计 算 机 科 学 丛 书

Pearson

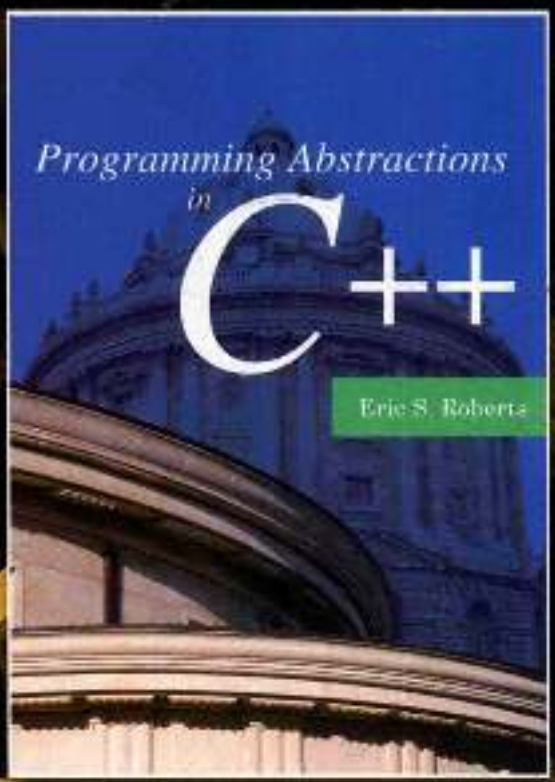
C++程序设计

基础、编程抽象与算法策略

[美] 埃里克 S. 罗伯茨（Eric S. Roberts） 著

李雁妮 译

Programming Abstractions in C++



机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

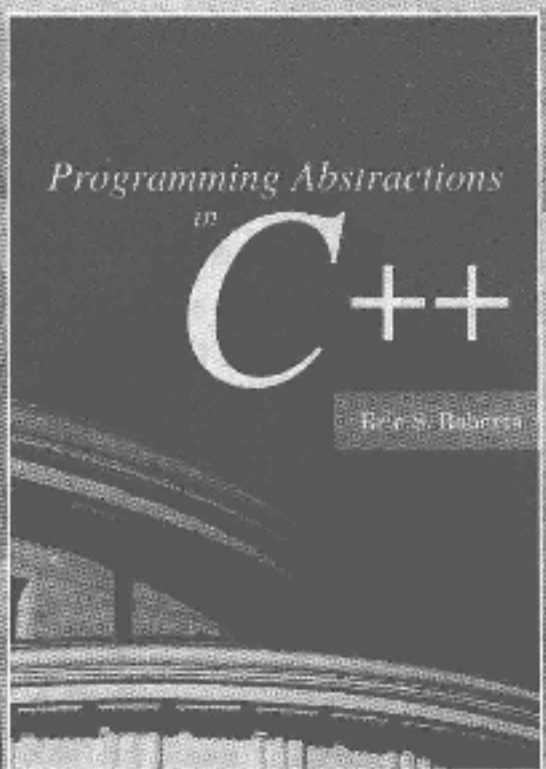
C++程序设计

基础、编程抽象与算法策略

[美] 埃里克 S. 罗伯茨（Eric S. Roberts）著

李雁妮 译

Programming Abstractions in C++



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

C++程序设计：基础、编程抽象与算法策略 /（美）埃里克 S. 罗伯茨（Eric S. Roberts）著；李雅娟译．—北京：机械工业出版社，2016.5
（计算机科学丛书）
书名原文：Programming Abstractions in C++

ISBN 978-7-111-54696-2

I. C… II. ①埃… ②李… III. C语言—程序设计 IV. TP312.8

中国版本图书馆CIP数据核字（2016）第225390号

本书版权登记号：图字：01-2013-8306

Authorized translation from the English language edition, entitled *Programming Abstractions in C++*, 9780133454840 by Eric S. Roberts, published by Pearson Education, Inc., Copyright © 2014.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2016.

本书中文简体字版由Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

本书是一本关于C++语言的优秀书籍，全书共计20章，主要介绍了C++的基本知识、函数和库、字符串、流、集合、类的设计、递归、递归策略、回溯算法、算法分析、指针与数组、动态内存管理、效率与表示、线性结构、映射、树、图、散列、迭代的策略等内容。本书重点突出，全面讲解了C++语言的基本概念，深入剖析了具体的编程思路。同时，每章后面都有配套的习题，有助于读者进一步理解和掌握相应的概念。

本书适合作为计算机专业及相关专业学生的教材或教学参考书，也适合希望学习C++语言的初学者和中高级程序员使用。

出版发行：机械工业出版社（北京市西城区百万庄大街22号　邮政编码：100037）
责任编辑：姜　杰　　责任校对：董纪丽
印　刷：北京市荣盛彩色印刷有限公司　　版　次：2016年11月第1版第1次印刷
开　本：185mm×260mm　1/16　　印　张：41
书　号：ISBN 978-7-111-54696-2　　定　价：120.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光／邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅奠划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brian W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037



华章科技图书出版中心

译者序

Programming Abstractions in C++

本书是美国斯坦福大学计算机科学系 C++ 编程课程多年来成功使用的优秀教材，我很荣幸能成为本书的译者。虽然在为时一年多的书稿翻译过程中，我倍感工作量的巨大与任务的艰辛，但却被本书严谨的结构、通俗精妙的语言以及丰富精巧的编程实例与习题等深深吸引，它驱动我尽最大努力翻译好这本优秀的 C++ 编程教材，以此呈现给国内教授 C++ 编程课程的高校教师和广大欲深入学习 C++ 编程的大学生、研究生及专业的 C++ 程序员。

本书突破了一般 C++ 编程教材大多仅注重介绍 C++ 语法特性的局限，以循序渐进的方式教授读者正确编写出可行高效的 C++ 程序。本书内容不仅涵盖了 2013 版 ACM 和 IEEE 所规定的计算机科学学科在程序设计课程中所定义的内容，而且为了缩短或消除“C++ 语言”和“C++ 编程抽象”之间的鸿沟，在其示例和习题中还包含了基本的数据结构及算法课程中的相关内容。从易于读者掌握并尽快提高 C++ 面向对象编程能力的角度，本书从第 3 章起便开始陆续介绍 C++ 标准类库中的类，并以示例程序和若干编程模式示范总结了如何使用 C++ 标准类库中的集合类、递归编程、面向对象程序设计和算法实现及分析等技术，同时提供了一个开源的、方便易用的图形化的可移植 C++ 类库——Stanford C++ 类库。衷心祝愿广大读者能从这本优秀的 C++ 编程教材中受益！

在本书翻译工作即将结束之时，衷心感谢机械工业出版社华章公司教材部宋劼女士，是她的努力才促成译者与华章公司在本书翻译上的合作。衷心感谢责任编辑缪杰对提高本书质量所做的大量细致的校对、修正等工作。

这里，要特别感谢王雅馨、黄一涵、刘磊、汪泰利、曹楠彦等同学在本书译校过程中的辛勤付出。由于时间仓促且译者水平有限，译文中难免存在欠妥、纰漏与错误之处，恳请广大读者不吝赐教与指正。

李磊妮

2016 年 7 月于西安电子科技大学

致学生

在过去的十年里，计算领域正令人振奋地高速发展着。我们随身携带的网络设备运行速度越来越快，价格越来越便宜，功能也越来越强大。谷歌和维基百科等基于网络的服务给我们提供了大量触手可及的信息。社交网络把我们同世界各地的人联系起来。流媒体技术和更快速的硬件让我们能在任何时候下载所需的音乐和影像。

然而，这些技术并不是突然而至的，而是人们创造了它们。遗憾的是，具备必需的软件开发技能的人现在正供不应求。在硅谷的高科技中心，很多公司找不到能把技术设想转化为现实应用的工程师。各个公司正在极力招聘懂得开发及维护大型系统的人，即懂得数据表示、效率、安全性、正确性和模块化等问题的软件开发人员。

尽管本书不会教给你关于这些主题和计算机科学领域的所有知识，但它会给你一个良好的开始。在斯坦福大学，每年有超过1000名学生选择使用本教材上课。他们中的大部分人觉得在暑期实习或实际工作中仅仅学习本教材中的知识远远不够。更多的学生选择继续学习更深入的课程以使自己在这个高速发展的领域获得更多的机会。

本书的主题除了会在计算机行业中给你提供机会外，同时它也寄乐于学。你在本书中学到的算法和策略有一部分是最近十年发明的，其他的都存在了超过2000年——它们充分体现了人类的聪明才智和创造力。这些算法和策略还非常实用，它们会帮助你成为一个富有经验的程序员。

在你学习本书中的材料时，请牢记，编程总是需要通过实际操作来学习的。阅读一种算法技术并不代表你就能够把那个算法应用到实际中去。只有通过练习和尝试去解决问题的调试，你才能真正学到算法的精髓。编程有时候使人感觉很沮丧，但是当你找到最后一个错误并且看到你的程序正确运行时，会欣喜若狂，它足以回报你在编程这条道路上所付出的任何努力。

致教师

本教材适合作为典型的大学课程中第二门编程课程的教材。它涵盖了ACM的Curriculum'78报告中定义的传统CS2课程中的材料。因此它包含了CS102和CS103课程指定的绝大多数主题。CS102和CS103分别由"ACM/IEEE-CS联合计算机课程2001版"报告及"计算机科学课程2013版"草稿中的AL/基本数据结构及算法单元中的材料定义。

本教材采用的教学策略在斯坦福大学已大获成功。

1. 数据结构的客户优先方法。传统的CS2课程由一系列基本数据结构组成。采用此模型，学生可同时学习如何使用一个特定的结构和如何实现它及理解它的性能特点。相比之下，本教材很早地展现了类的完整集合，让学生以客户的身份逐渐熟悉这些类。一旦学生透彻理解了这些内容，本书即开始展现它可能的实现机制和相关的计算特性。在斯坦福大学采用这种策略有助于学生轻松理解相关内容。自从做了这个改变，学生在需要使用集合类的考试中的分数也有了大幅度提高。

2. 稍晚呈现那些需要详细了解底层机器的 C++ 特性。尽管前两章给学生提供了 C++ 中基本类型和控制结构的总览，但初始的部分刻意地区分了基本指针和数组等依赖于对底层机器架构理解的主题。虽然这些细节是 CS2 的基本部分，但也没有必要在课程刚开始的时候就给学生过大的负担。尽早介绍类的集合使得学生能够掌握几个其他同等重要的主题，包括集合类、递归、面向对象设计和算法分析，但是不需要同时纠结于它的底层细节。

3. 一个方便易用的图形化可移植类库。使用 C++ 作为教学语言的一个问题是标准类库不提供图形化功能。而本书自带了一个免费发布的开源类库——Standford C++ 类库，它提供了一种进行图形交互的简单且宜教宜学的方法。Standford C++ 类库还包括集合类的简化实现，它支持一个更为简化且更加有效的表示规则。

补充资源[⊖]

对于学生

在 Pearson 网站（http://www.pearsonhighered.com/ericroberts/）上，读者可下载以下资源：

1. 书中每个示例程序的源代码文件
2. 运行示例的全彩 PDF 版本
3. 复习题的答案

对于教师

在 Pearson 网站上，有资格的教师可下载以下资源：

1. 书中每个示例程序的源代码文件
2. 运行示例的全彩 PDF 版本
3. 复习题的答案
4. 编程习题的答案
5. 每章的 PowerPoint 课件

Stanford C++ 类库

Stanford C++ 类库作为开源的开发项目可以免费获得。头文件、编译库和源代码可以通过 GitHub (http://www.github.com/eric-roberts/StanfordCPPLib) 或从作者的个人网站（http://cs.stanford.com/~eroberts/StanfordCPPLib）获得。

致谢

本教材有着有趣的发展历史，它在某些方面也反映了 C++ 语言自身的进化。就像 Bjarne Stroustrup 的第 1 版 C++ 是在 C 语言的基础上实现的，本书产生于我的另一本基于 C 语言的书——《C 程序设计的抽象思维》[⊖]，它由 Pearson 下属的 AddisonWesley 于 1998 年出版。十年前，我的斯坦福同事 Julie Zelenski 用 C++ 语言更新了它，在那一年我们开始在一系列的概述课程中使用它。尽管修订的教材版本在开始时效果很好，但这些年来我们演变的系列概述课

⊖ 关于本书教辅资源，凡书教师可向培生教育出版集团北京代表处申请，电话：010-57355169/57355171，电子邮件：service.cn@pearson.com。——编辑注

⊖ 此书中文版已由机械工业出版社出版，书号 978-7-111-38074-0。——编辑注

得表明它需要一个重新编写的教材版本，而这本书就是最终的产品。

我要感谢过去这些年在斯坦福的同事，首先要感谢 Julie Zelenski 在初始 C++ 版本上的卓越贡献。我的同事 Keith Schwarz、Jerry Cain、Stephen Cooper 和 Mehran Sahami 都对修订版做出了重要贡献。我还要向几届课程负责人和这些年我的学生表示感谢，他们都让本课程的教学变得更有激情。

另外，我还要向 Marcia Horton、Tracy Johnson 和 Pearson 团队的其他成员表达衷心的感谢，因为他们都对本书和它的旧版本的修订提供了大力支持。

最后，最衷心地感谢我的妻子 Lauren Rusk，她又一次作为我的开发编辑发挥了她的魔力。Lauren 梳理和润色了本书的文字。没有她，这本书不会成为现在这个样子。

埃里克 S. 罗伯茨
斯坦福大学

目 录

Programming Abstractions in C++

出版者的话
专家序
前言

第 1 章 C++ 概述 …… 1

1.1 你的第一个 C++ 程序 …… 1
1.2 C++ 的历史 …… 2
1.2.1 面向对象范型 …… 2
1.2.2 C++ 的演化 …… 3
1.3 编译过程 …… 3
1.4 C++ 程序结构 …… 4
1.4.1 注释 …… 5
1.4.2 包含的库文件 …… 6
1.4.3 函数原型 …… 6
1.4.4 主程序 …… 7
1.4.5 函数定义 …… 8
1.5 变量 …… 9
1.5.1 变量声明 …… 9
1.5.2 命名规则 …… 10
1.5.3 局部变量和全局变量 …… 11
1.5.4 常量 …… 11
1.6 数据类型 …… 12
1.6.1 数据类型的概念 …… 12
1.6.2 整数类型 …… 13
1.6.3 浮点类型 …… 13
1.6.4 布尔类型 …… 14
1.6.5 字符 …… 14
1.6.6 字符串 …… 15
1.6.7 枚举类型 …… 16
1.6.8 复合类型 …… 17
1.7 表达式 …… 17
1.7.1 优先级和结合律 …… 18
1.7.2 表达式中的混合类型 …… 19
1.7.3 整数除法和求余操作符 …… 19
1.7.4 类型转换 …… 20
1.7.5 赋值操作符 …… 20
1.7.6 自增和自减操作符 …… 21
1.7.7 布尔运算 …… 22
1.8 语句 …… 24
1.8.1 简单语句 …… 24
1.8.2 块 …… 24
1.8.3 if 语句 …… 24
1.8.4 switch 语句 …… 25
1.8.5 while 语句 …… 27
1.8.6 for 语句 …… 29
本章小结 …… 31
复习题 …… 32
习题 …… 33

第 2 章 函数与库 …… 37

2.1 函数概念 …… 37
2.1.1 数学中的函数 …… 37
2.1.2 编程中的函数 …… 37
2.1.3 使用函数的优点 …… 38
2.1.4 函数和算法 …… 38
2.2 库 …… 39
2.3 在 C++ 中定义函数 …… 41
2.3.1 函数原型 …… 41
2.3.2 重载 …… 42
2.3.3 默认形参数 …… 42
2.4 函数调用机制 …… 43
2.4.1 函数调用步骤 …… 43
2.4.2 组合函数 …… 44
2.4.3 追踪组合函数执行过程 …… 46
2.5 引用参数 …… 49
2.6 接口与实现 …… 52
2.6.1 定义 error 库 …… 53
2.6.2 导出数据类型 …… 54
2.6.3 导出常量定义 …… 56
2.7 接口设计原则 …… 58
2.7.1 统一主题的重要性 …… 58
2.7.2 简单性与信息隐藏原理 …… 59

2.7.3 满足用户需求 …… 60
2.7.4 通用工具的优势 …… 60
2.7.5 库稳定性的价值 …… 60
2.8 随机数库的设计 …… 61
2.8.1 随机数与伪随机数 …… 61
2.8.2 标准库中的伪随机数 …… 62
2.8.3 选择正确的函数集 …… 63
2.8.4 构建用户程序 …… 65
2.8.5 随机数库的实现 …… 65
2.8.6 初始化随机数种子 …… 69
2.9 Stanford 类库介绍 …… 73
2.9.1 简单的输入和输出类库 …… 73
2.9.2 Stanford 类库中的图形处理程序 …… 74
本章小结 …… 77
复习题 …… 78
习题 …… 79

第 3 章 字符串类 string …… 85
3.1 使用字符串作为抽象数据 …… 85
3.2 字符串操作 …… 87
3.2.1 操作符重载 …… 88
3.2.2 从一个字符串中选取字符 …… 89
3.2.3 字符串赋值 …… 90
3.2.4 提取字符串中的子串 …… 90
3.2.5 在一个字符串中进行搜索 …… 90
3.2.6 循环遍历字符串中的所有字符 …… 91
3.2.7 通过连接扩展字符串 …… 92
3.3 <cctype> 库 …… 93
3.4 修改字符串中的内容 …… 94
3.5 遗留的 C 风格字符串 …… 95
3.6 编写字符串应用程序 …… 95
3.6.1 回文识别 …… 96
3.6.2 将英语翻译成儿童黑话 …… 96
3.7 strlib.h 库 …… 99
本章小结 …… 100
复习题 …… 100
习题 …… 101

第 4 章 流类 …… 108
4.1 格式化输出 …… 108
4.2 格式化输入 …… 112
4.3 数据文件 …… 113
4.3.1 使用文件流 …… 114
4.3.2 单个字符的输入/输出 …… 115
4.3.3 面向行的输入/输出 …… 118
4.3.4 格式化输入/输出 …… 119
4.3.5 字符串流 …… 121
4.3.6 一个用于控制台输入的更鲁棒的策略 …… 122
4.4 类层次 …… 123
4.4.1 生物层次 …… 123
4.4.2 流类层次 …… 124
4.4.3 在流层次中选择正确的层次 …… 126
4.5 simpio.h 和 filelib.h 库 …… 127
本章小结 …… 128
复习题 …… 128
习题 …… 129

第 5 章 集合类 …… 133
5.1 Vector 类 …… 134
5.1.1 指定 Vector 的基类型 …… 134
5.1.2 声明 Vector 对象 …… 135
5.1.3 Vector 的操作 …… 135
5.1.4 从 Vector 对象中选择元素 …… 136
5.1.5 作为参数传递 Vector 对象 …… 137
5.1.6 创建预先定义大小的 Vector …… 138
5.1.7 Vector 类的构造函数 …… 141
5.1.8 Vector 中的操作符 …… 142
5.1.9 表示二维结构 …… 143
5.1.10 Stanford 类库中的 Grid 类 …… 143
5.2 Stack 类 …… 144
5.2.1 Stack 类结构 …… 145
5.2.2 栈和小型计算器 …… 145
5.3 Queue 类 …… 148
5.3.1 仿真和模型 …… 149
5.3.2 排队模型 …… 149
5.3.3 离散时间 …… 150
5.3.4 仿真时间中的事件 …… 150

5.3.5 实现仿真 ……151
5.4 Map 类 ……154
5.4.1 Map 类的结构 ……154
5.4.2 在一个应用中使用 Map 类 ……156
5.4.3 Map 类作为关联数组 ……157
5.5 Set 类 ……158
5.5.1 实现 <cctype> 库 ……159
5.5.2 创建单词列表 ……160
5.5.3 Stanford 类库中的 Lexicon 类 ……161
5.6 在集合上进行迭代 ……162
5.6.1 迭代顺序 ……163
5.6.2 再论几童黑话 ……164
5.6.3 计算单词的频率 ……165
本章小结 ……167
复习题 ……168
习题 ……168

第 6 章 类的设计 ……178

6.1 二维点的表示 ……178
6.1.1 将 Point 定义为结构类型 ……178
6.1.2 将 Point 定义为类 ……179
6.1.3 接口与实现的分离 ……182
6.2 操作符重载 ……184
6.2.1 重载插入操作符 ……184
6.2.2 判断两个点是否相等 ……186
6.2.3 为 Direction 类型增加操作符 ……189
6.3 有理数 ……191
6.3.1 定义新类的机制 ……192
6.3.2 采用用户的观点 ……193
6.3.3 确定 Rational 类的私有实例变量 ……193
6.3.4 为 Rational 类定义构造函数 ……193
6.3.5 为 Rational 类定义方法 ……194
6.3.6 实现 Rational 类 ……196
6.4 token 扫描器类的设计 ……198
6.4.1 用户想从记号扫描器中得到什么 ……199
6.4.2 tokenScanner.h 接口 ……200
6.4.3 实现 TokenScanner 类 ……202
6.5 将程序封装成类 ……205
本章小结 ……207
复习题 ……207
习题 ……208

第 7 章 递归简介 ……215

7.1 一个简单的递归例子 ……215
7.2 阶乘函数 ……217
7.2.1 Fact 的递归公式 ……217
7.2.2 跟踪递归过程 ……218
7.2.3 递归的稳步跳跃 ……221
7.3 斐波那契函数 ……222
7.3.1 计算斐波那契数列项 ……222
7.3.2 在递归实现中获得自信 ……223
7.3.3 递归实现的效率 ……224
7.3.4 递归不应被指责 ……225
7.4 检测回文 ……226
7.5 二分查找算法 ……228
7.6 间接递归 ……229
7.7 递归地思考 ……230
7.7.1 保持全局的观点 ……230
7.7.2 避免常见的错误 ……231
本章小结 ……232
复习题 ……233
习题 ……234

第 8 章 递归策略 ……237

8.1 汉诺塔 ……237
8.1.1 问题框架 ……238
8.1.2 找到一种递归策略 ……238
8.1.3 验证这个策略 ……240
8.1.4 编码方案 ……241
8.1.5 跟踪递归过程 ……242
8.2 子集求和问题 ……245
8.2.1 寻找一个递归解决方案 ……246
8.2.2 包含/排除模式 ……246
8.3 字符排列 ……247
8.4 图的递归 ……249
8.4.1 一个来自计算机艺术的实例 ……250

8.4.2 分形 ……252
本章小结 ……254
复习题 ……255
习题 ……255

第 9 章 回溯算法 ……264

9.1 迷宫的递归回溯 ……264
9.1.1 右手法则 ……264
9.1.2 寻找一种递归方法 ……265
9.1.3 确定简单情况 ……266
9.1.4 对迷宫问题的解决算法进行编码 ……267
9.1.5 说服你自己那个方案可行 ……269
9.2 回溯与游戏 ……271
9.2.1 拿子游戏 ……272
9.2.2 一个通用的双人游戏程序 ……276
9.3 最小最大算法 ……277
9.3.1 游戏树 ……278
9.3.2 评价位置和走法 ……278
9.3.3 限制递归搜索的深度 ……280
9.3.4 实现最小最大算法 ……280
本章小结 ……282
复习题 ……282
习题 ……283

第 10 章 算法分析 ……291

10.1 排序问题 ……291
10.1.1 选择排序算法 ……291
10.1.2 性能的经验评估 ……292
10.1.3 分析选择排序算法的性能 ……293
10.2 时间复杂度 ……294
10.2.1 大 O 符号 ……295
10.2.2 大 O 的标准简化 ……295
10.2.3 选择排序算法的时间复杂度 ……295
10.2.4 从代码中降低时间复杂度 ……296
10.2.5 最坏情况以及平均情况下的复杂度 ……297
10.2.6 大 O 符号的正式定义 ……298
10.3 递归的终止 ……300
10.3.1 分治策略的作用 ……300
10.3.2 合并两个矢量 ……301
10.3.3 归并排序算法 ……301
10.3.4 归并排序的计算复杂度 ……302
10.3.5 比较 N^2 与 $N\log_2 N$ 的性能 ……304
10.4 标准的算法复杂度类别 ……304
10.5 快速排序算法 ……306
10.5.1 划分矢量 ……308
10.5.2 快速排序算法的性能分析 ……310
10.6 数学归纳法 ……311
本章小结 ……313
复习题 ……314
习题 ……315

第 11 章 指针和数组 ……320

11.1 内存结构 ……320
11.1.1 位、字节和字 ……320
11.1.2 二进制和十六进制表示 ……321
11.1.3 表示其他数据类型 ……322
11.1.4 内存地址 ……323
11.1.5 为变量分配内存 ……325
11.2 指针 ……326
11.2.1 把地址当作数值 ……327
11.2.2 声明指针变量 ……327
11.2.3 基本的指针运算 ……328
11.2.4 指向结构和对象的指针 ……330
11.2.5 关键字 this ……331
11.2.6 特殊的指针 NULL ……331
11.2.7 指针和引用调用 ……332
11.3 数组 ……334
11.3.1 声明数组 ……334
11.3.2 数组元素的选择 ……335
11.3.3 数组的静态初始化 ……335
11.3.4 有效容量和分配容量 ……336
11.3.5 指针和数组的关系 ……336
11.4 指针运算 ……338
11.4.1 数组索引和内存地址 ……338
11.4.2 指针的自增自减运算 ……339
11.4.3 C 风格字符串 ……339
11.4.4 指针运算和程序风格 ……341
本章小结 ……341

复习题……342
习题……344

第 12 章　动态内存管理……347

12.1　动态分配和堆……347
12.1.1　new 操作符……348
12.1.2　动态数组……348
12.1.3　动态对象……349
12.2　链表……349
12.2.1　刚铎灯塔……349
12.2.2　链表内的迭代……352
12.2.3　迭代和列表……352
12.3　释放内存……352
12.3.1　delete 操作符……352
12.3.2　释放内存策略……353
12.3.3　析构函数……354
12.4　定义 CharStack 类……354
12.4.1　charstack.h 接口……355
12.4.2　选择字符栈的表示……357
12.5　堆－线图……361
12.6　单元测试……366
12.7　拷贝对象……368
12.7.1　深拷贝和浅拷贝……368
12.7.2　赋值和拷贝构造函数……369
12.8　关键字 const 的使用……371
12.8.1　常量定义……371
12.8.2　常量引用调用……372
12.8.3　const 方法……372
12.9　CharStack 类的效率……376
本章小结……377
复习题……378
习题……379

第 13 章　效率和表示……383

13.1　编辑文本的软件模式……383
13.2　设计简单的文本编辑器……384
13.2.1　编辑器命令……384
13.2.2　EditorBuffer 类的公有接口……385
13.2.3　选择一种底层表示……387
13.2.4　编写编辑器应用代码……388
13.3　基于数组的类实现……389
13.3.1　定义私有数据结构……390
13.3.2　缓冲区操作的实现……390
13.3.3　基于数组的编辑器的时间复杂度……393
13.4　基于栈的类实现……394
13.4.1　定义私有数据结构……394
13.4.2　缓冲区操作的实现……395
13.4.3　时间复杂度的比较……397
13.5　基于列表的类实现……397
13.5.1　链表缓冲区的插入操作……400
13.5.2　链表缓冲区的删除操作……402
13.5.3　链表表示法中光标的移动……403
13.5.4　缓冲区实现的完善……405
13.5.5　基于链表的缓冲区的时间复杂度……407
13.5.6　双向链表……408
13.5.7　时空权衡……408
本章小结……409
复习题……409
习题……410

第 14 章　线性结构……414

14.1　模板……414
14.2　栈的实现……416
14.2.1　使用动态数组实现栈……416
14.2.2　使用链表实现栈……420
14.3　队列的实现……426
14.3.1　基于数组的队列实现……427
14.3.2　队列的链表表示……433
14.4　实现矢量类……437
14.5　集成原型和代码……442
本章小结……442
复习题……443
习题……444

第 15 章　映射……446

15.1　使用矢量实现映射……447
15.2　查找表……449

15.3 哈希 ····· 451
15.3.1 设计数据结构 ····· 451
15.3.2 为字符串定义哈希函数 ····· 452
15.3.3 实现哈希表 ····· 454
15.3.4 线性哈希表的实现 ····· 456
15.3.5 调整桶的数目 ····· 457
15.4 实现 HashMap 类 ····· 458
本章小结 ····· 459
复习题 ····· 460
习题 ····· 461

第 16 章 树 ····· 463

16.1 家谱 ····· 463
16.1.1 用来描述树的术语 ····· 463
16.1.2 树的递归特性 ····· 464
16.1.3 用 C++ 语言表示家谱 ····· 464
16.2 二叉搜索树 ····· 465
16.2.1 使用二叉搜索树的动机 ····· 466
16.2.2 在二叉搜索树中寻找节点 ····· 467
16.2.3 在二叉搜索树中插入一个新节点 ····· 468
16.2.4 删除节点 ····· 471
16.2.5 树的遍历 ····· 472
16.3 平衡树 ····· 473
16.3.1 平衡树策略 ····· 474
16.3.2 可视化 AVL 算法 ····· 475
16.3.3 单旋转 ····· 476
16.3.4 双旋转 ····· 478
16.3.5 实现 AVL 算法 ····· 478
16.4 使用 BST 实现映射 ····· 482
16.5 偏序数 ····· 482
本章小结 ····· 485
复习题 ····· 485
习题 ····· 487

第 17 章 集合 ····· 494

17.1 集合作为一种数学抽象 ····· 494
17.1.1 隶属关系 ····· 494
17.1.2 集合运算 ····· 495
17.1.3 集合恒等式 ····· 496
17.2 集合接口的扩展 ····· 497
17.3 集合的实现策略 ····· 500
17.4 优化小整数的集合 ····· 504
17.4.1 特征向量 ····· 504
17.4.2 压缩的位数组 ····· 505
17.4.3 位操作符 ····· 506
17.4.4 实现特征向量 ····· 507
17.4.5 实现高级集合运算 ····· 509
17.4.6 模板特化 ····· 509
17.4.7 使用一种混合的实现 ····· 509
本章小结 ····· 510
复习题 ····· 511
习题 ····· 512

第 18 章 图 ····· 514

18.1 图的结构 ····· 514
18.1.1 有向图和无向图 ····· 515
18.1.2 路径和回路 ····· 516
18.1.3 连通性 ····· 516
18.2 表示策略 ····· 517
18.2.1 用邻接表表示连接关系 ····· 517
18.2.2 用邻接矩阵表示连接关系 ····· 518
18.2.3 用弧集合表示关系 ····· 519
18.3 一种低层的图抽象 ····· 520
18.4 图的遍历 ····· 524
18.4.1 深度优先搜索 ····· 524
18.4.2 广度优先搜索 ····· 527
18.5 定义图类 ····· 528
18.5.1 用类表示图、节点和弧 ····· 528
18.5.2 用参数化的类实现图 ····· 529
18.6 寻找最短路径 ····· 538
18.7 搜索网页的算法 ····· 542
18.7.1 谷歌的网页排名算法 ····· 542
18.7.2 网页排名算法的一个简例 ····· 543
本章小结 ····· 544
复习题 ····· 545

习题……546

第 19 章 继承……551

19.1 简单的继承……551
19.1.1 指定模板类中的类型……551
19.1.2 定义 Employee 类……552
19.1.3 C++ 中子类的局限性……554
19.2 图形对象的继承层次……556
19.2.1 调用父类的构造函数……561
19.2.2 将 GObject 类指针存储在矢量中……563
19.3 表达式的类层次……563
19.3.1 异常处理……565
19.3.2 表达式结构……566
19.3.3 表达式的递归定义……566
19.3.4 二义性……567
19.3.5 表达式树……568
19.3.6 exp.h 接口……570
19.3.7 Expression 子类的表示……573
19.3.8 表达式图解……573
19.3.9 方法的实现……574
19.4 解析表达式……577
19.4.1 语句解析和语法……577
19.4.2 考虑运算的优先级……578
19.4.3 递归下降语法分析器……579
19.5 多重继承……584
19.5.1 stream 类库中的多重继承……584
19.5.2 在 GObject 继承层次中添加 GFillable 类……585
19.5.3 多重继承的危险性……586
本章小结……586
复习题……587
习题……589

第 20 章 迭代策略……595

20.1 使用迭代器……595
20.1.1 简单的迭代器例子……595
20.1.2 迭代器的层次结构……597
20.2 使用函数作为数据值……598
20.2.1 函数指针……598
20.2.2 简单的画图应用……598
20.2.3 声明函数指针……600
20.2.4 实现 plot 函数……600
20.2.5 映射函数……601
20.2.6 实现映射函数……603
20.2.7 回调函数的限制……603
20.3 用函数封装数据……604
20.3.1 使用对象模拟闭包……604
20.3.2 函数对象的简单例子……605
20.3.3 向映射函数传递函数对象……606
20.3.4 编写以函数作为参数的函数……607
20.4 STL 算法库……607
20.5 C++ 的函数式编程……609
20.5.1 STL 库 <functional> 的接口……610
20.5.2 比较函数……612
20.6 迭代器的实现……612
20.6.1 为矢量类实现迭代器……612
20.6.2 将指针作为迭代器……616
20.6.3 typedef 关键字……617
20.6.4 为其他集合类实现迭代器……618
本章小结……618
复习题……619
习题……619

索引……624

C++ 概述

计划来自这些不同实践。这是我们的经验：计划不会从一个人或两个人，比如我们的头脑中产生，而是来自日常的工作中。

——斯托·卡迈克尔（Stokely Carmichael）和查尔斯汉·密尔顿（Charles V. Hamilton），《黑人权利》（*Black Power*），1967

在刘易斯·卡罗尔的《爱丽丝梦游仙境》（*Alice's Adventures in Wonderland*）中，国王向小白兔说道："从起点开始，然后继续一直到目的地，最后停止。"这是一个非常好的忠告，但仅限于你从起点处开始。本书设计为计算机科学的第二门课程教材，并且假设你已经开始了程序设计的学习。同时，由于第一门课程重点在于它所覆盖的内容，因此，对于一本教材的作者而言，很难依赖于你已掌握的任何材料。例如，你们中的一些人可能已经从其他密切相关的语言（如 C 或 Java 语言）的编程经验中理解了 C++ 的控制结构。然而，对另外一些人而言，C++ 的结构对他们来讲是不熟悉的。由于读者的知识背景不同，因此，最好的方法就是采用上述国王的忠告。因此，本章将从起点开始，并且向你介绍写一个简单的 C++ 程序所必需的那些部分。

1.1 你的第一个 C++ 程序

正如在下一节你将要学习到的一样，C++ 语言是极其成功的诞生于上世纪 70 年代初的 C 语言的一种扩展。在 C 语言的定义文档：《C 程序设计语言》，布莱恩·柯林汉（Brian Kernighan）和丹尼斯·里奇（Dennis Ritchie）在第 1 章开头便给出了以下建议：

> 学习一种新的程序设计语言的唯一途径就是开始编写程序。对于所有编程语言，写出的第一个程序都是一样的：
>
> 打印以下单词：
>
> ```
> hello,world
> ```

初学语言时，这是一个很大的障碍。要越过此障碍，首先必须创建程序文本，然后成功地对它进行编译，并加载、运行，最后再查看它所产生的输出结果。只有掌握了这些操作细节，剩下的就比较容易了。

如果你用 C++ 重写"Hello World"程序，它可能最终看起来像图 1-1 中的代码。

```
/*
 * File: HelloWorld.cpp
 * --------------------
 * This file is adapted from the example
 * on page 1 of Kernighan and Ritchie's
 * book The C Programming Language.
 */

#include <iostream>
using namespace std;

int main() {
   cout << "hello, world" << endl;
   return 0;
}
```

图 1-1 "Hello World"程序

此时，最重要的不是完全理解上述程序每一行是什么意思，因为之后会有足够的时间去掌握这些细节。你的任务（你应该决定是否接受它）是让 HelloWorld 程序运行起来。像图 1-1 中一样准确地输入程序，然后弄清楚你要怎样才能使它

工作。你需要使用的确切步骤取决于用来创建并运行C++程序的编译环境。如果用本书作
[2] 为课程的教科书，你的老师会为你提供所需的一些关于编译环境的参考资料。如果你正在阅读本书，无论你使用的是哪种C++编译环境，你都需要参考编译环境所附带的文档。

当你把所有这些细节解决后，你应该在电脑屏幕上的一个窗口中看到HelloWorld程序的输出。在我准备这本书的Apple Macintosh电脑上，输出窗口看起来如下图所示：



在你的电脑上，输出窗口可能会有些不同。除了程序的"hello, world"问候外，可能还会包含一些额外的状态消息。但"hello, world"消息一定会输出。Kernighan和Ritchie说道："其他一切都是相对容易的。"虽然这也许不是真的，但是你会取得一个重要的进展。

1.2 C++的历史

早期的计算机，程序采用**机器语言**（machine language）编写而成。机器语言是机器的基本指令，它可以直接被计算机执行。但用机器语言编写的程序难以阅读，主要是因为机器语言的
[3] 结构是针对机器硬件而不是针对程序员的需要设计的。更糟糕的是，每种计算机硬件都有它自己的机器语言，这就意味着在一台机器上编写的程序不能在其他类型的计算机硬件上运行。

20世纪50年代中期，在IBM的约翰·巴克斯的领导下的一组程序员有了深刻改变计算性质的想法。巴克斯和他的同事猜想：编写嵌有试图计算数学公式的程序并让计算机将这些公式自动转换成机器语言是否可行？1955年，该团队创造出了FORTRAN（formula translation名字的缩写）的最初版本，这是第一个**高级程序设计语言**（higher-level programming language）的范例。

从那以后，发明了许多新的程序设计语言，大多数都是基于已有的程序设计语言，并以演化的方式创造出新的语言。C++代表了其演化中连接的两个分支。其中一个先祖是C语言，C语言是在1972年由贝尔实验室的丹尼斯·里奇设计的，随后在1989年被**美国国家标准协会**（ANSI）修订并标准化。然而，C++也是那些支持不同编程风格的程序设计语言家族的后继，并在近年来已对软件开发产生了巨大的影响。

1.2.1 面向对象范型

近20年来，计算机科学和编程都经历了一些变革。像大多数变革一样，托马斯·库恩在1962年出版的《科学革命的结构》这本书中描写到：无论是政治巨变还是概念重组，新观点将推动改变，并挑战目前存在的正统观点。开始，两种观点竞争，至少在短期内，旧的观点占据主要的地位。随着时间的流逝，强大、流行的新观点成长起来，直到它开始取代旧的观点为止，库恩把这叫做**范型转移**（paradigm shift）。在编程中，较早的程序设计范型是**过程程序设计范型**（procedural paradigm），该范型的程序是由主程序和对数据进行操作的若干函数构成。新的范型称之为**面向对象范型**（object-oriented paradigm），该范型的程序被看做是由体现其特定特征和行为的一组数据对象构成。

面向对象编程并不是一个新观点。第一个面向对象语言是SIMULA，该语言是对编码

的仿真。它是在 1967 年由斯堪的纳维亚的计算机科学家奥利·约翰·达尔（Ole-Johan）和克利斯登·奈加特（Kristen Nygaard）设计发明的。SIMULA 的设计非常超前，它预见到了许多后来在编程中变得常见的观点，包括抽象数据类型和很多现代的面向对象范型。事实上，很多用来描述面向对象语言的术语都来自于 1967 年 SIMULA 的原始报告。

4

遗憾的是，在 SIMULA 诞生之后的很多年里，SIMULA 并没有引起太显的兴趣。第一个在计算机专业范围内具有意义的面向对象语言是 Smalltalk，它是由 20 世纪 70 年代晚期施乐公司的帕洛阿尔托研究中心开发的。该语言的目的在阿黛尔·戈德堡（Adele Goldberg）和大卫·罗伯森（David Robson）所写的《Smalltalk-80：语言及其实现》（*Smalltalk-80: The Language and Its Implementation*）一书中进行了描述，它让更多的人接触到了编程。就这点而言，Smalltalk 是施乐公司巨大努力的一部分，它提升了现代用户界面技术，而这些技术已成为当今个人电脑的标准。

尽管吸引人的特点和具有高度交互性的用户环境让编程过程更简单，但 Smalltalk 并没有取得商业上的成功。只有当它的核心思想被纳入并变成工业标准 C 语言的变种后，同行才对面向对象编程感兴趣。虽然有几个类似的工作来设计一个基于 C 的面向对象程序设计语言，最成功的语言是 20 世纪 80 年代早期由 AT&T 贝尔实验室的本贾尼·斯特劳斯特卢普（Bjarne Stroustrup）发明的 C++。C++ 兼容 C 标准，这使现有的 C 程序以一种循序渐进、演化的方式统一到 C++ 代码中成为可能。

尽管面向对象程序设计语言在面向过程语言的扩展中已十分流行，但它把面向对象和面向过程范型看作是互斥的，这可能是一个错误。程序设计范型并非没有很多竞争，但它们是互补的。面向对象和面向过程范型（连同其他重要的范型，比如在 LISP 中表现的函数编程范型）在实践中都有重要的应用。即使在单个应用的背景下，你也可能会发现使用了不止一种方法。作为一名程序员，你必须掌握不同的范型，这样你才能使用合适的模型去完成当前的任务。

1.2.2 C++ 的演化

像人类语言一样，程序设计语言也随着时间的推移发生着改变。多年来，C++ 已经逐步发展以满足用户不断变化的需求。国际标准化组织（ISO）一直在管理 C++ 新版本的发布，它在 1998 年、2003 年和 2011 年分别修订了 C++ 语言中的若干重要问题。最新的修订版称为 C++11，它引进了很多新特性，包括若干让 C++ 更容易学习的特性。

尽管每个人在某一时刻会用到 C++11 提供的很多特性，可是很多编译器还不支持 C++11 标准。在主要的编译器版本可以使用 C++ 11 之前，你可能不得不遵循 2003 年出台的旧 C++ 标准。本书会在合适的时候介绍 C++11 的特性，但是通常会概述用旧的编译器实现相同结果的策略。

5

1.3 编译过程

当你用 C++ 编写一个程序时，第一步是创建一个源文件（source file），该文件由程序文本构成。在运行程序之前，你需要将源文件转化为其可执行的格式，它需要调用被称为**编译器**（compiler）的程序。编译器将源文件转化为含有相应的机器语言指令的目标文件（object file）。该目标文件连同其他目标文件一起生成一个可以在系统中运行的可执行文件（executable file）。这些目标文件典型得包括预定义的称之为库（library）的目标文件，它包含机器语言指令用以程序所需的各种各样的通用操作。把所有多个目标文件组合成一个可执

行文件的过程称为链接（linking）。上述编译过程中的各步骤由图 1-2 表示说明。

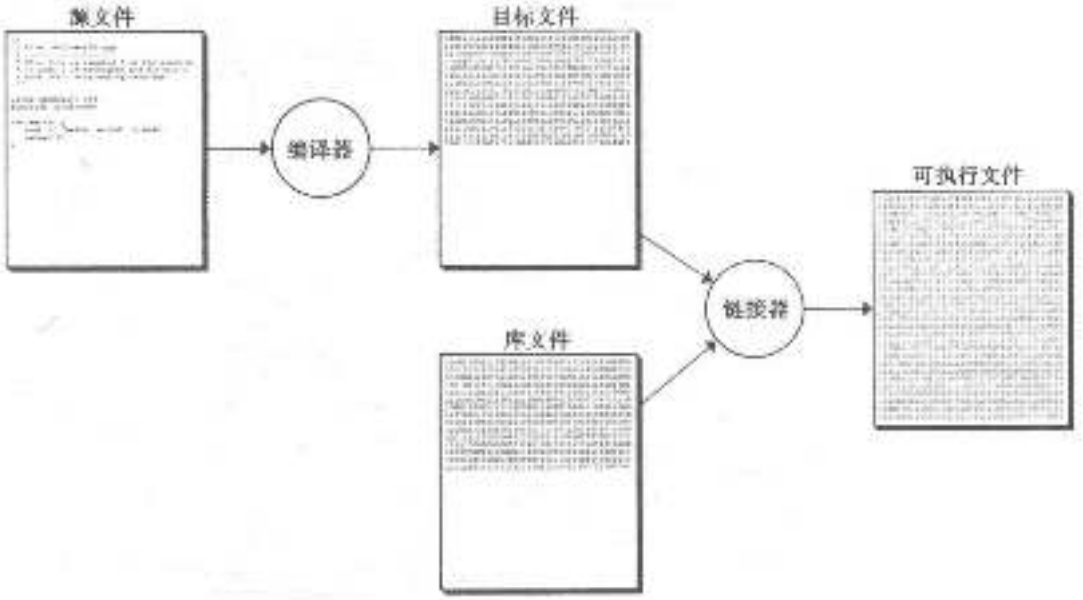


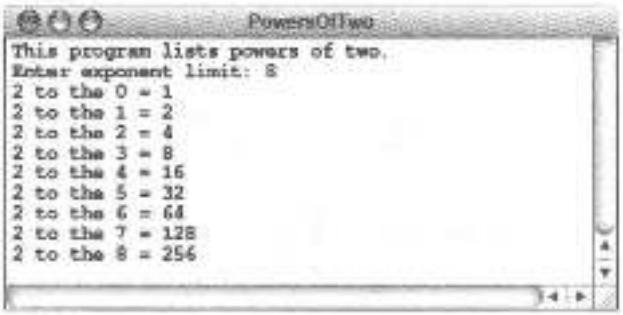
图 1-2 编译过程

正如本章早些时候讨论过的 HelloWorld 程序，在编译过程中，特殊的细节随机器不同而有所变化。没有哪一本教科书会像本书这样确切地告诉你应该用什么命令让你的程序在
6 你的系统中运行。好消息是 C++ 程序它们看起来都一样。用高级语言像 C++ 编程的一个优点是它允许你忽略硬件的不同特性，只创建的程序能够在很多不同的机器上运行。

1.4 C++ 程序结构

感受 C++ 程序设计语言的最好方法是研究一些示例程序，在你理解 C++ 语言的细节之前。这个 HelloWorld 程序是一个开始，但是它太简单了，因此在这个程序里它不能包含很多你想看到的语言特性。因为这本书适用于计算机科学中第二门课程，你几乎一定编写过程序，读取了用户的输入，用变量存储数值，使用循环来执行重复计算，并使用辅助函数来简化程序的结构。HelloWorld 程序没有做上述这些事情。为了说明更多的 C++ 特性，图 1-3 给出了一个程序源码，它计算输出 2 的幂，并且包含描述该程序各个部分的一些注释。

当你运行图 1-3 所示的 PowersOfTwo 程序时，计算机首先会询问你指数的限制，即指定该程序应产生 2 的多少次方。例如你输入 8，程序会产生一系列 2 的 0 到 8 次方的值，如下图所示：



```
/*
 * File: PowersOfTwo.cpp
 * ---------------------
 * This program generates a list of the powers of
 * two up to an exponent limit entered by the user.
 */

#include <iostream>
using namespace std;

/* Function prototypes */

int raiseToPower(int n, int k);

/* Main program */

int main() {
   int limit;
   cout << "This program lists powers of two." << endl;
   cout << "Enter exponent limit: ";
   cin >> limit;
   for (int i = 0; i <= limit; i++) {
      cout << "2 to the " << i << " = "
           << raiseToPower(2, i) << endl;
   }
   return 0;
}

/*
 * Function: raiseToPower
 * Usage: int p = raiseToPower(n, k);
 * ----------------------------------
 * Returns the integer n raised to the kth power.
 */

int raiseToPower(int n, int k) {
   int result = 1;
   for (int i = 0; i < k; i++) {
      result *= n;
   }
   return result;
}
```

程序注释

包含的库文件

函数原型

主程序

函数注释

函数定义

图 1-3 C++ 程序结构

计算机屏幕会显示你执行 PowersOfTwo 程序的运行结果。屏幕所显示的程序运行结果被称为示例运行（sample run）。在每个示例运行时，用户的输入呈现明亮的颜色以便你在程序中将输入数据和输出数据区分开。

正如图 1-3 中的注释所展示的那样，PowersOfTwo 程序可划分为多个部分，在下面的章节中我们会讨论它们。

1.4.1 注释

图 1-3 中的程序有很多由英文注释构成的文本。**注释**（comment）是被编译器忽略的文本，但注释会将程序的相关信息传递给程序员。一个注释是由符号 /* 和 */ 括起来的文本组成。它允许有多行。另外，你也可以采用单行注释，这种注释以字符 // 开始，直至该行的结束为止。除了当注释标志着一个程序还未完成，本书使用多行 /*…*/ 的注释方式，这种策略使得我们更易于发现一个程序未完成的部分。

正如你在图 1-3 中看到的那样，程序 PowersOfTwo 包含一个在开始用来对程序整体进行描述的注释，以及在 raiseToPower 函数定义之前对其功能进行更详细描述的一个注释。另外，这个程序还包括一对单行注释，它们扮演着英文文本节标题的角色。

1.4.2 包含的库文件

现代程序不用库（library）是无法编写的，库是提前编写的能执行有用操作的工具的集合。C++定义了几个标准库，其中最重要的是**输入输出流**（iostream）。这个库定义了一组简单的称之为**流**（stream）的数据结构的输入输出操作。流是用来管理流入数据源或从数据源发出信息流（如控制台或文件）的一种数据结构。

要想使用 iostream 库，你的程序必须包含这一行：

```
#include <iostream>
```

这一行指示 C++ 编译器从**头文件**（header file）中读取相关的定义。这行的一对尖括号表示该头文件是 C++ 标准中的一个系统库。从第 2 章开始，你将会有使用你自己编写或从其他库中获取头文件的需求。这些典型的头文件以后缀 .h 结尾，并且用一对引号括起来以代替一对尖括号。

在 C++ 中，仅仅读取使用 #include 所包含的相应的头文件还不足以让你访问系统库。为了确保定义在一个大系统中各个部分的程序的元素名字（如变量名、函数名等）不会相互混淆，C++ 的设计者将代码段切分成称之为**命名空间**（namespace）的结构来跟踪该结构中的名字。C++ 标准库的命名空间名为 std，这意味着你不能引用定义在标准头文件如 iostream 中的名字，除非你让编译器知道这些定义是在哪个命名空间中的。

越来越多专业的 C++ 程序员通过在所有来自 std 命名空间的名字前增加前缀 std:: 来明确指定其命名空间。如果你采用这种方式，HelloWorld 程序的第一行会变成：

9

```
std::cout << "hello, world" << std::endl;
```

如果你想向专业人士一样编码，你可以用这种方式。对一些刚学习 C++ 的人来说，这些 std:: 标记让程序变得很难阅读，所以这本书采用在库包含部分结束后增加下面这行代码的方式来省略程序中的 std:: 标记：

```
using namespace std;
```

有时你必须牢记标准库命名空间中的完整名字应包括 std:: 前缀，例如在第 2 章中，当你开始定义你自己的库接口时，这就非常重要。然而，截止目前，考虑使用下行代码：

```
using namespace std;
```

它可能是最容易想到的魔咒，C++ 编译器需要它的魔力为你的代码工作。

1.4.3 函数原型

从程序功能的角度上讲，C++ 程序的计算是在函数中进行的。一个**函数**（function）是命名为完成特定功能的一段代码。PowersOfTwo 程序包含两个函数：**主函数**（main）和 raiseToPower 函数，它们会在后面的章节进行更详细的描述。尽管这些函数的定义呈现在 PowersOfTwo 程序源文件的最后，但 PowersOfTwo 程序在包含的库文件语句后对 raiseToPower 函数进行了简洁的描述。这个简洁的形式称为**函数原型**（prototype），它使得在函数实际定义之前调用该函数成为可能。

一个 C++ 函数原型由函数定义的第一行后加一个分号组成，如以下函数原型：

```
int raiseToPower(int n, int k);
```

这个函数原型告诉编译器当它去调用出现在代码中的函数所需要知道的所有信息。正如

你将在第2章看到的函数扩展的讨论，这个函数原型表明函数将接受两个整数作为参数，并把一个整数作为返回值。

你必须在调用函数前，对每一个函数提供声明或定义。C++要求这样的函数原型声明，以使编译器可以判断函数的调用是否与函数的定义兼容。如果你不小心提供了错误的参数数量或者错误的参数类型，编译器就会报错，以便你能很容易地找到程序代码中的错误。 10

1.4.4 主程序

每一个C++程序必须有一个名为main的主函数。这个函数指定了程序计算的开始点，并且当程序开始运行时，main函数就被调用。当main函数结束它的工作并返回时，程序的执行也随之结束。

在PowersOfTwo程序中，main函数的第一行是一个**变量声明**（variable declaration）的示例，变量声明服务于程序对所使用到的值的存储空间。在这个例子中，这一行引入了一个新变量limit：

```
int limit;
```

它可以存放一个int类型的值，int标准类型代表整数。变量声明的语法之后会在“变量”一节进行详细的讨论。现在你需要知道的是：这个声明为一个整型变量创建了存储空间，你可以在main函数中使用这个空间。

main函数中第二行是：

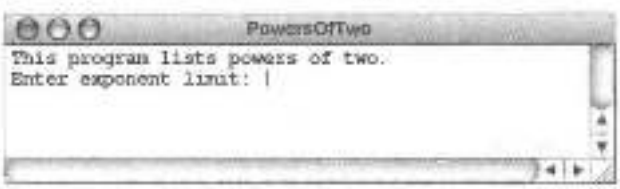
```
cout << "This program lists powers of two." << endl;
```

这一行跟HelloWorld中的声明有相同的作用，它向用户发送一条消息来表明程序的功能。标识符cout被称为**控制台输出流**（console output stream），控制台输出流是由iostream输入输出流接口定义的。这个声明的作用是将字符引号之间的字符串发送到cout流中，并伴随着行结束符endl。endl确保下一个输出操作会从新的一行开始。

接下来的两行代码要求用户输入变量limit的值。以下代码行：

```
cout << "Enter exponent limit: ";
```

仍然是向cout流打印一条消息，就像第一行所做的一样。这行代码的目的是让用户知道需要何种类型的输入值。这种消息一般被称为**提示**（prompt）。当你打印一个要求用户输入的提示时，传统上会省略endl的值，以便这个提示与用户输入能出现在同一行。当计算机执行到程序的这行代码时，计算机会显示提示，并将控制台光标（闪烁的竖线或方块）移到当前的输入位置上（即在这行的最后）以等待用户的输入，如下图所示：



对输入值实际请求的是以下这行代码： 11

```
cin >> limit;
```

标识符cin代表**控制台输入流**（console input stream），它对应cout用于从用户那里读入数

据。这条声明表明下一个来自 cin 流的值应该赋给变量 limit。然而，由于 limit 被声明为一个整型变量，操作符 >> 会自动地将用户输入的字符类型的值转化为相应的整数值。因此，当用户输入 8 并按回车键后，程序的执行效果是将 8 赋给 limit 变量。

主函数的下一行是一个 for 语句，其功能为重复执行一个代码块。像 C++ 中所有的控制语句一样，for 语句被分为定义控制操作性质的标题行（header line）和表明控制操作影响哪些语句的循环体（body）构成。该程序中的 for 语句所对应的上述划分如下所示：

```
for (int i = 0; i <= limit; i++) {          } 标题行
   cout << "2 to the " << i << " = "         }
        << raiseToPower(2, i) << endl;      } 循环体
}
```

在本章之后的“for 语句”一节中，你会有机会更详细地理解标题行。标题行表明无论循环体中的语句是什么，应该重复每一个 i 值，从 0 开始持续增至 limit，包括 limit。循环体的每一次执行都输出显示一行当前变量 i 值所对应的 2^i 的值。

循环体中的这条语句是：

```
cout << "2 to the " << i << " = "
     << raiseToPower(2, i) << endl;
```

这条语句表明了两点。第一，语句可以跨越多行，分号是语句结束的标志符，而非简单的每行的结尾。第二，这条语句展示了 cout 流链接多个输出值及将数字转换成可输出形式的能力。该输出的第一部分是以下字符序列：

```
2 to the
```

随后输出的是变量 i 的值，含有左右各一个空格的等号，函数调用

12

```
raiseToPower(2, i)
```

的值及最后行结束标志符。字符中的空格可确保数值不会连接在一起。

在程序可以打印输出行之前，它必须调用 raiseToPower 函数以查看其值是什么。调用 raiseToPower 会挂起 main 函数的执行，直到 main 函数收到调用函数的返回值为止。

与函数的典型情况一样，raiseToPower 需要从主程序 main 中获取某些信息以完成它的工作。如果你考虑增大一个数到幂所涉及的信息，你会很快意识到 raiseToPower 函数需要知道基数和指数，在该实例中基数是常数 2，指数是当前存放在变量 i 中的值。然而，这个变量是在 main 函数体内声明的，只能在 main 函数体内访问。如果 raiseToPower 要获得基数和指数值，主程序必须通过把它们放在函数名后的圆括号里，将它们作为参数传递到 raiseToPower 函数中。我们将在下一节讲授如何将这些值复制到相应的函数参数。

在 HelloWorld 程序中，main 函数的最后一条语句是：

```
return 0;
```

这条语句表明 main 函数的返回值是 0。为方便起见，C++ 采用 main 函数的返回值来报告整个程序的状态。0 值表示成功，其他值表示失败。

1.4.5 函数定义

因为较大的程序很难完全理解，因此，大多数程序被划分成几个较小的更易于理解的函

数。在程序 PowersOfTwo 中，函数 raiseToPower 是求一个整数的幂（C++中没有内置该操作），因此，它必须被明确地定义。

raiseToPower 函数的第一行是以下变量声明语句：

```
int result = 1;
```

该声明引入了一个新的变量 result，它可以存放 int 类型的值，并且将 result 初始化为 1。

函数中的第二条语句是一个类似于你在主函数中见到的 for 循环，它执行循环体 x 次。for 循环的循环体由下面这条语句构成： 13

```
result *= n;
```

该语句是 C++ 对英文句子"Multiply result by n"（result 乘以 n）的缩写。因为函数初始化 result 的值为 1，之后 result 乘以 n 做了 x 次，因此，变量 result 最后的值是 n^x。

raiseToPower 函数的最后一条语句是：

```
return result;
```

该语句表明函数应以 result 的值作为函数的返回值。

1.5 变量

在一个程序中的数值通常要存储在变量（variable）中，它是一个命名的能够存储特定类型值的一块内存区域。你已经在程序 PowersOfTwo 中看到了变量的实例，也肯定已经通过你早期的编程经历熟悉了变量的基本概念。本节的目的是概括 C++ 中的变量使用规则。

1.5.1 变量声明

在 C++ 中，你必须在使用变量前对其进行声明（declare）。声明变量的主要功能是将变量的名字和变量包含的值的类型关联起来。变量声明在程序中的位置也决定了变量的作用域（scope），它是一个变量能访问的区域。

常用的声明变量的语法是：

```
type namelist;
```

其中，*type* 表明变量的类型，而 *namelist* 是变量名列表，变量名之间用逗号分隔。在大多数情况下，每个声明引入一个变量名。例如，PowersOfTwo 程序中的 main 函数以这行代码开始：

```
int limit;
```

它声明变量 limit 是整数类型。然而，你也可以一次声明几个变量，就像下面声明了名字为 n1、n2、n3 的三个变量一样：

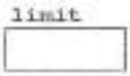
```
double n1, n2, n3;
```

在上述例子中，每个变量都声明为 double 类型（它是 C++ 中用来表示有小数部分的数值类型）。double 是双精度浮点（double-precision floating-point）的缩写，但没必要担心所有这些术语的含义。这个声明出现在图 1-4 所示的程序 AddThreeNumber 中，程序 AddThreeNumber 的功能是读取三个数，并计算输出它们之和。 14

```
/*
 * File: AddThreeNumbers.cpp
 * -------------------------
 * This program adds three floating-point numbers and prints their sum.
 */

#include <iostream>
using namespace std;

int main() {
   double n1, n2, n3;
   cout << "This program adds three numbers." << endl;
   cout << "1st number: ";
   cin >> n1;
   cout << "2nd number: ";
   cin >> n2;
   cout << "3rd number: ";
   cin >> n3;
   double sum = n1 + n2 + n3;
   cout << "The sum is " << sum << endl;
   return 0;
}
```

图 1-4 三个数相加的程序

重要的是需牢记：在变量的生存期内，所有变量的名字和其类型都是保持不变的，但变量的值一般会随着程序的运行而发生改变。为了强调变量值的动态特性，常常将变量示意成一个盒子，变量的名字像盒子的标签出现在盒子外面，变量的值出现在盒子里面。例如，你可以看到 limit 声明示意图如下所示：



赋给 limit 的值会覆盖盒子里之前的内容，但是不会改变它的名字和类型。

在 C++ 中，变量初值是没有被定义的。如果你想让一个变量有一个特定的初值，你需要对它进行明确地初始化。为此，你需要做的是在变量名后面加上一个等号和变量的初值。因此，以下声明：

15
```
int result = 1;
```

是下面这段将声明和赋初值分开的代码的简写：

```
int result;
result = 1;
```

初始值作为声明的一部分称为初始化（initializer）。

1.5.2 命名规则

变量、函数、类型、常量等名字统称为标识符（identifier）。在 C++ 中，标识符的构造规则为：

1. 名字必须以一个字母或者是下划线 "_" 开始。
2. 名字中的所有字符必须是字母、数字或者是下划线。不允许包含空格或者其他特殊的字符。
3. 名字不能包含表 1-1 中所示的 C++ 中的保留字。

在标识符中，大小写字母被认为是不同的。因此，名字 ABC 和 abc 是不同的。标识符可以取任意长度，但 C++ 编译器不会考虑任何超过 31 个字符的两个名字是否相同。

你可以通过采用规则的标识符以帮助读者识别其功能来改进你的编程风格。在本节中，

变量和函数的名字都是以小写字母开头，如limit和raiseToPower。类和其他编程者定义的数据类型的名字以大写字母开头，如Direction和TokenScanner。常量名全部用大写字母表示，如PI和HALF_DOLLAR。当一个标识符由几个英文单词构成时，通常的惯例是每个单词的首字母大写，从而使名字更易于阅读。因为这种命名方式不适合于常量，因此，程序员常采用下划线字符来分隔标识符中的单词。

表 1-1 C++中的保留字

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

1.5.3 局部变量和全局变量

很多变量声明在函数体内。这样的变量被称为**局部变量**（local variable）。局部变量的作用域可以扩展到它声明所在块的结束。当函数被调用时，为每个局部变量分配在整个函数调用时期的存储空间。当函数返回时，所有的局部变量都消亡。

若变量声明出现在函数定义以外，则这种声明将引入一个**全局变量**（global variable）。全局变量的作用域为它声明的那个文件，并且其生命期为该程序的整个运行期。因此，全局变量可以存储函数调用的值。虽然这可以看作为有用的特性，但全局变量的缺点仍超过其优点。首先，全局变量可以被程序中的任意函数操作，这很难避免函数间的相互干扰。由于函数间的紧耦合总是带来问题，因此，本书除了声明全局常量外，通常不采用全局变量，它们会在下节进行讨论。虽然全局变量有时看上去很迷人，但如果你避免使用全局变量，你会发现这样做会更易于管理你程序的复杂性。

1.5.4 常量

当你编写程序时，你会发现，在程序中相同的常量会用到很多次。例如，如果让你编写一个有关圆的几何计算程序，常量π将会频繁地出现。而且，如果这些计算要求的精度很高，这就意味着你可能会用3.141 592 653 589 793 238 46这个值来计算。一遍又一遍输入这个常数是非常枯燥的，而且如果你每次都是手工输入而不是复制、粘贴这个值的话，就很有可能导致错误。如果你给这个常量起一个名字并在程序中引用该名会更好。当然，你也可以像下行代码简单地声明pi为一个局部变量：

```
double pi = 3.14159265358979323846;
```

但是声明之后你只能按照pi被定义的方式使用它。更好的策略是声明一个全局的名为PI的常量，如下行代码所示：

```
const double PI = 3.14159265358979323846;
```

在这声明开始处的关键字const表明该值被初始化之后不能再改变，它确保了该值是一个常量。毕竟π的值不太可能会改变（尽管事实上，在1897年印第安纳州议会上的法案

试图找到了）。上述声明的其他部分由类型、常量名和值构成。该声明与其他声明的唯一不同在于C++中常量名中的字符为全部大写。

采用命名的常量有以下几个优点。首先，描述性常量名使得程序更易于阅读。更重要的是，使用常数名可以大大简化程序演化中维护代码时所出现的问题。尽管π的值不太可能发生改变，但有些"常量"的值会随着程序的演化而发生改变，尽管这些值仍为一个特定程序版本中的常数。

历史最容易说明上述原则的重要性。想象一下当你是一名20世纪60年代末的程序员，正进行着ARPANET的初期设计。ARPANET是第一个大型的计算机工作网络，它是现代互联网的前身。因为当时资源非常有限，你需要限制可以连接的主机的数量——正如真正的ARPANET设计者在1969年所做的那样。早期的ARPANET限制连接的主机数是127台。如果C++在当时已经出现了，有可能会向以下语句那样声明一个常量：

```
const int MAXIMUM_NUMBER_OF_HOSTS = 127;
```

然而，在之后的某个时间，网络的爆炸式增长会迫使你突破连接主机限制数。如果你在程序中采用命名的常量，该过程就会变得非常简单。将主机数量的限制增加到1023，仅需像如下语句修改声明即可：

```
const int MAXIMUM_NUMBER_OF_HOSTS = 1023;
```

如果你在程序中每一处使用最大值的地方都使用了常量MAXIMUM_NUMBER_OF_HOSTS，则程序会自动地将该常量的旧值替换为1023。

请注意，如果你使用数值字面值127，情况则会完全不同。此时，你需要通过搜索整个程序并将所有的127修改成更大的值。程序中的某些数值字面值127可能并不代表连接主机的限制数，此时至关重要的是不能改变这些值。若在上述事件中你出现了错误，那么你会在
18 追踪这些错误时非常难过。

1.6 数据类型

C++程序中的每个变量都可容纳一个限制于特定类型的值。作为声明语句的一部分，你必须设置变量的类型。到目前为止，你已经见过具有int和double类型的变量，但是这些类型仅仅是C++可用类型中的皮毛。如今的编程会用到很多数据类型。其中一些为编程语言的内置类型，另一些作为特定应用程序的一部分被定义。学会如何操作各种类型的数据是掌握任何一种编程语言（包括C++语言）基础的必不可少的一部分。

1.6.1 数据类型的概念

在C++中，每个数据值都有其相应的数据类型。从形式上看，**数据类型**（data type）由两个属性定义：**值集**（domain），即该类型值的集合；**操作集**（set of operation），它定义了该类型的行为。例如，int类型的值集为所有的整数，即

$$\cdots -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \cdots$$

等等，直至机器的硬件限制。例如，适合于int类型的操作集包括标准的算术运算，例如，加法和乘法。其他类型有其各自的值集和操作集。

正如你在下一章中会学到的，很多类似于C++的现代编程语言的能力来源于你可以从已存在的数据类型定义一个新的数据类型。为了启动这个过程，C++包含几种基本类型，这

些基本类型作为语言的一部分被定义。这些类型作为类型系统整体的建筑块，被称为**原子的**（atomic），即**基本类型**（primitive type）。这些预定义的基本类型分为5类：整数类型、浮点类型、布尔类型、字符类型和枚举类型——它们会在下面进行讨论。

1.6.2 整数类型

尽管整数的概念看起来是一个很简单的概念，但实际上C++中有几种不同的数据类型来表达整数值。在大多数情况下，你需要知道的是int类型，它对应着你所使用的计算机系统中整数的标准表示。然而，在某些情况下，你需要更加小心。像所有的数据一样，int类型的值存储能力有限。因此，这些值有最大值的限制，这就限制了你所使用的整数的范围。为了解决这个问题，C++定义了三种整数类型：short，int和long——它们由其值域的大小而相互区别。 19

遗憾的是，C++语言定义中没有指定这三种类型的确切的值域。因此，不同整数类型的值域取决于你所使用的机器和编译器。在计算的初期，int类型的最大值典型的是32 767，以今天的标准来看，它真是太小了。例如，如果你想执行计算一年中的总秒数，你就不能采用那些计算机的int类型是，因为计算结果值（31 536 000）要远大于32 767。现代的机器倾向于支持更大的整数，但是你只能依靠以下特性：

- 当你将一个整数从short类型转化成int类型再转换成long类型时，该整数的内存量是不能降低的。例如，C++编译器的设计者可以决定使short类型和int类型具有相同大小的内存，但绝不会让int类型比short类型的内存量小。
- int类型的最大值必须至少为32 767（$2^{15}-1$）。
- long类型的最大值必须至少为2 147 483 647（$2^{31}-1$）。

C++的设计者可以选择更确切地定义int类型的值域。例如，他们可以像Java的设计者一样在每台机器里都定义int类型的最大值为$2^{31}-1$。这样他们就会很容易将一个程序从一个系统移植到另一个系统，并且程序的行为方式不变。这种在不同机器之间移植程序的能力称为**可移植性**（portability），它是在编程语言的设计中需重点考虑的因素。

在C++中，每种整数类型int、long和short都可以在其类型名前加上关键字unsigned。增加unsigned之后，就构建出了一种新的不允许出现负值的数据类型。与有符号的整数类型相比，这种无符号整数可以提供两倍范围的值域。例如，在现代机器上，int类型的最大值典型的是2 147 483 647，但unsigned int类型最大值是4 294 967 295。C++允许类型unsigned int缩写成unsigned，实践中，大多数程序员更倾向于这种缩写形式。

一个整数常量通常写成一串十进制数字。然而，如果这个数以0开始，编译器会将这个值当做八进制数（基数8）。因此，常量040被认为是八进制数，代表的是十进制数32。如果你用字符0x作为一个数的前缀，则编译器会认为它是一个十六进制数（基数16）。因此，常量0xff代表十进制数255。你可以通过在数字串后加L来显式地指明一个整数常量的类型是long。因此，常量0L与0等价，但该值的类型是long。同样，如果用U作为一个整数常数的后缀，该常数就认为是一个无符号整数。 20

1.6.3 浮点类型

包含小数部分的数称为**浮点数**（floating-point number），在数学中它经常被用于近似实数。C++定义了三种不同的浮点类型：float，double和long double。尽管ANSI C++

并没有指定这些类型的确切表示，但若以类型所占的内存量的大小来考查类型的不同，则 long double 类型要比 double 类型长，double 类型比 float 类型长（更长的数据类型以占据更多的内存空间为代价，可表示数值的更多的精度）。然而，除非你是做严格的科学计算，否则这些类型之间的差异并不会很大。为了与 C++ 程序员的惯例一致，本书采用 double 类型作为浮点类型的标准。

在 C++ 中，浮点类型常量用带有小数点的十进制表示。因此，若程序中出现 2.0，则该数就被认为是一个浮点数。若在程序中写的是 2，那么该数就被认为是一个整数。浮点数还可以用科学计数法这种特殊的编程风格表示，其中数值被表示为一个浮点数乘以 10 的整数幂。用这种方式表示一个浮点数时，首先写一个标准的浮点数，其后书写字母 E 和一个整数指数，在整数前面选择 + 或 - 标志。例如：光速（单位为 m/s）的 C++ 表示形式为：

```
2.99792E+8
```

其中，E 代表将前面的数字乘以 10 的多少次幂。

1.6.4 布尔类型

当你编写程序时，常常需要测试一个特定的影响其代码行为的条件。通常，这一条件采用具有值为 true 或 false 的表达式来表示。这种仅具有合法常量值 true 或 false 的数据类型被称为**布尔数据**（Boolean data）——它是数学家乔治·布尔（George Boole）发明的一种处理该类数值的代数方法。

在 C++ 中，布尔类型名为 bool。你可以声明 bool 类型的变量，并且像操作其他类型 [21] 的量一样操作它。bool 类型适用的操作将在“布尔操作符”这一节中给出详细描述。

1.6.5 字符

早期，计算机被设计成只能处理数值数据，因此那时计算机常被称为**数值计算机**（number cruncher）。然而，现代计算机更多处理的是出现在键盘和屏幕上以字符表示的任何信息的文本，而非数值数据。现代计算机处理文本数据的能力推动了文字处理系统、在线参考库、电子邮件、社交网络和具有无限需求的各种令人兴奋的应用程序的发展。

表 1-2 ASCII 字符码

	0	1	2	3	4	5	6	7	8	9
0x	\000	\001	\002	\003	\004	\005	\006	\a	\b	\t
1x	\n	\v	\f	\r	\016	\017	\020	\021	\022	\023
2x	\024	\025	\026	\027	\030	\031	\032	\033	\034	\035
3x	\036	\037	space	!	"	#	$	%	&	'
4x	(	)	*	+	,	-	.	/	0	1
5x	2	3	4	5	6	7	8	9	:	;
6x	<	=	>	?	@	A	B	C	D	E
7x	F	G	H	I	J	K	L	M	N	O
8x	P	Q	R	S	T	U	V	W	X	Y
9x	Z	[	\	]	^	_	`	a	b	c
10x	d	e	f	g	h	i	j	k	l	m
11x	n	o	p	q	r	s	t	u	v	w
12x	x	y	z	{	|	}	~	\177		

文本数据最基本的元素是一个个单一的字符，一个字符在 C++ 中用预定义的基本类型 char 表示。char 类型的值域是可以出现在屏幕或键盘上的字母、数字、空格、标点符号和回车键等字符的集合。在机器内部，这些字符被表示成计算机赋给每个字符的数字代码。在大多数的 C++ 实现中，用于表示字符的代码系统被称为 ASCII，它代表“美国标准信息交换码（American Standard Code for Information Interchange）”。ASCII 字符集所对应的十进制值显示在表 1-2 中，任何字符的 ASCII 码值是该字符在表中所对应的行和列值之和。

尽管知道字符的内部码值很重要，但是知道一个数值所对应的特定字符通常更有用。当你键入字母 A，硬盘上的硬件逻辑会自动将字符 A 转化成 ASCII 码值 65，然后将 65 送入计 22
算机。类似地，当计算机将 ASCII 码值 65 输出到屏幕上时，屏幕上会显示字母 A。

在 C++ 中，你可用一对单引号括起来的一个字符来表示一个字符常量。因此，常量 'A' 代表大写字母 A 的机内编码。除了标准字符，C++ 还允许你书写以反斜杠（\）开始的由多个字符表示的一些特殊字符，这种形式被称为**转义序列**（escape sequence）。表 1-3 列出了 C++ 支持的转义序列。

1.6.6 字符串

当若干字符被聚集存储到一个连续的内存单元中时通常是非常有用的。在编程中，一个字符序列称为一个**字符串**（string）。截至目前，你已经在 HelloWorld 和 PowersOfTwo 程序中看到了字符串都被简单地用于在屏幕上显示信息，但是字符串的应用不仅限于此。

在 C++ 中，可采用一对双引号括起来的一串字符来表示一个字符串常量。与字符类型 char 一样，C++ 允许采用表 1-3 中的转义序列来表示字符串中的特殊字符。如果两个或两个以上字符串连续出现在程序中，编译器会自动将它们连接在一起。这条规则的最重要意义是，你可以将一个字符串分成几行书写，而不至于从左到右写一整行字符串。

由于字符串在很多应用中至关重要，因此，所有的现代编程语言都包含了字符串操作的相关特性。遗憾的是，C++ 定义了两种不同的字符串类型，让问题复杂化了。一种是从 C 语言继承下来的旧的字符串类型，另一种是更复杂的支持面向对象范型的 string 类。为了减少混乱，本书在能使用 string 的地方都用 string 类，对于大多数人而言，可能会不经意忽略有两种字符串形式的存在。在第 3 章概述中，这种复杂性显现出丑陋的一面，它会详细地涉及 string 类。那一刻你可以简单想象成 C++ 提供了一个内置的称为 string 的数据类型，其值域为所有字符序列的集合。你可以声明 string 类型的变量，并将字符串值作为形参和返回结果在函数之间进行传入和传出。

表 1-3 转义序列

\a	警报（蜂鸣声或响铃）
\b	退格
\f	进纸（从新一页开始）
\n	换行符（移动到下一行的起始处）
\r	回车符（移动到当前行的开始，没有前进）
\t	水平制表符（水平移动到下一个制表位）
\v	垂直制表符（垂直移动到下一个制表位）
\0	空字符（字符的 ASCII 码为 0）

（续）

\\	字符\本身
\'	字符'（只用当这个字符为常量的时候才用反斜杠）
\"	字符"（只用当这个字符为常量的时候才用反斜杠）
\ddd	任意字符

事实上，string是一个库类型，它的非内置特性确实有一些含意。如果你在程序中采用string类型，则需要在#include行中增加string库声明，如下所示：

```
#include <string>
```

另外，由于string类型是标准库命名空间的一部分，正如在本书中恒定不变所做的那样，仅当你在程序源文件的开头包括下行语句时，编译器才会识别string类型名：

```
using namespace std;
```

1.6.7 枚举类型

上一节讨论的ASCII码使我们明白：计算机通过对每个字符赋一个数值，用整数形式存储字符数据。这种通过整数元素的值域编码数据的策略实际上是一个更一般的原则。C++允许你通过列举它们值域中的元素来定义一种新的类型，该类型称为枚举类型（enumerated）。

定义一个枚举类型的语法为：

```
enum typename { namelist };
```

其中，*typename*是所创枚举类型名，*namelist*是其值域中以逗号相隔的常量列表。在本书中，所有类型名以大写字母开始，枚举常量名全部大写。例如，以下定义引入了一个新的枚举类型Direction，它的值是四个方向：

23
/
24

```
enum Direction { NORTH, EAST, SOUTH, WEST };
```

当C++编译器遇到该类型定义时，它会按常量名的顺序，从0开始给每个常量赋值。因此，NORTH被赋值为0，EAST被赋值为1，SOUTH被赋值为2，WEST被赋值为3。

C++允许你给每个枚举类型的常量显示地赋值。例如，以下类型声明：

```
enum Coin {
   PENNY = 1,
   NICKEL = 5,
   DIME = 10,
   QUARTER = 25,
   HALF_DOLLAR = 50,
   DOLLAR = 100
};
```

引入了一个表达美国货币的枚举类型，其中每一个常量定义了相应硬币的货币价值。如果你给枚举类型中的一些而非全部常量提供了初值，那么C++编译器会自动地给未赋值的常量赋以一个你所提供的最后一个常量值的后继整数值。因此，以下声明引入一个一年中各月份的类型，JANUARY赋值为1，FEBRUARY赋值为2，一直增加到DECEMBER赋值为12。

```
enum Month {
   JANUARY = 1,
   FEBRUARY,
   MARCH,
   APRIL,
   MAY,
   JUNE,
   JULY,
   AUGUST,
   SEPTEMBER,
   OCTOBER,
   NOVEMBER,
   DECEMBER
};
```

1.6.8 复合类型

在上述各节中，所描述的基本类型构成了允许你基于已存在的类型创建新类型的非常丰富的类型系统的基础。此外，因为C++是面向对象和过程的范式的集合，系统的类型包括 25
所有对象和传统的结构。在很大程度上，学会如何定义和操纵这些类型是这本书的主题。因此，就不在本章把这些类型完整地描述了，这是其余章节的内容。

在斯坦福大学教授本课程的这段时间，如果你有机会用高级方式使用面向对象编程，类和对象定义的细节会显示出来，你会更有可能掌握面向对象编程的概念。

本书采用的策略是在第6章之前推迟任何关于如何建立自己对象的讨论，到那时你会有足够的时间去发现眼下对象是如何使用的。

1.7 表达式

无论何时你想要程序完成其计算任务，你需要编写一个类似于数学表达式的表达式来指定所需的操作。例如，你想求解一个一元二次方程：

$$ax^2 + bx + c = 0$$

由高等数学可知，该方程通过以下式子计算得到两个解：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

第一个解通过在±符号的位置上用－获得，第二个解通过在±符号的位置上用＋获得。在C++中，你可以通过编写以下表达式来获得方程的第一个解：

```
(-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

与上述公式不同的是：在C++表达式中乘用“*”表示，除用“/”表示，平方根函数用名字sqrt表示（函数来自<cmath>库，将在第2章进行介绍），而不是数学符号。尽管如此，特别是你用任何现代高级程序设计语言编写过程序之后，会发现C++的表达式形式捕获了其对应的数学公式的意图是显而易见的。

在C++中，一个表达式由项和操作符组成。项（term）代表单个数据的值，必须是常量、变量，或函数的调用，例如在前面表达式中的变量a、b、c和常数2、4。操作符 26
（operator）是一个字符（有时候是短的字符序列），代表计算的操作。表1-4列出了在C++中出现的操作符。这个表包括数学操作符如＋、－，还有几个属于其他类型的操作符。会在以

后的章节中进行介绍。

1.7.1 优先级和结合律

选择在一个表中列出所有操作符的目的是建立这个操作符和其他操作符之间的优先级（precedence）关系。这是一个在不使用圆括号的情况下操作符在运算中如何结合的方法。如果两个操作符因同一个操作竞争，高优先级的先应用。因此，在以下表达式中：

```
(-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

* 的优先级高于 -，乘法 b*b 和 4*a*c 会先于减法进行运算。然而，非常重要的一点是：- 操作符出现两种形式。要求两个操作数的操作符称为**二元操作符**（binary operator），要求一个操作数的操作符称为**一元操作符**（unary operator）。当减号写在一个单独操作数的前面时，如 -b，它代表一元操作符表示取反。当减号出现在两个操作数之间，如 sqrt 中的参数形式，它代表二元操作符表示减法。一元操作符和二元操作符之间的优先级是不同的，在优先级表中分别列出了。

如果两个操作符有相同的优先级，它们通过**结合律**（associativity）来判断是否允许操作。结合性表示操作符是与左操作数还是右操作数相组合。在 C++ 中，大多操作符是左结合的（left-associative），这就意味着最左边的操作符最先求值。少数一些操作符是右结合的（right-associative），这就意味着它们从右到左组合。本章会在“赋值操作符”一节对它进行讨论。每个操作符的结合律在表 1-4 中列出。

一元二次公式说明了优先级和结合律规则的重要性。考虑一下如果你在表达式中没有给 2*a 加上一对圆括号会发生什么问题，如下式所示：

```
(-b + sqrt(b * b - 4 * a * c)) / 2 * a
```



没有圆括号，除（/）运算会先执行，因为操作符“/”的优先级与操作符“*”的优先级相同，它们的结合律均为左结合。这个例子表明在编程中使用错误的操作符会导致很严重的错误，因此，在你的程序中应避免此类错误。

表 1-4 C++ 中的操作符

以优先级组整理的操作符	结合性
() [] -> .	左
一元操作符：- ++ -- ! & * ~ (类型) sizeof	右
* / %	左
+ -	左
<< >>	左
< <= > >=	左
== !=	左
&	左
^	左
|	左
&&	左
||	左
?:	右
= op=	右

1.7.2 表达式中的混合类型

在 C++ 中，你可以编写一个包含各种数值类型值的表达式。如果 C++ 遇到一个具有不同类型操作数的操作符，编译器会根据表 1-5 所示的规则自动选择某种类型，将操作数转化为与此相同的类型。计算结果为转换后的类型，这种类型转换确保了计算结果尽可能地精确。

表 1-5 数值类型的类型转换层次

long double	最精确
double	
float	
unsigned long	
long	
unsigned int	
int	
unsigned short	
short	
char	最不精确

27 28

例如，假设 n 被声明为一个 int 类型，x 被声明为 double 类型。则表达式

```
n + 1
```

采用整数算术运算规则对表达式进行求值，产生一个 int 类型的结果。而表达式

```
x + 1
```

通过将整型 1 转换成浮点值 1.0，再采用双精度的浮点数算术规则进行相加求值，最终得到一个 double 类型的结果。

1.7.3 整数除法和求余操作符

事实上，在除法运算中比较有趣的是：将两个整数进行除法运算一般来说会得到一个整数结果。如果你编写一个以下表达式：

```
9 / 4
```

因为表达式中所有的操作数都是 int 类型，C++ 的运算规则表明这个操作的结果一定是一个整数。当 C++ 对这个表达式求值时，它用 9 除以 4，并且丢弃掉小数部分。因此，在 C++ 中，这个表达式的结果是 2，而不是 2.25。

如果你想精确地计算数学上 9 除以 4 的结果，那么，至少有一个操作数必须为浮点数。例如，以下三个表达式：

```
9.0 / 4
9 / 4.0
9.0 / 4.0
```

每一个计算结果都为 2.25。仅当表达式中的操作数全部为 int 类型时，小数部分才会被舍弃。舍弃掉小数部分的操作被称为舍去（truncation）。

C++ 中的操作符 / 与操作符 % 紧密相关。操作符 % 是当第一个操作数除以第二个操作数时，留下余数作为结果。例如，表达式

```
9 % 4
```

29

的值是 1，9 是 4 的两倍，余数是 1。下面是操作符 % 的一些应用实例：

```
0 % 4   =  0          19 % 4  =  3
1 % 4   =  1          20 % 4  =  0
4 % 4   =  0        2001 % 4  =  1
```

操作符 / 和 % 在编程中具有非常广泛的应用。例如，你可以用 % 来测试一个数是否能整除另一个数；例如，为了判断整数 n 是否能被 3 整除，你只需检查 n%3 的结果是否为 0 即可。

然而，如果参加 / 和 % 操作的操作数中有负数，谨慎使用这两种操作符是非常重要的，因为不同的机器可能会产生不同的结果。在大多数机器中，除法舍去操作的结果为 0，但它并未在 ANSI 标准中保证。通常来说，一个好的编程方法（正如本书示例中所使用的一样）在有负操作数的情况下应尽量避免使用这两种操作符。

1.7.4 类型转换

在 C++ 中，你可以通过被称之类型转换（type cast）的机制将一种类型明确地转换成另一种类型。在 C++ 中，类型转换需采用所期望转换的类型名后跟一对圆括号括起来的被转换的类型的值的书写格式。例如，如果 num 和 den 被声明为整数类型，你可以通过以下表达式来获得浮点类型的 quotient 值：

```
quotient = double(num) / den;
```

上述表达式计算的第一步是将 num 转换为 double 类型，之后，像在本章 1.7.2 一节所描述的那样进行浮点算术运算。

只要类型转换是按表 1-5 所示的类型层次向上移的，这种转换就不会丢失信息。反之，若你将一个值从更高精度的类型转换到较低精度的类型，则某些信息就可能会损失掉。例如，如果你将 double 类型转换成 int 类型，则小数部分就会被简单地舍弃。因此，表达式

```
int(1.9999)
```

的值为整数 1。

1.7.5 赋值操作符

在 C++ 中，对变量的赋值是一种内置的表达式结构。赋值操作符 = 需要两个操作数，像 + 和 - 操作符一样。赋值操作符规定其左操作数的值必须是可变的，通常为一个变量名。
30 当赋值操作被执行时，首先计算赋值操作符右边表达式的值，然后再将该值赋值给左边的变量。因此，当你计算以下表达式时：

```
result = 1
```

其结果是将值 1 赋值给变量 result。在大多数情况下，上述类型的赋值表达式常出现在简单语句中，即采用表达式之后加分号的形式，如下行语句所示：

```
result = 1;
```

这种语句称为**赋值语句**（assignment statement）。

赋值操作符会转换其右操作数的类型以使它与左操作数变量的类型相匹配。因此，如果变量 total 是 double 类型的话，你所写的以下赋值语句：

```
total = 0;
```

整型 0 被转换为 double 类型后再进行赋值。如果 n 被声明为 int 类型，那么赋值语句

```
n = 3.14159265;
```

的结果是将 3 赋值给 n，因为右操作数需进行类型转化以使它与左操作数的整数变量相匹配。

尽管赋值操作符经常出现在简单语句中，但它也可以被组合进更大的表达式中。此时，赋值操作符的功用就是简单的赋值。例如，表达式

```
x = (x = 6) + (y = 7)
```

的作用是将6赋值给x，7赋值给y，13赋值给z。在这个例子中，圆括号是必要的，因为赋值操作符=的优先级低于操作符+。作为较长表达式的一部分的赋值操作符被称为**嵌入式赋值**（embedded assignment）。

尽管嵌入式赋值使用方便，但它经常会让程序变得难以阅读，因为赋值嵌入在复杂表达式中很容易被忽视。因此，本书除了在一些赋值操作看起来更主要的特殊情况下，一般限制使用嵌入式赋值。当你想给几个变量赋同样的值时，赋值操作符会变得很重要。C++将赋值作为一种操作符的定义，使得同时给多个变量赋相同的值变为可能，而无须写多个赋值语句。仅需写如下的一条语句即可： [31]

```
n1 = n2 = n3 = 0;
```

该语句的作用是将n1、n2和n3三个变量全赋值为0。这条语句之所以能奏效，是因为C++的赋值操作符是从右向左进行的。因此，上述整个语句与下面的语句等价：

```
n1 = (n2 = (n3 = 0));
```

首先，计算表达式n3=0，它将n3赋值为0，然后，将0向左传递作为该赋值表达式的值。0被赋值给n2，然后再赋值给n1。这类语句被称为**多重赋值**（multiple assignment）。

