проверку типов приходится расплачиваться более тесной привязкой пользователя Employee к интерфейсу класса Employee. В этом случае объявление манипулятора в заголовочном файле уже не создает никаких проблем. Также можно выбрать путь, больше соответствующий «духу C++», и объявить манипулятор прототипа статической открытой переменной класса (листинг 8.4). В этом случае доступ

к прототипу в коде приложения осуществляется конструкцией Employee::exemplar вместо employeeExemplar:

```
Employee *aWorker =
            Employee::exemplar->make("Joe", 123456789);
```

ПРИМЕЧАНИЕ

Используйте прототипы в тех случаях, когда характеристики типа должны изменяться на стадии выполнения; например, изменение данных прототипа Employee может изменить поведение (характеристик типа) всех объектов Employee. Прототипы также удобны для управления всеми объектами некоторого типа; например, в них можно сосредоточить атрибуты и функции аудита или управления ресурсами данного класса. И все же главная область применения прототипов связана с идиомами, описанными в этой главе и в главе 9. Все эти идиомы направлены на получение дополнительной гибкости на стадии выполнения и снижение последствий от внесения изменений.

Листинг 8.4. Исходные файлы прототипа

Файл Employee.h:

```
class Employee {
public:
    static Employee *exemplar;
    ...
};
```

Файл Employee.c:

```
Employee *Employee::exemplar = new Employee(Exemplar());
```

8.2. Прототипы и обобщенные конструкторы

Прототипы позволяют повысить уровень обобщения при программировании на C++. Хотя C++ поддерживает виртуальные функции класса, «виртуальные конструкторы» не имеют смысла. Тем не менее, необходимость в некоем аналоге виртуальных конструкторов возникает в том случае, если *тип* создаваемого объекта зависит от параметров конструктора или глобального контекста. В этом разделе показано, как эмулировать эту семантику при помощи прототипов.

Предположим, у вас имеются некоторые анонимные данные (скажем, имя или идентификатор), содержащие достаточно информации, чтобы класс Employee мог определить фактический тип создаваемого объекта (начальник, штатный работник, внештатный работник и т. д.) по контексту. Данные могут читаться из файла на диске или вводиться оператором в интерактивном режиме. Класс Employee располагает всей необходимой информацией для интерпретации этих данных. Итак, нужно построить объект, который бы соответствовал типу работника, описанного данными. Но так как код построения объекта привязывается к типу объекта на стадии компиляции, а данные могут появиться лишь на стадии выполнения, система классов не позволит сделать это напрямую.

Одна из групп идиоматических решений была представлена ранее под обобщенным названием *виртуальных конструкторов* (см. главу 5). Прототипы также обеспечивают естественное решение этой задачи и могут рассматриваться как обобщение концепции виртуальных конструкторов. Вспомните, что объекты создаются на базе прототипов посредством вызова функции make для объекта-прототипа, а функция make может перегружаться, как и конструкторы. Анализируя свои параметры, функции make могут выбрать нужный класс и вернуть указатель на объект соответствующего типа.

Предположим, должность работника компании кодируется в первом символе идентификатора: M — начальник, L — штатный работник, V — вице-президент, и т. д. Примерная схема обобщенного построения объекта на основании пары «имя/идентификатор» показана в листинге 8.5. Здесь информация обо всех классах, производных от Employee, выделена в операцию make этого класса. Объект Employee в этом контексте играет две роли: базового класса, связывающего воедино поведение своих производных классов (то есть всех разновидностей работников), и прототипа, используемого для управления *всеми* типами Employee. При таком подходе следующая программа создает объект «типа» Manager:

```
int main() {
    Employee *joe = employee->make("Joe",
      EmployeeID("M012345678"));
    ...
}
```

Листинг 8.5. Обобщенный конструктор Employee

```
class Class {
public:
    virtual Class *make(const char*,EmployeeID);
    // ...
};

class Employee:  public Class {
public:
    Class *make(const char*, EmployeeID);
    // ...
};

Employee *employeeExemplar = new Employee(Exemplar());

class VicePresident: public Employee {
    // ...
};

class LineWorker:  public Employee {
    // ...
};
```

```
Class *
Employee::make(const char *name, EmployeeID id)
{
    Class *retval = 0;
    switch (id.firstChar()) {
    case 'M':    retval = new Manager(name, id);
                 break;
    case 'V':    retval = new VicePresident(name, id);
                 break;
    // ...
    }
    return retval;
}
```

Идиома централизованного управления коллекцией взаимосвязанных прототипов из единого прототипа-«попечителя» весьма полезна. Она называется *идиомой сообщества прототипов*: группа взаимосвязанных прототипов объединяется единым представителем, называемым *распорядителем сообщества*.

ПРИМЕЧАНИЕ

Сообщества прототипов позволяют объединить достоинства прототипов в области гибкости на стадии выполнения с достоинствами «виртуальных конструкторов».

8.3. Автономные обобщенные конструкторы

Рассмотренный пример отлично работает, если класс **Employee** располагает полной информацией о своих производных классах. Однако такое требование затрудняет возможную эволюцию программы: если при каждом добавлении нового производного класса в **Employee** придется вносить изменения, это нарушит абстракцию класса. Вместо этого согласно принципам объектно-ориентированного проектирования информация о системе должна распространяться в соответствующих классах.

Для этого нужно сделать следующее.

1. Определить набор взаимосвязанных объектов, которые будут интерпретироваться как происходящие от одного обобщенного прототипа (например, классы работников в предыдущем примере).
2. Создать глобальный список, доступный всем функциям этих классов. Элементы списка должны содержать указатели на объекты интересующих вас классов.
3. Организовать создание прототипа для каждого такого класса и его «регистрацию» в списке.
4. Проследить за тем, чтобы операция **make** каждого прототипа при вызове с неправильным параметром возвращала нулевой указатель.

5. Изменить функцию make главного прототипа так, чтобы она перебирала элементы списка и итеративно применяла операцию make к каждому прототипу с копией своих параметров. Вернуть нужно первое ненулевое возвращаемое значение.

Мы назовем этот подход идиомой *автономных обобщенных прототипов*. Каждый прототип наделяется «интеллектом», позволяющим ему проверить свои параметры и определить, ему ли они предназначены; если нет, та же возможность предоставляется всем прототипам того же сообщества. Представьте себе доску объявлений, вокруг которой собрался народ в ожидании объявлений о найме на работу. Когда появляется новое объявление, все ожидающие выстраиваются в заранее заданном порядке, и им последовательно предоставляется возможность отреагировать на это объявление. Первый, кто согласится, получает работу.

В примере, приведенном в конце главы, эта идиома используется для построения простого анализатора. Объекты-прототипы создаются как статические переменные, глобальные по отношению к каждому из производных классов Atom: Number, Name, Punct, Oper и т. д. Создание экземпляра любого из объектов-прототипов приводит к вызову конструктора базового класса Atom::Atom(Exemplar); этот конструктор включает объект-прототип в связанный список list, являющийся статической переменной класса Atom:

```
Atom(Exemplar /* Не используется */) {
    next = list; list = this;
}
```

Класс Atom заменяет все свои производные классы с позиций создания экземпляров. Пользователи пакета конструируют только новые объекты Atom, то есть объекты Number и Name обычно не создаются напрямую. Функция make(String&) класса Atom последовательно передает свой параметр функции make каждого объекта в списке. Первый класс, обнаруживший соответствие между своим типом и префиксом String, находит в строке совпадение как можно большей длины и возвращает ненулевое значение из своей функции make:

```
virtual Atom *make(String &s) {
    Atom *retval = 0;
    extern Atom *atomExemplar;
    if (this != atomExemplay) {  // Предотвращение бесконечной рекурсии
        for (Atom *a = list; a; a = a->next) {
            if (retval = a->make(s)) break;
        }
    }
    return retval;
}
```

Операции make возвращают результат вызова конструктора своего класса, если только конструктор не установит флаг ошибки, означающий, что входные данные не были распознаны как относящиеся к его типу. Функция Atom::make просто возвращает пользователю первое ненулевое значение, возвращенное функцией make производного класса, как свое возвращаемое значение.

Обратите внимание: в этом примере порядок регистрации прототипов производных классов в списке может оказаться существенным. Прототип должен

обязательно провести поиск подстрок наибольшей длины, прежде чем переходить к поиску более коротких.

К достоинствам прототипов следует отнести то, что классы утрачивают свою первостепенную роль. Знание внутреннего устройства (закрытых членов) объекта необходимо только при создании новых объектов и внутри функций класса. Поскольку объекты создаются не оператором new, а функцией make, для создания объектов не нужно разбираться во всех интерфейсах базового класса; вся необходимая информация сосредоточена в функции make объекта. В свою очередь, это означает, что знание структуры класса ограничивается самим классом; клиенты класса взаимодействуют с механизмом прототипов на более высоком уровне (эта тема рассматривается далее).

ПРИМЕЧАНИЕ

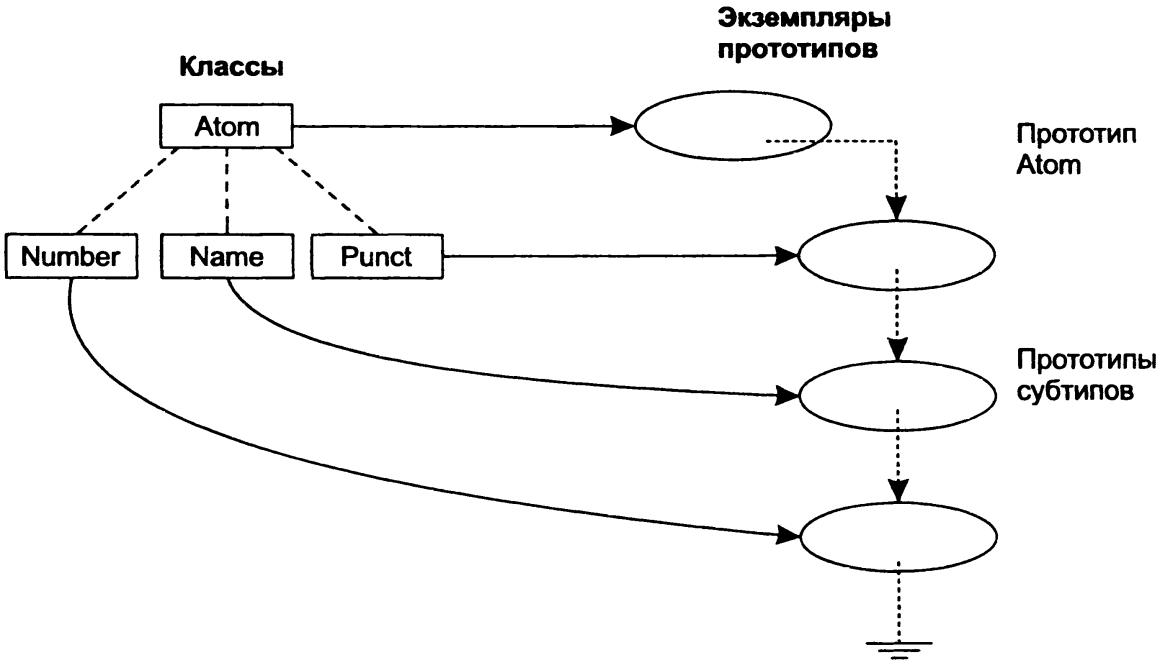
Идиома автономных обобщенных прототипов используется как механизм реализации семантики «виртуальных конструкторов» в тех ситуациях, в которых требуется избежать централизации сведений о классах в сообществе прототипов. Сведения об инициализации объектов остаются распределенными между классами, задействованными в реализации схемы.

8.4. Абстрактные базовые прототипы

Как же организуется взаимодействие объектов при использовании описанной идиомы? В соответствии с правилами C++ оно происходит через функции объектов (то есть без сообщений, транзакций и прочей экзотики). Любой клиент объекта-прототипа должен располагать объявлением интерфейса класса прототипа (сигнатурой) во время компиляции. Поскольку интерфейсы классов большинства объектов прототипов хранятся в исходных файлах, откуда берется эта сигнатура?

Ответ на этот вопрос связан с применением виртуальных функций для поддержки высокоуровневых архитектурных абстракций, называемых *абстрактными базовыми прототипами*. Конкретные классы реализаций объявляются производными от классов более высокого уровня, которые доступны глобально, а их объявления хранятся в глобальных заголовочных файлах. Классы высокого уровня немногочисленны и образуют неглубокую иерархию. На вершине иерархии находится абстрактный базовый класс Class, сигнатура которого отражает большую часть операций большинства классов в системе (рис. 8.1):

```
class Class {
public:
    virtual Class *make() = 0;
    virtual Class *make(const char *, EmployeeID) = 0;
    virtual long printPaycheck();
    virtual void logTimeWorked();
    virtual ~Class();
    ...
};
typedef CLass *Classp;
```



Все запросы приложения на создание новых объектов обрабатываются экземпляром прототипа Atom, который последовательно опрашивает свои подчиненные классы и предлагает им обработать запрос. Первый прототип в списке, который успешно разберет строку, переданную функцией Atom::make, получает право обработать запрос и возвращает объект своего класса.

Рис. 8.1. Автономные обобщенные прототипы для классов Atom

Другими словами, мы собираем объявления всех функций из группы взаимосвязанных классов и объединяем их в сигнатуре Class. Класс Class является абстрактным базовым классом, поэтому создание его экземпляров невозможно. Тем не менее, указатель Class* (или его эквивалент Classp) может использоваться для ссылок на любой объект, родословная которого восходит к Class.

При таком подходе классы утрачивают свою роль в качестве механизма классификации. Объявляя все функции базового класса Class виртуальными, мы обеспечиваем возможность обращения к любому объекту через указатель Classp; все выглядит так, словно все объекты относятся к классу Class. Программист работает с несколькими классами, предоставляющими интерфейс ко всем системным объектам; эти классы находятся на вершине неглубокой иерархии наследования, узкой в верхней части и широкой снизу. Таким образом, при проектировании акцент делается на функциях, а не на классах. Однако эти функции остаются полиморфными в том смысле, что работают со многими формами объектов Class. В каком-то смысле механизм наследования C++, называемый *полиморфизмом включения*, «выворачивается наизнанку». Полученная форма может рассматриваться как *параметрический полиморфизм*: сначала мы определяем полиморфные функции и собираем их в абстрактный базовый прототип, на основе которого создаются все производные классы приложения. С точки зрения архитектуры можно провести аналогию с *шаблонными функциями* C++ (см. главу 7): структура классов также образует широкую, неглубокую иерархию. В данном случае мы стремимся не к многократному использованию исходных текстов, как в случае шаблонных функций, а к полиморфизму времени выполнения, который шаблоны

обеспечить не могут. Различия между абстрактными базовыми прототипами и более традиционным «классовым» подходом проявляются не в механике, а в смысловом выделении. В обоих случаях функции классов идентифицируются на стадии выполнения, но при использовании абстрактных базовых прототипов детализированная иерархия классов сворачивается в более простую иерархию укрупненных интерфейсов.

Раз классы образуют иерархию, аналогичная иерархия должна существовать и для прототипов. Желательно, чтобы клиенты групп классов (таких, как Employee и Atom) работали с группой на наивысшем уровне абстракции, поэтому функции выделяются в классы, находящиеся на вершине иерархии. Каждое приложение имеет собственный абстрактный базовый прототип; для анализатора — это Atom, для бухгалтерской системы — Employee, и т. д. Конечно, можно разрешить клиентам доступ к иерархии на разных уровнях, чтобы они использовали низкоуровневые классы, когда возникнет необходимость в дополнительной специализации. Но при этом клиентский код начинает зависеть от интерфейса низкоуровневых классов, а для обращения к их сигнатурам клиентам приходится включать дополнительный заголовочный файл директивой #include.

Так мы приходим к важной альтернативе из области проектирования. На одном полюсе находится «толстый» абстрактный базовый класс, возглавляющий иерархию, и через него вызываются все функции. Пользователь такой иерархии общается только с одним абстрактным базовым классом. На другом полюсе находится более традиционная иерархия, в которой доступ к конкретным функциям классов осуществляется только из производных классов. Хотя общие операции всех объектов, созданных на основе иерархии, могут вызываться через абстрактный базовый прототип, для доступа ко всем операциям необходимо раскрыть интерфейсы классов нижних уровней. Таким образом, проектировщик выбирает между одной тщательно проработанной абстракцией и множеством более простых абстракций. При оценке подобных решений следует учитывать возможную эволюцию программы и ее последствия для перекомпиляции проекта.

Все функции класса Class объявлены виртуальными, причем в большинстве они являются чисто виртуальными; иначе говоря, они просто резервируют место и не имеют тела. К числу виртуальных функций, определенных для Class, также принадлежат функции make. Это означает, что если программа содержит указатель p на некоторый объект прототипа, а при ее компиляции использовался заголовочный файл Class.h, в ней можно вызвать функцию p->make для получения указателя на новый объект, клонированный с прототипа. Виртуальная функция make обычно строит новый объект, вызывая конструктор своего класса.

Как программа получает указатель на нужный прототип? Как было показано ранее, большинство прототипов сохраняет указатели на себя в глобальной переменной, предназначенной специально для этой цели; значение указателя присваивается при вызове соответствующего конструктора X::X(Exemplar). Вместо присваивания глобальных имен этим объектам также можно создать специальный объект (сервер имен), в котором будут регистрироваться все сформированные прототипы.

Подведем итог: код каждого класса и манипулятор его объекта-прототипа находятся вместе в своем исходном файле. Глобально экспортируемая информация складывается из манипуляторов глобальных прототипов и нескольких классов, интерфейсы которых объявляют объединение интерфейсов своих производных классов. Интерфейсы этих абстрактных прототипов определяются в заголовочном файле C++; этот файл должен включаться директивой #include всеми клиентами, желающими сконструировать или использовать объект этих типов.

8.5. Идиома фреймовых прототипов

Описанный механизм неплохо подходит для изоляции информации в классах; кроме того, он решает проблему нарушения инкапсуляции базовых классов при добавлении новых производных классов. И все же остается одна серьезная проблема: добавление новой функции в любой класс, входящий в сообщество прототипов, должно отражаться в некоем общем базовом классе (классе самого нижнего уровня, доступного для пользователей объектов). Вследствие этого в большинстве реализаций C++ придется перекомпилировать все эти классы.

Одно из возможных решений этой проблемы связано с другой альтернативой. Допустим, каждый класс содержит единственную функцию с именем doit, параметры которой определяют операцию и список аргументов этой операции. Мы можем значительно упростить интерфейс классов за счет ведения глобального списка операций, обновляемого при каждом включении в систему нового семантического интерфейса. Использование семантически слабых имен функций вроде doit наводит на мысль о слабом или случайном соответствии между сущностью и классом, отражающим ее семантику; стоит вспомнить предупреждения из 6.5. Тем не менее, включение новой семантической операции в интерфейс уже не требует добавления новой функции — просто в списке появляется операция doit с новым параметром, интерпретируемым соответствующим производным классом, что существенно снижает потребность в перекомпиляции. Поведение абстракции отражается уже не функциями класса, а таблицей, используемой функцией doit. Класс представляет «толстый» интерфейс уже не потому, что базовый класс содержит все функции, которые могут не понадобиться в производных классах, а потому что параметры функции класса могут адаптироваться для сколь угодно широкого спектра функций, операций и т. д. Такие операции являются аналогами *фреймов* из области искусственного интеллекта (также называемых *слотами*). Фрейм резервирует место для некоторой информации; чем больше фреймов (переменных состояния) заполнено, тем полнее объект описывает представляемую им абстракцию.

Вообще говоря, семантика такого интерфейса превышает уровень doit. В конце главы приводится расширенная версия класса Employee из предыдущего примера. В ней класс Class используется для обращения к объектам, представляющим практически все, от разных видов работников до денежных сумм. Класс Employee объявляется производным от Class, и у него имеются свои производные классы VicePresident, Manager и т. д.

В примере, приведенном в конце главы, все объекты интерпретируются как контейнеры с именованными фреймами, используемыми для чтения и записи значений, а также для выполнения специальных операций в контексте передаваемых параметров. Все объекты поддерживают стандартный набор базовых операций: функция at выполняет ассоциативную выборку из фрейма; функция atPut — ассоциативное присваивание; функции getNext и putNext рассматривают объект как последовательную составную структуру данных (например, для последовательного перебора фреймов). Функция atPut также может вызываться ради побочных эффектов; в этом случае параметр вместо данных, заносимых во фрейм, содержит данные для выполняемой операции; примером служит директива PrintPaycheck. В этом случае в ключевых полях передаются константы перечисляемого типа; менее эффективный, хотя и более наглядный подход со строковыми ключами кратко продемонстрирован в главе 6 (см. с. 233).

Поскольку полиморфизм поддерживается на уровне функций доступа (а не на уровне функций классов, как в случае виртуальных функций), полиморфное поведение и семантику наследования приходится частично кодировать в самих классах. Впрочем, делать это совсем несложно. В функции VicePresident::atPut команда выбора содержит секцию default для директив, не опознанных прототипом. В этом случае операция перенаправляется базовому классу VicePresident (то есть Employee), играющему роль делегата для подмножества операций atPut, выполняемых с VicePresident. Такое поведение имеет много общего с механизмом поиска методов в Smalltalk, при котором запрос передается по цепочке наследования в поисках класса, способного обработать «сообщение». Единственное отличие этой разновидности полиморфизма от виртуальных функций C++ заключается в отсутствии проверки типов на стадии компиляции, из-за чего на стадии выполнения могут происходить исключения. В нашем примере функции доступа сообщают о невозможности обработки запроса, возвращая нулевое значение; вместо этого можно было бы предпринять более радикальные меры (например, запустить исключение).

С практической точки зрения выбор между функциями и фреймами означает выбор между удобством и гибкостью. Добавление нового прототипа или функции требует минимальных административных хлопот. При использовании абстрактных прототипов добавление нового класса в программу может сводиться к компиляции нового типа и его параллельной дозагрузке в существующий код. Некоторые среды могут поддерживать параллельную компоновку нового объектного кода в загрузочных модулях (файл a.out в UNIX). В системах с непрерывным режимом работы, поддерживающих параллельную загрузку на стадии выполнения, новые прототипы могут интегрироваться в *выполняемую* программу (см. главу 9). Например, если бы анализатор, упоминавшийся ранее в этой главе, был частью языка обработки запросов в системе автоматического бронирования мест или билетов, то мы бы могли воспользоваться средствами операционной системы для загрузки классов новых языковых конструкций в работающую систему. Механизм загрузки программных модулей включает новый прототип в список, поддерживаемый распорядителем в идиоме сообщества прототипов. Включение

в список означает подключение нового прототипа к системе. Функциональность нового программного модуля становится доступной немедленно, а его установка не нарушает работы остальных программ.

Недостатком идиомы прототипов следует считать то, что она не относится к «естественным» средствам C++. Это затрудняет обучение персонала и может потребовать разработки нового инструментария для эффективного администрирования программ, написанных в этом стиле для большого проекта. Кроме того, снижаются гарантии того, что операции будут вызываться только для объектов соответствующего типа; авторы должны обеспечить корректное поведение класса при неожиданных результатах. Наконец, по быстродействию этот стиль программирования несколько уступает стилю со связыванием на стадии компиляции.

ПРИМЕЧАНИЕ

Используйте идиому фреймовых прототипов в тех ситуациях, когда гибкость времени выполнения важнее преимуществ проверки типов во время компиляции. Кроме того, прототипы снижают требования к перекомпиляции при внесении изменений, поэтому они могут применяться при построении пробных версий программ и аналитическом программировании.

8.6. Условные обозначения

В 6.4 была описана система условных обозначений связей между объектами и классами, основанная на «традиционной» модели классов и объектов C++. Введение прототипов означает, что эту систему необходимо расширить и включить в нее новые обозначения для новых связей.

Большинство изменений в основном относится не к смыслу, а к субъекту. Например, связь «IS-A» теперь может определять отношения между экземплярами: любой объект, создаваемый как клон другого объекта, находится с оригиналом в отношениях «IS-A». Между тем клон может ассоциироваться с другими объектами и перенаправлять им некоторые из своих операций для изменения своей функциональности. Если сигнатура клона сохраняет базовую семантику оригинала (см. главу 6), налицо отношение «IS-A».

При использовании идиомы прототипов классы отходят на второй план. Такие отношения, как «HAS-A», уже не разделяются на два разных случая для объектов и классов со слегка различающимся смыслом: все связи определяются между экземплярами. Отношение «CREATES-A» теряет свою особенность как связующее звено для объектов и классов; теперь все делается для объектов.

Конечно, классы присутствуют и в этой схеме; при желании можно адаптировать систему обозначений и выделить роль классов. Прототипы заменяют классы, а классы C++ самих прототипов являются классами этих «суррогатов классов». В некоторых языках эти «классы классов» называются *метаклассами*. Таким образом, каждый экземпляр прототипа связан специальным отношением «HAS-METACLASS» со своим классом C++.

Поскольку при использовании прототипов роль, обычно ассоциируемая с классами, отводится объектам, для наглядного изображения семантики программы

понадобится способ представления гибридных отношений между объектами и классами. На рис. 8.2 показано, как выглядит такая диаграмма для примера Atom; в основном она представляет собой «формализованную версию» рис. 8.1 с использованием записи, описанной в главе 6 в контексте простого приложения:

```
extern Atom *atom;
Expr parse(String s) {
    ...
    Atom *lhs = (*atom)->make(s);
    Atom *oper = (*atom)->make(s);
    Atom *rhs = (*atom)->make(s);
    Expr expr = (*exprExemplar)->make(lhs, oper, rhs);
   ...
    return expr;
}
```

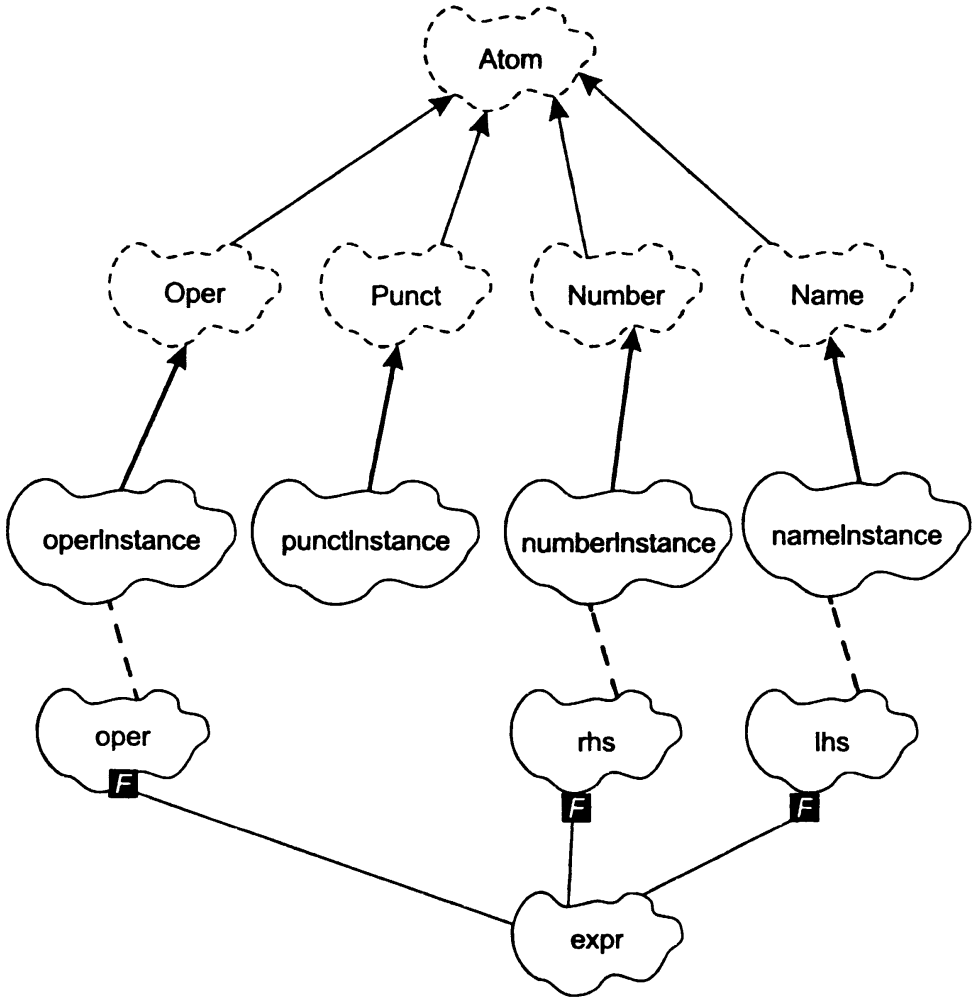


Рис. 8.2. Гибридная диаграмма объекта/класса

Классы Oper, Punct, Number и Name являются метаклассами приложения. Их «метаклассовые» отношения с экземплярами прототипов operInstance, punctInstance, numberInstance и nameInstance обозначены стрелками. Тем не менее, Oper, Punct, Number и Name реализованы как классы C++, поэтому здесь они изображены как

классы (пунктирные контуры). «Классы» приложения — operInstance, punctInstance, numberInstance и nameInstance — являются прототипами, которые в C++ представляют собой объекты, и это обстоятельство отражено на диаграмме. Отношения создания объектов между объектами приложения (oper, rhs и lhs) и их прототипами изображены в виде специальных связей, представленных на рисунке пунктирными линиями.

8.7. Прототипы и администрирование программ

Для такого стиля программирования характерны программы с малым количеством заголовочных файлов. Обычно для каждого класса создается отдельный исходный файл; объявление класса хранится в самом исходном файле, а *не в заголовке*. Иногда один исходный файл для класса получается слишком большим; в таких ситуациях объявление класса размещается в *локальном* заголовочном файле вместо *глобального*. В некоторых ситуациях объектам одного класса требуется предоставить более детализированный доступ к объектам другого класса, и тогда некоторые объявления классов приходится выносить в глобальные заголовки. Таким образом, объявления класса размещаются в трех местах: в исходных файлах модуля, в локальных заголовочных файлах и в глобальных заголовочных файлах.

Особое внимание следует уделить значениям перечисляемого типа, используемым для идентификации директив в идиоме фреймовых прототипов. Поскольку специфика функциональности конкретного объекта производного класса может отсутствовать в обобщенном интерфейсе абстрактного базового класса, в протоколах, определяемых интерфейсами абстрактных базовых классов, приходится определять дополнительные субпротоколы. По соображениям эффективности для таких протоколов может потребоваться связывание во время компиляции, а для предотвращения конфликтов необходимо организовать их централизованное администрирование. В результате в программе появляются длинные списки «типов сообщений», известных на стадии компиляции, которые в каком-то смысле являются моральным эквивалентом глобальных данных. Но так как разновидности сообщений фактически образуют перечисляемый тип, их можно линейно упорядочить, а добавление нового типа в конец перечисления не создает серьезных последствий для других частей программы — как в семантическом отношении, так и в отношении перекомпиляции.

Существует и другая, менее эффективная альтернатива — привязка индексов типов сообщений к последовательным целым числам на стадии выполнения. Прототип создает новый объект MessageType и связывает его с именем в собственном пространстве имен:

```
class MessageType {
public:
    MessageType(): typeField(nextType++) { }
    operator int() { return typeField; }
```

```
    int operator==(MessageType m) {
        return typeField == m.typeField;
    }
private:
    static int nextType;
    int typeField;
};

int MessageType::nextType = 0;

class AssociativeArray {
public:
    AssociativeArray {
    void atPut(void *element, MessageType key) {
        if (key == size) sizeFrame = *((int*)element);
        else if ...
    }
    void *at(MessageType key) {
        ..
    }
    static MessageType size, print, ...
private:
    int sizeFrame, ...
};

static MessageType AssociativeArray::size,
       AssociativeArray::print ..

int main() {
    AssociativeArray a;
    MessageType AssociativeArrayExample,
        CirclistExample;
    a.atPut((void*)233, AssociativeArrayExample);
    ...
    int size = a.at(a.size);
    ...
}
```

По эффективности такое решение уступает варианту с enum, но бесспорно превосходит вариант с классом String из исходной версии примера.

Упражнения

1. Измените простой анализатор, приведенный в конце главы, так, чтобы конструктор Atom мог многократно вызываться для одной строки; распознавание лексем и их преобразование в объекты должно сопровождаться деструктивным поглощением символов.

2. Напишите сервер имен для абстрактного базового прототипа (например, простого анализатора). Каждый прототип должен регистрироваться у сервера имен (когда?). Реализуйте для сервера имен функцию lookup(const char*), находящую прототип по имени. Где бы вы предложили хранить символическое имя прототипа?
3. Оптимизируйте сервер имен из предыдущего упражнения так, чтобы поиск прототипа по символическому имени производился только при первом вызове функции lookup с заданной строкой. Подсказка: организуйте сохранение ассоциаций «имя/прототип» между вызовами lookup.
4. Завершите приведенный далее пример с фреймовым прототипом.

Простой анализатор с прототипом

Ниже приведена реализация прототипа простого анализатора, предложенного в главе 5.

```
class Exemplar { public: Exemplar() { } };

class Atom {
protected:
    static Atom *list;
    Atom *next;
    static char errFlag;
public:  // Объединение сигнатур всех разновидностей Atom
    Atom(Exemplar) { next = list; list = this; }
    virtual Atom *make(String &s) {
        Atom *retval = 0;
        extern Atom *atomExemplar;
        for (Atom *a = list; a; a = a->next) {
            if (a != atomExemplar) {
                if (retval = a->make(s)) break;
            }
        }
        return retval;
    }
    Atom() { next = 0; errFlag = 0; }
    virtual ~Atom() {}
    virtual unsigned long value () { return 0; }
    virtual String name() { return String("error"); }
    virtual operator char() { return 0; }
};

Atom* Atom::list = 0;
static Atom atomInstance(Exemplar());
Atom *atomExemplar = &atomInstance;

class Number: public Atom {
```

```
public:
    Number(Exemplar a): Atom(a) { }
    Number(String &s): Atom() {
        sum = 0;
        for(int i=0; s[i] >= '0' && s[i] <= '9'; i++) {
            sum = (sum * 10) + s[i] - '0';
        }
        char c = s[i]; errFlag = !(ispunct(c) || isspace(c));
    }
    Number(Number &n) { sum = n.value(); }
    Atom *make(String &s) {
        Atom *retval = new Number(s);
        if (errFlag) { delete retval; retval = 0; }
        return retval;
    }
    ~Number() {}
    unsigned long value() const { return sum; }
private:
    unsigned long sum;
};

static Number numberInstance(Exemplar());

class Name: public Atom {
public:
    Name(Exemplar a): Atom(a) { }
    Name(String &s): Atom() {
        for(int i=0; s[i] >= 'a' && s[i] <= 'z'; i++)
            n = n + s(i,1);
        // Если значение не является именем, сообщить
        // вызывающей стороне установкой флага errFlag
        if (!isalnum(s[i])) errFlag++;
    }
    Atom *make(String &s) {
        Atom *retval = new Name(s);
        if (errFlag) { delete retval; retval = 0; }
        return retval;
    }
    Name(Name &m) { n = m.name(); }
    ~Name() {}
    String name() { return n; }
private:
    String n;
};

static Name nameInstance(Exemplar());

class Punct: public Atom {
public:
```

```
    Punct(Exemplar a): Atom(a) { }
    Punct(String &s): Atom() {
        if (!ispunct(s[0])) errFlag++;
        s = s(1, s.length()-1);
    }
    Atom *make(String &s) {
        Atom *retval = new Punct(s);
        if (!errFlag) { delete retval; retval = 0; }
        return retval;
    }
    Punct(Punct &p) { c = char(p); }
    operator char() { return c; }
    ~Punct() {}
private:
    char c;
};

static Punct punctInstance(Exemplar());

class Oper: public Atom {
    ...
};
```

Фреймовые прототипы

Ниже приведена простая иерархия классов Employee, Dollars и т. д. с общим абстрактным базовым прототипом Class. Класс Class определяет «толстый интерфейс» к классам, расположенным ниже него в иерархии наследования.

```
extern class Class *dollarsExemplar;  // Опережающая ссылка

class Class {
public:
    enum Directive { PrintPaycheck, TimeWorked,
      GiveRaise, Salary, Stock, ToString, Value };
    virtual Class *atPut(Directive key, Class *arg) = 0;
    virtual Class *at(Directive key) = 0;
    virtual Class *getNext() = 0;
    virtual Class *putNext(Class *) = 0;
    virtual Class *make(Class *param) = 0;
    virtual Class *make(int=0, int=0, int=0, int=0);
    virtual int toInt() = 0;
};

class Employee: public Class {
public:
    Class atPut(Directive key, Class *arg) {
        switch(key) {
```

```
        case Salary:      salary = arg->toInt(); return this;
        case Print:       cerr << *this; return this;
        default:          return 0;
        }
    }
    Class *at(Directive key) {
        switch(key) {
        case Salary:      return dollarsExemplar->make(salary);
        ...
        default:         return 0;
        }
    }
    int toInt() { return salary; }
    Class *make(int=0, int=0, int=0, int=0);
    Class *make(Class *param) = 0;
    Class *getNext() = 0;
    Class *putNext(Class *) = 0;
protected:
    Dollars salary;
    Days vacationAllotted, vacationUsed;
};

class VicePresident: public Employee {
public:
    Class atPut(Directive key, Class *arg) {
        switch(key) {
        case GiveRaisealary: salary = arg->toInt() + salary;
                             stockOptions *= 1.05;
                             break;
        default:          return Employee::atPut(key, arg);
                          break;
        }
        return this;
    }
    Class *at(Directive key) {
        switch(key) {
        case Stock:       return dollarsExemplar->make(stockOptions);
        ...
        default:          return Employee::at(key, arg);
        }
    }
    Class *make(Class *param) {
        name = param->getNext()->at(ToString);
        salary = (int)param->getNext()->at(toInt);
        stock = (int)param->getNext()->at(toInt);
    }
    Class *make(int i=0, int j=0, int k=0, int l=0) {
        Class *n = new VicePresident;
        n->salary = i; n->stockOptions = j; return n;
```

```
    }
    Class *getNext() { return 0; }
    Class *putNext(Class *) { return 0; }
    VicePresident(Exemplar) { }
private:
    Dollars stockOption;
};

class *vpExemplar = (Class *)new VicePresident(Exemplar());

class Dollars: public Class {
public:
    Class *atPut(Directive key, Class *arg) {
        switch(key) {
        case Value:       rep = (int)arg->at(ToInt); return this;
        default:          return Class::atPut(key, arg);
        }
    }
    Class *at(Directive key) {
        switch(key) {
        case Value:       return integerExemplar->make((Class*)rep);
        default:          return Class::atPut(key, arg);
        }
    }
    int toInt() { return rep; }
    Class *make(int i=0, int j=0, int k=0, int l=0) {
        Class * n = new Dollars;
        n->rep = i; return n;
    }
    Class *make(Class *param) {
        Class *n = new Dollars;
        n->rep = int(param);
        return n;
    }
    Class *getNext() { return 0; }
    Class *putNext(Class *) { return 0; }
    Dollars(Exemplar) { }
private:
    int rep;
};

Class *dollarsExemplar = (Class*)new Dollars(Exemplar());
```

Литература

Ungar, David and Randall B. Smith. «Self: The Power of Simplicity». SIGPLAN Notices 22,12 (December 1987).

Глава 9
Эмуляция символических языков на C++

C++ предоставляет в распоряжение программиста базовые механизмы для определения абстрактных типов данных и их использования при объектно-ориентированном программировании. В то же время язык C++ остается тесно связанным с C, поэтому в нем трудно обеспечить гибкость высокоуровневых объектно-ориентированных языков вроде Smalltalk или CLOS. Как в C, так и в C++ имя переменной прочно связывается с *адресом* блока памяти, в котором хранится соответствующий объект. Оно не может рассматриваться как ярлык, который можно снять с одного объекта и наклеить на другой по своему усмотрению. Сильное связывание позволяет компилятору убедиться в том, что переменная всегда ассоциируется с объектом правильного типа. Компилятор задействует это свойство, чтобы генерировать эффективный код и предотвращать некорректное использование объектов. За эффективность и безопасность типов приходится расплачиваться гибкостью на стадии выполнения; например, переменная, объявленная с типом `float`, не может ссылаться на объект `Complex` на стадии выполнения, хотя в поведении этих двух типов есть много общего.

Smalltalk и большинство объектно-ориентированных языков на базе Lisp поддерживают две возможности, основанные на слабой связи переменных и объектов. В C++ эти возможности не поддерживаются напрямую, но могут выражаться в виде идиом при использовании соответствующего стиля программирования. Первая возможность — автоматизация управления памятью (на базе подсчета ссылок или уборки мусора). В символических языках, где переменные представляют собой ярлыки для объектов, жизненный цикл объекта не зависит от ассоциированных с ним переменных. В символических языках объекты, на которые отсутствуют ссылки в программе, автоматически уничтожаются с применением специальных методов, основанных на ослаблении ассоциаций «имя-значение». В программах C++ ослабление связей имитируется посредством обращения к объектам через указатели, причем этот дополнительный уровень логической косвенности может использоваться системой автоматического управления памятью (за подробностями обращайтесь к разделам 3.5 и 3.6).

Вторая важная особенность высокоуровневых объектно-ориентированных языков — высокий уровень полиморфизма; идиомы его поддержки подробно рассматривались

в разделе 3.5. Расширенный полиморфизм повышает гибкость и приспособляемость программной архитектуры; жесткая привязка объектов ослабляется, что упрощает их независимое сопровождение.

Автоматизация управления памятью и расширенный полиморфизм считаются двумя главными достоинствами систем на базе Lisp, Smalltalk и других объектно-ориентированных языков из семейства символических. До определенной степени их можно эмулировать на C++, в результате присущая им гибкость становится доступной и для программистов C++. Однако за эту гибкость приходится расплачиваться снижением быстродействия и дополнительными затратами памяти. Ошибки в типах, которые ранее обнаруживались системой контроля типов компилятора, при использовании этих идиом проявляются только на стадии выполнения, а их обработка зависит от качества кода проверки типов, предоставленного пользователем. Однозначных рекомендаций по выбору одного из вариантов не существует, но опыт практического проектирования поможет выбрать среди идиом этой главы (а также других глав) те, которые оптимально подходят для вашего конкретного приложения.

В настоящей главе представлены три разновидности идиом. Первая объединяет идеи предыдущих глав, касающиеся обеспечения итеративного стиля за счет сокращения влияния изменений. Будет представлена каноническая форма, снижающая влияние изменений и закладывающая основу для двух других идиом. Вторая обеспечивает обновление программы во время ее выполнения. Третья автоматизирует процесс освобождения ресурсов и неиспользуемых объектов. Каждая идиома может использоваться независимо от других, или несколько идиом могут использоваться в сочетании друг с другом.

Адаптация второй идиомы для конкретной платформы может потребовать особого опыта и стараний — все зависит от подробностей реализации при представлении данных в классах C++. Реализация, описанная в тексте, основана на комментариях ANSI к базовому стандарту языка C++ [1]. В настоящее время эта реализация работает с версией 3 AT&T USL C++ Compilation System. Предполагается, что она будет адаптироваться к другим платформам и продуктам многих разработчиков.

Следует подчеркнуть, что приемы, описанные в этой главе, не должны рассматриваться как замена для Smalltalk или объектно-ориентированных сред программирования на базе Lisp. Представленные идиомы помогут вместе с C++ пройти несколько шагов в соответствующем направлении, но за это придется расплачиваться дополнительными сложностями при программировании и возможным снижением быстродействия. Настоящая глава призвана познакомить читателя с принципами, лежащими в основе инкрементного программирования на C++, а также с возможностями оперативного обновления систем, работающих в непрерывном режиме. Кроме того, она может использоваться в качестве модели для кода, автоматически сгенерированного программой под управлением генератора приложений или языкового компилятора очень высокого уровня, предназначенного для гибких, интерактивных приложений.

9.1. Инкрементное программирование на C++

Для чего нужно инкрементное программирование на C++? Чтобы вносимые изменения *быстро* вступали в силу, а вашим коллегам не приходилось подолгу мучиться с тестированием нового кода. Быстрые итерации играют важную роль при уточнении структуры системы и исследовании поведения приложений в свете новых или изменившихся требований. Чтобы такие итерации были эффективными, затраты на инкрементные изменения должны быть минимальными.

Инкрементный подход и объектно-ориентированное проектирование

Инкрементная разработка естественно подходит для объектно-ориентированного проектирования. Инкапсуляция деталей реализации в классах делает их естественными единицами итерации, а соблюдение единого протокола для всех классов в иерархии наследования упрощает включение новых листовых классов. Все эти средства могут использоваться при проектировании в C++, но перекомпиляция и загрузка обычно требуют гораздо больше времени, чем в системах на базе Lisp и Smalltalk. Хотя в наши дни появляются мощные среды программирования, наделяющие традиционную технологию поддержкой инкрементных режимов разработки, на некоторых платформах такие технологии остаются недоступными. Например, повышенная гибкость особенно актуальна для встроенных систем, работающих вне контекста «дружественных» операционных систем и традиционного вспомогательного окружения. Инкрементные методы, описанные в этой главе, уступают своим аналогам из интегрированных сред разработки C++, но зато могут обеспечивать инкрементную разработку для многих платформ.

