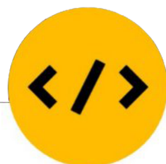
ДУМАЙ КАК ПРОГРАММИСТ

АНТОН СПРОЛ

КРЕАТИВНЫЙ ПОДХОД К СОЗДАНИЮ КОДА

C++ ВЕРСИЯ



@CODELIBRARY_IT

ANTON SPRAUN

THINK LIKE A PROGRAMMER

AN INTRODUCTION TO CREATIVE PROBLEM SOLVING

АНТОН СПРОЛ

ДУМАЙ КАК ПРОГРАММИСТ

КРЕАТИВНЫЙ ПОДХОД К СОЗДАНИЮ КОДА

БОМБОРА™
Москва 2018

УДК 004.4
ББК 32.973.26-018
С74

V. Anton Spraul
THINK LIKE A PROGRAMMER

Copyright 2012 by V. Anton Spraul. Title of English-language original: Think Like a Programmer, ISBN 978-1-59327-424-5, published by No Starch Press. Russian-language edition copyright 2018 by EKSMO Publishing House. All rights reserved.

Спрол, Антон.

С74 Думай как программист: креативный подход к созданию кода. C++ версия / Антон Спрол. — Москва : Эксмо, 2018. — 272 с. — (Мировой компьютерный бестселлер).

При помощи этой книги любой программист, особенно начинающий, может усовершенствовать свои навыки программирования. Автор разработал собственную программу, позволяющую получить навыки креативного решения разнообразных задач. Эти навыки необходимы в первую очередь тем, кто хочет создавать собственный код и действительно понимать и чувствовать основы программирования. Живой язык, множество примеров на языке C++ и уникальное авторское видение сделают чтение этой книги настоящим удовольствием.

УДК 004.4
ББК 32.973.26-018

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Антон Спрол

ДУМАЙ КАК ПРОГРАММИСТ

КРЕАТИВНЫЙ ПОДХОД К СОЗДАНИЮ КОДА

C++ ВЕРСИЯ

Директор редакции *Е. Капьёв.* Ответственный редактор *Е. Истомина*
Младший редактор *Е. Минина.* Художественный редактор *А. Гусев*

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел.: 8 (495) 411-68-86.
Home page: **www.eksmo.ru** E-mail: **info@eksmo.ru**

Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Қазақстан Республикасында дистрибьютор және өнім бойынша
арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8(727) 2 51 59 89,90,91,92, факс: 8 (727) 251 58 12 вн. 107; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ о техническом регулировании можно получить по адресу: http://eksmo.ru/certification/

Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 18.12.2017. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 22,04.
Тираж экз. Заказ

ISBN 978-5-04-089838-1

© Райтман М.А., перевод на русский язык, 2017
© Оформление. ООО «Издательство «Эксмо», 2018

ISBN 978-5-04-089838-1

Содержание

ОБ АВТОРЕ 9
ВВЕДЕНИЕ 10
Об этой книге 13
Предварительная подготовка 13
Выбор тем 13
Стиль программирования 13
Упражнения 14
Почему C++? 14

ГЛАВА 1. СТРАТЕГИИ РЕШЕНИЯ ЗАДАЧ 16
Классические головоломки 18
Лисица, гусь и кукуруза 18
Задача: как пересечь реку? 18
Вынесенные уроки 22
Головоломки со скользящими плитками 22
Задача: скользящая восьмерка 22
Задача: скользящая пятерка 24
Вынесенные уроки 26
Судоку 27
Задача: заполнить квадрат «Судоку» 27
Вынесенные уроки 28
Замóк Кварраси 29
Задача: открыть инопланетный замóк 29
Вынесенные уроки 32
Обшие подходы к решению задач 32
Планируйте решения 32
Переформулируйте задачи 34
Делите задачи 35
Начните с того, что знаете 36
Упрощайте задачи 36
Ищите аналогии 38
Экспериментируйте 39
Не расстраивайтесь 39
Упражнения 41

ГЛАВА 2. ИСТИННЫЕ ГОЛОВОЛОМКИ 42
Обзор языка C++, используемого в этой главе 43
Шаблоны вывода 43
Задача: половина квадрата 43
Задача: квадрат (упрощение задачи с половиной квадрата) 44
Задача: линия (еще большее упрощение задачи с половиной квадрата) 44
Задача: посчитать на уменьшение, считая на увеличение 45
Задача: равнобедренный треугольник 46
Обработка ввода 49
Задача: проверка контрольной суммы Луна 50
Разбиение проблемы на части 51
Задача: преобразовать символ цифры в целое число 53
Задача: проверка контрольной суммы Луна, фиксированная длина 54
Задача: проверка простой контрольной суммы, фиксированная длина 55
Задача: положительное или отрицательное 57
Соберем все детали вместе 58
Отслеживание состояния 60

Задача: декодирование сообщения 60
Задача: чтение трех- или четырехзначного числа 64
Задача: чтение трех- или четырехзначного числа, дальнейшее упрощение . . .65
Заключение 73
Упражнения. 73

ГЛАВА 3. РЕШЕНИЕ ЗАДАЧ С МАССИВАМИ 76

Обзор основных свойств массивов 77
Сохранение 78
Копирование 78
Извлечение и поиск. 79
Поиск определенного значения 79
Поиск по критерию 80
Сортировка 81
Быстрая и простая сортировка с помощью функции qsort 81
Легко модифицируемый алгоритм сортировки — сортировка вставками 82
Вычисление статистических показателей .84
Решение задач с помощью массивов 84
Задача: нахождение моды. 84
Рефакторинг 88
Массивы фиксированных данных 91
Нескалярные массивы 93
Многомерные массивы 95
В каких случаях использовать массивы 99
Упражнения 103

ГЛАВА 4. РЕШЕНИЕ ЗАДАЧ С УКАЗАТЕЛЯМИ И ДИНАМИЧЕСКОЙ ПАМЯТЬЮ. 105

Обзор основных свойств указателей 106
Преимущества использования указателей. .107
Структуры данных, размер которых определяется во время выполнения программы 107
Динамические структуры 108
Разделение памяти 108
В каких случаях использовать указатели. .109
Вопросы памяти 110
Стек и куча. 110
Объем памяти 113
Время существования переменной 115
Решение задач с указателями 115
Строка переменной длины. 116
Задача: операции со строками переменной длины 116
Проверка на специальные случаи 122
Копирование созданной динамически строки 123
Связные списки 126
Задача: отслеживание неизвестного количества студенческих карточек 126
Построение списка узлов 127
Добавление узлов в список 130
Обход списка 132
Заключение и дальнейшие шаги. 135
Упражнения 135

ГЛАВА 5. РЕШЕНИЕ ЗАДАЧ С КЛАССАМИ. 138

Обзор основных свойств классов 139
Цели использования классов. 141
Инкапсуляция 141

Повторное использование кода . . . 142
Разделение задачи . . . 142
Сокрытие . . . 143
Читабельность . . . 145
Выразительность . . . 146
Создание простого класса . . . 146
Задача: список класса . . . 146
Базовый фреймворк класса . . . 147
Служебные методы . . . 151
Классы с динамическими данными . . . 154
Задача: отслеживание неизвестного количества записей студентов . . . 155
Добавление узла . . . 157
Перегруппировка списка . . . 159
Деструктор . . . 163
Глубокое копирование . . . 164
Общий обзор классов с динамической памятью . . . 168
Ошибки, которых следует избегать . . . 169
Фальшивый класс . . . 169
Однозадачники . . . 170
Упражнения . . . 171

ГЛАВА 6. РЕШЕНИЕ ЗАДАЧ С ПОМОЩЬЮ РЕКУРСИИ . . . 173

Обзор основ рекурсии . . . 174
Головная и хвостовая рекурсия . . . 174
Задача: подсчет количества попугаев . . . 174
Подход 1 . . . 175
Подход 2 . . . 176
Задача: выявление лучшего клиента . . . 178
Подход 1 . . . 179
Подход 2 . . . 181
Большая рекурсивная идея . . . 183
Задача: вычисление суммы элементов целочисленного массива . . . 184
Распространенные ошибки . . . 186
Слишком много параметров . . . 187
Глобальные переменные . . . 188
Применение рекурсии к динамическим структурам данных . . . 189
Рекурсия и связные списки . . . 190
Задача: подсчет отрицательных чисел в односвязном списке . . . 191
Рекурсия и двоичные деревья . . . 192
Задача: нахождение наибольшего значения в двоичном дереве . . . 194
Функции-обертки . . . 195
Задача: нахождение количества листьев в двоичном дереве . . . 195
В каких случаях использовать рекурсию . . . 198
Аргументы против рекурсии . . . 198
Задача: отображение элементов связного списка в прямом порядке . . . 200
Задача: отображение элементов связного списка в обратном порядке . . . 201
Упражнения . . . 202

ГЛАВА 7. РЕШЕНИЕ ЗАДАЧ С ПОМОЩЬЮ ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ КОДА . . . 204

Хорошее и плохое повторное использование кода . . . 205
Основы работы с компонентами . . . 206
Кодовый блок . . . 206
Алгоритмы . . . 207

Шаблоны . . . 207
Абстрактные типы данных . . . 208
Библиотеки . . . 209
Изучение компонентов . . . 210
Исследовательское обучение . . . 210
Применение полученных знаний на практике . . . 211
Задача: староста . . . 211
Анализ решения задачи выбора старосты . . . 214
Обучение по мере необходимости . . . 215
Задача: эффективный обход . . . 215
Когда следует искать компонент . . . 216
Нахождение компонента . . . 217
Применение компонента . . . 218
Анализ эффективного решения задачи с обходом . . . 222
Выбор типа компонента . . . 223
Процесс выбора компонента . . . 225
Задача: выборочная сортировка . . . 225
Сравнение результатов . . . 229
Упражнения . . . 230

ГЛАВА 8. ДУМАЙТЕ КАК ПРОГРАММИСТ . . . 232

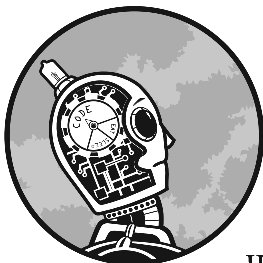
Разработка собственного мастер-плана . . . 233
Использование своих сильных и слабых сторон . . . 233
Планирование с учетом недостатков кодирования . . . 234
Планирование с учетом недостатков дизайна . . . 236
Планирование с учетом ваших сильных сторон . . . 238
Составление мастер-плана . . . 240
Решение любой задачи . . . 242
Задача: жульничество при игре в «Виселицу» . . . 243
Нахождение возможности для жульничества . . . 244
Необходимые операции для обмана в игре «Виселица» . . . 246
Исходный дизайн . . . 248
Первичное кодирование . . . 249
Анализ первоначальных результатов . . . 257
Искусство решения задач . . . 258
Изучение новых навыков программирования . . . 259
Новые языки . . . 260
Выделите время на учебу . . . 260
Начните с того, что вы знаете . . . 261
Изучите отличия . . . 261
Изучайте хорошо написанный код . . . 262
Новые навыки для языка, который вы уже знаете . . . 263
Новые библиотеки . . . 263
Выберите курс . . . 264
Заключение . . . 265
Упражнения . . . 266

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ . . . 268

Об авторе

Антон Спрол преподавал введение в программирование и информатику более 15 лет. Эта книга представляет собой квинтэссенцию методов, которые он использовал и оттачивал на протяжении множества занятий с испытывающими сложности программистами.

ВВЕДЕНИЕ



Написание программ вызывает у вас сложность, несмотря на то, что вы, вроде бы, понимаете языки программирования? Вы киваете головой, читая главу какой-нибудь книги по программированию, но не можете применить на практике то, что прочитали? А может быть, вы понимаете листинг программы, увиденный в Интернете, так хорошо, что можете объяснить кому-то, что делает каждая из строк, но все равно смотрите в пустой экран редактора, когда сталкиваетесь с задачей по программированию, и ощущаете, что ваш мозг отказывается работать?

Вы не одиноки. Я преподаю программирование уже более 15 лет, и большинство моих студентов подошли бы под это описание в определенные моменты процесса обучения. Я бы назвал это *навыком решения задач,* способностью воспринять описание поставленной задачи

и написать оригинальный программный код для ее решения. Не все аспекты программирования требуют решения большого количества задач. Если вы просто вносите незначительные изменения в код рабочей программы, выполняете отладку, добавляете код тестирования, процесс программирования может быть настолько механическим по своей сути, что вашим творческим возможностям в итоге никогда не будет брошен вызов. Но факт в том, что все программы в тот или иной момент требуют решения задач, и все хорошие программисты умеют эти задачи решать.

Делать это непросто. Конечно, есть люди, справляющиеся с различными задачами легко и непринужденно. Их называют the naturals — эквивалент одаренного атлета, наподобие Майкла Джордана в мире программирования. Для этих избранных любые высокоуровневые идеи легко трансформируются в исходный код. Если воспользоваться метафорой языка Java, то получится, что их мозг выполняет программы «нативно», а все остальные, включая нас с вами, вынуждены запускать виртуальную машину, интерпретирующую код по мере выполнения.

Однако нужно понимать, что программист может не относится к «the naturals», и это не фатально. Иначе в нашем мире было бы очень мало программистов. Тем не менее я видел слишком много хороших учеников, отчаянно бьющихся над решением задач на протяжении слишком долгого времени. В худшем случае они отказывались от программирования вовсе, убежденные в собственной неспособности стать программистами и считая, что единственные хорошие программисты — это люди с природным даром.

Почему научиться решать задачи по программированию так сложно?

Отчасти из-за того, что решение задач — это деятельность, отличная от изучения синтаксиса языка программирования, чтения кода программ и запоминания элементов интерфейса приложений; вышеперечисленные процессы в основном аналитические, относящиеся к деятельности левого полушария. Написание оригинальной программы с использованием ранее изученных инструментов и навыков — это творческий процесс, относящийся к правому полушарию головного мозга.

Предположим, что вам нужно убрать ветку, попавшую в один из дождевых желобов вашего дома, однако стремянка недостаточно высока и вы не можете дотянуться до ветки. Вы направляетесь в гараж в поисках чего-то или сочетания чего-то с чем-то, что поможет вам убрать ветку из желоба. Есть ли что-то такое, что может увеличить длину стремянки, чтобы вы смогли схватить или сбить ветку? Возможно, вы можете просто взобраться на крышу и достать ветку оттуда? Вот это и называется решением задач, и это деятельность творческая. Хотите верьте, хотите нет, но при проектировании оригинальной программы мыслительный процесс у вас в голове похож на то, что происходит

в голове человека, пытающегося придумать, как убрать ветку из желоба, и довольно сильно отличается от мыслительного процесса человека, занятого отладкой существующего цикла for.

Впрочем, внимание авторов большинства книг по программированию сосредоточено на синтаксисе и семантике. Изучение таких норм любого языка необходимо, но это лишь первый шаг на пути изучения программирования. На самом деле, большинство книг по программированию для начинающих учат, как читать код программ, но не как его писать. Книги, которые сосредоточены на написании, зачастую представляют собой «кулинарные книги», обучающие конкретным «рецептам» и их использованию в определенных ситуациях. Такие книги могут быть очень ценными для экономии времени, но не для обучения написанию оригинального кода. Подумайте о кулинарных книгах в исходном значении этих слов. Несмотря на то, что у замечательных поваров есть кулинарные книги, никто из тех, кто полагается исключительно на кулинарные книги, не может быть по-настоящему хорошим поваром. Замечательный повар понимает ингредиенты, методы готовки и знает, как их сочетать для получения вкусных блюд. Все, что нужно такому повару — это кухня с нужными ингредиентами и оборудованием. Таким же образом замечательный программист понимает синтаксис языка, среду разработки приложения, алгоритмы, принципы разработки и знает, как все это сочетать для создания отличных приложений. Дайте такому программисту технические спецификации, развяжите ему руки полнофункциональной средой программирования — и случится нечто удивительное.

Современное обучение программированию, по большей части, не предлагает много руководств по решению задач. Наоборот, предполагается, что если программисту дан доступ ко всем инструментам и от него требуют написать большое количество программ, то рано или поздно он научится писать такие программы и будет делать это хорошо. Путь от инициации до просветления может быть полон отчаяния и растерянности — и слишком многие из тех, кто его начинает, не доходят до конца.

Вместо того чтобы учиться методом проб и ошибок, вы можете научиться решать задачи, используя системный подход. Именно этому и посвящена данная книга. Вы изучите техники организации ваших мыслей, процедуры поиска решений и стратегии, которые могут быть применены к определенным классам задач. Изучив эти подходы, вы сможете высвободить свой потенциал. Он неизвестен, и никто не может с точностью сказать, как работает творческое сознание. Однако если мы можем разучить музыкальное произведение, воспринять совет по творческому написанию текстов или научиться рисовать, значит, мы можем также научиться творчески решать задачи. Это книга не скажет вам, что именно нужно делать, она поможет вам развить скрытые способности решения задач, так, что вы будете

знать, как следует поступить. Предназначение этой книги — помочь вам стать таким программистом, каким вы должны стать.

Цель, которую я ставлю перед вами и перед каждым читателем этой книги, — научиться системному подходу к выполнению каждой задачи по программированию и быть уверенным в том, что в конце концов вы сможете решать любые из них. Когда вы закончите читать эту книгу, я хочу чтобы *вы думали как программист* и *верили в то, что вы программист.*

Об этой книге

Объяснив необходимость этой книги, я должен также прокомментировать то, чем эта книга является, а чем нет.

Предварительная подготовка

Эта книга предполагает, что вы уже знакомы с базовыми синтаксисом и семантикой языка C++ и уже начали писать программы. В большинстве глав предполагается, что вам уже известны конкретные основы C++ — они будут начинаться с обзора этих основ. Если вы только постигаете язык, не переживайте: существует много замечательных книг по синтаксису C++, и вы можете параллельно изучать решение проблем и задач и синтаксис. Просто убедитесь, что вы изучили нужный синтаксис прежде, чем попытаетесь одолеть задачи в главе.

Выбор тем

Темы, обсуждаемые мной в этой книге, представляют те аспекты, в которых, по моему опыту, у начинающих программистов больше всего проблем. Они также представляют широкий срез различных аспектов программирования начального и среднего этапов.

Я должен, однако, подчеркнуть, что это не «кулинарная книга» с готовыми алгоритмами и шаблонами для решения конкретных задач. Несмотря на то, что в последних главах книги обсуждается применение широко известных алгоритмов или шаблонов, вам не следует использовать эту книгу как «шпаргалку», которая поможет вам решить конкретные задачи или сконцентрироваться лишь на главах, имеющих непосредственное отношение к вашим текущим задачам. Вместо этого я бы посоветовал вам проработать всю книгу и пропускать тот или иной раздел только в том случае, если вам не хватает знаний для усвоения материала.

Стиль программирования

Небольшое замечание о стиле программирования в данной книге: эта книга не посвящена высокопроизводительному программированию или запуску наиболее компактного и эффективного кода. Выбранный мной стиль, в первую очередь, отвечает задаче удобочитаемости. В некоторых случаях для четкой иллюстрации описанного

правила я даже делаю несколько шагов, несмотря на то, что это же действие можно было выполнить за один шаг.

Некоторые вопросы стиля программирования будут описаны в этой книге, но они относятся к большим вопросам, как, например, что должно, а что не должно быть включено в класс, а не к маленьким вопросам, таким как отступы строк программного кода. Как развивающийся программист, вы, разумеется, захотите применить наиболее удобочитаемый стиль для всех выполняемых работ.

Упражнения

В эту книгу я включил большое количество упражнений по программированию. Но это не учебник, поэтому вы не найдете ответов к упражнениям в конце книги. Упражнения лишь дают вам возможность применить на практике подходы, описанные в главе. Выполнять упражнения или нет — решать вам, однако практическое применение этих подходов очень важно. Простое чтение книги не даст вам ничего. Помните, что эта книга не скажет вам, что именно нужно сделать в той или иной ситуации. Применяя техники, показанные в этой книге, вы самостоятельно поймете, что нужно делать. Более того, повышение вашей самооценки — еще одна из целей этой книги — подразумевает достижение успеха. На самом деле, когда вы поймете, что можете решить любую задачу в той или иной области, это будет означать, что вы проработали достаточное количество упражнений. В конце концов, выполнение упражнений по программированию должно приносить вам удовольствие. Выполнение этих задач должно быть не рутиной, но вызовом вашим возможностям.

Воспринимайте эту книгу, словно полосу препятствий в вашей голове. Прохождение полосы препятствий увеличивает вашу силу, выносливость, ловкость и дает вам уверенность в собственных способностях. По мере прочтения этой книги и применения описанных в ней подходов на практике вы разовьете уверенность в себе и приобретете навыки решения задач, которые сможете использовать в любой ситуации при программировании. Встретив сложную задачу в будущем, вы будете знать подход к ее решению.

Почему C++?

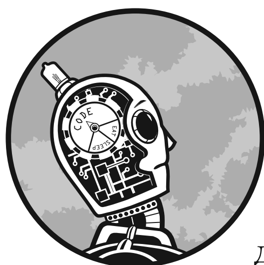
Программный код примеров в данной книге написан на языке C++. При этом следует отметить, что книга посвящена решению проблем, не обязательно характерных для этого языка. В книге вы не найдете большого количества советов и приемов, подходящих только для C++. Общие концепции, которым она обучает, могут быть реализованы с помощью любого языка программирования. Тем не менее мы не можем обсуждать программирование, не затрагивая программы, поэтому для примеров я выбрал конкретный язык для примеров.

C++ был выбран по многим причинам. Во-первых, этот язык популярен во многих проблемных областях. Во-вторых, так как C++

произошел от строго процедурного языка C, код на C++ может быть написан с использованием как процедурной, так и объектно ориентированной парадигмы. Объектно ориентированное программирование применяется сегодня настолько повсеместно, что не может быть исключено из обсуждения решения задач. Однако многие фундаментальные подходы к решению задач могут быть реализованы в строго процедурной парадигме, что упрощает как написание, так и изменение кода. В-третьих, будучи низкоуровневым языком с высокоуровневыми библиотеками, C++ позволяет мне обсудить оба уровня программирования. При необходимости лучшие программисты могут объединять низкоуровневые части с высокоуровневыми библиотеками и компонентами для сокращения времени разработки. Наконец, C++ — это отличный выбор, так как, научившись решать задачи на этом языке, вы сможете в принципе решать задачи на любом другом языке программирования. Изучение одного языка программирования облегчает последующее изучение других. Это особенно характерно для языка C++, так как он довольно сложен и основан на двух парадигмах одновременно. Возможно, это может вас испугать. Но преуспев в C++, в сможете не просто писать программы — вы по-настоящему станете программистом.

1

Стратегии решения задач



Эта книга посвящена решению задач, но что это значит на самом деле? Когда люди используют этот термин в разговоре, они зачастую имеют в виду нечто очень отличное от того, о чем я говорю в этой книге. Если из выхлопной трубы вашего старого автомобиля идет сизый дым, двигатель то и дело глохнет, а расход топлива вырос, то эта проблема, как задача, может быть решена с применением знаний автомобиля, диагностикой и заменой деталей при помощи инструментов автомеханика. Впрочем, если вы расскажете о своей проблеме друзьям, то один из них может вам ответить: «Эй, ты должен заменить свою старую машину на что-то поновее, и проблема будет решена». Однако предложение вашего друга на самом деле не будет *решением задачи*: это будет лишь способом ее *избежать*.

Задачи подразумевают ограничения, незыблемые правила, касающиеся задачи или способа ее решения. В случае со сломанной машиной одно из ограничений заключается в том, что вам нужно отремонтировать имеющийся автомобиль, а не купить новый. Ограничения также должны подразумевать общую стоимость ремонта, время на ремонт и невозможность приобретения инструмента для ремонта только этой поломки.

При решении проблемы в программе у вас также есть определенные ограничения. Среди общих ограничений язык программирования, платформа (будет ли программа работать на ПК, iPhone или ином устройстве?), производительность (для игровой программы может требоваться обновление графики до 30 кадров в секунду, бизнес-приложение должно иметь ограничение по максимальному времени отклика на ввод пользователя) или объем требуемой памяти. Иногда ограничения также подразумевают код, на который вы можете ссылаться: возможно, программа не может включать определенный открытый код или, наоборот, должна использовать только открытый код.

Таким образом, термин *решение задач* для программистов я могу определить как написание оригинальной программы, выполняющей определенный набор заданий и соответствующей всем установленным ограничениям.

Начинающие программисты зачастую так хотят выполнить первую часть определения — написать программу, выполняющую определенное задание — что забывают про выполнение второй части определения — соответствовать установленным ограничениям. Я называю такую программу, выдающую правильный результат, но нарушающую одно или несколько заявленных правил, *«Кобаяси Мару»*. Если это название вам незнакомо, значит вы не смотрели культовый в узких кругах кинофильм *«Звездный Путь 2: Ярость Хана»*. В этом фильме есть сюжетная линия про тест для начинающих офицеров Академии Звездного флота. Кадетов помещают на мостик симулятора звездного корабля или поручают роль капитана, выполняющего миссию, в ходе которой они встречаются с невозможным выбором. Гибель грозит людям на подбитом корабле *«Кобаяси Мару»*, однако, чтобы добраться до них, вам нужно пойти на самоубийственное сражение с врагами. Цель этого упражнения — проверить смелость и отвагу кадета, когда он находится под огнем. В данном сражении победить невозможно, и любой выбор ведет к провалу. Ближе к концу фильма мы обнаруживаем, что капитан Кирк модифицировал симулятор, сделав победу возможной. Кирк был хитер, но он не решил дилемму *«Кобаяси Мару»*, а лишь избежал ее.

К счастью, задачи, с которыми вы столкнетесь как программист, разрешимы, но многие программисты все еще прибегают к подходу Кирка. В некоторых случаях они делают это случайно. («О, черт! Мое решение работает только в том случае, если элементов данных

сто или меньше. Оно должен работать для неограниченного набора данных. Мне придется переосмыслить это решение».) В других случаях ограничения изменяют преднамеренно, прибегая к этому как к уловке, чтобы успеть к сроку, установленному начальником или преподавателем. В других случаях программист просто не знает, как соответствовать всем ограничениям. В худших случаях, которые я видел, студенты-программисты платили третьим лицам за написание программы. Независимо от мотивов, мы всегда должны проявлять максимум усердия во избежание «Кобаяси Мару».

Классические головоломки

По мере чтения книги вы заметите, что, несмотря на изменение частностей исходного кода от одной проблемной области к другой, некоторые схемы в моих подходах останутся неизменными. И это хорошо, потому что в конце концов позволит нам уверенно подходить к решению любой задачи, вне зависимости от того, есть ли у вас большой опыт в той или иной проблемной области. Эксперты в решении задач могут быстро распознать *аналогию*, т. е. пригодное для использования сходство между решенной и нерешенной задачей. Если мы распознаем, что некое свойство задачи А аналогично некоему свойству задачи Б, при условии, что мы уже решили задачу Б, у нас будет очень ценная наработка для решения задачи А.

В этом разделе мы обсудим классические задачи не из мира программирования, из которых мы можем вынести уроки, применимые при решении задач по программированию.

Лисица, гусь и кукуруза

Первая классическая задача, которую мы обсудим, — это загадка про крестьянина, которому нужно пересечь реку. Возможно, вы уже встречались с этой задачей в той или иной форме.

Задача: как пересечь реку?

Крестьянин с лисицей, гусем и мешком кукурузы должен пересечь реку. У крестьянина есть лодка, однако в ней есть место только для самого крестьянина и одного из перечисленных объектов. К сожалению, и лисица, и гусь голодны. Лисица не может быть оставлена наедине с гусем, так как она съест гуся. Точно так же гусь не может остаться наедине с мешком кукурузы, иначе он склюет кукурузу. Как крестьянину перевезти животных и кукурузу через реку?

Условие задачи проиллюстрировано на рис. 1.1. Если вы никогда ранее не встречались с этой задачей, сделайте небольшую паузу и потратьте несколько минут на попытки решить ее. Если же вы слышали эту загадку ранее, попытайтесь вспомнить решение и нашли ли вы его самостоятельно.

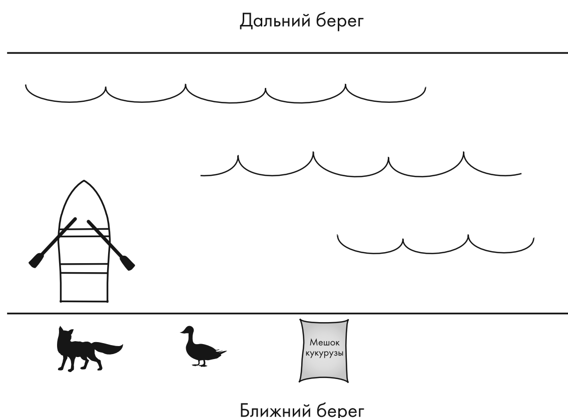


Рис. 1.1. Лисица, гусь и мешок кукурузы. Лодка может перевезти только один объект за раз. Лисицу нельзя оставить на одном берегу с гусем, точно так же, как и гуся нельзя оставлять на одном берегу с мешком кукурузы

Очень немногие могут решить эту загадку, по крайней мере без подсказки. Я тоже не смог. Вот как обычно идет рассуждение: поскольку крестьянин может брать в лодку только один из объектов за раз, ему потребуется несколько поездок, чтобы перевезти все на дальний берег. Если крестьянин возьмет в первую поездке лисицу, то гусь останется с мешком с кукурузы — и склюет кукурузу. Точно так же, если бы крестьянин взял мешок с кукурузой в первую поездку, а лисица осталась бы в компании с гусем, то лисица съела бы гуся. Таким образом, крестьянин должен взять гуся в первую поездку, в результате чего получится конфигурация, показанная на рис. 1.2.

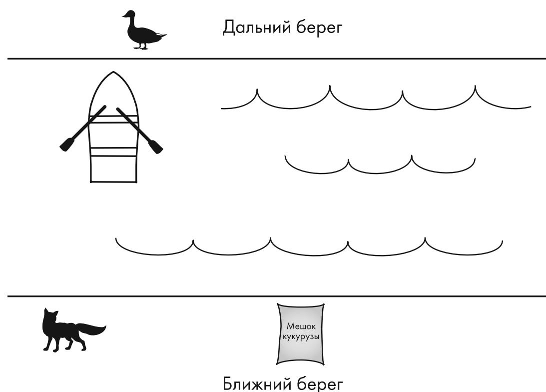


Рис. 1.2. Необходимый первый шаг решения задачи с лисицей, гусем и мешком кукурузы. Однако складывается ощущение, что все последующие шаги обречены на провал

Пока что все идет нормально. Но во вторую поездку крестьянин должен взять лисицу или кукурузу. Однако, что бы крестьянин ни взял, этот объект должен быть оставлен на дальнем берегу с гусем, на то время, пока крестьянин возвращается к ближнему берегу за оставшимся объектом. Это означает, что лиса и гусь останутся наедине

или что гусь и кукуруза будут оставлены вместе. Поскольку ни одна из этих ситуаций не приемлема, проблема кажется неразрешимой.

Повторимся, если вы уже встречали эту задачу, то, возможно, вы помните ключевой элемент решения. Как было объяснено выше, крестьянин должен взять в первую поездку гуся. Давайте предложим, что во вторую поездку крестьянин берет с собой лисицу. Однако вместо того, чтобы оставить гуся с лисицей, крестьянин *отвозит гуся обратно* на ближний берег. Затем крестьянин перевозит через реку мешок кукурузы, оставляя на дальнем берегу лисицу с кукурузой, и возвращается туда в четвертую поездку с гусем. Полное решение задачи показано на рис. 1.3.

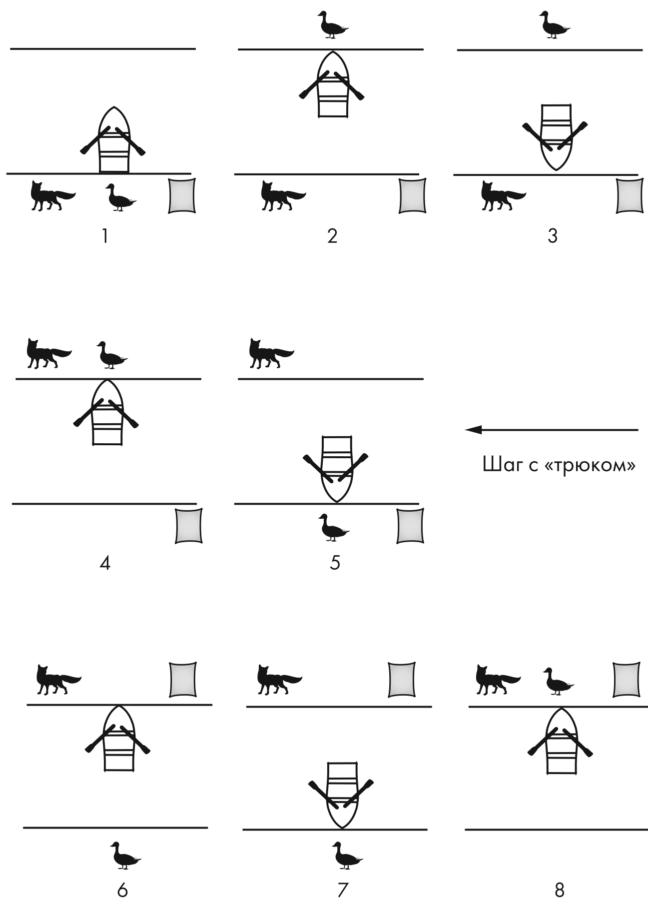


Рис. 1.3. Пошаговое решение головоломки с лисицей, гусем и кукурузой

Эта задача сложна потому, что большинство людей не рассматривает возможность перевезти один из элементов обратно с дальнего берега на ближний. Некоторые могут посчитать загадку нечестной и сказать что-то наподобие: «Вы не говорили, что я могу перевезти что-то обратно!» Это правда, впрочем, как правда и то, что ничего в

описании задачи не указывает на то, что перевозить объекты обратно запрещено.

Представьте, насколько проще было бы решить эту головоломку, если бы возможность перевезти один из объектов обратно на ближний берег была бы указана явно: *У крестьянина есть лодка,* ***которую он может использовать для перевозки объектов в обоих направлениях,*** *но в лодке есть место только для самого крестьянина и одного из трех объектов.* Теперь, когда эта подсказка видна невооруженным глазом, большее количество людей смогут решить задачу. Это позволяет проиллюстрировать важный принцип решения задач: если вам не известны все действия, которые вы можете предпринять, то, возможно, вы не сможете решить задачу. Мы можем называть эти действия операциями. Создав список всех возможных операций, мы сможем решить любую задачу, проверяя все возможные сочетания операций до тех пор, пока одно из этих сочетаний не решит задачу. В общем, переформулирование задачи более формальным языком позволяет нам открыть решения, которые ускользнули бы от нас в противном случае.

Давайте забудем, что мы уже знаем решение, и попытаемся более формально поставить условие данной конкретной задачи. В первую очередь, перечислим основные ограничения:

1. Крестьянин может взять в лодку только один объект за раз.
2. Лисицу нельзя оставить вместе с гусем.
3. Гуся нельзя оставить вместе с кукурузой.

Эта задача — хороший пример важности ограничений. Если мы уберем одно из вышеприведенных ограничений, то головоломка становится простой. Убрав первое ограничение, мы сможем запросто перевезти все три объекта через реку за одну поездку. Даже если мы можем взять только два объекта, то мы можем взять лисицу и кукурузу и перевезти их через реку, а затем вернуться за гусем. Убрав второе ограничение (но оставив в силе все прочие), то, нам нужно лишь быть внимательными и сначала взять гуся, затем переправить лисицу и, наконец, вернуться за кукурузой. Таким образом, если мы забудем или проигнорируем ограничения, то получим «Кобаяси Мару».

Теперь давайте перечислим все операции. Существует несколько способов формулировки операций этой головоломки. Мы могли бы составить список конкретных действий, которые, как нам кажется, мы можем предпринять.

1. Операция: Перевезти лисицу на дальний берег реки.
2. Операция: Перевезти гуся на дальний берег реки.
3. Операция: Перевезти кукурузу на дальний берег реки.

Помните, что цель формальной переустановки задачи — получение некоего аналитического вывода для решения. Если мы раньше не решали эту задачу и не находили возможную «скрытую» операцию, такую как перевозка гуся обратно на ближний берег, мы не сможем найти эту опе-

рацию, просто создав список действий. Вместо этого нам следует попытаться сделать операции шаблонными, или параметризованными.

1. Операция: Переплыть на лодке с одного берега на другой.
2. Операция: Если лодка пуста, то загрузить в нее один объект с берега.
3. Операция: Если лодка не пуста, то разгрузить с нее один объект на берег.

Обдумав задачу в наиболее общих терминах, второй список действий позволит нам решить проблему без необходимости догадываться, что можно перевезти гуся обратно на ближний берег. Если мы сгенерируем все возможные последовательности действий, при этом прекращая выполнение каждой последовательности, когда она нарушает одно из ограничений или приводит к конфигурации, которую мы видели ранее, то в конце концов мы составим последовательность, изображенную на рис. 1.3, и решим задачу. Мы сможем обойти имманентную сложность этой головоломки, переформулируя ограничения и операции.

Вынесенные уроки

Какой урок можно вынести из примера с лисицей, гусем и кукурузой?

Переустановка задачи более формальным языком — это отличная техника получения выводов, позволяющих решить эту задачу. Многие программисты обращаются к коллегам для обсуждения той или иной задачи не только потому, что коллеги могут иметь решение, но и потому, что озвучивание задачи зачастую наводит на новые полезные мысли. Переустановка задачи аналогична обсуждению этой задачи с другим программистом, с той лишь разницей, что вы играете обе роли.

Более широкий урок в том, что обдумывание задачи может быть столь же, а в некоторых случаях более продуктивным, чем обдумывание решения задачи. Во многих случаях правильный подход к решению — это и *есть* решение.

Головоломки со скользящими плитками

Размеры головоломок со скользящими плитками могут быть разными, что, как вы увидите позднее, влияет на конкретный механизм решения. Нижеследующее описание подходит для версии головоломки размером 3*3.

Задача: скользящая восьмерка

В сетке 3×3 расположено восемь плиток, пронумерованных от 1 до 8, и одна пустая ячейка. Изначально плитки в сетке находятся в случайной последовательности. Плитка может быть передвинута на примыкающую пустую клетку, освободив тем самым клетку под собой. Цель составить плитки по порядку, причем плитка с цифрой 1 должна находиться в верхнем левом углу.

Цель этой задачи проиллюстрирована на рис. 1.4. Если вы никогда не разгадывали подобную головоломку, то сейчас самое время это сделать. В Интернете можно найти большое количество симуляторов, но для наших целей будет лучше, если вы воспользуетесь игральными картами или же пронумеруете карточки, чтобы сделать собственный игральный набор. На рис. 1.5 показан пример начальной конфигурации.

1	2	3
4	5	6
7	8	

Рис. 1.4. Целевая конфигурация восьмиплиточной версии головоломки со скользящими плитками. Квадрат без цифры — это пустая ячейка, в которую может быть перенесена примыкающая плитка

4	7	2
8	6	1
3	5	

Рис. 1.5. Пример начальной конфигурации для головоломки со скользящими плитками

Эта головоломка довольно сильно отличается от головоломки с лисицей, гусем и кукурузой. Сложность прошлой задачи была в оставленной без внимания одной возможной операции. В данном же случае этого не происходит. Для любой заданной конфигурации максимум четыре плитки могут примыкать к пустой ячейке — и любая из этих плиток может быть передвинута в пустую ячейку. Это полностью перечисляет все возможные операции.

Источник сложности этой задачи — необходимость создания длинной цепочки операций. Последовательность операций может передвинуть часть плиток в правильную окончательную позицию, сдвигая при этом другие плитки с их правильных позиций, или же последовательность может передвинуть часть плиток ближе к верным позициям, сдвинув при этом часть плиток дальше от верных позиций. Из-за этого сложно сказать, будет ли результатом выполнения той или иной операции продвижение к цели. Не имея возможности оценить прогресс, сложно сформулировать стратегию. Многие из тех, кто пытается решить головоломку с плитками, просто в случайном порядке передвигают плитки

в надежде выйти на конфигурацию, из которой будет виден путь к цели.

Тем не менее существуют стратегии решения головоломок с плитками. Для иллюстрации одного из подходов давайте рассмотрим головоломку с меньшей и при этом прямоугольной, а не квадратной сеткой.

Задача: скользящая пятерка

В сетке 2×3 расположено пять плиток, пронумерованных от 4 до 8, и одна пустая клетка. Изначально плитки в сетке находятся в случайной последовательности. Плитка может быть передвинута в примыкающую пустую ячейку, освободив тем самым клетку под собой. Цель составить плитки по порядку, причем плитка с цифрой 4 должна находиться в верхнем левом углу.

Вы могли обратить внимание, что наши пять плиток пронумерованы от 4 до 8, а не от 1 до 5. Причины этого будут понятны чуть позже.

Несмотря на то, что это такая же базовая задача, как и скользящая восьмерка, из-за того, что плиток всего пять, ее проще решить. Попробуйте конфигурацию, изображенную на рис. 1.6.

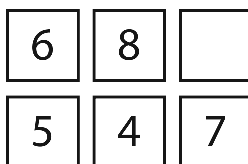


Рис. 1.6. Пример начальной конфигурации для уменьшенной головоломки со скользящими плитками 2×3

Если вы поиграете с плитками на протяжении нескольких минут, то вы, возможно, найдете решение. Я довольно быстро научился, играя с малым количеством плиток. Именно этот навык, вместе с наблюдением, которое мы вскоре обсудим, я использую для решения всех головоломок с плитками.

Я называю мою технику *кортеж*. Она основана на наблюдении, что схема позиций плиток с пустой ячейкой составляет кортеж — вереницу плиток, — которая может перемещаться по схеме, сохраняя при этом относительную упорядоченность плиток. На рис. 1.7 показан наименьший возможный кортеж с четырьмя позициями. Из начальной конфигурации плитка 1 может передвинуться в пустую ячейку, плитка 2 может передвинуться на место, освобожденное плиткой 1, и, наконец, плитка 3 может передвинуться на место, освобожденное плиткой 2. Такая последовательность оставляет пустую ячейку примыкать к плитке 1, что позволяет кортежу продолжать движение, и таким образом плитки могут быть двигаться по пути перемещения кортежа.

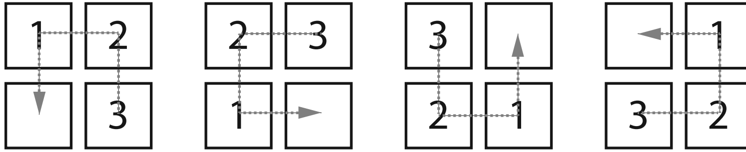


Рис. 1.7. «Кортеж» — путь плиток, который начинается от плитки, примыкающей к пустому квадрату, и скользящий по полю головоломки как кортеж из автомобилей.

Используя кортеж, мы можем перемещать наборы плиток, сохраняя при этом их взаимное расположение. Теперь давайте вернемся к предыдущей конфигурации с сеткой 2×3. Несмотря на то, что ни одна из плиток не находится в правильной финальной позиции, некоторые из плиток примыкают к плиткам, рядом с которыми они должны находиться в финальной конфигурации. Например, в финальной конфигурации плитка 4 будет над плиткой 7, и в данный момент эти плитки смежные. Как показано на рис. 1.8, мы можем использовать шестипозиционный кортеж для того, чтобы передвинуть плитки 4 и 7 на правильные позиции. Когда мы сделаем это, оставшиеся плитки займут почти что правильные позиции, нам останется лишь придвинуть плитку 8.

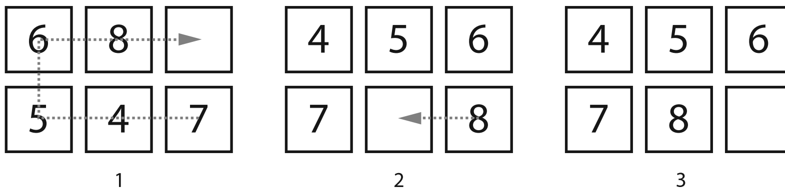


Рис. 1.8. Из конфигурации 1 две ротации вдоль выделенного «пути» приведут нас к конфигурации 2. Оттуда движение одной плитки приведет нас к цели, конфигурации 3.

Итак, каким же образом эта единственная техника позволяет нам решить любую головоломку с плитками? Рассмотрим нашу исходную конфигурацию 3×3. Мы можем использовать шестипозиционный кортеж для переноса смежных плиток 1 и 2 так, что плитки 2 и 3 станут смежными, как показано на рис. 1.9.

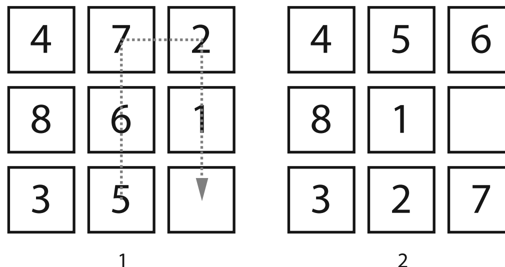


Рис. 1.9. Из конфигурации 1 плитки перемещаются вдоль выделенного «пути» и приходят к конфигурации 2.

Это действие сделало плитки 1, 2 и 3 смежными. Имея восьмипозиционный кортеж, мы можем переместить плитки 1, 2 и 3 так,

чтобы они оказались в правильных финальных позициях, как показано на рис. 1.10.

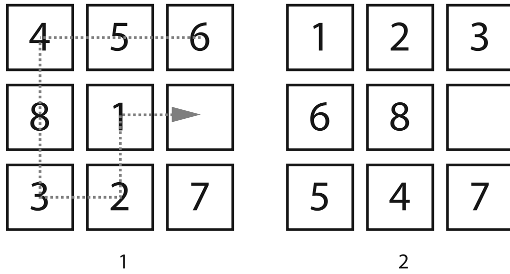


Рис. 1.10. Из конфигурации 1 плитки перемещаются вдоль выделенного «пути» и приходят к конфигурации 2, в которой оказываются в правильных финальных позициях

Обратите внимание на положение плиток 4–8. Эти плитки находятся в конфигурации, которую мы ранее задали для сетки 2*3, — и это ключевой момент. Поместив плитки 1–3 в правильные позиции, мы теперь можем решить оставшуюся сетку, как отдельную, меньшую и менее сложную задачу. Обратите внимание, чтобы данный метод сработал, нам нужно решить полностью либо столбец, либо строку; если бы мы передвинули плитки 1 и 2 в правильные позиции, а плитка 3 все еще находилась не на своем месте, то мы бы не смогли ничего передвинуть в правый верхний угол, не смещая одну или обе плитки верхнего ряда.

Эта же техника может использоваться для решения головоломок со скользящими плитками большего размера. Самая популярная головоломка насчитывает 15 плиток в сетке 4*4. Она может быть решена поэтапно. Для начала нужно перенести плитки 1–4 в правильные позиции, оставив сетку 3*4. Далее мы составим плитки самой левой колонки — и получим сетку 3*3. К этому моменту задача уменьшится и сведется к загадке с 8 плитками.

Вынесенные уроки

Какие уроки мы можем вынести из головоломок со скользящими плитками?

Количество движений плиток достаточно велико, поэтому сложно или вовсе невозможно заранее спланировать полноценное решение головоломки из начальной конфигурации. Однако невозможность спланировать полноценное решение не мешает нам составлять стратегии или использовать различные техники для систематичного решения головоломки. При решении задач программирования мы иногда сталкиваемся с такими ситуациями, когда невозможно увидеть четкий путь написания программного кода, необходимого для решения. Однако мы не должны использовать это как оправдание отказа от планирования и системного подхода. Лучше разработать стратегию, чем атаковать задачу методом проб и ошибок.

Я разработал свою технику с «кортежем», играя с маленькой головоломкой. Зачастую я использую эту же технику и в программировании. Столкнувшись со сложной задачей, я экспериментирую с уменьшенной версией этой задачи. Эти эксперименты часто приводят к ценным ноу-хау.

Еще один урок в том, что иногда задачи можно разделить невидимыми изначально способами. Так как перемещение плитки влияет не только на эту плитку, но также на возможные следующие ходы, кому-то может показаться, что такая головоломка должна решаться за один шаг, а не постепенно. Поиск способа решить задачу, как правило, занимает значительное количество времени. Даже если вы не можете найти четкого разделения, вы можете узнать нечто такое, что поможет вам решить эту задачу. При решении задачи всегда лучше работать с определенной целью в уме, нежели предпринимать случайные попытки, вне зависимости от того, удается вам достичь этих конкретных целей или нет.

Судоку

Игра «Судоку» стала необычайно популярной благодаря публикациям в газетах и журналах, а также благодаря веб- и мобильным версиям. У этой игры есть несколько разновидностей, однако мы вкратце обсудим традиционную версию.

Задача: заполнить квадрат «Судоку»

Сетка 9×9 частично заполнена цифрами (от 1 до 9), задача игрока — заполнить оставшиеся клетки, укладываясь при этом в определенные ограничения: в каждой строке или колонке каждая цифра может встречаться только один раз, более того, в каждой обозначенной области 3×3 каждая цифра может встречаться только один раз.

Если ранее вы уже играли в эту игру, то у вас уже, возможно, есть набор стратегий для заполнения каждого квадрата за минимальное время. Давайте сосредоточимся на ключевой начальной стратегии, взглянув на пример квадрата, показанный на рис. 1.11.

	9	1		6		7		
				8	2		3	9
5		3				2		
			9	1	3		6	2
		2	4		6	8		
1	4		8	2	5			
		9				5		7
6	7		1	5				
		5		4		6	9	

Рис. 1.11. Легкий вариант головоломки «Судоку»

Головоломки «Судоку» могут иметь разный уровень сложности, который определяется количеством пустых клеток. По этой классификации, приведенный вариант — очень легкая головоломка. Так как 36 клеток уже пронумеровано, нам осталось заполнить лишь 45. Вопрос, какие клетки нам следует заполнить первыми?

Помните об ограничениях этой головоломки. Каждая из девяти цифр должна появиться один раз в каждой строке, в каждой колонке и в каждой области 3*3, обозначенной жирной границей. Эти правила указывают нам, где мы должны начинать наши попытки. Область в центре головоломки уже содержит восемь из девяти цифр, таким образом, существует только одно возможное значение пустой клетки в центре, значение, не представленное в других клетках области 3*3. Именно здесь мы должны начать решение этой головоломки. Пропущенное число в этой области — 7, а, значит, мы можем вписать его в центральную клетку.

Вписав это значение, обратите внимание, что в центральной колонке теперь есть семь из девяти значений клеток, таким образом, нам осталось лишь заполнить две клетки, при этом в этих клетках должны быть значения, еще не представленные в этой колонке. Два недостающих числа — это 3 и 9. Ограничение для этой колонки позволит нам вписать любое число в любую клетку, однако обратите внимание, что в третьей строке уже есть число 3, а в седьмой строке есть число 9. Таким образом, ограничения строк говорят нам, что число 9 нужно вписать в среднюю колонку третьей строки, а число 3 — в среднюю колонку седьмой строки. Эти шаги резюмированы на рис. 1.12.

	9	1		6		7		
				8	2		3	9
5		3				2		
			9	1	3		6	2
		2	4		6	8		
1	4		8	2	5			
		9				5		7
6	7		1	5				
		5		4		6	9	

1

	9	1		6		7		
				8	2		3	9
5		3				2		
			9	1	3		6	2
		2	4	7	6	8		
1	4		8	2	5			
		9				5		7
6	7		1	5				
		5		4		6	9	

2

	9	1		6		7		
				8	2		3	9
5		3		9		2		
			9	1	3		6	2
		2	4		6	8		
1	4		8	2	5			
		9		3		5		7
6	7		1	5				
		5		4		6	9	

3

Рис. 1.12. Первые шаги решения примера головоломки «Судоку»

Я не буду решать всю головоломку, но эти первые шаги уже показывают важный момент: мы ищем клетки с наименьшим количеством вариантов возможных значений, в идеале — только с одним.

Вынесенные уроки

Основной урок из головоломки «Судоку» заключается в том, что следует искать наиболее ограниченную часть задачи. Несмотря на то, что ограничения — это то, что делает задачу сложной (вспомните про лисицу, гуся и кукурузу), они также могут упростить процесс обдумывания решения благодаря уменьшению количество вариантов.

Хотя в этой книге мы не будем обсуждать искусственный интеллект, в нем существует правило решения подобных задач, называется оно «наиболее ограниченная переменная». Это значит, что при решении задачи, где вы пытаетесь присвоить различные значения различным переменным, для соответствия ограничениям, вам следует начать с переменной, на которую наложено наибольшее количество ограничений или иными словами — переменной с наименьшим вариантом возможных значений.

Вот пример такого образа мышления. Предположим, что несколько коллег хотят вместе сходить на обед и попросили вас найти ресторан, который придется по вкусу каждому. Проблема в том, что каждый из коллег накладывает определенные ограничения на групповое решение: Пэм вегетарианка, Тодд не любит китайскую кухню и так далее. Если ваша цель — минимизировать время, которое потребуется для поиска ресторана, то вам нужно начать с коллеги с наиболее жесткими ограничениями. Если у Боба аллергия на большое количество продуктов, то логичнее всего было бы начать с выяснения списка ресторанов, в которых он знает, что сможет поесть, а не с Тодда, нелюбовь к китайской еде которого может запросто быть подавлена.

Такая же техника может использоваться и при решении задач по программированию. Если на некую часть задачи возложено большое количество ограничений, значит, это отличный отправной пункт, потому что вы можете продвигаться вперед без опасения, что тратите время на работу, которая впоследствии будет отменена. Связанный с этим вывод — вы должны начинать с наиболее очевидной части. Если вы можете решить часть задачи — смело приступайте и делайте, что сможете. Изучив собственный код, вы можете увидеть нечто такое, что подстегнет ваше воображение, и вы сможете решить оставшуюся часть задачи.

За́мок Кварраси

Возможно, вы уже встречали предыдущие головоломки ранее, но вы не должны знать о последней головоломке этой главы, если не читали данную книгу, так как я придумал эту задачу сам. Будьте внимательны, так как описание задачи может показаться несколько сложным.

Задача: открыть инопланетный за́мок

Враждебная инопланетная раса Кварраси совершила высадку на Земле, и вы были захвачены. Вы смогли победить своих охранников, несмотря на то, что они огромные и с большим количеством конечностей, однако, чтобы сбежать из (все еще находящегося на Земле) космического корабля, вы должны открыть массивную дверь. Как ни странно, инструкции по открытию двери написаны на английском, но это лишь незначительно упрощает задачу. Чтобы открыть дверь вы должны передвинуть три бруска Кратцца вдоль полозьев от правого рецептора к левому, находящемуся на краю двери, примерно в 3 метрах от вас.

Это довольно просто, но вы должны сделать все так, чтобы не включилась сигнализация, которая работает следующим образом. На каждом Кратцце есть несколько кристаллических силовых гемм Квиникрис, каждая из которых имеет форму звезды. Каждый рецептор оснащен четырьмя датчиками, которые загораются в случае, если количество Квиникрис в колонне над рецептором четное. Сирена включается, если количество загоревшихся датчиков равно одному. Обратите внимание, что оба рецептора оснащены отдельной сигнализацией, то есть у вас не может возникнуть ситуации, когда только один датчик горит для левого или для правого рецептора. Хорошая новость в том, что каждая сигнализация оснащена супрессором, не позволяющим сигнализации сработать, если нажата кнопка. Если бы вы могли нажать оба супрессора одновременно, то задача была бы легкой, но вы не можете, так как у вас лишь короткие человеческие руки, а не длинные щупальца, как у Кварраси.

Принимая во внимание вышеописанное, как вы передвинете Кратцц для открытия двери, не активируя при этом одну из сигнализаций?

Начальная конфигурация приведена на рис. 1.13. В этой конфигурации все три Кратцца находятся в правом рецепторе. Для пояснения на рис. 1.14 показан пример неудачного решения: перенос самого верхнего Кратцца в левый рецептор приводит к срабатыванию правого рецептора. Вам может показаться, что мы могли бы избежать срабатывания сигнализации, воспользовавшись супрессором, но помните, что мы только что переместили правый Кратцц в левый рецептор и, следовательно, находимся в трех метрах от супрессора правого рецептора.

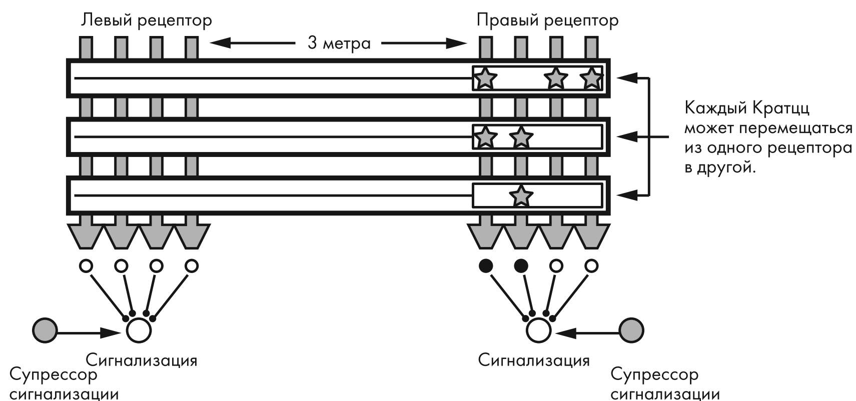


Рис. 1.13. Начальная конфигурация для задачи замок Кварраси. Вы должны перенести три бруска Кратцца, находящихся в данный момент в правом рецепторе, в левый, не активировав при этом сигнализацию. Датчик срабатывает, когда четное число звездообразных Квиникрис находится в колонне над этим датчиком, при этом сигнализация срабатывает, когда загорается один из подключенных датчиков. Супрессоры могут предотвратить срабатывание сигнализации, но только для того рецептора, рядом с которым вы находитесь

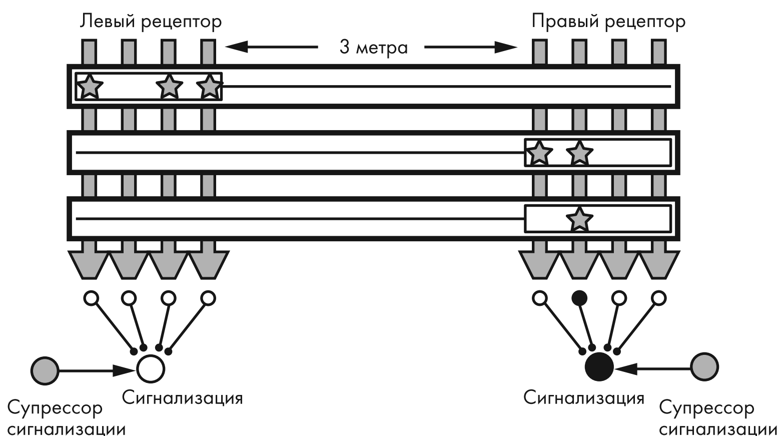


Рис. 1.14. Замок Кварраси в режиме сработавшей сигнализации. Вы только что передвинули Кратцц в левый рецептор, то есть правый рецептор для вас вне досягаемости. Второй датчик правой сигнализации загорелся, так как в колонне над ним находится четное количество Квиникрис, сигнализация срабатывает, когда загорается один из датчиков

Прежде чем продолжать чтение, возьмите небольшую паузу, чтобы изучить задачу и попытаться найти решение. В зависимости от вашей собственной точки зрения, эта задача может оказаться не такой сложной, какой кажется на первый взгляд. Серьезно, подумайте над ее решением прежде, чем продолжать чтение!

Вы подумали над задачей? Смогли ли вы найти решение? Существует два пути ответа на этот вопрос. Первый путь — это путь проб и ошибок: методически пробовать передвигать различные Кратццы и возвращаться на предыдущие шаги при достижении состояния срабатывания сигнализации до тех пор, пока не найдете работающую последовательность движений.

Второй путь — понять, что данная головоломка — лишь уловка. Если вы не поняли, в чем уловка, то объясню: перед нами лишь задача с лисицей, гусем и кукурузой в изысканной маскировке. Несмотря на то, что правила для срабатывания сигнализации описаны лишь в общем, существует только определенное количество сочетаний для каждого замка. Имея лишь три Кратцца, все, что нужно знать, — это то, какие комбинации Кратццев в рецепторе считаются допустимыми. Если мы обозначим три Кратцца как *верхний*, *средний* и *нижний*, то получим следующие комбинации, вызывающие срабатывание сигнализации, это «верхний и средний» и «средний и нижний». Если мы переименуем *верхний* в *лисицу*, *средний* в *гуся*, а *нижний* — в *кукурузу*, то комбинации, вызывающие трудности в этой задаче аналогичны комбинациям во второй: «лисица и гусь» и «гусь и кукуруза».

Таким образом, эта задача решается так же, как и задача с лисицей, гусем и кукурузой. Нам нужно передвинуть средний Кратцц (гусь) в левый рецептор, после этого передвинуть влево верхний Кратцц (лисица), удерживая при этом нажатой кнопку супрессора.

Далее мы начнем передвижение среднего Кратцца (гусь) обратно в правый рецептор. Затем мы перенесем нижний Кратцц (кукуруза) влево и, наконец, снова перенесем средний Кратцц (гусь) влево, открыв замок.

Вынесенные уроки

Главный урок здесь — это важность умения распознать аналогии. В данном примере продемонстрировано, что проблема замка Кварраси аналогична задаче с лисицей, гусем и кукурузой. Если мы сможем как можно раньше распознать аналогию, то сможем и избежать значительной части работы, просто взяв решение первой задачи, вместо создания нового. Большинство случаев сходства в решении задач не будут столь же прямыми, но по мере применения вы будете замечать это чаще и чаще.

Если вам было сложно увидеть связь между этой задачей и задачей с лисицей, гусем и кукурузой, это лишь потому, что я специально включил настолько большое количество пространных деталей, насколько это было возможно. История, включенная в постановку задачи Кварраси, не имеет отношения к ее решению, точно также все наименования инопланетных технологий нужны для увеличения чувства нахождения в незнакомой обстановке. Более того, механизм с четным/нечетным количеством брусков делает задачу на первый взгляд более сложной, чем она есть. Если вы посмотрите на положение Квиникрис, то вы увидите, что верхний Кратцц противоположен нижнему, поэтому они не могут взаимодействовать в сигнальной системе. Однако средний Кратцц взаимодействует с оставшимися двумя Кратццами.

Повторимся, если вы не разглядели аналогию, не переживайте. Вы начнете распознавать их эффективнее, если будете знать об их существовании.

Общие подходы к решению задач

Приведенные выше примеры демонстрируют множество важных подходов, применяемых при решении задач. В оставшейся части книги вы будете изучать конкретные проблемы программирования и искать способы их решения, но для начала нужно составить общий список подходов и методов. Как вы убедитесь позднее, в некоторых областях существуют специфические подходы, но правила ниже могут применяться практически в любой ситуации. Если вы сделаете их неотъемлемой частью вашего подхода к решению задач, ни одна проблема не поставит вас в тупик.

Планируйте решения

Это, вероятно, самое важное правило. У вас всегда должен быть план. Не стоит заниматься бессмысленной деятельностью.

К этому моменту вы, должно быть, поняли, что всегда можно составить план. Правда в том, что если вы не решили задачу мысленно, то у вас не может быть плана по реализации решения в виде программного кода. Это придет позже. Однако даже в самом начале у вас должен быть план, как вы будете искать решение.

Честно говоря, по ходу решения той или иной задачи план может потребоваться изменить или вообще отказаться от него в пользу нового плана. Тогда почему же это правило так важно? Генерал Дуайт Эйзенхауэр известен своей фразой: «Я всегда убеждался, что планы бесполезны, но без планирования не обойтись». Он имел в виду, что сражения настолько хаотичны, что невозможно предсказать все, что может произойти, или иметь вариант для любой ситуации. В этом смысле планы бесполезны на поле сражения (известная фраза прусского полководца Гельмута фон Мольтке гласит: «Ни один план не выживает после первого контакта с врагом»). Но никакая армия не может одержать победу без планирования и организации. Благодаря планированию генерал узнает возможности армии, как взаимодействуют различные армейские подразделения и так далее.

Точно так же у вас всегда должен быть план решения задачи. Он может и не выжить после первого контакта с врагом, вы можете отказаться от него, как только начнете писать код в редакторе, но план у вас быть должен.

Без плана вы просто хватаетесь за соломинку в надежде преуспеть, но ваши шансы на удачу близки к шансам обезьянки написать пьесу Шекспира, беспорядочно нажимая клавиши на клавиатуре. Счастливые случайности редки, а если такая случайность все же происходит, то ей все равно потребуется план. Многие слышали историю о том, как был открыт пенициллин: исследователь по имени Александр Флеминг забыл прикрыть на ночь чашку Петри, используемую для культивирования бактерий, а на утро обнаружил, что плесень замедлила рост бактерий в чашке. Однако Флеминг не сидел в праздном ожидании счастливой случайности, он ставил эксперименты, делал это тщательным и контролируемым образом и поэтому смог распознать важность того, что увидел в чашке Петри. Должен заметить: если бы я обнаружил плесень на чем-то, что оставил неприкрытым на ночь, это явно не привело бы к важному научному открытию.

Планирование также позволяет вам устанавливать промежуточные цели и достигать их. Без плана у вас лишь одна цель: решить задачу. И до того, как вы решите эту задачу, вы не будете чувствовать, что достигли чего-то. Возможно, как вы уже убедились на своем опыте, программы редко делают что-то полезное, пока не будут близки к завершению процесса создания. Таким образом, работа на достижение только основной цели неизбежно ведет к разочарованию, и вы не получите удовлетворения до тех пор, пока не дойдете до конца. Однако если вместо этого вы создадите план, в котором

установите несколько последовательных меньших целей, даже если некоторые из них кажутся напрямую не связанными с основной целью, вы все равно будете измеримо продвигаться к решению задачи и чувствовать, что потратили время не зря. По окончании каждой рабочей сессии вы сможете проверить элементы плана и обрести уверенность, что сможете решить задачу, а не получить еще больше разочарований.

Переформулируйте задачи

Как показал пример задачи с лисицей, гусем и кукурузой, переформулирование условий может привести к ценным результатам. В некоторых случаях задача, которая выглядит сложной, может оказаться очень простой при иной постановке или при использовании других определений. Переформулирование задачи похоже на то, как вы обходите гору, перед тем как взойти на нее. Прежде чем начать подъем, почему бы не рассмотреть склон со всех сторон в поиске самого легкого пути к вершине?

Переформулирование иногда показывает, что цель — это совсем не то, что мы изначально предполагали. Однажды я прочитал историю про бабушку, которая следила за внучкой, занимаясь при этом вязанием. Чтобы успеть довязать, бабушка посадила внучку в переносной манеж, но ребенку не нравилось сидеть в манеже, и девочка все время плакала. Бабушка перепробовала все возможные игрушки, чтобы сделать времяпрепровождение в манеже веселым, пока не поняла, что посадить ребенка в манеж было ошибкой изначально. Цель состояла в том, чтобы бабушка могла вязать в тишине. Решение: дать ребенку спокойно играть на ковре, пока бабушка вяжет, сидя в манеже. Переформулирование задачи — это мощный прием, но многие программисты не пользуются им, так как он не подразумевает непосредственно написание кода или хотя бы разработку решения. Это еще одна причина, почему просто необходимо иметь план. Без плана ваша единственная цель — получить работающий код, а переформулирование задачи отводит вас дальше от написания кода. Имея план, вы можете сделать «формальную переустановку задачи» первым пунктом.

Даже если переформулирование не приводит к очевидному ноу-хау, оно все равно может быть полезным. Например, если ваш босс или преподаватель поставил вам задачу, вы можете показать перефразированную версию человеку, который вам ее поставил, чтобы убедиться в правильном понимании проблемы. Кроме того, переформулирование задачи может быть необходимым предварительным шагом при использовании других общих техник, таких как уменьшение и разделение задачи.

В более широком смысле переформулирование может трансформировать все проблемные участки. Так, техника, которую я использую для рекурсивных решений и которой поделюсь в одной из глав, представляет собой метод переформулирования рекурсивных

задач так, чтобы у меня была возможность работать с ними как с итеративными задачами.

Делите задачи

Поиск способа разделить задачу на несколько шагов или этапов может значительно упростить задачу. Если вы сможете разделить задачу на две подзадачи, то вам может показаться, что решить каждую из подзадач будет вдвое проще, чем изначальную задачу; но обычно это еще проще.

Приведем аналогию. Она, возможно, уже знакома вам, если вы встречались с общими алгоритмами сортировки. Предположим, у вас есть 100 файлов, которые необходимо поместить в коробку в алфавитном порядке. Ваш базовый метод алфавитной сортировки — это, по сути, сортировка методом вставки. Вам нужно выбрать случайным образом один файл и поместить его в коробку, затем поместить второй файл в правильном отношении с первым файлом и продолжать так же, размещая каждый следующий файл в правильном отношении с предыдущими. Таким образом, файлы в коробке будут находится в алфавитном порядке в любой момент времени. Предположим, что кто-то изначально разделил файлы на 4 группы примерно одинакового размера, А–Ж, З–О, П–Ц и Ч–Я, и сказал вам отсортировать каждую группу файлов по алфавиту по отдельности, а затем поочередно сбросить их в коробку.

Если бы в каждой группе было примерно по 25 файлов, то кому-то могло бы показаться, что расстановка по алфавиту 4 групп из 25 файлов означала бы такое же количество работы, как и расстановка по алфавиту одной группы из 100 файлов. Однако на самом деле такая группировка означает значительно *меньшее* количество работы, так как оно возрастает по мере возрастания количества уже расставленных файлов: чтобы узнать, куда поместить новый файл, вам нужно будет посмотреть каждый файл в коробке. (Если вы сомневаетесь, представьте более экстремальную ситуацию: сравните упорядочение 50 групп по 2 файла, что вы сможете сделать, возможно, меньше чем за минуту, с упорядочением одной группы из 100 файлов.)

Таким же образом деление задачи может значительно снизить ее сложность. Сочетание техник программирования гораздо сложнее, чем использование этих техник по отдельности. Например, фрагмент кода, в котором используется последовательность инструкций if внутри цикла while, который также находится внутри цикла for, написать и прочитать сложнее, чем фрагмент кода, в котором все эти управляющие инструкции используются последовательно.

В последующих главах мы обсудим конкретные способы деления задач. Но вы всегда должны помнить об этой возможности. Помните, что такие задачи, как головоломка со скользящими плитками, зачастую скрывают потенциальную возможность разделения на подзадачи. Иногда метод деления задачи заключается в ее упрощении, что мы вскоре обсудим.

Начните с того, что знаете

Начинающим писателям часто дают совет: пишите о том, что знаете. Это не значит, что писатели должны пытаться создавать работы лишь на основе ситуаций и людей, которых они непосредственно наблюдали в жизни. Будь это так, то у нас не было бы романов в стиле фэнтези, исторических романов или произведений других популярных жанров. Но это значит, что, чем дальше писатель уходит от своего собственного опыта, тем сложнее писать.

Точно так же и в программировании, вам следует начинать с чего-то, что вы уже знаете, как реализовать, и продолжать двигаться вперед. Например, после того, как вы разделили задачу на несколько частей, завершите те части, которые вы уже знаете, как запрограммировать. Тот факт, что у вас есть уже частично работающее решение, может спровоцировать появление идей касательно решения оставшихся частей задачи. Также вы, возможно, заметили, что лейтмотивом решения всех задач является полезное продвижение к цели, что помогает обрести уверенность в том, что вы в конце концов сможете выполнить задание. Начав с того, что знаете, вы обретаете уверенность и получаете толчок по направлению к цели.

Озвученная максима «начните с того, что вы знаете» также применима и в случаях, когда вы еще не разделили задачу. Представьте, что кто-то создал полный список всех навыков программирования: написание классов C++, сортировка списков, поиск наибольшего элемента массива и так далее. В любой момент вашего развития как программиста у вас будут такие навыки, которыми вы владеете хорошо, навыки, которые вы можете использовать, но это требует усилий, а также навыки, которыми вы еще не овладели. Некая конкретная задача может быть или не быть полностью решаема с помощью навыков, которыми вы уже обладаете, но прежде, чем искать посторонней помощи, вам следует исследовать задачу с помощью тех навыков, каковыми вы уже овладели. Если представить, что решение задачи по программированию аналогично проекту ремонта в доме, то вам сначала нужно попробовать выполнить ремонтные работы с помощью имеющихся инструментов, прежде чем выдвигаться в хозяйственный магазин.

Эта техника также следует принципам, которые мы уже обсуждали. Она следует плану и привносит порядок в наши усилия. Начав исследование задачи с применением уже имеющихся навыков, мы сможем узнать больше о самой задаче и ее окончательном решении.

Упрощайте задачи

Эта техника предполагает, что, когда вы сталкиваетесь с задачей, которую не можете решить, вы уменьшаете ее объем, добавляя или снимая некоторые ограничения, создавая тем самым задачу, решение которой вам известно. Вы увидите эту технику в действии в последующих главах, а сейчас рассмотрим простой пример. Предположим,

вам дана последовательность координат в трехмерном пространстве, и вы должны найти координаты, наиболее близкие друг к другу. Если вы не сразу знаете, как решить эту задачу, существуют различные способы упрощения данной задачи с целью поиска ее решения. Например, что, если координаты находятся в двухмерном пространстве вместо трехмерного? Если это не помогает, то что, если точки лежат вдоль одной линии, так что координаты — это только отдельные числа (например, тип **double** в языке C++)? Теперь, по существу, вопрос лишь найти в списке чисел два с минимальным модулем разности.

Или вы могли бы упростить задачу, оставив координаты в трехмерном пространстве, но имея только три значения вместо последовательности произвольного размера. Поэтому вместо разработки алгоритма для нахождения наименьшего расстояния между любыми двумя координатами речь идет только о сравнении координаты А с координатой Б, затем Б с В, а потом А с В.

Эти редукции по-разному упрощают задачу. Первая устраняет необходимость вычисления расстояния между трехмерными точками. Может быть, мы еще не знаем, как это сделать, но еще до того, как выясним это, мы можем добиться прогресса в решении задачи. Вторая, напротив, фокусируется почти полностью на вычислении расстояния между трехмерными точками, но устраняет проблему нахождения минимального значения в последовательности значений произвольного размера.

Конечно, для решения первоначальной задачи нам, в конечном итоге, потребуются навыки, связанные с обоими упрощениями. Тем не менее упрощение позволяет нам работать над более простой задачей, даже если мы не можем найти способ разделить задачу на шаги. По сути, это похоже на преднамеренный, но временный «Кобаяси Мару». Мы знаем, что не работаем над полной задачей, но приведенная задача имеет достаточно много общего с полной, что позволит нам достигнуть прогресса в достижении окончательного решения. Зачастую программисты убеждаются, что у них есть все отдельные навыки, необходимые для решения задачи, и, написав код для решения каждого отдельного аспекта задачи, они понимают, как объединить различные фрагменты кода в единое целое.

Упрощение задачи также позволяет точно определить, где находится оставшаяся трудность. Начинающим программистам часто приходится искать помощь опытных программистов, но это может расстроить обоих, если программист, борющийся с задачей, не может точно описать необходимую помощь. Никто никогда не хочет свести все к фразе: «Вот моя программа, и она не работает. Почему?» Используя технику упрощения задач, можно точно определить то, какая помощь необходима, сказав что-то вроде: «Вот код, который я написал. Как видите, я знаю, как найти расстояние между двумя трехмерными координатами, и знаю, как проверить, меньше ли

одно расстояние другого. Но я не могу найти общего решения для нахождения пары координат с минимальным расстоянием».