为了编程方便，C++允许将赋值操作符与二元操作符相结合以产生一种称为**缩写赋值**（shorthand assignment）的形式。对任意二元操作符op，下述语句

```
variable op= expression;
```

等价于

```
variable = variable op (expression);
```

其中，括号用于强调整个表达式先于op计算。因此，下述语句

```
balance += deposit;
```

是下述语句的缩写

```
balance = balance + deposit;
```

即balance加上deposit。

这种缩写形式可用于C++中的任何二元操作符，你可以用下面这条语句中的-=操作符表达balance减去surcharge：

```
balance -= surcharge;
```

类似地，你可以用以下语句表达x除以10： [32]

```
x /= 10;
```

1.7.6 自增和自减操作符

除了缩写的赋值操作符，C++还提供了在编程中尤其经常使用的对变量进行加1或减1操作的更高级别的缩写形式。对一个变量加1称为**自增**（incrementing），减1称为**自减**

(decrementing)。为了以一种更简洁的形式来表示这些操作，C++ 采用符号 ++ 和 -- 来表示自增和自减。例如，在 C++ 中，语句

```
x++;
```

与语句

```
x += 1;
```

有相同的效果，而这条语句是以下语句的简写：

```
x = x + 1;
```

类似地，

```
y--;
```

与语句

```
y -= 1;
```

有相同的效果，或写为

```
y = y - 1;
```

确实自增和自减操作符比之前例子中所建议的操作更复杂。首先，自增和自减操作符都可以写成两种形式。操作符可以放在操作数的后面，如下所示：

```
x++
```

或者操作符放在操作数之前，如下所示：

```
++x
```

第一种形式，操作符在操作数之后，称为**后缀**（suffix）形式，第二种称为**前缀**（prefix）形式。

如果你所做的只是执行单独的 ++ 操作（正如你在一条语句或标准的 for 循环中那样），
33 前缀和后缀操作符具有完全相同的效果。只有当你在较长的表达式中使用自增或自减操作符时，你才会注意到前缀和后缀操作符的差别。像所有的操作符一样，++ 操作返回一个值，但该值与操作符相对于操作数的位置有关。有以下两种情况：

x++ 首先计算 x 的值，然后再自增 x。x 在自增前将其原始值返回给它临近的表达式。

++x x 首先自增，然后把自增后的新值作为一个整体使用。

-- 操作符除了执行自减外，它和自增操作符具有类似的行为。

你也许会奇怪为什么有人会用这种难懂的特性。++ 和 -- 操作符确实不是必需的。而且，并没有太多地出现这种情况，即在程序中的复杂表达式中，嵌入自增 / 自减操作符会比采用它们的简单形式更好。另一方面，在 C、C++ 和 Java 语言的历史传统中，++ 和 -- 是根深蒂固的。程序员是如此频繁地使用它们，以致这些操作已成为这些编程语言的惯用语法。鉴于它们在程序中的广泛使用，你必须理解这些操作以使于你能理解现有的程序代码。

1.7.7 布尔运算

C++ 定义了三种类别的操作符以操作布尔型数据：关系操作符、逻辑操作符和 ?: 操作符。**关系操作符**（relational operator）用于比较两个值。C++ 定义了以下六种关系操作符：

==	等于	<	小于
!=	不等于	>=	大于或等于
>	大于	<=	小于或等于

在程序中检查两个值是否相等时，注意使用“==”操作符，它是由两个等号构成的。单个等号是赋值操作符。由于使用双等号违背了传统的数学符号用法，因此，用一个等号代替双等号是特别常见的错误。这种错误也可能是难以追踪的，因为编译器并不是每次都能检查出这样的错误。单个等号通常将表达式转换为内嵌赋值，这在C++语言中是合法的，即使它并不是你所希望的。

关系操作符可用于比较基本类型的数值，例如整型数值、浮点型数值、布尔值和字符，但这些操作符也可用于类库中的许多类型，例如 string 类型。

除了关系操作符，C++语言定义了三种**逻辑操作符**（logical operator），它们采用布尔类型的操作数，并通过组合形成新的布尔值：

!	逻辑非（如果操作数是 false，结果为 true）。
&&	逻辑与（如果两个操作数均为 true，结果为 true）。
||	逻辑或（如果两个操作数全部或其中一个为 true，结果为 true）。

上述操作符的优先级依次递减。

尽管操作符“&&”、“||”和“!”分别与英语中的 and、or 和 not 相对应，但是用英语来理解逻辑操作符是不够准确的。为了避免这种不准确，以一种更形式化的、数学的方式来理解这些操作符会大有帮助。逻辑学家用**真值表**（truth table）定义了这些操作符，真值表说明了当操作数的值发生变化时，布尔表达式的值是如何变化的。真值表表 1-6 显示了每种逻辑运算的结果，并给出了变量 p 和 q 的所有可能取值。

任何C++程序以下述形式来计算一个表达式

exp_1 && exp_2

或者

exp_1 || exp_2

上述子表达式总是自左向右计算的，且一旦整个表达式的值可确定，就立刻结束计算。例如，如果 exp_1 在包含“&&”的表达式中为 false，那么不需要计算 exp_2 的值就可以确定整个表达式的值为 false。类似地，在含有“||”的表达式中，如果第一个表达式的值为 true，那么就不需要再计算第二个表达式了。这种在可以得到结果时就立刻结束的计算方式，称为**短路求值**（short-circuit evaluation）。

表 1-6 逻辑运算真值表

p	q	p && q	p || q	!p
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

C++语言还提供了在某些情况下十分有用的另外一种布尔操作符：?:操作符。在编程领域，虽然问号、冒号在代码中并不是紧挨着出现的，但它还是被称为“问号冒号”操作符。

与C++语言中的其他操作符不同，?:操作符分两部分书写，需要三个操作数。其一般形式如下：

```
(condition) ? exp1 : exp2
```

其中的圆括号并不是必需的，但是C++程序员通常在程序中使用它来强调条件的边界。

当C++程序遇到?:操作符时，它首先计算条件表达式，如果条件为true，则将exp1的值作为整个表达式的值；如果条件为false，则将exp2的值作为整个表达式的值。例如，可以像下述语句那样，使用?:操作符将x和y中较大的值赋值给max：

```
max = (x > y) ? x : y;
```

1.8 语句

C++程序由函数构成，函数由一系列语句组成。正如大多数编程语言，C++将语句划分为两种主要的类别：**简单语句**（sample statement）执行某些动作，而**控制语句**（control statement）控制程序的流程。本节回顾C++现有的主要语句形式，它们为你编写程序提供了工具。

1.8.1 简单语句

在C++中，常用的语句是**简单语句**，简单语句由表达式加分号组成：

```
expression;
```

在大多数情况下，表达式是一个函数调用、赋值，或者是变量的自增或自减操作。

1.8.2 块

正如在C++中定义的，控制语句典型地应用在一条单一的语句中。当你编写程序时，你经常想用一条特定的控制语句去控制一组语句。为了指明这一组语句序列是一个连贯单元
36 的一部分，我们用一对花括号将这些语句聚集成一个**块**（block），如下所示：

```
{
    statement1
    statement2
    ...
    statementn
}
```

当C++编译器遇见块，它会把整个块当做一条语句对待。因此，无论何时块中的助记符statement以一种控制形式模式出现时，你都可用一条简单语句或者一个块来代替它。就C++编译器而言，为了阐明块是由若干语句构成，块有时被称为**复合语句**（compound statement）。在C++中，任何块中的语句可以冠以变量的声明。

一个块内的语句常常在块内进行缩进。编译器忽略缩进，但是缩进的视觉效果会使代码易于阅读，因为它使程序的结构更清晰。经验表明每行缩进三个或四个空格有助于理解程序的结构；本书中的程序均采用缩进三个空格的形式。缩进是良好编程的关键，因此，你应该努力地开发具有一致缩进风格的程序。

1.8.3 if语句

编写程序时，你经常会检测一些条件是否满足，并采用这些检测结果来控制程序的后续

执行。这种类型的程序控制称为条件执行（conditional execution）。在C++中，表达条件执行最简单的方法就是使用if语句，它有以下两种形式：

```
if (condition) statement
if (condition) statement else statement
```

当你的解决策略只是在特定的布尔条件为真时调用一个语句的执行，你可以使用第一种形式。如果条件为假时，则跳过if语句或语句块。当你用第二种形式时，程序必须根据测试条件在两个独立的成套语句中选择其中一个。这种语句形式我们采用以下的程序进行说明，此程序显示了n是奇数还是偶数：

```
if (n % 2 == 0) {
   cout << "That number is even." << endl;
} else {
   cout << "That number is odd." << endl;
}
```

与任何控制语句一样，if语句中的控制语句可以是一条简单语句，或者是一个语句块。即使控制体是一条简单语句，为了提高程序的可读性，你需加上一对花括号。本书程序中的每一条控制语句体都非整个语句（控制形式和控制体）都被包含在一个块内，它是如此之短以至于它只占一行。

1.8.4 switch语句

对于那些有两种决策点，其条件要么是true要么是false，并且程序依据此条件执行其逻辑调用的应用，if语句是理想的选择。然而，某些应用要求更复杂的涉及若干互斥执行条件的决策结构，例如：一种情况，程序应该执行x；另外一种情况，应该执行y；第三种情况，需要执行z，等等。许多应用在这种情况下最合适使用switch语句，它遵循以下语法格式：

```
switch (e) {
 case c1:
   statements
   break;
 case c2:
   statements
   break;
   . . . more case clauses . . .
 default:
   statements
   break;
}
```

表达式e称为控制表达式（control expression）。当程序执行switch语句时，它计算控制表达式并将计算值与c1、c2等的值进行比较，每一个值必须是常量。如果c1、c2等任意一个常量和控制表达式的值相匹配，则语句跳转至该case子句执行。当程序执行到case子句最后的break语句时，则表明该case子句执行结束跳出该case子句，接着执行整个switch语句后面的语句。

default子句用来执行那些没有和控制表达式相匹配值的操作。default子句是可选的。如果没有匹配的case子句，也就没有default子句，程序不做任何处理，继续执行

switch 语句后面的语句。为了防止程序忽略一些意想不到的情况，除非你确定列举了所有的可能情况，否则在每个 switch 语句增加 default 语句是一个好的编程习惯。

我所用过的说明 switch 语句语法的代码模式都强力推荐在每个子句的后面添加 break 语句。事实上，C++ 定义了当没有 break 语句时，程序在执行完所选择的 case 语句后，会继续执行其后面的语句。当这种设计在某些情况下有用时，它会带来很多问题而不是解决方案。为了牢记在每个 case 子句后跳出的重要性，本书程序中的每个 case 子句后面都有 break 或 return 语句。

这个原则有一个例外，即多条 case 语句表示多个不同的常量时可接连出现。例如，switch 语句可能包含以下的代码：

```
case 1:
case 2:
  //otherwise
  break;
```

这表明：如果 select 表达式的值是 1 或者 2，特定的语句将被执行。C++ 编译器把上述结构当做两个 case 语句，其中第一个是空的。因为这个空语句没有 break 语句，则程序选择第一个 case 子语之后继续向下执行第二个 case 子句。然而，从这个概念角度上讲，如果你把这个结构看做是代表两种可能性的一条 case 子句，那样可能更好。

在 switch 语句中的常数必须是标量类型（scalar type），在 C++ 中，它定义成是一种其底层采用整数表示的类型。特别是字符经常用作 case 常量，正如以下所示的测试参数是否
[39] 是元音的函数：

```
bool isVowel(char ch) {
   switch (ch) {
    case 'A': case 'E': case 'I': case 'O': case 'U':
    case 'a': case 'e': case 'i': case 'o': case 'u':
      return true;
    default:
      return false;
   }
}
```

枚举类型也可以用作标量类型，如以下函数所示：

```
string directionToString(Direction dir) {
   switch (dir) {
    case NORTH: return "NORTH";
    case EAST: return "EAST";
    case SOUTH: return "SOUTH";
    case WEST: return "WEST";
    default: return "???";
   }
}
```

该函数将一个 Direction 类型的值转换为 string 类型。如果 dir 的值没有匹配到任何一个 Direction 常量，则 defalt 子句返回 "???"。

作为使用枚举类型的 switch 语句的第二个例子，以下函数返回一个特定年月的总的天数：

```
int daysInMonth(Month month, int year) {
   switch (month) {
    case APRIL:
    case JUNE:
    case SEPTEMBER:
    case NOVEMBER:
      return 30;
    case FEBRUARY:
      return (isLeapYear(year)) ? 29 : 28;
    default:
      return 31;
   }
}
```

上述代码认为测试 year 是否为闰年的函数 isLeapYear(year) 已经存在。你可以使用 || 和 && 操作符来实现 isLeapYear，如下所示：

```
bool isLeapYear(int year) {
   return ((year % 4 == 0) && (year % 100 != 0))
        || (year % 400 == 0);
}
```

该函数编写了如何判断闰年：一个闰年是任何一个能被 4 整除，但不能被 100 整除，或者能被 400 整除的年份。

返回布尔类型值的函数（如本节中的 isVowel 和 isLeapYear 函数）均称为判定函数（predicate function）。判定函数在程序设计中扮演着重要的角色，你会在本书中多次遇到它们。

1.8.5 while 语句

除了条件语句 if 和 switch 之外，C++ 还包含其他几种控制语句，它允许以循环的方式多次执行程序中的一部分。这样的控制语句称为**迭代语句**（interative statement）。在 C++ 中，最简单的迭代语句是 while 语句，它可以循环地执行语句直至条件表达式为 false。一般形式的 while 语句如下所示：

```
while (conditional-expression) {
   statement
}
```

当程序遇到 while 语句时，首先查看条件表达式的值是 true 还是 false。若条件表达式的值为 false，则循环**终止**（terminate），程序接着执行整个 while 语句后面的语句。当条件表达式值为 true 时，整个循环体被执行，然后程序返回到循环的开始再一次检查条件表达式的值。通过一次循环体语句构成了循环的一个**周期**（cycle）。

while 循环有两条重要的原则：

1. 在每个循环周期，包括第一次循环，条件表达式都会被测试。如果一开始测试结果为 false，则整个循环体一次也不会被执行。

2. 条件测试仅在循环开始的每个周期前进行。若在循环的某一时刻条件变为 false，程序不会注意到此情况直至完成全部的这个循环周期。之后，程序将再次测试条件。如果结果为 false，则循环结束。

以下函数展示了 while 循环操作，该函数计算一个整数的每位数字之和：

```
int digitSum(int n) {
   int sum = 0;
   while (n > 0) {
      sum += n % 10;
      n /= 10;
   }
   return sum;
}
```

该函数依赖下述观察：

- 表达式 n % 10 返回正整数 n 的最后一位数。
- 表达式 n / 10 返回整除运算后舍去 n 最后一位数的一个整数。

while 循环被设计成适合于重复操作执行前需首先进行某些条件测试的情况。若你试图解决的问题适合于这种结构，那么 while 循环是一种理想的工具。很遗憾，很多程序设计问题并不简单地适合于标准的 while 循环结构。某些问题并不是在操作的开始允许进行某种方便的测试，而大多数是要求你在循环的中间来编写判断循环是否能够结束的测试条件代码。

这种循环结构最常见的例子就是从用户那里读取数据，直到遇到某个表示输入结束的特定数值，即**信号量**（sentinel）。当用英语表达时，基于信号量的循环包括以下重复的步骤：

1. 读入一个值。
2. 当读入值与信号量相等，退出循环。
3. 否则，执行该值所需的处理。

遗憾的是，在循环开始时，没有可执行的测试以确定循环的结束。循环的终止条件是输入值与信号量的值相等。为了检查这一条件，程序首先需要读取一些数。如果程序还未读入一个数，终止条件就没有作用。

在测试终止条件前必须执行某些操作时，程序员碰到了称之为**循环和一半问题**（loop-and-a-half problem）的情况。C++ 提供的解决该问题的策略是使用 break 语句。除了 break 语句在 switch 语句中的应用外，它同样可以立即结束最内层的循环。采用
42 break，可能以下面遵循问题的本来结构的形式来编写循环结构，这种循环编程模式称为**读直到信号量模式**（read-until-sentinel pattern）：

```
while (true) {
   Prompt user and read in a value
   if (value == sentinel) break;
   Process the data value
}
```

注意

```
while (true)
```

代码行似乎引入了一个无限循环，因为常量 true 永远也不可能变为 false。该程序跳出循环的唯一方式是通过执行 while 语句中的 break 语句。图 1-5 所示的 AddIntegerList 程序采用读直到信号量模式来计算一系列整数之和，程序采用信号量值 0 来结束循环。

还有其他一些策略来解决循环和一半问题，这些策略大多数使用复制一段程序放在循环体之外，或者引入额外的布尔变量。已有的经验表明：在循环中使用 break 语句退出循环

相对于使用其他策略而言，更有可能编写出正确的程序。以上证据和我的经验使我相信：使用读直到信号量模式是解决循环和一半问题的最佳解决方案。

1.8.6 for 语句

在 C++ 中，最重要的控制语句就是 for 语句，它适合你以特定的循环次数来重复执行某个操作。所有现代编程语言都有一条对应的完成该功能的语句，但是在 C 语言家族中，for 语句在各种应用中其功能特别强大和有用。

在本节之前的 PowersOfTwo 程序中，你已经见到了两个 for 循环的实例。第一个出现在主程序中，它通过所期望的指数值来执行循环。该循环看起来如下所示：

```
for (int i = 0; i <= limit; i++) {
   cout << "2 to the " << i << " = "
        << raiseToPower(2, i) << endl;
}
```

43

```
/*
 * File: AddIntegerList.cpp
 * ------------------------
 * This program adds a list of integers.  The end of the input is
 * indicated by entering a sentinel value, which is defined by
 * setting the value of the constant SENTINEL.
 */

#include <iostream>
using namespace std;

/*
 * Constant: SENTINEL
 * ------------------
 * Defines the value used to terminate the input list.  This value must
 * be chosen so that it is not one that could naturally appear in the
 * input data.  In the AddIntegerList application, the value 0 is an
 * appropriate sentinel because the user can simply skip any 0 values
 * in the input.
 */

const int SENTINEL = 0;

/* Main program */

int main() {
   cout << "This program adds a list of numbers." << endl;
   cout << "Use " << SENTINEL << " to signal the end." << endl;
   int total = 0;
   while (true) {
      int value;
      cout << " ? ";
      cin >> value;
      if (value == SENTINEL) break;
      total += value;
   }
   cout << "The total is " << total << endl;
   return 0;
}
```

图 1-5 一系列整数之和的程序

第二个循环实例出现在函数 raiseToPower 的实现中，具有以下形式：

```
for (int i = 0; i < k; i++) {
   result *= n;
}
```

上述两个循环实例都展现了当你编写程序时会经常采用的惯用模式。

44

在上述两个模式中，第二个更常见。该模式的一般形式如下：

```
for (int var = 0; var < n; var++)
```

该循环的功能是执行循环体 n 次。你可以在这个模式中替换任意一个变量名，但那个变量通常在 for 循环体内根本不用，正如它在函数 raiseToPower 中的情况一样。

在主程序中，数据计数模式通常采用以下的一般循环形式：

```
for (int var = start; var <= finish; var++)
```

在该模式中，for 循环体通过对循环变量 var 设置为 start 和 finish 之间的（包括 finish）每一个值，其循环体被多次执行。因此，你可以采用取值为从 1 到 100 的循环变量 i 来表示如下的 for 循环语句：

```
for (int i = 1; i <= 100; i++)
```

你也可以使用 for 循环完成阶乘功能，它定义为 1 到 n 之间的所有整数的乘积，通常用数学式 n！表示。类似于函数 raiseToPower，其实现代码如下所示：

```
int fact(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

在上述实现中，for 循环确保循环变量 i 从 1 计数到 n。循环体通过给 result 乘以 i 来更新 result 值，从而依次得到它所期望的值。

在 for 循环中，使用的变量称为**循环变量**（index variable）。常用单个字符的变量名，例如 i 和 j，它们作为循环变量至少可以追溯到 FORTRAN 语言的早期版本，该语言要求循环变量为整型变量，且其变量名应为从字母表中已预定义的字母开头。尽管简短的变量名通常很少选用，因为它们传达了太少的关于该变量的目的信息，但事实上，这种循环变量简短的命名公约使得它在这种 for 语境中很合适。当你看到 for 语句中的循环变量 i、j 时，
45 你会很自然地确信这个变量是在某个值域内计数的。

不过，C++ 的 for 循环比之前所提到的实例更通用。for 循环的一般模式如下：

```
for (init; test; step) {
   statements
}
```

上述代码和下面的 while 语句等价：

```
init;
while (test) {
   statements
   step;
}
```

该代码段以 init 开始，这典型的是一个变量声明，并在循环开始之前用于对循环变量的初始化。例如，若你写了以下语句：

```
for (int i = 0) . . .
```

则循环以循环变量 i 被设置为 0 开始。如果循环以下述形式开始：

```
for (int i = -7; . . .
```

则变量 i 从 -7 开始，以此类推。

test 表达式是条件测试，正如 while 语句中的测试一样。当测试表达式为 true，循环继续。因此，以下循环：

```
for (int i = 0; i < n; i++)
```

i 从 0 开始直至小于 n，即以 0、1、2 等，直到最后一个值 n-1，总共执行 n 次循环。在以下这个循环中：

```
for (int i = 1; i <= n; i++)
```

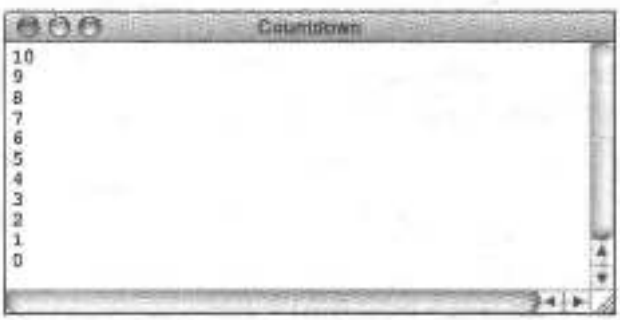
i 从 1 开始直至小于等于 n，即以 1、2 等，直到 n，总共执行 n 次循环。

step 表达式表明循环变量从一次循环到下一次循环的递增值。常用的 step 规范形式是使用 ++ 操作符来使循环变量自增，但这并不是唯一的形式。例如，也可采用反向计数的 -- 操作符，或不用 ++ 操作符采用 += 2 表示循环变量每次自增 2。 [46]

以下程序展示了一个反向计数的实例：

```
int main() {
   for (int t = 10; t >= 0; t--) {
      cout << t << endl;
   }
   return 0;
}
```

该程序的运行结果如下：



for 语句括号内的每个表达式都是可选的，但是分号不可少。如果没有 init 部分，就无初始化的执行。如果没有 test 部分，C++ 就默认为 true。如果没有 step 部分，在整个循环周期内循环变量就不会改变。

本章小结

本章是对 C++ 的概述，很难精简它为几个要点。目的是为了给你概要性地介绍 C++ 编程语言，快速地培养你使用这个语言编写一些简单的程序。本章集中在语言的底层结构、表达式和语句的概念上，将它们集合起来就可以定义函数。

本章的要点包括：

- 在 C++ 编程语言存在的 30 多年里，它已经成为世界上最广泛使用的编程语言。
- 典型的 C++ 程序由注释、包含的库文件、程序级定义、函数原型、当程序启动时所

[47] 调用名为 main 的函数，以及与 main 函数共同完成程序功能的一组辅助函数构成。

- C++ 中的变量在使用前必须声明。C++ 中大部分变量是局部变量，它们声明在函数体内，也仅能在声明它的那个函数体内使用。
- 一个数据类型是由其值域和其操作集定义的。C++ 中内置了几种基本类型，这些类型允许程序存储常见的数据值，包括整数、浮点数、布尔值和字符。正如在后面章节中你会学到的，C++ 允许程序员基于已存在的类型定义一些新的类型。
- C++ 完成输入输出操作最简单的方式是使用 iostream 库。这个库定义了三种涉及控制台的标准流：cin 从控制台读取输入数据，cout 向控制台编写正常程序的输出，cerr 向控制台报告错误信息。控制台输入习惯上用 >> 操作符表示，如以下语句：

```
cin >> limit;
```

它从控制台读取一个值存储到变量 limit 中。输出控制台使用 << 操作符表示，如以下语句所示：

```
cout << "The answer is " << answer << endl;
```

它在控制台上显示 answer 的值，及它的识别标签 " The answer is"。endl 确保了下一个控制台的输出显示在新的一行。

- C++ 中表达式的书写形式和其他大多数的程序设计语言一样，由各种操作符构成。C++ 的操作符以及它的优先级和结合性列示在表 1-4 中。
- C++ 中的语句分为两大类：简单语句和控制语句。一条简单语句是一个表达式（典型的是一条赋值表达式或函数调用表达式），后面连着分号。本章所讲述的控制语句有 if、switch、while 和 for 语句。前两个用于表示条件执行，后两个用于进行具体的循环。
- C++ 程序一般由若干个函数构成。你可以使用本章的例子作为你编写自己函数的模型，或者你可以等到第 2 章涵盖了更多的函数细节后你再编写自己的函数。

复习题

1. 当你编写一个 C 程序时，你是准备一个源文件还是一个目标文件？
2. C++ 程序中用什么字符标记注解？
3. 在 #include 这一行，库的头文件名可以用尖括号和双引号包含。这两种形式的标记法有什么
[48] 不同？
4. 如何定义一个值为 2.54，名字为 CENTIMETERS_PER_INCH 的常量？
5. 什么函数名必须在每个 C++ 程序中定义？在函数的结束处，典型的是哪种语句？
6. 当你编写 cout 输出流时，endl 的目的是什么？
7. 定义以下与变量有关的术语：name、type、values 和 scope。
8. 指出以下哪些是合法的 C++ 变量名：

```
a. x                          g. total output
b. formula1                   h. aVeryLongVariableName
c. average_rainfall           i. 12MonthTotal
d. $correct                   j. marginal-cost
e. short                      k. b4hand
f. tiny                       l. _stk_depth
```

9. 数据类型所定义的两个属性是什么？

10. short、int 和 long 类型的区别是什么？
11. ASCII 代表什么？
12. 列出所有 bool 类型的可能取值。
13. 从用户处读取数据并将其读入到 double 类型的变量 x 中，这段程序需要包含哪些语句？
14. 假设一个函数包含以下声明：

```
int i;
double d;
char c;
string s;
```

编写一条在屏幕上显示上述每个变量名和其值的语句，以便你可以区分这些变量。

15. 给出以下表达式的值和类型：

```
a.  2 + 3                    d.  3 * 6.0
b.  19 / 5                   e.  19 % 5
c.  19.0 / 5                 f.  2 % 7
```

16. 一元减和减法操作符的区别是什么？
17. 术语舍去（truncation）是什么意思？ [49]
18. 什么是类型转换？在 C++ 中如何表示它？
19. 计算下列表达式的结果：

```
a.  6 + 5 / 4 - 3
b.  2 + 2 * (2 * 2 - 2) % 2 / 2
c.  10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 * 2 + 1
d.  1 + 2 + (3 + 4) * ((5 * 6 % 7 * 8) - 9) - 10
```

20. 你如何指定一个缩写的赋值操作？
21. 表达式 ++x 和 x++ 的区别是什么？
22. 短路求值是什么意思？
23. 写出下列控制语句的一般语法形式：
    if、switch、while 和 for。
24. 用英语描述 switch 语句操作，包括在每个 case 语句后面使用的 break 语句。
25. 什么是哑元素？
26. 在下述情况中，你会用哪个 for 循环的控制段？
    a. 计数从 1 到 100。
    b. 计数从 0 开始，间隔为 7，直到数字超过两位数。
    c. 计数从 100 到 0 的间隔为 2 的倒序。

习题

1. 编写一个程序，它读取摄氏温度（C），显示对应的华氏温度（F）。摄氏温度与华氏温度的换算公式是：

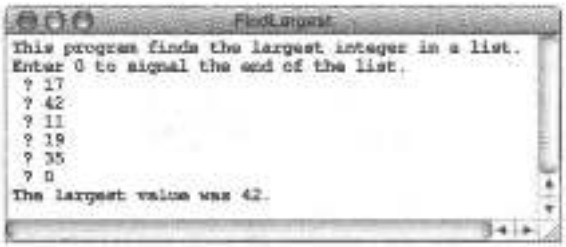
$$F = \frac{9}{5}C + 32$$

2. 编写一个程序，它将以米为单位的长度换算成相应的以英寸和英尺为单位的长度。你需要的换算关系为：
   1 英寸=0.0254 米
   1 英尺=12 英寸
3. 在数学史有这样一个故事，德国数学家卡尔·弗里德里希·高斯（1777~1855）在很小的时候就很有数学天赋。当高斯还在小学的时候，老师就让他计算从 1 到 100 的数字之和。他很快给出了答案： [50]

5050。编写一个程序，计算高斯的老师所提出的问题。

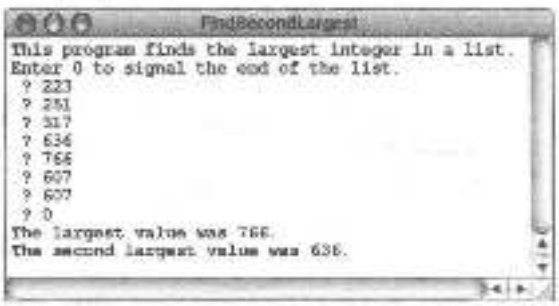
4. 编写一个程序，它读取正整数N，计算前N个奇数之和。例如，N是4，你的程序结果是计算1+3+5+7，结果为16。

5. 编写一个程序，从用户处读取数据直到用户输入了0信号量为止。当信号量出现时，程序需要显示读取数据的最大值，如以下运行结果：



确保你的信号量是个常量而且易于改变。并且确保你的程序当输入数据均为负数时也能正确运行。

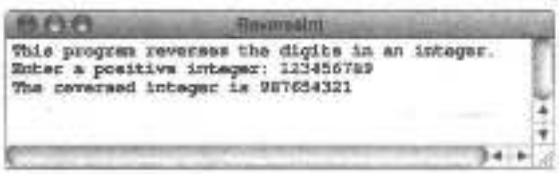
6. 做一些较为有趣的挑战，编写一个程序，在输入信号量后找出输入数据中最大和次大的数值。如果继续使用0作为信号量，程序运行结果如下所示：



51 这个例子使用的数据是乔安娜·凯瑟琳·罗琳《哈利·波特》系列英国精装版的页数。输出告诉我们最长的书《哈利·波特与凤凰令》有766页，次长的书《哈利·波特与火焰杯》有636页。

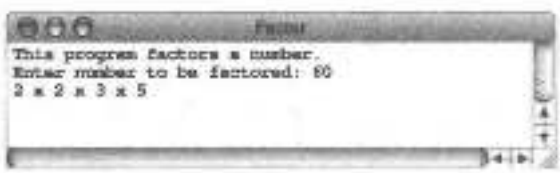
7. 以图1-5中的AddIntegerList程序为模板，编写程序AverageList，它读取成绩分数，并显示平均值。因为某些预备生的成绩可能实际为0，你的程序应该使用-1作为输入结束的信号量。

8. 使用"while语句"那节的digitSum函数作为模板，编写一个程序，读取一个整数，然后逆序输出该整数中的各位数，如以下示例运行结果：



9. 每个大于1的正整数可以表示成几个素数的乘积。这类因式分解是唯一的，被称为质因数分解(prime factorization)。例如，数字60可以被分解成2×2×3×5，每个因数都是素数。注意：在因式分解中，同样的素数可以出现多次。

编写一个程序，对数字n进行质因数分解，如以下示例运行结果：



10. 1979年，印第安纳大学认知科学教授道格拉斯·霍夫斯塔勒（Douglas Hofstadter），写了《哥德尔、艾舍尔、巴赫》一书，这本书他形容为"本着刘易斯·卡罗尔精神，在心灵与机器隐喻中的一首赋格曲"。该书获得了普利策文学奖，并且，多年来它已成为计算机科学的经典著作之一。这本书的魅力在于它包含了数学上可以用计算机程序的形式来表达的若干古怪和难解的问题。最有趣的一 52
个问题是通过对一个特定的正整数 n 重复地执行以下规则，便可形成一系列数：

- 如果 n 等于 1，那么已经到达这个序列数的终点，可以停止。
- 如果 n 是偶数，将它除以 2。
- 如果 n 是奇数，将它乘以 3 再加 1。

虽然这个数序列有很多名称，但通常称它为**冰雹序列**（hailstone sequence），因为这些值忽上忽下，如冰雹在云中的形成。

编写一个程序，让用户输入数据，然后从该数据产生冰雹序列，如以下程序运行结果：



正如你所看见的，该程序记录了它所执行的每个过程的数据变化。如 Hofstadter 在他书中描述的那样。

冰雹序列最迷人之处是直到目前还没有人证明它最终能停止。冰雹序列的计算过程可以有很多步，但不知何故，它总能跌回到 1。

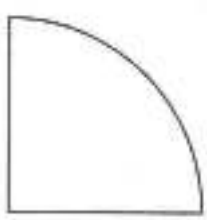
11. 德国数学家莱布尼兹（1646—1716）发现了一个引人注目的事实：数学常量 π 可以用下面的公式计算：

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$

该等式的右边代表了一个无穷级数：每个分数代表了该级数中的一项。所有奇数从 1 开始，减去三 53
分之一，加上五分之一，以此类推。你按上述公式一直做下去，所得到的值就会越来越接近于 π/4。

编写一个程序，计算包含莱布尼兹级数的前 10 000 项的 π 的近似值。

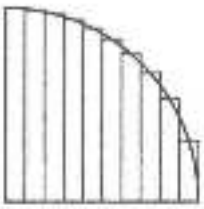
12. 你也可以通过划分圆近似计算 π。考虑下面的四分之一圆：



半径 r 等于 2 英寸[⊖]。从圆面积公式中，你可以确定四分之一圆面积为平方英寸。你可以通过增加

⊖ 1 英寸=0.025 4 米。

一系列的矩形近似计算该区域，每个矩形的宽度相同，高度由圆穿过矩形上底的中点确定。例如，你可以从左到右将这块区域划分为 10 个矩形，得到下面的图：



所有的矩形面积之和近似于四分之一圆。矩形越多，其面积就越接近于 π 的近似值。

每个矩形，宽度 w 是一个取决于将半径分为多少个矩形的常量。另外，高度 h 的变化依赖于矩形的位置。如果矩形中点的水平距离为 x，矩形的高度可以用平方根公式计算得出：

$$h = \sqrt{r^2 - x^2}$$

每个矩形的面积记为 h×w。

54 编写一个程序，计算将四分之一圆划分成 10 000 个矩形的面积。

函数与库

你的声就是你的乐园。

——伊拉斯谟（Desiderius Erasmus），《费舍尔在鹿特丹的研究》
（*Fisher's Study at Rotterdam*），1524

55

正如第1章中的示例，C++程序通常由一系列函数构成。本章将深入地阐述函数的基本概念，及其在C++中的具体实现思想。此外，本章将探讨如何把函数存储于库中，这使得函数在各种应用开发中的使用变得更加简单。

2.1 函数概念

在学习编程的早期阶段，最重要的职责就是学习如何在编程中更为有效地使用函数。幸运的是，编程中函数的概念与数学中的函数是类似的，因此，你可以完全不必从头开始学习这一新概念。同时，C++语言中的函数比数学中的函数更加通用，这也意味着你将不得不超越数学上的函数概念，来深入地思考你将怎样以一个程序员的身份去使用函数。接下来的章节先从数学上的函数概念开始，然后围绕着编程范围内的函数应用来进一步扩展这一概念。

2.1.1 数学中的函数

当你在中学学习数学时，你肯定已接触过函数的概念。例如，你可能曾经看到过如下的函数定义：

$$f(x) = x^2 + 1$$

上式表明了函数f是将数x首先转换到数x的平方然后再加上一的一个变换。对于x的任何取值，你能很容易地通过上述函数求值公式计算出其函数值。因此，$f(3)$的值是3^2+1，即10。

自从20世纪50年代FORTRAN语言诞生以来，程序设计语言已经在其自身的计算架构中组合了函数这一数学方法。例如，通过第1章有关函数的例子，你发现在C++中可以通过如下语句实现函数f：

```
double f(double x) {
   return x * x + 1;
}
```

在上述C++函数的定义中，虽然包括了一部分数学公式中未出现的语法，但其函数的基本思想是一致的。函数f通过变量x来获得输入值并返回表达式x*x+1的值。

56

2.1.2 编程中的函数

在程序设计语言中，函数的概念变得比在数学中更加宽泛。与数学中的函数相对应，C++中的函数可以指定其输入值，虽然有时并不一定非要如此。同样，C++中的函数不要求一定要返回结果。C++函数最基本的特征就是通过函数名与计算操作（一个称为函数体的代

码块）相关联。一旦某个函数被定义，程序其他部分就可以仅仅通过使用函数名来触发由函数所定义的计算操作，而无须重复编写函数定义中计算类作中的程序代码，因为它们已经在函数体中定义过了。

为了正确地理解C++中的函数，首先定义在编程中所用到的几个术语。函数（function）是一个被组织成具有特定名称的独立单元的代码块。通过使用函数名来调用函数代码（块）的行为称为函数调用（calling）。为了调用C++中的一个函数，你应该写下函数名，后面跟下一对用圆括号括起来的表达式列表。这些表达式称为函数的实参（argument），它允许函数调用者向被调用函数传递信息。如果一个被调用函数不需要调用者传递任何信息，函数就不需要实参，但是在定义和调用这种无实参的函数时，都必须在函数名后给出一对空的不含有任何实参的圆括号。

一旦函数被调用，函数将获取函数实参所提供的值，执行函数的功能（即执行函数体中的代码），然后返回程序的函数调用点。记忆主调程序的工作情况以便程序返回函数调用的确切位置是函数调用机制的主要特性之一。这种返回主调程序的操作称为从函数中返回（returning）。有时在函数返回时，向函数调用者返回一个函数值，它称为**返回值**（returning a value）。

2.1.3 使用函数的优点

函数在程序设计语言中扮演着重要的角色。首先，定义函数让编程人员可以将一段完成特定任务的操作代码仅编写一次，但可以多次使用。因此，将完成特定任务的一系列代码组织成一个函数，不仅可以显著地降低程序的规模，而且使程序更易于维护。如果你要对函数实现的操作进行改变，你会发现只出现一次的代码会比贯穿整个程序多次出现的相同代码更容易修改。

即使函数只在程序中使用了一次，定义这个函数也是值得的。函数最主要的作用就是
57 将一个大型程序分解成多个易于管理的小部分。这一过程称为**分解**（decomposition）。以往的编程经验告诉我们，将整个程序写成一个庞大的代码块必然会导致一场灾难。而你所需做的应是将一个高层次问题细分为一系列低层次的函数，每一个函数有其自己独立的功能。然而，找到问题正确的细分方法有很大的挑战，需要不断练习、思考与尝试。一个好的、独特的细分方法，会使每一个函数都是一个聚合紧密的单元，使得问题整体更加易于理解。如果你选择了一个不好的分解方法，它终会成为你解决问题的阻碍。但上并不存在准确的快速的法则让你找到最正确的问题分解方法，编程是一门艺术，好的问题分解策略主要来源于实际经验。

然而，作为一种通用的规则，问题的分解过程一般从程序的主程序开始。此时，我们将整个程序视为一个整体，并尝试从中分析并抓取出其主要部分。一旦程序的最主要部分被识别出来，就可以将它们定义为一些相互独立的函数。由于某些函数可能本身依然复杂，因此，通常需要将它们再分解为更小的部分。我们可以不断重复这一分解过程直到每个问题足够简单明了以使于解决。上述分解过程称为**自顶向下的程序设计**（top-down design）或**逐步求精的方法**（stepwise refinement）。

2.1.4 函数和算法

函数除了作为一种管理程序复杂性的工具之外，在编程中，重要的是，函数也为算

法（algorithm）的实现提供了基础性的保障。算法是一种用于解决计算问题的精确指定策略，这一术语起源于9世纪的波斯数学家穆罕默德·花拉子米（Muhammad ibn Mūsā al-Khwārizmī）的数学论著《代数学》（*Kitab al jabr w'al-muqabala*）中，后来产生了英文单词。然而，数学算法的产生可以追溯至历史上更早的时期，已确定最早延伸至古希腊、古中国、古印度文明时期。

历史上最著名的一个数学算法是以希腊数学家欧几里得的名字命名的，在托勒密一世统治时期（公元前323年~公元前283年），他居住在亚历山大。在欧几里得的数学著作《几何原本》中，描述了一个求两个整数x和y最大公约数（greatest common divisor，gcd）的过程，即一个可以同时整除x和y的最大整数的算法。例如，49和35的gcd是7，6和18的gcd是6，32和33的gcd是1。欧几里德算法可以描述如下：

1. 用x除以y并计算余数r。
2. 若r等于0，则算法结束，最大公约数是y。
3. 若r不等于0，则令x的值为y，y的值为r，重复该过程。 58

你可以很容易地将上述算法转换成C++代码：

```
int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

欧几里得算法相比你自己可能发现的任何计算策略显得更有效率，而且，该算法至今在包括网络安全的加密协议实现等很多情况下都得到了广泛的应用。

同时，我们很难清晰明确地看出该算法为什么会得到正确的结果。幸运的是，对于现在那些依赖于这个算法的人来说，欧几里得算法的正确性已经在《几何原本》第7章命题2中得到了证明。虽然并不是总有证据来证明算法对计算机应用的驱动作用，但这些证据能让你对程序的正确性更有信心。

2.2 库

当你编写一个C++程序时，计算机执行的大多数代码并不是你自己编写的代码，而是你从标准库中加载到应用程序中的代码。不管怎样，当今的程序就像海洋上的一座冰山，其大部分体积隐藏在水面之下。如果你想成为一个高效的C++程序员，那你就必须至少花费和你学一门语言本身一样多的时间来学习标准库。

在本书中你所看到的每一个程序，即使是第1章开头的HelloWorld小程序，都包含标准库<iostream>，这个库提供了cin和cout数据流。当你写HelloWorld和PowersOfTwo程序时，这些流是如何实现的对你来说并不重要。事实上，这些流的实现方法对于现在的你来讲还太过于遥远，也超出了本书讨论的范围。作为一个程序员，你所要做的就是知道如何使用这些库来实现它们所承诺的功能。

编程初学者有时候会对在不知函数的底层实现情况下调用函数这一概念感到不自然。然而，事实上，在数学领域你可能遇到这种情况已很久了。上中学时，你大概遇到过一些很有 59

用的函数，即使你当时不知道其原理也可能没兴趣知道。例如，在你的代数课上，你学习过对数函数和平方根函数。如果你选修了三角学课程，就会懂得正弦三角函数和余弦三角函数。想要知道这些函数的值，你无须手动计算其结果，只需要查询函数表，甚至可以将合适的值输入到计算器里来得到其结果。

编写程序需要和上述几乎一样的策略。如果你调用一些数学函数，那么你要做的会比正确地按计算器相应的按键更少。幸运的是，C++ 有一个强大的数学函数库，称为 <cmath>，它包含了编程中所需要的几乎所有的数学函数。表 2-1 列出了 <cmath> 中最常用的函数。不用担心自己不了解其中一些函数的功能，本书只需要很少的数学知识，并且会对超出基本代数内容的概念进行说明。

表 2-1 <cmath> 库中的一些基本函数

通用数学函数	
abs(*x*)	返回 *x* 的绝对值
sqrt(*x*)	返回 *x* 的平方根
floor(*x*)	返回小于或等于浮点数 *x* 的最大的整数值
ceil(*x*)	返回大于或等于浮点数 *x* 的最小的整数值
对数和指数函数	
exp(*x*)	返回 *x* 的指数函数值（e^x）
log(*x*)	返回 *x* 的自然对数值（基数为 e）
log10(*x*)	返回 *x* 的常用对数值（基数为 10）
pow(*x*, *y*)	返回 *x* 的 *y* 次方的值
三角函数	
cos(*theta*)	返回角 *θ* 的余弦值，其中 *θ* 为弧度，可用 *θ* * π/180 转换成度
sin(*theta*)	返回弧度 *θ* 的正弦值
tan(*theta*)	返回弧度 *θ* 的正切值
atan(*x*)	返回弧度 *θ* 的反正切值。结果为 −π/2 ~ π/2 之间角 *θ* 角的弧度值
atan2(*y*,*x*)	返回 *x* 轴与原点和点 (*x*,*y*) 所形成直线的夹角的弧度值

每当被程序包含的库在程序中发挥了作用时，计算机科学家便称该库导出了服务。例如，<iostream> 库导出了 cin 和 cout 流，<cmath> 库导出了 sqrt 函数以及表 2-1 中所列出的函数。

设计标准库的一个目标就是隐藏底层的复杂实现细节。通过导出 sqrt 函数，<cmath> 库的设计者让程序编写者更容易地使用该库。当你调用 sqrt 函数时，你不需要知道 sqrt 函数内部是怎样运行的。这些细节只与 <cmath> 库的设计实现人员有关。

懂得如何调用 sqrt 函数和懂得该函数是如何实现的，这两种能力很大程度上是相互独立的，但这两者都是程序员应具备的重要编程能力。优秀的程序员总是使用那些自己对其实现毫无头绪的函数。相反，实现库函数的程序员永远不能预测库函数的潜在用户。

为了强调库的实现者和使用者这两个概念的不同，计算机科学家为这两种人赋予了不同的名称。自然地，实现了库的程序员称为**库的实现者**（implementer）。相反地，调用库的程

序员称为库的用户（client）。当你阅读本书时，你将有机会同时从库的实现者与用户这两个角度来接触几个不同的库，首先是从用户的角度，然后是从实现者的角度。

2.3 在 C++ 中定义函数

显然你已经在第 1 章看到过几个函数，甚至也尝试过自己编写一些函数，但在深入研究怎样高效使用函数之前，你依然需要回过头来复习一下 C++ 语言中编写函数的方法。在 C++ 中，函数定义具有如下语法形式：

```
type name(parameters) {
   . . . body . . .
}
```

在这个例子中，*type* 是函数的返回类型，*name* 是函数名，*parameters* 是以逗号分隔的函数形参列表，它给出了函数中每个形参的类型和名字。形参是函数调用时用以传递实参的占位符。从某些方面看，形参就像一个局部变量。不同的是，每一个形参会自动地以实参进行初始化。如果一个函数没有形参，则函数的整个形参列表为空。

60 | 61

函数体是由若干语句及函数所需的若干局部变量的声明构成的完成一定功能的一个程序块。对于有返回值的函数来说，至少需要有一条 return 语句，该语句通常具有如下形式：

```
return expression;
```

执行 return 语句会使得函数当即将控制权返回给它的调用者，同时将表达式的值作为函数值返回。

函数可以返回任何类型的值。例如，下面的函数返回了一个布尔类型的值来表明参数 n 是否是一个偶数：

```
bool isEven(int n) {
   return n % 2 == 0;
}
```

一旦定义了这个函数，就可以在 if 语句中使用它：

```
if (isEven(i))
```

正像在第 1 章里所说的，返回布尔类型数值的函数称为**判定函数**（predicate function）。

然而，函数并不一定需要返回值。不返回值只执行规定步骤的函数称为**过程**（procedure）。以保留字 void 作为函数的返回类型来表明其函数为一个过程。过程通常在执行完函数体中的语句后结束。但是也可以像下面这样使用不带返回值的 return 语句来提前结束一个过程的执行：

```
return;
```

2.3.1 函数原型

当 C++ 编译器在程序中遇到函数调用时，它需要一些函数的相关信息来确保生成正确的程序代码。在大多数情况下，编译器不需要了解函数体所包含的所有语句。它所需要了解的仅是函数所需要的形参及函数的返回值类型。这些信息通常由**函数原型**（prototype）提供，它由函数的第一行后跟一个分号构成。

62

在第 1 章你已经见过函数原型的例子。例如，图 1-3 例子中的 PowersOfTwo 程序展示了 raiseToPower 函数的原型：

```
int raiseToPower(int n, int k);
```

这一函数原型声明告诉编译器 raiseToPower 函数需要两个整型实参并且返回一个整型值。在函数原型中，形参名是可选的，但一个好的形参名将有助于提高程序的可读性。

如果函数是先定义后调用，那就不需要编写函数原型。一些程序员喜欢将源代码组织成低级函数代码在前，调用低级函数的中间函数在中间，主程序代码在后的形式。使用这种编程风格可以让你少写几行代码，但是会让代码的阅读者对程序感到困惑。更糟糕的是，这种组织代码方式与自顶向下的程序设计风格是相悖的，因为他会将更经常使用的函数放在程序的后面。在本书中，为了让我们可以在任何地方定义函数，除主函数以外的每个函数都会提供一个明确的函数原型声明。

2.3.2 重载

在 C++ 中，函数名相同但函数的参数列表不同是合法的。当编译器遇到调用函数的函数名指代不唯一的情况时，编译器会检查调用函数时所传实参并选择最适合的函数版本。几个使用相同名字的不同版本函数称为函数名的**重载**（overloading）。函数的形参模型仅考虑其形参数量及其类型，而不考虑其形参名称，函数的形参模型称为函数**签名**（signature）。

下面举一函数名重载的实例，<cmath>库包含了几个版本的 abs 函数，且每一个对应一种内置的算法类型。例如，该库包含的函数如下：

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

也包含另一个相同名字的函数：

```
double abs(double x) {
   return (x < 0) ? -x : x;
}
```

63

这两个函数唯一的不同在于第一个 abs 函数需要一个 int 类型的实参，而第二个 abs 函数需要一个 double 类型的实参。编译器根据函数调用者在调用函数时所传入的实参类型来决定到底调用那一个 abs 函数。因此，如果 abs 被传入了一个 int 类型的实参，编译器将会执行 int 参数的 abs 函数并返回一个整型值。相反地，如果调用时传入的实参是 double 类型，编译器将选择使用 double 类型形参的 abs 函数。

函数重载最主要的好处就是让程序员可以更好地追踪那些名字相同、拥有相同操作但在应用上有细微差别的函数。在不支持函数重载的 C 语言中，在调用求绝对值函数时，你需要记住调用 fabs 函数来处理 float 类型的数值，调用 abs 函数来处理 int 类型的数值。而在 C++ 中，你只需要记住一个函数名 abs 就可以了。

2.3.3 默认形参数

C++ 可以指定函数中的某些形参具有默认值。形参变量依旧出现在函数的第一行，但函数声明中已赋值的形参可以在函数调用时不给其实参值，这种具有默认值的形参称为**默认形参数**（default parameter）。

为了表明一个函数形参值是可选的，你需要在函数声明的时候为其形参赋予一个初始值。假如你通过设计一组函数来实现一个字处理器，你可能会像下面这样编写一个函数原型：

```
void formatInColumns(int nColumns = 2);
```

formatInColumns 函数获取列数值来作为其实参，但在函数原型声明中的 =2 说明了这一实参有可能在调用的时候被省略。如果你在调用该函数时这样写：

```
formatInColumns();
```

那么实参 nColumns 的值会自动初始化为 2。

当使用默认参数值的时候，需要牢记以下两点：

- 对函数默认值的说明仅能出现在函数声明中，而不能出现在函数的定义中。
- 所有具有默认值的形参声明只能出现在函数形参列表的尾部。

默认形参值的函数在 C++ 中存在过度使用的问题。通常我们更倾向于使用 C++ 中的函数重载来完成与默认形参相同的功能。想象一下，为了定义一个需要传入 x 和 y 的坐标来作为参数的函数 setInitialLocation，那么该函数原型大概是这样的： 64

```
void setInitialLocation(double x, double y);
```

现在假设想修改该函数原型，使得传入的坐标初始位置是（0，0）。一种方法是在函数原型中给参数取默认值：

```
void setInitialLocation(double x = 0, double y = 0);
```

这一函数声明虽然可以达到预期的效果，但是在调用这一函数时，只传入一个参数这一现象会让人迷惑。达成这一目的一个更好方法是像下面这样使用函数的一个重载版本：

```
void setInitialLocation() {
   setInitialLocation(0, 0);
}
```

2.4 函数调用机制

尽管你可以通过直觉来理解函数调用过程，但是详细学习函数调用这一过程可以让你更精确地理解 C++ 中函数调用时后台发生的事，这一知识在理解第 7 章的递归函数时显得尤为有用。本节接下来将介绍函数调用时的细节，并通过设计一个简单的例子来使这一过程更加明朗。

2.4.1 函数调用步骤

当发生函数调用时，C++ 编译器会生成若干代码执行以下操作：

1. 主调函数通过将实参与自己上下文中的局部变量进行绑定来计算每个参数值。由于实参通常为表达式，因此在函数调用时其实参表达式的计算可能涉及操作符及其他函数调用；在新的函数开始执行之前，主调函数会对传入的实参的合法性进行验证。

2. 系统会为新的函数所需的所有局部变量（包括形参）创建新的存储空间。这些变量将被分配在内存中称为**栈帧**（stack frame）的区域中。

3. 每一个实参值被传入到函数相应的形参变量中。对于包含多个形参的函数，这些实参对形参的值拷贝将按对应函数形参的顺序执行：第一个实参被传给第一个形参，以此类推。 65
如果必要，编译器将像变量赋值过程一样执行从实参到函数形参的数据类型转换。举例来说，如果你向一个需要 double 类型形参的函数传入了一个 int 实参值，在发生实参到形参的值拷贝之前，实参整型数将被转换成函数形参的浮点型数值。

4. 执行函数体中的语句，直到遇到 return 语句或者没有多余可执行的语句。

5. 如果函数有返回值，函数体内 return 语句表达式的值将被计算，并作为函数值返回给主调函数。如果必要，编译器将执行数值的类型转换以确保返回值符合被调函数值的类

型要求。例如，如果给返回值类型为 int 的函数返回了一个浮点类型的值，则该返回结果的小数部分将会被截断以形成一个整数返回给主调函数。

6. 为函数调用所创建的线响将被删除。在这一过程中，所有的局部变量将被系统清理掉。

7. 将函数返回值代入到函数调用点位置。

虽然上述函数调用这一过程看起来简单明了，你可能还是需要通过一到两个例子来更深入地理解函数调用。通过阅读下一节的例子将让你领悟这一过程，更近一步说，如果你能使用自己编写的程序来感受这一过程的细节，会有助于理解这一过程。虽然你可以在纸上或者黑板上模拟这一过程，但最好设计一批 3×5 的索引卡片，并通过使用一张卡片来模拟栈帧的工作方式。使用索引卡片模型的优点在于你可以创建索引卡片的栈帧，并更贴近地建模计算机的程序执行过程。在模拟时，如果调用函数就增加一张卡片，如果函数返回就拿走一张卡片。

2.4.2 组合函数

函数调用过程可以很容易地通过特定例子的上下文进行说明。想象一下你有 6 枚硬币，每枚的面值分别是一美分、五美分、十美分、二十五美分、五十美分和一美元。如果你在其中随机选取 2 枚硬币，一共有多少种组合呢？图 2-1 列举出了所有的可能性，答案是 15 种。作为一个计算机科学家，应该思考一个更加普遍的问题：给定一个含有 n 个元素的集合，可以从中得到多少个包含 k 个元素的子集？可以通过如下**组合函数**（combinations function）C(n,k) 来得到答案：

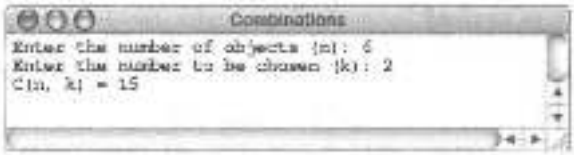
$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

66 其中，感叹号代表了阶乘函数，表明为从 1 到所指定的值中所有整数的乘积。



图 2-1 组合函数示例图

C++ 中计算组合函数的代码如图 2-2 所示，其中主函数从用户那请求 n 和 k 的值，然后显示函数 C(n,k) 的值。该程序的运行实例如下：



正如你在图 2-2 中看到的，Combinations 程序划分为三个函数。主函数 main 实现了与用户的交互界面。combinations 函数计算 C(n,k) 的值。最后，借用第 1 章中定义的 fact 函数来计算所需要的阶乘结果。

```
/*
 * File: Combinations.cpp
 * ----------------------
 * This program computes the mathematical function C(n, k) from
 * its mathematical definition in terms of factorials.
 */

#include <iostream>
using namespace std;

/* Function prototypes */

int combinations(int n, int k);
int fact(int n);

/* Main program */

int main() {
   int n, k;
   cout << "Enter the number of objects (n): ";
   cin >> n;
   cout << "Enter the number to be chosen (k): ";
   cin >> k;
   cout << "C(n, k) = " << combinations(n, k) << endl;
   return 0;
}

/*
 * Function: combinations(n, k)
 * Usage: int nWays = combinations(n, k);
 * --------------------------------------
 * Returns the mathematical combinations function C(n, k), which is
 * the number of ways one can choose k elements from a set of size n.
 */

int combinations(int n, int k) {
   return fact(n) / (fact(k) * fact(n - k));
}

/*
 * Function: fact(n)
 * Usage: int result = fact(n);
 * ----------------------------
 * Returns the factorial of n, which is the product of all the
 * integers between 1 and n, inclusive.
 */

int fact(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

图 2-2 计算组合函数的程序

2.4.3 追踪组合函数执行过程

或许我们会对 Combinations 程序的功能感兴趣，但是下面的例子的目的在于说明函数执行所包含的步骤。在 C++ 中，所有的程序都从主函数 main 开始执行。为了实现函数调用，系统（包括所使用的操作系统和硬件）会创建一个栈帧来保持对函数定义的局部变量的跟踪。在 Combinations 程序中，main 函数定义了两个变量，n 和 k，因此栈帧中将包含存储这两个变量的空间。

在本书的所有图表中，我们使用双划线围成的矩形来表示栈帧。每一个栈帧图中将显示一段程序源代码和一个表示程序代码执行位置的手指图标，以利于对程序执行点的跟踪。在栈帧中也包含了长方块来标记局部变量。因此，在开始执行 main 函数之前的栈帧如下图所示：

```
int main() {
   int n, k;
   cout << "Enter the number of objects (n): ";
   cin >> n;
   cout << "Enter the number to be chosen (k): ";
   cin >> k;
   cout << "C(n, k) = " << combinations(n, k) << endl;
   return 0;
}
```

k n

从 main 函数开始，系统依次执行语句，在控制台上输出提示，读取用户输入的数据，并在栈帧中以变量存储这些数据。如果用户输入了像上面示例一样的数值，那么当程序执行到如下语句时栈帧的状态如下图所示：

```
int main() {
   int n, k;
   cout << "Enter the number of objects (n): ";
   cin >> n;
   cout << "Enter the number to be chosen (k): ";
   cin >> k;
   cout << "C(n, k) = " << combinations(n, k) << endl;
   return 0;
}
```

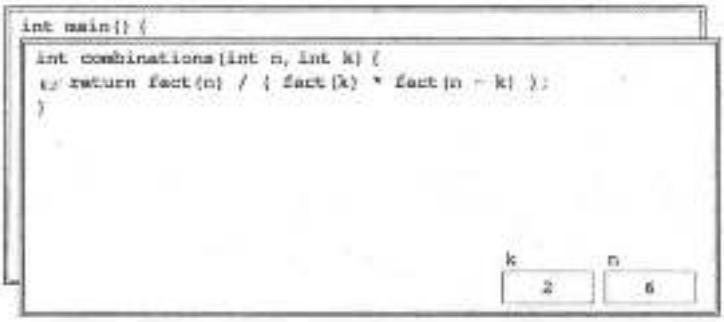
k n

2 6

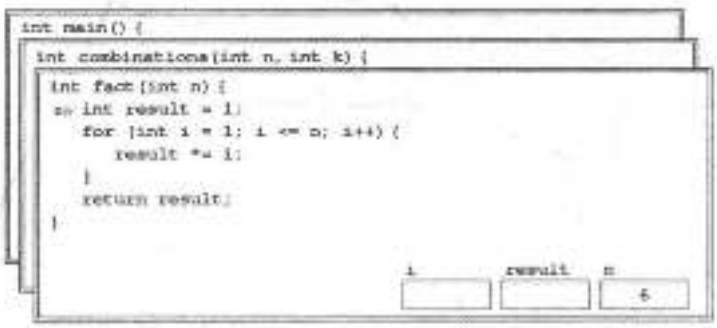
69

在程序输出执行结果之前，它必须调用函数 combinations(n,k)。此时，称 main 函数将调用 combinations 函数，这也意味着计算机需要先执行完调用函数所需的所有步骤再输出。

函数调用执行的第一步是计算当前栈帧中参数的值。变量 n 的值是 6，k 的值是 2。当计算机建立 combinations 函数的栈帧时，这些参数值将被传递给函数的形参 n 和 k。虽然 main 函数局部变量在这时候无法访问，但新的栈帧将建立在储存了 main 函数局部变量参数值的栈帧的上部。建立函数并初始化形参变量后的情形如下：

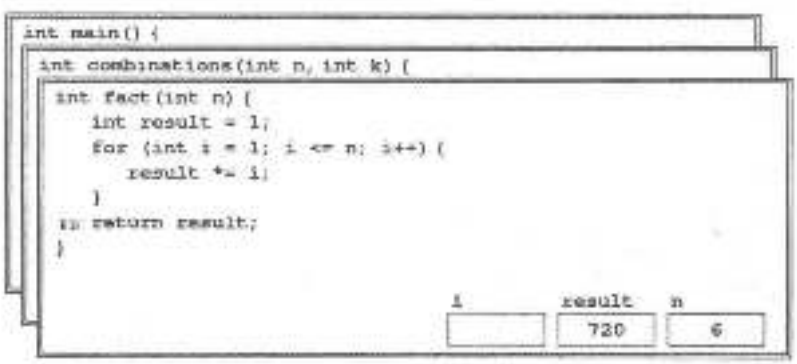


为了计算 combinations 函数的值，程序必须三次调用 fact 函数。在 C++ 中，三次 fact 函数的调用可以以任意顺序执行，但为了简单起见，我们从左到右依次调用 fact 函数。因此，第一次是调用 fact(n)，为了计算其函数值，系统必须创建另外一个栈帧，以下是向 fact 函数中传入数值 6 的情形：



不像之前的栈帧，fact 函数的栈帧中同时包含了形参和局部变量。形参 n 通过调用时传入的数值 6 被初始化。而函数中的两个局部变量 i 和 result 此时还没有被初始化。尽管如此，系统还是在栈帧中为它们保留了存储空间。直到给它们赋值之前，它们将包含之前赋予其中的不可预测的值——随机数。因此，必须牢记在使用局部变量前应对它们进行初始化，当然，最理想的情况是在声明时对它们进行初始化。 70

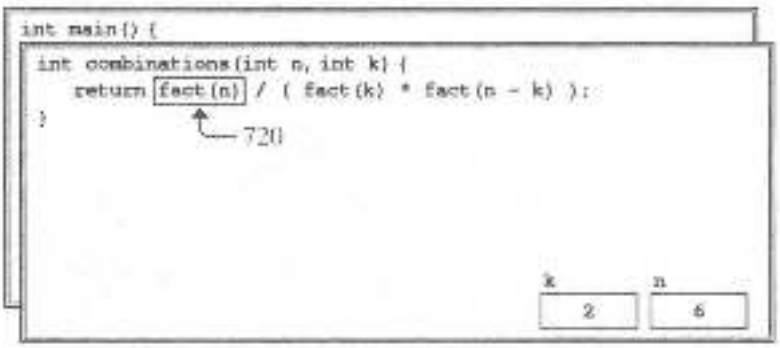
接下来系统会执行 fact 函数中的语句。在这一实例中，for 循环体被执行了 6 次。在每一次循环中，变量 result 的值都乘以循环变量 i，这意味着 result 变量最后的值是 720（1×2×3×4×5×6，即 6！）。当程序执行到 return 语句时，栈帧状态如下所示：



在上述示意图中，代表变量 i 的方框内是空的，因为 i 的值还没有定义。在 C++ 中，循环

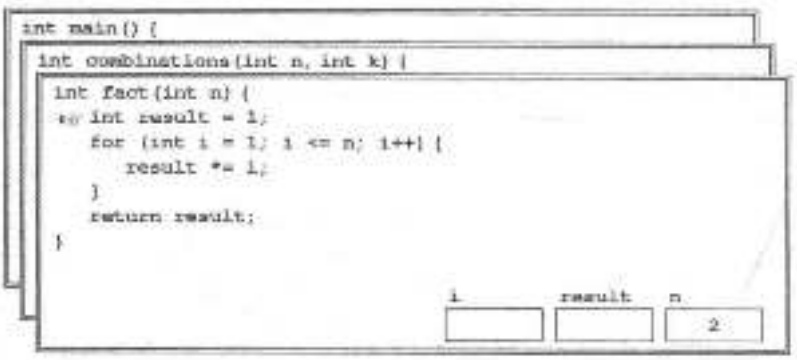
变量在 for 语句中定义，它也只在 for 语句体中有效。以空方框来代表 i 强调了 i 此时还无法访问。

从一个函数返回涉及将 return 表达式的值（即局部变量 result 的值）传回函数调用点这一操作。之后，fact 函数的栈帧被删除，这将导致函数进入以下状态：

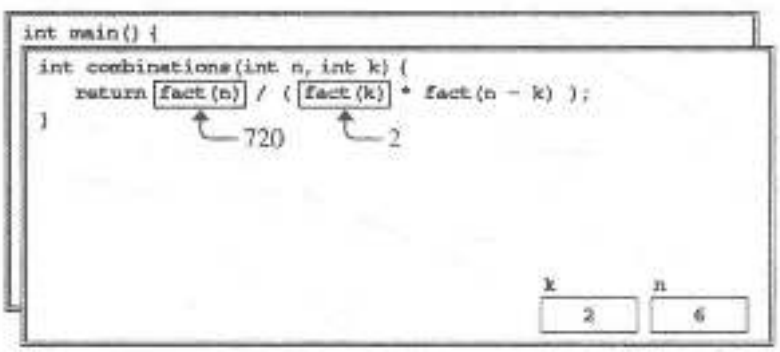


71

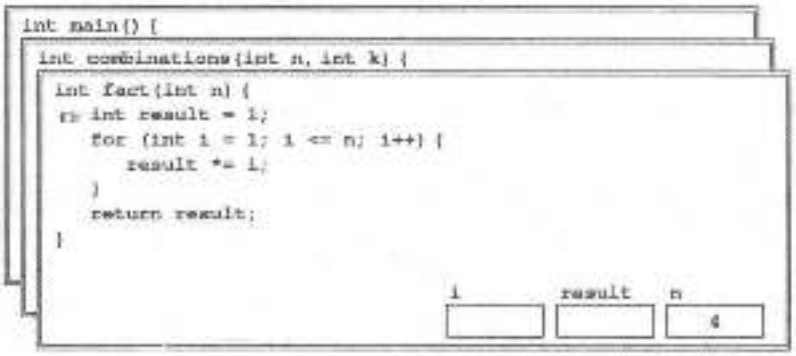
这个过程的下一步是进行 fact 函数的第二次调用，此时首先计算参数 k，然后在调用时用 k 的值 2 初始化新栈帧中的参数 n。其示意图如下所示：



fact(2) 的计算比之前介绍的 fact(6) 的计算要简单许多。在执行完成时，result 变量的值是 2，且这一数值像下面一样会被传回函数调用点：

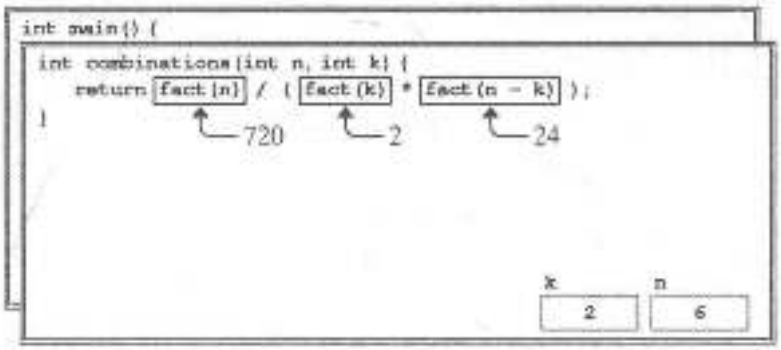


combinations 函数会第三次调用 fact 函数，这一次其函数参数为 n-k。像之前一样，系统会为本次函数调用创建包含值为 4 的变量 n 的栈帧：

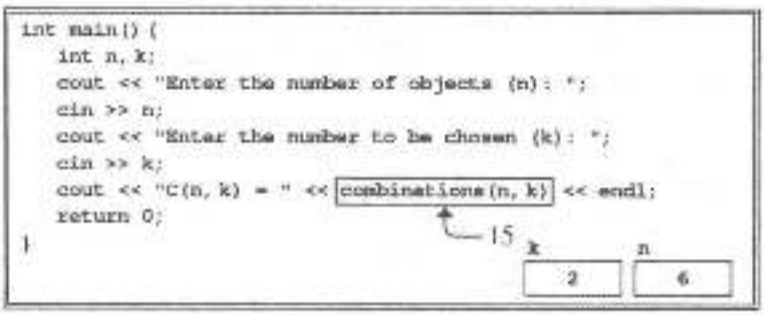


72

fact(4) 的值是 1×2×3×4，即 24。但本次调用返回时，系统便可以填写该程序所需的全部计算值：



计算机接下来会将 720 除以 2 和 24 的乘积，得到结果 15。这一数值会被返回到 main 函数中，使得 main 函数处于如下状态：



从此刻开始，所要做的事就只剩下处理输出，并从 main 函数返回以结束程序的执行过程。

2.5 引用参数

在 C++ 中，当你从一个函数向另一个函数传递一个普通变量时，被调函数将获得一个调用函数的值拷贝。被调函数中所传入的实参变量值仅改变被调用函数局部形参的值，但对主调函数中实参变量的值将不会有任何影响。例如，如果你执行以下代码，将一个变量值初始化为 0：

```
void setToZero(int var) {
   var = 0;
}
```



[73]

但是如果你如下调用函数，这段代码的执行将不会对变量 x 的值有丝毫影响：

```
setToZero(x);
```

不管 x 存储的值为多少，都将使用 x 的值拷贝来初始化参数 var。以下赋值语句：

```
var = 0;
```

将把函数中局部变量值设置为 0，但不会改变主调用程序中 x 的值。

如果你想要改变主调函数中实参的值（实际中也经常需要这么做），你可以将形参类型从普通的 C++ **值参数**（value parameter）改变为**引用参数**（reference parameter）。引用参数是在参数类型与参数名中间加一个“&”。与值参数不同，引用参数在函数调用时并不是实参值对形参的值拷贝，而实际上形参接受的是对实参的一个引用，这也意味着主调函数和被调函数是共享实参变量的统一存储空间的。setToZero 函数的改进版本如下：

```
void setToZero(int & var) {
   var = 0;
}
```

上述函数调用的参数传递风格被称为引用调用（call by reference）。当采用引用调用时，对应于引用形参的实参必须为可赋值的量，例如变量名。虽然调用 setToZero(x) 能正确地把 x 的值置为 0，但是像 setToZero(3) 这样的调用却是非法的，因为 3 是不能被引用赋值的。

在 C++ 中，最常使用引用调用的情况是函数需要返回多于一个值的情况。我们可以很容易地将单个值作为函数的值返回。但如果你试图从函数中返回多个值，返回值的方法将不再适用。最标准的解决方法是将函数转换为一个过程并通过实参列表向函数传递及从函数返回数值。

例如，假设你想编写一个算一元二次方程式的程序：

$$ax^2 + bx + c = 0$$

[74] 为了有一个好的编程风格，你可把这个程序结构按以下流程图设计为三部分：



图 2-3 所示的 Quadratic 程序向我们展示引用调用是如何将一个计算二次方程的程序分解为这种形式的。图 2-3 中所示的每个函数都各自对应着流程图中的一个阶段。主程序使用值参向被调函数传递信息。当被调函数向主程序返回值时，该函数将返回一个引用。solveQuadratic 函数分别使用了值参和引用两种类型的形参。其中，形参 a、b 和 c 传

入的是值参，代表了二次方程的三个系数，而形参 x1 和 x2 是引用参数用以输出函数的返回值，即允许程序传回二次方程的两个根。

Quadratic 程序还引入了一种新的报告错误的方法。一旦程序的输入无法使程序正确执行，程序将调用 error 函数在输出设备上打印出问题的详细信息并终止程序的执行。error 函数的代码如下所示：

```
void error(string msg) {
   cerr << msg << endl;
   exit(EXIT_FAILURE);
}
```

error 函数的代码使用了本书还未曾列出的两个 C++ 的新特性：cerr 输出流和 exit 函数。cerr 输出流与 cout 相似，但被用以报告错误。exit 函数可以即刻终止主程序的执行，使用形参值来报告程序现在的状态。常量 EXIT_FAILURE 在 <cstdlib> 库中被定义，用来说明发生了某种类型的错误。 75

```
/*
 * File: Quadratic.cpp
 * -------------------
 * This program finds roots of the quadratic equation
 *
 *         2
 *     a x   + b x + c = 0
 *
 * If a is 0 or if the equation has no real roots, the
 * program prints an error message and exits.
 */

#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;

/* Function prototypes */

void getCoefficients(double & a, double & b, double & c);
void solveQuadratic(double a, double b, double c,
                    double & x1, double & x2);
void printRoots(double x1, double x2);
void error(string msg);

/* Main program */

int main() {
   double a, b, c, r1, r2;
   getCoefficients(a, b, c);
   solveQuadratic(a, b, c, r1, r2);
   printRoots(r1, r2);
   return 0;
}

/*
 * Function: getCoefficients
 * Usage: getCoefficients(a, b, c);
 * --------------------------------
 * Reads the coefficients of a quadratic equation into the
 * reference parameters a, b, and c.
 */

void getCoefficients(double & a, double & b, double & c) {
   cout << "Enter coefficients for the quadratic equation:" << endl;
   cout << "a: ";
   cin >> a;
   cout << "b: ";
   cin >> b;
   cout << "c: ";
   cin >> c;
}
```

图 2-3 解二次方程的程序 76

```
/*
 * Function: solveQuadratic
 * Usage: solveQuadratic(a, b, c, x1, x2);
 * ---------------------------------------
 * Solves a quadratic equation for the coefficients a, b, and c.  The
 * roots are returned in the reference parameters x1 and x2.
 */

void solveQuadratic(double a, double b, double c,
                    double & x1, double & x2) {
   if (a == 0) error("The coefficient a must be nonzero.");
   double disc = b * b - 4 * a * c;
   if (disc < 0) error("This equation has no real roots.");
   double sqrtDisc = sqrt(disc);
   x1 = (-b + sqrtDisc) / (2 * a);
   x2 = (-b - sqrtDisc) / (2 * a);
}

/*
 * Function: printRoots
 * Usage: printRoots(x1, x2);
 * --------------------------
 * Displays x1 and x2, which are the roots of the quadratic equation.
 */

void printRoots(double x1, double x2) {
   if (x1 == x2) {
      cout << "There is a double root at " << x1 << endl;
   } else {
      cout << "The roots are " << x1 << " and " << x2 << endl;
   }
}

/*
 * Function: error
 * Usage: error(msg);
 * ------------------
 * Writes the string msg to the cerr stream and then exits the program
 * with a standard status value indicating that a failure has occurred.
 */

void error(string msg) {
   cerr << msg << endl;
   exit(EXIT_FAILURE);
}
```

图2-3 （续）

当然，error函数在除了解二次方程程序之外的其他程序中也能起到重要作用。虽然你可以很容易地将代码复制到另一个程序，但如果将error函数写入库中会更加方便。在
[77] 下面的章节中，你将有机会尝试这一做法。

2.6 接口与实现

当你定义一个C++库时，你需要提供库的两个部分。首先，你必须定义接口（interface），它可以让库用户在不了解库实现细节的情况下使用库中的库函数。第二，你需要定义库的具体实现（implementation），它说明库的底层实现细节。一个典型的接口可以提供多种定义，包括函数定义、类型定义和常量定义。每一个定义都称为一个接口条目（interface entry）。

在C++中，接口和实现通常写在两个不同的文件中。定义接口的文件后缀名是.h，也称为头文件。定义实现的文件取同样的名字，但其后缀名为.cpp。根据这些原则，error库将在error.h中定义，并在error.cpp中实现。

2.6.1 定义 error 库

图 2-4 中的程序给出了 error.h 库接口的内容。就像你所看到的，error.h 中包含了大量的注释。在其接口部分只包括了 error 函数的函数原型还有三行称为接口模板（interface boilerplate）的内容，即函数预编译头。这些预编译头在每个库接口中都会出现。预编译头包括了开始的 #ifndef 和 #define 指令，以及在结尾处对应的 #endif 指令。这些代码行确保了编译器不会重复编译同样的接口。#ifndef 指令检查 _error_h 标志是否已被定义。当编译器第一次读取这一语句时，检查的结果将是假的。然而在下一行，将定义这一标志。因此，如果编译器在接下来有访问同一接口的动作，此时的 _error_h 标志已经被定义，这一次编译器将跳过接口的定义。在本书中，#ifndef 行的标志通常由下划线开始，接下来是接口文件名，文件名其中的点将被下划线代替。

```
/*
 * File: error.h
 * -------------
 * This file defines a simple function for reporting errors.
 */

#ifndef _error_h
#define _error_h

/*
 * Function: error
 * Usage: error(msg);
 * ------------------
 * Writes the string msg to the cerr stream and then exits the program
 * with a standard status code indicating failure.  The usual pattern for
 * using error is to enclose the call to error inside an if statement that
 * checks for a particular condition, which might look something like this:
 *
 *     if (divisor == 0) error("Division by zero");
 */

void error(std::string msg);

#endif
```

图 2-4 error 库接口

然而，error 的函数原型定义与普通的函数定义有细微不同。在 error 函数的形参声明时，使用了类型名 std::string 来说明 string 类型来自于命名空间 std。接口一般来说是在代码行 using namespace std 之前读取的，因此如果不使用 std:: 修饰符来明确表示，我们将不能使用命名空间中的标识符。

图 2-5 是 error.cpp 的实现。该实现文件中的注释用于程序员维护库时使用。通常这种注释也比接口文件中的注释更少。在这里，error 函数体仅有两行，每一行的作用对于任何 C++ 程序员来说都可谓一目了然。

```
/*
 * File: error.cpp
 * ---------------
 * This file implements the error.h interface.
 */

#include <iostream>
#include <cstdlib>
#include <string>
#include "error.h"
using namespace std;

/*
```

图 2-5 error 库的实现

```
 * Implementation notes: error
 * ---------------------------
 * This function writes out the error message to the cerr stream and
 * then exits the program.  The EXIT_FAILURE constant is defined in
 * <cstdlib> to represent a standard failure code.
 */

void error(string msg) {
   cerr << msg << endl;
   exit(EXIT_FAILURE);
}
```

图 2-5 （续）

2.6.2 导出数据类型

前面章节介绍的 error.h 接口中只有一个函数。实际上，C++ 中大多数的接口中还会存在多个数据类型。这些类型大多数为类，而类是 C++ 提供的面向对象编程的基础。在第 6 章之前，你不会学到如何定义自己的类，所以在这里举有关类的例子还为时过早。为了便于理解，我们通过建立一个接口输出第 1 章介绍过的枚举类型，正如下面的 Direction 类型可以被用来编码指南针的四个方向：

```
enum Direction { NORTH, EAST, SOUTH, WEST };
```

实现这一数据类型的最简单方法就是定义一个只包含这一数据类型定义和预编译头的 direction.h 接口。如果你采用了这一方法，则无须提供这一接口的实现。

然而，如果这个接口输出一些简单的函数来操作 Direction 值，将会使得这个接口更好用。例如，输出 1.8.4 节定义的 directionToString 函数将是一个很好的选择，这个函数返回了一个枚举类型的值以代表方向。在本书接下来的一些程序里，定义的函数 leftFrom 和 rightFrom 将非常有用，它们可使 Direction 值的方向从特定方向旋转 90 度。例如，调用 leftFrom(NORTH) 将返回 WEST。如果你将上述函数加入到 direction.h 接口中，则必须提供 direction.cpp 文件来实现这些函数。图 2-6 和图 2-7 列出了这些文件。

函数 leftFrom 和 rightFrom 的实现有一些微妙的步骤需要我们对此进行进一步讲解。虽然 C++ 允许将枚举类型自动转换为整数类型，但是从整数类型转换为枚举类型时需要类型转换。这种类型转换可以通过 rightFrom 的实现来说明：

78
:
80

```
Direction rightFrom(Direction dir) {
   return Direction((dir + 1) % 4);
}
```

```
/*
 * File: direction.h
 * -----------------
 * This interface exports an enumerated type called Direction whose
 * elements are the four compass points: NORTH, EAST, SOUTH, and WEST.
 */

#ifndef _direction_h
#define _direction_h

#include <string>

/*
 * Type: Direction
```

图 2-6 direction 库的接口

```
 * ----------------
 * This enumerated type is used to represent the four compass directions.
 */

enum Direction { NORTH, EAST, SOUTH, WEST };

/*
 * Function: leftFrom
 * Usage: Direction newdir = leftFrom(dir);
 * ----------------------------------------
 * Returns the direction that is to the left of the argument.
 * For example, leftFrom(NORTH) returns WEST.
 */

Direction leftFrom(Direction dir);

/*
 * Function: rightFrom
 * Usage: Direction newdir = rightFrom(dir);
 * -----------------------------------------
 * Returns the direction that is to the right of the argument.
 * For example, rightFrom(NORTH) returns EAST.
 */

Direction rightFrom(Direction dir);

/*
 * Function: directionToString
 * Usage: string str = directionToString(dir);
 * -------------------------------------------
 * Returns the name of the direction as a string.
 */

std::string directionToString(Direction dir);

#endif
```

图 2-6 （续）

81

```
/*
 * File: direction.cpp
 * -------------------
 * This file implements the direction.h interface.
 */

#include <string>
#include "direction.h"
using namespace std;

/*
 * Implementation notes: leftFrom, rightFrom
 * -----------------------------------------
 * These functions use the remainder operator to cycle through the
 * internal values of the enumeration type.  Note that the leftFrom
 * function cannot subtract 1 from the direction because the result
 * might then be negative; adding 3 achieves the same effect but
 * ensures that the values remain positive.
 */

Direction leftFrom(Direction dir) {
   return Direction((dir + 3) % 4);
}

Direction rightFrom(Direction dir) {
   return Direction((dir + 1) % 4);
}

/*
 * Implementation notes: directionToString
 * ---------------------------------------
 * Most C++ compilers require the default clause to make sure that this
 * function always returns a string, even if the direction is not one
 * of the legal values.
 */
```

图 2-7 direction 库的实现

```
string directionToString(Direction dir) {
   switch (dir) {
    case NORTH: return "NORTH";
    case EAST: return "EAST";
    case SOUTH: return "SOUTH";
    case WEST: return "WEST";
    default: return "???";
   }
}
```

图2-7 (续)

在以下算术运算表达式中：

```
(dir + 1) % 4
```

[82] 操作符将自动把 Direction 值转换到对应方向的整型值：0代表 NORTH，1代表 EAST，2代表 SOUTH，3代表 WEST。从其中一个方向向右旋转方向需要将其值加一，但从 WEST 向右旋转时例外，需要将方向值循环到代表北的数值0。在循环类型的数据结构中，这是经常发生的事，因此经常使用求余符号来代替特殊情况对函数的测试。然而，在函数返回值之前，还需要将表达式求得的值转换成 rightFrom 函数定义的返回类型 Direction。

因为存在使用求余符号计算负数的危险，所以 leftFrom 函数不能像下面这样定义：

```
Direction leftFrom(Direction dir) {
   return Direction((dir - 1) % 4);
}
```



当你调用 leftFrom(NORTH) 时，上述函数将出现错误，但当变量 dir 的值是 NORTH 时，以下表达式：

```
(dir - 1) % 4
```

在大多数机器上将得到值-1，这不是 Direction 类型的一个合法值，幸运的是，我们可以通过以下定义的 leftFrom 函数来解决这一问题：

```
Direction leftFrom(Direction dir) {
   return Direction((dir + 3) % 4);
}
```

2.6.3 导出常量定义

接口除了可导出函数定义和类型定义，还经常可导出常量定义，这样做可以使多个用户共享这一常量而不用在每个文件中重新定义它。例如，当你编写有关几何计算的程序时，定义数学常量是非常有用的，按照约定俗成的习惯，通常会将常量名写为 PI。如果你像第1章那样声明常量 PI，则可写成：

```
const double PI = 3.14159265358979323846;
```

在C++中，像这样定义的常量在源文件中是私有的常量，它不能从接口中导出。为了从接口中能导出常量 PI，需要在其接口的常量定义声明和原型声明中都加上关键字 extern。如图2-8所示的 gmath.h 接口，这一接口导出 PI 和一些简单的函数来处理以度表示的角度。其对应的实现如图2-9所示。

[83]

```
/*
 * File: gmath.h
 * -------------
 * This file exports the constant PI along with a few degree-based
 * trigonometric functions, which are typically easier to use than
 * their radian-based counterparts in <cmath>.
 */

#ifndef _gmath_h
#define _gmath_h

/* Constants */

extern const double PI;          /* The mathematical constant pi */

/*
 * Function: sinDegrees
 * Usage: double sine = sinDegrees(angle);
 * ---------------------------------------
 * Returns the trigonometric sine of angle expressed in degrees.
 */

double sinDegrees(double angle);

/*
 * Function: cosDegrees
 * Usage: double cosine = cosDegrees(angle);
 * -----------------------------------------
 * Returns the trigonometric cosine of angle expressed in degrees.
 */

double cosDegrees(double angle);

/*
 * Function: toDegrees
 * Usage: double degrees = toDegrees(radians);
 * -------------------------------------------
 * Converts an angle from radians to degrees.
 */

double toDegrees(double radians);

/*
 * Function: toRadians
 * Usage: double radians = toRadians(degrees);
 * -------------------------------------------
 * Converts an angle from degrees to radians.
 */

double toRadians(double degrees);

#endif
```

图 2-8 gmath 库简化的接口

84

```
/*
 * File: gmath.cpp
 * ---------------
 * This file implements the gmath.h interface.  In all cases, the
 * implementation for each function requires only one line of code,
 * which makes detailed documentation unnecessary.
 */

#include <cmath>

#include "gmath.h"
extern const double PI = 3.14159265358979323846;

double sinDegrees(double angle) {
   return sin(toRadians(angle));
```

图 2-9 gmath 库的实现