Сокращение затрат на компиляцию

Первой составляющей инкрементного программирования на C++ является сокращение затрат на перекомпиляцию. Наиболее эффективный способ решения этой задачи — снижение *потребности* в перекомпиляции. В C++ на стадии компиляции формируется сильная связь между переменной C++, ее типом и той областью памяти, в которой она хранится. Если эволюция программы приводит к изменению характеристик типа или адреса переменной, то код, содержащий ссылки на имя переменной, придется заново компилировать или компоновать. Изменение одного символического имени может привести к смещению соседних символических имен, что приведет к необходимости перекомпиляции всех модулей, ссылающихся *на них*. Например, при любых изменениях в интерфейсе класса придется перекомпилировать весь код, ссылающийся на *любую* часть этого интерфейса.

Большинство способов предотвращения компиляции основано на включении дополнительного уровня адресации в ссылки на символические имена. Примерами

такого подхода служат идиома «конверт/письмо» (см. 5.5) и ее производные, в том числе и идиома прототипов из главы 8. Идиомы, представленные в настоящей главе, заимствуют многие идеи из идиомы прототипов.

Возможность внесения изменений с минимальными издержками на компиляцию порождает дополнительные затраты, обусловленные лишними операциями адресации во время выполнения. В традициях большинства символических языков программирования такой подход также характеризуется более низкой безопасностью типов по сравнению с «классическим» языком C ++. Эти два обстоятельства обязательно необходимо проанализировать перед тем, как задействовать эти идиомы в реальном проекте.

Сокращение затрат на компоновку и загрузку

Второй составляющей инкрементного программирования является сокращение времени компоновки и загрузки. *Компоновкой* называется операция, обычно используемая для построения исполняемого файла программы из откомпилированных объектных файлов, а *загрузкой* — перенос кода из файлов в память для выполнения. В некоторых системах эти две операции объединяются, а в большинстве систем обе операции сопряжены с привязкой символических имен к адресам.

Эффективность компоновки и загрузки особенно важна при разработке больших систем, когда объектно-ориентированные методы приносят ощутимую пользу. Традиционно многие операционные системы, ориентированные на миникомпьютеры и микропроцессоры, не сильны в инкрементной компоновке и загрузке; тем не менее, во многих новых версиях UNIX и других систем предусмотрены средства инкрементной компоновки. Инкрементная компоновка приводит к уменьшению размеров объектных модулей и существенному ускорению по сравнению с полной компоновкой перехода системы в фазу готовности.

Даже при быстрой компоновке повторная загрузка может стать «узким местом» системы. Если инициализация системы занимает много времени, то даже при инкрементной компоновке эти затраты будут включаться в каждую итерацию. Но если код может загружаться в инициализированную, работающую программу, возможно, при условии небольших изменений вам удастся еще быстрее привести систему к готовности.

Ускоренные итерации

Ускоренный итеративный процесс оказывается наиболее эффективным на ранней стадии исследования архитектуры, когда программист создает «пробные» версии для лучшего понимания приложения. С другой стороны, такой процесс может использоваться как один из этапов процесса разработки, но только в рамках стабильной архитектуры. Если правильно организовать ускоренный итеративный процесс интеграции совместимых изменений в систему, он может превратиться в успешную методику разработки. Однако оставляя основные интерфейсы открытыми для итераций, вы своими руками готовите катастрофу, повышая энтропию системы и разрушая системную структуру.

9.2. Символическая каноническая форма

Символическая идиома является альтернативой для ортодоксальной канонической формы, представленной в 3.1 (см. с. 53). Эмуляция символических парадигм в C++ «не ортодоксальна» (или, по крайней мере, находится на приемлемом уровне ортодоксальности), поэтому для нее нужны собственные правила и формы. При использовании этой альтернативной канонической формы для классов C++ код способен моделировать многие свойства символических языков программирования вообще. Однако при этом он утрачивает большую часть компактной выразительности, характерной для прямого применения ортодоксальной канонической формы.

ПРИМЕЧАНИЕ

Идиома символической канонической формы используется в ситуациях, требующих идеологической совместимости с C или C++ с сохранением гибкости символических языков программирования. Кроме того, она может пригодиться при организации взаимодействия между символическими языками и C++. Стили и идиомы, описанные в этой главе, позволяют создать систему протипизации в среде C++ (при соблюдении некоторых правил в прикладных программах). Наконец, она может применяться в системах, работающих в непрерывном режиме, для обновления программы во время выполнения и обеспечения долгосрочной эволюции.

Каноническая форма строится на концепциях управления памятью и полиморфизма, рассмотренных в предыдущих главах. Эти концепции расширяются для обеспечения дополнительного уровня инкрементности. Среди основных аспектов канонической формы можно выделить следующие:

- автоматическое управление памятью на базе классов писем с подсчетом ссылок (см. раздел 3.5);
- запрет на выполнение пользователем операций с указателями на объекты, но таким образом, что объекты вроде бы сохраняют всю гибкость указателей (см. раздел 3.5);
- использование виртуальных функций для обеспечения дополнительной гибкости на стадиях выполнения и загрузки;
- применение идиомы прототипов (см. главу 8).

Основным «строительным блоком» символической канонической формы является небольшой набор базовых классов, используемых для построения классов писем и конвертов. Объявления этих классов размещаются в заголовочном файле k.h, приведенном в листинге 9.1. Класс Top является базовым для всех классов конвертов, а класс This — для всех классов писем. На самом деле класс Thing тоже наследует от Top; тем самым обеспечивается логическая однородность системы, которая моделирует корневые базовые классы, присутствующие во многих средах символического программирования. Классы Top и Thing играют примерно те же роли, что и классы Object, Class и Behavior в Smalltalk, хотя между этими двумя решениями не существует очевидного соответствия.

Листинг 9.1. Заголовочный файл k.h

```
// Заголовочный файл k.h
class Top {
public:
    // Объекты этого класса не содержат данных кроме поля __vptr,
    // сгенерированного компилятором. Наследование всех классов
    // от этого класса гарантирует, что в большинстве реализаций
    // __vptr всегда будет первым элементом любого объекта.
    // В некоторых реализациях для обращения к __vptr
    // используются особые механизмы, но от данного свойства
    // в символической идиоме зависит только аспект,
    // связанный с динамической загрузкой.
    virtual ~Top() { /* Пусто */ }
    virtual Top *type() { return this; }
    // Оператор delete объявлен открытым для освобождения ресурсов
    // при обновлении на стадии выполнения.
protected:
    Top()  { /* Пусто */ }
    static void *operator new(size_t l) {
        return ::operator new(l);
    }
};

typedef unsigned long REF_TYPE;

class Thing: public Top {
    // Класс Thing определяет каноническую форму всех классов писем.
public:
    Thing(): refCountVal(1), updateCountVal(0) { }
    virtual REF_TYPE deref() {   // Уменьшение счетчика ссылок
        return --refCountVal;
    }
    virtual REF_TYPE ref() {     // Увеличение счетчика ссылок
        return ++refCountVal;
    }
    virtual Thing *cutover();       // Функция обновления класса
    virtual ~Thing() { /* Пусто */  }       // Деструктор
private:
    REF_TYPE refCountVal, updateCountVal;
};
```

Эти классы обеспечивают гибкость, а также поддерживают механизмы управления памяти, обновления на стадии выполнения и модель ослабленной типизации, которые мы пытаемся позаимствовать из символических языков программирования. Подробные описания обоих классов приводятся в двух следующих разделах.

Класс Top

Класс Top возглавляет иерархию системных классов, а его свойства наследуются всеми классами. Top не содержит собственных полей данных, однако большинство компиляторов C++ включает в него одну скрытую переменную для поддержки механизма виртуальных функций C++. Эта переменная называется vptr и ссылается на таблицу виртуальных функций (vtbl). Наличие виртуальной функции в классе Top гарантирует существование такого указателя.

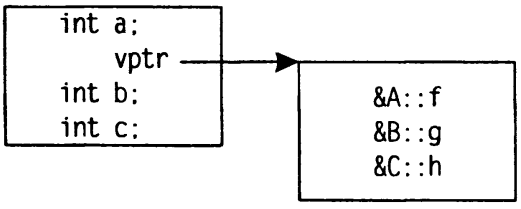
Во многих средах C++ применяются разные реализации механизма виртуальных функций, но в основном они являются вариациями одной темы. Для примера возьмем следующие три класса [1]:

```
class A {
public:
    int a;
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};

class B: public A {
public:
    int b;
    void g(int);
};

class C: public B {
public:
    int c;
    void h(int);
};
```

При таких объявлениях объект класса C будет иметь в памяти следующую структуру:



Если корневой класс не содержит данных, то все объекты интересующих нас классов будут начинаться с указателя vptr — это упрощает процедуру поиска. Располагая указателем на такой объект, любая функция сможет получить хранящийся в нем указатель vptr, а следовательно — организовать перебор содержимого таблицы vtbl для класса данного объекта. Этот принцип заложен в основу оперативной замены функции в работающих программах.

Класс Top также содержит конструктор по умолчанию (вызываемый без аргументов), находящийся в секции protected; тем самым предотвращается прямое создание объектов класса Top в программе. Виртуальный деструктор Top не делает ничего, а его виртуальность гарантирует, что вызов деструктора производного класса будет разрешаться на стадии выполнения.

Объявление оператора new для класса Top скрыто в секции protected, чтобы предотвратить возможное создание экземпляров классов конвертов в куче. Вследствие такого ограничения объекты классов конвертов могут объявляться только глобальными, локальными или встроенными объектами других классов, что позволяет компилятору полностью автоматизировать процесс их уничтожения. Оператор подменяется в производных классах писем, чтобы их экземпляры все же могли создаваться в куче. Динамическое создание и уничтожение объектов иерархии осуществляется классами писем, что гарантирует освобождение их памяти, когда надобность в ней отпадет.

Функция type подменяется в производных классах так, чтобы возвращать указатель на соответствующий прототип. Она используется для обновления классов на стадии выполнения (см. далее).

Работа класса Top зависит от компилятора. Большинство реализаций C++ генерирует код в описанном выше формате, но пользователь должен помнить, что адаптация программы для новой платформы может потребовать дополнительных усилий.

Класс Thing

Класс Thing является базовым классом для всех классов писем. Поскольку классы писем содержат основной «интеллект» приложения, семантическая сторона динамики объектов в основном сосредоточена в открытом интерфейсе Thing. В символической идиоме даже та часть управления памятью, которой обычно «заведует» класс конверта, реализуется классами в иерархии наследования Thing.

Функции deref и ref изменяют значение закрытого счетчика ссылок refCountVal. Эти функции объявлены виртуальными, чтобы они могли подменяться в производных классах. Тем не менее, для реальных приложений более характерно их объявление в виде невиртуальных, подставляемых функций. Эти функции созданы в первую очередь для удобства записи. Закрытая функция updateCountVal поддерживает инкрементную загрузку, а ее использование описано далее в этой главе.

Функция cutover требуется для преобразования существующих объектов заданного класса в объект, отражающий новую версию того же класса. Она обеспечивает преобразование данных существующих объектов из исходного формата в новый формат с введением новой версии класса в работающую систему. Обычно эта функция подменяется в производных классах, в которых она должна вызываться; ее стандартная семантика ограничивается простым возвращением указателя на исходный экземпляр. Использование функции cutover также рассматривается далее.

Следом идет пустой виртуальный деструктор, который просто гарантирует, что при вызове оператора delete для объекта класса, производного от Thing, будет выполнен завершающий код соответствующего производного класса.

Символическая каноническая форма для классов приложений

Разобравшись с базовыми классами из файла k.h, можно переходить к описанию канонических форм классов приложений, задействованных в символической идиоме. Эти классы делятся на две категории: конверты, управляющие созданием объектов и присваиванием, и письма, воплощающие большую часть прикладной семантики.

Один класс конверта может ассоциироваться с несколькими классами писем. Допустим, в некоторой архитектуре тип Number должен быть базовым типом для Double, BigInteger и Complex. В традиционном подходе C++, основанном на наследовании и виртуальных функциях, объекты этих классов делаются взаимозаменяемыми через интерфейс Number: класс Number является абстрактным базовым классом для трех других, а его экземпляры не могут создаваться в программе. В символической идиоме объекты трех производных классов тоже взаимозаменяемы через интерфейс Number. Однако класс Number в этом случае всего лишь является конвертом для классов писем Double, BigInteger и Complex. Последние три класса объявляются производными от общего базового класса NumericRep, характеризующего сигнатуру своих производных классов; этот класс представляет «обобщенное письмо» для данного приложения. Конверт Number содержит указатель, объявленный с типом NumericRep* и предназначенный для «хранения» письма. В свою очередь, обобщенный класс письма является производным от Thing, а конверт (Number) наследует от Top. Такая архитектура содержит дополнительные уровни логической адресации, обеспечивающие мощный полиморфизм и возможность обновления во время выполнения.

Обратите внимание: поскольку письмо является производным от Thing, а конверт — от Top, нам не удастся использовать общий базовый класс и для конвертов, и для писем, как это часто делалось в предыдущих примерах.

Символическая каноническая форма будет представлена на обобщенном примере с обобщенным классом конверта Envelope и обобщенным классом письма Letter. Составной объект, содержащий один подобъект Envelope и один подобъект класса, производного от Letter, представляет единую абстракцию, и эта абстракция может использоваться как гибкий компонент, наделенный мощью символической идиомы.

Любая программа или система может содержать несколько классов конвертов, следующих канонической форме Envelope, и несколько классов писем, построенных по образцу Letter и видоизмененных для соответствующей семантики. Например, программа может содержать класс Number, построенный по образцу Envelope, класс NumericRep, построенный по образцу Letter, а также семейство классов Complex и т. д., производных от NumericRep. В той же программе может присутствовать класс Shape, также производный от Envelope, и класс ShapeRep со своими производными

классами, образующие иерархию по образцу иерархии Letter. Хотя сообщества объектов Number и Shape существуют раздельно, они находятся в одной программе, причем каждое из них построено на основе символической идиомы.

Класс Envelope (листинг 9.2) представляет собой обобщенный класс, обеспечивающий создание экземпляра и присваивание (то есть он выполняет все «копирование» и большинство служебных операций управления памятью); часто бывает полезно привести его к полной ортодоксальной канонической форме (см. раздел 3.1), чтобы превратить в конкретный тип данных. Конверт напоминает ярлык, который можно прикреплять к разным объектам. В этом контексте присваивание сводится к установлению связи ярлыка с объектом; при снятии последнего ярлыка объект возвращается в пул.

Листинг 9.2. Класс Envelope

```
#include "k.h"        // См. выше

#include "letter.h"  // Для функций Envelope

extern Thing *envelope, *letter; // Указатели на прототипы

class Envelope: public Top {     // Top определяется в k.h
public:
    Letter *operator->() const { // Все операции
        return rep;              // перенаправляются rep
    }
    Envelope()    { rep = letter->make(); }
    Envelope(Letter&);
    ~Envelope() {
        if (rep && rep->deref() <= 0) delete rep;
    }
    Envelope(Envelope& x) {
        (rep = x.rep)->ref();
    }
    Envelope& operator=(Envelope& x) {
        if (rep != x.rep) {
            if (rep && rep->deref() <= 0) delete rep;
            (rep = x.rep)->ref();
        }
        return *this;
    }
    Thing *type() { return envelope; }
private:
    static void *operator new(size_t) {
        Sys_Error("heap Envelope");
    }
    static void operator delete(void *) { }
    Letter *rep;
};
```

Класс конверта ведет себя как абстракция с ослабленной типизацией, а его экземпляры имитируют нетипизованные ярлыки для объектов, поддерживаемые во многих символических языках. Например, функции конверта не выражают подробную семантику содержащихся в нем объектов. С другой стороны, объект конверта перенимает поведение объекта класса письма, используя механизм operator-> (см. раздел 3.5) — по аналогии с тем, как переменные в символических языках перенимают поведение тех объектов, с которыми они связываются. Тот факт, что интерфейс конверта передает некоторую информацию о своих классах писем, отражается в типе возвращаемого значения функции operator-> (а именно Letter*). Это одно из немногих мест символической идиомы, в которой указатель становится видимым для прикладного программиста, и только как временное значение, которое обычно не сохраняется пользователем.

Класс конверта содержит конструкторы, но в основном механизме инициализации объекта используются виртуальные функции make класса письма; как будет показано далее, это расширяет инкрементные возможности класса. Конструкторы конвертов являются «заготовками» для инициализации и преобразований, выполняемых компилятором. Два конструктора (конструктор по умолчанию и копирующий конструктор) входят в ортодоксальную каноническую форму. Наконец, также необходим конструктор для конструирования нового объекта конверта по экземплярам классов писем. Этот конструктор преобразует результаты вычислений, внутренних для классов писем, в объекты, которые могут использоваться обычными клиентами составных объектов «конверт/письмо».

Копирующий конструктор, оператор присваивания и деструктор работают со счетчиком ссылок объекта (см. раздел 3.5). Деструктор проверяет, равен ли счетчик нулю, и освобождает память объекта при освобождении ссылки на него последним конвертом.

Но самая важная роль отводится оператору ->, автоматически перенаправляющему вызовы функций конверта объекту письма. Того же результата можно было бы добиться таким дублированием сигнатуры письма в конверте, при котором каждая функция конверта просто передает управление своему аналогу в письме. Однако подобное решение требует удвоения усилий и включения новых функций в класс письма.

Класс Letter определяет интерфейс ко всем классам, обслуживаемым интерфейсом Envelope (листинги 9.3 и 9.4). Класс Letter сам по себе является базовым классом (обычно абстрактным) для группы классов, обслуживаемых интерфейсом Envelope. Один объект Envelope на протяжении своего жизненного цикла может предоставлять интерфейс к нескольким разным объектам писем. Например, объект Number изначально может предоставлять интерфейс к объекту Complex, но присваивание или иная операция может привести к замене исходного письма Number другим объектом, возможно, относящимся к другому классу.

Листинг 9.3. Класс Letter

```
class Letter: public Thing {
public:
    /* Здесь определяются все пользовательские операции.
     * Благодаря использованию оператора -> не обязательно
```

продолжение

Листинг 9.3 (*продолжение*)

```
     * воспроизводить эту сигнатуру в Envelope. Тем не менее,
     * в объявлении поля rep класса Envelope должен быть
     * указан соответствующий тип. Операторы присваивания
     * определяются не здесь, а в Envelope.
     *
     * Параметр возвращаемый_тип должен быть либо
     * примитивным типом, либо Envelope, либо Envelope&,
     * либо конкретным типом данных.
     */
    virtual void send(String name, String address);
    virtual double postage();
    // virtual возвращаемый_тип пользовательская_функция
    // ...
    virtual Envelope make();          // Конструктор
    virtual Envelope make(double);    // Другой конструктор
    virtual Envelope make(int days, double weight);
    virtual Thing *cutover(Thing*);  // Функция обновления
    Letter() { }
    ~Letter() { }
    Thing *type();
protected:
    friend class Envelope;
    double ounces;
    static void *operator new(size_t l) {
        return ::operator new(l);
    }
    static void operator delete(void *p) {
        ::operator delete(p);
    }
    String name, address;
    // ...
private:
    // ...
};
```

Листинг 9.4. Подставляемые функции класса Letter

```
/*
 * Здесь размещаются общие подставляемые функции.
 * Это сделано для выполнения соглашений о выводе
 * подставляемых функций из объявления класса.
 * Кроме того, размещение подставляемых функций
 * после Envelope и Letter помогает избавиться от
 * некоторых циклических зависимостей.
*/

inline double
Letter::postage() {
```

```
    if (ounces < 2) return 29.0;
    else return 29.0 + ((ounces - 1) * 23.0);
}

inline Thing *
Letter::type() {
    extern Thing *letter;     // Прототип
    return letter;
}
```

Объекты иерархии Letter «вкладываются» в объекты Envelope, а внешнему пользователю виден только объект Envelope. Пользователь никогда не обращается к сигнатуре Letter напрямую. Тем не менее, Letter добавляет свои функции в интерфейс Envelope посредством функции Envelope::operator->. Класс Letter не обязан быть конкретным типом данных, поскольку все «низкоуровневые аспекты» типа обрабатываются на уровне Envelope.

Класс Letter служит базовым для взаимосвязанных классов приложения, находящихся под управлением Envelope. Класс Envelope содержит поле rep с указателем на экземпляр Letter. Вся «настоящая» работа выполняется в объектах классов, производных от Letter.

Все функции приложения задаются в интерфейсе класса письма (обычно в виде чисто виртуальных функций). Некоторые функции, общие для всех классов, производных от класса письма, выделяются в класс письма, а их тела размещаются в определении базового класса письма. Объявление остальных функций чисто виртуальными гарантирует, что они будут определены в производных классах. Тем не менее, как показано далее, наличие чисто виртуальных функций недопустимо в расширенной форме этой идиомы, реализующей свойства объектов прототипов.

Пользовательские функции должны возвращать объекты встроенных или конкретных типов данных (иначе говоря — соответствующие ортодоксальной канонической форме), типа Envelope или ссылки на Envelope. Все типы указателей, присутствующие в сигнатуре Envelope, должны быть константными; возврат незащищенного указателя на динамически выделенный блок памяти создает угрозу для схемы управления памяти. Конечно, возвращаемые значения могут быть объявлены с типом void.

Функция make конструирует экземпляр класса, производного от Letter, и возвращает Letter* (см. главу 8). Может существовать несколько перегруженных функций make, каждая из которых выполняет всю необходимую работу по инициализации нового объекта; никакие ее части не должны оставаться на долю конструктора. Например, функция Letter::make может инициализировать объекты классов OverNight и FirstClass следующим образом:

```
Envelope
Letter::make(int days, double weight) {
    Letter *retval;
    if (days < 2 && weight <= 12) {
        retval = new OverNight;
```

```
    } else {
        retval = new FirstClass;
    }
    retval->ounces = weight;
    return Envelope(*retval);
}
```

Если конструктор не содержит сколько-нибудь значительной логики инициализации, его не придется изменять или перекомпилировать; данное обстоятельство полезно для инкрементной загрузки, поскольку виртуальные функции (такие, как make) могут подгружаться во время выполнения, а конструкторы — нет.

В общем случае производные от Letter классы не обязаны следовать ортодоксальной канонической форме. Они должны содержать конструктор по умолчанию и деструктор, но ни копирующий конструктор, ни оператор присваивания не являются обязательными.

Далее следуют классы, производные от Letter. Класс Envelope может «содержать» любые из них, и при правильном проектировании любой объект Envelope может присваиваться любому другому, причем такое присваивание прозрачно для программиста. В листинге 9.5 приведены примеры простых классов, производных от Letter. Каждый из этих классов также содержит конструктор по умолчанию.

Листинг 9.5. Специфические классы приложения, производные от Letter

```
class FirstClass : public Letter {
public:
    FirstClass();
    ~FirstClass();
    Envelope make();
    Envelope make(double weight);
    // ...
};

class OverNight : public Letter {
public:
    OverNight();
    ~OverNight();
    Envelope make();
    Envelope make(double weight);
    double postage() { return 8.00; }
    // ...
};
```

В духе идиомы прототипов каждый класс конверта содержит один глобально доступный объект, служащий его прототипом. Чтобы прототип отличался от «обычных» объектов приложения, его можно создавать специальным конструктором. Часто бывает удобно создать прототип для классов писем, чтобы прототип конверта мог ссылаться на некоторый экземпляр письма. Прототип письма перенаправляет вызовы make объекту письма оператором -> конверта. Прототип

письма может быть специальным экземпляром обобщенного базового класса письма (в данном случае Letter), если последний не является виртуальным базовым классом. В противном случае можно определить специальный производный класс письма с простыми заглушками для чисто виртуальных функций, чтобы сконструировать синглетный прототип письма.

Классы символической канонической формы используются по тому же принципу, что подсчитываемые указатели и объекты прототипов — то есть с применением оператора -> вместо оператора . (точка). Следующее простое приложение представляет собой пример использования классов Envelope и Letter:

```
static Envelope envelopeExemplar; // Никогда не используется напрямую
Envelope *envelope - &envelopeExemplar;

int main() {
    Envelope overnighter = (*envelope)->make(1, 3.0);
    overnighter->send("Addison-Wesley", "Reading, MA");
    Envelope acrosstown = (*envelope)->make(1.0);
    overnighter = acrosstown;
    acrosstown->send("Angwantibo", "Boston Common");
    return 0;
}
```

Так выглядят основные концепции символической канонической формы. Для более полного усвоения материала ниже приводится краткая сводка правил использования этой идиомы.

- Все обращения к классу Envelope должны производиться через оператор ->, а не через оператор . (точка). Это позволяет организовать автоматическое перенаправление операций классу Letter оператором ->.
- Функции классов писем должны быть виртуальными. Как будет показано в одном из следующих разделов, для виртуальных функций легко организуется инкрементная дозагрузка на стадии выполнения.
- Функция make выполняет всю работу конструктора; сам по себе конструктор почти ничего не делает. Это позволяет заменять код инициализации класса на стадии выполнения, что возможно только для виртуальных функций. Конструкторы присутствуют в классах, производных от Thing (они необходимы для работы механизма виртуальных функций C++), но они *не содержат* пользовательского кода.
- Программа должна содержать синглетный прототип для каждого класса; объект прототипа должен иметь возможность идентифицировать себя таковым (например, по факту его создания специальным конструктором). Обращения к прототипу никогда не производятся напрямую, а только через специальную переменную-указатель. Причины объясняются далее.
- Функции cutover(Thing*) субклассов Thing обеспечивают динамическую загрузку на стадии выполнения. Эти функции получают указатель на объект класса, которому они принадлежат; они должны *на месте* преобразовать старый

класс в соответствии с изменениями формата, структуры и типа нового класса. Если функция cutover не может преобразовать объект на месте, возможно, ей удастся произвести замену приемами, связанными со спецификой среды (то есть имитировать односторонний механизм BECOMES в Smalltalk). Дополнительные объяснения приводятся в следующем разделе.

- Помните, что объекты классов конвертов никогда не должны создаваться оператором new. Этот оператор объявляется закрытым, а любые попытки динамического создания объекта конверта приводят к ошибкам компиляции. Конверты должны объявляться только как автоматические переменные, члены других классов или глобальные переменные. В результате отказа от указателей программисту не придется помнить о необходимости освобождения памяти объектов; это позволяет использовать более гибкие полиморфные типы (такие как Number) в качестве конкретных типов данных вроде int.

Итак, в чем же эта идиома помогает программисту? Управление памятью в основном автоматизируется на уровне класса конверта, так как последний следит за появлением объектов, на которые не существует ни одной ссылки. А поскольку доступ к функциям и данным производится через дополнительный уровень адресации, клиент еще на один шаг изолируется от изменений, вносимых в класс. Следовательно, меньший объем программного кода зависит от подробностей реализации класса и нуждается в перекомпиляции при изменении класса. При наличии необходимых средств компоновки и загрузки эта методика даже позволяет оперативно модифицировать классы в работающей программе. Программа останавливается лишь на время, необходимое для настройки нового класса.

Описанный подход поднимает полиморфизм языка C++ на следующий уровень и создает иллюзию, будто характеристики типов могут изменяться во время выполнения программы. Далее приводятся дополнительные пояснения с примерами.

9.3. Пример обобщенного класса коллекции

Предположим, вы хотите написать программу, в которой должно быть организовано взаимозаменяемое использование трех контейнеров: массива с целочисленными индексами, B-дерева и хеш-таблицы. Для этого мы определяем класс конверта Collection, содержащий указатель на внутренний объект (см. выше). Чтобы класс Collection стал более гибким, он будет определен на базе шаблона — это позволит использовать его для хранения объектов произвольного класса. Так, следующее объявление создает коллекцию объектов Book, индексируемых по имени автора:

```
Collection<Book, Author> library;
```

Объекты писем в этом примере создаются на базе классов, производных от CollectionRep. Эти классы характеризуют варианты (см. раздел 7.7) CollectionRep. В данном примере три варианта CollectionRep (Array, Btree и HashTable) составляют набор альтернатив, используемых классом Collection в объекте контейнера.

Основные функции класса CollectionRep и его производных классов объявляются виртуальными, поэтому вызов функции Collection через указатель на CollectionRep будет перенаправлен соответствующей функции Array, Btree или HashTable. В интерфейсе класса CollectionRep должны быть объявлены операции для объединения всех функций этих классов. Из всех операций иерархии классов писем Collection знает лишь те, которые объявлены в самом классе CollectionRep. Поскольку не все производные классы писем подменяют эти функции, возможно, потребуется организовать обработку исключений для защиты от некорректных вызовов функций. Например, обращение к элементу массива может производиться только по целочисленному индексу, а хеш-таблицы могут использовать либо целые числа для индексной выборки, либо строки для ассоциативной выборки. Если объект Collection в настоящее время содержит объект Array, то при вызове функции operator[](S) для символьной строки произойдет исключение. Такова цена, которую приходится платить за гибкость символического стиля программирования.

В листинге 9.6 приведен «скелет» объявления CollectionRep — базового класса для классов писем в нашем примере.

Листинг 9.6. Базовый класс CollectionRep

```
#include "k.h"
#include "collection.h"

// Коллекция для хранения экземпляров класса T,
// индексируемых по значениям класса S

template<class T, class S>
    class CollectionRep: public Thing {
    public:
        virtual Collection<T, S> make();
        virtual Thing* cutover(Thing *);
        virtual T& operator[](int);
        virtual T& operator[](S);
        virtual void put(const T&);
        CollectionRep() { }
        ~CollectionRep() { }
        Thing *type();
    protected:
        friend class Collection<T, S>;
        static void *operator new(size_t l) {
            return ::operator new(l);
        }
        static void operator delete(void *p) {
            ::operator delete(p);
        }
    private:
        CollectionRep<T, S> *exemplarPointer;
    };
```

Классы иерархии CollectionRep реализуют функцию type несколько необычным образом. Поскольку класс CollectionRep является параметризованным, каждая специализация в новый класс требует собственного прототипа, поэтому функция type не может просто вернуть глобальный указатель. Вместо этого функция make прототипа сохраняет адрес прототипа в каждом создаваемом объекте (поле exemplarPointer), а функция type просто возвращает это значение. Пример:

```
class ClassDerivedFromCollectionRep<T, S>
    public CollectionRep<T, S> {
    ...
    Collection<T, S> make() {
        Collection<T, S> newObject;
        ...
        newObject.exemplarPointer = this;
        return newObject;
    }
    ...
};
Thing *CollectionRep::type() { return exemplarPointer; }
```

Класс CollectionRep сам по себе является полезной абстракцией и может использоваться в контексте другой идиомы. Логическая инкапсуляция иерархии CollectionRep внутри Collection имеет два преимущества. Во-первых, это позволяет классу Collection изменять свое представление во время выполнения. Динамическая типизация позволяет присвоить коллекцию одного типа коллекции другого типа, поэтому все типы коллекций становятся практически взаимозаменяемыми. Большие коллекции при достижении предельного размера (или по другим критериям) могут оперативно менять свой тип для повышения быстродействия; например, переключаться с линейного массива на ассоциативную хеш-таблицу. Во-вторых, класс Top, базовый для типа Collection, используя атрибуты класса Thing, реализует прозрачное управление памятью для идиомы прототипа. Эту концепцию иллюстрирует листинг 9.6. Механизм подсчета ссылок наследуется от базового класса Thing.

Класс Collection представлен в листинге 9.7. Он выполняет основные операции по управлению памятью (в основном при присваивании), а остальные операции перенаправляются классу письма. Класс Collection может создавать письма из набора классов по своему выбору, используя любые критерии. Для Collection можно определить дополнительные конструкторы, чтобы пользователь мог сам выбрать базовую структуру данных для коллекции.

Листинг 9.7. Класс Collection

```
#include "k.h"

template<class T, class S> class CollectionRep;

template<class T, class S>
    class Collection: public Top {
    public:
        CollectionRep<T, S> *operator->() const { return rep; }
        Collection();
```

```
        Collection(CollectionRep<T, S>&);
        ~Collection();
        Collection(Collection<T, S>&);
        Collection& operator=(Collection<T, S>&);
        T& operator[](int i) { return (*rep)[i]; }
        T& operator[](S s) { return (*rep)[s]; }
    private:
        static void *operator new(size_t) { return 0; }
        static void operator delete(void *p) {
           ::operator delete(p);
        }
        CollectionRep<T, S> *rep;
    };
```

В листинге 9.8 содержатся определения субклассов CollectionRep. Каждый объект Collection содержит письмо типа Array, Btree или HashTable. В каждом производном классе подменяются те функции, которые имеют смысл для его представления, а за остальными сохраняется стандартная реализация из CollectionRep. Поскольку производные классы избирательно подменяют небольшое подмножество функций базового класса, эти функции не могут объявляться чисто виртуальными в CollectionRep.

Листинг 9.8. Примеры классов реализации Collection

```
template<class T, class S>
    class Array: public CollectionRep<T, S> {
    public:
        Array();
        Array(Array<T, S>&);
        ~Array();
        class Collection<T, S> make();
        class Collection<T, S> make(int size);
        T& operator[](int i);
        void put(const T&);
    private:
        T *vec;
        int size;
    };

template<class T>
    struct HashTableElement {
        HashTableElement *next;
        T *element;
    };

template<class T, class S>
    class HashTable: public CollectionRep<T, S> {
    public:
        HashTable();
```

продолжение

Листинг 9.8 (*продолжение*)

```
        HashTable(HashTable<T, S>&);
        ~HashTable();
        class Collection<T, S> make();
        class Collection<T, S> make(int);
        T& operator[](int i);
        T& operator[](S);
        void put(const T&);
    private:
        int nbuckets;
        virtual int hash(int l);
        HashTableElement<T> *buckets;
    };
```

Обычно управление памятью для таких объектов организуется на базе подсчета ссылок. Как и прежде, классы из 3.5 дают хороший пример применения идиомы «конверт/письмо» для подсчета ссылок. В этом разделе показано, как оператор присваивания, конструктор и деструктор изменяют счетчик ссылок, хранящийся в переменной внутреннего объекта StringRep, и удаляют объект при уменьшении счетчика до нуля. То же самое происходит в классе CollectionRep. В разделе 9.5 будут рассмотрены другие способы уборки мусора.

Описанный подход может эффективно применяться на более высоком уровне полиморфизма, например, для объединения двух коллекций с произвольными внутренними структурами данных. Однако при этом необходимо учитывать ряд потенциальных опасностей. В частности, если классы, используемые таким образом, поддерживают бинарные операции (такие, как объединение двух коллекций оператором +), приходится учитывать возможность того, что переданные объекты относятся к разным реальным типам (например, при слиянии Array с Btree). Архитектура таких классов существенно усложняется, а выбор правильной операции для переданных объектов потребует немалых затрат на стадии выполнения (за аналогичными примерами обращайтесь к разделам 5.5 и 9.7). Теоретически необходимые преобразования можно было бы определить на уровне класса Top или Thing, добиваясь полноценного полиморфизма. Тем не менее, определения этих классов становятся очень сложными, и их придется изменять при каждом добавлении новых операций над объектами. Для определения классов, поддерживающих несколько внешних интерфейсов, можно воспользоваться множественным наследованием. Так, в примере, изображенном на рис. 9.1, класс List может проявлять свойства Array и LinkedList. Но и этот вариант слишком быстро усложняется.

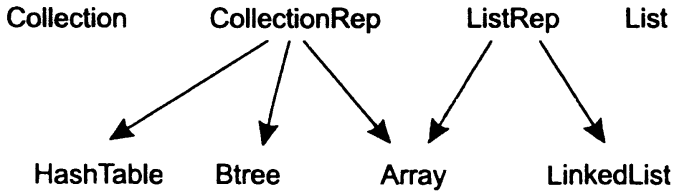


Рис. 9.1. Иерархия классов при множественном наследовании

9.4. Код и идиомы поддержки инкрементной загрузки

Если в вашем распоряжении имеется системный компоновщик с возможностью дозагрузки, часто можно написать загрузчик для включения нового кода в работающую программу. В листинге 9.9 приведена простая функция Load, которая получает имя откомпилированного объектного файла, загружает и запускает его. Допустим, объектный файл incr.o содержит единственную функцию с точкой входа, совпадающей с базовым адресом секции text объектного файла. Следующий фрагмент загружает и выполняет эту функцию во время работы программы:

```
int main() {
    typedef void (*PF) (...);  // Указатель на функцию
    PF anewfunc = (PF) load("incr.o");
    (*anewfunc)();
    return 0;
}
```

Программа работает на большинстве платформ Sun Microsystems; при ее загрузке должен использоваться флаг -n. Аналогичные программы можно написать для большинства современных операционных систем.

Листинг 9.9. Функция для загрузки файла на платформах Sun

```
#include <a.out.h>
#include <fcntl.h>
#include <sys/types.h>

caddr_t load(const char *filename) {
    char buf[64];
    caddr_t oadx = (caddr_t)sbrk(0);
    caddr_t adx = ((char*)oadx) + PAGSIZ -
        (((long)oadx) % PAGSIZ);
    sprintf(buf, "ld -N -Ttext %X -A a.out %s -o a.out.new",
        adx, filename);
    system(buf);
    int fd = open(filename, O_RDONLY);
    exec Exec;
    read(fd, (char *)&Exec, sizeof(exec));
    sbrk(PAGSIZ - (((long)oadx) % PAGSIZ));
    caddr_t ldadx = (caddr_t)sbrk(Exec.a_text +
        Exec.a_data + Exec.a_bss);
    read(fd, ldadx, Exec.a_text + Exec.a_data);
    close(fd);
    return ldadx;
}
```

Включение этого кода в интерактивную программу C++ позволяет организовать загрузку новых функций на стадии выполнения. Пока программа пребывает

в состоянии ожидания, мы можем откомпилировать новый код и подгрузить его. Далее останется связать функцию с существующим кодом, после чего работу программы можно продолжить.

ПРИМЕЧАНИЕ

Программная реализация идиом поддержки инкрементной загрузки может быть написана вручную или полуавтоматически сгенерирована средой разработки. Идиомы применяются в ситуациях, в которых необходимо иметь возможность изменять программу во время выполнения, в том числе при построении прототипов приложений и для поддержки больших приложений, работающих в непрерывном режиме.

В следующих разделах описываются идиомы и структуры данных, поддерживающие динамическую загрузку кода в работающую программу. Задача делится на две подзадачи: дозагрузку новых функций и преобразование форматов данных существующих объектов.

Загрузка виртуальных функций

Наиболее очевидным применением загрузчика является загрузка обновленных версий функций в работающую программу. Как было показано ранее, загрузить функцию несложно — необходимо лишь сгенерировать объектный файл, не содержащий ничего, кроме добавляемой функции. Впрочем, также придется связать вызовы существующей функции с новой реализацией. Именно этой теме посвящен настоящий раздел.

В разделе 9.2 упоминалась таблица указателей на виртуальные функции, или для краткости таблица виртуальных функций, ассоциированная с каждым классом. Символическая каноническая форма гарантирует, что указатель на эту таблицу является первым элементом объекта-прототипа любого класса (впрочем, это обстоятельство зависит от реализации компилятора). Вот почему все объекты конвертов объявляются производными от Top: указатель на виртуальную функцию объекта может использоваться для адресации таблицы виртуальных функций[1].

В среде C++ придется принять особые меры, гарантирующие, что программа содержит ровно одну копию таблицы виртуальных функций для каждого класса (кроме классов, созданных множественным наследованием). В одних системах компиляции C++ эта задача решается элементарно, в других требуется особая настройка. За дополнительной информацией обращайтесь к разработчикам компилятора.

Впрочем, одного указателя на таблицу виртуальных функций недостаточно: нужно знать, какой элемент таблицы соответствует обновляемой функции. Конечно, эта задача решалась бы очень просто, если бы таблица в памяти связывала имена функций с индексами элементов таблицы; в большинстве реализаций C++

[1] Решение подходит только для одиночного наследования; множественное наследование плохо обобщается и здесь не рассматривается.

это не так. Из-за этого нам придется выполнить некоторую работу *за пределами* программы, чтобы передать загрузчику всю необходимую информацию для идентификации нужного элемента таблицы.

Для этой цели можно написать небольшую *вспомогательную функцию*, единственная задача которой — вернуть составную характеристику функции, включающую адрес и индекс функции в таблице виртуальных функций. Вспомогательная функция кодируется вручную или генерируется автоматически при подготовке к обновлению функции. Процесс обновления делится на две фазы. В первой фазе вспомогательная функция загружается загрузчиком и вызывается для получения индекса элемента в таблице виртуальных функций. Во второй фазе происходит загрузка новой версии функции и связывание ее адреса с правильным элементом таблицы виртуальных функций.

Прежде чем рассматривать функции загрузки, стоит познакомиться с некоторыми вспомогательными структурами данных. Сначала определяется тип указателя на функцию:

```
typedef int (*vptp)();
```

Структура mptr определяет строение элемента таблицы виртуальных функций в большинстве существующих реализаций C++:

```
struct mptr {
    short d;
    short i;
    vptr f;
};
```

Это объявление, используемое в системах C++ на базе компилятора C++ cfront, характерно для большинства других систем. В некоторых компиляторах могут использоваться другие структуры; в этом случае приведенный код придется соответствующим образом изменить. Два поля short определяют смещения, требующиеся в основном для множественного наследования и здесь не рассматриваемые (при желании см. [1]). Для нас наибольший интерес представляет поле f, содержащее указатель на функцию для данного элемента таблицы.

Вспомогательная функция загрузки пишется легко: она просто возвращает адрес существующей версии функции, которую мы собираемся заменить. Эта функция (назовем ее functionAddress) выглядит примерно так:

```
extern vptp functionAddress() {
    // Код зависит от платформы и компилятора
    return (vptp)&Array::put;
}
```

При каждой загрузке функции сначала загружается новая копия functionAddress, которая либо заменяет предыдущую версию (с освобождением памяти), либо оставляет ее в виде несобранного «мусора», если в системе нет проблем, связанных с необходимостью максимально быстро освободить память.

Функция functionAddress способна разрешить неодозначности при загрузке функции с перегруженным именем. Предположим, класс Array содержит несколько версий функции put:

```
class Array {
public:
    ...
    void put(int, double);
    void put(int, int);
};
```

Для выбора функции, имеющей второй параметр типа double, может применяться присваивание с левосторонним значением нужного типа:

```
extern vptp functionAddress() {
    // Код зависит от платформы и компилятора
    typedef void ((Array::*TYPE)(int, double))
    TYPE retval = &Array::put;
    return (vptp)retval;
}
```

Следующая группа функций класса Top обеспечивает инкрементную загрузку и замену функций. Первая функция compareFuncs проверяет, соответствуют ли две характеристики одной и той же функции:

```
int
Top::compareFuncs(int vtbl, vptr vtblFptr, vptp fptr) {
    // Код зависит от платформы и компилятора
    return vtblIndex == (int)fptr;
}
```

Первая пара аргументов содержит информацию об элементе таблицы виртуальных функций: в int передается индекс, а в vptp — содержимое указателя на функцию (mptr::f) для данного элемента. Второй параметр определяет адрес замещаемой функции. Если по первым двум параметрам функция compareFuncs определяет, что заданная функция совпадает с той, которая определяется последним аргументом, она возвращает ненулевое значение. Вопрос о том, какие параметры реально требуются при сравнении, зависит от реализации. В нашем примере известно, что при получении адреса компилятор возвращает индекс функции в таблице, поэтому мы ограничиваемся сравнением индексов.

Вторая функция findVtblEntry определяется следующим образом:

```
mptr *
Top::findVtblEntry(vptp f) {
    // Код зависит от платформы и компилятора
    mptr **mpp = (mptr**) this;
    register mptr *vtbl = *mpp;
    for(int i = 1; vtbl[i].f; ++i ) {
        if (compareFuncs(i, vtbl[i].f, f)) {
```

```
            return vtbl + i;
        }
    }
    return 0;
}
```

Функция просто ищет в таблице виртуальных функций текущего объекта (на которую указывает первое слово объекта) элемент, соответствующий обновляемой функции и передаваемый в качестве параметра. Если поиск оказывается успешным, функция возвращает указатель на соответствующий элемент таблицы виртуальных функций (mptr).

Остается лишь загрузить новую виртуальную функцию и скомпоновать ее. Задача решается функцией Top::update:

```
extern "C" vptp load(const char *);

void
Top::update(const char *prepname, const char *loadname) {
    vptp findfunc = load(prepname);
    mptr *vtbl = findVtblEntry((*findfunc)());
    vtblEntry->f = load(filename);
}
```

При вызове функции указывается имя файла, содержащего вспомогательную функцию, и имя файла с загружаемой функцией. Используя определенные ранее функции (у функции load изменен тип, чтобы она возвращала vptr вместо caddr_t), она находит элемент таблицы виртуальных функций и заменяет текущее значение указателя указателем на новую загруженную функцию. Все последующие вызовы виртуальной функции адресуются новой версии.