Ищите аналогии

Аналогия для наших целей — это сходство между текущей задачей и уже решенной, которая может быть использована для решения текущей задачи. Сходство может принимать различные формы. Иногда это означает, что две задачи в действительности являются одной и той же задачей. Это та ситуация, с которой мы столкнулись при решении задачи с лисицей, гусем и кукурузой и задачи замка Кварраси.

Большинство аналогий не столь прямолинейны. Иногда сходство касается только части задач. Например, две задачи с обработкой с чисел могут быть разными во всех аспектах, за исключением того, что обе предполагают работу с номерами, требующими большей точности, чем предоставляют встроенные типы данных с плавающей точкой. Вы не сможете использовать эту аналогию для решения всей задачи, но если вы уже поняли, как справиться с задачей дополнительной точности, вы сможете справиться с этой же задачей снова.

Хотя распознавание аналогов — это самый важный способ увеличить вашу скорость и улучшить умение решать задачи, это также самый трудный для развития навык. Причина, по которой это так сложно, заключается в том, что вы не сможете находить аналогии, пока у вас не будет целого архива предыдущих решений, на которые вы сможете ссылаться.

Именно здесь малоопытные программисты часто пытаются сделать ход конем, находя код, похожий на необходимый, и изменяя его под свои нужды. Однако по нескольким причинам это ошибка. Во-первых, если вы не выполните свое решение самостоятельно, вы не будете полностью понимать и знать его работу изнутри. Проще говоря, очень сложно правильно изменить программу, которую вы не полностью понимаете. Вам не нужно иметь уже написанный код, чтобы полностью понять принципы его работы, но если вы не смогли написать код, ваше понимание его будет обязательно ограничено. Во-вторых, каждая успешная программа, которую вы пишете, больше, чем решение текущей задачи: это потенциальный источник аналогий для решения будущих задач. Чем больше вы полагаетесь на код других программистов сейчас, тем больше вам придется полагаться на него в будущем. Мы подробно поговорим о «хорошем повторном использовании» и «плохом повторном использовании» в главе 7.

Экспериментируйте

Иногда лучший способ добиться прогресса — это попробовать что-то и понаблюдать за результатами. Обратите внимание, что эксперимент — это не то же самое, что попытка угадать. Когда вы пытае-

тесь угадать, вы вводите код и надеетесь, что он сработает, не имея твердой уверенности в его работоспособности. Эксперимент — это контролируемый процесс. Вы выдвигаете гипотезу о том, что произойдет, когда будет выполнен определенный код, тестируете код и проверяете, верна ли ваша гипотеза. Из этих наблюдений вы получаете информацию, которая поможет вам решить исходную задачу.

Экспериментирование может быть особенно полезно при работе с интерфейсами программирования приложений или библиотеками классов. Предположим, вы пишете программу, которая использует библиотечный класс, представляющий вектор (в этом контексте, одномерный массив, который автоматически растет с добавлением большего количества элементов), но вы никогда раньше не использовали этот векторный класс и не уверены в том, что происходит при удалении элемента из вектора. Вместо того чтобы с трудом продвигаться вперед с решением исходной задачи, пока неопределенности вихрятся у вас в голове, вы можете написать небольшую отдельную программу, чтобы просто поиграть с векторным классом и специально опробовать ситуации, которые вас беспокоят. Если вы потратите немного времени на программу «вектор-демонстратор», она может стать вашим справочником для будущей работы с классом.

Другие формы экспериментов аналогичны отладке. Предположим, что какая-то программа производит результат, отличный от ваших ожиданий, например, если результат численный, то вы хоть и ожидаете появление чисел, но в обратном порядке. Если вы не видите, почему это происходит после анализа вашего кода, в качестве эксперимента вы можете попробовать изменить код, чтобы умышленно зеркально отобразить вывод (возможно, путем запуска цикла в обратном направлении). Получившиеся в результате этого изменения в программном выводе или отсутствие таковых может выявить задачу в изначальном исходном коде или может выявить пробел в вашем понимании. В любом случае, вы будете ближе к решению.

Не расстраивайтесь

Последний прием — это не столько прием, сколько максима: не расстраивайтесь. Когда вы расстроены, вы не будете думать так же четко, вы не будете работать так же эффективно, и работа займет больше времени и будет казаться сложнее. Хуже того, разочарование, как правило, растет само по себе, начинаясь с небольшого раздражения и заканчиваясь гневом.

Когда я даю этот совет новым программистам, они часто возражают, что, хотя они согласны с моей точкой зрения в принципе, они, тем не менее, не могут контролировать свои разочарования. Но просить программиста не расстраиваться из-за отсутствия успеха, это как попросить ребенка не кричать, если он наступил на гвоздь, не так ли? Ответ — нет. Когда некто наступает на гвоздь, через центральную нервную систему незамедлительно происходит отправ-

ка сильного сигнала, на который реагируют глубинные структуры мозга. Если вы не знаете, что вот-вот наступите на гвоздь, то для вас будет невозможно своевременно отреагировать и оказать противодействие автоматическому отклику мозга. Поэтому мы разрешим ребенку прокричаться.

Программист не находится с этим мальчиком в одной лодке. Рискуя быть похожим на гуру саморазвития, разочарованный программист не реагирует на внешние раздражители. Расстроенный программист не зол на исходный код, отображенный на мониторе, хотя программист в этом смысле тоже может выражать свое расстройство. Скорее всего, расстроенный программист зол на самого себя. Источник раздражения также является и конечным пунктом, и это сознание программиста.

Когда вы позволяете себе расстроиться — и я использую глагол «позволять» умышленно, — вы, по сути, даете себе оправдание в продолжающихся неудачах. Представьте, что вы работаете над сложной задачей и чувствуете как усиливается ваше разочарование. Спустя часы, взглянув на послеобеденное время, стиснутые зубы и карандаши, брошенные со злости в стену, вы говорите себе, что добились бы гораздо большего, если бы у вас получилось успокоиться. На самом же деле вы, возможно, решили, что поддаться злости гораздо проще, чем посмотреть в глаза сложной задаче.

В конечном итоге избегать расстройств — это то решение, которое вы должны принять. Однако есть некоторые мысли, которые вы можете использовать и которые вам помогут. Прежде всего, никогда не забывайте первое правило, которое гласит, что у вас всегда должен быть план и что несмотря на то, написание кода, который решает исходную задачу, — это цель этого плана, это не единственный шаг этого плана. Таким образом, если у вас есть план и вы следуете ему, то вы продвигаетесь вперед, и вы должны этому верить. Если вы выполнили все шаги по своему первоначальному плану, и вы все еще не готовы начать писать программный код, значит, пришло время сделать другой план.

Кроме того, когда дело доходит до ультиматума «разочароваться» или сделать «паузу», делайте паузу. Одна из хитростей — одновременно решать несколько задач. Таким образом, если решение одной задачи зашло в тупик, то вы можете перенаправить свои усилия на решение другой. Обратите внимание, что если у вас получится разделить задачу, то вы сможете воспользоваться этой техникой для решения одной и той же задачи: просто заблокируйте ту часть задачи, что завела вас в тупик, и начните работать над каким-то другим аспектом. Если у вас нет другой задачи, которую вы могли бы порешать, встаньте со стула и займитесь чем-то другим, что стимулирует кровообращение мозга, но не напрягает его: прогуляйтесь, постирайте белье, выполните зарядку с растяжками (если вы подписываетесь на работу программистом, который должен сидеть за компьютером це-

лый день, я очень рекомендую вам выполнять такую гимнастику!). Не терзайтесь над решением задачи до победного конца.

Упражнения

Помните: чтобы по-настоящему изучить что-то, что вы должны применить на практике, так что выполните как можно больше упражнений. В этой первой главе мы, конечно, еще не обсуждаем программирование, но даже несмотря на это я рекомендую вам попробовать некоторые упражнения. Воспринимайте эти вопросы как разминку для пальцев, прежде чем мы начнем играть настоящую музыку.

1.1. Попробуйте решить головоломку «Судоку» средней сложности (вы можете найти их по всему Интернету и в газетных киосках), экспериментируя с различными стратегиями и принимая во внимание результаты. Можете ли вы написать общий план решения «Судоку»?

1.2. Рассмотрим вариант головоломки со скользящими плитками, где на плитки нанесены изображения, а не цифры. Насколько это увеличивает сложность и почему?

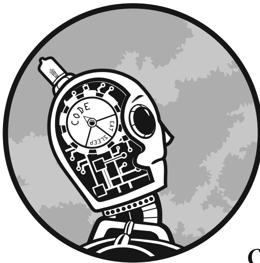
1.3. Найдите стратегию для решения головоломки со скользящими плитками, отличную от моей.

1.4. Найдите старомодные головоломки наподобие задачи про лисицу, гуся и кукурузу и попытайтесь их решить. Многие из великих головоломок были созданы или популяризированы Сэмом Лойдом, поэтому вы можете поискать его имя. Кроме того, как только вы откроете (или сдадитесь и прочитаете) решение, подумайте, как вы могли бы сделать более легкую версию головоломки. Что бы вы изменили? Ограничения или просто формулировку?

1.5. Попробуйте написать несколько явных стратегий для других традиционных игр с карандашом и бумагой, например для кроссвордов. С чего начать? Что вы должны делать, когда зашли в тупик? Даже простые газетные игры, такие как составление слов из набора букв, полезны для обдумывания стратегии.

2

Истинные головоломки



В этой главе мы начнем работать с программным кодом. Хотя для последующих глав потребуются знания программирования на среднем уровне, навыки программирования, требуемые в этой главе, настолько просты, насколько это возможно. Это не означает, что все эти головоломки будут легкими. Но это позволит вам сосредоточиться на решении задач, а не на синтаксисе языка. Это решение проблемы в самом чистом виде. Когда вы выясните, что вам нужно сделать, перевод ваших мыслей в код на языке C++ будет прост. Помните, что простое прочтение этой книги приносит лишь долю возможного результата. По ходу обсуждения вы должны прорабатывать каждую задачу, которая кажется вам нетривиальной, и пытаться решить ее самосто-

ятельно, прежде чем читать о моем подходе. В конце главы попробуйте выполнить некоторые упражнения, многие из которых будут продолжением обсуждаемых нами проблем.

Обзор языка C++, используемого в этой главе

В этой главе используется базовый язык C++, с которым вы должны быть знакомы, в том числе управляющие инструкции `if`, `for`, `while` и `do-while`, а также `switch`. Возможно, вы еще не очень уверенно решаете исходные задачи с этими инструкциями, но, в конце концов, книга-то как раз об этом. Однако вы должны понимать синтаксис написания этих инструкций или иметь под рукой хороший справочник по C++.

Вы также должны знать, как писать и вызывать функции. Для упрощения задачи мы будем использовать стандартные потоки `cin` и `cout` для ввода и вывода. Чтобы использовать эти потоки, включите в свой код необходимый заголовочный файл `iostream` и добавьте инструкции `using` для стандартных объектов потока:

```
#include <iostream>
using std::cin;
using std::cout;
```

Для краткости эти инструкции не будут приведены в листингах кода. Включение этих инструкций предполагается по умолчанию во всех программах по необходимости.

Шаблоны вывода

В этой главе мы разберем три задачи. Так как я буду обширно использовать приемы разделения и редукции задач, каждая из этих задач станет источником нескольких подзадач. В первом разделе давайте опробуем несколько программ, производящих шаблонный вывод некой геометрической фигуры. Такие программы развивают навыки написания циклов.

Задача: половина квадрата

Напишите программу, в которой используется только две инструкции вывода: `cout << "#"` и `cout << "\n"` для создания узора из символов # в виде половины квадрата 5×5 (или прямоугольного треугольника):

```
#####
####
###
##
#
```

Это еще один отличный пример важности ограничений. Если мы проигнорируем требование, что можем использовать только две

инструкции вывода, одна из которых выводит только один символ #, а вторая — символ переноса строки, мы можем написать «Кобаяси Мару» и тривиально решить эту задачу. Однако с этими ограничениями для решения задачи нам придется воспользоваться циклами.

Возможно, у вас в голове уже есть решение, но давайте предположим, что его нет. Первый хороший прием — это упрощение. Как мы можем упростить задачу до такой степени, где ее будет легко решить? Что если бы нам надо было вывести целый квадрат, а не его половину?

Задача: квадрат (упрощение задачи с половиной квадрата)

Напишите программу, в которой используется только две инструкции вывода: `cout << "#"` и `cout << "\n"` для создания узора из символов # в виде квадрата 5×5:

```
#####
#####
#####
#####
#####
```

Этого может быть достаточно для начала, но давайте предположим, что мы не знаем, как решить и эту задачу. Мы могли бы еще сильнее упростить задачу, поставив условие вывести одну линию, а не целый квадрат.

Задача: линия (еще большее упрощение задачи с половиной квадрата)

Напишите программу, в которой используется только две инструкции вывода: `cout << "#"` и `cout << "\n"` — для вывода линии из пяти символов #:

```
#####
```

Теперь перед нами возникает тривиальная задача, которую мы можем решить с помощью цикла `for`:

```
for (int hashNum = 1; hashNum <= 5; hashNum++) {
   cout << "#";
}
cout << "\n";
```

Отсюда вернемся к предыдущему упрощению: фигуре полного квадрата. Полный квадрат — это всего лишь повторяющаяся пять раз линия из символов #. Мы знаем, как создать повторяющийся код: нужно просто лишь написать цикл. Таким образом, мы можем превратить цикл во вложенный:

```
for (int row = 1; row <= 5; row++) {
   for (int hashNum = 1; hashNum <= 5; hashNum++) {
```

```
    cout << "#";
  }
  cout << "\n";
}
```

Мы поместили весь код из предыдущего листинга в цикл, таким образом, чтобы он повторялся пять раз, выводя пять строк, каждая из которых — линия из символов #. Мы уже приближаемся к окончательному решению. Как же изменить код, чтобы он выводил узор в виде половины квадрата? Если мы взглянем на последний листинг и сравним с нужным выводом в виде половины квадрата, то увидим, что вся проблема в условном выражении `hashNum <= 5`. Это условное выражение выводит одинаковую линию из символов # на каждой строке. Что нам требуется — это механизм, который настраивал бы количество символов на каждой строке так, чтобы на первой строке было пять символов, на второй четыре и так далее.

Чтобы понять, как это сделать, давайте поставим еще один эксперимент по упрощению программы. Повторимся, всегда проще работать над проблемным участком задачи, изолировав его. На минуту давайте забудем о символах # и поговорим только о числах.

Задача: посчитать на уменьшение, считая на увеличение

Напишите строку кода в указанном участке цикла нижеприведенного листинга. Программа распечатывает цифры от 5 до 1 в порядке убывания, при этом каждая цифра находится на отдельной строке.

```
for (int row = 1; row <= 5; row++) {
  cout << ❶ выражение << "\n";
}
```

Мы должны найти *выражение* ❶, оно равняется `5`, когда `row = 1` и `4`, когда `row = 2`. Если мы хотим получить выражение, которое будет уменьшаться по мере возрастания значения переменной `row`, то первая мысль, которая может нас посетить — это приписать знак минус к значениям `row`, умножив их на `-1`. Это приведет к появлению уменьшающихся чисел, но это будут не те числа, что нам нужны. Тем не менее мы будем ближе, чем думаем. Какова тогда разница между нужным значением и значением, полученным в результате умножения `row` на `-1`? В табл. 2.1 приводится анализ.

Табл. 2.1. Вычисление требуемого значения из значения переменной row

row	Требуемое значение	row*-1	Отклонение от Требуемого значения
1	5	-1	6
2	4	-2	6
3	3	-3	6
4	2	-4	6
5	1	-5	6

Отклонение представляет собой константу, равную 6. Это значит, что требуемое выражение — это `row * -1 + 6`. Вспомнив уроки алгебры, мы можем упростить это выражение до `6 – row`. Давайте проверим:

```
for (int row = 1; row <= 5; row++) {
   cout << 6 – row << "\n";
}
```

Отлично — все работает! Если бы код не заработал, то наша ошибка, скорее всего, была бы незначительной, благодаря тем тщательным шагам, которые мы предприняли. Повторимся, с такими маленькими и простыми блоками кода гораздо проще экспериментировать. Теперь давайте возьмем это выражение и воспользуемся им для ограничения вложенного цикла:

```
for (int row = 1; row <= 5; row++) {
   for (int hashNum = 1; hashNum <= 6 – row; hashNum++) {
      cout << "#";
   }
   cout << "\n";
}
```

Использование техники упрощения требует больше шагов, чтобы добраться от описания до завершенной программы, но сделать отдельный шаг гораздо проще. Представьте использование нескольких ремней и подвижных блоков для подъема тяжелого объекта: чтобы приложить столько же подъемной силы, вам потребуется тянуть веревку на большее расстояние, но каждое усилие будет иметь меньшую нагрузку на ваши мышцы.

Прежде чем мы пойдем дальше, давайте решим еще одну задачу с геометрической фигурой.

Задача: равнобедренный треугольник

Напишите программу, в которой используется только две инструкции вывода: `cout << "#"` и `cout << "\n"` для создания узора из символов # в виде равнобедренного треугольника:

```
#
##
###
####
###
##
#
```

Мы не будем проходить все шаги, которые прошли при решении предыдущей задачи, потому как нет в этом необходимости. Задача «Равнобедренный треугольник» аналогична задаче «Половина квадрата», а это значит, что при ее решении мы можем воспользоваться

теми сведениями, которые узнали при решении предыдущего упражнения. Помните максиму «начните с того, что вы уже знаете»? Давайте начнем с перечисления приемов и техник задачи «Половина квадрата», которые мы можем использовать при решении этой задачи. Мы знаем, как:

- вывести на экран строку символов конкретной величины с помощью цикла;
- вывести на экран последовательность строк с помощью вложенных циклов;
- отображать различное, а не одно фиксированное количество символов в каждой строке с помощью математической формулы;
- найти корректное алгебраическое выражение путем экспериментирования и анализа.

Рис. 2.1 резюмирует наше текущее положение. В первом ряду показана предыдущая задача «Половина квадрата». Мы также можем видеть требуемый узор из символов (а), узор «линию» (б), узор «квадрат» (в), а также числовую последовательность (г), которая трансформирует узор «квадрат» в узор «половина квадрата». Во втором ряду показана текущая задача «Равнобедренный треугольник». Мы опять видим требуемый узор из символов (д), узор «линию» (е), узор «квадрат» (ж), а также числовую последовательность (з).

```
#####              #####     5
####               #####     4
###       #####    #####     3
##                 #####     2
#                  #####     1

(а)       (б)      (в)       (г)

#                  ####      1
##                 ####      2
###                ####      3
####      ####     ####      4
###                ####      3
##                 ####      2
#                  ####      1

(д)       (е)      (ж)       (з)
```

Рис. 2.1. Различные компоненты, необходимые для решения задач с фигурами.

На данном этапе у нас не будет проблем с воспроизведением (е), так как этот узор почти идентичен (б). Мы также должны быть в состоянии вывести узор (ж), так как это просто узор (в) с большим количеством строк и меньшим количеством символов в строке. Наконец, если бы кто-то предложил нам алгебраическое выражение,

результатом выполнения которого была бы числовая последовательность (з), у нас не было бы сложностей с воспроизведением требуемого узора (д).

Таким образом, большая часть умственной работы, требующейся для решения задачи «Равнобедренный треугольник», уже выполнена. Более того, мы знаем точно, какую умственную работу осталось сделать: поиск выражения, которое выдало бы числовую последовательность (з). Таким образом, именно сюда мы должны направить свое внимание. Мы могли бы либо взять завершенный код задачи «Половина квадрата» и экспериментировать до тех пор, пока не получится произвести требуемую числовую последовательность, либо сделать предположение и составить таблицу наподобие табл. 2.1, чтобы посмотреть, подстегнет ли это наше воображение.

В этот раз давайте поэкспериментируем. В задаче «Половина квадрата» очень хорошо работало вычитание от ряда с большим числом. Поэтому давайте посмотрим, что у нас получится, если мы пропустим `row` через цикл от 1 до 7 и вычтем `row` из 8. Результат показан на рис. 2.2 (б). Это не то, что нам нужно. Какой мы можем сделать следующий шаг? В предыдущей задаче было необходимо, чтобы числа шли по убыванию, а не по возрастанию, поэтому мы вычитали переменную цикла из большего числа. В этой же задаче нам нужно сначала идти по увеличению, а потом по уменьшению. Будет ли правильно вычитать из числа где-то посередине? Если мы заменим выражение `8 - row` из предыдущего кода на `4 - row`, то также не получим правильной последовательности (рис. 2.2 (в)). Последовательность похожа на правильную, если не смотреть на знаки минус у последних трех чисел. Что если мы воспользуемся функцией выдачи модуля значения, чтобы убрать знак минус? Выражение `abs(4 - row)` выводит значения на рис. 2.2 (г). Мы так близки! Я практически уже чувствую вкус победы! Проблема лишь в том, что мы сначала идем по уменьшению, а потом по увеличению, хотя нам нужно идти в обратном направлении. Но как же нам прийти от имеющейся числовой последовательности к требуемой?

Давайте попробуем взглянуть на числа на рис. 2.2 (г) под другим углом. Что если бы мы считали количество пробелов вместо количества символов #, как показано на рис. 2.2 (д)? В случае если мы считаем количество пробелов, колонка (г) представляет собой *правильную* числовую последовательность. Чтобы получить правильное количество символов #, представьте, что в каждой строке четыре клетки, а затем вычтите количество пробелов. Если в каждом ряду четыре клетки, из которых `abs(4 - row)` пустые, значит количество клеток с символом # можно вычислить с помощью выражения `4 - abs(4 - row)`. Это выражение работает, подключите его и проверьте!

1	7	3	3	#
2	6	2	2	##
3	5	1	1	###
4	4	0	0	####
3	3	-1	1	###
2	2	-2	2	##
1	1	-3	3	#
(а)	(б)	(в)	(г)	(д)

Рис. 2.2. Различные компоненты, необходимые для решения задачи «Равнобедренный треугольник»

Мы избежали большей части работы через проведение аналогий и решили оставшуюся часть задачи через эксперименты. Это отличный подход, когда новая задача очень похожа на ту, что мы уже решили.

Обработка ввода

Предыдущие программы только производили вывод. Давайте теперь сменим тему и попробуем программы, полностью посвященные обработке ввода. У всех этих программ будет одно общее ограничение: ввод должен считываться символ за символом, а программа должна обрабатывать каждый символ перед считыванием следующего. Иными словами, программы не будут хранить символы в структурах данных для последующей обработки, но будут обрабатывать символы по мере их ввода.

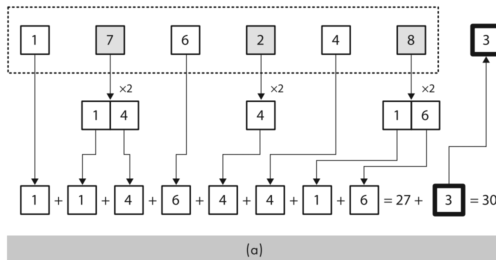
В первой задаче мы произведем проверку идентификационного номера. В современном мире почти у всего есть идентификационный номер, будь то ISBN или номер клиента. Иногда требуется ручной ввод этих номеров, что создает возможность возникновения ошибки. Если ошибочно введенный номер не соответствует ни одному действительному идентификационному номеру, то система может просто его отбраковать. А что если номер неверен, но при этом действителен? Что если кассир пытается увеличить кредит вашего счета за возвращенный товар и вводит при этом номер лицевого счета другого клиента? Ваш кредит будет передан другому клиенту. Чтобы избежать подобной ситуации, были созданы системы обнаружения ошибок в идентификационных номерах. Эти системы пропускают идентификационный номер через некую формулу, что приводит к генерации одной или двух дополнительных цифр, которые, в свою очередь, становятся частью расширенного идентификационного номера. При изменении любой цифры исходная часть номера больше не будет соответствовать добавочным цифрам — и система может отбраковать введенный номер.

Задача: проверка контрольной суммы Луна

Формула Луна — это широко используемая система проверки идентификационных номеров. Используя исходное число, увеличить вдвое значение каждой цифры. Затем сложить значения отдельных цифр (если удвоенное значение имеет две цифры, то сложить эти цифры по отдельности). Идентификационный номер проходит проверку, если получившаяся сумма делится на 10.

Напишите программу, которая принимала бы идентификационный номер произвольной длины и определяла бы по формуле Луна, действителен ли номер. Программа должна обрабатывать каждый символ перед чтением следующего.

Процесс звучит немного сложно, но пример позволит вам все прояснить. Наша программа будет только проверять идентификационный номер, без генерации проверочной цифры. Давайте пройдем через обе части процесса: вычисление проверочного номера и проверка результата. Данный процесс представлен на рис. 2.3. В части (а) мы вычисляем проверочную цифру. Исходный идентификационный номер 176248 показан в поле, обведенном пунктирной линией. Каждая вторая цифра, начиная с самого правого числа исходного номера (которая после добавления контрольной цифры становится второй справа), умножается на два. После этого все цифры складываются. Обратите внимание, что если при умножении на два результат — двузначное число, то каждая их цифр этого числа рассматривается по отдельности. Например, когда при умножении 7 на 2 в результате мы получаем 14, то к контрольной сумме прибавляется *14*, но по отдельности *1* и *4*. В данном случае контрольная сумма 27, значит контрольная цифра 3, потому как именно это значение приведет к общей сумме 30. Помните, что контрольная сумма последнего числа должна делиться на 10, а следовательно, заканчиваться на 0.



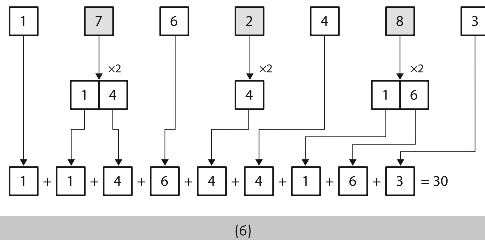


Рис. 2.3. Формула контрольной суммы Луна

В части (б) мы проверяем число 1762483, которое теперь содержит проверочную цифру. Такой процесс мы будем пользоваться для решения этой задачи. Как и прежде, мы удваиваем каждую вторую цифру, начиная с цифры слева от проверочной, и суммируем значения всех цифр для определения контрольной суммы. Так как контрольная сумма делится на 10, число проходит проверку.

Разбиение проблемы на части

Программа, которая решит эту задачу, содержит несколько сложностей, с которыми нужно разобраться. Одна из них — умножение цифр на 2, что непросто, так как мы определяем то, какие из цифр нужно умножать, начиная справа. Помните, что мы не будем считывать и сохранять все цифры для *последующей* обработки. Мы будем обрабатывать их по мере ввода. Проблема в том, что мы будем получать цифры слева направо, но чтобы знать, какие из цифр умножать на 2, нужно было бы получать их справа налево. Мы знали бы, которые из цифр умножать, если бы знали количество цифр в идентификационном номере, но мы этого не знаем, так как, по условию задачи, идентификационный номер имеет произвольную длину. Еще одна сложность в том, что если умноженное на 2 число больше 10, то мы должны обрабатывать цифры, составляющее получившееся число, по отдельности. Нам также нужно будет определить, когда идентификационный номер считан полностью. Наконец, нам нужно будет придумать, как считывать номер по одной цифре. Иными словами, пользователь будет вводить одно длинное число, но мы будем считывать его так, будто бы пользователь вводил цифры, как отдельные числа.

Так как нам всегда необходим план, следует создать список этих подзадач и разрешить их по отдельности:

- знать, какие цифры умножить на 2;
- обрабатывать число 10 и большие числа по отдельным цифрам;
- знать, что мы достигли конца номера;
- считывать каждую цифру по отдельности.

Для решения задач мы будем работать над отдельными частями прежде, чем писать окончательное решение. Таким образом, нет никакой необходимости работать над этими подзадачами в какой-то конкретной последовательности. Начните с той, которая выглядит проще остальных, или, если вы хотите побороться, начните с той, что кажется самой сложной. Или же просто выберите ту, что кажется самой интересной.

Давайте начнем с решения проблемы с числом 10 и больше. Это как раз та ситуация, когда ограничения задачи упрощают, а не усложняют ее решение. Вычисление суммы цифр целого числа произвольной длины может быть сложной задачей самой по себе. Но каков здесь диапазон возможных значений? Если мы начнем с отдельных

цифр 0–9 и умножим их на 2, то максимальное значение, которое мы можем получить — 18. Таким образом, возможности только две. Если удвоенное значение — однозначное число, то нам не нужно ничего больше делать. Если же удвоенное значение равно 10 и больше, то оно должно быть в диапазоне 10–18, таким образом, первая цифра всегда 1. Давайте поставим небольшой программный эксперимент, чтобы подтвердить этот подход:

```
  int digit;
  cout << "Enter a single digit number, 0-9: ";
  cin >> digit;
❶ int doubledDigit = digit * 2;
  int sum;
❷ if (doubledDigit >= 10) sum = ❸1 + doubledDigit % 10;
  else sum = doubledDigit;
❹ cout << "Sum of digits in doubled number: " << sum << "\n";
```

ПРИМЕЧАНИЕ. *Оператор % называется оператором деления с остатком. Для положительных целых чисел он вернет остаток от целочисленного деления. Например, результатом операции 12 % 10 будет 2, так как остаток деления 12 на 10 как раз 2.*

Это прямолинейный код: программа считывает цифру, умножает ее на 2 ❶, суммирует цифры удвоенного числа ❷ и выводит сумму ❹. Однако вся суть эксперимента заключается в подсчете суммы удвоенного числа, больше, чем 10 ❸. Как и в случае с подсчетом количества символов #, необходимых для конкретной строки в задачах с фигурами, выделение этого вычисления в короткую программу упростит экспериментирование. Даже если мы не получим правильную формулу с первого раза, мы все равно уверены, что вскоре ее найдем.

Прежде чем мы вычеркнем эту подзадачу из нашего списка, давайте преобразуем код в короткую функцию, которую сможем использовать для упрощения последующих листингов:

```
int doubleDigitValue(int digit) {
   int doubledDigit = digit * 2;
   int sum;
   if (doubledDigit >= 10) sum = 1 + doubledDigit % 10;
   else sum = doubledDigit;
   return sum;
}
```

Теперь давайте поработаем над считыванием отдельных цифр идентификационного номера. Мы могли бы попробовать разрешить любую другую подзадачу при желании, но я думаю, что именно эта– хороший выбор, так как она позволит нам естественным образом ввести идентификационный номер во время тестирования других частей нашей задачи.

Если мы считываем идентификационный номер в виде числовых данных (например, `int`), то получим лишь одно длинное чис-

ло, а впереди будет много работы. Кроме того, существует ограничение на длину целочисленного значения, которое мы можем считать, а по условию задачи идентификационный номер имеет произвольную длину. Таким образом, нам придется считывать символ за символом. Это значит, мы должны убедиться, что знаем, как считать символ, представляющий цифру, и привести его целочисленному типу, над которым мы сможем совершать математические действия. Чтобы увидеть, что произойдет, если мы возьмем символьное значение и используем его в целочисленном выражении напрямую, взгляните на следующий листинг, который включает также образец программного вывода.

```
  char digit;
  cout << "Enter a one-digit number: ";
❶ digit = cin.get();
  int sum = digit;
  cout << "Is the sum of digits " << sum << "? \n";

❷ Enter a one-digit number: 7
  Is the sum of digits 55?
```

Обратите внимание, что мы используем метод `get` ❶, так как базовый оператор извлечения (как в `cin >> digit`) пропускает пробелы. В данном случае это не проблема, но, как вы увидите далее, это вызовет трудности впоследствии. Вы также видите проблему в примере ввода и вывода ❷. Компьютеры работают с числовыми данными, поэтому отдельные символы представлены в виде целочисленных кодов символов. В разных операционных системах могут использоваться разные системы кодировки символов, но в этой книге мы сосредоточим внимание на общеупотребительной системе ASCII. В этой системе символу 7 присвоено значение кода символа 55, поэтому, когда мы обрабатываем этот символ как целое число, то получаем 55. Нам нужен механизм для преобразования *символа* 7 в *целое число* 7.

Задача: преобразовать символ цифры в целое число

Напишите программу, которая считывает символ, введенный пользователем и представляющий цифру от 0 до 9, а затем выводит целое число для демонстрации результата.

В задачах с геометрическими фигурами из предыдущего раздела у нас уже были переменные одного диапазона значений, которые мы хотели преобразовать в переменные с другим диапазоном. Мы составляли таблицу с колонками для текущих и требуемых значений, а затем выясняли разницу между значениями этих двух групп. Это аналогичная задача, а значит, мы вновь можем воспользоваться идеей с таблицей, как показано в табл. 2.2.

Табл. 2.2. Коды символов и требуемые целочисленные значения

Символ	Код символа	Требуемое целое	Разница
0	48	0	48
1	49	1	48
2	50	2	48
3	51	3	48
4	52	4	48
5	53	5	48
6	54	6	48
7	55	7	48
8	56	8	48
9	57	9	48

Разница между кодом символа и требуемым целым числом всегда 48, следовательно, нам нужно вычесть это значение. Вы, возможно, обратили внимание, что это код символа — 0. Это всегда будет так, потому, что все системы кодировки символов всегда хранят символы цифр в порядке возрастания, начиная с 0. Таким образом, мы можем создать более общее и более удобочитаемое решение, вычитая символ 0, а не используя некое предустановленное значение, каковым является 48:

```
char digit;
cout << "Enter a one-digit number: ";
cin >> digit;
int sum = digit – '0';
cout << "Is the sum of digits " << sum << "? \n";
```

Теперь мы можем перейти дальше к выяснению, какие именно цифры нужно умножить на 2. Эта часть проблемы может потребовать несколько шагов, прежде чем мы решим ее. Поэтому давайте попробуем упростить проблему. Что если бы мы изначально ограничили себя номером фиксированной длины? Это помогло бы подтвердить наше понимание общей формулы и продвинуться к окончательной цели. Давайте попробуем ограничить длину до шести, это достаточно длинный номер, который позволит представить общую сложность задачи.

Задача: проверка контрольной суммы Луна, фиксированная длина

Напишите программу, которая принимала бы идентификационный номер (и его проверочную цифру) длиной шесть цифр и определяла бы по формуле Луна, действителен ли номер. Программа должна обрабатывать каждый символ перед чтением следующего.

Как и прежде, мы можем еще упростить задачу, чтобы облегчить начало работы настолько, насколько это вообще возможно. Что если мы изменим формулу так, что ни одна из цифр не удваивается? Тогда программе остается лишь считывать цифры и суммировать их.

Задача: проверка простой контрольной суммы, фиксированная длина

Напишите программу, которая принимала бы идентификационный номер (и его проверочную цифру) длиной шесть цифр и определяла бы по простой формуле, где значения всех цифр суммируются, а результат подвергается проверке на делимость на 10, чтобы сказать, действителен ли номер. Программа должна обрабатывать каждый символ перед чтением следующего.

Поскольку мы знаем, как считывать отдельные цифры в виде символов, то можем довольно легко решить эту задачу с простой контрольной суммой и числом фиксированной длины. Нам нужно лишь считать шесть цифр, сложить их и определить, делится ли сумма на 10.

```
char digit;
int checksum = 0;
cout << "Enter a six-digit number: ";
for (int position = 1; position <= 6; position ++) {
   cin >> digit;
   checksum += digit - '0';
}
cout << "Checksum is " << checksum << ". \n";
if (checksum % 10 == 0) {
   cout << "Checksum is divisible by 10. Valid. \n";
} else {
   cout << "Checksum is not divisible by 10. Invalid. \n";
}
```

На этом этапе нам нужно добавить программную логику, которая бы действительно проверяла формулу Луна, что значит умножение на 2 каждой второй цифры, начиная со второй цифры справа. Так как в настоящий момент мы ограничили себя шестизначными номерами, нам нужно удвоить цифры в позициях один, три и пять, считая слева. Иными словами, мы умножаем цифру на 2, если находится в нечетной позиции. Мы можем различить четные и нечетные позиции с помощью оператора деления с остатком (%), поскольку число считается по определению четным, если делится на два без остатка. Таким образом, если результат выражения `position % 2` равен 1, значит, `position` — нечетное число — и мы должны его удвоить. Следует помнить, что в данном контексте *удвоить* значит не только умножение цифры на два, но и сложение цифр получившегося числа, если это число больше или равно 10. В этом нам поможет предыдущая функция. Когда по формуле Луна нужно удвоить цифру, мы просто передаем эту цифру нашей функции и пользуемся возвращенным результатом. Обобщив все вышесказанное, нам остается лишь изменить код цикла `for` из предыдущего листинга:

```
for (int position = 1; position <= 6; position ++) {
   cin >> digit;
   if (position % 2 == 0) checksum += digit - '0';
      else checksum += doubleDigitValue(digit - '0');
}
```

К этому моменту мы многого достигли на пути решения этой задачи, но предстоит сделать еще пару шагов, прежде чем мы сможем написать код для идентификационных номеров произвольной длины. Чтобы окончательно решить эту проблему, мы должны последовать правилу «разделяй и властвуй». Предположим, я попросил вас изменить код выше для работы с номерами длиной 10 или 16 цифр. Эта задача была бы тривиальной: вам понадобилось бы лишь заменить число 6, использованное в качестве верхнего предела цикла, на другое значение. Но предположим, что я попросил вас проверить семизначные номера. Это потребовало бы некоторого дополнительного изменения, так как если количество цифр нечетное и мы удваиваем каждую вторую цифру, начиная со второй цифры справа, то первая цифра *слева* уже не будет удваиваться. В этом случае вам нужно удвоить четные позиции: 2, 4, 6 и так далее. Отложив ненадолго этот вопрос, давайте разберемся, каким образом нам обрабатывать номера четной длины.

Первая сложность, с которой мы сталкиваемся, — это определение того, что мы достигли конца номера. Если пользователь вводит многозначное число и нажимает клавишу **Enter**, а мы считываем ввод цифра за цифрой, какой символ считывается после последней цифры? На самом деле это зависит от операционной системы, но мы лишь напишем экспериментальный код:

```
cout << "Enter a number: ";
char digit;
while (true) {
   digit = cin.get();
   cout << int(digit) << " ";
}
```

Цикл выполняется бесконечно, но делает свою работу. Я ввел номер `1234` и нажал клавишу **Enter**. Результатом было `49 50 51 52 10` (на основе ASCII, это будет зависеть от операционной системы). Таким образом, 10 — это то, что я ищу. Вооружившись этой информацией, мы можем заменить цикл `for` в предыдущем коде на цикл `while`:

```
   char digit;
   int checksum = 0;
❶ int position = 1;
   cout << "Enter a number with an even number of digits: ";
❷ digit = cin.get();
   while ❸(digit != 10) {
      ❹ if (position % 2 == 0) checksum += digit - '0';
         else checksum += doubledDigitValue(digit - '0');
      ❺ digit = cin.get();
      ❻ position++;
   }
   cout << "Checksum is " << checksum << ". \n";
   if (checksum % 10 == 0) {
      cout << "Checksum is divisible by 10. Valid. \n";
   } else {
      cout << "Checksum is not divisible by 10. Invalid. \n";
   }
```

В этом коде position больше не является управляющей переменной цикла for, значит, мы должны отдельно ее инициализировать ❶ и постепенно увеличивать ❻. Теперь циклом управляет условное выражение ❸, сверяющее ввод с кодом символа конца строки. Так как мы имеем значение для проверки при первом проходе по циклу, то считываем первое значение перед началом цикла ❷ и затем каждое последующее значение внутри цикла ❺, после обрабатывающего кода.

Повторюсь, этот код обработает номер любой четной длины. Для обработки номера нечетной длины нам потребовалось бы только изменить обрабатывающий код, обратив логику условия инструкции if ❹, чтобы удваивать числа в четных позициях, а не в нечетных.

Это наконец охватывает все возможности. Длина идентификационного номера может быть либо четной, либо нечетной. Если бы мы заранее знали длину номера, то знали бы, удваивать ли нам четные или нечетные позиции числа. Однако у нас нет такой информации до тех пор, пока мы не достигли конца числа. Неужели, учитывая все ограничения, решение проблемы невозможно? Если мы знаем, как решить эту проблему для нечетного количества цифр и для четного, но не знаем количество цифр в номере до тех пор, пока полностью не прочитаем число, как мы можем решить эту проблему?

Возможно, вы уже видите ответ для этой задачи. Если нет, то это не потому, что ответ сложен, а потому, что он спрятан в деталях. Чем мы могли бы воспользоваться здесь, так это аналогия, но мы еще не сталкивались с аналогичными ситуациями. Вместо этого мы создадим собственную аналогию. Давайте сформулируем задачу, явно посвященную этой самой ситуации, и посмотрим, будет ли нам полезно взглянуть проблеме прямо в глаза. Отчистите свое сознание от концепций, сложившихся у вас на основе проведенной работы, и прочитайте условие следующей задачи.

Задача: положительное или отрицательное

Напишите программу, которая считывала бы 10 целых чисел пользователя. По окончании ввода всех чисел пользователь может попросить отобразить количество положительных и количество отрицательных чисел.

Это простая задача, которая, кажется, не должна вызывать каких-либо сложностей. Нам понадобится лишь две переменных: одна для подсчета положительных чисел и еще одна для подсчета отрицательных чисел. Когда пользователь уточняет запрос в конце программы, то для ответа на этот запрос нужно лишь воспользоваться соответствующей переменной:

```
int number;
int positiveCount = 0;
int negativeCount = 0;
for (int i = 1; i <= 10; i++) {
```

```
   cin >> number;
   if (number > 0) positiveCount++;
   if (number < 0) negativeCount++;
}
char response;
cout << "Do you want the (p)ositive or (n)egative count? ";
cin >> response;
if (response == 'p')
   cout << "Positive count is " << positiveCount << "\n";
if (response == 'n')
   cout << "Negative count is " << negativeCount << "\n";
```

Пример демонстрирует метод, который нам необходимо использовать для решения проблемы контрольной суммы Луна: одновременно отслеживать увеличивающуюся контрольную сумму обоими способами, как будто бы введенный номер имеет нечетную длину и одновременно четную. Когда мы дойдем до конца числа и узнаем действительную длину, у нас будет правильная контрольная сумма в одной или другой переменной.

Соберем все детали вместе

Мы уже поставили галочки напротив всех пунктов нашего изначального списка задач. Теперь пришло время собрать все вместе и решить эту задачу. Так как мы уже по отдельности решили все подзадачи и знаем точно, что нужно делать, то можем использовать для справки предыдущие программы для быстрого достижения конечного результата:

```
char digit;
int oddLengthChecksum = 0;
int evenLengthChecksum = 0;
int position = 1;
cout << "Enter a number: ";
digit = cin.get();
while (digit != 10) {
   if (position % 2 == 0) {
      oddLengthChecksum += doubleDigitValue(digit - '0');
      evenLengthChecksum += digit - '0';
   } else {
      oddLengthChecksum += digit - '0';
      evenLengthChecksum += doubleDigitValue(digit - '0');
   }
   digit = cin.get();
   position++;
   }
   int checksum;
❶  if ((position - 1) % 2 == 0) checksum = evenLengthChecksum;
   else checksum = oddLengthChecksum;
   cout << "Checksum is " << checksum << ". \n";
   if (checksum % 10 == 0) {
      cout << "Checksum is divisible by 10. Valid. \n";
   } else {
      cout << "Checksum is not divisible by 10. Invalid. \n";
   }
```

Обратите внимание, что при проверке четности или нечетности длины введенного номера ❶ мы вычитаем 1 из `position`. Делаем мы

это потому, что последний считываемый символ в цикле — это символ конца строки, а не последняя цифра номера. Мы также могли бы написать такое проверочное выражение — `(position % 2 == 1)`, но оно сложнее для восприятия. Иными словами, проще сказать «если `position - 1` — четное число, использовать четную контрольную сумму», нежели сказав «если `position` — нечетное число, использовать четную контрольную сумму», помнить, почему это правильно.

Пока что это самый длинный из всех листингов, которые мы с вами видели, но у меня нет необходимости делать аннотации ко всем участком кода и описывать, как работает каждый блок, так как вы уже видели работу этих блоков по отдельности. И в этом большой плюс плана. Впрочем, следует отметить, что *мой* план необязательно будет *вашим.* Вероятно, те сложности, что я увидел в условии исходной задачи, и шаги, которые я предпринял, чтобы решить эту задачу, отличаются от того, что бы увидели и сделали вы. Ваш программистский опыт и те проблемы, которые вы успешно решили, определяют, какие из частей этой задачи для вас тривиальны, а какие сложны, а следовательно и то, какие шаги вам нужно предпринять для решения этой задачи. В предыдущем разделе мог быть такой момент, когда шаг, предпринятый мною, казался вам лишним отклонением для выяснения чего-то, что уже очевидно для вас. И наоборот, возможно был такой момент, когда я в спешке пропустил что-то сложное для вас. Кроме того, если бы вы решали эту задачу самостоятельно, возможно, ваша программа была бы столь же успешной, но значительно отличающейся от моей. Нет «правильных» решений проблемы, если любая программа, соответствующая ограничениям, считается решением, а для каждого решения отсутствует «правильный» путь достижения.

Просмотрев все шаги для решения задачи и учитывая относительную краткость кода, вы, возможно, захотите редуцировать какие-либо шаги в собственном процессе решения проблем. Я бы предупредил вас об опасности этого позыва. Всегда лучше сделать больше шагов, чем пытаться сделать слишком многое за один шаг, даже если какие-то из шагов кажутся чересчур тривиальными. Помните о целях решения проблем. Главная цель, разумеется, — найти программное решение, которое решало бы поставленную задачу и соответствовало бы всем ограничениям. Второстепенная цель — найти такое программное решение за кратчайшее время. Минимизация количества шагов не является целью, и никто не должен знать, сколько шагов вы сделали. Представьте, что вам нужно подняться на вершину крутого холма, и к этой вершине есть пологая, но длинная и извилистая тропа. Невнимание к тропе и восхождение на склон напрямую с подножья формально потребует меньше шагов, чем путь по тропе, но будет ли такое восхождение более быстрым? Наиболее вероятный результат такого прямого восхождения — вы просто упадете и сдадитесь.

Также вспомните последнее из моих правил решения проблем: *не расстраивайтесь*. Чем больше вы пытаетесь сделать за один шаг, тем настойчивее вы приглашаете в гости потенциальное расстройство. Даже если вы отступитесь от сложного шага и разобьете его на несколько подшагов, вред все равно будет нанесен, так как психологически вы будете ощущать, что движетесь назад, вместо того, чтобы двигаться вперед. Когда я обучаю начинающих программистов пошаговому подходу, иногда находится студент, который жалуется: «Эй, этот шаг слишком прост», на что я отвечаю: «На что ты жалуешься?» Если изначально задача выглядела сложной, но вы разбили ее на несколько частей, так, что каждая часть кажется легко решаемой, я скажу вам: «Мои поздравления! Это именно то, на что вы должны надеяться».

Отслеживание состояния

Последняя задача, которую мы рассмотрим в этой главе, также и самая сложная. Эта задача состоит из большого количества различных частей и имеет запутанное описание, что проиллюстрирует важность разбиения сложной проблемы.

Задача: декодирование сообщения

Некое сообщение было закодировано в виде текстового потока, который должен быть прочитан символ за символом. Поток содержит последовательность целых чисел, разделенных запятыми. Каждое из этих чисел — положительное целое, которое может быть представлено типом `int` языка C++. Однако то, какой символ, представлен тем или иным целым числом, зависит от текущего *режима декодирования*. Всего существует три режима: *верхний регистр*, *нижний регистр* и *пунктуация*.

В режиме *верхнего регистра* каждое целое число представляет прописную букву: целое число, взятое по модулю 27, означает букву алфавита (где 1 = A и так далее). Таким образом, введенное значение 143 в режиме верхнего регистра будет означать букву *H*, так как 143 по модулю 27 равняется 8, а H — восьмая буква латинского алфавита.

Режим *нижнего регистра* работает аналогичным образом, но со строчными буквами: остаток от деления целого числа на 27 представляет строчную букву (где 1 = a и так далее). Таким образом, введенное значение 56 в режиме верхнего регистра будет означать букву b, так как 56 по модулю 27 равняется 2, а b — вторая буква латинского алфавита.

В режиме *пунктуации* целое число уже берется по модулю 9 и согласно интерпретации в приведенной ниже табл. 2.3. Таким образом, 19 по модулю 9 будет означать восклицательный знак, так как 19 по модулю 9 равняется 1.

В начале каждого сообщения устанавливается режим декодирования верхнего регистра. Каждый раз, когда результат операции деления по модулю (27 или 9, в зависимости от режима) равен 0, происходит переключение режима декодирования. Если в текущий момент установлен режим декодирования верхнего регистра, то происходит переключение на режим декодирования нижнего регистра. Если же в текущий момент установлен режим декодирования нижнего регистра, то происходит переключение на режим декодирования пунктуации. Если установлен режим декодирования пунктуации, то происходит переключение на режим декодирования верхнего регистра.

Табл. 2.3. Режим декодирования пунктуации

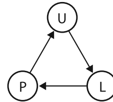
Число	Символ
1	!
2	?
3	,
4	.
5	(пробел)
6	;
7	"
8	'

Как и в случае с проверочной формулой Луна, мы познакомимся с конкретным примером, чтобы убедиться в том, что разобрались во всех шагах. На рис. 2.4 показан образец декодирования. Исходный поток ввода показан в верхней части рисунка. Этапы обработки указаны сверху вниз. В колонке (а) показано текущее число на вводе. В колонке (б) показан текущий режим, принимающий циклически одно из трех состояний: верхний регистр (U), нижний регистр (L) и пунктуация (P). В колонке (в) приводится делитель для текущего режима. В колонке (г) показан остаток от деления текущего ввода из колонки (а) на текущий делитель в колонке (в). Результат показан в колонке (д), и это либо символ, либо, если результат в колонке (г) равен 0, то переключение к следующему по порядку режиму.

Исходный ввод:
18,12312,171,763,98423,1208,216,11,500,18,241,0,32,20620,27,10

(а)	(б)	(в)	(г)	(д)
18	U	27	18	R
12312	U	27	0	→ L
171	L	27	6	i
763	L	27	7	g
98423	L	27	8	h
1208	L	27	20	t
216	L	27	0	→ P
11	P	9	2	?
500	P	9	5	
18	P	9	0	→ U
241	U	27	25	Y
0	U	27	0	→ L
32	L	27	5	e
20620	L	27	19	s
27	L	27	0	→ P
10	P	9	1	!

Цикл режимов:



Декодированное сообщение:
Right? Yes!

Рис. 2.4. Образец процессинга для задачи «Декодирование сообщения»

Как и в случае с предыдущей проблемой, мы можем начать с явного рассмотрения того, какие навыки потребуются для разработки решения. Нам нужно считать строку символов до тех пор, пока не достигнем конца строки. Символы представляют последовательность целых чисел, поэтому нам нужно считать символы цифр и конвертировать их в целые числа для последующей обработки. Получив целые числа, мы должны конвертировать целое число в один символ для вывода. Наконец, нам нужен некий способ для отслеживания режима декодирования, чтобы мы могли знать, должно ли текущее целое быть декодировано символ нижнего регистра, символ верхнего регистра или же в знак пунктуации. Давайте преобразуем вышесказанное в формальный список.

- Считывать символ за символом до достижения конца строки.
- Преобразовать последовательность символов, представляющие цифры, в целое число.
- Преобразовать целое число в диапазоне 1 — 26 в символ верхнего регистра.
- Преобразовать целое число в диапазоне 1 — 26 в символ нижнего регистра.
- Преобразовать целое число в диапазоне 1 — 8 в пунктуационный символ, в соответствии с табл. 2.3.
- Отслеживать режим декодирования.

Решить первый пункт мы можем благодаря предыдущей задаче. Более того, несмотря на то, что при решении задачи с проверочной формулой Луна мы только работали с отдельными цифрами, я подозреваю, что наши действия для решения задачи будут полезны и для второго элемента списка. Окончательный код алгоритма Луна еще, возможно, свеж в вашей памяти, но если между тем кодом и решением этой задачи вы отложите чтение книги, то по продолжении вам понадобится вернуться и перечитать код предыдущей задачи. Обычно, если описание текущей проблемы вызывает у вас дежа-вю, вы захотите найти похожий код в своих архивах для изучения.

Давайте займемся оставшимися элементами списка. Должно быть, вы заметили, что в списке каждое из преобразований вынесено отдельным элементом. Я подозреваю, что преобразование числа в букву нижнего регистра будет очень сильно походить на преобразование числа в букву верхнего регистра, однако, возможно, преобразование в пунктуационный символ потребует несколько иного подхода. В любом случае, нет ничего плохого в разбиении списка на такие детальные подпункты, это лишь означает, что вы сможете вычеркивать элементы чаще.

СОХРАНЕНИЕ КОДА ДЛЯ ПОСЛЕДУЮЩЕГО ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ

Сходства между элементами этой и предыдущей задач показывают важность отдельного сохранения исходного кода в виде, удобном для последующего повторного использования. Разработчики программного обеспечения очень много говорят о *повторном использовании кода*, которое происходит, когда вы используете участки старого программного обеспечения для создания нового. Зачастую это подразумевает использование инкапсулированного компонента или непосредственное повторное использование исходного кода. Однако очень важно иметь легкий доступ к ранее написанным решениям. Даже если вы не копируете старый код напрямую, это позволит вам повторно использовать уже приобретенные навыки и приемы без необходимости изобретать велосипед повторно. Для максимизации этого преимущества, старайтесь сохранять весь исходный код, который вы пишете (конечно же, не забывая о соглашениях по интеллектуальной собственности, возможно, заключенных между вами и вашими работодателями или клиентами).

Сможете ли вы использовать преимущества уже написанных программ, по большей части, зависит от того, с какой скрупулезностью вы сохраняете их: код, который вы не можете найти — это код, который вы не можете использовать. Если вы используете пошаговый подход и пишите отдельные программы для проверки своих идей, прежде, чем интегрировать эти отдельные программы в целое решение, обязательно сохраняйте и эти промежуточные программы. Позже, когда сходство программы, над которой вы работаете в данный момент, и старой программы будет находиться в сфере, для которой вы уже писали тестовую программу, вы найдете такой подход очень удобным.

Давайте начнем с преобразований целых чисел в символы. Из программы с формулой Луна мы знаем код, который требуется для считывания символьной цифры в диапазоне 0 — 9 и конвертации его в целое число в диапазоне от 0 до 9. Как мы можем расширить этот метод для работы с многозначными числами? Давайте рассмотрим простейший случай — двузначные числа. Это выглядит вполне понятно: в двузначном числе первая цифра представляет десятки, следовательно, нужно умножить эту цифру на 10, а затем прибавить значение второй цифры. Например, если бы число было 35, то после чтения отдельных составляющих цифр как символов 3 и 5 и преобразования этих символов в целые числа 3 и 5, выполнив выражение 3 * 10 + 5, мы получили бы нужное целое число. Давайте подтвердим это предположение с помощью кода:

```
cout << "Enter a two-digit number: ";
char digitChar1 = cin.get();
char digitChar2 = cin.get();
int digit1 = digitChar1 – '0';
int digit2 = digitChar2 – '0';
int overallNumber = digit1 * 10 + digit2;
cout << "That number as an integer: " << overallNumber << "\n";
```

Работает! Программа выводит то же самое двузначное число, что мы ввели. Однако мы сталкиваемся с проблемой, когда пытаемся расширить этот метод. Эта программа использует две разные

переменные для хранения двух введенных символов, и, хотя это не вызывает никаких проблем, мы конечно же не хотим использовать этот подход как общее решение. А если бы захотели, то нам понадобилось бы столько переменных, сколько цифр составляет введенное число. Это приведет к беспорядку, кроме того, такую программу было бы трудно изменить, если диапазон возможных чисел во входном потоке изменился. Нам нужно более общее решение этой подзадачи преобразования символов в целые числа. Первый шаг к поиску этого общего решения заключается в сокращении предыдущего кода до двух переменных: одной `char` и одной `int`:

```
  cout << "Enter a two-digit number: ";
❶ char digitChar = cin.get();
❷ int overallNumber = (digitChar - '0') * 10;
❸ digitChar = cin.get();
❹ overallNumber += (digitChar - '0');
  cout << "That number as an integer: " << overallNumber << "\n";
```

Мы достигаем этого за счет выполнения всех вычислений, касающихся первой цифры еще до считывания второй цифры. После считывания первого цифрового символа в первом шаге ❶, мы преобразуем его в целое число, умножаем на 10 и сохраняем результат во втором шаге ❷. После считывания второй цифры ❸ мы прибавляем ее целочисленное значение к общему значению ❹. Эти шаги аналогичны предыдущему коду, только в этот раз мы используем лишь две переменных: одну для последнего прочитанного символа и еще одну для общего значения целого числа. Следующий шаг — рассмотреть расширение этого метода для трехзначных чисел. Как только мы сделаем это, скорее всего, мы увидим закономерность, которая позволит нам создать общее решение для любого количества цифр.

Однако когда мы попытаемся реализовать этот план, то столкнемся с проблемой. С двузначными числами мы умножали левую цифру на 10, потому что левая цифра была в позиции разряда десятков, однако в случае с трехзначными числами самая левая цифра будет находиться в позиции разряда сотен, а значит, нам потребовалось бы умножить эту цифру на 100. После этого мы могли бы считать среднюю цифру, умножить ее на 10, прибавить к общему значению, а затем считать последнюю цифру и также прибавить ее к общему значению. Это должно сработать, однако такой подход не приближает нас к созданию общего решения. Вы видите проблему? Задумайтесь о вышесказанном: *самая левая цифра будет находиться в позиции разряда сотен*. В случае с общим решением мы не будем знать, из скольких цифр состоит число до тех пор, пока не достигнем следующей запятой. Самой левой цифре числа с неизвестным количеством цифр нельзя присвоить позицию сотен или вообще какую-либо другую позицию. Так каким же образом мы узнаем, какой множитель нам использовать с каждой цифрой прежде, чем прибавлять ее к общему значению? Или, может, нам нужен абсолютно иной подход?

Как всегда, зайдя в тупик, хорошей идеей будет создать упрощенную задачу и поработать над ней. Сложность здесь в том, что мы не знаем, из скольких цифр будет состоять число. Самая простая задача, в которой преодолевается та же самая сложность, будет предполагать работу с одним из двух возможных количеств цифр.

Задача: чтение трех- или четырехзначного числа

Напишите программу, которая считывала бы число символ за символом и преобразовывала бы его в целое число, используя только одну переменную типа `char` и одну переменную типа `int`. Число может быть либо трехзначным, либо четырехзначным.

Проблема незнания общего количества символов до самого конца, но потребности в общем количестве с самого начала аналогична проблеме с формулой Луна. В том случае мы не знали, состоит ли идентификационный номер из четного или нечетного количества символов. В том случае наше решение состояло в одновременном подсчете результата двумя способами и использовании подходящего результата в конце. Можем ли мы сделать что-то похожее и здесь? Если число состоит либо из трех, либо из четырех цифр, то существует лишь два варианта. Если число трехзначное, то самая левая цифра соответствует количеству сотен. Если же число четырехзначное, то самая левая цифра означает количество тысяч. Мы могли бы произвести вычисления одновременно, как будто бы перед нами трехзначное число и четырехзначное, а в конце выбрать подходящее число, однако условие задачи позволяет нам иметь только одну числовую переменную. Поэтому чтобы продвинуться в решении большой задачи, давайте ослабим это ограничение.

Задача: чтение трех- или четырехзначного числа, дальнейшее упрощение

Напишите программу, которая считывала бы число символ за символом и преобразовывала бы его в целое число, используя только одну переменную типа `char` и две переменные типа `int`. Число может быть либо трехзначным, либо четырехзначным.

Теперь мы можем применить метод «подсчет двумя способами». Мы обработаем первые три цифры двумя способами, а затем посмотрим, есть ли еще и четвертая цифра:

```
  cout << "Enter a three-digit or four-digit number: ";
  char digitChar = cin.get();
❶ int threeDigitNumber = (digitChar - '0') * 100;
❷ int fourDigitNumber = (digitChar - '0') * 1000;
  digitChar = cin.get();
  threeDigitNumber += (digitChar - '0') * 10;
  fourDigitNumber += (digitChar - '0') * 100;
  digitChar = cin.get();
  threeDigitNumber += (digitChar - '0');
  fourDigitNumber += (digitChar - '0') * 10;
  digitChar = cin.get();
```

```
   if ❸ (digitChar == 10) {
      cout << "Number entered: " << threeDigitNumber << "\n";
   } else {
    ❹ fourDigitNumber += (digitChar – '0');
      cout << "Number entered: " << fourDigitNumber << "\n";
   }
```

После считывания крайней левой цифры мы умножаем целочисленное значение на 100 и сохраняем результат в трехзначной переменной ❶. Мы также умножаем целочисленное значение на 1000 и сохраняем результат в четырехзначной переменной ❷. Эта же схема повторяется и для следующих двух цифр. Вторая цифра обрабатывается и как показатель десятков в трехзначном числе, и как сотен — в четырехзначном. Третья цифра обрабатывается и как показатель десятков, и как единиц. После считывания четвертого символа мы проверяем, является ли он символом конца строки, сравнив его с числом 10 ❸ (как и в предыдущей задаче, это значение может отличаться в зависимости от операционной системы). Если перед нами символ конца строки, а значит, было введено трехзначное число, в противном случае нам все еще нужно прибавить значение единиц к общему значению ❹.

Теперь нам нужно придумать способ, как избавиться от одной лишней целочисленной переменной. Предположим, мы полностью убрали переменную fourDigitNumber. Значение переменной threeDigitNumber как и прежде будет присвоено корректно, но когда мы дойдем до момента, где потребуется переменная fourDigitNumber, в нашем распоряжении ее просто не будет. Существует ли какой-либо способ определить значение, которое было бы присвоено переменной fourDigitNumber, используя значение переменной threeDigitNumber? Предположим, пользователь ввел 1234. После считывания трех первых цифр, значение переменной threeDigitNumber будет равно 123. Значение, которое было бы записано в переменную fourDigitNumber — это 1230. Так как множители для переменной fourDigitNumber в 10 раз больше множителей для threeDigitNumber, первая переменная всегда будет в 10 раз больше второй. Таким образом, нам нужна только одна целочисленная переменная, так как вторая переменная может быть просто при необходимости умножена на 10:

```
cout << "Enter a three-digit or four-digit number: ";
char digitChar = cin.get();
int number = (digitChar – '0') * 100;
digitChar = cin.get();
number += (digitChar – '0') * 10;
digitChar = cin.get();
number += (digitChar – '0');
digitChar = cin.get();
if (digitChar == 10) {
   cout << "Number entered: " << number << "\n";
} else {
   number = number * 10 + (digitChar – '0');
   cout << "Number entered: " << number << "\n";
}
```

Теперь у нас есть пригодный для использования шаблон. Рассмотрите расширение этого кода для обработки пятизначных чисел. После вычисления значения первых четырех цифр мы повторим тот же процесс, который использовали для вычисления четвертой цифры, вместо отображения результата, а именно: прочитаем пятый символ, проверим, является ли этот символ символом конца строки, и если так, то отобразим вычисленное значение, а в противном случае выполним умножение на 10 и прибавим цифровое значение текущего символа:

```
cout << "Enter a number with three, four, or five digits: ";
char digitChar = cin.get();
int number = (digitChar - '0') * 100;
digitChar = cin.get();
number += (digitChar - '0') * 10;
digitChar = cin.get();
number += (digitChar - '0');
digitChar = cin.get();
if (digitChar == 10) {
   cout << "Number entered: " << number << "\n";
} else {
   number = number * 10 + (digitChar - '0');
   digitChar = cin.get();
   if (digitChar == 10) {
      cout << "Number entered: " << number << "\n";
   } else {
      number = number * 10 + (digitChar - '0');
      cout << "Number entered: " << number << "\n";
   }
}
```

На данном этапе мы запросто смогли бы расширить этот код для обработки шестизначных чисел или чисел с меньшим количеством знаков. Шаблон вполне ясен: если следующий символ — это еще одна цифра, то умножить текущее общее значение на 10 и прибавить целочисленное цифровое значение этого символа. Поняв эту логику, мы можем написать цикл для обработки числа любой длины:

```
  cout << "Enter a number with as many digits as you like: ";
❶ char digitChar = cin.get();
❷ int number = (digitChar - '0');
❸ digitChar = cin.get();
  while ❹ (digitChar != 10) {
     ❺ number = number * 10 + (digitChar - '0');
     ❻ digitChar = cin.get();
  }
  ❼ cout << "Number entered: " << number << "\n";
```

Здесь мы считываем первый символ ❶ и определяем его цифровое значение ❷. Затем мы считываем второй символ ❸ и попадаем в цикл, в котором проверяем, не является ли только что прочитанный символ символом конца строки ❹. Если не является, мы умножаем текущее общее значение в цикле на 10 и добавляем цифровое значение ❺ текущего символа перед считыванием следующего символа ❻. Когда достигается конца строки, переменная `number` уже содержит значение, которое мы можем распечатать на экране ❼.

Вышеприведенный код обрабатывает преобразование одного набора символов в соответствующие целочисленные эквиваленты. В финальной версии программы мы будем считывать наборы чисел, разделенные запятыми. Каждое число должно быть отдельно считано и обработано. Как всегда лучше всего начать с размышления о простой ситуации, которая позволяет продемонстрировать проблему. Давайте рассмотрим ввод `101,22[EOL]`, где `[EOL]` для большей ясности явно означает конец строки. Этого нам будет достаточно для изменения условия цикла для проверки ввода запятой или символа конца строки. Затем нам потребовалось бы поместить код, обрабатывающий одно число внутрь большего цикла, повторяющегося до тех пор, пока не будут прочитаны все числа. Таким образом, выполнение внутреннего цикла должно прерываться запятой или `[EOL]`, тогда как выполнение внешнего цикла — только `[EOL]`:

```
❶ char digitChar;
  do {
     digitChar = cin.get();
     int number = (digitChar - '0');
     digitChar = cin.get();
     while ((digitChar != 10) && (digitChar != ',')) {
        number = number * 10 + (digitChar - '0');
        digitChar = cin.get();
     }
     cout << "Number entered: " << number << "\n";
  } while ❷ (digitChar != 10);
```

Это еще один замечательный пример того, как важны маленькие шаги. Несмотря на то, что это лишь короткая программа, ее хитросплетенная природа со вложенным циклом сделала бы написание такого кода довольно сложной задачей, попытайся мы написать эту программу с нуля. Впрочем, программа становится вполне прямолинейной, когда мы приходим к этому коду, делая шаг вперед от предыдущей программы. Объявление переменной `char digitChar` ❶ перенесено на отдельную строку, таким образом, что эта переменная остается видимой для всех участков кода. Оставшаяся часть кода идентичная предыдущему листингу за тем лишь исключением, что этот код помещен внутрь цикла `do-while`, выполнение которого продолжается до тех пор, пока мы не достигаем конца строки ❷.

Разобравшись с этой частью решения, мы можем сконцентрироваться на обработке отдельных чисел. Следующий элемент нашего списка — преобразование числа в диапазоне 1 — 26 в букву A-Z. Если вы попробуете решить эту задачу, то обнаружите, что этот процесс абсолютно противоположен тому процессу, который мы использовали для отдельных символов цифр в соответствующие целочисленные эквиваленты. Если мы можем извлечь код символа 0 для перевода этого символа из символьного диапазона 0 — 9 в целочисленный диапазон 0 — 9, значит, мы сможем прибавить символьный код для перевода 1 — 26 в A-Z. Что произойдет, если мы прибавили `'A'`? Ниже приведена эта попытка, а также пример ввода и вывода:

```
cout << "Enter a number 1-26: ";
int number;
cin >> number;
char outputCharacter;
outputCharacter = ❶ number + 'A';
cout << "Equivalent symbol: " << outputCharacter << "\n";

Enter a number 1-26: 5
Equivalent letter: F
```

Что-то пошло не так... Пятая буква латинского алфавита — **E**, а не **F**. Проблема в том, что мы прибавляем число из диапазона, начинающегося с 1. Когда мы совершали преобразования в обратном направлении, конвертируя символы цифр в целочисленные эквиваленты, мы работали с диапазоном, начинающимся с 0. Мы можем это исправить, изменив арифметическое выражение ❶ с `number + 'A'` на `number + 'A' – 1`. Обратите внимание, что мы могли бы просто посмотреть код символа буквы A (в кодировке ASCII это 65) и использовать значение на один меньше (например, `number + 64` в кодировке ASCII). Однако я предпочитаю первую версию, так как она более удобочитаема. Иными словами, если позже вы вернетесь и просмотрите этот код, то вы быстрее вспомните, что означает выражение `number + 'A' – 1`, чем `number + 64`, так как присутствие символа `'A'` в первом выражении напомнит вам о преобразовании в символы верхнего регистра.

Разобравшись в этом, мы запросто можем адаптировать эту идею для преобразования в символы нижнего регистра, изменив арифметическое выражение ❶ на `number + 'a'-1`. Преобразование таблицы пунктуационных символов не столь же компактно, так как символы пунктуации не появляются в таком порядке ни в таблице ASCII, ни в любой другой системе кодировки символов. Поэтому нам придется решить этот вопрос в лоб:

```
  cout << "Enter a number 1-8: ";
  int number;
  cin >> number;
  char outputCharacter;
❶ switch (number) {
    case 1: outputCharacter = '!'; break;
    case 2: outputCharacter = '?'; break;
    case 3: outputCharacter = ','; break;
    case 4: outputCharacter = '.'; break;
    case 5: outputCharacter = ' '; break;
    case 6: outputCharacter = ';'; break;
    case 7: outputCharacter = '"'; break;
    case 8: outputCharacter = ❷ '\''; break;
  }
  cout << "Equivalent symbol: " << outputCharacter << "\n";
```

Здесь мы используем инструкцию `switch` ❶ для вывода корректного знака пунктуации. Обратите внимание, что для отображения символа одинарной кавычки ❷ мы использовали обратный слеш в качестве знака перехода.

Перед тем как мы соберем все вместе, осталось решить лишь одну подзадачу: переключение между режимами каждый раз, когда последнее введенное значение декодируется в 0. Вспомните, что условие задачи требует, чтобы мы брали целочисленное значение по модулю 27 (если программа в настоящий момент находится в режиме верхнего или нижнего регистра) или 9 (если программа в режиме пунктуации). Когда результат операции равен 0, мы переключаемся к следующему режиму. Что нам потребуется, так это переменная, которая будет хранить текущий режим, а также некую логику внутри цикла «чтение и обработка следующего значения» для переключения режимов при необходимости. Переменная, отслеживающая текущий режим, может быть простым целым числом, но если мы воспользуемся для этих целей перечислением, код будет более удобочитаемым. На заметку: если переменная используется только для отслеживания состояния, и ни у одного из значений нет присущего смысла, то лучше использовать перечисление. В этом случае мы могли бы объявить переменную `int mode`, скажем, значение 1 которой означало бы верхний регистр, 2 — нижний регистр, а 3 — режим пунктуации. Однако в том, почему именно эти значения были выбраны для этих режимов, нет никакого скрытого смысла. Когда мы вернемся к этому коду позже, нам придется вспомнить, что значат некоторые инструкции, например `if (mode==2)`. Если мы воспользуемся перечислением, как в инструкции `(mode == LOWERCASE)`, то ничего не придется вспоминать, так как все и так написано. Далее приведен код, являющийся результатом вышеприведенных рассуждений, а также образец общения пользователя с программой:

```
enum modeType {UPPERCASE, LOWERCASE, PUNCTUATION};
int number;
modeType mode = UPPERCASE;
cout << "Enter some numbers ending with -1: ";
do {
   cin >> number;
   cout << "Number read: " << number;
   switch (mode) {
      case UPPERCASE:
         number = number % 27;
         cout << ". Modulo 27: " << number << ". ";
         if (number == 0) {
            cout << "Switch to LOWERCASE";
            mode = LOWERCASE;
         }
         break;
      case LOWERCASE:
         number = number % 27;
         cout << ". Modulo 27: " << number << ". ";
         if (number == 0) {
            cout << "Switch to PUNCTUATION";
            mode = PUNCTUATION;
         }
         break;
      case PUNCTUATION:
```

```
number = number % 9;
        cout << ". Modulo 9: " << number << ". ";
        if (number == 0) {
           cout << "Switch to UPPERCASE";
           mode = UPPERCASE;
        }
        break;
     }
     cout << "\n";
} while (number != -1);
```

```
Enter some numbers ending with -1: 2 1 0 52 53 54 55 6 7 8 9 10 -1
Number read: 2. Modulo 27: 2.
Number read: 1. Modulo 27: 1.
Number read: 0. Modulo 27: 0. Switch to LOWERCASE
Number read: 52. Modulo 27: 25.
Number read: 53. Modulo 27: 26.
Number read: 54. Modulo 27: 0. Switch to PUNCTUATION
Number read: 55. Modulo 9: 1.
Number read: 6. Modulo 9: 6.
Number read: 7. Modulo 9: 7.
Number read: 8. Modulo 9: 8.
Number read: 9. Modulo 9: 0. Switch to UPPERCASE
Number read: 10. Modulo 27: 10.
Number read: -1. Modulo 27: -1.
```

Мы вычеркнули все пункты нашего списка, поэтому настало время интегрировать эти отдельные листинги кода для создания решения всей программы. Мы могли бы подойти к такой интеграции по-разному. Мы могли бы совместить два фрагмента кода и построить решение на их основе. Например, можно объединить код для чтения и преобразования чисел, разделенных запятыми, с кодом для переключения режимов из крайнего листинга. После этого мы могли бы протестировать интеграцию и добавить код для преобразования каждого числа в соответствующую букву. Или же можно пойти другим путем: превратить листинг с преобразованием чисел в символы в несколько функций, которые мы вызывали бы из главной программы. На этом этапе мы уже перешли от решения задачи к проблемам технологий разработки программного обеспечения, а это совсем другая тема. Мы создали несколько блоков, что было сложной частью, а сейчас нужно лишь собрать их, как показано на рис. 2.5.