```
}

double cosDegrees(double angle) {
   return cos(toRadians(angle));
}

double toDegrees(double radians) {
   return radians * 180 / PI;
}

double toRadians(double degrees) {
   return degrees * PI / 180;
}
```

图2-9 （续）

2.7 接口设计原则

程序设计的一个难点是程序需要考虑到底层应用的复杂性。随着计算机处理的问题越来越复杂，程序也会变得越来越复杂难懂。

编写一个处理大型或复杂问题的程序将迫使你面对越来越惊人的程序复杂性的增长。有许多算法需设计，许多特殊情况需处理，用户的需求需满足，以及大量细节问题需要兼顾。为了提高程序的可管理性，你必须尽你所能地降低程序的复杂性。函数可以降低一部分程序复杂度；类库也提供了同样可用于降低复杂性的方法，但同时也需要你在建库时考虑更多的细节。函数向调用者提供了能完成特定功能的一组操作集。类库向用户提供了一组函数和数据类型，以实现计算机科学家所描述的**编程抽象**（programming abstraction）。一个特定的抽
85 象能多大程度地简化程序取决于你可以多好地设计接口。

为了设计一个高效的接口，你必须平衡多个方面的需求。通常，你应该以下面的准则来尝试构建接口：

- 统一性（unified）。一个接口必须依照一个统一的主题来定义一致的抽象。如果某个函数不符合这一主题，就必须使用其他方法来重新定义这一函数。
- 简单性（simple）。库的底层实现是复杂的，但是库的接口必须向用户隐藏这种复杂性。
- 充分性（sufficient）。当用户使用一种抽象时，接口必须提供足够的功能来满足用户的需求。如果接口缺少一些关键功能，用户可能会拒绝使用并自己开发更好的库。与简单性同样重要，库的设计者必须避免在简单化过程中使得库的功能大大减少。
- 通用性（general）。一个良好设计的接口必须有高度的适用性，它可以满足不同用户的需求。一个只为特定用户提供的小范围操作集的库，其可用性将大大低于一个可以在多种情况下使用的库。
- 稳定性（stable）。接口中定义的函数在底层实现改变时，也必须拥有一丝不变的函数接口结构和函数功能。函数功能的改变将迫使用户向接口的变化要求并修改程序。

下面，将详细讨论上述的接口设计准则。

2.7.1 统一主题的重要性

团结就是力量。

——伊索，《一捆柴枝》，一公元前600年

设计良好的接口所应具备的核心特征是该接口体现了一个统一的、一致的抽象。从

某种程度上说，这一库设计标准反映了库中提供的函数必须拥有一个一致的主题。因此，<cmath>库提供了数学函数，<iostream>库提供了cin、cout和cerr数据流以及执行输入输出操作符，2.6节介绍的error.h接口提供了一个报告错误的函数。这些库提供的每一个接口入口都符合接口的主题。例如，你不会让<string>库提供sqrt函数，因为sqrt函数更加符合<cmath>库的主题框架。

统一主题原理也对库接口中的函数设计产生影响。在一个接口中的函数应该尽可能在功能上表现出一致性。例如，<cmath>库中函数处理的角度值用弧度表示。如果有一些函数使用度数表示角度值，用户就不得不去记住每个函数要使用的参数的单位。 86

2.7.2 简单性与信息隐藏原理

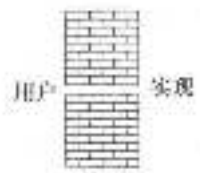
见素抱朴。

——老子，《老子》，一公元前550年

使用接口的首要目标就是降低编程的复杂性，因此很容易理解，简单性也是接口设计最追求的目标。通常来说，一个接口应该尽可能降低使用难度。虽然底层实现的操作可能极度错综复杂，但是应该让用户可以从一个更简单、更抽象的角度去了解这些操作。

在某种程度上，接口扮演着一个特定库抽象参考指南的角色。当你想要知道怎样使用error函数的时候，你只需要查阅error.h接口来了解使用方法。接口包含了你所需要的全部信息。对于用户来说，知道太多和太少的信息同样都是不利的，因为多余的细节信息将使界面变得更难以理解。通常，接口所含的数据信息应隐藏在接口中而不应对外暴露。

当你设计一个接口时，你应该尽可能地使用户远离其实现的复杂细节。从这个角度上讲，最好的办法是将接口设计成一堵分隔用户和实现的墙，而不是这两个部分间的通信渠道。



像希腊神话中分隔皮拉莫斯和提斯伯的墙一样，这堵墙在用户和实现之间提供了一道缝隙来使双方进行沟通。在编程时，这一缝隙包含了允许用户和实现之间进行沟通的界面入口。但是这堵墙的主要目标是保持用户和实现的分离。由于我们总是将它看成代表库的抽象的一道边界，所以接口有时也称作抽象边界（abstraction boundary）。理想情况下，库中所有复杂的部分都会被隔离在实现的一边。一个成功的接口将使用户远离那些复杂的实现。保持实现细节被限制在库的实现中又被称为信息隐藏（information hiding）。

信息隐藏原理在接口的设计上具有重要的现实意义。当你编写一个接口时，应该确保没有公开其实现的细节，包括注释部分，特别是当你同时编写接口和实现程序的时候，你可能会很不情愿要在接口中介绍你在实现库时用到的巧妙方法。试着放弃这种想法，接口的目标 87
是使用户感到方便，应该只包含用户需要知道的信息。

同样，你在接口中应该尽可能将函数设计得足够简单。如果你可以减少函数参数的数量，或者有减少函数发生特殊情况的方法，都将使函数变得更加简单易用。除此之外，限制一个接口提供的函数数量也是有利的，这样做可以使得用户不会在大量的函数中迷失自己而

缺乏对整个接口的了解。

2.7.3 满足用户需求

每一件事都应尽可能的简单，但不能过度。

——阿尔伯特·爱因斯坦

简单易用只是设计库要求的一部分。为了使库变得简单，最容易的做法就是删除库中复杂和难懂的部分。但这么做也会使你设计的库的可用性降低。有时用户需要执行一些具有内在复杂性的任务。所以为了接口的简单而拒绝用户的需求不是一个好想法。为了更好地服务于用户，你的库必须提供充足的函数。学会在设计库的时候平衡好简单性和完整性是一项重要而艰巨的任务。

在许多情况下，接口的用户考虑的不只是库是否提供了某项功能，他们同时也在考虑这项功能的实现是否是高效的。例如，如果你正在开发一个空中交通管理系统，并且该系统需要使用库中的函数，你会要求这些函数必须快速返回正确的数值。过慢的应答与错误的返回值都将酿成灾难性的后果。

在大多数时候，程序执行效率更多的与库的实现有关，而与接口无关。即使这样，你也会经常发现在设计接口时考虑实现策略是很有意义的。假如让你在两种设计中做出选择，其中一种设计的实现更加简单高效，而另一种并没有让你有不得不选的其他原因，那毫无疑问你会选择简单高效的设计。

2.7.4 通用工具的优势

给我们工具，我们将完成任务。

——温斯顿·丘吉尔，1941年BBC广播演讲

88

一个完全适合某种特定类型用户的接口，并不一定适合其他类型的用户。一个好的库抽象必须服务于多种类型用户的需求。为了达到此目的，我们必须在设计的时候提高库的通用性，以使得它可以解决多元化的问题而不是受限于某个非常特殊的应用情况。为了选择一种灵活性强的设计，你可以创建多用途的接口。

确保接口的通用性具有重要的现实意义。当你编写一个程序时，会经常发现你需要一个特殊的工具。如果你认为这个工具非常重要以至于需要把它放入到库中，那么你就需要转换你的思维模式了。当你为那个库设计接口的时候，你必须遗忘这个库最初的应用环境，而应为更通用的受众设计你的接口。

2.7.5 库稳定性的价值

人们改变了，却忘了彼此告知。太糟糕了——这导致了许多错误。

——莉莉安·海尔曼，《阁楼里的玩具》，1959

接口的另一特性使得它们在编程中具有关键的作用：它们可以在很长一段时间中保持自身的稳定。一个稳定的接口通过创建清晰的责任边界可以大大简化系统的维护。在库接口不变的情况下，实现者和用户都可以相对自由地改变自身负责部分的代码。

例如，想象一下你是<cmath>库的实现者。在你工作的时候，发现了一个实现sqrt函数的更精妙方法，这个方法可以使得计算平方根的时间缩小到一半。如果可以向用户告知你有一个sqrt函数的更好的实现方法，这个方法比以往更快，他们会非常高兴。但从另一

方面来说，如果你告诉用户函数的名称将改变或者说函数具有新的约束，那么用户会感到愤怒。为了使用你提供的“改进版”平方根函数，他们需要被迫修改自己的程序。修改程序是一项耗费时间、易于出错的活动，很多用户宁愿放弃程序的一部分性能提升，也不愿意他们的程序经修改后而无法运行。

接口只有在保持稳定的情况下才可简化程序的维护。当出现新算法或者应用需求变化时，程序经常被修改。然而在程序的这种进化中，其接口应尽可能地保持不变。在一个良好设计的系统中，修改实现是一个直截了当的过程。修改所涉及的复杂性只限定于抽象边界的实现部分。另外，修改接口通常会导致依赖于该接口的所有程序都必须进行修改。因此，修 89
改接口应是非常罕见的行为，这一行为应仅在用户参与时进行。

某些接口的修改造成的影响将会比其他接口的修改造成的影响更大。例如，在库中添加一个全新的函数是一个直截了当的工作，因为并没有用户的程序使用过这个函数。像这样接口修改使得以前使用该库的程序不用进行修改的行为称为接口的扩展（extending）。如果你发现在一个接口的生命周期中需要对该接口进行一次升级，最好的办法是通过扩展来达到此目的。

2.8 随机数库的设计

领会接口设计原则最容易的方法就是进行一次简单的设计实践。为达到此目的，本节将讲述开发 Stanford 类库中 random.h 接口的每个设计过程，它使编写像出租随机选择的程序成为可能。模拟随机过程是必须的，例如，如果你想要编写视频扑克或者掷骰子的电脑游戏，你就需要模拟随机过程，当然，这一过程在实际应用中还有着更重要的作用。模拟随机事件的程序称作不确定性的（nondeterministic）程序。

让计算机表现出随机选择的行为是比较复杂的。为了给用户程序提供便利，你必须隐藏接口背后的复杂过程。本节你将有机会从接口设计者、接口实现者和用户这几个不同的角度来了解这个接口。

2.8.1 随机数与伪随机数

使用计算机生成随机过程的概念经常被描述为生成特定范围内的一个随机数（random number），部分原因是早期计算机主要用于处理数值应用。从理论上讲，一个无法预测其值，且在其取值范围内其值等概率出现的数被称为随机数。例如，掷骰子时出现一个范围从 1 到 6 的随机数。如果骰子是正常的，我们将无法准确预测掷出的数字，并且六个值出现的概率均等。

虽然随机数的概念看起来如此直观，但是要将这一概念在计算机上实现是有困难的。因为计算机总是执行内存中一系列特定序列的指令，因此计算机的函数都是遵循确定的模式的。怎么让遵循确定序列指令的计算机产生不可预测的结果呢？如果一个数是一个确定过程 90
的结果，那么，任何用户都应该能通过一组相同的指令来预测出计算机的响应。

事实上，计算机正是通过使用特定的过程来产生所谓的随机数。这一策略之所以可行，是因为理论上用户可以通过同样的一组指令预测出计算机的结果，但实际上并没有人会无聊到去做这种事。在大多数现实应用中，产生的数字是否真正是随机的关系并不大，真正有影响的是该数字看起来是不是随机的。为了使产生的数字看起来是随机的，它们必须具备如下特点：（1）从统计的观点来看，该数表现的象一个随机数；（2）事先要预测这个数的值应该很难，以至于没有人想去预测它。通过计算机中的一个特定算法来产生的“随机数”也被称

作伪随机数（pseudorandom number），这一名称也强调了在该数的产生过程中并不涉及真正的随机活动。

2.8.2 标准库中的伪随机数

<cstdlib>类库提供了一个低级的函数 rand，调用该函数可以产生伪随机数。rand 函数的函数原型为：

```
int rand();
```

此函数原型表明 rand 函数无须传入参数，并且函数返回一个整型值。每一次对 rand 的调用将产生一个让用户难以预测其值的不同的随机数。rand 函数的结果是一个大于等于零且小于等于常量 RAND_MAX 的整数，而 RAND_MAX 被定义在 <cstdlib> 中。因此，每一次调用 rand 函数，这个函数将返回从 0 到 RAND_MAX 之间的任意一个整数。

如果你想要更深刻地体会 rand 函数的工作过程，一个方法就是编写一个简单的程序来测试它。图 2-10 的 RandTest 程序显示了 RAND_MAX 的值，然后打印出 10 次调用 rand 函数的结果。程序运行的实例如下：




```
/*
 * File: RandTest.cpp
 * ------------------
 * This program tests the random number generator in C++ and produces
 * the values used in the examples in the text.
 */

#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

const int N_TRIALS = 10;

int main() {
   cout << "On this computer, RAND_MAX is " << RAND_MAX << endl;
   cout << "The first " << N_TRIALS << " calls to rand:" << endl;
   for (int i = 0; i < N_TRIALS; i++) {
      cout << setw(10) << rand() << endl;
   }
   return 0;
}
```

图 2-10 测试 rand 函数的程序

正如你所看到的，rand 函数的值总是正的，并且从来不会大于 RAND_MAX 的值。而且，函数的值看起来在规定数字范围内不停变动，无法预测，这也正是你想要从一个伪随机过程得到的结果。

假设C++类库中已存在一个产生伪随机数的函数，你可能有充足的理由想问为什么我们要重新设计一个接口来支持这一过程。一部分的答案是rand函数本身并不总是返回用户原意所需要的数值。另一方面，RAND_MAX的值取决于硬件和软件环境。在大多数系统上，RAND_MAX被定义为整型数的最大值，通常为2 147 483 647，但在不同的系统上这一数值可能不一样。即使你能确定RAND_MAX拥有特定的值，也只有少部分应用情况会需要一个在0到2 147 483 647之间的值。作为一个用户，你更想要的是一个落在另一个数值范围内的随机数，通常这一范围会比上面列举的范围要小。例如，如果你想要模拟抛硬币的过程，你仅需要考虑两种输出可能性的函数：正面和反面。同样，如果你想要表示一个类骰子的过程，你需要在1到6中产生一个随机整数。如果你正在尝试模拟物理世界，你需要一个在连续范围内的随机数，这个数需用double类型来表示，而不是int类型。如果你可以设计一个符合以上用户需求的接口，那么这个接口会更加灵活，使用起来也会更加简单。 92

设计一个更高层次接口的另一个原因是：使用<cstdlib>中的低层次接口会使你在实现的过程中提高程序的复杂性，而这些复杂性正是用户所极力避免的。接口设计师的一部分工作就是要最大化地向用户隐藏这些复杂的实现过程。定义一个高层次的random.h接口使得这一设想成为可能，因为这一接口可以使得复杂性局限于实现过程中。

2.8.3 选择正确的函数集

作为一个接口设计者，你面临的首要挑战就是选择接口对外提供的函数。虽然接口设计看起来更偏向于艺术方面的不是科学方面，但依然有2.7节中所包含的若干通用原则可供设计中遵循。特别是，你在random.h中提供的函数必须足够简单，并且应尽可能隐藏其背后复杂的实现过程。同时，这些函数还必须提供必要的关键功能来满足用户的广泛需求，这也意味着你必须清楚了解用户有怎样的需求。理解这些需求的能力不仅取决于你自己的设计经验，也通常需要与潜在用户进行交流以更好地理解这些需求。

以我本人的编程经验，我知道用户期望从random.h中获得以下功能：

- 从一个特定的区域内选取一个随机整数。例如，如果你想要模拟掷一个具有六个面的标准骰子的过程，你需要从1到6中随机选择一个整数。
- 从一个特定的区域内选取一个随机实数。如果你想要一个物体处于空间中的一个随机位置，你需要在所有限制条件中随机生成坐标x和y值。
- 以某个特定概率模拟一个随机事件。如果你想要模拟抛硬币这一事件，你需要生成具有概率为0.5的heads值，这也意味着在抛硬币时，有50%的概率抛硬币的正面朝上。

将这些概念上的操作转换为一组函数原型是一个相对直接的工作。random.h中的三个函数（randomInteger、randomReal、randomChance）对应了这三个操作。一个完整的接口，还包括了另一个函数，称为setRandomSeed，这一函数出现在图2-11中，其功能将在稍后介绍。 93

```
/*
 * File: random.h
 * --------------
 * This file exports functions for generating pseudorandom numbers.
 */

#ifndef _random_h
#define _random_h

/*
 * Function: randomInteger
```

图2-11 随机数库接口

```
 * Usage: int n = randomInteger(low, high);
 * ----------------------------------------
 * Returns a random integer in the range low to high, inclusive.
 */

int randomInteger(int low, int high);

/*
 * Function: randomReal
 * Usage: double d = randomReal(low, high);
 * ----------------------------------------
 * Returns a random real number in the half-open interval [low, high).  A
 * half-open interval includes the first endpoint but not the second, which
 * means that the result is always greater than or equal to low but
 * strictly less than high.
 */

double randomReal(double low, double high);

/*
 * Function: randomChance
 * Usage: if (randomChance(p)) . . .
 * ---------------------------------
 * Returns true with the probability indicated by p.  The argument p must
 * be a floating-point number between 0 (never) and 1 (always).  For
 * example, calling randomChance(.30) returns true 30 percent of the time.
 */

bool randomChance(double p);

/*
 * Function: setRandomSeed
 * Usage: setRandomSeed(seed);
 * ---------------------------
 * Sets the internal random number seed to the specified value.  You can
 * use this function to set a specific starting point for the pseudorandom
 * sequence or to ensure that program behavior is repeatable during the
 * debugging phase.
 */

void setRandomSeed(int seed);

#endif
```

94

图2-11 （续）

正如你在图2-11中的注释和原型中所看到的，randomInteger函数需要传入两个整型参数并返回一个在这两个数范围内的随机整数。如果你想要模拟掷骰子的过程，只要像下面这样调用函数即可：

```
randomInteger(1, 6)
```

为了模拟一个欧洲大转盘（美国转盘除了1到36之外，还有0和00这两个位置），你应该这样调用函数：

```
randomInteger(0, 36)
```

randomReal函数理论上来说与randomInteger函数是类似的。但这个函数需要传入两个浮点参数：low和high，并且返回一个浮点数r，且满足：low≤r<high。例如，调用randomReal(0,1)将返回一个不小于0但小于1的浮点数。数学上，可等于某一端点的值而不能等于另一端点的值的一个实数区间称为半开区间（half-open-intrerval）。在一条数轴上，一个半开区间用一个空心点来标记其值不能等于端点值，例如：



在数学上，依照惯例是使用方括号来表示区间闭端，使用圆括号来表示区间的开端，所

以 [0, 1) 表示了图中所示的半开区间。

函数 randomChance 被用来模拟具有特定概率的随机事件，并且其事件发生概率可以设定。按照惯例，事件发生概率通过 0 到 1 中的一个数值来表示：0 表示不可能事件，1 表示必然事件。像这样 randomChance(p) 调用函数将以 p 为概率返回 true 值，因此，以 randomChance(0.75) 调用函数将有大约 75% 的概率返回 true 值。

你可以使用 randomChance 函数来模拟抛硬币的结果，正如以下函数说明的，函数以等概率返回 "heads" 或 "tails"：

```
string flipCoin() {
   if (randomChance(0.50)) {
      return "heads";
   } else {
      return "tails";
   }
}
```

[95]

2.8.4 构建用户程序

验证接口设计的最好方法就是编写一个程序并使用它。图 2-12 所示的程序给出了使用 randomInteger 函数来实现一种称为 craps 的赌场游戏。craps 的规则在程序开头的注释中进行了介绍，你也可以让程序向玩家介绍这一规则。在这个例子中，为了节省空间我们省略了打印玩家指南这一步骤。

尽管 Craps.cpp 程序是不确定性的，并且每次会产生不同的输出，运行该程序一个可能的实例如下：



2.8.5 随机数库的实现

直到现在，这一章看起来依旧停留在设计 random.h 接口这一步骤上。当你能在程序中实际使用这一接口之前，编写 random.cpp 来实现库显得尤为重要。在这一阶段你需要知道在 <cstdlib> 库中存在一个可以产生从 0 到 RAND_MAX 随机数的函数 rand，这一数值范围如下图所示：



例如，为了模拟掷骰子的过程，你需要将函数生成的随机整数转换为下面的任何一个离散数值：



为了达到这一目的，出现了很多并不高明的转换方法。很多书会采用以下方法：

96
```
int die = rand() % 6 + 1;
```



```
/*
 * File: Craps.cpp
 * ---------------
 * This program plays the casino game called craps, which is
 * played using a pair of dice.  At the beginning of the game,
 * you roll the dice and compute the total.  If your first roll
 * is 7 or 11, you win with what gamblers call a "natural."
 * If your first roll is 2, 3, or 12, you lose by "crapping
 * out."  In any other case, the total from the first roll
 * becomes your "point," after which you continue to roll
 * the dice until one of the following conditions occurs:
 *
 * a) You roll your point again, in which case you win.
 * b) You roll a 7, in which case you lose.
 *
 * Other rolls, including 2, 3, 11, and 12, have no effect
 * during this phase of the game.
 */

#include <iostream>
#include "random.h"
using namespace std;

/* Function prototypes */

bool tryToMakePoint(int point);
int rollTwoDice();

/* Main program */

int main() {
   cout << "This program plays a game of craps." << endl;
   int point = rollTwoDice();
   switch (point) {
    case 7: case 11:
      cout << "That's a natural.  You win." << endl;
      break;
    case 2: case 3: case 12:
      cout << "That's craps.  You lose." << endl;
      break;
    default:
      cout << "Your point is " << point << "." << endl;
      if (tryToMakePoint(point)) {
         cout << "You made your point.  You win." << endl;
      } else {
         cout << "You rolled a seven.  You lose." << endl;
      }
   }
   return 0;
}

/*
 * Function: tryToMakePoint
 * Usage: flag = tryToMakePoint(point);
 * ------------------------------------
 * Rolls the dice repeatedly until you either make your point or roll a 7.
 * The function returns true if you make your point and false if a 7 comes
 * up first.
 */

bool tryToMakePoint(int point) {
   while (true) {
      int total = rollTwoDice();
```

97

图 2-12 演示掷骰子游戏的程序

```
      if (total == point) return true;
      if (total == 7) return false;
   }
}

/*
 * Function: rollTwoDice
 * Usage: total = rollTwoDice();
 * ----------------------------
 * Simulates the process of rolling two dice.  The individual values of the
 * dice are printed on cout along with the sum, which is returned as the
 * value of the function.
 */

int rollTwoDice() {
   cout << "Rolling the dice . . ." << endl;
   int d1 = randomInteger(1, 6);
   int d2 = randomInteger(1, 6);
   int total = d1 + d2;
   cout << "You rolled " << d1 << " and " << d2
        << " - that's " << total << "." << endl;
   return total;
}
```

图 2-12（续）

这一句代码表面上看起来简单易懂。rand 函数总是返回一个正整数，这个数除 6 的余数必然会落在 0 到 5 之间。将所得余数加 1 使最终数值落在我们所需要的区域 1 到 6 的数值中。

问题的关键是 rand 函数只保证生成的值可以一致均匀地分布在 0 到 RAND_MAX 之间。也就是说，这个函数并不能保证结果除 6 之后得到的余数都具备同样的随机性。事实上，早期随 UNIX 操作系统一起发布的 rand 函数版本产生的值在奇数和偶数中间选择，而不管实际上这些数字在其取值范围内出现的概率是否相同。获取除 6 的余数也将在奇数和偶数之间做选择，这将使得结果很难符合随机数的定义。如果没有别的办法，我们无法连续两次随机 98
地掷骰子，这将导致掷一个骰子时可能为奇数而掷另一个骰子时只能为偶数。

你所要做的就是去除以 0 到 RAND_MAX 之间的整数，使得结果落在六个长度相等的区域来得到不同的输出，就像下图所示：



更通用的情况是，你首先需要将 0 到 RAND_MAX 的数轴划分成 k 个相等的区间，这 k 个区间代表了你想要的输出值范围。然后，你所需要做的就是将每个区间数映射为用户所需要的值。

将 rand 函数返回结果转换为有限区域内的一个整数这一过程可以简单地看成由以下四个步骤组成：

1. 规范化（normalize）rand 函数的整数结果 d，将其转换为浮点数，且满足：$0 \le d < 1$。

2. 量化（scale）d 值，将 d 乘以想要的数值范围来实现，使得数值 d 为落入到其数据范围内的一个正确的整数。

3. 翻译（translate）d 值，将 d 加上最低边界值使其数值范围从预想的点开始。

4. 转换（convert）d 值，调用 <cmath> 类库中的 floor 函数将 d 转换为小于其函数参数的最大整数。

图 2-13 说明了上述步骤，并且通过该过程可跟踪到一条可能的路径。如果 rand 函数的调用返回了数值 848 256 064，并且 RAND_MAX 的最常见值是 2 147 483 647，则规范化阶段将产生一个靠近 0.4 的值（图 2-13 中的值为了便取值符合图表已经将数值取为保留一位小数）。在量化阶段将此数值乘以 6 得到 2.4，之后的翻译阶段增加到 3.4。在转换阶段，以 3.4 为输入值调用 floor 函数，则程序输出数值 3。

编写代码来实现上述过程并不如想象中的容易，因为在上述过程中还存在一些可能不够完美的程序的失败陷阱。例如仅考虑规范化阶段，你不能通过如下调用将 rand 函数处在区间 [0, 1) 中的返回值转类为浮点型数值：

[99]
```
double d = double(rand()) / RAND_MAX;
```

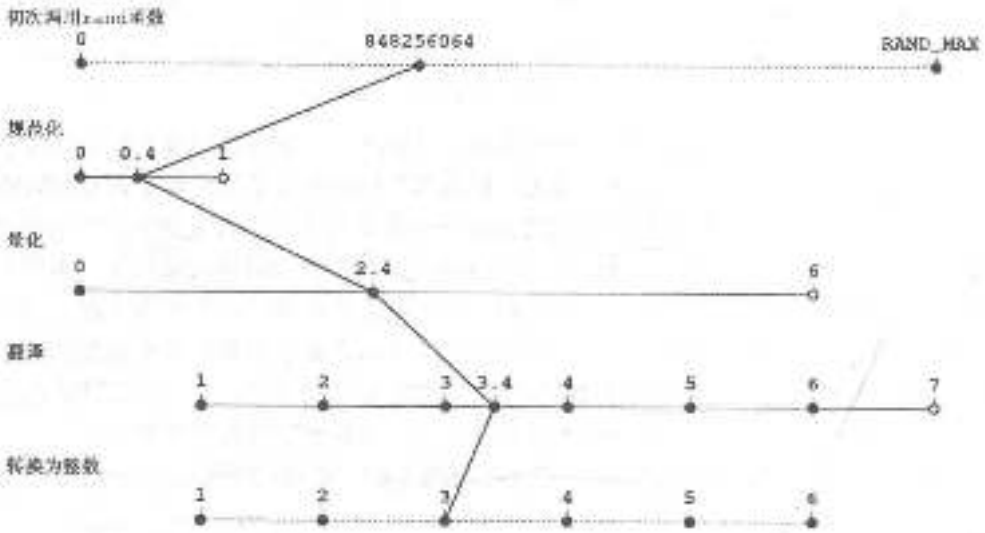


图 2-13 生成 1～6 区域的一个随机数所需的步骤

这里的主要问题是 rand 函数可能返回一个 RAND_MAX 值，这意味着变量 d 会被赋值为 1.0，而这一数值并不在半开区间内。你也不能像下面这样编写代码：

```
double d = double(rand()) / (RAND_MAX + 1);
```

虽然这里存在的问题更加微妙。像我们之前所说的，RAND_MAX 是 int 类型的最大正整数值。如果碰到这种情况，将 RAND_MAX 加 1 将会得到一个上溢的数值。

为了解决上述问题，你需要做的就是使用 double 类型数值而不是 int 类型数值，像下面这样：

```
double d = rand() / (double(RAND_MAX) + 1);
```

在量化阶段你会遇到类似的问题。数学上在 low 到 high 范围内的整数个数通过如下表达式计算获得：high-low+1。然而，如果 high 是一个大的正数而 low 是一个小的负
[100] 数，那么，这一表达式也必须使用双精度浮点数来表示。

考虑到上述所有实现的复杂性，randomInteger 函数的实现过程如下：

```
int randomInteger(int low, int high) {
   double d = rand() / (double(RAND_MAX) + 1);
   double s = d * (double(high) - low + 1);
   return int(floor(low + s));
}
```

在你解决 randomInteger 函数实现的复杂性之后，randomReal 和 randomChance 函数的实现过程相对比较简单了：

```
double randomReal(double low, double high) {
   double d = rand() / (double(RAND_MAX) + 1);
   double s = d * (high - low);
   return low + s;
}

bool randomChance(double p) {
   return randomReal(0, 1) < p;
}
```

2.8.6 初始化随机数种子

遗憾的是，前面介绍的 randomInteger、randomReal 和 randomChance 函数并不会像用户希望的那样工作。问题在于：与产生无法预测的结果相反，使用这些函数的程序总会产生相同的结果。举例来说，如果你运行 Craps 程序 20 次，每次你都会看到程序产生完全相同的输出。并不产生随机数。

为了弄明白为何函数的实现会产生这一结果，你可以回到函数 RandTest 程序并再一次运行该程序。此时，程序输出为：



101

这次程序的运行输出与第一次运行输出的结果相同。事实上，RandTest 程序每次都会产生完全相同的输出，因为 C++ 类库（该类库的基础也就是早期的 C 语言库）的设计者设计的 rand 函数每次运行都会产生相同的随机序列。

首先，你可能很难理解产生随机数的函数为什么会总是返回同一序列的值。毕竟这种确定的行为似乎与随机概念完全相反。然而，程序的这一行为很好解释：一个具有确定行为的程序更易于调试。

为了搞懂这种重复到底有什么意义，想象一下你编写了一个程序用来玩类似大富翁这种复杂的游戏。在编写一个新程序的时候，你的程序很大程度上会存在着一些漏洞。在一个复杂的程序中，漏洞常常会隐藏其中，除非现实中某种特殊情况触发了这一漏洞。假设你在玩

游戏时发现了游戏程序表现异常。这时你需要调试该程序，如果你可以重新恢复程序表现异常的状态并观察发生了什么，将会使调试变得非常方便。遗憾的是，如果程序处于一种随机状态，程序的第二次运行与第一次运行将会产生不同的表现。你第一次运行时出现的问题可能就不会在第二次运行时出现。因此，最初的C语言库设计者进行了总结，认为rand()函数必须以一个固定的方式运行来支持程序的调试。

与此同时，xrand函数又必须具备输出与不相同结果的功能。为了理解如何实现这一行为，我们必须了解rand函数内部的工作原理。rand函数通过对它最后产生的值进行一系列数学计算来产生一个新的随机数值。因为你不了解这一系列计算过程，因此将上述整个操作看成一个黑箱操作会更好。其中数据从黑箱的一端输入，新的伪随机数将从黑箱的另一端输出。因为第一次调用rand函数产生值1 103 527 590，第二次调用xrand函数将对应从黑箱的一端输入1 103 527 590，并在其另一端产生值377 401 575：



下一次调用rand函数时，实现过程将377 401 575输入到黑箱中，并返回662 824 084：



每一次调用rand函数将重复上述相同过程。黑箱内部的计算过程被设计为具有如下功能：

（1）产生的随机数将均匀地分布在其合法的取值区域内；

102 （2）产生的随机数序列会在很长一段时间后才会重复出现。

但为何在第一次调用rand函数的时候会返回1 103 527 590呢？因为要实现rand函数的计算过程，必须拥有一个开始点，必须存在一个整数S_0被输入到黑箱中，并产生值1 103 527 590：



这个初始值——即用于启动整个黑箱过程的值被称为随机数产生器的**种子**（seed）。在<cstdlib>类库中，你可以通过调用srand(seed)明确设置该种子值。

正如在多次运行RandTest程序时你所看到的，C++类库在每次程序启动时都初始化种子为一个常量值，这就是rand函数总是输出相同序列随机数的原因。然而这一行为仅在调试阶段会非常有用。很多现代程序设计语言都改变了默认值行为，以便随机数库中的函数每次运行时都返回不同的值，除非程序员进行了特殊设置。为了用户使用rand函数更加简单，其设计修改已体现在random.h接口中。但是允许用户能产生重复的随机数序列值仍然是必需的，因为这样做简化了调试程序过程。对这一选择的需求也是我们在random.h接口中提供setRandomSeed函数的原因。在调试阶段，你可以在main函数的开头增加以下语句：

```
setRandomSeed(1);
```

之后，对random.h接口函数的调用将产生重复的以1为初始种子的随机数序列。当确保程序可正常运行后，你可以删除这一语句以恢复程序产生不可预测的结果。

为了实现程序产生不可预测的结果，函数randomInteger、randomReal和randomChance必须在运行之前首先检查随机数种子是否已经被初始化，如果没有，将其

初始值设为某个用户难以预测的值。它通常从用户的系统时间获得。因为系统时间数值每一次运行程序时都不一样，随机数序列也随着时间不断变化。在 C++ 中，你可以通过调用函数 time 来获取系统时间，并将其转换为一个整型数。这一技术允许你编写如下语句以使伪随机数产生器被初始化为某个不可预测的值：

```
srand(int(time(NULL)));
```

尽管只需要上述一行代码，但利用系统时间将随机数种子初始化为一个不可预知的值这一操作还是相当晦涩难懂的。如果这一行语句出现在用户编写的代码中，用户就必须同时了解随机数种子的概念，srand 函数，time 函数和 NULL 常数（这一参数将在第 11 章中学习）这些知识。为了使用户使用起来更加简单，你必须将这些复杂的东西隐藏起来。 103

当你意识到初始化这一步骤在提交其他函数结果之前必须做且只能做一次，情况将变得更为复杂。为了保证初始化代码不会每次都执行，你需要有一个记录是否已初始化的布尔类型标志。遗憾的是，将这一标志声明为全局变量并不能达到目的，因为在 C++ 中不能指定全局变量初始化的次序。如果你声明了使用随机库函数产生的值初始化一些全局变量，并不能保证初始化标志已经被正确设置。

在 C++ 中，最好的办法就是在函数代码段中声明初始化标志来检查必要的初始化是否已经完成。然而该初始化标志变量不能被定义为一个传统的局部变量，因为这样做意味着每一次调用函数都将产生一个新的变量。为了使这一标志在每一次函数调用中都起作用，你需要将该变量声明为 static，如同下述代码：

```
void initRandomSeed() {
   static bool initialized = false;
   if (!initialized) {
      srand(int(time(NULL)));
      initialized = true;
   }
}
```

当使用关键字 static 来标记 initialized 变量时，它将成为一个**静态局部变量**（static local variable）。类似于其他的局部变量，一个静态局部变量只在函数体中可被访问。不同的是，对于该类变量，编译器只进行一次内存分配操作，而这一内存也被接下来的每一次 initRandomSeed 函数调用所共享。C++ 语法确保静态局部变量只被初始化一次，而且该初始化发生在该函数被第一次调用时的该变量定义点。定义 initRandomSeed 函数标志着 random.cpp 实现代码的完成，其代码如图 2-14 所示。

我们介绍整个编写 random.cpp 实现代码的目的并不是想要掌握其中的所有难点。我们希望让你明白为何作为 random.h 的用户你不用在意其实现的所有细节。random.cpp 中的所有代码都是极其精妙的，并且其中也存在许多陷阱使得想要单独实现它的程序员可能误入歧途。必须记住，库接口的首要目标就是将其内部所有实现的复杂性对用户隐藏起来。 104

```
/*
 * File: random.cpp
 * ----------------
 * This file implements the random.h interface.
 */

#include <cstdlib>
#include <cmath>
```

图 2-14 随机数库的实现

```
#include <ctime>
#include "random.h"
using namespace std;

/* Private function prototype */

void initRandomSeed();

/*
 * Implementation notes: randomInteger
 * -----------------------------------
 * The code for randomInteger produces the number in four steps.
 *
 * 1. Generate a random real number d in the range [0 .. 1).
 * 2. Scale the number to the range [0 .. N) where N is the number of values.
 * 3. Translate the number so that the range starts at the appropriate value.
 * 4. Convert the result to the next lower integer.
 *
 * The implementation is complicated by the fact that both the expression
 *
 *     RAND_MAX + 1
 *
 * and the expression for the number of values
 *
 *     high - low + 1
 *
 * can overflow the integer range.  These calculations must therefore be
 * performed using doubles instead of ints.
 */

int randomInteger(int low, int high) {
   initRandomSeed();
   double d = rand() / (double(RAND_MAX) + 1);
   double s = d * (double(high) - low + 1);
   return int(floor(low + s));
}

/*
 * Implementation notes: randomReal
 * --------------------------------
 * The code for randomReal is similar to that for randomInteger,
 * without the final conversion step.
 */

double randomReal(double low, double high) {
   initRandomSeed();
   double d = rand() / (double(RAND_MAX) + 1);
   double s = d * (high - low);
   return low + s;
}

/*
 * Implementation notes: randomChance
 * ----------------------------------
 * The code for randomChance calls randomReal(0, 1) and then checks
 * whether the result is less than the requested probability.
 */

bool randomChance(double p) {
   initRandomSeed();
   return randomReal(0, 1) < p;
}

/*
 * Implementation notes: setRandomSeed
 * -----------------------------------
 * The setRandomSeed function simply forwards its argument to srand.
 * The call to initRandomSeed is required to set the initialized flag.
 */

void setRandomSeed(int seed) {
   initRandomSeed();
```

105

図 2-14 (続)

```
   srand(seed);
}

/*
 * Implementation notes: initRandomSeed
 * ------------------------------------
 * The initRandomSeed function declares a static variable that keeps track
 * of whether the seed has been initialized.  The first time initRandomSeed
 * is called, initialized is false, so the seed is set to the current time.
 */

void initRandomSeed() {
   static bool initialized = false;
   if (!initialized) {
      srand(int(time(NULL)));
      initialized = true;
   }
}
```

图 2-14 （续）

[106]

2.9 Stanford 类库介绍

到目前为止，本书中的程序只使用了 C++ 中一小部分标准库特性，以保证这些程序可在任何类型硬件上的 C++ 编译器上运行。遗憾的是，C++ 标准类库并没有包含一个程序员所需要的所有特性。虽然大多数的现代应用使用图形显示，但是标准类库并不提供图形处理功能。每一个现代操作系统都提供了支持在屏幕上画图的类库，但是这些类库之间并不相互兼容。为了创建一个和你平常使用的一样有趣的应用，在某些地方你会使用非标准库。

作为本书的部分支持材料，斯坦福大学提供了一系列让你可以更加轻松愉悦进行 C++ 编程的类库（Stanford 类库）。这些类库包含了一个图形处理包，它与大多数普通计算平台上的类库工作方式相同。Stanford 类库也提供了本章所讲的该库的介绍主页。在我们讲解并熟知了定义和实现类似 error.h、direction.h、gmath.h 和 random.h 这些接口的设计之后，必须明白拒绝这些工具甚至强迫潜在用户拷贝相关的代码并不合理。由于每个接口最终都会在应用开发时带来便利，因此，最好的做法是将接口做成类库的一部分。然而，建立一个编译类库超出了本书的范围，主要原因是建立编译类库所涉及的细节在不同的机器平台上是不同的。本书在斯坦福的网站上有适合学生可能使用的所有主要平台的预编译版本类库。你可以从网上下载这些类库，其中的 .h 后缀文件定义了类库的接口。为了弄明白如何在你的工作环境下使用 Stanford 类库，你可能需要花些时间，但这么做的优点将在之后很快会弥补这段时间的损失。

2.9.1 简单的输入和输出类库

除了本章所介绍的类库接口之外，Stanford 类库还包括了其他几个能使编程变得更容易的接口。在这些接口中，最重要的就是 simpio.h 接口了，这一接口简化了从用户获取输入的过程，与以下几行代码不同：

```
int limit;
cout << "Enter exponent limit: ";
cin >> limit;
```

使用 simpio.h 允许你将上述代码行简化为下述一行代码：

```
int limit = getInteger("Enter exponent limit: ");
```

[107]

尽管使用 simpio.h 后代码明显缩短，但是使用该类库的真正优点是 getInteger 函

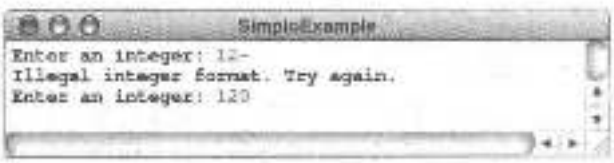
数可以检查用户的输入错误。例如，假设你的程序正在执行下面这行代码：

```
int n = getInteger("Enter an integer: ");
```

getInteger 函数将显示提示字符串 Enter an integer: 并等待用户的输入，如下图所示：



如果用户想要输入数值 120，但在该输入 0 的地方错误地输入了一个负号，getInteger 函数将会检查出用户输入的是一个非法的整型值，并给用户再次输入正确数值的机会，这一过程如下图所示：



与此相反，>> 操作符并不检查上述该类错误。如果用户在执行以下语句时输入了 12-：

```
cin >> n;
```

则变量 n 将会得到输入值 12，并将负号保留在输入流 cin 中。

除了 getInteger 之外，simple.h 接口还提供了用于读取浮点数的 getReal 函数和用于读取一整行字符串的 getLine 函数。在第 4 章你将会学习如何实现这些函数，现在你可以先通过使用来熟悉它们。

Stanford 类库还包括了一个控制台类库，它在程序开始运行时会在屏幕上弹出一个控制台窗口。控制台窗口的自动弹出解决了丹尼斯·里奇在其著作《C 程序设计语言》(*The C Programming Language*) 中所提及的问题，即运行程序时的一个挑战就是：结果要输出到哪里？为了使用这个库，你需要在程序的开始部分加上以下这一行代码：

```
#include "console.h"
```

如果加入了这一行代码，程序就会建立一个控制台窗口，如果程序中缺失这一行代码，它仍
108 会正常运行，只是不弹出控制台窗口而已。

2.9.2 Stanford 类库中的图形处理程序

将 C++ 作为一种教学语言的挑战就是 C++ 未提供标准图形库。虽然在不使用图形界面的情况下，我们可以很好地学习数据结构和算法，但在学习过程中加入图形库将使学习过程变得更加有趣。并且由于学习变得更加有趣你将更容易接受新知识，因此，Stanford 类库包含了一个可以在大多数普通平台上使用的二维图形处理类库包。

图形处理类库最主要的接口是 gwindow.h，该接口提供了一个可以创建图形窗口的类 GWindow。为了在屏幕上画图，需要声明一个 GWindow 类的对象，然后调用其中的对象方法。虽然 GWindow 类提供了大量的方法，但本书中的程序仅使用了表 2-2 中所示的方法。所有图形处理包基本上都提供了与表 2-2 所示功能类似的方法，因此，熟悉本书中对图形处理库的使用将使你更易于使用其他类似的类库。

图 2-15 是一个简单的图形应用程序，设计该简单应用程序的目的是用于说明表 2-2 中的方法，而不是为了绘画。程序的输出呈现在图 2-16 中。

表 2-2 GWindow 类中的方法

`GWindow gw;` `GWindow gw(width, height);`	创建一个图形窗口，它通常存储在变量 gw 中，该变量必须通过引用传递给完成图形操作。若参数值 *width* 和 *height* 省略，则构造函数创建了一个默认尺寸为 500×300 像素的窗口
`gw.drawLine(x0, y0, x1, y1);`	画一条从点 (x_0, y_0) 到点 (x_1, y_1) 的线段
`gw.drawPolarLine(x, y, r, theta);`	从点 (x, y) 开始画一条长度为 r 像素，与 x 轴间夹角为 θ 的线段。该方法将在第 5 章做更详细的介绍
`gw.drawRect(x, y, width, height);`	根据特定的参数画一个矩形框
`gw.fillRect(x, y, width, height);`	根据特定的参数填充矩形
`gw.drawOval(x, y, width, height);`	根据特定的参数画一个矩形的内切椭圆
`gw.fillOval(x, y, width, height);`	根据特定参数填充矩形的内切椭圆
`gw.setColor(color);`	设置当前的绘图颜色。形参 *color* 是一个命名颜色的字符串。定义的颜色名列表以电子文档给出
`gw.getWidth();`	返回图形窗口的宽度，单位为像素
`gw.getHeight();`	返回图形窗口的长度，单位为像素

109

```
/*
 * File: GraphicsExample.cpp
 * -------------------------
 * This program illustrates the use of graphics using the GWindow class.
 */

#include "gwindow.h"

/* Prototypes */

void drawDiamond(GWindow & gw);
void drawRectangleAndOval(GWindow & gw);

/* Main program */

int main() {
   GWindow gw;
   drawDiamond(gw);
   drawRectangleAndOval(gw);
   return 0;
}

/*
 * Function: drawDiamond
 * Usage: drawDiamond(gw);
 * -----------------------
 * Draws a diamond connecting the midpoints of the window edges.
 */

void drawDiamond(GWindow & gw) {
   double width = gw.getWidth();
   double height = gw.getHeight();
   gw.drawLine(0, height / 2, width / 2, 0);
   gw.drawLine(width / 2, 0, width, height / 2);
   gw.drawLine(width, height / 2, width / 2, height);
   gw.drawLine(width / 2, height, 0, height / 2);
}
```

图 2-15 GraphicsExample 程序代码

```
/*
 * Function: drawRectangleAndOval
 * Usage: drawRectangleAndOval(gw);
 * ------------------------------
 * Draws a blue rectangle and a gray oval inscribed in the diamond.
 */

void drawRectangleAndOval(GWindow & gw) {
   double width = gw.getWidth();
   double height = gw.getHeight();
   gw.setColor("BLUE");
   gw.fillRect(width / 4, height / 4, width / 2, height / 2);
   gw.setColor("GRAY");
   gw.fillOval(width / 4, height / 4, width / 2, height / 2);
}
```

图 2-15 （续）

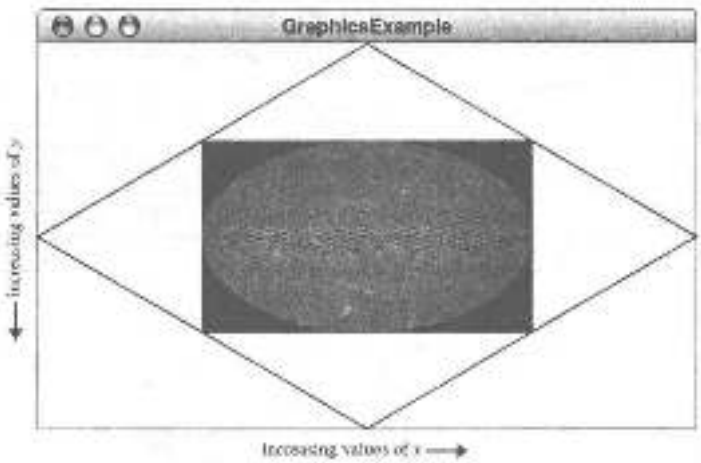


图 2-16 简单的图形示例输出

GraphicsExample 程序的第一行声明了一个用于实现图形操作的 GWindow 类对象，最简单的声明方法如下所示：

```
GWindow gw;
```

该声明创建了一个小型的图形窗口，并在屏幕上显示该窗口。同时，你可通过调用 GWindow 类的以下构造函数来设置窗口的大小：

```
GWindow gw(width, height);
```

上述构造函数的形参 width 和 height 的单位为像素（pixel），它是屏幕上的一个点。

对象 gw 的所有图形操作都由其所属类 GWindow 中的方法实现。因此，所有使用图形的代码段都必须通过变量 gw 来访问。如果你将程序分解为几个函数，那么需要以引用参数的方式在访问并操作窗口中图形的函数之间传递 gw 值。

图形窗口中的坐标使用数值对（x, y）来表示，其中，x 和 y 的值表示了它距离坐标原点（origin）（0, 0）的距离，该原点处于屏幕窗口的左上角。正如传统的笛卡尔坐标系，当你将点在窗口中向右移动时，变量 x 的值将增加。而原点在左上角使得变量 y 的值会随点坐标值向下移动而增加，这刚好与传统的笛卡尔坐标完全相反。计算机图形包反转了 y 坐标轴，因为这样做使文本在屏幕上看起来更自然。如果变量 y 的值随点的下移而增加，那么连续的多

行文本也将使得行数 y 逐行增加。

本章小结

本章我们学习了函数，函数使你可以用函数名来调用一系列特定的操作。函数作为一个从概念上降低编程复杂性的重要工具，允许程序员忽略其内部细节，只关注函数的最终功能。

本章包括以下要点：

- 函数是一个代码块，它组织成具有函数名并享有独立的内存空间。函数一旦被定义，则程序的其他部分可以调用该函数，通过实参列表向函数传递信息并通过函数的返回值传回结果。
- 函数在编程中起到以下重要作用。允许相同的指令集被多个不同的程序分享以降低程序的规模和复杂性。更重要的是，函数让程序可以被分解为更小、更易管理的多个代码片段。同时，函数是解决特定计算问题算法的实现基础。
- 当函数被聚集到类库中时，它们将变得更加有用，因为它们可被不同的应用程序所共享。每一个库都会定义若干个具有相同概念框架的函数。在计算机科学中，通过一个库使其中的一个函数可用的过程称作寻由该函数。
- <cmath>库提供了若干与数学概念上相似的函数，包括 sqrt、sin 和 cos。作为<cmath>库的用户，你不需要知道其中函数的实现细节，仅需知道如何调用它们即可。
- 在 C++ 中，函数在使用之前必须先声明。函数声明又称为函数原型。除了函数原型，函数还需有其实现，它定义了函数执行的每一个步骤。
- 有返回值的函数必须要有一个 return 语句，它指明了函数的返回结果。函数可以返回任意类型的结果。返回布尔类型的函数在编程中起着重要的作用，被称为判定函数。
- 仅执行函数中的语句完成特定功能但不返回值的函数称为过程。 112
- C++ 语言允许使用同名来定义多个函数，只要编译器能够通过函数参数的个数及类型来确定到底调用的是哪个函数即可。此多个同名的函数称为函数名的重载。用于区别不同重载函数版本的参数形式称为签名。C++ 语言可以定义默认参数，用户在调用函数时可省略相应的实参。
- 函数内定义的变量局部于该函数，不可在函数外使用。事实上，函数内定义的所有的局部变量被全部存储在栈帧中。
- 函数调用时，实参被求值后将其值拷贝到函数原型定义的形参变量中，并且，实参和形参顺序要一一对应。
- C++ 允许函数的调用者通过将变量用“&”标记传入函数，以便在函数中共享其变量值。这种类型的参数传递被称为引用调用。
- 当函数返回时，程序将返回函数调用点继续执行。计算机将该点称为返回地址并将其存储在栈帧中以便跟踪。
- 创建库的程序员称为库的实现者；使用库的程序员称为该库的用户。实现者和用户之间的连接件被称为接口。接口通常可提供函数、数值类型和常量定义，它们被统称为接口入口。

- 在C++中，接口被保存在以.h为后缀名的头文件中。每一个接口应该包含预编译头来确保编译器仅读取该接口一次。
- 当设计一个接口时，必须平衡多个设计需求。一个好的接口设计具有统一性、简单性、充分性、通用性和稳定性。
- random.h接口提供了若干可简化模拟随机行为的函数。
- 在本章所定义的几个接口（error.h、direction.h、gmath.h和random.h）是Stanford类库中的一部分，以便用户可以不必重写其代码而直接使用。Stanford类库还同时定义了另外几个有用的接口，包括用于简化输入/输出的库（simpio.h）、用于产生与控制台交互的库（console.h）、和在屏幕上用于显示图形窗口的库（gwindow.h）。

[113]

复习题

1. 用你自己的语言解释函数和程序的区别。
2. 定义函数中的几个专业术语：调用、实参、返回。
3. 判断题：C++程序中每个函数都需要一个函数原型。
4. <cmath>库中的sqrt函数的函数原型是什么？
5. 在一个函数体中可以存在多个return语句吗？
6. 什么是纯定函数？
7. 什么是函数重载？C++编译器是如何使用签名来实现函数重载的？
8. 如何为形参定义默认值？
9. 判断题：在函数第一个形参没有定义默认值的情况下可以为第二个形参定义默认值。
10. 什么是栈？
11. 描述实参与形参的关联过程。
12. 函数中定义的变量被称为局部变量。这一术语中“局部”是什么含义？
13. 术语引用调用的具体含义是什么？
14. 在C++程序中，如何表示引用调用？
15. 定义以下库中的术语：用户、实现、接口。
16. 如果你正在编写一个名称为mylib.h的接口，那么在预编译头中你会写上什么语句？
17. 描述使接口提供常量定义的过程。

[114]

18. 本章介绍的接口设计过程应遵循哪些核心准则？
19. 稳定性对于一个接口而言为何如此重要？
20. 什么是伪随机数？
21. 在大多数电脑上，RAND_MAX常量的值是多少？
22. 将rand函数的返回结果转换成不同区间上整数值的两个步骤是什么？
23. 如何使用randomInteger函数来生成1到100之间的伪随机数？
24. 通过手动执行randomInteger函数的每一条语句，判断该函数在负数条件下会产生什么结果。若调用函数randomInteger(-5,5)会产生什么结果？
25. 假设变量d1和d2已经被声明为整型变量，我们是否可以使用以下多重赋值语句：

```
d1 = d2 = RandomInteger(1, 6);
```

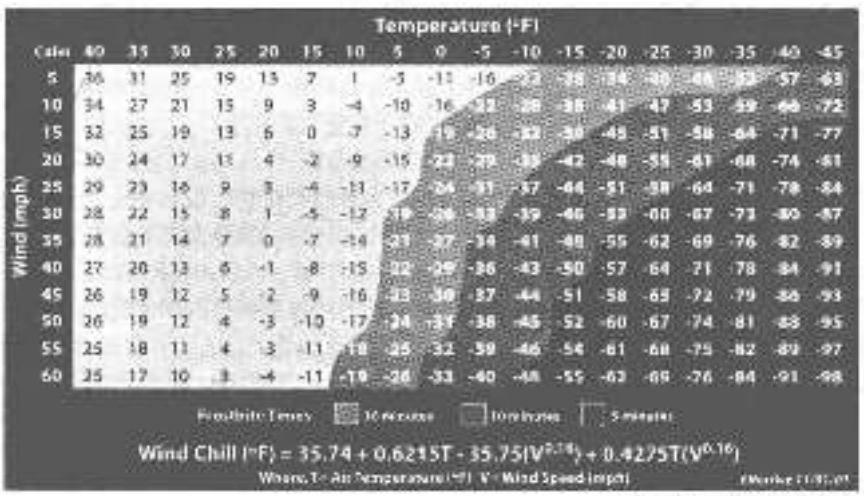
来模拟掷两颗骰子的过程？

26. 判断题：每一次程序运行，rand函数都会产生相同的随机数序列。
27. 在随机数中，什么是随机数种子？

28. 在介绍随机数时，本章为程序调试提出了什么建议？
29. 最新版本的 `<random>` 接口定义了哪些函数？在什么情况下会使用这些函数？

习题

1. 如果你还未完成第 1 章习题 1 中的摄氏度 – 华氏度转换程序，可以现在完成该程序，并使用函数来完成这种温度的转换。
2. 使用一个函数重新实现第 1 章习题 2 中的长度转换程序。此时，函数必须同时产生转换成公制的数值和转换成英寸的数值，这也意味着你需要通过引用调用来返回这些值。 115
3. 在 C++ 中，当一个浮点数被转换成整型数时，数值的小数部分会被直接舍去。因此，将 4.999 99 转换为整型数时，结果是 4。在许多场合，将浮点数转换为最接近的整型数将会更有用。现有一个浮点型变量 x，你可以通过将变量加上 0.5 并舍去小数部分来得到其最接近的整数。因为数值只会五入取整时会朝着数轴上 0 的方向，所以负数取整时需要为其数值减去 0.5，而不是加上 0.5。编写一个 `roundToNearestInt(x)` 函数，将浮点数取整为与其最接近的整数，并编写一个适当的 `main` 函数来检验它。
4. 如果你住上寒冷、大风的天气，你对温度的感觉不仅取决于温度，还取决于风速。风速超高，越觉得寒冷。为了量化风速对于温度感觉的影响，国家气象局做出了风寒指数报告，该部门网站上的风寒指数说明如图 2-17。



Source: National Weather Service

图 2-17 温度与风速的风寒指数函数 116

正如你在图 2-17 的底部所看到的，国家气象局使用以下公式计算风寒指数：

$$35.74 + 0.6215t - 35.75v^{0.16} + 0.4275tv^{0.16}$$

其中，t 是华氏温度，v 是风速，单位是英里 / 小时。

编写一个函数 `windChill`，参数为 `t` 和 `v`，返回风寒指数。为了达到这一目，你的函数必须考虑到以下两个情况：

- 如果没有风，`windChill` 必须返回原始温度 `t`。
- 如果温度高于 40°F，风寒指数未定义。此时你的函数必须使用正确的信息来调用 `error` 函数。

虽然在第 4 章学习如何格式化数据之后，你可以很轻松地编写出这种应用程序，但现在所学的知识已经足够使你将数据发向非列表输出类似图 2-17 所示的风寒指数表的所有行。如果你想要挑战自己，编写一个使用 `windChill` 函数的 `main` 函数来输出这个表格。

5. 希腊数学家热衷于寻找具值与自身所有真因子（proper division）的和相等的数。其中，真因子是指

除了该数本身的其他所有因子。数学上将符合上面定义的数称为完全数（perfect number）。例如，6是一个完全数，因为6的所有3个可以被6整除的真因子是1、2和3，它们之和正好等于6。类似地，28也是一个完全数，因为28的真因子1、2、4、7和14，其和为28。

编写一个判定函数 isPerfect，函数传入整型值 n，如果 n 是完全数，函数返回 true，如果不是则返回 false。通过编写一个 main 函数调用 isPerfect 检查从1到9999中的完全数来测试函数的实现。当找到一个完全数时，将它在屏幕上显示出来。程序启动的四行输出会是6和28。你的程序需要在给定的数值区间中找到另外两个完全数。

6. 一个比1大的整数如果除了自身和1之外没有其他因子，则被称为素数（prime）。例如17就是一个素数，因为除了1和17之外没有其他数可以整除它。91不是素数，因为它还可以被7和13整除。编写一个判定函数 isPrime(n)，如果整数 n 是素数，则返回 true，反之则返回 false。为了测试你的算法，编写一个 main 函数来列出1到100之间的素数。

7. 虽然 <cmath> 库的用户不需要了解函数 sqrt 的底层实现，但是这个库的实现者应会设计一个高
117 效的算法，并编写出其中蕴的代码来实现它。如果不使用类库来实现函数 sqrt，还有很多种方法可以采用。最简单的策略是采用逐步逼近法（successive approximation），这种方法通过在开始时做出预测，之后不断通过提高其求值精度来得到新值，使得其数值逼近最终结果。

你可以通过以下步骤使用逐步逼近法来求得x的平方根：

（1）开始时，将平方根值取为x/2，赋值给g。

（2）实际的平方根结果必须处于g和g/x之间。在逐步逼近法的每一步，通过计算g和g/x的平均值得到一个新的近似值。

（3）不断重复第2步直到g和g/x两值都足够接近，并达到硬件允许的精度要求为止。在C++中，最好的检验方法就是测试平均值与产生该值的两个数值是否相等。

使用上述方法编写你自己的 sqrt 函数。

8. 虽然在计算最大公约数的算法中，欧几里得算法是最古老的且非常实用的算法，但是在几个世纪之前也产生了很多其他算法。在中世纪，需要使用复杂算法的问题就是确定复活节的日期，它是每年春分后第一次圆月出现之后的第一个星期天。从其定义中我们知道：计算包括了每周日期的变换，月亮的轨道以及黄道带上太阳的位置。早期解决这一问题的算法要追溯到3世纪，并在8世纪被学者 Venerable Bede 记录在其著作中。1800年，德国数学家高斯出版了一本计算复活节日期的算法书，该算法通过计算得出，而不是查表。从德文翻译过来的该算法如图2-18所示。

编写一个过程：

```
void findEaster(int year, string & month, int & day);
```

它通过引用参数 month 和 day 为某个特定的年 year 返回其复活节日期。

遗憾的是，图2-18中的算法仅在18世纪和19世纪有效。然而，通过网络我们可以很轻易地找到适合所有年份的这一算法的扩展版本。在你完成高斯这一算法的实现之后，试着找到计算这一日
118 期更通用的算法。

I.某年份分别用19、4和7相除，并将其余数分别存入a、b和c中。若除数为偶数，则设其余数为0；不考虑商，以下除法与之相同。
II.用19a+20除以30，记其余数至d中。
III.最后，用2b+4c+6d+3或2b+4c+6d+4除以7，其中，第一个式子的年份应在1700和1799之间，而第二个式子的年份应在1800和1899之间，将其余数记入到e中。
然后，复活节日期应为3月22+d+e日，或者当d+e大于9时，其复活节日期应为4月d+e-9日。

译自卡尔·弗里德里克·高斯的“计算复活节的日期”，1800年8月。
http://gdz.sub.uni-goettingen.de/no_cache/dms/load/img/?IDDOC=137484

图2-18 计算复活节日期的高斯算法

9. 本章介绍的组合函数 C（n,k）可计算出从 n 个元素的集合中忽略排列顺序选取 k 个值共有多少种取法。如果考虑数值的排列情况，则在之前的硬币选择例子中，选择一个 2 角 5 分硬币再选择一个 1 角硬币，与先选择一个 1 角硬币再选择一个 2 角 5 分硬币是不一样的，要计算这种情况，你需要使用不同的函数，也就是排列函数（permutation）。排列函数计算从 n 个元素的集合中选取有序的 k 个元素共有多少种取法。这一函数用 P（n,k）标记，并具有如下数学公式：

$$P(n,k) = \frac{n!}{(n-k)!}$$

虽然从数学上来说，这一定义是正确的，但是在实际使用中并不合适，因为即使最终结果数值较小，其中包含的因子也有可能因为太大而无法存储在一个整型变量中。举例来说，如果你尝试使用这一公式计算从 52 张卡片中选取 2 张卡片的排列数，最终会得到下面的式子：

$$\frac{80\,658\,175\,170\,943\,878\,571\,660\,636\,856\,403\,766\,975\,289\,505\,440\,883\,277\,824\,000\,000\,000\,000}{30\,414\,093\,201\,713\,378\,043\,612\,608\,166\,064\,768\,844\,377\,641\,568\,960\,512\,000\,000\,000\,000}$$

虽然最后得到的结果并不大，为 2652（52 × 51）1。

编写一个函数 permutations（n,k），在不调用函数 fact 的情况下计算函数 P（n,k）。这一阶段你的部分工作就是找出如何高效地计算出这个数字。你可能会发现如果使用一些相对较小的数值来弄清楚公式中分子和分母的行为是很有用的。 119

10. 本章介绍的函数 C（n,k）和上述习题中的函数 P（n,k）总是能在计算机数学计算中见到，特别是在组合数学（combinatorics）这一领域，组合数学更专注于各种物体组合方法的数量。现在你已经使用 C++ 实现了这两个函数，接下来最好的办法就是将这两个函数放入到库中，这样你才能在许多不同的应用场合调用它们。

编写库文件 combinatorics.h 和 combinatorics.cpp，它提供函数 permutations 和 combinations。当你编写其实现时，确保你重写了 combinations 函数的实现代码，使其能利用你在习题 9 中的排列函数中的更高效算法。

11. 以 direction.h 接口为例，设计并实现一个 calendar.t 接口，该接口提供第 1 章中的数值类型 Month，还有函数 daysInMonth 和 isLeapYear。该接口还必须提供一个 monthToString 函数来返回 Month 类型值的常量名。编写一个 main 程序要求用户输入年份，然后输出该年份所有月份的天数来测试你的实现代码。就像以下程序输出：

```
TestCalendar
Enter a year: 2012
JANUARY has 31 days.
FEBRUARY has 29 days.
MARCH has 31 days.
APRIL has 30 days.
MAY has 31 days.
JUNE has 30 days.
JULY has 31 days.
AUGUST has 31 days.
SEPTEMBER has 30 days.
OCTOBER has 31 days.
NOVEMBER has 30 days.
DECEMBER has 31 days.
```



12. 编写一个程序 RandomAverage，它不断地产生一个 0 到 1 之间的随机实数，并在用户输入指定的输入次数之后显示其平均数。

13. 我从不相信上帝会掷骰子。

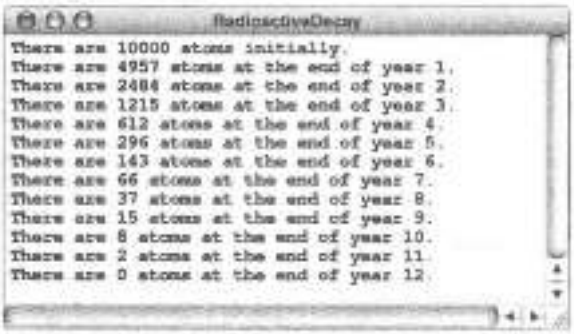
——阿尔伯特·爱因斯坦，1947

除了爱因斯坦形而上的反对声音之外，现在流行的物理模型，特别是量子力学都认为大自然实际上存在随机过程。例如，我们人类无法了解原子为什么衰变。实际上，原子在特定时期会随机生 120

地衰变。衰变有时发生，有时不发生，而且没有办法准确预测它们是否会发生。

因为物理学家将原子的衰变过程看成是一个随机过程，因此我们可以使用随机数来模拟这一过程一点也不奇怪。想象一下开始你拥有一些原子，每一个原子在任何时刻以一定的概率发生衰变。你可以通过将每个原子纳入考虑范围，并根据其概率随机决定其是否衰变来近似地模拟衰变过程。

编写一个模拟 10 000 个原子组成的材料的衰变过程的程序，其中，每个原子每年有 50% 的概率会发生衰变。程序必须输出在每年年末时还剩余多少个原子，如下图所示：



正如上图输出数据所表明的那样，每年有大约一半的原子样品会发生衰变现象。在物理上，这种现象我们就称这一样品的半衰期（half-life）为一年。

14. 随机数还提供了另一种逼近圆周率值的方法。想象一下墙上挂着一个靶子，这个靶子是画在正方形上的一个圆形，如下图所示：



[121]

如果你完全随机地向靶子掷出一连串飞镖，并且忽略所有没有落在正方形上的飞镖，这时会发生什么情况呢？有一些飞镖会落在灰色区域，但有一些却会落在圆外正方形的白色区域。如果你掷出时是随机的，则射在灰色区域飞镖数量与白色部分飞镖数量的比率将大致等于灰色部分面积和白色部分面积的比率。面积的比率与靶子的总面积似乎并不相关，这可以通过以下公式说明：

$$\frac{\text{圆内的灰色部分}}{\text{正方形内的灰色部分}} = \frac{\text{圆的面积}}{\text{正方形的面积}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

为了让程序模拟这一过程，想象一下靶子被画在标准的笛卡尔坐标系中，中心点在原点上，并且半径为 1 单位长度。掷飞镖的过程可以通过产生两个随机数 x 和 y 来建模，两个坐标变量取值都在 −1 和 1 之间。点 (x, y) 始终会落在正方形之中。如果符合以下条件，点 (x, y) 就落在圆中：

$$\sqrt{x^2 + y^2} < 1$$

然而这一条件可以通过将不等式两边同时取平方进行简化，使得计算更加高效：

$$x^2 + y^2 < 1$$

如果你多次模拟这一过程，并计算飞镖落入圆中的概率，结果会接近 $\pi/4$。

编写一个程序，模拟掷出 10 000 支飞镖，然后使用本习题中的模拟技术产生并显示圆周率的近似值。不用担心你得到的结果只在刚开始的几个小数位上是正确的。此方法在计算该问题上并不是特别精确，我们只是想通过这一例子来证明相似技术的可用性。在数学上，这一技术称为蒙特卡

洛积分法（Monte Carlo integration），这是一座坐落在摩纳哥首都旁边的城市，因其赌场而出名。

15. 正面…

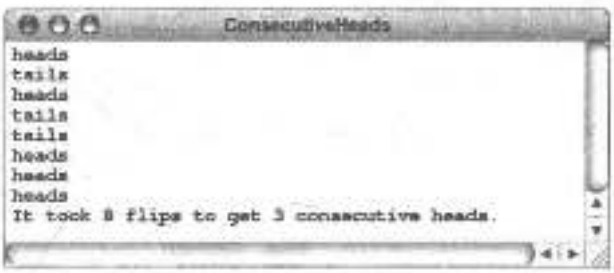
正面…

正面…

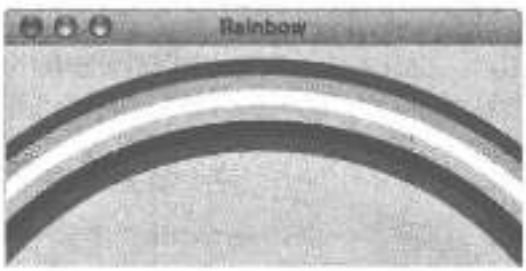
在一定的概率下，如果没有任何其他情况发生，一个赌者可能会重新审视他的信仰。

——汤姆·斯托帕德，《罗森克兰茨和吉尔登斯特恩已死》*Rosencrantz and Guildenstern Are Dead*，1967

编写一个模拟重复抛硬币过程的程序，一直抛直到连续三次正面朝上。此时，你的程序必须显示出共抛了多少次硬币。下面是一个可能的程序运行例子： 122

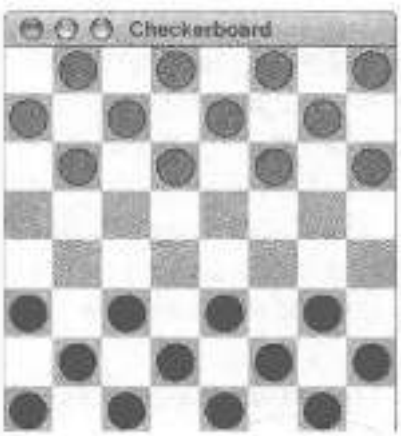


16. 使用图形库画出下图所示的彩虹：



先从上往下，彩虹的六条色带分别是：红、橙、黄、绿、蓝和品红，并用蓝绿色作为大空的可爱颜色。

17. 使用图形库编写一个在图形窗口中画出国际象棋棋盘的程序。你的图画必须包括红棋和黑棋，并且棋子的位置如以下游戏刚开始那样：



123

18. 道家哲学原理就是阴阳二气之间没有明显的界线，甚至在边界之间也混合着很多其他人可以明显看

到的相反的方面。这一思想体现在八卦图上，图中每一个区域都用有另一种颜色的小点：



编写一个图形处理程序，它可在图形窗口的中央画出八卦图。其挑战在于你只能使用表格 2-2
124 中的方法来分解绘画。这也意味着其中没有可以画出连续和平滑形的方法。

第 3 章

Programming Abstractions in C++

字符串类 string

> 地在这些弦上弹拨自低声的音乐。
>
> ——托马斯·斯特尔那斯·艾略特，《荒原》，1922

125

截至目前，你在本书中看到的大多数示例程序都使用数值作为程序的基本数据类型。如今，计算机被单纯用于处理数值的场合越来越少，更多的时候，计算机被用于处理**文本数据**（text data），而所谓的文本数据，一般指的是由多个独立字符构成的信息。现代计算机处理文本数据的强大能力已经保生了诸如手机短信、电子邮件、文字处理系统、网上图书馆和其他多种多样实用的应用程序。

在这一章中，我们将介绍 C++ 语言中的 <string> 类库，该类库向我们提供了方便操作字符串的抽象。熟练使用这一类库，可以使你更容易地编写出有趣的应用。在本章，我们还将介绍类（class）的概念，这一概念已经被计算机科学家采用，并应用于指代面向对象编程范式中的数据类型。虽然 C++ 语言定义了一个更原始的字符串类型，但是 string 类对象在大多数文本处理应用中具有更高的使用频率。在本章中，学习 string 类并掌握其应用将会加深你对于类这一概念的理解，同时为你在第 6 章中定义自己的类打下坚实的基础。

3.1 使用字符串作为抽象数据

从理论上来说，一个字符串（string）指的是一个特定的字符序列。例如，字符串 "hello,world" 中包含了 10 个字母、1 个逗号和 1 个空格的由 12 个字符组成的字符序列。在 C++ 语言中，string 类和其相关操作被定义在 <string> 类库里，因此，你必须在所有操作字符串数据的源代码中包含该类库。

在第 1 章中，你已经知道数据类型包含两个属性：值域和操作集。对于字符串来说，其定义域很容易进行界定：string 类型的定义域是字符串序列集。一个更有趣的问题是如何确定合适的操作集。早期的 C++ 语言版本效仿了旧版本的 C 语言，并且对字符串操作提供了少量支持。这一版本操作所提供的操作机制被定义在底层运算上，因此，执行相应操作要求你理解其底层表示。在此之后，C++ 语言设计者提出了字符串类 string，并很好地解决了该问题，string 类允许用户在更抽象的水平上处理字符串。

大多数情况下，你可以像使用 int 和 double 类型一样将 string 类当做基本数据类型来使用。例如，你可以声明各种各样的 string 类型变量，并赋给它一个初始值，这一操作与处理数值变量类似。当你声明一个 string 类型变量时，你一般赋给它一个**字符串字面值**（string literal）作为初始值，该值是用双引号括起来的一个字符序列。例如我们可以采用如下声明：

```
const string ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

将常量 ALPHABET 声明为包含 26 个大写字母的字符串。

126

你也可以使用操作符 >> 和 << 来读写 string 类型的值，当然这一操作必须小心谨慎。

例如，你可以通过以下语句改写 main 函数，使得 Hello 程序更具交互性：

```
int main() {
   string name;
   cout << "Enter your name: ";
   cin >> name;
   cout << "Hello, " << name << "!" << endl;
   return 0;
}
```

该程序从用户那里读取一个字符串并赋值给变量 name，之后将用户输入的名称作为问候语的一部分进行输出，该程序运行结果如下图所示：



但是，假如你输入了用户名有姓氏的全名而不是单独输入名称，你会发现程序产生了奇怪的运行结果。例如，若在输入时输入我的全名，程序将会产生如下输出：



显然该程序并没有包含用于将我的全名分解成两部分的代码，但当它输出问候语时，却只包含了我的名字而忽略了我的姓。为何程序能够产生如此口语化的输出呢？

为了回答这个问题，我们需要更深入地了解 >> 操作符是如何读取一个字符串值的。虽然你希望它能够读取一整行输入，但 >> 操作符一旦遇到空白字符（whitespace character）就会停止读取，这里，空白字符被定义为会在屏幕上显示空白间隔的字符。最常见的空白字符
[127] 就是空格，同时，在空白字符集中也包括制表符和行末标识符。

如果你想读取一个包含空白字符的字符串，则不能用 >> 操作符。最标准的方法是调用以下函数：

```
getline(cin, str);
```

该函数从控制台输入流 cin 读取一整行字符串，并存放到变量 str 中，其中，str 是通过引用进行传递的。包含 getline 函数的 HelloName 程序代码如图 3-1 所示，它可使程序输出用户的全名，如下所示：



实际操作中，读取一整行字符串这一操作比读取两端被空白字符分隔的字符串更加常见。因此，那些需要从用户那里读取完整字符串的程序更多地使用了 getline 函数（或者

2.9 节中 simple.h 头文件中包含的 getLine 函数），而不是 >> 操作符。

```
/*
 * File: HelloName.cpp
 * -------------------
 * This program extends the classic "Hello world" program by asking
 * the user for a name, which is then used as part of the greeting.
 * This version of the program reads a complete line into name and
 * not just the first word.
 */

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

int main() {
   string name;
   cout << "Enter your full name: ";
   getline(cin, name);
   cout << "Hello, " << name << "!" << endl;
   return 0;
}
```

图 3-1 “Hello World”程序的一个交互版本 128

3.2 字符串操作

如果你需要使用 <string> 类库来执行更复杂的操作，你会发现与传统数据类型相比，string 数据类型的表现并不完全相同。其中最主要的差别在于函数调用的语法。假如我们已经确定 <string> 库中提供了 length 函数，那么你可能认为可以通过如下函数调用来计算字符串长度：

```
int nChars = length(str);
```



在 C++ 语言中，调试程序时上述语句会被标记为错误。其原因在于 string 数据类型并不是一种传统的基本数据类型，它是一个类（class），这种类型被简单地定义为一个包含值集和相应操作集的模板。在面向对象编程语言中，属于一个类的所有值被称为该类的**对象**（object）。一个类可拥有多个不同对象，每一个对象被称作该类的一个**实例**（instance）。

应用于类实例的操作被称为**方法**（method）。在 C++ 语言中，方法的使用和操作与传统函数类似。但我们给它起一个新的名字是为了强调它们的不同。与函数不同，方法与它们所属的类紧密联系。强调这种差别是非常有意义的，传统的函数通常称为**自由函数**（free function），因为它们不被约束于特定的类。

在面向对象程序设计中，对象间通过信息发送和请求来实现对象间的通信。我们将传递的这些信息统称为**消息**（message）。对象间的消息发送通常理解为一个对象调用属于另一个对象的方法。为了与发送消息的概念模型达成一致，初始化方法的对象称为消息的**发送方**（sender），消息的目标对象称为**接收方**（receiver）。在 C++ 语言中，我们使用如下语法进行消息的发送：

```
receiver.name(arguments)
```

因此，在面向对象语言中，把字符串对象 str 的长度赋给 nChars 的语句如下：

```
int nChars = str.length();
```

129

表 3-1 列出了 <string> 类库中大部分常用方法，它们都使用以上接收方语法进行调用。

表 3-1 <string>库中常见的方法

字符串操作	
$str_1 + str_2$	连接字符串 str_1 和 str_2，返回一个连接后的新字符串。其中 str_1 或 str_2 可以被换为字符类型，但不允许被换为数字类型
str += str_2	将字符串 str_2 的拷贝添加到 str 的末尾。C++语言重载了这个操作符，使得 str_2 可以为字符类型
str_1 == str_2 str_1 != str_2 str_1 < str_2 str_1 <= str_2 str_1 > str_2 str_1 >= str_2	以这些操作符用于比较字符串 str_1 和 str_2，比较标准是按照字典序（lexicographic order），字典序由字符 ASCII 码值定义
str[k]	返回字符串 str 索引位置 k 上的字符，[] 操作符不检测 k 是否在其合法的范围内
读字符串内容方法	
str.length()	返回字符串 str 中字符的个数
str.at(k)	返回字符串 str 中索引位置 k 的字符。与 [] 操作符相反，如果 k 超出了其合法值范围，则 at 方法会产生异常
str.substr(pos, n)	返回一个新的字符串，该字符串是从 str 的 pos 位置开始，包含 n 个字符或直到 str 字符串末尾的子串。该方法的第二个参数是可选的，若无参数 n，子串总是会延伸至 str 字符串的末尾
str.compare(str_2)	比较接收方字符串 str 和 str_2，若两个字符串相等，则返回一个整数 0；如果 str 字典序在 str_2 之前，则返回一个负数；如果 str 字典序在 str_2 之后，返回一个正数。因为 C++ 重载了关系操作符，因此，程序员很少明确地调用 compare 方法
str.find($pattern$, pos)	在接收方字符串 str 中，从 pos 位置开始查找 $pattern$ 子串，若找到，则函数返回 $pattern$（可以是一个字符或一个字符串）所出现的第一个索引值；如果 $pattern$ 没有出现，find 返回常量 string::npos。方法的第 2 个参数是可选的；如果不提供第 2 个参数，则 find 方法会从字符串头开始搜索
修改接收方字符串内容方法	
str.erase(pos, n)	从 str 与 pos 处开始向后删除 n 个字符
str.insert(pos, str_2)	从 str 与 pos 处开始插入 str_2 的拷贝
str.replace(pos, n, str_2)	以 str_2 替换 str 中从 pos 处开始的 n 个字符
用 C++ 创建 C 风格的字符串方法	
string($carray$)	返回一个与 $carray$ 字符串相同字符的 C++ 字符串
string(n, ch)	返回一个包含 n 个 ch 字符的 C++ 字符串
str.c_str()	返回一个与 str 内容相同的 C 风格字符数组

130

3.2.1 操作符重载

正如你在表 3-1 的第一部分所看到的，<string>类库通过使用 C++ 语言的一种强大特性重新定义了标准操作符，这一 C++ 语言特性称为**操作符重载**（operator overloading），这一特性将依据操作数类型重新定义标准操作符执行的操作。在<string>类库中，最重要的重载操作符是 + 符号，当 + 被应用到数值时，它执行加法。当 + 被应用到字符串时，它完成字符串的**连接**（concatenation），这是一个精妙的符号，可以将两个字符串首尾相连组合

成新的字符串。

你也可以采用助记符 += 将新串连接到另一个字符串的尾部。+ 和 += 操作符都允许进行字符串或单个字符之间的连接，正如下面的例子，它将字符串变量 str 赋值为 "abcd"：

```
string str = "abc";
str += 'd';
```

如果你熟悉 Java 语言的编程，你可能认为 + 操作符能够支持将其他类型的值转换为字符串值，再与其他字符串进行连接的运算。但是这种情况在 C++ 中是不允许的，因为 C++ 是一种强类型语言，C++ 会把企图连接两个类型互不相容操作数的运算当作是一个错误。

C++ 语言也重载了关系操作符，相比于其他语言，包括 C 语言和 Java 语言，你能够更便捷地进行字符串的比较。例如，你可以使用下面的代码检测 str 的值是否等于 "quit"：

```
if (str == "quit")
```

关系操作符使用字典序来比较字符串，在底层这一次序通过 ASCII 码进行定义。字典序意味字母的大小写是有区别的，所以 "abc" 不等于 "ABC"。

3.2.2 从一个字符串中选取字符

在 C++ 中，一个字符串中字符的下标位置从 0 开始。例如，在字符串 "hello, world" 中以数字标识的字符下标位置如下图所示：

h	e	l	l	o	,		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10	11

其中，写在每个字符下面的位置数称为该字符的索引（index）。

<string> 类库提供了两种不同的机制来选取字符串中的特定字符。一种方法是将索引写在字符串后面的方括号中。例如，如果字符串变量 str 包含 "hello,world"，则以下表达式： 131

```
str[0]
```

指向字符串 str 的起始字符 'h'，虽然 C++ 程序员倾向于使用方括号语法来增加编程语言的可读性，但是调用 at 方法进行访问可能更好些。在 C++ 语言中，表达式 str[i] 和 str.at(i) 有着几乎相同的含义，唯一的不同在于 at 要检测并确保索引在字符串合法的取值范围内。

无论你使用哪种语法，从字符串中单独选取一个字符将会返回字符串中对应字符的引用，它允许你对该字符进行单独赋值。例如，你可以使用以下语句：

```
str[0] = 'H';
```

或者以下另一语句：

```
str.at(0) = 'H';
```

将字符串的值从 "hello, world" 更改为 "Hello, world"。第二种形式可以对函数返回的结果进行赋值这一点很容易让人困惑，并且在我的教学经验中发现学生对于使用 at 函数的写法不容易理解，所以虽然方括号的写法并不会执行范围检测，但在本书的示例程序中，我依旧选择使用方括号来选择字符串中的字符。

虽然字符串中的索引位置从概念上看是一个整数，但是 string 类依旧定义了 size_t

类型值来代表索引位置和字符串长度，这也使得该类的使用变得更复杂。如果你想让你的编码完全正确地通过编译，每一次你用一个变量存储一个字符串索引时，你应该使用 size_t 类型（该类型已经在 <string> 类库中默认自动定义）的变量。在斯坦福大学，我们已经发现使用 size_t 会使程序在概念上变得更加复杂，并且变得更加难以理解，因此，我们选择使用 int 类型来代替 size_t 类型。尽管程序编译时某些编译器可能会弹出一个警告消息，但是除非你的字符串比 2 147 483 647 个字符长度更长，否则使用 int 类型的效果会更好。

3.2.3 字符串赋值

为了减轻用户可以更改已存在字符串中的单个字符值这一概念问题的影响，C++ 采取了某些措施。C++ 重新定义了字符串赋值操作，允许我们将一个字符串赋值给另一个，这么做
132 将使用字符串的底层拷贝进行赋值。例如以下赋值语句：

```
str2 = str1;
```

用 str1 中字符的拷贝来重写 str2 之前的内容。变量 str1 和 str2 因此仍保持独立，这意味着改变 str1 中的字符并不影响 str2。类似地，C++ 在任意字符串中复制字符都是将字符作为一个参数值进行传递的。因此，除非字符串参数使用引用传递，否则在函数中对传入参数的更改并不会影响到函数外的实参值。

3.2.4 提取字符串中的子串

在连接较短字符串以形成更长的字符串运算后，你经常需要完成相反的工作：将一个字符串分解成数个短片段。一个较长字符串的一部分称为其子串（substring）。string 类提供了一个带有两个参数的方法 substr，其中第一个参数为提取字符串子串的起始位置，第二个参数为欲提取的子串字符个数。调用函数 str.substr(start, n)，将从 start 指定的索引位置开始在 str 中提取一个 n 个字符的子串。例如，如果 str 包含字符串 "hello, world"，如下调用方法：

```
str.substr(1, 3)
```

将返回包含 3 个字符的子串 "ell"。由于 C++ 中的下标从 0 开始，在索引位置 1 处为字符 'e'。在 substr 方法中第二个参数是可选的，如果第二个参数省略，substr 返回一个从指定位置开始持续到该字符串结尾的子串。因此，如以下调用方法：

```
str.substr(7)
```

将返回字符串 "world"。类似地，如果 n 是已知的，但指定的开始位置后面少于 n 个字符，substr 只返回原始字符串指定位置到字符串结尾的子串。

下面的函数调用将返回参数 str 的后半段子串，如果 str 的长度为奇数，则其子串将包含最中间的字符。

```
string secondHalf(string str) {
   return str.substr(str.length() / 2);
}
```

3.2.5 在一个字符串中进行搜索

有时，你会发现搜索一个字符串看它是否包含某个特定字符或子串是很有用的。为了完
133 成这样的搜索，string 类提供了一个名为 find 的方法，该方法有几种形式。最简单的调

门形式是：

```
str.find(search);
```

其中，search 是被搜索的内容，它可以是一个字符串，也可以是一个字符。当 find 方法被调用时，它会搜索 search 在字符串 str 第一次出现的索引位置。如果索引值被找到，则 find 返回搜索内容的开始的索引位置。如果搜索内容在字符串结束之前没有找到，则 find 返回常量 string::npos。与你在第 1 章看到过的常量不同，标识符 npos 被定义为 string 类的一部分，因此，一旦它在程序中出现，则要求必须用 string:: 限定符进行约束。

find 方法还有一个可选的第二参数，该参数指出从何处开始进行搜索。我们假设变量 str 包含字符串 "hello, world"，find 函数调用的两种版本将产生如下结果：

```
str.find('o')      → 4
str.find('o', 5)   → 8
str.find('x')      → str::npos
```

和字符串比较一样，搜索字符串的方法认为大小写字母是不同的。

3.2.6 循环遍历字符串中的所有字符

虽然 string 类提供的方法为你创造了基于从零开始的实现字符串应用的工具，但是通过利用已有的常见操作实现代码实例，通常能更容易地编写出程序。用编程术语来讲，这些示例性的程序被称为模式（pattern）。当你操作字符串时，其中一个最重要的模式就是循环遍历字符串中的字符，完成该功能所需要的代码如下：

```
for (int i = 0; i < str.length(); i++) {
   . . . body of loop that manipulates str[i] . . .
}
```

在上述代码运行的每次循环中，表达式 str[i] 都选取字符串中的第 i 个字符。因为循环的目的是为了处理字符串中的每个字符，所以循环会一直持续直到 i 的值达到字符串的长度为止。因此，可通过下面的函数来计算一个字符串中空格的数目：

```
int countSpaces(string str) {
   int nSpaces = 0;
   for (int i = 0; i < str.length(); i++) {
      if (str[i] == ' ') nSpaces++;
   }
   return nSpaces;
}
```

对于某些应用，需要反向循环遍历一个字符串，即从字符串的最后一个字符开始，反向迭代直至到达到字符串中的第一个字符。这种反向迭代需要以下的 for 循环语句：

```
for (int i = str.length() - 1; i >= 0; i--)
```

此时，字符串索引值 i 是从最后的索引位置开始，即开始时 i 的值等于字符串的长度减 1，然后它在每次循环中递减，直至减小到索引值为 0。

假如你已经理解了 for 语句的语法和语义，根据第一原则，你可以很容易地在每一次该模式出现时判断其中每一次迭代的情况。但是，这样做会极大地降低你的编码效率。这些迭代模式必须熟记，你不应该浪费任何时间去回想分析它们。当你确认你需要循环遍历字符串中的字符时，你的思维必须能很快地将迭代过程转化为以下代码：

```
for (int i = 0; i < str.length(); i++)
```

对于某些应用而言，你需要对基本迭代模式做出修改，使得开始和结束的索引位置不同。例如，下面的函数检测一个字符串是否以特定的前缀开始：

```
bool startsWith(string str, string prefix) {
   if (str.length() < prefix.length()) return false;
   for (int i = 0; i < prefix.length(); i++) {
      if (str[i] != prefix[i]) return false;
   }
   return true;
}
```

这段代码的开始部分是检测并确保 str 的长度不小于 prefix 的长度（万一出现这种情况，
135 结果一定为 false），然后循环遍历 prefix 中的字符而不是整个 str 字符串。

当你阅读 startsWith 函数的代码时，需要特别关注其中的两个 return 语句的位置。一旦发现 str 和 prefix 之间有一个不同的字符，在循环内部就立即返回 false。当代码检测完 prefix 的每个字符，并且在比较时没有发现任何的不同字符后，在循环的外部返回 true。在你阅读本书时，你会多次发现上述程序基本模式的实际应用例子。

在本章习题 1 中，你将有机会编写 startsWith 函数和类似的 endsWith 函数。虽然在 C++ 语言中，它们并不是标准 <string> 类库的一部分，但经验已经证明它们是非常有用的。它们是字符串函数库完美的候选者，你可以尝试通过应用第 2 章定义的 error 类库时使用的技术将这些函数包含在自定义类库中。Stanford 类库还包含一个能提供数个有用的字符串函数的接口，接口名为 strlib.h，我们将在 3.7 节对该接口进行更加详细的介绍。

3.2.7 通过连接扩展字符串

在学习字符串时，还有一个非常有意义的编程模式值得你去学习并铭记，这一模式通过一次一个字符，逐步创建一个完整字符串。循环结构本身将依赖这个应用，但是通过连续创建字符串的一般模式如下所示：

```
string str = "";
for (whatever loop header line fits the application) {
   str += the next substring or character;
}
```

作为一个简单的示例，下面的方法返回一个由 n 个 ch 字符拷贝组成的字符串：

```
string repeatChar(int n, char ch) {
   string str = "";
   for (int i = 0; i < n; i++) {
      str += ch;
   }
   return str;
}
```

上述 repeatChar 函数在某些场合中是很实用的，例如当你需要在控制台上输出某类章节分隔符的时候。要完成此目的，其中一个策略如下：

```
cout << repeatChar(72, '-') << endl;
```