Если еще немного потрудиться, программу можно адаптировать для включения новых виртуальных функций в класс (см. упражнения в конце главы). Тем не менее, организовать полноценную поддержку такой возможности с сохранением семантической корректности гораздо сложнее. Например, добавление новой виртуальной функции не ограничивается расширением таблицы виртуальных функций с сохранением порядка предыдущих элементов. Подумайте, что произойдет при добавлении новой виртуальной функции, имя которой замещает ранее существовавшую глобальную функцию: как определить, что именно было перекомпилировано и загружено?

Обновление структуры класса и функция cutover

Расширение методов оперативного обновления, предназначенных для корректного изменения структур данных, значительно повышает гибкость среды разработки и уровень поддержки программного продукта. Однако эта проблема сложнее, чем проблема обновления функций: механизм виртуальных функций содержит промежуточный уровень логической адресации, отсутствующий в данных. Тем не

менее, идиома «конверт/письмо» обеспечивает наличие такого уровня и может использоваться для организации корректной замены. В этом разделе описана методика поддержки оперативной замены данных в классах писем. Поскольку большая часть прикладного кода сосредоточена именно в классах писем, этот подход позволяет решить большинство проблем, связанных с изменением данных классов.

Обновление структуры данных класса (то есть повторная загрузка класса) фактически означает повторную загрузку его функций; не существует кода, соответствующего самой структуре класса, но информация об этой структуре распределена среди его операций. В предыдущем разделе было показано, как организовать оперативную загрузку функций, однако перезагрузка класса не сводится к простой перезагрузке всех его функций, поскольку существующие объекты также должны быть адаптированы к новой структуре класса.

Чтобы обеспечить такую возможность, каждый прототип отслеживает все создаваемые им объекты. При наличии списка созданных объектов он может преобразовать все представления данных своих классов писем при модификации их класса. Для хранения списка экземпляров удобно воспользоваться простым и универсальным классом List из библиотеки C++. Объект List объявляется статической переменной в классе прототипа.

Пользователь должен предоставить функцию cutover, которая умеет преобразовывать существующие объекты в памяти в новые версии; изменяя формат данных, функция сохраняет семантику. Эта функция, как и любая другая, может загружаться независимо. Момент вызова функции cutover после завершения определяется инфраструктурой самого приложения. Приложение может вызвать функцию cutover, когда система находится в известном, статическом состоянии, например, когда обновляемым функциям заведомо не может быть передано управление.

Существует много способов построения функций cutover. Следующие примеры дают представление о том, что вам предстоит сделать.

Добавление нового поля в класс

Предположим, в класс HashTable включается новое поле со списком блоков, содержащих дополнительные данные:

```
template <class T, class S>
    class HashTable: public CollectionRep<T, S> {
    public:
        HashTable();
        HashTable(HashTable<T, S>&);
        ...
        Thing *cutover();
    private:
        int nbuckets;
        virtual int hash(int l);
        HashTableElement<T> *buckets;
        HashTableElement<T> *overflow; // Новое поле
};
```

Обратите внимание: новые данные добавляются *в конец* класса. Это делается не случайно: довольно часто код, откомпилированный для старого интерфейса, может продолжить работу с объектами, построенными с новым интерфейсом. Кроме того, добавление данных в конец класса упрощает алгоритм перехода на новую версию класса. Также стоит заметить, что функция cutover подменяется в новом классе.

Если формат данных исходных элементов не изменяется при добавлении нового поля, реализация cutover получается весьма простой:

```
Thing *
HashTable::cutover() {
    HashTable *object = (HashTable*) this,
              *retval = new HashTable;
    // Копирование составляющей базового класса
    (*(CollectionRep *)retval) = (*(CollectionRep *)this);
    // Копирование полей класса
    retval->buckets = object->buckets;
    // Инициализация новых полей (возможно, по значениям старых):
    retval->overflow = 0;
    // Удаление старых данных
    object->buckets = 0;
    delete object;
    // Возвращение нового объекта
    return retval;
}
```

Существенные изменения в представлении класса

При значительном изменении внутреннего строения класса обновление существующих экземпляров требует нетривиальных мер. Новые экземпляры приходится строить на основе старых. Функция cutover должна иметь доступ к обеим версиям интерфейса класса, новой и старой. Нам потребуется копия объявления старого интерфейса, в которой все вхождения имени класса заменены некоторым временным именем. Функция cutover использует обе версии, поэтому она не ограничивается в способах построения нового объекта на базе старого.

Для примера возьмем класс Point:

```
class Point: public ShapeRep {
public:
    Shape make(double x, double y);
    void rotate(Shape& p);  // Поворот вокруг заданной точки
private:
    double x, y;
};
```

Требуется изменить класс Point так, чтобы он работал в круговых координатах (например, чтобы мы могли воспользоваться новым графическим устройством).

Интерфейс класса остается неизменным, поэтому преобразование существующих объектов не нарушит работу системы. Новая версия класса выглядит так:

```
class Point: public ShapeRep {
public:
    Shape make(double x, double y);
    void rotate(Shape& p);  // Поворот вокруг заданной точки
    Thing *cutover();
private:
    double radius;
    Angle theta;    // Легко строятся из double
};
```

Мы перерабатываем старый заголовочный файл и подставляем временное имя класса; получается приведенное ниже объявление. Обратите внимание: класс Point объявлен дружественным, чтобы функция cutover могла напрямую обращаться к его полям. В большинстве случаев функция cutover может получить все необходимое из открытой сигнатуры старого класса; тем не менее, после обновления старые функции могут стать недоступными!

```
class OLDPoint: public ShapeRep {
friend Point;
public:
    Shape make(double x, double y);
    void rotate(Shape& p);  // Поворот вокруг заданной точки
private:
    double x, y;
};
```

Далее мы пишем функцию cutover для нового класса:

```
Thing *
Point::cutover() {
    OLDPoint *old = (OLDPoint *)this;
    Point *newPoint = new Point;
    newPoint->radius = ::sqrt(old->x*old-> + old->y*old->y);
    if (::abs(old->x) < .000001) {
        newPoint->theta = ::atan(1) * 2;
    } else {
        newPoint->theta = ::atan2(old->y, old->x);
    }
    if (y < 0) newPoint->theta += ::atan(1) * 2;
    return newPoint;
}
```

Управление загрузкой функций и преобразованием объектов

Все управление загрузкой и преобразованием должно осуществляться самим приложением. Например, приложение должно знать, в какой момент может быть выполнено безопасное обновление (чтобы избежать попыток обновления в процессе

критических вычислений), или запросить данные у пользователя (скажем, имена файлов с объектным кодом, и т. д.). Во многих приложениях такой «управляющий код» может быть сгенерирован автоматически. В этом разделе приводятся некоторые соображения относительно архитектуры такого кода. Мы рассмотрим процесс обновления, используя класс Point в качестве примера.

После преобразования исходного текста все функции Point необходимо заново откомпилировать для нового интерфейса. Вспомогательная функция пишется для каждой виртуальной функции Point, после чего существующий прототип Point загружает новые функции в память, вызывая свою функцию Top::update. Функции загружаются по одной, причем каждая ассоциируется с парной вспомогательной функцией.

Итак, все функции загружены в память. В их число входит функция cutover, преобразующая все существующие экземпляры в новое представление. Все прототипы ведут список созданных экземпляров, поэтому задача преобразования существующих объектов классов, производных от класса письма, решается элементарно. Помните, что на концептуальном уровне классы писем находятся внутри соответствующих классов конвертов, поэтому мы знаем, что на любой объект иерархии писем может ссылаться только один класс конверта. Прототип этого класса отслеживает все свои экземпляры, он может последовательно перебрать их и проверить, какой тип письма в них содержится. Любое письмо, функция type которого возвращает указатель на прототип обновляемого письма, является кандидатом на обновление. А это означает, что конверт может последовательно выполнить операцию cutover с каждым из объектов, обновляя поле rep указателем на преобразованный экземпляр.

Предположим, мы заменяем класс Point, производный от ShapeRep. Прототип Shape содержит список всех созданных объектов Shape; он знает, что поле rep каждого из них ссылается на некоторый объект в иерархии ShapeRep. Перебирая экземпляры класса Shape, алгоритм обновления выделяет из них те экземпляры s, для которых истинно условие s.rep->type()==point (где point — указатель на прототип Point). Для каждого такого объекта s значение s.rep заменяется значением s.rep->cutover. В результате все объекты конвертов обновляются ссылками на объекты нового класса, полученные в результате преобразования старых объектов.

Остается внести еще одно изменение. Если одно письмо будет совместно использоваться несколькими конвертами, возникнет путаница между старыми и новыми версиями, и процесс преобразования нарушится. Каждое письмо, принадлежащее нескольким конвертам, должно обновляться только один раз. Для этого в каждый объект Thing включается дополнительный «теневой» счетчик. При переборе объектов мы проверяем значение этого счетчика; если оно равно нулю (значение инициализации), теневому счетчику присваивается текущее значение счетчика ссылок. Затем теневой счетчик последовательно уменьшается. Если он становится равным нулю, выполняется функция cutover; в противном случае объект пропускается. Таким образом, каждый общий объект письма преобразуется только в последнем экземпляре в процессе перебора.

Описанная логика размещается в новой функции Thing с именем docutover:

```
int
Thing::docutover() {
    if (!updateCountVal) {
        updateCountVal = refCountVal;
    }
    return !--updateCountVal;
}
```

Функция Shape::dataUpdate управляет процессом обновления данных в классах, производных от базового класса писем ShapeRep (листинг 9.10). Ее параметрами являются указатель на прототип обновляемого класса письма и указатель на прототип, который должен занять его место. После выполнения описанного алгоритма функция также обновляет объект прототипа. Полученная в результате программа отражает новую версию класса. Предполагается, что виртуальные функции новой версии класса уже загружены.

Листинг 9.10. Полный цикл обновления класса

```
typedef Thing *Thingp;

void
Shape::dataUpdate(Thingp &oldExemplar,
  const Thingp newExemplar) {
    Thing *saveRep;
    Shape *sp;
    for (Listiter<Shape*> p = allShapes;
        p.next(sp);  p++) {
        if (sp->rep->type() == &oldExemplar) {
        if (p->rep->docutover()) {
                saveRep = sp->rep;
                sp->rep = (ShapeRep*)sp->rep->cutover();
                delete saveRep;
            }
        }
    }
    saveRep = oldExemplar;
    oldExemplar = newExemplar;
    delete saveRep;
}
```

Инкрементная загрузка и автономные обобщенные конструкторы

Инкрементная загрузка эффективно работает в сочетании с автономными обобщенными конструкторами, описанными в разделе 8.3. Автономные обобщенные конструкторы позволяют специализированным прототипам (таким, как Number,

Name, Punctuation и т. д.) регистрироваться с применением более абстрактного обобщенного прототипа (Atom), называемого *автономным обобщенным прототипом*. Регистрируемые прототипы обычно являются производными от класса автономного обобщенного прототипа. Автономный обобщенный прототип служит для них агентом; набор регистрируемых прототипов может изменяться на протяжении жизненного цикла программы.

Методика инкрементной загрузки позволяет добавлять новые производные классы на протяжении жизненного цикла программы, создавать для них прототипы и регистрировать их в абстрактном базовом прототипе по мере появления. Например, мы можем включить в анализатор работающей системы классы BinaryOp и UnaryOp, создать для них объекты прототипов с использованием соответствующего конструктора и зарегистрировать в классе Atom. Далее система начинает принимать и правильно обрабатывать бинарные и унарные операторы в новых строках, передаваемых из пользовательского интерфейса, входных каналов данных и любых источников, в которых прототип Atom применяется для синтаксического анализа.

9.5. Уборка мусора

Важной особенностью символических сред программирования является то, что они избавляют программиста от хлопот с управлением памятью: объекты, на которые не осталось ни одной ссылки, автоматически уничтожаются средой времени выполнения (с помощью операционной системы, оборудования или того и другого). Этот процесс называется *уборкой мусора*. Уборка мусора создает впечатление, что объем памяти не ограничен, поэтому прикладные программисты могут просто забыть о тех объектах, которые решили свою задачу и стали лишними. Если система определяет, что объект недоступен для любых других частей программы, занимаемая им память освобождается и становится доступной для последующих запросов на выделение памяти. Механика и время освобождения объекта остаются незаметными для конечного пользователя.

Идиомы подсчета ссылок, представленные в разделе 3.5, являются упрощенной формой уборки мусора. В частности, идиома подсчета указателей (см. с. 78) обеспечивает уровень прозрачности управления памятью, присущий символическим средам. Однако алгоритмы, основанные на подсчете ссылок, не позволяют освобождать память циклических структур данных без применения механизмов рекурсивного сканирования, требующих больших затрат ресурсов. В высокоуровневых объектно-ориентированных языках применяются особые схемы уборки мусора, избавленные от этого ограничения, но в них редко используется подсчет ссылок. Идиома уборки мусора, описанная в этом разделе, представляет собой альтернативу подсчету ссылок.

Методы, не требующие подсчета ссылок, также лучше подходят для встроенных систем реального времени, в которых исключительные события (такие, как сбои памяти или процессора) могут стать причиной каскадных операций восстановления,

когда освобождение памяти затруднено. Если процесс становится неуправляемым и нуждается в аварийном завершении, возможно, у него не будет возможности выполнить свои деструкторы. Если он использует объекты с подсчетом ссылок совместно с другими процессами, то счетчики ссылок таких объектов никогда не уменьшатся до нуля, и эти объекты никогда не будут освобождены. В свете таких сбоев аудит памяти отличается большей гибкостью в отношении возврата освободившихся ресурсов, чем подсчет ссылок.

Во многих символических средах программирования детали уборки мусора скрываются в реализации компилятора и среды времени выполнения. Хотя некоторые ранние среды Smalltalk использовали алгоритмы освобождения памяти на базе подсчета ссылок, прикладные программисты писали код, в котором подсчет ссылок не применялся. Сравните с подходом C++, описанным в разделе 3.5, где управление памятью реализовано не как алгоритм, скрытый *внутри* компилятора и среды времени выполнения, а как языковая идиома. Некоторые схемы уборки мусора при помощи специализированного оборудования определяют, содержит ли слово памяти активный указатель на объект или простой блок данных. Управление памятью в символических языках, на чем бы оно ни было основано — на поддержке компилятора, средствах операционной системы или специализированном оборудовании — реализуется ниже уровня исходных текстов, написанных прикладными программистами. Термин «уборка мусора» обычно подразумевает именно такую прозрачность. Язык C++ настолько близок к оборудованию, что не существует «более низкого» уровня, на котором можно было бы реализовать абсолютно прозрачный и всеобъемлющий механизм уборки мусора. Если придерживаться простых правил написания программ, формирующих среду уборки мусора, вы сможете реализовать достаточно прозрачную уборку мусора для отдельных классов в своих программах C++.

Методика, описанная в этом разделе, прозрачна по отношению к механизму подсчета ссылок из раздела 3.5; освобождение памяти объекта в ней отделяется от «разрыва» последней ссылки на него. С другой стороны, она не обеспечивает уничтожения структур с циклическими ссылками. По быстродействию она несколько уступает методике подсчета ссылок, хотя количество сравнений зависит от особенностей использования и подробностей реализации. Разновидности представленной методики позволяют выполнять поэтапную уборку мусора, поэтому нормальную работу программы не придется надолго приостанавливать для освобождения памяти. Кроме того, в отличие от традиционной уборки мусора, этот механизм позволяет программисту выполнить некоторые действия при уничтожении объекта: уборщик мусора может вызвать деструкторы, чтобы правильно ликвидировать объект.

Для уборки мусора было разработано несколько алгоритмов. Одним из первых появился алгоритм *предварительной пометки* [2]: он анализирует объекты в памяти, проверяет, на какие объекты они ссылаются, и помечает объекты, на которые имеются ссылки. После того как все объекты обработаны, на следующем проходе непомеченные объекты уничтожаются.

Алгоритм *полупространственного копирования* гарантирует корректную ликвидацию объектов, на которые отсутствуют ссылки, а также ликвидацию групп объектов, не имеющих внешних ссылок. Прототип полупространственного копирования, алгоритм Бейкера [3], избавляется от лишних задержек, присущих алгоритму предварительной пометки, за счет дополнительных затрат памяти. В алгоритме Бейкера память делится на две половины, A и B. Половина A обычно называется *приемником*, а половина B — *источником* (причины вскоре станут ясны). Новые объекты создаются в половине A. В некоторый момент (например, при заполнении половины A или в период бездействия системы) все доступные объекты половины A перемещаются в один конец половины B. Указатели на объекты, хранящиеся в других объектах, изменяются в соответствии с изменившимся положением объектов в памяти. После завершения половина A заведомо не содержит доступных объектов, а только «мусор». Затем A и B меняются ролями, и начинается новый цикл.

Алгоритм Бейкера (как и большинство методов уборки мусора без подсчета ссылок) зависит от возможности различать в памяти данные и ссылки на объекты (указатели). Но если взять обычное слово в памяти C++, вы не сможете определить, что оно представляет — данные или указатель. Недавно появилось новое семейство алгоритмов уборки мусора, выполняющих *почти* полную уборку мусора и основанных на алгоритме предварительной пометки. Иногда эти алгоритмы не могут уничтожить мусор из-за *наложения указателей* — появления в памяти данных, содержимое которых совпадает с адресом объекта, хотя на самом деле значение представляет целочисленную величину или что-нибудь еще, совершенно не связанное с его интерпретацией в качестве адреса объекта. С другой стороны, при соблюдении программистом некоторых разумных правил эти алгоритмы не оставляют «висячих» ссылок. Примеры таких алгоритмов приведены в [4].

И все же в ограниченном мире символической идиомы, в мире конвертов и писем мы *можем* идентифицировать все объекты заданного класса конверта (просмотром списка, хранящегося в прототипе), а, следовательно, можем найти все ссылки на любой класс письма. Если изменить алгоритм выделения памяти для класса писем, чтобы он использовал фиксированный пул (см. раздел 3.6), то типы всех объектов в пуле будут известны (потому что это один и тот же тип). Также мы знаем все возможные адреса, по которым могут находиться объекты классов писем, поэтому появляется возможность отметить используемые объекты и уничтожить все остальные. Поскольку все объекты в пуле имеют одинаковый размер, исчезают проблемы с фрагментацией памяти. С другой стороны, объем памяти, занятой объектами всех классов, участвующих в уборке мусора, должен быть известен заранее.

ПРИМЕЧАНИЕ

Используйте уборку мусора, чтобы избавить пользователей от хлопот с управлением памятью, например, при ускоренной разработке пробных версий приложений. Уборка мусора также является мощным средством аудита ресурсов в системах с исключениями реального времени. Она позволяет гарантировать, что все ресурсы памяти будут освобождены даже в свете исключительных обстоятельств.

Используя класс Triangle в качестве примера, рассмотрим одну из возможных реализаций уборки мусора в символической идиоме «конверт/письмо». Для поддержки этой методики базовый класс письма ShapeRep необходимо дополнить новым битовым флагом; сброс этого бита помечает объект как доступный для алгоритма. Объектам также может понадобиться дополнительный битовый «флаг занятости», который показывает, свободна ли данная область памяти. Этот бит также может использоваться операторной функцией ShapeRep::operator new для поиска свободных областей памяти, которые можно было бы задействовать, чтобы удовлетворить запросы на создание объектов. В исходном состоянии все биты занятости и пометки сброшены; при выделении памяти под объект также устанавливается его бит занятости. Помимо этих битов каждый объект содержит биты A и B, определяющие его принадлежность к одному из полупространств A и B алгоритма Бейкера.

Пример иерархии геометрических фигур с уборкой мусора

В приложении Д представлен пример простого графического пакета, в котором применяются символические идиомы, описанные в настоящей главе. В примере демонстрируются упоминавшиеся ранее средства параллельной дозагрузки и расширенная форма уборки мусора. Методика уборки мусора не требует подсчета ссылок, а работает в духе алгоритмов Бейкера и предварительной пометки.

Алгоритм уборки мусора анализирует содержимое списка существующих объектов, хранящегося в прототипе Shape, просматривает поле rep каждого из них и проверяет, содержит ли оно указатель на объект Triangle (для чего возвращаемое значение функции type сравнивается с адресом прототипа Triangle). Для каждого найденного объекта устанавливается бит пометки. После завершения этого прохода алгоритм перебирает элементы фиксированного вектора, в котором создаются объекты Triangle, и ищет объекты со сброшенным битом пометки. После уничтожения объекта вызовом деструктора Triangle бит занятости сбрасывается. Напоследок бит пометки тоже сбрасывается для подготовки к следующему циклу уборки мусора.

Алгоритм в цикле обрабатывает все объекты Triangle, затем все объекты Line, все объекты Circle и т. д., пока не будут обработаны все классы, производные от ShapeRep, после чего все начинается заново. На более высоком уровне алгоритм последовательно применяется ко всем областям приложения (то есть к иерархии Shape, к иерархии Collection и т. д.). Планирование циклов уборки мусора должно производиться средой времени выполнения; возможно, его придется отдельно настраивать для каждого приложения. Алгоритм Бейкера даже позволяет выполнять уборку мусора поэтапно (см. упражнения в конце главы).

За подробностями реализации алгоритма обращайтесь к приложению Д. Структура классов уже описывалась, а иерархия наследования изображена на рис. 9.2. Все специализированные реализации находятся в классах, производных от ShapeRep, а пользователь имеет дело лишь с экземплярами Shape.

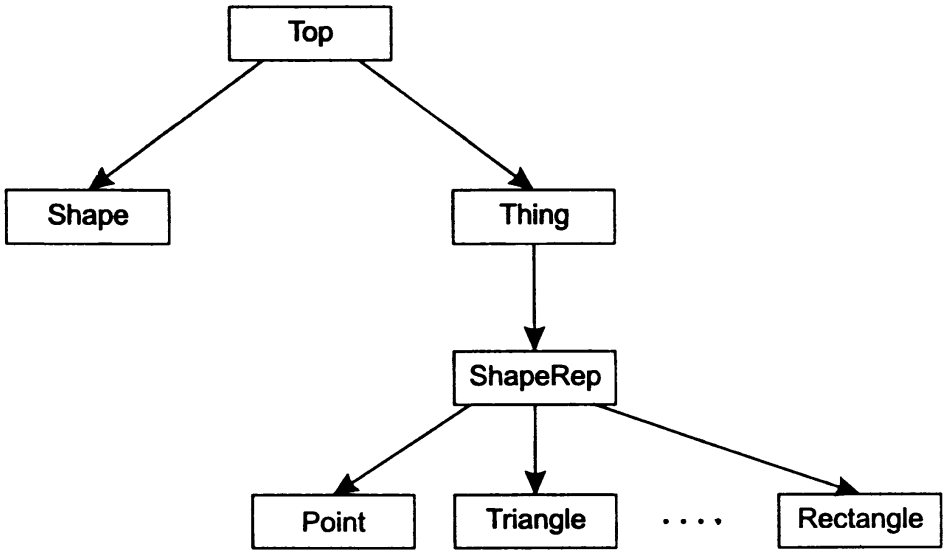


Рис. 9.2. Иерархия классов Shape в символической идиоме

Статическая функция Shape::init инициализирует некоторые глобальные структуры данных. Инициализация производится явно, без обращения к стандартным механизмам, предоставляемым средой C++, чтобы мы могли контролировать порядок инициализации. Функция Shape::init сначала инициализирует два объекта-списка. В первом списке (allShapes) отслеживаются все созданные экземпляры Shape, а во втором (allShapeExemplars) — все объекты прототипов для классов, производных от ShapeRep.

Далее Shape::init вызывает функцию init для каждого класса, производного от ShapeRep. В свою очередь, эти функции конструируют соответствующие объекты-прототипы. Каждый прототип регистрируется в Shape при помощи функции Shape::register, сохраняющей указатель на все прототипы в списке allShapeExemplars. Теперь понятно, почему списки Shape должны инициализироваться раньше прототипов. Недостаток такой схемы инициализации заключается в том, что класс Shape на стадии компиляции должен обладать информацией обо всех классах, производных от ShapeRep.

После завершения инициализации пользователь выдает запрос на создание объекта Shape, опираясь на идиому анонимного обобщенного конструктора:

```
Shape object = (*shape)->make(p1, p2, p3);
```

В результате вызова операция make прототипа Shape возвращает объект, удовлетворяющий заданным параметрам. Пользователь передает набор точек, определяющих геометрическую фигуру, а также может передать указатель на прототип, если точки не обеспечивают однозначного определения фигуры, например, две точки могут определять как прямоугольник, так и отрезок, поэтому нужен третий параметр для уточнения. По умолчанию три точки определяют треугольник, который и будет создан в результате вызова.

Обратите внимание: предыдущий вызов make также может быть выражен в виде

```
shape->operator->()->make(p1, p2, p3)
```

Прототип Shape возвращает указатель на экземпляр ShapeRep, который используется в качестве операнда для make. В идиоме «конверт/письмо» из раздела 5.5 все управление памятью (включая операции make) осуществлялось письмом. Но поскольку функция make объявлена виртуальной в базовом классе, ее новая версия может подгружаться во время работы программы с помощью описанных ранее приемов.

В результате вызывается перегруженная функция ShapeRep::make. Класс ShapeRep содержит две разновидности make. Первая вызывается с указателем на прототип для нужного типа объекта и координатами, определяющими представление объекта. Вторая версия получает только координаты и делает некое разумное предположение относительно того, какую фигуру нужно создать; для этого она просто вызывает более общую функцию make с подходящими параметрами. Так, для предыдущего вызова будет вызвана вторая версия make со следующими параметрами:

```
Shape ShapeRep::make(Coordinate pp1, Coordinate pp2,
        Coordinate pp3) {
    return make(pp1, pp2, pp3, triangle);
}
```

В свою очередь она вызывает такую функцию:

```
Shape ShapeRep::make(Coordinate pp1, Coordinate pp2,
        Coordinate pp3, Thingp type) {
    return ((ShapeRep*)type)->make(pp1, pp2, pp3);
}
```

Наконец, для построения треугольника по трем вершинам вызывается функция Triangle::make:

```
Shape Triangle::make(Coordinate pp1, Coordinate pp2,
        Coordinate pp3) {
    Triangle *retval = new Triangle;
    retval->p1 = pp1;
    retval->p2 = pp2;
    retval->p3 = pp3;
    retval->exemplarPointer = this;
    return *retval;
}
```

Функция make создает объект Triangle вызовом оператора new и заполняет этот объект данными вершин треугольника. Поле exemplarPointer (то есть поле типа) заполняется указателем на this, ссылающимся на прототип треугольника (значение this в этом контексте совпадает со значением указателя на глобальный прототип triangle). Наконец, команда return передает новый объект вызывающей стороне. Для преобразования объекта в правильный тип возвращаемого значения используется конструктор Shape(ShapeRep&).

При создании нового объекта Triangle оператором new вызывается собственный оператор new класса Triangle:

```
void *Triangle::operator new(size_t nbytes) {
    if (poolInitialized - nbytes) {
        gcCommon(nbytes, poolInitialized, PoolSize, heap);
        poolInitialized = nbytes;
    }
    Triangle *tp = (Triangle*) heap;
    while (tp->inUse) {
        tp = (Triangle*)(((char*)tp) + Round(nbytes));
    }
    tp->gcmark = 0;
    tp->inUse = 1;
    return (void*) tp;
}
```

Оператор new класса Triangle возвращает указатель на доступный блок памяти из пула объектов Triangle, размер которого соответствует размеру объекта. При первом вызове переменная poolInitialized равна нулю; она инициализируется на стадии компиляции. Если при сравнении с переменной nbytes, равной sizeof(Triangle), обнаруживается несовпадение, вызывается функция ShapeRep::gcCommon. Она используется как уборщиком мусора, так и в особых случаях — для инициализации пулов памяти в начале работы программы или при замене классов. В частности, в фазе инициализации функция gcCommon присваивает начальные значения всем флаговым битам в пуле. После возврата из gcCommon оператор new перебирает блоки пула Triangle (на который указывает Triangle::heap) в поисках блока с установленным битом занятости inUse, и возвращает первый найденный блок. Обработка возможного переполнения пула остается читателю для самостоятельного упражнения.

Все вновь созданные объекты Shape работают по одной схеме. При инициализации нового объекта Shape на базе существующего или присваивании одного объекта другому вызывается соответствующий перегруженный конструктор или оператор присваивания:

```
Shape::Shape(Shape &x) {
    Thing tp = this;
    allShapes->put(tp);
    rep = x.rep;
}

Shape& Shape::operator=(Shape &x) {
    rep = x.rep;
    return *this;
}
```

Таким образом, указатели на один объект класса, производного от ShapeRep, могут содержаться сразу в нескольких объектах Shape.

Согласно идиоме подсчета указателей (см. с. 78), объекты Shape не создаются в динамической куче; тем самым предотвращаются все проблемы с их уничтожением. Любой объект Shape, созданный как автоматический, уничтожается при выходе из области видимости. Любой объект Shape, являющийся членом класса, уничтожается при уничтожении объекта этого класса. Глобальные объекты Shape уничтожаются при завершении программы. Впрочем, мы все равно должны позаботиться об освобождении памяти и корректной ликвидации объектов классов, производных от ShapeRep. В этом нам поможет уборщик мусора.

Уборка мусора может быть инициирована в любой момент. В приложении Д она инициируется вручную в разных частях программы. Более удобный, хотя и более затратный подход заключается в вызове уборщика мусора при каждом создании нового объекта. Возможны и другие стратегии, например, можно освобождать неиспользуемую память только тогда, когда оператор new не найдет ни одного свободного элемента в пуле. В средах реального времени процесс уборки можно даже распределить во времени для поэтапного выполнения, если уборка сопряжена с длительными паузами в работе системы (см. упражнения в конце главы). Уборкой мусора управляет функция Shape::gc, которая проводит фазу пометки самостоятельно, а затем перепоручает уничтожение объектов конкретным объектам писем:

```
void Shape::gc() {
    Listiter<Topp> shapeIter = *allShapes;
    for ( Topp tp = 0; shapeIter.next(tp);  ) {
        ((Shape*)tp)->rep->mark();
    }
    Listiter<Thingp> shapeExemplarIter = *allShapeExemplars;
    for ( Thingp anExemplar = 0;
                shapeExemplarIter.next(anExemplar);  ) {
        ShapeRep *thisExemplar = (ShapeRep*)anExemplar;
        thisExemplar->gc(0);
    }
}
```

В первом цикле перебираются *все* существующие объекты Shape и устанавливается их бит пометки при помощи функции mark. Таким образом помечаются все объекты классов, производных от ShapeRep, на которые ссылаются существующие объекты Shape. Во втором цикле перебираются прототипы классов писем (по одному для каждого класса, производного от ShapeRep), которые и выполняют вторую фазу алгоритма, то есть уничтожение объектов.

Поскольку прототип каждого класса, производного от ShapeRep, поддерживает собственный пул памяти, из которого выделяется память для всех динамических объектов, каждый такой класс может легко найти и перебрать все свои динамически созданные экземпляры. В примере Shape пул представляет собой непрерывный блок памяти; возможны и другие реализации — важно лишь, чтобы прототип мог отслеживать все свои экземпляры.

Работа по уничтожению объектов выполняется статической общей функцией класса ShapeRep с именем gcCommon (листинг 9.11). Функция получает размер объекта в байтах (nbytes), размер объекта на момент последнего прохода уборки мусора (poolInitialized), количество объектов в пуле (PoolSize) и ссылку на указатель на кучу (heap). Эти параметры могут передаваться простой функцией gc производного класса, которая в свою очередь может вызываться из Shape без параметров.

Листинг 9.11. Фаза уничтожения объектов на уровне типов

```
void
ShapeRep::gcCommon(size_t nbytes, const size_t poolInitialized,
        const int PoolSize, Char_p &heap) {
    size_t s = nbytes? nbytes: poolInitialized;
    size_t Sizeof = Round(s);
    ShapeRep *tp = (ShapeRep *)heap;
    for (int i = 0; i < PoolSize; i++) {
        switch (nbytes) {
        case 0:   // Обычная уборка мусора
            if (tp->inUse) {
                if (tp->gcmark || tp->space != FromSpace) {
                    // Не уничтожать
                    tp->space = ToSpace;
                } else if (tp != tp->type()) {
                    // Объект необходимо ликвидировать
                    tp->ShapeRep::~ShapeRep();
                    tp->inUse = 0;
                    printf("ShapeRep::gcCommon ");
                    printf("Reclaimed Triangle object %c\n",
                     'A' + (((char *)tp-(char *)heap)/Sizeof));
                }
            }
            break;
        default:   // инициализация области памяти
            tp->inUse = 0;
            break;
        }
        tp->gcmark = 0;
        tp = (ShapeRep*)(Char_p(tp) + Sizeof);
    }
}
```

Как уже отмечалось, функция ShapeRep::gcCommon обеспечивает и инициализацию пула, и освобождение памяти. Логика уборки мусора сосредоточена в нулевой ветке команды switch внутри цикла. Она сохраняет объекты, которые были помечены или уже переведены в приемник (из алгоритма Бейкера). В данной реализации проверка принадлежности к полупространству не особенно важна, но если память будет освобождаться так, как показано выше, все меняется.

Объекты, непомеченные в источнике, подлежат уничтожению; их бит занятости обнуляется, в результате чего память возвращаются в пул для дальнейшего использования оператором new.

Обратите внимание: мы связали уничтожение объекта (вызов деструктора) с освобождением памяти (уборка мусора). Уничтожение объекта является семантической операций, связанной с приложением, а уборка мусора освобождает конкретный ресурс, с которым обычно возникает больше всего проблем во многих приложениях — память. Уничтожение объекта и освобождение памяти можно разделить на две операции (см. раздел 3.7), но тогда программисту придется вручную уничтожать объекты после того, как объект идентифицируется как неиспользуемый, но до того, как его память будет освобождена уборщиком мусора. Например, если объект содержит дескриптор открытого файла UNIX, одного лишь освобождения памяти будет недостаточно для возврата в систему ресурсов, связанных с этим файлом. Чтобы не перекладывать служебные операции на пользователя, мы объединили эти две функции.

Представленный подход предполагает, что корректная ликвидация объекта может быть отложена на неопределенный срок. Чтобы выполнить ее принудительно, пользователь может вызвать функцию уборки мусора сразу же после освобождения последней ссылки на объект.

9.6. Инкапсуляция примитивных типов

В большинстве символических сред экземпляры *всех* типов данных представляют собой объекты. С другой стороны, в C++ некоторые типы, особенно близкие к аппаратному уровню (int, char, long, short и т. д.), примитивнее полноценных объектов. Было бы желательно работать с этими типами в виде объектов, как в символических языках — в частности, они должны нормально уничтожаться в ходе уборки мусора.

Для каждого примитивного типа, который должен использоваться в символической среде, можно создать новый класс-«оболочку». Таким образом, мы построим библиотеку классов, моделирующих свои примитивные аналоги. Задача довольно однообразная, но на концептуальном уровне не сложная. Примером служит иерархия классов Number, описанная в разделе 5.5. После некоторых изменений в коде Number (прежде всего объявления соответствующих классов производными от Top и Thing) мы сможем применять полностью полиморфные объекты чисел, поддерживающие уборку мусора там, где обычно используются типы int, double, long и т. д.

Основные проблемы интеграции примитивных типов с символической идиомой возникают при выборе их места в иерархии наследования. Пример Number должен воплотить функциональность всех числовых типов, но зато мы перестаем различать типы double и float; char, int и unsigned int, и т. д. Как поступить со строковым классом String — выделить его в отдельный символический тип, как

предполагалось в главе 3, или отнести к ArrayedCollection, SequenceableCollection или Collection, как в иерархии Smalltalk? Как отмечено в главе 6, эти решения должны приниматься на основе анализа предметной области и потребностей приложения.

9.7. Мультиметоды в символической идиоме

Давайте попробуем представить, как программа должна интерпретировать сложение двух числовых объектов, относящихся к разным типам. При традиционном подходе система контроля типов компилятора располагает достаточной информацией, чтобы выполнить необходимое преобразование. Но при динамической типизации этот способ не подходит, поэтому в символическом программировании нужно другое решение. Допустим, каждое число с точки зрения программиста является экземпляром класса Number, а классы писем (такие, как Complex и BigInteger) остаются для него невидимыми. Объект Number изменяет характеристики типа на стадии выполнения, поэтому компилятор не располагает достаточной информацией, чтобы генерировать код преобразования. И преобразование, и выбор правильной операции сложения должны происходить во время выполнения.

Как выбрать алгоритм суммирования пары чисел? Например, можно произвольно поручить работу первому числу. Другими словами, нужная версия operator+ выбирается на основании типа первого операнда, как при использовании виртуальных функций. Но в этом случае каждый тип должен обладать полной информацией обо всех типах, которые могут с ним суммироваться. Числовые типы утрачивают свою автономность и независимость.

В идеальном случае желательно выбирать алгоритм на основании типов *обоих* операндов. Виртуальные функции C++ такой возможности не предоставляют, хотя она поддерживается некоторыми символическими языками. В объектно-ориентированных языках, интегрированных с Lisp, функция (или оператор), выбираемая на стадии выполнения в зависимости от типа нескольких операндов, называется *мультиметодом*.

Для имитации мультиметодов на C++ используются идиомы, основанные на селекторах типов (см. главу 4). Применять селекторы типов обычно нежелательно; предпочтительным механизмом выбора реализации среди нескольких производных классов считается механизм виртуальных функций. Далее показано, почему виртуальные функции в данной ситуации не подходят.

На с. 178 в разделе 5.6 приводится определение оператора класса низкочастотного фильтра LPF:

```
Value*
LPF::operator()(Value* f) {
    switch (f->type()) {
    case T_Data:
```

```
            myType = T_Data;
            cachedInput = f;
            return evaluate();
        case T_LPF:
            if (f->f1() > f1()) return this;
            else return f;
        case T_HPF:
            if (f->f1() > f1()) return new Notch(f1(), f->f1());
            else return new BPF(f->f1(), f1());
        case T_BPF:
            if (((Filter*)f)->f2() < f1()) return f;
            else return new BPF(f->f1(), f1());
        case T_Notch:
            cachedInput = f;
            return this;
        }
}
```

Там же показано, что анализ вариантов может быть преобразован в набор определений виртуальных функций. Но тогда увеличение числа типов фильтров приведет к размножению открытых виртуальных функций, что совершенно не способствует упрощению кода.

Семантика Filter::operator()(Value*) заключается в возврате значения одного из нескольких типов фильтров в зависимости от контекста. Контекст включает как тип фильтра, для которого была вызвана функция класса, *так и* тип объекта, переданного в параметре. Итак, мы имеем входной и выходной фильтры, и типы обоих фильтров определяют, какой алгоритм нужно применить и какой результат будет получен. В примере из 5.6 соответствующая логика распределена по классам, производным от Filter. В приведенном выше коде класса LPF она сосредоточена в выходных фильтрах. Если бы мы выбрали решение с виртуальными функциями, и вызов Filter::operator()(Value* input) просто обращался бы к функции входного фильтра для выполнения работы, то логика выполнения операции была бы сосредоточена во входных фильтрах.

Ни один из этих подходов не идеален. Было бы желательно обрабатывать эти запросы как гибридные операции, не являющиеся монополией одного или другого класса. Для этого требуется нечто вроде дружественных функций с глобальной перегрузкой (см. раздел 3.3):

Старое решение:

```
class Complex {
public:
    Complex operator+(Imaginary);
    operator double();
    ...
    ...
};
```

Новое решение:

```
Complex operator+(Complex c,
    Imaginary i) { ... }
class Complex {
friend Complex operator+(
    Complex, Imaginary);
public:
    operator double();
    ...
};
```

Однако здесь оператор выбирается во время компиляции на основании *объявленных* типов аргументов; не существует простого способа выбрать функцию во время выполнения в зависимости от *фактических* типов аргументов. Именно эта задача должна решаться мультиметодами.

В качестве простого примера возьмем класс Number; прежде всего нас интересует оператор +. Чтобы реализовать мультиметод operator+, мы воспользуемся коммутативностью сложения и сократим количество особых случаев вдвое. Для обращения к полю типа будет использоваться виртуальная функция Top::type. Несмотря на реализацию в виде виртуальной функции, она ведет себя как открытая переменная перечисляемого типа.

Для начала добавим в класс Number и в каждый из его производных классов новую функцию isA(const Top *const). Функция возвращает true при вызове для объекта класса A с аргументом, указывающим на объект класса B, если истинно условие «B является частным случаем A» (отношение «IS-A»). Проверка выявляет более общий из двух операндов, чтобы передать ему контроль над операцией.

Кроме того, добавим единственную функцию promote, заменяющую операторы преобразования из предыдущих примеров. Функция promote преобразует аргумент, относящийся к произвольной разновидности Number, в тип объекта, для которого она вызывается. Учитывая, что все преобразования теперь выполняются функцией promote, нам уже не придется перегружать сложение в сигнатуре каждого класса конверта; каждый класс содержит одну функцию add (имя operator+ заменяется именем add для предотвращения неоднозначности при добавлении нового оператора +). Теперь выбор в зависимости от типа параметра становится лишним, поскольку функция add каждого конверта может считать, что она всегда получает параметр типа своего класса:

```
class Number {
public:
    ...
    virtual Number *type() const;
    virtual int isA(const Number *const) const;
    virtual Number promote(const Number&) const;
    virtual Number add(const Number&) const;
    friend Number operator+(const Number&, const Number&) const;
    ...
};

class Complex: public Number {
public:
    ...
    int isA(const Number *const n) const {
        return n->type() == complexExemplar;
    }
    Number promote(const Number& n) const {
        // Всегда возвращает Complex
```

```
        if (n.type() == imaginaryExemplar) {
            return Number(0, n.magnitude());
        } else if (n.type() == integerExemplar) {
            return Number(n.magnitude(), 0);
        } else ...
        ...
    }
    // Оператор работает только с объектами Complex
    Number add(const Number&) const;
};

class Imaginary: public Complex {
public:
    ...
    int isA(const Number *const n) const {
        return n->type() == imaginaryExemplar ||
            Complex::isA(n);
    }
    // Функция promote отсутствует, поэтому
    // преобразование в Imaginary не выполняется.
    // Оператор работает только с объектами Imaginary
    Number add(const Number&) const;
};
```

Мультиметоды реализуются добавлением глобально перегруженного оператора +, применяемого компилятором к любым двум объектам Number:

```
Number operator+(const Number &n1, const Number &n2)
    throw(NumberTypeError)
{
    if (n1.isA(&n2)) {
        Number temporary = n2.promote(n1);
        return n2.add(temporary);
    } else if (n2.isA(&n1)) {
        Number temporary = n1.promote(n2);
        return n1.add(temporary);
    } else {
        throw NumberTypeError();
    }
}
```

Такое решение обладает рядом преимуществ по сравнению с представленным в разделе 5.5 на с. 148. Между функциями возникает более тесная связь: все преобразования типа локализуются, а бинарные математические операторы работают с операндами разных типов (необходимость в перегрузке операторов отпадает). Из определений функций классов исчезли команды switch.

Представленное решение имитирует принципы работы символических языков. В некоторых реализациях Smalltalk конкретные числовые классы напрямую

выполняют математические операции только в том случае, если оба операнда относятся к одному типу. Если типы не согласуются, класс Number должен выполнить необходимые преобразования и в конечном счете вызвать метод нужного субкласса. Ранее был представлен глобально перегруженный оператор:

```
operator+(const Number&. const Number&)
```

В схеме Smalltalk этот оператор выглядел бы так:

```
Number::operator+(const Number&)
```

В некоторых средах Lisp мультиметоды реализуются в виде каскадных вызовов виртуальных функций, с динамическим определением метода по типам нескольких аргументов.

Упражнения

1. Измените функцию Top::update и другие необходимые функции таким образом, чтобы в класс письма можно было добавлять *новые* виртуальные функции. Учтите, что исходная таблица виртуальных функций для класса размещается в памяти статически. Предположим, компилятор может обеспечить сортировку элементов новых функций в конце таблицы. Какие ограничения это накладывает на наследование от классов писем и как в них добавлять виртуальные функции?
2. Добавьте функцию Thing::backout(Thing*), которая возвращает класс к структуре данных и набору виртуальных функций, действовавшим до последнего обновления.
3. Измените программу в приложении Д таким образом, чтобы уборка мусора выполнялась при каждом выделении памяти под новый объект.
4. Чтобы сократить простои приложения во время уборки мусора, измените алгоритм уборки мусора из приложения Д. Любой одиночный вызов Shape::gc должен инициировать уборку мусора только в одном классе, производном от ShapeRep.
5. Чтобы уборка мусора имела еще более распределенный характер, организуйте возврат управления из Shape::gc в процессе пометки объектов. Последующие вызовы gc должны продолжать работу с того места, на котором она остановилась.
6. Повторите предыдущее упражнение так, чтобы в алгоритме Бейкера вместо фазы пометки прерывалась фаза уничтожения.
7. Объедините два предыдущих упражнения в единую стратегию.
8. Напишите систему управления памятью, в которой все классы одного размера используют один пул памяти и обрабатываются в одном цикле уборки мусора. Установления соответствия между классами и пулами должно происходить во время выполнения.