Практически каждая строка этой программы была взята из предыдущих листингов, приведенных в этом разделе. Каркас кода ❶ взята из программы, переключающей режимы. Центральный обрабатывающий цикл ❷ позаимствован из нашего кода для считывания последовательности чисел, разделенных запятыми. Наконец вы узнаете код, преобразующий целые числа в буквы верхнего и нижнего регистра и знаки пунктуации ❹. Небольшое количество нового кода отмечено меткой ❸. Инструкции `continue` переносят нас на следующую итерацию цикла, когда последний введенный символ был распознан как команда переключения режимов, пропуская строку `cout << outputCharacter` в конце цикла.

```
❶     char outputCharacter;
      enum modeType {UPPERCASE, LOWERCASE, PUNCTUATION};
      modeType mode = UPPERCASE;
❷     char digitChar;
      do {
         digitChar = cin.get();
         int number = (digitChar - '0');
         digitChar = cin.get();
         while ((digitChar != 10) && (digitChar != ',')) {
            number = number * 10 + (digitChar - '0');
            digitChar = cin.get();
         }
         switch (mode) {
            case UPPERCASE:
               number = number   27;
               outputCharacter = number + 'A' - 1;
               if (number == 0) {
                  mode = LOWERCASE;
❸                 continue;
               }
               break;
            case LOWERCASE:
               number = number   27;
               outputCharacter = number + 'a' - 1;
               if (number == 0) {
                  mode = PUNCTUATION;
                  continue;
               }
               break;
            case PUNCTUATION:
               number = number   9;
❹              switch (number) {
                  case 1: outputCharacter = '!'; break;
                  case 2: outputCharacter = '?'; break;
                  case 3: outputCharacter = ','; break;
                  case 4: outputCharacter = '.'; break;
                  case 5: outputCharacter = ' '; break;
                  case 6: outputCharacter = ';'; break;
                  case 7: outputCharacter = '"'; break;
                  case 8: outputCharacter = '\''; break;
               }
               if (number == 0) {
                  mode = UPPERCASE;
                  continue;
               }
               break;
         }
         cout << outputCharacter;
      } while (digitChar != 10);
      cout << "\n";
```

Рис. 2.5. Собранное решение задачи «Декодировать сообщение»

Несмотря на то, что это «копипаста», это *хорошая* разновидность «копипасты»: вы повторно используете код, который написали сами, и поэтому полностью понимаете его. Как и раньше, подумайте о том, насколько легок был каждый шаг в процессе решения задачи по сравнению с попыткой написать окончательный листинг сразу с нуля. Вне всяких сомнений, хороший программист должен писать окончательный листинг без необходимости выполнения промежуточных шагов но в таком случае были бы неправильные шаги, иногда код выглядит уродливо, строки кода закомментированы, а потом возвращены в программу. Благодаря маленьким шагам, мы заранее выполняем всю грязную работу, а код никогда не выглядит уродливо, так как код, над которым мы работаем в настоящее время, никогда не будет слишком длинным или запутанным.

Заключение

В этой главе мы изучили три задачи. В некотором смысле нам потребовалось пройти три разных пути для их решения. С другой точки зрения, мы проходили один и тот же маршрут каждый раз, потому что мы использовали одну и ту же базовую технику разбиения проблемы на компоненты, написания кода для решения этих компонентов по отдельности, а затем использования знаний, полученных при написании программ, или даже прямого использования строк кода из этих программ для решения исходной задачи. В последующих главах мы не будем явно использовать этот метод при решении каждой задачи, но эта основная идея будет присутствовать всегда: разрубить задачу на куски, с которыми можно справиться.

В зависимости от вашего опыта эти задачи изначально могут показаться вам лежащими в любом участке диапазона сложности: от очень сложных до тривиальных. Вне зависимости от того, насколько сложной задача может показаться изначально, я бы рекомендовал вам использовать эту технику с каждой задачей, с которой вы сталкиваетесь. Не нужно дожидаться встречи с безнадежно сложной задачей, чтобы опробовать эту новую технику. Помните, что одна из целей этой книги — развить вас уверенность в своих способностях решить любую задачу. Попрактикуйтесь в использовании этих техник на «простых» задачах — и у вас будет много энергии, когда вы повстречаетесь со сложными.

Упражнения

Как и прежде, я настоятельно рекомендую вам сделать столько упражнений, сколько сможете. Теперь, когда мы полностью вовлеклись в настоящее программирование, проработка этих упражнений необходима вам для развития ваших навыков решения задач.

2.1. Используя только инструкции вывода одного символа, которые выводят на экран символ #, пробел или конец строки, напишите программу, которая распечатывает следующую фигуру:

```
########
 ######
  ####
   ##
```

2.2. Или как насчет такой?

```
   ##
  ####
 ######
########
########
 ######
  ####
   ##
```

2.3. А вот особенно сложная:

```
#            #
 ##        ##
  ###    ###
   ########
   ########
  ###    ###
 ##        ##
#            #
```

2.4. Создайте свою собственную фигуру: Придумайте свою собственную симметричную фигуру из символов # и проверьте, сможете ли вы написать программу, которая следует правилам программ с фигурами.

2.5. Если вам понравилась задача с формулой Луна, попробуйте написать программу для другой системы с проверочной цифрой, например для 13-значной системы ISBN. Программа должна принимать идентификационный номер и верифицировать его либо принимать номер без проверочной цифры и генерировать такую цифру.

2.6. Если вы изучали двоичные числа и знаете, как преобразовывать десятичные числа в двоичные и наоборот, попробуйте написать программу для выполнения таких преобразований. При этом числа должны быть неограниченной длины (но вы можете предположить, что числа должны быть достаточно небольшими, чтобы их можно было сохранить в стандартной переменной `int` языка C++).

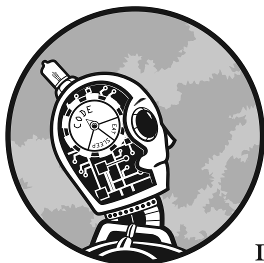
2.7. Вы изучали шестнадцатеричные числа? Попробуйте написать программу, которая позволяет пользователю ввести двоичное, десятичное или шестнадцатеричное число и вывести все три варианта.

2.8. Хотите дополнительный вызов? Обобщите код предыдущего упражнения для создания программы, конвертирующей любое число с базисом 16 или меньше в число с любым другим базисом. Так, например, программа должна смочь преобразовать число с базисом 9 в число с базисом 4.

2.9. Напишите программу, которая считывала бы строку текста, подсчитывая количество слов и вычисляя длину самого длинного слова, наибольшее количество гласных букв в слове и/или любую другую статистику, какую вы можете себе представить.

3

Решение задач с массивами



В предыдущей главе мы ограничились *скалярными переменными*, то есть переменными, которые могут одновременно хранить только одно значение. В этой главе мы рассмотрим задачи, в которых используется самый популярный агрегированный тип данных — массив. Несмотря на то, что массивы — это простые структуры с некоторыми фундаментальными ограничениями, их использование значительно увеличивает мощь наших программ.

В этой главе мы в первую очередь будем иметь дело с истинными массивами, то есть массивами, объявляемыми с помощью встроенного синтаксиса языка C++, например так:

```
int tenIntegerArray[10];
```

Однако обсуждаемые техники также хорошо применимы и к структурам данных с подобными атрибутами. Наиболее часто встре-

чающаяся разновидность таких структур данных — вектор. Термин *вектор* зачастую используется в качестве синонима одномерного массива, но мы будем использовать его более конкретно: для обозначения структуры с атрибутами массива без указания максимального количества элементов. Таким образом, в обсуждениях мы согласимся, что массив имеет фиксированный размер, тогда как размер вектора может увеличиваться или уменьшаться автоматически при необходимости. В каждой из обсуждаемых в этой главе задач будет некое ограничение, которое позволит использовать структуру с фиксированным количеством элементов. Задачи без таких ограничений могут быть адаптированы для использования векторов.

Более того, техники, используемые для массивов, могут зачастую также использоваться для структур данных, не обладающих каждым из вышеперечисленных свойств. Так, например, некоторые техники не требуют случайного доступа, то есть они могут использоваться применительно к таким структурам, как связанные списки. Массивы очень часто встречаются в программировании, и техники работы с ними используются и в других областях, поэтому это хороший тренировочный полигон для изучения решения задач со структурами данных.

Обзор основных свойств массивов

Вы уже должны знать, что такое массив, но давайте все же для большей ясности пробежимся по его основным свойствам. *Массив* —это набор переменных одного типа, собранных под одним именем, где отдельные переменные обозначаются числом. Мы называем отдельные переменные *элементами* массива. В C++ и большинстве других языков первый элемент имеет порядковый номер 0, но в некоторых языках ситуация может быть иной.

Основные свойства массива следуют сразу из его определения. Все значения, сохраненные в массиве, имеют один тип, тогда как агрегированные структуры данных могут хранить значения разных типов. Доступ к отдельному элементу осуществляется через порядковый номер, называемый *индексом*. В других разновидностях структур данных доступ к отдельному элементу может осуществляться по имени или ключевому значению.

Из этих основных свойств мы можем вывести несколько вторичных. Так как каждый элемент обозначается собственным порядковым номером, начиная с 0, мы можем с легкостью узнать все значения, сохраненные в массиве. В структурах данных других типов это может быть сложно, неэффективно или даже невозможно. Кроме того, тогда как к элементам некоторых структур данных, например связанных списков, доступ может быть получен только последовательно, массивы предлагают *случайный доступ*, таким образом, мы можем получить доступ к любому элементу в любое время.

Эти первичные и вторичные свойства определяют то, каким образом используются массивы. Работая с любой разновидностью агрегированной структурой данных, хорошо иметь в голове набор базовых операций при рассмотрении вариантов решения задачи. Воспринимайте эти базовые операции как инструменты: молотки, отвертки и гаечные ключи структур данных. Не всякая механическая проблема может быть решена с помощью общедоступных инструментов, но вам всегда следует смотреть, можно ли решить проблему с помощью этих инструментов прежде, чем отправляться искать новый. Далее приведен мой список базовых операций для массивов.

Сохранение

Это самая базовая операция. Массив — это набор переменных и мы можем присвоить значение каждой из них. Чтобы присвоить целое число 5 первому элементу (элементу 0) ранее объявленного массива, мы просто сообщаем:

```
tenIntegerArray[0] = 5;
```

Как и в случае с любой переменной, значения элементов внутри массива будут просто случайным «мусором» до тех пор, пока им не присвоят конкретное значение, таким образом, массивы следует инициализировать перед использованием. В некоторых случаях, особенно для тестирования, нам потребуется присвоить конкретное значение каждому элементу массива. Мы можем сделать это с помощью инициализатора сразу после объявления массива.

```
int tenIntegerArray[10] = {4, 5 , 9, 12, -4, 0, -57, 30987, -287, 1};
```

Вскоре вы увидите хороший пример использования инициализатора. Иногда вместо присваивания разных значений каждому элементу массива нам может потребоваться инициализировать все элементы массива одним значением. Существует несколько вариантов сокращений, позволяющих присвоить нули всем элементам массива в зависимости от ситуации или используемого компилятора (например, компилятор C++ в среде разработки Microsoft Visual Studio инициализирует все элементы массива нулем, если иное не указано пользователем). На данном этапе, однако, я бы всегда явно инициализировал массив вне зависимости от того, требуется ли такая инициализация, так как это повысит удобочитаемость кода, как в следующем примере кода, который устанавливает все элементы 10-элементного массива равными –1:

```
int tenIntegerArray[10];
for (int i = 0; i < 10; i++) tenIntegerArray[i] = -1;
```

Копирование

Мы можем сделать копию массива. Существует две часто встречающиеся ситуации, в которых это может оказаться полезным. Во-

первых, нам может потребоваться очень сильно изменить массив, но при этом иметь такой же массив в его первозданной форме для последующих операций. Восстановить исходный вид массива после проведения операций с ним может быть очень сложно или вовсе невозможно, если мы изменили хотя бы одно значение. Скопировав весь массив, мы можем работать с копией, не изменяя исходный массив. Все что нам нужно, чтобы скопировать весь массив, — это цикл и инструкция присваивания, точно так же, как и в случае с кодом инициализации:

```
int tenIntegerArray[10] = {4, 5 , 9, 12, -4, 0, -57, 30987, -287, 1};
int secondArray[10];
for (int i = 0; i < 10; i++) secondArray[i] = tenIntegerArray[i];
```

Эта операция доступна для большинства агрегированных структур данных. Вторая ситуация более специфична для массивов. Иногда нам может потребоваться скопировать часть данных из одного массива во второй, либо мы можем захотеть скопировать элементы одного массива в другой в рамках метода реорганизации порядка элементов. Если вы изучали алгоритм сортировки слиянием, то вы видели эту ситуацию в действии. Далее по тексту главы мы увидим несколько примеров копирования.

Извлечение и поиск

Имея возможность поместить значения в массив, мы также должны быть способны получать данные из массива. Извлечение значения из конкретной позиции выполняется прямолинейно:

```
int num = tenIntegerArray[0];
```

Поиск определенного значения

Обычно ситуация не так проста. Часто нам неизвестна нужная позиция, и требуется *произвести поиск* в массиве, чтобы определить позицию конкретного значения. Если элементы массива не упорядочены особым образом, то самое лучшее, что мы можем сделать, — это произвести последовательный поиск, в процессе которого мы будем перебирать все элементы массива, пока не найдем нужное значение. Ниже представлена базовая версия кода.

```
❶ const int ARRAY_SIZE = 10;
❷ int intArray[ARRAY_SIZE] = {4, 5, 9, 12, -4, 0, -57, 30987, -287, 1};
❸ int targetValue = 12;
❹ int targetPos = 0;
  while ((intArray[targetPos] != targetValue) && (targetPos < ❺ ARRAY_SIZE))
     targetPos++;
```

В этом коде присутствует константа, в которой хранится размер массива ❶, сам массив ❷, переменная для хранения значения, которое мы ищем в массиве ❸, а также переменная для хранения позиции этого значения ❹. В данном примере мы используем константу `ARRAY_SIZE` для ограничения количества итераций по нашему мас-

сиву ❺, чтобы не выйти за его пределы, если среди его элементов не обнаружится целевое значение targetValue. Вы могли бы «жестко закодировать» значение 10 вместо константы, однако использование константы делает код более общим, что облегчает его модификацию и повторное использование. Мы будем использовать константу ARRAY_SIZE в большей части фрагментов кода, приведенных в данной главе. Обратите внимание на то, что, если значение targetValue не будет обнаружено в массиве intArray, то после завершения цикла значение targetPos будет равно значению ARRAY_SIZE. Этого достаточно для обозначения данного события, поскольку значение ARRAY_SIZE не является допустимым номером элемента. Тем не менее проверить это предстоит коду, следующему далее. Также заметьте, что этот код не предусматривает никаких действий на случай, если целевое значение встретится более одного раза. При первом появлении целевого значения цикл завершится.

Поиск по критерию

Иногда искомое значение является не фиксированным, а основанным на отношении с другими значениями массива. Например, нам требуется найти максимальное значение в массиве. Механизм для решения этой задачи я называю «Царь горы» по аналогии с известной игрой. Пусть у вас будет переменная, представляющая наибольшее значение из обнаруженных в массиве *до сих пор*. Вы перебираете все элементы массива с помощью цикла, и каждый раз, когда сталкиваетесь со значением, превышающим предыдущее наибольшее значение, новое значение сбрасывает предыдущего царя с горы, занимая его место:

```
  const int ARRAY_SIZE = 10;
  int intArray[ARRAY_SIZE] = {4, 5, 9, 12, -4, 0, -57, 30987, -287, 1};
❶ int highestValue = intArray[0];
❷ for (int i = 1; i < ARRAY_SIZE; i++) {
      ❸ if (intArray[i] ❹ > highestValue) highestValue = intArray[i];
  }
```

В переменной highestValue хранится наибольшее значение, найденное в массиве до сих пор. При ее объявлении ей присваивается значение первого элемента массива ❶, что позволяет запустить цикл на втором элементе массива (это позволяет нам начать со значения i равного 1, а не 0) ❷. Внутри цикла мы сравниваем значение в текущей позиции со значением highestValue, при необходимости заменяя значение highestValue ❸. Обратите внимание на то, что для нахождения наименьшего значения вместо наибольшего достаточно просто заменить оператор сравнения «больше чем» оператором сравнения «меньше чем» (и изменить название переменной, чтобы не запутаться). Эта базовая структура может применяться ко всем видам ситуаций, в которых нам требуется рассмотреть каждый элемент массива, чтобы найти значение, в наибольшей степени иллюстрирующее конкретное качество.

Сортировка

Сортировка означает расположение данных в определенном порядке. Вам, вероятно, уже встречались алгоритмы сортировки массивов. Это классическая область для применения анализа эффективности, поскольку существует так много конкурирующих алгоритмов сортировки, каждый из которых имеет различные характеристики производительности, зависящие от особенностей базовых данных. Изучение различных алгоритмов сортировки могло бы стать предметом отдельной книги, поэтому мы не будем глубоко исследовать данную область. Вместо этого сосредоточимся на том, что можно применить на практике. В большинстве ситуаций вы сможете обойтись двумя алгоритмами сортировки — быстрым, простым в использовании, и хорошим, легким для понимания, которые вы сможете, при необходимости, с уверенностью модифицировать. Для быстрой и простой сортировки мы будем использовать стандартную библиотечную функцию `qsort`, а когда понадобится что-то настроить — сортировку методом вставок.

Быстрая и простая сортировка с помощью функции `qsort`

Для программистов C/C++ алгоритмом быстрой сортировки по умолчанию является функция стандартной библиотеки `qsort` (название предполагает применение быстрой сортировки (quicksort), однако реализатор библиотеки не обязан использовать этот алгоритм). Для использования функции `qsort` мы должны написать функцию сравнения. Ее будет вызывать функция `qsort`, когда понадобится сравнить два элемента в массиве, чтобы решить, какой из них должен отображаться раньше в отсортированном порядке. В эту функцию передается два указателя типа `void`. В данной книге мы еще не обсуждали указатели, однако пока вам достаточно знать то, что вы должны привести эти указатели `void` к указателям на тип элемента в вашем массиве. Затем функция должна возвратить значение `int`, которое может быть положительным, отрицательным или равным нулю, в зависимости от того, является ли значение первого элемента большим, меньшим или равным значению второго элемента. Конкретное возвращаемое значение не важно, главное то, положительное оно, отрицательное или равное нулю. Давайте проясним это с помощью небольшого примера сортировки массива из 10 целых чисел с использованием функции `qsort`. Наша функция сравнения выглядит следующим образом:

```
int compareFunc(❶const void * voidA, const void * voidB) {
  ❷ int * intA = (int *)(voidA);
    int * intB = (int *)(voidB);
  ❸ return *intA — *intB;
}
```

Список параметров состоит из двух указателей `const void` ❶. Так бывает всегда в случае с компаратором. Код внутри функции начинается с объявления двух указателей `int` ❷ и приведения двух

указателей void к указателям на тип int. Мы могли бы написать функцию без двух временных переменных; я использовал их здесь для ясности. Дело в том, что, как только мы закончим с этими объявлениями, intA и intB будут указывать на два элемента в нашем массиве, а *intA и *intB будут двумя целыми числами, подлежащими сравнению. Наконец мы возвращаем результат вычитания второго целого значения из первого ❸. Это позволяет получить нужный результат. Например, если *intA > *intB, нам нужно возвратить положительное число, а результат вычисления выражения *intA – *intB будет положительным, если *intA > *intB. Точно так же результат вычисления выражения *intA – *intB будет отрицательным, если *intB > *intA, и будет равен нулю в случае их равенства.

После добавления функции компаратора пример использования функции qsort выглядит следующим образом:

```
const int ARRAY_SIZE = 10;
int intArray[ARRAY_SIZE] = {87, 28, 100, 78, 84, 98, 75, 70, 81, 68};
qsort(❶intArray, ❷ARRAY_SIZE, ❸sizeof(int), ❹compareFunc);
```

Как вы можете заметить, функция qsort принимает четыре параметра: массив, который должен быть отсортирован ❶; количество элементов в этом массиве ❷; размер одного элемента в массиве, который, как правило, определяется, как и в данном случае, оператором sizeof ❸; наконец, функция компаратора ❹. Если у вас нет большого опыта передачи функций в качестве параметров в другие функции, обратите внимание на синтаксис, используемый для последнего параметра. Мы передаем непосредственно саму функцию, а не вызываем ее с последующей передачей результата этого вызова. Поэтому мы просто указываем имя функции без списка параметров или круглых скобок.

Легко модифицируемый алгоритм сортировки — сортировка вставками

В некоторых случаях вам может потребоваться написать собственный код сортировки. Иногда встроенная функция сортировки просто не подходит для решения стоящей перед вами задачи. Например, вам нужно отсортировать массив данных, основываясь на данных в *другом* массиве. При необходимости написания собственного кода вам нужен простой алгоритм сортировки, в котором вы будете уверены и который при необходимости можете без труда реализовать. Таким алгоритмом может стать *сортировка методом вставок*. Сортировка вставками напоминает сортировку карт при игре в бридж: игроки берут карты по одной за раз и размещают их в руке, поддерживая общий порядок и сдвигая другие карты, чтобы освободить для них место. Ниже представлена базовая реализация нашего целочисленного массива:

```
❶int start = 0;
❷int end = ARRAY_SIZE – 1;
❸for (int i = start + 1; i <= end; i++) {
```

```
    for (❹ int j = i; ❺ j > start && ❻ intArray[j-1] > intArray[j]; j--) {
        ❼ int temp = intArray[j-1];
        intArray[j-1] = intArray[j];
        intArray[j] = temp;
    }
}
```

Мы начинаем с объявления двух переменных, `start` ❶ и `end` ❷, обозначающих индекс первого и последнего элементов массива. Так код становится более удобным для чтения, а также при необходимости может быть легко модифицирован для сортировки только части массива. Внешний цикл выбирает следующую «карту», которая должна быть вставлена в наш все возрастающий отсортированный набор ❸. Обратите внимание на то, что при инициализации этого цикла счетчику `i` присваивается значение `start + 1`. Помните, в коде для нахождения наибольшего значения мы инициализировали переменную с наибольшим значением первым элементом массива и начали цикл со второго элемента массива. В этом случае идея та же. Если у нас есть только одно значение (или «карта»), то по определению она «упорядочена», и мы можем начать с рассмотрения вопроса о том, должно ли второе значение располагаться до или после первого. Внутренний цикл помещает текущее значение в правильную позицию, многократно меняя местами текущее значение с предшествующим ему, пока оно не окажется в нужной позиции. Начальным значением счетчика цикла `j` является `i` ❹, и цикл уменьшает это значение на 1, пока не будет достигнут нижний предел массива ❺ и не найдется правильная позиция для этого нового значения ❻. До тех пор мы будем использовать три инструкции присваивания для смещения текущего значения на одну позицию в массиве ❼. Другими словами, если бы у вас был набор из 13 игральных карт, из которых четыре крайние левые были уже отсортированы, то вы могли бы поместить пятую карту слева в правильную позицию, последовательно перемещая ее на одну позицию до тех пор, пока ее значение не превысит значение карты, расположенной слева от нее. Именно это делает внутренний цикл. Внешний цикл делает это для каждой карты, начиная с крайней левой. Таким образом, к моменту завершения цикла весь массив будет отсортирован.

В большинстве случаев сортировка вставками — не самый эффективный алгоритм, и, честно говоря, приведенный выше код — даже не самый эффективный способ выполнения такого рода сортировки. Тем не менее он подходит для небольших массивов и достаточно прост, чтобы его можно было запомнить, поэтому считайте его ментальным макросом. Вне зависимости от того, какой алгоритм сортировки вы выберете, у вас должен быть приличный метод код для которого вы можете легко создать самостоятельно. Недостаточно иметь доступ к чужому коду для осуществления сортировки, который вы не вполне понимаете. Вам не следует возиться с машиной, если вы не знаете, как она работает.

Вычисление статистических показателей

Последняя операция похожа на поиск в том плане, что перед возвратом значения нам нужно просмотреть каждый элемент массива. Отличие заключается в том, что это значение является не просто одним из элементов массива, а некоторым статистическим показателем, вычисленным на основании всех значений в массиве. Например, мы могли бы вычислить среднее арифметическое значение, медиану или статистическую моду, и сделаем это далее в этой главе. Примером вычисления простейшего статистического показателя могло бы быть среднее арифметическое значение оценок учащихся:

```
const int ARRAY_SIZE = 10;
int gradeArray[ARRAY_SIZE] = {87, 76, 100, 97, 64, 83, 88, 92, 74, 95};
double sum = 0;
for (int i = 0; i < ARRAY_SIZE; i++) {
     sum += gradeArray[i];
}
double average = sum / ARRAY_SIZE;
```

В качестве еще одного простого примера рассмотрим проверку данных. Пусть массив значений типа `double` с именем `vendorPayments` содержит данные о платежах поставщикам. Действительны только положительные значения, поэтому отрицательные указывают на проблемы целостности данных. В части отчета о проверке мы можем написать цикл для подсчета количества отрицательных значений в массиве:

```
const int ARRAY_SIZE = 10;
int countNegative = 0;
for (int i = 0; i < ARRAY_SIZE; i++) {
     if (vendorPayments[i] < 0) countNegative++;
}
```

Решение задач с помощью массивов

После того как вы разберетесь с общими операциями, решение задач с помощью массивов не будет сильно отличаться от их решения с использованием простых данных, которые разбирались в предыдущей главе. Рассмотрим один пример и проработаем все его этапы, используя методы из предыдущей главы и любую из часто применяемых операций для массивов, которые нам могут понадобиться.

Задача: нахождение моды

В статистике мода набора значений представляет собой самое часто встречающееся значение. Напишите код, который обрабатывает массив данных опроса, в ходе которого респонденты отвечали на вопрос с помощью цифры от 1 до 10, чтобы определить моду набора данных. Для нашей цели можно выбрать любую моду, если их окажется несколько.

В этой задаче нам требуется извлечь из массива одно значение. Используя методы поиска аналогии и начав с уже известного, мы можем надеяться на то, что сможем применить вариант поисковой техники, с которой познакомились, когда искали наибольшее значение в массиве. Принцип работы этого кода заключается в сохранении наибольшего из наблюдаемых до сих пор значений в переменной. Затем код сравнивает каждое последующее значение с этой переменной, заменяя ее при необходимости. В данном случае аналогичный метод заключается в том, что мы будем сохранять наиболее часто встречающееся до сих пор значение в переменной, а затем изменять ее при обнаружении более распространенного значения в массиве. Когда мы проговариваем это словами, кажется, что это может сработать, но когда думаем о реальном коде, то обнаруживаем проблему. Давайте рассмотрим образец массива и константу размера для этой задачи:

```
const int ARRAY_SIZE = 12;
int surveyData[ARRAY_SIZE] = {4, 7, 3, 8, 9, 7, 3, 9, 9, 3, 3, 10};
```

Модой среди этих данных является значение 3, потому что оно встречается четыре раза, то есть чаще, чем любое другое. Однако если мы обрабатываем этот массив последовательно, как в задаче с нахождением «наибольшего значения», в какой момент мы решаем, что значение 3 является нашей модой? Как мы узнаем, что оно встретилось в массиве в четвертый и последний раз? Кажется, что у нас нет никакой возможности выяснить это с помощью одной последовательной обработки данных массива.

Итак, обратимся к одному из других наших методов: упрощению задачи. Что, если бы мы упростили себе жизнь, собрав вместе все случаи появления одного и того же значения? Например, что, если бы наш массив с данными результатов опроса выглядел так:

```
int surveyData[ARRAY_SIZE] = {4, 7, 7, 9, 9, 9, 8, 3, 3, 3, 3, 10};
```

Теперь оба значения 7 собраны вместе, как и значения 9 и 3. После такой группировки данных кажется возможным последовательно обработать массив с целью нахождения моды. Обрабатывая массив вручную, легко подсчитать количество каждого из значений, поскольку вы просто отсчитываете элементы массива, пока вам не встретится другое значение. Тем не менее преобразовать наши мыслительные операции в программные инструкции может быть сложно. Поэтому, прежде чем пытаться написать код для этой упрощенной задачи, давайте напишем некоторый *псевдокод*, представляющий программные инструкции, написанные на языке, который представляет собой нечто среднее между человеческим языком и C++. Этот код будет напоминать о том, что мы пытаемся сделать с каждым утверждением, которое нужно написать.

```
   int mostFrequent = ❶?;
   int highestFrequency = ❶?;
   int currentFrequency = 0;
❷ for (int i = 0; i < ARRAY_SIZE; i++) {
     ❸ currentFrequency++;
     ❹ if (surveyData[i] IS LAST OCCURRENCE OF A VALUE) {
        ❺ if (currentFrequency > highestFrequency) {
              highestFrequency = currentFrequency;
              mostFrequent = surveyData[i];
           }
        ❻ currentFrequency = 0;
     }
}
```

Не существует правильного или неправильного способа написания псевдокода, и если вы собираетесь использовать данную технику, то вам следует выработать свой собственный стиль. Когда я пишу псевдокод, я стараюсь использовать действительный C++ для любой инструкции, в которой я уже уверен, а затем на человеческом языке пишу то, что я все еще обдумываю. В данном случае мы знаем, что нам понадобится переменная (`mostFrequent`) для хранения наиболее часто встречающегося до сих пор значения, которое после окончания цикла будет являться модой, если все будет сделано правильно. Нам также нужна переменная для хранения данных о том, насколько часто встречается это значение (`highestFrequency`), с которой мы можем производить сравнение. Наконец, нам нужна переменная, которую мы можем использовать для подсчета частоты отслеживаемого значения в процессе последовательной обработки массива (`currentFrequency`). Мы знаем, что нужно инициализировать наши переменные. Значение `currentFrequency` логически должно начинаться с 0, однако в отсутствие другого кода пока неясно, как следует инициализировать остальные переменные. Итак, давайте просто расставим вопросительные знаки ❶ в качестве напоминания.

Сам цикл представляет собой уже знакомый нам цикл для обработки массива, поэтому он уже представлен в окончательной форме ❷. Внутри цикла мы инкрементируем значение переменной, которая подсчитывает количество раз, когда встречается текущее значение ❸, а затем следует ключевая инструкция. Мы знаем, что нужно проверить, достигли ли мы последнего случая появления определенного значения ❹. Данный псевдокод позволяет пока пропустить этап выявления логики и набросать остальную часть кода. Однако если это *действительно* последний раз, когда встречается данное значение, мы знаем, что нужно делать, поскольку это похоже на код для нахождения «наибольшего значения»: нужно выяснить, превышает ли частота этого значения частоту значения, которое до сих пор было самым часто встречающимся. Если это так, то данное значение становится новым наиболее часто встречающимся значением ❺. Тогда, поскольку следующее прочитанное значение будет первым случаем появления нового значения, мы сбрасываем значение счетчика ❻.

Вернемся к логике инструкции if, которую мы пропустили. Как мы узнаем, что это последний случай появления значения в массиве? Поскольку значения в массиве сгруппированы, мы знаем, что значение встречается в последний раз, когда следующее значение в массиве отличается от него: в терминах C++, когда значения переменных surveyData[i] и surveyData[i + 1] не равны. Кроме того, последнее значение в массиве также представляет собой последний случай появления некоторого значения, даже если следующего значения нет. Мы можем узнать это, проверив, равно ли значение i == ARRAY_SIZE − 1, в случае чего это значение в массиве является последним.

Выяснив все это, давайте подумаем о первоначальных значениях для наших переменных. Помните, что в случае с кодом для обработки массива с целью нахождения «наибольшего значения» мы инициализировали переменную «с наибольшим значением, обнаруженным до сих пор» первым значением в массиве. Здесь «наиболее часто встречающееся» значение представлено двумя переменными – mostFrequent для самого значения и highestFrequency для количества случаев появления. Было бы хорошо, если бы была возможность инициализировать переменную mostFrequent первым значением в массиве, а переменную highestFrequency – значением счетчика частоты, однако не существует способа определить частоту первого значения, пока мы не доберемся до цикла и не начнем подсчет. В этот момент может показаться, что частота первого значения, каким бы она ни была, будет больше нуля. Поэтому, если мы зададим для переменной highestFrequency значение, равное нулю, то по достижению последнего случая появления первого значения, код все равно заменит значения переменных mostFrequent – highestFrequency показателями, соответствующими первому значению. Финальная версия кода выглядит следующим образом:

```
int mostFrequent;
int highestFrequency = 0;
int currentFrequency = 0;
for (int i = 0; i < ARRAY_SIZE; i++) {
    currentFrequency++;
  ❶ // if (surveyData[i] IS LAST OCCURENCE OF A VALUE)
  ❷ if (i == ARRAY_SIZE − 1 || surveyData[i] != surveyData[i + 1]) {
        if (currentFrequency > highestFrequency) {
            highestFrequency = currentFrequency;
            mostFrequent = surveyData[i];
        }
        currentFrequency = 0;
    }
}
```

В этой книге мы не будем много говорить о таких чисто стилевых проблемах, как стиль документирования (комментирования), однако, поскольку мы используем псевдокод в данной задаче, я хочу дать вам совет. Я заметил, что строки, написанные мной на «простом английском языке» в псевдокоде, являются теми строками, которые больше всего выигрывают от комментариев в итоговом коде, и сами

фразы на простом английским представляют собой замечательные комментарии. Я продемонстрировал это в приведенном здесь коде. Вы можете забыть точный смысл условных выражений в инструкции `if` ❷, однако комментарий на предыдущей строке прекрасно все проясняет ❶.

Что касается самого кода, он справляется со своей задачей, однако помните, что это требует группировки наших данных опроса. Группировка этих данных сама по себе может представлять проблему за исключением случая, когда мы *сортируем* массив. На самом деле мы не нуждаемся в сортировке, однако сортировка осуществит необходимую группировку. Поскольку мы не собираемся производить какой-то особый вид сортировки, давайте просто добавим вызов функции `qsort` в начало нашего кода:

```
qsort(surveyData, ARRAY_SIZE, sizeof(int), compareFunc);
```

Обратите внимание, что мы используем тот же метод `compareFunc`, который писали ранее для использования с функцией `qsort`. Добавив этап сортировки, мы получили полное решение исходной задачи. Итак, работа выполнена. Не так ли?

Рефакторинг

Некоторые программисты говорят о коде, который «дурно пахнет». Они говорят о работающем коде, в котором нет ошибок, но есть какие-то проблемы. Иногда это означает, что код слишком сложен или предусматривает слишком много специфических случаев, что затрудняет его модификацию и обслуживание. Бывает, что код не так эффективен, как мог бы быть, и хотя он подходит для решения тестовых задач, программист опасается, что производительность снизится, когда дело дойдет до более масштабных задач. Именно это заботит меня в данном случае. В нашем примере с крошечным массивом сортировка осуществляет почти мгновенно, но что, если массив будет огромным? Кроме того, я знаю, что алгоритм быстрой сортировки, который может использовать функция `qsort`, имеет самую низкую производительность, когда в массиве присутствует много повторяющихся значений, и весь смысл этой задачи состоит в том, что все наши значения находятся в диапазоне от 1 до 10. Поэтому я предлагаю произвести *рефакторинг* кода. *Рефакторинг* означает улучшение работающего кода, изменение не того, *что* он делает, а того, *как* он это делает. Мне нужно решение, которое было бы эффективным даже для огромных массивов, допуская, что их значения находятся в диапазоне от 1 до 10.

Давайте еще раз подумаем об операциях, которые мы умеем производить с помощью массивов. Мы уже изучили несколько версий кода для нахождения наибольшего значения. Мы знаем, что применение этого кода непосредственно к нашему массиву `surveyData` не даст нужных результатов. Есть ли массив, к которому мы могли

бы применить имеющийся у нас вариант кода для нахождения наибольшего значения, чтобы определить моду среди значений данных опроса? Ответ — да. Нужный нам массив — это гистограмма массива surveyData. Гистограмма представляет собой график, показывающий, как часто различные значения появляются в наборе данных, на котором он основан. Наш массив будет представлять собой набор данных для такой гистограммы. Другими словами, мы будем хранить в массиве из 10 элементов данные о том, как часто каждое из значений от 1 до 10 встречается в массиве surveyData. Ниже показан код для создания гистограммы:

```
  const int MAX_RESPONSE = 10;
❶ int histogram[MAX_RESPONSE];
❷ for (int i = 0; i < MAX_RESPONSE; i++) {
      histogram[i] = 0;
  }
❸ for (int i = 0; i < ARRAY_SIZE; i++) {
      ❹ histogram[surveyData[i] - 1]++;
  }
```

В первой строке мы объявляем массив для хранения данных гистограммы ❶. Вы заметите, что мы объявляем массив с 10 элементами, но значения данных опроса находятся в диапазоне от 1 до 10, а индексы для элементов этого массива — в диапазоне от 0 до 9. Таким образом, мы должны будем внести коррективы, поместив количество значений 1 в histogram[0] и так далее. (Некоторые программисты могут объявить массив с 11 элементами, оставив позицию[0] неиспользованной, чтобы индекс каждого элемента фактически соответствовал его позиции.) Мы явно инициализируем значения массива нулем с помощью цикла ❷, после чего готовы подсчитать количество каждого значения в массиве surveyData с помощью другого цикла ❸. Инструкцию внутри цикла ❹ следует прочитать очень тщательно; значение в текущей позиции массива surveyData говорит о том, какую позицию в массиве histogram необходимо инкрементировать. Чтобы было ясно, давайте рассмотрим пример. Предположим, что i присвоено значение 42. Мы обращаемся к surveyData[42] и обнаруживаем, допустим, значение 7. Таким образом, нам нужно инкрементировать значение счетчика 7. Мы вычитаем 1 из 7, чтобы получить 6, поскольку счетчик значений 7 находится в позиции [6] в массиве histogram, итак, мы увеличиваем на 1 значение histogram[6].

Теперь, когда данные гистограммы на месте, мы можем записать остальную часть кода. Обратите внимание, что код гистограммы создавался отдельно, чтобы его можно было отдельно протестировать. Невозможно сэкономить время, записывая весь код сразу в ситуации, когда задача легко разделяется на части, которые могут быть написаны и протестированы отдельно. Проверив приведенный выше код, мы теперь ищем наибольшее значение в массиве histogram:

```
❶ int mostFrequent = 0;
  for (int i = 1; i < MAX_RESPONSE; i++) {
       if (histogram[i] > ❷ histogram[mostFrequent]) ❸ mostFrequent = i;
  }
❹ mostFrequent++;
```

Несмотря на то, что этот код представляет собой адаптацию кода для нахождения наибольшего значения, он имеет некоторые отличия. Хотя мы ищем наибольшее значение в массиве `histogram`, в конечном итоге, нам нужно не само значение, а его позиция. Другими словами, в случае с массивом в примере мы хотим знать, что значение 3 встречается чаще, чем любое другое среди данных опроса, однако фактическое количество раз, когда оно встречается, для нас не важно. Таким образом, `mostFrequent` будет обозначать позицию наибольшего значения в массиве `histogram`, а не само это значение. Поэтому мы инициализируем его `0` ❶, а не значением в позиции `[0]`. Это также означает, что в инструкции `if` мы производим сравнение с `histogram[mostFrequent]` ❷, а не с `mostFrequent`, и присваиваем значение `i`, а не `histogram[i]` переменной `mostFrequent` ❸ при нахождении большего значения. Наконец, мы инкрементируем переменную `mostFrequent` ❹. Эта операция противоположна тому, что мы делали в предыдущем цикле, вычитая 1, чтобы получить правильную позицию в массиве. Например, если значение `mostFrequent` сообщает нам, что наивысшей позицией массива является 5, это означает, что наиболее часто встречающейся записью среди данных опроса является 6.

Решение с использованием гистограммы масштабируется линейно с увеличением числа элементов в нашем массиве `surveyData`, что соответствует нашим самым смелым ожиданиям. Следовательно, это решение лучше, чем наш исходный подход. Это не означает, что первый подход был ошибкой или пустой тратой времени. Конечно, можно было бы написать этот код, не прибегая к предыдущей версии, и нас можно простить за желание направиться к месту назначения напрямую, а не в обход. Однако я бы предостерег от битья себя по лбу в тех случаях, когда первое решение оказывается не окончательным. Написание оригинальной программы (помните, что это означает *оригинальной для пишущего ее программиста*) — это процесс обучения, который не может всегда продвигаться вперед прямолинейно. Кроме того, часто бывает так, что более длинный путь при решении одной задачи помогает найти более короткий путь для решения задач в будущем. В данном конкретном случае обратите внимание на то, что наше исходное решение (хотя оно не масштабируется достаточно хорошо для нашей конкретной задачи) могло бы быть правильным, если бы данные опроса не были строго ограничены небольшим диапазоном 1 — 10. Или предположим, что в дальнейшем вам потребуется написать код для нахождения *медианы* в наборе целочисленных значений (медиана — это значение, находящиеся в середине так, что половина значений набора больше него, а другая

половина меньше). Подход с использованием гистограммы никуда вас не приведет в случае с медианой, чего нельзя сказать о нашем первом подходе для нахождения моды.

Урок в данном случае заключается в том, что долгий путь — это не пустая трата времени, если вы научились чему-то такому, чему не научились бы, пойдя коротким путем. Это еще одна причина методично хранить весь код, который вы пишете, чтобы вы могли позднее легко найти его и повторно использовать. Даже код, который оказывается «тупиком», может стать ценным ресурсом.

Массивы фиксированных данных

В большинстве задач массив представляет собой хранилище данных, внешних по отношению к программе, например введенных пользователем, хранящихся на локальном диске или полученных с сервера. Однако чтобы получить максимальную отдачу от такого инструмента, как массив, вам нужно уметь распознавать другие ситуации, в которых его можно использовать. Часто бывает полезно создать массив, значения которого остаются неизменными после инициализации. Такой массив может допускать использование простого цикла или даже осуществление прямого поиска в массиве с целью замены всего блока управляющих инструкций.

В окончательном коде для решения задачи декодирования сообщения в конце предыдущей главы мы использовали инструкцию `switch` для замены декодированного входного числа (в диапазоне от 1 до 8) соответствующим знаком препинания, поскольку связь между числом и символом была произвольной. Несмотря на то, что это сработало, данная часть кода оказалась длиннее, чем эквивалентный код для режимов верхнего и нижнего регистра, кроме того, этот код не будет хорошо масштабироваться при увеличении количества знаков препинания. Для решения этой задачи вместо инструкции `switch` мы можем использовать массив. Во-первых, мы должны на постоянной основе назначить массиву знаки препинания в том же порядке, в каком они появляются в схеме кодирования:

```
const char punctuation[8] = {'!', '?', ',', '.', ' ', ';', '"', '\''};
```

Обратите внимание на то, что этот массив объявлен как константа, поскольку значения в нем никогда не изменятся. Благодаря этому объявлению мы можем заменить весь код инструкции `switch` одной инструкцией присваивания, которая ссылается на массив:

```
outputCharacter = punctuation[number – 1];
```

Поскольку входное число находится в диапазоне от 1 до 8, а нумерация элементов массива начинается с 0, то нам необходимо вычесть 1 из входного числа, прежде чем сослаться на массив. Эта корректировка аналогична той, что мы делали в версии программы

для нахождения моды с использованием гистограммы. Вы можете использовать тот же массив, чтобы пойти в другом направлении. Предположим, что вместо декодирования сообщения нам нужно его закодировать, т.е. дан ряд символов, которые требуется преобразовывать в числа, которые могут быть декодированы с использованием правил исходной задачи. Чтобы преобразовать знак препинания в соответствующее ему число, мы должны найти этот символ в массиве. Это операция извлечения, выполняемая с использованием приема последовательного поиска. Предположив, что символ должен быть преобразован и сохранен в переменной targetValue массива char, мы могли бы адаптировать код для осуществления последовательного поиска следующим образом:

```
const int ARRAY_SIZE = 8;
int targetPos = 0;
while (punctuation[targetPos] != targetValue && targetPos < ARRAY_SIZE)
   targetPos++;
int punctuationCode = targetPos + 1;
```

Обратите внимание на то, что так же, как нам пришлось вычитать 1 из значения number в предыдущем примере, чтобы получить правильную позицию в массиве, в данном примере необходимо добавить 1 к позиции массива, чтобы получить код из знаков препинания, преобразуя значение массива в диапазоне от 0 до 7 в значения нашего кода из знаков препинания в диапазоне от 1 до 8. Хотя этот код состоит не из одной строки, он все-таки намного проще, чем серия операторов switch, и хорошо масштабируется. Если бы мы удвоили количество знаков препинания в нашей схеме кодирования, это привело бы к удвоению количества элементов массива, однако длина кода осталась бы прежней.

В общем случае массивы const могут использоваться в качестве таблиц поиска вместо громоздкой серии управляющих инструкций. Предположим, вы пишете программу для вычисления стоимости бизнес-лицензии, которая зависит от значения валового объема продаж.

Табл. 3.1. Стоимость бизнес-лицензии

Категория бизнеса	Граница объема продаж	Стоимость лицензии
I	$ 0	$ 25
II	$ 50000	$ 200
III	$ 150000	$ 1000
IV	$ 500000	$ 5000

В этой задаче мы могли бы использовать массивы как для определения категории бизнеса на основе валовых продаж компании, так и для присвоения стоимости лицензии на основе категории бизнеса. Предположим, что в переменной типа double с именем grossSales хранится значение валового объема продаж компании и, исходя из этого показателя объема продаж, мы хотим присвоить подходящие значения переменным int category и double cost:

```
   const int NUM_CATEGORIES = 4;
❶ const double categoryThresholds[NUM_CATEGORIES ] =
      {0.0, 50000.0, 150000.0, 500000.0};
❷ const double licenseCost[NUM_CATEGORIES ] =
      {50.0, 200.0, 1000.0, 5000.0};
❸ category = 0;
❹ while (category < NUM_CATEGORIES &&
      categoryThresholds[category] <= grossSales) {
         category++;
      }
❺ cost = licenseCost[category - 1];
```

В этом коде используется два массива фиксированных значений. В первом массиве хранится значение границы объема продаж для каждой категории бизнеса ❶. Например, компания с годовым валовым объемом продаж в 65 000 долларов США относится к категории II, поскольку эта сумма превышает границу в 50 000 долларов США категории II, но не достигает границы в 150 000 долларов США категории III. Во втором массиве хранится значение стоимости бизнес-лицензии для каждой категории ❷. Определившись с массивами, мы инициализируем переменную `category` нулем ❸ и выполняем поиск в массиве `categoryThresholds`, пока не превысим границу валового объема продаж или пока не закончатся категории ❹. В любом случае, когда цикл завершится, категории 1–4 будут правильно присвоены, исходя из валового объема продаж. Последним шагом станет использование переменной `category` для того, чтобы сослаться на стоимость лицензии из массива `licenseCost` ❺. Как и раньше, мы должны внести небольшую корректировку, преобразовав значения категорий бизнеса в диапазоне 1–4 в диапазон 0–3 нашего массива.

Нескалярные массивы

До сих пор мы работали только с массивами таких простых типов данных, как `int` и `double`. Однако часто программисты вынуждены иметь дело с массивами сложных данных — структур или объектов (`struct` или `class`). Хотя использование сложных типов данных обязательно усложняет код, оно не обязано делать то же с ходом наших размышлений об обработке массивов. Обычно обработка массива включает только один элемент данных `struct` или `class`, и мы можем проигнорировать другие части структуры данных. Тем не менее иногда использование сложных типов данных требует от нас несколько изменить свой подход.

Например, рассмотрим задачу нахождения наибольшего значения среди оценок студентов. Предположим, что вместо массива `int` у нас есть массив структур, каждая из которых представляет собой набор данных о студенте:

```
struct student {
   int grade;
   int studentID;
   string name;
};
```

Одно из удобств при работе с массивами заключается в том, что целый массив можно легко инициализировать буквенными значениями для осуществления простого тестирования, даже в случае с массивом struct:

```
const int ARRAY_SIZE = 10;
student studentArray[ARRAY_SIZE] = {
   {87, 10001, "Fred"},
   {28, 10002, "Tom"},
   {100, 10003, "Alistair"},
   {78, 10004, "Sasha"},
   {84, 10005, "Erin"},
   {98, 10006, "Belinda"},
   {75, 10007, "Leslie"},
   {70, 10008, "Candy"},
   {81, 10009, "Aretha"},
   {68, 10010, "Veronica"}
};
```

Данное объявление означает, что в массиве studentArray[0] хранится значение 87 для переменной grade, 10001 — для переменной studentID, «Fred» — для переменной name и так далее для остальных девяти элементов массива. Что касается остальной части кода, то его написание могло бы сводиться к элементарному копированию кода из начала этой главы и дальнейшей замене каждой ссылки на форму intArray[subscript] на studentArray[subscript].grade. Это привело бы к следующему результату:

```
int highest = studentArray[0].grade;
for (int i = 1; i < ARRAY_SIZE; i++) {
     if (studentArray[i].grade > highest) highest = studentArray[i].grade;
}
```

Вместо этого предположим, что, поскольку у нас теперь есть дополнительная информация по каждому студенту, мы хотим найти имя студента с самой высокой оценкой, а не саму оценку. Это потребует дополнительной модификации. После завершения цикла мы имеем единственный статистический показатель — лучшую оценку, и это не позволяет напрямую определить ученика, которому она принадлежит. Мы должны снова осуществить поиск в массиве, чтобы найти структуру struct с соответствующим значением grade, что кажется дополнительной работой, которую мы не должны делать. Чтобы избежать этой проблемы, следует либо дополнительно отслеживать имя студента, соответствующее текущему значению в переменной highest, либо вместо отслеживания наивысшей оценки отслеживать позицию в массиве, в которой была найдена наивысшая оценка, как мы это делали ранее с histogram. Последний подход является наиболее общим, поскольку отслеживание позиции массива позволяет позднее извлечь *любой* элемент данных для конкретного студента:

```
❶ int highPosition = 0;
  for (int i = 1; i < ARRAY_SIZE; i++) {
```

```
        if (studentArray[i].grade > ❷ studentArray[highPosition].grade) {
           ❸ highPosition = i;
        }
}
```

Здесь переменная highPosition ❶ занимает место highest. Поскольку мы напрямую не отслеживаем наивысшую оценку, когда приходит время сравнить наивысшую оценку с текущей, мы используем highPosition в качестве ссылки на studentArray ❷. Если оценка в текущей позиции массива выше, то позиция в нашем цикле обработки присваивается переменной highPosition ❸. По завершении цикла мы можем получить доступ к имени студента с наивысшей оценкой, используя studentArray[highPosition].name, кроме того, мы можем получить доступ к любым другим данным, относящимся к этому студенту.

Многомерные массивы

До сих пор мы обсуждали только одномерные массивы, поскольку они наиболее распространены. Двумерные массивы необычны, а массивы с тремя или более измерениями еще более редки. Это связано с тем, что большинство данных одномерны по своей природе. Кроме того, данные, которые по своей сути многомерны, могут быть представлены в качестве нескольких одномерных массивов, поэтому использование многомерного массива всегда остается на усмотрение программиста. Рассмотрим данные по бизнес-лицензиям в табл. 3.1. Они явно многомерные. Взгляните на них. Это же таблица. Тем не менее я представил эти многомерные данные в виде двух одномерных массивов — categoryThresholds и licenseCost. Я мог бы представить таблицу с данными в виде двумерного массива, например, следующим образом:

```
const double licenseData[2][numberCategories] = {
    {0.0, 50000.0, 150000.0, 500000.0},
    {50.0, 200.0, 1000.0, 5000.0}
};
```

Трудно выявить какое-либо преимущество от объединения двух массивов в один. Ни одна часть нашего кода не упрощается, поскольку нет никаких причин для одновременной обработки всех данных таблицы. Однако ясно то, что мы ухудшили удобочитаемость и простоту использования табличных данных. В исходной версии имена двух отдельных массивов ясно дают понять, какие данные хранятся в каждом из них. В случае с объединенным массивом нам, программистам, придется помнить о том, что ссылки на форму licenseData[0][] относятся к границам валового объема продаж различных категорий бизнеса, а ссылки на форму licenseData[1][] — к стоимости бизнес-лицензий.

Однако иногда использование многомерного массива оправдано. Предположим, мы обрабатываем данные о ежемесячных объемах

продаж для трех агентов по продажам, и одна из задач заключается в нахождении самых высоких уровней ежемесячных продаж у любого агента. Имея все данные в одном массиве 3*12, мы можем обработать весь этот массив за один раз, используя вложенные циклы:

```
   const int NUM_AGENTS = 3;
   const int NUM_MONTHS = 12;
❶ int sales[NUM_AGENTS][NUM_MONTHS] = {
    {1856, 498, 30924, 87478, 328, 2653, 387, 3754, 387587, 2873, 276, 32},
    {5865, 5456, 3983, 6464, 9957, 4785, 3875, 3838, 4959, 1122, 7766, 2534},
    {23, 55, 67, 99, 265, 376, 232, 223, 4546, 564, 4544, 3434}
   };
❷ int highestSales = sales[0][0];
  for (❸int agent = 0; agent < NUM_AGENTS; agent++) {
     for (❹int month = 0; month < NUM_MONTHS; month++) {
        if (sales[agent][month] > highestSales)
           highestSales = sales[agent][month];
     }
   }
```

Несмотря на то, что это простая адаптация базового кода для нахождения наибольшего числа, здесь есть несколько нюансов. Когда мы объявляем двумерный массив, обратите внимание, что инициализатор организован по агентам, то есть в виде 3 групп по 12, а не 12 групп по 3 ❶. Как вы увидите в следующей задаче, это решение может иметь определенные последствия. Мы инициализируем `highestSales` первым элементом массива, как обычно ❷. Вы можете осознать, что при первом прохождении вложенных циклов значения обоих счетчиков циклов будут равны 0, поэтому мы будем сравнивать это начальное значение `highestSales` с самим собой. Это не влияет на результат, однако иногда начинающие программисты могут попытаться обойти эту небольшую проблему путем добавления второй инструкции `if` в тело внутреннего цикла:

```
if (agent != 0 || month != 0)
    if (sales[agent][month] > highestSales)
       highestSales = sales[agent][month];
```

Однако это гораздо *менее* эффективно, чем предыдущая версия, поскольку нам придется выполнять 50 дополнительных сравнений, избегая при этом только одного.

Также обратите внимание на то, что я использовал для переменных цикла выразительные имена: `agent` для внешнего цикла ❸ и `month` для внутреннего ❹. В одном цикле, который обрабатывает одномерный массив, описательный идентификатор мало что дает. Однако в двойном цикле, который обрабатывает двумерный массив, выразительные идентификаторы помогают мне поддерживать порядок в измерениях и индексах, поскольку я могу посмотреть и увидеть, что я использую `agent` в том же измерении, в котором я использовал `NUM_AGENTS` при объявлении массива.

Даже при наличии многомерного массива иногда лучше всего иметь дело только с одним измерением за раз. Предположим, ис-

пользуя тот же массив sales, что и в предыдущем коде, мы захотели бы отобразить наивысшее среднемесячное значение объема продаж. Мы могли бы сделать это, используя двойной цикл, как было показано ранее, однако код был бы более понятным для чтения и простым для написания, если бы мы обращались со всем массивом как с тремя отдельными массивами и обрабатывали их по отдельности.

Помните код, который мы неоднократно использовали для вычисления среднего значения в массиве целых чисел? Давайте переделаем его в функцию:

```
double arrayAverage(int intArray[], int ARRAY_SIZE) {
   double sum = 0;
   for (int i = 0; i < ARRAY_SIZE; i++) {
      sum += intArray[i];
   }
   double average = sum / ARRAY_SIZE;
   return average;
}
```

С помощью этой функции мы можем снова изменить базовый код для нахождения наибольшего числа, чтобы отыскать агента с самым высоким среднемесячным объемом продаж:

```
double highestAverage = ❶arrayAverage(sales[0], 12);
for (int agent = 1; agent < NUM_AGENTS; agent++) {
   double agentAverage = ❷arrayAverage(sales[agent], 12);
   if (agentAverage > highestAverage)
      highestAverage = agentAverage;
}
cout << "Highest monthly average: " << highestAverage << "\n";
```

Изменение здесь заключается в двух вызовах функции arrayAverage. Первым параметром, принятым этой функцией, является одномерный массив типа int. В первом вызове мы передаем sales[0] в качестве первого аргумента ❶, а во втором вызове передаем sales[agent] ❷. Поэтому в обоих случаях мы указываем индекс для первого измерения двумерного массива sales, но не для второго измерения. Из-за прямой взаимосвязи между массивами и адресами в C++ эта ссылка указывает адрес первого элемента конкретной строки, который затем может использоваться нашей функцией в качестве базового адреса одномерного массива, состоящего только из этой строки.

Если вы запутались, посмотрите еще раз на объявление массива sales, в частности, на инициализатор. Значения в инициализаторе перечислены в том же порядке, в котором они будут сохранены в памяти при выполнении программы. Таким образом, значение sales[0][0], которое равно 1856, будет первым, за ним последует значение sales[0][1] 498 и так далее до последнего значения для первого агента sales[0][11], равного 32. Затем будут перечислены значения для второго агента, начиная со значения sales[1][0],

равного 5865. Поэтому несмотря на то, что концептуально массив представляет собой 3 строки по 12 значений, в памяти он сохранен как одна большая последовательность из 36 значений.

Важно отметить, что эта техника работает из-за порядка, в котором мы поместили данные в массив. Если бы массив был организован вдоль другой оси, то есть по месяцам, а не по агентам, мы не смогли бы сделать то, что сделали. Хорошая новость заключается в том, что есть простой способ убедиться в правильности настройки массива — для этого достаточно проверить инициализатор. Если данные, которые требуется обработать индивидуально, не являются непрерывными в инициализаторе массива, это значит, что вы организовали данные неправильно.

Последнее, что следует отметить в этом коде, — это использование временной переменной `agentAverage`. Поскольку на значение среднемесячного объема продаж для текущего агента мы потенциально ссылаемся дважды, один раз в условном выражении в инструкции `if` и затем снова в инструкции присваивания в теле, эта временная переменная исключает возможность того, что функция `arrayAverage` будет вызвана дважды для одних и тех же данных агента.

Этот метод рассмотрения многомерного массива в качестве массива массивов непосредственно вытекает из основного принципа разбиения задачи на более простые компоненты и значительно упрощает концептуализацию задачи с многомерными массивами. Тем не менее эта техника может показаться вам несколько сложной, и если вы относитесь к большинству начинающих программистов на C++, то, вероятно, немного побаиваетесь адресов и стоящей за ними арифметики. Думаю, что лучший способ справиться с этим — это максимально разграничить измерения, поместив один уровень массива внутри структуры `struct` или класса `class`. Предположим, что мы создали структуру `agentStruct`:

```
struct agentStruct {
  int monthlySales[12];
};
```

Взяв на себя труд по созданию структуры `struct`, мы могли бы подумать о добавлении других данных, вроде идентификационного номера агента, однако это и так упростит ход наших размышлений. После добавления структуры `struct` вместо создания двумерного массива с данными о продажах мы создаем одномерный массив с данными об агентах:

```
agentStruct agents[3];
```

Теперь, когда мы вызываем нашу функцию, возвращающую среднее значение элементов в массиве, мы не используем специальный прием C++, а просто передаем одномерный массив. Например:

```
int highestAverage = arrayAverage(agents[1].monthlySales, 12);
```

В каких случаях использовать массивы

Массив — это всего лишь инструмент. Как и с любым другим инструментом, важная часть процесса освоения заключается в понимании, когда его следует использовать, а когда — нет. Обсуждавшиеся до сих пор примеры задач в самих описаниях предполагали использование массивов. Тем не менее в большинстве случаев никто вам этого не подскажет, и вам придется самим принять решение относительно использования массива. Самые распространенные ситуации, в которых может потребоваться принять это решение, — это те, в которых даны агрегированные данные, но не указано, как они должны храниться. Например, в задаче определения моды задание *«Напишите код для обработки массива данных опроса...* звучало бы так: *«Напишите код для обработки набора данных опроса...* В данном случае выбор относительно использования массива будет за вами. Как вы его сделаете?

Помните, что мы не можем изменить размер массива после его создания. Если бы у нас закончилось свободное пространство, в нашей программе произошел бы сбой. Таким образом, первое, о чем нужно подумать, — это будем ли мы знать в том месте нашей программы, где потребуется структура агрегированных данных, сколько значений мы будем хранить или, по крайней мере, каково их примерное максимальное количество. Это не значит, что при написании программы мы должны знать размер массива. C++, как и большинство других языков, позволяет создать массив, размер которого определяется во время выполнения программы. Предположим, что задача определения моды была изменена таким образом, что мы заранее не знаем, сколько у нас есть ответов на опрос, а вместо этого данное значение поступает в программу в виде пользовательского ввода. В этом случае мы могли бы объявить динамический массив для хранения данных опроса.

```
  int ARRAY_SIZE;
  cout << "Number of survey responses: ";
  cin >> ARRAY_SIZE;
❶int *surveyData = new int[ARRAY_SIZE];
  for(int i = 0; i < ARRAY_SIZE; i++) {
     cout << "Survey response " << i + 1 << ": ";
   ❷cin >> surveyData[i];
   }
```

Мы объявляем массив, используя нотацию указателей, инициализируя его через вызов оператора `new` ❶. Из-за связи между указателем и типами массивов в языке C++ доступ к этим элементам можно получить, используя массивную нотацию ❷, несмотря на то, что `surveyData` объявлен как указатель. Обратите внимание, что, поскольку память для данного массива выделяется динамически, в конце программы, когда массив нам больше не нужен, мы должны освободить память:

```
delete[] surveyData;
```

Для массивов используется оператор delete[], а не обычный оператор delete. Хотя в случае с массивом целых чисел это не имеет никакой разницы, при создании массива объектов оператор delete[] гарантирует то, что отдельные объекты в массиве будут удалены до удаления самого массива. Поэтому вам следует выработать привычку всегда использовать оператор delete[] с динамическими массивами.

Необходимость очищать динамическую память — это проклятие для программиста, работающего с C++, однако если вы программируете на данном языке, вам этого не избежать. Начинающие программисты часто уклоняются от этой ответственности, поскольку их программы настолько малы и выполняются в такие короткие сроки, что они никогда не замечают вредных последствий от утечек памяти (когда память, которая больше не используется программой, не освобождается и, следовательно, остается недоступной для остальной системы). Не развивайте эту дурную привычку.

Обратите внимание на то, что мы можем использовать динамический массив только в том случае, если пользователь заранее сообщит нам количество ответов на опрос. Рассмотрим другой вариант, когда пользователь начинает вводить ответы опроса, не сообщая нам их количество и указывая на отсутствие дальнейших ответов путем ввода –1 (метод записи данных, известный как *значение-метка*). Можем ли мы использовать массив для решения этой задачи?

Это «серая зона». Мы могли бы использовать массив, если бы точно знали максимальное количество ответов. В этом случае мы могли бы объявить массив конкретного размера и предположить, что код в безопасности. Тем не менее в долгосрочной перспективе у нас все равно могли бы возникнуть проблемы. Что, если в будущем количество участников исследования увеличится? Что, если мы захотим использовать ту же программу с другим участником? В общем, зачем создавать программу с известным ограничением, если этого можно избежать?

Тогда лучше использовать набор данных нефиксированного размера. Как говорилось ранее, класс векторов из стандартной библиотеки шаблонов C++ действует как массив, но увеличивается по мере необходимости. После объявления и инициализации вектор может обрабатываться точно так же, как и массив. Мы можем присвоить вектору значение или извлечь значение, используя стандартную массивную нотацию. Если вектор заполнил свой изначальный размер и нам нужно добавить дополнительный элемент, мы можем сделать это, используя метод push_back. Решение модифицированной задачи с помощью вектора выглядит следующим образом:

```
❶ vector<int> surveyData;
❷ surveyData.reserve(30);
  int surveyResponse;
  cout << "Enter next survey response or -1 to end: ";
❸ cin >> surveyResponse;
```

```
    while (surveyResponse != -1) {
      ❹ surveyData.push_back(surveyResponse);
      cout << "Enter next survey response or -1 to end: ";
      cin >> surveyResponse;
    }
❺ int vectorSize = surveyData.size();
    const int MAX_RESPONSE = 10;
    int histogram[MAX_RESPONSE];
    for (int i = 0; i < MAX_RESPONSE; i++) {
      histogram[i] = 0;
    }
    for (int i = 0; i < vectorSize; i++) {
      histogram[surveyData[i] - 1]++;
    }
    int mostFrequent = 0;
    for (int i = 1; i < MAX_RESPONSE; i++) {
      if (histogram[i] > histogram[mostFrequent]) mostFrequent = i;
    }
    mostFrequent++;
```

В этом коде мы сначала объявляем вектор ❶, а затем резервируем место для 30 ответов на опрос ❷. Второй шаг не является абсолютно необходимым, однако резервирование некоторого пространства сверх того, которое требуется для вероятного количества элементов, предотвращает частое изменение размера вектора по мере добавления в него новых значений. Мы читаем первый ответ перед циклом ввода данных ❸, — это метод, который мы впервые использовали в предыдущей главе и который позволяет проверять каждое введенное значение перед обработкой. В данном случае мы хотим избежать добавления значения-метки –1 к нашему вектору. Результаты опроса добавляются в вектор с помощью метода `push_back` ❹. После завершения цикла ввода данных мы извлекаем размер вектора, используя метод `size` ❺. Мы могли бы самостоятельно подсчитать количество элементов в цикле ввода данных, но, поскольку вектор уже отслеживает свой размер, это позволит избежать лишних усилий. Остальная часть кода аналогична предыдущей версии с массивом и фиксированным количеством ответов, за исключением других имен переменных.

Тем не менее в этом обсуждении векторов упущен важный момент. Если мы считываем данные, поступающие непосредственно от пользователя, не зная о том, что мы начинаем с массива или другого набора данных, нам может понадобиться только один массив — для гистограммы. Мы можем не хранить данные опроса в массиве, но обрабатывать их по мере считывания. Нам нужна структура данных только тогда, когда нужно прочитать все значения перед обработкой или нужно обрабатывать значения более одного раза. В этом случае не нужно ни то ни другое:

```
const int MAX_RESPONSE = 10;
int histogram[MAX_RESPONSE];
for (int i = 0; i < MAX_RESPONSE; i++) {
    histogram[i] = 0;
}
int surveyResponse;
```

```
cout << "Enter next survey response or -1 to end: ";
cin >> surveyResponse;
while (surveyResponse != -1) {
     histogram[surveyResponse - 1]++;
     cout << "Enter next survey response or -1 to end: ";
     cin >> surveyResponse;
}
int mostFrequent = 0;
for (int i = 1; i < MAX_RESPONSE; i++) {
     if (histogram[i] > histogram[mostFrequent]) mostFrequent = i;
}
mostFrequent++;
```

Несмотря на то что этот код легко было написать, используя предыдущие версии в качестве руководства, было бы еще проще просто прочитать данные пользователя в массив и использовать предыдущий цикл обработки дословно. Преимущество этого процесса «на ходу» заключается в эффективности. Мы избегаем необходимости хранить каждый ответ на опрос, когда нужно единовременно сохранять только один. Наше решение, основанное на применении вектора, было *неэффективным в плане объема*: оно требовало использовать больше места, чем было необходимо, не давая при этом дополнительных преимуществ. Кроме того, прочитывание всех ответов на опрос в вектор само по себе потребовало бы использования цикла в дополнение к циклам для обработки всех ответов на опрос и нахождения наибольшего значения в гистограмме. Это означает, что версия с вектором делает больше, чем вышеприведенная. Поэтому версия с вектором также *неэффективна во времени*: она выполняет больше работы, чем требуется, не давая при этом дополнительных преимуществ. В некоторых случаях разные решения предлагают компромиссы, а программисты должны выбрать что им важнее: объем или время выполнения. Тем не менее в данном случае использование вектора делает программу неэффективной во всех отношениях.

В этой книге мы не будем тратить много времени на отслеживание любой неэффективности. Программистам иногда приходится заниматься *настройкой производительности*, которая представляет собой систематический анализ и повышение эффективности программы в плане объема и времени выполнения. Настройка производительности программы во многом напоминает тюнинг гоночного автомобиля — это изнуряющая работа, в ходе которой небольшие корректировки могут иметь большие последствия и для выполнения которой необходимы профессиональные знания о том, как механизмы работают «за кадром». Тем не менее, даже если у вас нет времени, желания или знаний, чтобы полностью настроить производительность программы, вам все же следует избегать решений, ведущих к серьезным потерям в эффективности. Необязательное использование вектора или массива не похоже на двигатель со слишком бедной смесью топлива и воздуха, скорее это аналогично поездке на пляж на автобусе, когда все ваши вещи могли бы уместиться в небольшом автомобиле.

Если мы уверены в том, что нужно будет обрабатывать данные несколько раз, и мы хорошо представляем себе максимальный размер набора данных, то последним критерием при принятии решения относительно использования массива будет произвольный доступ. Позже мы обсудим такие альтернативные структуры данных, как списки, которые, подобно векторам, могут по мере необходимости увеличиваться, но в отличие от векторов и массивов доступ к их элементам может осуществляться только последовательно. То есть, если мы хотим получить доступ к 10-му элементу в списке, мы должны пробежать первые 9 элементов, чтобы добраться до него. Напротив, *случайный доступ* означает, что мы можем получить доступ к любому элементу в массиве или векторе в любое время. Поэтому последнее правило заключается в том, что нам следует использовать массив, когда нужен произвольный доступ. Если нужен только последовательный доступ, мы можем рассмотреть другую структуру.

Вы можете заметить, что многие из программ в этой главе не учитывают последний критерий; мы получаем доступ к данным последовательно, а не случайным образом, и все же используем массив. Это приводит нас к значительному исключению для всех этих правил. Если массив мал, то ни одному из предыдущих возражений не придается большой вес. То, что считается «малым массивом» может варьироваться в зависимости от платформы или приложения. Суть в том, что если вашей программе требуется коллекция из 1 или из 10 элементов, каждый из которых требует 10 байт, то вы должны рассмотреть, стоят ли потенциальные потери 90 байтов, которые могут возникнуть в результате выделения памяти для массива максимально необходимого размера, того, чтобы поискать лучшее решение. Используйте массивы мудро, но не позволяйте лучшему быть врагом хорошего.

Упражнения

Как всегда, я призываю вас попробовать выполнить как можно больше упражнений.

3.1. Вы разочарованы тем, что мы мало занимались сортировкой? Я знаю, как это исправить. Чтобы убедиться в том, что вы освоились с функцией `qsort`, напишите код, в котором эта функция используется для сортировки массива студенческой структуры `struct`. Сначала попробуйте сортировать по оценкам, а затем попытайтесь сделать это, используя идентификатор студента.

3.2. Перепишите код для нахождения агента с наивысшим среднемесячным объемом продаж так, чтобы он находил агента с самым высоким *медианным* объемом продаж. Как указывалось ранее, медиана набора значений представляет собой такое значение, находящееся в середине, что половина значений набора больше него, а другая половина меньше. При наличии четного количества значений медиана является простым средним двух

значений в середине. Например, в наборе 10, 6, 2, 14, 7, 9 значения, находящиеся в середине, — это 7 и 9. Среднее значение 7 и 9 равно 8, поэтому 8 — это медиана.

3.3. Напишите функцию `bool`, которой передается массив и количество элементов в этом массиве и которая определяет, будут ли данные в этом массиве сортироваться. Для этого требуется только одна передача!

3.4. Вот вариация задания с массивом значений `const`. Напишите программу для шифрования методом подстановки. При таком шифровании все сообщения состоят из прописных букв и знаков препинания. Исходное сообщение называется открытым текстом, зашифрованный текст создается путем замены каждой буквы другой буквой (например, каждая буква C может стать буквой X). Для этой задачи жестко закодируйте массив `const` из 26 элементов `char` для шифрования и сделайте так, чтобы ваша программа прочитывала открытый текст и выводила эквивалент сообщения в виде зашифрованного текста.

3.5. Сделайте так, чтобы предыдущая программа преобразовывала зашифрованный текст обратно в открытый текст для проверки корректности кодирования и декодирования.

3.6. Для усложнения задачи шифрования текста сделайте так, чтобы ваша программа случайным образом генерировала шифровальный массив вместо жестко закодированного массива `const`. Фактически это означает помещение случайного символа в каждый элемент массива, однако помните, что вы не можете заменить букву самой собой. Итак, первым элементом не может быть буква A, и вы не можете использовать ту же букву для двух подстановок, то есть, если первым элементом является буква S, ни один другой элемент не может быть буквой S.

3.7. Напишите программу, которой дан массив целых чисел и которая определяет *моду*, то есть наиболее часто встречающееся в массиве число.

3.8. Напишите программу, которая обрабатывает массив объектов `student` и определяет квартили оценок, то есть оценки, которые необходимо получить студенту, чтобы успевать также хорошо или лучше, чем 25% студентов, 50% студентов и 75% студентов.

3.9. Рассмотрите следующую модификацию массива `sales`: поскольку продавцы приходят и уходят в течение года, теперь мы отмечаем месяц, предшествующий найму торгового агента или следующий за последним месяцем его работы, значением –1. Перепишите код для нахождения самого высокого значения среднего объема продаж или наибольшего значения медианного объема продаж, чтобы это компенсировать.

4

Решение задач с указателями и динамической памятью

В этой главе мы научимся решать задачи с использованием указателей и динамической памяти, которые позволят нам писать гибкие программы, способные обрабатывать заранее неизвестные объемы данных. Использование указателей и динамическое управление памятью – это высший пилотаж в программировании. Если вы умеете писать программы, способные выделять память в процессе выполнения, объединять их в полезные структуры и высвобождать по завершении, то вы не просто «кодер», вы – самый настоящий программист.

Из-за сложности применения указателей и из-за того, что многие популярные языки, например Java отказываются от их использования, некоторые начинающие программисты могут решить

пропустить изучение данного вопроса. Но это было бы ошибкой с их стороны. Серьезное программирование всегда будет использовать указатели и косвенную адресацию, даже если они будут скрыты за конструкциями языка высокого уровня. Следовательно, чтобы по-настоящему мыслить как программист, вы должны уверенно применять указатели и решать связанные с ними задачи.

Однако перед тем как перейти к решению задач с использованием указателей, мы тщательно изучим все аспекты их функционирования, как лежащие на поверхности, так и подводные камни. Это исследование принесет нам двойную пользу. Во-первых, мы научимся применять указатели наиболее эффективным способом. Во-вторых, развеяв завесу тайны вокруг указателей, мы сможем избавиться от страха перед их использованием.

Обзор основных свойств указателей

Как и в отношении тем, рассматриваемых в предыдущих главах, вы уже должны знать, что такое указатели, но для большей ясности пробежимся по их основным свойствам.

В C++ указатели обозначаются с помощью символа звездочки (*). Причем, в зависимости от контекста звездочка может означать как объявление указателя, так и непосредственное обращение к памяти. Чтобы объявить указатель, мы ставим звездочку между типом данных и идентификатором переменной:

```
int * intPointer;
```

В примере объявлена переменная `intPointer`, которая является указателем на данные типа `int`. Обратите внимание, что звездочка относится к идентификатору, а не к типу. В следующем примере `variable1` — это указатель на `int`, а `variable2` обычная переменная типа `int`:

```
int * variable1, variable2;
```

Оператор & перед именем переменной возвращает *адрес* ячейки. Это позволяет нам записать адрес переменной `variable2` в указатель `variable1` следующим образом:

```
variable1 = &variable2;
```

Мы также можем напрямую присвоить значение одного указателя другому:

```
intPointer = variable1;
```

Но особенно важно то, что прямо во время выполнения программы мы можем выделять память, доступ к которой можно осуществить только с помощью указателя. Для этой цели обычно используют оператор new:

```
double * doublePointer = new double;
```

Доступ к содержимому памяти с помощью указателя называется операцией *разыменования* и осуществляется с помощью оператора *, который указывается слева от идентификатора указателя. То есть также как и при объявлении. А содержание операции определяется по контексту. Например:

```
❶ *doublePointer = 35.4;
❷ double localDouble = *doublePointer;
```

Мы записали числовое значение типа `double` в участок памяти, который был выделен в предыдущем примере ❶, а потом присвоили содержимое этой памяти переменной `localDouble` ❷.

Чтобы освободить память, выделенную при помощи оператора `new`, когда в ней больше нет нужды, мы используем ключевое слово `delete`:

```
delete doublePointer;
```

Этот процесс детально описан в разделе «Вопросы памяти» далее в этой главе.

Преимущества использования указателей

Использование указателей существенно расширяет возможности по сравнению с использованием статической памяти, а также позволяет использовать память более эффективно. Перечислим три основных преимущества, возникающие при использовании указателей:

- использование структур данных, размер которых определяется во время выполнения программы;
- использование динамических структур данных, которые могут изменять свой размер во время выполнения программы;
- разделение памяти.

А теперь давайте рассмотрим каждое из них подробнее.