136 它将打印由 72 个连字符号 "-" 组成的一行字符。

许多字符串处理函数将迭代和连接模式一起使用。例如，下面的函数将参数字符串进行逆序处理，例如，调用函数 reverse("desserts") 将返回字符串 "stressed"：

```
string reverse(string str) {
   string rev = "";
   for (int i = str.length() - 1; i >= 0; i--) {
      rev += str[i];
   }
   return rev;
}
```

3.3 <cctype> 库

由于字符由单个字符组成的，处理字符串中的单个字符而不是整个字符串的操作就显得非常必要。<cctype> 库提供了处理字符和字符的各种函数，其中最常见的函数如表 3-2 所示。

表 3-2 的第一部分定义了一组判断函数以检测字符串中的一个字符是否属于一个特定的类别。例如，如果 ch 是 '0' ~ '9' 之间范围内的一个数字字符，调用 isdigit(ch) 函数将返回结果 true。

表 3-2 在 <cctype> 库中的部分函数

检验字符类型的判断函数	
isalpha(ch)	如果 ch 是一个字母字符，则返回 true
isupper(ch)	如果 ch 是一个大写字母字符，则返回 true
islower(ch)	如果 ch 是一个小写字母字符，则返回 true
isdigit(ch)	如果 ch 是一个数字（'0' ~ '9'），则返回 true
isxdigit(ch)	如果 ch 是一个十六进制数字（'0' ~ '9'，'A' ~ 'F'，'a' ~ 'z'），则返回 true
isalnum(ch)	如果 ch 是一个字母数字混合的（alphanumeric）字符，则返回 true。一个字母数字意味着它要么是一个字母，要么是一个数字
ispunct(ch)	如果 ch 是一个标点符号，返回 true
isspace(ch)	如果 ch 是一个空白字符，则返回 true。' '（空格字符）、'\n'、'\r'、'\t'、'\v' 和 '\f' 都被认为是空白字符
isprint(ch)	如果 ch 是任意可打印的字符，则返回 true
大小写转换函数	
toupper(ch)	返回 ch 对应的大写字母（若 ch 不是一个字母则为 ch 本身）
tolower(ch)	返回 ch 对应的小写字母（若 ch 不是一个字母则为 ch 本身）

类似地，如果 ch 是任意一个在显示屏上显示为空白字符的字符，例如空格字符和制表符，则调用函数 isspace(ch) 将返回 true。表 3-2 第二部分的函数使得大写字母和小写字母之间的转换变得十分容易。例如，调用 toupper('a')，则返回字符 'A'。如果 toupper 或 tolower 函数的参数不是一个字母，函数返回它原来传入的参数，因此 tolower('7') 返回 '7'。

当处理字符串时，<cctype> 库中的函数的应用经常能够做到信手拈来。例如，如果参数 str 是一个非空的数字字符串，下面的函数返回 true，这意味着它代表一个整数：

```
bool isDigitString(string str) {
   if (str.length() == 0) return false;
   for (int i = 0; i < str.length(); i++) {
      if (!isdigit(str[i])) return false;
   }
   return true;
}
```

类似地，当忽略大小写时，如果字符串 s1 和 s2 相等，则下面的函数返回 true：

```
bool equalsIgnoreCase(string s1, string s2) {
   if (s1.length() != s2.length()) return false;
   for (int i = 0; i < s1.length(); i++) {
      if (tolower(s1[i]) != tolower(s2[i])) return false;
   }
   return true;
}
```

函数 equalsIgnoreCase 在忽略大小写情况下，一旦发现字符串 s1 和 s2 第一个不匹配的字符时，则返回 false，否则直到循环结束才返回 true。

3.4 修改字符串中的内容

与其他类似 Java 这样的语言不同，C++ 语言允许通过给字符串中的一个特定的索引位置赋新值来改变字符串中的字符。这使得你可以设计自定义字符串操作函数去改变一个字符串中的内容，就像删除（erase）、插入（insert）和替换（replace）方法所做的一样。然而，在大多
118 数情况下，最好编写函数以使它们能返回一个字符串的转换版本而不改变原有字符串的内容。

我们举一个例子说明对字符串内容进行修改的两种不同的方法。假设你想设计一个与 <cctype> 库中 toupper 函数对应的自定义函数，它等效地将字符串中的每个小写字符转换成大写字符。一种方法是实现一个会改变原来实参字符串中内容的过程，该实现方法如下所示：

```
void toUpperCaseInPlace(string & str) {
   for (int i = 0; i < str.length(); i++) {
      str[i] = toupper(str[i]);
   }
}
```

另一个替代的策略是编写一个函数，函数返回它的实参的大写版本拷贝，并且不改变原来的实参值。如果你同时使用迭代和连接模式，函数可能像下面这样：

```
string toUpperCase(string str) {
   string result = "";
   for (int i = 0; i < str.length(); i++) {
      result += toupper(str[i]);
   }
   return result;
}
```

上述修改字符串的策略有时非常高效，此外它更灵活，并且出现意外结果的可能性更低。然而，利用第一个程序代码，可将第二个程序代码进行改写，使其更加高效：

```
string toUpperCase(string str) {
   toUpperCaseInPlace(str);
   return str;
}
```

在上述这段代码实现中，C++ 自动复制实参字符串，因为参数是值传递的。鉴于 str 不再连接到调用域的参数字符串，在某些地方修改它然后返回一个复制字符串给它的调用者是完全可以接受的。

3.5 遗留的 C 风格字符串

早期 C++ 语言的成功部分归因于其包含了 C 语言作为子集，使得它能从一种语言逐渐过渡到另一种语言。然而，这种设计决策意味着 C++ 依然包含 C 语言的某些特性，而 C 语言的这些特性在现代面向对象程序设计语言中不再有意义，尽管如此，我们仍需保留 C++ 语言的兼容性。 139

在 C 语言中，字符串是以底层的字符数组来实现的，不提供 string 类中的高级机制。遗憾的是，使 C++ 和 C 保持兼容这项决策导致 C++ 必须支持两种风格。例如，string 字面值就是基于传统的 C 语言风格实现的。在大部分情况下，你都可以忽略其历史细节，因为当编译器能够确定你想要的是一个 C++ 字符串时，C++ 语言会自动将一个字符串字面值转换成一个 C++ 字符串。如果你用下面这行代码初始化一个 string 对象：

```
string str = "hello, world";
```

C++ 会自动将 C 风格的字符串字面值 "hello, world" 转换成一个 C++ 的 string 类型对象，因为你已经告诉编译器 str 是一个 string 类型的变量。但是，C++ 不允许你编写类似以下这样的声明语句：

```
string str = "hello" + ", " + "world";
```



即使该声明看起来和上条语句好像都会产生相同的结果。但此处的问题是该代码版本尝试将 + 操作符应用到非 C++ string 对象的字符串字面值上。

如果你需要回避这一问题，你可以通过显确调用 string 类对字符串字面值的处理函数，将一个字符串字面值转换成一个字符串对象。例如，下面的这行代码能正确地将 "hello" 转换成一个 C++ 的 string 对象，然后采用连接方式完成 string 对象初始值的计算：

```
string str = string("hello") + ", " + "world";
```

C++ 中字符串的两种不同表示产生了另外一个问题，即一些 C++ 库要求使用 C 风格的字符串来代替实现化的 C++ string 类。如果你在一个使用 C++ 字符串应用的上下文中使用这些抽象库函数，你必须在某个地方将 C++ 字符串对象转换成其对应的 C 风格字符串。这种转换非常简单：你所需要做的就是使用 C++ 版本字符串作为参数调用 c_str 方法，进而获取它的等价 C 风格字符串。然而，更重要的问题是，字符串的两种不同表示我们都需要掌握，这增加了 C++ 概念的复杂性，使得它更难以学习。

3.6 编写字符串应用程序

到目前为止，你看到的字符串例子都非常简单，尽管它们对于阐明特定字符串函数工作原理是很有用的，但这些例子都不能使你更清晰地了解如何编写一个有意义的字符串处理应用。本节通过开发两个操作字符串数据的应用解决上述缺陷。 140

3.6.1 回文识别

回文（palindrome）是指其字母排列正序与倒序均一致的词语，例如单词“level”或“noon”。本节的目的是编写一个判断函数以检测一个字符串是否属于回文。调用 isPalindrome("level") 应该返回 true；调用 isPalindrome("xyz") 应该返回 false。

和大多数编程问题一样，这里有解决该问题的几种合理策略。以我的经验，大部分学生可能首先尝试的方法是使用一个 for 循环依次读取字符串前半部分每一个索引位置上的字符。在每个位置上，代码将检测该字符是否与出现在字符串末尾对应对称位置的字符匹配。采取这种策略的代码如下：

```
bool isPalindrome(string str) {
   int n = str.length();
   for (int i = 0; i < n / 2; i++) {
      if (str[i] != str[n - i - 1]) return false;
   }
   return true;
}
```

考虑到使用本章遇见过的字符串处理函数，你也可以使用下面更简洁的形式编写 isPalindrome 函数：

```
bool isPalindrome(string str) {
   return str == reverse(str);
}
```

在上述两种实现方式中，第一个版本更为有效。第二个版本必须构造一个新的字符串，且其中的字符与原字符串中的字符顺序是相反的。更糟糕的是，它通过一次连接一个字符的方式，创建了一个和原字符串等长的临时字符串。第一个版本不需要创建任何字符串。它通过选择和比较字符串中的字符完成其功能，这被证明是一种低代价的运算。

除了二者在效率上的不同外，第二种编码也有许多优点，特别是对编程新手而言，可将其作为参考范例。其主要优点为：一方面，它通过使用 reverse 函数重用了已有的代码；另一方面，第一个版本需要计算字符串中字符的索引位置，而第二个版本则隐藏了涉及这方面的编程复杂性。对于大部分学生而言，它至少要花费一分钟或两分钟才明白为什么代码要包含选择表达式 str[n - 1 - i]，或者为什么它要在 for 循环检验中使用 < 操作符，而不是 <=。相比之下，以下这一行代码：

141
```
   return str == reverse(str);
```

读起来几乎和英语一样流畅：如果一个字符串和它相反顺序的字符串相等，则它是一个回文。

尤其是当你正在学习编程时，致力于程序的简洁性比关注其执行效率更为重要。鉴于现在计算机的速度，牺牲几个机器周期来使程序更易于理解是值得的。

3.6.2 将英语翻译成儿童黑话

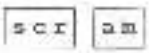
为了使你更多地了解如何实现对字符串操作的应用，本节列举了一个 C++ 程序，它读取用户输入的一行文字，然后将这行文字的每个单词从英语翻译成儿童黑话（Pig Latin），这是一种拼读的语言，世界上说英语的大部分孩子都对其熟知。在儿童黑话中，单词是通过应用下面的规则从它们对应的英语单词中构造而成：

1. 如果单词中不含元音字母，不作任何翻译。这意味着儿童黑话的单词和原单词一样。

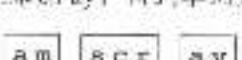
2. 如果单词以元音字母开始，则翻译出的儿童黑话包括原单词加上其后缀 way。

3. 如果单词以辅音字母开始，提取该辅音字符子串直到遇见第一个元音字母，移动该集的辅音字母到单词的结尾，然后添加后缀 ay，这样就形成了翻译后的儿童黑话。

下面举例说明，假设英语单词是 scram，因为该单词是以辅音字母开始的，故可将它分成两部分，一部分包括第一个元音之前的字母，另一部分包括该元音和剩余的字母：



然后你交换这两部分的位置并在末尾添加 ay，得到单词如下：



因此 scram 的儿童黑话单词是 amscray。对于一个以元音开始的单词，例如 apple，你只需简单地添加 way 到单词末尾，然后得到单词 appleway。图 3-2 给出了 PigLatin 的程序代码。主程序从用户处读取一行文本然后调用 lineToPigLatin 将这一行输入翻译成儿童黑话。然后函数 lineToPigLatin 调用 wordToPigLatin 将每个单词转换成对应的儿童黑话单词，不是该单词一部分的字符被直接复制到输出行，这样标点符号和间隔仍保持不变影响。 142

```
/*
 * File: PigLatin.cpp
 *
 * This program converts lines from English to Pig Latin.
 * This dialect of Pig Latin applies the following rules:
 *
 * 1. If the word contains no vowels, return the original
 *    word unchanged.
 *
 * 2. If the word begins with a consonant, extract the set
 *    of consonants up to the first vowel, move that set
 *    of consonants to the end of the word, and add 'ay'.
 *
 * 3. If the word begins with a vowel, add 'way' to the
 *    end of the word.
 */

#include <iostream>
#include <string>
#include <cctype>
using namespace std;

/* Function prototypes */

string lineToPigLatin(string line);
string wordToPigLatin(string word);
int findFirstVowel(string word);
bool isVowel(char ch);

/* Main program */

int main() {
   cout << "This program translates English to Pig Latin." << endl;
   string line;
   cout << "Enter English text: ";
   getline(cin, line);
   string translation = lineToPigLatin(line);
   cout << "Pig Latin output: " << translation << endl;
   return 0;
}
```

图 3-2 将英语翻译成儿童黑话的程序

```
/*
 * Function: lineToPigLatin
 * Usage: string translation = lineToPigLatin(line);
 * ------------------------------------------------
 * Translates each word in the line to Pig Latin, leaving all other
 * characters unchanged.  The variable start keeps track of the index
 * position at which the current word begins.  As a special case,
 * the code sets start to -1 to indicate that the beginning of the
 * current word has not yet been encountered.
 */

string lineToPigLatin(string line) {
   string result;
   int start = -1;
   for (int i = 0; i < line.length(); i++) {
      char ch = line[i];
      if (isalpha(ch)) {
         if (start == -1) start = i;
      } else {
         if (start >= 0) {
            result += wordToPigLatin(line.substr(start, i - start));
            start = -1;
         }
         result += ch;
      }
   }
   if (start >= 0) result += wordToPigLatin(line.substr(start));
   return result;
}

/*
 * Function: wordToPigLatin
 * Usage: string translation = wordToPigLatin(word);
 * ------------------------------------------------
 * Translates a word from English to Pig Latin using the rules
 * specified in the text.  The translated word is returned as the
 * value of the function.
 */

string wordToPigLatin(string word) {
   int vp = findFirstVowel(word);
   if (vp == -1) {
      return word;
   } else if (vp == 0) {
      return word + "way";
   } else {
      string head = word.substr(0, vp);
      string tail = word.substr(vp);
      return tail + head + "ay";
   }
}

/*
 * Function: findFirstVowel
 * Usage: int k = findFirstVowel(word);
 * ------------------------------------
 * Returns the index position of the first vowel in word.  If
 * word does not contain a vowel, findFirstVowel returns -1.
 */

int findFirstVowel(string word) {
   for (int i = 0; i < word.length(); i++) {
      if (isVowel(word[i])) return i;
   }
   return -1;
}

/*
 * Function: isVowel
 * Usage: if (isVowel(ch)) . . .
 * -----------------------------
 * Returns true if the character ch is a vowel.
 */
```

图 3-2 （续）

```
int findFirstVowel(string word) {
   for (int i = 0; i < word.length(); i++) {
      if (isVowel(word[i])) return i;
   }
   return -1;
}

/*
 * Function: isVowel
 * Usage: if (isVowel(ch)) . . .
 * ------------------------------
 * Returns true if the character ch is a vowel.
 */
```

图 3-2 （续）

上述示例程序的运行结果如下：



图 3-2 中 lineToPig_atin 和 wordToPigLatin 的代码实现值得仔细回味。函数 lineToPigLatin 在输入中找到单词的分界线，这提供了一种有用的模式以将一个字符串分割成单个单词。函数 wordToPigLatin 使用 substr 提取英语单词片段，然后用连接方式将它们以儿童黑话的形式组合在一起。在第 6 章，你就会学到一个更通用的称为记号扫描器（token scanner）的机制，它可以将一个字符串分成逻辑上相连的多个部分。 143 ⋮ 145

3.7 strlib.h 库

正如我多次在本书中所强调的，本章包含在库中的几个函数看起来很完美。一旦你将这些函数应用到实际编程中，你会感觉到在需要实现相同运算的其他应用场合中不使用它们是很浪费的。然而像 wordToPigLatin 这样的函数在其他地方不可能出现，而类似 toUpperCase 和 startsWith 这样的函数你会经常使用。因此为了避免重写和复制粘贴代码，将这些函数放在一个库中是很有意义的，这可以在你需要使用它们时提供极大的便利。

在本书中，Stanford 类库包含了一个接口，接口名为 strlib.h，它提供了如表 3-3 所示的函数。你可以在该表中看到若干函数，它们曾在本章中定义过。在习题中，你将有机会补全包括 endsWith 和 trim 在内的其他定义。表 3-3 中的头四个函数都与将数值转换成字符串形式相关，要求掌握的知识已超出了你目前掌握的 C++ 知识范围，但这只是暂时的。在第 4 章，你将学会如何使用一个新的数据类型，称作 stringstream 类，它能使这些函数的实现变得更加简单。

表 3-3 strlib.h 接口提供的函数

integerToString(*n*)	将整数 *n* 转换成对应的数字字符串
stringToInteger(*str*)	将数字字符串 *str* 转换成对应的整数
realToString(*d*)	将浮点数 *d* 转换成对应的字符串
stringToReal(*str*)	将实数字符串 *str* 转换成对应的实数值
toUpperCase(*str*)	返回一个新字符串，字符串中所有的小写字符将被换成它们的大写字符

（续）

`toLowerCase(str)`	返回一个新字符串，字符串中的内容是将 str 中所有的大写字符转换成它们的小写字符
`equalsIgnoreCase(s1, s2)`	在忽略大小写的情况下，如果 s_1 和 s_2 相等，则返回 `true`
`startsWith(str, prefix)`	若字符串 *str* 以指定的前缀 *prefix* 开始，其中，*prefix* 可以是一个字符串，也可以是一个字符，则返回 `true`
`endsWith(str, suffix)`	若字符串 *str* 以指定的后缀 *suffix* 结束，其中，*suffix* 可以是一个字符串，也可以是一个字符，则返回 `true`
`trim(str)`	从参数字符串 *str* 的开头至结尾处，删除其中所有的空白字符，然后返回一个新的字符串

146

本章小结

在本章，你已经学习了如何使用 <string> 类库，利用该类库你可以编写出字符串操作函数，并且不用担心底层表示的细节问题。本章的重点包括：

- <string> 类库提供了一个 string 类，它用于代表一个字符序列。尽管 C++ 语言也包括一个更原始的类型去维护对于 C 语言的兼容性，但在编写程序时最好使用 string 类。
- 如果你使用 >> 提取操作符读取字符串数据，输入将会在第一个空白字符处停止。如果你想从用户那里读取一整行文本，使用 C++ 标准库提供的 getline 函数会更好。
- string 类中提供的最常用方法显示在表 3-1 中。因为 string 是一个类，方法都使用接收方语法来代替传统的函数形式。因此，为了获取存储在 str 变量中字符串的长度，你需要调用 str.length()。
- string 类提供的几种方法破坏性地修改了接收字符串。给予用户自由的权限去使用这些能够改变一个对象内部状态的方法，这使得对象的完整性更难以保护。因此，本书中的程序将最大限度地减少这些方法的使用。
- string 类使用操作符重载以简化许多常见的字符串操作。对于字符串来说，最重要的操作符是 +（连接）操作符、[]（选择）操作符和关系操作符。
- 循环遍历一个字符串中字符的标准模式是：

```
for (int i = 0; i < str.length(); i++) {
   . . . body of loop that manipulates str[i] . . .
}
```

- 通过连接以增长一个字符串的标准模式是：

```
string str = "";
for (whatever loop header line fits the application) {
   str += the next substring or character;
}
```

147

- <cctype> 库提供了几种处理单个字符的函数。其中最重要的函数显示在表 3-2 中。

复习题

1. 字符串和字符的区别是什么？
2. 判断题：如果你执行以下代码

```
string line;
cin >> line;
```

程序将从用户那里读取一输行数据，然后将其存储在变量 line 中。

3. getline 函数的哪个参数是引用传递的？

4. 方法和自由函数之间的区别是什么？

5. 判断题：在 C++ 语言中，通过调用 length(str)，你可以判断存储在变量 str 中字符串的长度。

6. 如果你调用 s1.replace(0,1,s2)，那个字符串是接收方？

7. 当 + 操作符被作用到两个字符串运算对象时，它的作用是什么？

8. 当 C++ 计算表达式 s1<s2 时，string 类使用什么规则来比较字符串值？

9. 从字符串中选取一个单独字符时，本章描述了哪两种语法形式？在它们的实现中，这两种语法形式在表示上有何不同？

10. 当你从 C++ 字符串中选取一个单独的字符时，你可以使用 at 方法，也可以使用标准的下标操作符。其中，索引是放在方括号中的。请问从用户的角度看，这两种选择的区别是什么？

11. 判断题：如果你将字符串变量 s1 赋值给字符串变量 s2，字符串复制了字符，这样，一个字符串中字符的后续改变将不会影响到另一个字符串中的字符。

12. 判断题：一个字符串中的索引位置从 0 开始，并且扩展到字符串的长度减 1 位置。

13. substr 方法的参数是什么？如果你省略第二个参数，将会发生什么？ 148

14. 描述 compare 方法如何使用返回值表示两个字符串的相对顺序。为什么这个方法在实际中很少用到？

15. find 方法返回什么值来表示搜索的字符串没有出现？

16. 对于 find 方法，第二个可选参数的意义是什么？

17. 假设你这样声明和初始化变量 s 和 t：

```
string s = "ABCDE"
string t = "";
```

对于上述声明，下面每个调用的结果是什么？

a. `s.length()`	f. `s.replace(0, 2,"x")`
b. `t.length()`	g. `s.substr(0,3)`
c. `s[2]`	h. `s.substr(4)`
d. `s + t`	i. `s.substr(3,9)`
e. `t += 'a'`	j. `s.substr(3,3)`

18. 循环遍历字符串中每个字符这种程序的设计模式叫什么？

19. 如果你想从相反的顺序循环遍历字符串中的字符，即从字符串中的最后一个字符开始至第一个字符结束，习题 18 中的模式要怎样改变？

20. 通过连接增长一个字符串的模式是什么？

21. 在 <cctype> 库中，下面的每个调用将产生什么结果？

a. `isdigit(7)`	d. `toupper(7)`
b. `isdigit('7')`	e. `toupper('A')`
c. `isanum(7)`	f. `tolower('A')`

22. 为什么 C++ 同时支持 string 类和更原始的字符串类型？

23. 你怎样将一个原始的字符串值转换成一个 C++ 的字符串？你如何指定以相反的方向进行转换？

习题

1. 实现函数 endsWith(str, suffix)。函数的功能为：若 str 以 suffix 结束，则返回 true。类似于它对应的 startsWith 函数，endsWith 函数应该允许第二个参数是一个字符串或是一个字符。 149

2. strlib.h 函数提供了一个返回新的字符串的 trim(str) 函数，产生的新字符串是通过从头到尾删除 str 中的所有空白字符形成的。请编写对应的程序。

3. 不使用内置的字符串方法 substr，实现一个自由函数 substr(str, pos, n)。该函数返回从 str 的 pos 位置开始，最多包含 n 个字符的子串。确保你的函数正确地应用了以下规则：
   - 如果 n 缺失或者大于字符串的长度，子串应该延续到原字符串的末尾。
   - 如果 pos 大于字符串的长度，substr 应该使用一个合适的消息去调用 error。

4. 实现一个返回字符串的函数 capitalize(str)，即将 str 的首字符转换成大写（如果 str 的首字符是一个字母），其他所有的字母转换成小写的对应的字符串。str 中的非字母字符不受影响。例如，capitalize("BOOLEAN") 和 capitalize("boolean") 两个都返回字符串 "Boolean"。

5. 在大部分单词游戏中，单词中的每个字母是根据它的分数值记录得分的，分数值是和它在英语单词中的使用频率成反比的。在 Scrabble™ 中，分数分配如下：

分 数	单 词
1	*A*、*E*、*I*、*L*、*N*、*O*、*R*、*S*、*T*、*U*
2	*D*、*G*
3	*B*、*C*、*M*、*P*
4	*F*、*H*、*V*、*W*、*Y*
5	*K*
8	*J*、*X*
10	*Q*、*Z*

例如，在 Scrabble 中，单词 "FARM" 值9分：F 是4分，A 和 R 每个是1分，M 是3分。编写一个程序，在 Scrabble 中，程序读取单词并输出它们的得分，要求不计算游戏中出现的其他任何收益。计算分数时，你应该忽略除了大写字母的任何字符。特别是小写字母被认为代表空白，它可以代表任何字母但是其得分为0。

6. 首字母缩略词是一个通过结合一系列单词的初始字母而形成的单词。例如，单词 "scuba" 是一个首字母缩略词，它是由 "self-contained underwater breathing apparatus" 的第一个字母组成的。同理，AIDS 是由 "Acquired Immune Deficiency Syndrome" 组成的一个首字母缩略词。编写一个函数 acronym，它读取一个字符串，然后返回由该字符串形成的首字母缩略词。为了确保函数识别连字符的复合词（如 self-contained）算作是两个单词，它必须定义一个单词的首字母是任意的字母字符，它可以出现在一个字符串的开始位置，也可以出现在一个非字母字符后面。 [150]

7. 编写一个函数：

```
string removeCharacters(string str, string remove);
```

该函数返回一个 string 对象，str 中的字符删除 remove 中的字符后构成了该 string 对象。例如，若你调用以下语句：

```
removeCharacters("counterrevolutionaries", "aeiou")
```

函数应该返回 "cntrrvltnrs"，它是一个原始字符串删除它所有的元音字母后的一个新字符串。

8. 修改你在习题7中的问题解决方案，代替使用一个返回新字符串的函数，即定义一个 removeCharactersInPlace 函数，函数的作用是从字符串中删除若干字母，字符串是作为第一个参数传入函数中。

9. 浪费时间练习写这句为声音和字符为历史（也称为骚扰）在英语口头表达课的…

——查尔斯，1941

20世纪早期，英国和美国都有一个相当重要的问题，即简化拼写英语单词的规则，这一直是一个难题。作为这项运动的一部分，提出的一个建议是消除双写字母，这样bookkeeper会写作bokeper，并且committee会变成comite。编写一个返回新字符串的函数removeDoubledLetters(str)，在该函数中，str中任何重复的字符都被一个单独的该重复字符所代替。

10. 编写一个函数：

```
string replaceAll(string str, char c1, char c2);
```

它返回一个str的复制字符串，其中str中每一个c1都用c2代替。例如，调用

```
replaceAll("nannies", 'n', 'd');
```

应该返回"daddies"。 151

一旦你编写和测试完这个函数，编写一个重载的函数版本：

```
string replaceAll(string str, string s1, string s2);
```

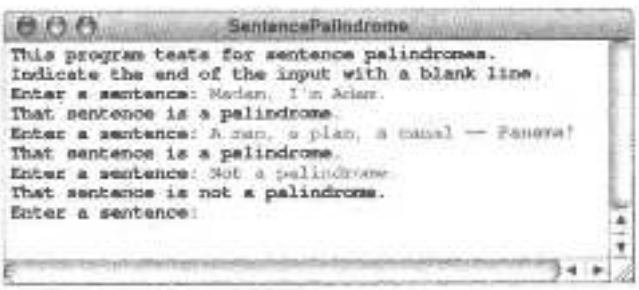
该函数将str中每一个s1字符串都用s2字符串代替。

11. 通过忽略标点符号和字母大小写的差异，回文的概念通常被扩展到整个句子。例如以下句子：

Madam, I'm Adam.

这是一个回文的句子，因为如果你只看字母并忽略大小写字母的区别，顺序读和倒序读是一样的。

编写一个判断函数isSentencePalindrome(str)，如果字符串str和回文句子的定义相匹配，则该函数返回true。你应该能够用该函数去编写一个主程序并产生以下的输出结果：



12. 编写一个函数createRegularPlural(word)，遵循以下标准的英语规则，函数返回word的复数形式：

a. 如果单词以s、x、z、ch或sh结尾，在单词末尾添加es。

b. 如果单词以y结尾，y前面有一个辅音，修改y为ies。

c. 其他情况，在单词末尾添加s即可。

编写一个测试程序，并设计一组测试示例来证实你程序能正常运行。

13. 像其他大部分语言一样，英语包含两种类型的数字：基数（cardinal number）（例如one、two、three和four）用在计数中，序数（ordinal number）（例如first、second、third和fourth）用来表示一个序列的位置。在本书中，序数通常用数字后面紧跟其相应序数的英语单词中的最后两个字母来表示。 152
因此，序数first、second、third和fourth经常表示为1st、2nd、3rd和4th。然而，序数11、12和13是11th、12th和13th。设计一种规则判断每个数字后面是否应该添加后缀，并且使用该规则编写函数createOrdinalForm(n)。该函数返回一个作为字符串出现的数字n的序数形式。

14. 在正式文稿（如论文、报纸和杂志等）中书写大的数字时，至少在美国，传统的方式是用逗号将数字分成三组。例如，数字一百万通常写成下面的形式：

1,000,000

为了使程序的更容易用这种方式显示数字，实现函数：

```
string addCommas(string digits);
```

该函数读取一个代表数字的十进制数字字符串，然后返回从右开始、每三位数字插入一个逗号所形成的字符串。例如，如果你执行以下主程序：

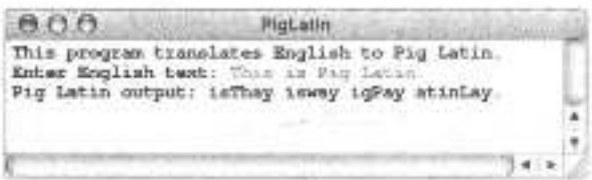
```
int main() {
   while (true) {
      string digits;
      cout << "Enter a number: ";
      getline(cin, digits);
      if (digits == "") break;
      cout << addCommas(digits) << endl;
   }
   return 0;
}
```

addCommas 函数的实现应该能产生以下示例中的运行结果：

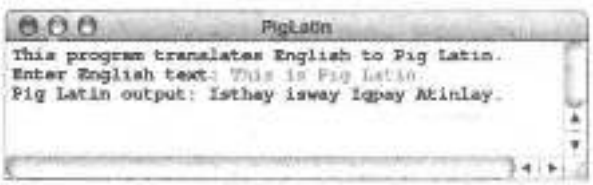


[153]

15. 图 3-2 中的 pigLatin 函数表现的很奇特。如果你输入一个字符串，该字符串包含以一个大写字母开始的单词，例如，如果你要利用句子的第一个单词来为儿童黑话讲者的名字，你应该能理解下面的输出：



重写 wordToPigLatin 函数，使得任何一个以大写字母开始的单词在儿童黑话中仍以一个大写字母开始。因此，在程序中做出必要的改变后，输出应该如下所示：



16. 大部分人（至少在说英语的国家），在他们生活中的某些地方已经玩过儿童黑话游戏。也有一些其他被发明的“语言”使用一些英语的简单转换创建单词。其中一种这样的语言被称作“Obenglobish”，在这种语言中，通过在英语单词的元音字母（a、e、i、o 和 u）前添加字母 ob 来创建单词。例如，在这种规则下，单词 english 中 e 和 i 前面添加字母 ob 形成 obengloish，这就是该语言获取它名字的方式。

在官方的 Obenglobish 中，ob 字符只添加在发出音来的元音字母前，这意味着一个像 game 的单词会变成 gobame，而不是 gobamobe，因为最后的 e 是不发音的。虽然完美地实现上述规则是不可能的，通过采用这样的规则，你可以完成一项完美的工作。规则是在英语单词中，ob 必须添加到每个元音字母前面，除了以下两级：

- 在其他元音字母后面的元音
- 出现在单词末尾的 e

154

编写一个函数 obenglobish，它读取一个英语单词，然后使用上述所给的翻译规则，返回它等价的 obenglobish 单词。例如，如果你在主程序中使用以下函数：

```
int main() {
   while (true) {
      string word = getLine("Enter a word: ");
      if (word == "") break;
      string trans = obenglobish(word);
      cout << word << " -> " << trans << endl;
   }
   return 0;
}
```

你应该能够产生下面的示例运行结果：



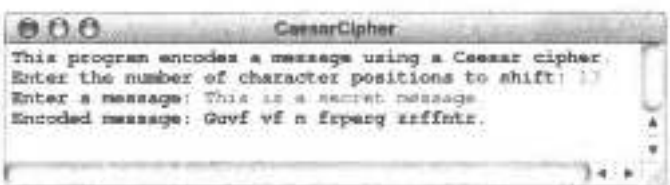
17. 如果你在青年加过密码，很有可能在某些地方用过循环码（cyclic cipher），它又经常被称为凯撒密码（Caesar cipher）。因为罗马历史学家苏维托尼乌斯记录到盖乌斯·尤利乌斯·凯撒使用过这项技术，运用这项技术你可以将原始消息中的每个字母用字母表中出现在它前面一个固定距离的字母来代替。作为一个例子，假设你想通过将每个字母向前移动三个位置来编码一个消息，运用这个密码技术，每个 A 变为 D，B 变为 E，以此类推。如果你到达了字母表的末尾，处理则周又回到开头，所以 X 变成 A，Y 变成 B，Z 变成 C。

为了实现一个凯撒密码，你首先应该定义以下函数：

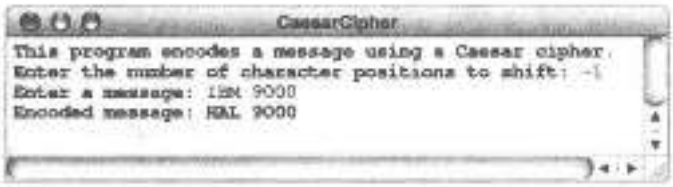
```
string encodeCaesarCipher(string str, int shift);
```

该函数返回一个新的字符串，它是将 str 中的每个字母转换成与它相隔 shift 处的字母形成的，如果有必要，通环回到字母表的开头。在你实现了 encodeCaesarCipher 函数后，编写程序产生以下示例的运行结果：

155



注意到翻译只运用于字母，其他任何字符都不经修改输出。此外，字母大小写不产生影响：即大小写字母被原样输出。你也应该这样编写你的程序。一个负的 shift 值意味着字母向着字母表开头的方向进行移动，而不是向结尾方向，正如以下示例的运行结果所表明的：

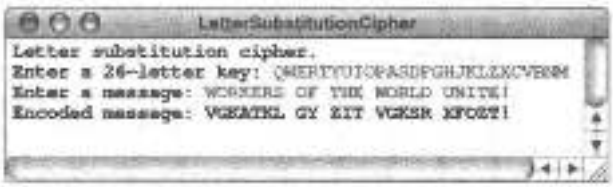


18. 尽管它们确实很简单，凯撒密码也极容易被破解。毕竟，只有25个字符可移动。如果你想破解一个凯撒密码，你只需尝试所有25种可能，然后观察哪一种翻译能将原始的消息转换成可读的。一个更好的策略是原始消息中的每个字母都被任意字母代替而不是被原始消息中相隔固定距离的字符代替。这样的话，编码运算的关键是一个翻译表，它展示了26个字母中每一个改变后的加密形式。这样的一个编码策略称作字母替换密码（letter-substitution cipher）。

字母替换密码的关键是一个由26个字符构成的字符串，该字符串依次指出了字母表中每个字符的翻译。例如，关键字"QWERTYUIOPASDFGHJKLZXCVBNM"表示编码过程应该使用下面的翻译规则：

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
| | | | | | | | | | | | | | | | | | | | | | | | |
Q W E R T Y U I O P A S D F G H J K L Z X C V B N M
```

156 采用字母替换密码编写一个能实现加密的程序。该程序应能够复制以下示例中的运行结果：



19. 使用前面习题中描述的字母替换密码的定义，编写函数 `invertKey`，该函数读取一个密钥，然后返回经过加密后对应的解密的报文。

20. 人类被基因控制。

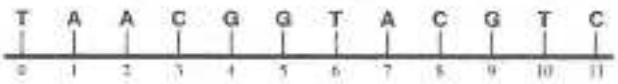
——电影 GATTACA 的标语，1997

所有生物的遗传密码都携带在它们的DNA（一种拥有非凡复制自身结构能力的分子）中。DNA分子的双螺旋结构包含两条相似且互相缠绕的化学长链。DNA的复制能力来源于它的4种基本组成成分：腺苷、胸嘧啶、鸟嘌呤和胸腺嘧啶。它们只能通过下面的方式互相组合起来：

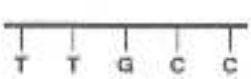
- 一条链上的胸嘧啶只和另一条链上的鸟嘌呤相连接，反之亦然。
- 腺苷只和胸腺嘧啶相连接，反之亦然。

生物学家用首字母缩写来表示基本成分的名字：A，C，G和T。

在细胞内部，一条DNA链表现的像一个模板，其他DNA链依附于这条链。作为一个例子，假设你有下面的DNA链，其中每个基本成分的位置如同它在一个C++字符串中一样用数字记录。

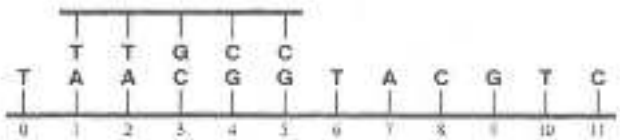


在这道习题中，你的任务是判断一条短的DNA链是否能依附于这条长链上。例如，如果你正好找到一条匹配的链：

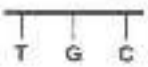


157

DNA 的规则表明只在位置 1 处，这条链可以与长链结合：



相比之下，以下这条链：



可以在位置 2 或位置 7 处与长链相匹配。

编写一个函数：

```
int findDNAMatch(string s1, string s2, int start = 0);
```

函数返回 DNA 链 s1 在 s2 匹配的第一个位置。正如 string 类中的 find 方法一样，可选择的 start 参数表示搜索应该开始的索引位置。如果不匹配，findDNAMatch 应返回 −1。 158

第4章
Programming Abstractions in C++

流　类

我们将不会满足，除非正义像水及泉水一样喷涌。

——牧师马丁·路德·金，《我有一个梦》，1963.8.28
（paraphrasing Amos 5:24）

159

自从 HelloWorld 出现在第1章起，本书中的程序已经使用了一种称为流（stream）的重要数据结构，C++利用这种数据结构管理来自或流向某个数据源的信息流。在前面的章节中，你已经使用了 << 和 >> 操作符，并且有机会使用 <iostream> 库提供的三种标准流：cin、cout 和 cerr。然而，你只抓住了你所能利用的标准流功能的表面。基于到目前为止你所见到的简单的 C++ 程序，为了提升编程水平，你必须学习更多关于流的知识，并且学会如何使用它们创建更复杂的应用。本章首先提供更多关于 << 和 >> 操作符的特征。之后介绍数据文件（data file）的概念，并且展示如何实现文件处理的应用。最后，本章通过探讨 C++ 流类的结构来对其进行总结，它可作为面向对象语言继承层次的一个代表性的实例。

4.1　格式化输出

在 C++ 中，产生格式化输出最简单的方式是使用 << 操作符。这个操作符称为**插入操作符**（insertion operator），因为它有着将数据插入到一个流的作用。该操作符的左操作数是输出流；右操作数是你想插入到该流中的数据。<< 操作符被重载使得右操作数可以是一个字符串或其他任意类型的值。如果右操作数不是一个字符串，在将它发送给输出流之前，<< 操作符会将其转换成字符串形式。这种特性使得它更易于显示变量值，因为 C++ 能自动处理输出转换。

通过使用 << 操作符返回流值，C++ 使得产生输出更加方便。正如你在本书几个例子中所看到的，这种设计决策使得将几个输出运算连接成一个链成为可能。例如，假设你想在一个输出行中显示变量 total 的值，变量以一些文本开始，目的是告诉用户这个值代表的意义。在 C++ 中，以下表达式：

```
cout << "The total is "
```

表示复制字符串 "The total is" 中的字符到 cout 流中。为了插入 total 值的十进制表示到流上，你所需要做的就是将变量 total 链接到 << 操作符上：

```
cout << "The total is " << total
```

160 这个表达式会得到预期的结果，因为 << 操作符返回流。因此，第二个 << 操作符的左操作数就是 cout，这意味着 total 的值会在输出流中显示出来。最后，可以使用 << 操作符的另一个实例：通过插入 endl 值来表示这一输出行的结束：

```
cout << "The total is " << total << endl;
```

如果 total 的值是 42，则在控制台上的输出结果如下：



即使从本书一开始，你就一直像这样使用语句，了解到 << 操作符通过该表达式传送 cout 值，正如它沿着插入操作符链移动一样，将会帮助你领会它在 C++ 中输出是如何工作的。

尽管 endl 看起来好像是一个简单的字符串常量，其值常常用来表示一个输出行的结束，事实上它是 C++ 中称为流操纵符（manipulater）的一个实例，流操纵符仅是一种用于控制格式化输出的一种特定类型值的有趣名称。C++ 类库提供了各种各样的流操纵符，可以使用它们来指定输出值的格式，它们中最常见的都显示在表 4-1 中。当在程序中包含了 <iostream> 库头文件时，这些流操纵符的绝大部分都是自动可用的。唯一例外是读取参数的流操纵符，例如 set (n)、setprecision (digits) 和 setfill (c)。为了使用这些流操纵符，你还需要包含 <iomanip> 库头文件。

流操纵符典型的作用是通过设置输出流的属性值来改变输出序列的格式。正如表 4-1 中逐一列出的各条目所阐明的那样，某些流操纵符的作用是短暂的（transient），这意味着它们只影响下一个输出的数据值。然而，大部分流操作符的作用是持久的（persistent），这意味着其作用一直有效，直到它们被明确地改变为止。

关于流操纵符一个最常见的应用就是指定输出域宽度以支持表格输出。例如，假设你想重写第 1 章的 PowersOfTwo 程序使得表格中的数字是纵向对齐的。为此，你所需要做的就是在输出语句中添加合适的流操纵符，如下所示：

```
cout << right << setw(2) << i
                << setw(8) << raiseToPower(2, i) << endl;
```

161

表 4-1　输出流操纵符

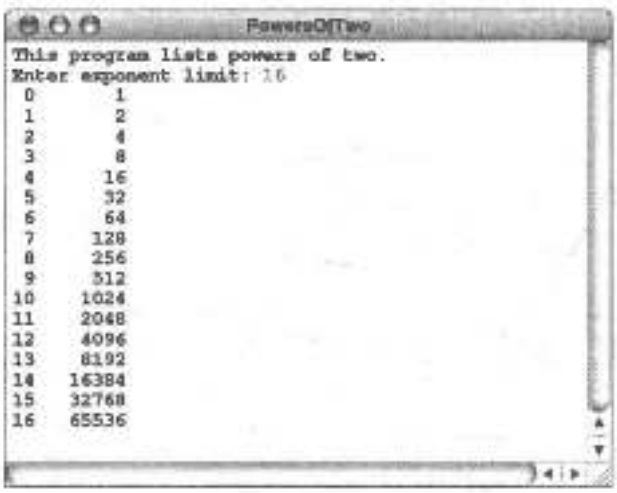
endl	将行结束序列插入到输出流，并确保输出的字符能被写到目的流中
setw(n)	将下一个输出字段的宽度设置为 *n* 个字符。如果输出值所需域宽小于 *n*，则额外的空间用空格填充。这种性质是暂时的，这意味着它只影响下一个插入到流中的数字值的输出宽度
setprecision(*digits*)	将输出浮点精度设置为 *digits*。精度说明的解释依赖于其环境的设置。如果你已经将格式设置为 fixed 或 scientific，*digits* 会指定小数点后数字的位数。如果没有设置以上两种模式，*digits* 表示有效数字的位数，并且不考虑这些数字出现在什么地方。这种性质是持久的，这意味着它一直保持有效，直到它被明确地改变为止
setfill(*ch*)	为所设置填充字符 *ch*。默认地，如果需要额外的字符填充到 setw 设置的字段宽度中，则空格作为填充字符输出。调用 setfill 使输出流可以改变填充字符。例如，调用 setfill('0') 意味着字段将用 0 填充。这种性质是持久的
left	指定输出字段为左对齐，这意味着任何填充字符都在输出值之后插入。这种性质是持久的

（续）

right	指定输出字段为右对齐，这意味着任何填充字符都在输出值之前插入。这种性质是持久的
fixed	指定之后的浮点数输出应该完整地呈现，并且不使用科学计数法。默认地，浮点数应该以最简洁的形式呈现。这种性质是持久的
scientific	指定之后的浮点数输出应该以科学计数法的形式呈现。这种性质是持久的
showpoint noshowpoint	这两个流操纵符控制浮点数中是否能出现小数点，这种控制同样适用于整数的情况。可以用 showpoint 强制要求出现小数点，然后通过 noshowpoint 来恢复默认设置。这种性质是持久的
showpos noshowpos	这两个流操纵符控制在一个正整数是否应有一个正号。默认地，正数前没有正号。这种性质是持久的
uppercase nouppercase	这两个流操纵符控制作为数据转换的一部分所产生的任意字母的大小写，例如科学计数法中的大写字符 E。默认地，字符以小写字母呈现。这种性质是持久的
boolalpha noboolalpha	这两个流操纵符控制布尔值的格式。它一般使用它们内在的数值表示呈现。使用 boolalpha 流操纵符导致它们以 true 或 false 的形式出现。这种性质是持久的

162

上述语句以宽度为 2 的字段输出 i 的值，以宽度为 8 的字段输出函数 raiseToPower(2,i) 的值。两个字段都是右对齐的，因为 right 流操纵符的作用是持久的。如果你使用这一行显示 2 的 0 次幂到 2 的 16 次幂，输出显示如下：



理解 setprecision(digits) 流操纵符的使用是复杂的，因为其参数的解释说明依赖于流的其他模式设置。在缺乏任何相反的说明的情况下，C++ 以十进制或科学计数法这些更简洁的形式来表示浮点数。如果你所关心的是显示该数值，C++ 允许任意选择上述其中一种表示法。然而，如果你想更精确地控制输出，你需要表明你想要 C++ 使用哪种输出格式。fixed 流操纵符指定浮点值应该一直作为一个数字字符串呈现，其中小数点出现在合适的位置上。相反地，scientific 流操纵符指定数值应该一直使用科学计数法的形式呈现，其中，字母 E 将指数和数值分开。上述每一种格式都用一种稍微不同的方式来解释 setprecision 流操纵符，这使得它更难提供一种关于 setprecision 是如何工

作的简明描述。

正如在编程中经常遇到的情况一样，要深入理解库的更多细节是如何工作的，其中一种最好的方法是编写简单的测试程序，它会使你通过屏幕输出结果看到流操纵符的作用效果。图 4-1 中的 PrecisionExample 程序展示了三个常量（以不同的浮点模式和精度呈现）：数学常量 π、以米/秒为单位的光速和描述电气联动作用的精细结构常量。程序的输出结果见图 4-2。 163

```
/*
 * File: PrecisionExample.cpp
 * --------------------------
 * This program demonstrates various options for floating-point output
 * by displaying three different constants (pi, the speed of light in
 * meters/second, and the fine-structure constant).  These constants
 * are chosen because they illustrate a range of exponent scales.
 */

#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

/* Constants */

const double PI = 3.14159265358979323846;
const double SPEED_OF_LIGHT = 2.99792458E+8;
const double FINE_STRUCTURE = 7.2973525E-3;

/* Function prototypes */

void printPrecisionTable();

/* Main program */

int main() {
   cout << uppercase << right;
   cout << "Default format:" << endl << endl;
   printPrecisionTable();
   cout << endl << "Fixed format:" << fixed << endl << endl;
   printPrecisionTable();
   cout << endl << "Scientific format:" << scientific << endl << endl;
   printPrecisionTable();
   return 0;
}

/*
 * Function: printPrecisionTable
 * -----------------------------
 * Generates a simple precision table for the current cout settings.
 */

void printPrecisionTable() {
   cout << " prec |      pi       |  speed of light  | fine structure" << endl;
   cout << "------+---------------+------------------+---------------" << endl;
   for (int prec = 0; prec <= 6; prec += 2) {
      cout << setw(4) << prec << "  |";
      cout << " " << setw(13) << setprecision(prec) << PI << " |";
      cout << " " << setw(16) << setprecision(prec) << SPEED_OF_LIGHT << " |";
      cout << " " << setw(14) << setprecision(prec) << FINE_STRUCTURE << endl;
   }
}
```

图 4-1 探索 setprecision 行为的程序 164

尽管格式化输入和输出（计算机科学家经常缩写为 I/O）机制在任何程序设计语言中都很有用，但是它们也趋向于注重细节。一般而言，了解最常见的格式化任务的实现细节是有意义的，而对于不常用的操作，只需在用到它们时再去了解。

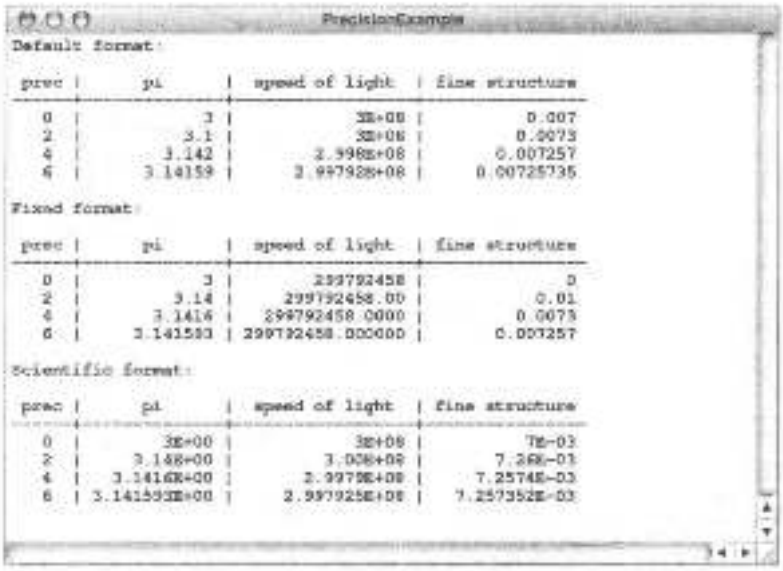


图 4-2 说明浮点输出的示例运行结果

4.2 格式化输入

C++ 的格式化输入已嵌入了流操作符 >>，你已经在各种各样的程序中用到过它。这个操作符称为提取操作符（extraction operator），因为它用于从一个输入流中提取格式化数据。到目前为止，你已经使用 >> 操作符从控制台请求输入数据，例如第 1 章中 PowersOfTwo 程序的语句行：

```
int limit;
cout << "Enter exponent limit: ";
cin >> limit;
```

165

默认地，>> 操作符在尝试读取输入数据之前忽略所有空白字符。如果有必要，可以使用 skipws 和 noskipws 流操纵符来改变这种行为，它们显示在表 4-2 的输入流操纵符的列表中。例如，当你执行下述语句时：

```
char ch;
cout << "Enter a single character: ";
cin >> noskipws >> ch;
```

用户可以输入一个空格字符或制表符来响应屏幕提示。一旦你省略了 noskipws 流操纵符，程序将会在存储 ch 的下一个输入字符之前跳过空白字符。

尽管提取操作符使得编写一个简单地从控制台读取输入数据的测试程序变得简单，但在实际中它并没有广泛采用。>> 操作符的主要问题是它几乎不提供任何支持检测用户输入是否有效的功能。众所周知，用户在向计算机中输入数据时是很草率的，他们会造成一些“笔误”，或者更糟糕的是，他们根本没有理解程序真正想要什么输入。设计良好的程序会检测用户的输入以确保它形式正确，并且在程序中是有意义的。

表 4-2　输入流操纵符

skipws noskipws	这两个流操纵符控制提取操作符 >> 在读取一个值之前是否忽略空白字符。如果指定 noskipws，提取操作符将所有的字符（包括空白字符）看作是输入字段的一部分。之后可以使用 skipws 恢复默认的行为。这个状态是持久的
ws	从输入流中读取字符，直到它不属于空白字符。因此，这个流操纵符的作用是跳过输入中的任何空白字符、制表符和换行符。不像 skipws 和 noskipws 改变的是流关于之后的输入操作行为，ws 流操纵符是立即起作用的

遗憾的是，>> 操作符根本就不能胜任检测输入是否合法这个任务，这正是在 2.9 节所介绍的 simpio.h 库存在的原因。如果你使用这个库提供的机制，从用户那里读取一个值的代码会从三行缩减至以下一行：

```
int limit = getInteger("Enter exponent limit: ");
```

正如在第 2 章所提到的，getInteger 函数同时会实现内部的必要错误检查，这使得它使用起来更安全。随后你将有机会理解 getInteger 是如何实现的，当你了解之后，它看起来将不再神秘。

166

4.3　数据文件

当你想在计算机上存储信息的时间长于一个程序的运行时间时，通常的方法是将这些数据整理成一个逻辑整体，并将其作为**文件**（file）存储在一个永久存储介质上。通常，文件是使用磁或光介质来存储的，例如安装在计算机上的硬盘或一个可移动的闪存或记忆棒中。然而，介质的特殊细节并不是决定性的，重要的是，你存储在计算机上的永久数据对象：文档、游戏、可执行程序和源代码等，它们都是以文件的形式存储的。

在绝大多数系统中，文件可以是各种各样的类型。例如，在编程领域，你与源文件、目标文件和可执行文件打交道，上述每一种文件都有其独特的表示方式。当你使用文件去存储一个程序使用的数据时，文件通常由文本构成，因此它也称为**文本文件**（text file）。可以将文本文件视为一个字符序列，该字符序列存储在一个永久介质中并以文件名加以识别。文件名和它所包含的字符与变量名和它的内容有相同的关系。

例如，以下文本文件包含刘易斯·卡罗尔在《镜中奇遇》打油诗“Jabberwocky”的第一节内容：

Jabberwocky.txt

```
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
```

该文件名为 Jabberwocky.txt，文件内容包含由诗的第一节组成的四行字符。

当你查看一个文件时，通常将它看作是一个二维结构：由单个字符构成的一系列字符行。然而，文本文件内部都用一个一维的字符序列表示。除了你能看到的输出的字符之外，文件也包含一个换行符用来标记每一行的结束。

在很多方面，文本文件和字符串是类似的。每一个文本文件都由一个带有一个特定结束符的有序字符集合组成。另一方面，字符串和文件在几个重要的方面是不同的。最重要的不同点是数据的持久性。当一个程序运行时，字符串是临时存储在计算机的内存中；而文件

是存储在一个长期的存储设备上，直到它被明确地删除。还有一个不同点是你如何对待字符
167 串中的字符与文件中的字符。因为字符串中的每个字符有一个索引，所以你可以简单地通过选择索引值以任意顺序来处理字符串中的字符。相反，一个文本文件中的字符只能被顺序处理。典型地，处理文件数据的程序在文件的开头开始，然后以它们的方式工作（或者从一个输入文件中读取已有的字符，或者向一个输出文件写入新行），直到文件的结尾。

4.3.1 使用文件流

正如你将在 4.4 节看到的，C++ 流库中的若干类形成了一个层次结构。为了帮助你从整体上来理解该层次结构，通过 <fstream> 库提供的两个流类（ifstream 和 ofstream）开始进行你的数据流学习是很有用的。一旦你熟悉了那些例子，将很容易从经验中进行概括和总结，以便从整体上理解流层次。

应用到文件流上最常见的方法显示在表 4-3 中。然而，研究表 4-3 不可能像学习几个处理文件的简单模式一样有用。任何编程语言的文件处理都趋向于符合语言习惯，在某种程度上，你需要学习一种通用的处理策略，然后再将这个策略应用到你编写的应用程序中。对于此规则，C++ 也不例外。

在 C++ 中，读或者写一个文件要求遵循以下步骤：

1. 声明一个指向某个文件的流变量。处理文件的程序通常为每一个活动文件声明一个流变量。因此，如果你正在编写一个读取输入文件的程序，然后再处理其中的数据以产生另一个输出文件，那么你需要声明以下两个变量：

```
ifstream infile;
ofstream outfile;
```

2. 打开文件。在可以使用一个流变量之前，需要在所声明的变量和一个实际的文件间建立关联。该操作称为打开（opening）文件，它是通过调用流方法 open 实现的。例如，如果你想读取包含在 Jabberwocky.txt 文件中的文本，通过执行以下方法调用可以打开这个文件：

```
infile.open("Jabberwocky.txt");
```

由于流库先于 string 类的介绍，因此 open 方法将 C 风格的字符串看作是文件名。于是，一个字符串字面值的文件名是可接受的。然而，如果文件名存储在名为 filename 的 string 变量中，你可以通过以下方法调用打开文件：

168
```
infile.open(filename.c_str());
```

表 4-3 流类中的有用方法

所有的流都支持的方法	
stream.fail()	如果流处于失效状态，则返回 true。这个条件通常发生在你尝试超出文件的结尾去读取数据的时候，但这也表示数据中出现了一个完整性错误
stream.eof()	如果流位于文件的结尾，则返回 true。基于 C++ 流库的语义，eof 方法只用在一个 fail 调用之后。此时，eof 调用允许你判断故障是不是由于到达文件的结尾引起的
stream.clear()	重置与流相关的状态位。当一个故障发生后，无论何时需要重新使用一个流，都必须调用这个函数
if(*stream*)…	判断流是否有效。就大部分情况而言，这个测试和调用 if(!*stream*.fail()) 的效果相同

（续）

所有文件流都支持的方法	
stream.open(*filename*)	尝试打开文件 *filename* 并将其附加到流中。流的方向和流的类型所决定：输入流对于输入打开，输出流对于输出打开。*filename* 参数是一个 C 风格的字符串，这意味着你将需要在任何 C++ 字符串上调用 c_str，然后调用 fail，可以检测 open 方法是否失败
stream.close()	关闭依附于流的文件
所有的输入流都支持的方法	
stream>>*variable*	将格式化数据读入到一个变量中。数据的格式是由变量类型控制的，并且无论输入流操纵符是什么，它都是有效的
stream.get(*var*)	将下一个字符读入到字符变量 *var* 中，*var* 是引用参数。返回值是流本身。如果没有更多的字符，设置 fail 标志
stream.get()	返回流的下一个字符。返回值是一个整数，它可识别以常量 EOF 表示文件结尾的字符
stream.unget()	复制流的内部指针以便当后续取的一个字符等于次被下一个 get 调用读取
getline(*stream*, *str*)	将流 *stream* 中的下一行读入到字符串变量 *str* 中。getline 函数返回流。它简化了文件结尾的测试
所有的输出流都支持的方法	
stream<<*expression*	将格式化数据写入到一个输出流。数据格式由表达式的类型所控制，并且对于任何输出流操纵符都有效
stream.put(*ch*)	将字符 *ch* 写入到输出流

169

如果请求的文件丢失，流会记录那个错误，并且可调用判定方法 fail 去检测它。从这些故障中恢复是你作为一个程序员的责任。在本章的后面，你将学到各种进行故障恢复的策略。

3. 传输数据。一旦你打开了数据文件，你之后会使用合适的流操作去实现实际的 I/O 操作。根据应用，可以选择任意的传输文件数据策略。最简单的是逐个字符地读或写文件。然而，在某些情况下，逐行处理文件会更方便。在更高的层面上，可以选择读或写格式化数据，它允许你将数值数据与字符串和其他的数据类型混合在一起。这些策略的细节将在后续章节中阐述。

4. 关闭文件。当你结束了所有的文件数据传输后，通过调用流方法 close 以告知文件系统关闭打开的文件，如下所示：

```
infile.close();
```

此操作称为**关闭**（closing）文件，它切断了流与所关联文件之间的关系。

4.3.2 单个字符的输入 / 输出

在许多应用中，处理文本文件中数据的最好方法是每次从文件中读取一个字符。C++ 类库中的输入流支持通过调用 get 方法以支持从文件中每次读取一个字符。get 方法有两种形式。最简单的策略是使用表 4-3 中 get 方法的第一种形式，它将来自流的下一个字符保存在一个引用传递的变量中，将来自 infile 流的第一个字符读入到变量 ch 中，如以下代码所示：

```
char ch;
infile.get(ch);
```

当然，对于绝大多数应用，其目的并不是每次从文件中读取单个字符，而是每次读取连续的多个字符，就像你浏览文件时一样。通过允许流在有条件的背景下使用，C++ 使得这个操作十分简单。读取一个文件的所有字符的一般模式如以下代码所示：

```
char ch;
while (infile.get(ch)) {
   Perform some operation on the character
}
```

170

get 方法将下一个字符读入到变量 ch 中并返回流。相应地，如果 get 操作成功，流解释为 true；否则，流解释为 false，这很可能是因为文件中再没有剩余字符。因此，上述代码的效果是对读取的每个字符执行一次 while 循环的主体，直到流到达文件尾部为止。

尽管这种策略极其简单，但许多 C++ 程序员仍使用返回一个字符的 get 方法版本，很大程度上是因为即使在过去的 C 语言版本中，这种策略也是可用的。这种 get 方法的形式有以下原型：

```
int get();
```

刚开始，你可能觉得函数的返回结果类型好像很古怪。它看起来更应返回一个 char 类型的量，但该原型表明 get 方法返回一个 int 类型的量。这种设计决策的原因是：返回一个字符会使程序更难以检测到输入文件的结尾。由于 char 类型只有 256 种可能的字符代码，一个数据文件可能包含这些值中的任何一个。不存在这样的值（或至少不存在这样的 char 类型值），让你可以使用它作为表示文件结束条件的信号量。定义 get 返回一个整数意味着函数可以返回合法字符代码范围之外的一个值来表示文件结束的条件。该值的符号名为 EOF。

如果你使用这种形式的 get 方法，那么逐个字符地读取一个完整文件的代码模式应该如下所示：

```
while (true) {
   int ch = infile.get();
   if (ch == EOF) break;
   Perform some operation on the character
}
```

代码的实现使用了读首到信号量的模式，这种模式在第 1 章的 AddList 程序中已经介绍。while 循环的主体部分将下一个字符读入到整型变量 ch 中，如果 ch 是文件结束信号量，则退出循环。

然而，许多 C++ 程序员使用下面稍短但明显更有意义的形式来实现这个循环：

```
int ch;
while ((ch = infile.get()) != EOF) {
   Perform some operation on the character
}
```

171

在这种形式的代码中，while 循环的检测表达式使用了嵌入式的组合操作，即首先读取字符然后再测试它是否为文件结束条件。当 C++ 计算这个测试时，首先计算子表达式：

```
ch = infile.get()
```

这个表达式读取一个字符并将它赋给变量 ch。在执行循环主体之前，程序要继续确保赋值结果不是 EOF。赋值语句周围的圆括号是必需的；没有它们，表达式会错误地将字符与 EOF 的比较结果赋给 ch。因为这种语言风格会频繁地出现在现存的 C++ 代码中，当它出现时，

你需要识别并理解它。在你自己的代码中，更容易基于 get 方法的引用调用版本来使用这种更简单的语言风格。

对于输出流，put 方法以一个 char 类型的值为其参数，然后将字符写入到输出流中。因此，一个典型的 put 方法调用如下所示：

```
outfile.put(ch);
```

作为一个 get 和 put 方法的使用实例，图 4-3 中的 ShowFileContents 程序在控制台上展示了一个文本文件的内容。假设存在 Jabberwocky.txt 文件，这个程序的示例运行结果如下所示：

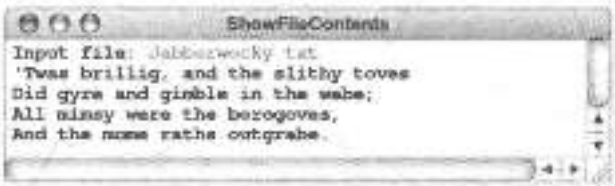


图 4-3 中的代码也包含了函数 promptUserForFile，它要求用户输入一个文件名，然后打开该文件作为输入。如果该文件不存在或由于某些原因不能打开，则 promptUserForFile 要求用户输入一个新的文件名，持续这个过程直到 open 调用成功为止。当用户输入一个不合法的文件名时，这种设计允许程序能优雅地恢复。例如，如果用户第一次忘记包含 .txt 扩展名，控制台最开始几行输出如下所示：



172

```
/*
 * File: ShowFileContents.cpp
 * --------------------------
 * This program displays the contents of a file chosen by the user.
 */

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

/* Function prototypes */

string promptUserForFile(ifstream & infile, string prompt = "");

/* Main program */

int main() {
   ifstream infile;
   promptUserForFile(infile, "Input file: ");
   char ch;
   while (infile.get(ch)) {
      cout.put(ch);
   }
   infile.close();
   return 0;
}
```

图 4-3 展示一个文件内容的程序

```
/*
 * Function: promptUserForFile
 * Usage: string filename = promptUserForFile(infile, prompt);
 * ------------------------------------------------------------
 * Asks the user for the name of an input file and opens the reference
 * parameter infile using that name, which is returned as the result of
 * the function. If the requested file does not exist, the user is
 * given additional chances to enter a valid file name. The optional
 * prompt argument is used to give the user more information about the
 * desired input file.
 */

string promptUserForFile(ifstream & infile, string prompt) {
   while (true) {
      cout << prompt;
      string filename;
      getline(cin, filename);
      infile.open(filename.c_str());
      if (!infile.fail()) return filename;
      infile.clear();
      cout << "Unable to open that file. Try again." << endl;
      if (prompt == "") prompt = "Input file: ";
   }
}
```

173

图 4-3 （续）

尽管 promptUserForFile 函数实现背后的逻辑很容易学习，但仍有几个重要的细节值得提及。例如，open 调用需要使用 c_str 将存储在 filename 中的 C++ 字符串转换为流库要求的旧风格的 C 字符串。类似地，需要在 while 循环体内的调用 clear 来确保在用户输入一个新的文件名之前，流中的故障状态标志被重置。

当你从一个输入文件中读取字符数据时，有些你会发现你处于这样一个尴尬位置：你不知道何时应停止读取字符，直到你已经读取到了超过你所需要的字符。例如，考虑一下，当 C++ 提取操作符尝试读取一个整数时，会发生什么，这个整数是用十进制数的字符串表示的。直到库实现函数读取一个不是数字的字符时，它才知道输入数据结束。然而，那个字符可能是一些随后输入的一部分，因此，不能丢失信息是至关重要的。

通过提供一个名为 unget 的方法，C++ 解决了这个关于输入流的问题。该方法具有以下形式：

```
infile.unget();
```

这个调用的作用是将大部分最近的字符“推送”回到输入流中，以便它在下一个 get 调用中返回。C++ 库的规格说明保证了它总是可以将一个字符推送回到输入文件中，但是你不应该依赖能够预先读取几个字符并且将它们全部推回。幸运的是，在绝大部分情况下，能够推送回一个字符已经足够了。

4.3.3 面向行的输入 / 输出

因为文件通常细分为单独的行，所以每次读取一整行数据常常是很有用的。实现这种操作的流函数为 getline，尽管它和来自 simple.h 头文件的 getline 函数有着类似的功能，但两者是不一样的。getline 函数（作为一个自由函数而不是类中的方法）有两个引用参数：从中读取数据行的输入流和向其写入结果的一个字符串变量。函数调用如下：

```
getline(infile, str);
```

将文件 infile 的下一行复制到变量 str 中，直到（但不包括）读入标记行结尾的换行符为

止。类似 get 方法，getline 函数返回输入流，这使得它很容易判断文件结束条件。你所需要做的是将 getline 调用本身作为一个测试表达式来使用。

getline 函数使得重写 ShowFileContents 程序成为可能，这样该程序可以每次从文件中读取一行。如果你采取这种方法，主程序中的 while 循环将如下所示：

```
string line;
while (getline(infile, line)) {
   cout << line << endl;
}
```

while 循环将 infile 文件中的每一行数据读入到 string 变量 line 中，直到流到达文件的结尾。循环体中使用 << 操作符将每行数据 line 发送给 cout，后面加一个换行符。

4.3.4　格式化输入 / 输出

除了逐个字符处理文件和逐行处理文件的方法外，也可以在文件流上使用 << 和 >> 操作符，正如你已经在控制台流上使用过的一样。例如，假设你想修改图 1-5 所示的 AddIntegerList 程序，使其能够从一个数据文件而不是控制台获取输入。最简单的方法就是打开一个用于输入的数据文件，使用 ifstream 代替 cin 去读取输入值。

在程序代码中，你仅需要做的一个改变是当输入结束时退出循环。控制台版本的程序使用了一个信号量值来表示输入的结束。对于从文件中读取输入数据的程序版本而言，循环应该持续到没有更多的数据值可以读取为止。判断该条件最简单的方法是使用 >> 操作符。因为 >> 返回这个输入流，所以它通过设置 fail 标志表示文件结束条件，C++ 将其解释为 false。

如果你在原始代码中做了这些改变，则程序如下所示：

```
int main() {
   ifstream infile;
   promptUserForFile(infile, "Input file: ");
   int total = 0;
   int value;
   while (infile >> value) {
      total += value;
   }
   infile.close();
   cout << "The sum is " << total << endl;
   return 0;
}
```

遗憾的是，尽管从技术角度来说这种实现策略是正确的，但不够完美。如果所有的数据都用正确的方式格式化，程序将返回正确的答案。然而，如果文件中有无关字符，那么在读取完所有的输入数据之前循环就会退出。更糟糕的是，程序不会给出错误发生的提示。

问题的核心是表达式中的提取操作符：

```
infile >> value;
```

这将在下述两种情况中的任意一种发生时设置故障提示器：

1. 到达文件的结尾，该处已没有更多的数据可供读取。
2. 试图从文件中读取不能转化为整数的数据。

通过检查当循环退出时文件是否已到达结尾，可以区分上述两种情况。例如，可以在 while 循环后添加下面一行代码，以使用户得知数据错误是否发生：

```
   if (!infile.eof()) {
      error("Data error in file");
   }
}
```

关于错误的根源，该错误提示并没有给用户提供太多的指导，但至少它比什么都没有要好。

另一个问题是就提取操作符所允许的数据格式而言，它过于自由。除非你已经指定，否则，>>操作符将所有空白字符序列作为数据分隔符接受。这样，输入文件不需要每一行只包含一个值，取而代之的是，可以用许多方法进行格式化。例如，你想使用你的应用添加开始的五个整数，你不需要在每一行只输入一个数据，如下面的数据文件所示：

```
1
2
3
4
5
```

将所有的值放在单独的一行也能够很好地工作，并且在实际中，它可能更方便。如下所示：

```
1  2  3  4  5
```

176

抓有如此大的灵活性所带来的问题是：更难发现某些种类的格式化错误。例如，如果你偶然地在一个整数中添加一个空格，将会发生什么？就目前情况看，SumIntegerFile 应用会简单地将空格前后的两个数字读作是两个分开的值。在这种情况下，坚持严格的格式化规则对于保持数据完整性显得更好。

在 SumIntegerFile 应用中，正如第一个示例文件一样，坚持每一行只出现一个数据值是有意义的。遗憾的是，如果你拥有的唯一工具是文件流和提取操作符，强制约束是很困难的。开始的一种方法是以每次一行来读取数据文件，然后在添加到整体之前将每一行转换成一个整数。如果你要采用这种方法，主程序将如下所示：

```
int main() {
   ifstream infile;
   promptUserForFile(infile, "Input file: ");
   int total = 0;
   string line;
   while (getline(infile, line)) {
      total += stringToInteger(line);
   }
   infile.close();
   cout << "The sum is " << total << endl;
   return 0;
}
```

唯一省略的内容是 stringToInteger 函数的实现。

尽管 C 库包含一个名为 atoi 的函数，它能将字符串转换成一个整数，但是由于该函数是在 <cstring> 类库之前出现的，因此，它要求使用 C 字符串，C 字符串使用起来更不方便。如果你能找到一种方法实现该转换，同时在 C++ 中又保持数据完整性，这将是非常幸运的事。你知道 C++ 库必须包含必要的代码，因为当 >> 操作符从文件中读取一个整数时，它必须实现转换。如果有一种方法能使用相同的代码从字符串中读取整数，那么采用 stringToInteger 的实现将很快得出结果。正如你将在后续章节所学到的，C++ 流库正好提供了那种功能，它被证明在多种多样的应用中是很有用的。

4.3.5 字符串流

鉴于文件和字符串都是字符序列，很自然地想到程序语言可能允许你同等地对待它们。C++在<sstream>库中提供了这种能力，<sstream>库提供了几种类允许你将一个流和一个字符串值关联起来，所使用的方法和<fstream>库允许你将一个流和一个文件关联起来的方法是很相像的。istringstream类与ifstream相似，它使你可以使用流操作符从字符串中读取数据。就输出而言，ostringstream类除了期望是一个字符串而不是一个文件之外，它与ofstream类的功能非常像。

istringstream类的存在使得你可以实现stringToInteger方法，它在最后一节中会有描述，如下所示：

```
int stringToInteger(string str) {
   istringstream stream(str);
   int value;
   stream >> value >> ws;
   if (stream.fail() || !stream.eof()) {
      error("stringToInteger: Illegal integer format");
   }
   return value;
}
```

函数的第一行介绍了变量声明的一种重要特征，这是你以前没见过的。如果你正在声明一个对象，那么C++允许你在变量名后添加参数来控制对象的初始化。在这个实现中，以下行：

```
istringstream stream(str);
```

声明了一个名为stream的变量，并且将其初始化为一个istringstream对象，该对象已经建立，用于从字符串变量str中读取数据。之后两行如下所示：

```
int value;
stream >> value >> ws;
```

这两行代码从流中读取一个整数值并将其存储在变量value中。在这个实现中，空白字符允许出现在值前面或后面。第一个>>操作符会自动地跳过任何出现在值前面的空白字符。位于行结尾的ws流操纵符会读取任何出现在值后面的空白字符，因此，它确保了：如果输入能正确地初始化，流的位置也是正确的。以下几行语句会检查确保输入合法：

```
if (stream.fail() || !stream.eof()) {
   error("stringToInteger: Illegal integer format");
}
```

如果字符串不能作为一个整数解析，stream.fail()将会返回true，从而触发错误消息。然而，如果字符串是以数字开始但是却包含另外的字符，stream.eof()将会是false，它也会触发错误消息。

如果你需要向另一个方向转化，可以使用ostringstream类。例如，下面的函数将一个整数转化为十进制数字字符串：

```
string integerToString(int n) {
   ostringstream stream;
   stream << n;
   return stream.str();
}
```

第二行的 << 操作符将值转化为它的十进制表示形式，正如它在文件中的作用一样。然而，在这种情况下，输出指向的是一个字符串，该字符串是作为 ostringstream 对象的一部分内部存储的。return 语句中的 str 函数复制了内部字符串的值，使得它可以返回给调用者。注意到一个有趣的事实：这个方向的转化实质上更容易，因为它不再需要考虑格式化错误。

4.3.6 一个用于控制台输入的更健壮的策略

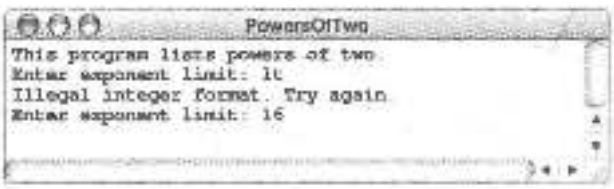
对于检测用户输入的格式是否正确的问题，字符串流也提供了一个解决方法。正如 4.3.4 节讨论的，>> 操作符不检查用户的输入错误。例如，如果你要求用户输入一个整数值，并且要使用来自 PowersOfTwo 程序的如下语句，考虑一下，将会发生什么：

```
int limit;
cout << "Enter exponent limit: ";
cin >> limit;
```

如果用户输入一个有效的整数，一切都顺利。但是如果用户在尝试输入值 16 时，手却滑到了键盘的下一行，并且录入了字符 c 而不是数字 6，将会发生什么？在一个理想的世界里，程序将被输入为 1c，并且抱怨输入的有效性。遗憾的是，如果你在 C++ 中使用提取类操作符，该错误将不会被发现。当要求程序将一个值读入整型变量 limit 时，>> 操作符会读取字符直到它在整数字段中发现一个不合法的字符。因此，输入在 c 处停止，但是值 1 仍是一个合法的整数，所以程序继续正确运行，limit 等于 1。

179

确保用户输入是有效的，最有效的方法是将一整行作为一个字符串读取，然后将该字符串转化成一个整数。这个策略收录在图 4-4 所示的函数 getInteger 中，正如 >> 操作符所做的那样，这个函数从用户处读取一个整数，但它同时也确保整数是有效的。函数 getInteger 的用法和前面章节中的 stringToInteger 函数相似。主要的不同点仅是 getInteger 给予用户一次重新输入正确值的机会，而不是以一条错误消息来结束程序，如下面运行的例子所示：



正如这个例子所解释的，getInteger 在第一行发现了录入错误，然后要求用户重新输入一个新值。当用户正确地输入值 16 并按回车键后，getInteger 将这个值返回给调用者。

```
/*
 * Function: getInteger
 * Usage: int n = getInteger(prompt);
 * ----------------------------------
 * Requests an integer value from the user.  The function begins by
 * printing the prompt string on the console and then waits for the
 * user to enter a line of input data.  If that line contains a
 * single integer, the function returns the corresponding integer
 * value.  If the input is not a legal integer or if extraneous
 * characters (other than whitespace) appear on the input line,
 * the implementation gives the user a chance to reenter the value.
 */
```

图 4-4 从控制台中读取一个整数的函数

```
int getInteger(string prompt) {
   int value;
   string line;
   while (true) {
      cout << prompt;
      getline(cin, line);
      istringstream stream(line);
      stream >> value >> ws;
      if (!stream.fail() && stream.eof()) break;
      cout << "Illegal integer format. Try again." << endl;
   }
   return value;
}
```

图 4-4 （续）

[180]

正如你在图 4-4 代码中所看到的，getInteger 函数有一个提示字符串，在函数读取输入行数据时，它显示给用户。这种设计决策直接遵循一个事实，即如果一个错误发生了，getInteger 函数需要这样的信息来重复提示字符串。

对于读取整数数据，getInteger 函数是一个比 >> 操作符更值得信赖的工具，因此在要求检查错误的应用中使用 getInteger 代替 >> 操作符是有意义的。正如你已经从第 2 章学到的，getInteger 是作为 simple 库的一部分包含在 Stanford 库中的。

4.4 类层次

当 C++ 的设计者承担着使输入/输出库现代化的任务时，他们选择采用一种面向对象的方法。这种决策的含义是流库中的数据类型是作为类实现的。相比于旧的表示数据的方法，类有许多的优势。其中，最重要的是，类提供了封装（encapsulation）的框架，将数据表示和与其相关的操作组合成一个一致的整体的过程称为封装，这个整体应尽可能少地揭示底层结构的细节。本章中你所看到的类已很好地展示了封装的概念，当你使用这些类时，你不知道它们是如何实现的。作为一个用户，你所需要知道的是什么方法是可用的，以及怎样调用它们。

除了封装之外，面向对象程序提供了另外的重要优势。特别地，面向对象语言中的类形成了一个层次，其中，每一个类自动地获取上面一个层次的类的特征。这种属性称为继承（inheritance）。相对于其他许多面向对象语言，尽管 C++ 趋向于较少地使用继承，然而它仍是一个使面向对象模式区别于早期程序模型的特征。

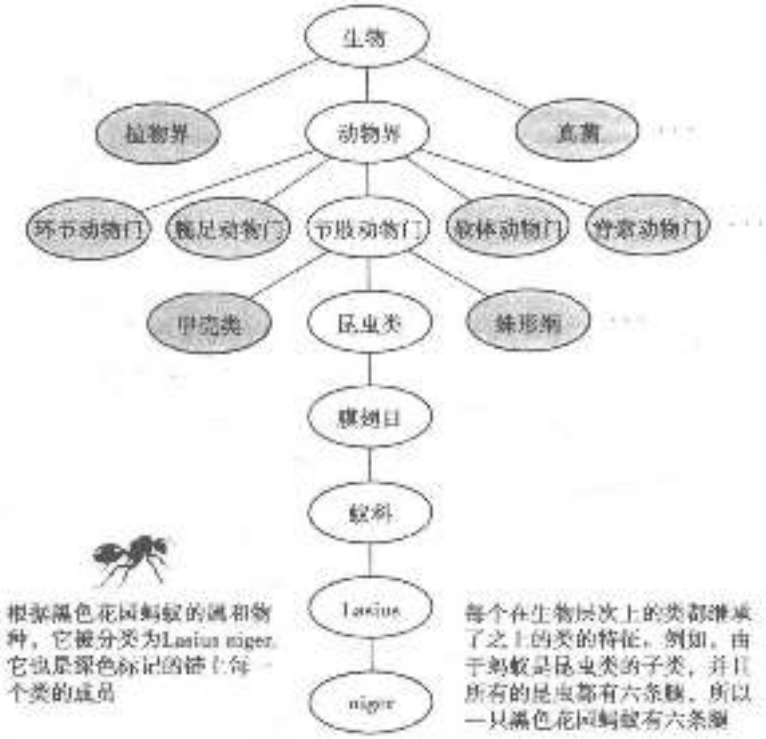
4.4.1 生物层次

一种面向对象语言的类层次结构在很多方面和生物分类系统是类似的。它是由 18 世纪瑞典植物学家卡尔·冯·林奈提出的，作为一种表示生物世界结构的方法。在林奈的概念中，生物首先细分为界（kingdom）。原始的系统只包含植物界和动物界，但是有一些形式的生命（例如真菌和细菌）不属于任何一个种类，并且现在有了它们自己的界。然后每个界进一步分解成包括门、纲、目、科、属和种的分层种类。每个属于底层的生物物种同时也属于每个较高层次的某些种类。

[181]

生物分类系统如图 4-5 所示，它展示了常见的黑色花园蚂蚁的分类，该蚂蚁有一个对应于它的属和种的学名 Lasius niger。然而这种蚂蚁也是蚁科的一部分，蚁科实际上是鉴别蚂蚁的一种分类。如果你在层次结构中从那儿向上行动，你会发现 Lasius niger 也属于

膜翅目（包括蜜蜂和黄蜂），昆虫类（由昆虫组成）和节肢动物门（例如，包括贝类和蜘蛛）。



182

图 4-5 生物世界的类层次

一种使这种生物分类系统很有用的属性是所有的生物都属于该层次中每一层的一个类别。因此，每一个单独的生命形式同时属于几个种类，并且继承了每一个类的属性。例如，物种 Lasius niger 同时是一只蚂蚁、一只昆虫、一个节肢动物和一个动物。每只蚂蚁拥有它自那些种类继承的属性。一种用于定义昆虫类的特征是昆虫有六条腿。因此，所有的蚂蚁必然有六条腿，因为蚂蚁是那个类的成员。

生物的比喻帮助解释类和对象之间的区别。尽管每一只常见的黑色花园蚂蚁都属于相同的生物类别，但有着许多常见黑色花园蚂蚁的个体。这样，每一只黑蚁



都是 Lasius niger 的一个实例。在面向对象程序语言中，Lasius niger 是一个类，并且每一只蚂蚁都是一个对象。

4.4.2 流类层次

流库中形成层次的类在很多方面和前面章节中介绍的生物层次是类似的。到目前为止，你已经见过两种类型的输入流（ifstream 和 istringstream）和两种类型的输出流（ofstream 和 ostringstream）。在每一对流中，它们都共享一组常见的操作。在 C++ 中，形成层次的这些类如图 4-6 所示。层次的顶端是类 ios，它代表一个最通用的流类型，

可以被用于任何一种输入/输出。

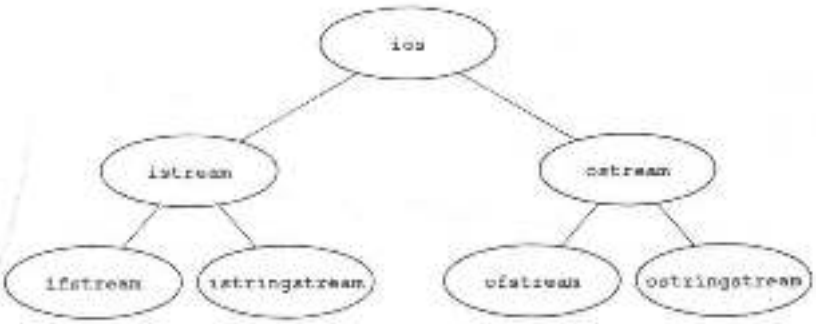


图 4-6　流层次中的一些类

然后该层次类被细分为两种类别（istream 和 ostream），分别概括了输入流和输出流的概念。自然而然地，C++ 文件和字符串流类就落在了层次中恰当的位置，如上所示。

图 4-6 提供了一个有用的框架来介绍与面向对象程序相关的一些专用术语。如图 4-5 所示，这个图同样使用了一些几何结构绘制而成，在该图中，每一个类都是出现在它上面层次的类的**子类**（subclass）。因此，istream 和 ostream 都是 ios 的子类。相反地，ios 被称作是 istream 和 ostream 共同的**父类**（superclass）。类似的关系存在于这个图的不同层次中。例如，ifstream 是 istream 的子类，ostream 是 ofstream 的父类。

子类和父类的关系在很多方面可以用英语词组“is a”很好地表达出来。每一个 ifstream 对象也是一个（is a）istream 对象，沿着类层次链向上，每一个 ifstream 对象也是一个（is a）ios 对象。正如在生物层次中的一样，这个关系表示任何类的特征都被它的子类所继承。在 C++ 中，这些对应于方法和其他定义的特征都和类有联系。因此，如果 istream 类提供了一个特殊的方法，对于任何 ifstream 和 istringstream 对象，该方法都是自动可用的。更全面地说，任何由 ios 类提供的方法对于图 4-6 层次中任意一个它的子类对象都是可用的。

尽管只展示类之间关系的简单图是很有价值的，但是扩展这些关系使其包含每一层类提供的方法也是很有用的，如图 4-7 所示。这个扩展图采用了部分的标准的方法来解释类的层次，这个方法称之为**统一建模语言**（Universal Modeling Language），或者简称为 UML。在 UML 中，每个类以一个矩形框表示，它的上部分包含了类的名字。类提供的方法出现在矩形框的下部分。UML 图使用空心的箭头以从子类指向它们的父类来表示继承关系。

如图 4-7 所示，UML 图的分类使得很容易确定对于图中的每一个类，什么方法是可用的。因为每个类继承了它的父类链中每个类的方法，一个特定类的对象都可以调用其父类所定义的方法。例如，该图表示 istream 对象可以调用下面的方法：

- 来自 ifstream 类本身的 open 和 close 方法。
- 来自 istream 类的 get 和 unget 方法以及 >> 操作符。
- 来自 ios 类的 clear，fail 和 eof 方法。

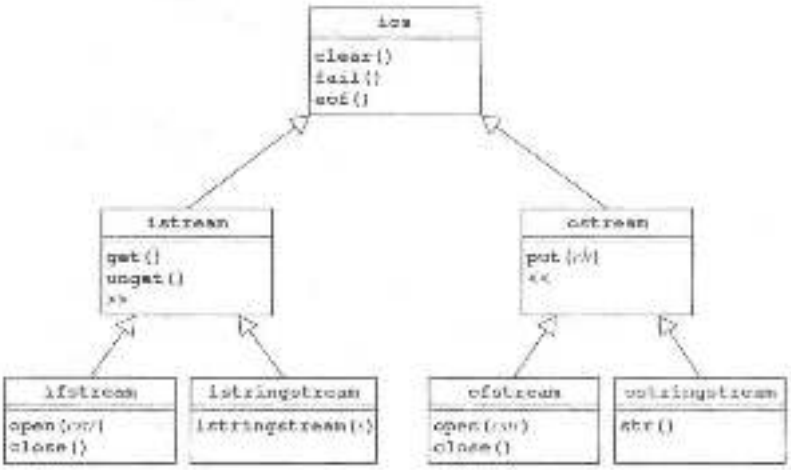


图 4-7 流层次简化的 UML 图

4.4.3 在流层次中选择正确的层次

你所需要作出的最重要的决策之一是：当使用对象层次时，你要选择适合工作的正确层次。作为一个一般的规则，最好是编写你自己的代码，这样它能使用类层次中最一般的类来支持你所需要的操作。采用这种规则能确保你的代码能尽可能灵活地支持最广泛的类型。

作为一个例子，你会经常发现定义一个 copyStream 方法是很有用的，它的作用是将来自一个输入流中的所有字符复制到另一个输入流中。如果当你使用文件流时，你已经想到这个方法，你可能会尝试像下面这样来实现这个方法：

```
void copyStream(ifstream & infile, ofstream & outfile) {
   while (true) {
      int ch = infile.get();
      if (ch == EOF) break;
      outfile.put(ch);
   }
}
```



尽管这种实现从技术上来说是正确的，但它对于错误定位仍会有很多问题。问题是如果你已
185 经选择了更多类型的参数，即使这段代码已经完美地适用于任何类型的输入输出流，这个方法仍只适用于文件流。copyStream 方法一种更灵活地实现如下所示：

```
void copyStream(istream & is, ostream & os) {
   char ch;
   while (is.get(ch)) {
      os.put(ch);
   }
}
```

新代码的优势在于你可以使用这个版本的 copyStream 方法处理所有的流类型。例如，已经给出了 copyStream 方法的这种实现，你可以在 ShowFileContents 中用单独的一行语句来替换 while 循环，如下所示：