9. Алгоритм уборки мусора с разбивкой на поколения [5] поддерживает отдельные пулы памяти для объектов, классифицируемых по возрасту. Пулы с более новыми объектами обрабатываются чаще, чем пулы старых объектов, поскольку было замечено, что «молодые» объекты обычно уничтожаются с большей вероятностью, чем старые. Напишите алгоритм уборки мусора с разбивкой на поколения, заменив единый пул памяти класса тремя пулами для разных поколений. Определите подходящий «возраст» объектов (измеряемый в количестве циклов пометки/уничтожения) для каждого поколения.
10. Перепишите классы Number из 5.5, используя конструкции символической идиомы.
11. Напишите символическую версию класса String, основанную на версии String из 3.5.

Литература

1. Ellis, Margaret A., B. Stroustrup. «The Annotated C++ Reference Manual». Reading, Mass.: Addison-Wesley, 1990, sect. 10.5.
2. McCarthy, J. «Recursive Functions of Symbolic Expressions and Their Computation by Machine», «Communications of the ACM 3 (1960)», 184.
3. H. G. Baker, «List Processing in Real Time on a Serial Computer», A.I.Working Paper 139, MIT-AI Lab, Boston (April 1977).
4. Caplinger, Michael. «A Memory Allocator with Garbage Collection for C», USENIX Association Winter Conference, (February 1988), 325–30.
5. Ungar, David. «Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm», SIGPLAN Notices 19,5 (May 1984).

Глава 10
Динамическое множественное наследование

Наследование в C++ применяется прежде всего для отражения отношений между двумя сходными абстракциями приложения, одна из которых является частным случаем другой. Структуры наследования могут рассматриваться как деревья классификации. Обычно они образуют жесткие иерархии, поскольку для нашего разума естественно интерпретировать сложные системы как иерархии.

В некоторых ситуациях бывает удобно рассматривать класс как наследующий свойства нескольких родителей. Например, поведение окна, предназначенного для редактирования текста (EditWindow), можно объединить с поведением конкретной технологии многооконного интерфейса (XWin или MSWindow), в результате чего появляется новый класс, сочетающий свойства обоих родителей.

C++ поддерживает статическое множественное наследование (то есть возможность наследования поведения от нескольких базовых классов). Однако структура наследования жестко фиксируется на стадии компиляции, а во многих реальных примерах требуется, чтобы способ объединения нескольких классов для создания нового класса зависел от контекста времени выполнения. Например, если система содержит комплект разных оконных классов, мы можем выбрать нужный класс, производный от Window, на стадии выполнения, и создать его экземпляр. После анализа параметров командной строки и других имеющихся данных программа может «смешать» интерфейсы EditWindow и XWin в одном случае, или EditWindow и MSWindow в другом. Конечно, существует альтернативное решение — заранее объявить все возможные комбинации классов во время компиляции, чтобы программа выбрала нужный готовый класс для построения окна во время выполнения. Однако из-за стремительного роста количества классов, полученных в результате всех возможных комбинаций типов окон, архитектура приложения становится излишне сложной и неэлегантной.

Множественное наследование часто используется для представления объектов, играющих разные роли во время своего жизненного цикла. Рассмотрим класс SeaPlane (гидросамолет), производный от Plane (самолет) и Boat (лодка)[1];

---

[1] Спасибо Тому Карджиллу (Tom Cargill) и Эндрю Кенигу (Andrew Koenig) за этот пример.

этот пример часто приводится как обоснование множественного наследования (см. с. 169). Тем не менее полученная абстракция SeaPlane в произвольный момент времени не отражает комбинацию свойств обоих родителей; она исполняет роль самолета в воздухе и роль лодки в воде. Статическое множественное наследование не обеспечивает должного отражения этой связи. Допустим, каждый из классов Plane и Boat содержит функцию mpg, возвращающую расход топлива в милях на галлон (или каждый класс является производным от класса Vehicle, который сам по себе содержит чисто виртуальную функцию mpg). Функция mpg класса SeaPlane должна соответствовать текущей роли объекта на момент вызова и выбирать между функциями mpg классов Boat и Plane. Интуиция подсказывает, что динамическое управление наследованием дает более качественную модель.

ПРИМЕЧАНИЕ

Идиома динамического множественного наследования используется в тех ситуациях, когда решения относительно свойств типа объекта, имеющего несколько родительских классов, должны откладываться до стадии выполнения. Динамическое множественное наследование является отдаленным аналогом концепции примесей в объектно-ориентированных языках программирования, встроенных в List.

Итак, иерархии множественного наследования приносят несомненную пользу, но было бы желательно имитировать создание и модификацию таких иерархий на стадии выполнения. В этой главе будет представлен один из способов имитации динамических иерархий множественного наследования с применением особых приемов, изолируемых во внутреннем представлении классов иерархии. Ограниченный вариант динамического множественного наследования становится возможным в результате применения обычного наследования C++, виртуальных функций и идиомы «конверт/письмо».

Кстати говоря, эта методика когда-то применялась для имитации множественного наследования в ранних версиях C++ до того, как множественное наследование стало поддерживаться на уровне языка.

10.1. Пример оконной системы с выбором технологии

Пример в листинге 10.1 позаимствован из примера множественного наследования, приведенного в главе 5 (см. с. 184). Как и в исходном примере, идея заключается в том, что программа создает экземпляр класса CursesWindow или XWindow, а затем использует его так, словно он относится к классу Window. Другие программы C++ также могут задействовать универсальный интерфейс Window.

Листинг 10.1. Применение наследования для определения классов оконных технологий

```
class Window {
public:
    virtual void addch(char);
    virtual void addstr(string);
```

```
    virtual void clear();
    ...
private:
    Color curColor;
};

class CursesWindow : public Window {
public:
    void addch(char);
    void addstr(String);
    void clear();
    ...
private:
    WINDOW *cursesWindow;
};
```

Хотя одна из целей заключалась в создании обобщенного класса Window, который бы мог использоваться разными программами, также желательно адаптировать класс Window к специфике конкретного приложения, например, редактора. Класс EditWindow является частным случаем Window с атрибутами, присущими редактору: хранением номеров строк документа для первой и последней строки экрана, возможностью переноса слов и т. д. Класс EditWindow можно было бы объявить производным от Window, но он также должен проявлять свойства CursesWindow или XWindow в зависимости от используемой технологии. Как организовать такое наследование?

У этой задачи существует несколько возможных решений. Например, можно сгенерировать полный комбинаторный набор всех перечисленных классов:

```
EditCursesWindow      EditXWindow
EditWindow            CursesWindow
XWindow               Window
```

Классы генерируются либо статическим множественным наследованием, либо внедрением объектов одного класса внутрь другого класса для выполнения запросов. Решение получается уродливым и негибким по отношению к добавлению новой оконной технологии или новой функциональности окна. В другом решении наследовать вообще не нужно; вместо этого создаются отдельные классы (а следовательно, объекты) типов Window, CursesWindow, XWindow и EditWindow. При необходимости объекты обращаются друг к другу через дружественные отношения. Однако предоставление дружественного доступа считается небезопасным, и этот подход плохо инкапсулирует изменения.

Другая идея — объявить класс EditWindow производным от класса Window, а затем объявить классы CursesWindow и XWindow производными от него. Но в этом варианте пользователи Window отягощаются лишним «грузом» класса EditWindow, но класс Window приобретает ненужную специализацию.

Также можно воспользоваться идиомой «конверт/письмо». Исходное наследование CursesWindow и XWindow от Window будет сохранено, что позволит разработчикам использовать класс Window в качестве обобщенного оконного интерфейса.

Класс EditWindow содержит указатель на Window, который в действительности является указателем на XWindow или CursesWindow (листинг 10.2). Все операции Window для объекта EditWindow вызывают функции Window (виртуальные) через этот внутренний указатель.

Листинг 10.2. Промежуточная логическая адресация для реализации дополнительного уровня полиморфизма

```
class EditWindow {
public:
    void addch(char x)        { window->addch(x); }
    void addstr(string x)     { window->addstr(x); }
    void clear()              { window->clear(); }
    ...
    EditWindow()              { if (для_среды_X) {
                                    window = new XWindow;
                                    ...
                                } else {
                                    window = new CursesWindow;
                                    ...
                              } }
private:
    Window *window;
    short topLine, bottomLine;
};
```

Данный прием хорошо известен как одно из обходных решений задачи динамического множественного наследования. Впрочем, у него есть свои тонкости: класс EditWindow остается производным от Window, поэтому функции редактора, знающие только о Window, смогут работать с EditWindow:

```
class EditWindow : public Window {
    ...
};
...
Window *screen = new EditWindow;
screen->addch('A');
```

Таким образом, мы имеем дело с простой разновидностью идиомы «конверт/письмо».

Конечно, в этом случае структура иерархии определяется классом EditWindow. Добавление аргументов в конструктор EditWindow позволит создателю EditWindow передать дополнительную информацию об иерархии наследования:

```
enum wType { Apollo, Curses };

EditWindow::EditWindow(wType t) {
    switch(t) {
        case Apollo:
```

```
            window = new XWindow;
            break;
        case Curses:
            window = new CursesWindow;
            break;
    }
}
...
Window *window = new EditWindow(Curses);
```

Остается последний штрих. Наследование должно быть реализовано так, чтобы конструкторы EditWindow фактически игнорировали конструктор базового класса (Window). Если бы этот конструктор выполнялся, он мог бы создать окно на экране. Создание окна должно быть предоставлено классу EditWindow, который вызывает конструктор CursesWindow или XWindow и сохраняет полученное значение в указателе Window* в своих закрытых данных. Задача решается при помощи аргументов по умолчанию:

```
Window::Window(bool doConstructor=true) {
        if (doConstructor) {
            ...
        }
    }
    ..
    EditWindow::EditWindow() : Window(false);
    ...
    Window      *ordinaryWindow = new Window;
                // То же, что "new Window(true)"
    Window      *editorWindow = new EditWindow;
    ...
```

Если в иерархии имеются базовые классы выше уровня Window, возможно, в их конструкторы также придется добавить проверку особых случаев. Например, наследование всех классов от Class при использовании идиомы прототипов (см. главу 8) потребует особых мер. Несмотря на команду if, конструктор Class все равно будет вызван, но он, конечно, никак не помешает инициализации окон. Такие инициализации базовых классов обычно безвредны, хотя их стоит внимательно проанализировать, чтобы избежать возможных сюрпризов.

10.2. Предостережение

Представленный механизм имеет не столь общий характер, как динамическое множественное наследование в тех языках, где оно поддерживается напрямую. Он позволяет *имитировать* множественное наследование во многих интересных ситуациях, а его полезность доказана практическим опытом.

Чтобы продемонстрировать ограниченность этой имитации, давайте снова вернемся к примеру Window. Мы выяснили, что цепочка наследования объекта может задаваться при конструировании:

```
Window *window = new EditWindow(Curses);
```

Нашу схему не удастся обобщить до такой степени, чтобы стала возможной следующая конструкция (если заранее не принять специальных мер):

```
Window *window = new Curses(EditWindowType);
```

Возможно, такой порядок наследования окажется предпочтительным для обеспечения другой иерархии замещения имен (например, чтобы функция Curses замещала функцию EditWindow, а не наоборот). Но даже если эту схему удастся реализовать дополнительным программированием, со следующим примером дело обстоит еще хуже:

```
Window *window = new Window(EditWindowType);
```

Чтобы реализовать такую возможность, придется предоставить информацию о классе EditWindow клиенту, который собирался работать только с классом Window.

Также стоит заметить, что программа *может* несколько усложниться, если класс участвует в нескольких отношениях наследования. К счастью, для пользователей классов эти сложности в большинстве остаются незаметными.

Глава 11
Системные аспекты

В предыдущих главах книги мы рассматривали механизмы, правила и идиомы применения объектов и классов как основных строительных блоков системы. Как было показано в главах 5 и 7, объектно-ориентированное программирование является полезным средством многократного использования кода и абстракции; оно естественным образом выражает принципы философии проектирования, описанные в главе 6. Комбинация объектно-ориентированного программирования с другими методами способствует созданию реализаций, отличающихся удобством управления, расширения и сопровождения. Вопрос смешанного использования парадигм на уровне реализации обсуждается в приложении А.

В этой главе тема объектно-ориентированного программирования рассматривается в более общей перспективе. Объектная парадигма хорошо согласуется со структурными аспектами больших систем, но в процессе конструирования системы приходится учитывать множество других архитектурных аспектов, в том числе обработку исключений, планирование, пакетирование компонентов и управление сообществами объектов. Некоторые из этих тем будут рассмотрены позже. Поскольку настоящая глава даже вместе с главой 6 не в состоянии охватить весь спектр проблем проектирования, основное внимание в ней уделяется тем областям, в которых язык программирования не подсказывает программисту, как нужно действовать, или C++ предоставляет особые возможности для решения проблем системного уровня. Глава не содержит особо глубокого или исчерпывающего материала, а всего лишь служит отправной точкой для описания конструкций и приемов, специфических для конкретных проектов.

Системные конструкции, о которых идет речь, делятся на две категории: статические и динамические. Статическая структура включает в себя отношения, представленные на диаграммах классов в главе 6, но к ней также причисляются абстракции более высокого уровня. Различные статические структуры отражают наш подход к проектированию систем и пакетированию компонентов.

Динамическая структура включает отношения между объектами, изменение поведения системы со временем и такие традиционные вопросы, как планирование процессорного времени, параллелизм и обработку исключений. В частности, в ней отражен наш подход к устойчивому поведению системы в приложениях. Мы последовательно изучим системные аспекты с этих двух точек зрения.

Читатели, знакомые с реализациями операционных систем, могут рассматривать эту главу как «ответ объектной парадигмы операционным системам» в контексте проектирования встроенных систем. Во многих распространенных методах проектирования за единицу абстракции принимается процесс; в объектно-ориентированном проектировании такой единицей является объект. Процессы также являются единицей планирования; а что считать такой единицей в объектной парадигме? Операционная система предоставляет базовый сервис ввода-вывода, выделения ресурсов и обработки ошибок; кто должен предоставлять этот сервис в объектно-ориентированной программе? Некоторые из решений, описанных далее, могут использоваться в сочетании с традиционными системами или заменять их в приложениях, предъявляющих особые требования к выделению ресурсов, синхронизации и сервису.

11.1. Статическая системная структура

При проектировании систем приходится иметь дело с понятиями более масштабными, чем объекты. В абстрактном смысле объекты могут использоваться для отражения сущностей, находящихся на самом высоком концептуальном уровне системы: мы можем построить сколь угодно глубокую иерархию объектов, инкапсулируя группу взаимосвязанных объектов в объекте большего размера, затем связать *этот* объект с другими объектами, и т. д.

Однако при этом возникают проблемы с рекомендациями по проектированию, приведенными в главе 6. Допустим, мы проектируем программное обеспечение автопилота: в нем используются такие абстракции, как элероны, рули и двигатели, а также операции с ними. Стоит ли объединить их в объект, представляющий самолет? Обратившись к стандартным правилам проектирования, мы спрашиваем себя, можно ли связать с таким объектом четкие, интуитивно понятные операции. Какими будут эти операции? Взлет, полет в крейсерском режиме, крен, повороты? С точки зрения системы управления полетом подобные операции не являются примитивными. Каждая из них требует десятков, сотен и тысяч взаимодействий между объектами системы, которые нелегко спроектировать как алгоритм или реализовать как функцию класса. Необходимы абстракции более высокого порядка.

Из сказанного следует очень важный вывод — добавление одной функциональной возможности в систему редко соответствует добавлению одного программного компонента. Например, включение фигуры высшего пилотажа «бочка» в программу автопилота потребует внесения многочисленных изменений в реализацию текущей функциональности системы. Изменения функциональности способны нарушить *любые* парадигмы проектирования или реализации (исключения из этого правила встречаются в узкоспециализированных приложениях, использующих особые языки и библиотеки, но в данном случае речь идет о больших, сложных системах). Если структурными единицами являются процедуры, большинство запросов конечного пользователя будет нарушать их границы. Объекты

обычно хорошо справляются с инкапсуляцией изменений, поскольку в них отражают самые нерушимые абстракции с точки зрения экспертов в предметной области, но и им присуща эта проблема. Конечно, изоморфизм между структурой приложения и структурой решения способствует инкапсуляции изменений по сравнению с процедурным структурированием системы, но этот вариант не идеален — некоторые изменения неизбежно нарушают даже самые общие предположения и реализуются только посредством реструктуризации классов. Короче, объектная парадигма — не панацея. Если можно что-то гарантировать, так только то, что крупные изменения обычно приводят к крупным последствиям, а мелкие изменения — к мелким последствиям.

В этом разделе мы рассмотрим несколько механизмов абстракции, работающих выше уровня объектов. Два из них — транзакционные диаграммы и каркасы — являются статическими средствами проектирования, ориентированными на внешнюю часть процесса разработки. Модули, подсистемы и библиотеки используются для организации уже спроектированного и написанного кода (хотя конечно, они могут рассматриваться как источники входных данных при нисходящем проектировании).

Транзакционные диаграммы

Хотя объектно-ориентированное проектирование уделяет основное внимание естественным «строительным блокам» систем, оно приучает нас рассматривать некоторые системы в контексте выполняемых ими функций, даже если эти функции не реализуются в виде примитивных операций. Для примера возьмем графический редактор: в нем определены классы для представления прямоугольников, кругов и других фигур, а также линий, соединяющих фигуры. Чтобы перемещение линии приводило к перемещению соединяемых ей фигур, в редактор нужно включить команду moveline. Другие линии следуют за фигурами, которые они соединяют.

Команда moveline должна поддерживаться соответствующими структурами данных и кодом функций объектов. Но где должен находиться код функции? Перемещение линии требует активного взаимодействия между объектом Mouse, объектом Screen, объектами фигур и самим объектом линии Line. Таким образом, moveline является не столько четко определенной операцией над одним объектом, сколько результатом взаимодействия объектов, направленным на достижение общей цели. Такие сложные взаимодействия называются *транзакциями*. В [1] совокупность объектов, объединенных вместе для выполнения транзакции или набора взаимосвязанных транзакций, называется *механизмом*. Транзакция не является ответственностью одного конкретного объекта; она происходит в результате взаимодействия всех объектов механизма для выполнения системной функции.

Многие операции в больших системах относятся к категории транзакций, и их удобнее рассматривать именно как транзакции, нежели как функции классов. Например, в области телефонии такие операции, как ожидание и перенаправление звонков и даже выписка счетов, должны оформляться не как объекты или их функции, а как транзакции. Рассмотрим другой пример из области авиации.

Можно ли при анализе высокоуровневых транзакций (таких как перенаправление звонка или приземление самолета) забыть об объектно-ориентированных методах? А если интерпретировать каждую транзакцию как процедуру с пошаговым уточнением (функциональной декомпозицией)? Объекты, характеризующие системные ресурсы, по-прежнему обеспечивают наилучшую инкапсуляцию в долгосрочной перспективе. Скорее всего, отказ от объектно-ориентированных методов и формирование системной структуры на базе транзакций приведет к ее разрушению со временем. Применение процедурного подхода распределяет информацию о системе не там, где следовало бы; большинство процедур зависит от внутреннего строения структур данных, внешних по отношению к ним самим. Это усложняет понимание и эволюцию программы. При объектно-ориентированном подходе информация распределяется между транзакциями — но если проектировщик понимает, что должна делать транзакция, ему очевидно, какие объекты участвуют в ее выполнении, и где нужно внести изменения.

Для идентификации объектов и их взаимодействия в приложении применяются *транзакционные диаграммы*. Часто клиенты предпочитают работать с проектировщиком на уровне деловых транзакций, слишком сложных для оформления в виде функций класса. Тем не менее, именно деловые транзакции в конечном счете реализуются системой при взаимодействии объектов через функции классов, и проектировщик должен воспроизвести это соответствие, отражая представления пользователя о функциональности системы.

Транзакционные диаграммы описывают функциональность механизма в целом. Механизмом может быть как вся *система*, так и одна из *подсистем*. В данном случае термины «система» и «подсистема» обозначают части приложения, автономные и доставляемые пользователю как независимое целое. Далее мы еще вернемся к подсистемам.

Каждый столбец транзакционной диаграммы описывает транзакцию (или механизм), а каждая строка соответствует объекту. Порядок столбцов и строк имеет значение: логически связанные функции и абстракции должны находиться в смежных позициях. Левый край может рассматриваться как логически прилегающий к правому краю; аналогичным образом связываются верх и низ. Перемещение слева направо часто соответствует перемещению по последовательным транзакциям в жизненном цикле системы; в частности, это относится к диаграмме на рис. 11.1[1].

Ячейки таблицы представляют функции, выполняемые объектом в ходе транзакции. В таблице нас интересуют закономерности: группы сходных функций, применяемых к взаимосвязанным объектам во взаимосвязанных транзакциях. Такие закономерности свидетельствуют о наличии сходных элементов в структуре объектов, которые могут быть вынесены в общие базовые классы. Например, можно заметить, что набор высоты сопровождается подъемом обоих рулей высоты; этот факт наводит на мысль, что между ними имеется сходство. Классы рулей высоты можно объявить производными от одного базового класса `Elevator` и включить в них функцию `climb` для выполнения действий, необходимых для набора высоты.

[1] Большое спасибо Нейлу Холлеру (Neil Haller) и Сьюзен Скотт (Suzan Scott), которые помогли приблизить этот пример к реальности.

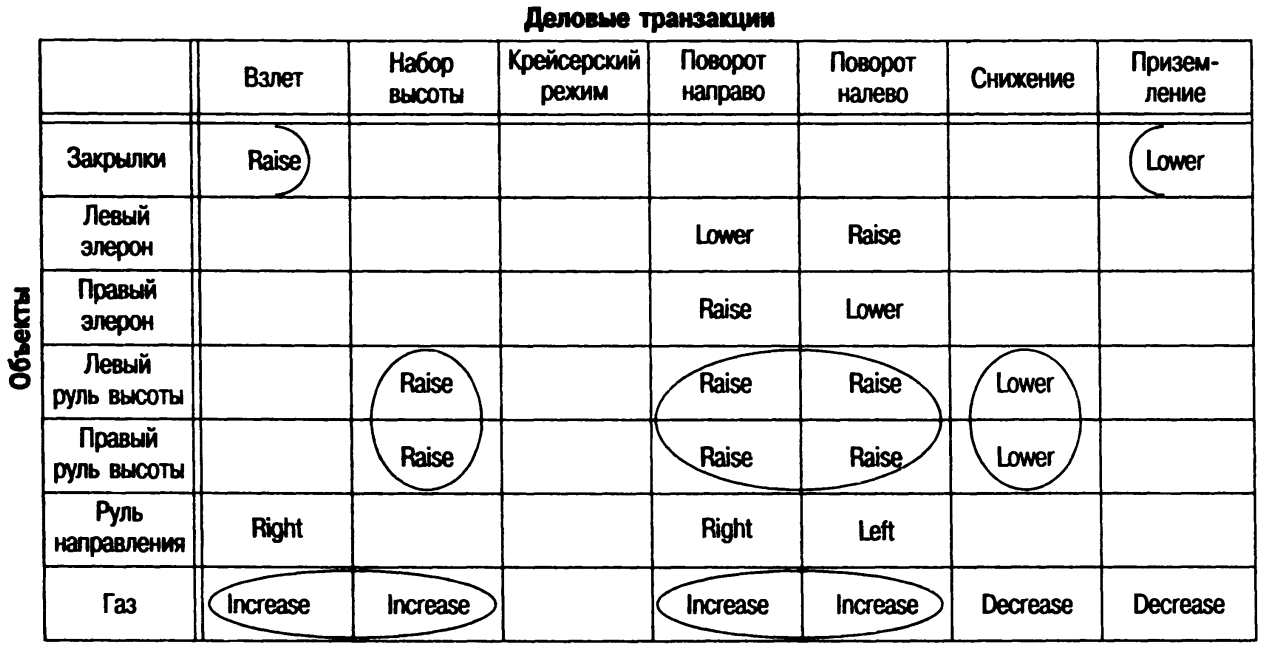


Рис. 11.1. Транзакционная диаграмма для программы управления самолетом

Также нетрудно заметить, что при обоих поворотах (налево и направо) происходит прибавление газа. Можно предположить, что для обоих поворотов должна вызываться некоторая глобальная функция, логика выполнения которой является общей для обеих транзакций. Эта функция не принадлежит одному конкретному объекту.

Если взглянуть на вещи еще шире, можно найти еще более удачное решение. Поскольку левый и правый рули высоты всегда ведут себя одинаково, с точки зрения архитектуры их можно рассматривать как единое целое. Каждый руль высоты представлен отдельным объектом общего класса, а упоминавшийся ранее класс Elevator, выполнявший *некоторые* общие операции рулей высоты, может выполнять их все. Две строки рулей высоты можно свернуть в одну строку.

Некоторые функции (такие, как функция right руля направления, компенсирующая вращение винта при взлете) группировке не подлежат. Их следует напрямую реализовать в виде функций классов, а не объединять с другими функциями в транзакции.

Следует подчеркнуть, что транзакционные диаграммы имеют неформальный характер — это всего лишь инструмент для выделения закономерностей и сходных черт объектов. По принципу применения они напоминают *карты Карно* — формальное средство, используемое в схемотехнике.

Модули

Структурной единицей программы является модуль. C++ поддерживает модули в той же степени, что и C — а именно за счет разделения кода на отдельные исходные файлы. Модуль может содержать как один класс, так и несколько классов; кроме того, он может экспортировать другие типы и константы.

Модуль состоит из двух частей: символических имен, экспортируемых им для использования клиентами, и закрытой реализации. Экспортируемая часть модуля находится в заголовочном файле, обычно имеющем расширение .h или .hpp, а закрытая часть — в файле с исходным текстом программы, или просто в исходном файле (с расширением .c или .cpp). Некоторые аспекты закрытой реализации также отражаются в заголовочном файле.

Заголовочный файл состоит в основном из объявлений, обеспечивающих нормальную работу системы контроля типов и удобство записи. Как правило, в заголовочные файлы выносятся объявления классов, используемых в модуле, и объявления глобальных переменных, определяемых внутри модуля, но доступных для его клиентов. В заголовочные файлы также выносятся другие сопутствующие определения типов, включая перечисляемые. Некоторые определения удобнее размещать в заголовочных файлах, особенно определения символических констант (как глобальных, так и действующих внутри класса).

Исходный файл содержит определения функций классов модуля, глобальных функций и глобальных объектов.

Содержимое исходного файла должно оставаться невидимым для клиентов модуля. Чтобы получить доступ к интерфейсам модуля, клиент включает заголовочный файл директивой #include. При этом возникает дилемма с подстановкой функций: подставляемые функции лучше всего размещать в отдельном исходном файле с ключевым словом inline. Этот файл включается в конец заголовочного файла модуля, если функции должны подставляться, или компилируется отдельно, если подстановка не нужна. Например, для класса Stack файл Stack.h может выглядеть так:

```
#ifndef _STACK_H
#define _STACK_H
class Stack {
public:
    int pop();
    ...
};

#ifndef inline
#include "StackInlines.c"
#endif
#endif
```

Примерный вид файла StackInlines.c:

```
#ifndef _STACKINLINES_C
#define _STACKINLINES_C
#include "Stack.h"
inline int Stack::pop() {
    ...
}
#endif
```

Наконец, файл Stack.c, содержащий не подставляемые функции Stack, выглядит так:

```
#include "Stack.h"
#ifdef inline
#include "StackInlines.c"
#endif

Stack::Stack() {
    ...
```

Если теперь при компиляции системы определить макрос inline как пустую строку, каждая функция Stack будет сгенерирована в одном экземпляре (вместо множественных расширений на месте):

```
CC -Dinline="" *.c
```

Подсистемы

Подсистемой называется набор модулей, образующих независимый логический узел. Функциональность подсистемы должна быть достаточно широкой и в то же время достаточно общей, чтобы такая группировка приносила пользу. Например, на проектирование и разработку подсистемы одна компания может заключить с другой компанией контракт.

Некоторые абстракции, воплощаемые в типах на ранней стадии проектирования системы, могут стать хорошими кандидатами для преобразования в подсистемы на стадии реализации. Хороший проектировщик расширяет масштаб своих абстракций для удовлетворения большинства потребностей приложения в заданной предметной области; в процессе этого расширения он старается идентифицировать модули и группы модулей, которые могут многократно использоваться в других приложениях (см. 6.3). Такие абстракции стоит выделить в подсистему, потратить время на их документирование для применения, не связанного с текущим проектом, и организовать их хранение в архивах готового кода.

Подсистема часто реализует *механизм* — совокупность классов, обеспечивающих высокоуровневые функции приложения, выходящих за рамки функций классов. В качестве примеров подсистем можно привести системы администрирования баз данных и сетевые интерфейсы.

Классы могут использоваться для группировки функций в статические функции класса; полученная абстракция ведет себя как подсистема. Такие функции обычно связаны друг с другом слабее, чем обычные (нестатические) функции, и не нуждаются в полиморфной интерпретации. Подсистема, как правило, не соответствует отдельному ресурсу, идентифицируемому как тип или сущность в объектно-ориентированном анализе. Скорее, она представляет синглетную сущность: в любую систему некоторая подсистема входит только один раз. Обычно подсистема имеет более широкий концептуальный масштаб, обладает более

разнородными внутренними состояниями и меньшей семантической связностью, чем объект. Простым примером может послужить интерфейс операционной системы:

```
class OperatingSystemInterface {
public:
    static int reboot();
    ...
    class Process {
        ...
    };
};
```

Тем не менее, то, что считается подсистемой на одном уровне, на другом уровне может оказаться объектом, а статическая функция класса при переходе на другой уровень может стать функцией экземпляра (то есть нестатической). Выбор представления для абстракции (объект или подсистема) зависит в основном от его логической связности с системной точки зрения, а также интуитивной связи с предметной областью. Если системный анализ показывает, что такой ресурс должен быть объектом, а не подсистемой, избегайте использования статических функций.

Класс, содержащий статические функции, можно сравнить с пакетом языка Ada. Помимо статических функций он может экспортировать другие классы, перечисления и константы.

Каркасы

Каркас определяет подсистему или механизм с возможностью настройки или расширения. *Архитектура* подсистемы инкапсулируется в наборе классов, тогда как ее *реализация* может быть задана лишь частично. Абстрактные базовые классы-заготовки оставляют определение некоторых функций классов приложениям. В качестве примеров объектно-ориентированных каркасов можно привести X Toolkit, MacApp и архитектуру MVC (Model-View-Controller — модель, представление, контроллер) в Smalltalk.

Каркас предоставляет вспомогательные функции, обычно ассоциируемые с операционной системой. В отличие от большинства операционных систем, каркас предоставляет услуги, ориентированные на конкретную область применения. Он выходит за рамки простого сервиса операционных систем и характеризует основные абстракции приложения. Пользователь каркаса предоставляет код, дополняющий незаполненные аспекты абстракций для построения системы.

Одним из механизмов параметризации программ для конкретного применения являются шаблоны C++. Каркасы обеспечивают более широкую и общую абстракцию, которую не удается удобно выразить при помощи шаблонов. Вы можете определять обобщенные каркасы в виде наборов абстрактных базовых классов,

которые определяют «контур» архитектуры и предоставляют пользователю возможность подстроить семантику под конкретное приложение. При создании каркасов часто применяется одна полезная идиома: открытые функции абстракций базовых классов почти не конкретизируются, а подробности реализации выносятся в виртуальные функции, подменяемые пользователем в производных классах. В результате каркас складывается из базовых классов, содержащих минимальное число открытых функций (листинг 11.1). Бóльшая часть открытых операций выполняется на обобщенном уровне в базовом классе. Функции базовых классов обеспечивают логику вызова специфических функций, определяемых в виде закрытых виртуальных функций в производных классах.

Листинг 11.1. Каркас для форм

```
class TaxFormFramework {
public:
    TaxFormFramework(int numberOfLines) {
        ..
    }
    void draw() {
        makeField(loc1, taxpayer.name());
        makeField(loc2, taxpayer.ssn());
        ...
        addFields();
    }
private:
    void makeField(Point, const char *const);
    // Производные классы должны содержать
    // собственную версию addFields
    virtual void addFields() = 0;
    Point loc1, loc2, ...
    ..
};

class ScheduleB: public TaxFormFramework {
public:
    ScheduleB(): TaxFormFramework(14) { }
private:
    Point locA, ...
    void addFields() {
        // Добавление строк для ScheduleB
        makeField(locA, mortgage.income());
        makeField(locB, other.income());
        ...
    }
    ...
};
```

Каркасы часто пакетируются в библиотеки для последующего распространения.

Библиотеки

Библиотекой называется упорядоченный набор программных компонентов, из которых приложение выбирает необходимые для работы компоненты. Библиотека отчасти сходна с подсистемой, но в большей степени является единицей пакетирования, нежели единицей структуризации или абстракции.

Библиотека обычно состоит из двух частей: объектного кода и набора заголовочных файлов. Пакетирование производится таким образом, чтобы пользовать мог работать с частью абстракций библиотеки без издержек, обусловленных неиспользуемыми частями.

Каждая библиотека должна строиться на базе определенной темы, подобно тому, как генеалогическая библиотека состоит из книг и других материалов по истории рода, а сельскохозяйственная библиотека содержит информацию о применении биологических наук для производства потребительских товаров. Как и подсистема, библиотека является самостоятельным продуктом, который может приобретаться отдельно. Но как и в случае с обычной библиотекой, для решения текущей задачи не обязательно использовать всю библиотеку полностью.

У каждой библиотеки имеется определенный набор правил или предположений, распространяющихся на все ее абстракции. Единые методы выделения памяти, стратегии обработки исключений и расширения классов (на базе шаблонов или производных классов) упрощают понимание и практическое использование библиотек. В идеальном случае все библиотеки, задействованные в проекте, проявляют подобную согласованность, но на практике этого добиться нелегко. Чтобы существующие библиотеки соответствовали локальным интерфейсным стандартам, для них можно построить набор простых интерфейсных классов. Согласование методов выделения памяти создает больше проблем, и многие библиотеки, не ориентированные на конкретные приложения, задействуют только средства выделения и освобождения памяти, описанные в спецификации языка. Если пользователь пожелает применить более экзотические средства (например, методику уборки мусора, описанную в главе 9), ему придется построить дополнительные классы на базе библиотечных классов. К счастью, большинство подобных преобразований выполняется без редактирования и перекомпиляции исходных файлов библиотек.

Будьте особенно внимательны при создании зависимостей между библиотеками. Преобразования между взаимосвязанными библиотечными классами должны осуществляться с учетом правил и рекомендаций, представленных в разделе 3.4 (см. с. 69). При создании библиотек шаблонов классов необходимо особо позаботиться о предотвращении дублирования кода (некоторые способы сокращения избыточности описаны в главе 7 на с. 261).

В соответствии с этими рекомендациями библиотеки могут использоваться для пакетирования кода отдельных конкретных типов данных, наборов базовых классов или целых каркасов. Продолжая этот логический ряд, они также подходят для пакетирования и распространения кода подсистем. Однако подсистема

обычно включается в приложение в виде монолитного кода, поэтому библиотеки подсистем редко позволяют избирательно применять компоненты.

Большинство рекомендаций по построению библиотек может рассматриваться как следствия принципов многократного использования кода, поэтому в этой области действуют все рекомендации из главы 7. Хороший стиль программирования и организации программного кода упрощает управление библиотеками; некоторые примеры приведены в [1, 2] .

11.2. Динамическая системная структура

Большинство методов проектирования ориентируется на статическую системную структуру. Динамические характеристики системы тоже учитываются, но полноценный анализ системной динамики требует обратной связи с прототипом программы или ранними вариантами реализации. Среди динамических аспектов следует выделить взаимодействие между отдельными экземплярами на стадии выполнения, соответствие дисциплин планирования системным ограничениям реального времени, а также распределенной обработки. Традиционно эти аспекты относились к проблематике операционных систем, но их стоит рассмотреть заново в свете объектно-ориентированных принципов.

Планирование

Планирование является одним из важнейших факторов проектирования, поскольку наш подход к модульности программ во многом определяется представлениями о процессах как единицах планирования. В языке C++ отсутствует прямая поддержка планирования. Впрочем, это вовсе не означает, что вопросы планирования чужды сообществу C++; изначально язык C++ создавался как язык с имитацией событий по образцу Simula 67 и для поддержки диспетчеризации задействовал библиотеку задач. Во многих современных средах C++ продолжают использоваться потомки этой библиотеки.

Прежде чем рассматривать тему планирования, необходимо сначала разобраться со смыслом некоторых терминов. Термины «процесс», «мультиобработка» и «параллельная обработка» встречаются часто, поэтому стоит разобраться, что они означают применительно к объектам. В этой главе мы будем понимать их следующим образом.

- *Процесс.* Основная единица *планирования*, которая также может быть единицей защиты, адресного пространства или пространства имен. *Структурными* единицами планирования обычно являются *программы.*
- *Мультиобработка.* Специальная методика, при которой основной единицей планирования считается процесс. Мультиобработка может быть реализована на одном процессоре, при этом каждый процесс как бы работает на отдельном виртуальном процессоре. Однопроцессорная среда мультиобработки также

называется средой *с разделением времени*. В таких средах полноценная асинхронность процессов отсутствует, хотя имитация асинхронности создает иллюзию параллелизма.

- *Параллельная обработка*. В параллельной (или распределенной) обработке задействовано несколько процессоров, или, говоря точнее, несколько параллельных программных потоков. Многие принципы мультиобработки применимы и к параллельной обработке, но при этом приходится дополнительно учитывать *полноценную асинхронность*, недостижимую в однопроцессорной системе.

Приложения существуют в невероятно широком спектре характеристик реального времени и планирования. Не существует единственно правильного механизма планирования, который подошел бы всем приложениям сразу. В языке было бы очень трудно реализовать поддержку мультиобработки, достаточно общую для многих приложений, а узкоспециализированные модели планирования часто бывает трудно применить в конкретном приложении. Например, в одних приложениях может потребоваться мультиобработка без параллелизма (скажем, при имитации автономных сущностей), а в других — передача управления с приоритетным прерыванием. Поддержка механизмов распространения тоже должна учитывать специфику конкретной ситуации. В одних средах параллелизм реализуется в низкоуровневых сетевых протоколах, в других он не поддерживается вообще, а в третьих используется реализация на базе общей памяти. Такое разнообразие усложняет языковую поддержку и даже разработку идиоматических конструкций, которые бы обеспечивали единую, достаточно общую модель параллелизма. В конкретных проектах либо усовершенствуется существующая схема, либо создаются совершенно новые подходы. Впрочем, языковая поддержка параллелизма все же существует; один из примеров приведен в [4].

В главе 6 было отмечено, что один из важнейших принципов проектирования, заложенных в основу объектной парадигмы, заключается в создании параллелей между структурой предметной области и структурой классов/объектов решения; мы назвали это свойство «принципом структурного изоморфизма». «Структура» в обеих областях характеризуется набором взаимосвязанных аспектов поведения. Существует ли аналогичный базовый принцип для планирования?

Практический опыт (возможно, с долей самоанализа) подсказывает ответ — возможно. Если модель вычислений является неотъемлемой и четко идентифицируемой частью характеристик приложения, то в духе принципа структурного изоморфизма на вопрос можно ответить положительно. Для примера возьмем архитектуру объектно-ориентированной базы данных; объекты существуют в общей среде, где с ними работают несколько процессоров. Объекты являются единицами синхронизации, а в механизме планирования должны быть задействованы высокоуровневые средства, обеспечивающие целостность данных. Это хороший пример прямой связи параллелизма с решаемой задачей; параллелизм непосредственно учитывается в постановке задачи.

С другой стороны, в некоторых ситуациях задача может не быть параллельной по своей сути, но параллелизм логично вписывается в ее решение. Хорошим примером является пакет графического воспроизведения компьютерной анимации или абстракция телефонного звонка в системе управления коммутируемой сетью: хотя «звонок» является единой концептуальной сущностью, одна его часть может обрабатываться одним процессором, находящимся вблизи от точки вызова, а другие части — другими процессорами, связанными с вызываемыми сторонами. В этом случае необходимость в распределении не только поднимает тему планирования, но и влияет на структуру решения.

В большинстве приложений совершенно неважно, какая схема планирования использована в их реализации, или на скольких процессорах они работают — одном или нескольких. Планирование обычно относится к механизму реализации, а не к характеристикам самого приложения. Одни приложения параллельны по своей сущности; для других устанавливаются жесткие ограничения производительности, вынуждающие применять распределенную обработку. Взаимодействие с существующими программами или использование специализированных процессоров, обусловленное спецификой приложения (сигнальные и математические процессоры, графические ускорители), также могут ограничивать решение некоторыми формами планирования. Как правило, различия между спецификациями области решения и ограничениями предметной области имеют искусственный характер; ограничения предметной области неявно присутствуют в структуре задачи.

Может ли каждый объект в объектно-ориентированной архитектуре рассматриваться как процесс? Во многих средах мультиобработки затраты не переключение контекстов и передачу сообщений слишком велики, чтобы это стало возможным. Подумайте, чем обернется переключение контекста или отправка сообщения операционной системы объекту строки при поиске очередного символа (кстати говоря, то, что в Smalltalk называется сообщениями, имеет весьма отдаленное отношение к планированию процессов и не подразумевает никакой асинхронности). Более того, в большинстве приложений такое соответствие не оправдано с точки зрения семантики. Мультиобработка существует для того, чтобы создать иллюзию выполнения каждого процесса на отдельном процессоре; нет необходимости создавать такую иллюзию между строкой и ее клиентом.

С другой стороны, если в программе потребуется абстракция для процесса, можно ли оформить ее в виде объекта Process? Все зависит от того, удастся ли спроектировать правильные, семантически четкие функции для такой абстракции. Если вы собираетесь использовать «процессную» сторону этих объектов, то ответ будет положительным. Рассмотрим сильно распределенную среду с сотней процессов. Если процессы должны часто вступать в вычисления и выходить из них, тогда интерпретация единиц планирования как архитектурных компонентов становится оправданной. Такие объекты могут быть производными от базового класса Process. Они называются *актерами*, потому что каждый из них, словно артист в пьесе, играет свою роль независимо от других.

Как упоминалось ранее, язык C++ не поддерживает актеров или других дисциплин планирования; тем не менее, приложения C++ могут работать на базе сред, поддерживающих такие модели [5, 6]. Модель актеров близко воспроизводится в распространенной библиотеке задач C++, входящей в стандартную поставку многих сред C++. Далее приводится краткое описание, за более подробной информацией обращайтесь к [7].

Актеры объединяют в единую абстракцию единицы структурирования и планирования. С одной стороны, их интерпретация в виде двух раздельных абстракций, не объединенных формальными связями, накладывает меньше ограничений. Одним из недостатков архитектур, в основу которых закладываются процессы, является тесная связь единиц структурирования, адресного пространства и планирования. С другой стороны, структура планирования может не соответствовать структуре объектов; выполнение происходит в конечных *программных потоках* (см. далее).

Задачи C++

В популярной библиотеке C++ в качестве единиц планирования используются *задачи*, а общая модель достаточно близка к модели актеров. Объекты задач служат как единицами планирования, так и структурными единицами. Кроме того, они обеспечивают разбиение пространства имен — каждая задача имеет собственную область видимости. Пакет задач C++ изначально проектировался для имитации дискретных событий, но он может применяться для любых схем, требующих невытесняющего планирования.

Ниже перечислены классы библиотеки, которые предоставляют поддержку многозадачности.

`task`

Объекты классов, производных от `task`, представляют активные ресурсы системы. Каждый объект ведет себя так, словно он обладает собственным счетчиком команд и работает на отдельном процессоре. Конструктор объекта `task` является аналогом функции `main`. Как и процесс, объект задачи обладает собственным набором данных и отдельно планируется. Задача может временно приостановить свое выполнение, заблокировав статус ресурса `object` (см. далее) или вызывая функцию `delay(n)` для передачи управления процессору на `n` квантов.

`object`

Класс `object` служит базовым классом для представления пассивных ресурсов системы. Такие ресурсы находятся либо в состоянии *готовности*, либо в состоянии *ожидания*, и объект `task` может заблокировать свое выполнение до изменения статуса одного или нескольких экземпляров `object`. Если задача ожидает изменения статуса нескольких ресурсов, она возобновляет выполнение при переходе любого из ожидаемых ресурсов в состояние готовности.

`queue`

Взаимодействие задач организуется через очереди общего назначения. У очереди имеются атрибуты начала и конца, представленные соответственно классами `qhead` и `qtail`. Оба эти класса являются производными от `object`, поэтому задача может блокироваться в ожидании поступления данных в начало очереди.

Хотя объекты task могут напрямую вызывать функции друг друга, такие вызовы осуществляются синхронно, что не соответствует семантике задач. В парадигме актеров взаимодействие между объектами обычно реализуется через интерфейсы сообщений, а получатель анализирует содержимое сообщения во время выполнения, чтобы интерпретировать его семантику. Интерфейс одного объекта задачи недоступен напрямую для других объектов задач, поэтому сигнатуры классов не фигурируют в архитектуре системы, основанной на задачах. Виртуальные функции для объектов task практически бессмысленны: полиморфизм основан на интерпретации (пользовательской) содержимого сообщения во время выполнения.