Структуры данных, размер которых определяется во время выполнения программы

Применение указателей позволяет создавать массивы, размер которых определяется во время выполнения программы, вместо того, чтобы определять их размер при проектировании приложения. Это уберегает нас от выбора между возможным переполнением массива и нерациональным использованием памяти в случае создания максимально больших массивов. Мы уже рассматривали структуры данных, размер которых определяется во время выполнения программы в разделе «В каких случаях использовать массивы». И вернемся к этому вопросу позже, в разделе «Строки переменной длины» далее в этой главе.

Динамические структуры

Также с помощью указателей можно создавать структуры данных, которые при необходимости способны увеличиваться или уменьшаться во время выполнения программы. Связный список, который вы, скорее всего, уже встречали, — самый простой пример динамической структуры данных. Несмотря на исключительно последовательный доступ к данным, связный список всегда занимает ровно столько памяти, сколько требуется для хранения его данных и ни байтом больше. Позже вы увидите, что более сложные структуры данных, основанные на указателях, используют упорядочивание и «формы», обеспечивающие лучшее отражение связей данных, чем это реализовано в массивах. По этой причине, даже учитывая произвольный доступ к данным массива, который структуры, основанные на указателях, обеспечить не могут, операция *поиска* (когда мы ищем элемент, наилучшим образом удовлетворяющий заданным критериям) может осуществляться намного быстрее в структурах, основанных на указателях. Позже в этой главе мы воспользуемся подобным преимуществом при создании структуры данных для хранения студенческих карточек, которая может увеличиваться по мере необходимости.

Разделение памяти

Указатели позволяют осуществлять совместный доступ к памяти, что повышает эффективность программного кода. Например, при вызове функции мы можем передать в нее указатель вместо того, чтобы копировать данные. Такой способ называется *передачей по ссылке.* Скорее всего, вы уже сталкивались с этим раньше. Это те самые параметры, у которых символ `&` указан между типом и именем в списке параметров функции:

```
void refParamFunction (int ❶& x) {
     ❷x = 10;
}

int number = 5;
refParamFunction(❸ number);
cout << ❹ number << "\n";
```

ПРИМЕЧАНИЕ. *Пробелы до и после символа* `&` *не обязательны, я поставил их по эстетическим соображениям. В коде других разработчиков вы можете встретить следующие формы записи:* `int& x`, `int &x` *и* `int&x`.

В примере параметр `x` ❶ является не копией аргумента `number` ❸, а представляет собой ссылку на участок памяти, в котором хранится значение переменной `number`. Таким образом, при изменении переменной `x` ❷ изменяется значение в ячейки памяти переменной `number` и в итоге на экран будет выведено 10 ❹. Передача по ссылке может также применяться и для возвращения результата из функции, как было показано в этом примере. В широком смысле передача по ссылке позволяет вызванной и вызывающей функциям совместно исполь-

зовать память, тем самым снижая накладные расходы. Если переменная, передаваемая в качестве аргумента, занимает килобайт памяти, то при передаче по ссылке вместо килобайта копируется только 32- или 64-битный указатель. При помощи ключевого слова const мы можем запретить функции изменять параметр, переданный по ссылке:

```
int anotherFunction(const int & x);
```

Ключевое слово const в объявлении ссылочного параметра x означает, что функция anotherFunction получит ссылку на аргумент, переданный при вызове, но не сможет изменять значение этого аргумента, так как он является константой.

В общем, мы можем использовать указатели подобным образом для того, чтобы различные части программы или структуры данных, задействованные в программе, могли использовать одни и те же данные без дополнительных затрат памяти.

В каких случаях использовать указатели

Как и массивы, указатели имеют свои недостатки и должны применяться только тогда, когда это действительно необходимо. Как понять, что применение указателя необходимо? Так как у нас есть только список возможностей, предоставляемых указателями, мы можем сказать, что их использование оправданно при возникновении потребности в одной или нескольких из этих возможностей. Если вашей программе требуется структура данных, но вы не можете заранее определить объем этих данных. Если вам требуется структура, которая может увеличиваться или уменьшаться во время выполнения программы. Или вы собираетесь передавать объемные данные между частями программы, то можете применять указатели. Если же таких потребностей нет, то вам следует держаться подальше от указателей и динамического распределения памяти.

Так как указатели имеют печальную репутацию одной из самых сложных особенностей языка C++, вы можете предположить, что программисты стараются не применять их без лишней необходимости. Я не единожды удивлялся, обнаружив обратное. Иногда программисты обманывают сами себя, предполагая, что указатели необходимы. Предположим, вы вызываете чужую функцию из библиотеки или интерфейса прикладного программирования со следующим прототипом:

```
void compute(int input, int* output);
```

Мы можем предположить, что эта функция написана на языке C, а не C++, и именно поэтому она использует указатель вместо ссылки (&) для создания «исходящего» параметра. При вызове данной функции программист может небрежно сделать что-то вроде этого:

```
int num1 = 10;
int* num2 = new int;
compute(num1, num2);
```

Этот код неэффективно использует память, так как он создает лишний указатель. Вместо памяти для двух переменных типа `int`, он захватывает место для двух переменных типа `int` и для указателя. Этот код также неэффективно использует время, так как выделение лишней памяти требует дополнительного времени (это мы рассмотрим в следующем разделе). И наконец, программист не должен забывать освобождать выделяемую память с помощью ключевого слова `delete`. Всего этого можно избежать, если использовать другой аспект оператора `&`, который позволяет получить адрес статической переменной, например:

```
int num1 = 10;
int num2;
compute(num1, &num2);
```

Строго говоря, мы все еще используем указатель во второй версии, но делаем это косвенно, без объявления или динамического выделения памяти.

Вопросы памяти

Чтобы разобраться, каким образом работает динамическое распределение памяти, нужно понимать механизм работы памяти в целом. Это одна из тех тем, ради которых начинающим программистам стоит изучать язык C++. В конце концов, все программисты должны понимать механизм работы системы памяти в современных компьютерах, и язык C++ ставит вас лицом к лицу с этой темой. Другие языки программирования скрывают большинство неприглядных подробностей работы системы памяти, поэтому начинающие программисты полагают, будто эти подробности их не касаются, что в корне неверно. Эти подробности никого не касаются, только пока все работает. Как только возникает проблема, дальнейшее пренебрежение основами работы памяти создает непреодолимое препятствие между программистом и ее решением.

Стек и куча

Язык C++ выделяет оперативную память в двух местах: стеке и куче. Как следует из названий, стек (от англ. *stack* – стопка) – аккуратный и организованный, а куча – бессвязная и беспорядочная. Название *стопка* очень меткое, так как помогает визуализировать устройство стека. Представьте стопку коробок наподобие той, что показана на рис. 4.1 (a). Когда вам нужно отправить коробку на хранение, вы просто ставите ее наверх стопки. Чтобы достать определенную коробку из стопки, вы сначала должны снять все коробки, стоящие сверху. Говоря языком программирования, однажды выделив блок памяти в стеке (коробку), вы уже не можете изменить его размер, так как следом расположены другие блоки памяти (коробки, стоящие сверху).

Язык C++ позволяет создавать ваш собственный стек, работающий по определенному алгоритму, но несмотря на это ваша программа все равно всегда будет использовать так называемый *стек вызовов*. Каждый раз при вызове функции (включая функцию `main`) будет выделяться блок памяти в голове стека. Такой блок называется *запись активации*. Разговор обо всем его содержимом выходит за рамки нашей книги, но для решения задач нам будет достаточно знать, что запись активации является местом хранения переменных. Память для всех локальных переменных, включая параметры функции, выделяется внутри записи активации. Давайте рассмотрим пример:

```
int functionB(int inputValue) {
  ❶ return inputValue - 10;
}
int functionA(int num) {
    int localVariable = functionB(num * 10);
    return localVariable;
}
int main()
{
    int x = 12;
    int y = functionA(x);
    return 0;
}
```

В этом коде функция `main` вызывает функцию `functionA`, которая, в свою очередь, вызывает функцию `functionB`.

На рис. 4.1 (б) можно увидеть упрощенную версию организации стека вызовов перед возвращением управления из `functionB` ❶. Записи активации всех трех функций будут расположены в стеке друг за другом, начиная с функции `main` в глубине стека. (Чтобы запутать вас еще больше, уточню, что стек может располагаться в памяти в обратном направлении, от больших адресов ячеек к меньшим. Однако ничего плохого не произойдет, если вы забудете об этом.) Логически запись активации `main` располагается в глубине стека, над ней лежит запись активации `functionA`, а еще выше запись активации `functionB`. Ни одна из нижних записей активации не может вернуть управление, пока не вернет управление `functionB`.

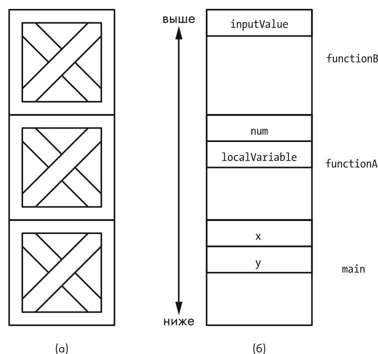


Рис. 4.1. Стопка коробок и стек вызовов функции

Если стек представляет собой хорошо организованную структуру, то куча, напротив, не имеет практически никакой организации. Представим, что вы опять храните коробки, но это хрупкие коробки и вы не можете поставить их друг на друга. У вас есть большая пустая комната для хранения коробок, и вы можете ставить их на пол в любое место. А так как коробки тяжелые, то вы стараетесь поставить их на ближайшее свободное к двери место. Эта структура имеет свои преимущества и недостатки по сравнению со стеком. С одной стороны, такая система хранения более гибкая и позволяет вам добраться до любой коробки в любой момент. С другой стороны, в комнате очень быстро начинается неразбериха. А если все коробки разного размера, то никак не получается использовать все полезное пространство пола. В итоге у вас пропадает слишком много места между коробками, так как туда уже ничего не помещается. Так как коробки сложно передвигать, то после выноса нескольких остаются промежутки, которые трудно заполнить. Если говорить языком программирования, то наша куча представляет собой как раз такой пол. Участок памяти идущих подряд адресов ячеек. Если программа предусматривает частые выделения и высвобождения памяти, то в итоге в памяти остается множество пустот между заполненными участками. Эта проблема носит название *фрагментация памяти*.

Для каждой программы создается своя собственная куча, память в которой распределяется динамически. Обычно в языке C++ это происходит с помощью оператора `new`, но также можно воспользоваться и старой функцией языка C для выделения памяти — `malloc`. Каждый вызов оператора `new` (функции `malloc`) выделяет кусок памяти в куче и возвращает указатель на него. Каждый вызов оператора `delete` (или функции `free`, если память выделили с помощью `malloc`) возвращает выделенный кусок в свободный резерв кучи. Из-за фрагментации можно использовать далеко не всю память из резерва. Если в начале программы в куче выделяется память под переменные A, B, и C, то можно ожидать, что они будут соседствовать. Если мы удаляем переменную B, то оставшийся от нее участок памяти можно заполнить только совпадающей или меньшей по размеру заявкой, пока переменные A или C не будут удалены.

Рис. 4.2 наглядно демонстрирует положение дел. В части (а) показан усеянный коробками пол комнаты. В какой-то момент пространство в комнате, возможно, было неплохо организовано, но теперь оно используется беспорядочно. Теперь некуда поставить маленькую коробку (б), при том что суммарное свободное пространство значительно превосходит ее по размерам. В части (в) представлена небольшая куча. Она поделена пунктирными линиями на минимальные (неделимые) ячейки памяти, которые могут быть однобайтовыми, размером с машинное слово или более крупными – это зависит от менеджера кучи. Участки, закрашенные серым, представляют собой выделенную память. Для наглядности проставлена нумерация в одном выделенном блоке. Как и в случае с фрагментированным полом, свободная память фрагментированной кучи разбита на отдель-

ные участки, что снижает возможность ее использования. Суммарно в куче 85 свободных ячеек памяти, но, как показывает стрелка, самый длинный блок составляет всего 17 смежных ячеек. Другими словами, если каждая ячейка размером 1 байт, то данная куча не может выполнить заявку на выделение памяти от оператора `new`, превышающую 17 байтов, несмотря на то, что свободно 85 байтов.

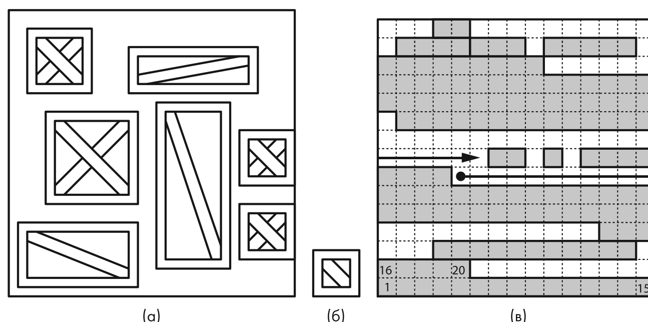


Рис. 4.2. Фрагментированный пол, коробка, которая не помещается, фрагментированная память

Объем памяти

При работе с памятью в первую очередь нужно разобраться с ограничениями ее использования. В современных компьютерных системах объемы памяти настолько велики, что можно принять ее за неограниченный ресурс, но на практике каждой программе выделяется ограниченный объем памяти. К тому же программистам стоит эффективнее работать с памятью, чтобы избежать общего замедления работы системы. В многозадачных операционных системах (а это практически все современные операционные системы) каждый байт, впустую растраченный одной программой, приближает момент нехватки памяти для всех запущенных программ. В этом случае операционная система начинает процесс *подкачки страницы*, или *свопинга* — перемещение отдельных блоков памяти из оперативной памяти во вторичное хранилище. Подкачка страницы существенно замедляет и даже практически останавливает работу системы. Такое состояние системы называется *пробуксовка*.

Стоит отметить, что для того, чтобы общий объем памяти, используемый программой, был минимальным, куча и стек должны быть максимально большими. Для доказательства давайте будем выделять по килобайту памяти из кучи, пока не начнутся проблемы:

```
const int intsPerKilobyte = 1024 / sizeof(int);
while (true) {
   int *oneKilobyteArray = new int[intsPerKilobyte];
}
```

Сразу поясню, что этот ужасный код написан исключительно для наглядной демонстрации вопроса. Если вы решите попробо-

вать его на своем компьютере, то для безопасности сначала сохранитесь. А произойти должно следующее: программа замедлится и операционная система выведет сообщение о том, что невозможно обработать исключение bad_alloc. Это исключение возникает, если выполняется оператор new, а в куче нет свободного блока памяти подходящего размера, чтобы удовлетворить запрос. Нехватка памяти в куче называется *переполнением кучи*. В некоторых системах может произойти общее переполнение кучи, в то время как в других программа начнет пробуксовывать задолго до того, как будет сгенерировано исключение bad_alloc (оператор new смог вызвать падение моей системы, только когда я стал выделять по два гигабайта памяти).

Похожая ситуация происходит и со стеком вызовов. Каждый вызов функции выделяет определенный объем памяти в стеке для каждой записи активации, даже для функций без параметров и локальных переменных:

```
❶ int count = 0;
  void stackOverflow() {
    ❷ count++;
    ❸ stackOverflow();
  }
  int main()
  {
    ❹ stackOverflow();
       return 0;
  }
```

В этом коде введена глобальная переменная ❶, которая в большинстве случаев — признак плохого стиля, но здесь мне необходима величина, которая будет присутствовать во время всех рекурсивных вызовов. Так как эта переменная объявлена вне функции и здесь нет никаких параметров и локальных переменных, то для нее не создается запись активации и не выделяется память. Функция увеличивает переменную count ❷ на единицу и производит рекурсивный вызов ❸. Рекурсия будет подробно рассмотрена в главе 6, но используется здесь для того, чтобы произвести максимальное количество вызовов функции. Запись активации функции остается в стеке, пока функция не прекратит свою работу. Таким образом, когда происходит первый вызов функции stackOverflow из функции main ❹, запись активации попадает в стек и не может быть удалена, пока не закончится первый вызов функции. А этого никогда не случится, так как функция осуществит второй вызов stackOverflow, и новая запись активации попадет в стек, а потом будет осуществлен третий вызов и так далее. Эти записи активации будут добавляться в стек, пока он не переполнится. В моей системе программа прекратила работу, когда значение переменной count достигло около 4900. Моя среда разработки Visual Studio выделяет под стек 1 Мб, следовательно, каждый вызов функции без локальных переменных и параметров создает запись активации размером около 200 байтов.

Время существования переменной

Временной *временем существования* промежуток между выделением и высвобождением памяти называется переменной. В случае с переменными, хранящимися в стеке — а это параметры и локальные переменные, — время существования определяется косвенно. Переменная появляется при вызове функции и исчезает, когда функция возвращает управление. В случае с переменными, хранящимися в куче — а это переменные, память под которые выделяется динамически с помощью оператора `new`, — мы можем напрямую задавать срок существования. Управление временем существования переменной — это бич каждого программиста C++. Наиболее очевидная проблема — это утечка памяти, ситуация, когда память в куче выделяется, но никогда не освобождается и недоступна с помощью указателей. Ниже показан простой пример:

```
❶ int *intPtr = new int;
❷ intPtr = NULL;
```

Мы объявляем указатель на переменную типа `int` ❶ и инициализируем ее путем выделения памяти под переменную типа `int` в куче. Во второй строке мы присваиваем указателю значение `NULL` ❷ (это, по сути, псевдоним нуля). Но переменная типа `int`, память под которую была выделена с помощью оператора `new`, все еще существует. И находится она, одинокая и всеми забытая, на своем месте в куче, ожидая удаления, которое может и вовсе не произойти. Мы не можем освободить память, выделенную под `int`, так как для вызова оператора `delete` необходим указатель, которого у нас больше нет. Если мы попытаемся продолжить вышеприведенный код командой `delete intPtr`, то получим ошибку, так как указатель нулевой.

Иногда вместо памяти, которую невозможно высвободить, мы получаем обратную проблему: пытаемся освободить память повторно, что вызывает ошибку времени выполнения. Может показаться, что эта проблема легко разрешима: следует просто не вызывать оператор `delete` дважды на одну и ту же переменную. Но не все так просто, ведь может оказаться, что множество переменных указывают на одну ячейку памяти. Если это действительно так и мы вызываем оператор `delete` для любой из этих переменных, то фактически мы очищаем память для всех переменных. Указатели, которым неявно присваивается значение `NULL`, называются *висячими*, а попытка применить к ним оператор `delete` вызывает ошибку времени выполнения.

Решение задач с указателями

Вот и настал момент, когда вы уже достаточно готовы к решению задач. Поэтому давайте рассмотрим несколько и попробуем применить указатели и динамическое распределение памяти для их решения. Для начала займемся динамическими массивами, на примере которых научимся отслеживать память в куче с помощью определенных

манипуляций. А потом мы проверим себя в деле с настоящими динамическими структурами.

Строка переменной длины

В первой задаче мы создадим функции для работы с переменными строкового типа. Здесь мы используем термин «строка» в его самом широком смысле — произвольной последовательности символов. Полагаю, нам необходимо обеспечить три функции нашей строки.

Задача: операции со строками переменной длины

Напишите, используя кучу, реализацию трех основных функций строковых переменных:

append Функция получает в качестве параметров строку и символ и добавляет символ к конец строки.

concatenate Функция получает две строки и добавляет символы второй строки к первой.

characterAt Функция получает строку и число и возвращает символ, находящийся на соответствующей числу позиции (нумерация символов начинается с нуля).

При написании кода учитывайте, что функция characterAt будет применяться часто, в то время как две другие функции — достаточно редко. Частота вызовов отразит относительную эффективность операций.

В этой задаче требуется таким образом реализовать строковую переменную, чтобы максимально быстро выполнять функцию `characterAt`, то есть обеспечить кратчайший путь достижения того или иного символа. Как вы, возможно, помните из темы предыдущей главы, именно массивы наилучшим образом обеспечивают произвольный доступ к данным. Давайте решим эту задачу, используя массив элементов типа `char`. Функции `append` и `concatenate` изменяют длину строки, из-за чего мы столкнемся со всеми теми проблемами, которые обсудили выше. Так как в условиях задачи не задан встроенный ограничитель длины строки, мы не можем объявить огромный массив и надеяться на лучшее. Вместо этого нам придется изменять размер массива во время выполнения программы.

Для начала давайте создадим псевдоним для нашего строкового типа с помощью спецификатора `typedef`. Мы знаем, что будем создавать динамические массивы, так что нам необходимо объявить строковый тип указателем на `char`.

```
typedef char * arrayString;
```

Сделав это, мы можем перейти к реализации функций. Руководствуясь принципом начинать с самого простого и понятного, мы можем быстро создать функцию `characterAt`.

```
char characterAt(arrayString s, int position) {
    ❶ return s[position];
}
```

Напомню из темы третьей главы, что если указатель указывает на массив, то мы можем получить доступ к элементам этого массива с помощью обычной индексной записи ❶. Замечу, однако, что неприятности могут начаться в том случае, если элемента с номером `position` больше нет в массиве `s`, так как этот код перекладывает ответственность за проверку второго параметра на вызывающего. Мы рассмотрим альтернативные решения в упражнениях в конце главы. А сейчас давайте перейдем к реализации функции `append`. В общих чертах нам известно, что должна делать эта функция, но, чтобы разобраться в деталях, давайте рассмотрим пример. Этот прием я называю *решением с помощью типичного примера*.

Начнем с необычного примера передачи данных в функцию или программу. Запишите все детали этого ввода, а также вывода. Тогда вы сможете написать код для типичной ситуации, а потом изменить его под свой пример, дважды перепроверяя каждый шаг, чтобы гарантировать достижение желаемого конечного состояния. Эта техника очень хорошо помогает при работе с указателями и динамическим распределением памяти, так как многое из происходящего в программе неочевидно на первый взгляд. Рассмотрение примера на бумаге помогает отслеживать все изменения значений в памяти, причем не только представленные в виде переменных, но и хранящиеся в куче.

Полагаю, что нам стоит начать со строковой переменной `test`, представляющей собой хранящийся в куче массив символьных переменных `t`, `e`, `s` и `t`, который мы хотим дополнить восклицательным знаком при помощи функции `append`. На рис. 4.3 можно увидеть состояние памяти «до» (а) и «после» (б) этой операции. На схеме слева от вертикальной пунктирной линии изображен стек (локальные переменные или параметры), а справа изображен участок памяти в куче, который был выделен динамически с помощью оператора `new`.

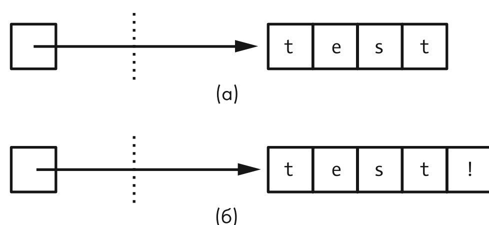


Рис. 4.3. Состояние памяти «до» (а) и «после» (б) вызова функции append.

Глядя на рисунок, я, кажется, уже предвижу возможные проблемы при работе нашей функции. Учитывая наш способ реализации строки, функция создаст новый массив, на один элемент длиннее предыдущего, и скопирует все символы из старого массива в новый. Но как мы узнаем размер первоначального массива? Из темы предыдущей главы известно, что мы должны самостоятельно следить за размером массива. Следовательно, здесь чего-то не хватает.

Если у вас есть опыт работы со строками стандартной библиотеки языка C/C++, вы сразу вспомните этот недостающий элемент, но если нет, то сможете быстро его вычислить. Давайте воспользуемся одной из наших техник решения задач, которая называется *поиск аналогий*. Возможно, стоит подумать о другой задаче с объектом неизвестной длины. В главе 2 мы оценивали корректность идентификационных номеров с произвольным количеством цифр для решения задачи «Проверка контрольной суммы Луна». В этой задаче нам было заранее неизвестно, сколько цифр введет пользователь. В итоге мы использовали цикл while, который повторялся до тех пор, пока не был введен символ окончания строки.

К сожалению, в конце массива нет никакого символа окончания строки. Но что если мы *вставим* символ окончания строки в последние элементы всех наших строковых массивов? Тогда мы сможем узнать размер массива так же, как до этого узнавали количество цифр в идентификационном коде. Единственный недостаток этого способа заключается в том, что символ окончания строки мы сможем использовать только в качестве *завершающего байта*. Нельзя сказать, что это очень существенное ограничение, но для большей гибкости нужно выбрать символ, который точно никто не сможет поместить в массив. Таким образом, мы будем использовать ноль для завершения массива, так как ноль является кодом отсутствия символа в ASCII и других системах кодирования символов. Это тот самый метод, который применяется в стандартной библиотеке C/C++.

После того как мы разобрались с этим вопросом, давайте рассмотрим подробнее, что же будет делать функция append с полученными данными. Как известно, функция получает два параметра: первый, arrayString, является указателем на массив символов, хранящийся в куче. Второй параметр представляет собой переменную типа char, которую нужно добавить в массив. Чтобы все стало понятно, давайте напишем предварительный вариант функции append и протестируем его:

```
void append(❶ arrayString& s, char c) {
}
void appendTester() {
  ❷ arrayString a = new char[5];
  ❸ a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] = 0;
  ❹ append(a, '!');
  ❺ cout << a << "\n";
}
```

Функция appendTester выделяет в куче память под нашу строку ❷. Обратите внимание, что длина массива составляет пять символов, чтобы мы могли поместить туда четыре буквы слова test, а также нулевой завершающий байт ❸. В следующей строке мы вызываем функцию append ❹, которая в этот момент является пустышкой. При ее написании я указал для параметра arrayString ссы-

лочный тип (&) ❶, так как функция будет создавать новый массив в куче. Ведь весь смысл как раз и заключается в использовании динамического распределения памяти для создания нового массива при изменении размера строки. Таким образом, значение переменной a при передаче в функцию отличается от того, которое возвращает функция, так как возвращаемая переменная указывает на новый массив. Обратите внимание, что раз наши массивы используют применяемый в стандартной библиотеке нулевой завершающий байт, то мы можем передать массив по указателю a прямо в стандартный поток вывода, чтобы проверить значение ❺.

На рис. 4.4 можно увидеть представление работы функции с тестовым примером. Завершающие байты находятся на своих местах и для простоты восприятия представлены как NULL. В части (б) показано состояние после выполнения функции. Очевидно, что переменная s указывает на новый участок памяти. Предыдущий массив окрашен серым, на этом рисунке я использую серый для освобожденных участков памяти. Изображение освобожденной памяти на рисунке напоминает нам фактически производить освобождение памяти.

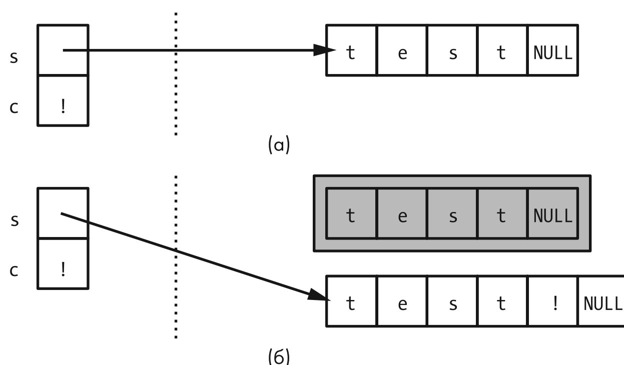


Рис. 4.4. Обновленные и доработанные состояния до (а) и после (б) выполнения функции append

Теперь мы можем записать функцию надлежащим образом со всеми подробностями.

```
void append(arrayString& s, char c) {
     int oldLength = 0;
   ❶ while (s[oldLength] != 0) {
          oldLength++;
     }
   ❷ arrayString newS = new char[oldLength + 2];
   ❸ for (int i = 0; i < oldLength; i++) {
          newS[i] = s[i];
     }
   ❹ newS[oldLength] = c;
   ❺ newS[oldLength + 1] = 0;
   ❻ delete[] s;
   ❼ s = newS;
}
```

В этом коде много чего происходит, так что давайте рассмотрим его шаг за шагом. В начале выполнения функции находится цикл, который находит завершающий байт массива и останавливается ❶. После выполнения цикла значение переменной `oldLength` равно количеству действительных символов в массиве (не считая нулевого завершающего байта). Мы выделяем в куче память для нового массива в размере `oldLength +2` ❷. Это одна из тех деталей, которую сложно реализовать, если держать их все в голове, но легко сделать правильно, если у вас есть рисунок. Рассматривая отображение нашего кода на рис. 4.5, мы видим, что значение переменной `oldLength` составляет 4. И мы знаем, что переменная `oldLength` должна равняться 4, так как в слове `test` четыре буквы, а также что новый массив на рис. 4.5 (б) содержит 6 символов, так как необходимо место для нового символа и для завершающего байта.

После выделения памяти для нового массива мы копируем в него все действительные символы из старого массива ❸ и добавляем в конец новый символ ❹, а также ставим нулевой завершающий байт на его законное место в новом массиве ❺. И опять схема помогает нам мыслить ясно. Чтобы прояснить все еще больше, рис. 4.5 демонстрирует, как было рассчитано значение переменной `oldLength` и на какую позицию это значение указывает в новом массиве. С таким наглядным напоминанием совсем не сложно расставить правильные значения в этих двух операциях присваивания.

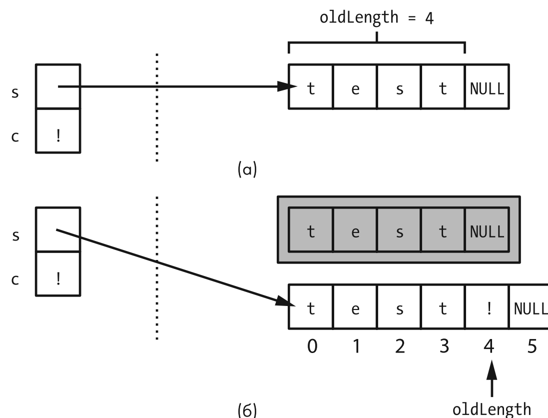


Рис. 4.5. Взаимосвязь локальной переменной, параметров и выделяемой памяти до и после применения функции `append`

Последние три строки кода в функции `append` относятся к серому прямоугольнику в части (б). Чтобы избежать утечки памяти, мы должны удалить из кучи старый массив, на который все еще ссылается параметр `s` ❻. И наконец, перед завершением вызова функции мы присваиваем параметру `s` указатель на новый, более длинный массив ❼. К сожалению, одна из причин того, что утечки памяти так распространены при программировании на языке C++, состоит в том, что, пока общий объем утечки памяти незначительный, ни програм-

ма, ни система не демонстрируют никаких симптомов происходящего. Поэтому утечка может быть и не замечена на этапе тестирования. Тем не менее мы как программисты должны тщательно подходить к вопросу времени существования выделенных участков памяти в куче. Если вы используете оператор `new`, то сразу думайте о том, где и когда нужно использовать соответствующий ему оператор `delete`.

Напомню, что при разработке этой функции нам очень помогла схема. Трудности программирования намного упрощаются с хорошей схемой, и я надеюсь, что все больше новых программистов будут составлять схемы, перед тем как писать код. Это возвращает нас назад к одному из основополагающих принципов решения задач: всегда составляйте план. Нарисовать хорошую схему для решения задачи — это все равно что нанести на карту маршрут до пункта назначения, перед тем как отравиться в дальний путь. Это требует чуть больших усилий вначале, но в итоге позволит вам сэкономить намного больше сил и времени.

СОСТАВЛЕНИЕ СХЕМ

Все, что вам потребуется для составления схемы — это карандаш и бумага. Однако если у вас есть время, то я бы рекомендовал использовать специальную программу для составления схем. Существуют специализированные инструменты с набором шаблонов для целей программирования, но для начала вполне подойдет и самый простой векторный графический редактор (термин *вектор* означает, что программа работает с линиями и кривыми, а не пикселами, как Photoshop). Я создал иллюстрации к этой книге в бесплатной программе Inkscape. Составление схем на компьютере позволяет организовать их хранение в том же месте, где вы будете писать код, который эти схемы иллюстрируют. Кроме того, диаграммы должны быть аккуратными, чтобы можно было понять их спустя какое-то время. И наконец, намного проще скопировать и изменить схему, нарисованную на компьютере как поступал я, когда подготавливал рис. 4.5 на основе рис. 4.4. А если вы хотите сделать какие-то быстрые пометки, то всегда можете распечатать себе экземпляр, чтобы покалякать на нем.

Давайте вернемся к функции `append`. Код выглядит солидно, но не стоит забывать, что мы написали этот код для типичного примера. То есть нам не стоит задаваться и утверждать, что код подойдет для всех возможных случаев. В первую очередь, рекомендуется проверить специальные случаи. *Специальный случай* в программировании — это ситуация, в которой достоверные данные спровоцируют нормальный код выдать ошибочный результат.

Следует отметить, что эта проблема начинается с недостоверных данных, таких как, например, недопустимые значения. При написании кода для этой книги я предполагал, что все данные, передаваемые в программы и функции, будут корректными. Например, если программа ожидает несколько числовых значений, разделенных запятыми, то я предполагаю, что именно эти данные программа и получит, а не посторонние символы, нечисловые значения и так

далее. Подобное предположение необходимо, чтобы не раздувать размеры кода и исключить повторение одних и тех же проверок вводимых данных из примера в пример. В реальной жизни, однако, мы должны принимать разумные меры предосторожности против некорректного ввода данных. Это так называемая устойчивость. *Устойчивая* к ошибкам программа способна работать даже при вводе некорректных данных. Например, такая программа может выдавать пользователю сообщение об ошибке вместо сбоя в работе.

Проверка на специальные случаи

Давайте опять обратимся к функции `append` и проведем проверку на специальные случаи. Другими словами, убедимся, что не возникнет никаких странных ситуаций с корректно введенными данными. Наиболее распространенными виновниками специальных случаев являются крайние значения, которые больше или меньше допустимого диапазона. Что касается функции `append`, то здесь нет верхних ограничений по величине нашего строкового массива, но лимитирован минимальный размер. Если у строки нет допустимых значений, то она будет представлять собой массив из одного элемента (нулевого завершающего байта). Как и раньше, давайте для прояснения ситуации нарисуем схему. Предположим, что мы добавили восклицательный знак к нулевой строке, как это показано на рис. 4.6.

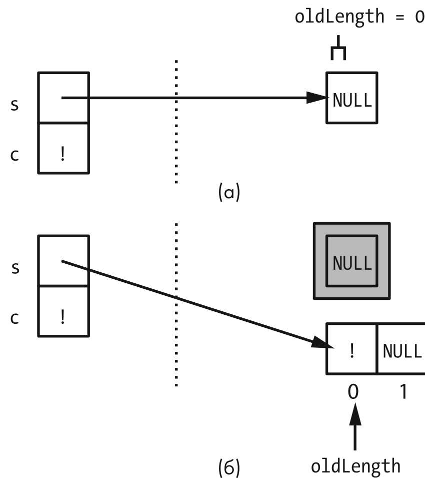


Рис. 4.6. Тестирование функции append в случае минимальной длины строки

Если вы взглянете на схему, то не заметите, что этот случай похож на специальный, но, чтобы убедиться, нам следует запустить нашу функцию с такими условиями. Давайте добавим несколько строк в код функции `appendTester`:

```
arrayString b = new char[1];
b[0] = 0;
append(b, '!');
cout << b << "\n";
```

Здесь тоже все работает. Теперь, когда мы убедились, что функция append работает корректно, полностью ли она нас устраивает? Код кажется простейшим, и я не чувствую от него никакого «запашка», но кажется, что он немного длинноват для такой простой операции. Когда я обдумывал функцию concatenate, мне пришло в голову, что, как и функция append, функция concatenate нуждается в определении длины строкового массива, или, может быть, двух строковых массивов. Так как обе операции предполагают использование цикла, который будет определять нулевой завершающий байт, то мы можем поместить этот код в отдельную функцию. А потом при необходимости сможем вызывать эту функцию из функций append и concatenate. Давайте приступим к реализации этой задумки и модифицируем соответствующим образом функцию append:

```
int length(arrayString s) {
  ❶int count = 0;
    while (s[count] != 0) {
        count++;
    }
    return count;
}
void append(arrayString& s, char c) {
  ❷int oldLength = length(s);
   arrayString newS = new char[oldLength + 2];
   for (int i = 0; i < oldLength; i++) {
       newS[i] = s[i];
  }
  newS[oldLength] = c;
  newS[oldLength + 1] = 0;
  delete[] s;
  s = newS;
}
```

Код в функции length ❶ — это тот же самый код, с которого раньше начиналась функция append. А в самой функции append мы заменили этот фрагмент кода на вызов функции length ❷. Функция length представляет собой *вспомогательную функцию*, то есть она инкапсулирует общую для нескольких других функций операцию. Кроме того, она уменьшает объем кода, а избавление от излишков делает код более надежным и гибким. Это также способствует решению нашей задачи, так как вспомогательные функции делят код на небольшие фрагменты, что упрощает возможность его повторного использования.

Копирование динамически созданной строки

Теперь пришло время разобраться с функцией concatenate. Мы будем пользоваться тем же подходом, что и при создании функции append. Сначала мы напишем пустую заготовку для функции, чтобы определить параметры и их типы. Затем нарисуем схему тестового примера и в итоге напишем код, соответствующий схеме. Давайте взглянем на заготовку и пример для тестирования.

```
void concatenate(❶arrayString& s1, ❷arrayString s2) {
}
void concatenateTester() {
     arrayString a = new char[5];
     a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] = 0;
     arrayString b = new char[4];
     b[0] = 'b'; b[1] = 'e'; b[2] = 'd'; b[3] = 0;
     concatenate(a, b);
}
```

Не забывайте, что в описании этой функции говорится, что символы второй строки (второго параметра) добавляются в конец первой строки. Таким образом, первый параметр функции будет ссылочным ❶, по той же причине, что и первый параметр функции `append`. Второй параметр ❷, однако, не будет изменяться внутри функции, так что он будет передаваться по значению. Теперь перейдем к условиям для общего случая: мы конкатенируем строковые переменные `test` и `bed`. Схема содержимого памяти до и после изображена на рис. 4.7.

Детали схемы нам уже знакомы из работы с функцией `append`. Для начала у нас есть два динамически созданных в куче массива, на которые указывают параметры `s1` и `s2`. Когда работа функции будет закончена, `s1` будет указывать на новый массив, длина которого будет составлять девять символов. Массив, на который сначала указывал `s1`, будет удален. Указатель `s2` и его массив останутся без изменений. В данный момент может показаться бессмысленным включение указателя `s2` и массива `bed` в нашу схему. Но если пытаться избежать ошибок при написании кода, отслеживание того, что не меняется, так же важно, как и отслеживание того, что меняется. Здесь я также пронумеровал элементы старого и нового массивов, так как это нам очень пригодилось при написании функции `append`. Теперь, когда все разложено по полочкам, давайте приступим к написанию функции.

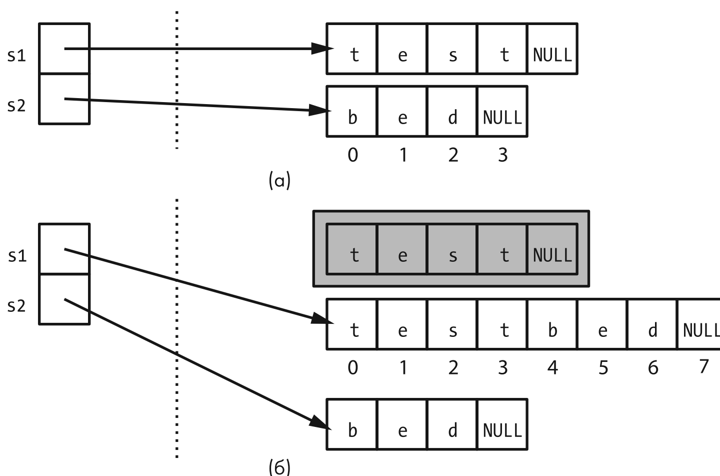


Рис. 4.7. Состояние памяти «до» (а) и «после» (б) вызова функции concatenate

```
void concatenate(arrayString& s1, arrayString s2) {
  ❶int s1_OldLength = length(s1);
    int s2_Length = length(s2);
    int s1_NewLength = s1_OldLength + s2_Length;
  ❷arrayString newS = new char[s1_NewLength + 1];
  ❸for(int i = 0; i < s1_OldLength; i++) {
        newS[i] = s1[i];
    }
    for(int i = 0; i < s2_Length; i++) {
        newS[❹s1_OldLength + i] = s2[i];
    }
  ❺newS[s1_NewLength] = 0;
  ❻delete[] s1;
  ❼s1 = newS;
}
```

Для начала мы определяем длину обеих строк, которые будем складывать ❶, а потом складываем эти величины и получаем длину объединенной строки. Не забывайте, что при определении длины строк мы учитываем только действительные члены массива без нулевого завершающего байта. Так что, когда мы создаем в куче массив для хранения новой строки ❷, мы выделяем память для количества элементов на один больше, чем объединенная длина двух массивов, чтобы добавить в конце нулевой завершающий байт. Затем мы копируем символы двух первоначальных строк в новую строковую переменную ❸. Первый цикл совсем не сложный, но не забывайте о расчете индексов во втором цикле ❹. Мы копируем элементы из начала массива `s2` в середину массива `newS`. У нас уже был другой пример с переводом значений из одного диапазона в другой, который мы выполняли в главе 2. Глядя на номера элементов на моей схеме, я могу увидеть, какие величины я должен сложить вместе, чтобы правильно определить индекс элемента-цели. В оставшейся части функция добавляет нулевой завершающий байт в конец новой строки ❺. Так же как и в функции `append`, мы удаляем массив, указателем на который является первый параметр ❻, и перенаправляем первый параметр на новый строковый массив ❼.

Этот код выглядит вполне рабочим, но, как и прежде, нужно убедиться, что мы создали функцию, которая успешно работает не только для тестового примера, но и во всех остальных случаях. С большой вероятностью проблемы могут начаться в случае, когда один или оба параметра окажутся строками нулевой длины (содержащими только нулевой завершающий байт). Мы должны проверить этот случай, прежде чем двигаться дальше. Напомню, что при проверке корректности кода, содержащего указатели, следует проверить именно сами указатели, а не только величины в куче, на которые они ссылаются. Рассмотрим тестовый пример:

```
  arrayString a = new char[5];
  a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] = 0;
  arrayString c = new char[1];
  c[0] = 0;
  concatenate(c, a);
  cout << a << "\n" << c << "\n";
❶ cout << (void *) a << "\n" << (void *) c << "\n";
```

Я хотел бы удостовериться, что после завершения вызова функции `concatenate` с параметрами `a` и `c` они оба ссылаются на строки с одинаковым значением `test`. Не менее важно, однако, чтобы они ссылались на *разные* строки, как это показано на рис. 4.8 (а). Я проверяю это с помощью второй операции вывода, заменив тип переменных на `void *`, что позволяет вывести содержимое непосредственно указателя ❶. Если указатели сами по себе содержат одинаковые значения, тогда можно сказать, что они являются *перекрестными ссылками*, как показано на рис. 4.8 (б). Если перекрестные ссылки появляются в программе непреднамеренно, то возникает очень коварная проблема, которую трудно обнаружить в большой программе. Изменение содержимого одной переменной в куче таинственным образом влечет за собой изменение содержимого другой, а, по сути, той же самой переменной. Также не стоит забывать, что если два указателя являются перекрестными, то при удалении одного из них второй превращается в висячую ссылку. Таким образом, мы должны быть очень внимательны при просмотре кода и всегда проверять возможные перекрестные ссылки.

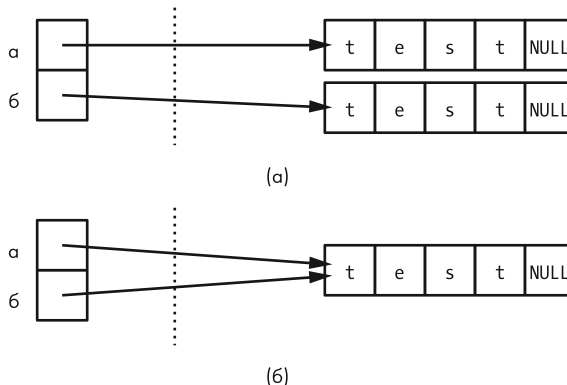


Рис. 4.8. Функция `concatenate` должна выдавать в результате две отдельные строки (а), а не два перекрестных указателя (б)

Теперь мы реализовали все три функции: `characterAt`, `append` и `concatenate`, следовательно, задача решена.

Связные списки

Давайте теперь попробуем решить что-то похитрее. Использовать указатели станет сложнее, но мы будем пользоваться схемами для прояснения ситуации.

Задача: отслеживание неизвестного количества студенческих карточек

В этой задаче вы должны будете реализовать функции для создания и управления студенческими карточками. Такая карточка содержит числовые данные: номер студента и балл. Нужно реализовать две функции:

addRecord Функция получает указатель на коллекцию студенческих карточек, номер студента и балл и добавляет новую запись с этими данными в коллекцию.

averageRecord Функция получает указатель на коллекцию студенческих карточек и возвращает среднее значение баллов студентов этой коллекции в формате double.

Коллекция может быть любого размера. Предполагается, что функция addRecord будет использоваться часто, следовательно, она должна быть реализована наиболее эффективным способом.

Из всего многообразия методов, соответствующих техническому заданию, мы выберем связные списки, которые позволят попрактиковаться в работе с указателями. Вы, должно быть, уже сталкивались со связными списками раньше, а если нет, то знайте, что создание связного списка представляет собой нечто радикально отличающееся от всего, что мы обсуждали раньше в этой книге. Хороший специалист по решению задач мог бы прийти к любому из предыдущих решений, потратив достаточное количество времени и тщательно все обдумав. Большинство программистов, однако, не смогли бы разработать концепцию связных списков без посторонней помощи. Тем не менее, однажды увидев и овладев основами, вы сможете придумывать другие связные структуры по накатанной. Связный список является по-настоящему динамической структурой. Наши строковые массивы хранились в динамически выделенной памяти, но, будучи однажды созданными, они становились статическими структурами, которые не меняли своего размера и только иногда перемещались. В отличие от них, связный список постепенно увеличивается звено за звеном, как гирлянда.

Построение списка узлов

Давайте создадим простейший связный список студенческих карточек. Для объявления связного списка вам понадобится создать структуру с помощью ключевого слова struct, которая содержит указатель на такую же структуру вдобавок к тем данным, которые вы хотите сохранить в коллекции, реализованной с помощью связного списка. В нашем случае структура struct будет содержать номер студента и его балл.

```
  struct ❶listNode {
    ❷int studentNum;
      int grade;
    ❸listNode * next;
  };
❹ typedef listNode * studentCollection;
```

Мы создали структуру listNode ❶. Структуры, которые создаются для реализации связного списка, всегда называются *узлами*. Можно предположить, что они получили это название по аналогии с ботаническим термином, обозначающим точку на стволе, из которой растут новые ветви. Узел содержит номер студента ❷ и его балл, которые составляют полезное содержимое узла. Также узел содержит указатель на ту же структуру, которую мы только что описали ❸. Когда

большинство программистов видит эту конструкцию в первый раз, они не могут понять, как же можно создать структуру, ссылающуюся на саму себя. Но все это вполне законно, и скоро вы поймете это. Обратите внимание, что указатели, ссылающиеся на самих себя, носят названия типа *next* (от англ. следующий), *nextPtr* и тому подобные. И наконец, в коде объявляется указатель на список узлов ❹. Это улучшит читаемость наших функций. Теперь давайте создадим пробный связный список с использованием этих типов:

```
❶ studentCollection sc;
❷ listNode * node1 = new listNode;
❸ node1->studentNum = 1001; node1->grade = 78;
  listNode * node2 = new listNode;
  node2->studentNum = 1012; node2->grade = 93;
  listNode * node3 = new listNode;
❹ node3->studentNum = 1076; node3->grade = 85;
❺ sc = node1;
❻ node1->next = node2;
❼ node2->next = node3;
❽ node3->next = NULL;
❾ node1 = node2 = node3 = NULL;
```

Вначале мы объявляем структуру `studentCollection`, `sc` ❶, которая в конечном счете станет названием нашего связного списка. Затем мы объявляем переменную `node1` ❷, которая является указателем на `listNode`. Еще раз повторю, `studentCollection` – синоним `listNode *`, но для лучшей читаемости я использую тип `studentCollection` только для переменной, которая будет ссылаться на весь список студенческих карточек. После объявления указателя `node1` и его инициализации с помощью динамически созданного в куче узла ❷ мы присваиваем значение полям `studentNum` и `grade` в узле ❸. В этот момент следующее поле не определено. У нас не справочник по синтаксису, но если вы раньше не встречали нотацию `->`, то поясню, что она используется для обращения к полям (свойствам) структуры (или класса). Таким образом, запись `node1->studentNum` означает «поле `studentNum` в переменной типа `struct`, на которую указывает `node1`» и эквивалентна записи `(*node1).studentNum`. Потом мы повторяем ту же процедуру для узлов `node2` и `node3`. На рис. 4–9 можно увидеть состояние памяти после присвоения значений полям последнего узла. Чтобы показать узел `struct`, на этой схеме мы используем метод разделенных ящиков, который мы уже применяли раньше для массивов.

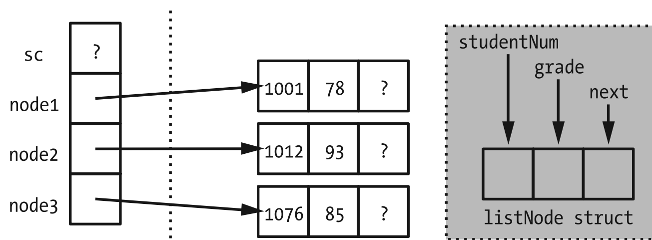


Рис. 4.9. На полпути к построению связного списка

Теперь, когда у нас уже есть все узлы, мы можем выстроить их между собой в связный список. Это как раз то, что делает оставшийся код из предыдущего листинга. Во-первых, мы направляем указатель `studentCollection` на первый узел ❺. Затем мы направляем поле `next` первого узла на второй узел ❻, а после этого направляем поле `next` второго узла на третий узел ❼. На следующем шаге мы присваиваем значение `NULL` (еще раз уточню, что это всего лишь псевдоним нуля) полю `next` третьего узла ❽. Мы делаем это по тем же причинам, по которым ставили нулевой символ в конце массивов в предыдущей задаче: чтобы обозначить окончание структуры. Точно так же, как нам был необходим специальный символ для обозначения завершения массива, нам нужен ноль в поле `next` последнего узла нашего связного списка, чтобы мы точно знали, что это последний узел. Наконец, чтобы подчистить хвосты и избежать возможной проблемы перекрестных ссылок, мы присваиваем значение `NULL` каждому из индивидуальных указателей на узлы ❾. Конечное состояние памяти можно увидеть на рис. 4.10.

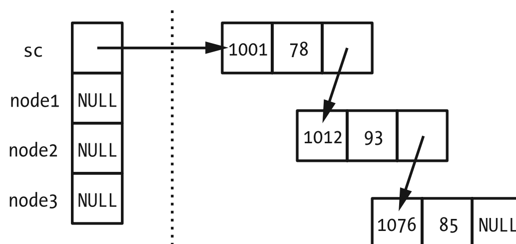


Рис. 4.10. Завершенный связный список

Глядя на рисунок, легко понять, почему структура называется связным списком: каждый узел списка связан со следующим узлом. Вы будете часто встречать линейное изображение связных списков, но я предпочитаю распределенный по памяти вид этой схемы, так как он подчеркивает, что эти узлы не имеют никакого отношения друг к другу, кроме ссылок. Каждый из них может храниться в любом уголке кучи. Удостоверьтесь, что вы разобрались с кодом, перед тем как согласиться со схемой.

Обратите внимание, что в конечном состоянии остался только один указатель на стек переменная типа `studentCollection sc`, которая указывает на первый узел. Указатель, внешний для списка (не является полем `next` в узле списка), который указывает на первый узел связного списка, называется *головной указатель*. Чисто символически эта переменная представляет список целиком, но, конечно, по факту она указывает только на первый узел. Чтобы достичь второго узла, нам нужно идти через первый, а чтобы добраться до третьего – через первый и второй и т.д. Это значит, что связный список допускает только последовательный доступ, в отличие от произвольного доступа, который предоставляют массивы. Последовательный доступ является слабым местом связного списка. А сильная сторона

связного списка, как уже обсуждалось раньше — это способность увеличиваться или уменьшаться в размерах путем добавления или удаления узлов, без создания абсолютно новых структур и копирования данных, как мы это делали при работе с массивами.

Добавление узлов в список

Теперь давайте перейдем к реализации функции addRecord. Эта функция будет создавать новый узел и добавлять его к существующему связному списку. Мы будем пользоваться той же техникой, что и при решении предыдущей задачи. Во-первых, заготовка и пробный вызов. Для тестирования мы будем дорабатывать код предыдущего листинга, таким образом, sc уже существует в качестве головного указателя на связный список с тремя узлами.

```
  void addRecord(studentCollection& sc, int stuNum, int gr) {
  }
❶ addRecord(sc, 1274, 91);
```

Еще раз повторю, что ❶ вызов функции произойдет в конце предыдущего листинга. Раз у нас теперь есть заготовка функции с параметрами, то мы можем нарисовать схему состояния «до» вызова, как это показано на рис. 4.11.

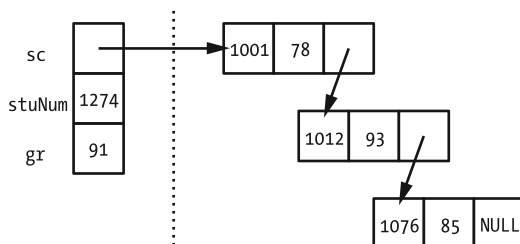


Рис. 4.11. Состояние «до» вызова функции addRecord

Что касается состояния «после», то у нас есть выбор. Нетрудно догадаться, что нам нужно создать новый узел, хранящийся в куче, и скопировать в него параметры stuNum и gr. Но открытым остается вопрос о позиции, на которую новый узел будет добавлен в список. Наиболее очевидный выбор — это добавление в конец: тогда просто нужно направить нулевой указатель в поле next на новый узел. Это будет соответствовать рис. 4.12.

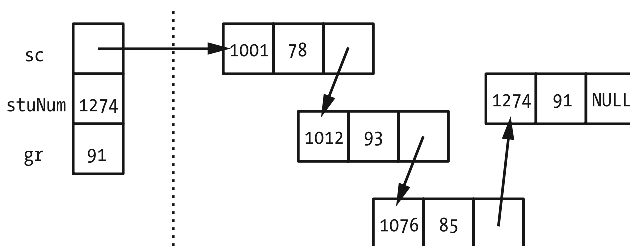


Рис. 4.12. Предполагаемое состояние «после» вызова функции addRecord

Но если мы предполагаем, что порядок записей не имеет значения (нам не нужно поддерживать записи в определенном порядке, в котором они были добавлены в коллекцию), то это неверный выбор. Чтобы понять, почему, давайте рассмотрим коллекцию не из трех студенческих карточек, а из 3000. Чтобы добраться до последней записи нашего связного списка с целью изменить ее поле next, требуется пройти через все 3000 узлов. Это непозволительно неэффективно, так как мы можем добраться до нового узла и не проходя через *каждый* из существующих узлов. На рис. 4.13 можно увидеть, как это сделать. После создания нового узла он добавляется в *начало* списка, а не в конец. В состоянии «после» наш головной указатель sc указывает на новый узел, в то время как поле next нового узла указывает на бывший первый узел списка, тот, в котором хранится запись о студенте с номером 1001. Стоит отметить, что, пока мы присваиваем значение полю next нового узла, изменяется только один указатель sc и ни одно из значений в существующих узлах не меняется и даже не проверяется. Напишем код, ориентируясь на схему:

```
void addRecord(studentCollection& sc, int stuNum, int gr) {
  ❶ listNode * newNode = new listNode;
  ❷ newNode->studentNum = stuNum;
    newNode->grade = gr;
  ❸ newNode->next = sc;
  ❹ sc = newNode;
}
```

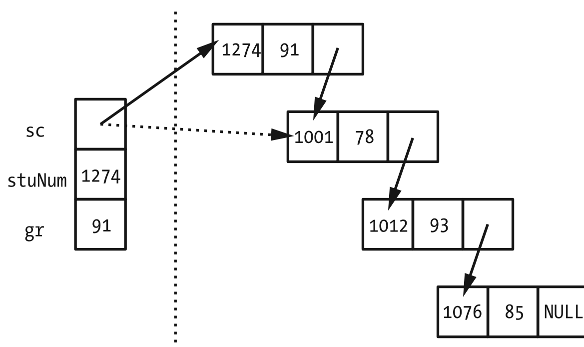


Рис. 4.13. Приемлемое состояние «после» вызова функции addRecord. *Пунктирная стрелка указывает на предыдущее значение указателя* sc

Еще раз подчеркиваю, что разобраться со схемой и этим кодом намного проще, чем пытаться представить и понять все это у себя в голове. Код напрямую вытекает из иллюстрации. Мы создаем новый узел ❶ и копируем номер студента и его балл из параметров ❷. Затем привязываем новый узел к списку первым, направив указатель его поля next на бывший первый узел (копируя значение указателя sc) ❸, а далее перенаправляем указатель sc на новый узел ❹. Стоит отметить, что два последних шага происходят именно в этом порядке. Мы должны использовать первоначальное значение указателя sc, перед тем как изменить его. Также стоит отметить, что раз

мы меняем значение указателя `sc`, то он должен быть ссылочным параметром.

Как всегда после написания кода для типичного примера, мы должны проверить возможные специальные случаи. В этом примере мы будем проверять, что функция работает в ситуации с пустым списком. В примере со строковым массивом пустая строка была также и указателем, так как у нас по-прежнему был массив, на который можно было указывать и который содержал только нулевой завершающий байт. Здесь же количество узлов совпадает с количеством студенческих карточек, и поэтому пустой список будет головным указателем, содержащим `NULL`. Будет ли наш код по-прежнему работать, если мы попытаемся соединить типичный пример с нулевым указателем? На рис. 4.14 представлены состояние «до» и желаемое состояние «после».

Запустив этот пример с кодом, мы видим, что все прекрасно работает. Новый узел создается, как и раньше. Так как указатель `sc` содержит `NULL` перед началом работы функции, то именно это значение копируется в поле `next` нового узла ❸. Это соответствует нашим ожиданиям, и список, содержащий один узел, получает корректное окончание. Стоит отметить, что, если мы продолжим реализовывать другую идею – добавление нового узла в конец связного списка – изначально пустой список станет *специальным* случаем, так как это будет единственный случай, когда указатель `sc` будет меняться.

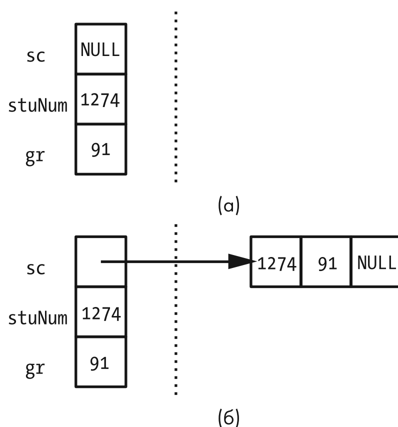


Рис. 4.14. Состояния «до» и «после» вызова функции `addRecord` *в случае нулевого списка*

Обход списка

И вот наконец настало время для того, чтобы разобраться с функцией `averageRecord`. Как и раньше, мы начнем с заготовки и схемы. Рассмотрим заготовку функции и вызов типичного примера. Полагаю, что типичный вызов ❶ происходит после создания нашего первоначального типичного списка, как показано на рис. 4.10.

```
  double averageRecord(studentCollection sc) {
  }
❶ int avg = averageRecord(sc);
```

Как вы видите, я решил вычислять среднее значение в формате int, как мы делали это с массивами в предыдущей главе. В зависимости от задачи, однако, может быть лучше рассчитывать среднее как значение с плавающей точкой. Сейчас нам нужна схема, но мы уже рассматривали примерно то же самое состояние «до» на рис. 4.9. Нам не нужна схема состояния «после», потому что эта функция не вносит никаких изменений в нашу динамическую структуру, а только выдает результат. Нам только нужно знать ожидаемый результат, который в данном случае составляет примерно 85,3333.

Так как же мы будем рассчитывать средний балл? Концепция понятна из нашего опыта расчета среднего значения всех величин в массиве. Мы должны просуммировать все значения в коллекции, а потом разделить сумму на количество значений. Когда мы писали код для расчета среднего значения в массиве, то использовали цикл for от 0 до длины массива минус один, используя итератор цикла как индекс массива. В случае со связным списком мы не можем использовать цикл for, так как нам заранее неизвестно количество итераций. Мы должны перебирать узлы один за другим, пока не достигнем нулевого значения в поле next очередного узла, что будет означать окончание списка. Такие условия предполагают использование цикла while, что-то подобное мы использовали ранее в этой главе для перебора массива неизвестной длины. Подобный проход по связному списку от начала к концу называется *обход списка*. Это одна из базовых операций по работе со связным списком. Давайте реализуем идею обхода для решения этой задачи:

```
double averageRecord(studentCollection sc) {
  ❶ int count = 0;
  ❷ double sum = 0;
  ❸ listNode * loopPtr = sc;
  ❹ while (loopPtr != NULL) {
      ❺ sum += loopPtr->grade;
      ❻ count++;
      ❼ loopPtr = loopPtr->next;
    }
  ❽ double average = sum / count;
    return average;
}
```

Сначала мы объявляем переменную count для хранения количества пройденных узлов списка ❶, в итоге там будет храниться количество узлов в списке, которое мы будем использовать для расчета среднего значения. Дальше мы объявляем переменную sum для хранения кумулятивной суммы баллов из списка ❷. Затем мы объявляем указатель listNode * с именем loopPtr, который будет использоваться для обхода списка ❸. Это эквивалент итератора цикла for, который отражает наше текущее положение в связном списке, но не в виде номера позиции, а сохраняя указатель на узел, который мы в данный момент проходим.

С этого момента начинается обход. Цикл обхода длится до тех пор, пока указатель цикла не дойдет до значения NULL ❹. Внутри

цикла мы добавляем значение поля grade текущего узла к значению переменной sum ❺. Мы инкрементируем переменную count ❻, а затем копируем значение поля next текущего узла в указатель цикла ❼. Это приводит к перемещению нашего обхода на один узел вперед. Это достаточно сложная часть кода, так что давайте убедимся, что вы все поняли правильно. На рис. 4.15 я продемонстрировал, как меняется указатель цикла. Буквы от (а) до (г) отмечают различные моменты во время выполнения кода нашего типичного примера, демонстрируя различные моменты времени существования указателя loopPtr, а также источники его значений. Момент (а) представляет начало цикла, указатель loopPtr только что был инициирован с помощью значения указателя sc. Таким образом, loopPtr указывает на первый узел списка, также как и указатель sc. Дальше во время первой итерации цикла значение балла первого узла 78 добавляется в переменную sum. Значение поля next первого узла присваивается указателю loopPtr, то есть теперь loopPtr указывает на второй узел в списке, и это уже момент (б). Во время второй итерации мы добавляем 93 к значению переменной sum и копируем значение поля next второго узла в указатель loopPtr, и это момент (в). И наконец, во время третьей и последней итерации мы добавляем 85 к значению переменной sum и копируем NULL из поля next третьего узла в указатель loopPtr, и это момент (г). Когда мы подходим к началу цикла while в следующий раз, цикл заканчивается, потому что указатель loopPtr содержит NULL. Так как мы инкрементировали переменную count в каждой итерации, то она равна трем.

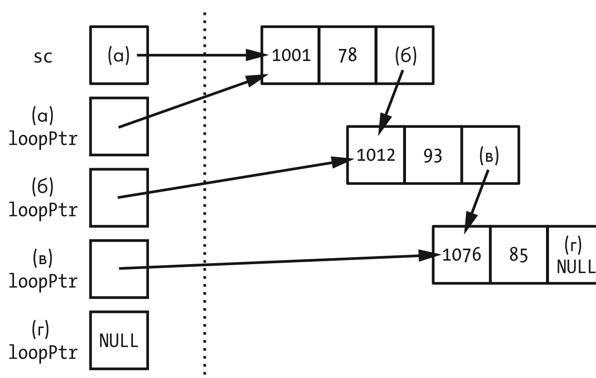


Рис. 4.15. Изменение локальной переменной loopPtr *в итерациях цикла при вызове функции* averageRecord

Когда выполнение цикла заканчивается, мы просто делим sum на count и возвращаем результат ❽.

Для типичного случая код работает, но, как всегда, мы должны проверить потенциальные специальные случаи. И опять мы будем проверять самый очевидный специальный случай при работе со списком — пустой список. Что случится с нашим кодом, если указатель sc будет содержать NULL при вызове функции?

Догадались? Код рухнет. Я просто был обязан сделать так, чтобы один из специальных случаев закончился плохо, иначе вы не будете воспринимать меня всерьез. На самом деле нет никаких проблем в работе цикла, если у нас пустой связный список. Если указатель sc содержит NULL, то указатель loopPtr инициализируется значением NULL, цикл заканчивается сразу после начала, и переменная sum остается равной нулю, что совершенно обоснованно. Проблема возникает в тот момент, когда мы совершаем действие деления для вычисления среднего значения ❸, переменная count тоже равна нулю, что означает деление на ноль, которое приводит либо к обрушению программы, либо к неверному результату. Чтобы разобраться с этим специальным случаем, нужно в конце цикла проверять переменную count на нулевое значение. Но почему бы не разрешить эту ситуацию путем проверки указателя sc? Давайте добавим следующий код в качестве новой первой строки в функцию averageRecord:

```
if (sc == NULL) return 0;
```

Как мы убедились на этом примере, решать специальные случаи совсем не сложно. Мы только должны убедиться, что верно их определили.

Заключение и дальнейшие шаги

Эта глава всего лишь слегка затронула задачи, которые решаются с помощью указателей и динамического распределения памяти. Вы еще столкнетесь с указателями и выделением памяти в куче по мере изучения этой книги. Например, в методах объектно ориентированного программирования, которые будут рассматриваться в главе 5, использование указателей особенно полезно. Они позволяют инкапсулировать указатели таким образом, чтобы обезопасить нас от утечек памяти, висячих указателей и прочих распространенных проблем при работе с указателями.

Даже при том, что еще многое осталось неизученным в данной теме, вы вполне можете развивать свои навыки работы со структурами, основанными на указателях, увеличивая сложность задач, если будете следовать основным идеям, описанным в этой главе. Во-первых, применяйте основные методы решения задач. Во-вторых, применяйте специфические методы для работы с указателями и составляйте схемы для визуализации каждого решения, перед тем как приступить к написанию кода.

Упражнения

Я не шучу по поводу выполнения упражнений. Вы же не собираетесь просто читать главу за главой, не так ли?

4.1. Придумайте свое собственное упражнение: возьмите задачу, которую вы умеете решать с применением массивов, но которая

ограничена размером массива. Перепишите код, сняв ограничение с помощью динамических массивов.

4.2. Создайте для нашей динамической строки функцию `substring`, которая получает три параметра: массив `arrayString`, начальную позицию в числовом формате и количество элементов подстроки, также в числовом формате. Функция возвращает указатель на вновь созданный динамический массив. Этот строковый массив содержит символы из первоначальной строки, начиная с указанной позиции и указанной длины. Первоначальная строка остается неизменной. Так что, если изначально строка представляла собой `abcdefg`, позиция была 3, а длина 4, то новая строка будет содержать `cdef`.

4.3. Создайте для нашей динамической строки функцию `replaceString`, которая получает три параметра в формате `arrayString: source, target` и `replaceText`. Функция заменяет каждое появление строки `target` в строке `source` на строку `replaceText`. Например, если `source` указывает на массив, содержащий `abcdabee`, `target` указывает на `ab`, а `replaceText` — на `xyz`, то после завершения работы функции `source` будет указывать на массив, содержащий `xyzcdxyzee`.

4.4. Измените реализацию нашей строки таким образом, чтобы элемент массива `location[0]` содержал размер массива (и, соответственно, элемент массива `location[1]` содержал первый хранящийся в массиве символ) и уже не пришлось бы использовать нулевой завершающий байт. Реализуйте каждую из трех функций `append`, `concatenate` и `charactertAt`, используя, насколько это возможно, преимущество сохраненной информации о размере массива. Так как мы больше не будем использовать соглашение о нулевом завершающем байте, которое необходимо для работы со стандартным потоком вывода, то вам необходимо написать свою собственную функцию `output`, которая с помощью цикла переберет строковый параметр и выведет на экран символы.

4.5. Реализуйте функцию `removeRecord`, которая принимает в качестве параметров указатель на коллекцию `studentCollection` и номер студента, а затем удаляет учетную карточку с этим студенческим номером из коллекции.

4.6. Напишите реализацию строковой переменной, которая использует связный список для хранения символов вместо динамического массива. То есть у нас будет связный список, в котором в качестве полезных данных будут храниться одиночные символы. Такая реализация позволит увеличивать размер строки без создания нового массива. Вам стоит начать с реализации функций `append` и `charactertAt`.

4.7. В продолжение предыдущего упражнения реализуйте функцию `concatenate`. Не забудьте, что, если мы вызываем функцию

`concatenate(s1, s2)`, где оба параметра — это указатели на первые узлы соответствующих связных списков, функция должна создавать копию каждого узла из списка `s2` и добавлять ее в конец списка `s1`, а не просто помещать указатель на первый узел списка `s2` в поле `next` последнего узла списка `s1`.

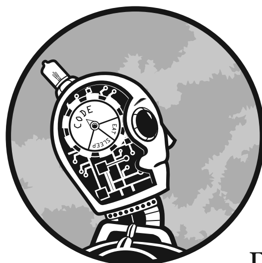
4.8. Добавьте в связный список, реализующий строковую переменную, функцию `removeChars`, которая получает в качестве параметров позицию и длину, а затем удаляем подстроку символов из первоначальной строки. Например, при вызове функции `removeChars(s1, 5, 3)` из строки будут удалены три символа, начиная с пятой позиции. Удостоверьтесь, что удаляемые узлы были надлежащим образом удалены из памяти.

4.9. Представьте, что существует связный список, в узлах которого вместо символов хранятся цифры от 0 до 9 в формате `int`. Используя подобный связный список, мы можем реализовать положительные числа любого размера. Число 149, например, будет представлено в виде связного списка, в первом узле которого будет храниться цифра 1, во втором – 4, а в третьем и последнем — 9. Напишите функцию `intToList`, которая принимает в качестве параметра число и создает подобный связный список. Подсказка: возможно, вам покажется легче строить связный список в обратном порядке, например, если число 149, то в первом узле хранится 9.

4.10. Создайте для списка с цифрами из предыдущего упражнения функцию, которая принимает в качестве параметров два таких списка и возвращает новый список, представляющий сумму их чисел.

5

Решение задач с классами



В этой главе мы обсудим классы и объектно ориентированное программирование. Как и раньше, я предполагаю, что вы уже знакомы с объявлением классов в языке C++ и понимаете принципы синтаксиса при создании класса, вызовах методов класса и так далее. Я проведу небольшой обзор в следующем разделе, но в основном обсуждение коснется проблем решения задач в классах.

Это другой аспект, в котором, я считаю, язык C++ выигрывает по сравнению с другими языками. Дело в том, что C++ гибридный язык и программист на C++ может создавать классы там, где это необходимо, но совершенно не обязан этого делать. В отличие от языков Java или

C#, весь код которых должен быть заключен в объявление класса. В руках опытного программиста это не вызывает больших проблем, но в руках новичка может привести к плохим привычкам. Для программиста на Java или C# все является объектом. Хотя весь код, написанный на этих языках, должен быть инкапсулирован в объекты, результат не всегда отражает разумное объектно ориентированное проектирование. Объект должен быть осмысленным и тесно связывать набор данных и код, который работает с этими данными. Он не должен напоминать капризную шляпу фокусника, наполненную объедками. Поскольку мы программируем на языке C++ и, следовательно, имеем возможность выбора между процедурным и объектно ориентированным программированием, то поговорим о хорошем проектировании классов, а также о том, когда следует использовать классы, а когда нет. Для достижения высокого уровня программирования существенен навык распознавания ситуаций, при которых классы полезны. Однако не менее важно уметь распознавать ситуации, когда классы все только портят.

Обзор основных свойств классов

Как всегда, я предполагаю, что вы знакомы с основами и правилами синтаксиса языка C++, тем не менее, давайте рассмотрим основы синтаксиса классов, чтобы мы использовали одинаковую терминологию. *Класс* – схема для создания конкретного набора кода и данных. Каждая переменная, созданная согласно схеме класса, называется *объектом* этого класса. Находящийся вне класса код, который создает и использует объект класса, называется *клиент* класса. *Объявление класса* именует класс и содержит список всех *членов* или элементов, находящихся внутри класса. Каждый элемент является либо *полем класса* – переменной, объявленной внутри класса, — либо *методом* (также известным как *функция класса*), то есть функцией, объявленной внутри класса. Среди функций класса встречается специальный тип, называемый *конструктор* — эта функция имеет то же имя, что и класс, и вызывается неявно при объявлении нового объекта класса. Помимо обычных атрибутов объявления переменной или функции (таких как тип и, для функции, список параметров) у каждого члена класса есть *спецификатор доступа*, определяющий, какие функции имеют доступ к этому члену. *Публичный член* доступен любому коду, использующему этот объект: коду внутри класса, клиенту класса или коду в подклассе, то есть в классе, который «наследует» весь код и данные существующего класса. *Приватный член* доступен только коду внутри класса. *Защищенные члены*, которые будут кратко рассмотрены в этой главе, похожи на приватные с единственным отличием, что методы в подклассах могут также обращаться к ним. Однако как приватные, так и защищенные члены недоступны из клиентского кода.

В отличие от других атрибутов, например типа возвращаемого значения, спецификатор доступа внутри объявления класса действует до тех пор, пока не будет заменен другим спецификатором. Таким образом, обычно каждый спецификатор используется только один

раз, а члены класса группируются по принципу доступа. Поэтому программисты используют термины «публичная секция» или «приватная секция» класса, например, так: «Мы должны поместить этот метод в приватной секции».

Давайте взглянем на крошечный пример объявления класса:

```
  class ❶sample {
❷ public:
     ❸ sample();
     ❹ sample(int num);
     ❺ int doesSomething(double param);
  private:
     ❻ int intData;
  }❼;
```

Это объявление начинается с имени класса ❶ и после этого sample становится именем типа. Объявление начинается с public спецификатора доступа ❷, и, таким образом, пока мы не достигнем private спецификатора ❻[1], все является публичным. Многие программисты ставят публичную секцию в начало, подразумевая, что публичный интерфейс более интересен другим пользователям. В данном случае публичными объявлениями являются два конструктора с названием sample (❸ и ❹) и метод doesSomething ❺. Эти конструкторы неявно вызываются при объявлении объекта этого класса.

```
sample object1;
sample object2(15);
```

Здесь объект object1 вызовет первый конструктор ❸, называемый *конструктор по умолчанию.* У этого конструктора нет параметров. А объект object2 вызовет второй конструктор ❹, поскольку он содержит одно целое значение и, таким образом, соответствует параметрам сигнатуры второго конструктора.