```
copyStream(infile, cout);
```

该行语句的作用是将文件中的内容复制到 cout 中。之前的版本（其中第二个参数被声明为一个 ofstream 类而不是更一般的类 ostream）失败了，因为 cout 不是一个文件流。

4.5 simpio.h 和 filelib.h 库

第 3 章介绍了几种新的函数，将它们打包成类似 strlib.h 库的形式是很有用的。在这一章中，你已经见过几种有用的处理流的工具，在 Stanford 类库中，流已封装为两种接口：第 2 章介绍的 simpio.h 接口和用于文件密切相关的方法的 filelib.h 接口。

尽管可以在这本书的排序表中将这些接口的内容都列出来，但大部分现代的接口在书中都没有描述。随着网络的扩展，程序员使用在线参考材料多于打印的材料。例如，图 4-8 展示了关于 simpio.h 库的基于网络版的在线文档。

虽然表列描述非常适合打印在纸张上，但基于网络版的在线文档更适合在线浏览，但你还有另一种选择来学习一个接口中什么资源是可用的：你可以阅读 .h 文件。如果一个接口能有效地被设计并记录，阅读 .h 文件可以提供你所需的所有信息。在任何情况下，如果你想精通 C++，阅读 .h 文件是你需要培养的一项编程技能。

为了使你在阅读接口方面进行一些实践，你应该阅读由 Stanford 类库提供的 filelib.h 接口。这个接口包括了本章给出的 promptUserForFile 函数，以及许多其他的函数，当你处理文件时，它们迟早会有用。 186

simpio.h

simpio.h

这个接口提供了一组简化C++输入/输出操作的函数，并且提供一些关于控制台输入的错误检测。

函数

getInteger(prompt)	从cin中读取一个完整的行，并且尝试将它当作一个整数输入
getReal(prompt)	从cin中读取一个完整的行，并且尝试将它当作是一个浮点数输入
getLine(prompt)	从cin中读取一个完整的行，并且将该行文本作为一个字符串返回

函数细节

```
int getInteger(string prompt = "");
```

从cin中读取一个完整的行，并且将它当作是一个整数输入。如果输入成功，返回整数值。如果参数不是一个合法的整数或者字符串中出现无关字符（除了空白），给予用户一次重新输入值的机会。如果提交，可选择的prompt字符串将会在读取值之前打印。

用法：

```
int n = getInteger(prompt);
```

```
double getReal(string prompt = "");
```

从cin中读取一个完整的行，并且将它当作是一个浮点数输入。如果输入成功，返回浮点数值。如果参数不是一个合法的数或者字符串中出现无关字符（除了空白），给予用户一次重新输入值的机会。如果提交，可选择的prompt字符串将会在读取值之前打印。

用法：

```
double x = getReal(prompt);
```

```
string getLine(string prompt = "");
```

从cin中读取一行文本，并且将该行文本作为一个字符串。结束输入的新一行的字符不会作为返回值的一部分进行存储。如果提交，可选择的prompt字符串将会在读取值之前打印。

用法：

```
string line = getLine(prompt);
```

图 4-8　关于 simpio.h 接口的在线文档 187

本章小结

在本章，你已经学会了如何使用流层次中的库去支持涉及控制台、字符串和数据文件的输入/输出操作。本章要点包括：

- <iostream>库提供了三种标准流：cin、cout 和 cerr。
- <iomanip>库使得控制输出格式成为可能。这个库提供了一组流操纵符，其中最重要的流操纵符出现在表 4-1 中。<iomanip>库也包含了输入流操纵符，但它们在实际中不如输出流操纵符重要。
- C++ 中的 <fstream> 库提供了读写数据文件的功能。应用在文件流中的最重要的方法列举在表 4-3 中。
- 当读取一个文件时，C++ 流库允许你任意选择几种不同策略类的任意一种。你可以使用 get 方法逐个字符读取文件，也可以使用 getline 方法逐行读取文件，或者使用 >> 提取操作符格式化数据。
- <sstream>库使得我们使用 >> 和 << 操作符来读写字符串数据成为可能。
- 流库中的类形成了一个类层次，其中子类继承了父类的行为。当你设计函数处理流时，重要的是选择类层次中最一般的类，其中必要的操作都有定义。
- 在前 3 章的很多地方，文中定义的新函数被证明在许多情况下是很有用的。这些函数通过组成 Stanford C++ 库的几个接口被导出。这确保了你不需要复制代码就可以使用它们。

复习题

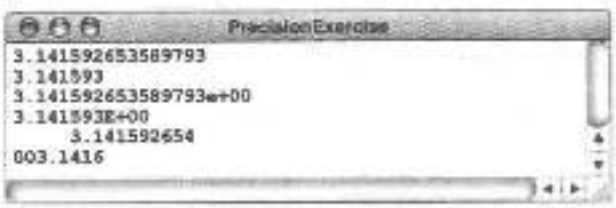
1. <iostream>库定义的三个标准文件流是什么？
2. << 和 >> 操作符的正式名称是什么？
3. << 和 >> 操作符返回什么值？这个值为什么重要？
4. 什么是流操纵符？
5. 短暂性和持久性的区别是什么？
6. 用你自己的话描述 fixed 和 scientific 流操纵符怎样改变浮点输出的格式。如果你不指定这些选择，将会发生什么？

[188]

7. 假设常量 PI 定义如下：

```
const double PI = 3.14159265358979323846;
```

你将会使用什么输出流操纵符来产生下面例子中的每一行运行结果：

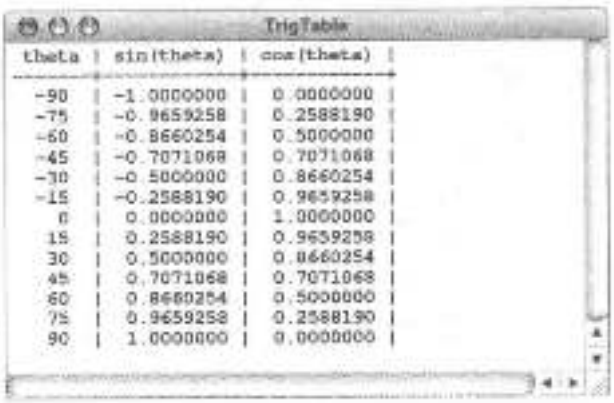


8. 类 ifstream 和 ofstream 的功能是什么？
9. open 参数必须是一个 C 风格的字符串。这个要求是如何影响你编写的打开一个文件的代码？
10. 你是如何确定流中的一个 open 操作是否成功？
11. 当你使用 get 方法去逐个字符读取文件时，你怎样发现一个文件的结尾？

12. 为什么 get 的返回类型是 int 而不是 char？
13. unget 方法的目的是什么？
14. 当你使用 getline 方法去逐行读取文件时，你怎样发现一个文件的结尾？
15. <astream> 库支持什么类？这些类和 <fstream> 提供的类有什么区别？
16. 下面术语的含义是什么：子类、父类和继承？
17. 判断题：图 4-7 中 strnam 类层次展示了 istrnam 是 istringstream 的一个子类。
18. 为什么 copyStream 函数与 istream 和 ostream 类型的参数代替 ifstream 和 ofstream 类型的参数？
19. 相比使用 >> 提取操作符，使用来自 simpio.h 的 getInteger 和 getReal 函数有什么优势？
20. 如果文中没有用表列形式来描述一个库提供的函数，你有哪些选择去学习如何使用这个库？ 189

习题

1. <iomanip> 库使得程序员更好地控制输出格式，例如，使得创建格式化表变得容易。编写一个程序，展示一个三角函数正弦和余弦值的表，如下所示：



数值列应该右对齐，包含三角函数（度数以 15 度间隔列举）的列应该在小数点后有七位数字。

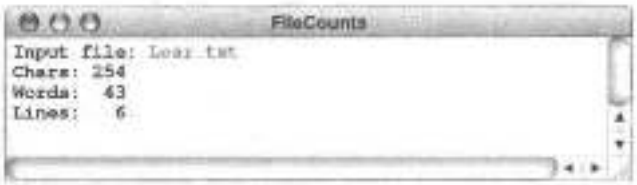
2. 在第 2 章的习题 4 中，你编写过一个函数 windchill，该函数根据给出的温度和风速计算风寒指数。编写一个程序，要求使用这个函数用列表的形式显示这些值，正如图 2-17 中来自国家气象局的表所示。
3. 编写一个程序，要求输出用户选择的文件中最长的行。如果有几行是相同的长度，你的程序应该输出第一个满足的行。
4. 编写一个程序，要求读取一个文件并输出文件行数、单词总数和字符总数。为此，一个单词是由一个连续的字符序列组成的，除了空白字符。作为一个例子，假设文件 lear.txt 包含下面这些来自莎士比亚李尔王的段落：

lear.txt

```
Poor naked wretches, wheresoe'er you are,
That bide the pelting of this pitiless storm,
How shall your houseless heads and unfed sides,
Your loop'd and window'd raggedness, defend you
From seasons such as these?  O, I have ta'en
Too little care of this!
```

190

你的程序应该能产生以下的运行结果：



输出的数值必须具有适当的数据位并且是右对齐的。例如，如果你在一个包含整个乔治·艾略特的米德尔马契（Middlemarch）文本的文件，你的程序输出应该如下所示：



5. filelib.h 接口提供了几个函数使得处理文件名变得容易。尤其是函数 getRoot 和 getExtension 将一个文件分成根部（文件名中点的前半部分）和扩展名（表示文件类型）。例如，给出文件名 Middlemarch.txt，根部是 Middlemarch，扩展名是 .txt（注意到 filelib.h 定义扩展名包含点）。编写必要的代码实现这些函数。为了发现怎样处理特殊情况，例如文件名不包含一个点，你可以阅读 filelib.h 接口或者查阅在线文档。
6. filelib.h 中另一个有用的方法是：

```
string defaultExtension(string filename, string ext);
```

如果 filename 没有扩展名，它就将 ext 添加到 filename 的末尾。例如：

```
defaultExtension("Shakespeare", ".txt")
```

将会返回“Shakespeare.txt”。如果 filename 已经有了扩展名，将返回不做任何改变的文件名，以下语句：

```
defaultExtension("library.h", ".cpp")
```

将会忽略这个指定的扩展名，并返回不作改变的“library.h”。然而，如果 ext 在点之前包含了一个星号，defaultExtension 将去除 filename 中现存的扩展名，并添加新的扩展名（减去星号）。因此，以下语句：

191

```
defaultExtension("library.h", "*.cpp")
```

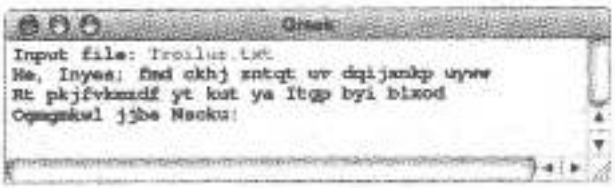
将会返回“library.cpp”。为 defaultExtension 函数编写代码，使得它像本题描述的那样运行。
7. 有时，出版商发现不被实际单词分心地评价布局和格式设计是很有用的。为此，他们有时用这样的方法来排版示例页面：将所有原始的字母用随机的字母来替换。结果文本有原始的空格和标点结构，但单词采用了这种方法的设计后不再表达任何的含义。这种替换文本的出版项目的方法是 greek，大概是在古老的谚语“It's all Greek to me”后，它才被应用在莎士比亚的《凯撒大帝》中的一行文字中。

编写一个程序，从一个输入文件中读取字符，进行适当的随机替换后，将它们显示在控制台上。你的程序应该将输入中的每个大写字母用随机的大写字母替换，每个小写字母用随机的小写字母替换。非字母表字符应该保持不变。例如，假设输入文件 Troilus.txt 包含下面的来自莎士比亚的《特洛伊罗斯克瑞西达》(*Troilus and Cressida*) 的文本：

```
Troilus.txt
Ay, Greek; and that shall be divulged well
In characters as red as Mars his heart
Inflamed with Venus:
```

你的程序产生的输出应该如下所示：

```
Greek
Input file: Troilus.txt
He, Inyes; fmd ckhj zntqt uv dqijzmkp uyww
Rt pkjfvkmzdf yt kut ya Itgp byi blxod
Oqmgmkwl jjbe Nsoku:
```



8. 尽管注释对人类读者来说很重要，但编译器会简单地忽略它们。如果你正在编写一个编译器，因此需要能够识别并消除出现在源文件中的注释。

编写一个函数：

```
void removeComments(istream & is, ostream & os);
```

192

除了出现在 C++ 注释中的字符，该函数将来自输入流 is 中的字符复制到输出流 os 中。你实现的程序应该能识别两种注释公约：

- 任何以 /* 开始并以 */ 结束的文本，可能其中有很多行。
- 任何以 // 开始的文本，扩展到行的结尾。

真正的 C++ 编译器需要检测确保这些字符不包含在引用字符串内，但是忽略该细节你应该感到很舒服，问题是它十分狡猾。

9. 在詹姆斯·瑟伯的童话故事“奇怪的字母 O（The Wonderful O）”中，奥罗岛被私人侵略了，侵略者准备将字母 O 从字母表中删除。和现在的知识相比，这样的审查制度将更容易。编写一个程序，要求用户提供一个输入文件、一个输出文件和一个将要被消除的字母字符串。然后程序要将输入文件的内容复制到输出文件中，删除任何出现在被检查过的字母中的字符，无论它们以大写字母形式出现还是以小写字母形式出现。

作为一个例子，假设你有一个包含瑟伯的小说开头几行文字的文件，如下所示：

```
TheWonderful0.txt
Somewhere a ponderous tower clock slowly
dropped a dozen strokes into the gloom.
Storm clouds rode low along the horizon,
and no moon shone.  Only a melancholy
chorus of frogs broke the soundlessness.
```

如果你使用这样的输入来运行你的程序：



```
BanishLetters
Input file: TheWonderful0.txt
Output file: TheWnderful.txt
Letters to banish: o
```

它应该显示出下面这样的文件：

193

TheWonderful.txt

```
Smewhere a pnderus twer clck slwly
drpped a dzen strkes int the glm.
Strn cluds rde lw alng the brizn,
and n mn shne.  nly a melanchly
chrus f frgs brke the sundleasness.
```

如果你尝试更贪心一点，想要消除所有的元音字母，你可以通过输入 aeiou 来对应这个命令，输出文件的内容如下所示：

```
Smwhr  pndrs twr clck slwly
drppd  dzn strks nt th glm.
Strn clds rd lw lng th hrzn,
nd n mn shn.  nly  mlnchly
chrs f frgs brk th sndlssnss.
```

10. 某些文件使用制表符使数据分成不同的列。然而，这样做会产生应用问题，即应用不能直接使用制表符。对于这些应用，一个有用的方式是编写一个程序，将输入文件中的制表符用若干空格符代替，空格要求一直持续到下一个制表符停止的位置。在程序中，通常制表符停止的位置设置在每八列处。例如，假设输入文件包含下面形式的一行字符：

```
abc ——→| pqrst —→| xyz
```

其中 → 符号表示空格被制表符所代替，区别依赖于它在这一行中的位置。如果制表符停止的位置设置在每八列处，第一个制表符必须被五个空格代替，第二个制表符被三个空格代替。

编写一个程序，将输入文件中的内容复制到输出文件，并将其中所有的制表符用适当数目的空格代替。

11. 使用函数 stringToInteger 和 integerToString 作为一个模型，编写实现函数 stringToReal 和 realToString 所需的必要代码。

194 12. 通过实现函数 getReal 和 getLine 来完成 simpio.h 接口的实现。

集 合 类

这样，我就有了一个很有价值的收藏了。

——马克·吐温，《国外漫游》(*A Tramp Abroad*)，1880 [195]

正如你的编程经验所告诉你的，数据结构可以进行组合以形成层次结构。像 int、char、double 这样的基本类型代表这些层次结构的基本构造块。为了表示更复杂的信息，你可以把不同的基本类型组合在一起以形成一个更大的结构。在不断地扩充过程中，这些较大的结构可以再次被组装成一个更大的结构。这些组合的结构被统称为**数据结构**（data structure）。

随着不断地深入学习编程，你会发现特殊的数据结构是很有用的，并且值得深入学习。此外，了解如何去高效地使用这些数据结构远比了解它们的内部表示要重要得多。例如，即使一个字符串在机器内被表示为一个字符数组，它还是有一些抽象行为超出了它的内部表示。一种类型是根据它的行为而不是其内在表示定义的，我们称这种类型为**一种抽象数据类型**（abstract data type），通常简写为 ADT。抽象数据类型是面向对象编程风格的核心，它鼓励编程人员用全局的方式来考虑数据结构。

本章介绍五个类——Vector，Stack，Queue，Map 和 Set，其中每一个类都代表了一种重要的抽象数据类型。此外，它们都包含了一些简单类型值的集合。因此，这些类被称为**集合类**（collection classe）。暂时你不需要了解这些类是如何实现的，因为你应重点关注的是作为一个用户去学习如何使用这些类。在后续的章节中，你将有机会去探索各种类的实现策略，并且学习一些必要的算法和数据结构以获得更高效的实现。

把类的行为与其底层的实现相分离是面向对象编程的一项基本技术。作为一种设计策略，维持这种分离可提供以下优势：

- **简单性**（simplicity）。对用户隐藏类的内部表示意味着用户只有较少的细节需要去理解。
- **灵活性**（flexibility）。因为一个类是根据其对外公开的行为来定义的，实现一个类的程序员可以自由地改变该类的内部私有表示。和任何抽象一样，只要保持类的对外接口不变就可以适当地改变其内部实现。
- **安全性**（security）。类的接口扮演了防火墙的角色，它确保了类的实现和用户的彼此分离。如果一个用户程序有权访问类的内在表示，则它可以以出乎预料的方式来改变类中的内部数据结构中的值。设定类中的数据为私有的，防止了用户对其做出改变。 [196]

为了使用本章中所介绍的任意集合类，你必须包含适当的接口，就像在前面的章节中你使用库一样。每个集合类的接口仅仅由类的名字的首字母小写以及在最后跟着一个扩展名 .h 拼写而成。例如，为了在程序中使用 Vector 类，你必须在你的源文件开始处添加下面这行代码：

```
#include "vector.h"
```

本书中使用的集合类是受一个更高级的类集合所启发，并由此衍生而来。这个类集合被称之为**标准模板库**（Standard Template Library），或简称为 STL。尽管 STL 足够强大，但无论你是作为类的用户还是作为实现者，理解它都是很困难的。使用 STL 简单版本的一个好处就是：在你学习完本书时，你可以完全理解它的实现。了解类是如何实现的会让你对之后标准模板库的用途有更深入的理解。

5.1 Vector 类

最有价值的集合类之一就是 Vector 类，它提供了一种类似于你在早期编程中曾遇到过的具有数组功能的机制。早期的编程大量使用数组。和大多数编程语言一样，C++ 也支持数组，你将在第 11 章学习数组是如何工作的。然而，C++ 中的数组有若干缺点，主要包括：

- 数组被分配具有固定大小的内存，以致于程序员不能在以后对其大小进行改变。
- 即使数组有固定的大小，C++ 也不允许程序员获得这个大小。因此，典型的含有数组的程序需要一个附加的变量来记录数组元素的个数。
- 传统的数组不支持插入和删除数组元素的操作。
- C++ 对数组越界不做检查。例如：如果你创建了一个含有 25 个元素的数组，之后你试图挑选出索引为 50 的数组元素值，C++ 仅查看在索引为 50 内存地址中是否有数据存在。

Vector 类通过以抽象数据类型的方式重新实现数组解决了上述问题。你可以在任何应用中使用 Vector 类代替传统的数组，通常在源代码中只需进行很少的修改和较小的删减就会产生出人意料的高效结果。实际上，一旦你有了 Vector 类，在很多场景中就不会再使用数组，除非你实际上必须实现和 Vector 一样的类，这个类使用数组作为它隐藏的数
197 据结构。然而，作为一个 Vector 类的用户，你可能不会对其隐藏的数据结构感兴趣，而把相关的数组结构问题留给程序员，让他们去实现这个抽象数据类型。

作为一个 Vector 类的用户，你会面临一组不同的问题并且需要回答以下问题：

1. 如何指定包含在一个 Vector 中对象的类型？
2. 如何创建一个对象，它是一个 Vector 类的实例？
3. 在 Vector 中存在什么样的方法来实现它的抽象行为？

接下来的三节会依次探索上述问题中的答案。

5.1.1 指定 Vector 的基类型

在 C++ 中，集合类通过在类的名字后面添加一对包含类型名称的尖括号来指定其包含对象的类型。例如：类 Vector<int> 指定了其元素类型为整数，Vector<char> 指定了其元素类型为字符，Vector<string> 指定了其元素类型为字符串。尖括号内的类型被称为集合的**基类型**（base type）。

包含基类型规格说明的类在面向对象中被称之为**参数化的类**（parameterized class）。在 C++ 中，参数化的类通常被称为**模板**（template），它反映了 C++ 编译器将 Vector<int>、Vector<char>，以及 Vector<string> 这些共享相同结构的类当作独立的类的事实。Vector 这个名字扮演了产生只有其包含的值的类型不同的类模板的角色。现在你需要理解

的是如何使用类模板，实现基本模板的过程将在第 14 章中阐述。

5.1.2 声明 Vector 对象

抽象数据类型背后的其中一个哲学原理是：用户应能够想到它们好像就是一个内置的基本类型。因此，正如你会用下面的声明语句来声明一个整型变量一样：

```
int n;
```

通过编写下面的声明语句，也应该可以声明一个新的 Vector 对象：

```
Vector<int> vec;
```

198

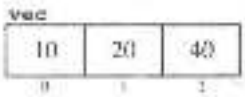
在 C++ 中，那恰恰是你要做的。在上述的声明语句中，定义声明了一个 Vector 类的名为 vec 的新变量，且正如其尖括号内的模板参数所指明的，它的基类型为整型。

5.1.3 Vector 的操作

当你声明了一个 Vector 对象时，它的初值是一个空矢量（empty vector），这意味着它里面没有包含任何元素。因为空的 Vector 用处不大，你需要学习的第一件事就是如何向一个 Vector 对象里面添加新元素。通常的方法是调用 Vector 的 add 方法，它会在 Vector 对象的最后添加一个新的元素。例如，如果 vec 是一个之前声明过的基类型为整数的空 Vector 对象，执行以下代码：

```
vec.add(10);
vec.add(20);
vec.add(40);
```

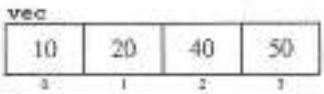
便会将 vec 变为含有值分别为 10、20 和 40 的三个元素的 Vector 对象。和字符串中的字符一样，C++ 中 Vector 对象元素的索引从 0 开始，这就意味着你可以用下图所示来表示 vec 中的内容：



与第 11 章将要介绍的大多数的原始数组类型不同，Vector 对象的大小是不固定的，这意味着你可以在任何时候向其添加新元素。例如，在上述代码后你可以调用以下语句：

```
vec.add(50);
```

它会在 Vector 对象 vec 的最后添加一个值为 50 的元素，如下图所示：

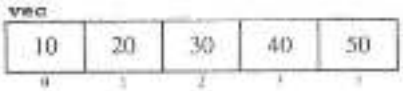


Vector 的 insert 的方法允许你在一个 Vector 对象中添加新元素。insert 方法的第一个参数是插入元素的索引号，新的元素将会插入在索引号之前的位置上。例如，调用以下语句：

```
vec.insert(2, 30);
```

将在索引位置为 2 的元素前面插入一个值为 30 的元素，如下图所示：

199

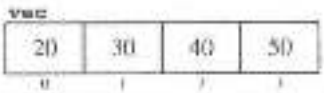


Vector 类在其内部实现这个操作需要扩大数组的存储空间，并且将值为 40 和 50 的元素向后移动一位从而为值为 30 的元素提供存储空间。从用户的观点看，这个操作的实现很少考虑那些细节，并且也不需要理解它是如何实现的。

Vector 类同时也允许你删除其中的元素。例如，调用以下语句：

```
vec.remove(0);
```

将删除索引位置为 0 的元素，之后 vec 中的元素如下图所示：

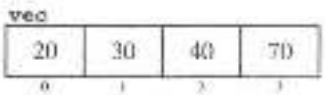


Vector 类包含两个查询和修改单个元素的方法。get 方法获取索引值所对应的元素，并将某元素返回。例如，给定上图所示的 vec 的值，调用 vec.get（2）将会返回值 40。

与此相对应，你可以使用 set 方法来改变 vec 中已经存在的元素值。例如，调用以下语句：

```
vec.set(3, 70);
```

将索引位置为 3 的值由 50 改变为 70，如下图所示：



get、set、insert 和 remove 方法全都会检查和确保你提供的索引值对于 Vector 对象来说是合法的。例如，如果你调用 vec.get（4），get 方法将会调用 error，并报告索引值为 4 对 vec（索引值在 0～3 之间）而言太大了，已越界。对于 get、set 和 remove 方法，Vector 的实现将会检查其索引值是否大于等于 0 且小于元素的总数。insert 方法允许索引值等于元素的个数，这种情况下新的元素值将会被插入到数组的最后，这正与 add 方法相同。

测试一个索引是否合法的操作称为**边界检查**（bounds-checking）。边界检查更易于捕获编程中的错误，而传统的数组操作常常忽视这些错误。

5.1.4 从 Vector 对象中选择元素

虽然 get 和 set 方法易于使用，但几乎每个人都不调用这些方法。让 C++ 与其他大多数编程语言不同的特性之一是在类中可以重载标准操作符。这种特性使得 Vector 类支持用方括号去指定想要得到的特定索引的元素值这种更传统的语法变为可能。因此，为了查找位置 i 处的元素，就像使用传统数组那样，你可以使用表达式 vec[i]。此外，你可以通过指定一个新的值赋给 vec[i] 来改变其值。例如，你可以在 vec 中查找索引为 3 的元素，并将其赋值为 70：

```
vec[3] = 70;
```

上述结果的语法要比调用 set 方法更加简短，且它与数组操作更为相似，因此 Vector 类去设计模仿这种方法。

数组中用来查找一个元素的索引可以是任何与整数相等价的表达式。其中最常见的一个索引表达式为 for 循环的索引，循环遍历的每一个索引值是有序的。在一个 Vector 对象中，对于通过改变索引位置的循环大体模式都如下所示：

```
for (int i = 0; i < vec.size(); i++) {
   loop body
}
```

在上述循环体内，你可以引用当前的元素 vec[i]。

作为一个例子，下面的代码输出 vec 中元素的内容，其中元素内容是以方括号括起来并以逗号相分隔：

```
cout << "[";
for (int i = 0; i < vec.size(); i++) {
   if (i > 0) cout << ", ";
   cout << vec[i];
}
cout << "]" << endl;
```

如果你执行以上这段代码并假定 vec 的内容是最新的，你将会在屏幕上得到下面的输出： 201



5.1.5 作为参数传递 Vector 对象

上一小节结束处的代码是非常有用的（尤其是你在调试并且需要查看一个 Vector 对象中所包含的元素时），为此值得定义一个函数。在某种程度上，把代码封装在一个函数内很简单，你所要做的就是在上述代码的基础上添加一个合适的函数头部，如下所示：

```
void printVector(Vector<int> & vec) {
   cout << "[";
   for (int i = 0; i < vec.size(); i++) {
      if (i > 0) cout << ", ";
      cout << vec[i];
   }
   cout << "]" << endl;
}
```

然而，函数头部那行所涉及的精妙之处就是在你高效使用集合类之前你必须理解它。正如第 2 章所描述的，在参数名字前面的“&”表明参数 vec 是通过引用传递的。这就意味着函数内的值和调用者的值是共享的，引用调用比 C++ 默认的需要复制 Vector 对象中每一个元素的值调用更高效。在 printVector 例子中，没有必要去复制 Vector 对象中的元素。使用引用调用（尤其是对于像集合类这样的大的数据结构而言）可以节省大量的执行时间。

也许更为重要的是，使用引用调用可以通过编写一个函数来改变 Vector 对象中的内容。例如，下面的函数在一个元素类型为整数的 Vector 对象中删除任何零值元素：

```
void removeZeroElements(Vector<int> & vec) {
   for (int i = vec.size() - 1; i >= 0; i--) {
      if (vec[i] == 0) vec.remove(i);
   }
}
```

202 上述程序中的 for 循环遍历了 Vector 对象 vec 的每一个元素，并且检查其元素值是否为 0。如果其值为 0，该函数则调用 remove 方法将这个元素从 vec 中删除。为了保证删除一个元素并不会改变待检查元素的位置，该 for 循环是从 vec 的最后一个元素开始，然后反向的检查并操作元素。

这个函数依赖于引用调用的使用。如果你忽视函数头部那行的 &，removeZeroElements 将不会有效。函数中的代码将会删除函数内复制的 vec 的零值，并不是调用者提供的 Vector 对象的零值。当 removeZeroElements 返回时，复制的 vec 将会消失，导致原始 Vector 对象的值并未改变。这种错误很容易产生，当你的程序出错时，你应该学会找到这类错误。

图 5-1 中的 ReverseFile 程序展现了一个完整的 C++ 程序，它使用 Vector 从一个文件中以逆序来显示其中的行。你会发现函数 promptUserForFile 以及 readEntireFile 在多个应用中都会使用到。基于此，这两个函数连同许多其他对文件有用的函数，都被包含在 filelib.h 库中。

5.1.6 创建预先定义大小的 Vector

你看到过的例子相当于开始是一个空的 Vector 对象，然后再逐个地向其中添加元素。在许多应用中，每次创建的 Vector 对象中只有一个元素是很麻烦的，尤其是你预先已经知道了 Vector 对象的大小。这种情况下，在声明部分指明元素的个数是明智的。

例如，假设你想创建一个 Vector 对象用来保存高尔夫课程上的 18 个洞各自的分数。你已经知道的解决方法就是创建一个空的 Vector<int>，然后利用 for 循环向其中添加 18 个元素值，如下述代码所示：

```
const int N_HOLES = 18;

Vector<int> golfScores;
for (int i = 0; i < N_HOLES; i++) {
   golfScores.add(0);
}
```

但一个更好的方法就是将 Vector 对象的大小作为参数添加到对象声明中，如下所示：

```
Vector<int> golfScores(N_HOLES);
```

这个声明创建了一个含有 N_HOLES 个元素的 Vector 对象 golfScores，由于 Vector 的基类型为 int，每个初始值都为 0。这两段不同代码的功能是相同的。每段代码都创建了一个包含 18 个零值的 Vector<int> 对象。但第一段代码要求用户去初始化其中的每个元
203 素，而第二段代码的 Vector 类有自身初始化其元素的功能。

```
/*
 * File: ReverseFile.cpp
 * ---------------------
 * This program displays the lines of an input file in reverse order.
 */

#include <iostream>
#include <fstream>
#include <string>
#include "filelib.h"
#include "vector.h"
using namespace std;

/* Function prototypes */

void readEntireFile(istream & is, Vector<string> & lines);

/* Main program */

int main() {
   ifstream infile;
   Vector<string> lines;
   promptUserForFile(infile, "Input file: ");
   readEntireFile(infile, lines);
   infile.close();
   for (int i = lines.size() - 1; i >= 0; i--) {
      cout << lines[i] << endl;
   }
   return 0;
}

/*
 * Function: readEntireFile
 * Usage: readEntireFile(is, lines);
 * ---------------------------------
 * Reads the entire contents of the specified input stream into the
 * string vector lines.  The client is responsible for opening and
 * closing the stream.
 */

void readEntireFile(istream & is, Vector<string> & lines) {
   string line;
   while (getline(is, line)) {
      lines.add(line);
   }
}
```

图 5-1 以逆序显示文件中每行内容的程序 [204]

一个更重要的例子就是当你想要声明一个具有固定大小的 Vector 对象，考虑图 5-2 中的程序 LetterFrequency，它的功能为计算 26 个字母在一个数据文件各自出现的次数。这些计算值存储在变量 letterCounts 中，它以下述语句进行声明：

```
Vector<int> letterCounts(26);
```

该 Vector 对象 letterCounts 中包含的每个计数对应的字母的顺序和字母表中索引的顺序相同。也就是字母 A 的个数存储在 letterCounts[0] 中，字母 B 的个数存储在 letterCounts[1] 中，以此类推。对于文件中的每一个字母，该程序所做的就是在 Vector 对象 letterCounts 中的恰当位置上增加其相对应的值。

这个恰当位置是通过利用字符的 ASCII 码值通过算术计算获得。该计算表达式如下：

```
letterCounts[toupper(ch) - 'A']++;
```

用当前出现的字符 ch 的大写字母的字符 ASCII 码值减去字母 "A" 的 ASCII 码值就得到了字母表中 ch 的索引。接着，这个表达式就更新了 Vector 对象中该元素的计数值。

接下来，程序最关心的就是以列对齐的方式格式化地输出文件中每个字母的计数值。例

如，计算 George Eliot 的《Middlemarch》中字母计数结果的 letterCounts 程序如下所示：




```
/*
 * File: LetterFrequency.cpp
 * -------------------------
 * This program counts the frequency of letters in a data file.
 */

#include <iostream>
#include <iomanip>
#include <fstream>
#include <cctype>
#include "filelib.h"
#include "vector.h"
using namespace std;

/* Constants */

static const int COLUMNS = 7;

/* Main program */

int main() {
   Vector<int> letterCounts(26);
   ifstream infile;
   promptUserForFile(infile, "Input file: ");
   char ch;
   while (infile.get(ch)) {
      if (isalpha(ch)) {
         letterCounts[toupper(ch) - 'A']++;
      }
   }
   infile.close();
   for (char ch = 'A'; ch <= 'Z'; ch++) {
      cout << setw(COLUMNS) << letterCounts[ch - 'A'] << " " << ch << endl;
   }
   return 0;
}
```

图 5-2 计数一个文件中字母的个数

5.1.7 Vector 类的构造函数

到目前为止，本章的示例程序中声明一个 Vector 对象用了两种不同的方法。图 5-1 的 ReverseFile 程序采用以下声明定义了一个空的字符串向量：

```
Vector<string> lines;
```

LetterFrequency 程序用以下语句声明了一个含有 26 个零值的 Vector 对象： 205 ; 206

```
Vector<int> letterCounts(26);
```

这些声明实际运行的行为要比我们见到的要复杂得多。当你声明一个基本类型的变量时，例如：

```
double total;
```

C++ 不会初始化这个变量。内存习惯于给这个变量 total 存放一个该地址原本存在的值，这就导致了我们常常得到一些不可预测的结果。因此，基本类型的变量声明通常包含一个明确的初始化表达式来给该变量设置一个我们想要的初始值，就像如下语句：

```
double total = 0.0;
```

当你声明一个变量是 C++ 类的实例的时候，这种情况就不一样了。在这种情况下，C++ 自动地通过调用一种称为类的构造函数（constructor）的特殊方法来初始化这个对象。例如以下声明：

```
Vector<string> lines;
```

C++ 调用了 Vector 类的构造函数，该方法将变量 lines 初始化为一个属于参数化类 Vector<string> 的空的 Vector 对象。以下声明：

```
Vector<int> letterCounts(26);
```

调用 Vector 类的另一个构造函数初始化并形成了一个含有 26 个元素的 Vector<int> 对象。

和调用重载函数一样，C++ 的编译器通过查看类声明中构造函数的参数列表来决定调用类中哪一个版本的构造函数。lines 变量的声明没有提供参数，这就告诉编译器调用的构造函数不需要参数，通常我们称这种构造函数为默认构造函数（default constructor）。letterCounts 变量的声明提供了一个整型参数，它告诉编译器调用含有一个整数来指明 Vector 对象大小的构造函数。Vector 类中这两种版本的构造函数连同 Vector 的其他方法列于表 5-1 中。

如果你认真阅读了表 5-1 构造函数的描述，你会发现第二种形式的构造函数接受一个可选择的参数来指明每个元素的初始值。在那些元素初始值为基本类型的默认值（default value）的场景中，这个参数通常被省略，例如数值类型的初始值是 0，bool 类型的初始值是 false，char 类型的初始值是 ASCII 码值为 0 所对应的那个字符。对于类而言，默认值 207 是通过调用默认构造函数来确定的。

表 5-1 Vector.h 接口中的条目

构造函数	
Vector<*type*> ()	创建一个空的 Vector 对象
Vector<*type*> (*n*, *value*)	创建一个含有 *n* 个元素的 Vector 对象，每个元素都被初始化为 *value*，如果 *value* 实参省略，则取 *value* 类型的默认值

（续）

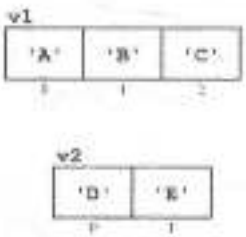
方法	
size()	返回 Vector 对象中元素的个数
isEmpty()	若 Vector 对象为空，返回 true
get(*index*)	返回指定索引位置 *index* 的元素。尝试获取 Vector 对象边界外的元素会发生错误
set(*index*, *value*)	给指定位置 *index* 的元素设置新的值 *value*。尝试给 Vector 对象边界外的元素设置新值会发生错误
add(*value*)	在 Vector 对象的尾部添加一个新的元素 *value*
insertAt(*index*, *value*)	在指定位置 *index* 之前插入一个新值 *value*
removeAt(*index*)	删除指定位置 *index* 上的元素
clear()	删除 Vector 对象中的所有元素

操作符	
vec[*index*]	查找指定位置 *index* 上的元素。尝试获取 Vector 对象边界外的元素会发生错误
v_1 + v_2	连接 v_1 和 v_2，返回一个包含 v_1 和 v_2 中的所有元素的 Vector 对象
vec += e_1, e_2,...	在 Vector 对象的尾部添加元素 e_1、e_2 等

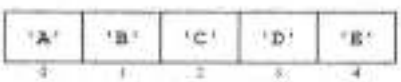
5.1.8 Vector 中的操作符

除表 5-1 所列的方法之外，Vector 类也定义了一些适用于 vector 对象的操作符。你已经见过方括号的使用了，这使得利用数组的传统的查找方法从一个 Vector 对象中查找元素变为可能。Vector 类还定义了操作符 + 和 += 作为连接两个 Vector 对象和在一个已
208 经存在的 Vector 对象的尾部添加元素的简洁形式。

Vector 对象中的 + 操作符与其在字符串中的使用相同。如果 Vector 类的基类型为字符，字符向量 v1、v2 包含的值为：



计算表达式 v1+v2，将产生一个新的含有五个元素的 Vector 对象，如下所示：



+= 操作符把元素添加到一个已经存在的 Vector 对象的结尾处，但是它在初始化一个 Vector 对象的内容时也很有用。例如，你可以声明和初始化一个 Vector 对象 v1，如下所示：

```
Vector<char> v1;
v1 += 'A', 'B', 'C';
```

如果你使用 2011 年新颁布的 C++ 新标准 C++11，你可以简化这个初始化，如下所示：

```
Vector<char> v1 = { 'A', 'B', 'C' };
```

5.1.9 表示二维结构

在 Vector 类中的基类型参数可以是任意的 C++ 类型，甚至可以是该模版类本身。特别是你可以通过声明一个基类型为 Vector 类型本身的 Vector 对象来创建一个二维结构的 Vector。以下声明：

```
Vector< Vector<int> > sudoku(9, Vector<int>(9));
```

将变量 sudoku 初始化为一个含有九个元素的 Vector 对象，其中每个元素又都是含有九个元素的 Vector 对象。内部的 Vector 对象的基类型为 int，外部的 Vector 对象的基类型是 Vector<int>。整个集合的类型是

```
Vector< Vector<int> >
```

209

尽管某些 C++ 编译器在不断地扩展以至于对象的空间是任意的，但这种声明坚持了 C++ 标准，即在内部类型参数周围留有空间。这个空间确保了尖括号中类型参数被正确的解释。如果你用下面的声明代替上面的声明：

```
Vector<Vector<int>> sudoku(9, Vector<int>(9));
```



许多 C++ 编译器会将 ">>" 解释为一个单个的操作符，并且不能编译通过这一代码行。

5.1.10 Stanford 类库中的 Grid 类

尽管使用嵌套的 Vector 对象能够代表二维结构，但这种方法很不便利。为了简化那些需要二维结构的应用开发，Stanford 集合类库提供了一种名为 Grid 的类，尽管在标准模板库中并没有对应的类。grid.h 中的条目如表 5-2 所示。

表 5-2 grid.h 接口中的条目

构造函数	
Grid<*type*>()	创建一个空的 grid 对象。那些使用默认构造函数的用户需要通过调用 resize 方法指定 grid 对象的维数
Grid<*type*>(*rows*, *cols*)	创建一个指定具体行数和列数的 grid 对象。其中每个元素被初始化为元素类型的默认值
方法	
numRows()	返回 grid 对象中的水平行数
numCols()	返回 grid 对象中的垂直列数
inBounds(*row*, *col*)	如果指定的行和列组成的坐标在 grid 对象中，返回 true
get(*row*, *col*)	返回 grid 对象中指定的行和列所在的元素
set(*row*, *col*, *value*)	给指定坐标位置上的元素设置新的值 value
resize(*rows*, *cols*)	通过给定的行数 *rows* 以及列数 *cols* 来改变 grid 对象的尺寸。grid 对象中先前的内容都被丢弃
操作符	
grid[*row*][*col*]	在 grid 对象中选择指定行和列所在的元素

210

5.2 Stack 类

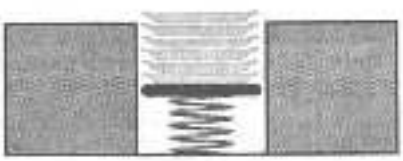
依据集合类所支持的方法来进行评估，最简单的集合类就是 Stack 类，由于它的简单性，使得它在很多程序应用中非常有用。从概念上讲，一个栈（stack）提供了数据值集合的存储，并且删除一个栈中的值的方向必须要和值添加进来的方向相反。这就告诉我们最后一个添加进来的值总是第一个被删除。

由于栈在计算机科学中的重要作用，故它有自己的术语。向一个栈中添加一个新的元素被称为入栈（push），从栈中删除最近的元素称为出栈（pop）。此外，栈中处理的顺序有时候被称为 LIFO（Last In First Out），它表示“后进先出”。

词语“栈”、“入栈”以及“出栈”的一种常见的（可能是假的）解释就是栈类型继承于咖啡馆的盘子存储方式。如果你在一个咖啡馆中，那里的客人在一个自助的流水线的开始处拿起他的盘子，这些盘子用弹簧顶住被排成一列，使得客人能够方便地拿到最上面的盘子，如下图所示：



当洗碗工添加新盘子时，将它放在盘子的最上面，随着弹簧被压缩，其他的盘子轻微地下降，如下图所示：



客人们只能拿到最上面的盘子。当他们拿走盘子后，剩下的盘子将会上升。最后一个添加进来的盘子是第一个被客人拿走的。

栈对于编程很重要的一个基本原因就是：函数嵌套调用如同栈的类作。例如：如果主函数调用了一个名为 f 的函数，f 函数的栈帧（stack frame）将会添加进 main 函数的栈帧中的
211 顶端，如下图所示：

```
int main() {
  void f() {
    cout << "This is the function f" << endl;
    g();
  }
```

如果 f 调用 g，g 的一个新的栈帧将会添加进 f 的栈帧的顶端，如下图所示：

```
int main() {
  void f() {
    void g() {
      cout << "This is the function g" << endl;
    }
```

当 g 返回时，它的栈帧从栈中弹出，将 f 恢复到栈顶，如原图所示。

5.2.1 Stack 类结构

和 Vector 类以及 Grid 类一样，Stack 类也是一个需要指明元素类型的集合类。例如，Stack<int> 代表了一个元素类型为整型的栈，Stack<string> 代表了一个元素类型为字符串的栈。类似地，如果你定义了类 Plate 以及 Frame，你可以使用这些类的对象作为元素来创建 Stack<Plate> 和 Stack<Frame> 对象。stack.h 中的条目如表 5-3 所示。

表 5-3 stack.h 接口中的条目

构造函数	
Stack<type>()	创建一个空栈可存储指定类型值的栈对象
方法	
size()	返回当前栈中的元素个数
isEmpty()	如果栈为空，则返回 true
push(value)	将值 value 压入栈顶
pop()	将栈顶元素出栈，并且将栈顶元素返回给调用者。在空栈中调用 pop 方法会产生错误
peek()	返回栈顶元素的值但并不出栈。在空栈中调用 peek 方法会产生错误
clear()	删除栈中的所有元素

212

5.2.2 栈和小型计算器

栈一个最有趣的应用就是电子计算器，它被用来存储计算的中间结果。尽管栈在大多数计算器的操作中担演了一个核心角色，但是在那些需要用户输入逆波兰表示法（reverse Polish notation，RPN）的早期科学计算器中能很明显地看到栈的作用。

在逆波兰表示法中，操作符在那些它们作用的操作数之后输入。例如，为了用逆波兰式计算器计算表达式

```
8.5 * 4.4 + 6.9 / 1.5
```

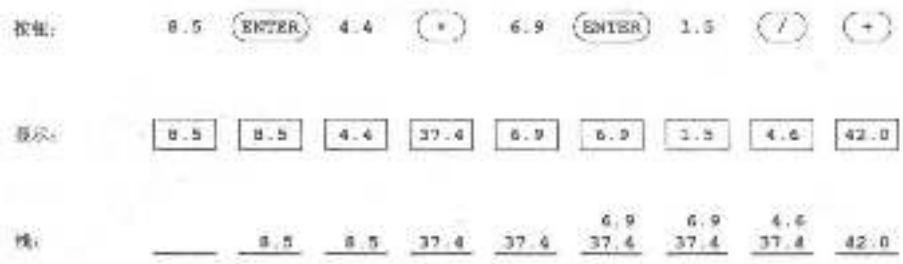
的结果，你将会按照下面的顺序输入操作数和操作符：

8.5 (ENTER) 4.4 (*) 6.9 (ENTER) 1.5 (/) (+)

当按下 ENTER 按钮时，计算器获得了先前的值并将该值压入栈。当按下操作符按钮时，计算器首先会检查用户是否已经输入了一个值，如果已经输入了，那么会自动地将其从栈中弹出。然后通过以下步骤计算应用操作符的结果：

- 从栈顶弹出两个值。
- 对这些值进行由按钮所指定的算术操作。
- 将结果压回到栈。

除了当用户在输入数值时，计算器总是显示栈顶的值。因此，图 5-3 显示了在计算中的每一时刻，计算器所显示的内容以及栈中所包含的值。



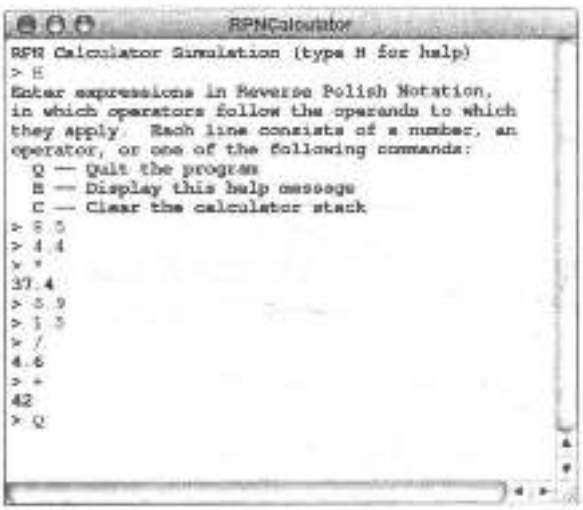
213

图 5-3 RPN 计算器的计算序

用 C++ 实现逆波兰式计算器需要在用户接口设计上做出一些改变。在一个真正的计算器中，数字和操作符都是出现在键盘上的。在这种实现方法下，我们可以很简单的想象用户在命令行中输入的每一行，并且这些行内容的形式有以下几种：

- 一个浮点数。
- 一个从 +、-、* 和 / 中选出的算术操作符。
- 字母 Q，它的作用是终止程序。
- 字母 H，它的作用是输出帮助消息。
- 字母 C，它的作用是清除当前栈中所有的值。

一个正在运行的计算器程序的例子如下图所示：



由于用户以 RETURN 键作为每一个数字输入的结束，并且它只是表明一个数字输入完成，因此没有必要对 RETURN 按钮做一个副本。当用户输入数字时，计算器程序仅仅是将这些数字添加到栈中。当计算器读到一个操作符时，它会弹出栈顶的两个元素，再基于操作符计算出结果，并且展示这个结果，最后再将结果压入栈。

214 图 5-4 是计算器应用的完整的实现方法。

```
/*
 * File: RPNCalculator.cpp
 * -----------------------
 * This program simulates an electronic calculator that uses
```

图 5-4 实现一个简单的 RNP 计算器程序

```
 * This program simulates an electronic calculator that uses
 * reverse Polish notation, in which the operators come after
 * the operands to which they apply.  Information for users
 * of this application appears in the helpCommand function.
 */

#include <iostream>
#include <cctype>
#include <string>
#include "error.h"
#include "simpio.h"
#include "stack.h"
#include "strlib.h"
using namespace std;

/* Function prototypes */

void applyOperator(char op, Stack<double> & operandStack);
void helpCommand();

/* Main program */

int main() {
   cout << "RPN Calculator Simulation (type H for help)" << endl;
   Stack<double> operandStack;
   while (true) {
      string line = getLine("> ");
      if (line.length() == 0) line = "Q";
      char ch = toupper(line[0]);
      if (ch == 'Q') {
         break;
      } else if (ch == 'C') {
         operandStack.clear();
      } else if (ch == 'H') {
         helpCommand();
      } else if (isdigit(ch)) {
         operandStack.push(stringToReal(line));
      } else {
         applyOperator(ch, operandStack);
      }
   }
   return 0;
}
/*
 * Function: applyOperator
 * Usage: applyOperator(op, operandStack);
 * ---------------------------------------
 * Applies the operator to the top two elements on the operand stack.
 * Because the elements on the stack are popped in reverse order,
 * the right operand is popped before the left operand.
 */

void applyOperator(char op, Stack<double> & operandStack) {
   double result;
   double rhs = operandStack.pop();
   double lhs = operandStack.pop();
   switch (op) {
    case '+': result = lhs + rhs; break;
    case '-': result = lhs - rhs; break;
    case '*': result = lhs * rhs; break;
    case '/': result = lhs / rhs; break;
    default:  error("Illegal operator");
   }
   cout << result << endl;
   operandStack.push(result);
}

/*
 * Function: helpCommand
 * Usage: helpCommand();
 * ---------------------
 * Generates a help message for the user.
 */

void helpCommand() {
```

图 5-4 (续)

```
   cout << "Enter expressions in Reverse Polish Notation," << endl;
   cout << "in which operators follow the operands to which" << endl;
   cout << "they apply.  Each line consists of a number, an" << endl;
   cout << "operator, or one of the following commands:" << endl;
   cout << "  Q -- Quit the program" << endl;
   cout << "  H -- Display this help message" << endl;
   cout << "  C -- Clear the calculator stack" << endl;
}
```

215
:
216

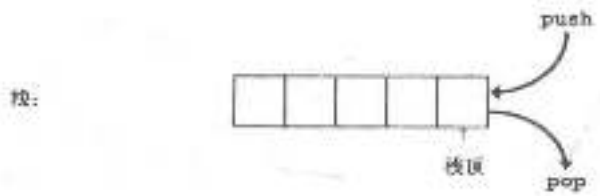
图 5-4 （续）

5.3 Queue 类

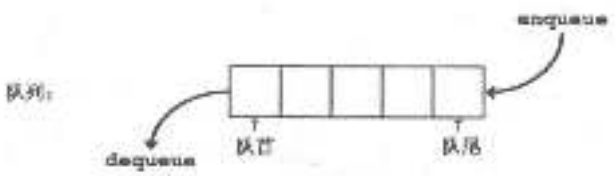
正如你在 5.2 节所学过的，栈的典型特点就是最后输入的总是最先输出。并强调了这种行为经常在计算机科学中作为 LIFO 被提到，其中 LIFO 是短语 "last in, first out" 的首字母缩写。LIFO 原则在编程环境中非常有用，因为它反映了函数的调用操作，最近调用的函数总是最先返回。然而，在现实社会中，"last in, first out" 模型相对来说很少。实际上，在人类社会中，我们集体的公平的分配表示法被表达成为 "先来先被服务（first come, first served）"。在编程中，这种顺序策略的短语是 "先进先出（first in, first out）"，习惯上简写为 FIFO。

一个使用 FIFO 原则存储数据的数据结构被称为队列（queue）。队列基本的操作（和栈的操作 push 以及 pop 相似）被称为入队（enqueue）和出队（dequeue）。入队操作在队列的最后添加一个新元素，通常被称之为队尾（tail）。出队操作删除队列开头的元素，通常被称之为队首（head）。

栈和队列结构之间的差异可以用下图简单地阐明。在一个栈中，用户必须在数据结构的同一端（栈顶）添加和删除元素，如下图所示：



在队列中，用户在一端添加元素而在另一端删除元素，如下图所示：



可能和你期望的一样，上述模型是非常熟悉的，Queue 类看起来非常像 Stack 类的一个对应物。表 5-4 的条目证实了这个假定。它们只在术语上存在差异，它反映了元素顺序的不同。

217

表 5-4 queue.h 接口中的条目

构造函数	
Queue<type>()	创建一个空的能存储指定类型值的队列对象

（续）

方法	
size()	返回当前队列中元素的个数
isEmpty()	若队列为空，则返回 true
enqueue(value)	将值 value 添加到队尾
dequeue()	删除队首元素并将此元素返回给调用者。在空队列中调用 dequeue 方法将产生错误
peek()	返回队首元素的值但并不将其从队列中删除。在空队列中调用 peek 方法将产生错误
clear()	删除队列中的所有元素

队列这种数据结构在编程中有很多应用，队列出现在很多以确保服务能够被公正对待为目的的先进 - 先出原则的场景中。例如，如果你的工作环境中只有一台打印机可共享给多台电脑，那么打印软件被设计成所有的打印请求都将进入一个队列。因此，如果几个用户决定输入打印请求，队列结构能确保每个用户的请求按照收到的请求的顺序执行。

程序中出现的队列大多数都是相同的，它们都是模仿排队的行为。例如，如果你想要决定一个超市里面需要多少个收银员。那么通过编写一个程序来模拟顾客在商店中的行为是值得的。这样的一个程序基本上会涉及队列，因为一个结账台是以先进 - 先出这种方式进行的。那些完成购物的顾客们进入结账队列中等待付账。每一个顾客最终到达队列的前端，在这里收银员计算总的付款金额然后收钱。模拟的这个行为代表了应用程序的一个重要类，它值得花费一些时间来理解这个模拟是如何工作的。

5.3.1 仿真和模型

在编程世界之外，真实世界中有永无止境的事情和过程（尽管它们的确都很重要），它们都太复杂了以致于难以理解。例如，知道各种污染物是如何影响臭氧层以及臭氧层改变时 218
如何影响全球气候是非常有用的。类似地，如果经济学家和政治领导人对国家经济是如何运转的有一个更加全面准确的理解，他们将能够对降低资本利益税是刺激投资还是加剧已经存在的贫富差距做出评估。

当遇到像这样的大规模问题时，通常给出一个理想的能够代表简化的真实世界过程的模型是必要的。大多数问题都太复杂了以至于无法完全理解，这些问题都有太多的细节。构建一个模型的主要原因在于：除了特定问题的复杂性外，我们能够在没有影响这个问题的基本特性的前提下做出一些必然的假设来帮助你简化一个复杂的过程。对于一个过程可以想出一个合理的模型，你可以把这个模型动态地翻译成能够捕获该模型的行为的程序。这样的程序称为**仿真**（simulation）。

创建一个仿真通常由两个步骤组成，记住这一点很重要。第一个步骤是设计一个概念模型，用来仿真真实世界中的行为。第二个步骤是编写一个能够实现这个概念模型的程序。这两个步骤可能都会发生错误，因此保持对仿真以及它们在真实世界中的适用性的怀疑是明智的。在现实社会中，计算机提供的“结果”只在一定条件下才可以相信，认识到仿真永远比不上它们所基于的模型是至关重要的。

5.3.2 排队模型

假设你想设计一个超市排队行为模型的仿真。通过模拟排队，你可以得到一些有用的排队模型的性质，并且这些性质可能会帮助公司作出决策。例如需要多少个收银员以及需要给

队列提供多大的空间等。

在编写一个结账队列仿真的过程中，第一步是开发出一个排队模型，在这个模型中你可以确定任何简化的假设。例如，为了使这个仿真最初的实现尽可能的简单，你可以假设这里只有一个收银员只为一个队列进行服务，且顾客随机过来排队。每当收银员空闲，并且队列里面还有人，收银员就开始为那个顾客服务。在经过一个适当的服务时期（这在某种程度上
219 需要建模），收银员完成了当前的交易，接着为队列中的下一个顾客服务。

5.3.3 离散时间

模型中另一个经常用到的假设就是对精确度等级的限制。在仿真结账队列时，收银员服务一个顾客所花费的时间在一定限度内是不断变化的。一个顾客可能要花费两分钟，另一个顾客可能要花费六分钟。然而，考虑用分钟数来衡量时间是否让仿真变得充分地精确，这是非常重要的。如果你有一个非常精确的秒表，你可能会发现一个顾客花费了 3.141 592 65 分钟。你需要解决的问题是你需要什么样的精度。

对于大多数模型而言，尤其是这些打算作为仿真的模型来说，引入一个简化的假设是非常有用的，即模型中所有事件发生的时间都是离散的整数时间。假设你能找到一个单位时间（建模的目的）并使用它，那么你可以将其看成是不可分割的。总之，在一个仿真中用到的单位时间一定要足够小，以至于在一个单个的时间单元中发生超过一个事件的可能性是可以忽略的。例如，在结账队列仿真中，分钟可能不够精确，两个顾客很有可能在同一分钟内到达。但是，你可以使用秒作为单位时间以降低两个顾客在同一秒钟内恰好同时到达的可能性。

接下来的章节将采取用秒作为度量时间的策略。然而，没有理由要你必须用传统的时间单位来度量时间。当你编写一个仿真程序时，你可以定义任何适合模型结构的单位时间。例如，你可以定义五秒钟为一个单位时间，然后在一系列五秒的时间间隔中运行仿真。

5.3.4 仿真时间中的事件

使用离散时间单位的一个优点是可以使用 int 类型变量而不是采用较低效的 double 类型变量。使用离散时间单位的另一个更为重要的优点是它允许你把仿真结构作为一个循环，而每个时间单元代表其中的一次单独的循环。当你用这种方法处理问题时，仿真程序可能具有如下形式：

```
for (int time = 0; time < SIMULATION_TIME; time++) {
    Execute one cycle of the simulation.
}
```

在上述循环体内，程序在一个仿真单位时间内执行必要的操作。

220 思考一下在结账队列仿真中的每个单位时间可能发生的事件。一种可能就是一个新的顾客到达。另一种可能就是收银员完成了对当前顾客的服务，然后再去为队列中的下一个顾客服务。这些事件带来了一些有趣的问题。为了完成这个模型，你需要知道顾客多久来一次以及他们在收银机前所花费的时间。你可以通过观察商店中真实的结账队列来收集近似的信息。即使你收集了这些信息，你仍然需要对这些信息进行简化：包含足够多的真实世界中的行为是有用，并且就这个模型来说它是易于理解的。例如，你的调查表明顾客平均每隔 20 秒就有一人进入队列。对于这个模型的输入来说，这个平均的到达率是非常有用的。另一方

间，你可能对每隔 20 秒就有一个顾客到达的仿真不是很有信心。这样的实现可能会违反真实世界中顾客到达是随机的情况，以及他们经常是同时到达的情况。

由于上述原因，到达过程通常以指定在任意的一个离散的单位时间内顾客到达的概率建模，而不是根据到达者之间的平均时间来进行建模。例如，如果你的研究表明：每隔 20 秒有一个顾客到达，那么在任何一秒钟顾客到达的平均概率是 1/20 或者 0.05。如果假设在每一个单位时间内顾客以等概率到达，那么到达模型就形成了一种模式，这种模式以法国数学家 Siméon Poisson（1781 — 1840）的名字命名，被数学家称为泊松分布（Poisson distribution）。

你可能还会选择对收银员为一个顾客服务多长时间做出一个简化的假设。例如，如果你假设每个顾客所需的服务时间在一定范围内是分布均匀的，那么这个程序将很容易编写。如果你那样做的话，你可以从 random.h 接口使用 randomInteger 函数来选择服务时间。

5.3.5 实现仿真

即使这个程序比本章其他程序要长，但是这个程序代码更易于编写。图 5-5 给出了程序 checkoutLine 的代码。该仿真的核心是一个循环，它根据参数 SIMULATION_TIME 所指定的秒的数量来运行。每一秒，该仿真都会完成下面的操作：

1. 检查是否有顾客到达，如果有的话，那么将该顾客加入到队列中。
2. 如果收银员当前处于服务状态，提示收银员还需要为当前的顾客服务多长时间。最终，这个要求服务的时间结束，收银员变为空闲。
3. 如果收银员是空闲的，将会为队列中的下一个顾客服务。 221

```
/*
 * File: Checkoutline.cpp
 * ----------------------
 * This program simulates a checkout line, such as one you
 * might encounter in a grocery store.  Customers arrive at
 * the checkout stand and get in line.  Those customers wait
 * in the line until the cashier is free, at which point
 * they are served and occupy the cashier for some period
 * of time.  After the service time is complete, the cashier
 * is free to serve the next customer in the line.
 *
 * In each unit of time, up to the constant SIMULATION_TIME,
 * the following operations are performed:
 *
 * 1  Determine whether a new customer has arrived.
 *    New customers arrive randomly, with a probability
 *    determined by the constant ARRIVAL_PROBABILITY.
 *
 * 2  If the cashier is busy, note that the cashier has
 *    spent another minute with that customer.  Eventually,
 *    the customer's time request is satisfied, which frees
 *    the cashier.
 *
 * 3  If the cashier is free, serve the next customer in line.
 *    The service time is taken to be a random period between
 *    MIN_SERVICE_TIME and MAX_SERVICE_TIME.
 *
 * At the end of the simulation, the program displays the
 * simulation constants and the following computed results:
 *
 * o  The number of customers served
 * o  The average time spent in line
 * o  The average number of people in line
 */
```

图 5-5 仿真结账队列程序

```
#include <iostream>
#include <iomanip>
#include "queue.h"
#include "random.h"
using namespace std;

/* Constants */

const double ARRIVAL_PROBABILITY = 0.05;
const int MIN_SERVICE_TIME =  5;
const int MAX_SERVICE_TIME = 15;
const int SIMULATION_TIME = 2000;

/* Function prototypes */

void runSimulation(int & nServed, int & totalWait, int & totalLength);
void printReport(int nServed, int totalWait, int totalLength);

/* Main program */

int main() {
   int nServed;
   int totalWait;
   int totalLength;
   runSimulation(nServed, totalWait, totalLength);
   printReport(nServed, totalWait, totalLength);
   return 0;
}

/*
 * Function: runSimulation
 * Usage: runSimulation();
 * -----------------------
 * Runs the actual simulation.  This function returns the results
 * of the simulation through the reference parameters, which record
 * the number of customers served, the total number of seconds that
 * customers were waiting in a queue, and the sum of the queue length
 * in each time step.
 */

void runSimulation(int & nServed, int & totalWait, int & totalLength) {
   Queue<int> queue;
   int timeRemaining = 0;
   nServed = 0;
   totalWait = 0;
   totalLength = 0;
   for (int t = 0; t < SIMULATION_TIME; t++) {
      if (randomChance(ARRIVAL_PROBABILITY)) {
         queue.enqueue(t);
      }
      if (timeRemaining > 0) {
         timeRemaining--;
      } else if (!queue.isEmpty()) {
         totalWait += t - queue.dequeue();
         nServed++;
         timeRemaining = randomInteger(MIN_SERVICE_TIME, MAX_SERVICE_TIME);
      }
      totalLength += queue.size();
   }
}

/*
 * Function: printReport
 * Usage: printReport(nServed, totalWait, totalLength);
 * ----------------------------------------------------
 * Reports the results of the simulation in tabular format.
 */

void printReport(int nServed, int totalWait, int totalLength) {
   cout << "Simulation results given the following constants:"
        << endl;
   cout << fixed << setprecision(2);
   cout << "  SIMULATION_TIME:      " << setw(4)
        << SIMULATION_TIME << endl;
   cout << "  ARRIVAL_PROBABILITY: " << setw(7)
        << ARRIVAL_PROBABILITY << endl;
   cout << "  MIN_SERVICE_TIME:     " << setw(4)
```

图 5-5 （续）

```
        << MIN_SERVICE_TIME << endl;
    cout << "  MAX_SERVICE_TIME:     " << setw(4)
        << MAX_SERVICE_TIME << endl;
    cout << endl;
    cout << "Customers served:       " << setw(4) << nServed << endl;
    cout << "Average waiting time:   " << setw(7)
        << double(totalWait) / nServed << " seconds" << endl;
    cout << "Average queue length:   " << setw(7)
        << double(totalLength) / SIMULATION_TIME << " people" << endl;
}
```

图 5-5 （续）

等待队列很自然地能被表示为一个队列。顾客到达队列的时间值被存储在队列中，它能用来确定顾客到达队首所花费的时间。

该仿真被以下常量所控制：

- SIMULATION_TIME：指定了仿真的持续时间。
- ARRIVAL_PROBABILITY：确定了在一个单位时间内一个新的顾客到达队列的概率。为了符合标准的统计学规定，这个概率被表示为一个在 0 到 1 之间的实数。
- MIN_SERVICE_TIME，MAX_SERVICE_TIME：这两个常量分别定义了顾客被服务时间的范围。对于任何一个顾客来说，收银员在他们身上所花费的时间的总和是在这个范围内的一个随机整数。

223 / 224

当一个仿真结束后，程序会报告该仿真的常量值以及下面的结果：

- 被服务的总顾客数。
- 顾客在队列中平均等待时间。
- 队列的平均长度。

例如，下面的运行示例展现了给定常量值的仿真运行之后的结果：

```
CheckoutLine
Simulation results given the following constants:
  SIMULATION_TIME:      2000
  ARRIVAL_PROBABILITY:   0.05
  MIN_SERVICE_TIME:         5
  MAX_SERVICE_TIME:        15

Customers served:         117
Average waiting time:     8.73 seconds
Average queue length:     0.51 people
```

仿真的行为很大程度上取决于用于控制其行为的变量的值。例如，假设单位时间内顾客到达队列的概率从 0.05 增加到 0.10，那么用这些参数运行这个仿真将得到下面的结果：

```
CheckoutLine
Simulation results given the following constants:
  SIMULATION_TIME:      2000
  ARRIVAL_PROBABILITY:   0.10
  MIN_SERVICE_TIME:         5
  MAX_SERVICE_TIME:        15

Customers served:         176
Average waiting time:    68.95 seconds
Average queue length:     6.49 people
```

正如你所看到的，顾客到达概率变为双倍后，造成了平均等待时间从低于九秒钟增加

到超过一分钟，显然这种增加是动态递增的。造成这种糟糕的性能的原因在于：仿真运行过程中每秒钟顾客到达的概率意味着新的到达的概率也就是他们服务的概率。当到达的概率增加时，队列的长度以及平均的等待时间开始快速地增加。这种仿真能够用不同参数值进行试
[225] 验。这些试验反过来能够在相对应的真实社会系统中验证潜在的资源问题。

5.4 Map 类

本节介绍另外一种名为 map 的集合类，它和字典从概念上很相似。字典允许你查阅一个单词来了解它的含义。一个 map 是这个概念的概括，它提供了一个被称为键（key）的标签和一个相关联的被称为值（value）的值之间的联系，这可能是一个更大更为复杂的结构。在字典例子中，键就是你所要查找的单词，值就是这个词的具体定义。

Map 在编程中有许多应用。例如，一种编程语言的解释器要能够给变量赋值，然后以该变量的名字作为引用。一个 Map 可以很简单地存储变量的名字和其所对应的值之间的联系。当它们在这种环境下使用时，Map 经常被称为**符号表**（symbol table）。

除了本节所描述的 Map 类之外，Stanford 类库中也提供了一种具有几乎相同结构和行为的名为 HashMap 的类。HashMap 类更加有效，但它在某些应用中不太便利。现在，你最好将注意力集中在 Map 类上，直到你大体上理解了 Map 类是如何工作的。在第 15 章和第 16 章中，你将有机会学习上述两种不同的 Map 类的实现方法。

5.4.1 Map 类的结构

和本章前面所介绍的集合类一样，Map 是由一个键类型以及值类型参数化的模板类实现的。例如，如果你想要仿真一个每个单词都和其定义相联系的字典，可以先声明一个名为 dictionary 的变量，如下所示：

```
Map<string,string> dictionary;
```

类似地，如果你正在实现某种编程语言，你可以使用 Map 通过将变量名与其值相关联的方式来存储一个浮点变量的值，如下所示：

```
Map<string,double> symbolTable;
```

上述声明语句创建了两个没有任何键和值的空的 Map 对象。对上述任意一个 Map 对象，你随后都可向其添加键－值对。在字典中，你可以从一个数据文件中读取内容。对于符
[226] 号表，一旦出现了一个赋值语句，你将会向符号表中添加一个新的变量与值的关联对。

表 5-5 给出了 Map 类最常见的一些方法。在这些方法中，实现 Map 概念的基本方法是 put 以及 get。put 方法在键和值之间创建了一个联系。这个操作和 C++ 中将一个值赋给一个变量类似：如果一个键已存在一个与之相对应的值，则这个旧的值将会被一个新的值所取代。get 方法检索最近和该键相关联的值，因此和使用变量名检索变量值相一致。在 Map 对象中，如果对于一个特定的键没有与其对应的值，则用这个键调用 get 方法会返回值类型的默认值。你可以通过调用 containsKey 方法来检查一个键是否出现在该 Map 对象中，返回的结果是 true 还是 false 取决于该键是否存在于 Map 对象中。

表 5-5 map.h 接口中的条目

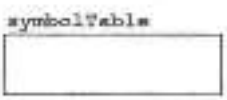
构造函数	
Map<*key type, value type*> ()	创建一个空的关联键和值的 Map 对象

（续）

方法	
`size()`	返回 Map 对象中含有的键和值进行关联值对个数
`isEmpty()`	如果 Map 对象是空的，返回 `true`
`put(key, value)`	将特定的键和值进行关联。如果 *key* 在先前的 Map 对象中没有定义，则一个新的值被添加进来；如果 Map 对象中已存在该键 *key*，则旧值将被新值所替代
`get(key)`	返回在 Map 对象中当前与键 *key* 相关联的值。如果该键没有定义，则 `get` 方法返回值类型的默认值
`remove(key)`	从 Map 对象中删除键 *key* 及其所对应的值。如果该键不存在，调用不会对 Map 对象作出任何改变
`containsKey(key)`	检查键 *key* 是否存在与之相关联的值。若存在，则返回 `true`；反之，则返回 `false`
`clear()`	删除 Map 对象中所有的键 - 值对
操作符	
map[*key*]	和 `get` 方法功能一样，该操作符在 Map 对象中返回与键 *key* 相关联的值。如果 Map 对象中不存在该键，则该选择操作符会创建一个新的键 - 值对，且设置其对应的值为值类型的默认值。使用方括号查找和改变某一特定键的值的 Map 对象通常被称为关联数组（associative array）

227

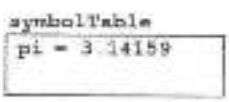
一些简单的图将会帮助你加深对 Map 类型操作的理解。假设你已经声明了一个类型为 Map<sting, double> 的变量 symbolTable，正如你在前面所看到的。该声明创建了一个空的 Map 对象，表示包含一个空的无键 - 值对的集合，如下图所示：



一旦你有了 Map 对象，你可以使用 put 方法来创建新的键与值之间的关联。例如，你可以调用以下语句：

```
symbolTable.put("pi", 3.14159);
```

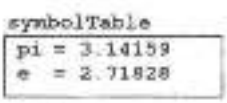
概念上的效果将会在键 "pi" 与值 3.141 59 之间添加一种关联，如下图所示：



类似地，调用以下语句：

```
symbolTable.put("e", 2.71828);
```

将会在键 "e" 与值 2.718 28 之间添加一个新的关联，如下图所示：

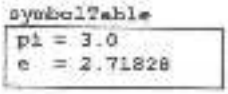


然后你可以使用 get 来检索这些值。调用 symbolTable.get（"e"）将得到值 2.718 28，调用 symbolTable.get（"pi"）将返回 3.141 59。

尽管就数学角度来说，这很难讲得通，但是你可以通过调用 put 方法来修改 Map 对象中的值。例如，你可以通过调用以下语句重置 "pi" 所对应的值（1897 年印第安纳州大会之前试图尝试做的事情）：

```
symbolTable.put("pi", 3.0);
```

将导致 Map 对象变成下面的这种状态：



在这种情况下，调用 symbolTable.containsKey("pi") 将会返回 true；与之相对应，
228 调用 symbolTable.containsKey("x") 将返回 false。

5.4.2 在一个应用中使用 Map 类

如果你经常乘坐飞机的话，你可以快速地发现世界各地的每一个机场都有一个由国际飞机运输协会（International Air Transport Association，IATA）颁布的三字母代码。例如，纽约市的约翰 F·肯尼迪机场的代码为 JFK。然而，其他的这些代码很难被分辨。大多数基于网络的交通运输系统提供了一些查询这些代码的方法，来作为对它们顾客的一种服务。

假设你被要求编写一个简单的 C++ 程序，用来从用户处读取一个机场的三字母代码，然后再向用户返回这个机场的位置。你需要的数据在一个名为 AirportCodes.txt 的文本文件中，这个文件包含了上千条由国际飞机运输协会已经颁布的飞机场的代码。文件中的每一行是由一个三字母代码、一个等号以及一个与之对应的机场位置所组成的。如果这个文件是按照 2009 年机场旅客流量的降序排列，并且由国际机场委员会编辑，那么这个文件将会如图 5-6 所示的一样。

AirportCodes.txt

```
ATL=Atlanta, GA, USA
ORD=Chicago, IL, USA
LHR=London, England, United Kingdom
HND=Tokyo, Japan
LAX=Los Angeles, CA, USA
CDG=Paris, France
DFW=Dallas/Ft Worth, TX, USA
FRA=Frankfurt, Germany
PEK=Beijing, China
MAD=Madrid, Spain
DEN=Denver, CO, USA
AMS=Amsterdam, Netherlands
JFK=New York, NY, USA
HKG=Hong Kong, Hong Kong
LAS=Las Vegas, NV, USA
IAH=Houston, TX, USA
PHX=Phoenix, AZ, USA
BKK=Bangkok, Thailand
SIN=Singapore, Singapore
MCO=Orlando, FL, USA
```

图 5-6 包含部分机场代码与位置的数据文件

229 Map 类的存在使这个应用很容易实现。图 5-7 展示了这个完整的应用程序。

```
/*
 * File: AirportCodes.cpp
 * ----------------------
 * This program looks up a three-letter airport code in a Map object.
 */

#include <iostream>
#include <fstream>
#include <string>
#include "error.h"
#include "map.h"
#include "strlib.h"
using namespace std;

/* Function prototypes */

void readCodeFile(string filename, Map<string,string> & map);

/* Main program */

int main() {
   Map<string,string> airportCodes;
   readCodeFile("AirportCodes.txt", airportCodes);
   while (true) {
      string line;
      cout << "Airport code: ";
      getline(cin, line);
      if (line == "") break;
      string code = toUpperCase(line);
      if (airportCodes.containsKey(code)) {
         cout << code << " is in " << airportCodes.get(code) << endl;
      } else {
         cout << "There is no such airport code" << endl;
      }
   }
   return 0;
}

void readCodeFile(string filename, Map<string,string> & map) {
   ifstream infile;
   infile.open(filename.c_str());
   if (infile.fail()) error("Can't read the data file");
   string line;
   while (getline(infile, line)) {
      if (line.length() < 4 || line[3] != '=') {
         error("Illegal data line: " + line);
      }
      string code = toUpperCase(line.substr(0, 3));
      map.put(code, line.substr(4));
   }
   infile.close();
}
```

图 5-7 查找三字母机场代码的程序 230

AirportCodes 应用中的主程序读取三字母代码，查找其相对应的机场位置，然后将位置在控制台中输出，正如下述示例程序的运行结果：



5.4.3 Map 类作为关联数组

Map 类重载了用于数组查找的方括号操作符，因此代码

```
map[key] = value;
```

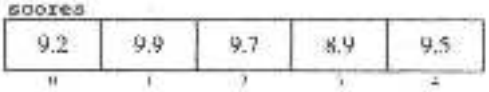
扮演了代码

```
map.put(key, value);
```

的简写形式。

类似地，与 map.get(key) 所做的一样，表达式 map[key] 返回了 Map 对象中与 key 相对应的值。毫无疑问，使用 put 以及 get 方法的简写方式非常方便，但是基于数组和 Map 类是两种完全不同的结构，在 Map 类中使用数组表示法可能令人惊讶。然而，如果你从更抽象的角度考虑 Map 类和数组，则和你刚开始的怀疑相比，你会发现它们是很相似的。

要统一这两种表面上看起来不一样的结构，你可以把数组看成是一种将位置索引和元素值进行映射的结构。例如，假设你有一个数组，或者一个等价的 Vector 对象，其中包含了一系列由你记录的体育比赛的分数：



这个数组将键 0 映射为值 9.2，将键 1 映射为值 9.9，将键 2 映射为值 9.7 等等。因此，你可以将一个数组看成是一个用整数作为键的 Map 对象。相反地，你也可以把一个 Map 对象看
231 成是一个使用键作为索引的数组，这也正是 Map 类取选择语法所提倡的。

使用数组语法来完成 Map 类的操作在编程语言中正逐步流行，它甚至已超出了 C++ 的领域。许多流行的脚本语言都用 Map 类来实现数组，这可使数组使用索引值不一定为整数。用 Map 类作为其底层实现者的数组被称为关联数组（associative array）。

5.5 Set 类

集合类中最有用的一个就是 Set 类，表 5-6 展示了其相关的条目。该类通常用于建模**集合**（set）的数学抽象，即它是一个集合，其中的元素是无序的且每个元素的值仅出现一次。Set 类在某些算法应用中极为有用，因此值得花费单独的一节来介绍它。在你阅读 17 章中更为详细的 Set 类的论述之前，举几个关于 Set 类的例子是非常值得的，这能让你对 Set 类是如何工作以及它在应用中为何非常有用有一个更好的理解。

表 5-6 set.h 接口中的条目

构造函数	
Set<*type*>()	创建一个包含特定类型值的空的 Set 对象
方法	
size()	返回 Set 对象中元素的个数
isEmpty()	如果 Set 对象为空则返回 true
add(*value*)	向 Set 对象中添加值 *value*。如果 Set 对象中已经存在该值，不产生任何错误，并且 Set 对象保持不变
remove(*value*)	从 Set 对象中删除该值 *value*。如果该值不存在，不产生任何错误，并且 Set 对象保持不变
contains(*value*)	如果 Set 对象中存在该值 value，则返回 true

（续）

clear()	删除 Set 对象中的所有元素
isSubsetOf(*set*)	如果 Set 对象是通过参数传递的 Set 对象的子集，则返回 true
first()	返回 Set 对象中由预类型确定顺序后的第一个元素

操作符

s_1 + s_2	返回 s_1 和 s_2 的并运算（union）结果。其中包含的元素是两个原始集合中的所有元素
s_1 * s_2	返回 s_1 和 s_2 的交运算（intersection）结果。其中包含的元素是两个原始集合中共同的元素
s_1 - s_2	返回 s_1 和 s_2 的差运算（difference）结果。其中包含的元素是只出现在 s_1 而未出现在 s_2 中的元素
s_1 += s_2，s_1 -= s_2，s_1 *= s_2	就像数值运算中的 +、-、* 一样，这些操作符可以和赋值表现在一起。对于 += 和 -=，s_2 的值可以是一个集合、单个值，或者是用逗号隔开的一系列值

5.5.1 实现 <cctype> 库

在第 3 章，你已经学习了 <cctype> 库，它导出了一些测试一个字符类型的判定函数。例如，调用 isdigit(ch) 将测试字符 ch 是否为一个数字字符。你可以通过测试 ch 是否超过了单个数字字符的值范围来实现函数 isdigit，如下所示：

```
bool isdigit(ch) {
   return ch >= '0' && ch <= '9';
}
```

对于其他一些函数，情况可能变得更为复杂一些。用同样的方式实现 ispunct 函数可能会有点困难，因为标点符号分布在 ASCII 范围的好几个间隔中。如果你将所有的标点符号定义在一个集合中，事情就会变得很简单，在这种情况下，实现 ispunct（ch）你所要做的就是检查字符 ch 是否出现在这个集合中。

图 5-8 展示了用集合实现 <cctype> 中的判定函数。这段代码首先对于每一种字符类型都创建了一个 Set<char>，然后定义了判定函数，这样它们仅仅在适当的 Set 对象中调用 contains 就可以实现判定函数。例如，为了实现 isdigit，cctype 的实现定义了一个包含所有数字字符的 Set 对象，如下所示：

```
const Set<char> DIGIT_SET = setFromString("0123456789");
```

出现在图 5-8 中的 setFromString 函数仅仅是一个辅助函数，它通过向 Set 对象中依次添加参数字符串中的字符来创造一个 Set 对象。这个函数让定义 Set 对象变得很简单，例如，对于定义标点符号字符的 Set 对象，你只需要列出适合其描述的字符即可。

对于那些抽象和高级的操作来说，使用 Set 类的好处之一就是让这些操作更易于思考。尽管在 cctype.cpp 中大多使用 setFromString 从实际字符中来创建 Set 对象，但是还有一些是使用 + 操作符，+ 操作符在 Set 类中被重载了，该操作可以返回两个 Set 对象的并运算结果。例如，一旦你定义了 Set 对象 LOWER_SET 和 UPPER_SET，使得它们可以包含小写字母和大写字母，你就可以通过编写如下代码来定义 ALPHA_SET：

```
const Set<char> ALPHA_SET = LOWER_SET + UPPER_SET;
```

232 | 233

```
/*
 * File: cctype.cpp
 * ----------------
 * This program simulates the <cctype> interface using sets of characters.
 */

#include <string>
#include "cctype.h"
#include "set.h"
using namespace std;

/* Function prototypes */

Set<char> setFromString(string str);

/*
 * Constant sets
 * -------------
 * These sets are initialized to contain the characters in the
 * corresponding character class.
 */

const Set<char> DIGIT_SET = setFromString("0123456789");
const Set<char> LOWER_SET = setFromString("abcdefghijklmnopqrstuvwxyz");
const Set<char> UPPER_SET = setFromString("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
const Set<char> PUNCT_SET = setFromString("!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~");
const Set<char> SPACE_SET = setFromString(" \t\v\f\n\r");
const Set<char> XDIGIT_SET = setFromString("0123456789ABCDEFabcdef");
const Set<char> ALPHA_SET = LOWER_SET + UPPER_SET;
const Set<char> ALNUM_SET = ALPHA_SET + DIGIT_SET;
const Set<char> PRINT_SET = ALNUM_SET + PUNCT_SET + SPACE_SET;

/* Exported functions */

bool isalnum(char ch) { return ALNUM_SET.contains(ch); }
bool isalpha(char ch) { return ALPHA_SET.contains(ch); }
bool isdigit(char ch) { return DIGIT_SET.contains(ch); }
bool islower(char ch) { return LOWER_SET.contains(ch); }
bool isprint(char ch) { return PRINT_SET.contains(ch); }
bool ispunct(char ch) { return PUNCT_SET.contains(ch); }
bool isspace(char ch) { return SPACE_SET.contains(ch); }
bool isupper(char ch) { return UPPER_SET.contains(ch); }
bool isxdigit(char ch) { return XDIGIT_SET.contains(ch); }

/* Helper function to create a set from a string of characters */

Set<char> setFromString(string str) {
   Set<char> set;
   for (int i = 0; i < str.length(); i++) {
      set.add(str[i]);
   }
   return set;
}
```

234

图 5-8 基于集合的 <cctype> 的实现

5.5.2 创建单词列表

在本章前面对 Map 类的讨论中，用于解释底层概念的示例之一就是：一个字典中的键都是一个个独立的单词，并且其对应的值就是该单词的定义。在某些应用中，例如一个拼写检查程序或者 Scrabble 程序（一种拼字游戏），你不需要知道一个词的定义，你所要知道的就是一个字母的组合是否是一个合法的单词。在那样的应用中，Set 类是一个理想的工具。与一个 Map 对象既包含单词又包含定义不一样，使用 Set 类你所需要的就是包含所有合法单词的 Set<string>。如果单词包含在 Set 对象中，那么它就是合法的，反之，则是非法的。

一个只有单词没有与之对应的解释的集合我们称之为字典（lexicon）。如果你有一个包含了所有英文单词的名为 EnglishWords.txt 的文本文件，并且每个单词只占一行，你

可以通过下面的这段代码创建一个英文字典：

```
Set<string> lexicon;
ifstream infile;
infile.open("EnglishWords.txt");
if (infile.fail()) error("Can't open EnglishWords.txt");
string word;
while (getline(infile, word)) {
   lexicon.add(word);
}
infile.close();
```

5.5.3 Stanford 类库中的 Lexicon 类

尽管对于一个字典来说，用 Set 类作为其底层表示表现得相当好，但它并不是很有效率。因为一个高效的字典表示法可能会给编程项目带来许多令人激动的事情。Stanford 类库中包含了一个名为 Lexicon 的类，该类是 Set 类中一个用于存储单词集合的优化的定制版本。表 5-7 展示了由 Lexicon 类导出的条目。正如你所看到的一样，它们和 Set 类中的大多数条目是一样的。

这个库中同时还包含了一个名为 EnglishWords.dat 的数据文件，这是一个已编译的包含了所有英文单词的字典。使用英文字典的程序通常都使用以下声明语句对其进行初始化：

```
Lexicon english("EnglishWords.dat");
```

在像 Scrabble 一样的文字游戏中，尽可能多地记住两个字母的单词是很有用的，因为知道两 235
个字母的单词能让你更容易地知道字典中以这两个字母为基础的新单词。假设你有一个包含英文单词的字典，你可通过生成所有的两个字母的字符串来创建这样一个列表，然后再用这个字典检查上述两个字母的字符串的组合是否为一个单词。上述代码的实现如图 5-9 所示。

表 5-7 lexicon.h 接口导出的条目

构造函数	
Lexicon()	创建一个空的 Lexicon 对象
Lexicon(*file*)	通过从文件 file 中读取数据来初始化一个 Lexicon 对象
方法	
size()	返回 Lexicon 对象中单词的总数
isEmpty()	如果 Lexicon 对象为空，返回 true
add(*word*)	如果该单词在 Lexicon 对象中不存在的话，则向其中添加一个新的单词 word。所有的单词都以小写字母形式存储在 Lexicon 对象中
addWordsFromFile(*file*)	将参数名为 file 文件中的所有单词添加到 lexicon 对象中。file 要么是一个其单词为分行存储的文本文件，要么是为 Lexicon 对象而特别设计其格式的已编译的数据文件向 Lexicon 对象中添加 file 中所有的单词
contains(*word*)	如果单词 word 存在于 Lexicon 对象中，则返回 true
containsPrefix(*prefix*)	如果 Lexicon 对象中的任何一个单词都是以特定的前缀 prefix 开头，则返回 true
clear()	删除 Lexicon 对象中的所有的单词

在下一节，你会发现：可以通过浏览字典，然后输出长度为 2 的单词的方法来解决这个问题。然而，基于在字典中有超过 100 000 个英文单词，而只有 676（26×26）个单词是由

两个字母组合而成，因此采用图 5-9 程序代码中所使用的策略将更加高效。

5.6 在集合上进行迭代

图 5-9 中介绍的 TwoLetterWords 程序通过生成两个字母的所有可能组合，然后再查阅字典，检查这些两个字母的组合是否出现在英文字典中的方式，产生了一个由两个字母组成的单词的清单。另一种达到同样效果的策略是浏览字典中的每一个单词，然后再将长度为 2 的单词显示出来。要做到这一点，你所要做的就是以某种方式每次一个单词地遍历
236 Lexicon 对象中的每个单词。

```
/*
 * File: TwoLetterWords.cpp
 * ------------------------
 * This program generates a list of the two-letter English words.
 */

#include <iostream>
#include "lexicon.h"
using namespace std;

int main() {
   Lexicon english("EnglishWords.dat");
   string word = "xx";
   for (char c0 = 'a'; c0 <= 'z'; c0++) {
      word[0] = c0;
      for (char c1 = 'a'; c1 <= 'z'; c1++) {
         word[1] = c1;
         if (english.contains(word)) {
            cout << word << endl;
         }
      }
   }
   return 0;
}
```

图 5-9 生成所有由两个字母构成的单词的程序

对于任何集合类而言，迭代其中的元素是一个基本的操作。此外，如果集合类包设计得很好，用户应该能够使用同样的策略来实现这些操作，无论是一个 Vector 对象还是 Grid 对象来循环地遍历其中的每个元素，或者是遍历一个 Map 对象中的所有键，或者是遍历一个 Lexicon 对象中的所有单词。标准模板库提供了一个强有力的名为**迭代器**（iterator）的机制来做上述事情。遗憾的是，理解标准迭代器需要熟悉 C++ 某些特定的底层细节，尤其是指针的概念。鉴于本节的一个主要目标就是推迟掩盖这些细节，直到你了解了高层的思想，对于实现一个实际上很简单的例子，引入标准迭代器会牵扯许多复杂的事物。你所要做的就是表达下面伪代码所建议的算法思想：