Библиотека задач предоставляет в распоряжение программиста ограниченные средства синхронизации. Во-первых, полноценной синхронизации быть не может, потому что все задачи работают на одном процессоре в одном программном потоке. Во-вторых (что более существенно), синхронизация операций пользовательского уровня осуществляется при помощи механизмов блокировки в объектах object и queue; подробности реализации скрываются от пользователя.

В систему, использующую объекты задач, можно добавить схему приоритетной диспетчеризации. Например, запросы могут планироваться на основании приоритетов, ассоциируемых либо с запросами, либо с очередями, их содержащими. Приоритеты особенно полезны в системах с интенсивным вводом-выводом, где периферийным операциям назначается высокий приоритет для параллельного выполнения с текущими вычислениями.

ПРИМЕЧАНИЕ

Объекты задач полезны для имитации дискретных событий и в других приложениях для однопроцессорных систем, в которых требуется имитировать параллелизм. Используйте эту идиому, когда единицы планирования образуют абстракции, соответствующие логике приложения, а не являющиеся артефактами решения.

Программные потоки

Программные потоки формируют для объектов естественный механизм планирования и являются альтернативой модели актеров. Если актеры объединяют единицы планирования и структуры, то программные потоки позволяют разделить их. Проектировщику, представления которого о планировании базируются исключительно на модели процессов, потоки могут показаться чем-то необычным — так представление о классах как об абстракциях кажется необычным тому, чей опыт проектирования ограничивается методами функциональной декомпозиции. Программные потоки обладают рядом преимуществ для систем реального времени, использующих объектно-ориентированные методы.

На рис. 11.2 показано, как осуществляется планирование потоков. Программный поток запускается по *запросу*. Предположим, запрос с пометкой 1 поступил от сообщения, сгенерированного где-то снаружи. Доставку всех запросов обеспечивает объект-диспетчер, связывающий типы запросов с объектами, которые должны их получить, и вызываемыми функциями. Эта информация входит в большинство запросов и предоставляется объектом-отправителем; в этом смысле запросы напоминают сообщения или элементы очередей библиотеки task.

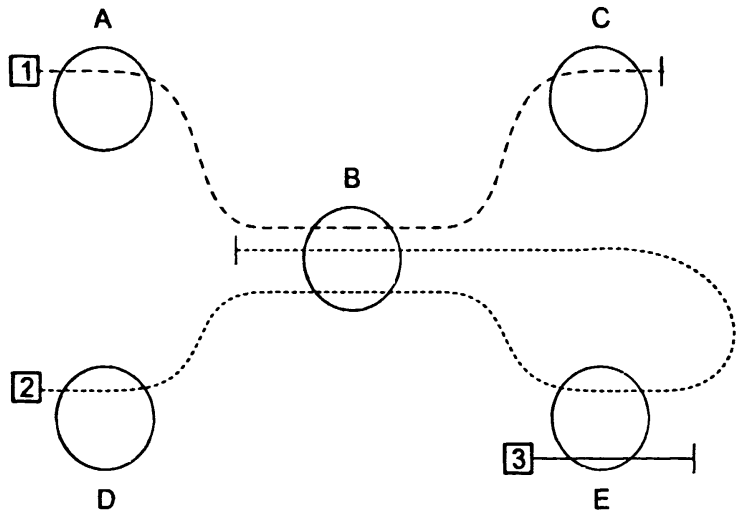


Рис. 11.2. Планирование программных потоков для функций объектов

Диспетчер отправляет запрос 1 функции объекта A, которая начинает выполняться. Эта функция вызывает функцию объекта B, которая в свою очередь вызывает функцию объекта C. Функция объекта C выполняет свою работу и возвращает управление. В конечном счете управление будет возвращено диспетчеру сразу же за точкой передачи управления A, и на этом работа потока завершается. Поток выполняется *без прерывания* от начала до конца; его код не блокируется в ожидании ресурса и не уступает процессор вплоть до завершения. Чтобы гарантировать завершение потока в течение заданного промежутка времени, можно воспользоваться системным таймером. Диспетчер запускает таймер в начале работы потока и останавливает его при завершении. Если таймер срабатывает в процессе выполнения потока, система решает, что текущий поток вышел из-под контроля, и инициирует операцию восстановления.

В ходе выполнения поток может генерировать другие запросы. Когда при завершении потока управление возвращается диспетчеру, последний запускает новый поток по запросу, сгенерированному предыдущим потоком. При отсутствии запросов диспетчер остается в пассивном состоянии до тех пор, пока поток не будет снова запущен по внешнему запросу. В ходе выполнения потока 1 может быть сгенерирован запрос 2, который запустит собственный программный поток.

Благодаря отсутствию явной блокировки программные потоки существенно упрощают синхронизацию. Между потоками объект не находится в «частично обработанном» состоянии из-за того, что функция блокируется в ожидании ресурса.

Планирование программных потоков естественно обобщается для распределенных многопроцессорных сред. Любой поток работает на своем процессоре, а все межпроцессорные взаимодействия осуществляются при передаче запросов между средами времени выполнения процессоров.

Управление потоками может осуществляться на базе системы приоритетов — тогда диспетчер выбирает порядок обработки запросов на основании типа запроса и других подходящих критериев.

ПРИМЕЧАНИЕ

Используйте программные потоки в приложениях реального времени, когда время отклика на происходящие события особенно важно. Потоки также хорошо подходят для распределенных архитектур, в которых связывание объектов с процессорами производится на поздней стадии проектирования (или является частью реализации).

Контексты

Большинство нетривиальных проблем в программировании обычно сводится к именам и тому, что за ними стоит. В объектно-ориентированных архитектурах имена многих классов и их функций являются глобальными, как и манипуляторы многих системных объектов. В системах с десятками или сотнями классов верхнего уровня довольно сложно найти нужный объект, оценить последствия от внесения изменений и организовать управление ресурсами на стадии выполнения.

Если проанализировать абстракции статической системной структуры — классы, модули, подсистемы, каркасы, библиотеки (и в меньшей степени транзакции), — обнаружится, что каждая из них на интуитивно понятном уровне связана с некоторой совокупностью исходного кода программы. *Контекст* описывает отношения между существующими, работающими объектами, что означает существенное ослабление связи со статическими абстракциями, а следовательно — и со структурой исходного кода. Очень важно, чтобы программист воспринимал такие абстракции как административные единицы времени выполнения по тем же причинам, по которым процессы считаются важными абстракциями во многих распространенных методах проектирования.

Сущности в большинстве приложений группируются в сообщества; другими словами, большинство объектов плотно взаимодействует лишь с небольшим подмножеством других объектов. Такие сообщества часто могут рассматриваться как локальные группы, которые инициализируются и администрируются независимо от других групп. Одним из примеров такой группировки являются иерархии объектов — инкапсуляция экземпляров классов внутри других объектов. В других случаях группировка объектов отражает абстракции, которые сами по себе не могут быть нормально представленными в виде объектов, но и иерархии объектов не могут нормально отразить связи между ними. Например, классы Mouse, Menu, Keyboard, Screen и Window явно связаны между собой, но их было бы трудно объединить в рамках одной абстракции класса (хотя классы Screen, Keyboard и Mouse можно сгруппировать в класс Terminal). До определенной степени такие сообщества являются подсистемами. Однако один объект может принадлежать *нескольким* сообществам; предположим, объект Window принадлежит сообществу объектов, реализующих графический редактор, но при этом он также может принадлежать диспетчеру окон. Объекты Telephone и Call могут представлять конкретную географическую точку; тем не менее, любой телефон может принадлежать одновременно двум таким сообществам, если он использует режим ожидания для задержки звонка на время разговора с другим абонентом. Ни «большие объекты», ни подсистемы не способны выразить такие отношения сколь-нибудь

удовлетворительно. Для обозначения таких перекрывающихся сообществ мы будем использовать термин *контекст*.

У каждого контекста имеется свой *серверный объект*, который можно рассматривать как крошечный аналог операционной системы; объекты общаются друг с другом через сервер, и сервер доступен только из обслуживаемого им контекста. Сервер поддерживает рудиментарный набор функций, как правило — службу имен и некоторые базовые средства управления памятью. Функция службы имен ассоциирует символические имена с объектами; символические имена могут представлять собой строки или обычные константы перечисляемого типа. В результате объекты внутри контекста становятся доступными друг для друга без загромождения пространства имен всей программы. Обычно объект в момент создания регистрируется у одного или нескольких серверов.

Контекст, как и традиционные процессы, определяет пространство имен. Другие классические функции процессов — такие, как управление памятью и устранение сбоев — тоже могут вписываться в контексты (вероятно, планирование все же должно осуществляться на более высоком уровне, который бы теснее ассоциировался с процессами). Сервер также может выделять память для всех объектов, создаваемых для контекста или создаваемых из кода в работающем контексте.

Взаимодействие между пространствами имен

Распределенная среда должна обеспечивать общую поддержку взаимодействия между объектами в разных процессах и на разных процессорах. Если распределенные вычисления интегрируются с механизмами, заложенными в основу взаимодействий между объектами внутри процесса, то поддержка распределенности становится прозрачной для программиста. Затраты на переключение контекстов желательно свести к минимуму, для чего производится оптимизация удаленных взаимодействий — вместо использования единого механизма диспетчеризации с постоянными, хотя и высокими затратами. В этом разделе рассматриваются идиомы C++, обеспечивающие прозрачную распределенность, а также баланс между универсальностью и эффективностью.

Снова о процессах и объектах

Независимо от того, что представляют главные абстракции архитектуры — процессы или объекты — эти абстракции должны взаимодействовать друг с другом. Передача управления от одной абстракции другой (добровольная или принудительная) называется *переключением контекста*.

В архитектуре процессов в переключении контекста часто задействуются сообщения. Если передача данных важнее передачи управления, вместо сообщений можно воспользоваться общей памятью и избавиться от затрат на формирование сообщений. В обоих случаях операционная система должна сохранять состояние процессов при передаче управления между процессами. Это обычно та же информация, которая сохраняется при вызове одной функции из другой — счетчик

команд и содержимое регистров. Операционная система также должна активизировать другой процесс, то есть восстановить его состояние и передать управление туда, где оно было прервано или должно быть возобновлено в результате поступления запроса или изменения состояния (скажем, получения сигнала). Все взаимодействия между процессами сопряжены с затратами на минимум одно переключение контекста, а возможно — и на создание сообщения.

Если основными архитектурными единицами являются не процессы, а объекты, они тоже должны обладать механизмом переключения контекста. Для обычных объектов C++ таким механизмом является простой вызов функции. Передача управления осуществляется напрямую, без таблиц отображения. Стековый протокол вызова функции сохраняет контекст вызывающей стороны, который автоматически восстанавливается при возвращении. Вместо передачи в сообщениях аргументы заносятся в стек. Все взаимодействия между процессами требуют стандартных затрат, как и на вызов функции.

У обоих подходов есть свои достоинства и недостатки. Передача управления между процессами может осуществляться в произвольном порядке, что упрощает реализацию приоритетных схем диспетчеризации, не зависящих от порядка поступления сообщений. Распределить процессы между процессорами проще, чем объекты; механизм вызова функций, используемый для взаимодействия между объектами, не расширяется на многопроцессорные конфигурации. За универсальность приходится платить, и затраты на переключение контекстов и форматирование сообщений могут оказаться весьма высокими.

В объектно-ориентированном мире желательно вызывать функции напрямую, и нести затраты, связанные с использованием сообщений, лишь при необходимости (для распределенной обработки). Кроме того, желательно, чтобы распределенный характер среды был как можно более прозрачным. О том, как это сделать, рассказано в следующем разделе.

Прозрачность

По мере развития системы количество процессоров может увеличиваться, а объекты будут переноситься с одного процессора на другой. Изменения в технологии или в требованиях задачи могут привести к необходимости перераспределения объектов между процессами или процессорами. Такие изменения возникают не только в ходе долгосрочного сопровождения, но и во время исходной реализации, когда проектировщик рассматривает и опробует различные варианты. Используя программные потоки и идиому «манипулятор/тело», можно организовать распределение объектов между процессорами с минимальными хлопотами.

Любое взаимодействие между парой объектов состоит из двух фаз: запроса и передачи. Каждый объект состоит из двух частей: *манипулятора* и *тела* (см. раздел 3.5). Чтобы обеспечить прозрачность распределенного характера среды, объект-манипулятор перехватывает сообщения и передает их своему телу. На одном процессоре манипулятор представляет собой простой класс конверта, а тело — класс письма; эти два класса объединяются при помощи указателя (рис. 11.3).

Любой экземпляр тела существует на одном процессоре, но он может иметь *представителей* на нескольких процессорах, включая тот, на котором работает тело. Манипулятор, находящийся на одном процессоре с телом, обращается к телу напрямую как к классу письма (как в однопроцессорной системе). Удаленный доступ к объекту, как обычно, осуществляется через класс-манипулятор, но этот класс передает запросы не телу, а представителю.

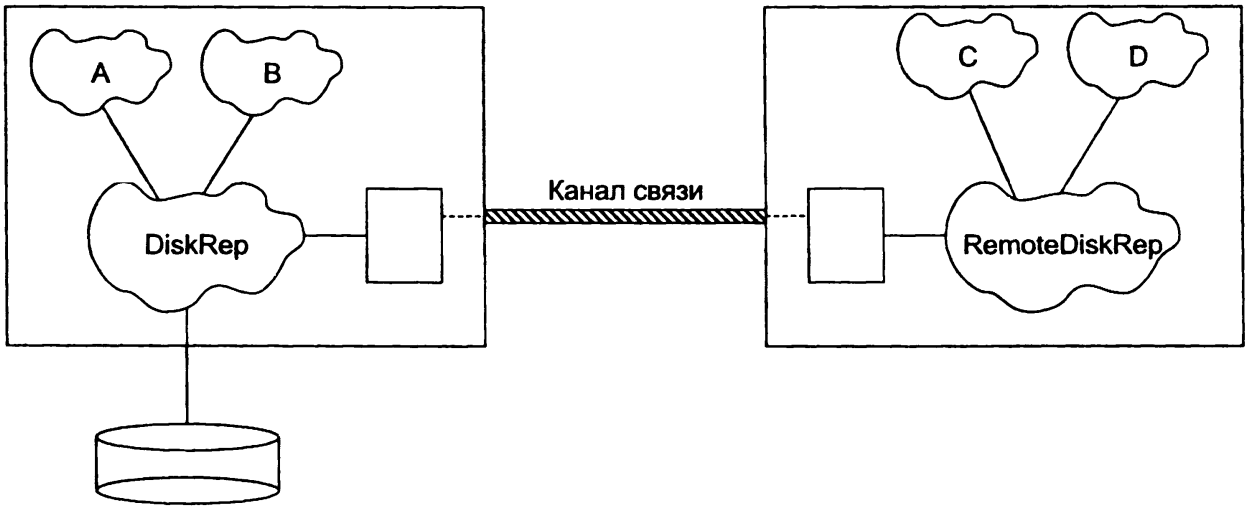


Рис. 11.3. Классы-представители

Представитель пакетирует аргументы в сообщение и производит его прозрачную пересылку «настоящему» классу тела во внешнем пространстве имен. Например, представитель `RemoteDiskRep` пакетирует запросы `read` и `write` от объектов `C` и `D` и пересылает их объекту `DiskRep` на другом процессоре. Диспетчер операционной системы направляет сообщение в канал связи для передачи удаленному процессору. Затем диспетчер запускает новый поток на передающем конце. На приемном конце операционная система доставляет сообщение объекту тела `DiskRep`. Доставка может осуществляться специальной функцией, которая декодирует сообщение и выполняет соответствующую операцию с `DiskRep`.

Учтите, что организовать распределенное выполнение функций с возвращаемыми значениями нелегко, поскольку межпроцессорный вызов требует, чтобы между вызовом и возвратом проходила пауза в реальном времени. В среде планирования потоков не предусмотрен готовый механизм сохранения адреса возврата и других динамических данных потока (автоматических переменных) во время таких пауз. Важная особенность программных потоков состоит в том, что бóльшая часть информации о состоянии системы хранится не в стеке, а в данных объекта, что может упростить анализ ошибок и восстановление.

Если возврат значений абсолютно необходим, его можно реализовать путем обратного вызова. Генерируя запрос на обслуживание, инициатор запроса задает функцию обратного вызова в виде указателя на функцию, или функтора. Когда объект тела `DiskRep` завершает свою работу, он отправляет сообщение процессору, от которого исходил вызов. Сообщение доставляется объекту-представителю, который вызывает функцию обратного вызова. Если объект класса манипулятора

способен удовлетворить запрос при помощи объекта тела на том же процессоре (например, A), то тело может вызвать функцию обратного вызова сразу же после завершения обработки запроса. Если тело может удовлетворить запрос без паузы в реальном времени, то обратный вызов инициируется в том же программном потоке, что и исходный запрос. Если запрос удовлетворяется в удаленном режиме или с паузой в реальном времени, обратный вызов инициируется в другом потоке. Если не считать продолжительности обработки, этот механизм прозрачен с точки зрения клиента, обращающегося с запросом на обслуживание к A, B, C или D.

Обработка исключений

Исключением называется событие, инициированное в результате возникновения аномальной ситуации и требующее особого внимания. Примерами исключений служат деление на ноль, прерывания (например, сигнал о поступлении данных от периферийного устройства) и многочисленные программные ошибки (скажем, попытка вызова несуществующей виртуальной функции). Обработка исключений определяется спецификой приложения, и ни одно общее решение не подойдет для всех случаев. В этом разделе рассматриваются некоторые типичные способы обработки исключений, а также приводятся рекомендации по проектированию отказоустойчивых программ.

В таких программах, как редакторы, компиляторы, игры и т. д., при возникновении ошибки может быть достаточно освободить ресурсы (закрыть открытые файлы, освободить семафоры, изменить счетчики ссылок общих ресурсов и т. д.), вывести сообщение и завершить работу. Подобная «зачистка» обычно выполняется автоматически в деструкторах объектов, поэтому она выполняется при стандартном уничтожении объектов в ходе обработки исключения.

При выборе модели обработки исключений необходимо ответить на важный вопрос: что должно считаться единицей восстановления? В традиционных миникомпьютерных средах (и многих других) единицей восстановления является процесс, а исключения приводят к завершению или перезапуску процессов. Поскольку для объектов более естественной дисциплиной планирования являются не процессы, а актеры или потоки, придется искать новую абстракцию восстановления. В одной из моделей обработки исключений единицами восстановления являются функции (функции классов, глобальные или статические функции). Такой механизм обработки исключений предлагается в качестве стандартного для языка C++ [8].

Так как обработка исключений еще не стала устоявшейся частью языка, а база для создания идиом еще не сформирована, в этой главе данная тема рассматривается крайне поверхностно. Обычно механизмы и конструкции обработки исключений зависят от приложения, но общие познания в этой области подскажут вам некоторые полезные идеи, которые могут стать отправными точками для дальнейших исследований.

При обработке исключений на базе функций основные проблемы возникают с уничтожением локальных переменных функций. Если исключение возникло при многократно вложенном вызове, необходимо уничтожить объекты для всех функций, для которых в стеке были созданы контексты вызова. Сначала нужно дать возможность восстановиться выполняемой функции, затем функции, которая вызвала ее, и т. д. — вплоть до верхнего уровня, чтобы ошибка не распространялась дальше.

В схеме обработки исключений C++ все фрагменты кода, для которых должна активизироваться обработка исключений, выполняются в блоках try:

```
void foo() {
    try {
        // Вызовы, находящиеся в блоке try,
        // участвуют в обработке исключений.
    }
    // А вызовы, находящиеся вне блока, в
    // обработке исключений не участвуют
    // (если только функция foo не была
    // вызвана из другого блока try).
}
```

Исключения генерируются средой времени выполнения (например, при делении на ноль) или в результате выполнения команды throw с выражением, определяющим тип инициируемого исключения. Исключения, сгенерированные в блоке try или в функциях, вызванных из блока try, передают исключения обработчику. Обработчик исключения программируется в виде секции catch. В заголовке секции объявляется тип обрабатываемого исключения, а тело секции содержит команды, выполняемые при передаче управления обработчику. Простой пример:

```
void foo() {
    try {
        bar();
    }
    catch (const char *message) {
        // Получает указатель на параметр throw
    }
    catch (const List<String> &message) {
        // Получает временную копию параметра throw
    }
}

void bar() throw(List<String>, const char*) {
    List<String> errorReport;
    ...
    throw errorReport;   // Вызывается первая секция catch
    ...
    throw "Help!";       // Вызывается вторая секция catch
    ...
}
```

Обратите внимание: в интерфейсе функции bar указано, какие исключения она генерирует. Механизм обработки исключений не предусматривает продолжения выполнения с той точки, в которой произошло исключение.

Описанная схема хорошо подходит для уничтожения локальных объектов функций, активных на момент возникновения исключения, а также для простых контейнерных классов, строк и других классов, входящих в библиотеки общего назначения. Но обработка исключений C++ не решает проблемы уничтожения объектов, созданных в куче. Кроме того, она не справляется с асинхронными или каскадными ошибками; эти проблемы решаются в области обработки сигналов.

На практике сложные отказоустойчивые системы нуждаются в более глобальной стратегии восстановления. Во-первых, они должны справляться с событиями, которые обычно не ассоциируются с абстракциями языков программирования, например, ошибками контроля четности памяти и сбоями устройств. Такие ошибки асинхронны по отношению к «основному» программному потоку, а процедура восстановления не связана с тем, что делает программа в момент их возникновения. Такая отказоустойчивость связана с событиями, происходящими «за кулисами», и делится на *субпарадигматическое восстановление* (для низкоуровневых прерываний вроде сбоев четности) и *суперпарадигматическое восстановление* (восстановление на уровне контекста, процесса или процессора). Эмпирическое правило гласит, что восстановление в системах, работающих в непрерывном режиме, должно осуществляться на самом нижнем из доступных технологическом уровне.

Во-вторых, если механизм try/throw/catch восстанавливает вложение вызовов функций (то есть смежные контексты вызовов в стеке), в некоторых ситуациях нужно, чтобы единицей восстановления был объект. Функции обработки исключений могут отложить повторную инициализацию на более высокие уровни. Поврежденный объект восстанавливается вызовом конструктора для существующего объекта (см. 3.7) или другим подходящим способом.

В-третьих, обработка таких ошибок системного уровня, как нехватка памяти, должна производиться на общесистемном уровне. Схема восстановления может освободить наименее критические ресурсы, чтобы обеспечить возможность функционирования системы в целом. Некоторые ресурсы можно вернуть, используя контрольные точки состояния системы. Для разных систем приходится задействовать разные методы восстановления — как и в случае с планированием, язык программирования не может предложить единого решения, подходящего для всех случаев.

Наконец, некоторые ошибки перехватываются не сразу после их возникновения. Если ошибка обнаруживается в результате проявления эффектов второго, третьего или четвертого порядка на каких-нибудь данных, расположенных далеко от точки возникновения ошибки, у обработчика исключений не будет контекста для решения проблемы. Восстановление придется передать на более глобальный уровень. Обработка исключений является одним из механизмов такой передачи,

но это не единственный механизм, а его эффективное применение в системах непрерывного режима работы требует глобальной стратегии восстановления.

На момент написания книги обработка исключений поддерживается не всеми средами C++. Некоторые суррогатные решения описаны в [9].

Уборка мусора как средство повышения надежности

Освобождение неиспользуемой памяти в сочетании с автоматизацией вызова деструкторов недоступных объектов является важной частью восстановления. Идиомы для реализации этих методов описаны в главе 9. Если система считает, что объект утратил логическую целостность, было бы сомнительно доверять деструктору освобождение всех его ресурсов. Обычно безопаснее просто возвращать память таких объектов в пул; вызов деструкторов может вызвать каскад ошибок из-за появления недействительных или неопределенных указателей. Но в этом случае возникает опасность появления объектов, на которые не существует ни одной ссылки — именно эту проблему решают механизмы уборки мусора. Скорее всего, субпарадигматическим обработчикам не удастся выполнить действия, «положенные» для объектно-ориентированной парадигмы, поэтому они тоже могут просто освободить память объектов без их деинициализации. Скажем, процесс в системе с разделением времени, аварийно завершившийся из-за деления на ноль, может не закрывать все свои открытые файлы.

Механизм уборки мусора выполняет три операции, способствующие восстановлению.

- Освобождение памяти, связанной с недоступными объектами (то есть объектами, на которые не существует ссылок).
- Если механизм уборки мусора вызывает деструкторы для недоступных объектов, помимо памяти освобождаются другие ресурсы (файловые дескрипторы, регистры виртуальной памяти, семафоры и т. д.).
- Если механизм уборки мусора вызывает деструкторы, то объекты, ссылки на которые присутствуют только в недоступных объектах, тоже освобождаются, и т. д. — теоретически это может приводить к каскадному освобождению целых ветвей ресурсов.

Если такие ресурсы не освобождаются, это может привести к постепенному истощению системных ресурсов и полному краху системы. Но, как и прежде, эту опасность следует сравнить с опасностью каскадных ошибок от вызова деструкторов для объектов, утративших логическую целостность.

Уборка мусора обходится недешево, а ее реализация бывает утомительной и однообразной. Тем не менее, она вполне подходит для кода C++, созданного генераторами приложений, а также может использоваться при ручном кодировании в системах с повышенными требованиями к надежности.

Контекст как единица восстановления

При использовании программных потоков концепция процесса перестает существовать, поэтому вы не сможете завершить и перезапустить процесс в рамках

восстановительных мероприятий. В то же время контексты управляют своими ресурсами и могут рассматриваться в качестве единиц восстановления. Например, можно обратиться к серверному объекту с запросом на повторную инициализацию всех локальных объектов контекста, возможно — с особой обработкой объектов, общих для нескольких контекстов.

Зомби

Даже в самых надежных схемах восстановления возможны непредвиденные события. Что произойдет при попытке обращения к исчезнувшему объекту? Обычно начинается полный хаос с каскадными ошибками, и отследить первопричину происходящего бывает нелегко. Проблему можно решить заменой уничтоженных объектов *объектами-зомби* с управляемым, отказоустойчивым поведением.

Строение объекта-зомби зависит от среды C++, в которой вы работаете. В большинстве случаев зомби конструируются как списки указателей на таблицу виртуальных функций. Объект не содержит ничего, кроме этих указателей, причем все указатели ссылаются на одну и ту же таблицу. Эта таблица должна быть, по крайней мере, не меньше самой большой таблицы виртуальных функций в системе, и в ней хранятся указатели на функцию восстановления. Объект-зомби может находиться на месте любого другого объекта и направлять все вызовы своих виртуальных функций на функцию восстановления.

Объекты-зомби могут создаваться на месте удаляемых объектов системным оператором delete или перегруженными операторами delete отдельных классов.

Функция восстановления, на которую ссылается таблица виртуальных функций, не делает ничего (пассивная операция), генерирует исключение (чтобы вызывающая сторона знала о попытке разыменования «висячего» указателя) или предпринимает необходимые действия, чтобы система восстановления интерпретировала текущий объект как недостоверный. Выбор зависит от системной политики восстановления.

С вызовами невиртуальных функций классов через «висячие» указатели дело обстоит сложнее. В общем случае предотвратить их довольно сложно.

Литература

1. Booch, Grady. «Object-Oriented Design with Applications». Redwood City, Calif.: Benjamin/Cummings, 1991.
2. Coggins, James M. and Gregory Bollela. «Managing C++ Libraries». SIGPLAN Notices 24, 6 (June 1989), 37.
3. Stroustrup, B. «The C++ Programming Language, 2nd ed.» Reading, Mass.: Addison-Wesley, 1991, ch.12.
4. Gehani, N., and W. D. Roome. «Concurrent C Programming Language», Summit, New Jersey: Silicon Press, 1989.

5. Agha, Gul A. «Actors: a model of concurrent computation in distributed systems». Cambridge, Mass.: MIT Press, 1986.
6. Kafura, Dennis, and Keung Hae Lee. «ACT++: Building a Concurrent C++ with Actors», TR89-18, Blacksburg, Va.: Virginia Polytechnic Institute and State University, Department of Computer Science.
7. Shopiro, Jonathan E. «Extending the C++ Task System for Real-Time Control», Proceedings of the USENIX C++ Workshop, Santa Fe: USENIX Association Publishers (November 1987).
8. Ellis, Margaret A., and B. Stroustrup. «The Annotated C++ Reference Manual», Reading, Mass.: Addison-Wesley, 1990, Chapter 15.
9. Miller, W. M. «Exception Handling Without Language Extensions», Proceedings of the C++ Workshop, Denver: USENIX Association Publishers (October 1988).

Приложение А
C в среде C++

Язык C++ близок к C — большая часть того, что мы видим в программах C++, позаимствовано непосредственно из C, а многие программы C требуют минимальной доработки для превращения в программы C++[1]. В этом приложении показано, что нужно сделать для наделения программы C «стилем C++». Мы не пытаемся обучать читателя языку C. Если вам понадобится хороший учебник, обращайтесь к [2].

A.1. Вызовы функций

В C++ появился новый уровень видимости, не поддерживавшийся в C. Он определяется новой языковой конструкцией, называемой *классом* и предназначенной для создания *абстрактных типов данных*, или пользовательских типов. Таким образом, если в C практически любое имя могло быть локальным или глобальным, в C++ имена могут быть локальными, глобальными или принадлежащими некоторому классу. Правда, в C переменные также могли находиться на разных уровнях вложенных фигурных скобок внутри функций, а имена в структурах могли повторять имена, встречающиеся в локальной и глобальной областях видимости. Тем не менее, в C этого нельзя было сказать о функциях: все функции были либо глобальными, либо статическими (то есть попросту «глобальными по отношению к исходному файлу»). В C++ функции еще могут объявляться внутри классов — тогда они называются *функциями классов*. Таким образом, если в программе существуют глобальная функция `foo` и класс `F`, содержащий функцию `foo`, эти две функции существуют независимо друг от друга. Говоря о них, для функции `foo` класса `F` мы используем обозначение `F::foo`, а для глобальной функции — обозначение `::foo` (или просто `foo`).

Допустим, в классе `F` присутствует конструкция вида

```
foo();
```

С первого взгляда трудно сказать, какую функцию *хотел* вызвать программист — то ли функцию `foo` из класса `F`, то ли глобальную функцию `foo`. Возможно, для получения ответа на этот вопрос читателю программы придется проанализировать

---

[1] Различия между C++ и C подробно обсуждаются в [1].

код от точки вызова, выйти за пределы класса и в конечном итоге добраться до глобальной области видимости. Чтобы упростить чтение и понимание таких программ, можно воспользоваться синтаксисом, несколько проясняющим ситуацию для программ С++ с классами. В этом синтаксисе глобальные функции вызываются с префиксом уточнения глобальной области видимости С++:

```
::foo();
```

Впрочем, это всего лишь условное обозначение, а не стандарт программирования. Оно не изменяет сгенерированный код (в данном примере) и не упрощает работу компилятора (при отсутствии неоднозначностей). Тем не менее, эта запись помогает избежать недоразумений, поэтому она применяется в книге; читателю также рекомендуется применять ее в своей работе.

А.2. Параметры функций

Стиль объявления параметров функций С++ несколько отличается от стиля классического языка С. Возможно, он покажется знакомым программистам, работающим на Паскале:

Код С:

```
int main(argc, argv)
int argc;
char *argv[];
{
    ...
}
int func()
{
    ...
}
```

Код С++:

```
int main(int argc, char *argv[])
{
    ...
}

int func()
{
    ...
}
```

Объединенная синтаксическая форма, описывающая интерфейс с функцией, служит двум целям. Во-первых, она начинает тело функции в ее определении. Во-вторых, она используется как спецификация интерфейса функции. Эта спецификация, называемая *прототипом функции*, должна предшествовать вызову функции.

А.3. Прототипы функций

Грамотно написанный исходный файл на С начинается с объявления функций, используемых в этом файле, даже если определения этих функций размещаются в других файлах. Одни программисты включают эти объявления в каждую функцию, чтобы обозначить внешние функции, требующиеся данной функцией; другие собирают их в начале файла, третьи размещают все объявления в заголовочном файле, включаемом в начало файла директивой #include.

В C++ объявления лучше всего размещать в заголовочных файлах. Если вы предоставляете в распоряжение пользователя некоторую функцию или класс, обычно при этом объявление функции или класса передается в заголовочном файле, имеющем расширение .h и находящемся в специальном каталоге, отведенном под заголовочные файлы. Пользователи вашей программы включают заголовочный файл директивой #include и таким образом получают доступ к его функциям с проверкой типов. Присутствие объявления в заголовочном файле не только помогает обеспечить безопасность типов, но и избавляет от необходимости заново объявлять функцию или класс при очередной ссылке.

В большинстве сред программирования C++ все системные функции объявляются в заголовочном файле, поставляемом вместе с компилятором. Например, в операционной системе UNIX среда C++ содержит заголовочные файлы с объявлениями функций ядра и библиотечными функциями, упоминаемыми в разделах 2 и 3 руководства по UNIX. Чтобы получить доступ к объявлению функции, следует включить соответствующий заголовочный файл директивой #include. Если вы не знаете, какой именно файл должен включаться, обратитесь к администратору или консультанту, или просмотрите файл самостоятельно.

В «классическом» языке C функции не объявляются перед использованием [2]; обычно компилятор не выдает предупреждения и генерирует код. Он предполагает, что все необъявленные функции возвращают целые числа, а типы их параметров *точно* соответствуют типам передаваемых аргументов.

С другой стороны, C++ таких предположений не делает, а заставляет программиста четко выразить свои намерения: все идентификаторы и функции должны объявляться перед использованием. Следующие примеры демонстрируют различия между классическим языком C и C++:

Код C:

```
extern double myfunc();
extern double cos();
```

Код C++:

```
extern double myfunc(int, char);
extern "C" double cos(double);
```

В ANSI C используются те же правила, что и в коде C++ из приведенного примера, но без указания спецификатора компоновки **extern "C"**.

Обратите внимание: *тип компоновки* (то есть язык, на котором написана объявляемая функция — спецификатор **extern "C"** обозначает язык C) может явно указываться в объявлении. В общем случае он должен указываться для всех символических имен (как функций, так и данных) из кода, написанного на другом языке программирования, с которыми вы собираетесь работать в программе. В большинстве реализаций выбор компоновки C для данных ни на что не влияет, хотя в объектном коде имя, сгенерированное для функций с компоновкой C++, обычно отличается от имени той же функции с компоновкой C. Если тип компоновки не задан, предполагается компоновка C++ (как для **myfunc**).

Явное указание типов параметров и возвращаемого значения полезно в двух отношениях. Во-первых, оно гарантирует, что функция получит именно те типы аргументов, которые она ожидает получить, и не попытается обработать непредвиденные данные. Во-вторых, компилятор получает информацию, достаточную для того, чтобы при необходимости преобразовать один тип объекта в другой.

Например, С++ умеет автоматически преобразовывать целые числа в вещественные, что позволяет делать следующее:

Код С:

```
extern double sin();

sin(1);  /* Не определено */
sin((double)1);
sin(2.0);
sin(3.0);
```

Код С++:

```
extern double sin(double);


sin(1); // Автоматически
sin(2); // преобразуется в double
sin(3.0);
```

А.4. Передача параметров по ссылке

Стандартная семантика передачи параметров в С (и С ++) требует создания копии каждого передаваемого параметра. Этот механизм называется *передачей по значению*. После выхода из функции временная копия уничтожается. Иногда в программах С в качестве параметра передается адрес переменной; естественно, что функция всегда сможет разыменовать параметр оператором *, чтобы прочитать или присвоить его значение. Таким образом, функция может изменять значение переменной, переданное вызывающей стороной. В С++ поддерживается механизм передачи по ссылке, снимающий необходимость в указателях и их явном разыменовании. Следующие функции incr возвращают старое значение своего аргумента, увеличивая его текущее значение:

Код С:

```
int incr(i)
int *i;
{
    return (*i)++;
}

int main()
{
    int i, j = 5;
    i = incr(&j);
      /* j = 5, j = 6 */
    ...
}
```

Код С++:

```
int incr(int &i)
{
    return i++;
}


int main()
{
    int i, j = 5;
    i = incr(j);
    /* j = 5, j = 6 */

}
```

В синтаксисе С++ int &i означает *ссылку* на переменную i. Формальный параметр функции (i) становится синонимом другого значения или объекта, переданного в качестве аргумента (j). Имена i и j ссылаются на один объект; собственно, в этом и заключается смысл «передачи по ссылке». В общем случае при создании ссылочной переменной всегда можно указать уже существующий объект:

```
int m = 5;
int &n = m;  // n означает m , а m означает n
n = 6;       // И m, и n теперь равны 6
```

Как правило, ссылки применяются для передачи параметров функциям. Еще одно распространенное идиоматическое использование — возврат по ссылке, позволяющий избавиться от лишнего копирования при возврате значения из функции. Возврат по ссылке особенно полезен для часто копируемых, больших объектов (см. приложение В).

А.5. Переменное количество параметров

Иногда требуется определить функцию, которая может получать любое количество аргументов любого типа. По контексту функция определяет, сколько параметров ей было передано, и как их обрабатывать. Типичным примером служит функция printf языка С. В этом разделе кратко показано, как объявить, вызвать и реализовать подобную функцию.

Функция с неизвестным количеством аргументов неизвестных типов объявляется так:

```
extern "C" int printf(...);
```

Многоточие (...) означает, что тип или количество параметров могут меняться, или они просто не известны на стадии компиляции. Эта конструкция обходит систему проверки типов, что позволяет при вызовах передавать произвольное число аргументов любых типов. В разновидности этой формы объявляются типы любого фиксированного числа начальных параметров, за которыми следует неизвестное количество параметров неизвестных типов:

```
extern "C" int printf(const char *, ...);
```

Такие объявления обычно публикуются в заголовочных файлах, включаемых в приложения директивой #include. Заголовочные файлы для примитивов операционной системы и библиотеки С входят в поставку большинства компиляторов С++.

Функция с переменным количеством параметров вызывается в С ++ точно так же, как в С:

```
#include <stdio.h>
// В stdio.h объявляется функция printf
int main() {
    printf("hello world\n");
    int i = 1, j = 1;
    printf("%d plus %d equals %d\n", i, j, i+j);
    return 0;
}
```

Функции с неизвестным количеством параметров неизвестных типов определяются почти так же, как в С. Заголовочный файл stdarg.h, являющийся адаптированной версией файла varargs.h, позволяет писать программы с переменным количеством параметров независимым от платформы способом. В отличие от

классического языка С, С++ требует, чтобы в объявление функции входил хотя бы один формальный параметр с известным типом. Наличие параметра обеспечивает единственный гарантированно переносимый способ обращения к аргументам из тела вызываемой функции:

```
#include <stdarg.h>
#include <stdio.h>
int printf(const char *p1, ...) {
    va_list ap;
    va_start(ap, p1);
    ...
    switch(fmt) {
    case 'f':
        double d = va_arg(ap, double);
    case 's':
        char *string = va_arg(ap, char);
    case 'd': case 'c':
        int val = va_arg(ap, int);
        ...
    }
    va_end(ap);    // Вызывается перед возвратом
}
```

Применение стандартного пакета для работы с аргументами отличается от других приемов, встречающихся в классических программах С и основанных на предположениях о структуре стека или физической смежности лексически смежных аргументов [2]. Единственно безопасный, заведомо переносимый путь работы с фактическими аргументами, переданными функции с переменным числом аргументов, основан на механизме stdarg.h.

Правила хорошего стиля программирования не рекомендуют использовать функции с переменным списком параметров, чтобы не ухудшать переносимость и удобочитаемость программы. Пакет stdargs обеспечивает максимальную переносимость и удобство программирования, а объявление с многоточием удобнее записи, принятой в классическом языке С. Довольно часто stdargs удается заменить перегрузкой функций и аргументами по умолчанию; программа упрощается и становится более безопасной по отношению к типам. За дополнительной информацией обращайтесь к описанию varargs или stdargs в электронной документации вашей системы.

А.6. Указатели на функции

Указатели на функции в С++ работают практически так же, как в С. Единственное различие состоит в том, что при их объявлении должен указываться тип аргументов:

Код C:

```
void error(p) char *p;
{
    ...
}
void (&efct)();
int main()
{
    efct = &error;
    (*efct)("error");
    efct("also calls error");
    return 0;
}
```

Код C++:

```
void error(const char *p)
{
    ...
}
void (*efct)(const char *);
int main()
{
    efct = &error;
    (*efct)("error");
    efct("also calls error");
    return 0;
}
```

Другой пример — вызов системной функции signal. В следующем фрагменте signal объявляется как функция с двумя параметрами, int и SIG_PF (определяется в заголовочном файле). Сама функция просто возвращает указатель на функцию с параметром int:

Код C:

```
int efct2();

typedef inf(*SIG_PF) ();
SIG_PF fptr;

extern (*signal()) ();
fptr = signal(3, &efct2);
```

Код C++:

```
int efct2(int);

typedef inf(*SIG_PF) (int);
SIG_PF fptr;

extern (*signal(int, SIG_PF))(int);
fptr = signal(3, &efct2);
```

Похожий пример, но с более простыми типами аргументов:

Код C:

```
extern (*f())();

(*f(1,'a',123))(1.0,2);
(*(f(1,'a',123)))(1.0,2);
f(1,'a',123)(1.0,2);
```

Код C++:

```
extern int
  (*f(int,char,long))(double,int);
(*f(1,'a',123))(1.0,2);
(*(f(1,'a',123)))(1.0,2);
f(1,'a',123)(1.0,2);
```

Указатели на функции часто объявляются в typedef. Вот как выглядит версия примера signal с использованием определения типа:

Код C:

```
typedef int (*SIG_PF) ();
extern (*signal()) ();
```

Код C++:

```
typedef int (*SIG_PF) (int);
extern SIG_PF signal(int, SIG_PF);
```

Напоследок приведем более запутанный пример с typedef:

Код C:

```
typedef int (*FP) ();
extern FP (*f)();

f(1,'a',123)(1.0,2);
```

Код C++:

```
typedef int (*FP) (double,int);
extern FP f(int,char,long);

f(1,'a',123)(1.0,2);
```

А.7. Модификатор const

С++ позволяет преобразовать переменную в константу при помощи модификатора типа const. Такие объявления заменяют препроцессорные директивы #define. Модификатор const дает компилятору возможность организовать проверку присваивания указателям (или тем объектам, на которые они указывают), или записи в переменную после ее инициализации. Ниже приводятся три примера.

Пример 1. Использование модификатора const вместо директивы #define

Директива С #define часто используется (или должна использоваться) для определения мнемонических, наглядных имен для констант. Обращение к константам по имени помогает документировать их предполагаемую реализацию, а также централизовать определение и администрирование констант, неоднократно встречающихся в программе. В С++ для определения констант вместо директивы #define может указываться ключевое слово const. Вариант с const обеспечивает более качественную проверку типов на стадии компиляции, а в некоторых системах он также расширяет возможности символической отладки.

Код С:

```
#define CTRL_X '\030'
#define msg "an error"
```

Код С++:

```
const char CTRL_X '\030';
const char *msg = "an error";
```

В обоих вариантах:

```
int main() {
    char c;
    ...
    if (c == CTRL_X) { ... }
    printf("%s\n", msg);
    return 0;
}
```

Компилятор С++ гарантирует, что символические имена, определяемые подобным образом, остаются неизменными; программа не сможет случайно изменить константу. Намеренное изменение константной величины потребует явного преобразования (см. 2.9).

Пример 2. Модификатор const и указатели

Указатели считаются «аналогом команды goto в мире данных». К сожалению, указатели играют важную роль в программах С и С++; они позволяют эффективно организовать передачу больших записей данных или массивов между функциями. Иногда программисту требуется предоставить пользователю доступ к данным через указатель, но ограничить его чтением.

Ключевое слово const используется для управления модификацией данных — как непосредственно, так и через указатель. Возьмем следующий пример:

```
char *const a = "example 1";
```

Переменная-указатель a объявляется константной; иначе говоря, ее содержимое не может быть изменено. Но в данном случае «содержимое a» означает *адрес* некоторого блока памяти, в котором хранится последовательность символов, завершенная нулем. Таким образом, приведенное объявление означает, что мы не можем перевести указатель на другой блок памяти, но ничего не говорит об изменении тех данных, *на которые* он указывает:

```
a = "example 2";    // Компилятор выдает сообщение об ошибке
a[0] = '2';         // Успешно создается строка "example 2"
```

Выражение, которое может находиться в левой части операции присваивания, называется *l-значением*. В данном случае a не является допустимым l-значением, тогда как a[8] им является.

Чтобы указатель не мог использоваться как l-значение для модификации тех данных, на которые он ссылается, необходимо изменить объявление и объявить константными данные вместо указателя. Следующее объявление b делает содержимое строки "example 2" неизменяемым, тогда как объявление c запрещает модификацию как указателя, так и тех данных, на которые он ссылается. Символическое имя c может рассматриваться как неизменяемая константа:

```
const char *b = "example 2";
const char *const c = "example 3";
```

Переменная a не является l-значением, но им является *a. Указатель b является l-значением, а *b — нет. Наконец, c и *c не являются l-значениями. Обратите внимание: ключевое слово const ассоциируется с непосредственно следующим за ним объявлением:

```
a = "ex4";      // Нельзя
*a = 'E';       // Можно. a является l-значением
b = "ex4";      // Можно. b является l-значением
*b = 'E';       // Нельзя
c = "ex4";      // Нельзя
*c = 'E';       // Нельзя
```

Пример 3. Объявление функций с константными аргументами

Чтобы константность объекта сохранялась при передаче управления между функциями, можно добавить ключевое слово const в объявление формальных параметров. При попытке передать константный аргумент функции, ожидающей получить неконстантный формальный параметр, компилятор выдает сообщение об ошибке.