Объявление заканчивается приватным полем класса intData ❻. Помните, что объявление класса заканчивается закрывающей скобкой и точкой с запятой ❼. Возможно, эта точка с запятой слегка озадачивает, поскольку мы не ставим их после функций, блоков инструкций if или любых других закрывающихся скобок. В действительности наличие точки с запятой показывает, что объявление класса может быть объявлением объекта. Мы могли бы поместить идентификаторы между закрывающей скобкой и точкой с запятой и создать объекты так же, как создаем классы. Однако это не слишком распространенная практика в языке C++, особенно учитывая тот факт, что большинство программистов размещают объявления классов в файлы, отдельные от программ, которые их используют. Также точка с запятой появляется после закрывающей скобки при использовании ключевого слова struct.

Говоря о ключевом слове struct, вы должны знать, что в языке C++ слова struct и class обозначают примерно одно и то же. Един-

[1] По аналогии следовало бы поставить (6) перед словом private *(примеч. пер.).*

ственное отличие касается членов (данных или методов), объявленных до первого спецификатора доступа. В struct эти члены будут публичными, а в class они будут приватными. Однако хорошие программисты используют эти две структуры разными способами. Аналогичным образом любой цикл for может быть написан как цикл while, однако хороший программист может сделать код более читаемым, используя циклы for для большинства циклов, считываемых вперед. Большинство программистов приберегают struct для простых структур, в которых либо отсутствуют поля класса кроме конструкторов, либо в тех, которые планируется использовать в качестве параметров в методах больших классов.

Цели использования классов

Чтобы распознавать правильные и неправильные ситуации для использования классов, а также правильные и неправильные способы создания классов, мы сначала должны определить цели использования классов. Говоря об этом, мы должны помнить, что использование классов необязательно. То есть классы не предоставляют нам новые возможности, как, например, массивы или структуры, основанные на указателях. Если вам нужна программа, которая использует массив для сортировки 10 000 записей, ее нельзя написать без массивов. Если вам нужна программа, которая, основываясь на возможностях связного списка, со временем растет или сокращается, вы не сможете добиться такого эффекта, не используя связные списки или аналогичные, основанные на указателях структуры. Однако если вы уберете классы из объектно ориентированной программы и перепишете ее, программа будет выглядеть иначе, но ее возможности и эффективность не уменьшатся. В самом деле, ранние компиляторы языка C++ работали как предпроцессоры. Компилятор C++ прочитывает исходный код языка C++ и выдает на лету новый код, соответствующий синтаксису языка C. Этот модифицированный исходный код передается компилятору C. Таким образом, основное дополнение, которое имеет язык C++ по сравнению с языком C, связано не с функциональными возможностями языка, а с тем, как исходный код читается программистом.

Следовательно, выбирая основную цель проектирования классов, мы выбираем цель помощи нам, программистам, выполнять наши задачи. В частности, поскольку эта книга посвящена решению задач, мы будем думать, как классы помогают решать эти задачи.

Инкапсуляция

Слово *инкапсуляция* — это причудливый способ сказать, что классы соединяют многочисленные части данных и кода в единый пакет. Если вы когда-либо видели желатиновую медицинскую капсулу, наполненную маленькими шариками, то это хорошая аналогия: пациент принимает одну капсулу и проглатывает все отдельные ингредиенты, заключенные в ней.

Инкапсуляция — это механизм, позволяющий достичь большинства обозначенных выше целей, но при этом он имеет собственное достоинство, состоящее в организации кода. В листингах длинных программ чистого процедурного кода (в языке C++ это означает код с функциями, но без классов) бывает очень сложно определить верный порядок для функций и директив компилятору, позволяющий легко запомнить их расположение. Вместо этого мы вынуждены полагаться на среду разработки, которая будет искать наши функции для нас. Инкапсуляция объединяет те элементы, которые используются вместе. Если вы работаете с методом класса и понимаете, что вам надо посмотреть или модифицировать другой код, скорее всего, этот другой код встретится в другом методе этого же класса, и, следовательно, он где-то рядом.

Повторное использование кода

С точки зрения решения задач, инкапсуляция позволяет нам легче использовать код предыдущей задачи для решения текущей. Часто, даже если мы уже решили задачу, сходную с нашим текущим проектом, повторное использование кода, как мы уже узнали, все-таки требует много усилий. Полностью инкапсулированный класс может работать подобно внешнему USB-носителю. Вы просто вставляете его, и он работает. Однако для того, чтобы это произошло, мы должны корректно спроектировать класс и убедиться, что код и данные на самом деле инкапсулированы и настолько независимы от чего-либо за пределами класса, насколько это возможно. Например, класс, который использует глобальную переменную, нельзя скопировать в другой проект без копирования этой глобальной переменной. Помимо повторного использования классов в разных программах, классы часто предоставляют более непосредственную форму повторного использования: наследование. Вспомните, как в главе 4 мы говорили об использовании вспомогательных функций для «выделения» кода, доступного двум или более функциям. Наследование использует эту идею в большем масштабе. Используя наследование, мы создаем родительские классы с методами, доступными для двух или большего количества дочерних классов, таким образом, «выделяя» не просто несколько строк кода, а целые методы. Наследование — большая тема сама по себе, и мы рассмотрим эту форму повторного использования кода позднее в этой главе.

Разделение задачи

Один способ, к которому мы будем возвращаться снова и снова, состоит в делении сложной задачи на маленькие фрагменты, с которыми проще работать. Классы отлично подходят для деления программ на функциональные элементы. Инкапсуляция не только хранит вместе данные и код в пакете, который можно использовать неоднократно; также она отделяет эти данные и код от остальной программы, позволяя нам работать в этом классе отдельно от чего-то еще. Чем больше классов мы создадим в программе, тем сильнее эффект разделения задачи.

Таким образом, там, где возможно, мы разрешим классу быть нашим методом разделения сложной задачи. Если классы хорошо спроектированы, это усилит функциональное разделение, и задачу можно будет решить проще. Как побочный результат, возможно, мы обнаружим, что классы, созданные нами для решения одной задачи, могут использоваться в других, даже если мы полностью не рассматривали такую возможность, при их создании.

Сокрытие

Некоторые программисты используют термины *сокрытие* и *инкапсуляция* как синонимы, однако здесь мы их разделим. Как было показано ранее в этой главе, инкапсуляция — это совместная упаковка данных и кода. Сокрытие означает отделение интерфейса структуры данных — определение процессов и их параметров — от реализации структуры данных или кода внутри функций. Если сокрытие являлось целью при написании класса, тогда возможно изменение реализации методов, без изменения в клиентском коде (код, который использует этот класс). И снова мы должны четко определить термин *интерфейс*. Он означает не только название методов и список их параметров, но и также объяснение (возможно отраженное в документации), что делают разные методы. Говоря об изменении реализации без изменения интерфейса, мы имеем в виду, что меняем *как* методы класса работают, но не *что* они делают. Некоторые авторы книг по программированию относятся к этому, как к своего рода неявному соглашению между классом и клиентом: класс соглашается никогда не менять результаты существующих процессов, а клиент соглашается использовать классы строго на основе их интерфейса и игнорировать детали реализации. Предположим, существует универсальный пульт, с помощью которого можно управлять любым телевизором, начиная от старых моделей с ЭЛТ-экраном и заканчивая моделями с жидкокристаллическим или плазменным экраном. Вы нажимаете 2, затем 5, потом «ввод» и любой экран покажет двадцать пятый канал, хотя механизм, который позволит этому произойти, очень сильно отличается в зависимости от технологии, лежащей в его основе.

Невозможно сокрытие без инкапсуляции, однако в контексте определенной нами терминологии возможна инкапсуляция без сокрытия. Самый очевидный способ сделать это — объявить все поля класса как `public`. В этом случае класс по-прежнему инкапсулирован, так как это пакет, в котором соединены вместе код и данные. Однако сейчас клиентский код имеет доступ к важным деталям реализации класса: переменным и типам, которые класс использует для хранения данных. Даже если клиентский код непосредственно не изменяет данные класса, а только изучает их, клиентский код требует особую реализацию класса. Любые изменения в классе, которые меняют имя или тип любой переменной доступной клиентскому коду, требуют соответствующих изменений в клиентском коде.

Возможно, первое, что вы подумали, что сокрытие гарантировано, пока все данные реализованы как приватные, а мы потратили достаточно времени на проектирование списка функций класса и списка их параметров так, что нам никогда не потребуется их менять. Хотя все это требуется для сокрытия, этого не достаточно, потому что проблема сокрытия более тонкая. Помните, что по устоявшейся договоренности класс не может менять действия любого метода, независимо от ситуации. В предыдущих главах мы решали задачу, в которой функция обрабатывала пустой список, а также задачу с нестандартной ситуацией, как, например, поиск среднего значения в массиве, для которого параметр, содержащий размер массива, равнялся нулю. Изменение результата метода даже для странных случаев представляет собой изменение интерфейса. Этого следует избегать. Это еще одна причина, почему в программировании важно явно рассматривать специальные случаи. Многие программы перестают работать, когда обновляется лежащая в их основе технология или программный интерфейс приложения (API), и некоторые системные вызовы, которые надежно возвращали –1, в случаях, когда один из параметров был ошибочным, теперь возвращают вроде бы случайную отрицательную величину. Чтобы избежать такой проблемы, одним из наилучших способов является определение результатов специальных методов в документации метода или класса. Если ваша собственная документация говорит, что вы возвращаете код ошибки –1 в случае особых ситуаций, следует подумать дважды о возможном возврате чего-то еще.

Итак, как же сокрытие влияет на решение задач? Принципы сокрытия говорят программисту отложить в сторону детали реализации класса при работе над клиентским кодом или, в более широком смысле, заниматься конкретной реализацией класса только внутри класса. Если вы можете выкинуть из головы детали реализации, вы можете избавиться от отвлекающих мыслей и сконцентрироваться на решении текущих задач.

Однако мы должны осознавать ограничения сокрытия при решении задач. Иногда детали реализации все-таки имеют значение для клиента. В предыдущих главах мы видели преимущества и недостатки некоторых структур, основанных на массивах или указателях. Структуры, основанные на массивах, допускают случайный доступ, но не могут легко увеличиваться и уменьшаться, в то время как структуры, основанные на указателях, допускают только последовательный доступ, но позволяют добавление или удаление фрагментов без повторного создания всей структуры. Следовательно, класс, созданный на основе структуры, основанной на массиве, имеет качественные различия с классом, построенном на структуре, основанной на указателях.

В информатике мы часто говорим о концепции *абстрактного типа данных*, что является сокрытием в самом чистом виде: тип данных определяется только во время работы. В главе 4 мы обсуждали концепцию стека и описали стек программы как смежные блоки

памяти. Однако как абстрактный тип данных, стек, это любой тип данных, где вы можете добавить и удалить отдельный элемент, при этом элементы удаляются в обратном порядке по сравнению с тем, в котором они добавлялись. Этот порядок известен как «последним пришел — первым ушел», или LIFO. Ничто не заставляет стек быть смежными блоками памяти, мы можем реализовать стек, используя связный список. Поскольку смежные блоки памяти и связный список имеют разные свойства, у стека, имеющего одну или другую реализации, также будут разные свойства, и это может привести к большим различиям для клиента, использующего этот стек.

Смысл всего этого в том, что сокрытие будет для нас полезной целью при решении задач в той мере, в которой оно позволяет разделить задачу и работать отдельно с различными частями программы. Однако мы не можем позволить себе полностью игнорировать детали реализации.

Читабельность

Хороший класс повышает читабельность программы. Объекты могут соответствовать тому, как мы их видим в реальном мире, и, следовательно, вызов метода часто имеет англоязычную читабельность. Кроме того, взаимоотношения между объектами часто понятнее, чем взаимоотношения между отдельными переменными. Повышение читабельности увеличивает возможности решения задач, поскольку мы можем лучше понять свой собственный код во время разработки и потому что повторное использование встречается чаще, когда понятно, как работать со старым кодом.

Чтобы максимизировать преимущества хорошей читабельности классов, мы должны подумать о том, как методы нашего класса будут использоваться на практике. Имена методов надо выбирать, думая об отражении наиболее точного смысла результатов метода. Например, рассмотрим класс, осуществляющий расчеты финансовых инвестиций, в котором содержится метод расчета будущей стоимости. Имя `compute` сообщает намного меньше информации, чем `computeFutureValue`. Даже выбор правильной части речи для имени может быть важным. Имя `computeFutureValue` — это глагол, а `futureValue` — существительное. Посмотрите, как используются имена в следующем примере кода.

```
  double FV;
❶ investment.computeFutureValue(FV, 2050);

❷ if (investment.futureValue(2050) > 10000) { ...
```

Если вы задумаетесь, то обнаружите, что первый вариант имеет смысл для самостоятельного вызова, то есть функция, возвращающая `void`, в которой будущая стоимость возвращается к вызвавшему ее клиенту с помощью ссылочного параметра ❶. Последний вариант имеет больше смысла для вызова, который используется в выражении, то есть будущая стоимость возвращается как значение функции ❷.

Позднее в этой главе мы увидим специфические примеры этого, однако основной принцип максимизации читабельности — это при написании любой части интерфейса класса постоянно думать о клиентском коде.

Выразительность

Конечная цель хорошо спроектированного класса состоит в выразительности или в том, что в широком смысле может быть названо «писабельностью» — легкостью, с которой код пишется. Однажды написанный хороший класс облегчает написание остального кода так же, как делает это хорошая функция. Классы эффективно расширяют язык, становясь высокоуровневыми партнерами для основных низкоуровневых инструментов, таких как циклы, инструкции `if` и так далее. В языке C++ даже основной функционал, такой как ввод и вывод, не является неотъемлемыми частями языка, а реализован как набор классов, которые необходимо явно включать в программу для их использования. При помощи классов программные действия, которые раньше требовали большого количества этапов, могут быть реализованы за несколько или вообще за один шаг. Решая задачи, мы должны присвоить этой цели особый приоритет. Мы всегда должны думать: «Как этот класс позволит облегчить написание остальной программы или будущих программ, которые смогут его использовать?»

Создание простого класса

Теперь, когда мы знаем, какие цели наш класс должен преследовать, наступило время применить теорию на практике и создать некоторый класс. Прежде всего, мы разработаем наш класс для решения следующей задачи.

Задача: список класса

Спроектируем класс или набор классов для использования в программе, которая поддерживает список класса. Для каждого студента будем хранить его фамилию, идентификатор и результат выпускного экзамена в числовом диапазоне от 0 до 100. Программа должна позволять добавлять и удалять записи, отображать записи по конкретному студенту, идентифицируемому по ID с оценкой, представленной в числовом или буквенном формате, и отображать среднюю оценку класса. Соответствующая буквенная оценка для каждого результата представлена в табл. 5.1.

Табл. 5.1. Буквенные оценки

Диапазон результатов	Буквенная оценка
93–100	A
90–92	A–
87–89	B+
83–86	B
80–82	B–

77–79	C+
73–76	C
70–72	C–
67–69	D+
60–66	D
0–59	F

Мы начнем с рассмотрения базового фреймворка класса, который составляет основу для большинства классов. Затем мы рассмотрим пути расширения базового фреймворка.

Базовый фреймворк класса

Лучше всего изучать базовый фреймворк класса с помощью класса-примера. В нашем случае мы начнем со структуры студентов из главы 3 и разовьем ее в полноценный класс. Для удобства повторим исходную структуру:

```
struct student {
   int grade;
   int studentID;
   string name;
};
```

Даже в такой простой структуре у нас есть инкапсуляция. Вспомним, что в главе 3 мы создавали массив данных о студентах в виде структуры `struct` и без использования `struct` нам пришлось создать три параллельных массива, один для оценок, второй для ID и третий для имен — ужас! Однако с использованием `struct` мы не достигаем сокрытия. Базовый фреймворк класса дает нам сокрытие, объявляя все данные приватными и затем добавляя публичные методы, с помощью которых клиентский код может опосредованно получать или менять эти данные.

```
   class studentRecord {
❶ public:
     ❷ studentRecord();
        studentRecord(int newGrade, int newID, string newName);
     ❸ int grade();
     ❹ void setGrade(int newGrade);
        int studentID();
        void setStudentID(int newID);
        string name();
        void setName(string newName);
❺private:
     ❻int _grade;
       int _studentID;
       string _name;
   };
```

Как договаривались, это объявление класса разделено на публичную секцию с функциями класса ❶ и приватную секцию ❺, которая содержит те же данные, что и исходная `struct` ❻. В классе восемь функций: два конструктора ❷ и пара функций класса для каждого поля класса. Например, у поля `_grade` есть две связанных функции

класса, `grade` ❸ и `setGrade` ❹. Первый из этих методов будет использоваться клиентским кодом для получения оценки для конкретной `studentRecord`, а второй — для установки новой оценки для конкретной `studentRecord`.

Методы получения и установки, связанные с полем класса, настолько распространены, что их обычно называют краткими терминами `get` и `set` (геттер и сеттер). Как вы видите, я включил слово `set` в названия методов, которые устанавливают новое значение для поля. Многие программисты также включили бы слово `get` в другие названия, например, `getGrade` вместо `grade`. Почему я так не сделал? Потому что в этом случае я бы использовал бы глагол для функции, использующейся как существительное. Однако некоторые могли бы возразить, что термин `get` настолько общепринят и, следовательно, его значение настолько очевидно, что его использование перевешивает прочие доводы. В конечном счете, это дело личного стиля.

Хотя я и указывал в этой книге на преимущества C++ по сравнению с другими языками, я должен признать, что более современные языки, такие как C#, превосходят C++ в том, что касается методов `get` и `set`. В языке C# есть встроенный механизм, называемый свойством, который действует и как метод `get`, и как `set`. После определения клиентский код получает доступ к свойствам, как если бы они были полями класса, а не через вызов функции. Это большой шаг в увеличении читабельности и выразительности. Поскольку в языке C++ такой механизм отсутствует, важно определить некоторую конвенцию наименований и постоянно ее придерживаться.

Обратите внимание, что моя конвенция наименований распространяется и на поля класса, которые в отличие от исходной `struct` все начинаются с подчеркивания. Это позволяет мне называть функции `get` (почти) также как и поля класса, значения которых они получают. Кроме того, это позволяет легко распознавать обращения к полям класса в коде, повышая тем самым его читабельность. Некоторые программисты используют ключевое слово `this` для обращения ко всем полям класса вместо подчеркивания. Так вместо инструкции:

```
return _grade;
```

у них будет:

```
return this.grade;
```

Возможно, вы не встречались с ключевым словом `this` ранее. Это ссылка на объект, в котором оно появляется. То есть если вышеприведенное строка присутствует в методе класса, а в этом методе объявлена локальная переменная с именем `grade`, выражение `this.grade` будет относиться к полю класса `grade`, а не к локальной переменной с тем же именем. Использование этого ключевого слова таким образом дает особое преимущество в средах программирования с автоматическим дополнение синтаксиса: программист может набрать только

this, нажать точку, и выбрать поле класса из списка, избежав дополнительного набора и возможных опечаток. Однако в обоих вариантах выделяются поля класса, это то, что действительно важно.

Теперь, когда мы рассмотрели объявление класса, давайте посмотрим реализацию методов. Начнем с первой пары get/set.

```
int studentRecord::grade() {
    ❶ return _grade;
}
void studentRecord::setGrade(int newGrade) {
    ❷ _grade = newGrade;
}
```

Это самая простая форма пары get/set. Первый метод, grade, возвращает текущее значение соответствующего поля класса, _grade ❶. Второй метод, setGrade, присваивает полю класса _grade значение параметра newGrade ❷. Однако, если бы это было все, что делает наш класс, у нас бы ничего путного не вышло. Хотя этот код предусматривает сокрытие, поскольку передает данные в обоих направлениях без рассмотрения и модификации, он лучше, чем объявление переменной _grade публичной, поскольку оставляет нам возможность изменить имя или тип полю класса. Метод setGrade должен как минимум выполнить несколько примитивных проверок; необходимо удостовериться, что значение, получаемое полем класса _grade из переменной newGrade, имеет смысл как оценка. Однако надо быть осторожными, решая задачи спецификации, и не делать предположений о данных, базируясь только на своем опыте, без обсуждения с пользователем. Возможно, оценка варьируется от 0 до 100, а может и нет, если, например, в школе предусмотрены дополнительные баллы или используется оценка –1 для отчисленных. Поскольку в данном случае у нас есть описание ситуации, мы можем применить эти знания для проверки.

```
void studentRecord::setGrade(int newGrade) {
   if ((newGrade >= 0) && (newGrade <= 100))
      _grade = newGrade;
}
```

В данном случае проверка является только привратником. Однако, в зависимости от описания задачи, возможно, следует добавить в метод вывод сообщения об ошибке, запись в файл сообщения об ошибке или еще какой-то механизм управления ошибками.

Остальные пары get/set работают точно так же. Несомненно, существуют правила формирования номеров ID студентов в конкретной школе. Эти правила надо использовать при проверке. Однако при вводе имени студента лучше всего отклонять строки со странными символами, как % или @, но сейчас это вряд ли возможно.

Заключительный этап создания нашего класса — это написание конструкторов. В базовый фреймворк мы включаем два конструктора: конструктор по умолчанию, без параметров, который назначает

полям класса разумные значения по умолчанию, и конструктор с параметрами для каждого поля класса. Второй конструктор важен для нашей цели *выразительности*, поскольку позволяет одномоментно создать объект нашего класса и инициализировать его. После того как вы написали код для остальных методов, этот второй конструктор почти написался сам по себе.

```
studentRecord::studentRecord(int newGrade, int newID, string newName) {
   setGrade(newGrade);
   setStudentID(newID);
   setName(newName);
}
```

Как видите, конструктор просто вызывает соответствующие методы `set` для каждого параметра. В большинстве случаев это правильный подход, поскольку позволяет избежать дублирования кода и гарантирует, что конструктор использует преимущества проверок, сделанных в методах `set`.

Иногда конструктор по умолчанию бывает немного сложнее, но не из-за сложности кода, а потому что не всегда очевидно, какие именно должны быть значения по умолчанию. Выбирая значения по умолчанию для полей класса, помните о ситуациях, при которых будет использоваться конструктор по умолчанию и, особенно, будет ли объект по умолчанию легитимным для данного класса. Это подскажет вам, заполнять ли поля класса полезными значениями по умолчанию или значениями, сигнализирующими о том, что объект неверно инициирован. Для примера рассмотрим класс, представляющий коллекцию значений и инкапсулирующий связный список. В этом случае *существует* связный список по умолчанию, а именно пустой связный список. В этом случае мы устанавливаем в полях класса легитимный, но пустой, связный список. Однако в нашем примере с базовым классом, не существует правильного определения студента по умолчанию; мы не хотим давать верный номер ID объекту `studentRecord` по умолчанию, так его теоретически можно спутать с легитимным объектом `studentRecord`. Следовательно, мы должны выбрать очевидно нелегитимные значения по умолчанию для поля `_studentID`, например –1:

```
studentRecord::studentRecord() {
   setGrade(0);
   setStudentID(-1);
   setName("");
}
```

Мы задаем оценку с помощью метода `setGrade`, который устанавливает свой параметр. Это значит, что мы должны передать корректную оценку, в данном случае 0. Поскольку в поле номера ID установлено неверное значение, вся запись может быть легко идентифицирована как нелегитимная. Следовательно, корректное значение оценки не имеет значения. Если бы нам было бы важно, мы могли бы установить некорректное значение прямо в поле `_grade`.

Тем самым мы завершаем базовый фреймворк класса. Мы создали группу приватных полей класса, соответствующих разным атрибутам одного и того же логического объекта, в данном случае записи класса студентов. У нас есть функции класса для получения или изменения с соответствующими проверками данных объекта. И у нас есть набор полезных конструкторов. Мы создали хорошую основу класса. Однако возникает вопрос: нужно ли делать что-то еще?

Служебные методы

Служебный метод – это метод класса, который не получает и не устанавливает данные. Некоторые программисты называют их вспомогательными методами, методами-помощниками или как-то еще. Их можно назвать как угодно, однако именно они делают класс чем-то большим, чем базовый фреймворк класса. Часто именно хорошо спроектированный набор служебных методов делает класс поистине полезным.

Для определения возможных служебных методов, подумайте, как будет использоваться класс. Можем ли мы ожидать от клиентского кода совершения каких-то обычных действий с полями нашего класса? В нашем примере программа, для которой мы проектируем класс, выдает оценки студентов не только в цифровом, но и в буквенном формате. Тогда давайте создадим служебный метод, который возвращает оценку студента в виде буквы. Сначала добавим объявление метода в публичную секцию объявления класса.

```
string letterGrade();
```

Теперь нам надо реализовать метод. Функция будет преобразовывать цифровые значения, хранящиеся в поле `_grade`, в соответствующую переменную формата `string`, основываясь на таблице оценок, представленной в задаче. Мы могли бы справиться с этим серией условных инструкций `if`, но нет ли более чистого, более элегантного способа? Если вы сейчас подумали: «Эй, это ж похоже на то, как мы конвертировали доход в категории бизнес-лицензии в главе 3», то поздравляю — вы обнаружили подходящую программистскую аналогию. Мы можем адаптировать тот код с параллельными константными (`const`) массивами для хранения буквенных оценок и нижней границы цифровой оценки соответствующей этим буквам для конвертации цифровых оценок в цикле.

```
string studentRecord::letterGrade() {
   const int NUMBER_CATEGORIES = 11;
   const string GRADE_LETTER[] = {"F", "D", "D+", "C-", "C", "C+", "B-",
   "B", "B+", "A-", "A"};
   const int LOWEST_GRADE_SCORE[] = {0, 60, 67, 70, 73, 77, 80, 83, 87, 90,
   93};
   int category = 0;
   while (category < NUMBER_CATEGORIES && LOWEST_GRADE_SCORE[category] <=
   _grade)
      category++;
   return GRADE_LETTER[category - 1];
}
```

Этот метод — прямая адаптация функции из главы 3, поэтому здесь нет ничего нового, что требовалось бы объяснить. Однако адаптация в метод класса все же демонстрирует некоторые проектные решения. Первое, на что можно обратить внимание, состоит в том, что мы не создали нового поля класса для хранения буквенных оценок, но определяем соответствующую буквенную оценку на лету в ответ на каждый запрос. Альтернативный подход предполагает создание поле класса `_letterGrade` и изменение метода `setGrade` для обновления поля `_letterGrade` вместе с `_grade`. Тогда метод `letterGrade` стал бы простым методом `get`, возвращающим значение уже определенного поля класса.

Причины такого подхода лежат в *избыточности данных*. Этот термин описывает ситуацию, когда хранимые данные либо являются в буквальном смысле дубликатом других данных, либо могут быть прямо получены из других данных. Эта проблема обычно возникает в базах данных, и их проектировщики стараются избежать создания избыточных данных в своих таблицах. Однако при нашей неосторожности избыточность данных может появиться в любой программе. Рассмотрим программу, хранящую медицинские записи о возрасте и дате рождения пациентов. Дата рождения пациента несет в себе информацию, которой нет в данных о возрасте. Эти два набора данных не тождественны, но возраст не говорит нам ничего, что мы не могли бы получить из даты рождения. А что если эти два параметра не согласуются между собой (что может, в конечном счете, произойти, если возраст не рассчитывается автоматически)? Какому значению мы верим? Мне вспоминается знаменитое (хотя вероятно апокрифическое) высказывание халифа Умара, произнесенное при приказе сжечь Александрийскую библиотеку. Он воскликнул, что если книги в библиотеке согласны с Кораном, то они избыточны и их не стоит хранить, но если они не согласны с Кораном, они пагубные и их следует уничтожить. Избыточность данных – это ожидаемая проблема. Единственным оправданием может быть производительность, когда мы полагаем, что изменения поля `_grade` будут редки, а вызовы `letterGrade` часты, однако тяжело представить общее существенное увеличение производительности программы.

Однако этот метод можно улучшить. Тестируя его, я заметил проблему. Хотя метод выдает верные результаты для корректных значений поля `_grade`, метод перестает работать в случае отрицательного значения переменной `_grade`. При выполнении цикла `while` отрицательное значение поля `_grade` немедленно приводит к ошибке. Следовательно, переменная `category` остается нулевой и выражение `return` пытается обратиться к `GRADE_LETTER[ -1]`. Мы могли бы избежать этой проблемы, инициализировав переменную `category` единицей вместо нуля, но это означало бы, что отрицательная отметка связана с «F», при том что она не должна быть связана ни с какой строкой, поскольку некорректное значение оценки не подходит никакой категории.

Вместо этого мы могли бы проверить поле _grade до преобразования его в буквенную оценку. Мы уже проверяем значение оценки в методе setGrade, так что вместо добавления нового проверочного кода в метод letterGrade, нам следует «выявить» общую часть кода в этих методах и сделать третий метод. (Возможно, вы удивитесь, откуда может взяться некорректная оценка, если мы проверяем оценки при их установке. Однако мы могла решить устанавливать некорректную оценку в конструкторе как сигнал об отсутствии легитимной оценки.) Это еще один служебный метод, являющийся эквивалентом на уровне класса концепции общей вспомогательной функции, предложенной в предыдущей главе. Давайте реализуем этот метод и исправим другие наши методы:

```
❶bool studentRecord::❷isValidGrade(❸int grade) {
     if ((grade >= 0) && (grade <= 100))
         return true;
     else
         return false;
  }
  void studentRecord::setGrade(int newGrade) {
     if (❹isValidGrade(newGrade))
         _grade = newGrade;
  }
  string studentRecord::letterGrade() {
     if (❺!isValidGrade(_grade)) return "ERROR";
     const int NUMBER_CATEGORIES = 11;
     const string GRADE_LETTER[] = {"F", "D", "D+", "C-", "C", "C+", "B-",
                                     "B", "B+", "A-", "A"};
     const int LOWEST_GRADE_SCORE[] = {0, 60, 67, 70, 73, 77, 80, 83, 87,
                                        90, 93};
     int category = 0;
     while (category < NUMBER_CATEGORIES && LOWEST_GRADE_SCORE[category]
     <= _grade)
         category++;
     return GRADE_LETTER[category - 1];
  }
```

Тип нового метода проверки оценок bool ❶ и, поскольку его значение да-или-нет, я выбрал название isValidGrade ❷. Тем самым достигается наиболее англоязычное чтение вызовов этого метода, такое как в методах setGrade ❹ и letterGrade ❺. Также обратите внимание, что метод принимает оценку для проверки как параметр ❸. Хотя letterGrade уже проводит проверку значения в поле класса _grade, setGrade проверяет значение, которое может быть установлено в поле класса. Таким образом, чтобы быть полезным обоим методам, метод isValidGrade должен принимать оценку в качестве параметра.

Хотя метод isValidGrade уже реализован, один вопрос по его поводу остался: какой у него должен быть уровень доступа? То есть должны ли мы поместить его в публичную или приватную секцию класса? В отличие от методов set и get базового фреймворка класса, которые всегда помещаются в публичной секции, служебные методы могут быть в зависимости от их использования, как публичными, так и приватными. Чего мы добьемся, объявив метод isValidGrade публичным? Очевидно,

клиентский код получит доступ к этому методу. Поскольку кажется, что, чем больше в классе публичных методов, тем он полезнее, многие начинающие программисты объявляют публичными все методы, которые могут использоваться клиентом. Однако такой подход игнорирует обратную сторону назначения публичного доступа. Помните, что публичная секция определяет интерфейс нашего класса и мы не сможем изменить метод после того, как класс будет встроен в одну или несколько программ, поскольку с большой вероятностью такие изменения потребуют имений во всем клиентском коде. Таким образом, размещение метода в публичной секции фиксирует интерфейс метода и его результаты. Предположим, в нашем случае, что некоторый клиентский код, основываясь на исходном коде в методе `isValidGrade`, ориентируется на диапазон оценок от 0 до 100, но впоследствии правила выставления оценок становятся более сложными. Клиентский код в этом случае может перестать работать. Чтобы избежать этого, нам, видимо, придется создать второй проверочный метод внутри класса, не трогая первый.

Предположим, мы планируем, что метод `isValidGrade` не будет широко использоваться клиентом, и решаем не делать его публичным. Мы могли бы объявить его приватным, но это не единственный вариант. Поскольку функция не ссылается непосредственно на поля класса или другие методы класса, мы могли бы объявить функцию вообще вне класса. Однако в данном случае мы сталкиваемся не только с такими ограничениями, которые публичный доступ налагает на изменяемость, но также снижаем инкапсуляцию, потому что теперь эта функция, требуемая классом, не является его частью. Также мы могли бы оставить метод в классе, но сделать его *защищенным* вместо приватного. Разница будет видна в подклассе. Если метод `isValidGrade` защищенный, его смогут вызывать методы подклассов; если метод `isValidGrade` приватный, его могут использовать только другие методы класса `studentRecord`. Это та же дилемма выбора между публичным и приватным доступом, но в меньшем масштабе. Ожидаем ли мы, что наш метод будет очень полезен методам в подклассе, и ожидаем ли мы, что его результаты или интерфейс могут измениться в будущем? В большинстве случаев наиболее безопасный вариант — объявить все служебные методы приватными и оставить публичным только те служебные методы, которые написаны для клиента.

Классы с динамическими данными

Одна из важнейших причин создания класса лежит в инкапсуляции структур с динамическими данными. Как мы обсудили в главе 4, программисты сталкиваются с реальной проблемой отслеживания динамического выделения памяти, назначения указателей и освобождения памяти для того, чтобы мы могли избежать утечек памяти, висячих ссылок и ссылок на неверную память. Упаковка всех ссылок на указатели в класс не устраняет кропотливую работу, но сделав ее один раз хорошо, мы можем безопасно вставлять этот код в другие

проекты. Также это означает, что все проблемы, связанные со структурами с динамическими данными, изолированы внутри класса, тем самым облегчая отладку.

Давайте создадим класс с динамическими данными и посмотрим, как это работает. Для нашей задачи мы используем модифицированную версию основной задачи главы 4.

Задача: отслеживание неизвестного количества записей студентов

В этой задаче вы создадите класс с методами для хранения и манипулирования коллекцией записей студентов. Запись студента содержит номер студента, оценку, оба параметра целочисленные, и фамилию студента (строковая переменная). Необходимо реализовать следующие функции:

addRecord Этот метод принимает номер, фамилию и оценку студента и добавляет новую запись с этими данными в коллекцию.

recordWithNumber Эта функция принимает номер студента и возвращает из коллекции запись для студента с этим номером.

removeRecord Эта функция принимает номер студента и удаляет из коллекции запись студента с этим номером.

Коллекция может быть любого размера. Ожидается, что операция **addRecord** будет выполняться часто, потому она должна быть реализована эффективно.

Основное отличие между этим описанием и исходной версией в том, что мы добавили новую операцию, `recordWithNumber`, и таким образом ни одна из операций не требует в качестве параметра указатель. Это главный выигрыш от использования класса для инкапсуляции связного списка. Клиент может знать, что класс реализует коллекцию записей о студентах в виде связного списка и может даже учитывать это (помните нашу дискуссию об ограничениях сокрытия). Однако клиентский код непосредственно не взаимодействует ни со связным списком, ни с каким-либо указателем в классе.

Поскольку в этой задаче необходимо хранение той же информации, как и в предыдущей, у нас есть возможность использовать класс еще раз. В нашем связном списке узлового типа вместо отдельных полей для каждой из частей информации о студенте, мы используем один объект `studentRecord`. Использование объекта одного класса в качестве типа данных во втором классе называется *композиция*.

У нас достаточно информации для предварительного объявления класса:

```
class studentCollection {
private:
  ❶ struct studentNode {
    ❷ studentRecord studentData;
      studentNode * next;
    };
❸ public:
    studentCollection();
```

```
      void addRecord(studentRecord newStudent);
      studentRecord recordWithNumber(int idNum);
      void removeRecord(int idNum);
   private:
      ❹ typedef studentNode * studentList;
      ❺ studentList _listHead;
   };
```

Ранее я говорил о тенденции программистов начинать классы с публичных объявлений, однако здесь мы сделаем исключение. Начнем с приватного объявления узла `struct studentNode` ❶, который будем использовать для создания связного списка. Это объявление должно быть раньше публичной секции, поскольку некоторые публичные функции класса ссылаются на этот тип. В отличие от узла в главе 4, у этого узла нет индивидуальных полей для полезных данных, но он включает в себя члена класса `studentRecord` ❷. Публичные функции класса следуют напрямую из условия задачи. Кроме того, у нас, как всегда, есть конструктор. Во второй приватной секции мы для ясности объявили псевдоним `typedef` ❹ для указателя на наш узел, так как мы делали в главе 4. Затем мы объявили указатель на начало списка, разумно названный `_listHead` ❺.

Этот класс объявляет два приватных типа. Классы могут объявлять типы данных также как и функции класса и поля класса. Как и любые другие члены класса, типы, появляющиеся в классе, могут быть объявлены с любым спецификатором доступа. Хотя, что касается полей класса, вы должны по умолчанию объявлять их приватными и только в случае явных причин делать их менее закрытыми. Объявление типов обычно лежит в основе того, как класс действует за сценой, и, как следствие, они имеют жизненно важное значение для сокрытия. Кроме того, в большинстве случаев клиентскому коду не нужны типы, которые вы объявили в своем классе. Исключения бывают тогда, когда тип, объявленный в классе, используется как тип возвращаемого значения в публичном методе или как тип параметра публичного метода. В этом случае тип должен быть публичным или клиентский код не сможет воспользоваться публичным методом. Класс `studentCollection` предполагает, что структура типа `studentRecord` будет отдельно объявлена, но мы могли бы сделать ее частью класса. Если бы мы это сделали, нам надо было бы объявить ее в секции `public`.

Теперь мы можем реализовать методы нашего класса, начав с конструктора. В отличие от предыдущего примера, у нас будет только конструктор по умолчанию. Конструктора, принимающего параметры для инициализации полей, в классе не будет. Основная цель нашего класса спрятать детали связного списка, поэтому мы не хотим, чтобы клиент даже думал о `_listHead`, не говоря уж о манипуляции с ним. Все что нам надо сделать в конструкторе по умолчанию, это установить `NULL` как значение указателя на начало списка:

```
studentCollection::studentCollection() {
    listHead = NULL;
}
```

Добавление узла

Перейдем к функции addRecord. Поскольку условия задачи не требуют хранения записей о студентах в каком-то особом порядке, мы можем непосредственно адаптировать функцию addRecord из главы 4.

```
void studentCollection::addRecord(❶studentRecord newStudent) {
    ❷studentNode * newNode = new studentNode;
    ❸newNode->studentData = newStudent;
    ❹newNode->next = _listHead;
    ❺_listHead = newNode;
}
```

Между этим кодом и исходной функцией всего два отличия. Здесь нам достаточно только одного параметра в списке ❶. Этим параметром является объект studentRecord, который мы собираемся добавить в нашу коллекцию. Этот объект инкапсулирует все данные о студенте, тем самым уменьшая количество необходимых параметров. Кроме того, нам не надо передавать указатель на начало списка, поскольку он уже хранится в нашем классе как _listHead и мы обращаемся к нему по необходимости. Как и в функции addRecord из главы 4, мы создаем новый узел ❷, копируем данные по новому студенту в новый узел ❸, указываем в новом узле в поле «следующий» на предыдущий первый узел списка ❹, и наконец, устанавливаем указатель _listHead на новый узел ❺. Обычно я рекомендую составлять схемы для всех манипуляций с указателями, однако, поскольку мы повторяем уже проделанные нами манипуляции, мы можем воспользоваться старыми схемами.

Теперь обратимся ко второй функции класса в списке, recordWithNumber. Это название слегка труднопроизносимо, и, возможно, некоторые программисты выбрали retrieveRecord или что-то подобное. Однако, следуя своим изложенным выше принципам наименования, я решил использовать существительное, поскольку этот метод возвращает значение. Этот метод будет похож на метод averageRecord, поскольку ему также потребуется пробежаться по списку; однако отличие будет в том, что как только мы обнаружим соответствующую студенческую запись, мы сможем остановиться.

```
studentRecord studentCollection::recordWithNumber(int idNum) {
  ❶ studentNode * loopPtr = _listHead;
  ❷ while (loopPtr->studentData.studentID() != idNum) {
        loopPtr = loopPtr->next;
    }
  ❸ return loopPtr->studentData;
}
```

В этой функции мы инициализируем указатель нашего цикла на начале списка ❶ и пробегаем по списку, пока не обнаружим желаемый идентификатор ❷. Наконец, достигнув желаемого узла, мы возвращаем всю соответствующую запись как значение функции ❸. Этот код выглядит отлично, однако, как всегда, нужно учесть специальные случаи. Случай, который всегда необходимо рассматривать, имея дело со

связным списком, это исходный головной указатель, имеющий значение NULL. Здесь это определенно вызовет проблему, поскольку мы не проверяем это и код вызовет сбой, если мы попытаемся разыменовать указатель loopPtr после входа в цикл. Кроме того, мы должны рассмотреть возможность, что идентификатор, переданный клиентским кодом, вообще не соответствует никаким записям в коллекции. В этом случае, даже если _listHead не NULL, loopPtr в конце концов станет NULL, когда мы достигнем конца списка.

Таким образом, основная идея состоит в том, что мы должны остановить цикл, если значение loopPtr становится NULL. Это не сложно, но что мы вернем в такой ситуации? Очевидно, мы не можем вернуть loopPtr->studentData, поскольку loopPtr будет NULL. Вместо этого мы можем создать и вернуть фиктивную запись studentRecord с очевидно некорректными значениями.

```
studentRecord studentCollection::recordWithNumber(int idNum) {
   studentNode * loopPtr = _listHead;
   while (❶loopPtr != NULL && loopPtr->studentData.studentID() != idNum) {
      loopPtr = loopPtr->next;
   }
   if (❷loopPtr == NULL) {
    ❸ studentRecord dummyRecord( -1, -1, "");
      return dummyRecord;
   } else {
      return loopPtr->studentData;
   }
}
```

В этой версии метода, если цикл закончен, а указатель цикла содержит значение NULL ❷, мы создаем фиктивную запись с пустой строкой вместо фамилии студента и значениями –1 для его оценки и идентификатора ❸ и возвращаем эту запись. Возвращаясь к циклу, мы проверяем, равен ли NULL указатель loopPtr, что может произойти, если список пуст либо мы безуспешно полностью его просмотрели. Обратим внимание, что выражение условия цикла ❶ — составное выражение с условием loopPtr != NULL, идущим первым. Именно так и должно быть. Язык C++ использует для оценки составных логических выражений механизм, известный как *сокращенные вычисления*. Проще говоря, правая часть составного логического выражения не рассматривается, если значение всего выражения уже известно. Поскольку && означает логическое *и*, то если левая часть выражения && оказывается ложной, все выражение также оказывается ложью, независимо от значения правой части. Для повышения эффективности, C++ использует этот факт, опуская расчет правой части выражения &&, если левая часть оказывается ложной (для || или логического *или* правая часть по той же причине не рассчитывается, когда левая часть истинна). Следовательно, когда loopPtr равен NULL, выражение loopPtr != NULL ложно, и правая часть выражения && не рассчитывается. Без сокращенных вычислений правая часть рассчитывалась *бы* и указатель на NULL разыменовывался бы, вызывая сбой программы.

Эта реализация позволяет избежать потенциальной поломки, характерной для первой версии, однако она оказывает слишком много доверия клиентскому коду. То есть функция, вызывающая этот метод, должна проверить вернувшуюся обратно запись studentRecord и до выполнения дальнейших операций убедиться, что она не фиктивная. Если вы похожи на меня, то вам должно быть немного не по себе.

> **ИСКЛЮЧЕНИЯ**
>
> *Есть еще один способ. C++, как и многие другие языки программирования, предлагает механизм, известный как исключения, позволяющий методу или глобальной функции недвусмысленно сообщить об ошибке туда, откуда она была вызвана. Это сделано для ситуации, которая сложилась в этом методе, когда нет хорошего ответа на вопрос, что возвращать, если ввод был некорректным. Исключения сложнее, чем мы можем позволить себе рассматривать сейчас, и, к сожалению, реализация исключений в C++ не решает проблему доверия, описанную в предыдущем абзаце.*

Перегруппировка списка

Метод removeRecord похож на recordWithNumber в той, что мы должны пройти список для нахождения узла, который необходимо удалить. Однако есть задачи сверх этого. Удаление узла из списка требует внимательности, поскольку остальные узлы должны остаться связным списком. Проще всего зашить сделанную нами дырочку соединением узла, который шел перед удаленным, с тем, который шел следом за ним. Нам не нужно набрасывать функцию, потому что у нас уже есть прототип функции в объявлении класса, поэтому перейдем к тестам.

```
  studentCollection s;
  studentRecord stu3(84, 1152, "Sue");
  studentRecord stu2(75, 4875, "Ed");
  studentRecord stu1(98, 2938, "Todd");
  s.addRecord(stu3);
  s.addRecord(stu2);
  s.addRecord(stu1);
❶ s.removeRecord(4875)
```

Мы создали объект s класса studentCollection, а также три объекта класса studentRecord, каждый из которых был добавлен в коллекцию. Обратите внимание, что мы могли бы использовать одну и ту же запись, меняя значения между вызовами addRecord, но не стали этого делать для упрощения тестового кода. Последняя строка в тесте вызывает метод removeRecord ❶, который в данном случае удалит вторую запись, то есть запись для студента по имени «Ed». Используя тот же стиль схем указателей, что и в главе 4, проиллюстрируем на рис. 5.1 состояние памяти «до» и «после» этого вызова.

На рис. 5.1 (а) мы видим связный список, созданный нашим тестовым кодом. Обратите внимание, что, поскольку мы используем класс, наши соглашения о схемах слегка искажены. В левой части рисунка

изображен стек/куча, в котором находятся _listHead, являющийся приватным полем внутри объекта s класса studentCollection, и idNum, являющийся параметром метода removeRecord. В правой части рисунка изображен сам список в куче. Помните, что addRecord размещает новую запись в начало списка, поэтому записи расположены в обратном порядке по сравнению с тем, как они добавлялись в тестовом коде. Идентификатор среднего узла, «Ed» совпадает с параметром 4875, поэтому он должен быть удален из списка. Рис. 5.1 (б) показывает результат вызова метода. Первый узел списка, «Todd», теперь указывает на узел, который ранее был в списке третьим, то есть на «Sue». Узел «Ed» больше не связан с большим списком и удален.

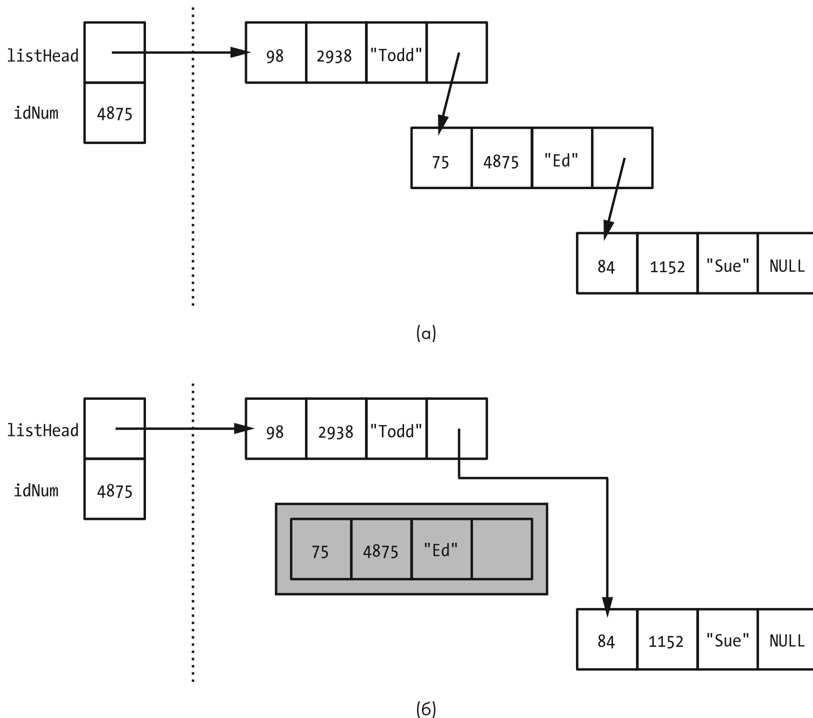


Рис. 5.1. Ситуация «до» и «после» вызова метода removeRecord *в тестовом примере*

Теперь, зная, какого результата мы хотим добиться, начнем писать код. Поскольку мы знаем, что нам надо найти узел с соответствующим идентификатором, мы могли бы начать с цикла while из метода recordWithNumber. По окончанию это цикла указатель будет указывать на узел, который надо удалить. К сожалению, нам надо кое-что еще, чтобы закончить удаление. Посмотрите на рис. 5.1. Чтобы закрыть дыру и восстановить связный список, мы должны изменить поле next узла «Todd». Если все, что у нас есть, это указатель на узел «Ed», способа указать на «Todd» не существует, потому что каждый узел в связном списке указывает на своего последователя, а не предшественника. (Из-за подобной ситуации некоторые связные списки связывают в обоих направлениях, такие списки называются

двойные связные списки, но нужда в них возникает редко.) Таким образом, в дополнение к указателю на удаляемый узел (который называется `loopPtr`, если мы применим код предыдущей функции), нам необходим указатель на предыдущий узел: давайте назовем такой указатель `trailing`. Рис. 5.2 демонстрирует эту концепции применимо к нашему примеру.

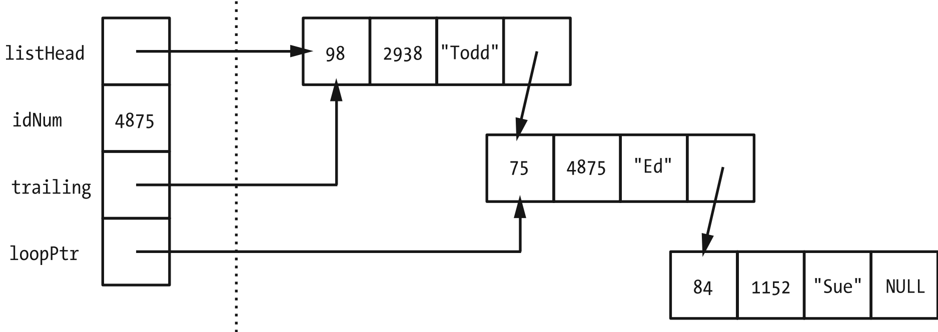


Рис. 5.2. Указатели, необходимые для удаления узла с номером `idNum`

С указателями `loopPtr` ссылающимся на удаляемый узел и `trailing`, ссылающимся на предыдущий узел, мы можем удалить желаемый узел и сохранить список.

```
void studentCollection::removeRecord(int idNum) {
     studentNode * loopPtr = _listHead;
  ❶ studentNode * trailing = NULL;
     while (loopPtr != NULL && loopPtr->studentData.studentID() != idNum) {
        ❷trailing = loopPtr;
        loopPtr = loopPtr->next;
     }
  ❸ if (loopPtr == NULL) return;
  ❹ trailing->next = loopPtr->next;
  ❺ delete loopPtr;
}
```

Первая часть этой функции похожа на метод `recordWithNumber`, с тем исключением, что мы объявили указатель `trailing` ❶ и внутри цикла присвоили ему старое значение `loopPtr` ❷ до перехода `loopPtr` к следующему узлу. Таким образом, указатель `trailing` всегда находится на один узел позади `loopPtr`. Благодаря нашей работе с предыдущей функцией мы уже знаем про специальный случай. Поэтому, когда цикл закончится, мы проверяем, не равен ли `loopPtr` значению `NULL`. Если равен, это означает, что мы не нашли узел с заданным идентификатором и немедленно возвращаемся ❸. Я называю выражение `return` в середине функции «смываться». Некоторые программисты возражают против этого, потому что функцию с несколькими точками выхода тяжело читать. Однако альтернативный вариант в этом случае еще один уровень вложения для инструкции `if`, которое идет следом, так что я лучше смоюсь.

Определив, что удаляемый узел существует, пора удалить его. На диаграмме видео, что необходимо установить, чтобы поле `next` узла

trailing указывало на узел, на который в настоящий момент указывает поле next узла loopPtr ❹. Затем мы можем безопасно удалить узел, на который указывает loopPtr ❺.

Этот код работает для нашего тестового примера, но, как всегда, необходимо проверить его на возможных специальных случаях. Мы уже разобрались с возможностью, что номер idNum отсутствует в записях нашей коллекции, но нет ли еще какой-нибудь проблемы? Посмотрим на пример. Изменится ли что-нибудь, если мы попробуем удалить первый или третий узел вместо среднего? Тестирование показывает отсутствие проблем при удалении третьего (последнего) узла. Однако удаление первого узла и в самом деле вызывает трудности из-за того, что в этом случае отсутствует предыдущий узел, на который должен указывать trailing. Для этого мы должны манипулировать указателем _listHead. Рис. 5.3 иллюстрирует ситуацию, сложившуюся по окончании цикла while.

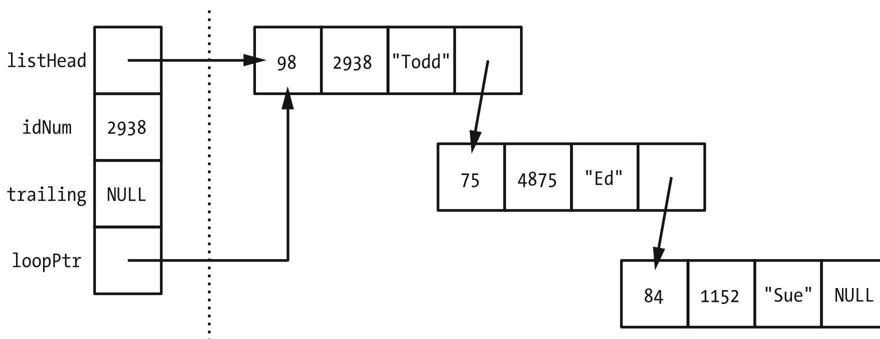


Рис. 5.3. Положение до удаления первого узла в списке

В этой ситуации нам нужно перенаправить _listHead на бывший второй узел в списке, то есть на узел «Ed». Давайте перепишем наш метод, чтобы решить данную проблему.

```
void studentCollection::removeRecord(int idNum) {
   studentNode * loopPtr = _listHead;
   studentNode * trailing = NULL;
   while (loopPtr != NULL && loopPtr->studentData.studentID() != idNum) {
      trailing = loopPtr;
      loopPtr = loopPtr->next;
   }
   if (loopPtr == NULL) return;
❶ if (trailing == NULL) {
   ❷ _listHead = _listHead->next;
   } else {
      trailing->next = loopPtr->next;
   }
   delete loopPtr;
}
```

Как видите, как проверка условия ❶, так и код, решающий проблему ❷, просты, поскольку мы тщательно проанализировали ситуацию, прежде чем писать его.

Деструктор

Реализовав эти три метода, мы можем подумать, что класс studentCollection готов. Однако это не так. Первое, чего не хватает нашему классу, это *деструктор*. Это специальный метод, вызываемый, когда объект выходит из области видимости (когда функция, объявившая этот объект, закончила работу). Если в классе отсутствуют динамические данные, обычно деструктор не нужен. Однако если такие данные присутствуют, без деструктора не обойтись. Помните, для того, чтобы не допустить утечек памяти, мы должны применить ключевое слово delete для всех объектов, которым была выделена память с помощью ключевого слова new. Если объект нашего класса studentCollection содержит три узла, необходимо освободить память, которую занимает каждый из них. К счастью, это несложно. Нам необходимо пройти по нашему связному списку, удаляя узлы. Однако вместо того, чтобы делать это непосредственно, давайте напишем служебный метод, удаляющий все узлы в studentList. В приватной секции нашего класса добавим объявление:

```
void deleteList(studentList &listPtr);
```

Код для этого метода будет выглядеть так:

```
void studentCollection::deleteList(studentList &listPtr) {
   while (listPtr != NULL) {
      ❶ studentNode * temp = listPtr;
      ❷ listPtr = listPtr->next;
      ❸ delete temp;
   }
}
```

Во время обхода указатель на текущий узел копируется во временную переменную ❶, передвигается к следующему узлу ❷, а затем удаляется узел, на который указывает временная переменная ❸. С помощью этого кода мы можем написать деструктор очень просто. Прежде всего, добавим деструктор в публичную секцию объявления нашего класса:

```
~studentCollection();
```

Обратите внимание, что, как и у конструктора, имя деструктора совпадает с названием класса и в нем отсутствует возвращаемый тип. Тильда перед именем позволяет отличить деструктор от конструктора. Реализация будет выглядеть так:

```
studentCollection::~studentCollection() {
     deleteList(_listHead);
}
```

Код в этом методе прост, однако очень важно протестировать деструктор. Хотя плохо написанный деструктор и может вызвать сбой вашей программы, большинство проблем с деструктором вызывают

не ошибку программы, но утечки памяти или, что хуже, необъяснимое поведение программы. Следовательно, важно протестировать деструктор с помощью отладчика среды разработки, так чтобы вы могли видеть, действительно ли деструктор вызывает `delete` для каждого узла.

Глубокое копирование

Осталась еще одна серьезная проблема. В главе 4 мы кратко обсудили концепцию перекрестных связей, когда две переменных типа указатель имеют одно и то же значение. Даже если это разные переменные, они указывают на одну и ту же структуру данных; следовательно, изменение структуры одной переменной изменяет обе. Эта проблема часто всплывает в классах, включающих динамически распределенную память. Чтобы увидеть, почему это проблема, рассмотрим следующий элементарный код на C++:

```
  int x = 10;
  int y = 15;
  x = y;
❶ x = 5;
```

Предположим, я спросил бы вас, какое влияние последнее выражение ❶ оказывает на переменную y. Вы, вероятно, удивились бы не оговорился ли я. Последнее выражение не оказывает совсем никакого эффекта на y, только на x. А теперь посмотрите на это:

```
  studentCollection s1;
  studentCollection s2;
  studentRecord r1(85, 99837, "John");
  s2.addRecord(r1);
  studentRecord r2(77, 4765, "Elsie");
  s2.addRecord(r2);
❶ s1 = s2;
❷ s2.removeRecord(99837)
```

Предположим, я спрошу вас, какое влияние окажет последнее выражение ❷ на `s1`. К сожалению, влияние действительно будет. Хотя `s1` и `s2` — два разных объекта, они отныне не полностью отдельные объекты. По умолчанию, когда один объект приравнен к другому, как мы сейчас приравняли объекты `s1` и `s2` ❶, язык C++ производит то, что известно как *поверхностное копирование*. При поверхностном копировании каждое поле класса одного объекта присваивается другому. Так если, `_listHead` — наше единственное поле класса, публично, `s2 = s1` будет аналогично `s1._listHead = s2._listHead`. Получается, что поле `_listHead` обоих объектов указывает на одно и то же место в памяти: узел для «Elsie», который указывает на другой узел, а именно на узел «John». Поэтому после удаления узла «John», он удаляется из двух списков, поскольку на самом деле существует только один список. Рис. 5.4 иллюстрирует положение, сложившееся к концу данного кода.

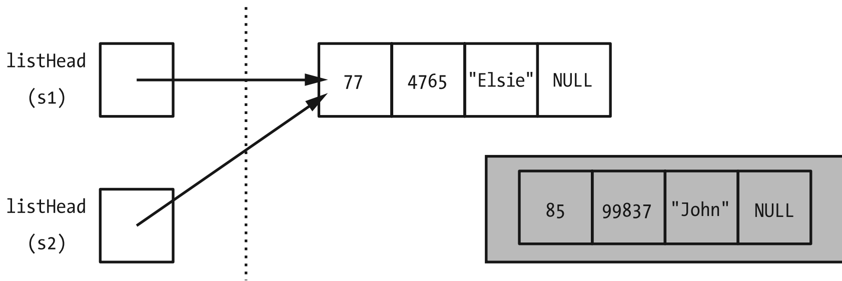


Рис. 5.4. Результаты поверхностного копирования при перекрестных связях; удаление узла «John» из одного списка удаляет из обоих

Однако все может быть еще хуже. Что, если последняя строка кода удалит первую запись — узел «Elsie»? В этом случае указатель _listHead объекта s2 станет указывать на узел «John», а узел «Elsie» будет удален. Однако указатель `_listHead` объекта `s1` по-прежнему будет указывать на удаленный узел «Elsie», то есть возникнет опасная висячая ссылка, как показано на рис. 5.5.

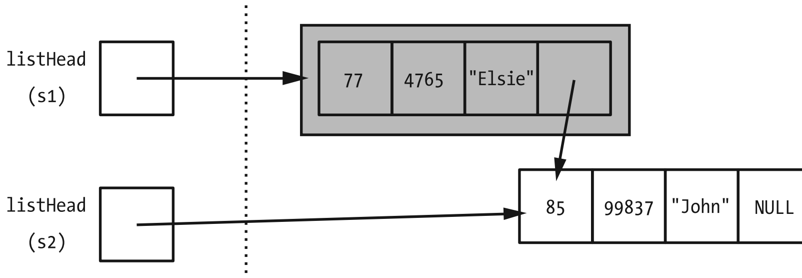


Рис. 5.5. Удаление из s2 приводит к висящей ссылке в s1

Для решения этой проблемы применяют *глубокое копирование*, то есть не только копирование указателя на структуру, но копирование всей структуры. В нашем случае это означает копирование всех узлов списка. Тем самым происходит истинное копирование. Как и ранее, давайте начнем с приватных служебных методов, то есть в нашем случае с метода, копирующего `studentList`. Объявление в приватной секции класса выглядит примерно так:

```
studentList copiedList(const studentList original);
```

Как и ранее, я выбрал существительное для названия метода, возвращающего значение. Реализация метода выглядит так:

```
❶ studentCollection::studentList studentCollection::copiedList(const
  studentList original) {
  ❷ if (original == NULL) {
        return NULL;
    }
    studentList newList = new studentNode;
  ❸ newList->studentData = original->studentData;
  ❹ studentNode * oldLoopPtr = original->next;
  ❺ studentNode * newLoopPtr = newList;
    while (oldLoopPtr != NULL) {
```

```
        ❻ newLoopPtr->next = new studentNode;
          newLoopPtr = newLoopPtr->next;
          newLoopPtr->studentData = oldLoopPtr->studentData;
          oldLoopPtr = oldLoopPtr->next;
      }
  ❼ newLoopPtr->next = NULL;
  ❽ return newList;
  }
```

В этом методе происходит много всего, поэтому давайте будем разбираться постепенно. Прежде всего, обратим внимание, что, определяя тип возвращаемого значения, мы должны указать в префиксе имя класса ❶. В противном случае компилятор не поймет, о каком типе мы говорим. (Внутри метода это необязательно, поскольку компилятор уже знает, частью какого класса является метод – немного запутанно!) Мы проверяем, не пуст ли входящий список. Если он пуст, смываемся ❷. Теперь, зная, что у нас есть список для копирования, копируем данные первого узла до начала цикла ❸, поскольку для этого узла нам необходимо изменить указатель на начало нового списка.

Затем мы устанавливаем два указателя для прохода по списку. Указатель `oldLoopPtr` ❹ проходит через пришедший список; он всегда будет указывать на узел, который мы собираемся копировать. Указатель `newLoopPtr` ❺ проходит через новый, скопированный, список и всегда указывает на последний созданный узел, то есть узел, до которого мы будем добавлять следующий узел. Как и в методе `removeRecord` нам нужен здесь хвостовой указатель. В цикле ❻ мы создаем новый узел, передвигаем `newLoopPtr`, чтобы он указывал на него, копируем данные из старого узла в новый и передвигаем указатель `oldLoopPtr`. После цикла мы завершаем новый список, присваивая значение `NULL` полю `next` последнего узла ❼ и возвращаем (`return`) указатель на новый список ❽.

Как же этот служебный метод решает обозначенную выше проблему? Сам по себе никак. Но, добавив этот код, мы можем перегрузить оператор присваивания. *Перегрузка операторов* — особенность языка C++, позволяющая изменять действия, производимые встроенными операторами над определенными типами данных. В нашем случае мы хотим перегрузить оператор присваивания (=) так, чтобы вместо установленного по умолчанию поверхностного копирования, он вызывал наш метод `copiedList` для выполнения глубокого копирования. В публичной секции класса добавим следующее:

```
❶studentCollection& ❷operator=(❸const studentCollection & ❹rhs);
```

Перегружаемый оператор определяется названием метода с использованием ключевого слова `operator`, за которым идет оператор, который мы хотим перегрузить ❷. Имя выбранное мной для параметра (`rhs` ❹) обычный выбор для перегрузки операторов, поскольку это сокращение от словосочетания «*правая сторона*» (right-hand side). Это помогает программистам упрощать себе жизнь. Так, в присвоении, о котором идет речь `s2 = s1`, объект `s1` будет правой стороной оператора присваивания, а `s2` — левой стороной. На правую часть мы

ссылаемся через параметр, а на левую — непосредственно через поле класса, так же как и при любом другом методе класса. Итак, наша задача состоит в создании списка, на который будет указывать `_listHead` и который будет являться копией списка, на который указывает `_listHead` объекта `rhs`. Таким образом, при вызове `s2 = s1` будет создан объект `s2` как истинная копия объекта `s1`.

Тип параметра — это постоянная ссылка на наш класс ❸. Тип возвращаемого значения — это всегда ссылка на класс ❶. Скоро вы увидите, почему параметр — это ссылка. Вас, возможно, удивляет, почему метод возвращает все подряд, начиная с того, что мы изменяем поле класса непосредственно в методе. Это происходит из-за того, что язык C++ позволяет присвоение по цепочке, например `s3 = s2 = s1`, в котором возвращаемое значение первого присвоения становится параметром следующего.

Теперь, когда синтаксис понятен, код оператора присваивания становится вполне очевидным:

```
studentCollection& studentCollection::operator=(const studentCollection &rhs) {
  ❶ if (this != &rhs) {
        ❷ deleteList(_listHead);
        ❸ _listHead = copiedList(rhs._listHead);
     }
  ❹ return *this;
}
```

Чтобы избежать утечки памяти, мы сначала должны удалить все узлы из левостороннего списка ❷. (Именно для этой цели мы отдельно написали вспомогательный метод `deleteList`, а не включили его код непосредственно в деструктор.) Удалив предыдущий левосторонний список, мы копируем правосторонний список, используя наш другой вспомогательный метод ❸. Однако до выполнения этих шагов мы проверяем, что объект с правовой стороны отличается от объекта с левой (то есть, это не что-то подобное `s1 = s2`) проверяя, отличаются ли указатели ❶. Если указатели одинаковые, нет необходимости что-то делать, при этом это не вопрос эффективности. Если мы выполним глубокое копирование для одинаковых указателей, когда мы будем удалять узлы в левостороннем списке, мы также будем удалять узлы и в правостороннем. Наконец, мы возвращаем указатель на левосторонний объект ❹; это случится независимо от того, действительно ли скопировали мы что-нибудь или нет, потому что, хотя выражение, подобное `s3 = s2 = s1`, выглядит странным, мы хотим, чтобы оно работало, если кто-нибудь попытается его выполнить.

Сделав служебный метод для копирования списка, мы должны создать и *копирующий конструктор*. Это конструктор, принимающий другой объект того же класса как объект. Копирующий конструктор вызывается явно, когда нам надо создать дубликат существующего списка `studentCollection`, но может вызываться и неявно, если объект этого класса передается как параметр по значению в функцию. Из-за этого вы должны предусмотреть передачу объекта по ссылке

с ключевым словом const вместо передачи по значению, за исключением тех случаев, когда функция, получающая объект, должна изменить его копию. В противном случае ваш код будет делать много ненужной работы. Например, рассмотрим коллекцию из 10 000 записей о студентах. Коллекция может быть передана по ссылке одним указателем. В противном случае копирующий конструктор будет вызываться каждый раз во время длительного прохода по коллекции и распределять память 10 000 раз. Затем для этих копий в конце работы функции будет вызван деструктор с очередным долгим проходом по коллекции высвобождением памяти 10 000 раз. Вот поэтому правосторонний параметр перегрузки оператора присваивания использует параметр по ссылке с ключевым словом const.

Для добавления копирующего конструктора в наш класс, добавим прежде его объявление в публичной секции объявления нашего класса.

```
studentCollection(const studentCollection &original);
```

Как и во всех конструкторах, здесь нет типа возвращаемого значения. И, как и в случае перегрузки оператора присваивания, передаваемый параметр — это ссылка на наш класс с ключевым словом const. Поскольку мы уже создали служебный метод, реализация будет простой.

```
studentCollection::studentCollection(const studentCollection &original) {
   _listHead = copiedList(original._listHead);
}
```

Теперь мы можем выполнить подобное объявление:

```
studentCollection s2(s1);
```

Это объявление создает s2 и копирует туда все узлы s1.

Общий обзор классов с динамической памятью

Мы сделали многое в этом классе, завершив методы, указанные в условиях задачи, а теперь давайте взглянем, что у нас получилось. Вот так выглядит объявление нашего класса.

```
class studentCollection {
private:
   struct studentNode {
      studentRecord studentData;
      studentNode * next;
   };
public:
   studentCollection();
   ~studentCollection();
   studentCollection(const studentCollection &original);
   studentCollection& operator=(const studentCollection &rhs);
   void addRecord(studentRecord newStudent);
   studentRecord recordWithNumber(int idNum);
   void removeRecord(int idNum);
```

```
private:
   typedef studentNode * studentList;
   studentList _listHead;
   void deleteList(studentList &listPtr);
   studentList copiedList(const studentList original);
};
```

Данный урок состоит в том, что при создании классов с динамической памятью необходимо добавлять новые части. В дополнении к сущностям базового фреймворка класса — приватным данным, конструктору и методам отправления данных в объект и получения их оттуда — мы должны добавить методы, работающие с выделением и высвобождением динамической памяти. Как минимум мы должны добавить конструктор и деструктор копирования, а также перегрузить оператор присваивания, если есть вероятность, что он кому-нибудь понадобится. Создание этих дополнительных методов часто облегчается созданием служебных методов копирования или удаления динамических структур данных.

Кажется, что это большая работа. И это действительно может быть так. Но важно понимать, что со всем, что вы добавите в класс, вам так или иначе придется разбираться. Другими словами, даже если у нас нет класса для коллекции записей о студентах, основанной на связном списке, мы все равно должны удалять узлы в списке после прохода по ним. Нам все равно придется думать о перекрестных связях, все равно придется проходить сквозь список и копировать узел за узлом, если нам нужна истинная копия исходного списка и так далее. Размещение всего в класс требует немного больше начальной подготовки, но как только все заработает, клиентский код может не задумываться обо всех деталях, связанных с распределением памяти. Кроме того, инкапсуляции и сокрытие существенно облегчают работу со структурами динамических данных.

Ошибки, которых следует избегать

Мы поговорили о том, как создать хороший класс в языке C++, давайте теперь обсудим несколько подводных камней, которые следует избегать.

Фальшивый класс

Как я уже говорил в начале этой главы, я считаю C++ отличным языком для изучения объектно ориентированного программирования, поскольку это гибридный язык, включающий как процедурную, так и объектно ориентированную парадигму. В этом случае создание класса всегда является выбором программиста. В языках, подобных Java, никогда не стоит вопрос: «Надо ли мне делать класс?» Вместо него возникает вопрос: «Как мне поместить это все в класс?» Требование поместить все в класс приводит к появлению того, что я

называю *фальшивым классом,* то есть классом без последовательного проектирования, который корректен синтаксически, но не имеет смысла. Слово *класс*, как его используют программисты, соответствует смыслу английского слова, означающего группу вещей с общими атрибутами, и хороший класс языка C++ соответствует этому определению.

Фальшивый класс может быть создан по нескольким причинам. Например, программист хочет использовать глобальные переменные, но не по обоснованным причинам (такие причины редки, но все-таки существуют), а из-за лени — чтобы избегать передачи параметров от функции к функции. При этом если программист понимает, что широкое использование глобальных переменных это признак плохого стиля, он или она полагает, что нашел лазейку. Все или большинство функций программы сваливают в класс, и теперь переменные, которые были бы глобальными, становятся полями этого класса. Функция main такой программы создает объект этого фальшивого класса и вызывает какой-то «главный» метод в классе. Технически программа не использует глобальные переменные, но фальшивый класс означает, что этой программе присущи все недостатки программы с глобальными переменными.

Другой тип фальшивых классов возникает из-за того, что программист полагает, что объектно ориентированное программирование всегда «лучше», и применяет его в тех ситуациях, где оно не подходит. В этих случаях программист часто создает класс, который инкапсулирует специфический функционал, имеющий смысл только в контексте исходной программы, для которой он написан. Есть два способа проверить, пишете ли вы фальшивый класс или нет. Задайте себе вопрос: «Могу я назвать класс кратко и понятно?» Если имя будет выглядеть как `PayrollReportManagerAndPrintSpooler`, по-видимому, у вас проблемы. Другой тест выглядит так: «если мне придется писать другую программу с похожим функционалом, можно ли этот класс использовать еще раз с небольшими изменениями? Или его придется кардинальным образом переписывать?»

Даже в языке C++ появление фальшивых классов неизбежно, например, когда нам надо инкапсулировать данные для использования в классах-коллекциях. Однако такие классы обычно маленькие и элементарные. Когда мы избегаем создания фальшивых классов, наш код становится лучше.

Однозадачники

Если вам доводилось смотреть телевизионное шоу `Good Eats`, вы знаете, что ведущий Элтон Браун тратит массу времени, рассуждая, как оборудовать кухню с максимальной эффективностью. Часто он восстает против кухонных приспособлений, которые называет *однозадачниками*, подразумевая инструменты, хорошо решающие одну задачу, но не приспособленные более ни для чего. Разрабатывая классы,

мы должны стремиться сделать их настолько общими, насколько это возможно, последовательно включая все функции необходимые для нашей программы.

Один из способов сделать это состоит в использовании шаблонных классов. Это продвинутый способ с немного загадочным синтаксисом, однако он позволяет нам создавать классы, где у одного или более полей тип определяется в момент создания объекта класса. Шаблонные классы позволяют нам «выделить» основной функционал. Например, наш класс `studentCollection` содержит много кода одинакового для любого класса, инкапсулирующего связный список. Вместо этого мы можем сделать шаблонный класс для общего связного списка, такого, чтобы тип данных внутри узлов списка определялся в момент создания объекта шаблонного класса, а не программировался как `studentRecord`. Тогда полем нашего класса `studentCollection` будет объект шаблонного класса связного списка, а не указатель на начало списка, и наш класс больше не сможет непосредственно управлять связным списком.

Мы не будем рассматривать шаблонные классы в этой книге, однако, развивая ваши навыки проектировщика классов, вы должны всегда стремиться делать классы «многозадачниками». Когда вы обнаруживаете, что текущую задачу можно решить, используя класс, написанный тогда, когда вы даже не подозревали о ее существовании, вы испытываете чувство глубокого удовлетворения.

Упражнения

Вы знаете, что я хочу вам сказать, не так ли? Вперед, попытаетесь сами!

5.1. Давайте попробуем реализовать класс, используя базовый фреймворк. Рассмотрим класс, хранящий данные по автомобилям. У нас есть три элемента данных: название производителя, название модели (оба строковые) и год выпуска – целочисленное. Создайте класс с методами `get/set` для каждого поля класса. Убедитесь, что вы приняли верные решения о деталях, например имени переменных. Необязательно пользоваться моей конвенцией наименования, важнее грамотно подходить к принятию решений и быть в них последовательными.

5.2. Добавьте в наш автомобильный класс из предыдущего упражнения служебный метод, возвращающий полное описание автомобиля в формате строки, например, «`1957 Chevrolet Impala`». Добавьте второй служебный метод, возвращающий возраст автомобиля в годах.

5.3. Возьмите функции для строк переменной длины из главы 4 (`append`, `concatenate` и `characterAt`) и используйте их для создания класса для строк переменной длины. Убедитесь, что вы реализовали все необходимые конструкторы, деструктор, а также перегрузили оператор присваивания.