```
For each element in a particular collection {
   Process that element
}
```

大多数现代编程语言为了准确地表达算法思想而定义了一种对应的语法形式。但遗憾的
237 是，尽管它已被提议将在未来的某个版本中发布，但 C++ 语法中至今仍不包含这样的机制。然而，好消息是可以使用 C++ 预处理的宏定义功能来准确地实现你想要出现在程序中的东西。尽管这个实现超过了本章集合类的范围，但是这些集合类——包括标准模板库里的集合类和本章的简化版本——支持一种新的被称为**基于范围的循环**（range-based for loop）的控制模式，如下所示：

```
for (type variable : collection) {
   body of the loop
}
```

例如，如果你想要迭代英语字典中所有的单词，并且挑选出只含有两个字母的单词，你可以这样编写代码，如下所示：

```
for (string word : english) {
   if (word.length() == 2) {
      cout << word << endl;
   }
}
```

基于范围的循环是C++11的新特性，C++11是2011年发布的C++的新版本。因为这个版本是最近发布的，因此，C++11所扩展的特性还没有被完全的合并到所有C++编程环境中，包括了几个主要的特性。如果你使用的是一个旧的编译器，你将不能够使用基于范围的循环的标准形式。但是也不必绝望，标准C++库中包含了一个名为foreach.h的接口，它使用了C++预处理，用一个很熟悉的方式定义了一个名为foreach的宏：

```
foreach (type variable in collection) {
   body of the loop
}
```

foreach宏与基于范围的循环的唯一不同在于关键字的名字。foreach中使用关键词in而不是一个冒号。和基于范围的循环一样，foreach对于Stanford类库和标准模板库中实现的集合类都起作用。

5.6.1 迭代顺序

当你使用基于范围的循环时，有时候理解迭代处理元素的顺序是很有用的。这没有什么通用的规则。对于迭代顺序，基于效率上的考虑，每个集合类都定义了关于其自身的迭代顺序策略。你之前见过的类，对关于元素值的顺序都进行了下面的保证：

- 当你对一个Vector类中的元素进行遍历时，基于范围的循环以索引位置为序来对元素进行遍历，因此，索引位置为0的元素首先被遍历，紧接着是索引位置为1的元素，直到这个Vector类中的最后一个元素被遍历。因此，迭代的顺序与传统的 238
for循环模式顺序是一样的：

```
for (int i = 0; i < vec.size(); i++) {
   code to process vec[i]
}
```

- 当你迭代一个Grid类中的元素时，基于范围的循环首先依次浏览行数为0的各元素，然后再浏览行数为1的各元素，依此类推。Grid类的迭代策略和下面使用的for循环相类似：

```
for (int row = 0; row < grid.numRows(); row++) {
   for (int col = 0; col < grid.numCols(); col++) {
      code to process grid[row][col]
   }
}
```

这种外循环出现行下标，它的遍历顺序被称为**行优先次序**（row-major order）。

- 当你对 Map 类中的元素进行遍历时，基于范围的循环返回以键为序的且由其类型决定的所有键值。例如，一个键类型为整型的 Map 对象将会按照数字升序的顺序排列其键值。一个键类型为字符串类型的 Map 对象将会按照字典序（lexicographic order）排列其键值，字典序是通过比较其内部的 ASCII 码值来决定其顺序的。
- 当你对一个 Set 类或者 Lexicon 类中的元素进行遍历时，基于范围的循环返回的元素的顺序总是由其值的类型所确定的。在 Lexicon 类中，基于范围的循环返回小写字母的所有单词。
- 你不能使用基于范围的循环去遍历 Stack 类和 Queue 类。当只有一个元素（这个元素是栈顶元素或者是队首元素）是可见的时候，允许自由地访问这些结构将会违背栈和队列的读取原则。

5.6.2 再论儿童黑话

像 3.2 节所描述的一样，当你将英语转化成儿童黑话时，大多数单词将转变成某种与传统的英语截然不同的、听起来模糊的类拉丁文语言。然而，有几个单词在它们转化后恰好与英文单词相同。例如，trash 的儿童黑话是 ashtray，entry 的儿童黑话是 entryway。但这些单词并不是一种普遍情况，存储在 EnglishWords.dat 文件中的字典，有超过 100 000 个英语单词，但仅有 27 个单词符合上述情况。采用基于范围的循环和第 3 章中 PigLatin 程序的 translateWord 函数，可以容易地编写出图 5-10 中列出所有这些单词的程序。

239

```
/*
 * File: PigEnglish.cpp
 * --------------------
 * This program finds all English words that remain words when
 * you convert them to Pig Latin, such as 'trash' (which becomes
 * "ashtray") and 'entry' (which becomes 'entryway').  The code
 * ignores words containing no vowels (mostly Welsh-derived
 * words like 'cwm'), which don't change form under the Pig Latin
 * rules introduced in Chapter 3.
 */

#include <iostream>
#include <string>
#include <cctype>
#include "lexicon.h"
using namespace std;

/* Function prototypes */

string wordToPigLatin(string word);
int findFirstVowel(string word);
bool isVowel(char ch);

/* Main program */

int main() {
   cout << "This program finds words that remain words"
        << " when translated to Pig Latin." << endl;
   Lexicon english("EnglishWords.dat");
   for (string word : english) {
      string pig = wordToPigLatin(word);
      if (pig != word && english.contains(pig)) {
         cout << word << " -> " << pig << endl;
      }
   }
   return 0;
}

/* The code for the helper functions appears in Figure 3-2 */
```

例 5-10 列出所有的英文单词与 Pig Latin 单词一致的单词程序

5.6.3 计算单词的频率

图 5-11 中的 WordFrequency 程序是另一个迭代起重要作用的应用程序。采用之前示例中你已经用过的机制，则必要的程序代码将相当简单。将一行划分成一个个单词的实现策略与你在第 3 章中已经见过的 PigLatin 程序很类似。为了记录每个单词以及它出现的频率值，你明显需要一个 Map<string, int> 对象。 240

```
/*
 * File: WordFrequency.cpp
 * -----------------------
 * This program computes the frequency of words in a text file.
 */

#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <cctype>
#include "filelib.h"
#include "map.h"
#include "strlib.h"
#include "vector.h"
using namespace std;

/* Function prototypes */

void countWords(istream & stream, Map<string,int> & wordCounts);
void displayWordCounts(Map<string,int> & wordCounts);
void extractWords(string line, Vector<string> & words);

/* Main program */

int main() {
   ifstream infile;
   Map<string,int> wordCounts;
   promptUserForFile(infile, "Input file: ");
   countWords(infile, wordCounts);
   infile.close();
   displayWordCounts(wordCounts);
   return 0;
}

/*
 * Function: countWords
 * Usage: countWords(stream, wordCounts);
 * --------------------------------------
 * Counts words in the input stream, storing the results in wordCounts.
 */

void countWords(istream & stream, Map<string,int> & wordCounts) {
   Vector<string> lines, words;
   readEntireFile(stream, lines);
   for (string line : lines) {
      extractWords(line, words);
      for (string word : words) {
         wordCounts[toLowerCase(word)]++;
      }
   }
}

/*
 * Function: displayWordCounts
 * Usage: displayWordCounts(wordCount);
 * ------------------------------------
 * Displays the count associated with each word in the wordCount map.
 */

void displayWordCounts(Map<string,int> & wordCounts) {
   for (string word : wordCounts) {
      cout << left << setw(15) << word
           << right << setw(5) << wordCounts[word] << endl;
   }
```

图 5-11 计算单词频率的程序

```
}

/*
 * Function: extractWords
 * Usage: extractWords(line, words);
 * ---------------------------------
 * Extracts words from the line into the string vector words.
 */

void extractWords(string line, Vector<string> & words) {
   words.clear();
   int start = -1;
   for (int i = 0; i < line.length(); i++) {
      if (isalpha(line[i])) {
         if (start == -1) start = i;
      } else {
         if (start >= 0) {
            words.add(line.substr(start, i - start));
            start = -1;
         }
      }
   }
   if (start >= 0) words.add(line.substr(start));
}
```

图 5-11 （续）

对于那些使用现代化工具的应用来说，计算单词频率变得很有用起初是令人惊讶的。例如，过去的几十年中，在解决那些有争议的著作的问题上，计算机分析已经变得非常重要。这里有几个伊丽莎白一世时代的戏剧，虽然它们并不是莎士比亚传统的著作集中的一部分，但这些戏剧仍被认为可能是由莎士比亚书写的。相反地，有一些被归于莎士比亚著作集的戏剧听起来并不像他的其他作品，这些作品实际上可能是由其他人写的。为了解决这样的问题，研究莎士比亚的学者准备计算出现在这些戏剧中的特定单词的频率，然后看这些频率与根据莎士比亚已知的作品中分析得到的单词频率是否匹配。

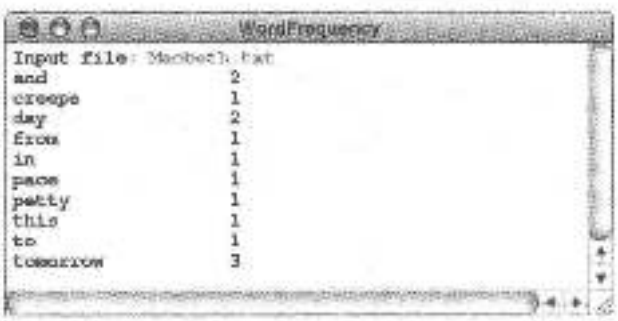
241
:
242

例如，假设你有一个包含了莎士比亚的一段文章的文本文件，例如以下所示的《麦克白》中著名的句子：

Macbeth.txt

```
Tomorrow, and tomorrow, and tomorrow
Creeps in this petty pace from day to day
```

如果你试图确定莎士比亚作品中单词的相关频率，你可以使用 WordFrequency 程序计算数据文件中每个单词出现的次数。因此，给定文件 Macbeth.txt，你的程序可能如下所示：



本章小结

本章介绍了C++的Vector、Stack、Queue、Map和Set类，它们一起代表了存储集合的一个强大的框架。目前，你只需要从用户的角度来看待这些类。在后续的章节中，你将有机会更深入地学习它们是如何实现的。鉴于当你完成本书的学习后，可能会实现这些集合类，虽然它们也提供了一些非常类似的方法集合，但这里介绍的类都是标准模板库中vector、stack、queue、map和set类的某种程度上的简化。

本章重点包括：

- 根据其行为而不是其表示进行定义的数据结构被称为**抽象数据类型**（abstract data type）。与基本数据类型相比，抽象数据类型有一些重要的优点。这些优点包括简单性、灵活性和安全性。
- 包含其他对象并作为一个完整集合的元素类被称为**集合类**（collection classe）。在C++中，集合类的定义使用了**模板**（template），即**参数化类型**（parameterized type），其中元素的类型名出现在集合类名字之后的尖括号中。例如，类Vector<int>表示一个包含元素值类型为int的Vector类。 243
- 矢量Vector类是一种抽象数据类型，它的行为与一个一维数组很像，但更加强大。和数组不一样的是，一个Vector对象可以随着元素的增加和减少其尺寸可动态地变化。Vector类也更加安全，因为它检查确保所有的索引都在其范围中。尽管你可以使用Vector对象中包含多个Vector对象来创建一个二维结构，但是使用Stanford类库中的Grid类将更加简单。
- 栈Stack类表示了一种对象的集合，这些对象的行为表现为从一个栈中删除元素的方向与向栈中添加元素的方向相反：即后进先出（LIFO）。Stack类的基本操作是push，也就是向栈中添加一个元素，另一个基本操作是pop，即删除并返回最近添加的元素值。
- 队列Queue类和栈Stack类相似，但有一点不同。从一个队列中删除元素和添加元素的顺序相同：即先进先出（FIFO）。一个队列的基本操作是enqueue，也就是在队列的末尾添加一个元素，另一个基本操作是dequeue，即从队列的开始删除一个元素并且返回该元素值。
- Map类在某种程度上实现了键（key）与值（value）的关联，以便能够高效地检索这些关联。一个Map对象的基本操作是put，也就是向Map对象中添加一个键－值对，另一个基本操作是get，即返回一个特定的键所关联的值。
- Set类表示一个集合，和数学中的集合一样，这个集合中的元素是无序的并且每个元素只能出现一次。一个Set对象的基本操作包含了add，也就是向Set对象中添加一个新的元素，同时也包含了contains，即检查一个元素是否已存在于Set对象中。
- 除了Stack和Queue类，所有的集合类都支持foreach模式，它使循环遍历集合中的元素变得很简单。和在“迭代顺序”这一节中所描述的一样，每一个集合类都定义了自己的元素迭代顺序。
- 另外，对于Map类和Set类，Stanford类库都提供了非常相关联的HashMap和HashSet类。Map类与HashMap类的唯一不同（或者Set与HashSet类之间）

在于其基于范围的循环的迭代元素顺序不同。Map 和 Set 以其元素类型值的升序来
244 迭代元素，而 HashMap 和 HashSet 类更高效，它们似乎以随机顺序来迭代元素。

复习题

1. 判断题：一个抽象的数据类型是以其行为定义而不是根据其表示来定义的。
2. 本章中列举的将一个类的行为与其基本实现相分离的三个好处是什么？
3. 什么是标准模板库（STL）？
4. 如果你想在一个程序中使用 Vector 类，你需要在你程序的开始加上什么样的 #include？
5. 列出至少三个与 C++ 中提供的基本的数组机制相比，Vector 类更具有的优势。
6. 边界检查这个术语的含义是什么？
7. 什么是一个参数化类型？
8. 你将使用什么样的类型名来存储一个元素类型为布尔类型的 Vector？
9. 判断题：Vector 类的默认构造函数创建了一个含有 10 个元素的 Vector 对象，不过你可以在之后将其变大。
10. 如何初始化一个含有 20 个元素，并且其元素值都等于 0 的 Vector<int> 对象？
11. 你可以调用什么样的方法来确定一个 Vector 对象中的元素个数？
12. 如果一个 Vector 对象含有 N 的元素，insert 方法的第一个参数的合法范围是什么？remove 方法的参数的合法范围是什么？
13. 让 Vector 类能够避免明确地使用 get 和 set 方法的特点是什么？
14. 为什么通过传递引用的方式传递 Vector 对象和其他集合对象是很重要的？
15. 你会使用什么样的声明去创建一个名为 chessboard，并且元素类型都为字符的 8×8Grid 对象？
16. 对于上个习题中给出的 chessboard 变量，如何给最左边以及最右边的角落赋值字符的 'R'（在国际象棋的表示法中，这代表了白色的车）。
245 17. 首字母缩写词 LIFO 和 FIFO 代表了什么？这些术语是怎么适用在栈和队列中的？
18. 对于一个栈来说，两个基本操作的名称是什么？
19. 队列的基本操作是什么？
20. 在 Stack 类和 Queue 类中，peek 操作都做了什么？
21. 用你自己的语言表述在仿真程序中出现的离散时间的意义？
22. Map 类中使用的两种类型参数是什么？
23. 如果在一个 Map 对象中，采用 get 调用一个不存在于该对象中的键将会发生什么？
24. 你把一个 Map 对象看成是一个关联数组，那么 get 和 set 方法的语法缩写形式是什么？
25. 为什么 Stanford 类库中包含了一个单独的 Lexicon 类，即使这个类可使用 Set 类来简单的实现？
26. 哪两种类型的数据文件支持 Lexicon 类的构造函数？
27. 基于范围的循环模式的一般形式是什么？
28. 本章中对于 Stack 类和 Queue 类不能使用基于范围的循环的原因是什么？
29. 对于本章介绍的每一种集合类，描述用基于范围的循环处理其元素时的顺序。

习题

1. 编写以下重载函数：

```
void readVector(istream & is, Vector<int> & vec);
void readVector(istream & is, Vector<double> & vec);
void readVector(istream & is, Vector<string> & vec);
```

这些函数都是从指定的输入流 is 向 Vector 对象 vec 中输入数据。在输入流中，每一个

Vector 对象的元素出现在其自身的一行中。函数一直读取元素直到遇到了空行或者文件的结尾。为了解释这个函数的操作，假设你有一个数据文件：

SquareAndCubeRoots.txt

```
1.0000
1.4142
1.7321
2.0000

1.0000
1.2599
1.4422
1.5874
1.7100
1.8171
1.9129
2.0000
```

并且你已经对这个文件创建了一个名为 infile 的输入流。此外，假设你已经声明了一个名 roots 的变量，如下所示：

```
Vector<double> roots;
```

第一次调用 readVector（infile, roots）将 roots 初始化为一个包含数据文件前四个元素的对象。第二次调用将 roots 的值修改为在数据文件末尾出现的八个元素。第三次调用将 roots 变为一个空的 Vector 对象。

2. 在统计学里，一个数据值的集合经常被称为一个分布（distribution）。统计分析的一个主要目标是找到一些方法来把完整的数据集合压缩成为一个简明扼要的统计资料，进而从整体上表达其分布特性。最常见的统计方法就是平均值（mean），也就是传统的平均。对于分布 x_1、x_2、x_3、…、x_n，平均值通常用符号 $\bar{x}$ 表示。试编写以下函数：

```
double mean(Vector<double> & data);
```

返回 Vector 对象中数据的平均值。

3. 另一种常见的统计方法就是标准差（standard deviation），它表明了分布 x_1、x_2、x_3、…、x_n 中的值偏离平均数的多少。在数学形式中，标准差（σ）被表达成如下形式，对照此例，如果你正在计算一个完整的分布的标准差：

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x - x_i)^2}{n}}$$

希腊字母 Σ 表示的是每一个单独的数据值与平均值差的平方和。编写一个函数：

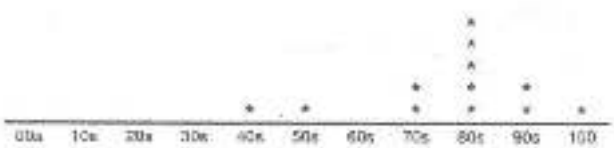
```
double stddev(Vector<double> & data);
```

返回数据分布的标准差。

4. 直方图是一个图，它通过把数据划分进单独的范围，然后指出每个范围内中有多少个数据值落入该范围的方式来显示一组值。例如，给出一组考试分数：

100，95，47，88，86，92，75，89，81，70，55，80

一个传统的直方图有如下形式：



直方图中的星号表示：有一个分数在 40 ～ 49 之间，有一个分数在 50 ～ 59 之间，有五个分数在 80 ～ 89 之间，依此类推。

然而，当你用计算机生成直方图的时候，将其放在一页的一边是很简单的，如以下这个示例运行结果：



编写一个程序，这个程序从一个数据文件中向一个元素类型为整型的 Vector 对象中输入数据，然后在一个直方图中显示这些数字，这个直方图被划分的范围依次是 0 ～ 9、10 ～ 19、20 ～ 29，
248 依此类推，直到只包含值 100 的范围。你的程序应该尽可能产生和示例程序一样的输出。

5. 通过定义一个名为 hist.h 的接口来扩展之前习题的灵活性，这个接口给出了用户对于直方图的形式的更多的控制。你的接口的最低要求是：应该允许用户指定最大值和最小值，以及每个直方图范围的大小，但你也可以额外的添加一些其他的功能。

6. 公元前 3 世纪，古希腊天文学家埃拉托色尼发明了一种用于发现不超过 N 的所有素数的算法。为了应用这种算法，你首先写出一组在 2 到 N 之间的整数。例如，如果 N 是 20，你可以写出如下所示的一组数字：

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

然后，你给列表中的第一个数字划上一个圆圈，表明你已经找出了一个素数。当你给一个数标记为素数时，你检查列表中剩余的数字并且划掉这个素数的倍数，因为这些倍数都不可能为素数。因此，当你执行完这个算法的第一轮时，你将会圈出数字 2，并且划掉了数字 2 的倍数，如下所示：

② 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~

为了完成这个算法，你只需要重复一个过程，这个过程首先圈出第一个既没有被圈出没有被划掉的数字，然后再划掉这个数字的倍数。在这个例子中，你将会圈出 3 作为一个素数，然后在剩余列表中划掉 3 的倍数，这将会导致列表变成下面的状态：

② ③ ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~

最终，列表中既没有被圈出没有被划掉的数字如下图所示：

② ③ ~~4~~ ⑤ ~~6~~ ⑦ ~~8~~ ~~9~~ ~~10~~ ⑪ ~~12~~ ⑬ ~~14~~ ~~15~~ ~~16~~ ⑰ ~~18~~ ⑲ ~~20~~

这些被圈出的数字都是素数，被划掉的数字都是合数。这个算法被称为埃拉托色尼筛法（sieve of Eratosthenes）。

编写一个程序，使用爱拉托色尼筛法产生一个 2 到 1000 之间素数的列表。

249 7. 与使用 Vector 类不同，使用 Grid 类的一个问题就是很难创建一个带有特殊初始值的集合。这就需要你使用操作符 += 来添加你想要加入的元素。一种使这个过程合理化的方法就是定义一个函数来初始化一个 Grid 对象：

```
void fillGrid(Grid<int> & grid, Vector<int> & values);
```

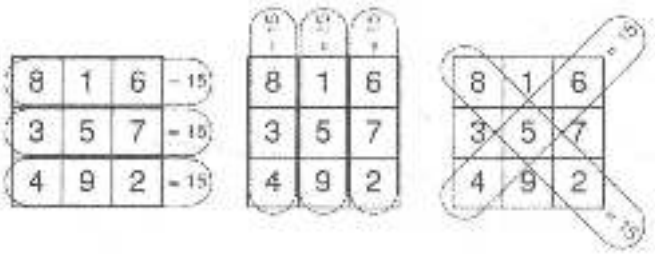
该函数从 Vector 对象中取值，然后向 Grid 对象中添加元素。例如，以下代码：

```
Grid<int> matrix(3, 3);
Vector<int> values;
values += 1, 2, 3;
values += 4, 5, 6;
values += 7, 8, 9;
fillGrid(matrix, values);
```

初始化变量 matrix，则为一个包含如下图所示值的 3×3 的 Grid 对象：

1	2	3
4	5	6
7	8	9

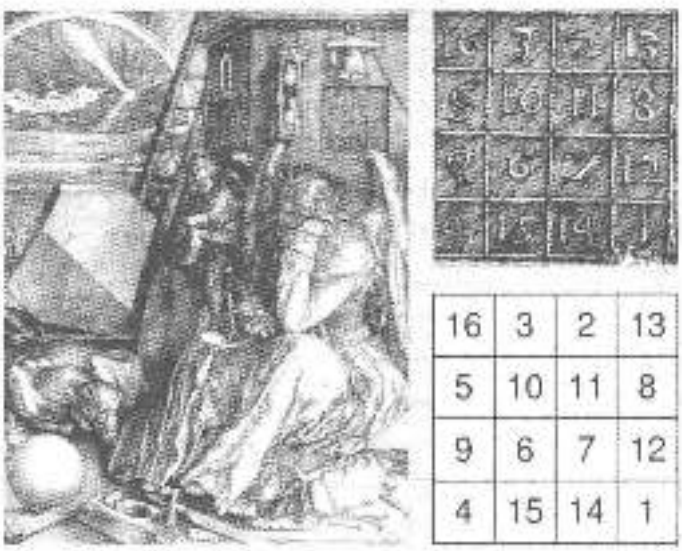
8. 魔方是一维的等距 Grid 对象，并且这个对象的行、列以及对角线的元素加起来都等于相同值。图 5-12 展现的是一个著名的魔方，它出现在 1514 年由阿尔布雷特·丢勒（Albrecht Dürer）雕刻的 Melencolia I 上。在这个雕像的右上角时钟的下面有一个 4×4 的魔方。在丢勒的魔方中，可以很容易地看到在图右边放大的插图中，每一行、每一列和两条对角线上的元素相加之和都等于 34。一个更加熟悉的例子就是下面 3×3 的魔方，其中每一行、每一列和对角线上的元素相加之和都等于 15，如下图所示：



实现一个函数：

```
bool isMagicSquare(Grid<int> & square);
```

250



16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

图 5-12 阿尔布雷特·丢勒在 Melencolia I 上的魔方

请测试一个 Grid 对象包含的是否是一个魔方。你的程序应该可以在任何大小的 Grid 对象中运行。如果你对一个行数和列数不相同的 Grid 对象调用 isMagicSquare，函数应该只返回 false。

9. 在过去的几年里，一个名为数独（Sudoku）的新的逻辑问题在全世界广泛地流行。在数独中，开始的时候有一个元素类型为整型的 9×9 的 Grid 对象，并且其中的一些单元格已经填上了 1 到 9 之间的整数。在这个问题中，你的工作是在空白的单元格中填上 1 到 9 之间的整数，使得每个整数在每一行、每一列和每一个小的 3×3 的正方形中只出现一次。每一个数独问题都要很小心的创建以便它
[251] 们只有一个答案。例如，图 5-13 左边展现了一个典型的数独问题，右边是这个问题的唯一答案。

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

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

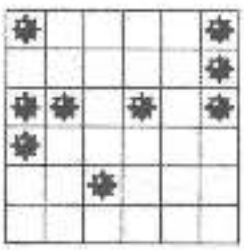
图 5-13 典型的数独问题和它的解

尽管在第 9 章之前你不需要发现解决数独问题的算法策略，但是你可以编写一个方法检查一个被提议的答案是否遵循了每一行、每一列以及 3×3 正方形中不能有重复的数独规则。编写一个函数：

```
bool checkSudokuSolution(Grid<int> & puzzle);
```

执行这个检查，并且当 puzzle 是有效的答案时，返回 true。程序应该检查以确保 puzzle 包含了一个元素类型为整型的 9×9 的 Grid 对象，并且当不满足这个情况时报告错误。

10. 在扫雷游戏中，对于一个很小的网格，一个玩家在一个矩形的网格中寻找隐藏的雷可能像下面这样：



在 C++ 中，一种代表这种网格的方法是使用元素类型为布尔类型的 Grid 对象，并且 Grid 对象中元素的值标示了雷的位置。即元素值为 true 表示这个位置有一个雷，采用布尔形式。这个例
[252] 子的 Grid 对象应该如下图所示：

T	F	F	F	F	T
F	F	F	F	F	T
T	T	F	T	F	T
T	F	F	F	F	F
F	F	T	F	F	F
F	F	F	F	F	F

给出一个表示雷位置的 Grid 对象，编写一个函数：

```
void fixCounts(Grid<bool> & mines, Grid<int> & counts);
```

使用引用参数 counts 来返回一个元素类型为整型的 Grid 对象，在这个对象中，每一个元素表明了在 mines 这个 Grid 对象中的每个位置相应附近的雷的数目，一个位置的附近包括了它自身所在的位置以及任何八个在 Grid 对象边界内的相邻位置。例如，如果 mineLocations，包含了这一页开始时所展示的元素类型为布尔的 Grid 对象，以下代码：

```
Grid<int> mineCounts;
fixCounts(mineLocations, mineCounts);
```

将 mineCounts 初始化为如下形式：

1	1	0	0	2	2
3	3	2	1	4	3
3	3	2	1	3	2
3	4	3	2	2	1
1	2	1	1	0	0
0	1	1	1	0	0

253

11. Grid 类中的 resize 方法改变了一个 Grid 对象的维数，同时也将该 Gird 对象的每一个元素初始化为默认值。编写一个函数：

```
void reshape(Grid<int> & grid, int nRows, int nCols);
```

调整 Grid 对象的大小，但填写的数据须通过拷贝的方式按标准的行主序（从左到右/从上到下）方式进行。例如，如果 myGrid 初始时包含值，如下图所示：

1	2	3	4
5	6	7	8
9	10	11	12

调用函数

```
reshape(myGrid, 4, 3)
```

将 myGrid 的维数及内容修改成如下图所示：

1	2	3
4	5	6
7	8	9
10	11	12

如果新的 Grid 对象没有足够的空间用于存储原始的值，原 Grid 对象底下的值会被舍弃掉，例如，如果你调用以下函数：

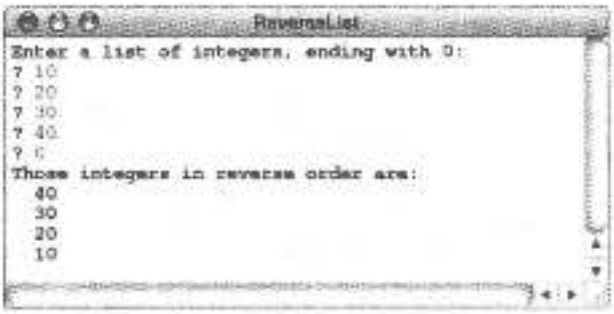
```
reshape(myGrid, 2, 5)
```

这里没有足够的空间来容纳最后两个元素，所以新的 Grid 对象如下图所示：

1	2	3	4	5
6	7	8	9	10

相反地，如果原来的 Grid 对象中没有足够的元素来填充其空间，新的 Grid 对象中最后的元素保持它们的默认值。

12. 编写一个程序，使用栈从控制台中逆序读取一个整数序列。其中，控制台的每一行只包含一个整数，示例程序的运行结果如下图所示：



254

13. 现在是第一个
将变成最后一个
就对同写言，它们是一个变革的时代。

——鲍勃·迪伦，"变革的时代（*The Times They Are a-Changin*）"，1963

从鲍勃·迪伦的歌曲中获得灵感，编写一个函数：

```
void reverseQueue(Queue<string> & queue);
```

颠倒一个队列中的元素。记住你没有接触过队列的内部表示方法，因此，你必须想出一个算法（假定涉及其他的结构）用于完成这个任务。

14. 编写一个程序，检查一个字符串中的括号（圆括号、方括号和花括号）是否匹配。考虑以下字符串：

```
{ s = 2 * (a[2] + 3); x = (1 + {2}); }
```

如果你仔细检查这个字符串，会发现所有的括号都是正确嵌套的，即每一个左圆括号和一个右圆括号相匹配，每一个左方括号和一个右方括号相匹配，依此类推。另一方面，下面的字符串都是不匹配的，并且已经给出原因：

{(])}	这行缺少了一个右括号
)(	右括号在左括号之前出现
[{] }	括号操作符不恰当地嵌套

15. 本书中的图表都是使用由 Adobe 公司在 20 世纪 80 年代早期开发的强大的图形语言 PostScript® 创建的。PostScript 程序将其数据存在一个栈中。PostScript 中可使用的操作符在某种程度上对栈的操作都有效果。例如，你可以调用 pop 操作符，这将会弹出栈顶元素，或者调用 exch 操作符，这将会交换栈中的两个元素。

PostScript 操作符中一个最有趣（并且是最有用）的操作符是 roll 操作符，它需要两个参数：n 和 k。调用 roll（n, k）的效果是将一个栈顶的 n 个元素转动 k 个位置，旋转的方向一般朝着栈顶。更加特殊地是，roll（n, k）的效果是移动栈顶的 n 个元素，将上面的元素循环至最底下的位置 k 次，然后在栈中更换重新排序的元素。图 5-14 展示了三个调用 roll 函数之前和之后的不同例子的图片。

255

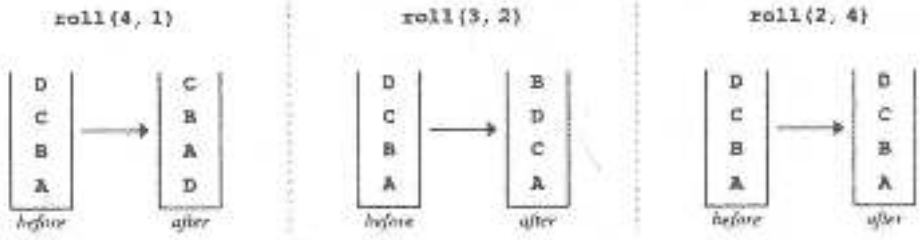


图 5-14 栈 roll 函数的调用示例

编写一个函数，在一个指定的栈中实现 roll(n,k)：

```
void roll(Stack<char> & stack, int n, int k)
```

你的实现应该检查 n 和 k 都是非负的，并且 n 不大于栈的大小。如果违反了其中的任何一个条件，你的实现应该调用 error 并提供以下消息：

```
roll: argument out of range
```

然而，k 可以比 n 大。在这种情况下，roll 操作将会持续经过一个完整的循环。图 5-14 最后举例说明了这种情况，栈顶的两个元素滚成了四次，导致栈中的内容和开始时完全一样。

16. 你可以扩大到 5-5 中所示的结账队列仿真来研究队列是如何排队的这个重要的实际问题。首先，和超市中经常出现的情况一样，对于有多个独立的队列重写仿真。一个顾客到达结账台的区域时，会寻找最短的结账队列，然后进入这个队列。修改后的仿真应该和本章中的仿真报告同样的结果。
17. 作为结账队列仿真的第二个扩展，修改上述习题中的程序使得只有单个队列并且被多个收银员服务（近年来一种很常见的形式）。在仿真的每个循环中，任何收银员只要空闲了，就会为队列中下一个顾客服务。如果你比较这个练习题与之前习题产生的数据，对于组织一个结账队列的这两种方式它们各自具有的优势，你有什么想说的？ 256
18. 编写一个程序，对下面的试验进行仿真。这个试验出现在 1957 年的迪斯尼电影《点了我们的朋友》（*Our Friend the Atom*）中，用来说明核裂变中涉及的链式反应。试验的环境是一个立方形的盒子，这个盒子的底部被 625 个陷阱完全覆盖，并且 625 个陷阱形成了一个每边有 25 个陷阱的正方形。每个陷阱开始的时候填充了两个乒乓球。在仿真开始时，一个额外的乒乓球从盒子顶部释放，然后落到一个陷阱上。这个陷阱被触发，然后使其自己的两个乒乓球弹出空气中。这两个乒乓球在盒子的侧面跳跃，最终落在盒底。在落下的位置上，它们可能触发更多的陷阱。

    在编写这个仿真的过程中，你应该做出以下这些简化的假设：

    - 每一个乒乓球都落在一个陷阱上，这个陷阱是通过在网格中挑选一个随机的行和列来随机挑选的。如果这个陷阱依旧填充了乒乓球，那么会将其乒乓球释放到空中。如果这个陷阱已经被触发，落入一个球对其没有影响。
    - 一旦一个球落入一个陷阱中，无论这个陷阱是否已经被触发，这个球都将停止，并且不会对接下来的仿真产生影响。
    - 从陷阱中发射的乒乓球在盒子的空间中跳跃，直到经过一个随机的时间间隔后才会落下。这个随机的间隔对于每一个乒乓球都是独立的，并且这个时间间隔总是处于一个到四个循环之间。

    你的仿真应该一直运行，直到空中没有乒乓球。在那个时候，程序应该报告从程序开始运行到现在一共经过了多长时间，被触发陷阱所占的百分比，以及在仿真过程中空中出现的乒乓球的最大数目。
19. 1884 年的冬月，萨缪尔·摩尔斯通过电报机从华盛顿向巴尔的摩发送了一条内容为“上帝创造了什么”的消息，预示着电子通信技术时代的来临。为了能够让只使用单音信号的电报和消失来交流信息，摩尔斯设计了一个编码系统。在这个系统中，字母和其他的符号表示成一个短的和长的音调编码序列，这里上称为点（dot）和破折号（dash）。在摩尔斯编码中，字母表中的 26 个字母表示为如图 5-15 所示的编码。

    编写一个程序，从用户处读取若干行信息，要么将每一行信息翻译为摩尔斯编码，要么将摩尔

斯编码翻译为原信息，并根据每一行的首字符对每一行进行翻译：

- 如果这行以一个字母开始，并且你想要将其翻译为摩尔斯编码。除了26个字母以外的其他字符都应该被忽视。

257

A •—	H ••••	O ———	V •••—
B —•••	I ••	P •——•	W •——
C —•—•	J •———	Q ——•—	X —••—
D —••	K —•—	R •—•	Y —•——
E •	L •—••	S •••	Z ——••
F ••—•	M ——	T —	
G ——•	N —•	U ••—	

图5-15 摩尔斯编码

- 如果这一行以一个原点（点）或者一个连字号（破折号）开始，它将会被读成一系列的摩尔斯编码，并且你需要将它们翻译为字母。你可能假设：在输入字符串中每一个点和破折号的序列都以空格隔开，并且可以自由地忽视任何其他出现的字符。由于单词之间的空格没有编码，当你的程序百签翻译时，被翻译的消息的所有字符都会连在一起。

当用户输入空行时，程序停止运行。一个运行该程序的示例（节选自泰坦尼克号与卡帕西亚号之间的消息）可能如下图所示：



20. 美国和加拿大的电话号码被组织成不同的三位区号（area code）。一个单独的州或者省可能有很多个区号，但是一个区号不会在一个州或者一个省的边界上交叉。这个规则能够让你列出一个数据文件中每个区号的地理位置。对于这个问题，假设你能够使用一个名为 AreaCodes.txt 的文件，

258 这个文件列出了每个区号及其对应的位置，文件中开始的几行如下图所示：

AreaCodes.txt

```
201-New Jersey
202-District of Columbia
203-Connecticut
204-Manitoba
205-Alabama
206-Washington
```

使用图5-7中的程序 AirportCodes 作为一个模型，编写必要的代码将这个文件读取到一个 Map<int,string> 对象中，这个对象的键是区号，值是对应的位置。一旦读取完数据，编写一个主程序，让用户反复地输入区号，然后查询对应的位置。如以下示例运行结果：



然而，提示表明，程序也应该允许用户输入州或者省的名字，然后列出所有服务该地区的区号。如以下示例运行结果：



21. 当你为之前的习题编写 FindAreaCode 程序时，通过查找整个地图为某个州产生其区号列表，并且打印出映射到那个州的任意区号。尽管这种策略对于单区号实例这种小的地图是合适的，但这种策略在针对很大的数据地图时效率将会是一个问题。

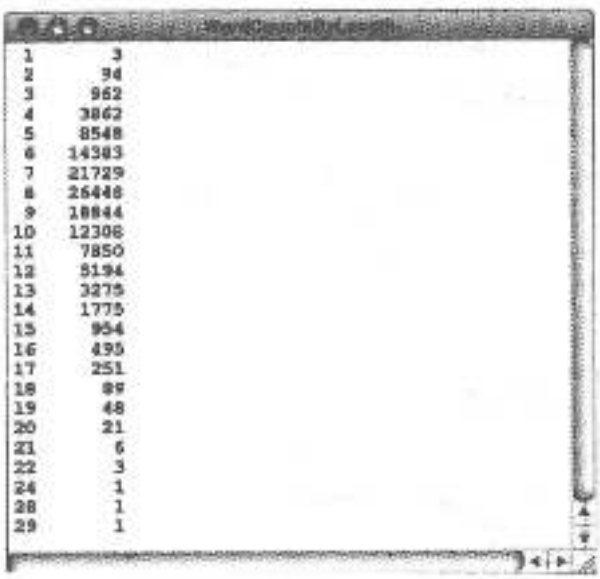
一个可选择的方法是使映射的关键字与值反转，以便你可以在任何一个方向上执行查询操作。然而，你不可以声明一个形如 Map<String,Int> 的映射反转，因为对于一个州，这里经常有超过一个的区号与其对应。替代的方法是将反转的映射声明为一个 Map<String,Vector<Int>> 对象，以便这个 map 能够将每个州的名字与服务该州的区号匹配。重写程序 FindAreaCode 以便 259
在读取完数据文件之后，程序创建了一个反转的映射，然后使用这个映射列出一个州的区号。

22. 3.6 节定义了一个名为 isPalindrome 的函数来检查一个单词从前和从后读是否是一样的。将这个函数与英语字典一起使用，列出所有的回文单词。

23. 在一种名为 Scrabble 的拼字游戏中，已知由两个字母构成的单词列表是很重要的，因为那些短的单词使得它易于“构出”一个已存在的新的单词。拼字游戏所产生的另一个由二个字符所构成的单词列表，它由在二个字符所构成的单词的前面或者后面增加一个字符形成。编写能够生成该列表的程序。

24. Scrabble 拼字游戏最重要的策略性原则就是保存你的 s 构造块，因为英语单词的复数规则是许多单词以 s 结尾。当然，某些单词，大约有 680 个单词，允许只在其单词后添加一个 s 便可构造出一个新的单词，例如，单词 cold 和 bot。编写一个程序，产生所有符合条件的单词列表。

25. 编写一个程序显示一个表格，它根据单词的长度进行排序，并显示了不同长度的单词在英文字典中出现的次数。对于 EnglishWords.dat 中的字典，这个程序的输出如下图所示：



1	3
2	94
3	962
4	3862
5	8548
6	14383
7	21729
8	26448
9	18844
10	12306
11	7850
12	5194
13	3275
14	1775
15	954
16	495
17	251
18	89
19	48
20	21
21	6
22	3
24	1
28	1
29	1

260

第 6 章
Programming Abstractions in C++

类的设计

你不会理解，也许我有条……

261 ——马龙·白兰度在影片《码头风云》中的角色，1954

虽然你已经广泛地使用了本书中大量的类，但是还没有定义过自己设计的类。本章的目标就是通过提供给你实现新类所需的一些工具从而弥补这一空缺。但是本章仅呈现自定义类的一些基础。在后续的章节中，你将有机会学习类设计的其他重要方面，包括内存管理和继承。

6.1 二维点的表示

类的有用特性之一（但绝不是唯一的一个）在于它能将几个相关的信息片断组织成一个复合值，使你可以整体地对其进行操作。举一个简单的例子，你正在对 $x-y$ 网格坐标系上的坐标进行操作，其中，x 和 y 坐标值均为整数。虽然你可以将 x 和 y 的值独立处理，但是定义一种抽象数据类型将 x 和 y 的值组合在一起会更方便。在几何学中，这一对坐标点值称为点（point），因此，将这一数据类型命名为 Point 将更有意义。C++ 语言提供了几种策略来定义 Point 类型，其范围包括从 C 语言家族提供的简单结构类型到采用现代的面向对象设计风格的自定义类型。以下各节将逐步探索这些策略，首先从基于结构的模型开始，然后再到基于类的模式。

6.1.1 将 Point 定义为结构类型

在以前的编程中，你几乎肯定接触过将几个已存在的简单类型数值组合在一起，称作记录（record）或结构（structure）的数据类型。其中，记录这一术语在计算机科学领域中使用得更为广泛，而第二个术语结构则在 C++ 程序员中更为普遍。若 C++ 支持已有的 C 语言机制，你就可以采用 C 语言风格的结构定义将 Point 类型定义为结构类型：

```
struct Point {
   int x;
   int y;
};
```

这段代码将 Point 类型定义成具有两个分量的传统结构类型。在一个结构中，其分量称为域（field）或成员（member）。在本例中，Point 结构分别包括了名字分别为 x 和 y 的两个域，它们均为 int 类型。

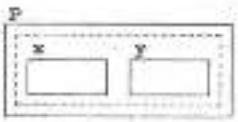
当你在 C++ 语言中采用结构或类时，你应该时刻牢记：新的定义只是引入了一种新的
262 数据类型，并没有声明任何变量。一旦有了数据类型的定义，你就可以像使用其他类型一样声明该类型变量。例如，如果你在一个函数中有如下局部变量声明：

```
Point p;
```

编译器将会在栈帧中为 Point 类型变量 p 分配存储空间，正如以下声明语句：

```
int n;
```

将在栈帧中为 int 类型的变量 n 分配存储空间一样。上述两个声明语句的唯一不同在于 Point 类型变量 p 中包含了内部域 x 和 y 的值。若画一个变量 p 的盒图，它看起来如下图所示：



变量 p 中拥有一个组合值，即其内部域 x 和 y 的值。

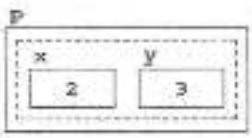
给定一个结构类型的变量，可以采用下述方式使用点操作符来选择其中的某个域：

```
var.name
```

其中，var 是一个结构体变量，而 name 是欲选取的某个域名。例如，可以通过表达式 p.x 和 p.y 来访问存储在 Point 类型变量 p 中的域 x 和 y 的坐标值。选择表达式是可赋值的，因此，可以通过下面的代码初始化变量 p 的各成员值，以表示点（2,3）：

```
p.x = 2;
p.y = 3;
```

上述语句使得变量 p 的状态如下图所示：



结构类型最基本的特性就是你可以将其看作一组独立数据域的集合，并将其整体看作一个值。在结构类型的底层实现中，存储于结构体各个域中的成员值可能是最重要的。但是在其细节的更高层次上，我们更关注将结构作为一个整体看待。 263

C++ 语言通过将一个结构作为整体看待，并为其定义一组重要操作，使其从更高的层次上更易于维护。例如，假定已有一个 Point 类型值，你可以把该值赋给一个 Point 类型的变量，或者将它作为参数传递给一个函数，或者将其作为函数的返回值。同样，用户可以选择该变量的 x 域或者 y 域以单独访问其成员值。但通常将其看作一个整体使用已经足够了。结构类型的设计决策意味着你可以在应用的不同层面来传递一个结构类型变量。但是除非结构类型在最底层，否则其底层实现细节并不重要。

6.1.2 将 Point 定义为类

虽然结构类型是 C++ 语言历史的一部分，甚至比 C++ 语言本身更早出现，但是它已经在很大程度上被更通用更灵活的类所取代。前面小节中的 Point 结构类型与下面的类定义完全相同：

```
class Point {
public:
   int x;
   int y;
};
```

正如你从这个例子中所看到的，对于类中的域——也称为**实例变量**（instance variable），其声明与结构类型中域的声明语法相同。两者唯一的语法差别在于类中的域又划分成公有部分和私有部分，以控制程序对这些域的访问权限。关键字 public 引入了类的**公有部分**（public section），其中所包含的域对于该类的所有用户而言都是可访问的。然而在一个类的定义中，还可以包括由 private 关键字标识的类的**私有部分**（private section）。声明在私有部分的域仅对该类本身可见，而对其他的任何用户不可见。在如今的 C++ 版本中，结构类型和类基本上以同样的方式实现，唯一的不同在于结构的入口默认访问权限是 public，而类的入口默认访问权限是 private。

正如现在 Point 类的定义，x 和 y 域处于类的公有部分，它们对类的用户可见。可以使用点操作符来选择该类对象中的公有域。例如，如果有一个 Point 类的对象 pt，即 Point 类的一个实例，可以通过以下方法来选取其中的 x 域：

```
pt.x
```

264 这正和前面章节中访问 Point 结构类型的方法一样。

然而，现代面向对象编程不鼓励在类中声明 public 的实例变量。今天，通常的做法是将类中的所有实例变量都声明为 private，这意味着用户不能直接访问类中的这些内部变量。用户只能通过类中的 public 方法来间接地访问类中内部变量的值。正如第 5 章所介绍的，我们将实现细节与用户相隔离，这保证了程序的简洁性、灵活性和安全性。

使得类中的实例变量变为 private 非常容易，你仅须将访问权限从 public 变为 private 即可，如下所示：

```
class Point {
private:
   int x;
   int y;
};
```



这一定义存在的问题是用户将永远无法访问 Point 类型对象中存储的信息，这使得该定义形式的 Point 类无法使用。用户至少需要某种方式去创建 Point 对象，并能从中单独地获取其 x 或 y 的坐标值。

正如你在第 5 章中所学到的，创建对象是**构造函数**（constructor）的责任，其构造函数名与类名完全一样。类通常定义了多个构造函数以适应其多种初始化对象的方式。特别是，大多数类都定义了无参数的构造函数，它称为**默认构造函数**（default constructor）。这种默认构造函数在初始化对象时无须提供参数列表。在 Point 类中，定义一个获取一对二维坐标值的构造函数是非常有用的。

在计算机科学中，获取实例变量值的函数一般称为**访问器**（accessor），亦通常称为**读取器**（getter）。为方便起见，读取器的命名通常以单词 get 为前缀，后跟首字母大写的实例变量名。因此，Point 类的读取器名为 getX 和 getY。

为了保持本书先举例，然后再详细解释示例中出现的每一个新概念的一贯策略，图 6-1 展示了 Point 类的核心定义，其中包括具有 public 访问权限的两个构造函数，getX 和 getY 方法，以及 toString 方法。在此例子中，已经省略了用以说明类定义结构以及类中
265 方法的相关注释。

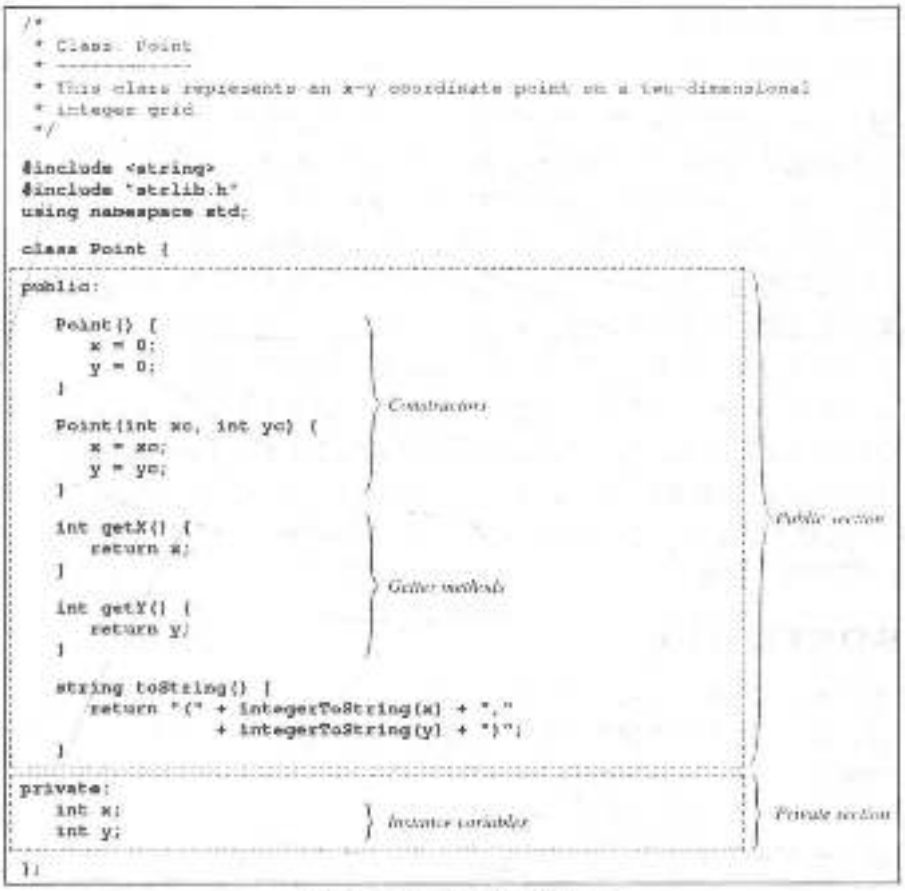


图 6-1 Point 类的简单版本

图 6-1 所示的大部分代码应该很容易理解。唯一难以理解并可能引起混淆的是第二个构造函数的形参名。逻辑上，如果构造函数需要分别传入一个 x 和 y 坐标值，那么构造函数的形参名应毫无疑问应为 x 和 y，而不是 xc 和 yc（这里的字母 c 代表单词 coordinate）。遗憾的是，若在这里使用 x 和 y 来命名构造函数的参数将使对变量的指代发生歧义，即无法分 266
辨是对形参变量 x 的引用，还是对类对象的同名实例变量的引用。

程序代码内层块中的变量隐藏外层块中同名变量的行为称为**遮蔽**（shadowing）。在第 11 章，你将学习解决该二义性的一种简单技术，但遗憾的是，该技术所需的概念已超出了你的知识范围。因此，本书中的例子暂时通过为形参和实例变量选取不同的名称以避免参数遮蔽这一问题。

在阅读了图 6-1 所示的程序代码后，你可能会产生另一个问题：为什么类中没有包含一些其他的方法？虽然读取器方法允许我们获得 Point 类对象中的信息，但是该类并没有提供更新这些域值的方法。为此，在某些情况下，一种有效的办法就是在类中对外提供为特定的实例变量设置值的方法。这样的方法称为**设值方法**（mutator），通常也称为**设值器**（setter）。如果 Point 类对外提供了 setX 和 setY 方法以允许用户改变相应域的值，那么你就能很容易地将之前使用旧版本结构类型的 Point 类型替换为全新的 Point 类版本。而你所要做的就是将以下形式的每一个赋值语句：

```
pt.x = value;
```

更改为如下形式的方法调用：

```
pt.setX(value);
```

类似地，每一个不作为赋值语句中的左值而对 pt.y 变量的引用，必须重写为 pt.getY()。

然而，在刚刚认识到设置实例变量的访问权限为私有的重要性时，你可能会觉得在类中增加设置器这一行为是不可取的。毕竟，将实例变量的访问权限设置为私有部分的原因是为了阻止用户不受任何约束地访问这些变量。因此，为类中的每个实例变量编写一个具有公共访问权限的设置器将赋予用户绕过类中所设的约束权限，并将抵消程序员在开始时将实例变量访问权限设置为私有所带来的优势。通常，只允许用户读取实例变量值会比允许用户改变其值更安全。因此，在面向对象程序设计中，读取器比设置器更为常见。

事实上，许多程序员采取在更高的层次上将类设计为完全不可更改的建议，即一旦对象被创建，其实例变量值不可改变。这种设计风格的类称为是不可变的（immutable）。Point 类就是不可变的，至少在 C++ 语言的定义中是倾向于不可变的。虽然我们还可以通过将
[267] 一个 Point 类对象赋给另一个 Point 对象以改变其内容，但是我们无法单独地改变一个 Point 对象中的各个域值。

6.1.3 接口与实现的分离

只有在图 6-1 中展示的 Point 类与使用该类的代码放在同一个源文件的情况下，它才能发挥作用。通常，在类库中提供了类的定义，并使得这一定义可以应用于大多数的情况，这是更好的做法。为此，你必须创建一个 point.h 文件以表示类的接口，而另一个分开的 point.cpp 文件包含了相应的类的实现。

正如你在第 2 章所看到的，接口通常只包含函数的原型而不包含其完整的实现，这同样适用于类中的方法。类的接口定义只包含方法原型，而将其具体代码推迟到其实现中。因此，类的头文件与其他类库的头文件是类似的，唯一的不同在于：对类而言，方法原型定义在类中。但是类的实现文件结构与类库相比却是不同的。

在 C++ 中，当将类的接口与实现进行分离时，类自身的定义仅存在于它的 .h 文件中。其对应的实现放在 .cpp 文件中，它作为独立方法定义，而不像方法原型嵌套在类的定义中。因此，这些方法的定义必须以不同的方式表明自己属于哪个类。在 C++ 中，通过在方法名前添加类名作为**限定符**（qualifier），并使用双冒号分隔类名与方法名。因此，Point 类中的 getX 方法全名是 Point::getX。

一旦排除了小小的语法瑕疵，类的实现代码对你就没有任何障碍。图 6-2 提供了 Point 类的完整接口，其对应的实现代码显示在图 6-3 中。

```
/*
 * File: point.h
 * -------------
 * This interface exports the Point class, which represents a point on
 * a two-dimensional integer grid.
 */

#ifndef _point_h
#define _point_h

#include <string>

class Point {
```

图 6-2 Point 类的初步接口

```
public:

/*
 * Constructor: Point
 * Usage: Point origin;
 *        Point pt(xc, yc);
 * ------------------------
 * Creates a Point object.  The default constructor sets the coordinates
 * to 0; the second form sets the coordinates to xc and yc.
 */

   Point();
   Point(int xc, int yc);

/*
 * Methods: getX, getY
 * Usage: int x = pt.getX();
 *        int y = pt.getY();
 * ------------------------
 * Return the x and y coordinates of the point, respectively.
 */

   int getX();
   int getY();

/*
 * Method: toString
 * Usage: string str = pt.toString();
 * ----------------------------------
 * Returns a string representation of the Point in the form "(x,y)".
 */

   std::string toString();

private:

   int x;                       /* The x-coordinate */
   int y;                       /* The y-coordinate */

};

#endif
```

图 6-2 （续）

```
/*
 * File: point.cpp
 * ---------------
 * This file implements the point.h interface.
 */

#include <string>
#include "point.h"
#include "strlib.h"
using namespace std;

/*
 * Implementation notes: Constructors
 * ----------------------------------
 * The constructors initialize the instance variables x and y.  In the
 * second form of the constructor, the parameter names are xc and yc
 * to avoid the problem of shadowing the instance variables.
 */

Point::Point() {
   x = 0;
   y = 0;
}

Point::Point(int xc, int yc) {
   x = xc;
   y = yc;
}

/*
 * Implementation notes: Getters
 * -----------------------------
```

图 6-3 Point 类的初步实现

```
 * The getters return the value of the corresponding instance variable.
 * No setters are provided to ensure that Point objects are immutable.
 */

int Point::getX() {
   return x;
}

int Point::getY() {
   return y;
}

/*
 * Implementation notes: toString
 * ------------------------------
 * The implementation of toString uses the integerToString function
 * from the strlib.h interface.
 */

string Point::toString() {
   return "(" + integerToString(x) + "," + integerToString(y) + ")";
}
```

图 6-3 （续）

6.2 操作符重载

通过之前章节关于类库的经验已知，C++ 允许我们扩展标准操作符使其适用于新的数据类型。这一技术称为**操作符重载**（operator overloading）。例如，string 类重载了 + 操作符，使 + 操作符能应用到字符串上，以产生与其在基本类型上不同的效果。当 C++ 编译器看到 + 操作符时，它会根据 + 操作符操作数的类型来确定其操作语义。这一行为与通过参数签名来决定使用哪个重载版本的函数类似。如果 C++ 编译器检测到 + 操作符作用于两个整数，那么它将执行两个整数相加并产生整数结果的指令。如果操作对象是 string 类型，编译器将产生对 string 类中提供的重载函数 + 的调用来实现字符串的连接。

268
:
270

操作符重载是 C++ 语言的一个强大特性，这一特性使得程序更易于阅读，但前提是每一种操作符对各种类型的解释是一致的。重载了 + 操作符的类使用这一操作符来实现概念上与加法相似的操作，例如连接字符串。对两个 string 类型变量，编写以下表达式：

```
s1 + s2
```

我们可以很容易地理解这一操作：它将两个字符串串接到一起，形成一个新的字符串。但是，如果你重新定义了让读者无法理解其语义的操作符，则操作符重载会使程序变得晦涩难懂。因此，学会有限制地使用操作符重载这一特性以提高程序的可读性显得尤为重要。

以下各节将说明怎样在你自己定义的类中实现操作符重载，我们将以 6.1 节中的 Point 类作为该示例的开始，然后对第 1 章中引入的 Direction 类增加一些有用的操作符以进行操作符重载的实践。

6.2.1 重载插入操作符

正如你在图 6-2 所示的文件 point.h 接口中看到的，Point 类提供了 toString 方法，它将 Point 对象中包含的信息转换成由一对圆括号括起来的坐标值字符串。引入这一方法的主要是为了更容易地显示 Point 类对象的值。当调试一个程序时，显示一个变量的值通常可以带来便利。为了显示 Point 类型变量 pt 的值，所要做的就是在程序中添加以下语句：

```
cout << "pt = " << pt.toString() << endl;
```

操作符重载可以进一步简化这一过程。C++已经重载了流插入操作符<<，以便它可以显示字符串及基本类型数据。如果你重载了这一操作符以支持Point类，可以将上述语句简化为：

```
cout << "pt = " << pt << endl;
```

这只是一个细微的改变，但输出一个点的值也因此更加简便。

C++中的每一个操作符都与定义其重载操作符行为的重载函数名相关。大多数情况下，重载函数名由关键字operator后跟操作符构成。例如，如果你想为某种新类型重新定义操作符+，你必须定义一个名为operator+的函数，并传入该类型的实参。类似地，可以 [271]
通过重新定义函数operator<<来重载插入操作符。

编写operator<<函数时遇到的最大挑战是如何给这个函数编写函数原型。<<操作符的左操作数是一个输出流，但<iostream>类库中已经定义了一个完整的输出流层次结构。在大多数情况下，使用最通用的类来实现必要的操作是一个很好的选择。在输出流层次结构中，最通用的类是ostream。<<操作符的右操作数是你想要嵌入到输出流中的Point类对象的信息。因此，<<操作符的重载定义需要传入两个实参：一个是ostream类型，另一个是Point类型。

然而，完成该函数原型你还需考虑这种情况：流对象是不能拷贝的。这一约束意味着ostream类型参数必须采用引用传递。同样地，这一约束也在插入操作符<<返回时产生影响。正如在本章开头所介绍的那样，插入操作符通过返回输出流在连接流语句中起到重要作用，其返回结果可以参与到链中的下一个<<操作符运算。为了避免在该操作过程结束后拷贝流，operator<<的定义也必须通过引用来返回结果。

通过引用来传递函数参数要比通过引用来返回函数结果要常见得多。类似刚才的<<操作符重载例子，对于那些通过引用返回结果的应用，你只需要知道定义引用返回与定义引用参数传递的语法大致相同：只需要在函数返回类型后面加上一个&符号即可。

综合上述要点，我们应该像下面这样定义operator<<函数的重载版本：

```
ostream & operator<<(ostream & os, Point pt);
```

该函数的实现必须在输出流中输出pt对象的字符串表示，然后通过引用返回这个流，使得该流可以在程序中继续使用。如果依次实现了这些步骤，会得到以下代码：

```
ostream & operator<<(ostream & os, Point pt) {
   os << pt.toString();
   return os;
}
```

然而，也可以像下面这样将上述实现精简为一行代码：

```
ostream & operator<<(ostream & os, Point pt) {
   return os << pt.toString();
}
```

[272]

本书中定义的类均使用了上述第二种形式的代码，它重点强调了<<操作符返回输出流这一事实。

6.2.2 判断两个点是否相等

如果你阅读过 Standford 类库中最终版本的 point.h 接口，你会发现 Point 类除了提供流插入操作之外，还提供了其他操作符。例如，给定两个点 p1 和 p2，可以采用 == 操作符来判断这两个点是否相等，这与检验字符串和基本类型是否相等一样。

C++ 提供了两种机制用以重载内置的操作符，以保证它们可适用于新定义类的对象：

1. 可以在类中用一个方法来重载一个操作符。当采用这种方式在类中重载一个二元操作符时，其左操作数为该类型的对象，而右操作数则作为形参传递进来。

2. 可以在类外使用一个自由函数（free function）来重载定义一个操作符。如果采用这种方式，则二元操作符的两个操作数都必须通过形参传递进来。

如果采用基于方法的操作符重载方法，在 Point 类中重载 == 操作符的第一步就是在其 point.h 接口中增加 == 操作符的函数原型，如下所示：

```
bool operator==(Point rhs);
```

该方法是 Point 类的一部分，因此必须定义在公有部分。相关的实现将放置在 point.cpp 文件中，其中可能需要增加如下代码：

```
bool Point::operator==(Point rhs) {
   return x == rhs.x && y == rhs.y;
}
```

与类中通过其接口导出的其他方法一样，operator== 的实现必须通过在方法名中增加 Point:: 前缀来声明其属于 Point 类。

调用这个方法的用户代码看起来如下所示：

```
if (pt == origin) ...
```

假设 pt 和 origin 均为 Point 类型的变量，编译器在执行 pt==origin 表达式时，会从 Point 类中调用 == 操作符。因为 operator== 是一个方法，所以，编译器将把变
273 量 pt 指派为接收者，并拷贝 origin 变量的值传送给形参 rhs。在 operator== 方法体中，无任何限制符的变量 x 和 y 是指变量 pt 的数据域，而 rhs.x 和 rhs.y 指的是变量 origin 的数据域。

operator== 方法的代码展示了面向对象编程的一个重要特性。operator== 方法显然可以访问当前对象的 x 和 y 域，因为一个类中的任意方法可以访问该类的私有变量。其中比较难理解的一点是，为何 operator== 方法在所属对象完全不同的情况下也可以同时访问 rhs 对象的私有变量。在 C++ 中，这种引用是合法的，因为类中定义的私有部分对该类来说是私有的，但对对象而言并非如此。类中方法的代码可以引用该类型的任何对象的实例变量。

根据我的经验，学生通常会对基于方法形式的操作符重载感到迷惑，因为编译器处理左操作数和右操作数的方式是不同的，它会把左操作数指派为接收者，并将右操作数以形参传递进来。恢复这一操作数对称性的最简单方法就是选择其他方式将操作符重载定义为自由函数。如果采用这一策略，则需要在 point.h 接口中包含如下函数原型：

```
bool operator==(Point p1, Point p2);
```

这一函数原型声明了一个自由函数，因此，它必须出现在 Point 类的定义之外。其对应的

实现如下述代码所示，它不再包含 Point:: 前缀，因为该函数不是类的一部分：

```
bool operator==(Point p1, Point p2) {
   return p1.x == p2.x && p1.y == p2.y;
}
```

虽然这个实现由于它同等地处理了形参 p1 和 p2 而更易于效仿，但这段代码却面临着一个严重的问题：它无法真正运行。事实上，如果在 Point 类中增加这段定义，代码将不能通过编译。问题的症结在于 == 操作符是通过自由函数的方式定义的，因此该代码并不能访问类中的私有实例变量 x 和 y。

这里并没有添加一个错误图标，因为 == 操作符实现代码的最终版本正是这个例子中所呈现的代码。幸运的是，C++ 语言提供了另外一种用于解决访问权限问题的方法。由于 == 操作符出现在 point.h 接口中，它概念上与 Point 类相关联，因此在某种程度上被当成该收者而拥有访问类中私有变量的权限。 274

为了使这种设计可行，Point 类必须让 C++ 编译器知道：对于一个特定的函数，== 操作符重载版本可适当地允许访问类中的私有实例变量。为了使这种访问合法，Point 类必须将 operator== 函数定义为友元（friend）函数。此时，友元的特性与在社交网络中的友情特性是类似的。其私有信息一般都不会在社会中大范围地共享，而只会对你所认可的朋友开放。

在 C++ 中，将自由函数声明为友元函数的语法如下：

```
friend prototype;
```

其中，prototype 就是函数原型。在这个例子中，通过书写如下语句将 operator== 函数声明为 Point 类的友元函数：

```
friend bool operator==(Point p1, Point p2);
```

这行语句是类定义的一部分，因此也必须是 point.h 接口的一部分。

在 C++ 中，一个类可以用以下声明使其成为另一个类的友元，从而访问该类的私有信息：

```
friend class name;
```

其中，name 是类名。在 C++ 中，这种关于友元的声明并非自动是双向的。如果两个类都需要获取对方私有变量的访问权限，则这两个类都必须显式地将另一个类声明为其友元类。

每当为一个类重载操作符 == 时，最好的做法是同时为该类提供 != 重载操作符。毕竟用户希望判断两个点是否不同与判断这两个点是否相同同样容易。C++ 并未默认 == 操作符和 != 操作符运算返回相反的结果；如果你想要得到这种相反的结果，你必须单独重载这两个操作符。但是，在实现 operator!= 时，可以利用 operator== 的实现，因为 operator== 是类的一个公有方法。因此，最直截了当地重载 != 操作符的函数看起来如以下代码所示：

```
bool operator!=(Point p1, Point p2) {
   return !(p1 == p2);
}
```

Point 类的最终版本在下几页中给出。图 6-4 包含了 point.h 接口，图 6-5 包含了对应的 point.cpp 的实现。 275

```
/*
 * File: point.h
 * -------------
 * This interface exports the Point class, which represents a point on
 * a two-dimensional integer grid.
 */

#ifndef _point_h
#define _point_h

#include <iostream>
#include <string>

class Point {

public:

/*
 * Constructor: Point
 * Usage: Point origin;
 *        Point pt(xc, yc);
 * ------------------------
 * Creates a Point object.  The default constructor sets the coordinates
 * to 0; the second form sets the coordinates to xc and yc.
 */

   Point();
   Point(int xc, int yc);

/*
 * Methods: getX, getY
 * Usage: int x = pt.getX();
 *        int y = pt.getY();
 * ------------------------
 * These methods return the x and y coordinates of the point.
 */

   int getX();
   int getY();

/*
 * Method: toString
 * Usage: string str = pt.toString();
 * ----------------------------------
 * Returns a string representation of the Point in the form "(x,y)".
 */

   std::string toString();

/* Private section */

private:

/* Friend declaration */

   friend bool operator==(Point p1, Point p2);

/* Instance variables */

   int x;                       /* The x-coordinate */
   int y;                       /* The y-coordinate */

};

/*
 * Operator: <<
 * Usage: cout << pt;
 * ------------------
 * Overloads the << operator so that it is able to display Point values.
 */

std::ostream & operator<<(std::ostream & os, Point pt);

/*
 * Operator: ==
 * Usage: p1 == p2
 * ---------------
```

图 6-4 Point 类的完整接口

```
 * Implements the == operator for points.
 */

bool operator==(Point p1, Point p2);

/*
 * Operator: !=
 * Usage: p1 != p2
 * ---------------
 * Implements the != operator for points.  It is good practice to
 * overload this operator whenever you overload == to ensure that
 * clients can perform either test.
 */

bool operator!=(Point p1, Point p2);

#endif
```

图 6-4 （续）

```
/*
 * File: point.cpp
 * ---------------
 * This file implements the point.h interface.  The comments have been
 * eliminated from this listing so that the implementation fits on a
 * single page.
 */

#include <string>
#include "point.h"
#include "strlib.h"
using namespace std;

Point::Point() {
   x = 0;
   y = 0;
}

Point::Point(int xc, int yc) {
   x = xc;
   y = yc;
}

int Point::getX() {
   return x;
}

int Point::getY() {
   return y;
}

string Point::toString() {
   return "(" + integerToString(x) + "," + integerToString(y) + ")";
}

bool operator==(Point p1, Point p2) {
   return p1.x == p2.x && p1.y == p2.y;
}

bool operator!=(Point p1, Point p2) {
   return !(p1 == p2);
}

ostream & operator<<(ostream & os, Point pt) {
   return os << pt.toString();
}
```

图 6-5 Point 类的完整实现

6.2.3 为 Direction 类型增加操作符

虽然在类定义中操作符重载比较常见，但是 C++ 也允许你扩展操作符定义，使得该操作符可以应用到枚举类型。这一特性允许我们给第 2 章介绍的 direction.h 接口中添加两个操作符，它使得 Direction 枚举类型更易于使用。

与为 Point 类重载 << 操作符的原因完全相同，为 Director 类型重载 << 操作符将使该类型更为有用。如果 direction.h 接口提供了 directionToString 函数，我们可以直接实现 operator<< 函数的扩展版本：

```
ostream & operator<<(ostream & os, Direction dir) {
   return os << directionToString(dir);
}
```

与类库接口中的其他函数类似，该函数体存在于 direction.cpp 文件中，而它的函数原型则必须出现在 direction.h 文件中。

在介绍下一个更重要和更精妙的操作符之前，必须认识到 Direction 类型提供的功能是受限制的。正如将在第 9 章中所看到的，迭代 Direction 类型中的元素是非常有用的，即依次循环地在 NORTH、EAST、SOUTH 和 WEST 这四个值中进行迭代。为此，所要做的就是像下面这样毫不犹豫地使用 for 循环：

```
for (Direction dir = NORTH; dir <= WEST; dir++) ...
```

遗憾的是，该语句不适用于当前定义的 Direction 类型的数据。问题的关键在于 ++ 操作符不能操作枚举类型。为了使 dir++ 这一语句可以运行，必须如下所示编写一个并不优雅的表达式：

```
dir = Direction(dir + 1)
```

再一次，操作符重载有助于解决问题。为了使标准 for 循环可以像往常一样精准运行，必须为 Direction 类型重载 ++ 操作符。然而，做这件事并不简单。在 C++ 中，++ 和 -- 操作符是特殊的，因为它们彼此都分别有前缀和后缀两种形式。当它们在表达式中作前缀操作符时，如在表达式 ++x 中，操作符先运算，表达式的最终结果是运算完成后的变量值。当它们作为后缀操作符时，如在表达式 x++ 中，变量的值将会发生同样的改变，但是该表
279 达式的值是变量参加 ++ 运算之前的值。

当用 C++ 重载 ++ 或 -- 操作符时，必须告诉编译器你想要重载的操作符是前缀形式还是后缀形式。C++ 的设计者选择了通过传入一个无意义的整型参数的方式来说明其操作符为后缀形式，用以区别其前缀形式。因此，为了重载 Direction 类型的前缀 ++ 操作符，定义如下函数：

```
Direction operator++(Direction & dir) {
   dir = Direction(dir + 1);
   return dir;
}
```

为了重载其后缀操作符，应如下所示定义函数：

```
Direction operator++(Direction & dir, int) {
   Direction old = dir;
   dir = Direction(dir + 1);
   return old;
}
```

注意，dir 参数必须以引用方式进行传递，以保证函数可以改变其变量值。这个例子也说明了：在 C++ 中，如果不需要使用形参值，则可以不用此形参名。

若重载这一操作符的目的只是保证标准 for 循环语句的正常运行，那么类库版本的 Direction 类型只重载了操作符的后缀形式。这种扩展对于那些需要进行元素迭代的枚举类型具有重要意义。

一旦你完成了本节所介绍的 ++ 操作符的重载之后，Direction 类型将变得更易于使用。例如，如果执行以下语句：

```
for (Direction dir = NORTH; dir <= WEST; dir++)
   cout << dir << endl;
```

会得到如下输出：



如果未进行 ++ 操作符的重载，要产生这一输出将会困难得多。 280

6.3 有理数

虽然 6.1 节定义的 Point 类表明了定义一个新类的基本方法，但是要对这一章的主题形成一个牢固的认识体系却需要我们接触更多更复杂的例子。本节以**有理数**（rational number）类为例来让你更深入地了解类的设计。其中，有理数是指所有可以用两个整数的商来表示的数。在中学阶段，你们可能将这类数称为**分数**（fraction）。

在某些方面，有理数与你在第 1 章中使用的浮点数类似。这两种类型的数都可以用来表示分数值，例如 1.5 就是有理数 3/2。它们的不同之处在于有理数是精确的，而浮点数却因为受限于硬件的精度只能是近似值。

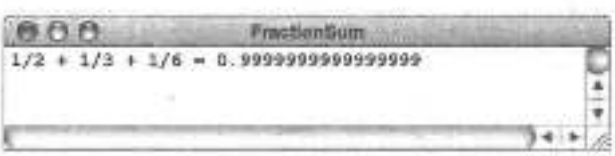
为了更好地理解两者差别的重要性，考虑求下述分数之和这样一个数学问题：

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{6}$$

基本算术（甚至是直觉）都清晰地告诉我们算术精确结果为 1，但如果你使用 double 类型进行计算将很难得到这一结果。下面的程序通过使用双精度浮点数来计算其和并显示 16 位精度的计算结果来说明这一问题：

```
int main() {
   double a = 1.0 / 2.0;
   double b = 1.0 / 3.0;
   double c = 1.0 / 6.0;
   double sum = a + b + c;
   cout << setprecision(16);
   cout << "1/2 + 1/3 + 1/6 = " << sum << endl;
   return 0;
}
```

如果运行该程序，会得到以下结果：



问题在于计算机用来存储数值的内存单元的存储能力是有限的，它只能提供有限的数字精度。在双精度算术的限制下，1/2 加上 1/3 再加上 1/6 的和更接近 0.999 999 999 999 999 9， 281

而不是1.0。更糟糕的是，计算所得的和会小于1，并且会在测试程序中如实输出。当程序运行结束时，表达式sum<1的值为true，而sum==1的值为false。这个结果从数学角度上来说是完全错误的。

与此相反，有理数计算不会出现四舍五入错误，因为计算过程并不涉及任何取近似值的操作。此外，有理数遵循如图6-6所总结的良好定义的计算规则。但是C++的预定义类型中并不包括有理数。如果你想在C++中使用有理数，那么必须定义一个类来表示有理数。

6.3.1 定义新类的机制

使用面向对象语言定义新类是你必须掌握的一项重要技能。在大多数编程中，设计新类既是一门科学更是一门艺术。开发高效的类设计需拥有良好的美学素养，同时需要考虑将类作为工具的用户的需求和期望。理论与实践是最好的老师，但是遵循一个普遍适用的设计框架可以对你设计高效的类提供帮助。

根据我个人的编程经验，我发现遵循按部就班的方法常常是很有用的：

1. 从普遍性的角度出发，思考用户会如何使用一个类。在设计过程的最开始，你就必须确立一种思想：类库是为了满足用户的需求而不是为了方便类的实现者。从专业的角度上说，保证新设计的类更好迎合用户需求的最佳方法就是让用户参与到设计过程中。不管怎样，你至少应该站在用户的角度来描绘类的设计蓝图。



282

图6-6 有理数的算术运算法则

2. 确定什么信息属于类的每个对象的私有部分。虽然类中的私有部分从概念上说是类实现的一部分，但是对于该类的对象必须包括哪些信息有一个初步的了解可以简化接下来的类设计阶段。许多情况下，你可以写下将要放置在私有部分的实例变量。虽然在这个阶段并不需要做到如此清晰的划分，但是对于类的内部结构有一个清晰的印象可以使得构造函数和方法的定义变得更加简单。

3. 定义一组多套的构造函数以创建新的类对象。因为类通常会定义多个重载形式的构造函数，所以站在用户的角度考虑用户需要创建的对象类型及在创建对象时将会传入的信息将会非常有用。典型地，每个类都会提供一个默认构造函数，该函数允许用户声明该类对象并在之后对其进行初始化。在这个阶段，还必须思考构造函数是否需要设置约束条件来确保最终生成对象的合法性。

4. 列举出将成为类的公有方法的所有操作。这一阶段的目标是为类中提供的方法编写其原型，从而将你刚开始时开发的类框架转化成详细说明。也可以在这个阶段中细化整体设计，它遵循第2章提出的准则：统一性、简单性、充分性、通用性和稳定性。

5. 编码并测试其实现。一旦完成了类的接口设计，就需要编写代码来实现该类。编写实现过程不仅是为了得到一个可以运行的程序，还要为设计提供验证。在编写实现代码时，你还需要不时回顾其接口设计，例如，当你发现很难将设计中一种特性的性能控制在一个预定的水平

时。作为一个实现者，你也有责任测试你的实现代码以确保类提供了接口中计划实现的功能。接下来的各节将遵循上述步骤来设计 Rational 类。

6.3.2 采用用户的观点

设计 Rational 类的第一步，你需要考虑用户可能期望得到的类特性。在一个大公司里，你可能拥有多个实现小组，他们都需要使用有理数，并且都可能向你提供他们对于这个类的期望。在这种情况下，让这些用户参与到类的设计中，并且共同制定设计目标将是非常有帮助的。

然而，由于这个例子是基于教科书的背景设计的，因此，你不可能组织与预期用户之间 283
的会议。这个例子的最主要目的是说明在 C++ 中类定义的结构。鉴于上述约束，以及需要控制管理该实例的复杂性，因此，可以对 Rational 类的设计目标加以限定，使得该类只实现图 6-6 中所示的算术操作。

6.3.3 确定 Rational 类的私有实例变量

对于 Rational 类而言，其私有部分易于说明。一个有理数被定义成两个整数的商。因此，每个有理数对象都必须始终保持对这两个整数值的追踪。所以类中私有部分中声明的实例变量如下：

```
int num;
int den;
```

这些变量名是数学术语分子（numerator）和分母（denominator）的简称。这两个术语分别代表了分数的上半部分和下半部分。

有趣的是，我们会发现 Point 类和 Rational 类的实例变量只是名称不同，其他特性均相同。这两个类所维护的值都由一对整数构成。这两个类所不同的是这些整数所代表的实际含义不同，这也反映在每个类所支持的操作中。

6.3.4 为 Rational 类定义构造函数

给定代表两个整数商的一个有理数，则其中一个 Rational 类的构造函数将会获取代表一个有理数的两个整数。例如，有这样一个构造函数，可以通过调用 Rational(1,3) 来定义有理数 1/3。这种构造函数的原型是类接口的一部分，其原型为：

```
Rational(int x, int y);
```

虽然在这一阶段没有必要考虑其实现细节，但是在脑海中保留实现过程的初步印象可以使得你接下来的工作更加轻松。在这一阶段，必须意识到像下面这样实现构造函数是不合适的：

```
Rational(int x, int y) {
   num = x;
   den = y;
}
```



284

这个实现的问题是该构造函数没有考虑到算术法则对于分子和分母数值形式的约束，这一约束必须体现在构造函数中。其中最明显的约束条件就是分母的值不能为零。构造函数必须检查这一情况，并且在这种情况可能发生时抛出错误提示。除此之外，还存在其他更加细微的约束条件。如果用户传入了不符合规格的分子和分母值，那么同一个有理数可能会有很多种

不同的表现方式。例如，有理数 1/3 可以写成以下几种分数形式：

$$\frac{1}{3} \qquad \frac{2}{6} \qquad \frac{100}{300} \qquad \frac{-1}{-3}$$

以上所给出的这些分数都表示同一个有理数，因此，在一个 Rational 对象中允许任意的分子、分母值组合是不优雅的。如果每一个有理数都有一个一致、唯一的表示，将简化其实现过程。

在数学上，通过遵循以下规则来达到上述目标：

- 分数始终以最简项来表示，这意味着分子、分母的所有公因子都必须约掉。事实上，对分数进行化简的最简单方法是将分子和分母除以它们的最大公约数，而最大公约数的计算可以使用 2.1 节介绍的 gcd 函数。
- 分母总是正数，这也意味着有理数值的符号存储在分子中。
- 有理数 0 通常用 0/1 表示。

可以采用下面的代码实现构造函数：

```
Rational(int x, int y) {
   if (y == 0) error("Rational: Division by zero");
   if (x == 0) {
      num = 0;
      den = 1;
   } else {
      int g = gcd(abs(x), abs(y));
      num = x / g;
      den = abs(y) / g;
      if (y < 0) num = -num;
   }
}
```

285 作为一个通用规则，每一个类都应该拥有一个默认的构造函数，在声明该类型对象时无须传入任何参数的情况下使用它。默认有理数最理想的值是零，该值表示为 0/1。因此，其默认构造函数的实现代码如下：

```
Rational() {
   num = 0;
   den = 1;
}
```

最后，定义第三个版本的构造函数是非常有用的，它允许用户创建整数 Rational 对象，此时，分母始终为 1：

```
Rational(int n) {
   num = n;
   den = 1;
}
```

6.3.5 为 Rational 类定义方法

之前提到限制 Rational 类的功能是为了实现其最基本的算术操作，因此，特别是在 C++ 中，弄清楚该类提供的方法相对比较简单。包括 Java 语言在内的许多面向对象编程语言只能通过方法来定义算术操作，比如定义函数 add、subtract、multiply 和 divide 来实现四则算术运算。更糟糕的是，你必须使用接收者所提供的语法来调用这些

函数。而不是编写如下直观上令人满意的声明：

```
Rational sum = a + b + c;
```

像 Java 语言会要求你编写成如下形式：

```
Rational sum = a.add(b).add(c);
```

尽管不难理解该表达式的含义，但是这种方式在重新定义默认提供的操作符方面缺少灵活性和表现力。

对此，C++ 有更好的语言机制。C++ 允许通过重载类作符 +、-、* 和 / 来实现针对 Rational 对象的有理数算术运算。与 6.2 节的 Point 类一样，将这些操作符重载函数定义成自由函数而不是类的方法将更加方便。这也意味着这四个操作符的函数原型如下所示：

```
Rational operator+(Rational r1, Rational r2);
Rational operator-(Rational r1, Rational r2);
Rational operator*(Rational r1, Rational r2);
Rational operator/(Rational r1, Rational r2);
```

286

类似于 Point 类中的 == 操作符，上述算术操作符需要访问 Rational 类对象 r1 和 r2 的域。这意味着这些操作符重载函数必须声明为 Rational 类的友元函数。

虽然 Rational 类的专业实现还必须包括很多其他有用的方法和操作，但是在本例中我们想要增加的只有 toString 方法和一个重载的 << 操作符，而添加这些方法是让你养成以后在设计类时自觉实现这些机制的习惯。让一个类可以将其对象所包含的值以用户可读的方式显示出来与程序调试和编译是同等重要的，而后者是程序开发过程的重要阶段。

这些设计决策可使我们顺利地完成 rational.h 接口的定义，其详细代码如图 6-7 所示。

```
/*
 * File: rational.h
 * ----------------
 * This interface exports a class for representing rational numbers.
 */

#ifndef _rational_h
#define _rational_h

#include <string>
#include <iostream>

/*
 * Class: Rational
 * ---------------
 * The Rational class is used to represent rational numbers, which
 * are defined to be the quotient of two integers.
 */

class Rational {

public:

/*
 * Constructor: Rational
 * Usage: Rational zero;
 *        Rational num(n);
 *        Rational r(x, y);
 * -------------------------
 * Creates a Rational object.  The default constructor creates the
 * rational number 0.  The single-argument form creates a rational
 * number equal to the specified integer, and the two-argument form
 * creates a rational number corresponding to the fraction x/y.
 */

   Rational();
```

图 6-7 Rational 类的接口

```
   Rational(int n);
   Rational(int x, int y);

/*
 * Method: toString()
 * Usage: string str = r.toString();
 * ---------------------------------
 * Returns the string representation of this rational number.
 */

   std::string toString();

/* Declare the operator functions as friends */

   friend Rational operator+(Rational r1, Rational r2);
   friend Rational operator-(Rational r1, Rational r2);
   friend Rational operator*(Rational r1, Rational r2);
   friend Rational operator/(Rational r1, Rational r2);

/* Private section */

private:

/* Instance variables */

   int num;     /* The numerator of this Rational object   */
   int den;     /* The denominator of this Rational object */

};

/*
 * Operator: <<
 * ------------
 * Overloads the << operator so that it is able to display Rational values
 */

std::ostream & operator<<(std::ostream & os, Rational rat);

/*
 * Operator: +
 * Usage: r1 + r2
 * --------------
 * Overloads the + operator so that it can add rational numbers.
 */

Rational operator+(Rational r1, Rational r2);

/*
 * Operator: -
 * Usage: r1 - r2
 * --------------
 * Overloads the - operator so that it can subtract rational numbers.
 */

Rational operator-(Rational r1, Rational r2);

/*
 * Operator: *
 * Usage: r1 * r2
 * --------------
 * Overloads the * operator so that it can multiply rational numbers.
 */

Rational operator*(Rational r1, Rational r2);

/*
 * Operator: /
 * Usage: r1 / r2
 * --------------
 * Overloads the / operator so that it can divide rational numbers.
 */

Rational operator/(Rational r1, Rational r2);

#endif
```

图 6-7 （续）

6.3.6 实现 Rational 类

完成 Rational 类的最后一步是编写图 6-8 所示的代码。其中唯一的复杂部分就是类

构造函数的实现，它的核心代码你已经看过了，因此 rational.cpp 的内容已相当直观。

特别是当你将像作符重载函数以自由函数的形式实现时，你会发现其实现过程将直接遵循图 6-6 所示的数学定义。例如，operator+ 函数的实现过程如下：

```
Rational operator+(Rational r1, Rational r2) {
   return Rational(r1.num * r2.den + r2.num * r1.den,
                   r1.den * r2.den);
}
```

它是对以下有理数 r1 和 r2 加法运算法则的直接翻译：

$$r1 + r2 = \frac{r1_{num}\, r2_{den} + r2_{num}\, r1_{den}}{r1_{den}\, r2_{den}}$$

Rational 类的实现还包括一个名为 gcd 的私有方法，它实现了求最大公约数的欧几里得算法，这个方法在 2.1 节已进行了介绍，其实现在 rational.cpp 文件中。然而，私有方法的原型必须放置在类的私有部分中，因此，即使用户不必调用这一方法，其方法原型也必须是 rational.h 文件中的一部分。如果你仅以用户的身份来使用该类，即使 C++ 要求类的私有部分必须包含在类定义中，你也必须忽略类中的私有部分。

287 ↓ 289

```
/*
 * File: rational.cpp
 * ------------------
 * This file implements the Rational class.
 */

#include <string>
#include <cstdlib>
#include "error.h"
#include "rational.h"
#include "strlib.h"
using namespace std;

/* Function prototypes */

int gcd(int x, int y);

/*
 * Implementation notes: Constructors
 * ----------------------------------
 * There are three constructors for the Rational class.  The default
 * constructor creates a Rational with a zero value, the one-argument
 * form converts an integer to a Rational, and the two-argument form
 * allows you to specify a fraction.  The constructors ensure that
 * the following invariants are maintained:
 *
 * 1. The fraction is always reduced to lowest terms.
 * 2. The denominator is always positive.
 * 3. Zero is always represented as 0/1.
 */