Возьмем следующее объявление:

```
extern char *strcpy(char*, const char*);
```

Пусть программа содержит такие объявления:

```
char a[20];
const char *b = "error message";
char *c = "user prompt> ";
```

Тогда компилятор разрешит следующие вызовы:

```
::strcpy(a, b); // Копирование const в неконстантную память
::strcpy(a, c); // Параметр const char* может быть неконстантным
::strcpy(c, b); // Аналог ::strcpy(a,b)
```

Однако такой вызов невозможен:

```
::strcpy(b, c); // Для неконстантного параметра
                // нельзя передавать const char*
```

Конечно, ограничение обходится явным преобразованием:

```
::strcpy((char *)b, c);  // Теперь нормально
```

Но тогда становится непонятно, зачем было объявлять b как const char*?

Ключевое слово const играет важную роль в канонических формах, упоминавшихся в книге.

А.8. Взаимодействие с кодом С

Многие системы содержат низкоуровневые вставки или унаследованный код, корни которого теряются в истории проекта. Даже новые системы нередко содержат смесь кода С и С++, что объясняется историческими или социологическими причинами. Поэтому в языке С++ была предусмотрена возможность взаимодействия с модулями, написанными на С. Настоящий раздел посвящен теме внешних связей с точки зрения С++: вы узнаете, как организовать сосуществование С++ с другими языками в одной программе. Сначала мы рассмотрим архитектурные аспекты такого сосуществования, а затем синтаксис и проблемы администрирования.

А.8.1. Архитектурные аспекты

Проблемы объединения кода С и С++ в одной программе не сводятся к синтаксису, механике общих объявлений и компоновки программ. Программист должен проанализировать С, С++ и другие языки на пригодность для данной задачи и той роли, которую они сыграют в ее решении, а также на совместимость с инструментарием проекта.

Главной причиной для применения С++ является возможность выражения архитектурных концепций с помощью абстрактных типов данных и объектов. Код С редко отражает эти элементы стиля программирования; чаще он реализует функциональную декомпозицию или генерируется автоматически (например, в результате работы генерирующего анализатора или CASE-системы). Таким образом, смесь кода С и С++ означает объединение минимум двух разных стилей проектирования.

Такая смесь часто кроется в истории развития продукта или в организационных недочетах. С++ может применяться для написания нового кода как в объектно-

ориентированных, так и в процедурных архитектурах. Смешение языков обычно объясняется наличием готового кода, писать который заново было бы слишком дорого, или применением генераторов кода С (скажем, yacc). Проектировщик должен решить, когда и как объединять разные стили.

Ниже приводятся примеры типичных ситуаций с объединением кода С и С++.

- *Библиотеки объектов в среде С.* Допустим, имеется система управления базами данных (СУБД), написанная на С++ и ориентированная в основном на приложения С++. Существующее приложение С требуется адаптировать для работы с этой СУБД. Программа С должна вызывать функции С++ для объектов СУБД, причем код С и С++ использует общие структуры данных. «Основная» часть программы пишется на С, а объекты С++ применяются как пассивная библиотека серверных объектов.
- *Библиотеки С в среде С++.* На С++ написано великое множество программ — как общедоступных, так и коммерческих. Построение программ С++ на этой базе значительно упрощается. Такие программы должны вызывать функции С и работать со структурами данных С (минимальный интерфейс между С и С++). Для примера можно привести программу С++, написанную на базе готовой графической библиотеки для рабочей станции или PC, или использование интерфейсов операционной системы из нового кода С++.
- *Инженерный анализ.* В некоторых ситуациях работающий код С переписывается на С++, чтобы задействовать преимущества проверки типов, структурной организации программ и другие расширенные возможности. Фрагменты готового кода С могут инкапсулироваться в классах С++ — либо с переводом кода С на С++, либо с предоставлением интерфейса для прозрачной работы с существующим кодом из абстракций С++.

  Многие существующие программы проектировались с использованием метода Джексона [4], анализа потока данных Йордона [5] или других методов, позаимствованных из литературы или обусловленных локальной спецификой проекта. Возможно, перевод проекта на объектно-ориентированную архитектуру позволит задействовать характерную для объектной парадигмы поддержку эволюции, инкапсуляции и защиты данных. Далее описаны две основные категории методов перевода существующей программной базы на объектную парадигму.
- *Индуктивные методы.* Начиная с конкретной информации о низкоуровневых компонентах существующей архитектуры (функциях, структурах данных и переменных), проектировщик выделяет комбинации этих компонентов, которые должны стать классами и объектами новой системы.

  Индуктивные методы делятся на управляемые данными и управляемые процедурами. В методах, управляемых данными, структуры данных (struct) рассматриваются как доминирующие архитектурные концепции, определяющие классы (рис. А.1). Иерархии наследования формируются на основе сходства между структурами данных. Функции, близко взаимодействующие с этими структурами, становятся функциями соответствующих классов. Хотя этот подход часто

приводит к разумной иерархии наследования, учтите, что выделение общих данных само по себе не определяет наследования. Гораздо более важную роль играют связи между семантиками базового и производного классов (глава 6). Часто бывает невозможно понять эти семантики, пока не будут идентифицированы функции классов. Получение правильной структуры требует хорошего знания предметной области и последовательного уточнения посредством построения прототипов.

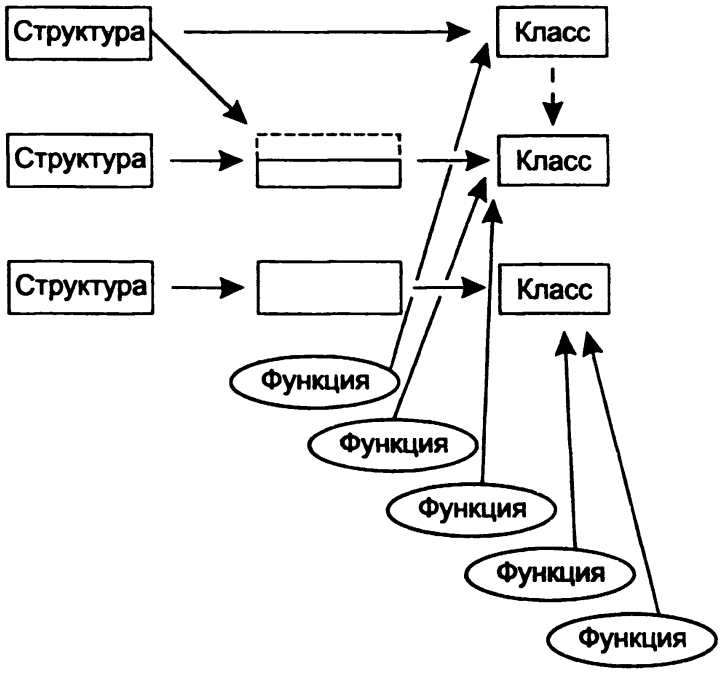


Рис. А.1. Индуктивный метод формирования классов, управляемый данными

- *Дедуктивные методы.* Дедуктивные методы начинают с абстракций и двигаются к конкретике, поэтому они больше подходят для реализации новых систем на основе спецификаций, нежели для эволюции существующих систем. Тем не менее, если система имеет четко определяемые абстракции высокого уровня, мы можем воспользоваться ими для построения структур классов и объектов, используемых в объектно-ориентированных системах (рис. А.2).

  Абстракции могут выявляться в процессе анализа ранней проектной документации; вряд ли они обнаружатся в коде, написанном на языке, в котором основной абстракцией являются процедуры. Тем не менее, понимание архитектуры поможет проектировщику выявить фрагменты, обусловленные общей структурой. Возможно, готовые фрагменты удастся без изменений использовать в новой структуре, которая сохранит и усилит исходные конструкции.

  Наследование более естественно формируется при дедуктивных методах, чем при индуктивных. Дедуктивные методы лучше отвечают принципам, описанным в главе 6. Впрочем, инженерный анализ создает определенный риск, например, в качестве классов новой системы могут быть идентифицированы абстракции, относящиеся не к пространству задачи, а к пространству решения (скажем, процессы).

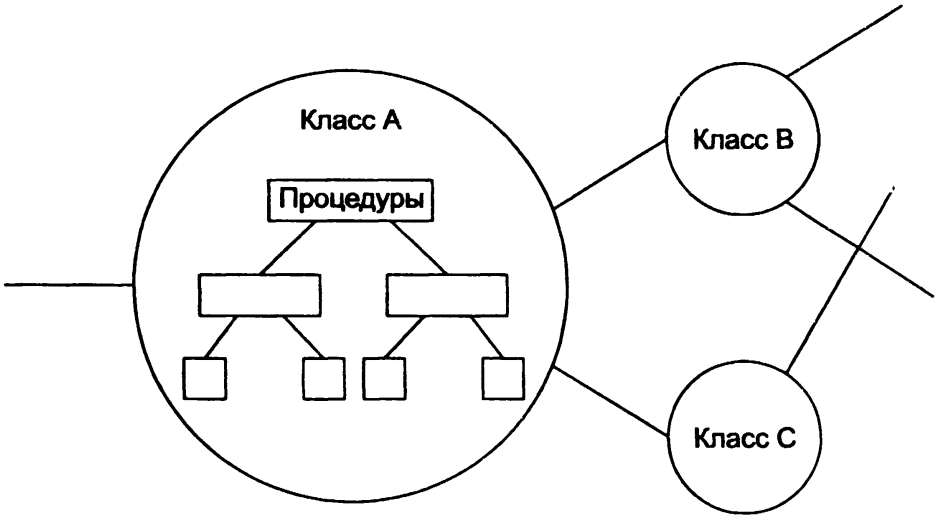


Рис. А.2. Дедуктивный метод формирования классов с внутренним пошаговым уточнением

Одна парадигма может использоваться для создания уточненных структур в рамках абстракций другой парадигмы — скажем, функциональная декомпозиция может применяться внутри классов. В традиционной архитектуре встречается независимая группировка взаимосвязанных функций. Функции в верхней части таких иерархий могут оказаться хорошими кандидатами на роль открытых функций класса или закрытых служебных функций, скрываемых внутри класса для использования другими функциями того же класса. Низкоуровневые функции иерархии преобразуются в закрытые функции класса или скрываются средствами структурного программирования, описанными в приложении Е.

У обоих вариантов имеется общий недостаток: в них делается попытка извлечь информацию проектирования из готовой реализации. Это примерно то же самое, что пытаться воссоздать трехмерное изображение по фотографии. Для реализаций, изначально основывавшихся на объектно-ориентированных методах или моделировании данных, подобный анализ способен принести плоды, но для реализаций, основанных на процедурной декомпозиции, его польза в лучшем случае сомнительна. Вместо того чтобы пытаться строить архитектурную модель на базе кода, лучше попробуйте вернуться к исходным системным спецификациям и использовать их для определения системных классов. Вполне возможно, что после этого существующие компоненты реализации после небольших изменений удастся разместить в структуре классов. Нет гарантий, что существующий процедурный код, полученный в результате методики Йордона, удастся точно отобразить на объектно-ориентированный код, а какая-то его часть может пережить такой переход. Методика системного анализа Джексона неплохо подходит для ранних фаз объектно-ориентированного проектирования, а системы, построенные на базе этой модели, могут упростить переход на объектную парадигму. И все же методика Джексона не дает вразумительных советов относительно того, как ее абстракции могут использоваться в структурах наследования.

Существующие фрагменты кода С могут инкапсулироваться в «модулях С++». Такое решение не требует переделки архитектуры, это всего лишь способ упаковки; тем не менее, оно предоставляет удобный механизм для подключения логически сплоченного кода С к новому коду С++.

В общем случае переводить стабильно работающий код С на С++ не рекомендуется: это не приносит сколько-нибудь существенной выгоды в долгосрочной перспективе, хотя и требует немалых краткосрочных затрат. В крупных проектах одноразовое полное преобразование вообще неприемлемо, а за пошаговое преобразование приходится расплачиваться наличием нескольких сред поддержки и усложнением интерфейса между преобразованным и непреобразованным кодом.

А.8.2. Языковая компоновка

С++ поддерживает перегрузку имен; иначе говоря, две функции могут иметь одинаковые имена, если нужная функция однозначно определяется по контексту вызова (то есть по типу переданных параметров). Заголовочные файлы программы могут содержать несколько объявлений одноименных функций с разными параметрами при условии согласования типов их возвращаемых значений. Для каждого такого объявления в исходном файле С ++ определяется свое тело функции.

В С++ 1.2 иногда возникали неприятности из-за порядка следования директив **#include** в начале файла. Допустим, файлы **cookie.h** и **cake.h** содержат объявления функции **print**:

```
cookie.h:       extern void print(Cookie*);
cake.h:         extern void print(Cake*);
```

Программист использует эти функции следующим образом:

```
overload print;   // Конструкция перегрузки в C++ 1.2
#include "cookie.h"
#include "cake.h"

int f1() {
    Cookie c;
    Cake d;

    print(&c);    // Вызывается функция с внутренним именем print
    print(&d);    // Вызывается функция с закодированным именем
```

Для первой версии **print** используется внутреннее имя **print**, а вторая функция во внутреннем представлении снабжается суффиксом, отличающим ее от первой функции. Все идет хорошо, если в другой части системы не произойдет следующее:

```
overload print;   // Конструкция перегрузки в C++ 1.2
#include "cake.h"
#include "cookie.h"
```

```
int f2() {
    Cookie c;
    Cake d;

    print(&c);    // Вызывается функция с закодированным именем
    print(&d);    // Вызывается функция с внутренним именем print
```

Порядок следования директив **#include** определяет вызываемую функцию! Одно из возможных решений — потребовать, чтобы все перегруженные объявления находились в одном файле **#include**. Для распространенных имен вроде **print**, встречающихся во многих библиотеках, такое решение неразумно.

Многие современные компиляторы C++ генерируют уникальные внутренние имена для всех функций, при этом типы аргументов шифруются в суффиксе, присоединяемом к имени функции. Ключевое слово **overload** стало пережитком прошлого. Но теперь возникает другая проблема: вызов **read** ассоциируется с внутренним именем **read_***нечто*, где суффикс *нечто* определяется типом аргументов функции. Если функция **read** написана на C, все работает нормально. Но если программист просто хотел вызвать функцию **read(2)**, написанную на C, программа не пройдет компоновку.

Для решения этой проблемы функции, написанные не на C++, должны помечаться особым образом. Например, для вызова C-версии **read** следует использовать объявление

```
extern "C" int read(int, char*, int);
```

Объявление **extern** может относиться сразу к нескольким функциям C:

```
extern "C" {
    int read(int, char*, int);
    int open(char*, int, int=0);
    int write(int, char*, int);
}
```

Такие объявления *не могут* использоваться внутри области видимости классов или функций; они должны определяться на внешнем уровне видимости. Лучше всего разместить их в заголовочных файлах, включаемых директивой **#include** для получения доступа к функции.

Некоторые существующие заголовочные файлы ANSI C содержат все необходимое для C++, за исключением конструкции **extern "C"**, определяющей компоновку функции. Чтобы использовать такой заголовочный файл в программе C++, просто включите директиву **#include** в объявление типа компоновки:

```
extern "C" {
#include <cfuncs.h>
};
```

Если заголовочный файл содержит другие объявления (например, структур или констант), это несущественно; они будут обработаны правильно. В частности, эта конструкция не влияет на функции классов, объявленные внутри блока **extern "C"**.

А.8.3. Вызов функций С++ из С

Функции классов С++ не могут напрямую вызываться из кода С. Вызовы должны проходить через промежуточные глобальные функции С++, связывающие эти два мира.

Было бы неправильно решать проблему на уровне отображения имен С++ в имена С системой компиляции С++ и вызывать «украшенные» имена на уровне исходного текста программы:

```
struct Stack {
    ...       // Объявление С, имитирующее
    ...       // структуру класса Stack в С++
};
struct Stack s;
...
push__5StackFi_v(&s,10);      /* Не рекомендуется */
```

Полученный код, во-первых, зависит от платформы; во-вторых, требует наличия информации о внутренних механизмах работы компилятора, отсутствующей у большинства пользователей; в-третьих, может отказаться работать в следующей версии компилятора, даже если он успешно работает в текущей; и, в-четвертых, слишком труден для понимания.

Гораздо правильнее написать в модулях компиляции С++ интерфейсные функции С, которые могут вызываться из обычных функций С:

Код С++:

```
class Stack {
    ...
};
extern "C" void CStackPush(Stack *s, int v) { s->push(v); }
```

Интерфейсные функции формируют «проходы» между двумя разными парадигмами. При их написании необходима осторожность, а наиболее успешно такие функции работают, когда они остаются невидимыми для пользователя.

А.8.4. Совместное использование заголовочных файлов в С и С++

Объявления функций в заголовочных файлах сообщают информацию о типах аргументов и возвращаемого значения. Если разработчик и пользователь функции будут применять один заголовочный файл, включая его в программу директивой #include, безопасность типов гарантируется даже для перегруженных функций. Было бы желательно, чтобы этот механизма работал с С-совместимыми функциями как при вызове из других функций С, так и при вызове из кода С++. Рассмотрим объявления ANSI С:

```
extern char *foo(short, long);
extern void bar(char, const char*);
```

```
#define size 10
extern char *text[SIZE];
```

Эти объявления отличаются от классического синтаксиса С:

```
extern char *foo();
extern void bar();
#define size 10
extern char *text[SIZE];
```

В свою очередь, классический синтаксис С отличается от синтаксиса С++:

```
extern "C" char *foo(short, long);
extern "C" void bar(char, const char*);
const unsigned char SIZE = 10;
extern char *text[SIZE];
```

Возможно ли использование одного общего заголовочного файла клиентами С и С++? В отдельных случаях удается выделить «наименьшее общее кратное» для всех трех языков. Например, спецификация конструкций SIZE и text С++ может использоваться в стиле С. С другой стороны, объявления в стиле С++ позволяют компилятору обеспечить более полную отладочную информацию или сгенерировать более эффективный код. Для объединения разных языковых стилей в одном заголовочном файле лучше воспользоваться *условной компиляцией*, то есть препроцессорными макросами #if, #ifdef и другими директивами этого семейства.

Компиляторы ANSI С определяют препроцессорное символическое имя __STDC__; компиляторы С++ определяют имя __cplusplus. Если вы работаете только с ANSI С и С ++, решение может выглядеть примерно так:

```
#if __cplusplus
extern "C" {
#endif
    extern char *foo(short, long);
    extern void bar(char, const char*);
#if __STDC__
#define SIZE 10
#endif
#if __cplusplus
const unsigned char SIZE = 10;
#endif
extern char *text[SIZE];
...
#if __cplusplus
}
#endif
```

Результат не идеален: литерал 10 встречается в двух местах, что создает опасность десинхронизации кода С и С++ по мере эволюции программы. В следующем решении значение, представленное именем SIZE, изменяется только в одном месте:

```
#if __cplusplus
extern "C" {
#endif
    extern char *foo(short, long);
    extern void bar(char, const char*);
#define COMMON_SIZE 10
#if __STDC__
#define SIZE COMMON_SIZE
#endif
#if __cplusplus
const unsigned char SIZE = COMMON_SIZE;
#endif
char *text[SIZE];
...
#if __cplusplus
}
#endif
```

Старайтесь выделять как можно большего общего текста, это способствует координации изменений.

Другая распространенная ситуация — среда, в которой классический язык С используется совместно с С++:

```
#if __cplusplus
extern "C" {
#endif
    extern char *foo(
#           if __cplusplus
                short, long
#           endif
    );
    extern void bar(
#           if __cplusplus
                char, const char*
#           endif
    );
#define COMMON_SIZE 10
#if __cplusplus
    const unsigned char SIZE = COMMON_SIZE;
#else
#   define SIZE COMMON_SIZE
#endif
extern char *text[SIZE];
    ...
#if __cplusplus
}
#endif
```

При слиянии всех трех стилей файл усложняется:

```
#if __cplusplus
extern "C" {
#endif
    extern char *foo(
#           if __cplusplus||__STDC__
                short, long
#           endif
    );
    extern void bar(
#           if __cplusplus||__STDC__
                char, const char*
#           endif
    );

#define COMMON_SIZE 10

#if !__cplusplus
#   define SIZE COMMON_SIZE
#else
    const unsigned char SIZE = COMMON_SIZE;
#endif

extern char *text[SIZE];
    ...
#if __cplusplus
}
#endif
```

Как правило, чтение такого кода занимает на порядок больше времени, чем его написание. Возможно, чтобы избавиться от хлопот с параллельным сопровождением, проще использовать одну директиву **#ifdef** для каждого языка и продублировать все содержимое.

В отдельных случаях единый общий синтаксис одинаково хорошо подходит для ANSI C и C++. Например, следующая форма может использоваться в C++ и в ANSI C для объявления функций без параметров:

```
тип_возвращаемого_значения  функция(void);
```

В программе ANSI C это объявление однозначно указывает на то, что функция вызывается без параметров. При отсутствии ключевого слова void компилятор ANSI C будет принимать вызовы функции с любым количеством параметров. Хотя в C++ ключевое слово void не обязательно, оно допустимо и обладает той же семантикой, что и в ANSI C. Другая форма, общая для обоих языков, может использоваться как основа для объявления символических констант:

```
#if __cplusplus
extern "C" {
```

```
#endif
static const тип имя1 = инициализатор;
static const тип имя2;
#if __cplusplus
}
#endif
```

Присутствие ключевых слов static и extern играет ключевую роль для согласования синтаксиса двух языков.

Впрочем, все эти принципы — не более чем информация к размышлению. В каждой конкретной среде следует применять их так, как подсказывает здравый смысл, а если обнаружатся более удобные альтернативы — использовать их на практике.

А.8.5. Импорт форматов данных С в С++

Ранее рассказывалось, как организовать вызов функций С в С++ и наоборот. А как насчет данных? Если у вас имеется структура С, к которой нужно обращаться из С++, нередко удается использовать один заголовочный файл с объявлением структуры в обоих языках. Если объявление структуры соответствует синтаксису С, применение ее «экземпляров» в коде С и С++, объединенном в одной программе, не создает никаких проблем.

А.8.6. Импорт форматов данных С++ в С

Совместимость «снизу вверх» работает на нас при написании кода С++, использующего существующие форматы данных С, но с обратным преобразованием дело обстоит сложнее. В общем случае нельзя добавить код С в существующее приложение С++ и ожидать, что код С будет работать с форматами данных С++ — понадобятся дополнительные преобразования.

Если в программе С++ имеется простая конструкция struct языка С, не содержащая функций, она может включаться в программы С и С++ и использоваться так, как описано выше.

В нетривиальных ситуациях применяются два основных решения.

- (*Не рекомендуется*). Определите формат данных в классе С++ и продублируйте его в структуре С. Класс с виртуальными функциями может содержать скрытые поля, для которых в структуру должны включаться заглушки (фиктивные члены). Расположение и размер этих полей зависят от реализации. Если среда С++ позволяет генерировать объектный код С по исходному коду С++, возможно, вам удастся сгенерировать заголовочный файл С по заголовочному файлу С++, содержащему объявление класса (для этого файл обрабатывается компилятором или транслятором с соответствующими ключами). Данное решение приводит к появлению заголовочных файлов, содержащих странные имена полей, к тому же для его реализации программист С должен хорошо разбираться во внутренних алгоритмах конкретного компилятора. Синхронизация двух интерфейсов выполняется вручную, что повышает риск ошибок.

- Все обращения к данным производятся через специальные функции C++ с использованием приемов, упоминавшихся ранее.

В листинге А.1 приведен пример кода С, который работает с полями класса EmployeeRecord, написанного на C++.

Листинг А.1. Работа с кодом C++ из кода С

```
extern "C" const char *EmployeeRecordName(void*);
extern "C" short EmployeeRecordYOS(void*);
extern "C" double EmployeeRecordSalary(void*);

class EmployeeRecord {
friend const char *EmployeeRecordName(void*);
friend short EmployeeRecordYOS(void*);
friend double EmployeeRecordSalary(void*);
public:
    EmployeeRecord(const char*, short, double);
    EmployeeRecord();
    void giveRaise(double);
    virtual void disbursePay();
private:
    char name[NAME_SIZE];
    short yearsOfService;
    double monthlySalary;
};

extern "C" {
    void aCFunction(void *RecordPtr);

    const char *EmployeeRecordName(void *RecordPtr) {
        return ((EmployeeRecord *)RecordPtr)->name;
    }

    short EmployeeRecordYOS(void *RecordPtr) {
        return ((EmployeeRecord *)RecordPtr)->yearsOfService;
    }

    double EmployeeRecordSalary(void *RecordPtr) {
        return ((EmployeeRecord *)RecordPtr)->monthlySalary;
    }
}
    ...
    EmployeeRecord aRecord;
    aCFunction(&aRecord); // Вызов функции в приложении С
                          // с передачей указателя на данные C++
```

Код С получает указатель void* и использует его в параметрах функций, осуществляющих разыменование:

```
extern char *EmployeeRecordName(void*);
extern short *EmployeeRecordYOS(void*);
```

```
extern double EmployeeRecordSalary(void*);

void* aCFunction(void *RecordPtr) {
    char *name = RmployeeRecordName(RecordPtr);
    short YOS = EmployeeRecordYOS(RecordPtr);
    double salary = EmployeeRecordSalary(RecordPtr);
    ...
}
```

Упражнения

1. Напишите упрощенную версию функции printf, обрабатывающую только форматы %d и %s.
2. Напишите функцию быстрой сортировки qsort со следующим интерфейсом:

```
void qsort(void *base, int nel,
  int elementSize, int (*compar)(void*, void*));
```

   Здесь base — указатель на базовый элемент сортируемой таблицы; nel — количество элементов в таблице; elementSize — размер одного элемента; compar — указатель на функцию, которая сравнивает два элемента и возвращает целое число, меньшее, равное или большее нуля, если первый аргумент соответственно меньше, равен или больше второго аргумента. Можете воспользоваться исходным текстом любой существующей функции сортировки.
3. Напишите объявления функций UNIX printf, malloc, read, write, pipe, open, close и signal.

Литература

1. Koenig, A. and B. Stroustrup. «C++: As Close as Possible to C — But No Closer», C++ Report 1,7 (July/August 1989).
2. Kernighan, Brian W., and Dennis M. Ritchie. «The C Programming Language», 2nd ed. New York: Prentice-Hall, 1988.
3. DeMarco, T. «Structured Analysis and System Specification», New York: Yourdon Press, 1978.
4. Jackson, M. A. «Principles of Program Design». London: Academic Press, 1975.

Приложение Б
Программа Shapes

```
#include "Shapes.h"
/*-------------------------------------------------------------*/
/* File main.c -- управляющая программа для примера Shapes   */
/*-------------------------------------------------------------*/

const int XSIZE = 800;
const int YSIZE = 1024;

void
rotateList(Shape *s, Angle a)
{
    for (Shape *f = s;  f;  f = f->next())
        f->rotate(a);
}

int
main()
{
    Coordinate origin (0,0);

    /*
     * windowBorder не только определяет границы окна, но и служит
     * заголовком списка фигур
     */


    Rect windowBorder(origin, XSIZE, YSIZE);

    Shape * t = new Triangle(
        Coordinate(100, 100),
        Coordinate(200, 200),
        Coordinate(300, 100)
    );
    windowBorder.append(t);

    Shape * c = new Circle(Coordinate(500,652), 150);
    t->insert(c);
```

```
    rotateList(&windowBorder, 90.0);
    return 0;
}

#include <math.h>
#include <X11/X.h>

/*
 * Файл Shapes.h
 * Автор:  Дж. Коплин
 *
 * Заголовочный файл содержит объявления классов, используемых
 * в демонстрационной библиотеке Shape. Многие функции классов
 * определяются подставляемыми.
 */

/*-------------------------------------------------------------*/
/* Класс Angle представляет угол в градусах или радианах.     */
/* Он оформлен как класс, что позволяет легко адаптировать    */
/* его для систем, работающих как в градусах, так и           */
/* в радианах, а также изменить математическую точность       */
/* представления (float, double или int).                      */
/*-------------------------------------------------------------*/

class Angle {
friend double cos(Angle);
friend double sin(Angle);
public:
    Angle(double d)       { radians = d; }
    Angle ()              { radians = 0; }
    Angle(const Angle& a) { radians = a.radians; }
    inline Angle &operator=(const Angle& a) {
        radians = a.radians;
        return *this;
    }
    Angle operator+(const Angle&) const;
private:
    double    radians;
};

/*-------------------------------------------------------------*/
/* Класс Coordinate описывает точку на декартовой плоскости.   */
/* Значения координат могут задаваться в микрометрах,          */
/* пикселах, дюймах или в любых других удобных единицах.       */
/*-------------------------------------------------------------*/

class Coordinate {
public:
    inline long x() const       { return x_rep; }
```

```
    inline long y() const       { return y_rep; }
    Coordinate    &operator=(const Coordinate &c) {
        x_rep = c.x_rep;
        y_rep = c.y_rep;
        return *this;
    }
    Coordinate(long x, long y) {
        x_rep = x; y_rep = y;
    }
    Coordinate(const Coordinate &c) {
        x_rep = c.x_rep;
        y_rep = c.y_rep;
    }
    Coordinate()        { x_rep = y_rep = 0;}
    ~Coordinate()       { }

    void rotateAbout(Coordinate,Angle);
private:
    long x_rep, y_rep;
};

/*-------------------------------------------------------------*/
/* Следующие функции перегружают обычные тригонометрические  */
/* функции таким образом, чтобы они могли вызываться          */
/* с аргументом Angle. Поскольку перегрузка не может          */
/* производиться только на базе возвращаемого значения        */
/* (обязательно должен изменяться тип аргументов),            */
/* перегрузка функции atan, которая должна уметь возвращать   */
/* объект типа Angle, невозможна. Ее приходится определять    */
/* как новую функцию.                                         */
/*-------------------------------------------------------------*/

inline double cos(Angle a)      { return ::cos(a.radians); }
inline double sin(Angle a)      { return ::sin(a.radians); }
inline Angle Angleatan(double t) {
    Angle a(::atan(t)); return a;
}

/*-------------------------------------------------------------*/
/* Класс Color инкапсулирует реализацию кодировки цветов      */
/* для используемого графического пакета. Его внутренние      */
/* данные доступны только для функций низкоуровневого         */
/* интерфейса, тогда как высокоуровневые функции работают     */
/* на более общем уровне. Такой подход делает возможным       */
/* адаптацию программы для широкого спектра платформ.         */
/* Flashing (мигание) - атрибут цвета. Часто используемые     */
/* константы Black и White объявлены в виде глобальных        */
/* константных объектов типа Color.                           */
/*-------------------------------------------------------------*/
```

```
enum Flash { FlashBlack, FlashWhite, FlashOff };

class Color {
public:
    inline    double    red()         { return red_rep;  }
    inline    double    green()       { return green_rep;  }
    inline    double    blue()        { return blue_rep;  }
            Color(double r=0.0, double g=0.0,
                  double b=0.0, Flash f=FlashOff) {
       red_rep = r;
       green_rep = g;
      blue_rep = b;
       flashing = f;
    }
    inline    Color&    operator=(const Color& c) {
       red_rep = c.red_rep;
       green_rep=c.green_rep;
       blue_rep = c.blue_rep;
       return *this;
    }
            ~Color()        { }
private:
    double    red_rep, green_rep, blue_rep;
    Flash    flashing;
};

const Color Black, White = Color(1.0, 1.0, 1.0);

/*-------------------------------------------------------------*/
/* Shape - базовый класс для всех геометрических фигур.        */
/* Он определяет сигнатуру (совокупность выполняемых           */
/* операций), а также предоставляет фактическую реализацию     */
/* операций, общих для всех фигур (например, операция          */
/* перемещения move является общей с отдельными исключениями). */
/*-------------------------------------------------------------*/

class Shape {
public:
    // Поворот относительно центра
    virtual    void    rotate(Angle a)        { }
    virtual    void    move(Coordinate xy) {
       erase(); center_val=xy; draw();
    }
    virtual    void    draw()  = 0;      // Прорисовка
    virtual    void    erase() {
       color = Black; draw();
    }

    virtual    void    redraw() {
       erase(); draw();
```

```
    }
    Shape();
    virtual   ~Shape();
    virtual   Coordinate origin() const  { return center_val; }
    virtual   Coordinate center() const  { return center_val; }
    inline    Shape   *next() const   { return next_pointer; }
    inline    Shape   *prev() const   { return prev_pointer; }
    inline    void    insert(Shape* i) {
        i->next_pointer= this;
        i->prev_pointer=prev_pointer;
        prev_pointer->next_pointer=i;
        prev_pointer = i;
    }
    inline    void    append(Shape* i)    {
                        i->next_pointer=next();
                        i->prev_pointer = this;
                        next_pointer->prev_pointer=i;
                        next_pointer = i;
                      }
protected:
    /*
     * Следующие поля объявлены защищенными, а не закрытыми,
     * чтобы они были доступны для функций производных классов
     */

    Coordinate    center_val;    // Номинальный центр
    Color        color;          // Цвет фигуры

    /*
     * Следующие статические переменные класса используются
     * базовым пакетом оконного интерфейса (например, X)
     * для общей настройки графики.
    */

    static        Display        display;
    static        Window        window;
    static        GContext        gc;

private:
    Shape        *next_pointer;  // Указатель на следующую
                                 // фигуру в списке
    Shape        *prev_pointer;  // Указатель на предыдущую
                                 // фигуру в списке
};

/*-------------------------------------------------------*/
/* Line -- особая разновидность фигур, которая обычно    */
/* используется для прорисовки других фигур. Из всех фигур */
/* только она содержит функцию rotateAbout, используемую  */
```

```
/* как примитив для вращения других фигур. Операция rotate */
/* является вырожденной формой rotateAbout. Центр отрезка  */
/* определяется как средняя точка между его концами.       */
/* началом считается первая из заданных конечных точек.    */
/*---------------------------------------------------------*/

class Line : public Shape {
public:
    void    rotate(Angle a)    { rotateAbout(center(),a); }
    void    rotateAbout(Coordinate,Angle);
    void    draw();
    inline Line &operator=(const Line &l) {
     (void)this->Shape::operator=(*this);
         p=Coordinate(l.p); q=Coordinate(l.q); return *this;
    }
    Line(Coordinate,Coordinate);
    Line(Line& l) {
        p = l.p; q = l.q;
        center_val = l.center_val;
    }
    Line()            { p = Coordinate(0,0);
                        q = Coordinate(0,0); }
   ~Line()            { erase(); }
    inline  Coordinate      e() const   { return q; }
    inline  Coordinate origin() const   { return p; }
private:
    Coordinate    p, q;            // Конечные точки отрезка
};

/*-------------------------------------------------------------*/
/* Прямоугольник (Rect) состоит из четырех отрезков. Началом   */
/* прямоугольника считается его левый верхний угол, задаваемый */
/* первым параметром конструктора, а его центром считается     */
/* геометрический центр. Операция erase наследуется от Shape.  */
/* Стороны объявлены защищенными, а не закрытыми, чтобы они    */
/* могли использоваться производным классом Square.            */
/*-------------------------------------------------------------*/

class Rect : public Shape {
public:
    void        rotate(Angle);
    void        draw();
    void        move(Coordinate);
            // Левый верхний угол, ширина, высота
            Rect(Coordinate,long,long);
            ~Rect()            { erase(); }
    inline    Coordinate    origin() const { return l1.origin(); }
protected:
    Line        l1, l2, l3, l4;
};
```

```
/*-------------------------------------------------------*/
/* Центром эллипса (Ellipse) является его геометрический */
/* центр; эта же точка считается началом фигуры.         */
/* Класс Ellipse не реализован.                          */
/*-------------------------------------------------------*/

class Ellipse : public Shape {
public:
    void        rotate(Angle);
    void        draw();
            // Центр, большая ось, малая ось
            Ellipse(Coordinate,long,long);
            ~Ellipse()        { erase(); }
protected:
    long        major, minor;
};

/*-----------------------------------------------------*/
/* Треугольник (Triangle) состоит из трех отрезков.    */
/* Его центр вычисляется как средняя точка между       */
/* тремя вершинами.                                    */
/*-----------------------------------------------------*/

class Triangle : public Shape {
public:
    void        rotate(Angle);
    void        draw();
    void        move(Coordinate);
            Triangle(Coordinate,Coordinate,Coordinate);
            ~Triangle()        { erase(); }
private:
    Line        l1, l2, l3;
};

/*-----------------------------------------------------------*/
/* Квадрат (Square) представляет собой вырожденный           */
/* прямоугольник. Все его функции наследуются от Rect;       */
/* только конструктор имеет собственную реализацию, которая */
/* ограничивается простым вызовом конструктора Rect.         */
/*-----------------------------------------------------------*/

class Square : public Rect {
public:
            Square(Coordinate ctr, long x):
                Rect(ctr, x, x)        { }
};

/*-------------------------------------------------*/
/*                                                 */
/* Файл Shapes.c содержит примерную реализацию Shape */
/*                                                 */
/*-------------------------------------------------*/
```

```
extern "C" {
    extern void doXInitialization(Shape&);
    extern void
       XDrawLine(Display, Window, GContext, int, int, int, int);
}

/*
 * Класс ANGLE
 */

Angle
Angle::operator+(const Angle& angle) const {
    Angle retval = angle;
    retval.radians += radians;
    return retval;
}

/*
 * Класс COORDINATE
 */

void
Coordinate::rotateAbout(Coordinate pivot, Angle angle)
{
    long xdistance = pivot.x()-x(), ydistance = pivot.y()-y();
    long xdistsquared = xdistance * xdistance,
        ydistsquared = ydistance * ydistance;
    double r = ::sqrt( xdistsquared + ydistsquared );
    Angle newangle = angle +
        Angleatan(double(ydistance)/double(xdistance));
    x_rep = pivot.x() + long(r * ::cos(newangle));
    y_rep = pivot.y() + long(r * ::sin(newangle));
}

/*
 * Класс SHAPE
 */

/* Флаг, используемый базовым графическим пакетом */

static int X_initialized = 0;

Shape::Shape() {
    center_val = Coordinate(0,0);
    next_pointer=prev_pointer=0;
    color = White;
    if( !X_initialized) {
        doXInitialization(*this);
```

```
        X_initialized = 1;
    }
}

/*
 * Класс LINE
 */

void
Line::rotateAbout(Coordinate c, Angle angle) {
    erase();
    p.rotateAbout(c, angle);
    q.rotateAbout(c, angle);
    draw();
}

void
Line::draw() {
    if (p.x()-q.x() && p.y()-q.y()) {
      XDrawLine(display, window, gc, p.x(),
                    p.y(), q.x(), q.y());
    }
}

Line::Line(Coordinate p1, Coordinate p2) {
    p = p1; q = p2;
    center_val = Coordinate((p.x()+q.x())/2, (p.y()+q.y())/2);
    color = Color(White);
    draw();
}

/*
 * Класс RECTANGLE
 */

void
Rect::rotate(Angle angle) {
    erase();
    l1.rotateAbout(center(), angle);
    l2.rotateAbout(center(), angle);
    l3.rotateAbout(center(), angle);
    l4.rotateAbout(center(), angle);
    draw();
}

void
Rect::draw() {
    l1.draw();
    l2.draw();
```

```
    l3.draw();
    l4.draw();
}

void
Rect::move(Coordinate c) {
    /*
     * Аргумент определяет новое положение центра.
     * Функция определяет смещение центра и производит
     * соответствующее смещение остальных точек.
     */

    erase();
    long xmoved = c.x() - center().x();
    long ymoved = c.y() - center().y();
    center_val = c;
    l1 = Line(Coordinate(l1.origin().x()+xmoved,
                         l1.origin().y()+ymoved),
            Coordinate(l1.e().x()+xmoved,l1.e().y()+ymoved) );
    l2 = Line(Coordinate(l2.origin().x()+xmoved,
                         l2.origin().y()+ymoved),
            Coordinate(l2.e().x()+xmoved,l2.e().y()+ymoved) );
    l3 = Line(Coordinate(l3.origin().x()+xmoved,
                         l3.origin().y()+ymoved),
            Coordinate(l3.e().x()+xmoved,l3.e().y()+ymoved) );
    l4 = Line(Coordinate(l4.origin().x()+xmoved,
                         l4.origin().y()+ymoved),
            Coordinate(l4.e().x()+xmoved,l4.e().y()+ymoved) );
    draw();
}

Rect::Rect(Coordinate topLeft, long xsize, long ysize) {
    Coordinate    a(topLeft);
    Coordinate    b(a.x()+xsize, a.y());
    Coordinate    c(a.x(),a.y()+ysize);
    Coordinate    d(b.x(),c.y());
    l1 = Line(a,b);
    l2 = Line(b,c);
    l3 = Line(c,d);
    l4 = Line(d,a);
    center_val = Coordinate((l1.origin().x()+l2.e().x())/2,
                    (l4.origin().y()+l4.e().y())/2);
    draw();
}

/*
 * Класс TRIANGLE
 */
```

```
void
Triangle::rotate(Angle angle) {
    erase();
    l1.rotateAbout(center(), angle);
    l2.rotateAbout(center(), angle);
    l3.rotateAbout(center(), angle);
    draw();
}

void
Triangle::move(Coordinate c) {
    /*
     * Аргумент определяет новое положение центра фигуры.
     * Функция определяет смещение центра и производит
     * соответствующее смещение остальных точек.
     */

    erase();
    long xmoved = c.x() - center().x();
    long ymoved = c.y() - center().y();
    center_val = c;
    l1 = Line(Coordinate(l1.origin().x()+xmoved,
                         l1.origin().y()+ymoved),
            Coordinate(l1.e().x()+xmoved,l1.e().y()+ymoved) );
    l2 = Line(Coordinate(l2.origin().x()+xmoved,
                         l2.origin().y()+ymoved),
            Coordinate(l2.e().x()+xmoved,l2.e().y()+ymoved) );
    l3 = Line(Coordinate(l3.origin().x()+xmoved,
                         l3.origin().y()+ymoved),
            Coordinate(l3.e().x()+xmoved,l3.e().y()+ymoved) );
    draw();
}

void
Triangle::draw() {
    l1.draw(); l2.draw(); l3.draw();
}

Triangle::Triangle(Coordinate a, Coordinate b, Coordinate c) {
    l1 = Line(a,b);  l2 = Line(b,c);  l3 = Line(c,a);
    center_val =
              Coordinate((l1.e().x()+l2.e().x()+l3.e().x())/3,
                         (l1.e().y()+l2.e().y()+l3.e().y())/3);
    draw();
}
```

Приложение В
Ссылочные возвращаемые значения операторов

Ссылочная переменная не является ни указателем, ни объектом или переменной в традиционном понимании. Скорее, она определяет дополнительное *имя* для объекта. При создании ссылки появляется возможность обращаться к уже существующему объекту по новому имени.

Почему же функции operator= и operator[] возвращают ссылочные значения? На высоком уровне ответ звучит так: потому что ссылка позволяет использовать возвращаемое значение как имя переменной. Другими словами, всюду, где может находиться имя переменной, допускается существование ссылки на переменную (того же типа). Например, имя может применяться в качестве левостороннего значения (нечто такое, что может находиться в левой части оператора присваивания), чего нельзя сказать о выражениях вообще. Возврат ссылок операторными функциями operator= и operator[] означает, что такие выражения могут быть левосторонними. Данный факт особенно важен для векторов, чтобы индексируемым элементам можно было присваивать новые значения (см. 3.3).

Следующая идиома (из приложения А) демонстрирует простейшую передачу параметров по ссылке:

Код C:

```
int incr(i)
int *i;
{
    return (*i)++;
}
int main()
{
    int j = 5;
    incr(&j);
}
```

Код C++:

```
int incr(int &i)
{
    return i++;
}

int main()
{
    int j = 5;
    incr(j);
}
```

Идиома передачи по ссылке распространяется на возвращаемые значения функций, в том числе и перегруженных операторных функций. В действительности многие встроенные бинарные операторы возвращают ссылки для обеспечения ассоциативности. Рассмотрим следующий пример:

Код C:

```
struct String {
    char *rep;
};

char index(t, i)
struct String *t; int i;
{
    return t->rep[i];
}

int main()
{
    struct String x; char c;
    ...
    c = index(x, 5);
    index(x,6) = 'a';
    return 0;
}
```

Код C++:

```
1  class String {
2  public:
3      char operator[](int i) {
4          return t[i];
5      }
6  private:
7      char *t;
8  };
9
10
11 int main()
12 {
13     String x; char c;
14     ...
15     c = x[5];
16     x[6] = 'a';
17     return 0;
18 }
```

Для версии C компилятор выдает сообщение об ошибке:

```
"t2.c", line 16: addressable expression required
```

Для версии C++ сообщение выглядит иначе:

```
"t2.c", line 16: error: assignment to generated
    function call (not an lvalue)
```

Но если добавить в версию C промежуточный уровень логической адресации (и объявить ссылочное возвращаемое значение в версии, написанной на C++), все работает нормально.