5.4. Для класса строк переменной длины из предыдущего упражнения замените метод `characterAt` перегруженным оператором `[]`. Например, если `myString` является объектом нашего класса, тогда выражение `myString[1]` должно возвращать то же значение, что и `myString.characterAt(1)`.

5.5. Для класса строк переменной длины из предыдущих упражнений добавьте метод `remove`, принимающий начальную позицию и количество символов и удаляющий это количество символов из середины строки. Например, `myString.remove(5,3)` удалит три символа, начиная с пятой позиции. Убедитесь, что ваш метод работает, когда значение любого из параметров некорректно.

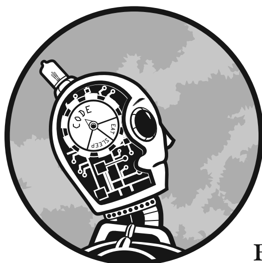
5.6. Подумайте, можно ли ваш класс строк переменной длины улучшить. Например, нет ли общего функционала, который можно выделить в приватный служебный метод?

5.7. Возьмите функции записей о студентах из главы 4 (`addRecord` и `averageRecord`) и используйте их для создания класса, представляющего коллекцию записей о студентах, как ранее. Убедитесь, что реализовали все необходимые конструкторы, деструктор, а также перегрузили оператор присваивания.

5.8. Классу коллекции записей о студентах из предыдущего упражнения добавьте метод `RecordsWithinRange`, принимающий нижнее и верхнее значение оценки как параметры и возвращающий новую коллекцию, состоящую из записей в этом диапазоне (исходная коллекция не меняется). Например, `myCollection.RecordsWithinRange(75, 80)`вернет коллекцию всех записей с оценками в диапазоне от 75 до 80 включительно.

6

Решение задач с помощью рекурсии



Эта глава посвящена *рекурсии*, при которой функция прямо или косвенно вызывает саму себя. Рекурсивное программирование выглядит так, будто оно должно быть простым. Действительно, хорошее рекурсивное решение часто имеет простой, почти элегантный вид. Тем не менее очень часто путь к этому решению весьма непрост. Это связано с тем, что рекурсия требует от нас мыслить не так, как в случае с другими типами программирования. Когда мы обрабатываем данные с помощью циклов, мы думаем о последовательной обработке, однако при обработке данных с использованием рекурсии, наш обычный процесс последовательного мышления не помогает. У многих хороших, но еще не оперившихся программистов возникают проблемы с рекурсией, поскольку они не могут найти способ применения уже освоенных ими

навыков решения задач к задачам с рекурсией. В этой главе мы обсудим систематический подход к решению таких задач. Ответ заключается в том, что мы будем называть *Большой Рекурсивной Идеей*, далее по тексту — БРИ. Эта идея настолько проста, что напоминает трюк, но она работает.

Обзор основ рекурсии

Синтаксис рекурсии не очень сложен. Трудность возникает, когда вы пытаетесь использовать рекурсию для решения задач. Рекурсия имеет место всегда, когда функция сама себя вызывает, поэтому синтаксис рекурсии представляет собой просто синтаксис вызова функции. Наиболее распространенной формой является *прямая рекурсия*, когда вызов функции происходит в теле этой же функции. Например:

```
int factorial(int n) {
  ❶ if (n == 1) return 1;
     else return n * ❷factorial(n - 1);
}
```

Эта функция, которая является общей, но неэффективной демонстрацией рекурсии, вычисляет факториал числа n. Например, если *n* равно 5, то факториал — это произведение всех чисел от 5 до 1, или 120. Обратите внимание на то, что в некоторых случаях рекурсии не происходит. В этой функции, если параметр равен 1, мы просто возвращаем значение напрямую без какой-либо рекурсии ❶, это так называемый *базовый случай*. В противном случае мы делаем рекурсивный вызов ❷.

Другой формой рекурсии является *косвенная рекурсия* — например, когда функция A вызывает функцию B, которая далее вызывает функцию A. Косвенная рекурсия редко используется в качестве метода решения задач, поэтому мы не будем здесь ее обсуждать.

Головная и хвостовая рекурсия

Прежде чем обсуждать БРИ, нам нужно понять разницу между головной и хвостовой рекурсией. В случае *головной рекурсии* рекурсивный вызов, когда он происходит, предваряет другие процессы обработки в функции (считайте, что он происходит вверху или в голове функции). В случае *хвостовой рекурсии* все наоборот — обработка происходит перед рекурсивным вызовом. Выбор между двумя рекурсивными стилями может показаться произвольным, однако в этом выборе заключается все различие. Чтобы проиллюстрировать это различие, давайте рассмотрим две задачи.

Задача: подсчет количества попугаев

Пассажиры Райской тропической железной дороги (РТЖД) с нетерпением ждут того момента, когда смогут увидеть из окон поезда многочисленных красочных птиц. По этой причине руководство железной дороги очень заинтересовано в состоянии здоровья

местной популяции попугаев и решает подсчитать количество попугаев, находящихся в пределах видимости с каждой железнодорожной платформы вдоль главной линии. На каждую платформу направлен сотрудник РТЖД (см. рис. 6.1), который, безусловно, способен подсчитать количество попугаев. К сожалению, работа усложняется из-за примитивной телефонной системы. С каждой платформы можно позвонить только на соседние платформы. Как передать общее число попугаев на терминал главной линии?

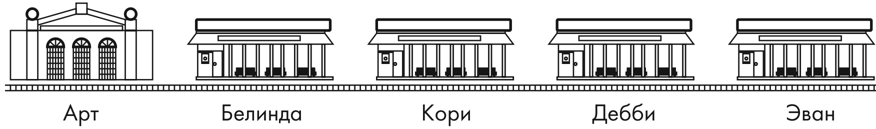


Рис. 6.1. Сотрудники пяти станций могут общаться только со своими непосредственными соседями

Предположим, что Арт насчитал на главном терминале 7 попугаев, Белинда — 5 попугаев, Кори — 3 попугая, Дебби — 10 попугаев, и 2 попугая насчитал Эван на последней станции. Таким образом, общее число попугаев составляет 27. Вопрос в том, как сотрудники собираются работать вместе, чтобы сообщить об этом Арту? Любое решение этой задачи потребует последовательной передачи сообщения от основного терминала до конца линии и обратно. Сотрудник на каждой платформе должен будет подсчитать количество попугаев, а затем сообщить о своих наблюдениях. Тем не менее существуют два разных подхода к реализации этой цепочки сообщений, которые соответствуют головной и хвостовой рекурсии в программировании.

Подход 1

При этом подходе мы сохраняем промежуточную сумму попугаев по мере продвижения по цепи исходящих сообщений. Каждый сотрудник, связываясь с сотрудником на следующей станции, сообщает количество виденных до сих пор попугаев. Когда мы доберемся до конца линии, Эван первым узнает общее количество попугаев, которое он передаст Дебби, которая передаст его Кори и т.д. (как показано на рис. 6.2).

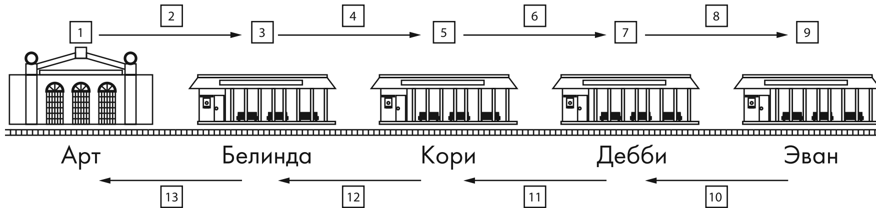


Рис. 6.2. Нумерация шагов, предпринятых при использовании подхода 1 для решения задачи подсчета попугаев

1. АРТ начинает с подсчета попугаев вокруг своей платформы. Он насчитывает 7 попугаев.
2. АРТ БЕЛИНДЕ: «На главном терминале 7 попугаев».

3. БЕЛИНДА насчитывает 5 попугаев вокруг своей платформы, промежуточная сумма равна 12.
4. БЕЛИНДА КОРИ: «Вокруг первых двух станций 12 попугаев».
5. КОРИ начитывает 3 попугая.
6. КОРИ ДЕББИ: «Вокруг первых трех станций 15 попугаев».
7. ДЕББИ насчитывает 10 попугаев.
8. ДЕББИ ЭВАНУ: «Вокруг первых четырех станций 25 попугаев».
9. ЭВАН насчитывает 2 попугая и обнаруживает, что общее количество попугаев составляет 27.
10. ЭВАН ДЕББИ: «Общее количество попугаев — 27».
11. ДЕББИ КОРИ: «Общее количество попугаев — 27».
12. КОРИ БЕЛИНДЕ: «Общее количество попугаев — 27».
13. БЕЛИНДА АРТУ: «Общее количество попугаев — 27».

Этот подход аналогичен хвостовой рекурсии. В хвостовой рекурсии рекурсивный вызов происходит после обработки — рекурсивный вызов является последним шагом в функции. Обратите внимание, что в вышеприведенной цепочке сообщений «работа» сотрудников (подсчет попугаев и суммирование) имеет место до передачи сообщения следующему сотруднику. Вся работа происходит в цепочке исходящих, а не входящих сообщений. Вот шаги, выполняемые каждым из сотрудников.

1. Подсчет попугаев, видимых с платформы станции.
2. Добавление полученного значения к сумме, переданной с предыдущей станции.
3. Передача промежуточной суммы попугаев на следующую станцию.
4. Ожидание передачи общей суммы попугаев со следующей станции и передача этого значения на предыдущую станцию.

Подход 2

При использовании этого подхода мы суммируем количества попугаев с другого конца. Каждый сотрудник, связываясь со следующей станцией вниз по линии, запрашивает общее количество попугаев с этой станции. Затем этот сотрудник прибавляет количество попугаев, виденных со своей собственной станции, и передает эту новую сумму вверх по линии (как показано на рис. 6.3).

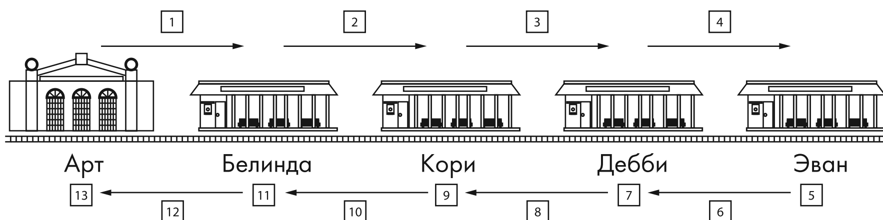


Рис. 6.3. Нумерация шагов, предпринятых при использовании подхода 2 для решения задачи подсчета попугаев.

1. АРТ БЛИНДЕ: «Каково общее количество попугаев с вашей станции до конца линии?»
2. БЕЛИНДА КОРИ: «Каково общее количество попугаев с вашей станции до конца линии?»
3. КОРИ ДЕББИ: «Каково общее количество попугаев с вашей станции до конца линии?»
4. ДЕББИ ЭВАНУ: «Каково общее количество попугаев с вашей станции до конца линии?»
5. ЭВАН находится в конце линии. Он насчитывает 2 попугая.
6. ЭВАН ДЕББИ: «Общее количество попугаев здесь в конце линии — 2».
7. ДЕББИ насчитывает 10 попугаев на своей станции, поэтому общая сумма от ее станции до конца линии составляет 12.
8. ДЕББИ КОРИ: «Общее количество попугаев отсюда до конца равно 12».
9. КОРИ насчитывает 3 попугая.
10. КОРИ БЕЛИНДЕ: «Общее количество попугаев отсюда до конца равно 15».
11. БЕЛИНДА насчитывает 5 попугаев.
12. БЕЛИНДА АРТУ: «Общее количество попугаев отсюда до конца равно 20».
13. АРТ насчитывает 7 попугаев на главном терминале, что составляет в общей сложности 27.

Этот подход аналогичен головной рекурсии. При головной рекурсии рекурсивный вызов происходит перед прочими операциями обработки. В данном случае вызов следующей станции происходит до подсчета попугаев или суммирования. «Работа» откладывается до тех пор, пока сотрудники станций, расположенных вниз по линии, не сообщат свои итоговые значения. Ниже перечислены шаги, которые выполняет каждый из сотрудников.

1. Вызов следующей станции.
2. Подсчет попугаев, видимых с платформы станции.
3. Добавление полученного значения к сумме, переданной со следующей станции.
4. Передача итоговой суммы на предыдущую станцию.

Возможно, вы заметили два практических эффекта от использования разных подходов. При использовании первого подхода в конечном итоге сотрудники всех станций узнают общее количество попугаев. При использовании второго подхода только Арт, работающий на главном терминале, узнает итоговое значение, однако заметьте, что Арт является единственным сотрудником, которому требуется это итоговое значение.

Другой практический эффект станет более важным для нашего анализа, когда мы переключим внимание на конкретный программный код. При использовании первого подхода каждый сотрудник передает «промежуточную сумму» на следующую станцию вниз по линии при выполнении запроса. При использовании второго подхода сотрудник просто запрашивает информацию со следующей станции, не передавая какие-либо данные вниз по линии. Этот эффект типичен для головной рекурсии. Поскольку рекурсивный вызов предваряет любые другие операции обработки, новая информация не передается рекурсивному вызову. В общем, головная рекурсия позволяет передать рекурсивному вызову минимальный набор данных. Теперь давайте рассмотрим еще одну задачу.

Задача: выявление лучшего клиента

Менеджер DelegateCorp должен определить, какой из восьми клиентов приносит его компании наибольшую прибыль. Два фактора усложняют эту в прочих отношениях простую задачу. Во-первых, определение общей прибыли от клиента требует перебора всех данных этого клиента и подсчета значений на десятков заказов и квитанций. Во-вторых, сотрудники DelegateCorp, как следует из названия, любят делегировать, и каждый сотрудник при любой возможности старается передать задачу своему подчиненному. Чтобы ситуация не вышла из-под контроля, менеджер устанавливает правило, согласно которому при делегировании сотрудник должен выполнить часть работы самостоятельно и поручить подчиненному меньше работы, чем досталось ему самому.

В табл. 6.1 и 6.2 перечислены сотрудники и клиенты DelegateCorp.

Табл. 6.1. Должности и ранги сотрудников DelegateCorp

Должность	Ранг
Менеджер	1
Вице-менеджер	2
Заместитель менеджера	3
Помощник менеджера	4
Младший менеджер	5
Стажер	6

Табл. 6.1. Клиенты DelegateCorp

Номер клиента	Прибыль
№0001	$172 000
№0002	$68 000
№0003	$193 000
№0004	$13 000
№0005	$256 000
№0006	$99 000

Исходя из установленного компанией правила делегирования работы, вот что произойдет с шестью файлами клиентов. Менеджер возьмет один файл и определит, какую прибыль для компа-

нии сгенерировал данный клиент. Менеджер делегирует изучение остальных пяти файлов вице-менеджеру. Он обработает один файл и передаст остальные четыре заместителю менеджера. Этот процесс будет продолжаться до тех пор, пока мы не дойдем до шестого сотрудника, стажера, которому передадут один файл, и он должен будет просто обработать его, не имея возможности для дальнейшего делегирования.

На рис. 6.4 показаны линии связи и разделения труда. Однако, как и в предыдущем примере, существует два различных подхода к реализации цепи сообщений.

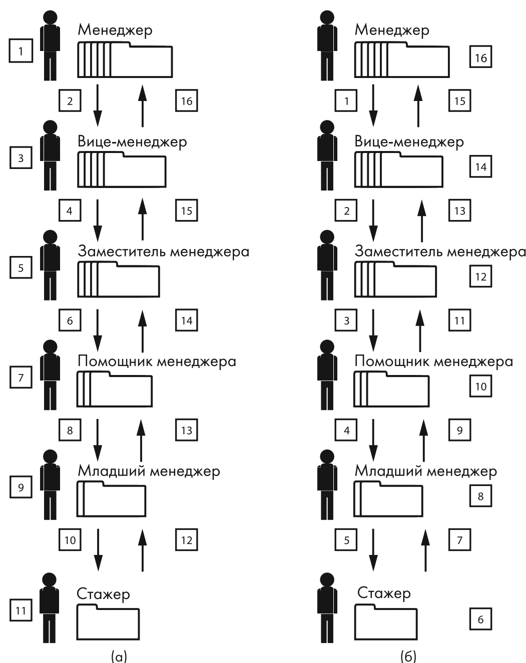


Рис. 6.4. Нумерация шагов при использовании Подхода 1 (а) и Подхода 2 (б) для определения клиента, принесшего наибольшую прибыль

Подход 1

При использовании этого подхода, делегируя оставшиеся файлы, сотрудник также передает самое высокое значение прибыли, наблюдаемое до сих пор. Это значит, что сотрудник должен подсчитать общую прибыль в одном файле и сравнить полученное значение с предыдущим наибольшим значением прибыли, прежде чем делегировать обработку оставшихся файлов другому сотруднику. Вот пример того, как это может происходить на практике.

1. МЕНЕДЖЕР подсчитывает прибыль от клиента №0001, которая составляет 172 000 долларов США.
2. МЕНЕДЖЕР ВИЦЕ-МЕНЕДЖЕРУ: «Самое высокое значение прибыли, которое мы наблюдали до сих пор, составляет 172 000

долларов США, клиент №0001. Изучите эти пять файлов и определите максимальное значение прибыли».

3. ВИЦЕ-МЕНЕДЖЕР подсчитывает прибыль, полученную от клиента №0002, которая составляет 68 000 долларов США. Наибольшим из наблюдаемых до сих пор значений прибыли по-прежнему является 172 000 долларов США, клиент №0001.
4. ВИЦЕ-МЕНЕДЖЕР ЗАМЕСТИТЕЛЮ МЕНЕДЖЕРА: «Самое высокое значение прибыли, которое мы наблюдали до сих пор, составляет 172 000 долларов США, клиент №0001. Изучите эти четыре файла и определите максимальное значение прибыли».
5. ЗАМЕСТИТЕЛЬ МЕНЕДЖЕРА подсчитывает прибыль, полученную от клиента №0003, которая составляет 193 000 долларов США. Наибольшим из наблюдаемых до сих пор значений прибыли теперь является 193 000 долларов США, клиент №0003.
6. ЗАМЕСТИТЕЛЬ МЕНЕДЖЕРА ПОМОЩНИКУ МЕНЕДЖЕРА: «Самое высокое значение прибыли, которое мы наблюдали до сих пор, составляет 193 000 долларов США, клиент №0003. Изучите эти три файла и определите максимальное значение прибыли».
7. ПОМОЩНИК МЕНЕДЖЕРА подсчитывает прибыль, полученную от клиента №0004, которая составляет 13 000 долларов США. Наибольшим из наблюдаемых до сих пор значений прибыли по-прежнему является 193 000 долларов США, клиент №0001.
8. ПОМОЩНИК МЕНЕДЖЕРА МЛАДШЕМУ МЕНЕДЖЕРУ: «Самое высокое значение прибыли, которое мы наблюдали до сих пор, составляет 193 000 долларов США, клиент №0003. Изучите эти два файла и определите максимальное значение прибыли».
9. МЛАДШИЙ МЕНЕДЖЕР подсчитывает прибыль, полученную от клиента №0005, которая составляет 256 000 долларов США. Наибольшим из наблюдаемых до сих пор значений прибыли теперь является 256 000 долларов США, клиент №0005.
10. МЛАДШИЙ МЕНЕДЖЕР СТАЖЕРУ: «Самое высокое значение прибыли, которое мы наблюдали до сих пор, составляет 256 000 долларов США, клиент №0005. Изучите последний файл и определите максимальное значение прибыли».
11. СТАЖЕР подсчитывает прибыль, полученную от клиента №0006, которая составляет 99 000 долларов США. Наибольшим из наблюдаемых до сих пор значений прибыли по-прежнему является 256 000 долларов США, клиент №0005.
12. СТАЖЕР МЛАДШЕМУ МЕНЕДЖЕРУ: «Наибольшее значение прибыли составляет 256 000 долларов США, клиент №0005».
13. МЛАДШИЙ МЕНЕДЖЕР ПОМОЩНИКУ МЕНЕДЖЕРА: «Наибольшее значение прибыли составляет 256 000 долларов США, клиент №0005».
14. ПОМОЩНИК МЕНЕДЖЕРА ЗАМЕСТИТЕЛЮ МЕНЕДЖЕРА:

«Наибольшее значение прибыли составляет 256 000 долларов США, клиент №0005».

15. ЗАМЕСТИТЕЛЬ МЕНЕДЖЕРА ВИЦЕ-МЕНЕДЖЕРУ: «Наибольшее значение прибыли составляет 256 000 долларов США, клиент №0005».
16. ВИЦЕ-МЕНЕДЖЕР МЕНЕДЖЕРУ: «Наибольшее значение прибыли составляет 256 000 долларов США, клиент №0005».

В этом подходе, показанном на рис. 6.4 (а), используется хвостовая рекурсия. Каждый сотрудник обрабатывает один файл клиента и сравнивает вычисленное значение прибыли, полученной от этого клиента, с самым высоким значением, которое было обнаружено до сих пор. Затем этот сотрудник передает результат сравнения подчиненному. Рекурсия — делегирование задач — происходит после других операций обработки. Процесс работы каждого сотрудника включает следующие шаги:

1. подсчет значения прибыли в одном файле клиента;
2. сравнение этого значения с самым высоким значением прибыли, наблюдаемым начальством в других файлах клиентов;
3. передача оставшихся файлов клиентов подчиненному вместе с наибольшим значением прибыли, наблюдаемым до сих пор;
4. когда подчиненный сообщит самое высокое значение прибыли из всех файлов клиентов, требуется передать это значение начальнику.

Подход 2

При использовании этого подхода сотрудник сначала откладывает один файл, а затем передает оставшиеся файлы подчиненному. В данном случае перед подчиненным не ставится задача подсчета наибольшего значения прибыли из всех файлов, а только из тех, которые были ему переданы. Как и в первой задаче, это упрощает запросы. При использовании тех же данных, что и в первом подходе, цепочка сообщений выглядит следующим образом:

1. МЕНЕДЖЕР ВИЦЕ-МЕНЕДЖЕРУ: «Изучите эти пять файлов и сообщите мне максимальное значение прибыли».
2. ВИЦЕ-МЕНЕДЖЕР ЗАМЕСТИТЕЛЮ МЕНЕДЖЕРА: «Изучите эти четыре файла и сообщите мне максимальное значение прибыли».
3. ЗАМЕСТИТЕЛЬ МЕНЕДЖЕРА ПОМОЩНИКУ МЕНЕДЖЕРА: «Изучите эти три файла и сообщите мне максимальное значение прибыли».
4. ПОМОЩНИК МЕНЕДЖЕРА МЛАДШЕМУ МЕНЕДЖЕРУ: «Изучите эти два файла и сообщите мне максимальное значение прибыли».
5. МЛАДШИЙ МЕНЕДЖЕР СТАЖЕРУ: «Изучите этот файл и сообщите мне максимальное значение прибыли».

6. СТАЖЕР подсчитывает прибыль, полученную от клиента №0006, которая составляет 99 000 долларов США. Это единственный файл, который видел СТАЖЕР, поэтому данное значение прибыли является наибольшим.
7. СТАЖЕР МЛАДШЕМУ МЕНЕДЖЕРУ: «Наибольшее значение прибыли в моем файле составляет 99 000 долларов США, клиент №0006».
8. МЛАДШИЙ МЕНЕДЖЕР подсчитывает прибыль, полученную от клиента №0005, которая составляет 256 000 долларов США. Наибольшим из известных данному сотруднику значений является 256 000 долларов США, клиент №0005.
9. МЛАДШИЙ МЕНЕДЖЕР ПОМОЩНИКУ МЕНЕДЖЕРА: «Наибольшее значение прибыли в моих файлах составляет 256 000 долларов США, клиент №0005».
10. ПОМОЩНИК МЕНЕДЖЕРА подсчитывает прибыль, полученную от клиента №0004, которая составляет 13 000 долларов США. Наибольшим из известных данному сотруднику значений является 256 000 долларов США, клиент №0005.
11. ПОМОЩНИК МЕНЕДЖЕРА ЗАМЕСТИТЕЛЮ МЕНЕДЖЕРА: «Наибольшее значение прибыли в моих файлах составляет 256 000 долларов США, клиент №0005».
12. ЗАМЕСТИТЕЛЬ МЕНЕДЖЕРА подсчитывает прибыль, полученную от клиента №0003, которая составляет 193 000 долларов США. Наибольшим из известных данному сотруднику значений является 256 000 долларов США, клиент №0005.
13. ЗАМЕСТИТЕЛЬ МЕНЕДЖЕРА ВИЦЕ-МЕНЕДЖЕРУ: «Наибольшее значение прибыли в моих файлах составляет 256 000 долларов США, клиент №0005».
14. ВИЦЕ-МЕНЕДЖЕР подсчитывает прибыль, полученную от клиента №0002, которая составляет 68 000 долларов США. Наибольшим из известных данному сотруднику значений является 256 000 долларов США, клиент №0005.
15. ВИЦЕ-МЕНЕДЖЕР МЕНЕДЖЕРУ: «Наибольшее значение прибыли составляет 256 000 долларов США, клиент №0005».
16. МЕНЕДЖЕР подсчитывает прибыль от клиента №0001, которая составляет 172 000 долларов США. Наибольшим из известных данному сотруднику значений является 256 000 долларов США, клиент №0005.

В этом подходе, показанном на рис. 6.4 (б), используется головная рекурсия. Каждый сотрудник по-прежнему должен подсчитать значение прибыли в одном файле клиента, однако это действие откладывается до тех пор, пока подчиненный не определит самое высокое значение в остальных файлах. Процесс работы каждого из сотрудников включает следующие этапы:

1. передача подчиненному сотруднику всех файлов клиентов, кроме одного;
2. получение от подчиненного сотрудника наибольшего значения прибыли среди этих файлов;
3. определение значения прибыли в одном файле клиента;
4. передача начальнику большего из этих значений прибыли.

Как и в задаче с подсчетом количества попугаев, головная рекурсия позволяет каждому сотруднику передать подчиненному минимальный объем информации.

Большая рекурсивная идея

Теперь мы подошли к Большой Рекурсивной Идее. На самом деле, если вы прочитали все этапы решения приведенных выше задач, вы уже видели БРИ в действии.

Как это? В обеих задачах используется некая форма рекурсивного решения. Каждый человек в коммуникационной цепочке выполняет одни и те же действия со все уменьшающимся подмножеством исходных данных. Однако важно отметить, что эти задачи *вообще не предусматривают никакой рекурсии.*

В первой задаче каждый сотрудник железной дороги запрашивает данные у сотрудника следующей станции вниз по линии, и, выполняя этот запрос, следующий сотрудник повторяет те же шаги, что и предыдущий. Однако ничто в формулировке запроса не требует от сотрудника выполнения этих конкретных шагов. Например, когда Арт позвонил Белинде, используя подход 2, он попросил ее подсчитать общее количество попугаев от ее станции до конца линии. Он не требовал применения какого-то определенного метода для вычисления этого значения. Если бы он подумал об этом, то мог бы понять, что Белинде придется произвести те же действия, что и ему, однако ему не нужно это учитывать. Для решения своей задачи Арту было достаточно того, чтобы Белинда сообщила правильный ответ на заданный им вопрос.

Точно так же, во второй проблеме каждый сотрудник старался делегировать подчиненному максимально возможное количество работы. Например, помощник менеджера может хорошо знать младшего менеджера и ожидать, что младший менеджер передаст стажеру все файлы, кроме одного. Однако у помощника менеджера нет причин задумываться о том, обработает ли младший менеджер все оставшиеся файлы или передаст некоторые из них своему подчиненному. Помощник менеджера заботится только о том, чтобы младший менеджер сообщил ему правильный ответ. Поскольку помощник менеджера не собирается повторять действия младшего менеджера, помощник менеджера просто предполагает, что результат, полученный от младшего менеджера является правильным, и использует эти данные для решения общей задачи, которую помощник менеджера получил от заместителя менеджера.

В обеих задачах, когда сотрудники обращаются к другим сотрудникам за информацией, они думают о том, *что*, а не о том, *как*. Вопрос задан; ответ получен. Таким образом, Большая Рекурсивная Идея сводится к следующему: если вы следуете определенным правилам в процессе кодирования, *вы можете сделать вид, что рекурсии не происходит*. Вы даже можете использовать дешевый трюк (описан ниже), чтобы перейти от итеративной реализации к рекурсивной реализации, явно не рассматривая то, как рекурсия на самом деле решает проблему. Со временем вы выработаете интуитивное понимание того, как работают рекурсивные решения, однако, пока этого не произошло, вы можете реализовать рекурсию и быть уверенными в своем коде.

Давайте применим эту концепцию на практике с помощью примера кода.

Задача: вычисление суммы элементов целочисленного массива

Напишите рекурсивную функцию, которой в качестве параметров передан массив целых чисел и размер массива. Эта функция должна возвратить сумму целых чисел в массиве.

Вашей первой мыслью, вероятно, было то, что эту проблему легко можно решить итеративно. Действительно, давайте начнем с итеративного решения этой проблемы:

```
int iterativeArraySum(int integers[], int size) {
   int sum = 0;
   for (int i = 0; i < size; i++) {
      sum += integers[i];
   }
   return sum;
}
```

Вы видели код, очень похожий на этот, в главе 3, поэтому эта функция должна быть простой для понимания. Следующим шагом будет написание кода, который находится посередине между итерационным решением и конечным требуемым рекурсивным решением. Мы оставим итеративную функцию и добавим вторую функцию, которую будем называть *диспетчером*. Он передаст большую часть работы ранее написанной итеративной функции и использует эту информацию для решения общей задачи. Чтобы написать функцию-диспетчер, мы должны следовать двум правилам.

1. Диспетчер должен быть в состоянии полностью справиться с самым тривиальным случаем, не вызывая итеративную функцию.
2. При вызове итеративной функции диспетчер должен передать меньшую версию задачи.

Применяя первое правило к этой задаче, мы должны решить, что собой представляет самый тривиальный случай. Если значение `size` равно 0, то функции концептуально передается массив «null» с суммой

элементов равной 0. Можно поспорить, что самый тривиальный случай должен иметь место, когда значение `size` равно 1. В этом случае в логическом массиве будет только одно число, мы можем вернуть это число в качестве суммы. Любая из этих интерпретаций будет работать, однако выбор в пользу первого варианта позволяет функции обрабатывать специальный случай. Обратите внимание на то, что в исходной итеративной функции не произойдет сбой при значении `size` равном нулю, поэтому поддержание этой гибкости является предпочтительным.

Чтобы применить второе правило к этой задаче, мы должны найти способ передачи меньшей версии задачи от диспетчера итеративной функции. Нет простого способа передачи меньшего массива, однако мы можем легко передать меньшее значение `size`. Если диспетчеру передано значение `size` равное 10, то от функции требуется вычислить сумму 10 значений в этом массиве. Если диспетчер передает итеративной функции значение `size` равное 9, то он запрашивает сумму первых 9 значений в массиве. Затем диспетчер может добавить значение одного оставшегося элемента массива (10-го) для вычисления суммы всех 10 значений. Обратите внимание на то, что уменьшение размера на 1 при вызове итеративной функции максимизирует работу итеративной функции и тем самым минимизирует работу диспетчера. Этот подход всегда является предпочтительным — как менеджеры DelegateCorp, функция-диспетчер старается избежать как можно большего количества работы.

Объединив эти идеи, мы получаем следующую функцию-диспетчер для этой задачи:

```
int arraySumDelegate(int integers[], int size) {
  ❶ if (size == 0) return 0;
  ❷ int lastNumber = integers[size - 1];
    int allButLastSum = ❸iterativeArraySum(integers, size - 1);
  ❹ return lastNumber + allButLastSum;
}
```

Первое утверждение обеспечивает соблюдение первого правила диспетчеров: оно проверяет тривиальный случай и полностью его обрабатывает, в данном примере, возвращая 0 ❶. В противном случае управление переходит к оставшемуся коду, который обеспечивает соблюдение второго правила. Последнее число в массиве хранится в локальной переменной `lastNumber` ❷, а затем сумма всех остальных значений в массиве вычисляется посредством вызова итеративной функции ❸. Этот результат сохраняется в другой локальной переменной `allButLastSum`, и, наконец, функция возвращает сумму значений двух локальных переменных ❹.

Если мы создали функцию-диспетчер правильно, мы уже создали рекурсивное решение. Это Большая Рекурсивная Идея в действии. Превращение этого итеративного решения в рекурсивное требует лишь одного простого шага: нужно сделать так, чтобы функция-делегат вызывала саму себя там, где она ранее вызывала итеративную функцию. Затем мы можем полностью удалить итерационную функцию.

```
int ❶arraySumRecursive(int integers[], int size) {
    if (size == 0) return 0;
    int lastNumber = integers[size − 1];
    int allButLastSum = ❷arraySumRecursive(integers, size − 1);
    return lastNumber + allButLastSum;
}
```

В предыдущий код были внесены только два изменения. Имя функции было изменено для лучшего описания ее новой формы ❶, и теперь функция вызывает сама себя там, где ранее она вызывала итеративную функцию ❷. Логика функций arraySumDelegate и arraySumRecursive одинакова. Каждая функция проверяет тривиальный случай, когда сумма уже известна, в данном примере это массив размером 0 с суммой элементов равной 0. В противном случае каждая функция вычисляет сумму значений в массиве, вызывая функцию для вычисления суммы всех значений, кроме последнего. Наконец, каждая функция добавляет это последнее значение к возвращенной общей сумме. Разница лишь в том, что первая версия функции вызывает другую функцию, в то время как рекурсивная версия вызывает саму себя. БРИ говорит нам о том, что если мы будем следовать изложенным выше правилам написания диспетчера, то сможем проигнорировать это различие.

Вам не обязательно буквально следовать всем приведенным выше шагам для реализации БРИ. В частности, вы обычно не будете применять итерационное решение задачи до реализации рекурсивного решения. Написание итеративной функции в качестве одного из этапов — это дополнительная работа, которая в конечном итоге будет отброшена. Кроме того, рекурсия лучше всего применима к ситуациям, в которых итерационное решение трудно реализуемо, о чем мы поговорим далее. Тем не менее вы можете следовать схеме БРИ без фактического написания итерационного решения. Ключ в том, чтобы рассматривать рекурсивный вызов в качестве вызова другой функции, независимо от свойств этой функции. Таким образом, вы устраняете сложности рекурсивной логики из рекурсивного решения.

Распространенные ошибки

Как показано выше, при правильном подходе рекурсивные решения часто могут быть очень легки в написании. Однако так же легко можно придумать неправильную рекурсивную реализацию, либо рекурсивное решение, которое «работает», но является неуклюжим. Большинство проблем рекурсивной реализации связано с двумя основными ошибками: чрезмерным обдумыванием задачи или началом реализации при отсутствии четкого плана.

Чрезмерное обдумывание рекурсивных задач свойственно программистам-новичкам, поскольку ограниченный опыт и недостаток уверенности при работе с рекурсией заставляют их считать задачу более сложной, чем она есть на самом деле. Код, полученный в результате такого чрезмерного обдумывания, легко узнать по его слишком

аккуратному виду. Например, рекурсивная функция может иметь несколько особых случаев, когда ей требуется только один.

Слишком раннее начало реализации может привести к чрезмерно сложному, «заумному» коду, где непредвиденные взаимодействия приводят к необходимости внесения поправок в исходный код.

Давайте рассмотрим некоторые конкретные ошибки и способы их избежать.

Слишком много параметров

Метод головной рекурсии может уменьшить количество данных, передающихся рекурсивному вызову, тогда как метод хвостовой рекурсии может привести к передаче рекурсивному вызову дополнительных данных. Программисты часто застревают на хвостовой рекурсии, потому что слишком много думают и слишком рано приступают к реализации.

Рассмотрим нашу задачу рекурсивного вычисления суммы элементов целочисленного массива. При написании итеративного решения этой задачи программист знает, что потребуется переменная с «промежуточной суммой» (в предлагаемом итеративном решении я назвал ее `sum`) и что значения массива будут суммироваться, начиная с первого элемента. Учитывая рекурсивное решение, программист, естественно, представляет себе реализацию, которая наиболее непосредственным образом отражает итерационное решение с переменной, в которой хранится промежуточная сумма, и рекурсивным вызовом, обрабатывающим первый элемент в массиве. Однако этот подход требует того, чтобы рекурсивная функция сообщала значение промежуточной суммы и место, где следующий рекурсивный вызов должен начать обработку. Такое решение будет выглядеть следующим образом:

```
int arraySumRecursiveExtraParameters(int integers[],❶int size, ❷int sum,
int currentIndex) {
   if (currentIndex == size) return sum;
   sum += integers[currentIndex];
   return arraySumRecursiveExtraParameters(integers, size, sum,currentIndex
   + 1);
}
```

Этот код так же короток, как и другая рекурсивная версия, но значительно более сложный семантически из-за дополнительных параметров, `sum` ❶ и `currentIndex` ❷. С точки зрения клиентского кода, дополнительные параметры не важны и всегда должны иметь значение 0 в вызове, как показано в этом примере:

```
int a[10] = {20, 3, 5, 22, 7, 9, 14, 17, 4, 9};
int total = arraySumRecursiveExtraParameters(a, 10, 0, 0);
```

Эту проблему можно обойти с помощью *функции-обертки*, как описано в следующем разделе, однако поскольку мы не можем полностью устранить данные параметры, это не лучшее решение. Итеративная функция для этой задачи и исходная рекурсивная функция отвечают на вопрос относительно суммы значений массива с данным

количеством элементов. Напротив, у второй рекурсивной функции спрашивается, какова сумма значений этого массива, если он содержит столько-то элементов, мы начинаем с такого-то конкретного элемента и имеем такую-то сумму всех предшествующих элементов.

Проблему чрезмерного количества параметров можно обойти, если выбрать параметры своей функции до обдумывания рекурсии. Другими словами, заставьте себя использовать тот же список параметров, что и в том случае с итеративным решением. Если вы используете полный процесс БРИ и на самом деле сначала пишите итеративную функцию, то сможете автоматически избежать этой проблемы. Однако если вы не реализуете весь процесс формально, вы все равно сможете использовать эту идею концептуально, если выпишете список параметров, имея в виду итеративную функцию.

Глобальные переменные

Избегая чрезмерного количества параметров, программисты иногда совершают другую ошибку — используют глобальные переменные для передачи данных от одного рекурсивного вызова другому. Использование глобальных переменных, как правило, является плохой практикой программирования, хотя иногда это допустимо из соображений производительности. Использования глобальных переменных в рекурсивных функциях следует избегать, насколько это возможно. Давайте рассмотрим конкретную задачу, чтобы увидеть, как программисты убеждают сами себя совершить эту ошибку. Предположим, нас попросили написать рекурсивную функцию, которая подсчитывает количество нулей в целочисленном массиве. Эту задачу легко решить с помощью итерации:

```
int zeroCountIterative(int numbers[], int size) {
  ❶ int count = 0;
    for (int i = 0; i < size; i++) {
       if (numbers[i] == 0) count ++;
    }
    return count;
}
```

Логика этого кода проста. Мы просто просматриваем массив от первого до последнего элемента, подсчитывая при этом нули и используя локальную переменную, `count` ❶, в качестве трекера. Однако если мы имеем в виду подобную функцию при написании рекурсивной функции, то можем предположить, что нам нужна переменная-трекер и в этой версии кода. Мы не можем просто объявить `count` в качестве локальной переменной в рекурсивной версии, поскольку тогда это будет новая переменная в каждом рекурсивном вызове. Поэтому у нас может возникнуть соблазн объявить ее глобальной переменной:

```
int count;
int zeroCountRecursive(int numbers[], int size) {
   if (size == 0) return count;
   if (numbers[size - 1] == 0) count++;
   zeroCountRecursive(numbers, size - 1);
}
```

Этот код работает, однако глобальная переменная совершенно не нужна и вызывает все проблемы, которые обычно возникают из-за глобальных переменных, такие как плохая читаемость и усложнение обслуживания кода. Некоторые программисты могут попытаться облегчить эту проблему, сделав переменную локальной, но статической:

```
int zeroCountStatic(int numbers[], int size) {
  ❶ static int count ❷= 0;
     if (size == 0) return count;
     if (numbers[size - 1] == 0) count++;
     zeroCountStatic(numbers, size - 1);
}
```

В языке C++ локальная переменная, объявленная как *статическая*, сохраняет свое значение от одного вызова функции к другому; таким образом, локальная статическая переменная `count`❶ будет действовать так же, как глобальная переменная в предыдущей версии. Так в чем проблема? Инициализация этой переменной нулем ❷ происходит только при первом вызове функции. Это необходимо для того, чтобы объявление `static` было полезным, однако это означает, что функция вернет правильный ответ только при первом вызове. Если эту функцию вызывать дважды — сначала с массивом, в котором содержится три нуля, а затем с массивом, в котором содержится пять нулей то функция вернет значение 8 для второго массива, поскольку значение переменной `count` будет начинаться с того места, где оно остановилось.

В данном случае, чтобы избежать использования глобальной переменной, можно применить БРИ. Мы можем предположить, что рекурсивный вызов с меньшим значением `size` вернет правильный результат и на его основе вычислит правильное значение для всего массива. Это приведет к решению, предполагающему головную рекурсию:

```
int zeroCountRecursive(int numbers[], int size) {
     if (size == 0) return 0;
  ❶ int count = zeroCountRecursive(numbers, size - 1);
  ❷ if (numbers[size - 1] == 0) count++;
  ❸ return count;
}
```

В этой функции у нас все еще есть локальная переменная, `count` ❶, но здесь не делается попытки сохранить ее значение от одного вызова к другому. Вместо этого в этой переменной сохраняется значение, возвращаемое из нашего рекурсивного вызова; мы при необходимости инкрементируем эту переменную ❷ перед ее возвращением ❸.

Применение рекурсии к динамическим структурам данных

Рекурсия часто применяется к таким динамическим структурам, как связные списки, деревья и графы. Чем сложнее структура, тем больше процесс кодирования может выиграть от применения рекурсивного решения. Зачастую обработка сложных структур напоминает

поиск пути сквозь лабиринт, а рекурсия позволяет нам возвращаться к предыдущим этапам нашего процесса обработки.

Рекурсия и связные списки

Начнем с самой простой из динамических структур — со связного списка. Для целей обсуждения, приведенного в этом разделе, предположим, что у нас есть простейшая структура узлов для нашего связного списка — всего один тип данных int. Вот наши объявления типов:

```
struct listNode {
   int data;
   listNode * next;
};
typedef listNode * listPtr;
```

Применение БРИ к односвязному списку следует той же общей схеме вне зависимости от конкретной задачи. Рекурсия требует от нас разделения задачи, чтобы мы могли передать рекурсивному вызову уменьшенную версию исходной задачи. Существует только один практический способ разделения односвязного списка — на первый узел в списке и остальную часть списка.

На рис. 6.5 мы видим пример списка, разделенного на неравные части: первый узел и остальные узлы. Концептуально мы можем рассматривать «остальную часть» исходного списка как отдельный список, начинающийся со второго узла исходного списка. Именно эта точка зрения обеспечивает гладкую работу рекурсии.

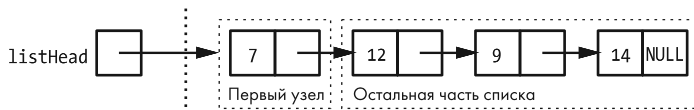


Рис. 6.5. Список, разделенный на первый узел и «остальную часть списка»

Опять же, нам необязательно изображать все этапы рекурсии, чтобы обеспечить ее работу. С точки зрения того, кто пишет рекурсивную функцию для обработки связного списка, это может быть концептуализировано как первый узел, с которым нам приходится иметь дело, и остальная часть списка, с которой мы не будем работать и, следовательно, не обращаем на нее внимание. Такое отношение изображено на рис. 6.6.

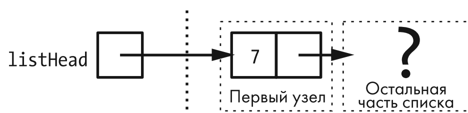


Рис. 6.6. Список, изображенный так, как его должен представлять программист, использующий рекурсию: первый узел и «остальная часть списка» в виде туманной формы, передаваемой рекурсивному вызову

Обеспечив таким образом разделение труда, мы можем сказать, что рекурсивная обработка односвязных списков будет происходить

в соответствии со следующим общим планом. При наличии связного списка L и вопроса Q...

1. При минимальном L мы напрямую присваиваем значение по умолчанию. В противном случае...
2. Используем рекурсивный вызов для получения ответа на Q для «остального» списка L (списка, начинающегося со второго узла L).
3. Проверяем значение в первом узле L.
4. Используем результаты предыдущих двух шагов, чтобы ответить на Q для всего L.

Как вы можете видеть, это прямое применение БРИ с учетом практических ограничений на разбиение связного списка. Теперь давайте применим этот план к конкретной задаче.

Задача: подсчет отрицательных чисел в односвязном списке

Напишите рекурсивную функцию, которой дан односвязный список с целочисленным типом данных. Функция должна возвращать количество отрицательных чисел в этом списке.

Вопрос Q, на который мы хотим ответить: сколько отрицательных чисел в списке? Поэтому наш план можно сформулировать так.

1. Если в списке нет узлов, значением по умолчанию является 0. В противном случае...
2. Используем рекурсивный вызов для подсчета количества отрицательных чисел в «остальной части» списка.
3. Смотрим, является ли значение в первом узле списка отрицательным.
4. Используем результаты предыдущих двух шагов, чтобы определить количество отрицательных чисел во всем списке.
5. Ниже показана реализация функции, непосредственно вытекающая из данного плана:

```
int countNegative(listPtr head) {
    if (head == NULL) return 0;
    int listCount = countNegative(head->next);
    if (head->data < 0) listCount++;
    return listCount;
}
```

Обратите внимание на то, что этот код следует тем же принципам, что и код в предыдущих примерах. Он будет подсчитывать отрицательные числа «в обратном направлении», от конца списка к его началу. Также обратите внимание на то, что в коде используется метод головной рекурсии; мы обрабатываем «остальную часть» списка перед обработкой первого узла. Как и прежде, это позволяет нам

избежать передачи дополнительных данных в рекурсивный вызов и использования глобальных переменных.

Также обратите внимание на то, как первое правило для связного списка, «если список L минимален», интерпретируется в конкретной реализации этой задачи как «если список не имеет узлов». Дело в том, что мы можем сказать, что список без узлов имеет ноль отрицательных значений. Тем не менее в некоторых случаях не существует никакого содержательного ответа на наш вопрос Q для списка без узлов, а минимальным случаем является список с одним узлом. Предположим, наш вопрос звучит так: «Каково наибольшее число в этом списке?» На этот вопрос нельзя ответить в случае списка без значений. Если вы не понимаете, почему, представьте, что вы учитель начальной школы и ваш класс состоит из одних девочек. Если бы директор вашей школы спросил вас, сколько мальчиков из вашего класса поют в хоре, вы могли бы просто ответить «ноль», поскольку в вашем классе нет мальчиков. Если бы директор школы попросил вас назвать самого высокого мальчика в вашем классе, вы не смогли бы дать осмысленный ответ на этот вопрос, поскольку для этого в вашем классе должен быть хотя бы один мальчик. Точно так же, если вопрос о наборе данных требует наличия по крайней мере одного значения, чтобы на него можно было осмысленно ответить, то минимальный набор данных включает один элемент. Однако вы все равно можете возвратить *какой-то* ответ для случая, когда размер равен нулю, хотя бы для обеспечения гибкости при использовании функции и для предотвращения сбоя.

Рекурсия и двоичные деревья

Все примеры, которые мы обсуждали до сих пор, включают не более одного рекурсивного вызова. Однако для более сложных структур может потребоваться несколько рекурсивных вызовов. Чтобы понять, как это работает, рассмотрим структуру, известную как *двоичное дерево*, в котором каждый узел содержит ссылки на «левые» и «правые» узлы. Вот типы, которые мы будем использовать:

```
struct treeNode {
    int data;
    treeNode * left;
    treeNode * right;
};
typedef treeNode * treePtr;
```

Поскольку каждый узел в дереве указывает на два других узла, рекурсивные функции обработки дерева требуют двух рекурсивных вызовов. Мы концептуализировали связные списки как состоящие из двух частей: первого узла и остальной части списка. Для применения рекурсии мы будем концептуализировать деревья как имеющие три части: узел на вершине, известный как *корневой узел*; все узлы, к которым ведет левая ссылка корня, известные как *левое поддерево*; и все узлы, к которым ведет правая ссылка корня, известные как *правое поддерево*. Эта концептуализация показана на рис. 6.7. Как и в случае со связны-

ми списками, мы как разработчики рекурсивного решения просто сосредоточиваемся на существовании левого и правого поддеревьев, не учитывая их содержимое. Это показано на рис. 6.8.

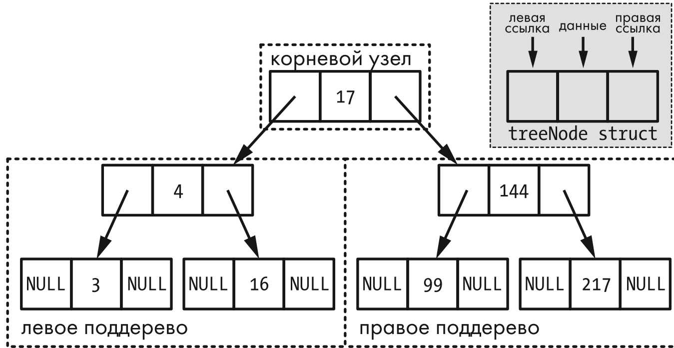


Рис. 6.7. Двоичное дерево, разделенное на корневой узел и левое и правое поддеревья

Как всегда, при рекурсивном решении задачи с двоичными деревьями мы хотим использовать БРИ. Мы сделаем рекурсивные вызовы функций и предположим, что они возвращают правильные результаты, не беспокоясь о том, как рекурсивный процесс решает задачу в целом. Как и в случае со связными списками, мы будем работать с естественным делением двоичного дерева. Эта дает следующий общий план. Чтобы ответить на вопрос Q для дерева T:

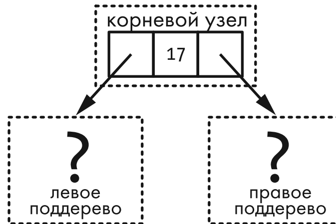


Рис. 6.8. Двоичное дерево, каким его должен представлять программист, использующий рекурсию: корневой узел с левым и правым поддеревьями неизвестной и неучитываемой структуры

1. Если дерево T имеет минимальный размер, следует напрямую присвоить значение по умолчанию. В противном случае...
2. Сделать рекурсивный вызов, чтобы ответить на вопрос Q для левого поддерева T.
3. Сделать рекурсивный вызов, чтобы ответить на вопрос Q для правого поддерева T.
4. Проверить значение в корневом узле дерева T.
5. Использовать результаты предыдущих трех шагов, чтобы ответить на вопрос Q для всего дерева T.

Теперь давайте применим общий план к решению конкретной задачи.

Задача: нахождение наибольшего значения в двоичном дереве

Напишите функцию, которая при передаче двоичного дерева, где каждый узел содержит целое число, возвращает наибольшее целое число в дереве.

Применение общего плана к этой конкретной задаче предполагает выполнение следующих действий:

1. Если корень дерева не имеет дочерних элементов, верните значение в корневой узел. В противном случае...
2. Сделайте рекурсивный вызов, чтобы найти наибольшее значение в левом поддереве.
3. Сделайте рекурсивный вызов, чтобы найти наибольшее значение в правом поддереве.
4. Проверьте значение в корневом узле.
5. Верните наибольшее из значений, найденных на предыдущих трех этапах.

Имея в виду эти шаги, мы можем прямо написать код для решения этой задачи:

```
int maxValue(treePtr root) {
  ❶ if (root == NULL) return 0;
  ❷ if (root->right == NULL && root->left == NULL)
        return root->data;
  ❸ int leftMax = maxValue(root->left);
  ❹ int rightMax = maxValue(root->right);
  ❺ int maxNum = root->data;
     if (leftMax > maxNum) maxNum = leftMax;
     if (rightMax > maxNum) maxNum = rightMax;
     return maxNum;
}
```

Обратите внимание на то, что минимальное дерево для этой задачи представляет собой единственный узел ❷ (хотя случай с пустым деревом учитывается по соображениям безопасности ❶). Это связано с тем, что на вопрос, который мы задаем, можно дать осмысленный ответ с помощью минимум одного значения. Рассмотрим практическую проблему, когда базовым случаем является пустое дерево. Какое значение мы могли бы вернуть? Если мы возвращаем ноль, мы неявно требуем наличия в этом дереве нескольких положительных значений; если все значения в дереве отрицательные, то ноль будет ошибочно возвращен как наибольшее значение в дереве. Мы могли бы решить эту проблему, возвращая наименьшее (самое отрицательное) целое число, но в этом случае нам бы пришлось осторожно адаптировать код для других числовых типов. Сделав базовым случаем единственный узел, мы полностью избегаем необходимости принятия этого решения.

Остальная часть кода проста. Мы используем рекурсию для нахождения максимальных значений в левом ❸ и правом поддере-

вьях ❹. Затем мы находим наибольшее из трех значений (значение в корневом узле, наибольшее значение в левом поддереве, наибольшее значение в правом поддереве), используя вариант алгоритма «Царь горы», который мы использовали на протяжении всей этой книги ❺.

Функции-обертки

В предыдущих примерах в этой главе мы обсуждали только саму рекурсивную функцию. Однако в некоторых случаях эту рекурсивную функцию требуется «настраивать» с помощью второй функции. Чаще всего это происходит, когда мы пишем рекурсивные функции внутри структур классов. Это может привести к несоответствию между параметрами, необходимыми для рекурсивной функции, и параметрами, необходимыми для общедоступного метода класса. Поскольку классы обычно обеспечивают сокрытие информации, код клиента класса может не иметь доступа к данным или типам, который требуется рекурсивной функции. Эта проблема и ее решение показаны в следующем примере.

Задача: нахождение количества листьев в двоичном дереве

Для класса, реализующего двоичное дерево, добавьте общедоступный метод, который возвращает количество листьев (узлов без дочерних элементов) в дереве. Подсчет листьев должен выполняться с использованием рекурсии.

Давайте обрисуем схему того, как мог бы выглядеть этот класс, прежде чем мы попытаемся реализовать решение этой задачи. Для простоты мы будем включать только соответствующие части класса, игнорируя конструкторы, деструктор и даже методы, которые позволили бы нам построить дерево, чтобы сосредоточиться на нашем рекурсивном методе.

```
class binaryTree {
  public:
    ❶ int leafCount();
  private:
       struct binaryTreeNode {
          int data;
          binaryTreeNode * left;
          binaryTreeNode * right;
       };
       typedef binaryTreeNode * treePtr;
       treePtr _root;
};
```

Обратите внимание на то, что наша функция для подсчета листьев не принимает параметров ❶. С точки зрения интерфейса это корректно. Рассмотрим образец вызова для ранее построенного объекта `binaryTree bt`:

```
int numLeaves = bt.leafCount();
```

В конце концов, если мы спрашиваем у дерева, сколько у него листьев, то какую информацию мы могли бы предоставить этому объекту, которую он еще о себе не знает? Так же, как это является правильным для интерфейса, это не является правильным для рекурсивной реализации. Если не существует параметра, что же меняется от одного рекурсивного вызова к другому? В этом случае ничто не может измениться, кроме как благодаря глобальным переменным, использования которых, как говорилось ранее, следует избегать. Если ничего не изменяется, то для прогресса или прекращения рекурсии нет возможности.

Чтобы обойти эту проблему, нужно сначала написать рекурсивную функцию, концептуализируя ее как функцию вне класса. Другими словами, мы напишем эту функцию для подсчета листьев в двоичном дереве в том же стиле, который мы использовали при написании функции для нахождения наибольшего значения в двоичном дереве. Единственным параметром, который нам необходимо передать, является указатель на нашу структуру узла.

Это дает нам еще одну возможность использовать БРИ. В чем заключается вопрос Q в данном случае? Сколько листьев в дереве? Применение общего плана рекурсивной обработки двоичных деревьев к этой конкретной задаче приводит к следующей последовательности шагов.

1. Если корень дерева не имеет дочерних элементов, то дерево имеет только один узел. Этот узел является листом по определению, поэтому возвратите 1. В противном случае...
2. Сделайте рекурсивный вызов для подсчета листьев в левом поддереве.
3. Сделайте рекурсивный вызов для подсчета листьев в правом поддереве.
4. В данном случае нет необходимости проверять корневой узел, поскольку, если мы добрались до этого этапа, значит, корневой узел не может быть листом. Поэтому...
5. Возвратите сумму значений, полученных на этапах 2 и 3.

Переведя этот план в код, мы получим следующее:

```
struct binaryTreeNode {
   int data;
   binaryTreeNode * left;
   binaryTreeNode * right;
};
typedef binaryTreeNode * treePtr;
int leafCount(treePtr rootPtr) {
   if (rootPtr == NULL) return 0;
   if (rootPtr->right == NULL && rootPtr->left == NULL)
       return 1;
   int leftCount = leafCount(rootPtr->left);
```

```
  int rightCount = leafCount(rootPtr->right);
  return leftCount + rightCount;
}
```

Как вы можете видеть, код представляет собой прямой перевод плана. Вопрос в том, как мы переходим от этой независимой функции к тому, что мы можем использовать в классе? Именно здесь неосторожный программист может легко попасть в неприятности, посчитав, что нам нужно использовать глобальную переменную или сделать корневой указатель общедоступным. Нам не нужно этого делать; мы можем оставить все внутри класса. Трюк заключается в использовании *функции-обертки*. Сначала мы помещаем независимую функцию с параметром `treePtr` в приватный раздел нашего класса. Затем мы пишем публичную функцию, которая будет служить «функцией-оберткой» для приватной функции. Поскольку публичная функция имеет доступ к приватному элементу данных `_root`, она может передать его рекурсивной функции, а затем возвратить клиенту результаты следующим образом:

```
  class binaryTree {
    public:
      int publicLeafCount();
    private:
      struct binaryTreeNode {
          int data;
          binaryTreeNode * left;
          binaryTreeNode * right;
      };
      typedef binaryTreeNode * treePtr;
      treePtr _root;
      int privateLeafCount(treePtr rootPtr);
  };
❶ int binaryTree::privateLeafCount(treePtr rootPtr) {
    if (rootPtr == NULL) return 0;
    if (rootPtr->right == NULL && rootPtr->left == NULL)
        return 1;
    int leftCount = privateLeafCount(rootPtr->left);
    int rightCount = privateLeafCount(rootPtr->right);
    return leftCount + rightCount;
  }
❷ int binaryTree::publicLeafCount() {
    ❸ return privateLeafCount(_root);
  }
```

Несмотря на то, что язык C++ позволяет обеим функциям иметь одно и то же имя, для ясности я использовал разные имена, чтобы различать публичную и приватную функции для «подсчета листьев». Код в функции `privateLeafCount` ❶ точно повторяет код в нашей предыдущей, независимой функции `leafCount`. Функция-обертка `publicLeafCount` ❷ проста. Она вызывает функцию `privateLeafCount`, передавая приватный элемент данных `_root`, и возвращает результат ❸. По сути, она стимулирует рекурсивный процесс. Функции-обертки очень полезны при написании рекурсивных функций внутри классов, однако их можно использовать в любое время при несоответствии между списком параметров, которые требуются функции, и желаемым списком параметров вызывающей функции.

В каких случаях использовать рекурсию

Начинающие программисты часто задаются вопросом, зачем использовать рекурсию. Возможно, они уже узнали, что любую программу можно создать, используя базовые управляющие структуры, такие как выбор (утверждения `if`) и итерация (циклы `for` и `while`). Если рекурсию использовать труднее, чем базовые управляющие структуры, и она не является необходимой, возможно, ее следует просто игнорировать.

На этого существует несколько возражений. Во-первых, рекурсивное программирование помогает программистам думать рекурсивно, а рекурсивное мышление применяется повсюду в мире информатики в таких областях, как проектирование компилятора. Во-вторых, некоторые языки просто требуют использования рекурсии, поскольку в них отсутствуют некоторые элементарные управляющие структуры. Чистые версии языка Lisp, например, требуют рекурсии почти в каждой нетривиальной функции.

Тем не менее остается вопрос: если программист изучил рекурсию достаточно для того, чтобы ее понять, и использует такой полнофункциональный язык, как C++, Java или Python, следует ли ему применять рекурсию? Имеет ли рекурсия практическую ценность в таких языках, или это просто упражнение для ума?

Аргументы против рекурсии

Чтобы изучить этот вопрос, давайте перечислим недостатки рекурсии.

Концептуальная сложность

В большинстве случаев среднему программисту бывает сложнее решить задачу с помощью рекурсии. Даже если вы понимаете Большую Рекурсивную Идею, в большинстве ситуаций будет проще написать код с использованием циклов.

Производительность

Вызовы функций очень требовательны к ресурсам компьютера. Рекурсия предусматривает множество вызовов функций и, следовательно, может приводить к замедлению работы системы.

Требования к пространству

Рекурсия не просто предусматривает много вызовов функций; она также вкладывает их один в другой. То есть в итоге вы можете получить длинную цепочку вызовов функций, ожидающих завершения других вызовов. Каждый вызов функции, который начался, но еще не закончился, занимает дополнительное пространство в системном стеке.

На первый взгляд этот список свойств представляет собой сильное обвинение против рекурсии, характеризуя ее как сложный, медленный и требовательный к пространству метод. Тем не менее эти аргументы работают не во всех случаях. Поэтому самым базовым

правилом для выбора между рекурсией и итерацией является следующее: *выбирайте рекурсию, когда эти аргументы не работают.*

Рассмотрим нашу функцию, которая подсчитывает количество листьев в двоичном дереве. Как бы вы решили эту задачу без рекурсии? Это возможно, но вам бы понадобился явный механизм для поддержания «тропы из хлебных крошек» для обозначения узлов, дочерние элементы которых в левом поддереве уже были посещены, а в правом — нет. К этим узлам пришлось бы вернуться в какой-то момент, чтобы иметь возможность посетить дочерние элементы в правом поддереве. Вы могли бы хранить эти узлы в такой динамической структуре, как стек. Для сравнения далее приведена реализация функции, которая использует класс stack из стандартной библиотеки шаблонов C++:

```
int binaryTree::stackBasedCountLeaves() {
    if (_root == NULL) return 0;
    int leafCount = 0;
  ❶stack< ❷binaryTreeNode *> nodes;
  ❸nodes.push(_root);
    while (❹!nodes.empty()) {
        treePtr currentNode = ❺nodes.top();
      ❻nodes.pop();
        if (currentNode->left == NULL && currentNode->right == NULL)
            leafCount++;
        else {
           if (currentNode->right != NULL) wnodes.push(currentNode->right);
           if (currentNode->left != NULL) wnodes.push(currentNode->left);
        }
  }
  return leafCount;
}
```

Этот код следует той же схеме, что и оригинал, однако, если вы никогда раньше не использовали класс stack, здесь будут уместны некоторые комментарии. Класс stack работает как системный стек, который мы обсуждали в главе 3; вы можете добавлять и удалять элементы только на вершине. Обратите внимание на то, что мы могли бы выполнить операцию подсчета листьев, используя любую структуру данных, которая не имеет фиксированного размера. Например, мы могли бы использовать вектор, однако использование стека лучше всего отражает исходный код. Когда мы объявляем стек ❶, то указываем тип элементов, которые собираемся в нем хранить. В данном случае мы будем хранить указатели на структуру binaryTreeNode ❷. В этом коде мы будем использовать четыре метода класса stack. Метод push ❸ помещает элемент (в данном случае указатель на узел) на вершину стека. Метод empty ❹ говорит нам, остались ли в стеке какие-либо элементы. Метод top ❺ предоставляет нам копию элемента на вершине стека, а метод pop ❻ удаляет верхний элемент из стека.

Данный код решает задачу, помещая на стек указатель на первый узел, а затем последовательно удаляя из стека указатель на узел,

проверяя, является ли он листом, увеличивая счетчик на 1, если это так, и помещая на стек указатели на дочерние узлы, если они существуют. Таким образом, стек отслеживает узлы, которые мы обнаружили, но еще не обработали, так же, как цепочка рекурсивных вызовов в рекурсивной версии отслеживает узлы, которые мы должны повторно посетить. Сравнивая эту итеративную версию с рекурсивной, мы видим, что ни одно из стандартных возражений против рекурсии не имеет в данном случае достаточной силы. Во-первых, этот код длиннее и сложнее, чем его рекурсивная версия, поэтому против рекурсивной версии нельзя выдвинуть аргумент, связанный с концептуальной сложностью. Во-вторых, посмотрите, сколько вызовов функций совершает `stackBasedCountLeaves` — на каждое посещение внутреннего узла (то есть узла, который не является листом) эта функция совершает до пяти вызовов функций: по одному для методов `empty`, `top` и `pop` и один или два для метода `push`. Рекурсивная версия совершает только два рекурсивных вызова для каждого внутреннего узла. (Обратите внимание на то, что мы можем избежать вызовов функцией объекта стека, включив в функцию логику стека. Однако это еще больше увеличит сложность функции.) В-третьих, хотя эта итеративная версия не использует дополнительное пространство системного стека, она явно использует приватный стек. Справедливости ради следует сказать, что при этом расход пространства меньше, чем расход пространства системного стека при рекурсивных вызовах, однако это по-прежнему расходование системной памяти, пропорциональное максимальной глубине двоичного дерева, которое мы обходим.

Поскольку в данной ситуации возражения против рекурсии смягчаются или сводятся к минимуму, рекурсия — хороший выбор для решения этой задачи. В общем случае, если задачу легко решить итеративно, то итерация должна быть вашим первым выбором. Рекурсию следует использовать, когда решение с помощью итерации представляется сложным. Часто это требует использования описанного здесь механизма создания «тропы из хлебных крошек». Обход ветвящихся структур, таких как деревья и графы, по своей сути рекурсивен. Обработка линейных структур, таких как массивы и связные списки, обычно не требует рекурсии, однако существуют исключения. Вы никогда не ошибетесь, если начнете решать задачу с помощью итерации и посмотрите, как далеко вас это заведет. В качестве последних примеров рассмотрим следующие задачи со связными списками.

Задача: отображение элементов связного списка в прямом порядке

Напишите функцию, которой передается указатель на головной элемент односвязного списка, где типом данных каждого узла является целое число, и которая отображает эти целые числа по одному на строке в порядке их следования в списке.

Задача: отображение элементов связного списка в обратном порядке

Напишите функцию, которой передается указатель на головной элемент односвязного списка, где типом данных каждого узла является целое число, и которая отображает эти целые числа по одному на строке в порядке, обратном порядку их следования в списке.

Поскольку эти задачи являются зеркальными версиями друг друга, естественно предположить, что их решения также будут зеркальными. Это действительно так в случае рекурсивных реализаций. Далее приведены рекурсивные функции для решения обеих этих задач с использованием приведенных ранее типов `listNode` и `listPtr`:

```
void displayListForwardsRecursion(listPtr head) {
   if (head != NULL) {
      ❶ cout << head->data << "\n";
      ❷ displayListForwardsRecursion(head->next);
   }
}
void displayListBackwardsRecursion(listPtr head) {
   if (head != NULL) {
      ❸ displayListBackwardsRecursion(head->next);
      ❹ cout << head->data << "\n";
   }
}
```

Как видите, код в этих функциях одинаков, за исключением порядка следования двух утверждений внутри инструкции `if`. В этом все различие. В первом случае мы отображаем значение в первом узле ❶, прежде чем делать рекурсивный вызов для отображения остальной части списка ❷. Во втором случае мы совершаем вызов для отображения остальной части списка, ❸ прежде чем отображать значение в первом узле ❹. Благодаря этому порядок отображения оказывается обратным.

Поскольку обе эти функции одинаково лаконичны, можно предположить, что рекурсия уместна для решения обеих этих задач, однако это не так. Чтобы убедиться в этом, давайте рассмотрим итеративные реализации обеих этих функций.

```
void displayListForwardsIterative(listPtr head) {
  ❶ for (listPtr current = head; current != NULL; current = current->next)
        cout << current->data << "\n";
}
void displayListBackwardsIterative(listPtr head) {
  ❷ stack<listPtr> nodes;
  ❸ for (listPtr current = head; current != NULL; current = current->next)
        nodes.push(current);
  ❹ while (!nodes.empty()) {
     ❺ nodePtr current = nodes.top();
     ❻ nodes.pop();
     ❼ cout << current->data << "\n";
  }
}
```

Функция для отображения элементов списка в прямом порядке — это не что иное, как цикл прямого обхода ❶, вроде обсуждаемого нами в главе 4. Однако функция для отображения элементов списка в обратном порядке сложнее. Она страдает от той же необходимости в использовании «тропы из хлебных крошек», как и наши задачи с двоичным деревом. Отображение узлов связного списка в обратном порядке по определению требует возврата к предыдущим узлам. В случае с односвязным списком этого нельзя сделать, используя сам список, поэтому требуется вторая структура. В данном примере нам нужен другой стек. После объявления стека ❷ мы помещаем все узлы нашего связного списка в стек, используя цикл `for` ❸. Поскольку это стек, в котором каждый элемент добавляется поверх предыдущих элементов, первый элемент в связном списке будет находиться внизу стека, а последний элемент связного списка — на его вершине. Мы вводим цикл `while`, который выполняется до тех пор, пока стек не опустеет ❹, последовательно захватываем указатель на верхний узел в стеке ❺, удаляем из стека этот указатель на узел ❻, а затем отображаем данные в узле, на который ссылаемся ❼. Поскольку данные на вершине стека являются последними данными в связном списке, в результате данные связного списка оказываются отображенными в обратном порядке.

Как и в случае с показанной ранее итеративной функцией двоичного дерева, эту функцию можно было написать без использования стека (путем создания второго списка в пределах функции, которая является обратной версией оригинала). Тем не менее нет никакого способа сделать вторую функцию такой же простой, как и первая, или избежать фактического обхода двух структур вместо одной. Сравнивая рекурсивную и итерационную реализации, нетрудно заметить, что итеративная «прямая» функция настолько проста, что у использования рекурсии отсутствуют практические преимущества и при наличии нескольких практических недостатков. Напротив, рекурсивная «обратная» функция проще, чем итеративная версия, и от нее следует ожидать примерно такой же производительности, как от итеративной версии. Таким образом, «обратная» функция представляет собой разумное использование рекурсии, в то время как «прямая» функция, хоть и является хорошим упражнением в рекурсивном программировании, не может считаться хорошим примером практического применения рекурсии.

Упражнения

Как всегда, вам настоятельно рекомендуется опробовать идеи, представленные в данной главе!

6.1. Напишите функцию для вычисления суммы только положительных чисел в целочисленном массиве. Сначала решите эту задачу, используя итерацию. Затем, используя технику, описанную в этой главе, преобразуйте вашу итеративную функцию в рекурсивную.

6.2. Рассмотрите массив, представляющий двоичную строку, где значение каждого элемента равно либо 0, либо 1. Напишите функцию

`bool`, чтобы проверить двоичную строку на нечетность (установить, содержит ли двоичная строка нечетное количество единичных битов). Подсказка: помните, что рекурсивная функция возвратит `true` (нечетность) или `false` (четность), а не количество единичных битов. Решите эту задачу, используя сначала итерацию, а затем рекурсию.

6.3. Напишите функцию, которой передается целочисленный массив и «целевое» число и которая возвращает количество случаев, когда это целевое число встречается в массиве. Решите эту задачу, используя сначала итерацию, а затем рекурсию.

6.4. Придумайте свою собственную задачу: найдите задачу на обработку одномерного массива, которую вы уже решали или тривиальную, учитывая ваш текущий уровень мастерства, и решите ее с помощью рекурсии.

6.5. Решите задачу 6.1 снова, используя связный список вместо массива.

6.6. Решите задачу 6.2 снова, используя связный список вместо массива.

6.7. Решите задачу 6.3 снова, используя связный список вместо массива.

6.8. Придумайте свою собственную задачу: попробуйте найти задачу на обработку связных списков, которая трудно решается с помощью итерации, но может быть решена с использованием рекурсии.

6.9. Некоторые слова в программировании имеют более одного значения. В главе 4 мы узнали о куче, из которой мы выделяем память с помощью оператора `new`. Термин *куча* также описывает двоичное дерево, в котором значение каждого узла выше любого значения в левом или правом поддереве. Напишите рекурсивную функцию, чтобы определить, является ли двоичное дерево кучей.

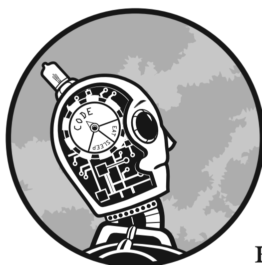
6.10. *Двоичное дерево поиска* — это двоичное дерево, в котором значение каждого узла больше любого значения в левом поддереве этого узла, но меньше любого значения в правом поддереве узла. Напишите рекурсивную функцию, чтобы определить, является ли двоичное дерево двоичным деревом поиска.

6.11. Напишите рекурсивную функцию, которой передается корневой указатель двоичного дерева поиска и новое значение, подлежащее вставке, и которая создает новый узел с новым значением, помещая его в нужное место для поддержания структуры двоичного дерева поиска. Подсказка: попробуйте сделать параметр корневого указателя опорным параметром.

6.12. Придумайте свою собственную задачу: рассмотрите простые статистические показатели для набора числовых значений, например среднее арифметическое, медиана, мода и т.д. Попробуйте написать рекурсивные функции для вычисления этих статистических показателей для двоичного дерева, состоящего из целых чисел. Некоторые из них легче написать, чем другие. Почему?

7

Решение задач с помощью повторного использования кода



Эта глава сильно отличается от предыдущих. Ранее я подчеркивал важность нахождения собственных решений для задач. В конце концов, именно этому и посвящена данная книга — написанию оригинальных решений для задач в области программирования. Тем не менее даже в предыдущих главах мы говорили о том, что вы всегда учитесь на написанном ранее коде, и поэтому вам следует сохранять весь этот код на будущее. В этой главе мы сделаем еще один шаг вперед и поговорим о том, как использовать код и идеи других программистов для решения стоящих перед нами задач.

Если вы помните, с чего началась эта книга, то эта тема может показаться странным вкраплением. Вначале я говорил о том, какой ошибкой была бы попытка решения сложных задач путем модифи-

кации чужого кода. Мало того, что такой подход имеет мало шансов на успех, но даже когда он срабатывает, вы не приобретаете обучающего опыта. И если это все, что вы умеете, вы никогда не станете настоящим программистом и мало чем сможете обогатить сферу разработки программного обеспечения. Тем не менее, когда задача создания программы является достаточно сложной, неразумно ожидать от программиста того, что он будет разрабатывать ее решение с нуля. Это неэффективное использование времени программиста, и оно необоснованно подразумевает, что программист является экспертом во всем. Кроме того, это скорее приведет к созданию программы полной ошибок или сложной в плане сопровождения.

Хорошее и плохое повторное использование кода

Нам следует различать хорошее повторное использование кода, позволяющее нам писать программы лучше и быстрее, и плохое повторное использование кода, позволяющее на какое-то время притвориться программистом, но в конечном итоге приводящее к плохому развитию как кода, так и самого программиста. В табл. 7.1 приведены эти различия. В левом столбце перечислены свойства хорошего повторного использования кода, а в правом столбце — плохого. При рассмотрении вопроса о возможности повторного использования кода спросите себя, какие из свойств вы, скорее всего, продемонстрируете — из правого или левого столбца.