Rational::Rational() {
   num = 0;
   den = 1;
}

Rational::Rational(int n) {
   num = n;
   den = 1;
}

Rational::Rational(int x, int y) {
   if (y == 0) error("Rational: Division by zero");
   if (x == 0) {
      num = 0;
      den = 1;
```

图 6-8 Rational 类的实现

```
   } else {
      int g = gcd(abs(x), abs(y));
      num = x / g;
      den = abs(y) / g;
      if (y < 0) num = -num;
   }
}
/* Implementation of toString and the << operator */

string Rational::toString() {
   if (den == 1) {
      return integerToString(num);
   } else {
      return integerToString(num) + "/" + integerToString(den);
   }
}

ostream & operator<<(ostream & os, Rational rat) {
   return os << rat.toString();
}

/*
 * Implementation notes: arithmetic operators
 * ------------------------------------------
 * The implementation of the operators follows directly from the definitions.
 */

Rational operator+(Rational r1, Rational r2) {
   return Rational(r1.num * r2.den + r2.num * r1.den, r1.den * r2.den);
}

Rational operator-(Rational r1, Rational r2) {
   return Rational(r1.num * r2.den - r2.num * r1.den, r1.den * r2.den);
}

Rational operator*(Rational r1, Rational r2) {
   return Rational(r1.num * r2.num, r1.den * r2.den);
}

Rational operator/(Rational r1, Rational r2) {
   return Rational(r1.num * r2.den, r1.den * r2.num);
}

/*
 * Implementation notes: gcd
 * -------------------------
 * This implementation uses Euclid's algorithm to calculate the
 * greatest common divisor.
 */

int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

290
|
291

图 6-8 （续）

6.4 token 扫描器类的设计

在第3章，字符串处理过程中最复杂的例子就是儿童黑话翻译器。如图 3-2 所示，PigLatin 程序将问题分解成两个部分：lineToPigLatin 函数将输入划分为一个个单词，然后调用 wordToPigLatin 函数将每个单词转换为儿童黑话。但是其第一阶段的单词分解在儿童黑话领域显得并不专业。许多应用需要将字符串分解成单词，更常见的，将其分解成包含多于一个字母的逻辑单元。在计算机科学领域中，这种逻辑单元称为记号（token）。

由于将字符串分解成独立的记号这一问题在各种应用中普遍采用，因此为这一目标单独

建立一个类包是很有必要的。本节将介绍一个用来完成这一任务的 TokenScanner 类。首要任务就是建立一个使用简单、用法灵活并且能满足多种用户需求的类包。

6.4.1 用户想从记号扫描器中得到什么

与往常一样，开始设计 TokenScanner 类的最好方法就是站在用户的角度审视问题。想使用扫描器的每个用户都从一个记号源开始，它们可能是一个字符串或者是应用从文件中读取的数据输入流。无论上述哪种情况，用户所需要的都是以某种方法从这个记号源中抽取出一个个特定的记号。

可以采用多种机制设计 TokenScanner 类，使其能提供必需的功能。例如，可以让记号扫描器返回一个囊括整个记号列表的矢量。但是，这种策略并不适合对大型输入文件的扫描，因为在这种情况下，扫描器必须创建一个单独的矢量来保存整个记号列表。一个更加空间有效的方法是让扫描器一次传送一个记号。当使用这种设计时，扫描器读取记号过程的伪码形式如下：

```
Set the input for the token scanner to be some string or input stream.
while (more tokens are available) {
   Read the next token.
}
```

这段伪码结构直接表明了 TokenScanner 类必须支持的方法。从这个例子中，可能希望 TokenScanner 类提供以下方法：

- 一个允许用户指明记号来源的 setInput 方法。理想情况下，这个方法应该重载，使得输入可以是一个字符串或者是一个输入流。 292
- 一个用于判断扫描器扫描过程中是否还有待扫描记号的 hasMoreTokens 方法。
- 扫描并返回下一个记号的 nextToken 方法。

上述这些方法定义了记号扫描器的操作结构，并且它在很大程度上与特定的应用独立。然而，不同的应用有各种各样的记号，因此，TokenScanner 类必须向用户提供控制来决定扫描器到底识别的是哪种类型的记号。

我们可以很容易地通过一些例子来说明用户期望识别不同类型记号的需求。首先，回顾一下将英语翻译成儿童黑话的问题，这对我们目前的设计极具启发性。如果你使用记号扫描器来重写 PigLatin 程序，那么必须牢记在这一阶段不能忽略空格和标点符号，因为这些字符将成为输出的一部分。在儿童黑话问题中，记号会以下述两种形式之一出现：

1. 一个由字母和数字字符组成用来表示一个单词的连续字符串。
2. 一个包括空格或标点符号在内的单字符字符串。

如果你给记号扫描器以下输入：

```
this is 'pig latin'
```

并多次调用 nextToken，则会返回以下 9 个有序的记号：

```
this | is | ' | pig | | latin | '
```

但是其他应用可能会定义不同类型的记号。例如，C++ 编译器使用记号扫描器将程序划分为许多记号，这些记号包括限定符、常量、操作符和其他定义语法结构的符号。例如，如果你向编译器的记号扫描器输入以下语句：

```
cout << "hello, world" << endl;
```

你会得到以下记号序列：

```
cout << "hello, world" << endl ;
```

在这两个应用领域中，记号的定义略微不同。在儿童黑话转换器中，所有非字母数字的字符序列的元素都被处理成单字符记号返回。在编译器的例子中，情况变得更加复杂。因为编程语言经常定义两个字符组成的操作符，如 << 操作符，这些操作符必须被整体地当成一个记号。与此类似，字符串常量 "hello,world!" 仅当它被记号扫描器当成一个单独的实体才具有明确的意义。也许有点不明显，除非该空格出现在字符串常量中，否则编译器的记号扫描器忽略输入中的所有空格。

293 正如你将在编译课中学到的，创建一个允许用户通过输入精确规则定义合法输入类型的记号扫描器是可能的。这种设计提供了最大的通用性。但是，通用性有时也会降低程序的简洁性。如果你强迫用户为记号信息设置规则，将导致用户需要学习如何正确书写规则，这在很大程度上与学习一种新语言类似。更糟糕的是，记号信息规则的制定对于用户来说是复杂和困难的，特别是如果你试图定义编译器识别规则的数量。

如果你设计接口的目标是使其最简化，将 TokenScanner 类设计成允许用户指明期望在应用中得到的记号类型会更加理想。如果你只需要记号扫描器搜集组成单词的字母数字序列串信息，可以只使用 TokenScanner 类的最简单的配置。例如，如果你想要 TokenScanner 类识别一个 C++ 程序，可以选择性地设置扫描器，使得扫描器忽略空格字符，将引号包括的字符串划分为一个独立单元，并且将特定组合的标点符号看成为多字符的操作符。

6.4.2 tokenscanner.h 接口

C++ Stanford 类库提供了一个 TokenScanner 类，在不牺牲其简单性的前提下，该类提供了很大的灵活性。TokenScanner 类提供的方法如表 6-1 所示，其中接口的大部分方法提供了改变扫描器默认属性的选择。例如，你可以通过将记号扫描器初始化为以下状态使其忽略所有空格字符：

```
TokenScanner scanner;
scanner.ignoreWhitespace();
```

如果你想要初始化 TokenScanner 类使之遵循 C++ 语言的记号规则，你可以采用以下代码：

```
TokenScanner scanner;
scanCPlusPlusTokens(scanner);
```

294 其中，scanCPlusPlusToken 方法的定义如图 6-9 所示。

表 6-1 类库 TokenScanner 类提供的方法

构造函数	
TokenScanner() **TokenScanner(*str*)** **TokenScanner(*infile*)**	初始化一个扫描器对象。记号源来自特定的字符串或者是一个输入文件。如果不提供任何记号源，用户必须在从扫描器中读取记号之前调用 setInput 函数

（续）

读取记号方法	
`hasMoreTokens()`	若读取的输入源中还有剩余的记号，则返回 true
`nextToken()`	返回扫描器扫描的下一个记号。如果在没有可访问记号的情况下调用 nextToken 函数，则返回一个空字符串
`saveToken(token)`	将特定记号存储为扫描器的内部状态，以保证在调用 nextToken 函数时可以返回该标记。库实现允许用户保存任意数量的标记，这些标记会被存储在一个类似栈的存储结构中
控制扫描规则的方法	
`ignoreWhitespace()`	将扫描器设置为忽略空格字符
`ignoreComments()`	将扫描器设置为忽略注释。这些注释可以是 /* 或者是 // 类型的注释
`scanNumbers()`	将扫描器设置为接受合法数字并将其整体当成一个记号。数字的语法与 C++ 中的一样
`scanStrings()`	将扫描器设置为返回一个将双引号括起来的字符串作为一个独立的记号。引号（可能是单引号也可能是双引号）被包括在记号中，使得用户可以区分字符串类型记号和其他类型的记号
`addWordCharacters(str)`	将 *str* 变量中的字符添加到合法的单词中
`addOperator(op)`	定义一个新的多字符操作符。扫描器将返回符合定义的最长的所定义的操作符，并且将至少返回一个字符
其他方法	
`setInput(str)` `setInput(inFile)`	将扫描器的输入源设置为 str 表示的字符串或者 inFile 表示的输入流。扫描之前输入源获得的记号将被清空
`getPosition()`	返回扫描器在当前输入字符串中的位置
`isWordCharacter(ch)`	如果字符 *ch* 与在单词中存在匹配项，则返回 true
`verifyToken(expected)`	读取下一个记号，并检查该记号与字符串 expected 是否匹配
`getTokenType(token)`	返回记号类型，包括：EOF、SEPARATOR、WORD、NUMBER、STRING、OPERATOR

295

```
/*
 * Function: scanCPlusPlusTokens
 * Usage: scanCPlusPlusTokens(scanner);
 * ------------------------------------
 * Sets the necessary options for the scanner so that it can
 * read C++ source code.
 */

void scanCPlusPlusTokens(TokenScanner & scanner) {
   scanner.ignoreWhitespace();
   scanner.ignoreComments();
   scanner.scanNumbers();
   scanner.scanStrings();
   scanner.addWordCharacters("_");
   scanner.addOperator("++");
   scanner.addOperator("--");
   scanner.addOperator("==");
   scanner.addOperator("!=");
   scanner.addOperator("<=");
   scanner.addOperator(">=");
   scanner.addOperator("<<");
   scanner.addOperator(">>");
   scanner.addOperator("&&");
   scanner.addOperator("||");
   scanner.addOperator("+=");
   scanner.addOperator("-=");
```

图 6-9 初始化 TokenScanner 用于扫描 C++ 的记号

```
    scanner.addOperator("*=");
    scanner.addOperator("%=");
    scanner.addOperator("^=");
    scanner.addOperator("&=");
    scanner.addOperator("|=");
    scanner.addOperator("<<=");
    scanner.addOperator(">>=");
    scanner.addOperator("->");
    scanner.addOperator("::");
}
```

图 6-9 （续）

图 6-9 的 scanCPlusPlusTokens 的实现向编译器传递以下信息：必须忽略空格字符和注释，数值与字符串必须被归类为一个单一标记，下划线在标识符中是合法的，并且扫描器必须检测 addOperator 方法中定义的多字符的操作符（有些并不常用，但在 C++ 中有明确定义）。

tokenscanner.h 接口使得编写各种应用变得更为简单，这些应用包括你在本书已看到过的。例如，可以使用它通过重写以下的 lineToPigLatin 函数来简化图 3-2 中的
296 PigLatin 程序：

```
string lineToPigLatin(string line) {
   TokenScanner scanner(line);
   string result = "";
   while (scanner.hasMoreTokens()) {
      string word = scanner.nextToken();
      if (isalpha(word[0])) word = wordToPigLatin(word);
      result += word;
   }
   return result;
}
```

新版本的 lineToPigLatin 函数相比老版本显得更加短小精悍，从概念上讲它已经进行了真正的简化。老版本代码只能在单个字符上进行操作，而新版本可以在整个单词上进行操作，因为 TokenScanner 类已实现了对底层细节的管理。

6.4.3 实现 TokenScanner 类

特别地，在给定该类所支持的有限操作之后，将整个 TokenScanner 类的完整实现作为一个有效的实例将太复杂。图 6-10 和图 6-11 给出了记号扫描器包的简化版本，该版本自定义了以下方法：

- 一个传入字符串参数的构造函数，以及一个默认构造函数。
- 一个 setInput 方法，它将扫描器的输入定义为字符串。
- 一个 nextToken 方法，它返回字符串中的下一个记号。
- 一个 hasMoreTokens 方法，它允许用户查看是否有剩余记号未被扫描。
- 一个 ignoreWhitespace 方法，它将扫描器设置为忽略空格字符。

```
/*
 * File: tokenscanner.h
 * --------------------
 * This file exports a simple TokenScanner class that divides a string
 * into individual logical units called tokens.
 */
```

图 6-10 简化的 TokenScanner 类的接口

```
#ifndef _tokenscanner_h
#define _tokenscanner_h

#include <string>

/*
 * Class: TokenScanner
 * -------------------
 * This class is used to represent a single instance of a scanner.
 * In this simplified version of the class, tokens come in two forms:
 *
 * 1. Strings of consecutive letters and digits representing words
 * 2. One-character strings representing punctuation or separators
 *
 * The use of the TokenScanner class is illustrated by the following code
 * pattern, which reads the tokens in the string variable input:
 *
 *      TokenScanner scanner;
 *      scanner.setInput(input);
 *      while (scanner.hasMoreTokens()) {
 *          string token = scanner.nextToken();
 *          ... process the token ...
 *      }
 *
 * This version of the TokenScanner class includes the ignoreWhitespace
 * method.  The other options available in the library version of the
 * class are included as exercises in the text.
 */

class TokenScanner {

public:

/*
 * Constructor: TokenScanner
 * Usage: TokenScanner scanner;
 *        TokenScanner scanner(str);
 * ---------------------------------
 * Initializes a scanner object.  The initial token stream comes from
 * the string str, if it is specified.  The default constructor creates
 * a scanner with an empty token stream.
 */

   TokenScanner();
   TokenScanner(std::string str);

/*
 * Method: setInput
 * Usage: scanner.setInput(str);
 * -----------------------------
 * Sets the input for this scanner to the specified string.  Any
 * previous input string is discarded.
 */

   void setInput(std::string str);

/*
 * Method: hasMoreTokens
 * Usage: if (scanner.hasMoreTokens()) . . .
 * -----------------------------------------
 * Returns true if there are additional tokens for this scanner to read.
 */

   bool hasMoreTokens();

/*
 * Method: nextToken
 * Usage: token = scanner.nextToken();
 * -----------------------------------
 * Returns the next token from this scanner.  If called when no tokens
 * are available, nextToken returns the empty string.
 */

   std::string nextToken();
```

图 6-10 (续)

```
/*
 * Method: ignoreWhitespace()
 * Usage: scanner.ignoreWhitespace();
 * ----------------------------------
 * Tells the scanner to ignore whitespace characters.  By default, the
 * nextToken method treats whitespace characters (typically spaces and
 * tabs) just like any other punctuation mark and returns them as
 * single-character tokens.  Calling
 *
 *    scanner.ignoreWhitespace();
 *
 * changes this behavior so that the scanner ignores whitespace characters.
 */

   void ignoreWhitespace();

private:

/* Instance variables */

   std::string buffer;          /* The input string containing the tokens */
   int cp;                      /* The current position in the buffer     */
   bool ignoreWhitespaceFlag;   /* Flag set by a call to ignoreWhitespace */

/* Private methods */

   void skipWhitespace();

};

#endif
```

297 ~ 299

图 6-10 （续）

```
/*
 * File: tokenscanner.cpp
 * ----------------------
 * This file implements the TokenScanner class.  Most of the methods
 * are straightforward enough to require no additional documentation.
 */

#include <cctype>
#include <string>
#include "tokenscanner.h"
using namespace std;

TokenScanner::TokenScanner() {
   /* Empty */
}

TokenScanner::TokenScanner(string str) {
   setInput(str);
}

void TokenScanner::setInput(string str) {
   buffer = str;
   cp = 0;
}

bool TokenScanner::hasMoreTokens() {
   if (ignoreWhitespaceFlag) skipWhitespace();
   return cp < buffer.length();
}

/*
 * Implementation notes: nextToken
 * -------------------------------
 * This method starts by looking at the current character, which is
 * indicated by the index cp.  If the index is past the end of the string,
 * nextToken returns the empty string.  If the character is alphanumeric,
 * nextToken scans ahead until it finds the end of a word; if not,
 * nextToken returns the character as a one-character string.
 */

string TokenScanner::nextToken() {
   if (ignoreWhitespaceFlag) skipWhitespace();
```

图 6-11 简化的 TokenScanner 类的实现

```
    if (cp >= buffer.length()) {
       return "";
    } else if (isalnum(buffer[cp])) {
       int start = cp;
       while (cp < buffer.length() && isalnum(buffer[cp])) {
          cp++;
       }
       return buffer.substr(start, cp - start);
    } else {
       return string(1, buffer[cp++]);
    }
}

/*
 * Implementation notes: ignoreWhitespace and skipWhitespace
 * ---------------------------------------------------------
 * The ignoreWhitespace method simply sets a flag.  The private method
 * skipWhitespace is called only if that flag is true
 */

void TokenScanner::ignoreWhitespace() {
   ignoreWhitespaceFlag = true;
}

void TokenScanner::skipWhitespace() {
   while (cp < buffer.length() && isspace(buffer[cp])) {
      cp++;
   }
}
```

图 6-11 （续）

ignoreWhitespace 方法被设计为可在本程序包内得到为其他选项设置的模型，你可以在习题中自己完善所有这些设置的实现。但是，增加从数据文件读取记号的功能依赖第 11 章中引入的一些概念，因此，你必须学完之后章节的内容才能完成 TokenScanner 类的实现。

6.5 将程序封装成类

你在本章所看到的大多数的类定义创建了一种新的抽象类型，你可以像使用基本类型变量一样来使用这种抽象类型。例如，一旦定义了 Rational 类，你可以像使用 C++ 中的其他基本类型一样使用 Rational 类。你可以声明 Rational 类型的变量以保存其值，也可对这些变量赋以新值，使用操作符组合这些变量，在 cout 上输出这些变量的值，并且可将它们存储在各种集合类对象中。使用了有理数类的程序将会在运行时创建许多 Rational 对象，所有这些对象都是同一个 Rational 类的实例。

然而，当你不需要为一个特定的类创建一个以上的对象时，类仍然在程序中扮演着重要作用。例如，以类来组织程序比使用自由函数集合来构建程序更加有效。这样做的最大优点就是类提供了更好的封装性。这种封装性使得对类私有实例变量的访问只能限制于类内，这意味着使用私有实例变量要比使用全局变量共享信息具有更高的安全性。

作为这一技术的说明，图 6-12 所示的程序向我们展示了如何将图 5-5 中的收银台排队模拟程序重新设计为一个类。在这一新的设计中，自由函数 runSimulation 和 printReport 被重新设计为 CheckoutLineSimulation 类的公有方法。这些方法的实现与原函数体相同，这一改变仅对代码的复杂性产生了一点细微的影响。但是这一改变的好处是：程序运行时，方法之间的共享信息现在可通过实例变量来存储，而无需通过参数传递。类中的所有方法对其数据访问权的共享大幅度降低了其方法参数列表的规模和复杂性，在这个例子中，方法形参的数量从三个缩减至零。

300 ↓ 301

```
/*
 * File: CheckoutLineClass.cpp
 * ---------------------------
 * This program duplicates the CheckoutLine program from Chapter 5,
 * but embeds the entire program in a class definition.
 */

#include <iostream>
#include <iomanip>
#include "queue.h"
#include "random.h"
using namespace std;

/* Constants */

const double ARRIVAL_PROBABILITY = 0.05;
const int MIN_SERVICE_TIME =  5;
const int MAX_SERVICE_TIME = 15;
const int SIMULATION_TIME = 2000;

/*
 * Class: CheckoutLineSimulation
 * -----------------------------
 * This class encapsulates the code and data for the simulation.
 */

class CheckoutLineSimulation {

public:

   void runSimulation() {
      ... same as in Figure 5.5 ...
   }

   void printReport() {
      ... same as in Figure 5.2 ...
   }

private:
   int nServed;          /* Number of customers served                */
   int totalWait;        /* Sum of all customer waiting times         */
   int totalLength;      /* Sum of the queue length at each time step */

};

/* Main program */

int main() {
   CheckoutLineSimulation simulation;
   simulation.runSimulation();
   simulation.printReport();
   return 0;
}
```

图 6-12 基于类的收银台排队模拟程序版本

当你使用上述方法实现程序时，将使 main 函数变得更加简短。main 函数声明了一个封装了程序操作过程的类对象，然后调用其公有方法来运行程序，最后的 main 函数定义如图 6-12 所示：

```
int main() {
   CheckoutLineSimulation simulation;
   simulation.runSimulation();
   simulation.printReport();
   return 0;
}
```

随着程序复杂性的增大，使用类来组织程序的优点将更加明显。本书只在使用这一技术可以简化代码的情况下才会采用这一设计理念。

本章小结

本章的主要目的就是介绍在设计和实现自定义类型时需要的若干工具。本章的示例重点强调了类将数据和其操作封装为一个整体的特性，我们将在第 19 章介绍更加复杂的类继承特性。

本章的要点包括：

- 在大多数应用中，将许多独立数据值组合成一个单独的抽象数据类型是很有用的。C++ 为这种数据封装提供了多种机制。在最底层，C++ 延续了对 C 语言风格数据结构定义的支持。但是在现代编程实践中，我们更多的是使用类来实现数据的封装。
- 在 C++ 中，一个类被划分为不同的部分以实现用户对该部分数据域和方法的访问控制。类的公有部分对所有类型的用户开放；类的私有部分仅对其实现开放。在现代面向对象编程风格中，实例变量被声明为类的私有部分。一个类可以通过将其他函数和类声明为该类的友元（friend）来赋予它们访问类中私有数据的权限。 302 : 303
- 给定一个要么是一个类要么是一个结构类型的混合对象，你可以通过使用点操作符来选择该对象所包含的独立成员。用户只能选择该对象中声明为公有部分的数据域。但是，类的实现代码可访问该类所有对象的私有成员。
- 典型地，类的定义可导出一个或多个负责初始化该类对象的构造函数。通常，所有的类定义中都包含一个不接收任何参数的默认构造函数。
- 允许用户访问实例变量数值的方法被称为读取器；允许用户改变实例变量数值的方法称为设置器。拒绝用户在创建对象后改变对象数值的类是不可变的。
- 典型地，类的定义可实现其接口与实现的分离。其中，.h 文件只包含类中的方法原型；而其方法体包含在 .cpp 文件中。在 C++ 语言中，类的实现细节包含在文件 .cpp 中。它必须通过在方法名前增加类名和 :: 标签来表明该方法属于哪个类。
- 类定义中可以通过两种方式来重载基本操作符。将一个操作符重载定义为类的方法意味着该操作符是类的一部分，因此可以自由访问类中的私有成员。将一个操作符重载定义成自由函数使代码变得简单易读，但也意味着操作符函数必须被定义成类的友元函数，这样操作符才能访问类中的私有成员。
- 需要重载的一个最有用的操作符就是插入操作符 <<，因为这样做可以很容易将类的值显示在控制台上。本书大多数的类都重载了 << 操作符，并且定义了 toString 方法，它将类中的数值转换为对应的字符串。
- 设计一个新类既是一门科学也是一种艺术。尽管本章提供了设计类的一些通用的指南，但是必须牢记实践和练习才是最好的老师。
- Stanford 类库提供了 TokenScanner 类，它支持将输入文本分解成多个独立的被称为记号的一个个单元。TokenScanner 类库版本提供了多种选择设置，使得该类包应用范围更加广泛。
- 对于太过复杂，需要维护的内部变量超过接受限度的应用，将这个程序封装成一个类将是很好的选择。当你使用这种设计风格时，main 程序将创建该类的一个对象，然后调用类中的方法使得程序正常运行。 304

复习题

1. 给出以下术语的定义：对象、结构类型、类、实例变量和方法。

2. 在C++的类定义中，关键字public和private的含义是什么？
3. 判断题：在C++语言中，关键字struct和class之间的唯一区别就是struct的默认数据域是公有的。
4. C++语言使用什么操作符从对象中选择一个实例变量？
5. C++语言构造函数的语法是怎样的？
6. 调用表达式或函数需要传入多少个参数？
7. 什么是读取器？什么是设置器？
8. 一个类被称为不可变的是什么意思？
9. 当你将类的接口和实现分离时，类的实现如何让编译器知道一个特定的方法定义属于哪个类？
10. 本章为了防止用户访问类中私有部分内容，在.h文件中采用了哪种机制？
11. 在C++语言中，你会使用何种方法名来重载%操作符？
12. C++语言怎样区别++和--操作符的前缀版本和后缀版本？
13. 为什么重载<<操作符的实现部分需要返回变量的引用？
14. 判断题：在C++语言中，引用返回和引用调用的使用频率一样。
15. 叙述基于方法和基于自由函数这两种类操作符重载方法的不同点。这两种风格各自的优缺点是什么？

305

16. 在一个类中声明一个方法或者其他类声明为该类的友元意味着什么？
17. 本章为Direction类型提供++操作符重载的原因是什么？
18. 本章中提出关于设计一个类的五个步骤是什么？
19. 什么是有理数？
20. Rational类构造函数为num和den变量的值设置了什么约束？
21. 图6-8中的Rational类构造函数代码包括了对x是否为零的检查。如果这一检查被去除，Rational类还能不能正常工作？
22. 在图6-7中的rational.h文件中，为何有必要将操作符函数+、-、*和/声明为友元函数，但是插入操作符<<函数却不能进行类似声明？
23. 什么是记号？
24. 从一个字符串中读取所有记号的标准模式是什么？
25. 怎样初始化TokenScanner对象使得该对象忽略输入中的空格、制表符和其他空白字符？
26. 用你自己的语言解释将程序嵌入到一个类中的技术。

习题

1. **多米诺骨牌**（dominos）游戏通常使用在两个表面标示代表数值白点的黑色长方形骨牌。如下图所示：



上图所示的骨牌称为4-1骨牌，该牌的左边部分有四个点，右边部分有一个点。

定义一个简单的Domino类来代表传统的多米诺骨牌。你的类必须提供如下功能：

- 一个创建0-0骨牌的默认构造函数
- 一个需要传入骨牌各边数值参数的构造函数
- 一个输出代表骨牌字符串的toString方法

306

- 两个名称分别为getLeftDots和getRightDots的访问器方法

编写一个domino.h接口和一个domino.cpp实现来提供该类。参照书本中的例子，所有实例变量必须为私有，并且接口必须重载<<操作符以输出代表多米诺骨牌的字符串。

编写一个创建从 0-0 到 6-6 一整套骨牌的程序来测试 Domino 类的实现，并且将这些骨牌都显示在控制台中。一套完整的多米诺骨牌需要每个骨牌样式的一张牌，不允许通过翻转产生重复的骨牌。因此，多米诺骨牌堆中存在 4-1 骨牌则不能存在 1-4 骨牌。

2. 定义一个 Card 类来表示标准扑克牌，牌面通过两个组成因素进行区别：等级（rank）和花色（suit）。等级以整数 1 到 13 的形式存储，其中 A 是 1，J 是 11，Q 是 12，K 是 13。花色是以下枚举类型中的四个常量之一：

```
enum Suit = { CLUBS, DIAMONDS, HEARTS, SPADES };
```

Card 类必须提供以下方法：

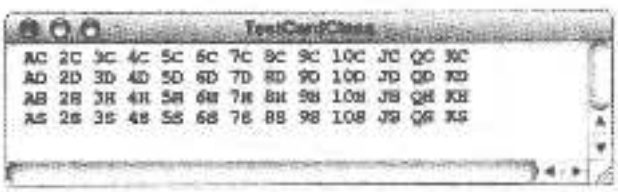
- 一个创建一张扑克牌并允许在随后的程序中对其进行赋值的默认构造函数
- 一个接受传入类似 "10s" 或 "2D" 等短字符串的构造函数
- 一个分别传入等级和花色参数的构造函数
- 一个返回代表扑克牌牌面的短字符串的方法 toString
- 访问器方法 getRank 和 getSuit

编写 card.h 接口和 card.cpp 实现来提供 Card 类。除 Card 类之外，card.h 接口还必须提供 Suit 类型，花色常量名（ACE,JACK,QUEEN,KING）在实际中将比数值更常用，并且你必须运行如下 main 程序进行测试：

```
int main() {
   for (Suit suit = CLUBS; suit <= SPADES; suit++) {
      for (int rank = ACE; rank <= KING; rank++) {
         cout << " " << Card(rank, suit);
      }
      cout << endl;
   }
   return 0;
}
```

307

你的程序必须产生如下输出：



3. gtypes.h 接口提供了多个实用的可以与图形类库一同工作的类。在这些类中，最简单的是 GPoint 类，它与本章介绍的 Point 类相似，不同点只是在运行时使用了浮点数来表示坐标而非使用整数来表示坐标。另一个实用的类是 GRectangle 类，它使用 x 和 y 坐标表示一个长方形区域的左上角顶点，并用 width 和 height 变量来表示长方形的长和宽。以在线文档介绍的 GRectangle 类为参考，实现 GRectangle 类。

4. 上一个习题中介绍了 gtypes.h 接口中提供的类，使得创建复杂图像的模式变得更加简单，其中，部分原因是因为这些类的实现使得在集合类和其他抽象数据类型中存储坐标信息的过程变得更加简单。例如在本题中，你将体会到 GPoint 对象构成的矢量带来的乐趣。想象一下你在一个矩形边界中插入大头针，并且要求这些大头针必须在四条边上方的空间中均匀分布，其中上部和下部边界上的大头针称为 N_ACROSS，左部和右部边界上的称为 N_DOWN。为了使用图形窗口对这一过程建模，你需要创建一个 Vector<GPoint> 来保存每一个大头针的坐标，这些点从左上角的点开始，并且依照顺时针方向环绕矩形一周，如下图所示：



从这里开始，你通过在大头针之间连线勾画出了矩形的轮廓，连线从0号大头针开始，然后依照数值顺序间中间次移动一定距离的空间，这一距离在常量DELTA中定义。例如，如果DELTA的
308 值为11，第一条线是从0号大头针到11号，第二条线从11号到22号。第二条线需按顺时针方向数过11个大头针经过起始点，从22号移动到5号。这一过程一直持续，直到划线返回到0号大头针。像往常一样，使用%操作符可以使环绕特性的实现变得非常简单。

编写一个程序，在图像窗口中使用更大的N_ACROSS和N_DOWN数值来模拟这一过程。例如，N_ACROSS的值为50，N_DOWN的值为30，并且DELTA的值为67的时候，程序的输出如图6-13所示。通过改变这些常量值，你可以创建其他更奇妙的只有直线构成的图画。

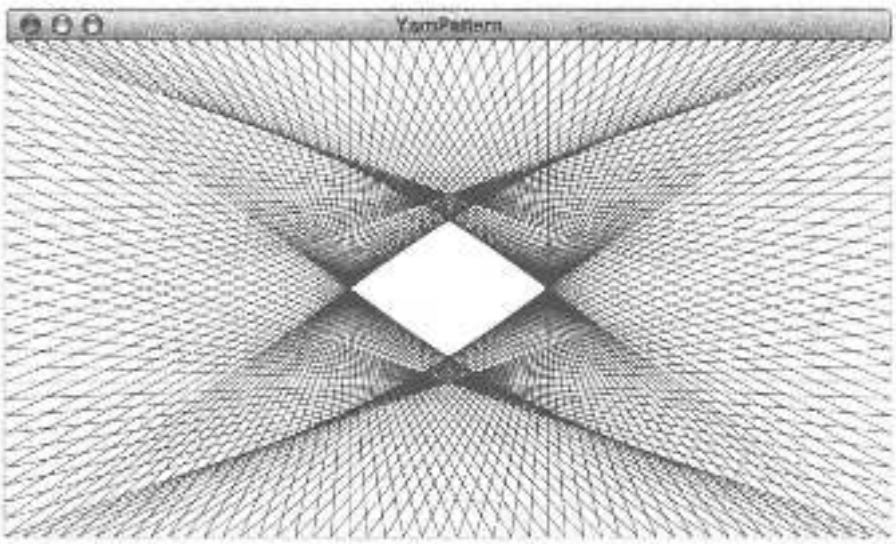


图6-13 yarn_pattern程序的运行实例

5. 扩展第2章习题11中的calendar.h接口，使得该Date类提供以下方法：

- 一个将日期设为1900年1月1日的默认构造函数。
- 一个传入月、日和年并将Date对象初始化为包含这些值的构造函数。该函数的调用例子如下：

```
Date moonLanding(JULY, 20, 1969);
```

309 该语句将创建一个moonLanding变量，该变量表示1969年7月20日。

- 一个重载的构造函数，它将传入的前两个参数为了方便不同地区的用户使用而颠倒了顺序。这一改变使得moonLanding变量的声明可以被写成：

```
Date moonLanding(20, JULY, 1969);
```

- 访问器方法getDay、getMonth和getYear。
- 一个以dd-mmm-yyyy格式返回时间的toString方法，其中dd是一个一位或者两位数的日期，mmm是一个三字母的英文月份简写，yyyy是一个四位数的年份。因此，调用toString

(moonLanding) 会返回字符串 "20-Jul-1969"。

6. 通过添加以下重载操作符来扩展 calendar.h 接口：
   - 插入操作符 <<
   - 关系操作符 ==、!=、<、<=、> 和 >=
   - 表达式 date+n 将返回 date 之后 n 天的日期
   - 表达式 date-n 将返回 date 之前 n 天的日期
   - 表达式 $d_1 - d_2$ 将返回 d_1 和 d_2 之间相差的天数
   - 复合赋值操作符 -= 和 --，该操作符右边传入一个整型数
   - ++ 和 -- 操作符的前缀和后缀形式

   例如，假设你已经定义了如下变量：

```
Date electionDay(6, NOVEMBER, 2012);
Date inaugurationDay(21, JANUARY, 2013);
```

则 electionDay<inaugurationDay 表达式的值为 true，因为 electionDay 日期在 inauguration Day 之后。计算 inaugurationDay-electionDay 将返回值 76，这是这两天之间的日期间隔天数。除此之外，这些操作符的定义允许你编写如下 for 循环：

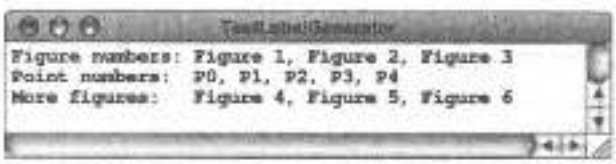
```
for (Date d = electionDay; d <= inaugurationDay; d++)
```

这一循环将遍历两个日期中的每一天，包括这两个日期本身。

7. 对于大多数应用来说，产生一系列以特定顺序组合起来的名称是非常有用的。例如，如果你编写一个程序对纸上的图进行编号，使得机器自动返回有序字符串："Figure1"、"Figure2"、"Figure3" 等等，这一功能将非常方便。除此之外，你可能需要标记几何图形中的点，使得标记 310
之间相似但是可以加以区别，比如 "P0"、"P1"、"P2" 等等。

   如果让这个问题变得更加广泛，你需要一个标记产生工具，该工具允许用户定义任意序列标记，每一个标记由前部（比如之前例子中的 "Figure" 或 "P"）和一个按序排列的整型数组成。因为用户可能会需要不同的序列同时被激活，所以将标记产生器定义为 LabelGenerator 类将非常有用。为了初始化一个新的标记产生器，用户需要提供标记前缀字符串和初始化的下标值作为参数传递给构造函数 LabelGenerator。在产生器创建后，用户可以通过调用 LabelGenerator 类的 nextLabel 方法返回新的标记序列。

   为了说明接口如何工作，图 6-14 中的 main 程序运行结果如下图所示：



   编写文件 labelgen.h 和 labelgen.cpp 来实现该类。

```
int main() {
   LabelGenerator figureNumbers("Figure ", 1);
   LabelGenerator pointNumbers("P", 0);
   cout << "Figure numbers: ";
   for (int i = 0; i < 3; i++) {
      if (i > 0) cout << ", ";
      cout << figureNumbers.nextLabel();
   }
   cout << endl << "Point numbers:  ";
   for (int i = 0; i < 5; i++) {
      if (i > 0) cout << ", ";
```

图 6-14 测试标记产生器的程序

```
      cout << pointNumbers.nextLabel();
   }
   cout << endl << "More figures:  ";
   for (int i = 0; i < 3; i++) {
      if (i > 0) cout << ", ";
      cout << figureNumbers.nextLabel();
   }
   cout << endl;
   return 0;
}
```

311

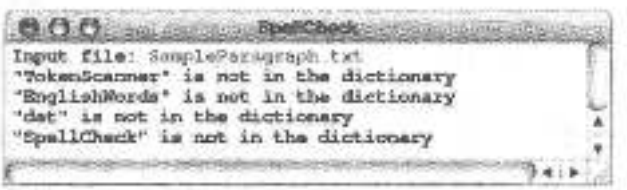
图 6-14 （续）

8. 本书中的 Rational 类定义了操作符 +、-、* 和 /，但是仍然需要定义其他操作符来完善该类。请在接口和实现中增加以下操作符：
   - 关系类操作符 ==、!=、<、<=、> 和 >=
   - 复合赋值操作符 +=、-=、*= 和 /=
   - ++ 和 -- 操作符的前缀和后缀形式

9. 重新实现图 5-4 中的 RPN 计算器，使得该计算器内部使用有理数进行计算而不是浮点数。例如，你的程序运行时应能产生以下输出（该例子演示了有理数运算的精确性）：



10. 编写一个程序，检查文件中单词的拼写。你的程序必须使用 TokenScanner 类从输入文件中读取记号，然后再依照第 5 章介绍的 EnglishWords.dat 文件存储的字典对每个单词进行检查。如果单词不在字典中，程序必须输出这一检查结果。例如，如果运行程序并输入包括以下这段话的文本，SpellCheck 程序将产生以下输出：



312

11. 编写一个程序，实现一个简单的计算器。计算器的输入包括了由数字（整型数或者实数）和四则运算操作符 +、-、* 和 / 组成的表达式。针对每一行输入，程序都必须显示从左到右运算所产生的结果。你应该使用记号扫描器来读取术语和操作符，同时让扫描器忽略所有的空白字符。程序必须在用户输入一个空行后退出。运行该程序的一个实例如下：



上述实例中的最后一行的四则运算问题是由数学家诺顿·贾斯特（Norton Juster）写的儿童故事《神奇的收费亭》（*The Phantom Toll booth*）中 Mathemagician 给 Milo 出的问题。

12. 扩展之前习题中的程序，使得表达式中可以出现变量名，该变量可像下图所示一样通过简单的赋值表达式进行赋值：



13. 为 TokenScanner 类实现 saveToken 方法。这一方法存储特定的记号使得随后对 nextToken 方法的调用直接返回存储的记号，而不用再次对输入中的字符进行扫描。你的实现必须允许用户存储多个标记，其中最后存储的记号将会被第一个返回。

14. 为 TokenScanner 类实现 scanStrings 方法。但调用 scanStrings 方法时，记号扫描器将把引号当成一个单独的字符返回。字符串有可能通过单引号或者双引号进行标示，并且 nextToken 方法必须将引号与字符串一同返回。 313

15. 为 TokenScanner 类实现 scanNumbers 方法。该方法将符合 C++ 语言标准的数值作为一个单独的记号返回。这一扩展的难点在于理解什么是合法的数值字符串，并且寻找一种高效实现这些规则的方法。说明这些规则的最简单方式就是使用计算机科学中的有限状态机（finite-state machine），它通常被表示成图，其中使用圆圈来表示有限状态机的所有可能状态。之后这些圆圈将被一组标示箭头进行连接，这些箭头表明了该状态机从一种状态转换到另一种状态的过程。图 6-15 展示了扫描一个实数的有限状态机。

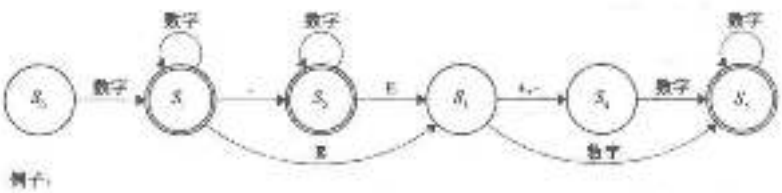


图 6-15 扫描数字的有限状态机

当你使用一个有限状态机时，你将从状态 S_0 开始，然后遵循标记箭头来处理输入的每一个字符直到没有可以匹配输入字符的箭头时为止。如果你在一个双线圈处停止，那么你已经成功地扫描到了一个数值。这表明扫描成功的状态被称为终态（final state）。图 6-15 展示了有限状态机在扫描各种类型的数字的一个实例。

编写 scanNumbers 代码来扫描记号的最简单方法就是模拟有限状态机的运行过程。你的代码必须保持对有限状态机中状态的追踪，然后每次验证输入的一个字符。每一个字符都可能标示
314 若数值的结尾或者该状态机进入一个新的状态。

第7章
Programming Abstractions in C++

递归简介

通常来说，对梦想的坚定不移让我们梦想成真。

——威廉·詹姆斯，《信仰的意志》(*The Will to Believe*)，1897 [315]

大多数用于解决程序问题的算法策略在计算领域之外都有与之对应的事物。当你重复执行一个任务时，你就在使用迭代。当你做一个决定时，你正运用条件控制。由于这些都是人们熟悉的操作，大多数人在遇到相关的小问题时学会了使用 `for`、`while` 以及 `if` 这样的控制语句。

然而，在处理一些复杂的程序任务之前，必须学习一种强大的问题处理策略，它在现实世界中很少有直接对应的事物。这种策略被称为递归（recursion），它被定义为将大问题通过简化成相同形式的小问题来解决问题的一种技术。在递归定义中，“相同形式”这个词是最重要的，它从其他方面描述了逐步求精法的基本策略。递归和逐步求精这两种策略都涉及分解，但递归的特殊之处在于：在其解决方案中，子问题和原问题具有相同的形式。

如果你和大多数初级程序员一样，那么当你第一次听到将一个问题分解成相同形式的子问题的思想时，你可能觉得它意义不大。不像重复或者条件检验，递归不是日常生活中出现的一个概念。正因为它不常见，因此学习如何使用递归可能很困难。为此，你必须培养必要的直觉，这种直觉能够让递归看起来和其他所有的控制结构一样自然。对于大多数正在学习编程的学生而言，理解递归需要大量的时间和练习。即便如此，努力学习使用递归无疑是值得的。作为一种解决问题的工具，递归非常强大，以至于它有时看起来近乎神奇。另外，经常使用递归使得用极其简洁的方式编写一个复杂的程序成为可能。

7.1 一个简单的递归例子

为了更好地理解递归，想象一下：假如你已经被任命为一个大慈善组织的资金协调员，这个组织有很多志愿者，但是缺乏资金。你的工作就是筹集 1 000 000 美元以满足组织的开支。

如果你认识一些愿意捐献 1 000 000 美元的人，你的工作就很简单。但是，你可能不会太幸运的拥有那些慷慨且为百万富翁的朋友。此时，你必须一小笔一小笔地来筹集这 1 000 000 美元。如果平均捐款为 100 美元，你可能会选择一个不同的行动方针：给 10 000 个朋友打电话，请求他们每人捐款 100 美元。但话说回来，你可能没有 10 000 个朋友，那么你该怎么办呢？

最常见的情况是，当你面临一个超出你能力的任务，这个问题的答案就是将你的部分工作分派给其他人。你的组织有相当多的志愿者。如果你能够在这个国家的不同 [316] 地区发现 10 个志愿者，然后任命他们为地区协调员，这 10 个人每人负责筹集 100 000

美元。

筹集100 000美元比筹集1 000 000美元简单，但这也并非易事。你任命的地区协调员应该做什么？如果他们采取相同的策略，他们进而将部分工作分派给其他人。如果他们每个人招募10名资金筹集志愿者，这些志愿者每人只需要筹集10 000美元。这个分派任务的过程可以一直继续下去，直到志愿者能够自己筹集到所分派的钱。由于平均捐款为100美元，资金筹集志愿者可以从每个捐款者那里筹集100美元，这就无须再分派任务。

如果你用伪码表示这个筹集资金的策略，它具有以下结构：

```
void collectContributions(int n) {
   if (n <= 100) {
      Collect the money from a single donor
   } else {
      Find 10 volunteers.
      Get each volunteer to collect n/10 dollars.
      Combine the money raised by the volunteers.
   }
}
```

这段伪码翻译时最重要的就是以下这行：

```
Get each volunteer to collect n/10 dollars
```

它只是简单地将原始问题的规模减小。任务的基本特征（筹集n美元）问题依然和原来完全一样；唯一不同的是n的值变小了。此外，由于问题是相同的，你可以调用原始的函数来求解它。因此，伪码中前面一行最终可以用下面这行替换：

```
collectContributions(n / 10);
```

如果平均捐款数大于100美元，注意collectContributions函数调用自身的结束条件是非常重要的。在程序中，一个函数直接或间接地调用自身正是所定义的递归函数的重要特点。

collectContributions函数的结构是典型的递归函数。通常，递归函数体都具有
317 如下形式：

```
if (test for simple case) {
   Compute a simple solution without using recursion
} else {
   Break the problem down into subproblems of the same form
   Solve each of the subproblems by calling this function recursively
   Reassemble the subproblem solutions into a solution for the whole
}
```

这个结构提供了编写递归函数的模板，因此被称为递归范型（recursive paradigm）。你可以将这种技术运用到编程问题中，只要该问题符合以下条件：

1. 你一定能够识别那些答案很容易被确定的简单情况（simple case）。

2. 你一定能够确定一个递归分解（recursive decomposition），这个递归分解可让你将任何复杂问题分解成相同形式的更简单的问题。

collectContributions这个例子说明了递归的强大。和任何递归技术一样，原始问题通过分解成为更小的子问题来解决，子问题只是在规模上与原问题有差别，这里，原始问题是要筹集到1 000 000美元。在第一级的分解中，每一个子问题是要筹集到100 000美

元。然后，这些问题进而被细分为更小的问题，直到问题足够简单，甚至不需要再细分就能解决为止。由于该解决方法依赖于将复杂问题分解成相同形式的更简单的问题，因此，这种递归形式的解决方法通常被称为分治（divide-and-conquer）算法。

7.2 阶乘函数

尽管 collectContributions 这个例子说明了递归的思想，但是它并没有揭示递归在实际中是如何使用的，这大部分原因是组成解决方法的步骤，例如招募 10 名志愿者然后筹钱，都不能在 C++ 程序中简单地表示出来。为了得到对递归性质的一个实际理解，你需要思考那些更容易适用于编程领域的问题。

对于大多数人来说，理解递归最好的方法就是从简单的数学函数开始。其中，递归的结构直接伴随着问题的描述出现而很容易被理解。其中，最常见的是阶乘函数（数学上习惯表示为 n!），它被定义为在 1 到 n 之间的所有整数的乘积。在 C++ 中，与之等价的问题是编写具有如下原型的函数：

```
int fact(int n);
```

该函数从参数中获取一个整数 n，并返回其阶乘结果。

正如你可能以你的编程经验中所了解的那样，使用 for 循环编写 fact 函数非常简单，正如以下实现代码所展示的：

```
int fact(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

该实现代码使用了一个 for 循环来循环遍历 1 到 n 之间的每个整数。而在递归实现中，并不存在这个循环。取而代之的是通过直接递归调用以产生相同的结果。

使用循环（典型的通过使用 for 和 while 语句）的实现方法被称为迭代（iterative）。迭代和递归通常被看成是完全相反的两种策略，因为它们用完全不同的方法来解决相同的问题。然而，这两种策略并非相互排斥。递归函数内部有时候也包含迭代。尽管本章的例子纯粹全部采用递归，但是你将会在第 8 章见到这种技术的例子。

7.2.1 fact 的递归公式

然而，fact 的迭代实现并没有利用阶乘的一个重要的数学性质。每一个阶乘都与下一个更小的整数的阶乘相关，如下所示：

$$n! = n \times (n-1)!$$

因此，4! 是 4×3!，3! 是 3×2!，以此类推。为了确保阶乘计算过程在某处终止，数学上定义了 0! 为 1。因此，阶乘函数的传统数学定义如下：

$$n! = \begin{cases} 1 & \text{若 } n = 0 \\ n \times (n-1)! & \text{其他} \end{cases}$$

这个定义是递归的，因为它根据n－1的阶乘定义了n的阶乘。新的问题（计算n－1的阶乘）和原始的问题有同样的形式，这种形式是递归的基本特征。你之后可以使用相同的过程
319 根据（n－2）！来定义（n－1）！。此外，你可以一步一步地向前递推这个过程，直至解决方案被表达成0！，根据定义0！等于1。

从程序员的观点来看，递归数学定义的实际影响是，它为其实现方法提供了一个模板。用C++你可以实现一个如下的计算其参数的阶乘函数fact：

```
int fact(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * fact(n - 1);
   }
}
```

如果n等于0，fact函数的结果为1。如果n不等于0，该实现通过调用fact（n－1），然后将这个结果乘以n来计算结果。该实现直接遵循了阶乘函数的数学定义，并且恰好具有递归结构。

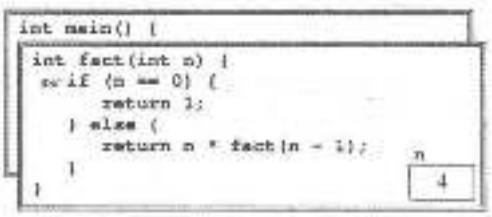
7.2.2 追踪递归过程

如果根据递归的数学定义，编写fact的递归实现会很简单。然而，尽管依据定义易于编写出代码，但过于简短的答案似乎令人怀疑。当你第一次学习递归时，fact的递归实现看起来遗漏了某些东西。即使它清楚地反映了数学定义，递归公式使你难以确定实际的计算步骤。例如，当你调用fact函数时，你想要计算机给出答案。在这个递归实现中，你能看到的只是一个公式，它将一个fact调用转化成另一个fact调用。由于计算步骤不明显，因此当计算机给出正确答案时，它看起来有点神奇。

然而，如果你跟踪计算机的函数调用逻辑，你会发现其中没有涉及任何魔法。当计算机计算一个递归函数fact的调用时，和计算其他任何函数调用一样经历了相同的过程。为了使这个过程可视化，假设你已经执行了main函数中的以下这条语句：

```
cout << "fact(4) = " << fact(4) << endl;
```

320 当main函数调用fact函数时，计算机创建了一个新的栈帧并将实参值复制给形参变量n。fact函数的栈帧暂时取代了main函数的栈帧，如下图所示：



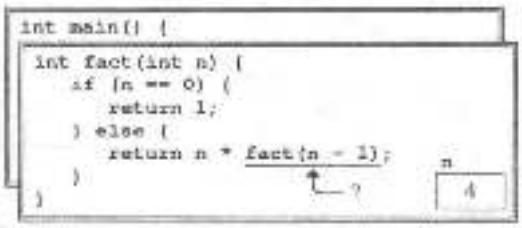
图中栈帧内部显示了fact函数体代码，它使你很容易跟踪程序中的当前位置。图中当前位置指示符出现在代码的开始处，因为所有的函数调用都从函数体的第一条语句开始。

现在，从if语句开始，计算机开始执行函数体。由于n不等于0，控制行进至else

分句，此时程序必须计算并返回以下表达式的值：

```
n * fact(n - 1)
```

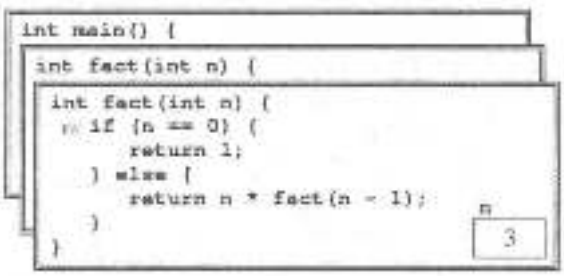
计算该表达式需要计算 fact(n-1) 的值，它引入了一个递归调用。当该调用返回时，程序必须将得到的结果乘以 n。因此，当前的计算状态可用下图表示：



一旦调用 fact(n-1) 返回值，其结果将取代图中划线的表达式，以便计算过程继续进行下去。

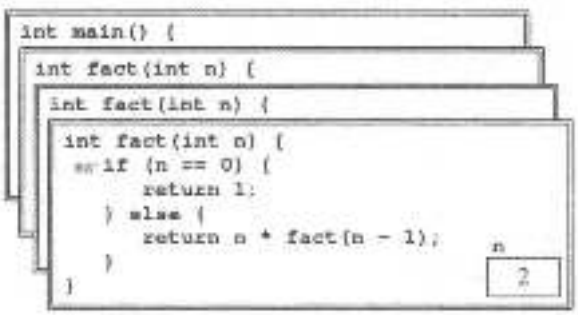
计算的下一步为进行函数调用 fact(n-1)，首先计算该函数的参数表达式。由于 n 的当前值为 4，则参数表达式 n-1 的值为 3。之后计算机为 fact 函数创建了一个新的栈帧，其中，形参变量被初始化为 3。因此，下一栈帧看起来如下图所示：

321

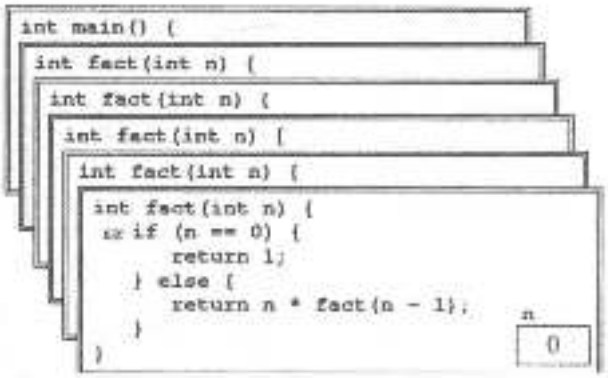


现在，这里有两个被标记为 fact 的栈帧。在最近的栈帧中，计算机正开始计算 fact(3)。这个新的栈帧覆盖了之前的 fact(4) 栈帧，直到 fact(3) 计算完成才会再次出现 fact(4) 栈帧。

以测试 n 的值开始再次计算 fact(3)。由于 n 仍不为 0，else 分句通知计算机计算 fact(n-1)。和之前一样，这个过程需要创建一个新的栈帧，如下图所示：



遵循同样的逻辑，现在，程序必须调用 fact(1)，继而调用 fact(0)，创建两个新的栈帧，如下图所示：

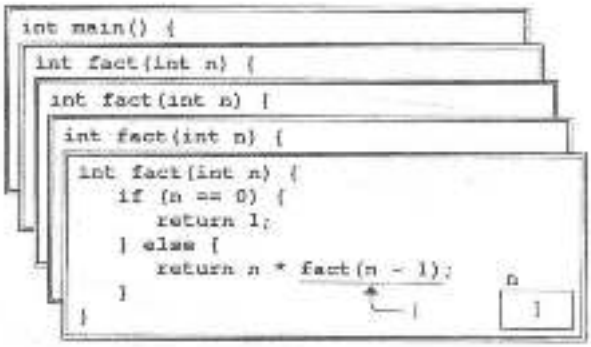


322

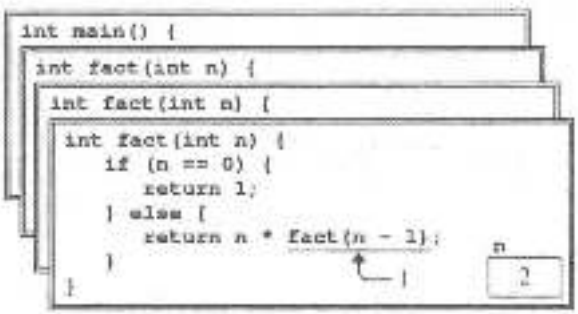
然而，此时情况发生了改变。由于n的值为0，函数将通过执行以下语句立即返回结果：

```
return 1;
```

值1被返回到调用栈帧，这将恢复它在上层栈帧中的位置，如下图所示：

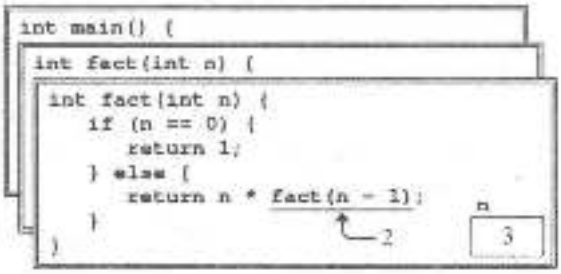


依据上述方式，通过每次递归调用返回使计算执行下去，即通过返回值来完成每一层的计算。例如，在当前栈帧中，调用fact(n-1)能被值1替换，正如关于栈帧的图所示。在这个栈帧中，n的值为1，因此该调用的结果就是1。该结果被返回给它的调用者并被下图中的上一层栈帧替换。

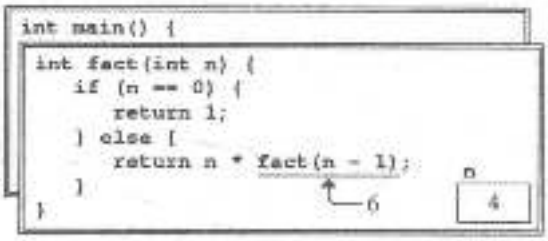


由于n现在为2，return语句的执行结果为2，并且该值将会被返回给前一级，如下图

323 所示：



在此阶段，程序将3×2返回给它的前一级。因此，初始调用 fact 的栈帧如下图所示：



计算过程的最后一步包括计算4×6以及将值24返回给主程序。

7.2.3 递归的稳步跳跃

包含 fact(4) 计算的完整跟踪过程的意义使你确信：计算机将递归函数与其他函数同等对待。当你面对一个递归函数时，至少在理论上，你可以模拟计算机的操作，并且弄明白它将做什么。通过画出所有栈帧和跟踪所有的变量值，你可以复制出完整的操作并给出答案。然而，如果你这样做，你会经常发现其过程的复杂性会以计算过程无法继续跟踪而结束。

每当你试图理解一个递归程序时，将基本细节隐藏，取而代之的是应集中于某个层次上的操作是非常有用的。在那个层次上，你可以假设：只要那层调用的参数在某种意义上比原始的参数简单，所有的递归调用都能自动得到正确的答案。这种心理上的策略（假设任何更简单的递归调用将正确地工作）被称为**递归的稳步跳跃**（recursive leap of faith）。在实际应用中使用递归时，学会运用这种策略是非常重要的。 124

例如，考虑一下：当n的值为4时，计算 fact(n) 会发生什么？为此，递归实现必须计算以下表达式的值：

```
n * fact(n - 1)
```

将n代入到上述表达式，则得到：

```
4 * fact(3)
```

此时，计算 fact(3) 比计算 fact(4) 简单。因此，递归的稳步跳跃允许你假设它已起作用了。因此，你可以假设调用 fact(3) 能正确地计算出3！的值，它是3×2×1或者6。因此，调用 fact(4) 的结果是4×6或者24。

正如你在本章剩余部分的例子中所看到的，总是尝试专注于大局而非细节。一旦你已经完成了递归分解并且确定了简单情况，则相信计算机可以处理剩余的部分。

7.3 斐波那契函数

在1202年出版的《算法之书》(*Liber Abbaci*)中的一个数学模型中，意大利数学家列奥纳多·斐波那契提出了一个在很多领域，包括计算机科学领域都具有重要影响的一个问题。该问题作为人口生物学（最近几年正逐渐变得十分重要的领域）中的一个例子首次被提出。斐波那契的问题是关于兔子的总数是如何一代一代地增长的，如果兔子生育是根据以下公认的规则进行：

- 每一对成年的兔子每个月生产一对新的兔子。
- 兔子在生下来的两个月之后变成成年兔子。
- 老的兔子永远不会死亡。

假设在某年的一月有一对新出生的兔子，当这年结束时共有多少对兔子？

通过在这一年每个月记录下兔子的总数目，你可以简单地解决这个斐波那契问题。在一月初，没有兔子，由于在这个月的某个时候第一对兔子刚被引进，这导致了二月一号只有一对兔子。由于初始的这对兔子是新出生的，它们在二月的时候还没有成年，这意味着在三月
325 一号依然只有原来的那对兔子。然而，在三月，这对兔子到了生产的年龄，这也就意味着一对新的兔子出生了。在四月一号的时候，这对新出生的兔子增加了种群的数量（若用“对”来计算的话）则达到了两对。在四月，原来的那对兔子继续生产，但是三月出生的那对兔子还太年轻。因此，在五月开始的时候，这里有三对兔子。从此，随着每个月有越来越多的兔子成年，兔子的总数开始增加得越来越快。

7.3.1 计算斐波那契数列项

此时，将迄今为止的兔子按月总数记录成一个数字序列是非常有用的，该序列用下标值 t_i 表示，t_i 表示从某年的一月一号的实验开始到第 i 个月兔子的总对数。这个序列被称为**斐波那契数列**（Fibonacci sequence），且以下述项开始，它表示到目前为止的计算结果：

t_0	t_1	t_2	t_3	t_4
0	1	1	2	3

你可以通过仔细地观察来简化这个数列中更多项的计算。由于问题中的兔子永远不会死，前一个月中所有的兔子在这个月中仍然存在。此外，每一对成年的兔子生产出一对新的兔子。能够繁殖的成年兔子的数量就是前一个月中所有的兔子的数量。净效应是序列中的每个新项一定是前面两项之和。因此，斐波那契数列中接下来的几项看起来如下所示：

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
0	1	1	2	3	5	8	13	21	34	55	89	144

因此，这年结束时兔子的总对数是144。

从编程的角度来看，它帮助我们使用下面更加数学化的形式来表达生成这一斐波那契数列新项的规则：

$$t_n = t_{n-1} + t_{n-2}$$

在该类型的表达式中，一个序列的每个元素由其之前的元素确定，这被称为递归关系（recurrence relation）。

单独的递归关系不足以定义斐波那契数列。虽然上述公式使得计算该数列的新项变得更加
326 容易，但是这个过程必须从某个地方开始。为了应用此公式，你至少需要两个已知项，这意味

若序列中开始的两项（t_0 和 t_1）必须被明确地定义。因此，斐波那契数列完整的描述应为：

$$t_n = \begin{cases} n & \text{若 } n \text{ 是 0 或 1} \\ t_{n-1} + t_{n-2} & \text{其他} \end{cases}$$

这个数学公式对于函数 fib(n) 的递归实现是一个理想的模型，它的作用是计算斐波那契数列中的第 n 项。你所需要做的就是将简单示例以及递归关系插入到标准的递归模式中。fib(n) 的递归实现方法展示在图 7-1 中，它同时包含了一个测试程序，其作用是显示斐波那契数列中两个给定的项。

```
/*
 * File: fib.cpp
 * -------------
 * This program lists the terms in the Fibonacci sequence with
 * indices ranging from MIN_INDEX to MAX_INDEX.
 */

#include <iostream>
#include <iomanip>
using namespace std;

/* Constants */

const int MIN_INDEX  = 0;   /* Index of first term to generate */
const int MAX_INDEX = 20;   /* Index of last term to generate  */

/* Function prototypes */

int fib(int n);

/* Main program */

int main() {
   cout << "This program lists the Fibonacci sequence." << endl;
   for (int i = MIN_INDEX; i <= MAX_INDEX; i++) {
      if (i < 10) cout << " ";
      cout << "fib(" << i << ")";
      cout << " = " << setw(4) << fib(i) << endl;
   }
   return 0;
}

/*
 * Function: fib
 * Usage: int f = fib(n);
 * ----------------------
 * Returns the nth term in the Fibonacci sequence using the
 * following recursive formulation:
 *
 *    fib(0) = 0
 *    fib(1) = 1
 *    fib(n) = fib(n - 1) + fib(n - 2)
 */

int fib(int n) {
   if (n < 2) {
      return n;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

图 7-1 列出斐波那契数列的程序

7.3.2 在递归实现中获得自信

既然你已经有了函数 fib 的一个递归实现，那么如何使自己确信它能够工作？你可以从不断地跟踪程序的运行逻辑开始。例如，考虑一下，如果你调用了 fib(5) 将会发生什么？

由于这里没有出现 if 语句中列举的简单情况，因此将通过执行以下语句得出计算结果。

```
return fib(n - 1) + fib(n - 2);
```

它等价于以下语句：

```
return fib(4) + fib(3);
```

此时，计算机计算 fib(4) 的结果，再加上调用 fib(3) 的结果而得到最终结果，然后作为 fib(5) 的值返回求和结果。

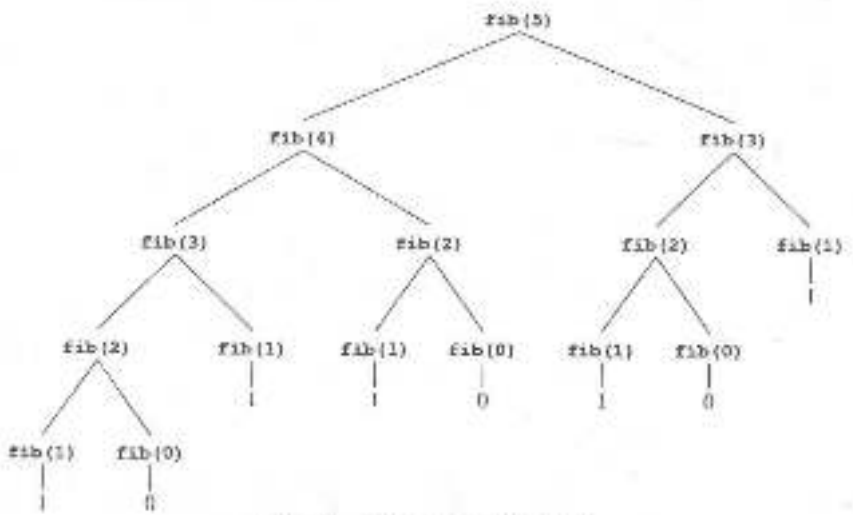
但是计算机是如何计算 fib(4) 以及 fib(3) 的呢？当然，答案是它使用了完全相同的策略。递归的本质就是将原问题分解成更简单的问题，并且这些更简单的问题可以通过调用完全相同的函数解决。这些调用被分解成更简单的调用，这种分解一直持续下去，直到达到最简单的情况为止。

另一方面，最好是将完整的机制视为无关紧要的细节。取而代之的是，仅牢记递归的稳步跳跃即可。此时，你的工作就是理解 fib(5) 的调用是如何工作的。在逐步了解函数的执行过程时，你已经成功地将问题转换成计算 fib(4) 和 fib(3) 之和。因为函数的参数值更小，上述每一个调用都代表了一种更简单的情况。运用递归的稳步跳跃，你可以假设程序在没有被仔细检查所有步骤的情况下能够正确地计算出每一个函数调用值。为了达到验证递归策略的目的，你可以看看表格中的答案：fib(4) 是 3，fib(3) 是 2。因此，调用 fib(5) 的结果就是 3+2，即 5，这确实是正确的答案，示例结束。你不需要了解所有的细节，这些细节最好交给计算机处理。

327
|
328

7.3.3 递归实现的效率

然而，如果你决定检查关于计算 fib(5) 调用的细节，你很快会发现这个计算的效率是非常低的。递归分解产生了很多冗余的调用，其中，计算机在多次计算斐波那契数列中相同的项后终止。图 7-2 说明了这种情况，它展示了计算 fib(5) 所需的所有递归调用。正如你从图中所看到的，这个程序最终调用了一次 fib(4)、两次 fib(3)、二次 fib(2)、五次 fib(1)，以及三次 fib(0)。假设斐波那契函数可以用迭代高效地实现，则递归实现所需的指数级增长的步骤就有些令人烦恼。



329 图 7-2 计算 fib(5) 的步骤

7.3.4 递归不应被指责

在发现图 7-1 中给出的 fib(n) 的实现效率很低之后，很多人都忍不住将矛头指向递归。然而，斐波那契例子的问题与递归本身无关，相反的是与递归的使用方式有关。通过采用一种不同的策略，编写一个可消除图 7-2 所示的大规模低效之处的 fib 函数的递归实现是可能的。

使用递归时，避免低效是常态，其关键在于采用一种更通用的方法以找到一种更高效的解决方案。斐波那契数列并不是唯一的由以下递归关系定义其项的序列：

$$f_n = f_{n-1} + f_{n-2}$$

根据你选择头两项的方式，你可以产生许多不同的序列。传统的斐波那契数列：

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ···

由定义 $t_0=0$ 和 $t_1=1$ 得到。例如，如果定义了 $t_0=3$ 和 $t_1=7$，取而代之，你会得到以下序列：

3, 7, 10, 17, 27, 44, 71, 115, 186, 301, 487, 788, 1275, ···

类似地，若定义 $t_0=-1$ 以及 $t_1=2$，将会产生以下序列：

-1, 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, ···

这些序列都使用了相同的递归关系，这个递归关系指出了每一个新项都是前两项之和。上述序列唯一不同之处在于其开始两项的选择不同。作为一种通用的类别，将遵循这种模式的序列称为**可加序列**（additive sequence）。

这种可加序列的概念能够将求斐波那契数列中的第 n 项的问题转化成更一般的问题，即找出初始两项为 t_0 和 t_1 的可加序列的第 n 项。这样的函数需要三个参数，并且在 C++ 中可以被表示成一个具有以下原型的函数：

```
int additiveSequence(int n, int t0, int t1);
```

如果你有这样一个函数，那么使用它来实现 fib 是很简单的。你所需要做的就是提供开始两项的正确值。如下所示：

330

```
int fib(int n) {
   return additiveSequence(n, 0, 1);
}
```

该函数体仅包含了一行代码，它所做的是只调用另一个传递几个额外参数的函数。这类仅简单地返回另一个通常以某种方式改变参数后的函数的结果的函数被称为包装器（wrapper）函数。包装器函数在递归编程中非常普遍。在大多数情况下，一个包装器函数被用于为一个辅助函数提供额外的参数来解决一个更一般的问题。

由此，剩下的一个任务就是实现函数 additiveSequence。如果你花费几分钟思考一下这个更一般的问题，你会发现可加序列自身有一个很有趣的递归特性。递归的简单情况包括 t_0 和 t_1 两项，它们的值是序列定义的一部分。在 C++ 实现中，这些项的值作为参数被传递。例如，如果你需要计算 t_0，你所需做的只是返回参数 t_0。

然而，如果你被要求找出序列中更大的项呢？例如，假设你想找出可加序列中的 t_5，其中，这个可加序列的初始两项是 3 和 7。通过查看下面这个序列项的列表：

t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
3	7	10	17	27	44	71	115	186	301	...

你可以看出正确的值是 71。然而，一个有趣的问题是你如何使用递归来确定这个结果。

你需要发现的关键点是任何可加序列的第 n 项是从可加序列开始一步一步推进到第 n－1 项的。例如，上例展示的序列中的 t_6 仅是以 7 和 10 开始的，直至可加序列中的 t_5 项之和：

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
7	10	17	27	44	71	115	186	301	...

331 这种深入的观察使得可实现如下的 additiveSequence 函数：

```
int additiveSequence(int n, int t0, int t1) {
   if (n == 0) return t0;
   if (n == 1) return t1;
   return additiveSequence(n - 1, t1, t0 + t1);
}
```

如果你跟踪采用这种技巧实现的 fib（5）的计算步骤，会发现这个计算没有涉及困扰前面递归公式那样的冗余计算。这些步骤直接解决问题，如下图所示：

```
fib(5)
  = additiveSequence(5, 0, 1)
    = additiveSequence(4, 1, 1)
      = additiveSequence(3, 1, 2)
        = additiveSequence(2, 2, 3)
          = additiveSequence(1, 3, 5)
            = 5
```

即使新的实现是完全递归的，它和传统迭代版本的斐波那契函数相比更高效。事实上，我们能够使用更复杂的数学方法编写一个 fib（n）函数完整的递归实现，这种实现方法被认为比迭代策略更高效。你将有机会在第 10 章的习题中亲自编写这个实现。

7.4 检测回文

尽管阶乘和斐波那契函数提供了很好的关于递归函数是如何运行的例子，但是它们本质上都属于数学范畴，因此，可能会传递一个错误的印象，即递归只适用于数学函数。事实上，你可以将递归运用到任何问题中，只要这个问题可以被分解成相同形式的更小问题。因此，多考察几个递归实例，并将关注点放在非数学的特性上将是非常有用的。例如，本节用一个简单的字符串应用说明递归的使用。

一个回文（palindrome）是一个其正序和倒序读取都是完全相同的字符串，例如“level”和“noon”均为回文。尽管通过迭代字符串中的字符来检查其是否是一个回文很简单，但回文也可以被递归地定义。你需要深入观察的就是多于一个字符的任何回文在其内部一定包含了一个更短的回文。例如，字符串“level”由回文“eve”和一个出现在两端的“l”组成。因此，为了检查一个字符串是否是一个回文——假设这个字符串足够长，不至于它是仅
332 由一个字符构成的字符串这种最简单的情况——那么你所需要做的就是：

1. 检查其首字符和最后一个字符是否相同。
2. 检查删除首字符和最后一个字符之后产生的子串是否是一个回文。

若以上两个条件均满足，则这个字符串就是一个回文。

编写一个解决回文问题的递归方案，你必须考虑的唯一问题是：其简单情况是什么？显然，任何由单个字符构成的字符串都是一个回文，因为颠倒这一字符串中的字符对字符串本身没有影响。因此，单个字符的字符串代表了一种简单情况，但这并不是唯一的一种。空串（不包含任何字符的字符串）也是一个回文，在这种情况下，任何递归方案也必须对这种情况进行正确地操作。

图 7-3 包含了函数 isPalindrome 的一个递归实现，如果它的参数是一个回文，则函数返回 true。这个函数首先检测参数字符串的长度是否小于 2。如果是，则该串一定是一个回文。反之，函数检测确认这个串是否都满足以上两个必要条件。

遗憾的是，即使容易遵循这个递归分解，但是图 7-3 展示的实现仍然是低效的。通过做出以下改变，你可以改进函数 isPalindrome 的性能：

- 只计算参数的长度一次。函数 isPalindrome 原始的实现在每次递归分解层次上都计算了字符串的长度。只调用一次 length 方法，然后通过递归调用链将信息向下传递，这种方法效率更高。
- 不要在每次调用中都生成一个子串。在 isPalindrome 的第一个版本中，其低效的原因是重复调用用来删除首字符和最后一个字符的 substr 函数。你可以通过传递索引来跟踪预期的子串开始和结束位置，从而完全避免调用 substr。

```
/*
 * Function: isPalindrome
 * Usage: if (isPalindrome(str)) . . .
 * ------------------------------------
 * Returns true if str is a palindrome, which is a string that
 * reads the same backwards and forwards.  This implementation
 * uses the recursive insight that all strings of length 0 or 1
 * are palindromes and that longer strings are palindromes if
 * their first and last characters match and the remaining substring
 * is a palindrome
 */

bool isPalindrome(string str) {
   int len = str.length();
   if (len <= 1) {
      return true;
   } else {
      return str[0] == str[len - 1] && isPalindrome(str.substr(1, len - 2));
   }
}
```

图 7-3 检查回文的程序

上述每一种改变都要求递归函数获取另外的参数。图 7-4 展示了 isPalindrome 的一个改进版本，它以一个包装器函数实现，该函数的作用是调用辅助函数 isSubstringPalindrome 来实现所需的工作。函数 isSubstringPalindrome 获取额外的指明其应检查的字符串的始末索引位置的参数 p1 和 p2。

```
/*
 * Function: isPalindrome
 * Usage: if (isPalindrome(str)) . . .
 * ------------------------------------
 * Returns true if str is a palindrome, which is a string that reads the
 * same backwards and forwards.  This level of the implementation is
 * simply a wrapper for isSubstringPalindrome, which does the real work
 */
```

图 7-4 isPalindrome 函数更高效的实现

```
bool isPalindrome(string str) {
   return isSubstringPalindrome(str, 0, str.length() - 1);
}

/*
 * Function: isSubstringPalindrome
 * Usage: if (isSubstringPalindrome(str, p1, p2)) . . .
 * ----------------------------------------------------
 * Returns true if the characters in str from p1 to p2, inclusive, form
 * a palindrome.  This implementation uses the recursive insight that
 * all strings of length 0 or 1 are palindromes (the simple cases) and
 * that longer strings are palindromes only if their first and last
 * characters match and the remaining substring is a palindrome.
 */

bool isSubstringPalindrome(string str, int p1, int p2) {
   if (p1 >= p2) {
      return true;
   } else {
      return str[p1] == str[p2] && isSubstringPalindrome(str, p1 + 1, p2 - 1);
   }
}
```

333
:
334

图 7-4 （续）

7.5 二分查找算法

当你处理一系列存储在一个矢量中的数值时，一个最常见的操作就是在这个矢量中查找某个特定的元素。例如，如果你经常要处理字符串矢量，拥有以下函数是很有用的：

```
int findInVector(string key, Vector<string> & vec);
```

该函数遍历 vec 中的每一个元素，寻找一个其值等于 key 的元素。如果找到匹配者，findInVector 将返回这个元素的下标索引。如果该值不存在，则函数返回 -1。

如果你对这个矢量的元素顺序没有确切的认识，那么函数 findInVector 的实现必须依次检查其中的每个元素，直到发现了匹配的元素或者遍历完所有元素为止。这种策略被称为**线性查找算法**（linear-search algorithm），如果矢量很大，那么它会耗费大量的时间。然而，如果你已知矢量中的元素是按字典顺序排列的，那么，你可以采取另一种更高效的方法。你需要将这个矢量划分成两半，然后用 ASCII 字符编码定义的被称为**字典顺序**（lexicographic order）的顺序将你正在试图查找的关键字与最接近矢量中间的那个元素相比较。如果你正在寻找的关键字在中间元素之前，那么这个关键字（如果它存在）一定在前半部分。相反地，在字典顺序中，如果关键字在中间元素的后面，你需要在后半部分查找元素。这种策略被称为**二分查找算法**（binary-search algorithm）。由于二分查找能够使你在每一步查找的过程中排除一半的可能元素，因此，相对于线性查找已排序的矢量而言，已证明它具有更高的效率。

图 7-5 所示的二分查找算法也是分治策略的一个完美实例。因此，二分查找存在一个自然的递归实现并不令人惊讶。注意到函数 findInSortedVector 是作为一个包装器函数实现的，它将真正的工作留给了递归函数 binarySearch，这个函数有两个额外的参数（索引 p1 和 p2）以限制查找的范围。

binarySearch 函数的简单情况如下：

1. 在矢量当前查找部分中没有该元素。这个条件被标记成索引 p1 比索引 p2 大，这也意味着没有剩下可供查找的元素。

2. 中间的元素和查找的关键字匹配。由于这个关键字刚好有匹配者，findInSorted

Vector 可以简单地返回这个匹配元素的索引。 [335]

```
/*
 * Function: findInSortedVector
 * Usage: int index = findInSortedVector(key, vec);
 * ------------------------------------------------
 * Searches for the specified key in the Vector<string> vec, which
 * must be sorted in lexicographic (character code) order.  If the
 * key is found, the function returns the index in the vector at
 * which that key appears.  (If the key appears more than once in
 * the vector, any of the matching indices may be returned.)  If the
 * key does not exist in the vector, the function returns -1.  This
 * implementation is simply a wrapper function; all of the real work
 * is done by the more general binarySearch function.
 */

int findInSortedVector(string key, Vector<string> & vec) {
   return binarySearch(key, vec, 0, vec.size() - 1);
}

/*
 * Function: binarySearch
 * Usage: int index = binarySearch(key, vec, p1, p2);
 * --------------------------------------------------
 * Searches for the specified key in the Vector<string> vec, looking
 * only at indices between p1 and p2, inclusive.  The function returns
 * the index of a matching element, or -1 if no match is found.
 */

int binarySearch(string key, Vector<string> & vec, int p1, int p2) {
   if (p1 > p2) return -1;
   int mid = (p1 + p2) / 2;
   if (key == vec[mid]) return mid;
   if (key < vec[mid]) {
      return binarySearch(key, vec, p1, mid - 1);
   } else {
      return binarySearch(key, vec, mid + 1, p2);
   }
}
```

图 7-5 “分而治之”的二分查找实现

如果上述两种情况都不满足，算法实现可以将问题简化为在矢量中挑选合适的一半进行查找，然后以这个更新后的查找范围来递归地调用自己。

7.6 间接递归

截止目前，上述每一个例子中的递归函数都是直接调用自己，即函数体内包含一个自身的调用。尽管你遇到的大多数的递归函数都可能是这种风格，但是递归定义实际上要更宽泛一些。为了成为一个递归函数，它必须在计算过程中的某一时刻调用自己。如果一个函数分 [336]
解为若干子函数，那么这个递归调用可以发生在更深层次的嵌套调用中。例如，一个函数 f 调用了另外一个函数 g，反过来函数 g 调用函数 f，这种形式的函数调用依旧被认为是递归的。由于函数 f 和 g 彼此调用，这种类型的递归被称为间接递归（mutual recursion）。

正如一个简单的例子展示的一样，使用递归来测试一个数字是奇数还是偶数被证明是很简单的，尽管它非常低效。例如，图 7-6 所示的代码通过采用下面的形式化定义实现了 IsEven 和 IsOdd 函数：

- 如果一个数的前趋是奇数，则这个数为偶数。
- 如果一个数不是偶数，则它必为奇数。
- 数字 0 被定义为偶数。

尽管这些规则看上去很简单，只要数字是非负的，它们形成了一种有效地区分奇数和偶数的

策略基础。

图 7-6 中的代码通过让函数 isEven 和 isOdd 获取的参数类型为 unsigned 的方式确保了实现条件，C++ 使用 unsigned 类型表示不小于 0 的整数。

```
/*
 * Function: isOdd
 * Usage: if (isOdd(n)) . . .
 * ---------------------------
 * Returns true if the unsigned number n is odd.  A number is odd
 * if it is not even.
 */

bool isOdd(unsigned int n) {
   return !isEven(n);
}

/*
 * Function: isEven
 * Usage: if (isEven(n)) . . .
 * ---------------------------
 * Returns true if the unsigned number n is even.  A number is even
 * either (1) if it is zero or (2) if its predecessor is odd.
 */

bool isEven(unsigned int n) {
   if (n == 0) {
      return true;
   } else {
      return isOdd(n - 1);
   }
}
```

图 7-6 isEven 和 isOdd 的间接递归定义

7.7 递归地思考

对于大多数人来说，递归不是一个易于掌握的概念。学习有效地使用递归需要大量的练习，并且它迫使你用全新的方法来解决问题。成功的关键在于形成正确的习惯——学习如何递归地进行思考。本章其余部分将帮助你实现这个目标。

7.7.1 保持全局的观点

当你学习编程时，我认为牢记整体论与简化论的哲学理念将会对你有很大的帮助。简单地说，简化论（reductionism）就是这样一种理念，它仅仅通过理解构成对象的某一部分就可理解整个对象。与之对立的就是整体论（holism），它认为整体总是比构成它的部分总和更为重要。当你学习编程时，它帮助你能够交错这两种视角，有时候将程序的行为当作一个整体，而在其他时刻，需要探究执行的细节。然而，当你试图理解递归时，这种平衡似乎被改变了。递归地思考需要你从整体的角度来思考。在递归领域，简化论是理解的敌人，它总是妨碍你的理解。

为了保持全局的视角，你必须习惯于采用本章 7.2 节所介绍的递归的稳步跳跃的理念。无论你编写一个递归程序或者尝试理解一个递归程序的行为，你都必须找到在一个单独的递归调用中那些可以被忽略的细节。只要你选择了正确的分解，确定了适当的简单情况，并且正确地实现了你的策略，这些递归调用将会起作用，你不必考虑它们的具体实现细节。

遗憾的是，除非你有大量处理递归函数的经验，否则有效地运用递归的稳步跳跃这一概念并非易事。采用递归需要暂缓你的怀疑，并且对程序的正确性做出假设，它完全违反你以往的经验。举例，当你编写一个程序时，这是很有可能的（即使你是一个有经验的程序员），

你的程序第一次也不会正确地工作。事实上，很可能是由于你选择了错误的分解，搞乱了简单情况的定义，或者在你试图实现你的策略时莫名其妙的将事情弄得一团糟。如果你已经做了这些事中的任何一个，那么你的递归调用将不会工作。 337 ↓ 338

当事情出错时（由于它们不可避免地会发生），你必须牢记在合适的地方寻找错误。问题是你在递归实现中某个地方发生了错误，不是递归机制本身有什么问题。如果递归程序有问题，你应该能够通过合找单个的递归层次来找到其中的错误。向下查找递归调用的其他层次不会有帮助。如果简单情况能工作，并且递归分解是正确的，那么子调用会正常工作。否则，问题一定出现在递归分解的公式中。

7.7.2 避免常见的错误

随着你不断地获得递归相关的经验，编写和调试递归程序将会变得更自然。然而，刚开始找出你在一个递归程序中需要注意的东西是很困难的。下面是一个清单，它将帮助你辨别最常见错误的根源。

- 你的递归实现是以检查简单情况开始吗？在你尝试通过将一个问题转化成一个递归的子问题的方式来解决之前，必须先检查这个问题是否足够简单以至于这样的分解是不是必须的。在绝大多数情况下，递归函数以关键字 `if` 开始。如果你的函数不是这样，应该仔细地检查你的程序，并确保你知道自己正在做什么。
- 你是否已经正确地解决了这些简单情况？由于使用错误的方法解决简单情况，会在递归程序中产生令人惊讶的错误数量。如果简单情况是错误的，更复杂问题的递归方案将会继承同样的错误。例如，如果你错误地将 `fact(0)` 定义成 0 而不是 1，用任何参数调用 `fact` 都将返回 0。
- 你的递归分解让问题变简单了吗？对于能正确运行的递归，求解问题会变得越来越简单。更正式地讲，这里一定有某种度量标准（metric）（一种根据问题的难度从而给该问题赋一个数值的标准的测量方法）其值会随着计算过程的进行而逐渐地变小。对于像 `fact` 以及 `fib` 这样的数学函数来说，以整型参数值作为度量标准。在每一次递归调用中，参数值都将变小。对于函数 `isPalindrome` 来说，由于字符串在每一次调用中都不断地变短，因此合适的度量标准就是字符串参数的长度。如果问题实例没有变得更简单，分解过程将会不断地产生越来越多的调用，会产生类似于死循环一样的被称为无穷递归（nonterminating recursion）的递归调用。
- 这些简单化的处理最终能到达简单情况吗，或者你遗漏了一些其他的可能性？错误的一般根源是，对于所有可以作为递归分解的结果情况中，没有包括简单情况测试。 339 例如，在如图 7-3 所示的 `isPalindrome` 实现中，函数中检查空字符以及单个字符的情况是非常重要的，即使用户从来不打算以空字符串调用函数 `isPallindrome`。随着递归分解的进行，字符串参数在每一层的递归调用中都将缩短两个字符。如果原始字符串参数的长度是偶数，那么递归分解永远不会进入单个字符的情况。
- 你的函数递归调用表示与原始问题的子问题在形式上是完全相同的吗？当你使用递归来分解一个问题时，子问题和原问题具有相同的形式是很重要的。如果分解调用改变了问题的本质或者违背了一个初始的假设，那么整个过程将会失败。正如本章中的一些示例所示，定义公有接口的函数作为一个简单的包装器函数是非常有用的，它调用一个更一般的私有的递归函数。由于私有函数具有更一般的形式，因此，它

通常更易于将原始问题进行分解，并使得子问题仍具有递归的结构。

- 当你运用递归跳步跳跃时，递归子问题的求解是否为原始问题提供了一个完整的解决方案？将一个问题分解成递归的子问题只是递归过程的一部分。一旦你获得了这些子问题的解，你还必须能够将它们重新整合以形成问题的一个完整解。检查其处理过程是否产生了问题的真实解的方法就是核查分解，这需要严谨地运用递归的跳步跳跃。检查当前函数调用的每一步，但假设每一个递归调用都生成了正确的答案。如果遵循这一过程并且产生了正确的解，你的程序应该能正常工作。

本章小结

- 本章介绍了递归的概念，它是一种强大的编程策略，其中，复杂的问题可以被分解成形式相同的更简单的问题。本章的重点包括：
- 递归与逐步求精法类似，两种策略都是将一个问题分解成更易于处理的简单问题。
340 递归的不同之处在于简单的子问题必须和原始问题具有相同的形式。
- 为了使用递归，你必须能够识别问题解的简单情况，以及允许你将任何复杂问题分解成具有相同类型的更简单问题的递归分解。
- 采用C++，递归函数通常都有以下范例形式（范型）：

```
if (test for simple case) {
   Compute a simple solution without using recursion.
} else {
   Break the problem down into subproblems of the same form.
   Solve each of the subproblems by calling this function recursively.
   Reassemble the subproblem solutions into a solution for the whole.
}
```