Код C:

```
struct String {
    char *rep;
};

char *index(t, i)
struct String *t; int i;
{
    return t->rep + i;
}

int main()
{
    char x[80], c;
    c = *(index(x, 5));
    *(index(x,6)) = 'a';
    return 0;
}
```

Код C++:

```
class String {
public:
    char &operator[](int i) {
        return t[i];
    }
private:
    char *t;
};


int main()
{
    String x; char c;
    c = x[5];
    x[6] = 'a';
    return 0;
}
```

Приложение Г
Поразрядное копирование

До выхода версии 1.2 транслятора C++ присваивание структур (и классов) выполнялось точно так же, как в языке C: содержимое источника просто копировалось в приемник. Такой способ называется *поразрядным копированием.*

Обычно программист при определении класса C++ использует каноническую форму (см. 3.1), чтобы предотвратить поразрядное копирование. Он «перехватывает» все операции, приводящие к копированию: присваивание (operator=), передачу аргументов и возвращаемых значений (X::X(X&)). Затем пользователь может сделать то же, что происходит при поразрядном копировании, или нечто более интересное (например, обновить счетчик ссылок). Очень важно, чтобы эта специальная обработка выполнялась при хранении в объекте *указателя* на динамически выделенную память; иначе возможны непредсказуемые последствия.

Вернемся к нашему примеру String:

```
class String {
public:
    String(int n)             { rep = new char[size = n]; }
    ~String()                 { delete rep; }
    inline int length() const { return size; }
private:
    char *rep;
    int size;
};
```

Теперь посмотрим, что произойдет, если объявить в программе два объекта String, а затем присвоить один другому:

```
int main() {
    String s1(10), s2(20);
    s1 = s2;
    return 0;
}
```

Программа сначала вызывает конструктор String(int) для каждого из объектов s1 и s2; в свою очередь, каждый объект выделяет память под символьный вектор и сохраняет указатель на него в своей переменной rep. Затем происходит присваивание, и каждое поле s1 становится в точности равным соответствующему

полю s2. В результате s1.rep и s2.rep ссылаются на один вектор из 20 элементов, а исходный вектор из 10 элементов, изначально связанный с s1, становится недоступным! Даже сам по себе этот факт выглядит странно. Но это не все: при выходе из программы автоматически вызываются деструкторы s1 и s2 (их вызов обеспечивается компилятором). Каждый конструктор вызовет оператор delete для своего поля rep, а следовательно, один и тот же блок памяти будет освобожден дважды. Обычно это приводит к нарушению логической целостности кучи и аварийному завершению программы.

Чтобы решить эту проблему, следует перехватить присваивание и либо создать локальную копию вектора s1, либо обновить счетчик ссылок. Вот почему в любом нетривиальном классе определяются функция operator= и конструктор X::X(const X&), как это сделано в ортодоксальной канонической форме.

ВНИМАНИЕ

Копирование на уровне членов класса — не панацея.

C++ автоматически генерирует требуемый оператор = и конструктор X::X(const X&) для тех классов, в которых они отсутствуют. Конструктор X::X(const X&) инициализирует текущий объект копированием соответствующих членов параметра; функция operator= последовательно присваивает их значения. Однако при этом не всегда используется поразрядное копирование: X::X(const X&) инициализирует переменные объекта при помощи *их* конструктора X::X(const X&), а присваивание выполняется рекурсивно. Эти функции могут увеличивать счетчики ссылок на динамически выделенную память, принадлежащую внутренним объектам, или выполнять другие служебные операции для предотвращения упоминавшегося на примере String хаоса.

Эта схема хорошо работает для объектов классов. Но *указатели* (точнее, указатели, освобождаемые вызовом оператора delete в деструкторе) нарушают ее работу, и мы приходим к той же проблеме, с которой все началось: повторному освобождению динамически выделенной памяти. Поэтому во всех классах, в которых деструктор вызывает оператор delete для одного из членов, важно определять функцию operator= и конструктор X::X(const X&).

Одним из поводов для перегрузки функции operator-> (см. раздел 6.5) в C++ 2.0 стало стремление к использованию исключительно объектов; в этом случае автоматическое копирование на уровне членов класса всегда работает правильно, поскольку указатели уходят со сцены. Тем не менее, такой подход еще не получил достаточного распространения в сообществе программистов C++, и вместо него разрабатываются другие методики.

Приложение Д
Иерархия геометрических фигур в символической идиоме

```
//*************************************************************//
//                                                             //
//     Ф А Й Л :    M P T R . H                                //
//                                                             //
//         Объявления структур данных компилятора C++          //
//                                                             //
//*************************************************************//

#define _MPTR_H

// Следующие структуры данных обеспечивают работу
// механизма виртуальных функций C++. Обобщенный
// указатель на функцию vptp используется для адресации
//  (но не для вызова) функций классов.Тип vvptp определяет
// указатель на функцию, возвращающую указатель на функцию;
// он используется некоторыми идиомами оперативной
// модификации данных класса. Структура mptr описывает
// элемент таблицы виртуальных функций; ее формат зависит
// от специфики построения объектного кода вашим
// компилятором C++.

typedef int (*vptp)();
typedef vptp (*vvptp)();
struct mptr {short d; short i; vptp f; };

//*************************************************************//
//                                                             //
//      Ф А Й Л :   K . H                                      //
//                                                             //
//         Объявления классов Thing и Top                      //
//                                                             //
//*************************************************************//
```

```
#ifndef _MPTR_H
#include "mptr.h"
#endif
#define _K_H
#include <String.h>

//  ЗАВИСИТ ОТ ПЛАТФОРМЫ И КОМПИЛЯТОРА
const unsigned int WRDSIZE = sizeof(void*);

inline size_t
Round(size_t s) {
    return (size_t)((s + WRDSIZE - 1)/WRDSIZE)*WRDSIZE;
}

// Часто используемый тип указателя на символ
typedef char *Char_p;

// Фиктивный тип, используемый для устранения неоднозначностей
// между конструкторами классов, производных от ShapeRep
enum Exemplar { };

//  ЗАВИСИТ ОТ ПЛАТФОРМЫ И КОМПИЛЯТОРА

class Top {
public:
    // Объекты этого класса не содержат данных кроме поля __vtbl,
    // сгенерированного компилятором. Наследование всех классов
    // от этого класса гарантирует, что в большинстве реализаций
    // __vtbl всегда будет первым элементом любого объекта.
    // Если в вашем компиляторе данное условие не выполняется,
    // __vtbl придется искать другими средствами. Возможно,
    // это потребует изменения реализации (а именно findVtblEntry).

    // Поле __vtbl резервируется для внутреннего
    // использования системой.

    virtual ~Top() { }
    mptr* findVtblEntry(vptp);
    void update(String, String, const char *const = "");
    static void operator delete(void *p) {
        ::operator delete(p);
    }
    // Функция общего назначения doit упрощает управление
    // процессом обновления.
    virtual Top *doit();
protected:
    Top()  { }
    static void *operator new(size_t l) {
```

```
        return ::operator new(l);
    }
private:
    // Сравнение двух указателей на функции.
    int compareFuncs(int, vptp, vptp);
};

typedef Top *Topp;

class Thing: public Top {
    // Все поля "rep" наследуются от Thing; этот класс
    // определяет каноническую форму всех классов писем.
public:
    virtual Thing *type() { return this; }
    Thing() { }
    virtual Thing *cutover();       // Функция обновления поля
    virtual ~Thing() { }            // Деструктор
    int docutover();
};

typedef Thing *Thingp;

//*************************************************************//
//                                                             //
//     Ф А Й Л :    K   C                                      //
//                                                             //
//         Код класса Thing                                    //
//                                                             //
//*************************************************************//

#include "k.h"

Thing *
Thing::cutover() {
    // Функция, предоставляемая пользователем для управления
    // преобразованием старого формата данных класса
    // в новый формат. Функция вызывается для экземпляра
    // старого класса и возвращает экземпляр нового класса.
    return this;
}

int
Thing::docutover() {
    // Возможно, пользователь не захочет преобразовывать
    // некоторые объекты в процессе преобразования данных
    // класса. Функция docutover возвращает для отдельного
    // объекта логическую величину (true или false),
    // указывающую,нужно ли преобразовывать объект в новый
```

```
    // формат.Обычно это делается с объектами, совместно
    // используемыми несколькими классами конвертов:
    // преобразование объекта должно быть выполнено ровно
    // один раз, а НЕ для каждого конверта. Функция docutover
    // определяет, нужно ли преобразовывать общий объект
    // (например, анализом счетчика ссылок,ведением теневого
    // счетчика и т. д.).
    return 1;
}

//*************************************************************//
//                                                             //
//     Ф А Й Л :    T O P . C                                  //
//                                                             //
//         Код класса Top                                      //
//                                                             //
//*************************************************************//

#include "k.h"
#include <sys/types.h>

Top *
Top::doit() {
    // Функция предоставляет удобную точку входа, через
    // которую пользователь может загрузить собственный
    // код для управления обновлением класса.
    return 0;
}

int
Top::compareFuncs(int vtblindex, vptp, vptp fptr) {
    // Функция сравнивает два абстрактных указателя на функции.
    // Первый указатель на функцию характеризуется индексом
    // в таблице виртуальных функций и адресом;
    // второй характеризуется только адресом.
    // Использование параметров зависит от системы.
    // В данном случае адрес первого указателя не требуется.
    return vtblindex == (int)fptr;
}

mptr *
Top::findVtblEntry(vptp f) {
    // Функция ищет в таблице виртуальных функций "текущего"
    // объекта указатель на функцию, равный параметру f,
    // и возвращает адрес его структуры mptr (полное содержимое
    // элемента таблицы виртуальных функций).

    // Указатель mpp содержит адрес указателя на таблицу
    // виртуальных функций; структура наследования гарантирует,
```

```
    // что указатель на таблицу виртуальных функций находится
    // в начале каждого объекта, представляющего интерес
    // для нас (все объекты являются производными от Top).
    mptr ** mpp = (mptr**) this;

    // Разыменование указателя для получения адреса таблицы
    // виртуальных функций, находящейся в начале объекта.
    mptr * vtbl = *mpp;

    printf("Top::findVtblEntry(%d):  vtbl = 0x%x\n", f, vtbl);

    // Перебор таблицы виртуальных функций текущего
    // объекта и поиск в ней заданной функции.
    // Элемент с нулевым значением завершает таблицу,
    // а элемент с нулевым индексом не используется.
    for(int i = 1; vtbl[i].f; ++i ) {
        if (compareFuncs(i, vtbl[i].f, f)) {
            return vtbl + i;
        }
    }
    return 0;
}

// Объявление внешней функции C "load"
extern "C" vptp load(const char *);

void
Top::update(    String filename,
                String fname,
                const char *const TypedefSpec) {

    // Загрузка в память функции fname из файла filename.
    // Последний необязательный параметр задает тип указателя
    // на функцию: он необходим в случае перегрузки функции.
    // При загрузке происходит соответствующее обновление
    // таблицы виртуальных функций. Обновляться могут только
    // виртуальные функции.

    const String temp = "/tmp";

    printf("Top::update(\"%s\", \"%s\", \"%s\")\n",
        (const char *)filename, (const char *)fname,
        (const char *)TypedefSpec);
    String prepname = temp + "/" + "t.c";
    String Typedef, cast = "";
    if (strlen(TypedefSpec)) {
        Typedef = String("// TYPE used to disambiguate\
        overloaded function\n\t\t\ttypedef ") + TypedefSpec;
```

```
    } else {
        Typedef = "typedef vptp TYPE";
        cast = "(vptp)";
    }
    // Вспомогательная функция prepname возвращает адрес
    // обновляемой функции.
    FILE *tempFile = fopen(prepname, "w");
    fprintf(tempFile, "#\
        include \"includes.h\"\n\
        extern vptp functionAddress() {\n\
            %s;\n\
            TYPE retval = %s&%s;\n\
            return (vptp)retval;\n\
        }\n",
        (const char*)Typedef,
        (const char *)cast,
        (const char*)fname);
    fclose(tempFile);

    // Компиляция вспомогательной функции
    String command = String("DIR=`pwd`; cd ") + temp + ";\
        CC +e0 -I$DIR -c -g " + prepname;
    system(command);
    unlink(prepname);
    String objectname =
            prepname(0, prepname.length() - 2) + ".o";

    // Загрузка вспомогательной функции. Вспомните, что
    // функция load возвращает адрес новой функции.
    vvptp findfunc = (vvptp)load(objectname);
    unlink(objectname);
    printf("Top::update: calling findVtblEntry(%d)\n",
                    (*findfunc)());

    // Поиск правильного элемента в таблице виртуальных функций
    // данного класса. Вызов вспомогательной функции сообщает
    // findVtblEntry, какую функцию нужно найти.
    mptr *vtblEntry = findVtblEntry((*findfunc)());

    // Загрузка новой версии функции и сохранение ее адреса
    // в таблице виртуальных функций.
    printf("Top::update: old vtblEntry->f = 0x%x\n",
        vtblEntry->f);
    printf("Top::update: calling load(\"%s\")\n",
        (const char *) filename);
    vtblEntry->f = load(filename);
    printf("Top::update: complete, new vtblEntry->f = 0x%x\n",
        vtblEntry->f);
}
```

```
//************************************************************//
//                                                            //
//    Ф А Й Л :    C O O R D . H                              //
//                                                            //
//         Интерфейс структуры Coordinate                     //
//                                                            //
//************************************************************//

#define _COORDINATE_H

// Простой открытый класс, представляющий декартовы
// координаты на графическом экране.

struct Coordinate {
    int x, y;
    Coordinate(int xx = 0, int yy = 0) { x = xx; y = yy; }
};

//************************************************************//
//                                                            //
//    Ф А Й Л :    E S H A P E . H                            //
//                                                            //
//         Интерфейс класса Shape                             //
//                                                            //
//************************************************************//

// Класс Shape обеспечивает пользовательский интерфейс
// ко всему графическому пакету - все остальные классы
// просто используются во внутренней реализации

#ifndef _SHAPE_H
#define _SHAPE_H
#include "k.h"
#endif
#include "List.h"

// Опережающее объявление
class ShapeRep;

class Shape: public Top {    // Top определяется в k.h
public:
    // Все операции перенаправляются rep
    ShapeRep *operator->() const {
        return rep;
    }

    // Конструкторы и ортодоксальная каноническая форма
    Shape();
    Shape(ShapeRep&);
```

```
    ~Shape();
    Shape(Shape& x);
    Shape& operator=(Shape& x);

    // Получение типа объекта письма
    Top *type();
    // Замена прототипа обновленной версией
    // с обновлением всех существующих экземпляров
    // при помощи функции cutover,
    // предоставленной пользователем
    void dataUpdate(Thingp&, const Thingp);

    // Функция общего назначения, которая может загружаться
    // пользователем для выполнения любых действий
    Top *doit();

    // Уборщик мусора для иерархии Shape
    void gc();

    // Функции регистрации и разрыва связи прототипов с Shape
    void Register(ShapeRep*);
    void UnRegister(ShapeRep*);

    // Инициализация переменных класса
    static void init();    // Инициализация класса
private:
    friend ShapeRep;
    // Оператор new вызывается только из ShapeRep.
    // Оператор delete не используется.
    static void *operator new(size_t s) {
        return ::operator new(s);
    }
    static void operator delete(void *) { }

    // Список всех экземпляров своего класса
    static List<Topp> *allShapes;

    // Список всех прототипов классов, производных от ShapeRep
    static List<Thingp> *allShapeExemplars;

    // Указатель на фактическую реализацию Shape
    ShapeRep *rep;
};

typedef Shape *Shapepointer;

// Указатель на прототип Shape, динамически
// созданный функцией Shape::init
extern Shape *shape;          // Прототип
```

```
class Point: public Shape {
public:
    // ...
    ShapeRep *operator->() const;
    Point();
    ~Point();
} point;

// Пользователь не обязан включать заголовочный файл
// ShapeRep и вообще знать о его существовании.

#ifndef _SHAPEREP_H
#include "shaprp.h"
#endif

//************************************************************//
//                                                            //
//     Ф А Й Л :    S H A P E . C                             //
//                                                            //
//         Реализация класса Shape                            //
//                                                            //
//************************************************************//

#include "Shape.h"
#ifndef _SHAPEREP_H
#include "ShapeRep.h"
#endif
#ifndef _TRIANGLE_H
#include "Triangle.h"
#endif
#ifndef _RECTANGLE_H
#include "Rectangle.h"
#endif
#include "List.h"

// Класс Shape выполняет большую часть работы по управлению
// памятью для всех разновидностей Shape.

// Списки всех существующих экземпляров Shape
List<Topp> *Shape::allShapes = 0;

// Списки всех прототипов, созданных для субклассов
// класса ShapeRep
List<Thingp> *Shape::allShapeExemplars = 0;

// Прототип Shape
extern Shape *shape = 0;
```

```
void
Shape::init() {
    // Функция инициализирует все глобальные структуры данных
    // Shape, чтобы функция main могла управлять порядком
    // инициализации.
    allShapes = new List<Topp>;
    allShapeExemplars = new List<Thingp>;
    shape = new Shape;

    // Управление инициализацией типов ShapeRep
    ShapeRep::init();
    Triangle::init();
    Rectangle::init();
}

Top *
Shape::doit() {
    // Вспомогательная функция, которая может перегружаться
    // пользователями для собственных целей.
    return 0;
}

Shape::Shape() {
    // Конструктор Shape по умолчанию
    Topp tp = this;

    // Регистрация в списке allShapes
    allShapes->put(tp);

    // Фактическая разновидность пока неизвестна:
    // создаем фиктивный объект
    rep = new ShapeRep;
}

Shape::~Shape() {
    Listiter<Topp> tp = *allShapes;
    Topp t;
    for ( tp.reset(); tp.next(t); ) {
        if (t == (Thingp)this) {
            tp.remove_prev(t);
            break;
        }
    }
    if (allShapes->length() == 0) {
        // Завершающая уборка мусора
        gc();
    }
}
```

```
Shape::Shape(Shape &x) {
    // Копирующий конструктор
    Thingp tp = (Thingp) this;

    // Регистрация в списке allShapes
    allShapes->put(tp);

    // Сохранение указателя на поле rep параметра
    rep = x.rep;
}

Shape::Shape(ShapeRep &x) {
    // Построение Shape на базе ShapeRep:
    // требуется в основном для преобразования
    // объектов ShapeRep, построенных классами
    // Triangle, Rect и т. д., в объекты Shape,
    // возвращаемые пользователю
    // (внутренние классы остаются невидимыми
    // для пользователя).
    Topp tp = this;

    // Регистрация в списке allShapes
    allShapes->put(tp);

    // Вызывающая сторона должна предоставить копию!
    rep = &x;
}

Shape&
Shape::operator=(Shape &x) {
    // Присваивание. Не стоит беспокоиться об объекте,
    // на который раньше ссылался указатель rep:
    // он будет уничтожен в ходе уборки мусора.
    rep = x.rep;
    return *this;
}

Top *
Shape::type() {
    // В качестве типа возвращается прототип Shape
    return shape;
}

void
Shape::dataUpdate(Thingp &oldExemplar,
                  const Thingp newExemplar) {
    // Функция обеспечивает замену субклассов ShapeRep.
    // Предполагается, что все виртуальные функции были
    // перекомпилированы и подгружены с новым определением
```

```
    // класса. Также предполагается, что программист
    // предоставил функцию cutover, которая вызывается
    // для старого экземпляра и возвращает указатель
    // на семантически эквивалентную копию.

    ShapeRep* savedExemplar = (ShapeRep*) oldExemplar;
    Topp tp = 0;

    // Создание статической копии для перебора, чтобы
    // исключить из него вновь добавляемые экземпляры.
    // Копия будет уничтожена при выходе из функции.
    List<Topp> staticCopy = *allShapes;

    // Замена прототипа
    oldExemplar = newExemplar;

    // Преобразование всех подобъектов для данного прототипа
    Listiter<Topp> shapeIter = staticCopy;
    for ( shapeIter.reset(); shapeIter.next(tp);  ) {
        Shapepointer sp = (Shapepointer)tp;
        if (sp->rep->type() == (Thingp)savedExemplar) {
            if (sp->rep->docutover()) {
                ShapeRep *oldrep = sp->rep;
                printf("\tchanging shape 0x%x to new format\n",
                    oldrep);
                sp->rep = (ShapeRep*)sp->rep->cutover();

                // Старый подобъект rep необходимо
                // уничтожить вручную:уборщик мусора
                // его не уничтожит, поскольку он выходит
                // за пределы области видимости.
                oldrep->ShapeRep::~ShapeRep();
            }
        }
    }

    // Исключение прототипа из списка всех прототипов Shape
    UnRegister(savedExemplar);
}

void
Shape::gc() {
    // Уборщик мусора Shape обеспечивает уничтожение
    // недоступных объектов всех субклассов
    // ShapeRep. Использует алгоритм Бейкера.

    // Первая половина алгоритма Бейкера: пометка
    // всех доступных объектов
    Listiter<Topp> shapeIter = *allShapes;
    shapeIter.reset();
```

```
    for ( Topp tp = 0; shapeIter.next(tp);  ) {
        Shapepointer sp = (Shapepointer)tp;
        if (sp->rep) {
            sp->rep->mark();
        }
    }

    // Вторая половина алгоритма Бейкера: уничтожение.
    // Каждый подтип самостоятельно освобождает память
    // в собственном пуле, используя функцию gc.
    Listiter<Thingp> shapeExemplarIter = *allShapeExemplars;
    shapeExemplarIter.reset();
    for ( Thingp anExemplar = 0;
                shapeExemplarIter.next(anExemplar);  ) {
        ShapeRep *thisExemplar = (ShapeRep*)anExemplar;
        thisExemplar->gc(0);
    }

    // Перестановка источника и приемника алгоритма Бейкера.
    ShapeRep::FromSpace ^= 1;
    ShapeRep::ToSpace ^= 1;
}

void
Shape::Register(ShapeRep *s) {
    // Функция для регистрации прототипов ShapeRep в Shape
    Thingp tp = s;
    allShapeExemplars->put(tp);
}

void
Shape::UnRegister(ShapeRep *s) {
    // Отмена регистрации прототипа (при замене новой версией).
    Thingp tp = 0;
    Listiter<Thingp> shapeIter = *allShapeExemplars;
    for ( shapeIter.reset(); shapeIter.next(tp); ) {
        if (tp == (Thingp)s) {
            shapeIter.remove_prev(tp);
        }
    }
}

//************************************************************//
//                                                            //
//     Ф А Й Л:    S H A P E R E P . H                        //
//                                                            //
//         Интерфейс класса ShapeRep.                         //
//                                                            //
//************************************************************//
```

```
#define _SHAPEREP_H
#ifndef _SHAPE_H
#include "Shape.h"
#endif
#ifndef _COORDINATE_H
#include "Coordinate.h"
#endif

class ShapeRep: public Thing {
public:
    /* Здесь определяются все пользовательские операции.
     * Благодаря оператору -> не обязательно
     * воспроизводить эту сигнатуру в Shape. Тем не менее,
     * в объявлении поля rep класса Shape должен быть
     * указан соответствующий тип. Операторы присваивания
     * определяются не здесь, а в Envelope.
     *
     * Возвращаемый_тип должен быть либо примитивным типом,
     * либо Shape, либо Shape&, либо конкретным
     * типом данных.
     */
    virtual void rotate(double);
    virtual void move(Coordinate);
    virtual void draw();
    virtual void erase();

    // Конструкторы
    Shape make(Coordinate,Thingp);
    Shape make(Coordinate,Coordinate,Thingp);
    Shape make(Coordinate,Coordinate,Coordinate,Thingp);

    // Функции подменяются в производных классах;
    // по умолчанию они вызывают функции, приведенные выше,
    // с разумным выбором аргумента Thingp.
    virtual Shape make();
    virtual Shape make(Coordinate);
    virtual Shape make(Coordinate,Coordinate);
    virtual Shape make(Coordinate,Coordinate,Coordinate);

    // Функция пометки объекта по алгоритму Бейкера
    virtual void mark();

    // Функция type возвращает указатель на прототип объекта
    Thing *type();

    // Уборка мусора. Если аргумент равен нулю, выполняется
    // очистка текущего пула; в противном случае функция
    // строит новый пул для объектов заданного размера
```

```
    // и забывает про старый пул.
    virtual void gc(size_t = 0);

    // Конструкторы класса
    ShapeRep();
    ~ShapeRep();
    ShapeRep(Exemplar);

    // Функция управляет порядком инициализации
    // статической информации класса.
    static void init();
protected:
    friend class Shape;
    Coordinate center;

    // "Поле типа"
    Top *exemplarPointer;

    // Переменные состояния управления памятью.
    // Переменная space принимает значения FromSpace
    // или ToSpace; переменная gcmark содержит
    // бит пометки объектов по Бейкеру; переменная
    // inUse означает, что объект был возвращен
    // вызывающей стороне для использования и еще
    // не освобожден уборщиком мусора.
    unsigned char space:1, gcmark:1, inUse:1;
protected:
    // Общая обработка для фазы уничтожения
    // уборки мусора по Бейкеру
    static void gcCommon(size_t nbytes,
        const size_t poolInitialized,
        const int PoolSize, Char_p &h);

    // Следующие две "константы" являются признаками
    // источника и приемника в алгоритме Бейкера.
    // При каждом цикле уборки мусора они меняются местами
    static unsigned char FromSpace, ToSpace;

    // Общие операторы управления памятью
    // используют пул, с которым работает gc
    static void *operator new(size_t);
    static void operator delete(void *);

    // Фигура, лишенная типа: используется
    // как возвращаемое значение по умолчанию.
    static Shape *aShape;
};
```

```
//*************************************************************//
//                                                             //
//     Ф А Й Л :    ShapeRep.c                                 //
//                                                             //
//         Код класса ShapeRep                                 //
//                                                             //
//*************************************************************//

// Класс ShapeRep является базовым классом для всех классов
// писем,использующих конверт Shape. Вся логика конкретных
// разновидностей геометрических фигур сосредоточена в классах,
// производных от ShapeRep.

#include "ShapeRep.h"
#ifndef _RECTANGLE_H
#include "Rectangle.h"
#endif
#ifndef _TRIANGLE_H
#include "Triangle.h"
#endif

// "Константа" используется в качестве вырожденного
// возвращаемого значения.
Shape *ShapeRep::aShape = 0;

// Биты, обозначающие источник и приемник
// в алгоритме Бейкера; меняются местами при каждом цикле.
unsigned char ShapeRep::FromSpace = 0, ShapeRep::ToSpace = 1;

void ShapeRep::init() {
    // Инициализация структур данных класса.
    aShape = new Shape;
}

void *
ShapeRep::operator new(size_t l) {
    // Подменяется в производных классах
    return ::operator new(l);
}

void
ShapeRep::operator delete(void *p) {
    // Подменяется в производных классах
    ::operator delete(p);
}

Thing *
ShapeRep::type() {
```

```
    // Тип любого класса, производного от ShapeRep,
    // хранится в exemplarPointer
    return (Thingp)exemplarPointer;
}

Shape
ShapeRep::make() {
    // Производитель объектов ShapeRep по умолчанию
    return *aShape;
}

Shape
ShapeRep::make(Coordinate c1, Thingp) {
    // Одна пара координат: объект Point
    return point->make(c1);
}

Shape
ShapeRep::make(Coordinate c1, Coordinate c2, Thingp type) {
    // Конструктор для объектов Shape с двумя парами
    // координат: отрезки и прямоугольники
    return ((ShapeRep *)type)->make(c1, c2);
}

Shape
ShapeRep::make(Coordinate c1, Coordinate c2, Coordinate c3,
           Thingp type) {
    // Конструктор для объектов Shape с тремя парами
    // координат: треугольники, дуги, параллелограммы
    return ((ShapeRep*)type)->make(c1, c2, c3);
}

Shape
ShapeRep::make(Coordinate c1) {
    // Построение Shape с одной парой координат.
    // По умолчанию создается точка.
    return make(c1, (Thingp)&point);
}

Shape
ShapeRep::make(Coordinate c1, Coordinate c2) {
    // Построение Shape с двумя парами координат.
    // По умолчанию создается прямоугольник.
    return make(c1, c2, rectangle);
}

Shape
ShapeRep::make(Coordinate c1, Coordinate c2, Coordinate c3) {
```

```
    // Построение Shape с тремя парами координат.
    // По умолчанию создается треугольник.
    return make(c1, c2, c3, triangle);
}

void
ShapeRep::gcCommon(size_t nbytes, const size_t poolInitialized,
        const int PoolSize, Char_p &heap) {
    // Уборка мусора, общая для всех ShapeRep.
    // Именно в этой функции выполняется большая часть
    // "уничтожений" во второй половине алгоритма Бейкера.
    // Функция напрямую вызывается функциями gc
    // производных классов с соотвтствующими параметрами.

    // Вычисление размера объекта s. Если задано значение
    // nbytes, использовать его: оно задает изменение размера
    // (или это изменение задается при инициализации).
    // В противном случае использовать значение, хранящееся
    // в poolInitialized.
    size_t s = nbytes? nbytes: poolInitialized;

    // Округление для выравнивания
    size_t Sizeof = Round(s);

    // Если параметр отличен от нуля, он определяет
    // размер объекта; это означает, что мы хотим
    // ликвидировать старый пул и создать новый.
    // Это происходит в начале работы программы
    // и после обновления класса.
    if (nbytes) heap = new char[PoolSize * Sizeof];

    ShapeRep *tp = (ShapeRep *)heap;

    // Уничтожение объектов в пуле
    for (int i = 0; i < PoolSize; i++) {
        switch (nbytes) {
        case 0:   // Нормальный случай уборки мусора
            // Если объект помечен и находится
            // в источнике - уничтожить.
            if (tp->inUse) {
                if (tp->gcmark || tp->space != FromSpace) {
                    // Не уничтожать
                    tp->space = ToSpace;
                } else if (tp != tp->type()) {
                    // Объект необходимо деинициализировать
                    tp->ShapeRep::~ShapeRep();
                    tp->inUse = 0;
                    printf("ShapeRep::gcCommon ");
```

```
                    printf("Reclaimed Shape object %c\n",
                     'A' + (((char *)tp-(char *)heap)/Sizeof));
                }
            }
            break;
        default:   // Инициализация памяти
            tp->inUse = 0;
            break;
        }
        tp->gcmark = 0;
        tp = (ShapeRep*)(Char_p(tp) + Sizeof);
    }
}

ShapeRep::ShapeRep(Exemplar) {
    // Конструктор для построения прототипов ShapeRep.
    // Как уже отмечалось, фиктивный тип Exemplar
    // используется для того, чтобы этот конструктор
    // отличался от конструктора по умолчанию.
    exemplarPointer = this;
    shape->Register(this);
}

void
ShapeRep::rotate(double) {
    // Действует как чисто виртуальная функция
    printf("ShapeRep::rotate(double)\n");
}

void
ShapeRep::move(Coordinate) {
    // Действует как чисто виртуальная функция
    printf("ShapeRep::move(Coordinate)\n");
}

void
ShapeRep::draw() {
    // Действует как чисто виртуальная функция
    // В текущей версии функция отображения
    // просто выводит имя типа.
    printf("<Shape>");
}

void
ShapeRep::erase() {
    // Действует как чисто виртуальная функция
    printf("ShapeRep::erase()");
}
```

```
void
ShapeRep::gc(size_t) {
    // Действует как чисто виртуальная функция
}

void
ShapeRep::mark() {
    // Используется Shape в фазе пометки алгоритма Бейкера
    gcmark = 1;
}

ShapeRep::ShapeRep() {
    // Конструктор по умолчанию
    exemplarPointer=this;
    gcmark=0;
    space=ToSpace;
    inUse=1;
}

ShapeRep::~ShapeRep() {
    // Деструктор ничего не делает
}

//*************************************************************//
//                                                             //
//    Ф А Й Л :    L O A D . C                                 //
//                                                             //
//        Код загрузчика функций C                             //
//                                                             //
//*************************************************************//

#include <a.out.h>
#include <fcntl.h>
#include "mptr.h"
#include "String.h"

static String symtab;
static char y = 'a';

extern "C" vptp load(const char *filename) {
    // Функция load сначала компонует заданный файл в новый
    // a.out, используя предыдущий файл a.out как базу
    // для разрешения символических ссылок.
    // Затем функция открывает новый файл, вычисляет размеры
    // его секций .text и .data, после чего загружает их
    // в выделенный блок памяти. Она работает как динамический
    // загрузчик, предназначенный для взаимодействия с внешним
    // инкрементным компоновщиком.
```

```
int errcode = 0;
String newfile;
char buf[256];
long adx, oadx;
unsigned char *ldadx;
struct exec Exec;
int fd, wc;

// При первом проходе используются разумные значения
// по умолчанию: таблица символических имен находится
// в файле a.out. При каждой подгрузке имя файла
// заменяется тем, которое генерируется компоновщиком.
// В процессе происходит уничтожение старых файлов.

if (!symtab.length()) {
    symtab = "a.out"; newfile = "b.out";
} else {
    symtab = String(++y) + ".out";
    newfile = String(y+1) + ".out";
}

// Выравнивание по границе страницы.
oadx = (long)sbrk(0);
adx = oadx + PAGSIZ - (oadx%PAGSIZ);

// Построение команды загрузки для инкрементной компоновки
// переданного файла .o с текущим файлом a.out.
sprintf(buf, "ld -N -Ttext %X -A %s %s -o %s",
    adx, (const char*)symtab, filename,
    (const char*)newfile);
printf("<%s>\n", buf);
if ((errcode=system(buf)) != 0) {
    printf("load: link edit returned error code %d\n",
        errcode);
}
if (symtab != "a.out") unlink(symtab);

// Открытие файла для загрузки в память.
fd = open(newfile, O_RDONLY);
if (fd < 0) {
    printf("load: open of \"%s\" failed\n", newfile);
    return 0;
}

// Чтение заголовка файла для получения размеров
// секций .text и .data.
read(fd, (char *)&Exec, sizeof(struct exec));
```

```
    // Выделение памяти для новой программы
    sbrk(int(PAGSIZ-(oadx%PAGSIZ)));
    ldadx = (unsigned char *)
        sbrk(int(Exec.a_text + Exec.a_data + Exec.a_bss));

    // Загрузка скомпонованного файла в работающий процесс
    // по только что вычисленному адресу.
    wc = read(fd, (char *)ldadx,
    int(Exec.a_text + Exec.a_data));
    close(fd);

    // Возврат адреса загрузки
    return (vptp) ldadx;
}

//*************************************************************//
//                                                             //
//     Ф А Й Л :    M A I N . C                                //
//                                                             //
//   Управляющая программа для примера с фигурами              //
//                                                             //
//*************************************************************//
#include <a.out.h>
#include <fcntl.h>
#include "Shape.h"
#ifndef _COORDINATE_H
#include "Coordinate.h"
#endif
extern void doClassUpdate();
extern int compile(const String &fileName);
extern int mkfile(const String &fileName,
                  const String &contents);

int main2() {
    Shape::init();
    Coordinate p1, p2, p3;
    Shape object = (*shape)->make(p1, p2, p3);
    printf("object is "); object->draw(); printf("\n");

    // Демонстрация обновления виртуальных функций
    object->move(p1);
    compile("ev2tri.c");
    String include = "includes.h";
    mkfile(include,
    "#include \"ek.h\"\n#include \"ev2tri.h\"\n");
    object->update("v2Triangle.o", "Triangle::move");
    object->move(p1);
    doClassUpdate();
```

```
    object->move(p1);
    {
        Shape object3 = (*shape)->make(p1, p2, p3);
        printf("object3 is "); object3->draw(); printf("\n");
    }
printf("main: making object2\n");
    Shape object2 = (*shape)->make(p1, p2, p3);
    shape->gc();        // Уборка мусора
    printf("object2 is "); object2->draw(); printf("\n");
printf("main: made object2, calling object2->move\n");
    object2->move(p1);
    shape->gc();        // Уборка мусора
printf("exiting\n");
    return 0;
}

int main() {
    int retval = main2();
    shape->gc();
    return retval;
}

void
doClassUpdate() {
    extern Shape *triangle;
    const String include = "includes.h";
    mkfile(include,
    "#include \"ek.h\"\n#include \"ev2tri.h\"\n");

    compile("ev3tria.c");
    (*triangle).update("ev3tria.o", "Triangle::make",
        "Shape (Triangle::*TYPE)()");

    compile("ev3trib.c");
    (*triangle).update("ev3trib.o", "Triangle::make",
        "Shape (Triangle::*TYPE)\
            (Coordinate,Coordinate,Coordinate)");

    compile("ev3trim.c");
    (*triangle).update("ev3trim.o", "Triangle::move");

    compile("v3tric.c");
    (*triangle).update("v3tric.o",
                    "Triangle::cutover");

    mkfile(include, "#include \"ek.h\"\n\
       #include \"ev3tri.h\"\n");
```

```
    mkfile("ev3doit.c",  "#include \"ek.h\"\n\
       #include \"ev3tri.h\"\n\
        Top * Shape::doit() {\n\
            printf(\"v3 Shape::doit (new) called\\n\");\n\
            Thingp Ttriangle = triangle;\n\
            shape->dataUpdate(Ttriangle,\n\
                new Triangle(Exemplar(0)));\n\
            triangle = (ShapeRep*) Ttriangle;\n\
            printf(\"Shape::doit: did data update\\n\");\
            return 0;\n\
        }\n\n\
        Triangle::Triangle(Exemplar e): ShapeRep(e) { }\n");
    compile("ev3doit.c");
    printf("doClassUpdate:\
        calling shape->update(\"ev3doit.o\",\
        \"Shape::doit\")\n");
    shape->update("ev3doit.o", "Shape::doit");
    shape->doit();
    shape->gc();        // Периодическая уборка мусора
    unlink("ev3doit.c");
    unlink("ev3doit.o");
}

#include <sys/stat.h>

int compile(const String& fileName) {
    struct stat dotC, dotO;
    String fileNameDotO =
        fileName(0,fileName.length()-2) + ".o";
    stat(fileName, &dotC);
    stat(fileNameDotO, &dotO);
    if (dotC.st_mtime < dotO.st_mtime) {
        printf("\"%s\" is up to date\n", (const char*)fileName);
        return 0;
    } else {
        String command = String("CC +eO -c -g ") + fileName;
        printf("compile: <%s>\n", (const char*)command);
        return system(command);
    }
}

extern int mkfile(const String &fileName,
                  const String &contents) {
    FILE *inc = fopen(fileName, "w");
    printf("mkfile: creating <%s>\n", (const char *)fileName);
    fprintf(inc, (const char*)contents);
    return fclose(inc);
}
```

```
//************************************************************//
//                                                            //
//    ФАЙЛ:   TRIANGLE.H                                      //
//                                                            //
//        Интерфейс класса Triangle                           //
//                                                            //
//************************************************************//

// Интерфейс класса Triangle, реализующего семантику
// геометрической фигуры "треугольник".

#define _TRIANGLE_H
#ifndef _SHAPEREP_H
#include "ShapeRep.h"
#endif
#ifndef _COORDINATE_H
#include "Coordinate.h"
#endif

class Triangle: public ShapeRep {
public:
    // Конструкторы
    Shape make();
    Shape make(Coordinate, Coordinate, Coordinate);

    // Управление памятью
    void *operator new(size_t);
    void operator delete(void *);
    void gc(size_t = 0);

    // Пользовательская семантика
    void draw();
    void rotate(double);
    void move(Coordinate);

    // Функции класса
    Triangle(Exemplar);
    Triangle();
    static void init();
private:
    // Никогда не должны вызываться
    Shape make(Coordinate);
    Shape make(Coordinate, Coordinate);
private:
    // Переменные состояния экземпляра
    Coordinate p1, p2, p3;
private:
    // Данные управления памятью
```

```
    static char *heap;
    static size_t poolInitialized;
    enum { PoolSize = 10 };
};

// Объявление указателя на прототип Triangle
extern ShapeRep *triangle;

//*************************************************************//
//                                                             //
//    F I L E :    T R I A N G L E. C                          //
//                                                             //
//         Интерфейс класса Triangle                           //
//                                                             //
//*************************************************************//

#include "Triangle.h"

// Переменные, специфические для класса
ShapeRep *triangle = 0;
char *Triangle::heap = 0;
size_t Triangle::poolInitialized = 0;

void
Triangle::init() {
    // Инициализация статических и глобальных переменных,
    // специфических для класса Triangle.
    // Некоторые структуры данных инициализируются
    // при первом вызове ShapeRep::gcCommon
    triangle = new Triangle(Exemplar(0));
}

Shape
Triangle::make()
{
    // Построение вырожденного треугольника
    Triangle *retval = new Triangle;
    retval->p1 = retval->p2 = retval->p3 = Coordinate(0,0);
    retval->exemplarPointer = this;
    return *retval;
}

Shape
Triangle::make(Coordinate pp1, Coordinate pp2, Coordinate pp3)
{
    // Создание и возврат нового объекта Triangle
    Triangle *retval = new Triangle;
    retval->p1 = pp1;
```

```
    retval->p2 = pp2;
    retval->p3 = pp3;
    retval->exemplarPointer = this;
    return *retval;
}

void
Triangle::gc(size_t nbytes) {
    // Передача структур данных Triangle общей функции
    // уборки мусора базового класса.
    gcCommon(nbytes, poolInitialized, PoolSize, heap);
}

void
Triangle::draw() {
    // В этой версии просто выводится информация о положении
    // треугольника в пуле.
    int Sizeof = poolInitialized? poolInitialized:
        Round(sizeof(Triangle));
    printf("<Triangle object %c>",
        'A' + (((char *)this-(char *)heap)/Sizeof));
}

void
Triangle::move(Coordinate) {
    // ...
}

void
Triangle::rotate(double) {
    // ...
}

void *
Triangle::operator new(size_t nbytes) {
    // Если пул еще не инициализировался или класс Triangle
    // только что обновился, управление передается
    // уборщику мусора.
    if (poolInitialized - nbytes) {
        gcCommon(nbytes, poolInitialized, PoolSize, heap);
        poolInitialized = nbytes;
    }

    // Поиск свободного элемента
    Triangle *tp = (Triangle *)heap;
    // Нужно добавить проверку переполнения
    while (tp->inUse) {
        tp = (Triangle*)(((char*)tp) + Round(nbytes));
    }
```

```
    // Инициализация битов памяти
    tp->gcmark = 0;
    tp->inUse = 1;
    return (void*) tp;
}

void Triangle::operator delete(void *) {
    // Оператор delete никогда не должен вызываться, но C++
    // настаивает на его присутствии, если определен
    // оператор new.
}

Triangle::Triangle() { }  // Информация о размере и vptr

Triangle::Triangle(const Triangle& t) {
    // Копирующий конструктор
    p1 = t.p1;
    p2 = t.p2;
    p3 = t.p3;
}

Triangle::Triangle(Exemplar e) : ShapeRep(e) {
    // Построение прототипа Triangle
}

Shape
Triangle::make(Coordinate) {
    // Фиктивная функция - никогда не должна вызываться.
    return *aShape;
}

Shape
Triangle::make(Coordinate, Coordinate) {
    // Фиктивная функция - никогда не должна вызываться.
    return *aShape;
}

//*************************************************************//
//                                                             //
//    Ф А Й Л :    V 2 T R I A N G L E. C                      //
//                                                             //
//        Измененный код класса Triangle (версия 2)            //
//                                                             //
//*************************************************************//

#include "v2Triangle.h"

void
Triangle::move(Coordinate)
```

```
{
    printf("version 2 Triangle::move of size %d\n",
        sizeof(*this));
}

//*************************************************************//
//                                                             //
//    Ф А Й Л :    V 2 T R I A N G L E. H                      //
//                                                             //
//        Измененный интерфейс класса Triangle (версия 2)      //
//                                                             //
//*************************************************************//

#define _TRIANGLE_H
#ifndef _SHAPEREP_H
#include "ShapeRep.h"
#endif
#ifndef _COORDINATE_H
#include "Coordinate.h"
#endif
class Triangle: public ShapeRep {
public:
    // Соответствует Triangle.h. но с добавлением нового
    // определения Triangle::move
    Shape make();
    Shape make(Coordinate, Coordinate, Coordinate);
    Triangle();
    void draw();
    void move(Coordinate);
    void rotate(double);
    void *operator new(size_t);
    void operator delete(void *);
    void gc(size_t = 0);
    Triangle(Exemplar);
    static void init();
private:
    static void poolInit(size_t);
    Shape make(Coordinate) { return *aShape; }
    Shape make(Coordinate, Coordinate) { return *aShape; }
    Coordinate p1, p2, p3;
private:
    static char *heap;
    static size_t poolInitialized;
    enum { PoolSize = 10 };
};

extern ShapeRep *triangle;
```

```
//*************************************************************//
//                                                             //
//     Ф А Й Л :    V 3 T R I A N G L E. H                     //
//                                                             //
//         Структуры данных для версии 3 класса Triangle       //
//                                                             //
//*************************************************************//

#define _TRIANGLE_H
#ifndef _SHAPEREP_H
#include "ShapeRep.h"
#endif
#ifndef _COORDINATE_H
#include "Coordinate.h"
#endif

// Объявление НОВОЙ (третьей) версии класса Triangle
// с добавлением атрибута color

class Triangle: public ShapeRep {
public:
    Shape make();
    Shape make(Coordinate, Coordinate, Coordinate);
    Triangle();
    void draw();
    void move(Coordinate);
    void rotate(double);
    void *operator new(size_t);
    void operator delete(void *);
    void gc(size_t = 0);
    Thing *cutover();
    Triangle(Exemplar);
    static void init();
private:
    static void poolInit(size_t);
    Shape make(Coordinate) { return *aShape; }
    Shape make(Coordinate, Coordinate) { return *aShape; }
    Coordinate p1, p2, p3;
    enum Color { Black, White } color;
private:
    static char *heap;
    static size_t poolInitialized;
    enum { PoolSize = 10 };
};

extern ShapeRep *triangle;
```

```
//*************************************************************//
//                                                             //
//    Ф А Й Л :    V 3 T R I A N G L E A. C                    //
//                                                             //
//         Код версии 3 класса Triangle                        //
//                                                             //
//*************************************************************//

#include "v3Triangle.h"

Shape
Triangle::make()
{
    printf("Triangle::make() entered\n");
    Triangle *retval = new Triangle;;
    retval->p1 = retval->p2 = retval->p3 = Coordinate(0,0);
    retval->exemplarPointer = this;
    color = Black;
    return Shape(*retval);
}

// Статически расширяется для использования в приведенной
// выше функции make
Triangle::Triangle() { }  // Информация о размере и vptr

//*************************************************************//
//                                                             //
//    Ф А Й Л :    V 3 T R I A N G L E B. C                    //
//                                                             //
//         Код версии 3 класса Triangle                        //
//                                                             //
//*************************************************************//

#include "v3Triangle.h"

Shape
Triangle::make(Coordinate pp1, Coordinate pp2, Coordinate pp3)
{
    Triangle *retval = new Triangle;
    retval->p1 = pp1;
    retval->p2 = pp2;
    retval->p3 = pp3;
    retval->exemplarPointer = this;
    return *retval;
}

Triangle::Triangle() { }  // Информация о размере и vptr
```

```
//************************************************************//
//                                                            //
//    Ф А Й Л :    V 3 T R I A N G L E C U T O V E R . C      //
//                                                            //
//    Управление преобразованием для версии 3 кода Triangle   //
//                                                            //
//************************************************************//

#include "v3Triangle.h"
#include "Map.h"

// ------------------------------------------------------------

class v2Triangle: public ShapeRep {
public:
    Shape make();
    Shape make(Coordinate, Coordinate, Coordinate);
    v2Triangle();
    void move(Coordinate);
    void *operator new(size_t);
    void operator delete(void *);
    void gc(size_t = 0);
    void draw();
    v2Triangle(Exemplar);
    static void init();
private:
    friend Thing *Triangle::cutover();  // Для преобразования
    static void poolInit(size_t);
    Shape make(Coordinate) { return *aShape; }
    Shape make(Coordinate, Coordinate) { return *aShape; }
    Coordinate p1, p2, p3;
};

// ------------------------------------------------------------

// Отображение содержит информацию обо всех старых объектах,
// которые мы должны преобразовать, и о новых объектах,
// в которые они преобразуются. В этом случае многократные
// запросы на преобразование одного и того же объекта
// будут отображены на одно возвращаемое значение.

Map<Thingp, Thingp> objectMap;

Thing *
Triangle::cutover() {
    // Функция вернет указатель на преобразованный треугольник
    Triangle *retval = this;

    // Переданный экземпляр в действительности представляет
    // старый объект Triangle. Объявление старой версии
    // сохраняется под именем v2Triangle; класс Triangle
```

```
    // определяется в версии 3.
    v2Triangle *old = (v2Triangle *)this;
    Thingp oldtp = this;
    ShapeRep *oldsr = (ShapeRep*)this;

    if (objectMap.element(oldtp)) {
        // Если объект уже был преобразован, не преобразовывать
        // его заново -просто вернуть старое преобразованное
        // значение
        retval = (Triangle*)(objectMap[oldtp]);
    } else {
        // Создание треугольника новой (третьей) версии.
        // Сохранение нескольких указателей разных типов.
        retval = new Triangle;
        ShapeRep *newsr = retval;
        Thingp newtp = retval;

        // Копирование подобъекта базового класса (ShapeRep)
        *newsr = *oldsr;

        // Инициализация полей нового объекта
        retval->exemplarPointer = triangle;
        retval->p1 = old->p1;
        retval->p2 = old->p2;
        retval->p3 = old->p3;
        retval->color = Black;

        // Сохранить преобразованный объект на будущее
        objectMap[oldtp] = newtp;
    }
    return retval;
}

Triangle::Triangle() { }

//************************************************************//
//                                                            //
//     Ф А Й Л :    V 3 T R I A N G L E M O V E . C           //
//                                                            //
//     Код реализации move версии 3 класса Triangle           //
//                                                            //
//************************************************************//

#include "v3Triangle.h"

void
Triangle::move(Coordinate)
{
    printf("version 3 Triangle::move of size %d\n",
        sizeof(*this));
}
```

Приложение E
Блочно-структурное программирование на C++

Нисходящее проектирование — проверенная временем методика программирования, которая лежит в основе большинства методов объектно-ориентированного проектирования, использовавшихся за два последних десятилетия. Объектно-ориентированное проектирование предлагалось как замена этой методики, особенно при моделировании больших, сложных систем. Тем не менее, методы нисходящего проектирования (такие, как функциональная декомпозиция) по-прежнему могут успешно применяться, если алгоритмы хорошо понятны, или же задача достаточно автономна, а структура ее решения известна заранее.