Табл. 7.1. Хорошее и плохое повторное использование кода

Хорошее повторное использование кода	Плохое повторное использование кода
Следование схеме	Копирование чужой работы
Усиливает и расширяет ваши возможности	Фальсифицирует ваши возможности
Помогает вам научиться	Позволяет избежать обучения
Экономит время в краткосрочной и долгосрочной перспективе	Может сэкономить время в краткосрочной перспективе, но удлинить рабочий процесс в долгосрочной перспективе
Результат — рабочая программа	Может привести к созданию программы, которая в любом случае не работает

Важно отметить, что различие между хорошим и плохим повторным использованием кода зависит не от самого кода или того, как вы его используете, а от вашего отношения к этому коду и концепциям, которые вы заимствуете. Однажды при написании курсовой работы по литературе я обнаружил, что часть знаний, полученных при прохождении курса, имела отношение к теме моей работы, поэтому я включил их в нее. Когда я показал черновик своей работы профессору, она сказала, что я должен был оформить эту информацию в качестве цитаты. В замешательстве я спросил своего профессора, в какой момент я могу просто использовать свои знания в работе, не указывая ссылку на источник. Она ответила, что я смогу

перестать ссылаться на других при использовании знаний, находящихся в моей голове, когда я стану таким экспертом, что другие начнут ссылаться на меня.

Выражаясь в терминах программирования, хорошее повторное использование кода имеет место, когда вы сами пишете код, основываясь на чьем-то описании общей концепции, или когда вы используете код, который могли бы написать сами. На протяжении всей этой главы мы будем говорить о том, как вы можете усвоить концепции создания кода, чтобы быть уверенными, что повторное использование помогает вам стать более умелым, а не более ленивым программистом.

Позвольте мне также обратить внимание на последнюю строку табл. 7.1. Попытки плохого повторного использования кода часто оказываются неудачными. Это неудивительно, потому что при этом программист использует код, который он или она фактически не понимает. В некоторых ситуациях заимствованный код поначалу работает, однако когда программист пытается изменить или расширить заимствованную кодовую базу, отсутствие глубокого понимания не позволяет подойти к задаче организованно. Поэтому программист начинает действовать наобум путем проб и ошибок, тем самым нарушая первое и самое главное из наших общих правил решения задач, которое заключается в том, чтобы всегда иметь план.

Основы работы с компонентами

Теперь, когда мы знаем, к какому результату стремимся, давайте классифицируем различные способы повторного использования кода. В этой книге я буду использовать термин *компонент* для обозначения того, что было создано одним программистом и что может быть повторно использовано другим для решения задачи программирования. Компоненты могут существовать в любом месте континуума от абстрактного до конкретного, от идеи до полностью реализованного кода. Если мы подумаем о решении задачи программирования, как о проекте в стиле «сделай сам», то изученные нами методы для решения задач будут похожи на инструменты, а компоненты — на специальные части. Описанные далее компоненты представляют собой различные способы повторного использования результатов работы других программистов.

Кодовый блок

Кодовый блок представляет собой не что иное, как блок кода, скопированный из одного листинга программы в другой. На сленге это называется «*копипастой*». Это самая низкоуровневая форма использования компонентов и часто пример плохого повторного использования кода со всеми вытекающими из этого проблемами. Конечно, если код, который вы копируете, ваш собственный, это не наносит

реального вреда, правда, вы можете рассмотреть возможность упаковки существующего кода в виде библиотеки классов или другой структуры, чтобы его можно было повторно использовать более чистым и удобным образом.

Алгоритмы

Алгоритм — это программный рецепт; особый метод достижения цели, который выражается либо простым языком, либо наглядно, как в блок-схеме. Например, в главе 3 мы обсудили операцию *сортировки* массивов и различные способы ее реализации. Одним из таких способов является алгоритм сортировки вставками, и я продемонстрировал примерную реализацию этого алгоритма. Важно отметить, что данный код представлял собой одну из реализаций такой сортировки, однако она сама по себе является алгоритмом — способом сортировки массива, а не конкретным кодом. При сортировке вставками берется каждое последующее неотсортированное значение в массиве, а отсортированные значения многократно сдвигаются на одну позицию, пока не освободится место, подходящее для значения, подлежащего вставке. Любой код, который использует этот метод сортировки массива, представляет собой сортировку вставками.

Алгоритмы — это высокоуровневая форма повторного использования кода, и обычно они характеризуются хорошими свойствами этого процесса. Алгоритмы — это, по сути, просто идеи, и вам как программисту необходимо реализовать идеи, используя свои навыки программирования и глубокое понимание самого алгоритма. Алгоритмы, которые вы обычно будете использовать, хорошо изучены и имеют предсказуемую производительность в различных ситуациях. С алгоритмом в качестве схемы вы можете быть уверены в правильности вашего кода и его эффективности.

Тем не менее у использования алгоритма в качестве основы для кода существуют некоторые недостатки. Когда вы используете алгоритм, вы начинаете с концептуального уровня. Поэтому впереди вас ожидает длинный путь до готового кода для конкретного раздела программы. Алгоритм, безусловно, экономит время, поскольку этап решения задачи по сути завершен, однако в зависимости от алгоритма и его конкретного применения в вашей работе, реализация алгоритма может оказаться нетривиальной задачей.

Шаблоны

В программировании *шаблон* (или *шаблон проектирования*) представляет собой схему для конкретного метода программирования. Эта концепция напоминает алгоритм, но отличается от него. Алгоритмы подобны рецептам для решения конкретных задач, а шаблоны — это общие методы, используемые в конкретных ситуациях в процессе программирования. Задачи, которые решают шаблоны, обычно

входят в структуру самого кода. Например, в главе 6 мы обсудили проблему, представленную рекурсивной функцией в классе связных списков: рекурсивной функции был нужен «головной» указатель на первый узел в списке в качестве параметра, однако эти данные должны были оставаться приватными. Решение заключалось в создании *обертки*, функции, которая адаптировала бы один список параметров к другому. Техника, предполагающая использование обертки, представляет собой шаблон проектирования. Мы можем использовать этот шаблон для решения проблемы рекурсивной функции в классе, однако его можно использовать и другими способами. Например, предположим, что у нас есть класс `linkedList`, который позволяет вставлять или удалять элементы в любой точке списка, но нам нужен класс `stack`, то есть список, который позволяет вставлять и удалять элементы только на одном конце. Мы могли бы создать новый класс `stack`, который предусматривал бы общедоступные методы для типичных операций стека, таких как `push` и `pop`. Эти методы просто вызывали бы функции-члены на объекте `linkedList`, являющемся приватным членом данных нашего класса `stack`. Таким образом, мы бы повторно использовали функциональность класса связного списка, предоставляя при этом интерфейс класса `stack`.

Как и алгоритмы, шаблоны представляют собой высокоуровневую форму использования компонентов, а изучение шаблонов является отличным способом пополнения вашего профессионального инструментария. Однако шаблоны разделяют некоторые из потенциальных проблем алгоритмов. Знать о существовании шаблона — это не то же самое, что уметь реализовывать его на определенном языке, который вы выбрали для решения задачи программирования, кроме того, шаблоны часто бывает сложно реализовать правильно или с максимальной производительностью. Например, существует шаблон, известный как *одиночка (singleton)*, представляющий собой класс, который позволяет создавать только один объект класса. Создать класс-одиночку легко, однако создать класс-одиночку, который не создает единственный допустимый экземпляр объекта до тех пор, пока он на самом деле не понадобится, бывает на удивление сложно, а лучший способ решения этой задачи может отличаться в зависимости от языка.

Абстрактные типы данных

Абстрактный тип данных, как говорилось в главе 5, представляет собой тип, определяемый его операциями, а не тем, как эти операции выполняются. Хорошим примером является тип стек, который мы уже несколько раз использовали в этой книге. Абстрактные типы данных похожи на шаблоны в плане определения эффектов операций, но они конкретно не определяют, как эти операции реализуются. Тем не менее, как и в случае с алгоритмами, для этих операций существуют хорошо известные методы реализации. Например, стек может быть реализован с использованием любого количества таких основополагающих структур данных, как связный список или

массив. Однако когда мы выбираем конкретную структуру данных, решения, связанные с ее реализацией, иногда оказываются уже принятыми. Предположим, мы реализовали стек с помощью связного списка и не можем обернуть существующий связный список, но нам требуется написать собственный код списка. Поскольку стек является структурой типа «последним пришел — первым ушел» (last-in-first-out), нам имеет смысл добавлять и удалять элементы только на одном конце связанного списка. Кроме того, имеет смысл добавлять и удалять элементы только в начале списка. Теоретически вы можете добавлять и удалять элементы в конце, однако это приведет к неэффективному обходу всего списка при каждой вставке или удалении. Чтобы избежать этих обходов, потребуется двусвязный список с отдельным указателем на последний узел в списке. Вставка и удаление элементов в начале списка делает возможной простейшую, наиболее эффективную реализацию, поэтому практически все реализации стеков на основе связных списков одинаковы.

Таким образом, хотя слово *абстрактный* в названии *абстрактный тип данных* означает, что тип является концептуальным и не содержит деталей, относящихся к реализации, на практике, если вы решите реализовать абстрактный тип данных в своем коде, вам не придется разрабатывать реализацию с нуля. Вместо этого в качестве руководства вам послужат существующие реализации данного типа.

Библиотеки

В программировании *библиотека* представляет собой набор связанных фрагментов кода. Как правило, библиотека включает код в скомпилированной форме вместе с необходимыми объявлениями исходного кода. Библиотеки могут включать автономные функции, классы, объявления типов или что-либо еще, что может использоваться в коде. В языке C++ наиболее очевидными примерами являются стандартные библиотеки. Функция `strcmp`, которую мы использовали в предыдущих главах, была взята из старой библиотеки языка C *cstring*, такие контейнерные классы, как `vector`, — из стандартной библиотеки шаблонов C ++ и даже макрос `NULL`, который мы использовали во всем коде на основе указателей, не является частью языка C++, а определяется в заголовочном файле библиотеки, *stdlib.h.* Поскольку так много базовых функций содержится в библиотеках, их использование в современном программировании неизбежно.

Вообще, использование библиотеки является примером хорошего повторного использования кода. Код включен в библиотеку, поскольку он обеспечивает функциональность, которая обычно необходима в различных программах, — библиотечный код помогает программистам избежать необходимости «изобретать колесо». Тем не менее как разработчики программ при использовании библиотечного кода мы должны стремиться научиться на своем опыте, а не просто использовать обходной путь. Далее в этой главе мы рассмотрим пример.

Обратите внимание на то, что, хотя многие библиотеки являются библиотеками общего назначения, другие разработаны как *интерфейсы прикладного программирования (API)*, предоставляющие программисту, работающему с высокоуровневым языком, упрощенное или более последовательное представление об основополагающей платформе. Например, язык Java включает в себя API под названием JDBC, который предоставляет классы, позволяющие программам взаимодействовать с реляционными базами данных стандартным способом. Другой пример — это DirectX, который предоставляет программистам игр Microsoft Windows обширную функциональность для работы со звуком и графикой. В обоих случаях библиотека обеспечивает связь между высокоуровневой программой и фундаментальным аппаратным и программным обеспечением — с движком базы данных в случае JDBC и графическим и звуковым оборудованием в случае DirectX. Более того, в обоих случаях повторное использование кода не только допустимо, но и желательно для всех практических целей. Программист базы данных, работающий с языком Java, или графический программист, пишущий код C++ для Windows, будет использовать API, если не перечисленные выше API, то что-то еще, но программист не будет с нуля разрабатывать новое соединение с платформой.

Изучение компонентов

Компоненты настолько полезны, что программисты используют их при любой возможности. Тем не менее, чтобы использовать компонент для решения задачи, программист должен знать о его существовании. В зависимости от того, насколько точно вы их определяете, доступные компоненты могут исчисляться сотнями или даже тысячами, а начинающий программист будет знать только о некоторых из них. Поэтому хороший программист всегда должен стараться обогащать свои знания о компонентах. Такое накопление знаний происходит двумя разными способами: программист может специально выделить время для изучения новых компонентов или поискать компонент для решения конкретной задачи. Мы будем называть первый подход *исследовательским обучением*, а второй — *обучением по мере необходимости*. Чтобы развиваться как программисту, вам нужно будет использовать оба подхода. После освоения синтаксиса выбранного вами языка программирования, изучение новых компонентов — один из основных способов самосовершенствования в качестве программиста.

Исследовательское обучение

Давайте начнем с примера исследовательского обучения. Предположим, нам нужно узнать больше о шаблонах проектирования. К счастью, существует общее соглашение относительно того, какие шаблоны проектирования являются наиболее полезными или часто

используемыми, поэтому мы могли бы начать с любого количества ресурсов по этой теме и быть достаточно уверенными в том, что не упустим ничего важного. Мы бы выиграли от простого нахождения списка шаблонов проектирования и его изучения, однако мы бы достигли большего понимания, если бы реализовали некоторые из этих шаблонов.

Один шаблон, который мы можем найти в типичном списке, называется *стратегией* или *политикой*. Это идея, позволяющая выбрать алгоритм или часть алгоритма во время выполнения программы. В самой чистой форме, в форме стратегии, этот шаблон позволяет изменить способ функционирования метода или функции, не изменяя результат. Например, метод класса, который сортирует данные или предполагает их сортировку, может допускать выбор метода сортировки (например, быстрая сортировка или сортировка вставками). Результат — отсортированные данные — во всех случаях будет одним и тем же, однако предоставление клиенту возможности выбора метода сортировки может обеспечить некоторые преимущества в плане производительности. Например, клиент может избежать использования быстрой сортировки для данных с высокой степенью дублирования. В форме политики выбор клиента влияет на результат. Например, предположим, что класс представляет собой набор игральных карт. Политика сортировки может определить, считаются ли тузы самыми старшими картами (старше короля) или самыми младшими (младше 2).

Применение полученных знаний на практике

Изучив предыдущий абзац, вы узнали, что представляет собой шаблон стратегия/политика, но вы не сделали его своим инструментом. Это подобно разнице между просмотром инструментов в магазине и их фактической покупкой и использованием. Поэтому давайте возьмем этот шаблон проектирования с полки и попробуем его в деле. Самый быстрый способ опробовать новую технику — включить ее в уже написанный вами код. Давайте сформулируем задачу, которая может быть решена с использованием этого шаблона и построена на уже написанном нами коде.

Задача: староста

В каждом классе школы назначается староста («первый ученик»), который отвечает за поддержание порядка в классе в отсутствие учителя. Первоначально это звание присваивалось ученику с наивысшим уровнем успеваемости, однако теперь некоторые преподаватели считают, что староста должен определяться по старшинству, то есть по наименьшему идентификационному номеру ученика, поскольку они назначаются последовательно. Другая часть преподавателей считает традицию избрания старосты смехотворной и намеревается протестовать против нее, просто выбрав ученика, имя которого идет первым по алфавиту в ведомости успеваемости класса. Наша задача — изменить класс коллекции учеников, добавив метод для извлечения из этой коллекции старосты, при этом учитывая критерии отбора различных групп учителей.

Как видите, решение этой задачи подразумевает использование шаблона в форме политики. Мы хотим, чтобы наш метод, возвращающий имя старосты, возвращал имена разных учеников в зависимости от выбранного критерия. Чтобы реализовать это средствами языка C++, мы будем использовать указатели на функции. В главе 3 мы кратко рассмотрели эту концепцию в действии на примере функции qsort, принимающей указатель на функцию, которая сравнивает два элемента в массиве, подлежащем сортировке. Мы сделаем что-то подобное и здесь; мы используем набор функций сравнения, которые возьмут два наших объекта studentRecord и определят, является ли первый ученик «лучше» второго, исходя из их оценок, идентификационных номеров или имен.

Для начала мы должны определить тип для наших функций сравнения:

```
typedef bool ❶(* firstStudentPolicy)(studentRecord r1, studentRecord r2);
```

Это объявление создает тип с именем firstStudentPolicy в качестве указателя на функцию, которая возвращает bool и принимает два параметра типа studentRecord. Круглые скобки вокруг * firstStudentPolicy ❶ необходимы, чтобы исключить интерпретацию объявления как функции, которая возвращает указатель на bool. После добавления этого объявления мы можем создать три функции политики:

```
bool higherGrade(studentRecord r1, studentRecord r2) {
   return r1.grade() > r2.grade();
}
bool lowerStudentNumber(studentRecord r1, studentRecord r2) {
   return r1.studentID() < r2.studentID();
}
bool nameComesFirst(studentRecord r1, studentRecord r2) {
   return ❶strcmp(r1.name().c_str()❷, r2.name().c_str()❷ )❸< 0;
}
```

Первые две функции очень просты: higherGrade возвращает значение true, когда первая запись содержит более высокую оценку, а lowerStudentNumber возвращает значение true, когда первая запись содержит меньший идентификационный номер ученика. Третья функция nameComesFirst по сути представляет собой то же самое, но требует использования библиотечной функции strcmp ❶, которая ожидает две строки в «стиле C», то есть нуль-терминированные массивы символов вместо объектов string. Поэтому мы должны вызывать метод c_str() ❷ в строках name в обеих записях с данными об учениках. Функция strcmp возвращает отрицательное число, когда первая строка следует по алфавиту перед второй, поэтому мы проверяем возвращенное значение, чтобы посмотреть, не превышает ли оно значение 0 ❸. Теперь мы готовы изменить сам класс studentCollection:

```
class studentCollection {
private:
   struct studentNode {
      studentRecord studentData;
```

```
        studentNode * next;
    };
public:
        studentCollection();
        ~studentCollection();
        studentCollection(const studentCollection &copy);
        studentCollection& operator=(const studentCollection &rhs);
        void addRecord(studentRecord newStudent);
        studentRecord recordWithNumber(int IDnum);
        void removeRecord(int IDnum);
    ❶ void setFirstStudentPolicy(firstStudentPolicy f);
    ❷ studentRecord firstStudent();
private:
    ❸ firstStudentPolicy _currentPolicy;
        typedef studentNode * studentList;
        studentList _listHead;
        void deleteList(studentList &listPtr);
        studentList copiedList(const studentList copy);
};
```

Это объявление класса, которое мы видели в главе 5 с тремя новыми членами: приватным членом данных, `_currentPolicy` ❸, для хранения указателя на одну из наших функций-политик; методом `setFirstStudentPolicy` ❸ для изменения этой политики; и самим методом `firstStudent` ❷, который возвращает имя старосты в соответствии с текущей политикой. Код для метода `setFirstStudentPolicy` прост:

```
void studentCollection::setFirstStudentPolicy(firstStudentPolicy f) {
    _currentPolicy = f;
}
```

Нам также нужно изменить конструктор по умолчанию для инициализации текущей политики:

```
studentCollection::studentCollection() {
    _listHead = NULL;
    _currentPolicy = NULL;
}
```

Теперь мы готовы создать метод `firstStudent`:

```
studentRecord studentCollection::firstStudent() {
❶ if (_listHead == NULL || _currentPolicy == NULL) {
        studentRecord dummyRecord(-1, -1, "");
        return dummyRecord;
    }
    studentNode * loopPtr = _listHead;
❷ studentRecord first = loopPtr->studentData;
❸ loopPtr = loopPtr->next;
    while (loopPtr != NULL) {
        if (❹_currentPolicy(loopPtr->studentData, first)) {
            first = loopPtr->studentData;
        }
     ❺loopPtr = loopPtr->next;
    }
    return first;
}
```

Этот метод начинается с проверки специальных случаев. При отсутствии списка для проверки или политики ❶ мы возвращаем фиктивную запись. В противном случае мы обходим список, чтобы обнаружить ученика, лучше всего соответствующего текущей политике, используя базовые методы поиска, которые многократно применяли в этой книге. Мы присваиваем запись в начале списка переменной `first` ❷, запускаем нашу переменную цикла на второй записи в списке ❸ и начинаем обход. Внутри цикла обхода вызов текущей функции политики ❹ сообщает нам, является ли рассматриваемый нами ученик «лучшим учеником» по сравнению с лучшим учеником, определенным нами до сих пор, исходя из текущего критерия. После завершения цикла мы возвращаем имя старосты, то есть «первого ученика» ❺.

Анализ решения задачи выбора старосты

Решив задачу с помощью шаблона стратегии/политики, мы скорее распознаем ситуации, в которых применим этот метод, чем если бы мы просто о нем прочитали, но так и не опробовали. Мы также можем проанализировать нашу задачу, чтобы начать формировать собственное мнение о ценности метода, о том, когда его использование может быть уместным, а когда ошибочным или по крайней мере принести больше неприятностей, чем пользы. Одна мысль, которая могла у вас возникнуть по поводу этого конкретного шаблона, заключается в том, что он ухудшает инкапсуляцию и сокрытие информации. Например, если клиентский код предоставляет функции политики, он требует доступа к типам, которые обычно остаются внутренними для класса, в данном случае к типу `studentRecord`. (Мы рассмотрим способ решения этой проблемы в упражнениях). Это означает, что в клиентском коде может произойти сбой, если мы когда-либо изменим этот тип, и мы должны сопоставить эту проблему с преимуществами шаблона, прежде чем применять его в других проектах. В предыдущих главах мы уже говорили, что знание того, когда следует использовать тот или иной метод, а когда не следует, так же важно, как знание того, как его использовать. Изучение собственного кода позволяет вам ответить на этот крайне важный вопрос.

Для дальнейшей практики вы можете просмотреть свою библиотеку готовых проектов, чтобы найти код, который можно переработать с использованием этого метода. Помните о том, что большая часть «настоящего» программирования предполагает дополнение или изменение существующей базы кода, поэтому данная практика отлично подходит для подобных модификаций, а также развивает ваши навыки работы с конкретным компонентом. Более того, одним из преимуществ хорошего повторного использования кода является то, что мы учимся на этом, а данная практика позволяет получить от обучения максимальную пользу.

Обучение по мере необходимости

Предыдущий раздел был посвящен тому, что можно было бы назвать «блуждающим обучением». Хотя такие странствия ценны для программистов, иногда нам приходится двигаться к определенной цели. Если вы работаете над конкретной задачей, особенно если вам нужно успеть к какому-то сроку и вы считаете, что компонент может вам очень помочь, вам не нужно бесцельно блуждать по миру программирования в надежде наткнуться на то, что вам нужно. Вместо этого вы хотите как можно быстрее найти компонент или компоненты, непосредственно применимые к вашей ситуации. Однако проблема вот в чем: как вы найдете то, что вам нужно, если вы точно не знаете, что ищете? Рассмотрим следующую задачу.

Задача: эффективный обход

В проекте программы будет использоваться ваш класс studentCollection. Клиентский код нуждается в возможности обхода всех учеников в коллекции. Очевидно, что для обеспечения сокрытия информации клиентскому коду не может быть предоставлен прямой доступ к списку, однако эффективность обходов — непременное условие.

Поскольку ключевым словом в этом описании является *эффективность*, давайте уточним, что это значит в данном случае. Предположим, что отдельный объект нашего класса studentCollection содержит 100 учеников. Если бы у нас был прямой доступ к связному списку, мы могли бы написать цикл для обхода этого списка, который бы обрабатывал список 100 раз. Это максимально эффективный обход списка. Любое решение, требующее, чтобы мы обрабатывали список более 100 раз для определения результата, будет считаться неэффективным.

Без требования эффективности мы можем попытаться решить эту задачу, добавив в наш класс простой метод recordAt, который будет возвращать запись с данными об ученике в конкретной позиции в коллекции, присвоив первой записи номер 1:

```
studentRecord studentCollection::recordAt(int position) {
   studentNode * loopPtr = _listHead;
   int i = 1;
❶ while (loopPtr != NULL && i < position) {
      i++;
      loopPtr = loopPtr->next;
   }
   if (loopPtr == NULL) {
   ❷ studentRecord dummyRecord(-1, -1, "");
      return dummyRecord;
   } else {
    ❸ return loopPtr->studentData;
   }
}
```

В этом методе используем цикл ❶, чтобы обходить список до тех пор, пока мы не достигнем желаемой позиции или конца списка.

По завершении цикла, если был достигнут конец списка, мы создаем и возвращаем фиктивную запись ❷ или возвращаем запись в указанной позиции ❸. Проблема в том, что мы производим обход для нахождения только одной записи ученика. Этот обход необязательно полон, поскольку мы остановимся, когда достигнем желаемой позиции, но тем не менее это обход. Предположим, что клиентский код пытается усреднить оценки учащихся:

```
int gradeTotal = 0;
for (int recNum = 1; recNum <= numRecords; recNum++) {
   studentRecord temp = sc.recordAt(recNum);
   gradeTotal += temp.grade();
}
double average = (double) gradeTotal / numRecords;
```

Для этого сегмента кода предположим, что `sc` — это ранее объявленная и заполненная коллекция `studentCollection`, а `recNum` представляет собой переменную типа `int`, в которой хранится количество записей. Предположим, что значение `recNum` равно `100`. При поверхностном взгляде на этот код вам может показаться, что вычисление среднего значения требует всего 100 запусков цикла, однако, поскольку каждый вызов `recordAt` сам по себе является частичным обходом списка, этот код предполагает 100 обходов, каждый из которых будет включать около 50 запусков цикла для среднего случая. Таким образом, вместо 100 шагов, что считалось бы эффективным, для этого может потребоваться около 5000 шагов, что было бы очень неэффективно.

Когда следует искать компонент

Теперь мы столкнулись с реальной проблемой. Предоставить клиенту доступ к членам коллекции для совершения обхода легко; обеспечить эффективность такого доступа сложно. Разумеется, мы могли бы решить эту задачу, используя только собственные возможности, однако мы могли бы достичь решения намного быстрее с помощью компонента. Первым шагом при поиске неизвестного компонента, который может помочь нам решить задачу, является предположение о том, что такой компонент на самом деле существует. Иными словами, вы не найдете компонент, если не начнете его искать. Поэтому, чтобы извлечь из компонентов максимальную пользу, вам нужно уметь определять ситуации, где они могут пригодиться. Если вы застрянете на каком-то аспекте проблемы, попробуйте сделать следующее.

1. Сформулируйте проблему в общем виде.
2. Спросите себя: может ли это быть распространенной проблемой?

Первый шаг важен, потому что если мы сформулируем задачу так: «Позволить клиентскому коду эффективно вычислять среднюю оценку учеников в связном списке записей, инкапсулированных в классе», то это покажется характерным для нашей ситуации. Одна-

ко если мы сформулируем задачу так: «Позволить клиентскому коду эффективно обходить связный список без предоставления прямого доступа к указателям списка», то мы начнем понимать, что это может быть распространенной проблемой. Поскольку программы часто хранят связные списки и другие структуры с последовательным доступом в классах, мы можем предположить, что другие программисты должны были выяснить, как можно обеспечить эффективный доступ к каждому элементу структуры.

Нахождение компонента

Теперь, когда мы готовы искать, пришло время найти наш компонент. Чтобы все было ясно, давайте переформулируем исходную задачу программирования как исследовательскую проблему: «Найдите компонент, который можно использовать для изменения класса studentCollection таким образом, чтобы клиентский код мог эффективно обходить внутренний список». Как мы можем решить *эту* задачу? Мы могли бы начать с рассмотрения любого из наших типов компонентов: шаблонов, алгоритмов, абстрактных типов данных или библиотек.

Предположим, что мы начали с рассмотрения стандартных библиотек C++. Мы не обязательно будем искать класс для «включения» в решение, вместо этого мы могли бы поискать идеи в библиотеке классов, напоминающей класс studentCollection. Это подразумевает применение стратегии поиска аналогии, которую мы использовали для решения задач по программированию. Если мы найдем класс с аналогичной проблемой, то сможем воспользоваться его аналогичным решением. В ходе нашего предыдущего рассмотрения библиотеки C++ мы познакомились с такими его контейнерными классами, как vector, и нам следует искать контейнерный класс, который больше всего похож на класс studentCollection. Если мы обратимся к нашему любимому ресурсу, посвященному C++, будь то книга или интернет-сайт, и рассмотрим контейнерные классы C++, мы увидим там «контейнер последовательности» под названием list, который соответствует нашему запросу. Позволяет ли класс list клиентскому коду совершать эффективный обход? Да, используя объект, известный как *итератор*. Мы видим, что класс list предусматривает методы begin и end, создающие итераторы, которые являются объектами, которые могут ссылаться на определенный элемент в списке и инкрементироваться, чтобы итератор мог сослаться на следующий объект в списке. Если intList представляет собой list<int>, заполненный целыми числами, а iter — это list<int>::iterator, то мы могли бы отобразить все целые числа в списке с помощью следующего кода:

```
iter = intList.begin();
while (iter != intList.end()) {
   cout << *iter << "\n";
   iter++;
}
```

Благодаря использованию итератора класс `list` решил проблему, предоставив клиентскому коду механизм для эффективного обхода списка. Сейчас мы можем подумать о том, чтобы поместить сам класс `list` в класс `studentCollection`, заменив наш самодельный связный список. Затем мы могли бы создать методы `begin` и `end` для нашего класса, которые обернули бы те же методы из встроенного объекта списка, и проблема была бы решена. Однако это напрямую связано с проблемой хорошего или плохого повторного использования кода. Когда мы полностью поймем концепцию итератора и сможем самостоятельно воспроизвести ее в собственном коде, включение в код существующего класса из стандартной библиотеки шаблонов будет хорошим, а вероятно, и лучшим вариантом. Если мы не сможем этого сделать, то использование класса `list` станет обходным путем, который не поможет нам вырасти в области программирования. Иногда, конечно, нужно использовать компоненты, которые мы не можем воспроизвести сами, однако если мы привыкнем к зависимости от других программистов при решении задач, то рискуем никогда не научиться решать задачи самостоятельно.

Итак, давайте самостоятельно реализуем итератор. Прежде чем мы это сделаем, давайте кратко рассмотрим другие способы, с помощью которых мы могли бы прийти к тому же самому результату. Мы начали поиск со стандартных библиотек шаблонов, однако могли бы начать искать в другом месте. Например, мы могли бы просмотреть список распространенных шаблонов проектирования. Под заголовком «поведенческие шаблоны» мы бы нашли шаблон *iterator*, в котором клиенту предоставляется последовательный доступ к коллекции элементов без раскрытия базовой структуры этой коллекции. Это именно то, что нужно, однако мы могли бы обнаружить это решение только, осуществив поиск в списке шаблонов или вспомнив о том, что оно нам уже попадалось в ходе предыдущего исследования шаблонов. Мы могли бы начать поиск с абстрактных типов данных, потому что *список* вообще и *связный список* в частности — распространенные абстрактные типы данных. Тем не менее во многих обсуждениях и реализациях абстрактного типа данных `list` обход списка клиентским кодом не считается базовой операцией, поэтому вопроса о концепции итератора никогда не возникает. Наконец, если мы начнем свой среди алгоритмов, то вряд ли найдем что-нибудь полезное. Алгоритмы, как правило, описывают сложный код, а код для создания итератора довольно прост, как мы вскоре увидим. Поэтому в данном случае библиотека классов была самым быстрым путем к месту назначения, за которым следуют шаблоны. Однако, как правило, вы должны учитывать все типы компонентов при поиске того, который может оказаться для вас полезным.

Применение компонента

Теперь мы знаем, что собираемся создать итератор для класса `studentCollection`, однако все, что нам показал класс стандартной библиотеки `list`, — это то, как методы итератора работают вовне.

Если бы мы застряли на этапе реализации, то могли бы проверить исходный код list и его классы-предки, однако, учитывая сложность чтения больших фрагментов незнакомого кода, это было бы крайней мерой. Вместо этого давайте просто подумаем над этим. Используя предыдущий пример кода в качестве руководства, можно сказать, что итератор определяется четырьмя центральными операциями.

1. Метод в классе коллекции, предоставляющий итератор, который ссылается на первый элемент в коллекции. В классе list это был метод begin.
2. Механизм для проверки того, прошел ли итератор последний элемент в коллекции. В предыдущем примере это был метод под названием end в классе list, который создал специальный объект итератора для тестирования.
3. Метод в классе итератора, который перемещает итератор, чтобы он ссылался на следующий элемент в коллекции. В предыдущем примере это был перегруженный оператор ++.
4. Метод в классе итератора, который возвращает элемент коллекции, на который в данный момент ссылается итератор. В предыдущем примере это был перегруженный префиксный оператор*.

В плане написания кода здесь нет ничего сложного. Нужно лишь расставить все по своим местам. Из приведенных выше описаний следует, что наш итератор, который назовем scIterator, должен хранить ссылку на элемент в коллекции studentCollection и должен иметь возможность перейти к следующему элементу. Таким образом, итератор должен хранить указатель на studentNode. Это позволит ему возвратить содержащуюся внутри запись studentRecord, а также перейти к следующему studentNode. Поэтому приватный раздел класса итератора будет содержать следующий член данных:

```
studentCollection::studentNode * current;
```

У нас сразу возникает проблема. Тип studentNode объявляется в приватном разделе коллекции studentCollection, поэтому приведенная выше строка не сработает. Нашей первой мыслью является то, что, возможно, studentNode не должен быть приватным, однако этот ответ неверен. Приватность — неотъемлемое свойство типа узла, поскольку мы не хотим, чтобы случайный клиентский код зависел от конкретной реализации типа узла, тем самым создавая код, в котором может произойти сбой при изменении нашего класса. Тем не менее мы должны предоставить итератору scIterator доступ к нашему приватному типу. Мы делаем это с помощью объявления friend. В общедоступном разделе studentCollection добавляем:

```
friend class scIterator;
```

Теперь scIterator может получить доступ к приватным объявлениям внутри studentCollection, включая объявление для studentNode. Мы также можем объявить несколько конструкторов:

```
scIterator::scIterator() {
   current = NULL;
}
scIterator::scIterator(studentCollection::studentNode * initial) {
   current = initial;
}
```

Давайте на секунду перейдем к studentCollection и напишем наш метод begin, возвращающий итератор, который ссылается на первый элемент в нашей коллекции. Следуя схеме именования, которую я использовал в этой книге, данный метод должен иметь в качестве имени существительное, например, firstItemIterator:

```
scIterator studentCollection::firstItemIterator() {
   return scIterator(_listHead);
}
```

Как видите, все, что нам нужно сделать, — это поместить указатель на головной элемент связного списка в объект scIterator и возвратить его. Если вы похожи на меня, то мельтешащие указатели могут вас немного нервировать, однако обратите внимание на то, что scIterator будет просто держаться за ссылку на элемент в списке studentCollection. Он не будет выделять собственную память, поэтому нам не нужно беспокоиться о глубоких копиях и перегруженных операторах присваивания.

Вернемся к scIterator и напишем другие наши методы. Нам нужен метод для перехода итератора к следующему элементу, а также метод для определения того, достигли ли мы конца коллекции. Мы должны думать обо всем этом одновременно. При перемещении итератора нужно знать, какое значение должен иметь итератор при выходе за пределы последнего узла в списке. Если мы не делаем ничего особенного, итератор, естественно, получит значение NULL, так что это будет самое простое для использования значение. Обратите внимание на то, что мы инициализировали итератор значением NULL в конструкторе по умолчанию, поэтому, когда мы используем NULL для обозначения конца коллекции, между этими двумя состояниями теряется всякое различие, однако в случае данной задачи это не проблема. Ниже показан код для методов:

```
❶ void scIterator::advance() {
   ❷ if (current != NULL)
      ❸ current = current->next;
   }
❹ bool scIterator::pastEnd() {
      return current == NULL;
   }
```

Помните, что мы просто используем концепцию итератора для решения исходной задачи. Мы не пытаемся дублировать точную спецификацию итератора стандартной библиотеки шаблонов C++, поэтому нам необязательно использовать тот же самый интерфейс. В данном случае вместо перегрузки оператора ++ я применяю метод `advance` ❶, который проверяет, что указатель `current` не имеет значения `NULL` ❷, прежде чем перемещать его к следующему узлу ❸. Точно так же я считаю громоздким создание специального «конечного» итератора для сравнения, поэтому я просто использую метод `bool` под названием `pastEnd` ❹, который определяет, закончились ли у нас узлы.

Наконец, нам нужен способ получения объекта `studentRecord`, на который мы в данный момент ссылаемся:

```
studentRecord scIterator::student() {
  ❶ if (current == NULL) {
        studentRecord dummyRecord(-1, -1, "");
        return dummyRecord;
  ❷ } else {
        return current->studentData;
    }
}
```

Как и ранее, в целях безопасности, если наш указатель имеет значение `NULL`, мы создаем и возвращаем фиктивную запись ❶. В противном случае мы возвращаем запись, на которую в данный момент ссылаемся ❷. Это завершает реализацию концепции итератора с нашим классом `studentCollection`. Для ясности далее приведено полное объявление класса `scIterator`:

```
class scIterator {
public:
   scIterator();
   scIterator(studentCollection::studentNode * initial);
   void advance();
   bool pastEnd();
   studentRecord student();
private:
   studentCollection::studentNode * current;
};
```

Теперь, когда код готов, мы можем протестировать его с помощью пробного обхода. Давайте реализуем процесс вычисления средней оценки для сравнения:

```
   scIterator iter;
   int gradeTotal = 0;
   int numRecords = 0;
❶ iter = sc.firstItemIterator();
❷ while (!iter.pastEnd()) {
      numRecords++;
    ❸ gradeTotal += iter.student().grade();
    ❹ iter.advance();
   }
   double average = (double) gradeTotal / numRecords;
```

В этом листинге используются все наши методы, имеющие отношение к итератору, поэтому он представляет собой хороший тест для нашего кода. Мы вызываем firstItemIterator для инициализации объекта scIterator ❶. Мы вызываем метод pastEnd в качестве теста на завершение цикла ❷. Мы вызываем метод объекта итератора student для получения текущей записи studentRecord, чтобы иметь возможность извлечь оценку ❸. Наконец, для перемещения итератора на следующую запись мы вызываем метод advance ❹. Когда этот код работает, мы можем быть в достаточной степени уверенными в том, что мы правильно реализовали различные методы и, более того, четко усвоили концепцию итератора.

Анализ эффективного решения задачи с обходом

Как и прежде, если код работает, это не значит, что мы не можем научиться на нем чему-то еще. Мы должны внимательно посмотреть, что сделали, проанализировать положительные и отрицательные последствия, а также рассмотреть возможность расширения базовой идеи, которую только что реализовали. В данном случае мы можем сказать, что концепция итератора определенно решает исходную проблему неэффективного обхода нашей коллекции клиентским кодом, и после ее реализации использование итератора является элегантным и легко читаемым решением. С другой стороны, нельзя отрицать, что неэффективный подход, основанный на методе recordAt, был намного более простым в написании. При решении вопроса о ценности реализации итератора для конкретной ситуации мы должны спросить себя, как часто будут иметь место обходы, сколько элементов, как правило, будет содержаться в нашем списке и т.д. Если обходы редки, а список мал, то неэффективность, вероятно, не будет иметь значения, однако если мы ожидаем увеличения списка или не можем гарантировать, что этого не произойдет, то нам может потребоваться итератор.

Конечно, если бы мы решили использовать объект list из стандартной библиотеки шаблонов, нам больше не пришлось бы беспокоиться о сложности реализации итератора, поскольку не нужно было бы самим его реализовывать. В следующий раз, когда возникнет подобная ситуация, мы сможем использовать класс list без ощущения того, что мы обманываем себя или настраиваемся на трудности в будущем, поскольку мы изучили списки и итераторы достаточно, чтобы понимать, что происходит за кулисами, даже если мы никогда не рассматривали фактический исходный код.

Заходя еще дальше, мы можем подумать о более широком применении итераторов и об их возможных ограничениях. Например, предположим, что нам понадобился итератор, который мог бы эффективно перемещаться не только к следующему элементу нашей studentCollection, но и к предыдущему. Теперь, когда мы знаем, как работает итератор, понимаем, что не можем решить эту зада-

чу с помощью нашей текущей реализации `studentCollection`. Если итератор поддерживает ссылку на определенный узел в списке, то переход к следующему узлу требует простого следования по ссылке в узле. Тем не менее возврат к предыдущему узлу требует повторного обхода списка вплоть до этой точки. Вместо этого нам бы потребовался двусвязный список, узлы которого имеют указатели в обоих направлениях — как на следующий узел, так и на предыдущий. Мы можем обобщить эту мысль и начать рассматривать различные структуры данных и то, какие виды обходов или доступа к данным могут быть эффективно предложены клиентам. Например, в предыдущей главе, посвященной рекурсии, мы кратко описали структуру двоичного дерева. Существует ли способ позволить клиенту совершить эффективный обход этой структуры в своей стандартной форме? Если нет, какие изменения нам нужно внести для обеспечения эффективных разворотов? Что такое правильный порядок узлов в двоичном дереве, подлежащем обходу? Размышление по поводу таких вопросов помогает нам стать более хорошими программистами. Мы не только приобретем новые навыки, но и узнаем больше о достоинствах и недостатках различных компонентов. Знание плюсов и минусов компонента позволит нам использовать его с умом. Игнорирование ограничений конкретного подхода может заводить в тупик, и чем больше мы знаем о компонентах, которые используем, тем меньше вероятность того, что это с нами произойдет.

Выбор типа компонента

Как мы видели в этих примерах, одна и та же проблема может быть решена с использованием различных типов компонентов. Шаблон может выражать идею решения, алгоритм может описывать реализацию этой идеи или другую идею, которая решает ту же проблему, абстрактный тип данных может инкапсулировать концепцию, а класс в библиотеке может содержать полностью проверенную реализацию абстрактного типа данных. Если каждый из них — это выражение той концепции, которая требуется нам для решения проблемы, как мы узнаем, какой тип компонента следует выбрать из нашего инструментария?

Одно из основных соображений заключается в том, сколько усилий может потребоваться для интеграции компонента в наше решение. Привязывание библиотеки классов к нашему коду часто является быстрым способом решения проблемы, тогда как реализация алгоритма из описания псевдокода может занять много времени. Еще одно важное соображение заключается в степени гибкости, обеспечиваемой предлагаемым компонентом. Часто компонент изначально имеет удобную, готовую к использованию форму, но когда он интегрируется в проект, программист обнаруживает, что, хотя компонент выполняет большую часть необходимых задач, он дела-

ет не все. Например, возвращаемое значение одного метода имеет неправильный формат и требует дополнительной обработки. Если компонент все равно используется, то в будущем могут обнаруживаться другие проблемы до тех пор, пока компонент не будет совершенно отброшен, а для этой части проблемы не будет разработан с нуля новый код. Если программист выбрал компонент более высокого концептуального уровня, например шаблон, то полученная в результате реализация кода идеально впишется в проблему, поскольку она создавалась специально для ее решения.

На рис. 7.1 показано взаимодействие этих двух факторов. Как правило, код из библиотеки готов к использованию, но его нельзя изменить напрямую. Его можно модифицировать косвенно либо с помощью шаблонов C++, либо в том случае, если код, о котором идет речь, реализует нечто вроде шаблона-*стратегии*, о котором говорилось ранее в этой главе. С другой стороны, шаблон может быть представлен в качестве идеи («класс, который может иметь только один экземпляр»), обеспечивая максимальную гибкость в плане реализации, но требуя от программиста большой работы.

Конечно, это всего лишь общее руководство, а конкретные случаи будут иметь свои отличия. Возможно, класс из библиотеки, который мы используем, находится на таком низком уровне в нашей программе, что гибкость не пострадает. Например, мы можем обернуть класс коллекции нашего собственного дизайна вокруг базового контейнерного класса, вроде `list`, который имеет такие широкие возможности, что, даже если нам придется расширить функциональность нашего контейнерного класса, мы можем ожидать того, что класс `list` с этим справится. Прежде чем использовать шаблон, возможно, мы уже реализовали конкретный шаблон раньше, поэтому мы не столько создаем новый код, сколько адаптируем ранее написанный.

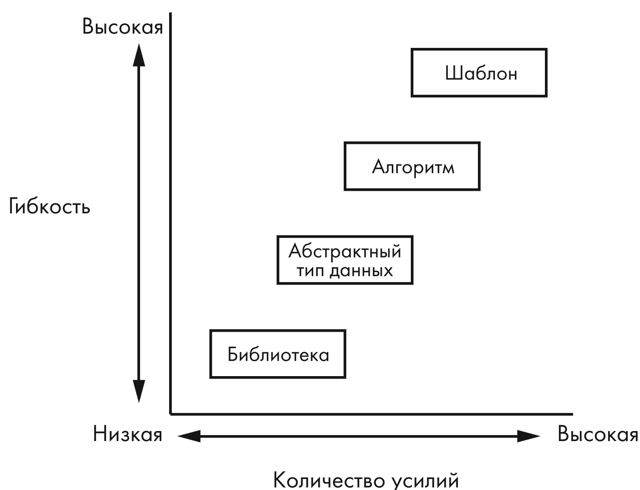


Рис. 7.1. Типы компонентов: степень гибкости против количества требуемых усилий

Чем больше у вас опыта использования компонентов, тем более уверены вы можете быть в том, что начинаете с нужного места. Пока вы не наработаете этот опыт, вы можете использовать компромисс между гибкостью и количеством необходимых усилий в качестве общего руководства. В каждой конкретной ситуации задайте себе следующие вопросы.

- Могу ли я использовать компонент как есть, или для его внедрения в мой проект мне потребуется написать дополнительный код?
- Уверен ли я в том, что понимаю всю суть проблемы или ту ее часть, которая относится к этому компоненту, а также в том, что она не изменится в будущем?
- Расширю ли я свои знания в области программирования, выбрав этот компонент?

Ответы на эти вопросы помогут вам оценить, сколько усилий вам потребуется, а также то, какую пользу вы получите от каждого из возможных подходов.

Процесс выбора компонента

Теперь, когда мы понимаем общую идею, давайте рассмотрим небольшой пример, чтобы разобрать конкретные детали.

Задача: выборочная сортировка

Проект требует того, чтобы вы отсортировали массив объектов `studentRecord` по оценкам, однако существует один подвох. Другая часть программы использует специальное значение оценки –1, обозначающее студента, запись которого не может быть перемещена. Таким образом, в то время как все остальные записи должны перемещаться, те, которые имеют значение оценки –1, должны оставаться на своем месте, в результате получается массив, в котором отсортированные объекты перемежаются объектами со значением оценки равным –1.

Это сложная задача, и существует много способов, с помощью которых мы могли бы попытаться ее решить. Для простоты, давайте ограничимся двумя вариантами: либо мы выберем алгоритм, то есть процедуру сортировки, вроде сортировки вставками, и модифицируем его так, чтобы проигнорировать объекты `studentRecord` с оценкой –1, либо найдем способ использования библиотечной процедуры `qsort` для решения этой проблемы. Возможны оба варианта. Поскольку мы уже освоились с кодом сортировки вставками, нам не составит труда добавить несколько инструкций `if`, чтобы конкретно проверить и пропустить записи с оценкой –1. Чтобы метод `qsort` сделал за нас всю работу, нам потребуется воспользоваться обходным путем. Мы могли бы скопировать записи учеников с реальными оценками в отдельный мас-

сив, отсортировать их с помощью qsort, а затем скопировать их обратно, убедившись в том, что мы не копируем ни одну из записей с оценкой –1.

Давайте рассмотрим оба варианта, чтобы понять, как выбор типа компонента влияет на итоговый код. Мы начнем с такого компонента, как алгоритм, написав собственный модифицированный код сортировки вставками для решения этой задачи. Как обычно, мы будем решать задачу поэтапно. Во-первых, давайте уменьшим ее, удалив всю проблему объектов с оценкой –1 и просто отсортировав массив объектов studentRecord без каких-либо специальных правил. Если sra — это массив, содержащий объекты arraySize типа studentRecord, то итоговый код будет выглядеть так:

```
int start = 0;
int end = arraySize - 1;
for (int i = start + 1; i <= end; i++) {
   for (int j = i; j > start && ❶sra[j-1].grade() > sra[j].grade(); j--) {
   ❷ studentRecord temp = sra[j-1];
      sra[j-1] = sra[j];
      sra[j] = temp;
   }
}
```

Этот код очень похож на код сортировки вставками для целых чисел. Отличия состоят лишь в том, что для сравнения требуются вызовы метода grade ❶, а наш временный объект, используемый в качестве подкачки, поменял свой тип ❷. Этот код работает нормально, однако существует один нюанс, касающийся тестирования этого и других блоков кода приведенных далее в данном разделе: наш класс studentRecord проверяет данные, и, как было сказано ранее, не принимает оценку –1, поэтому убедитесь, что вы внесли необходимые изменения. Теперь мы готовы закончить эту версию решения. Нам нужно, чтобы алгоритм сортировки вставками игнорировал записи с оценкой –1. Сделать это не так просто, как кажется. В базовом алгоритме сортировки вставками мы всегда меняем местами соседние позиции в массиве — в приведенном выше коде это j и j - 1. Тем не менее, если мы оставим на месте записи с оценкой –1, то расстояние между записями, подлежащими перестановке, может быть произвольным.

На рис. 7.2 приведен пример для иллюстрации этой задачи. Если массив изображен в исходной конфигурации, то стрелки указывают местоположение первых записей, которые нужно поменять местами, и они не являются соседними. Кроме того, в конце концов, последняя запись (для Арта) должна быть перемещена с позиции [5] на [3], а затем с [3] на [0], поэтому все замены, необходимые для сортировки этого массива (поскольку мы его сортируем), касаются записей, которые не являются соседними.

[0]	[1]	[2]	[3]	[4]	[5]
Том	Глэдис	Сэм	Джейн	Джон	Арт
87	-1	-1	84	-1	72
11523	83764	65342	11523	11764	77663

Рис. 7.2. Произвольное расстояние между записями, подлежащими перестановке в модифицированном коде для сортировки вставками

В процессе размышления над решением этой задачи я искал аналогию и нашел ее в обработке связных списков. Во многих алгоритмах связных списков необходимо поддерживать указатель не только на текущий узел в процессе обхода нашего списка, но и на предыдущий. Поэтому в конце тела цикла мы часто назначаем текущий указатель предыдущему указателю, прежде чем перемещать текущий указатель. Здесь должно происходить что-то подобное. Нам нужно отслеживать последнюю «реальную» запись студента по мере линейного продвижения по массиву, чтобы найти следующую «реальную» запись. Практическая реализация этой идеи выглядит следующим образом:

```
for (int i = start + 1; i <= end; i++) {
  ❶ if (sra[i].grade() != -1) {
    ❷ int rightswap = i;
      for (int leftswap = i – 1;
        leftswap >= start
        && (sra[leftswap].grade() > sra[rightswap].grade()
         ❸ || sra[leftswap].grade() == -1);
        leftswap--)
      {
      ❹ if(sra[leftswap].grade() != -1) {
          studentRecord temp = sra[leftswap];
          sra[leftswap] = sra[rightswap];
          sra[rightswap] = temp;
        ❺ rightswap = leftswap;
       }
    }
  }
}
```

В базовом алгоритме сортировки вставками мы многократно вставляем несортированные элементы в постоянно растущую отсортированную область внутри массива. Внешний цикл выбирает следующий несортированный элемент, подлежащий вставке в отсортированный порядок. В этой версии кода мы начинаем с проверки того, что значение оценки в позиции `i` не равно –1 ❶ внутри внешнего цикла. Если это так, мы просто перейдем к следующей записи, оставив эту запись на месте. Как только мы установили, что запись студента в позиции `i` может быть перемещена, мы инициализируем переменную `rightswap` значением этой позиции ❷. Затем запускаем внутренний цикл. В базовом алгоритме сортировки вставками каждая итерация внутреннего цикла меняет местами соседние

элементы. Однако в нашей версии, поскольку оставляем на месте записи с оценкой –1, мы осуществляем обмен только тогда, когда в местоположении j не находится оценка –1 ❹. Затем мы меняем местами позиции leftswap и rightswap и присваиваем переменной rightswap значение leftswap ❺, настраивая следующий обмен во внутреннем цикле, если он есть. Наконец, нам нужно изменить условие в нашем внутреннем цикле. Обычно при сортировке вставками внутренний цикл прекращает выполняться, когда мы достигаем переднего конца массива или когда находим меньшее значение по сравнению с тем, которое вставляем. Здесь мы должны сформулировать составное условие с использованием логического *или* так, чтобы цикл продолжал выполняться после нахождения оценки –1 ❸ (поскольку значение –1 всегда будет меньше любой допустимой оценки, что приведет к преждевременной остановке цикла).

Этот код позволяет нам решить задачу, но может испускать некоторый «неприятный запах». Стандартный код сортировки вставками легко читается, особенно если вы понимаете суть того, что он делает. Однако эта модифицированная версия сложна для восприятия и, возможно, требует нескольких строк с комментариями, если мы хотим иметь возможность позднее его понять. Возможно, здесь может быть уместен рефакторинг, однако давайте попробуем использовать для решения этой задачи другой подход и рассмотрим соответствующий ему код.

Первое, что нам понадобится, — это функция сравнения для использования с методом qsort. В данном случае мы будем сравнивать два объекта studentRecord, и наша функция будет вычитать одну оценку из другой:

```
int compareStudentRecord(const void * voidA, const void * voidB) {
   studentRecord * recordA = (studentRecord *) voidA;
   studentRecord * recordB = (studentRecord *) voidB;
   return recordA->grade() - recordB->grade();
}
```

Теперь мы готовы отсортировать записи. Мы сделаем это в три этапа. Во-первых, скопируем все записи, которые не содержат оценки –1, во вторичный массив, без пропусков. Затем вызовем метод qsort для сортировки вторичного массива. Наконец, скопируем записи из вторичного массива обратно в исходный массив, пропустив записи с оценкой –1. Итоговый код выглядит следующим образом:

```
❶ studentRecord * sortArray = new studentRecord[arraySize];
❷ int sortArrayCount = 0;
  for (int i = 0; i < arraySize; i++) {
  ❸ if (sra[i].grade() != -1) {
         sortArray[sortArrayCount] = sra[i];
         sortArrayCount++;
     }
  }
❹ qsort(sortArray, sortArrayCount, sizeof(studentRecord),
  compareStudentRecord);
```

```
❺ sortArrayCount = 0;
❻ for (int i = 0; i < arraySize; i++) {
  ❼ if (sra[i].grade() != -1) {
        sra[i] = sortArray[sortArrayCount];
        sortArrayCount++;
     }
  }
```

Хотя этот код имеет примерно ту же длину, что и другое решение, он более прост и удобен для восприятия. Мы начинаем с объявления вторичного массива `sortArray` ❶, имеющего тот же размер, что и исходный массив. Переменная `sortArrayCount` инициализируется нулем ❷; в первом цикле мы будем использовать ее для отслеживания количества записей, скопированных во вторичный массив. Внутри этого цикла при каждом нахождении записи, не содержащей оценки –1 ❸, мы присваиваем ей следующий доступный слот в `sortArray` и инкрементируем `sortArrayCount`. После завершения цикла мы сортируем вторичный массив ❹. Значение переменной `sortArrayCount` сбрасывается на 0 ❺; мы будем использовать ее во втором цикле для отслеживания количества записей, скопированных из вторичного массива обратно в исходный массив. Обратите внимание на то, что второй цикл обходит *исходный* массив ❻, ища слоты, которые необходимо заполнить ❼. Если мы подойдем к этому другим способом, пытаясь обработать циклом вторичный массив и поместив записи в исходный массив, нам понадобился бы двойной цикл, причем внутренний цикл искал бы в исходном массиве следующий слот с реальной оценкой. Это еще один пример того, как наша концептуализация задачи может сделать ее более легкой или трудной.

Сравнение результатов

Оба решения работают и представляют собой разумные подходы. Для большинства программистов первое решение, в котором мы модифицировали алгоритм сортировки вставками, чтобы оставить на месте некоторые записи, обойдя их в процессе сортировки, сложнее в плане написания и восприятия. Тем не менее второе решение, похоже, несколько неэффективно, поскольку требует копирования данных во вторичный массив и обратно. Здесь могут пригодиться знания в области анализа алгоритмов. Предположим, мы сортируем 10 000 записей — если бы мы сортировали намного меньше записей, то не заботились бы об эффективности. Мы не можем точно знать, какой алгоритм лежит в основе вызова метода `qsort`, однако в худшем случае универсальная сортировка потребовала бы поменять местами записи 100 миллионов раз, а в лучшем случае — около 130 000. Вне зависимости от того, в каком месте диапазона мы находимся, копирование 10 000 записей туда и обратно нанесет по производительности не столь серьезный удар по сравнению с сортировкой. Кроме того, мы должны учесть, что любой алгоритм, используемый методом `qsort`, может оказаться гораздо эффективнее, чем наша простая сортировка вставками, нивелировав любое преимущество, которое

мы могли бы получить, избегая копирования данных во вторичный массив и обратно.

Таким образом, в этом сценарии второй подход, использующий метод qsort, оказывается победителем. Его проще реализовать, проще читать и, следовательно, легче сопровождать, кроме того, мы можем ожидать того, что он окажется таким же, а, возможно, и более эффективным, чем первое решение. Самое лучшее, что мы можем сказать о первом подходе, — это то, что мы могли бы освоить навыки, которые можно было бы применить к решению других задач, тогда как второй подход, в силу своей простоты, этого не предполагает. Как правило, когда вы находитесь на этапе программирования, где пытаетесь освоить максимальное количество навыков, вам следует отдавать предпочтение компонентам более высокого уровня, таким как алгоритмы и шаблоны. Когда вы пытаетесь максимизировать свою эффективность как программиста (или ограничены во времени), вы должны отдавать предпочтение компонентам более низкого уровня, выбирая, по возможности, готовый код. Конечно, если время позволяет, попытка использовать несколько разных вариантов, как мы сделали это здесь, позволяет получить лучшее из всех подходов.

Упражнения

Опробуйте как можно больше компонентов. Как только вы научитесь осваивать новые компоненты, ваши способности как программиста быстро начнут расти.

7.1. Жалоба, поступающая в адрес шаблона *политики/стратегии*, заключается в том, что она требует разоблачения некоторых внутренних данных класса, например типов. Измените программу для определения старости, рассмотренную ранее в этой главе, так, чтобы все функции политики хранились в классе и выбирались путем передачи значения кода (например, нового нумерованного типа) вместо передачи самой функции политики.

7.2. Перепишите наши функции studentCollection из главы 4 (addRecord и averageRecord) так, чтобы вместо непосредственной реализации связного списка вы использовали класс из библиотеки C++.

7.3. Рассмотрим коллекцию объектов studentRecord. Мы хотим иметь возможность быстрого нахождения конкретной записи на основе номера ученика. Сохраните записи учеников в массиве, отсортируйте массив по номеру ученика, а также исследуйте и реализуйте алгоритм *интерполяционного поиска.*

7.4. Решите задачу из пункта 7.3 путем реализации абстрактного типа данных, который позволяет хранить произвольное количество элементов и извлекать отдельные записи на основе значения ключа. Общим названием для структуры, которая может эффективно хранить и извлекать элементы на основе значения ключа,

является *таблица символов*, а распространенными реализациями идеи таблицы символов — *хеш-таблицы* и *двоичные деревья поиска*.

7.5. Решите задачу в пункте 7.3, используя класса из библиотеки C++.

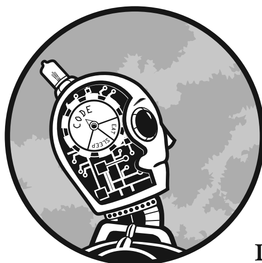
7.6. Предположим, вы работаете над проектом, в котором может потребоваться дополнить конкретную запись `studentRecord` одним из следующих фрагментов данных: названием курсовой работы, годом поступления или значением `bool`, указывающим, прослушивает ли студент данный курс без получения оценки. Вы не хотите включать все эти поля данных в базовый класс `studentRecord`, зная, что они не будут использоваться в большинстве случаев. Вашей первой мыслью будет создание трех подклассов, каждый из которых имеет одно из полей данных с такими именами, как `studentRecordTitle`, `studentRecordYear` и `studentRecordAudit`. Затем вам сообщают, что некоторые записи студентов будут содержать два из этих дополнительных полей данных или, возможно, все три. Создание подклассов для каждого возможного варианта нецелесообразно. Найдите шаблон дизайна, направленный на разгадывание этой загадки, и реализуйте решение.

7.7. Разработайте решение задачи, описанной в пункте 7.6, которое не использует обнаруженный вами шаблон, но вместо этого решает проблему с помощью классов библиотеки C ++. Вместо того, чтобы сосредоточиваться на трех конкретных полях данных, описанных в предыдущем вопросе, попробуйте реализовать общее решение: версию класса `studentRecord`, которая позволяет добавлять в конкретные объекты дополнительные поля данных. Так, например, если `sr1` представляет собой запись `studentRecord`, вы можете сделать так, чтобы клиентский код вызывал `sr1.addExtraField("Title", "Problems of Unconditional Branching")`, а затем `sr1.retrieveField("Title")` возвращал «проблему безусловного ветвления».

7.8. Придумайте собственную задачу. Возьмите задачу, которую вы уже решили, и решите ее снова, используя другой компонент. Не забудьте проанализировать результаты, сравнив их со своим исходным решением.

8

Думайте как программист



Пришло время подвести итог всему, что мы узнали в предыдущих главах, чтобы завершить превращение из новичка в программиста, умеющего решать задачи. В предыдущих главах мы решали задачи, относящиеся к самым разным областям. Я считаю, что эти области наиболее полезны для развивающегося программиста, но, разумеется, процесс обучения ими не ограничивается, и многие задачи потребуют навыков, не описанных в этой книге. Итак, в данной главе мы собираемся объединить общие концепции решения задач и применить знания, которые усвоили по ходу нашего путешествия, чтобы разработать *мастер-план* для решения любой задачи по программированию. Хотя мы могли бы назвать этот план общим, в некотором отношении он на самом деле очень конкретен: это будет *ваш* план, а не чей-то. Мы также рас-

смотрим множество способов, с помощью которых вы можете расширить свои профессиональные знания и навыки.

Разработка собственного мастер-плана

В первой главе мы узнали о первом правиле решения задачи, которое заключается в том, чтобы всегда иметь план. Если сформулировать его точнее, то вы всегда должны придерживаться *своего* плана. Составьте мастер-план, который максимизирует ваши сильные стороны и сводит к минимуму слабые, а затем примените его к каждой решаемой вами задаче.

За многие годы преподавания я видел учеников, обладающих разными способностями. Под этим я не просто имею в виду, что некоторые программисты обладают большими способностями, чем другие, хотя это, конечно, правда. Даже среди программистов с одинаковыми способностями существует большое разнообразие. Я много раз удивлялся тому, как еще недавно отстающий студент быстро овладевает особым навыком или как талантливый ученик испытывает сложности с освоением новой области. Так же, как нет двух одинаковых отпечатков пальцев, нет двух одинаковых мозгов, и темы, простые для одного человека, сложны для другого.

Представьте, что вы — футбольный тренер, планирующий стратегию для следующей игры. Из-за травмы двух защитников вы не уверены, какой из них сможет начать игру. Оба защитника — высокопрофессиональные игроки, но, как у всех, у них есть свои сильные и слабые стороны. План игры, обеспечивающий лучшие шансы на победу для одного защитника, может оказаться провальным для другого.

При создании своего мастер-плана вы являетесь тренером, а ваш набор навыков — вашим игроком. Чтобы максимизировать свои шансы на успех, вам нужен план, который учитывает как ваши сильные, так и слабые стороны.

Использование своих сильных и слабых сторон

Ключевым шагом в составлении собственного мастер-плана является определение собственных сильных и слабых сторон. Это не сложно, но требует усилий и довольно честной самооценки. Чтобы извлечь пользу из своих ошибок, вы должны не только исправлять их в программах, в которых они появляются, но и отмечать их, по крайней мере, мысленно или еще лучше, документировать. Так вы можете определить поведенческие паттерны, которые вы в противном случае пропустили бы.

Я расскажу о недостатках в двух различных категориях — в кодировании и в дизайне. *Недостатки кодирования* — это области, где вы склонны повторять ошибки при написании кода. Например, многие программисты часто пишут циклы, количество итераций в которых оказывается на единицу меньше или больше чем нужно. Это

известно как *ошибка заборного столба*, получившая свое название от старой загадки о том, сколько заборных столбов необходимо для создания 50-метрового забора с поперечинами длиной в 10 метров между столбами. Большинство людей сразу отвечают «пять», но если вы как следует подумаете, то получите «шесть», как показано на рис. 8.1.

Большинство недостатков кодирования связано с ситуациями, когда программист допускает семантические ошибки, создавая код слишком быстро или без достаточной подготовки. Напротив, *недостатки в дизайне* — это проблемы, которые обычно возникают на этапе решения задач или проектирования. Например, вы можете обнаружить, что сталкиваетесь с проблемами на начальном этапе или на этапе интеграции ранее написанных подпрограмм в итоговое решение.

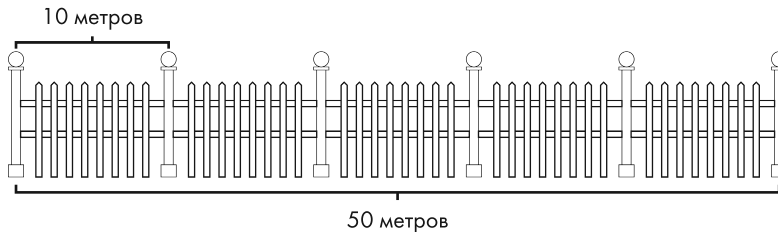


Рис. 8.1. Загадка заборного столба

Несмотря на то, что две эти категории несколько пересекаются, два вида недостатков, как правило, создают разные проблемы, и предотвращать их нужно разными способами.

Планирование с учетом недостатков кодирования

Вероятно, наиболее неприятная деятельность в программировании — это многочасовое отслеживание семантической ошибки, исправить которую очень просто, как только удается ее обнаружить. Поскольку никто не совершенен, нет возможности полностью устранить подобные ситуации, но хороший программист сделает все возможное, чтобы избежать повторения одних и тех же ошибок.

Я знал программиста, который устал совершать, пожалуй, наиболее распространенную семантическую ошибку в программировании на языке C++, которая заключается в подмене оператора равенства (==) оператором присваивания (=). Поскольку условные выражения в C++ являются целочисленными, а не строго логическими, утверждение, подобное следующему, синтаксически допустимо:

```
if (number = 1) flag = true;
```

В данном случае целочисленное значение 1 присваивается переменной `number`, а затем значение 1 используется как результат условного выражения, которое C++ оценивает как `true`. Конечно, программист имел в виду следующее:

```
if (number == 1) flag = true;
```

Устав от повторения такого типа ошибок, программист приучил себя всегда писать тесты равенства по-другому, указывая числовой литерал слева, например:

```
if (1 == number) flag = true;
```

Благодаря этому, если программист по ошибке заменит оператор равенства, выражение `1 = number` перестанет быть допустимым синтаксисом С++, что приведет к синтаксической ошибке, которая будет обнаружена во время компиляции. Исходная ошибка представляет собой допустимый синтаксис, поэтому является всего лишь семантической ошибкой, которая будет обнаружена во время компиляции или не будет обнаружена вообще. Поскольку я сам много раз совершал эту ошибку (и сходил с ума, пытаясь ее найти), я использовал этот метод, помещая числовой литерал в левой части оператора равенства. При этом я обнаружил кое-что любопытное. Поскольку это противоречит моему обычному стилю, при написании условных утверждений необходимость помещения числового литерала в левую часть заставляет меня на мгновение остановиться. Я думаю: «Мне нужно не забыть поместить литерал слева, чтобы суметь поймать себя в случае, если я по ошибке использую оператор присваивания». Как и следовало ожидать, благодаря этой мысли я никогда не использовал оператор присваивания вместо оператора равенства. Теперь я больше не помещаю литерал в левую часть оператора равенства, но по-прежнему останавливаюсь и позволяю этой мысли пронестись в моей голове, что не позволяет мне использовать неправильный оператор.

Урок здесь заключается в том, что знания о своих недостатках на уровне кодирования часто достаточно для того, чтобы избежать их. Это хорошая новость. Плохая новость заключается в том, что вам все равно придется поработать, чтобы осознать свои слабые стороны как кодировщика. Ключевой метод заключается в том, чтобы спросить себя, почему вы допустили ту или иную ошибку, вместо того, чтобы просто исправить ее и двигаться дальше. Это позволит вам определить общий принцип, которому вы не последовали. Например, предположим, что вы написали следующую функцию для вычисления среднего значения положительных чисел в целочисленном массиве:

```
double averagePositive(int array[ARRAYSIZE]) {
   int total = 0;
   int positiveCount = 0;
   for (int i = 0; i < ARRAYSIZE; i++) {
      if (array[i] > 0) {
         total += array[i];
         positiveCount++;
      }
   }
❶  return total / (double) positiveCount;
}
```

На первый взгляд эта функция выглядит нормально, но при ближайшем рассмотрении в ней обнаруживается проблема. Если в этом массиве нет положительных чисел, тогда значение переменной `positiveCount` будет равно нулю, когда цикл завершится, и это приведет к делению на ноль в конце функции ❶. Поскольку это деление с плавающей точкой, в программе может произойти не сбой, а скорее она продемонстрирует странное поведение в зависимости от того, как значение этой функции используется всей программой.

Если вы хотите быстро запустить свой код, то, обнаружив эту проблему, можете добавить код для обработки случая, когда значение переменной `positiveCount` равно нулю, и двигаться дальше. Но если вы хотите расти как программист, вы должны спросить себя, какую ошибку вы совершили. Разумеется, конкретная проблема заключается в том, что вы не учли возможности деления на ноль. Тем не менее, если ваш анализ ограничится этим, то в будущем вам это не поможет. Конечно, вы можете столкнуться с другой ситуацией, когда делитель может оказаться равным нулю, однако это бывает нечасто. Вместо этого мы должны спросить, какой общий принцип был нарушен. Ответ: мы всегда должны искать специальные случаи, которые могут нарушить работоспособность кода.

Если мы учтем этот общий принцип, мы с большей вероятностью заметим в наших ошибках закономерности и, следовательно, с большей вероятностью отследим эти ошибки в будущем. Вопрос: «Есть ли здесь возможность деления на ноль?» — не так полезен, как вопрос: «Каковы специальные случаи для этих данных?» Постановка более широкого вопроса будет напоминать не только о возможном делении на ноль, но и о пустых наборах данных, о данных вне ожидаемого диапазона и т.д.

Планирование с учетом недостатков дизайна

Чтобы справиться с недостатками дизайна, требуются другие подходы. Тем не менее первый шаг тот же: вы должны обнаружить эти недостатки. У многих людей возникают проблемы на этом этапе, поскольку они не любят смотреть на себя столь критично. Мы приучены скрывать личные недостатки. Это похоже на то, когда на собеседовании при приеме на работу вас спрашивают о вашем самом большом недостатке, и в ответ вы говорите какую-то глупость о том, что вы слишком сильно переживаете о качестве своей работы, вместо того, чтобы рассказать о *настоящем* недостатке. Однако так же, как у Супермена есть свой криптонит, даже лучшие программисты имеют свои слабые стороны.

Вот примерный (разумеется, не исчерпывающий) список недостатков программистов. Посмотрите, узнаете ли вы себя в любом из этих описаний.

Замысловатый дизайн

Программист с таким недостатком создает программы, которые имеют слишком много частей или слишком много шагов. Несмо-

тря на то, что программы работают, они не внушают доверия — как изношенная одежда, которая выглядит так, будто готова развалиться, стоит потянуть за нитку, кроме того, эти программы явно неэффективны.

Сложности с началом работы

Этот программист имеет высокую степень инерции. Либо из-за недостатка уверенности в своих навыках решения задач, либо из-за банальной прокрастинации, этот программист ждет слишком долго, прежде чем приступить к делу.

Отсутствие тестирования

Этот программист не любит формально тестировать код. Часто код подходит для общих случаев, но не справляется со специальными случаями. В других ситуациях код работает нормально, но не «масштабируется» для более крупных задач, которые этот программист не тестировал.

Чрезмерная уверенность

Уверенность в себе — это прекрасная вещь, данная книга призвана повысить уверенность читателей, однако слишком большая уверенность иногда может быть такой же проблемой, как слишком маленькая. Чрезмерная уверенность проявляется по-разному. Чрезмерно уверенный в себе программист может попытаться реализовать более сложное решение, чем нужно, или выделить слишком мало времени на завершение проекта, в результате чего получается недоработанная и полная ошибок программа.

Слабость в определенной области

Эта категория охватывает множество разнообразных проблем. Работа некоторых программистов протекает достаточно гладко, пока они не столкнуться с определенными концепциями. Возьмем темы, обсуждавшиеся в предыдущих главах этой книги. Даже после выполнения упражнений большинство программистов будут ориентироваться в некоторых областях лучше, чем в других. Например, возможно, программист теряется при работе с указателями или у него голова идет кругом из-за рекурсии. Может быть, у программиста проблемы с разработкой сложных классов. Дело не в том, что программист не может разобраться и решить задачу, а в том, что это требует тяжелой работы и напоминает езду по грязи.

Существуют разные способы противостоять своим крупным недостаткам, и как только вы их осознаете, вы легко сможете спланировать свою работу с их учетом. Например, если вы программист, который часто пропускает этап тестирования, сделайте тестирование неотъемлемым пунктом своего плана написания каждого модуля и не переходите к следующему модулю, пока не поставите галочку в соответствующем месте. Кроме того, вы можете рассмотреть парадигму

под названием «*разработка через тестирование*», в рамках которой сначала пишется тестовый код, а затем код для прохождения этого теста. Если у вас возникают проблемы с началом работы, используйте принципы деления или уменьшения задач и начинайте писать код, как можно быстрее, понимая, что позднее вам, возможно, придется его переписать. Если ваши проекты часто оказываются слишком сложными, добавьте специальный этап рефакторинга в свой мастер-план. Суть в том, что, какие бы недостатки как у программиста у вас ни были, если вы их осознаете, вы можете спланировать свою работу с их учетом. В этом случае ваши слабости больше не будут являться слабостями, а будут просто препятствиями на пути, которые вы будете объезжать, двигаясь к успешному завершению проекта.

Планирование с учетом ваших сильных сторон

Планирование с учетом слабых сторон в значительной степени направлено на избегание ошибок. Однако хорошее планирование заключается не только в том, чтобы не допустить ошибку. Смысл в том, чтобы достичь наилучшего из возможных результатов, учитывая существующие способности и любые ограничения, в которых вам приходится работать. Это означает, что вы также должны включить в мастер-план свои сильные стороны.

Вы можете подумать, что этот раздел не про вас или, по крайней мере, пока не про вас. В конце концов, если вы читаете эту книгу, вы все еще находитесь на этапе становления программистом. Вы можете усомниться в наличии у себя каких-либо сильных сторон на данном этапе своего развития. Я утверждаю, что у вас есть сильные стороны, даже если вы о них еще не знаете. Далее приведен список распространенных сильных сторон программистов, который ни в коем случае не является исчерпывающим, с описаниями каждой из них и подсказками, помогающими вам понять, насколько тот или иной пункт относится к вам.

Зоркий глаз на детали

Такой программист может предвидеть специальные случаи, предвосхищать потенциальные проблемы в плане производительности и никогда не позволяет общей картине заслонить важные детали, которые необходимо учесть, чтобы программа представляла собой завершенное и корректное решение. Программисты с такой сильной стороной обычно проверяют свои планы на бумаге перед кодированием, пишут код медленно и часто его тестируют.

Быстрое обучение

Такой программист быстро осваивает новые навыки, независимо от того, изучает ли он новую технику на уже известном языке или работает с новым каркасом приложений. Он получает удовольствие от изучения нового и может выбирать проекты, исходя из этого предпочтения.