На уровне языка поддержка нисходящего проектирования воплощена в блочно-структурном программировании. В настоящем приложении показано, как применение некоторых идиом C++ предоставляет в распоряжение программиста интересную разновидность блочно-структурного программирования. Эти идиомы требуют более точного указания областей видимости, чем в языке Паскаль или Modula-2, что объясняется спецификой ограничения доступа в C++. Возможно, кто-то из читателей не будет полностью удовлетворен результатом — все зависит от личного вкуса и наклонностей. Однако показанное решение можно настроить по своему желанию, и здесь будут представлены некоторые варианты.

Макросы препроцессора C определяются как часть языка C++. В этой главе читатель получит представление о том, как комбинирование макросов с конструкциями C++ обеспечивает функциональность и выразительность, недоступные для «базового языка C++».

E.1. Концепция блочно-структурного программирования

Вероятно, нисходящее проектирование является основным методом структурирования программ, используемым в наше время. А из всех приемов нисходящего проектирования на практике чаще всего встречается *функциональная декомпозиция*, или *пошаговое уточнение*, идея которого была предложена Никласом Виртом в начале 1970-х годов. При функциональной декомпозиции система изначально

характеризуется некоторой высокоуровневой функцией. Затем эта функция разбивается на составные части, каждая из которых проходит дальнейшую декомпозицию, и т. д. Конечный результат представляет собой набор модулей с четко определенной семантикой, напрямую реализуемых программистом.

При функциональной декомпозиции данным отводится безусловно вторичная роль. На каждом уровне уточнения для каждой процедурной единицы проектируется структура данных. Она определяет архитектуру модулей следующего уровня. Процедуры рассматриваются как «владельцы» этих данных. Жизненный цикл блока данных совпадает с жизненным циклом стекового кадра той процедуры, которой он принадлежит. Область видимости блока данных также определяется структурой соответствующей процедуры.

За прошедшие годы во многих языках была реализована прямая поддержка нисходящего проектирования. Одним из важнейших языковых средств поддержки нисходящего проектирования является *блочно-структурное программирование*. Языки с поддержкой блочно-структурного программирования содержат конструкции изменения видимости, позволяющие вкладывать процедуры друг в друга. Языки C и C++ называются блочно-структурными, но только в ограниченном смысле. Хотя они отделяют локальные переменные от глобальных и допускают вложение блоков объявлений в процедурах, их конструкции вложения блоков не распространяются на процедуры. По этому критерию ни C, ни C++ не являются полноценными языками блочно-структурного программирования, в отличие от таких языков, как Алгол 68, PL/1, Modula-2, Ada и Паскаль.

Но даже простые средства C++ при творческом применении создают основу для построения блочно-структурных программ. Элементы стиля и приемы, используемые для имитации блочно-структурного программирования, мы будем называть *блочно-структурной идиомой*. Сначала эта идиома описывается в ограниченной форме, предоставляющей доступ к символическим именам соседних блоков, а затем расширяется до более универсальной формы.

ПРИМЕЧАНИЕ

Блочно-структурное программирование хорошо подходит для прямолинейных задач, в которых алгоритмы и процедуры доминируют над отношениями и структурами данных. Используйте его для задач с последовательной логикой, решения которых состоят не более чем из нескольких сотен строк кода, а алгоритм достаточно понятен.

E.2. Основные строительные блоки структурного программирования на C++

Чтобы строить на C++ иллюзию блочно-структурного программирования, необходимо уметь создавать для каждой функции область видимости (или пространство имен), которая также может содержать новые функции. Структуры (struct) хорошо отвечают этим требованиям. В C++, в отличие от C, объявления структур могут содержать функции, причем функции, локальные по отношению к структуре (которая локальна по отношению к своей функции), обладают соответствующей

видимостью. Кроме того, структуры могут использоваться для инкапсуляции архитектуры данных на каждом уровне функциональной декомпозиции. Вскоре вы убедитесь, что эта инкапсуляция упрощает анализ блочно-структурных программ C++ по сравнению с их аналогами из языка Алгол.

Хотя структуры обеспечивают необходимую семантику для поддержки блочно-структурного программирования, синтаксис программы загромождается и выглядит неестественно по сравнению с процедурными блоками языков Алгол и Паскаль. Макросы препроцессора помогают донести намерения программиста до читателя программы. С их помощью программист обозначает границы новых областей видимости, создаваемых в каждом теле функции. Мы разместим этим макросы в заголовочном файле **block.h**:

```
// Файл block.h

#include <generic.h>

#define LocalScope(function)           \
  struct name2(function,ForLocalScope) { public
#define EndLocalScope } local
```

Объявление **LocalScope** просто содержит открывающие элементы объявления **struct**. Макрос **name2**, определенный в **generic.h**, соединяет два своих аргумента в одно новое имя. Параметр макроса **LocalScope** должен определять имя функции, внутри которой он находится. Макрос **EndLocalScope** завершает объявление **struct** и объявляет экземпляр структуры с именем **local**, которое используется для обращения к контексту, созданному этой структурой.

В листинге E.1 приведен пример простой блочно-структурной реализации алгоритма пузырьковой сортировки [1]. Алгоритм пузырьковой сортировки декомпозируется до тех пор, пока текущий модуль **CompareExchange** не будет выделен в самостоятельную функцию, абстрагированную внутри **BubbleSort**. Функция **BubbleSort** управляет вызовами **CompareExchange** и организует работу алгоритма сортировки.

Листинг E.1. Блочно-структурная реализация пузырьковой сортировки

```
#include <block.h>
#include <iostream.h>
#include <string.h>

void BubbleSort(int n, char *records[], char *keys[])
{
  // Алгоритм пузырьковой сортировки по [1]
  LocalScope(BubbleSort):
    int bound, t;
    void CompareExchange(int j, char *records[],
                         char *keys[]) {
      if (::strcmp(keys[j], keys[j+1]) > 0) {
        char *temp = records[j];
        records[j] = records[j+1];
        records[j+1] = temp;
```

продолжение

Листинг E.1 (*продолжение*)

```
        temp = keys[j];
        keys[j] = keys[j+1];
        keys[j+1] = temp;
        t = j;
      }
    }
  EndLocalScope:

  local.bound = n;
  do {
    local.t = -1;
    for (int j = 0; j < local.bound-1; j++) {
      local.CompareExchange(j, records, keys);
    }
    local.bound = local.t + 1;
  } while (local.t != -1);
}
```

Обратите внимание: хотя функция **BubbleSort** вызывается как обычная функция, функция **CompareExchange** не видна за пределами **BubbleSort**. Блочное структурирование обеспечивает сокрытие информации на процедурном уровне, а алгоритмические подробности инкапсулируются в абстракциях более высокого порядка.

Ниже приводится пример использования приведенной реализации пузырьковой сортировки:

```
char *records[] = {
  "Stroustrup, Bjarne",
  "Lippman, Stan",
  "Hansen, Tony",
  "Koenig, Andrew"
};

char *keys[] = {
  "bs",
  "stan",
  "hansen",
  "ark"
};

int main() {
  for (int i = 0; i < 4; i++) {
    cout << records[i] << endl;
  }
  BubbleSort(4, records, keys);
  for (i = 0; i < 4; i++) {
    cout << records[i] << endl;
  }
  return 0;
}
```

При проектировании иерархий функциональной декомпозиции стоит руководствоваться несколькими практическими правилами. Правило первое — объявляйте локальные переменные функции в ее блоке LocalScope. Размещение «локальных» переменных в этих блоках делает их доступными как для функции, создавшей блок, так и для функций следующего уровня функциональной декомпозиции. Например, переменные t и bound в приведенной выше программе находятся в области видимости BubbleSort; они «принадлежат» BubbleSort, но доступны для CompareExchange. Обратите внимание на уточнение обращений из BubbleSort префиксом local и оператором . (точка); это синтаксическая аномалия, обусловленная реализацией идиомы в C++.

Такой подход объясняется тем, что в блочно-структурных языках функции внутреннего блока могут изменять переменные внешнего блока, включая переменные, локальные для вмещающей функции. В некотором смысле это нарушает инкапсуляцию вмещающей функции. Функциональная декомпозиция естественным образом создает логические связи, направленные от корня иерархии к листьям, и абстракции нижних уровней зависят от абстракций верхних уровней. Тем не менее, мы не хотим, чтобы высокоуровневые абстракции слишком сильно зависели от низкоуровневых. Переменные процедуры принадлежат только ей, и любые обращения к этим переменным со стороны низкоуровневых, более детализированных процедур должны легко обнаруживаться простым просмотром программного кода. Обращения к локальным переменным из вложенных блоков нарушают «принцип наименьшей неожиданности» в эволюции и сопровождении программы.

Представленное решение ограничивает доступ к переменным блока так, что обращения к ним могут производиться только из области видимости «владельца» или максимум со следующего уровня. Следующий уровень (CompareExchange) напрямую виден из текущего (BubbleSort), а это означает, что анализ переменных области видимости не потребует долгих поисков.

К некоторым тривиальным переменным (например, счетчикам циклов) это не относится. Эти переменные не являются частью архитектуры данных, поэтому с ними можно обойтись неформально с использованием стандартного синтаксиса C++.

Другое полезное правило гласит, что предпочтительным механизмом обращения к данным другого уровня является передача параметров. Для передачи данных между вложенными блоками могут использоваться глобальные переменные или данные в LocalScope, но это создаст проблемы с сопровождением, когда потребуется найти все ссылки на такие данные в программе. Локализация данных во вложенном блоке помогает, но данные блока остаются доступными для любых функций внутри этого блока. Передача данных в параметрах и возврат результатов в возвращаемых значениях делают интерфейс более четким. Более того, это необходимо для переменных, не входящих в LocalScope; например, параметры BubbleSort недоступны напрямую для CompareExchange. Чтобы сделать их доступными, требуется передать переменные в качестве параметров, как это сделано в примере

пузырьковой сортировки. Это способствует повышению качества архитектуры и организации хорошо определенных, четко выраженных интерфейсов между модулями.

Е.3. Альтернативное решение с глубоким вложением областей видимости

Блочно-структурная идиома предоставляет доступ только к соседней области видимости, хотя и с несколько неуклюжим синтаксисом. В большинстве блочно-структурных языков символические имена доступны во всех областях видимости, вложенных по отношению к их области видимости, однако описанная идиома этой возможности не поддерживает. В нашем случае символические имена доступны только из их «собственной» области видимости (как переменные bound и t доступны из BubbleSort с префиксом local.) и из функций следующего уровня, содержащихся в этой области (как те же символические имена доступны для CompareExchange). В жестких рамках конструкций C++ было бы трудно обеспечить полную доступность из вложенных областей видимости, характерную для блочно-структурных языков. Можно возразить, что представленное решение делает программу более понятной, чем, скажем, в языке Паскаль: использование символического имени не может быть удалено от его объявления. В языках Паскаль и Modula допускается объявление символического имени в нескольких уровнях видимости от места его использования. Тем не менее, если бы мы смогли обеспечить полный доступ из вложенных областей видимости, выраженный таким способом, который бы упрощал поиск объявления символического имени, и реализовать его в рамках C++, то в нашем распоряжении появилась бы идиома, сочетающая в себе основные свойства явного уточнения видимости и вложения структур. В качестве примера такого решения ниже приводится динамическая *блочно-структурная идиома.*

ПРИМЕЧАНИЕ

Задействуйте эту идиому в тех же случаях, когда используется блочно-структурная идиома (с. 452), но при необходимости доступа к символическим именам из глубоко вложенных областей видимости. Такая необходимость часто возникает при функциональной декомпозиции, поскольку вложение абстракций архитектуры данных часто противоречит процедурным абстракциям. (Если вам потребуется решение, более динамическое и одновременно более безопасное по отношению к типам, обратитесь к разделу «Упражнения» в конце этого приложения.)

В качестве примера рассмотрим программу, приведенную в конце приложения — частичную реализацию простой видеоигры. Правила игры иллюстрирует рис. Е.1. В игре используются клавиатура и примитивный дисплей, способный отображать символы (но не графику) в произвольной позиции экрана. Показанный на рисунке игровой экран содержит стену, состоящую из символов W, биту (символы b) и шарик (символ o). Шарик двигается по экрану вверх, вниз и по диагонали, изменяя направление полета при столкновениях со стеной или битой. Игрок перемещает биту влево и вправо клавишами l и r. Его задача — проломить шариком стену [2].

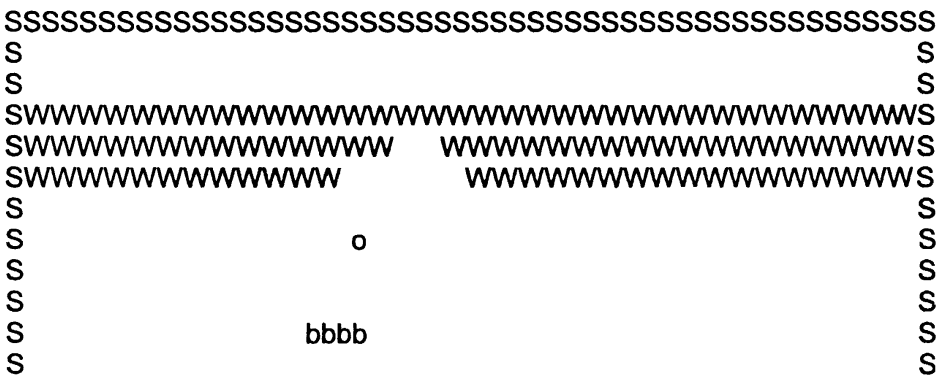


Рис. E.1. Иллюстрация видеоигры

Этот пример нисходящего проектирования заметно сложнее пузырьковой сортировки; его структура изображена на рис. E.2. В программе будут добавлены специальные средства, при помощи которых пользователь сможет получить доступ к внешним областям видимости.

Начнем с построения примитивного механизма областей видимости на уровне макросов. Наш механизм областей видимости поддерживает *динамическую цепочку*, или *список истории*, вызовов функций для текущей области видимости. Программист может перебрать элементы списка в обратном направлении, обратиться к любым данным или вызвать любые функции внешних областей. Для решения этой задачи в макросе LocalScope определяется конструктор области видимости, автоматически вызываемый при входе в нее (см. 2.3). По правилам имя конструктора совпадает с именем структуры, а его работа обычно сводится к инициализации состояния новых переменных, созданных для данного типа структуры. Конструктор получает один параметр — переменную this, которая инициализируется компилятором и содержит указатель на внешний объект. Новый файл block.h начинается так:

```
#include <generic.h>

#define LocalScope(function)              \
  struct name2(function,ForLocalScope) {     \
    void *parent,                 \
    name2(function,ForLocalScope)(void *p)   \
      { parent = p; }             \
  public
#define EndLocalScope                \
  } local(this)
```

Макрос LocalScope дополнен объявлением parent (указателем на родительскую область видимости) и конструктором, инициализирующим parent. Обратите внимание на следующий вызов макроса:

```
LocalScope(SomeFunction)
```

Этот вызов расширяется во фрагмент кода, эквивалентный следующему:

```
struct SomeFunctionForLocalScope {
  void *parent;
  SomeFunctionForLocalScope(void *p) { parent = p; }
```

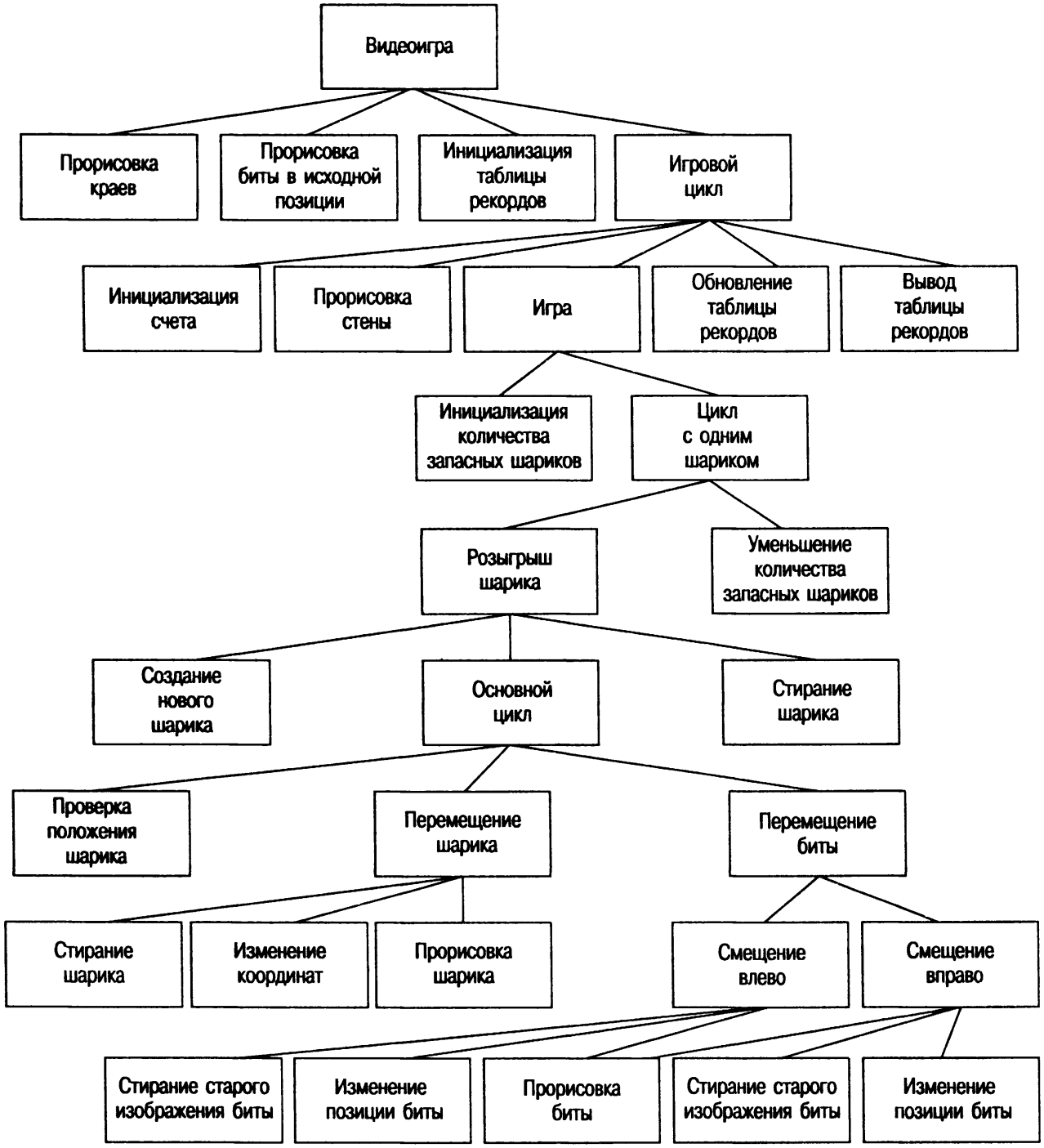


Рис. Е.2. Иерархическая диаграмма для видеоигры

Макрос EndLocalScope завершает объявление структуры, содержащей информацию области видимости, и определяет переменную local для хранения информации о ее состоянии. Переменная local инициализируется значением this в EndLocalScope. Переменная this ссылается на вмещающую область видимости, потому что внутренний класс находится внутри функции вмещающей области видимости. В свою очередь, значение local передается в качестве параметра конструкторам всех внутренних областей. Именно этот параметр устанавливает связь области видимости с ее вмещающей областью (то есть вмещающим объектом local).

«Крайний» внешний уровень должен обрабатываться особым образом, потому что он не имеет вмещающего объекта LocalScope. Внешний блок LocalScope имеет одну внешнюю область видимости (глобальную), но между ними не существует формальной связи. Глобальные переменные не подчиняются ограничениям блочной видимости — к ним всегда можно обращаться без уточнения в любой точке программы. Поскольку крайний внешний блок LocalScope не может создать связь с указателем this вмещающего блока, он должен иметь собственный макрос-завершитель, который попросту обнуляет указатель на родительскую область видимости. Назовем этот макрос EndOuterScope:

```
#define EndOuterScope } local(0)
```

Обращение к родительской области видимости производится с помощью макроса Parent. Это не более чем синтаксическое украшение для преобразования указателя void* parent и разыменования одного из его полей:

```
#define Parent(type,c)  \
  ((name2(type,ForLocalScope)*)(c)->parent)
```

Для обращения к внешней области видимости в первом параметре макроса Parent передается имя функции внешней области видимости, а во втором — указатель this. Типом возвращаемого значения является указатель на объект области видимости нужной функции:

```
outerProcLocalScope *scopePtr = Parent(outerProc,this);
```

Макрос Parent возвращает указатель this для своей вмещающей области видимости. Его последовательное применение позволяет обращаться к областям видимости более высоких уровней:

```
PlayGameLocalScope *p  = Parent(PlayGame, this);
BallGameLocalScope *p2 = Parent(BallGame, p);
int &y = p2->ball.yposition;
```

Простая программа в листинге E.2 показывает, как работает механизм многоуровневых областей видимости. Она выдает следующий результат:

```
bar: a=5, 3, 1; b=6, 4, 2
foo: a=1, b=2
```

Листинг E.2. Многоуровневые обращения к внешним областям видимости

```
int main() {
  LocaScope(main);
    int a;
    int b;
      LocalScope(foo);
        int a, b;
        void bar() {
          local.a = 3; local.b = 4;
          a = 5; b = 6;
          printf("bar: a=%d, %d, %d; b=%d, %d, %d\n",
```

продолжение

Листинг Е.2 (*продолжение*)

```
            a, local.a,
            Parent(main.this)->a,
            b, local.b,
            Parent(main.this)->b);
        }
      EndLocalScope;
      local.bar();
      printf("foo: a=%d, b=%d\n", a, b);
    }
  EndOuterScope;
  local.a = 1;
  local.b = 2;
  local.foo();
}
```

Е.4. Проблемы реализации

В C++ не существует синтаксиса раздельного определения или компиляции тела функции класса, вложенного в другую функцию. Следовательно, определения функций должны находиться внутри того класса, в котором они объявлены. А это означает, что ваша среда компиляции C++ может интерпретировать тела таких функций как подставляемые. Многие среды компиляции C++ выполняют подстановку только для достаточно простых функций; более сложные функции компилируются как статические, и их использование сопряжено с теми же затратами, что и вызов обычных функций.

Если ваша среда компиляции C++ предоставляет такую возможность, подумайте об отключении подстановки при использовании этой идиомы. Если этого не сделать, для вызовов функций будет сгенерирован код, многократно превышающий их объем, и объектный модуль станет слишком громоздким. Если в вашей среде компиляции имеется эвристический анализатор, предотвращающий подстановку слишком больших функций, возможно, удастся обойтись и без этого.

Может показаться, что невозможность раздельной компиляции вложенных функций создает некоторые неудобства. В самом деле, компиляция таких функций требует затрат на компиляцию их вмещающих функций. На практике это ограничение ни на что не влияет. В большинстве компиляторов блочно-структурных языков раздельная компиляция вложенных функций либо отсутствует вообще, либо считается возможностью крайне экзотической.

Упражнения

1. Поскольку идиома, представленная в настоящем приложении, базируется на преобразовании данных void* в различные классы, она небезопасна по отношению к типам. Передача макросу неверного имени структуры приведет к использованию ошибочного значения во время выполнения, причем без выдачи

каких-либо предупреждений на стадии компиляции или выполнения. Результаты выполнения становятся непредсказуемыми и почти наверняка ошибочными. Рассмотрим следующую альтернативу файла block.h:

```
#include <generic.h>
#include <string.h>
#include <iostream.h>

struct Scope {
  Scope *parent;
  virtual char *ScopeName() = 0;
};

#define LocalScope(function)                    \
struct name2(function,ForLocalScope):public Scope {       \
  char *ScopeName() { return "function"; }           \
  name2(function,ForLocalScope)(Scope *p) { parent = p; }
  public
#define EndLocalScope } local(this)
#define EndOuterScope } local(0)
#define Parent(type,c)
  ((name2(type,ForLocalScope)*)_Parent("type",c))

inline Scope *_Parent(const char * type, Scope *c)
{
  register Scope *s = c;
  while(s && strcmp(s->ScopeName(), type)) s = s->parent;
  if (!s) cerr << "couldn't find scope " << type << endl;
  return s;
}
```

Используйте эту версию заголовочного файла для компиляции и запуска программы, приведенной в конце приложения. Какую дополнительную защиту она обеспечивает по сравнению с предыдущей версией файла block.h?

2. Измените макросы из упражнения 1 так, чтобы они обеспечивали трассировочную печать на стадии выполнения. Режим трассировки должен включаться или выключаться по желанию пользователя.
3. Реализуйте файл block.h с применением макросов m4 [3] вместо макросов препроцессора C (или воспользуйтесь любым другим достаточно мощным макроязыком). Используя дополнительные возможности m4, сведите к минимуму количество вызовов Parent, необходимых для обращения к некоторой области видимости, а также число параметров макроса Parent.

Код блочно-структурной видеоигры

```
#include <block.h>
#include <iostream.h>
#include <curses.h>
```

```
int main()
{
  LocalScope(BallGame):
    void DrawSides() {
      for(int i = 0; i < 66; i++) mvaddch(0, i, 'S');
      for(i = 0; i < 13; i++) {
        mvaddch(i, 0, 'S');
        mvaddch(i, 65, 'S');
      }
      leftWall = 0;
      rightWall = 65;
    }

    struct Ball {
      int yposition, xposition;
      int yspeed, xspeed;
      Ball() {
        yposition = 6;
        xposition = 33;
        yspeed = 1;
        xspeed = 1;
      }
    } ball;

    struct Bat {
      int yposition, position, length, xspeed;
      Bat() { position = 40;
        yposition = 11;
        length = 4;
        xspeed = 0;
      }
    } bat;

    void DrawBat() {
      for (int i = bat.position;
              i < bat.position + bat.length; i++) {
        mvaddch(bat.yposition, i, 'b');
      }
    }

    void EraseBat() {
      for (int i = bat.position;
              i < bat.position + bat.length; i++) {
        mvaddch(bat.yposition, i, ' ');
      }
    }
```

```
short leftWall, rightWall;

int score;

void DrawWall() {
  for(int i = 0; i < 66; i++) {
    mvaddch(3, i, 'W');
    mvaddch(4, i, 'W');
    mvaddch(5, i, 'W');
  }
}

char ballIsInPlay() {
  return ball.xposition < rightWall &&
    ball.xposition > leftWall &&
    ball.yposition <= bat.yposition;
}

void PlayGame() {
  LocalScope(PlayGame):
    char key;  // Вводится с клавиатуры

    short ballsLeft;
    void PlayABall() {
      LocalScope(PlayABall):
        void CheckBallPosition() {
          PlayGameLocalScope *p =
                    Parent(PlayGame, this);
          BallGameLocalScope *p2 =
                    Parent(BallGame, p);
          int &y = p2->ball.yposition;
          int &x = p2->ball.xposition;
          if (x <= p2->leftWall+1 ||
            x >= p2->rightWall-1) {
            p2->ball.xspeed = -p2->ball.xspeed;
          }
          if (y <= 0) {
            p2->ball.yspeed = -p2->ball.yspeed;
          }
          char c = mvinch(y + p2->ball.yspeed,
                 x + p2->ball.xspeed);
          switch (c) {
          case 'W':
            mvaddch(y + p2->ball.yspeed,
              x + p2->ball.xspeed, ' ');
            p2->score++;
          case 'b':
```

```
    p2->ball.yspeed = -p2->ball.yspeed;
    p2->ball.xspeed = p2->bat.xspeed;
    break;
  }
}

void MoveBall() {
  PlayGameLocalScope *p =
          Parent(PlayGame, this);
  BallGameLocalScope *p2 =
          Parent(BallGame, p);
  mvaddch(p2->ball.yposition,
    p2->ball.xposition, ' ');
  p2->ball.xposition += p2->ball.xspeed;
  p2->ball.yposition += p2->ball.yspeed;
  mvaddch(p2->ball.yposition,
    p2->ball.xposition, 'O');
}

void MoveBat(char key) {
  LocalScope(MoveBat);
    void MoveLeft(Bat& bat) {
      PlayABallLocalScope *p =
            Parent(PlayABall, this);
      PlayGameLocalScope *p2 =
            Parent(PlayGame, p);
      Parent(BallGame, p2)->EraseBat();
      if (bat.position >
        Parent(BallGame, p2)->leftWall) {
        bat.position--;
      }
      Parent(BallGame, p2)->DrawBat();
    }

    void MoveRight(Bat& bat) {
      PlayABallLocalScope *p =
            Parent(PlayABall, this);
      PlayGameLocalScope *p2 =
            Parent(PlayGame, p);
      Parent(BallGame, p2)->EraseBat();
      if (bat.position <
        Parent(BallGame, p2)->rightWall -
        bat.length) {
        bat.position++;
      }
      Parent(BallGame, p2)->DrawBat();
    }
  EndLocalScope;
```

```
            PlayGameLocalScope *p =
                    Parent(PlayGame, this);
            switch (key) {
            case 'l':
              local.MoveLeft(
                    Parent(BallGame, p)->bat);
              Parent(BallGame, p)->bat.xspeed = -1;
              break;
            case 'r':
              local.MoveRight(
                    Parent(BallGame, p)->bat);
              Parent(BallGame, p)->bat.xspeed = 1;
              break;
            default:
              break;
            }
          }
        EndLocalScope;

        while (Parent(BallGame, this)->ballIsInPlay()) {
          local.CheckBallPosition();
          local.MoveBall();
          refresh();
          key = getch();
          local.MoveBat(key);
        }
      }
    EndLocalScope;

    local.ballsLeft = 4;
    while (local.ballsLeft > 0) {
      refresh();
      local.key = getch();
      local.PlayABall();
      --local.ballsLeft;
    }
  }

EndOuterScope;

int bestScore = 0;

initscr();
cbreak();
noecho();

local.DrawSides();
local.DrawBat();
```

```
  for (;;) {
    local.score = 0;
    local.DrawWall();
    local.PlayGame();
    if (local.score > bestScore) {
      bestScore = local.score;
    }
    cout << "best score is " << bestScore << "\n";
  }
}
```

Литература

1. Knuth, Donald E. «Sorting and Searching». Reading, Mass.: Addison-Wesley, 1973.
2. Bell, Doug, Ian Morrey, and John Pugh. «Software Engineering: A Programming Approach». Englewood Cliffs, N.J.: Prentice Hall, 1987, ff.43.
3. American Telephone and Telegraph Company, «UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools». Englewood Cliffs, N.J.: Prentice-Hall, 1990.

Приложение Ж
Список терминов

В этом приложении перечислены основные термины, используемые в книге.

Термин, используемый в книге	Оригинальный термин
Абстрактный базовый класс	Abstract base class
Абстрактный базовый прототип	Abstract base exemplar
Абстрактный тип данных	Abstract data type
Абстракция	Abstraction
Автономный обобщенный конструктор	Autonomous generic constructor
Автономный обобщенный прототип	Autonomous generic exemplar
Актер	Actor
Ассоциативная выборка	Associative retrieval
Ассоциативный массив	Associative array
Базовый класс	Base class
Библиотека	Library
Вариант	Variant
Вектор	Vector
Виртуальная функция	Virtual function
Висячая ссылка	Dangling reference
Время выполнения	Run time
Время компиляции	Compile time
Встроенный тип данных	Built-in type
Выборка	Retrieval
Высокоуровневое управление каналом	High-Level Data Link Control (HDLC)
Глобальная перегрузка	Global overloading
Глубокое копирование	Deep copy
Делегирование	Delegation
Деструктор	Destructor
Динамическая типизация	Dynamic typing
Динамическое множественное наследование	Dynamic multiple inheritance
Диспетчер памяти	Memory manager
Загрузка	Loading
Задача	Task
Закрытое наследование	Private derivation, или private inheritance

Термин, используемый в книге	Оригинальный термин
Закрытый член класса	Private member
Защищенный член класса	Protected member
Зомби	Zombie
Идентификация операций на стадии выполнения	Run-time operator identification
Иерархия реализации	Implementation hierarchy
Изоморфная структура	Isomorphic structure
Инициализация	Initialization
Интерфейс	Interface
Исключение	Exception
Итератор	Iterator
Каркас	Framework
Каркасное программирование	Frame-based programming
Класс	Class
Класс конверта	Envelop class
Класс письма	Letter class
Класс-заместитель	Stand-in class
Компоновка	Linking
Конверт	Envelop
Конкретный тип данных	Concrete data type
Конструктор	Constructor
Конструктор по умолчанию	Default constructor
Контейнер	Container
Контейнерный класс	Container class
Контекст	Context
Копирующий конструктор	Copy constructor
Корневой базовый класс	Root base class
Косвенный шаблон	Indirect template
Курсор	Cursor
Куча	Heap
Левостороннее значение	Left-hand value (l-value)
Логическое копирование	Logical copy
Манипулятор	Handle
Массив	Array
Международная организация по стандартизации	International Standards Organization (ISO)
Мета-возможности	Meta-features
Метакласс	Metaclass
Метод	Method
Метод близнецов	Buddy system
Механизм	Mechanism
Множественное наследование	Multiple inheritance
Модель, представление, контроллер	Model-View-Controller (MVC)
Модуль	Module
Мультиметод	Multi-method

Термин, используемый в книге	Оригинальный термин
Мультиобработка	Multiprocessing
Мусор	Garbage
Наложение указателей	Pointer aliasing
Наследование	Inheritance
Наследование с переопределением	Inheritance with overriding
Наследование с расширением	Inheritance with addition
Наследование с сокращением	Inheritance with cancellation
Неоднозначность	Ambiguities
Непереносимый код	Nonportable code
Неполный класс	Partial class
Область видимости	Scope
Обобщенная объектная система Lisp	Common Lisp Object System (CLOS)
Обобщенный конструктор	Generic constructor
Обобщенный прототип	Generic exemplar
Оболочка	Wrapper
Обработка исключений	Exception handling
Обратный вызов	Callback
Объединение	Union
Объект	Object
Объектная инверсия	Object inversion
Объектно-ориентированное программирование	Object-oriented programming
Объектно-ориентированное проектирование	Object-oriented design
Объявление	Declaration
Обязательство	Responsibility
Оператор	Operator
Операторная функция	Operator function
Операторная функция класса	Member operator function
Опережающая ссылка	Forward reference
Открытое наследование	Public derivation, или public inheritance
Открытый член класса	Public member
Пакетирование	Packaging
Параллельная обработка	Parallel processing
Параметризованный тип	Parameterized type
Параметрический полиморфизм	Parametric polymorphism
Перегрузка	Overloading
Перегрузка в классе	Member overloading
Переключение контекста	Context switch
Перенаправление	Forwarding
Подмена	Overriding
Переопределение	Redefining
Письмо	Letter
Планирование	Scheduling
Поверхностное копирование	Shallow copy

Термин, используемый в книге	Оригинальный термин
Подмножество	Subset
Подсистема	Subsystem
Подставляемая функция	Inline function
Подстановка	Substitution
Подсчет ссылок	Reference counting
Подсчитываемый указатель	Counted pointer
Полиморфизм	Polymorphism
Полиморфизм включения	Inclusion polymorphism
Полупространственное копирование	Semispace copying
Пользовательский тип	User-defined type
Пометка	Tag
Помеченная структура	Tagged structure
Помеченный класс	Tagged class
Предварительная пометка	Mark-and-sweep
Представитель	Ambassador
Преобразование типа	Type conversion
Примесь	Mix-in
Принцип подстановки Лисков	Liskov substitution principle
Программный поток	Thread
Производный класс	Derived class
Протокол доступа к каналу для данных	Link Access Protocol for Data (LAPD)
Прототип	Exemplar, или prototype
Распорядитель сообщества прототипов	Community manager
Селектор типа	Type selector
Семантическая сеть	Semantic network
Сильная типизация	Strong typing
Сильное связывание	Strong binding
Синглетный класс письма	Singleton letter class
Синглетный объект	Singleton object
Система контроля типов	Type system
Слабая типизация	Weak typing
Слот	Slot
Создание экземпляра	Instantiation
Сообщество прототипов	Exemplar community
Составной объект	Composite object
Состояние объекта	State of object
Спецификатор доступа	Access specifier
Ссылка	Reference
Статическая функция	Static function
Статическое множественное наследование	Static multiple inheritance
Структура	Structure
Субкласс	Subclass
Субпарадигматическое восстановление	Subparadigmatic recovery

Термин, используемый в книге	Оригинальный термин
Суперкласс	Superclass
Суперпарадигматическое восстановление	Superparadigmatic recovery
Сущность	Entity
Счетчик ссылок	Reference count
Таблица виртуальных функций	Virtual function table
Тело	Body
Тип	Type
Толстый интерфейс	Fat interface
Тонкий класс	Skinny class
Традиционная простая телефонная услуга	Plain Old Telephone Service (POTS)
Транзакционная диаграмма	Transaction diagram
Транзакция	Transaction
Уборка мусора	Garbage collection
Указатель	Pointer
Физическое копирование	Physical copy
Фрейм	Frame
Функтор	Functor
Функция класса	Member function
Цифровая сеть с комплексными услугами	Integrated Services Digital Network (ISDN)
Чисто абстрактный базовый класс	Pure abstract base class
Чисто виртуальная функция	Pure virtual function
Шаблон	Template
Шаблонная функция	Function template
Эволюция схемы	Schema evolution
Экземпляр прототипа	Exemplar instance

Алфавитный указатель

К

Л

М

Н

О

П

P

C

Т

У

Ф

Джеймс О. Коплиен

Программирование на C++. Классика CS

Перевел с английского Е. Матвеев

Главный редактор	*Е. Строганова*
Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Жданов*
Научный редактор	*Е. Матвеев*
Литературный редактор	*А. Жданов*
Иллюстрации	*А. Санжаревский*
Художник	*Л. Адуевская*
Корректоры	*А. Моносов, И. Смирнова*
Верстка	*Р. Гришанов*

Лицензия ИД № 05784 от 07.09.01.
Подписано в печать 11.11.04. Формат 70×100/16. Усл. п. л. 38,7.
Тираж 3000 экз. Заказ № 1044.
ООО «Питер Принт». 194044, Санкт-Петербург, пр. Б. Сампсониевский, дом 29а.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.
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.

Дж. Коплиен

ПРОГРАММИРОВАНИЕ НА C++

КНИГИ, КОТОРЫЕ НЕ СТАРЕЮТ!

КЛАССИКА COMPUTER SCIENCE

Эта книга поможет программистам проверять свои программы на соответствие классическому стилю и идиомам языка программирования в процессе разработки.

В книге представлены:

- преимущества и потенциальные опасности нетривиальных приемов программирования на C++;
- небольшие, но очень важные примеры абстракций C++, которые с успехом можно комбинировать;
- советы по преобразованию объектно-ориентированных проектов в конкретные реализации на C++;
- возможности многократного использования кода, доступные благодаря шаблонам;
- важнейшие аспекты разработки крупномасштабных систем, включая создание библиотек, обработку исключений и распределенную обработку.

Книга может быть полезной для системных и прикладных программистов, пишущих на C++.

Читатель должен иметь определенные базовые знания в области C++.

Джеймс Коплиен изучал электронику и компьютеры в Университете штата Висконсин, получив степени бакалавра и магистра. Он работает в *Bell Laboratories* Института AT&T, где специализируется на средствах разработки, платформах и отладке объектно-ориентированных систем.

ISBN 5-469-00189-X

9 785469 001898