Быстрое кодирование

Такому программисту не требуется много времени изучать справочник, чтобы создать функцию. Когда приходит время набирать текст, код течет из-под кончиков пальцев такого кодера без особых усилий и с малым количеством синтаксических ошибок.

Упорство

Некоторые программисты воспринимают досадную ошибку как личное оскорбление, которое нельзя игнорировать. Это похоже на то, что программа ударила программиста по лицу кожаной перчаткой, и ему необходимо ответить на вызов. Кажется, что такой программист всегда остается уравновешенным, решительным и никогда сильно не расстраивается, будучи уверенным в том, что при достаточных усилиях победа будет гарантирована.

Превосходные навыки решения задач

Вероятно, когда вы купили эту книгу, вы еще не были мастером решения задач, но теперь, после изучения этого руководства, вам многое может начать казаться более легким. Программист с этой чертой начинает предполагать потенциальные решения задачи еще во время чтения ее условия.

Энтузиазм

Для такого программиста рабочая программа похожа на удивительный ящик с игрушками. Энтузиаст очень любит, когда компьютер выполняет его или ее приказы, и обожает ставить перед машиной все новые задачи. Это может выражаться в добавлении в рабочую программу все большего количества функций — симптом, известный под названием *ползучий «улучшизм»*. Кроме того, такой программист может многократно подвергать программу рефакторингу для повышения ее производительности или дорабатывать ее так, чтобы она просто казалась программисту или пользователю более красивой.

Не у многих программистов проявляется сразу более двух из перечисленных сильных сторон, на самом деле, некоторые из них, как правило, отменяют друг друга. Однако у каждого программиста есть свои сильные стороны. Если вы не узнали себя ни в одном из приведенных описаний, это просто означает, что вы еще недостаточно себя знаете или ваши сильные стороны просто не вписываются ни в одну из выделенных мной категорий.

Как только вы определите свои сильные стороны, вам нужно будет учесть их в своем мастер-плане. Предположим, вы быстро пишете код. Очевидно, что это поможет довести любой проект до завершения, однако как вы можете использовать эту сильную сторону систематически? В области формальной разработки программного обеспечения существует подход, называемый *быстрым прототипированием*, в рамках которого программа сначала пишется, минуя этап подробного планирования, а затем улучшается с помощью

последовательных итераций до тех пор, пока результаты не начинают удовлетворять требованиям задачи. Если вы быстро пишете код, вы можете попробовать использовать этот метод, приступая к кодированию, как только у вас появится основная идея, и руководствуясь в процессе создания дизайна и разработки окончательного кода программы своим черновым прототипом.

Если вы быстро учитесь, возможно, вам следует начинать каждый проект с освоения новых ресурсов или методов, способных помочь вам в решении текущей задачи. Если вы являетесь программистом, который учится не очень быстро, но не расстраивается из-за неудач, возможно, вам следует начать проект с тех областей, которые кажутся вам самыми трудными, чтобы уделить достаточно времени каждой из них.

Таким образом, какими бы ни были ваши сильные стороны, убедитесь в том, что вы используете их в процессе своей работы. Составьте свой мастер-план так, чтобы как можно больше времени тратилось на то, что вы делаете лучше всего. Так вы получите не только наилучшие результаты, но и удовольствие от своей работы.

Составление мастер-плана

Давайте рассмотрим пример составления мастер-плана. Его составными частями являются все разработанные нами методы решения задач, а также анализ наших сильных и слабых сторон. В этом примере я буду использовать свои сильные и слабые стороны.

Что касается методов решения задач, то я использую все методы, которые я описал в этой книге, но особенно мне нравится техника «уменьшения задачи», поскольку ее использование позволяет мне ощутить конкретный прогресс в достижении своей цели. Если я в настоящее время не могу найти способ написания кода, который соответствует всей спецификации, я просто отбрасываю часть этой спецификации, пока не наберу скорость.

Моим самым большим недостатком в кодировании является чрезмерное рвение. Я люблю программировать, потому что мне нравится видеть, что компьютеры следуют моим инструкциям. Иногда это заставляет меня думать: «Попробую-ка я сделать это и посмотрю, что произойдет», — когда мне следует анализировать правильность того, что я только что написал. Опасность здесь заключается не в том, что в программе произойдет сбой, а в том, что программа либо покажется успешной, но не охватит все специальные случаи, либо, несмотря на свою эффективность, окажется не лучшим из возможных решений.

Мне нравятся элегантные дизайны программ, которые легко расширять и повторно использовать. Часто, когда я создаю код для крупных проектов, я трачу много времени на разработку альтернативных версий дизайна. В целом, это хорошая черта, но иногда это приводит к тому, что я трачу слишком много времени на этапе

проектирования, что не оставляет достаточно времени на фактическую реализацию выбранного дизайна. Кроме того, иногда из-за этого решение получается чрезмерно разработанным. То есть иногда решение оказывается более элегантным, расширяемым и надежным, чем необходимо. Поскольку каждый проект ограничен в плане времени и денег, лучшее решение должно сбалансировать стремление к высокому качеству программного обеспечения с необходимостью экономии ресурсов.

Я считаю, что моя самая сильная сторона в программировании заключается в том, что я хорошо усваиваю новые концепции и люблю учиться. В то время как некоторым программистам нравится снова и снова использовать одни и те же навыки, мне нравится работать над проектом, на котором я могу научиться чему-то новому, и я всегда с радостью принимаю вызов.

Далее представлен мой мастер-план для нового проекта, учитывающий мои сильные и слабые стороны.

Чтобы справиться со своей основной слабой стороной в области дизайна, я строго ограничу свое время на этапе проектирования или ограничу количество различных версий дизайна, которые я буду рассматривать, прежде чем двигаться дальше. Некоторым читателям это может показаться опасной идеей. Разве мы не должны потратить максимально возможное количество времени на этапе проектирования, прежде чем переходить к кодированию? Разве не правда, что большинство проектов терпят неудачу из-за того, что на начальных этапах было потрачено недостаточно времени, что привело к необходимости многочисленных компромиссов в конце? Это правомерные опасения, однако помните, что моей целью не являлось создание общего справочника по разработке программного обеспечения. Я создаю свой личный мастер-план для решения проблем программирования. Моя слабая сторона заключается в чрезмерной, а не в недостаточной разработке, поэтому правило, ограничивающее время на этапе проектирования, является для меня актуальным. Для другого программиста такое правило может оказаться катастрофическим, а некоторым программистам может понадобиться правило, заставляющее их тратить больше времени на разработку.

После завершения своего первоначального анализа я собираюсь посмотреть, предоставляет ли проект возможность для изучения новых методов, библиотек и т.д. Если да, я собираюсь написать небольшую пробную версию программы, чтобы опробовать эти новые навыки, прежде чем пытаться включить их в разрабатываемое мною решение.

Чтобы справиться с чрезмерным рвением, я мог бы включить небольшой этап для обзора кода после завершения этапа кодирования каждого модуля. Тем не менее это потребует от меня применения силы воли — после завершения модуля мне захочется опробовать его. Просто надеяться на то, что я смогу всякий раз себя

уговорить, — это все равно, что оставить открытый пакет с картофельными чипсами рядом с голодным человеком и удивиться, когда пакет опустеет. Лучше попытаться справиться со слабыми сторонами с помощью плана, который не требует от программиста борьбы с собственными инстинктами. Итак, что, если я создам две версии проекта: приблизительную версию, с которой можно делать все что угодно, и отшлифованную версию, которую можно сдать заказчику? Если я позволю себе поиграть с первой версией по своему усмотрению, но не позволю себе включать код в отшлифованную версию до тех пор, пока он не будет полностью проверен, я с большей вероятностью преодолею свой недостаток.

Решение любой задачи

Теперь, когда у нас есть мастер-план, мы готовы ко всему. В конечном счете, суть этой книги сводится к следующему: начните с задачи, с любой задачи, и найдите способ ее решения. Во всех предыдущих главах описания задач подталкивали нас в определенном направлении, однако в реальном мире большинство задач не подразумевает требований, связанных с использованием массива или рекурсии или инкапсуляцией части функций программы в класс. Эти решения принимает программист по ходу рабочего процесса.

Поначалу может показаться, что меньшее количество требований делает задачу более простой. В конце концов, требование к дизайну является ограничением, а разве ограничения не затрудняют решение задачи? Хотя это верно, верно и то, что все задачи подразумевают ограничения, просто в некоторых случаях они изложены более четко, чем в других. Например, если вам не было сказано, что конкретная задача требует динамически выделяемой структуры, не означает, что это решение является бесполезным. Более общим ограничениям задачи, касаются ли они производительности, модифицируемости, скорости разработки или чего-то еще, может быть сложнее или вообще невозможно удовлетворить, если мы сделаем неправильный выбор в плане дизайна.

Представьте, что группа друзей попросила вас выбрать фильм для совместного просмотра. Если один из друзей определенно хочет посмотреть комедию, другой не любит старые фильмы, а третий перечисляет пять фильмов, которые он недавно видел, поэтому не хочет смотреть их снова, то эти ограничения затруднят выбор. Однако если ни у кого нет никаких пожеланий, кроме «просто какого-нибудь хорошего фильма», ваша задача становится еще более сложной, и вы, скорее всего, выберете то, что вообще не понравится по крайней мере одному человеку из группы.

Поэтому крупные, пространно определенные, слабо ограниченные задачи являются самыми трудными из всех. Тем не менее к ним применимы рассмотренные в этой книге методы; их решение про-

сто требует чуть большего количества времени. Знания этих методов и ваш мастер-план помогут вам решить любую задачу.

Чтобы продемонстрировать суть того, о чем я говорю, я проведу вас по первым этапам программы, которая играет в классическую детскую игру «Виселица», но с подвохом.

Прежде чем перейти к описанию задачи, давайте рассмотрим основные правила этой игры. Первый игрок выбирает слово и говорит второму игроку, сколько в этом слове букв. Второй игрок пытается угадать букву. Если названная буква присутствует в слове, первый игрок показывает, в каком месте слова эта буква находится; если эта буква встречается в слове более одного раза, указываются все ее положения. Если названная буква в слове отсутствует, первый игрок добавляет фрагмент к рисунку повешенного. Если второй игрок угадывает все буквы в слове, то он побеждает, если первый игрок завершает рисунок, то побеждает он. Существуют разные правила относительно количества фрагментов, составляющих рисунок повешенного, поэтому в общем случае мы можем сказать, что игроки заранее договариваются о том, сколько «промахов» должен допустить второй игрок, чтобы первый выиграл.

Теперь, когда мы обговорили основные правила, давайте рассмотрим конкретную задачу, включая подвох.

Задача: жульничество при игре в «Виселицу»

Напишите программу, которая будет Игроком 1 в текстовой версии игры «Виселица» (то есть на самом деле вам не нужно рисовать повешенного, нужно только отслеживать количество неправильных догадок). Игрок 2 будет задавать уровень сложности игры, указывая длину слова, которое требуется угадать, а также количество неправильных догадок, которые приведут к проигрышу.

Подвох заключается в том, что программа будет жульничать. Вместо того чтобы выбирать слово в начале игры, программа может избегать выбора слова так, что, когда Игрок 2 проиграет, программа отобразит слово, соответствующее всей информации, предоставленной Игроку 2. Правильно угаданные буквы должны указываться в соответствующих им положениях, а неправильно угаданные буквы вообще не могут появиться в этом слове. Когда игра закончится, Игрок 1 (программа) сообщит Игроку 2 слово, которое было выбрано. Поэтому Игрок 2 никогда не сможет доказать, что игра его обманывает; просто вероятность выигрыша для Игрока 2 низкая.

Это не слишком крупная задача по стандартам реального мира, однако она достаточно велика, чтобы на ее примере можно было продемонстрировать проблемы, с которыми мы сталкиваемся при решении задачи программирования, которая обозначает результаты, но не содержит никакой методологии. Основываясь на описании задачи, вы можете запустить свою среду разработки и начать писать код с любого места. Это, конечно, было бы ошибкой, потому что нам всегда следует создавать код по плану, поэтому я собираюсь применить свой мастер-план к этой конкретной ситуации.

Первая часть моего мастер-плана ограничивает количество времени, которое я трачу на этапе проектирования. Чтобы реализовать это, мне нужно тщательно продумать дизайн, прежде чем приступать к созданию кода. Тем не менее я считаю, что в данном случае для нахождения решения этой задачи мне понадобится провести несколько экспериментов. Мой мастер-план также позволяет мне создать два проекта — грубый прототип и окончательное отшлифованное решение. Поэтому я позволю себе начать кодирование прототипа в любое время до начала реальной работы над дизайном, но не приступлю к кодированию итогового решения, пока не удостоверюсь в том, что дизайн разработан так, как нужно. Это не гарантирует того, что я буду полностью удовлетворен дизайном второго проекта, но максимизирует вероятность этого.

Теперь пришло время разделить задачу на части. В предыдущих главах мы иногда перечисляли все подзадачи, необходимые для итогового решения задачи, поэтому я хотел бы провести инвентаризацию подзадач. Тем не менее на данном этапе это было бы сложно сделать, поскольку я не знаю, что программа сделает на самом деле, чтобы обмануть второго игрока. Мне нужно более подробно исследовать этот вопрос.

Нахождение возможности для жульничества

Обман в игре «Виселица» достаточно специфичен, поэтому я не ожидаю найти какое-либо руководство в обычных источниках с компонентами; там вряд ли есть шаблон под названием *БесчестнаяСтратегия*. На данном этапе у меня есть лишь смутное представление о том, как можно сжульничать в этой игре. Я думаю, что загадаю исходное слово и буду придерживаться его, пока Игрок 2 выбирает буквы, которые отсутствуют в этом слове. Как только Игрок 2 назовет букву, которая на самом деле есть в этом слове, я выберу другое слово, если можно будет найти такое, в котором нет ни одной из названных до сих пор букв. Другими словами, я буду стараться как можно дольше отказывать Игроку 2 в правильности его догадки. Это идея, но мне нужно нечто большее, чем идея, — мне нужно что-то, что я могу реализовать.

Чтобы развить свои идеи, я разберу пример на бумаге, взяв на себя роль Игрока 1 и работая со списком слов. Чтобы не усложнять, я предположу, что Игрок 2 запросил слово, состоящее из трех букв, и что полный список известных мне трехбуквенных слов представлен в первом столбце табл. 8.1. Я предполагаю, что первым загаданным мной словом является первое слово в списке — *bat*. Если Игрок 2 назовет любую букву, кроме *b*, *a* или *t*, я скажу «нет», и мы подойдем на один шаг ближе к завершению рисунка повешенного. Если Игрок 2 угадает букву в слове, то я выберу другое слово, которое не содержит эту букву.

Тем не менее, глядя на свой список, я не уверен, что эта стратегия является лучшей. В некоторых ситуациях она, вероятно, име-

ет смысл. Предположим, что Игрок 2 называет букву *b*. Ни одно из оставшихся в списке слов не содержит букву *b*, поэтому я могу выбрать в качестве загаданного слова любое из них. Это также означает, что я минимизировал ущерб; я исключил из своего списка только одно возможное слово. Но что, если Игрок 2 назовет букву *a*? Если я просто скажу «нет», мне придется исключить все слова, содержащие букву *a*, при этом останется только три слова во втором столбце табл. 8.1, из которых я смогу выбрать. Если бы вместо этого я решил признать наличие буквы *a* в загаданном слове, у меня осталось бы пять слов на выбор, как показано в третьем столбце. Однако заметьте, что этот расширенный выбор существует только потому, что все пять слов содержат букву *a* в одном и том же положении. Как только я признаю догадку правильной, я должен буду точно указать, где в слове находится буква. Я буду чувствовать себя гораздо спокойнее, если у меня будет больше вариантов слов, среди которых я могу сделать выбор в ответ на будущие догадки.

Табл. 8.1. Образец списка слов

Все слова	Слова без буквы a	Слова с буквой a
bat	dot	bat
car	pit	car
dot	top	eat
eat		saw
pit		tap
saw		
tap		
top		

Кроме того, даже если мне удастся избежать раскрытия букв в начале игры, мне следует ожидать того, что Игрок 2 в конечном итоге сделает правильное предположение. Например, Игрок 2 может начать с перечисления всех гласных. Поэтому в какой-то момент мне нужно будет решить, что делать, когда буква будет открыта, и, судя по моему эксперименту с образцом списка, мне нужно будет найти местоположение (или местоположения), в которых эта буква появляется чаще всего. Это наблюдение заставило меня понять, что я неправильно подошел к обдумыванию возможного обмана. Мне никогда не следует загадывать слово, даже на время, мне нужно просто отслеживать все возможные слова, из которых при необходимости я мог бы выбрать одно.

Имея в виду эту идею, я могу теперь по-другому определить способ обмана: следует хранить как можно больше слов в списке слов-кандидатов. При каждой высказанной Игроком 2 догадке программа должна принять решение. Признает ли она догадку правильной или не правильной? Если догадка правильная, то, в каком месте слова находится угаданная буква? Я сделаю так, что моя программа будет постоянно сокращать список слов-кандидатов и после каждой высказанной догадки принимать решение, при котором в этом списке остается максимальное количество слов.

Необходимые операции для обмана в игре «Виселица»

Теперь я понимаю задачу достаточно хорошо, чтобы создать список подзадач. В случае проблемы такого размера существует довольно большая вероятность того, что список, составленный на раннем этапе, позволит обойтись без некоторых операций. Это нормально, поскольку мой мастер-план предусматривает возможность того, что я не создам идеальный дизайн с первого раза.

Сохранение и поддержание списка слов

Эта программа должна содержать список допустимых английских слов. Поэтому программе необходимо будет прочитать список слов из файла и сохранить его внутри в каком-то формате. Во время игры, когда программа будет обманывать игрока, этот список будет уменьшаться или из него будут извлекаться слова.

Создание подсписка слов определенной длины

Учитывая мое намерение поддерживать список слов, которые могут быть загаданы (список слов-кандидатов), я должен начать игру со списком слов указанной Игроком 2 длины.

Отслеживание выбранных букв

Программа должна будет помнить, какие буквы были названы Игроком 2, сколькие из них были неправильными, а также в каком месте загаданного слова находятся буквы, которые были сочтены правильными.

Подсчет слов, в которых отсутствует названная буква

Чтобы облегчить обман Игрока 2, мне нужно знать, сколько слов в списке не содержит последней названной буквы. Помните, что программа решит, присутствует ли последняя названная Игроком 2 буква в загаданном слове, руководствуясь целью оставить в списке слов-кандидатов максимальное количество слов.

Определение наибольшего количества слов, исходя из буквы и положения

Эта операция кажется самой сложной. Предположим, что Игрок 2 только что угадал букву *d*, а загаданное в текущей игре слово состоит из трех букв. Возможно, текущий список слов-кандидатов содержит всего 10 слов, включающих букву *d*, но это не важно, поскольку программе нужно будет указать, в каком месте загаданного слова находится эта буква. Давайте назовем расположение букв в слове шаблоном. Таким образом, *d??* — это трехбуквенный шаблон, который указывает, что первой буквой в слове является *d*, а двумя другими — буквы отличные от *d*. Рассмотрим табл. 8.2. Предположим, что список в первом столбце содержит все трехбуквенные слова с буквой *d*, известные программе. Остальные столбцы разбивают этот список согласно шаблону. Наиболее часто встречается шаблон *??d* — в 17 словах. Это число, 17, будет сравниваться с количеством слов в списке

слов-кандидатов, которые не содержат буквы *d*, чтобы решить, как следует оценивать догадку — как промах или как попадание.

Создание подсписка слов, соответствующих шаблону

Когда программа сообщает, что Игрок 2 угадал букву, она создает новый список слов-кандидатов, содержащий только те слова, которые соответствуют выбранному буквенному шаблону. В предыдущем примере, если мы объявили, что буква *d* угадана правильно, то третий столбец в табл. 8.2 становится новым списком слов-кандидатов.

Игра до конца

После разработки всех остальных операций мне нужно будет написать код, объединяющий все вместе и фактически играющий в игру. Программа должна многократно запрашивать у Игрока 2 (пользователя) букву, определять, станет ли список слов-кандидатов более длинным, если догадка будет отклонена или признана правильной, соответственно уменьшать список слов, а затем отображать получившееся загаданное слово с открытыми правильно угаданными буквами, наряду с перечислением всех названных ранее букв. Этот процесс будет продолжаться до окончания игры, пока не выиграет один из игроков, условия чего мне еще предстоит сформулировать.

Табл. 8.2. Слова, состоящие из трех букв

Все слова	?dd	??d	d??	d?d
add	add	aid	day	did
aid	odd	and	die	
and		bad	doe	
bad		bed	dog	
bed		bid	dry	
bid		end	due	
day		fed		
did		had		
die		hid		
doe		kid		
dog		led		
dry		mad		
due		mod		
end		old		
fed		red		
had		rid		
hid		sad		
kid				
led				
mad				
mod				
odd				
old				
red				
rid				
sad				

Исходный дизайн

Хотя может показаться, что в предыдущем списке необходимых операций перечислены просто сырые факты, мы принимаем решения, касающиеся дизайна. Рассмотрим операцию «Создание подсписка слов, соответствующих шаблону». Эта операция будет иметь место в моем решении или по крайней мере в его первоначальной версии, однако, строго говоря, эта операция вовсе не является *обязательной*. Это же касается операции «Создание подсписка слов определенной длины». Вместо поддержания постепенно уменьшающегося списка слов-кандидатов я мог бы на протяжении всей игры хранить исходный главный список слов. Тем не менее это усложнило бы большинство других операций. Операция «Подсчет слов, в которых отсутствует буква» не может подразумевать простой перебор списка слов-кандидатов и подсчет слов, не содержащих указанную букву. Поскольку она осуществляла бы поиск в основном списке, ей также пришлось бы проверять длину каждого слова и то, соответствует ли это слово открытым до сих пор буквам. Я считаю, что выбранный мной путь в целом более прост, однако я должен знать, что решения, принятые даже на этих ранних этапах, влияют на итоговый дизайн.

Тем не менее кроме первоначального разбиения задачи на подзадачи, мне предстоит принять и другие решения.

Хранение списков слов

Основной структурой данных программы будет список слов, который программа будет уменьшать на протяжении всей игры. Выбирая структуру, я сделал следующие наблюдения. Во-первых, я не думаю, что мне потребуется произвольный доступ к словам в списке, вместо этого я всегда буду обрабатывать список целиком, от начала до конца. Во-вторых, мне не известен размер нужного мне исходного списка. В-третьих, мне предстоит часто сокращать список. Наконец, в-четвертых, вероятно, в этой программе пригодятся методы стандартного класса `string`. На основе всех этих наблюдений я принимаю решение, что моим первоначальным выбором для этой структуры будет стандартный шаблонный класс `list` с типом элемента `string`.

Отслеживание отгаданных букв

Концептуально выбранные буквы представляют собой набор, то есть буква либо выбрана, либо нет, ни одна буква не может быть выбрана более одного раза. Таким образом, вопрос сводится к тому, является ли конкретная буква алфавита частью набора «выбранных букв». Поэтому я собираюсь представить выбранные буквы в виде массива типа `bool` размером 26. Если массив называется `guessedLetters`, то `guessedLetters[0]` имеет значение `true`, если буква *a* была угадана ранее, в противном случае значением будет `false`; `guessedLetters[1]` соответствует букву *b* и т.д. Я буду использовать методы преобразования диапазона, кото-

рые мы уже обсуждали в этой книге, чтобы обеспечить преобразование между буквой в нижнем регистре и соответствующей ей позицией в массиве. Если `letter` — это символ, представляющий букву в нижнем регистре, то `guessedLetters[letter - 'a']` — это соответствующее ей место.

Хранение шаблонов

В одной из операций, код для которых мне предстоит написать, «Создание подсписка слов, соответствующих шаблону», будет использоваться шаблон положений буквы в слове. Этот шаблон будет создаваться операцией «Определение наибольшего количества слов, исходя из буквы и положения». Какой формат я буду использовать для этих данных? Шаблон представляет собой ряд чисел, соответствующих местоположению конкретной буквы. Существует множество способов для хранения этих чисел, но для простоты я буду использовать другой список `list` с типом элемента `int`.

Пишу ли я класс?

Поскольку я кодирую эту программу на языке C++, я могу использовать или не использовать объектно ориентированное программирование по своему усмотрению. Моя первая мысль заключается в том, что многие из операций в моем списке можно естественным образом объединить в класс под названием `wordList`, возможно, с методами, позволяющими удалять слова на основе заданных критериев (то есть длины и шаблона). Однако, поскольку сейчас я пытаюсь избежать принятия решений, касающихся дизайна, которые мне придется в дальнейшем отменить, я собираюсь сделать свою первую, черновую программу полностью процедурной. Когда я разработаю все сложные аспекты программы и фактически напишу код для всех операций в своем списке, я смогу принять решение относительно применимости принципов объектно-ориентированного программирования к окончательной версии программы.

Первичное кодирование

Теперь начинается самое интересное. Я запускаю свою среду разработки и берусь за дело. В этой программе будет использоваться несколько классов из стандартной библиотеки, поэтому для ясности позвольте мне сначала установить их все.

```
#include <iostream>
using std::cin;
using std::cout;
using std::ios;
#include <fstream>
using std::ifstream;
#include <string>
using std::string;
#include <list>
using std::list;
using std::iterator;
#include <cstring>
```

Теперь я готов начать кодирование операций из моего списка. В какой-то мере я мог бы кодировать операции в любом порядке, однако я собираюсь начать с написания функции для чтения простого текстового файла со словами в выбранную мною структуру list<string>. На данном этапе я понимаю, что мне нужно найти существующий мастер-файл со словами — я не хочу набирать его самостоятельно. К счастью, на поисковый запрос *word list* система Google выдает ряд сайтов, содержащих списки английских слов в текстовом формате, по одному слову на строку файла. Я уже знаком с чтением текстовых файлов средствами языка C++, но если бы это было не так, я сначала написал бы небольшую тестовую программу, чтобы освоить этот навык, прежде чем интегрировать эту способность в программу, которая обманывает игрока в игре «Виселица». Эта практика обсуждается далее в главе.

Теперь, когда у меня есть файл, я могу написать функцию:

```
list<string> readWordFile(char * filename) {
  list<string> wordList;
❶ ifstream wordFile(filename, ios::in);
❷ if (wordFile == NULL) {
     cout << "File open failed. \n";
     return wordList;
  }
  char currentWord[30];
❸ while (wordFile >> currentWord) {
  ❹ if (strchr(currentWord, '\'') == 0) {
        string temp(currentWord);
        wordList.push_back(temp);
    }
  }
  return wordList;
}
```

Эта функция проста, поэтому я сделаю всего несколько коротких комментариев. Если вы никогда не встречали его раньше, объект ifstream ❶ представляет собой входной поток, который работает так же, как cin, за исключением того, что данные считываются из файла, а не из стандартного потока ввода. Если конструктор не может открыть файл (обычно это означает, что файл не найден), объект будет иметь значение NULL, что я явно проверяю ❷. Если файл существует, то он обрабатывается в цикле ❸, который считывает каждую строку файла в массив символов, преобразует массив в объект string и добавляет его в список list. Выбранный мной файл с английскими словами содержит слова с апострофами, которые недопустимы в нашей игре, поэтому я явно исключаю их ❹.

Далее я пишу функцию для отображения всех слов в моем списке list<string>. Это не входит в мой требуемый список операций, и я бы не использовал ее в игре (в конце концов, это бы только помогло Игроку 2, которого я пытаюсь обмануть), однако это хороший способ проверить, правильно ли работает моя функция readWordFile:

```
void displayList(❶const list<string> & wordList) {
   ❷ list<string>::const_iterator iter;
      iter = wordList.begin();
      while (iter != wordList.end()) {
         cout << ❸iter->c_str() << "\n";
         iter++;
      }
}
```

По сути это тот же код обхода списка, который был представлен в предыдущей главе. Обратите внимание на то, что я объявил параметр как ссылку const ❶. Поскольку в начале список может быть довольно большим, наличие параметра типа ссылки снижает расход ресурсов, связанный с вызовом функции, в то время как параметр типа значения предполагал бы копирование всего списка. При объявлении этого ссылочного параметра const означает, что функция не изменит список, что делает код более удобным для чтения. Константный список требует использования константного итератора ❷. Поток cout не может выводить строковый объект, поэтому данный метод создает эквивалентный нуль-терминированный массив символов с помощью c_str ()❸.

Я использую эту же базовую структуру для записи функции, которая подсчитывает количество слов в списке, не содержащих указанную букву:

```
int countWordsWithoutLetter(const list<string> & wordList, char letter) {
   list<string>::const_iterator iter;
   int count = 0;
   iter = wordList.begin();
   while (iter != wordList.end()) {
      ❶ if (iter->find(letter) == string::npos) {
            count++;
         }
         iter++;
   }
   return count;
}
```

Как видите, это тот же базовый цикл обхода. Внутри я вызываю метод find класса string ❶, который возвращает позицию своего параметра char в объекте string, возвращая специальное значение npos, если символ не найден.

Я использую эту же базовую структуру для записи функции, которая удаляет все слова из списка, которые не соответствуют указанной длине:

```
void removeWordsOfWrongLength(list<string> & wordList, int acceptableLength)
{
   list<string>::iterator iter;
   iter = wordList.begin();
   while (iter != wordList.end()) {
      if (iter->length() != acceptableLength) {
      ❶ iter = wordList.erase(iter);
      } else {
      ❷ iter++;
      }
   }
}
```

Эта функция является хорошим примером того, как каждая написанная вами программа позволяет углубить понимание принципов своей работы. Мне было легко написать данную функцию, поскольку я понимал, что происходит «за кадром», благодаря написанным мною ранее программам. В этой функции используется базовый код обхода из предыдущих функций, однако внутри цикла код становится интереснее. Метод `erase()` удаляет элемент, указанный итератором `iterator`, из объекта `list`. Но из опыта реализации шаблона итератора для связного списка, описанного в главе 7, я знаю, что `iterator` почти наверняка является указателем. Из опыта работы с указателями, описанного в главе 4, я знаю, что указатель бесполезен и часто опасен, когда является висячей ссылкой на то, что было удалено. Поэтому я знаю, что после этой операции мне необходимо присвоить действительное значение `iter`. К счастью, разработчики метода `erase()` предусмотрели возможность возникновения такой проблемы и сделали так, чтобы этот метод возвращал новый `iterator`, который указывает на элемент, следующий за тем, который мы только что стерли, поэтому я могу присвоить это значение `iter` ❶. Также обратите внимание на то, что я явно продвигаю `iter` ❷ только тогда, когда текущая строка не удаляется из списка, потому что присвоение `erase()`возвращенного значения фактически продвигает `iterator`, и я не хочу пропускать какие-либо элементы.

Теперь перейдем к трудной части — к поиску наиболее распространенного конкретного буквенного шаблона в списке оставшихся слов. Это еще одна возможность для использования техники разделения задачи. Я знаю, что одной из подзадач этой операции является определение того, соответствует ли конкретное слово определенному шаблону. Помните, что шаблон представляет собой `list<int>`, причем каждый элемент `int` соответствует местоположению буквы в слове, а для того, чтобы слово соответствовало шаблону, буква не просто должна находиться в указанных позициях в слове, но и *не* должна находиться больше нигде в слове. Учитывая это, я собираюсь проверить строку на соответствие путем ее обхода. Для каждой позиции в строке, если указанная буква присутствует, я удостоверюсь, что эта позиция находится в шаблоне, а если там находится какая-то другая буква, я удостоверюсь, что эта позиция в шаблоне отсутствует.

Упрощая еще больше, я сначала напишу отдельную функцию, чтобы проверить, отображается ли номер конкретной позиции в шаблоне:

```
bool numberInPattern(const list<int> & pattern, int number) {
  list<int>::const_iterator iter;
  iter = pattern.begin();
  while (iter != pattern.end()) {
    if (*iter == number) {
      return true;
    }
    iter++;
  }
  return false;
}
```

Этот код было довольно просто написать, основываясь на предыдущих функциях. Я просто обхожу `list` в поисках `number`. Если я нахожу его, то возвращаю значение `true`, а если добираюсь до конца списка, то возвращаю `false`. Теперь я могу реализовать общий тест на соответствие шаблону:

```
bool matchesPattern(string word, char letter, list<int> pattern) {
   for (int i = 0; i < word.length(); i++) {
      if (word[i] == letter) {
         if (!numberInPattern(pattern, i)) {
            return false;
         }
      } else {
         if (numberInPattern(pattern, i)) {
             return false;
         }
      }
   }
   return true;
}
```

Как видите, эта функция соответствует описанному ранее плану. Для каждого символа в строке, если он соответствует значению `letter`, код проверяет, находится ли текущая позиция в шаблоне. Если символ не соответствует значению `letter`, код производит проверку, чтобы удостовериться в том, что позиция не находится в шаблоне. Если хотя бы одна позиция не соответствует шаблону, слово отклоняется; в противном случае будет достигнут конец слова, и оно будет принято.

Сейчас мне кажется, что найти наиболее распространенный шаблон будет проще, если каждое слово в списке содержит указанную букву. Поэтому я пишу быструю функцию для отбрасывания слов, не содержащих эту букву:

```
void removeWordsWithoutLetter(list<string> & wordList, char requiredLetter) {
   list<string>::iterator iter;
   iter = wordList.begin();
   while (iter != wordList.end()) {
      if (iter->find(requiredLetter) == string::npos) {
         iter = wordList.erase(iter);
      } else {
         iter++;
      }
   }
}
```

Этот код представляет собой просто комбинацию идей, использованных в предыдущих функциях. Теперь, когда я думаю об этом, мне понадобится и обратная функция, которая отбрасывает все слова, которые *содержат* указанную букву. Я буду использовать ее для уменьшения списка слов-кандидатов, когда программа признает последнюю догадку промахом:

```
void removeWordsWithLetter(list<string> & wordList, char forbiddenLetter) {
   list<string>::iterator iter;
   iter = wordList.begin();
```

```
   while (iter != wordList.end()) {
      if (iter->find(forbiddenLetter) != string::npos) {
         iter = wordList.erase(iter);
      } else {
         iter++;
      }
   }
}
```

Теперь я готов найти наиболее распространенный шаблон в списке слов для данной буквы. Я рассмотрел ряд подходов и выбрал тот, который, как мне кажется, я мог бы легче всего реализовать. Во-первых, я воспользуюсь вызовом вышеприведенной функции, чтобы удалить все слова, не содержащие указанную букву. Затем я возьму первое слово в списке, определю его шаблон и подсчитаю количество других слов в списке, соответствующих этому же шаблону. Все эти слова будут стираться из списка по мере их подсчета. Затем процесс повторится снова с любым словом, находящимся в начале списка, и так будет продолжаться, пока список не опустеет. Результат выглядит следующим образом:

```
void mostFreqPatternByLetter(❶ list<string> wordList, char letter,
                             ❷ list<int> & maxPattern,
                             ❸ int & maxPatternCount) {
 ❹ removeWordsWithoutLetter(wordList, letter);
   list<string>::iterator iter;
   maxPatternCount = 0;
 ❺ while (wordList.size() > 0) {
      iter = wordList.begin();
      list<int> currentPattern;
   ❻ for (int i = 0; i < iter->length(); i++) {
         if ((*iter)[i] == letter) {
            currentPattern.push_back(i);
         }
      }
      int currentPatternCount = 1;
      iter = wordList.erase(iter);
   ❼ while (iter != wordList.end()) {
         if (matchesPattern(*iter, letter, currentPattern)) {
            currentPatternCount++;
            iter = wordList.erase(iter);
         } else {
            iter++;
         }
      }
   ❽ if (currentPatternCount > maxPatternCount) {
            maxPatternCount = currentPatternCount;
            maxPattern = currentPattern;
      }
      currentPattern.clear();
   }
}
```

Список `list` представляет собой параметр типа значения ❶, поскольку в процессе обработки данная функция будет уменьшать список, пока в нем ничего не останется, и я не хочу влиять на параметр, передающийся вызывающим кодом. Обратите внимание на то, что `maxPattern` ❷ и `maxPatternCount` ❸ являются только исходящими па-

раметрами; они будут использоваться для отправки наиболее часто встречающегося шаблона и количества случаев его появления обратно в вызывающий код. Я удаляю все слова, не содержащие значение `letter` ❹. Затем я вхожу в основной цикл функции, который продолжается до тех пор, пока список не опустеет ❺. Код внутри цикла имеет три основных раздела. Во-первых, цикл `for` создает шаблон для первого слова в списке ❻. Затем цикл `while` подсчитывает, сколько слов в списке соответствует этому шаблону ❼. Наконец, мы проверяем, превышает ли это значение наибольшее из определенных до сих пор значений, используя стратегию «Царь горы», описанную в главе 3 ❽.

Последняя нужная мне служебная функция будет отображать все отгаданные до сих пор буквы. Помните, что я храню их как массив из 26 значений `bool`:

```
void displayGuessedLetters(bool letters[26]) {
   cout << "Letters guessed: ";
   for (int i = 0; i < 26; i++) {
      if (letters[i]) cout << ❶(char)('a' + i) << " ";
   }
   cout << "\n";
}
```

Обратите внимание на то, что я добавляю базовое значение одного диапазона, в данном случае символ `a`, к значению из другого диапазона ❶, этот метод мы впервые использовали в главе 2.

Теперь, когда все ключевые подзадачи сформулированы, я готов попытаться решить задачу целиком, однако у меня есть много функций, которые не были полностью протестированы, и я хотел бы протестировать их как можно скорее. Поэтому, вместо того, чтобы пытаться решить оставшуюся часть задачи за один шаг, я собираюсь уменьшить эту задачу. Я сделаю это, превратив в константы некоторые переменные, например размер загаданного слова.

Поскольку я собираюсь отбросить эту версию, я спокойно могу включить всю игровую логику в функцию `main`. Тем не менее из-за внушительного объема кода я буду представлять его поэтапно.

```
int main () {
❶ list<string> wordList = readWordFile("wordlist.txt");
   const int wordLength = 8;
   const int maxMisses = 9;
❷ int misses = 0;
❸ int discoveredLetterCount = 0;
❹ removeWordsOfWrongLength(wordList, wordLength);
❺ char revealedWord[wordLength + 1] = "********";
❻ bool guessedLetters[26];
   for (int i = 0; i < 26; i++) guessedLetters[i] = false;
❼ char nextLetter;
   cout << "Word so far: " << revealedWord << "\n";
```

Этот первый раздел кода устанавливает константы и переменные, которые нам понадобятся для игры. Большая часть этого кода не

требует пояснений. Список слов создается из файла ❶, а затем урезается до указанной длины слова, которая в данном случае имеет постоянное значение 8 ❹. В переменной misses ❷ хранится количество неправильных догадок Игрока 2, в то время как discoveredLetterCount ❸ отслеживает количество позиций, занятых в слове угаданной буквой, (поэтому если буква *d* присутствует в слове дважды, то угадывание буквы *d* увеличивает это значение на две единицы). В переменной revealedWord хранится загаданное слово в том виде, в котором оно в настоящее время известно Игроку 2, со звездочками вместо букв, которые еще не были угаданы ❺. Массив логических значений guessedLetters ❻ отслеживает конкретные угаданные до сих пор буквы; цикл устанавливает все значения на false. Наконец, nextLetter ❼ сохраняет текущую догадку Игрока 2. После вывода начального значения переменной revealedWord я готов к запуску основного игрового цикла.

```
❶ while (discoveredLetterCount < wordLength && misses < maxMisses) {
      cout << "Letter to guess: ";
      cin >> nextLetter;
   ❷ guessedLetters[nextLetter - 'a'] = true;
   ❸ int missingCount = countWordsWithoutLetter(wordList, nextLetter);
      list<int> nextPattern;
      int nextPatternCount;
   ❹ mostFreqPatternByLetter(wordList, nextLetter, nextPattern,
                              nextPatternCount);
      if (missingCount > nextPatternCount) {
        ❺ removeWordsWithLetter(wordList, nextLetter);
         misses++;
      } else {
        ❻ list<int>::iterator iter = nextPattern.begin();
         while (iter != nextPattern.end()) {
            discoveredLetterCount++;
            revealedWord[*iter] = nextLetter;
            iter++;
         }
         wordList = reduceByPattern(wordList, nextLetter, nextPattern);
      }
      cout << "Word so far: " << revealedWord << "\n";
      displayGuessedLetters(guessedLetters);
}
```

Существует два условия, которые могут привести к окончанию игры. Либо Игрок 2 угадает все буквы в слове, так что значение discoveredLetterCount достигнет значения wordLength, либо промахи Игрока 2 позволят завершить рисунок повешенного, и в этом случае значение misses будет равно значению maxMisses. Таким образом, цикл продолжает выполняться до тех пор, пока одно из этих условий не окажется истинным ❶. Внутри цикла после получения от пользователя очередной догадки обновляется соответствующая позиция в массиве guessedLetters ❷. Затем начинается жульничество. С помощью countWordsWithoutLetter программа определяет, сколько слов-кандидатов останется в списке слов, если догадка будет объявлена промахом ❸, а с помощью mostFreqPatternByLetter она определяет максимальное количество оставшихся слов, если догадка будет объявлена попаданием ❹. Если первое значение будет больше второго,

то слова с угаданной буквой отбрасываются, а значение misses инкрементируется ❺. Если второе значение больше первого, мы возьмем шаблон, заданный mostFreqPatternByLetter, и обновим значение revealedWord, удалив при этом все слова из списка, которые не соответствуют этому шаблону ❻.

```
  if (misses == maxMisses) {
      cout << "Sorry. You lost. The word I was thinking of was '";
      cout << ❶(wordList.cbegin())->c_str() << "'.\n";
  } else {
    cout << "Great job. You win. Word was '" << revealedWord << "'.\n";
  }
  return 0;
}
```

Остальная часть кода представляет собой то, что я называю *вскрытием цикла*, в котором действие, предпринимаемое после завершения цикла, определяется условием, которое «убило» этот цикл. Здесь либо нашей программе удастся обмануть пользователя и выиграть, либо Игрок 2, несмотря ни на что, заставит программу раскрыть слово целиком. Обратите внимание на то, что в случае победы программы в списке должно остаться хотя бы одно слово, поэтому я просто отображаю первое слово ❶ и утверждаю, что именно оно и было загадано с самого начала. Более хитрая программа могла бы случайным образом выбрать одно из оставшихся слов, чтобы уменьшить вероятность того, что противник обнаружит обман.

Анализ первоначальных результатов

Я собрал весь код воедино и протестировал. Он работает, однако, очевидно, существует множество возможностей для его улучшения. Помимо дизайна, программе недостает большого количестве функций. Она не позволяет пользователю указать длину загаданного слова или количество допустимых неправильных догадок. Она не проверяет, называлась ли угаданная буква раньше. В этом отношении она даже не проверяет, является ли входной символ строчной буквой. В этой программе также отсутствует множество полезных функций интерфейса, например сообщающих пользователю количество оставшихся допустимых промахов. Я думаю, было бы неплохо, если бы программа предлагала пользователю сыграть снова, чтобы ему не приходилось повторно ее запускать.

Что касается дизайна, то, когда я начну обдумывать готовую версию программы, я серьезно рассмотрю методы объектно-ориентированного дизайна. Класс wordlist сейчас представляется естественным выбором. Основная функция кажется мне слишком большой. Мне нравится модульный, простой в обслуживании дизайн, поэтому основная функция должна быть короткой и просто направлять трафик между подпрограммами, которые выполняют настоящую работу. Таким образом, моя основная функция должна быть разбита на несколько функций. Вероятно, мне следует переосмыслить некоторые

из моих первоначальных вариантов дизайна. Например, оглядываясь назад, сохранение шаблонов в виде `list<int>` кажется громоздким. Может быть, мне стоит попробовать использовать массив значений типа `bool` аналогично тому, как я использовал массив `guessedLetters`?

Или, может быть, мне следует поискать совершенно другую структуру. Теперь мне пришло время отступить, чтобы посмотреть, существуют ли какие-либо возможности для изучения новых методов решения этой задачи. Мне интересно, есть ли еще не рассмотренные мной специализированные структуры данных, которые могут оказаться полезными. Даже если я в конечном итоге останусь при своем первоначальном выборе, в процессе исследования я мог бы многому научиться.

Несмотря на то, что все эти решения по-прежнему не окончательны, я ощущаю прогресс в работе над этим проектом. Хорошо иметь рабочую программу, которая отвечает основным требованиям задачи. Я могу легко поэкспериментировать с различными дизайнерскими идеями на этой черновой версии, будучи уверенным в том, что у меня уже есть решение и я просто ищу лучший его вариант.

СОЗДАНИЕ ТОЧКИ ВОССТАНОВЛЕНИЯ

Операционная система Microsoft Windows создает то, что называется точкой восстановления, перед установкой или модификацией компонентов системы. Точка восстановления содержит резервные копии ключевых файлов, например реестра. Если установка или обновление приведет к серьезной проблеме, его можно фактически «откатить» или отменить, скопировав файлы из точки восстановления.

Я настоятельно рекомендую использовать тот же подход при работе с вашим собственным исходным кодом. Когда у вас есть рабочая программа, которую вы предполагаете позднее изменить, сделайте копию всего проекта и измените только копию. Это быстро, и в будущем может сэкономить вам время, если с вашими изменениями возникнут проблемы. Программисты могут легко попасть в ловушку, думая, что если они сделали что-то один раз, то смогут сделать это снова. Как правило, это так, но есть большая разница между пониманием того, что вы можете что-то сделать снова, и тем, чтобы иметь возможность восстановить старый исходный код, к которому вы можете мгновенно обратиться.

Вы также можете использовать программное обеспечение для управления версиями, которое автоматизирует копирование и хранение файлов проекта. Программное обеспечение для управления версиями делает больше, чем создание «точки восстановления»; например, оно также может позволить нескольким программистам работать независимо друг от друга над одними и теми же файлами. Хотя описание таких инструментов выходит за рамки этой книги, вам следует исследовать их в процессе своего профессионального развития.

Искусство решения задач

Узнали ли вы все методы решения задач, которые я использовал в своей программе до сих пор? У меня был план решения задачи. Как всегда, это самый важный метод. Я начал с того, что знал, при созда-

нии первой версии своего решения и использовал пару структур данных, с которыми был очень хорошо знаком, — массивы и класс `list`. Я уменьшил функциональность, чтобы упростить процесс написания черновой версии и иметь возможность протестировать свой код раньше, чем я мог бы это сделать в противном случае. Я разделил задачу на операции и сделал каждую операцию отдельной функцией, чтобы иметь возможность работать над частями программы по отдельности. Когда я не был уверен в способе обмана, я экспериментировал, что позволило мне переформулировать «обман» как «максимизацию размера списка слов-кандидатов», что представляло собой конкретную концепцию, подлежащую кодированию. В процессе кодирования операций я применил методы, аналогичные тем, которые использовались в этой книге.

Мне также удавалось не расстраиваться, хотя, полагаю, тут вам придется поверить мне на слово.

Прежде чем двигаться дальше, позвольте мне пояснить, что я продемонстрировал шаги, которые предпринял *я*, чтобы добраться до этого этапа в процессе решения данной задачи. Вы не обязательно пойдете тем же путем. Приведенный выше код не является лучшим решением задачи, и он не обязательно превосходит то, что могли бы придумать вы. Надеюсь, что это продемонстрирует вам то, что любая задача, независимо от размера, может быть решена с использованием вариаций одних и тех же базовых методов, которые использовались в этой книге. Если вы столкнетесь с задачей вдвое большей, чем эта, или в 10 раз большей, то это может испытать ваше терпение, но не помешает вам ее решить.

Изучение новых навыков программирования

Есть еще одна тема, которую следует обсудить. Осваивая описанные в этой книге методы решения задач, вы делаете ключевой шаг на пути становления программистом. Однако, как и в случае с большинством профессий, это бесконечная дорога, потому что вы всегда должны стремиться к тому, чтобы профессионально расти. Как и во всем остальном, в программировании у вас должен быть план для изучения новых навыков и методов, не стоит рассчитывать на хаотичное приобретение новых знаний.

В этом разделе мы обсудим несколько областей, в которых вы, возможно, захотите приобрести новые навыки, а также некоторые систематические подходы для каждой из них. Все эти области объединяет необходимость применять полученные знания на практике. Именно поэтому каждая глава этой книги заканчивается упражнениями — и вы их выполняете, не так ли? Чтение ресурсов, посвященных новым идеям в области программирования — это важный первый шаг в их изучении, но это только первый шаг. Чтобы достичь точки, в которой вы можете уверенно использовать новую технику

для решения реальной задачи, сначала вы должны опробовать эту технику на меньшей, искусственной задаче. Помните, что один из наших основных методов решения задач состоит в разбиении сложной задачи, либо путем ее разделения на подзадачи, либо путем ее временного уменьшения, так что каждое состояние, с которым мы имеем дело, имеет только один нетривиальный элемент. Вам не следует пытаться решить нетривиальную задачу одновременно с освоением нового навыка, который будет центральным в вашем решении, потому что тогда ваше внимание будет разделено между двумя трудными задачами.

Новые языки

Я считаю, что C++ — это отличный язык программирования для создания производственного кода, и в первой главе я объяснил, почему я думаю, что это отличный язык для изучения. Тем не менее ни один язык программирования не является лучшим для всех ситуаций, поэтому хорошим программистам следует изучать несколько языков.

Выделите время на учебу

По возможности вам следует выделять время на изучение нового языка, прежде чем пытаться создавать производственный код на одном из них. Если вы попытаетесь решить нетривиальную задачу на языке, который вы никогда раньше не использовали, вы быстро нарушите важное правило решения задач, которое заключается в том, чтобы избегать разочарования. Поставьте себе цель изучить язык и достигните ее, прежде чем писать «реальные» программы на этом языке.

Конечно, в реальном мире мы иногда не полностью контролируем процесс назначения проектов. В любой момент кто-то может попросить нас написать программу на определенном языке, и этот запрос может сопровождаться крайним сроком, который помешает нам не спеша изучить язык перед решением настоящей задачи. Лучшая защита от возникновения такой ситуации заключается в том, чтобы начать изучение других языков программирования *до того*, как от вас потребуется владение ими. Исследуйте языки, которые вас интересуют или которые используются в тех областях, где вы планируете работать на протяжении своей карьеры. Это еще одна ситуация, когда деятельность, которая кажется тратой времени в краткосрочной перспективе, может принести большие дивиденды в долгосрочной. Даже если окажется, что в ближайшем будущем вам не понадобится язык, который вы изучили, освоение другого языка может улучшить ваши навыки работы с языками, которые вы уже знаете, поскольку это заставляет вас думать по-новому, избавляя от старых привычек и позволяя свежим взглядом посмотреть на свои навыки и методы работы. Считайте это эквивалентом перекрестного обучения в области программирования.

Начните с того, что вы знаете

Когда вы начинаете изучать новый язык программирования, вы по определению ничего о нем не знаете. Тем не менее, если это не первый ваш язык программирования, вы уже многое знаете о самом программировании. Поэтому хорошим первым шагом при изучении нового языка является понимание того, как код, который вы уже умеете писать на другом языке, может быть написан на новом языке.

Как говорилось ранее, вам следует учиться на практике, а не ограничиваться чтением. Возьмите программы, написанные вами на других языках, и перепишите их на новом языке. Систематически исследуйте отдельные элементы языка, такие как управляющие инструкции, классы, другие структуры данных и т.д. Цель состоит в том, чтобы перенести в новый язык как можно больше знаний, полученных вами ранее.

Изучите отличия

Следующий шаг заключается в изучении того, что отличает новый язык. Хотя два высокоуровневых языка программирования могут иметь большое сходство, *какие-то* отличия должны быть, иначе не было бы причин выбирать этот язык вместо любого другого. Опять же, учитесь на практике. Например, простое чтение о том, что инструкция множественного выбора языка допускает диапазоны (вместо отдельных значений инструкции `switch` языка C++) не так полезно для вашего профессионального становления, как фактическое написание кода, в котором осмысленно используется данное свойство.

Этот шаг, очевидно, важен для языков, которые заметно отличаются друг от друга, но в равной степени важен и для языков, имеющих общего предка, таких как C++, C# и Java, которые являются объектно ориентированными потомками языка C. Синтаксические сходства могут заставить вас ошибочно полагать, что вы знаете о новом языке больше, чем на самом деле. Рассмотрим следующий код.

```
integerListClass numberList;
numberList.addInteger(15);
```

Если бы эти строки были представлены вам как код, созданный на C++, вы бы поняли, что первая строка создает объект `numberList` класса `integerListClass`, а вторая строка вызывает метод `addInteger` на этом объекте. Если этот класс действительно существует и имеет метод под таким названием, который принимает параметр `int`, то этот код имеет смысл. Теперь предположим, я сказал вам, что этот код написан на языке Java, а не на C++. С точки зрения синтаксиса в этих двух строках нет ничего недопустимого. Однако в языке Java простое объявление переменной объекта класса фактически не создает объект, поскольку объектные переменные, по сути, являются ссылками, то есть они ведут себя аналогично указателям. Для выполнения эквивалентных действий на языке Java следует использовать такой код:

```
integerListClass numberList = new integerListClass;
numberList.addInteger(15);
```

Вероятно, вы быстро осознаете эту конкретную разницу между Java и C++, однако многие другие различия могут оказаться довольно тонкими. Если вы не выделите времени на их обнаружение, то они могут усложнить отладку при работе с новым языком. В процессе сканирования своего кода ваш внутренний интерпретатор языка программирования будет предоставлять вам некорректную информацию о том, что вы читаете.

Изучайте хорошо написанный код

В этой книге я несколько раз говорил о том, что вы не должны пытаться учиться программированию путем модификации чужого кода. Однако бывают моменты, когда изучение чужого кода жизненно важно. Хотя вы можете развить навыки работы с новым языком, написав серию оригинальных программ, чтобы стать мастером, вам нужно будет найти код, написанный программистом, хорошо владеющим этим языком.

Вы не будете «списывать» этот код; вы не будете использовать этот код для решения конкретной задачи. Вместо этого вы будете исследовать существующий код, чтобы обнаружить «лучшие практики» в этом языке. Посмотрите на код, написанный экспертом, и спросите себя не только о том, *что* делает программист, но и *почему* он это делает. Если этот код сопровождается пояснениями программиста, еще лучше. Делайте различия между решениями, принятыми с учетом стиля, и преимуществами с точки зрения производительности. Этот шаг позволит вам избежать распространенной ловушки. Слишком часто программисты изучают лишь жизненно необходимые основы нового языка, в результате чего получается слабый код, который не использует всех функций этого языка. Например, если бы вам как программисту C++ было необходимо написать код на языке Java, вы не довольствовались бы написанием кода на упрощенном C++; вместо этого вы решили бы научиться писать настоящий код Java, как это делают программисты, работающие с этим языком.

Как и в случае со всем остальным, вам необходимо применять полученные навыки на практике. Возьмите исходный код и измените его так, чтобы он мог сделать что-то новое. Уберите код с глаз и попытайтесь воспроизвести его. Цель состоит в том, чтобы познакомиться с кодом достаточно хорошо для того, чтобы суметь ответить на вопросы другого программиста об этом коде.

Важно подчеркнуть, что этот шаг происходит после других. Прежде чем мы доберемся до стадии изучения чужого кода, написанного на новом языке, мы уже изучим синтаксис и грамматику нового языка и применим навыки решения задач, освоенные при работе с другим языком, к новому языку. Если мы попытаемся ускорить процесс, начав изучение нового языка с изучения длинных образцов

программ и модификации этих образцов, существует реальный риск того, что этим наши навыки и ограничатся.

Новые навыки для языка, который вы уже знаете

Достижение этапа, на котором вы можете сказать, что «знаете» язык, не означает, что вы знаете о нем все. Даже если вы освоили синтаксис языка, всегда будут существовать новые способы комбинирования существующих языковых особенностей для решения задач. Большая часть этих новых способов будет относиться к одному из компонентов, описанных в предыдущей главе, в которой мы обсудили процесс их освоения. Важным фактором является усилие. Научившись решать задачи определенным образом, легко успокоиться на том, что вы уже знаете, и перестать расти как программист. В этот момент вы становитесь похожим на бейсбольного питчера, который умеет совершать только прямую подачу. Некоторые питчеры построили успешную профессиональную карьеру на одной подаче, однако игрок, который хочет превратиться из заменяющего в нападающего, нуждается в большем.

Чтобы максимально раскрыть свой потенциал как программиста, вам нужно искать новые знания и новые методы и применять их на практике. Ищите препятствия и преодолевайте их. Изучайте работу экспертов-программистов, работающих с выбранными вами языками.

Помните, что необходимость — это мать изобретения. Ищите задачи, которые не могут быть удовлетворительно решены с помощью вашего нынешнего набора навыков. Иногда вы можете изменить уже решенные вами задачи для постановки новых. Например, вы написали программу, которая отлично работает с небольшим набором данных, но что произойдет, если этот набор разрастется до гигантских размеров? Или вы написали программу, которая хранит свои данные на локальном жестком диске, но вы хотите, чтобы данные хранились удаленно? Что, если вам требуется несколько исполнений программы, которые могут одновременно получать и обновлять данные, хранящиеся удаленно? Начиная с рабочей программы и добавляя новые функции, вы можете сосредоточиться только на новых аспектах программирования.

Новые библиотеки

Современные языки программирования неотделимы от своих основных библиотек. Например, в процессе изучения языка C++ вы неизбежно что-то узнаете о стандартных библиотеках шаблонов, а при изучении Java — о стандартных классах этого языка. Однако помимо библиотек, поставляемых вместе с языком, вам необходимо изучать сторонние библиотеки. Иногда они представляют собой общие каркасы приложений вроде .NET Framework компании Microsoft, которые могут использоваться с несколькими высокоуровневыми

языками. В других случаях библиотека создается специально для определенной области, например OpenGL для графики, или является частью программного пакета стороннего производителя.

Как и при изучении нового языка, вы не должны пытаться изучать новую библиотеку во время работы над проектом, требующим ее применения. Вместо этого изучите основные компоненты библиотеки отдельно в ходе тестового проекта с нулевой важностью, прежде чем использовать их в реальном проекте. Поставьте перед собой ряд задач с увеличивающейся сложностью. Помните, что цель заключается не в решении любой из них, а только в том, чтобы научиться на этом процессе, поэтому вам не обязательно шлифовать решения или даже завершать их после того, как вы успешно применили изучаемую вами часть библиотеки. В ходе дальнейшей работы эти программы могут служить в качестве руководства. Когда вы зайдете в тупик из-за того, что не можете вспомнить, как, скажем, наложить 2D-изображение на 3D-сцену в OpenGL, нет лучшего решения, чем открыть старую программу, созданную специально для демонстрации этого метода и написанную в вашем собственном стиле, поскольку ее создавали вы сами.

Кроме того, как и при изучении нового языка, как только вы освоитесь с библиотекой, вам следует изучить код, написанный экспертами в области использования этой библиотеки. Большинство крупных библиотек имеют свои особенности, которые не раскрываются в официальной документации и узнать о которых при отсутствии длительного опыта работы с ними можно только от других программистов. По правде говоря, для того, чтобы добиться больших успехов в работе с некоторыми библиотеками, требуется первоначальное руководство другого программиста. Важно не слишком сильно полагаться на созданный другими код и быстро достичь стадии, на которой вы способны воссоздать тот код, который был вам показан. Вы удивитесь тому, как много вы можете узнать в процессе воссоздания созданного кем-то кода. Вы можете заметить в исходном коде вызов функции библиотеки и понять, что аргументы, переданные в этом вызове, приводят к определенному результату. Однако, когда вы откладываете этот код в сторону и пытаетесь воспроизвести данный эффект самостоятельно, вам придется исследовать документацию функции, все конкретные значения, которые могут принимать аргументы, а также то, почему они должны быть именно тем, чем они являются, чтобы получился желаемый результат.

Выберите курс

Как преподаватель я считаю, что мне следует завершить этот раздел, сказав о курсах. Независимо от области программирования, которую вы хотите изучать, вы найдете того, кто готов учить вас, будь то в традиционной классной комнате или в некоторой онлайн-среде. Тем не менее курс — это лишь катализатор для обучения, а не само

обучение, особенно в случае с программированием. Независимо от того, насколько знающим или воодушевленным является ваш инструктор, при фактическом изучении новых возможностей программирования вы оказываетесь сидящим перед своим компьютером, а не в лекционном зале. Как я уже неоднократно повторял в этой книге, вы должны воплощать идеи программирования на практике, и вы должны сделать их своими, чтобы по-настоящему их изучить.

Это не значит, что курсы не имеют ценности, потому что часто она огромна. Некоторые концепции программирования по своей сути сложны или запутаны, и если у вас есть доступ к преподавателю, обладающему талантом объяснять сложные концепции, это может сэкономить вам массу времени и нервов. Кроме того, курсы позволяют вам оценить свой прогресс в освоении новых знаний. Опять же, если вам повезет с инструктором, вы сможете многое узнать из оценки вашего кода, что упростит процесс обучения. Наконец, успешное завершение курса предоставляет нынешним или будущим работодателям некоторые доказательства того, что вы разбираетесь в предмете (если вам не повезло с инструктором, вы можете утешиться хотя бы этим).

Помните, что вы сами отвечаете за свое образование в области программирования, даже при прохождении курса. Курс предоставляет структуру для получения оценки и кредита в конце семестра, но эта структура не ограничивает вас в процессе обучения. Рассматривайте время изучения курса как прекрасную возможность узнать как можно больше о предмете помимо целей, перечисленных в учебном плане.

Заключение

Я с радостью вспоминаю свой первый опыт программирования. Я написал короткий текстовый симулятор пинбольной машины и должен сказать, что сейчас мне это тоже кажется бессмысленным, но это имело смысл в то время. Тогда у меня не было компьютера, а у кого он был в 1976 году? Однако в офисе моего отца был терминал телетайпа, который по сути представлял собой огромный матричный принтер с клавиатурой, соединенный с мэйнфреймом в местном университете через акустический модем. (Нужно было взять трубку, набрать номер, и когда вы слышали электронный зуммер, нужно было положить телефонную трубку на специальный рычаг, подключенный к терминалу.) Каким бы примитивным и бессмысленным не был мой пинбольный симулятор, в тот момент, когда программа заработала и компьютер начал действовать согласно моим инструкциям, меня зацепило.

Чувство, которое я испытал в тот день, — что компьютер подобен бесконечной куче кусочков конструкторов Lego, Erector Set и Lincoln Logs, позволяющих мне построить все, что я только мог себе представить, — это то, что питает мою любовь к программированию.

Когда моя среда разработки объявляет о чистой сборке, и мои пальцы касаются клавиши, которая запускает выполнение моей программы, я всегда волнуюсь, ожидая успеха или неудачи, и хочу увидеть результаты своих усилий, независимо от того, пишу ли я простой тестовый проект, добавляю окончательный штрих к крупному решению, создаю красивую графику или просто разрабатываю клиентскую часть приложения базы данных.

Надеюсь, что у вас возникают похожие чувства в процессе программирования. Даже если вы все еще испытываете сложности с некоторыми из описанных в этой книге областей, я надеюсь, вы понимаете, что, пока программирование волнует вас достаточно сильно, чтобы продолжать им заниматься, не существует проблем, которые вы не можете решить. Все, что требуется, — это желание приложить усилия и правильный подход к этому процессу. Обо всем остальном позаботится время.

Вы уже начали думать как программист? Если вы выполнили упражнения, приведенные в конце каждой главы, вы должны думать как программист и быть уверенным в своей способности решать задачи. Если вы выполнили мало упражнений, то у меня есть для вас предложение, и я готов поспорить, что вы можете догадаться, какое именно: выполните больше упражнений. Если вы пропустили некоторые из предыдущих глав, не начинайте с упражнений, приведенных в этой главе, — вернитесь туда, где вы остановились, и начните свой путь оттуда. Если вы не хотите выполнять больше упражнений, потому что вам не нравится программировать, тогда я не могу вам помочь.

Как только вы начнете думать как программист, можете гордиться своими навыками. Если кто-то назовет вас кодером, а не программистом, скажите, что хорошо обученную птицу можно научить печатать код, — вы же не просто пишете код, а используете код для решения задач. При прохождении собеседования с будущим работодателем или клиентом вы будете знать, что независимо от того, что потребует от вас эта работа, вы сможете с ней справиться.

Упражнения

Вы должны были знать, что вас ожидает последний набор упражнений. Они, конечно, более сложные и менее однозначные по сравнению с любым упражнением из предыдущих глав.

8.1. Напишите полную реализацию задачи с обманом в игре «Виселица», которая была бы лучше моей.

8.2. Расширьте свою программу с игрой «Виселица» так, чтобы пользователь мог выбрать роль Игрока 1. Пользователь по-прежнему будет выбирать количество букв в слове и количество промахов, однако в этот раз высказывать догадки должна программа.

8.3. Перепишите свою программу с игрой «Виселица» на другом языке, с которым вы в настоящее время мало знакомы или вообще не знакомы.

8.4. Сделайте свою игру «Виселица» графической, фактически отображающей рисунок виселицы и повешенного по мере ее составления. Вы пытаетесь думать как программист, а не как художник, поэтому не беспокойтесь о качестве искусства. Тем не менее вы должны сделать настоящую графическую программу. Не рисуйте виселицу с помощью ASCII-текста, — это слишком просто. Возможно, вам захочется исследовать библиотеки 2D-графики для C++ или выбрать другую, графически ориентированную платформу, например, Flash. Наличие графического изображения может потребовать ограничения количества неправильных догадок, однако вы можете найти способ предложить по крайней мере ряд вариантов для этого количества.

8.5. Создайте собственное упражнение: используйте навыки, которые вы приобрели, работая над игрой «Виселица», чтобы решить совершенно другую задачу, предполагающую манипулирование списком слов, например, создайте еще одну игру в слова, вроде «Скрабла», средство проверки орфографии или что-нибудь еще.

8.6. Создайте собственное упражнение: найдите задачу программирования на C++ такого размера или уровня сложности, которую вы не могли решить ранее, и решите ее.

8.7. Создайте собственное упражнение: найдите интересующую вас библиотеку или API, которые вы еще не использовали в программе. Затем исследуйте эту библиотеку или API и используйте ее в полезной программе. Если вас интересует общее программирование, рассмотрите библиотеку Microsoft .NET или библиотеку баз данных с открытым исходным кодом. Если вам нравится низкоуровневая графика, рассмотрите OpenGL или DirectX. Если вы хотите попробовать создавать игры, подумайте о таком движке с открытым исходным кодом, как Ogre. Подумайте о типах программ, которые вы хотите написать, найдите библиотеку, которая вам подходит, и приступайте.

8.8. Создайте собственное упражнение: напишите полезную программу для новой платформы (новой для вас), например для платформы мобильного или веб-программирования.

Предметный указатель

B

bad_alloc исключение, 114

C

C++
  free функция, 112
  malloc функция, 112
  this, 148
  typedef, 156
  typedef спецификатор, 116
  исключение, 159

D

delete оператор, 107
DirectX, 210

J

Java, 138
JDBC, 210

N

new оператор, 106
NULL указатель, 115

Q

qsort, функция, 81

T

this ключевое слово, 148
typedef, 156

А

Абстрактные типы данных, 144, 208
Алгоритм, 207

Б

Базовые техники решения задач, 32
Библиотеки, 209
Большая рекурсивная идея (БРИ), 174, 183, 185
Быстрое прототипирование, 239

В

Вектор, 77, 100
Висячая ссылка, 154
  из-за перекрестных связей, 165
Висячие указатели, 115, 126
Время существования переменной, 115
Вспомогательная функция, 123
Выделение памяти в классе, 154
  причины минимализации, 113
  пробуксовка, 113

Г

Глобальные переменные, 188
Глубокое копирование, 164
Головная рекурсия, 174, 177
Головной указатель, 129, 156–157, 166, 171

Д

Двоичные деревья, 192
  корневой узел, 192
  левое поддерево, 192
  поиска, 203, 231
  правое поддерево, 192
Двойной связный список, 161
Декодирование сообщения, 60
Деление на ноль, 135
Деструктор, 163
Динамическая память, 100
Динамические структуры данных, 108, 189
Динамический массив, 99
Диспетчер, 184
Допустимость данных, 121

Ж

Живучая программа, 122

З

Запись активации, 111, 114
И (Булева алгебра), 149, 151, 153, 158, 161–162

И

Избыточность данных, 152
Изучение языка, 259
  библиотеки, 263
  время, 260
  курс, 264
  навыки для уже изученного языка, 263
  отличия между языками, 261
  хорошо написанный код, 262
Инкапсуляция, 141, 142
Интерполяционного поиска, алгоритм, 230
Интерфейсы прикладного программирования (API), 210
  DirectX, 210
  JDBC, 210
Исключение, 159
Итератор, 217

К

Класс
  get и set, 148
  базовый фреймворк, 147
  выразительности, 146, 148, 150
  глубокое копирование, 164
  деструктор, 163
  динамические структуры данных, 154
  защищенный член, 139
  инкапсуляция, 141, 155
  интерфейс, 143
  композиция, 155
  конвенция наименований, 148
  конструктор, 139, 150
  метод, 139
  название метода, 145
  объявление класса, 139
  объект, 139
  однозадачник, 170
  поверхностное копирование, 164
  подкласс, 139
  приватный член, 139
  проверка, 153
  публичный член, 139
  служебный метод, 151
  сокрытие информации, 143
  спецификатор доступа, 139, 153
  фальшивый, 169–170
Классические головоломки, 18
  кварраси замок, 29
  лисица, гусь и кукуруза, 18
  скользящие плитки, 22
  судоку, 27
Клиент, 139
Кобаяси Мару, 17
Кодовый блок, 206
Композиция, 155
Компоненты
  выбор типа, 223
  изучение, 210
  обзор, 206
Конструктор, 139–140, 149–150, 156
  копирования, 167, 169
  по умолчанию, 140, 150
Контейнер последовательности, 217
Корневой узел, 192
Куча, 112
  переполнение, 114

Л

Левая сторона, 166
Левое поддерево, 192
Логическое И (&&)
  сокращенные вычисления, 158

М

Массив, 77
  динамический, 99
  динамическое выделение памяти, 117
  использование, 99
  многомерный, 95
  нескалярный, 93
  поиск, 79
  решение задач с помощью, 84
  сортировка, 81
  структура, 93
  фиксированных данных, 91
Мастер-план
  использование сильных и слабых сторон, 233
  разработка, 233
  составление, 240
Медиана, 90
Метод, 139
  get, 148
  set, 148
Многомерные массивы, 95

Н

Недостатки в дизайне, 234
планирование с учетом, 236
Недостатки кодирования, 233
планирование с учетом, 234
Нескалярные массивы, 93

О

Объект, 139
Одиночка, шаблон, 208
Оператор
->, 157
&& (логическое И), 158
-> (обращение к полям структуры), 128
& оператор (получение адреса), 110
присваивания (=), 166
* (разыменование), 107, 157, 167
Отслеживание состояния, 60
Ошибка заборного столба, 234

П

Перегрузка оператора, 166
Перекрестные ссылки, 126, 164
Планирование
с учетом недостатков дизайна, 236
с учетом недостатков кодирования, 234
с учетом сильных сторон, 238
Поверхностное копирование, 164
Повторное использование кода, 204
хорошее и плохое, 205
Поиск аналогий, 118
Поиск по критерию, 80
Поле класса, 140, 148, 149
Последовательный поиск, 79
Правая сторона, 166
Правое поддерево, 192
Представление неэффективное использование памяти, 110
Программное обеспечение для управления версиями, 258

Р

Разделение задачи
с использованием класса, 142
Разработка через тестирование, 238
Рекурсия, 173
базовый случай, 174
головная, 174, 177
двоичные деревья, 192
динамические структуры данных, 189
когда использовать, 198
косвенная, 174
недостатки, 198
основы, 174
ошибки, 186
прямая, 174
связные списки, 190
хвостовая, 174, 176
Рефакторинг, 88
Решение с помощью типичного примера, 117, 113

С

Свойство (C#), 148
Связные списки, 127, 190
добавление узлов, 130
обход, 132
последовательный доступ, 129
построение, 127
пустой, проверка, 134
узел, 127
головной указатель, 156
двойной связный список, 161
добавление узла, 157
обход, 159
удаление узла, 159
узел, 156
Символ
* (объявление указателя), 106
& (ссылочный параметр), 108
Скалярные переменные, 76
Служебный метод, 151
Сокращенные вычисления, 158
Сокрытие информации, 143
Сортировка, 81
вставками, 81, 82
проверка, 122, 157, 162
Специальный случай, 121
Спецификатор доступа, 139, 153
Ссылочный параметр, 108
Статистические показатели, 84
Статическая переменная, 189
Стек, 110, 114, 199

вызовов, 111
Стратегия
шаблон, 211
строка, 116
представление в виде массива, 116
Строки завершающий байт, 118
копирование, 123
Структура, 93
данных, размер которых определяется во время выполнения программы, 107

Т

Таблица символов, 231
Тестирование
тестовый пример, 118
Точка восстановления, 258

У

Узел
связный список, 156
Узоры
половина квадрата, 43
равнобедренный треугольник, 46
Указатели схемы, 121
Указатели, 81
когда применять, 109
на функции, 212
объявление, 106
приемущества использования, 107
Утечка памяти, 115
предотвращение, 120

Ф

Факториал, 174
Фиктивная запись, 158
Формула Луна, 50
Фрагментация памяти, 112
Функции
множнесколько точек выхода, 161
название, 145
Функция-обертка, 187, 195, 197

Х

Хвостовая рекурсия, 174, 176
Хеш-таблица, 231

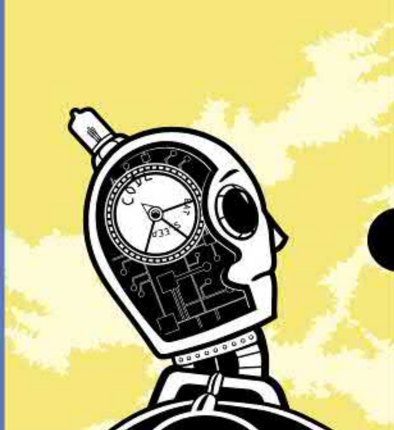
Ч

Читабельность, 145

Ш

Шаблон проектирования, 207
одиночка, 208
стратегия, 211
шаблонных классов, 171

КРЕАТИВНЫЙ ПОДХОД К СОЗДАНИЮ КОДА



C++ ВЕРСИЯ

Программирование - это творчество!

Если вы хоть раз сталкивались с трудностями при создании кода – эта книга для вас! Ее автор говорит о том, что программистов, талантливых от природы, не так много, однако можно развить в себе этот талант, если приучить свой мозг решать разнообразные задачи и делать это креативно.

В этой книге вы найдете:

- Примеры задач и их решения
- Множество интересных упражнений
- Полезные выводы и рекомендации

В этой книге вы не найдете:

- Жестких инструкций
- Готовых шаблонов
- Скучных объяснений

Основной проблемой начинающих программистов является неумение разложить программу на небольшие, понятные им составные части, свести задачу к ранее решенной и обобщить написанный код. Эта книга нацелена на развитие именно таких навыков. В ней на простых примерах показано, как приступать к решению различных задач, и рассмотрены основные техники написания программ. Я рекомендую данную книгу начинающим программистам в дополнение к учебнику по C++.

Николай Белов,
Senior C++ developer, Teknavo

www.nostarch.com

ISBN 978-5-04-089838-1

БОМБОРА

Бомбора — это новое название Эксмо Non-fiction, лидера на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bomborabooks
www.bombora.ru