- 和其他任何函数调用一样，递归函数也是使用完全相同的机制实现的。每次调用都创建了一个新的包含了调用中的局部变量的栈帧。由于计算机为每一次函数调用创建了一个单独的新栈帧，因此每一层递归分解的局部变量都是相互独立。
- 在你能够有效地使用递归之前，必须学会将你的分析限定在递归分解的一个单独的层次上，并且在没有跟踪整个计算过程的前提下确保所有更简单的递归调用的正确性。相信这些更简单的调用能够正确地工作被称为递归的跳步跳跃。
- 数学函数经常用递归关系的形式来表达它们的递归性质，其中，一个序列中的每个元素都根据它前面的元素定义。
- 即使某些递归函数可能与它们对应的迭代表示相比效率更低，但是递归本身并没有问题。和所有典型的各类算法一样，某些递归策略要比其他策略更为有效。
- 为了确保一个递归分解产生的子问题与原问题具有相同的形式，经常有必要使问题一般化。因此，采用一种简单的包装器函数实现一个特定问题的求解方案通常是非常有用的，包装器函数的唯一目的是调用一个辅助函数处理更一般的情况。
- 递归不一定由一个调用自身的函数组成。它可能涉及几个在一个循环模式中彼此调用的函数。涉及不止一个函数的递归被称为间接递归。
- 如果你能保持全局的观点而非简化的视角，那么你会在理解递归程序上更加成功。

以正确的方式思考递归问题并非易事。学习有效地使用递归需要不断地练习。对于大多数学生而言，掌握递归的概念就要花费数年。但是，由于递归将会成为你编程技能中最强大

的技术之一，因此值得花费时间去学习。 341

复习题

1. 定义术语递归和迭代。一个函数可以同时使用这两种策略实现吗？
2. 递归和传统的逐步求精法根本的不同之处是什么？
3. 在函数 collectContributions 的伪码中，if 语句如下所示：

```
if (n <= 100)
```

使用 <= 操作符代替简单地检查 n 是否等于 100 为什么很重要？

4. 标准的递归范型是什么？
5. 采用递归来有效地求解一个问题，该问题必须拥有的两个性质是什么？
6. 为什么术语分而治之适用于递归技术？
7. 递归的跳步飞跃的含义是什么？作为一个程序员，这个概念对你为什么很重要？
8. 7.2.2 节给出了很长的一段分析来说明当 fact(4) 被调用时内部发生了什么。采用这节作为模型，跟踪 fib(3) 的执行过程，并画出递归过程中创建的每一个栈帧。
9. 什么是递归关系？
10. 通过引入额外的规则，即一对兔子在生下三对兔子之后将会停止繁殖。请修改斐波那契兔子问题。这个假设会如何改变其递归关系？你需要在简单情况上做出什么改变？
11. 当使用图 7-1 给出的递归实现来计算 fib(n) 时，fib(1) 被调用了多少次？
12. 什么是包装器函数？为什么在编写递归函数时，它经常很有用？ 342
13. 如果你在函数 additiveSequence 中删除了 if(n==1) 的检测语句，将会发生什么？其实现代码如下所示：

```
int additiveSequence(int n, int t0, int t1) {
   if (n == 0) return t0;
   return additiveSequence(n - 1, t1, t0 + t1);
}
```

这个函数还能工作吗？为什么能或者不能？

14. 为什么在图 7-3 中函数 isPalindrome 的实现对空串以及单个字符的字符串的检查是很重要的？如果这个函数不检查单个字符的情况，取而代之的是只检查字符串的长度是否为 0，将会发生什么？这个函数还能正确地工作吗？
15. 解释图 7-4 给出的 isPalindrome 实现中的以下函数调用结果：

```
isPalindrome(str, p1 + 1, p2 - 1)
```

16. 什么是间接递归？
17. 如果像下面一样定义 isEven 和 isOdd，将会发生什么：

```
bool isEven(unsigned int n) {
   return !isOdd(n);
}

bool isOdd(unsigned int n) {
   return !isEven(n);
}
```

这个例子说明了 7.7.2 节中的哪一种错误？

18. 下面关于 isEven 和 isOdd 的定义也是不正确的：

```
bool isEven(unsigned int n) {
   if (n == 0) {
      return true;
   } else {
      return isOdd(n - 1);
   }
}
bool isOdd(unsigned int n) {
   if (n == 1) {
      return true;
   } else {
      return isEven(n - 1);
   }
}
```

343

给出一个例子，展示这个实现是怎么失败的？这里展示的常见错误是什么？

习题

1. 球体（例如炮弹）可以被堆叠成成一个顶上只有一个炮弹的金字塔，这个炮弹坐落在一个由四个炮弹组成的正方形上，这四个炮弹又坐落在一个由九个炮弹组成的正方形上，依此类推。编写一个递归函数 `cannonball`，这个函数以一个金字塔的高度作为参数，并返回这个金字塔所包含的炮弹的总数。你的函数必须递归地操作，并且不允许使用任何迭代的结构，例如 `while` 和 `for`。
2. 和许多编程语言不一样，C++ 并不包含一个预定义的求幂操作符。作为对于这个缺陷的一个补救，函数

```
int raiseToPower(int n, int k)
```

的递归的实现使它能够计算 n^k。你需将求解这个具有以下数学性质的问题进行递归函数的实现。

$$n^k = \begin{cases} 1 & \text{若 } k \text{ 为 } 0 \\ n \times n^{k-1} & \text{其他} \end{cases}$$

3. 在 18 世纪，天文学家尤翰·丹尼尔·提丢斯（Johann Daniel Titius）为了计算太阳与当时已知的每一个星球之间的距离制定了一个规则，这个规则后来被约翰·埃勒特·波得（Johann Elert Bode）记载下来。为了运用这个规则，即著名的**提丢斯－波得定律**（Titius-Bode Law），你可以通过书写以下序列开始：

$$b_1 = 1 \quad b_2 = 3 \quad b_3 = 6 \quad b_4 = 12 \quad b_5 = 24 \quad b_6 = 48$$

其中，序列中的每个其后的元素都是其前一个元素的两倍。运用以下公式，我们可以计算出这个序

344

列中第 i 个星球到太阳的大致距离：

$$d_i = \frac{4 + b_i}{10}$$

距离 d_i 被表示成天文单位（AU），它相当于地球到太阳的平均距离（大约是 93 000 000 英里）。除了火星和木星之间有一个空缺，提丢斯－波得定律在当时给出了已知的七个星球与太阳之间的合理的大致距离：

Mercury	0.5AU
Venus	0.7AU
Earth	1.0AU
Mars	1.6AU

?	2.8AU
Jupiter	5.2AU
Saturn	10.0AU
Uranus	19.6AU

天文学家对序列中出现的空缺的关注，导致发现了小行星带，这显示了这个小行星带可能是在很久之前太阳的12年之一的行星的残骸。

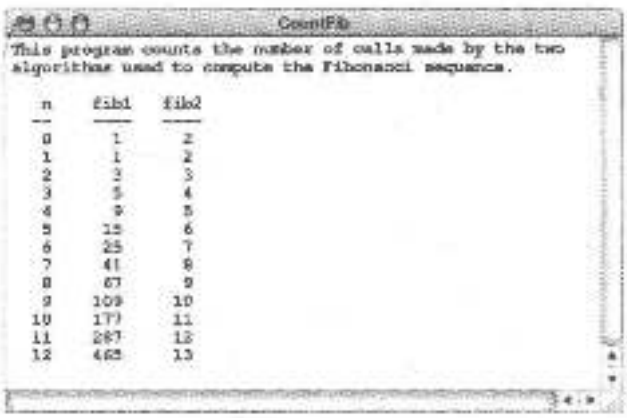
编写一个计算太阳和第k个星球之间的距离的递归函数 `getTitiusBodeDistance(k)`，将水星编号为1，并且向外逐渐编号。编写一个测试函数，并用表格形式展示这些星球到太阳的距离。

4. 两个非负整数的最大公约数（经常被简写成 gcd）就是能够这两个数整除的最大整数。在公元前3世纪，希腊数学家欧几里得发现x和y的最大公约数总是可以像下面这样计算：
   - 若x能够整除y，那么y就是最大公约数。
   - 否则，x和y的最大公约数总是等于y和x除以y之后所得的余数的最大公约数。

   使用欧几里得理论编写一个计算x和y的最大公约数的递归函数 `gcd(x, y)`。

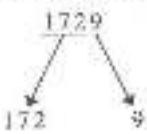
5. 编写一个关于函数 `Fib(n)` 迭代的实现。

6. 对于本章中出现的函数 `fib(n)` 的两个递归实现版本，编写一个递归函数（你可以调用两个函数 `countFib1` 和 `countFib2`），在相应的 Fibonacci 计算过程中计算函数调用的次数。编写一个 345
   主程序，要求它使用这些函数展示一个表格，其内容为在n取不同值时，每一个算法的函数调用次数，程序运行结果如下图所示：



7. 编写一个递归函数 `digitSum(n)`，它的参数为一个非负的整数，并且返回它的各位数字之和。例如，调用 `digitSum(1729)` 应该返回1+7+2+9，也就是19。

   `digitSum` 的递归实现依赖于这样一个事实，即非常容易通过除以10的方式来将一个整数分成两部分。例如，给出整数1729，你可以如下图所示将其分解成两部分：



   结果中的每一个整数都比原来的整数要小，因此，它代表了一种更简单的情况。

8. 一个整数n的数根（digit root）被定义为：重复地将其各位数相加直到保留最终的单个数字。例如，1729可以用下面的步骤来计算它的数根：

   步骤1：1+7+2+9→19

   步骤2：1+9→10

   步骤3：1+0→1 346

由于在步骤3执行后得到的是单个数字1，故这个值就是数根。

编写一个返回其参数数根的函数 digitalRoot(n)。尽管很容易使用习题7中的 origami 函数和一个 while 循环来实现 digitalRoot，但编写一个不直接使用任何循环结构的递归函数也是这个问题的一部分挑战。

9. 正如你在第2章中所了解到的，数学组合函数 c(n, k) 通常被定义为阶乘的形式，如下所示：

$$c(n,k) = \frac{n!}{k! \times (n-k)!}$$

c(n, k) 的值也可以按照几何学的方式排列形成一个三角形，在这个三角形中，随着向下三角形的后移动，n 的值增加；从左向右移动，k 的值增加。最终的结构以法国数学家布莱士·帕斯卡（Blaise Pascal）命名，被称为帕斯卡三角形（Pascal's Triangle），其排列如下所示：

c(0, 0)

c(1, 0) c(1, 1)

c(2, 0) c(2, 1) c(2, 2)

c(3, 0) c(3, 1) c(3, 2) c(3, 3)

c(4, 0) c(4, 1) c(4, 2) c(4, 3) c(4, 4)

帕斯卡三角形有一个有趣的性质，即除了左边和右边的数字之外（值为1），其他数字都是其上面两个数字之和。例如，如以下帕斯卡三角形中圈出的数字所示：

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

1 6 (15) 20 15 6 1

1 7 21 35 35 21 7 1

347 它对应于 c(6, 2)，是出现在这个数上的两个数 5 和 10 之和。利用帕斯卡三角形中数字间的关系，编写一个函数 c(n, k) 的递归实现，要求这个函数不使用循环、不使用乘法运算以及不调用 fact 函数。

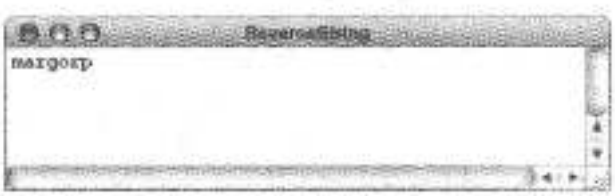
10. 编写以一个字符串作为参数的函数，要求函数返回这个字符串的逆序字符串。这个函数的原型为：

```
string reverse(string str);
```

语句

```
cout << reverse("program") << endl;
```

应该显示如下结果：



你的解决方案应该是完全递归的，并且它不应该使用任何迭代结构，例如 while 或者 for。

11. strlib.h 库包含了一个函数 integerToString。在不使用该函数的前提下，通过使用习题8中描

348 述的对一个整数的递归分解来重新实现这个函数。

第 8 章
Programming Abstractions in C++

递归策略

未战而庙算不胜者，得算少也。

——孙子，公元前 5 世纪

349

若递归分解直接从一个数学定义中得出，正如第 7 章中的 fact 和 fib 函数，采用递归并不会特别困难。大多数情况下，你可以通过将合适的表达式插入到标准的递归范型中，将数学定义直接转换成一个递归实现。然而，当你开始解决一些更复杂的问题时，情况会发生变化。

本章将引入几个编程问题，它们看起来（至少在表面上）比第 7 章的问题更难。事实上，如果你尝试不采用递归来解决这些问题，而是依赖于更熟悉的迭代技术，你会发现它们很难。相比之下，这些问题都有一个惊人简短递归的求解方案。如果你发挥递归的威力，对于每个问题只需要几行代码就足够了。

然而，这些解决方案的简洁性赋予了它们一种假象，即问题简单。解决问题困难部分与代码的长度毫不相关。编写这些程序的困难点在于：首先要找到递归分解的办法。有时这样做需要一些巧妙的思维，但是你真正需要的是信心。你必须接受递归的稳步跳跃。

8.1 汉诺塔

本章的第一个例子是一个简单的谜题，即广为人知的**汉诺塔**（Towers of Hanoi）问题。这个谜题是由法国数学家爱德华·卢卡斯在 1880 年提出的，汉诺塔问题迅速在欧洲流行。它的成功部分归功于围绕这一谜题的逐渐增长的传奇，它在法国数学家亨利·德·巴微（Henri de Parville）所著的《自然之谜》(*La Nature*)（由数学史学家鲍尔翻译）中描述如下：

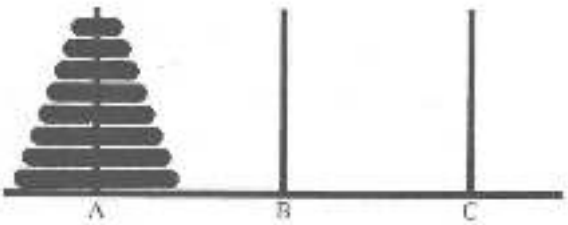
> 在世界中心贝拿勒斯圆顶之下的一座圣庙里，放置了一个黄铜板，有三根宝石针固定在上面，每一根针都有一肘高，厚度和一个蜜蜂的身体一样。创建世界时，梵天在其中一根针上面放了 64 个纯金圆盘，最大的金圆盘放在铜板上，其他的金圆盘随着高度升高，越来越小，最上面的一个是最小的一个。这就是梵天寺之塔。昼夜不断，牧师将金圆盘从一根宝石针移动到另一根上，根据梵天寺固定不变的法则，它要求牧师的职责是每次移动的金圆盘数不能超过一个。他必须把这个金圆盘放在一根针上，并且在这个金圆盘下面没有更小的金圆盘。当所有的金圆盘都从梵天穿好的那根针上移到另外一根针上时，世界将在一声雷霆中毁灭，而梵塔、庙宇和众生也都将同归于尽。

很多年过去了，环境已经从印度转移到越南，但问题及其说明仍保持不变。 350

据我所知，汉诺塔谜题除了向学计算机科学的学生教授递归之外，并没有实际作用。在那个领域，它有巨大的价值，因为解决方法只涉及递归。和大部分对应于现实世界问题的递

旧算法相比，汉诺塔问题没有额外的复杂之处，否则可能会干扰你对问题的理解，并使你不明白递归方案是如何解决问题的。因为作为一个例子，它能很好地解决问题，汉诺塔已经包含在大部分处理递归的教科书中，并且它已经成为（像第1章的“hello,world”程序）计算机科学家共同分享的文化遗产中的一部分。

在这个问题的商业版本中，64个传奇的金圆盘被替换成8个木质的或塑料的圆盘，这使得问题更易于解决（而不是说成本更低）。问题的初始状态看起来如下图所示：



一开始，8个圆盘都在塔杆A上，你的目的是将这8个圆盘从塔杆A移到塔杆B。同时要遵循以下规则：

- 每次只能移动一个圆盘。
- 不允许将一个大圆盘移动到小圆盘之上。

8.1.1 问题框架

为了将递归应用到汉诺塔问题，你必须首先在更一般的条件下才清楚问题的框架。尽管最终的目的是将所有的圆盘从A移到B，问题的递归分解涉及在各种结构下将更小的圆盘子塔从一个塔杆移到另一个塔杆。在更一般的情况下，你需要解决的问题是将一个给定高度的塔从一个塔杆移到另一个塔杆，使用第三个塔杆作为一个临时仓库。为了确保所有的子问题适合于原始的形式，因此，你的递归程序必须包含以下参数：

1. 需要移动的圆盘数目。
2. 所有圆盘开始所在的塔杆名字。
3. 所有圆盘最终应该放置的塔杆名字。

[351] 4. 用于临时存储圆盘的塔杆名字。

需要移动的圆盘数目显然是一个整数，使用char类型表示被标记为A、B和C的塔杆，表明此时哪一个塔杆将被涉及。了解类型使你可以为移动塔这个操作编写一个函数原型，如下所示：

```
void moveTower(int n, char start, char finish, char tmp);
```

为了移动例子中的8个圆盘，初始调用为：

```
moveTower(8, 'A', 'B', 'C');
```

该函数调用对应于英文指令“Move a tower of size 8 from spire A to spire B using spire C as a temporary.”随着递归分解的逐步进行，moveTower将会以不同的参数形式被调用，它以不同的配置移动更小的塔。

8.1.2 找到一种递归策略

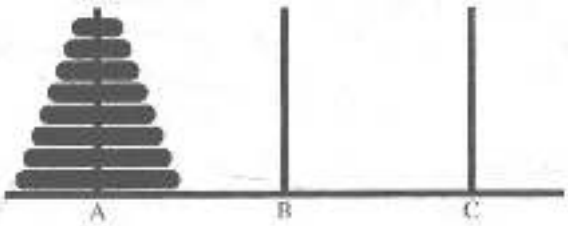
既然你已经有了对这个问题更一般的定义，那么就可以转到寻找某种策略以移动一个大

塔这一问题上来。为了应用递归，你必须首先确保问题满足以下条件：

1. 必然存在一种简单的情况。在这个问题中，n 等于 1 即为简单情况，这意味着只有一个圆盘需要移动。只要你没有违反将一个大的圆盘放在小的圆盘之上这一规则，你就可以用一步操作移动一个圆盘。

2. 必然有走递归分解。为了实现一个递归求解，必须可以将原问题分解成与原问题具有同样形式的更简单的问题。递归分解这部分更难，并且需要仔细检查。

为了弄明白如何通过解决一个更简单的子问题帮助解决一个大的问题，需要回过头来考虑原来有 8 个圆盘的例子。



这个问题的目的是将 8 个圆盘从塔柱 A 移到塔柱 B。你需要自问一下：如果你能解决具 352
有更小数目的圆盘的同样问题，这对你解决原问题有什么帮助。特别是，你应该思考一下，能够解决 7 个圆盘的问题怎么能帮助你解决 8 个圆盘的情况。

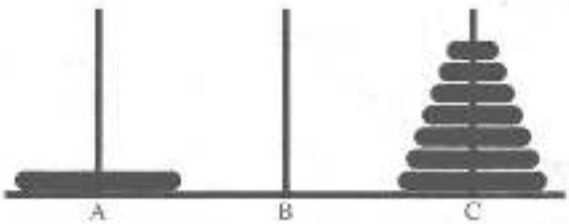
如果你思考这个问题一段时间，问题求解思路便会逐渐清晰：可以将问题分成以下三个步骤来进行求解：

1. 将堆叠在塔柱 A 上面的 7 个圆盘全部移到塔柱 C。

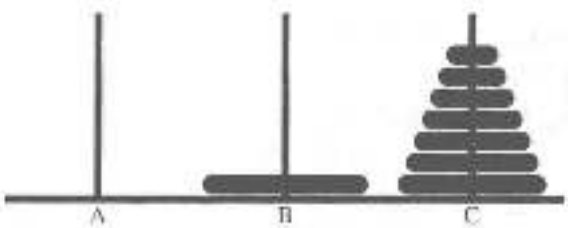
2. 将塔柱 A 最底下的一个圆盘移到塔柱 B。

3. 将塔柱 C 上的 7 个圆盘移到塔柱 B。

执行第一步后，你会看到下面圆盘所处的位置：



一旦你移除了最大圆盘上面的 7 个圆盘，第二步就是简单地将该圆盘从塔柱 A 移到塔柱 B，这将会产生以下的结构：



剩下的问题就是将 7 个圆盘从塔柱 C 移到塔柱 B，这又再次成为原形式的一个更小的问题。这个操作是上述递归策略中的第三个步骤。它使问题达到最终我们所期望的结果：



153

就是它！你完成了。你已经将移动8个圆盘的塔问题降低到移动7个圆盘的塔问题。更重要的是，这个递归策略可以推广到以下N个圆盘的塔问题：

1. 将最上面的N－1个圆盘从开始所在的塔柱移动到临时塔柱上。
2. 将最底下单独的一个圆盘从开始所在的塔柱移到最终所在的塔柱上。
3. 将N－1个圆盘从临时塔柱上移动到最终所在的塔柱上。

此刻，很难避免对自己说："好，我可以将问题降到只移动N－1个圆盘的塔，但是我如何完成它？"毫无疑问，答案是你使用同样的方法来移动N－1个圆盘的塔。你将这个问题分解成只需要移动N－2个圆盘的塔，这个问题随后分解成只需要移动N－3个圆盘的塔，依此类推，一直持续到只需移动一个圆盘为止。然而，从心理学上讲，最重要的是避免一次性询问所有的问题。递归的跳步跳跃应该已经足够了。你已经在不改变问题本身形式的情况下，将它的规模减小了。这是一项艰难的工作。接下来就是记流水账，这些最好让计算机去处理。

一旦你已经确定了简单情况和递归分解，你所需要做的就是将它们整合成标准的递归框式，这会产生下面的伪码程序：

```
void moveTower(int n, char start, char finish, char tmp) {
   if (n == 1) {
      Move a single disk from start to finish
   } else {
      Move a tower of size n - 1 from start to tmp
      Move a single disk from start to finish
      Move a tower of size n - 1 from tmp to finish
   }
}
```

8.1.3 验证这个策略

尽管事实上，以上的伪码策略是正确的，但是问题推导到这一步还有些问题。当你使用递归去分解一个问题时，你必须确保新问题和原问题的形式相同。将N－1个圆盘从一个塔柱移到另一个塔柱的任务听起来好像是原问题的一个实例，并且它适用于函数moveTower原型。即使这样，仍有一个微小但重要的不同点。在原问题中，目标塔柱和临时塔柱都是空的。但你移动N－1个圆盘的塔到临时塔柱作为递归策略的一部分时，在开始所处的塔柱上剩下了一个圆盘。这个圆盘的存在是否会改变问题的性质并因此证明递归策略是无效的？

154

为了回答这个问题，你需要根据游戏规则来思考问题。如果递归分解不是以违反规则结束的，那么一切都好。第一个规则（每次只有一个圆盘可以被移动）不是问题。如果不止一个圆盘，递归分解将问题分解，以产生一个更简单的问题。伪码中实际移动圆盘的步骤只能每次移动一个圆盘。第二条规则（不允许将较大的圆盘放在较小的圆盘上面）是一个关键问题。你需要说服自己：你不会在递归分解中违反这条规则。

一个重要的观察结果是：当你将子塔从一个塔柱移到另一个塔柱时，你遗留在原来塔柱的圆盘（在操作之前的任何阶段，那些被遗留的圆盘）一定比当前子塔中的圆盘大。因此，当你在塔柱之间移动这些圆盘时，它们下面的唯一一个圆盘尺寸一定比它们大，这是与规则一致的。

8.1.4 编码方案

为了完成汉诺塔的解决方案，唯一的缺失步骤就是替换剩余伪码的函数调用。移动一个完整的塔的任务需要递归调用 moveTower 函数。其他唯一的操作就是将一个单独的圆盘从一个塔柱移到另一个塔柱。为了编写能显示解决方案的各步骤的测试程序，你所需做的就是编写一个将这些操作输出在控制台上的函数。例如，你可以如下所示实现函数 moveSingleDisk：

```
void moveSingleDisk(char start, char finish) {
   cout << start << " -> " << finish << endl;
}
```

moveTower 代码本身如下所示：

```
void moveTower(int n, char start, char finish, char tmp) {
   if (n == 1) {
      moveSingleDisk(start, finish);
   } else {
      moveTower(n - 1, start, tmp, finish);
      moveSingleDisk(start, finish);
      moveTower(n - 1, tmp, finish, start);
   }
}
```

其完整的实现显示在图 8-1 中。 355

```
/*
 * File: Hanoi.cpp
 * ---------------
 * This program solves the Towers of Hanoi puzzle.
 */

#include <iostream>
#include "simpio.h"
using namespace std;

/* Function prototypes */

void moveTower(int n, char start, char finish, char tmp);
void moveSingleDisk(char start, char finish);

/* Main program */

int main() {
   int n = getInteger("Enter number of disks: ");
   moveTower(n, 'A', 'B', 'C');
   return 0;
}

/*
 * Function: moveTower
 * Usage: moveTower(n, start, finish, tmp);
 * ----------------------------------------
 * Moves a tower of size n from the start spire to the finish
 * spire using the tmp spire as the temporary repository.
 */
```

图 8-1 解决汉诺塔问题的程序

```
void moveTower(int n, char start, char finish, char tmp) {
   if (n == 1) {
      moveSingleDisk(start, finish);
   } else {
      moveTower(n - 1, start, tmp, finish);
      moveSingleDisk(start, finish);
      moveTower(n - 1, tmp, finish, start);
   }
}

/*
 * Function: moveSingleDisk
 * Usage: moveSingleDisk(start, finish);
 * ------------------------------------
 * Executes the transfer of a single disk from the start spire to the
 * finish spire.  In this implementation, the move is simply displayed
 * on the console; in a graphical implementation, the code would update
 * the graphics window to show the new arrangement.
 */

void moveSingleDisk(char start, char finish) {
   cout << start << " -> " << finish << endl;
}
```

356

图 8-1 （续）

8.1.5 跟踪递归过程

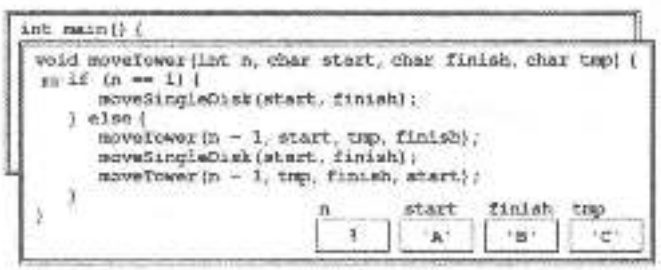
moveTower实现的唯一问题是它看起来很不可思议。如果你像大多数学生一样第一次学习递归，这个解决方案代码看起来如此简短，以至于你感觉一定有某些东西被遗漏了。圆盘塔的移动策略在哪里？计算机如何知道首先需要移动哪个圆盘，它移动到哪里？

答案正是递归过程（将一个问题分解成相同形式的更小的子问题，然后提供简单示例的解决方法），它是你所需要求解问题的根本。如果你相信递归的神奇跳跃，你已经完成任务了。你可以跳过这一章节，继续阅读下一章。另一方面，如果你对此仍有所怀疑，你可能就需要跟踪完整过程的步骤，看一看发生了什么。

为了使问题更易于控制，让我们考虑一下：如果原来的塔中只有3个圆盘会发生什么。因此，主程序的调用应该为如下语句：

```
moveTower(3, 'A', 'B', 'C');
```

为了跟踪这个调用在移动3个圆盘的塔所需的计算步骤，你所需要做的就是跟踪这个程序的执行过程，这和第7章中阶乘的例子所使用的策略完全相同。对于每个函数调用，你可以采用一个显示该调用参数值的栈帧。例如，初始调用moveTower创建了以下栈帧：



正如代码中的箭头所示，由于该函数正被调用，因此，函数体中第一条语句开始执行。n的当前值不等于1，这意味着程序向前跳至else分句并执行语句：

```
moveTower(n-1, start, tmp, finish);
```

与任何函数调用一样，第一步都是计算参数。为此，你需要求形参变量 n、start、tmp 和 finish 的值。每当你需要得到某个变量的值时，你就可以使用当前栈帧所定义的那个变量值。因此，moveTower 调用等价于： 357

```
moveTower(2, 'A', 'C', 'B');
```

然而，这个操作表明了另一个函数调用，这意味着当前操作被挂起，直到新的函数调用完成为止。为了跟踪新函数调用的操作，你需要生成一个新的栈帧并重复该过程。一如既往，新栈帧中的参数以它们出现的顺序从调用参数中复制而来。因此，新栈帧如下图所示：



如上图所示，新栈帧有它自己的一组参数，它们临时取代了前一个函数调用中的参数。因此，一旦程序执行到这个栈帧，n 的值就会变为 2，start 变为 'A'，finish 变为 'C'，tmp 变为 'B'。直到对 moveTower 的调用所代表的子任务完成，上层栈帧中的旧值才会出现。

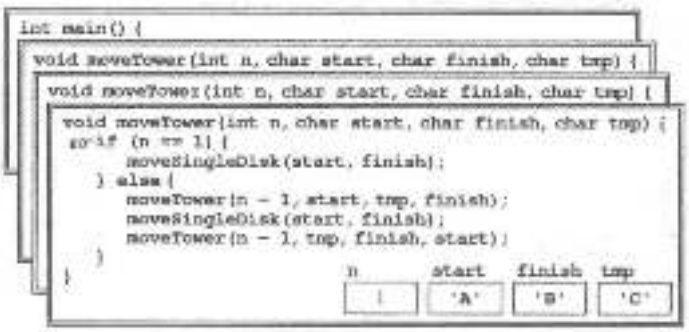
moveTower 的每层递归调用的执行过程都完全一样。n 再一次不等于 1，这需要另一个调用：

```
moveTower(n-1, start, tmp, finish);
```

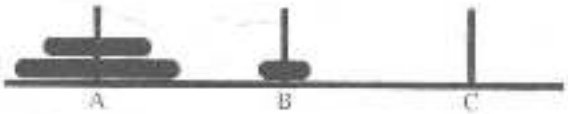
然而，由于这个调用来自另一个不同的栈帧，因此各变量的值便和原来调用中的变量值不同。如果你计算当前栈帧的参数，你会发现这个函数调用相当于以下函数调用：

```
moveTower(1, 'A', 'B', 'C');
```

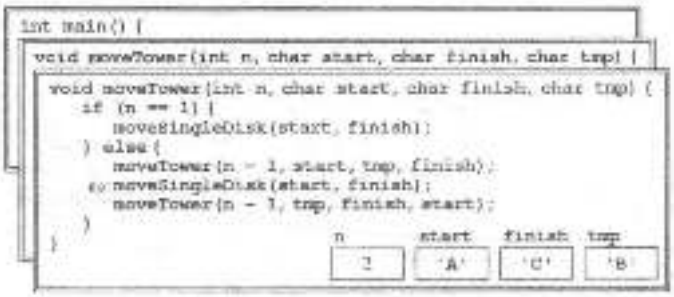
该调用的效果是引入了 moveTower 函数调用的另外一个栈帧，如下图所示： 358



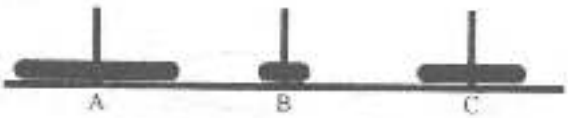
然而，这个 moveTower 调用的确表示简单情况。因为 n 等于 1，程序调用 moveSingleDisk 函数将一个圆盘从 A 移到 B，其问题帧结构如下：



此时，最近的moveTower函数调用完成并返回。在这一过程中，它的栈帧被丢弃，并返回到刚执行完的else语句后的第一条语句的栈帧，如下图所示：



moveSingleDisk调用再次代表一种简单操作，它使程序处于以下状态：



359

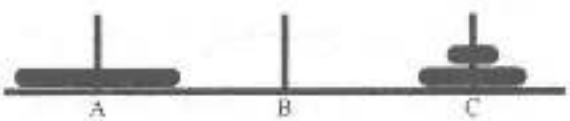
随着moveSingleDisk操作的完成，完成moveTower调用的唯一步骤是在函数中执行最后一条语句：

```
moveTower(n-1, tmp, finish, start);
```

对当前栈帧中的参数进行计算后，上述函数调用等价于如下语句：

```
moveTower(1, 'B', 'C', 'A');
```

这个函数调用需要创建一个新栈帧。然而，此时你应该能明白：该函数调用的结果就是利用塔柱A作为一个临时仓库，将1个圆盘的塔从塔柱B移到塔柱C。实质上，该函数求出n的值为1，然后调用moveSingleDisk使问题达到以下状态：



这个操作再次完成了一个moveTower调用，并将结果返回给调用它的子任务。这个子任务将2个圆盘的塔从塔柱A移到塔柱C。以上调用过程丢弃刚完成的子任务的栈帧，从而使栈帧变化成以下状态：



下一步是调用moveSingleDisk，将最大的圆盘从塔柱A移到塔柱B，这导致了圆盘的位置如下表所示：



剩下的唯一操作是调用以下语句：

```
moveTower(n-1, tmp, finish, start);
```

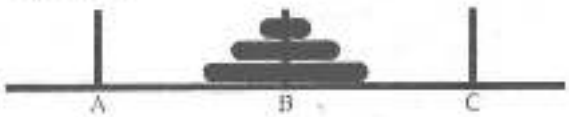
该调用的参数值来自当前栈帧，如下所示：

```
moveTower(2, 'C', 'B', 'A');
```

如果你仍对这个递归过程有所怀疑，你可以画出这个函数调用创建的栈帧，并持续跟踪它的调用过程以得出最终结论。然而，在某些地方，你可确信这个调用过程，并将函数调用看作是一个单独的具有以下英文指令作用的操作是非常重要的，该英文指令如下：

Move a tower of size 2 from C to B, using A as a temporary repository.

如果你用整体形式来思考这一过程，马上会发现完成这步将会使两个圆盘从塔柱C移到塔柱B，导致了我们所期望的结果：



8.2 子集求和问题

尽管汉诺塔问题精彩地展示了递归的威力，但作为一个例子，由于它没有任何实际应用，其效用受到了折损。许多人被编程所吸引，是因为编程能使他们解决实际问题。如果所有递归问题都像汉诺塔问题那样，那么很轻易就会得出这样一个结论：即递归仅适用于解决抽象问题。实际情况远非如此。递归策略产生了对于实际问题极其有效的解决方法（尤其是第10章介绍的排序问题），但用其他方法却很难解决。

本节所涉及的问题被称为**子集求和问题**（subset-sum problem），可定义如下：

给定一个整数集合和一个目标值，确定是否可以找到这个整数集合的一个子集，子集的和等于指定的目标值。

例如，给定集合{-2,1,3,8}和目标值7，子集求和问题的答案是“是”，因为子集{-2,1,8}的元素之和等于7。然而，如果目标值是5，答案将为“否”，因为没有一种方法能选择出整数集合{-2,1,3,8}的一个子集，其元素之和为5。

361

很容易就能将子集求和问题的想法翻译成C++语句。具体的目标是编写一个判定函数：

```
bool subsetSumExists(Set<int> & set, int target);
```

该函数获取所需要的信息，如果通过相加来自set的某些元素，可以产生值target，就返回true。

尽管子集求和问题可能开始看起来就像汉诺塔问题一样难懂，但它在计算机理论与实践方面都非常重要。正如你将在第10章所看到的，子集求和问题是一类重要的计算问题中的

一个实例，它难以有效地被解决。然而，这个事实绝不像子集求和问题在信息加密领域的应用中非常有用。例如，公钥密码学的第一种实现就使用了子集求和问题的一个变种来作为它的数学基础。现代的加密策略是通过将其操作附加在一个被证明很难的问题上，采用这样的密码难以破解。

8.2.1 寻找一个递归解决方案

采用传统的迭代方法很难求解子集求和问题。想要取得进展，你需要递归地思考问题。因此，一如既往，你需要确定一个简单情况和递归分解。在涉及集合的应用中，简单情况往往会在集合为空时发生。如果集合为空，没有任何一种方法将元素加起来能产生目标值，除非目标值为0。这个发现暗示 subsetSumExists 的代码将会像这样开始：

```
bool subsetSumExists(Set<int> & set, int target) {
   if (set.isEmpty()) {
      return target == 0;
   } else {
      Find a recursive decomposition that simplifies the problem.
   }
}
```

在这个问题中，困难的部分是找到递归分解的方法。

当你正在寻找一种递归分解时，你需要密切留意输入中的某些值（它们被转换成用C++表示的问题公式化的参数），你可以使这些参数变小。对于本问题的求解，你所需要做的就是使集合变小，因为你尝试做的就是向集合为空时发生的简单情况靠拢。如果你从某个集合
362 取出一个元素，则它就少了一个元素，集合变小。Set 类提供的操作能很容易地从一个集合中选取一个元素并确定所剩余的元素。你所需的是以下代码：

```
int element = set.first();
Set<int> rest = set - element;
```

first 方法返回集合中按迭代顺序排序的第一个元素，上述表达式涉及重载 - 操作，它产生了一个包含 set 集合中除去 element 值的所有其他元素的集合。element 在迭代顺序中排在第一个的事实在这里并不重要。你真正所需要的是选择一个元素，然后将你所选的元素从原来的集合中删除，创建一个更小的集合。

然而，使集合变小是不足以解决这个问题。就结构而言，你知道 subsetSumExists 必须在更小的集合中递归地调用自己，更小的集合当前存储在变量 rest 中。你还不能确定的是这些递归的子问题的解决方法是如何帮助解决原问题的。你所需使用的策略将在下一节中描述，并且将展示一个通用的编程模式，它已被证明在很多应用中是有用的。

8.2.2 包含 / 排除模式

为了完成 subsetSumExists 函数的实现，你所需的关键的洞察力是：在你确定了一个特定的元素后，可以采用两种方法产生期望的目标和。一种可能是你找到了包含该目标元素的子集。既然如此，很有可能是取出集合中的剩余元素并产生 target-element。另一种可能是你所寻找的子集不包含目标元素，那么很有可能仅使用了集剩余的元素产生值 target。对于完成 subsetSumExists 函数的实现，这种洞察力已经足够了。

```
bool subsetSumExists(Set<int> & set, int target) {
   if (set.isEmpty()) {
      return target == 0;
   } else {
      int element = set.first();
      Set<int> rest = set - element;
      return subsetSumExists(rest, target)
          || subsetSumExists(rest, target - element);
   }
}
```

因为这个递归策略将一般情形组分为两个分支，一种包含特定的元素，另一种不包含该特定元素，这种策略有时也称为**包含/排除模式**（inclusion/exclusion pattern）。当你做本章后面的习题时，你会发现这种相同的策略，它可能来自于各种各样不同的应用问题。尽管当你处理集合时，这种模式是最容易识别的，它也出现在一些涉及矢量和字符串的应用中，此时，你应该格外留意它。 363

8.3 字符排列

许多单词游戏和谜题需要能将一组字母重新排列以形成一个单词。因此，如果你想编写一个拼字游戏程序，具有对于一个给定的字符集合可以产生其所有的排列组合这一机制将会是非常有用的。在单词游戏中，这样的排列一般被称为**重组字**（anagram）。在数学上，它被称为**排列**（permutation）。

假设你想编写以下函数：

```
Set<string> generatePermutations(string str);
```

该函数返回一个包含字符串所有排列情况的集合。例如，如果你调用以下函数：

```
generatePermutations("ABC")
```

应该返回一个包含下列元素的集合：

```
{ "ABC", "ACB", "BAC", "BCA", "CAB", "CBA" }
```

你如何实现 generatePermutations 函数呢？如果你局限于迭代控制结构，找到一个通用的方法来处理任意长度的字符串是困难的。另一方面，递归地思考这个问题将会产生一个相对简单的解决方案。

对于递归程序而言，通常求解过程中最难的部分是如何将原问题分解成相同形式的，但却更简单的点问题的实例。此时，为了产生字符串的所有排列情况，你需要发现如何能够产生一个更短字符串的所有排列情况，这可能对于求解最终的问题有所帮助。

在你看到后面的解决方案之前，请停下来思考这个问题几分钟。当你第一次学习递归时，很容易看懂一个递归方案，并且你也能自己实现这一方案。然而，没有第一次尝试，你很难知道是否能提出必要的递归策略。

考虑一个实例将有助于给你自己更多的关于这个问题的感受。假设你想产生一个拥有 5 个字符的字符串的所有排列情况，例如 "ABCDE"，在你的解决方案中，你可以采用递归来产生任何较短的字符串的所有排列情况。假设递归调用起作用并且可以完成该任务。再一次强调：决定性的问题是排列较短的字符串怎样能够帮助你排列原来的 5 个字符的字符串。 364

如果你致力于将 5 个字符的字符串排列问题分解成若干 4 个字符的字符串排列实例，很

快就会发现5个字符的字符串"ABCDE"的排列包含下面的字符串：

- 字符'A'后面跟着"BCDE"的每一种可能的排列。
- 字符'B'后面跟着"ACDE"的每一种可能的排列。
- 字符'C'后面跟着"ABDE"的每一种可能的排列。
- 字符'D'后面跟着"ABCE"的每一种可能的排列。
- 字符'E'后面跟着"ABCD"的每一种可能的排列。

更宽泛地说，你可以通过顺序选择集合中的每个字符，构建包含长度为n的字符串的所有排列情况的集合，首字母有n种可能性，对于其中的每一种可能性，将选择的字符连接到剩下的n－1个字符的每一个可能的排列的开头。产生n－1个字符的所有的排列情况是相同类型的更小的问题，因此它可以递归地解决。

一如既往，你也需要定义一种简单情况。一种可能性是检测字符串是否包含一个单独的字符。计算含有1个字符的字符串的所有排列情况是很容易的，因为它只有一种可能的顺序。然而，在字符串处理过程中，简单情况的最好选择并不是处理含有1个字符的字符串的情况，因为有一个更简单的选择：即不包含任何字符的空字符串。正如只有1个字符的字符串，它只有一种顺序，也只有一种方法来书写空字符串。如果你调用generatePermutations(""),你可以得到一个包含一个元素的集合，这个元素就是空字符串。

一旦你有了简单情况和递归的洞察力，编写generatePermutations函数的代码就变得相当简单。generatePermutations函数的代码及其简单的测试程序显示在图8-2中，
365 该测试程序要求用户输入一个字符串，然后它输出该字符串中字符的每一种可能的排列情况。

```
/*
 * File: Permutations.cpp
 * ----------------------
 * This file generates all permutations of an input string
 */

#include <iostream>
#include "set.h"
#include "simpio.h"
using namespace std;

/* Function prototypes */

Set<string> generatePermutations(string str);

/* Main program */

int main() {
   string str = getLine("Enter a string: ");
   cout << "The permutations of \"" << str << "\" are:" << endl;
   for (string s : generatePermutations(str)) {
      cout << "  \"" << s << "\"" << endl;
   }
   return 0;
}

/*
 * Function: generatePermutations
 * Usage: Set<string> permutations = generatePermutations(str);
 * -------------------------------------------------------------
 * Returns a set consisting of all permutations of the specified string.
 * This implementation uses the recursive insight that you can generate
 * all permutations of a string by selecting each character in turn,
 * generating all permutations of the string without that character,
 * and then concatenating the selected character on the front of each
 * string generated.
 */
```

图8-2 产生一个字符串所有排列情况的程序

```
Set<string> generatePermutations(string str) {
   Set<string> result;
   if (str == "") {
      result += "";
   } else {
      for (int i = 0; i < str.length(); i++) {
         char ch = str[i];
         string rest = str.substr(0, i) + str.substr(i + 1);
         for (string s : generatePermutations(rest)) {
            result += ch + s;
         }
      }
   }
   return result;
}
```

图 8-2 （续）

366

如果你运行 Permutations 程序并输入字符串 "ABC"，你会看到以下输出：



在这个应用中，使用集合确保了程序按照字母表的顺序生成所有的排列情况，即使在输入的字符串中有重复的字母，每一种不同的字符顺序也只出现一次。例如，当你输入字符串 AABB，它仅生成下图所示的六种排列顺序：



递归过程中调用了 add 方法 24（4！）次，但是 Set 类的实现确保了不会出现重复值。

通过改变图 8-2 所示的主程序，你可以使用 generatePermutations 函数来产生某个单词的所有重组词，以便将每一个字符串与英语字典中的单词进行核对。假如你输入字符串 "aeinrst"，你会得到以下输出，一个认真的拼字游戏玩家可立刻识别的单词列表：



367

8.4 图的递归

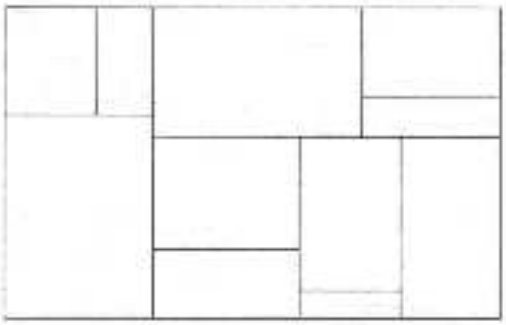
某些指令人兴奋的递归应用使用图来创建复杂的图片，其中以不同的规模来重复一个特定的主题。本章其余部分提供了几个关于图递归的实例，它们均利用了第 2 章结尾简要介绍

过的 GWindow 类。这些对于学习递归而言并不重要，如果你还没准备好使用图形库，可以跳过它。另外，学习这些例子将会使递归看起来更强大，而不是更有趣。

8.4.1 一个来自计算机艺术的实例

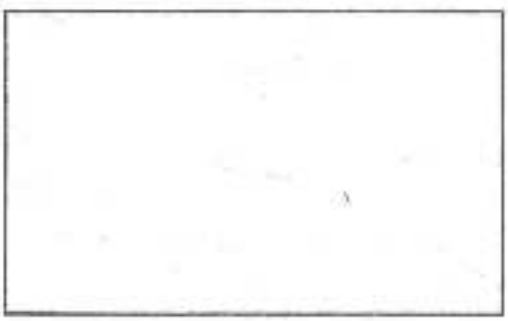
20 世纪初，在巴黎发生了一场有争议的艺术运动，这很大程度上受到了毕勃罗·毕加索和乔治·布拉克的影响。立体派（正如评论家对他们的称谓）拒绝了传统的关于透视法和表象主义的艺术概念，取而代之的是产生了基于简单的几何图形的高度分散的作品。在立体派的强烈影响下，荷兰画家彼特·蒙德里安（1872～1944）创造了一系列的基于水平和垂直线条的作品。这些图画的递归结构使得它们成为计算机仿真的理想候选者。

例如，假设你想产生一幅类似蒙德里安风格的作品，如下图所示：

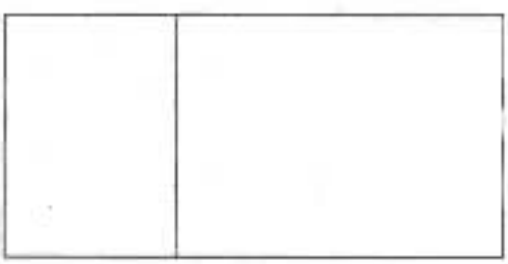


采用图形库，你如何设计一个一般的策略来创建这样一个图形？

将创建图的过程看作是一种连续的分解，将有助于理解一个程序是怎样产生这样一个图
368 形的。首先，画布只是一个空的矩形，如下图所示：



如果你想使用水平和垂直线条将画布细分，开始最容易的方法是画一条单独的直线将这个矩形一分为二：



如果你正在递归地思考这个问题，值得注意的是你现在有两个空的矩形画布，其中每个

画布在尺寸上都比原来的大画布小。细分这些矩形的任务和之前一样，所以你可以使用递归的相同的步骤去完成它。

完成一个递归策略唯一需要的是一个简单情况。分解矩形的过程不能无限地进行下去。随着矩形变得越来越小，在某一时刻这个过程必须停止。一种方法是在你开始之前，查看一下每个矩形的面积。一旦一个矩形的面积低于某个阈值，就不需要继续细分了。

图 8-3 中的 Mondrian.cpp 程序利用完整的图形窗口作为初始的画布，实现了这个递归算法。在 Mondrian.cpp 程序中，递归函数 subdivideCanvas 做了所有的工作。参 369
数给出了画布中当前矩形的位置和尺寸。在分解的每一步，该函数只是简单地检测所观察的矩形是否足够大进而能被再分解。如果是，函数将检测观察哪一个尺寸（宽或高）更大，并以此为依据使用垂直或水平的线条将矩形分开。对于上述每一种情况，函数只画一条单独的直线，图形中剩余的直线是在随后的递归调用中画出的。

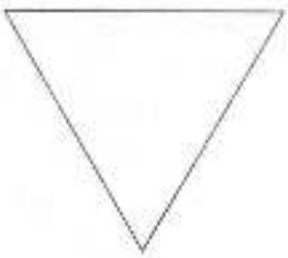
```
/*
 * File: Mondrian.cpp
 * ------------------
 * This program creates a line drawing in a style reminiscent of Mondrian
 */

#include <iostream>
#include "gwindow.h"
#include "random.h"
using namespace std;

/* Constants */

const double MIN_AREA = 10000;   /* Smallest square that will be split */
const double MIN_EDGE = 20;      /* Smallest edge length allowed       */

/* Function prototypes */

void subdivideCanvas(GWindow & gw, double x, double y,
                                   double width, double height);

/* Main program */

int main() {
   GWindow gw;
   subdivideCanvas(gw, 0, 0, gw.getWidth(), gw.getHeight());
   return 0;
}

/*
 * Function: subdivideCanvas
 * Usage: subdivideCanvas(gw, x, y, width, height);
 * ------------------------------------------------
 * Decomposes the specified rectangular region on the canvas recursively
 * by splitting that rectangle randomly along its larger dimension.  The
 * recursion continues until the area falls below the constant MIN_AREA
 */

void subdivideCanvas(GWindow & gw, double x, double y,
                                   double width, double height) {
   if (width * height >= MIN_AREA) {
      if (width > height) {
         double mid = randomReal(MIN_EDGE, width - MIN_EDGE);
         subdivideCanvas(gw, x, y, mid, height);
         subdivideCanvas(gw, x + mid, y, width - mid, height);
         gw.drawLine(x + mid, y, x + mid, y + height);
      } else {
         double mid = randomReal(MIN_EDGE, height - MIN_EDGE);
         subdivideCanvas(gw, x, y, width, mid);
         subdivideCanvas(gw, x, y + mid, width, height - mid);
         gw.drawLine(x, y + mid, x + width, y + mid);
      }
   }
}
```

图 8-3 使用了类似蒙德里安风格将平面细分的程序

8.4.2 分形

20世纪70年代末，一名IBM的研究员本华·曼德博（1924~2010），出版了一本关于分形的书籍，该书引起了人们极大的兴趣。分形是相同的模式在许多不同的尺度下被重复使用的几何结构。尽管数学家了解分形很长一段时间了，但在20世纪80年代又重新引起了人们对它的兴趣，部分原因是：由于计算机的发展，人们对分形可以做出以前更多的事情。

最早的分形例子就是海里格·冯·科赫（1870~1924）发明的**科赫雪花分形**（Koch snowflake）。科赫雪花分形是以一个等边三角形开始的，如下图所示：

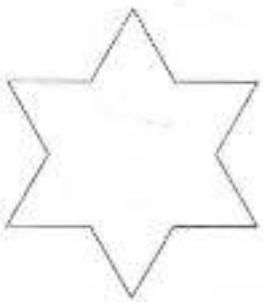


每条边都为直线的三角形被称为0阶科赫曲线。然后逐步修正这个图以产生相继更高阶的分形。在其中的每个阶段，图中的每一条线段都被一个分形所代替，其中中间的第二部分包含一个从图中向外突出的三角形凸块。因此，第一步将三角形中的线段用这样的线来代替，如下图所示：

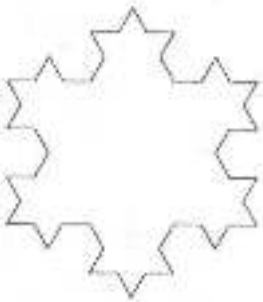


370
:
371

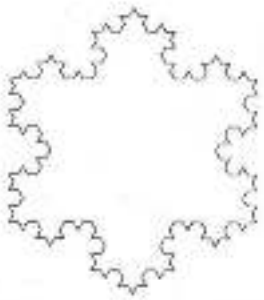
将这种转换应用到原三角形的每一条边，产生1阶科赫雪花分形，如下图所示：



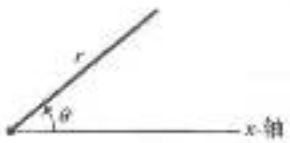
然后，如果你用一条新的线来替换这个图中的每一条线段，新的线将再次包含一个三角楔形，你就创建了以下2阶的科赫雪花分形：



继续替换所有这些线段，会产生3阶分形，看起来更像一片雪花，如下图所示：



因为像科赫雪花分形这样的图用计算机比用手更容易画出来，因此编写这样一个程序是有意义的，该程序使用了 graphics.h 接口提供的工具来产生这个设计。尽管可以仅 372
使用 drawLine 方法来画不规则的雪花图形，但使用 GWindow 类中的 drawPolarLine 方法通常显得更容易，它能让你确定一条线的长度和方向。在数学中，一条分割线的长度和方向按照惯例是用符号 r 和 θ 表示，它被称为极坐标（polar coordinate）。下图展示了极坐标的用法，其中实线长度为 r，并从起点向外延伸，实线的位置是沿 x 轴逆时针旋转 θ 度：



drawPolarLine 方法读取起点坐标（无论是作为一个单独的坐标还是作为一个 GPoint 对象），并返回线段另一个端点的坐标，从而更容易将连续的线段连接在一起。例如，下面的代码画出了一个向下的等边三角形，它的左上角处于 pt 的原始值处：

```
pt = gw.drawPolarLine(pt, size, 0);
pt = gw.drawPolarLine(pt, size, -120);
pt = gw.drawPolarLine(pt, size, +120);
```

这段代码创建了0阶雪花分形。为了产生更高阶的分形，你需要将 drawPolarLine 调用以一个名为 drawFractalLine 的函数替换，该函数拥有（除了图形窗口）一个额外的参数表示分形线的阶数，如下所示：

```
pt = drawFractalLine(gw, pt, size, 0, order);
pt = drawFractalLine(gw, pt, size, -120, order);
pt = drawFractalLine(gw, pt, size, +120, order);
```

剩下的唯一任务就是实现 drawFractalLine 函数，如果你能递归地思考，它很容易完成。当 order 为0时，drawFractalLine 的简单情况就会发生，此时，函数只需简单地画一条确定了长度和方向的直线。如果 order 大于0，分形线就会被分成四部分，每一部分都是一条更低阶数的分形线。像 drawPolarLine 函数一样，drawFractalLine 函数返回它画的上一条线段的终点，这样下一条分形线可以在上一条分形线结束的地方开始。Snowflake 程序的完整实现显示在图 8-4 中，其中包含了 drawFractalLine 的完整代码。 373

```
/*
 * File: Snowflake.cpp
 * -------------------
 * This program draws a Koch fractal snowflake.
 */

#include <iostream>
#include <cmath>
#include "gwindow.h"
using namespace std;

/* Constants */

const double SIZE = 200;          /* Size of the order 0 fractal in pixels */
const int ORDER = 4;              /* Order of the fractal snowflake        */

/* Function prototypes */

GPoint drawFractalLine(GWindow & gw, GPoint pt,
                       double r, double theta, int order);

/* Main program */

int main() {
   GWindow gw;
   cout << "Program to draw a snowflake fractal." << endl;
   double cx = gw.getWidth() / 2;
   double cy = gw.getHeight() / 2;
   GPoint pt(cx - SIZE / 2, cy + sqrt(3.0) * SIZE / 6);
   pt = drawFractalLine(gw, pt, SIZE, 0, ORDER);
   pt = drawFractalLine(gw, pt, SIZE, -120, ORDER);
   pt = drawFractalLine(gw, pt, SIZE, +120, ORDER);
   return 0;
}

/*
 * Function: drawFractalLine
 * Usage: GPoint end = drawFractalLine(gw, pt, r, theta, order);
 * -------------------------------------------------------------
 * Draws a fractal edge starting from pt and extending r units in direction
 * theta.  If order > 0, the edge is divided into four fractal edges of the
 * next lower order.  The function returns the endpoint of the line.
 */

GPoint drawFractalLine(GWindow & gw, GPoint pt,
                       double r, double theta, int order) {
   if (order == 0) {
      return gw.drawPolarLine(pt, r, theta);
   } else {
      pt = drawFractalLine(gw, pt, r / 3, theta, order - 1);
      pt = drawFractalLine(gw, pt, r / 3, theta + 60, order - 1);
      pt = drawFractalLine(gw, pt, r / 3, theta - 60, order - 1);
      return drawFractalLine(gw, pt, r / 3, theta, order - 1);
   }
}
```

374

图 8-4 画科赫雪花分形的程序

本章小结

本章引入了较少的新概念，因为递归的基本思想已经在第7章中介绍过了。本章的重点是介绍递归例子的复杂之处，这些问题用其他方法很难解决。鉴于这些问题较高的复杂性，学生刚开始学的时候经常会发现这些问题比前面章节中遇到的问题更难理解。这些问题的确更难，但是递归就是一个解决难题的工具。为了掌握它，你需要针对这个级别的复杂性问题进行实战。

本章的重点包括：

- 每当你想要将递归应用到一个程序问题中时，你必须设计一种策略，将原问题转换成相同形式的更简单的问题。当你找到了一个递归策略的切入点，才会有办法应用递归的知识。

- 一旦你定义了一个递归方法，对你而言，重要的是：检测你的策略并确保它不会违反问题施加的任何条件。
- 当你尝试解决的问题的复杂性有所增加时，接受递归的稳步跳跃的重要性也会增加。
- 递归并不神奇。如果你需要这样做，可以通过画出每个过程的栈帧内容来模拟计算机的操作，这些栈帧在问题解决的过程中被调用。另一方面，不要报以怀疑的态度是至关重要的，因为怀疑将会强迫你去看所有基本的细节。

复习题

1. 用你自己的话，描述解决汉诺塔问题必要的递归思路。
2. 下面这种解决汉诺塔问题的策略在结构上和书中所使用的策略相似：

   a. 将最上面的圆盘从开始的塔柱移到临时塔柱上。

   b. 将 N - 1 个圆盘从开始的塔柱移到最终的塔柱上。

   c. 将当前处于临时塔柱上的圆盘移回到最终的塔柱上。

   为什么这个策略失败了？
3. 如果你调用

   ```
   moveTower(16, 'A', 'B', 'C')
   ```

   375

   作为解决方案的第一步，moveSingleDisk 将会显示什么语句？这个解决方案的最后一步是什么？
4. 什么是排列？
5. 用你自己的话，解释一下枚举一个字符串中字符排列情况所需的递归的思路。
6. 字符串 "axyz" 有多少种排列情况？
7. Mordrian.cpp 中，什么样的简单情况终止了递归过程？
8. 画出 1 阶雪花分形图。
9. 有多少条线段出现在 2 阶雪花分形中？

习题

1. 遵循 moveTower 函数的逻辑，编写一个递归函数 countDiscMoves(n)，它计算解决 n 个圆盘的汉诺塔问题所需要移动圆盘的次数。
2. 为了使程序的操作稍微容易解释，在本章，moveTower 的实现使用了：

   ```
   if (n == 1)
   ```

   作为它的简单情况测试。当你看到一个递归程序使用 1 作为它的简单情况时，这有点值得怀疑；在大部分应用中，0 是一个更合适的选择。重写汉诺塔程序，使得 moveTower 的检测条件为 n 是否等于 0。此时，moveTower 实现代码的长度会发生什么变化？
3. 重写汉诺塔程序，它使用一个显式的待处理任务栈来代替递归。此时，一个任务最容易表示为包含将要移动的圆盘的数目，作为开始、结束和临时仓库的塔柱的名称的一种结构。在这个过程的一开始，你的栈放置了一个移动整个塔的任务。然后，程序反复出栈并执行任务，直到在栈中发现没有剩余的任务为止。除了简单情况，执行一个任务的过程导致了更多任务的创建，这些任务将被放进栈中留于后来的执行。
4. 在 8.2 节介绍的子集求和问题中，通常有几种方法可以产生期望的目标值。例如，给定集合 {1,3,4,5}，有两种不同的方法产生目标值 5：

   376

   - 选择 1 和 4。
   - 只选择 5。

   相比之下，没有办法来划分集合 {1,3,4,5} 得到 11。

编写一个函数：

```
int countSubsetSumWays(Set<int> & set, int target);
```

它返回对于给定的集合为产生目标值你能选择的子集数。例如，假设 sampleSet 已经被如下初始化：

```
Set<int> sampleSet;
sampleSet += 1, 3, 4, 5;
```

给定 sampleSet 的定义，调用

```
countSubsetSumWays(sampleSet, 5);
```

应该返回 2（有两种方法可以得到 5）。调用

```
countSubsetSumWays(sampleSet, 11)
```

应该返回 0（没有方法得到 11）。

5. 编写程序 EmbeddedWords，它能找出所有的英语单词，这些单词可以通过提取来自一个给定的初始单词的字母的子集来组成。例如，给出初始的单词“happy”，你一定可以产生单词“a”、“ha”、“hap”和“happy”，其中字母是连续出现的。你也可以产生单词“hay”和“ay”，因为这些字母在 happy 中是以从左到右的顺序出现的。然而，你却不能产生单词“pa”或“pap”，因为这些字母（即使它们出现在单词中）并不是以正确的顺序出现的。程序的一个示例运行结果如下图所示：



377

6. 我是父母唯一的进行权衡、度量和定价一切的孩子；
对他们而言，任何东西都不能被权衡、度量、定价和存在。

——查尔斯·狄更斯，《小杜丽》(*Little Dorrit*)，1857

在狄更斯时代，商人使用砝码和有两个盘子的天平来称量很多商品——今天，这个惯例在世界许多地方仍持续着。然而，如果你使用有限的一组砝码，只能称量一定的数量。例如，假设你只有两个砝码：一个 1 盎司[⊖]的砝码和一个 3 盎司的砝码。有了这两个砝码，你可以轻易地称量出 4 盎司，如下图所示：



一个更有趣的发现是：通过将 1 盎司的砝码放在天平的另一边，你也可以称量出 2 盎司，如下图所示：



⊖ 1 盎司 = 28.3495 克。

编写一个递归函数

```
bool isMeasurable(int target, Vector<int> & weights)
```

该函数的作用是利用给定的一组砝码（它们存储在矢量 weights 中），判断是否可以称量出期望的目标数量。

例如，假设 sampleWeights 函数已经如下所示被初始化：

```
Vector<int> sampleWeights;
sampleWeights += 1, 3;
```

根据这些值，函数调用

```
isMeasurable(2, sampleWeights)
```

应返回 true，因为它可以使用前面图形中所示的砝码的放法，称量出 2 盎司。然而，调用

```
isMeasurable(5, sampleWeights)
```

378

应返回 false，因为它不可能使用 1 盎司和 3 盎司的砝码称量出 5 盎司。

7. 在被称为**克里比奇**（Cribbage）的纸牌游戏中，一副五张扑克牌加起来的分数组成了游戏的一部分。分数的其中一个组成部分是不同的扑克牌组合中的各个扑克牌上的分值，要求这些扑克牌组合其扑克牌上的分值之和等于 15，A 的分数记为 1，所有的花牌（J，Q 和 K）分数记为 10。例如，考虑下面的扑克牌：



扑克牌分值之和等于 15 有三种不同的组合，如下所示：

AD＋10S＋4H　　AD＋5C＋9C　　5C＋10S

考虑第二种示例，组合中的扑克牌如下图所示：



它包含以下八种不同组合，它们的扑克牌分值之和都等于 15：

5C＋JC	5D＋JC	5H＋JC	5S＋JC
5C＋5D＋5H	5C＋5D＋5S	5C＋5H＋5S	5D＋5H＋5S

编写一个函数：

```
int countFifteens(Vector<Card> & cards);
```

它接收一副扑克牌值的矢量（正如第 6 章习题 2 中定义的），并返回你可以从这副牌中选取其分值之和为 15 的组合方法的数量。解决该问题，你不需要对 Card 类了解太多。你唯一需要的是 getRank 方法，它返回每张扑克牌所对应的排序整数值。你可以假设 card.h 接口提供了名为 ACE、JACK、QUEEN 和 KING 的常量，它们的值分别为 1、11、12 和 13。

379

8. 第 8 章展示的递归分解对于解决产生排列的问题并不是唯一有效的策略。另一种实现递归的方法如下所示：

a）去除字符串中的首字母，并将其存储在变量 ch 中。

b）产生包含剩余字符所有排列情况的集合。

c) 将 ch 插入到这些排列情况中每一个可能的位置，形成一个新的集合。

重写 Permutations 程序，要求它使用这个新的策略。

9. 设计实现 Permutations 程序的策略是为了遍历它递归的字符。最终的代码并非特别有效，大部分是因为它以产生集合结束，并且这些集合随后就被抛弃。也因为它应用了类似 substr 的方法，该方法要求复制字符串中的字符。使用下面的递归公式表述可以消除上述的无效性：

a) 在每一层，传递整个字符串以及一个索引，该索引表示排列过程的开始位置。字符串中在该索引前面的子字符串保持不变；而在这个索引位置上或在索引位置后的字符必须出现在它们所有的排列情况中。

b) 当索引到达了字符串的结尾，简单情况就发生了。

c) 递归操作的过程就是将字符串中索引位置处的字符和其他字符相交换，然后产生下一个更高索引位置开始的每一种排列情况，然后将字符交换回去确保原来的顺序能被还原。

使用上述策略实现函数：

```
void listPermutations(string str);
```

该函数在标准输出设备上列举了字符串 str 的所有排列情况。并且不使用任何集合，除了 length 和选择方法之外，也不使用其他字符串方法。listPermutations 函数本身必须是一个包装器函数，它作为包含索引的第二个函数。

如果你不考虑字符串中的重复字符，这个函数相对容易实现。只有当你改变算法的结构时，有趣的挑战就会产生，这样它列举每一个唯一的排列一次，并且不使用集合来完成该任务。然而，你
180 不应该担心 listPermutations 输出的顺序。

10. 在一部手机键盘上，数字被映射到字母表上，如下图所示：



为了使它们的电话号码更难忘记，服务供应商喜欢寻找能够拼写出一些单词（称之为记忆术）的数字，这些单词适合他们的生意，使得电话号码更容易记住。

想象一下，你刚被一家本地的手机公司雇用，要求你编写一个 listMnemonics 函数，该函数将产生对应于一个给定数字的所有字母组合情况，数字用一个数字字符串表示。例如，调用

```
listMnemonics("723")
```

后应列举出以下对应于前缀的 36 种可能的字母排列情况：

```
PAD PBD PCD QAD QBD QCD RAD RBD RCD SAD SBD SCD
PAE PBE PCE QAE QBE QCE RAE RBE RCE SAE SBE SCE
PAF PBF PCF QAF QBF QCF RAF RBF RCF SAF SBF SCF
```

11. 重写习题 10 的程序，要求它使用 Lexicon 类和 englishWords.dat 文件，以便程序只列举有效的英语单词的记忆方法。

12. 现在，手机键盘上的字母并不是用于增进记忆，而是为了发短信。使用键盘输入文本是有问题的，因为键盘上的键要比字母表上的字母少很多。一些手机使用了一个“多按式”的用户界面，你可以按一次“2”键得到 a，按两次得到 b，按三次得到 c，它们感觉冗长乏味。一个精简的选择是使用一个预测的策略，手机使用这个策略猜测你打算输入的字母，它是以到目前为止的输入序列和

它可能的完成概率为基础的。 381

例如，如果你要打出数字序列72，有12中可能性：pa，pb，pc，qa，qb，qc，ra，rb，rc，sa，sb和sc。只有这四对字母（pa，ra，sa和sc）看起来有希望，因为它们都是普通的英语单词（像party，radio，sandwich和scanner）的前缀。其他的可以被忽略，因为没有常见的单词是以这些字母序列开始的。如果用户输入"9956"，有144（4×4×3×3）种可能的字母序列，但是你可以确定用户表示的是xylo的意思，因为只有唯一的一个序列是英语单词的前缀。

编写函数：

```
void listCompletions(string digits, Lexicon & lex);
```

输出字典中的所有单词，要求这些单词可以通过扩展给定的数字序列来形成。例如，调用

```
listCompletions("72547", english)
```

后应该产生下面的示例输出：



如果你只关心获取答案，解决这个问题最容易的方法是迭代字典中的单词，输出和指定的数字序列匹配的每个单词。这个方法不需要递归，也不怎么用思考。然而，你的经理认为浏览字典中的每个单词速度太慢，他要求你只使用字典一次来检测给出的字符串是否是一个单词或是一个英语单词的前缀。有了这个约束，你需要解决如何从数字字符串中生成所有可能的字母序列。这个任务很容易递归地解决。

13. 很多蒙德里安的几何图画用一些颜色来填充矩形区域。扩展文中的Mondrian程序，使它可以使用随机选择的颜色来填充创建的矩形区域的某些部分。 382

14. 像美国这样的国家仍使用传统的英国测量系统，一个直尺上的每个英寸[⊖]都是使用刻度标记划分的，如下图所示：



最长的刻度标记处在半英寸的位置，两个较小的刻度标记表示四分之一英寸，更小的刻度标记被用来标记八分之一英寸和十六分之一英寸。编写一个递归程序，在图形窗口的中间画一条1英寸的直线，然后画出如上图所示的刻度标记。假设表示半英寸位置的刻度标记的长度已经由常数定义给出：

```
const double HALF_INCH_TICK = 0.2;
```

并且每一个更小的刻度标记的尺寸是下一个更大刻度标记的一半。

15. 分形引起人们如此大兴趣的一个原因是：它们被证明在一些意外的现实语境中很有用。例如，最成功的技术是由计算机的山图像和确定的其他景观特征，它们都涉及分形几何学的使用。

作为这个问题提出时的一个简单的例子，考虑一下将两个点A和B用一个分形连接的问题，它们看起来就像一张地图上的海岸线。最简单的可行策略是在两点之间画一条直线：

⊖ 1英寸= 0.0254米。



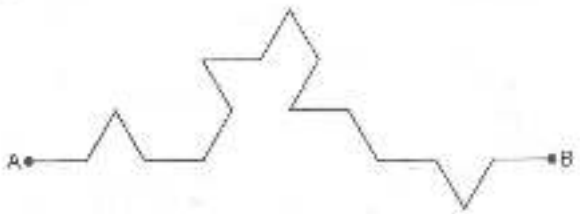
这个 0 阶海岸线，代表途旦的基本情况。

当然，现实的海岸线沿着它的长度在某些地方可能会有小半岛或入口，所以你期待一种更现实的面孔，画出的海岸线偶尔向内或向外突出。作为第一次近似，你可以使用相同的被用于创建雪花分形的分形线来代替直线，如下图所示：

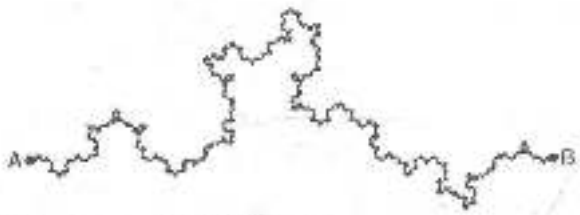


383 这个过程创建了 1 阶海岸线。然而，海岸线中的锯齿形状并不总是指向同一个方向。因此，画一些右时向上或右时向下的三角锯形是很重要的，假设它们以等概率出现。

然后，如果你使用一条随机方向的分形线代替 1 阶分形中的每一条直线段，你就得到了 2 阶海岸线，如下图所示：



继续这个过程，直到产生一幅能表达出某种现实意义的图画，如 5 阶海岸线：

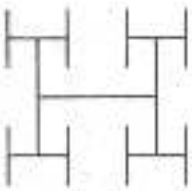


编写一个程序，在图形窗口中画一条分形的海岸线。

16. 如果你搜索网络上的分形设计，你会发现往往比本章展示的科赫雪花分形更复杂的图形。一个是 H- 分形（H-fractal），其中重复图案的形状像一个正方形中拉长的字母 H。因此，0 阶的 H- 分形如下图所示：

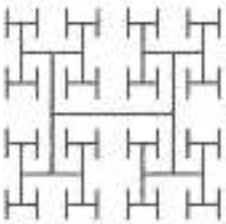


为了创建 1 阶分形，你所需要做的是在 0 阶分形的每个开口端添加四个新的 H- 分形（每个分形尺寸都是原分形的一半），如下图所示：



384

为了创建2阶分形，你仅需要将更小的H-分形（尺寸是它们所连接的分形的一半）添加到每一个开放的端点。这个过程产生了下面的2阶分形：



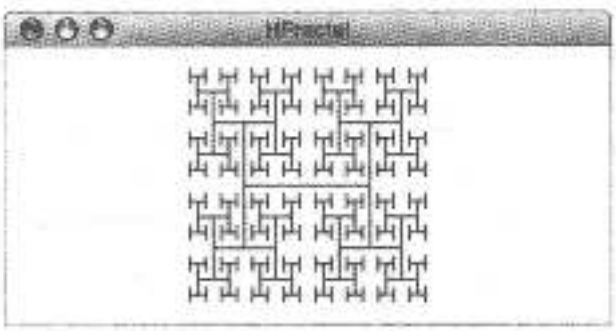
编写一个递归函数：

```
drawHFractal(GWindow & gw, double x, double y,
                          double size, int order);
```

其中x和y是H-分形的中心坐标，size指定了宽度和高度，order表示分形的阶数。作为一个例子，以下主程序：

```
int main() {
   GWindow gw;
   double xc = gw.getWidth() / 2;
   double yc = gw.getHeight() / 2;
   drawHFractal(gw, xc, yc, 100, 3);
   return 0;
}
```

将会在图形窗口的中间画出3阶分形，如下图所示：

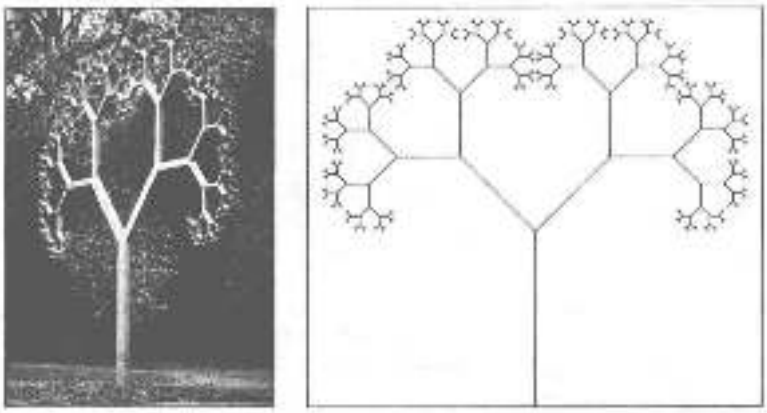


17. 2008年，为了庆祝牛津大学莫德林学院550周年庆，他们委托美国艺术家马克·涅林格创建一个名为Y的雕塑，它有一个明确地递归结构。这个雕塑的照片出现在图8-5的左边，展示它的分形设计的图出现在右边。鉴于它的分支结构，在涅林格的雕塑中，基本的图案被称作一个分形树（fractal tree）。树是以一个简单的树干开始的，树干使用一条直的垂直线表示，如下图所示： 385

这些分支本身可以分裂形成新的分支，这些新的分支又可以分裂形成新的分支。



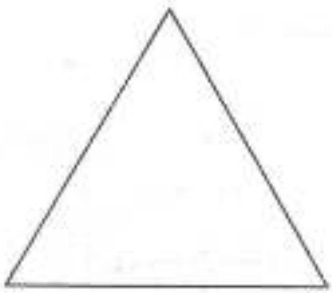
编写一个程序，利用图形库画出课林格的雕塑中的分形树。如果你将这个过程应用到 8 阶分形上，会得到图 8-5 中右边的图形。



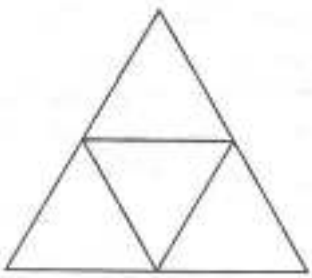
386

图 8-5 马克·课林格分形树

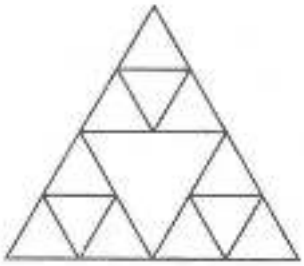
18. 另一个有趣的分形是谢尔宾斯基三角形（Sierpinski Triangle），它是以发明者瓦茨瓦·谢尔宾斯基（1882—1969）命名的。0 阶的谢尔宾斯基三角形是一个等边三角形：



为了创建一个 N 阶的谢尔宾斯基三角形，你可以画三个 N－1 阶谢尔宾斯基三角形，它们每一条边的长度都是原三角形的一半。这三个三角形被放在大三角形的三个角上。这意味着 1 阶的谢尔宾斯基三角形如下图所示：

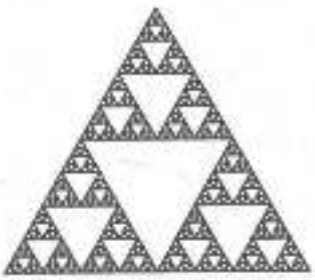


该图中间向下的三角形并不是明确地画出来的，而是由其他三个三角形的边构成的。而且，面积也不是递归地被细分，它在每一层的分形分解中将保持不变。因此，2 阶的谢尔宾斯基三角形在图中同有着相同的开放的区域：



387

如果你在三个递归层上继续这个过程，你将得到 5 阶谢尔宾斯基三角形，如下图所示：



编写一个程序，要求用户输入一条边的长度和分形的阶数，并在图形窗口中间画出最终的谢尔宾斯基三角形。

388

第9章 |
Programming Abstractions in C++

回溯算法

发现真理需要的是探索，而不是汇报。真理总是来源于实践。

389 ——西蒙娜·韦伊，《纽约札记》(*The New York Notebook*)，1942

对于现实世界中的许多问题，其解决过程是由你自己的一系列选择构成的，每一次选择将导致你在某些路上走得更远。如果你做出了一组正确的选择，最终会解决问题。然而，如果你走进死胡同或者发现你某些地方做出了错误的选择，那么将不得不回溯到前面的某个选择，并且尝试另一条不同的路径。采用这种方式的算法被称为回溯算法（backtracking algorithm）。

如果你把回溯算法当作是一条重复探索路径直到遇到问题的求解方案，那么其过程似乎具有某种迭代特性。然而，目前大多数这种形式的问题采用递归方法更容易解决。递归的基本思想很简单：一个回溯问题有一个解决方案，当且仅当至少有一个更小的回溯问题有其解决方案，这些更小的回溯问题又来源于所做的每一种可能的初始化选择。本章的示例旨在阐明这一过程，并且展示递归算法在这一领域的威力。

9.1 迷宫的递归回溯

很久很久以前，在希腊神话时代，地中海的克里特岛被暴君迈诺斯统治着。迈诺斯不时地向雅典索取年轻的男女作为贡品来祭祀弥诺陶洛斯（一个人身牛头的怪物）。为了安置这个致命的怪物，迈诺斯迫使他的侍者代达鲁斯（一个工程天才，后来通过建造一对翅膀而逃脱）在克诺索斯建造了一个广阔的地下迷宫。那些来自雅典的年轻的贡品将被引入迷宫，并在逃出迷宫之前被弥诺陶洛斯吃掉。这个悲剧一直延续到雅典的特休斯自愿成为其中的一个贡品为止。听从了迈诺斯的女儿阿里阿德涅的建议，特休斯带着一把剑和一个线团进入了迷宫。在屠杀了怪物以后，他凭借着解开的线索独自沿原路返回。

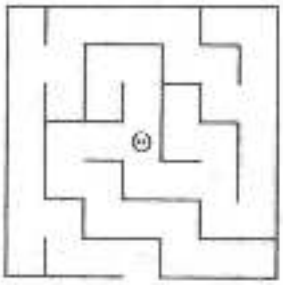
9.1.1 右手法则

阿里阿德涅的策略是逃离迷宫的一种方法，但不是每个困在迷宫里的人都有幸拥有一卷线子。幸运的是，逃出迷宫还有其他的可取之径。在这些策略中，最著名的方法被称为右手法则（right-hand rule），它可以用下面的伪码表示：

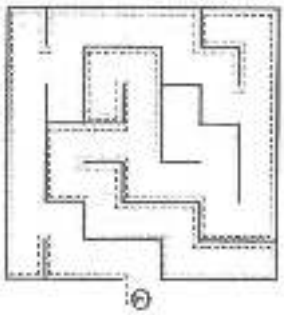
```
Put your right hand against a wall
while (you have not yet escaped from the maze) {
   Walk forward keeping your right hand on a wall
}
```

390

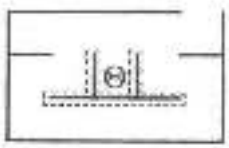
为了可视化右手法则的操作，假设特休斯现在已经成功杀死了弥诺陶洛斯，并且现在他站在第一幅图中由特休斯名字的第一个字母（即希腊字母 Θ）标记的位置：



如果特休斯把他的右手放在墙上，然后从那儿开始遵循右手法则，他将探寻出下图由虚线勾勒的出路：



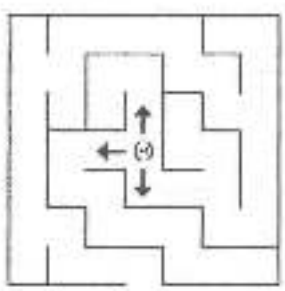
遗憾的是，右手法则并不是在每一个迷宫中都奏效。如果在开始位置周围有一个循环，特休斯将会落入无限循环中，正如下面简单的迷宫示例展示的一样：



9.1.2 寻找一种递归方法

伪码中 while 循环清楚地表明：右手法则是一种迭代策略。然而，你也可以从递归的角度思考一下解决迷宫问题的过程。为此，你必须采取一种不同的思维模式。你可以从不再寻找一条完整的路径来思考这个问题。取而代之的是，你的目标是寻找一个递归策略来简化问题，一步一个脚印。一旦简化了这个问题，你就可以用相同的过程解决该问题所产生的每一个子问题。 391

让我们回到最初用到右手法则的迷宫图上，即把你的右手放在特休斯的位置。从最开始的布局开始，你有三个选择，如下图箭头所示：



如果存在出口，它一定就是这些出路中的其中一条。此外，如果你选择了正确的方向，你将离解决问题更进一步。那么沿着这条路，迷宫也会因此变得更简单，这才是递归解法的关键。这种观察指出了必不可少的递归思维。当且仅当至少能够解决图 9-1 中的一个新迷宫，那么原始的迷宫便可得以解决。在每个迷宫的 × 标记着原始出发的方格，而且它作为之后的任何递归解法的禁区，因为最佳的解法永远不需要回溯到这个方格。

如果你观察图 9-1 中所示的迷宫，很容易看到（至少以你的全局视角）二级迷宫标签（a）和（c）表示一条死路，唯一的解法开始于二级迷宫（b）所示的方向。然而，如果你递归地思考，就不需要不断地分析直到找到所有的解法。你已经将问题分解成了更简单的问题。你所需要做的就是依赖递归的力量去解决一个个子问题，然后你就大功告成了。你仍然必须确定一系列的简单情况，从而递归才能结束，但是困难的工作已经完成了。

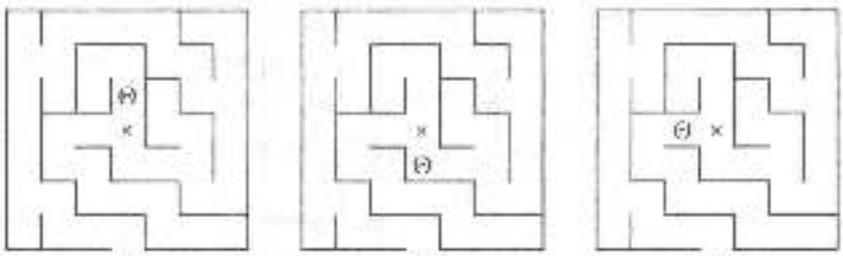
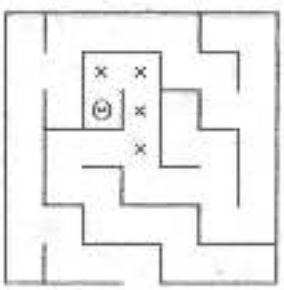


图 9-1 迷宫的递归分解

9.1.3 确定简单情况

迷宫问题中，什么构成了简单情况？一种可能是你已经站在迷宫外面了。如果这样的话，你已经完成了。显然，这一处境代表一种简单情况。然而，还有其他的可能性。你也可能会到达一个死胡同，此时，你不能往其他地方移动了。例如，如果你尝试解决下图的迷宫实例，通过向北移动，然后沿着那条路一直调用递归，你将最终站在你尝试解决的下面的迷宫中的位置：



此时，你已经尝试完所有可移动的空间了。从新的位置延伸出来的每一条路要么被标记了，要么被墙挡住了，很明显，从这个点出发，迷宫是没有解的。因此，迷宫问题有了第二种简单情况，即前面方格的每一个方向都被堵住了，要么是一堵墙，要么是一个标记过的方格。

当你考虑可以移动的方向时，你在这些方格上进行递归调用，而不是检查已标记过的方格，那将很容易对递归算法进行编码。在程序开始时，如果你检查当前的方格是否被标记过，就可以以此作为条件来终止递归。毕竟，如果你发现自己站在一个被标记过的方格上，

就必须折返，这意味着最佳的解决肯定在其他方向上。

因此，此问题的两种简单情况如下所示：

1. 如果当前的方格在迷宫之外，那么迷宫问题就解决了。
2. 如果当前方格被标记过，那么迷宫问题没有解决，至少目前沿着你选的这条路不行。

392 / 393

9.1.4 对迷宫问题的解决算法进行编码

尽管在概念层面上，用来解决问题的递归思维和简单情况是你所需要的，但编写一个完整的程序来导航迷宫也需要你考虑许多实现的细节。例如，你需要决定迷宫本身的表示，标出端的位置，记录当前位置，表示一个特定的方格被标记过，并且判断你是否已经逃离迷宫。虽然为迷宫设计一个合适的数据结构本身是一个有趣的编程挑战，但是这与我们理解递归算法——我们讨论的焦点，关系并不大。但是如果数据结构的细节设计得不好，也很可能让你迷失，并且使你在整体上理解算法策略变得更加困难。幸运的是，通过引入一个新的接口，我们可以把数据结构的细节放置一边来隐藏复杂性。图 9-2 的 maze.h 接口提供了一个被称为 Maze 的类，它将所有在迷宫中记录通路以及将迷宫展现在图形窗口的必要信息封装在其中。

```
/*
 * File: maze.h
 * ------------
 * This interface exports the Maze class.
 */

#ifndef _maze_h
#define _maze_h

#include <string>
#include "grid.h"
#include "gwindow.h"
#include "point.h"

/*
 * Class: Maze
 * -----------
 * This class represents a two-dimensional maze contained in a rectangular
 * grid of squares.  The maze is read from a data file in which the
 * characters '+', '-', and '|' represent corners, horizontal walls, and
 * vertical walls, respectively; spaces represent open passageway squares.
 * The starting position is indicated by the character 'S'.  For example,
 * the following data file defines a simple maze:
 *
 *       +-+-+-+-+-+
 *       |     |
 *       + +-+ + +-+
 *       |S  |     |
 *       +-+-+-+-+-+
 */

class Maze {

public:

/*
 * Constructor: Maze
 * Usage: Maze maze(filename);
 *        Maze maze(filename, gw);
 * ----------------------------
 * Constructs a new maze by reading the specified data file.  If the
 * second argument is supplied, the maze is displayed in the center
 * of the graphics window.
 */

   Maze(std::string filename);
   Maze(std::string filename, GWindow & gw);
```

图 9-2 maze.h 接口

```
/*
 * Method: getStartPosition
 * Usage: Point start = maze.getStartPosition();
 * ---------------------------------------------
 * Returns a Point indicating the coordinates of the start square.
 */

   Point getStartPosition();

/*
 * Method: isOutside
 * Usage: if (maze.isOutside(pt)) . . .
 * ------------------------------------
 * Returns true if the specified point is outside the boundary of the maze.
 */

   bool isOutside(Point pt);

/*
 * Method: wallExists
 * Usage: if (maze.wallExists(pt, dir))
 * ------------------------------------
 * Returns true if there is a wall in direction dir from the square at pt.
 */

   bool wallExists(Point pt, Direction dir);

/*
 * Method: markSquare
 * Usage: maze.markSquare(pt);
 * ---------------------------
 * Marks the specified square in the maze.
 */

   void markSquare(Point pt);

/*
 * Method: unmarkSquare
 * Usage: maze.unmarkSquare(pt);
 * -----------------------------
 * Unmarks the specified square in the maze.
 */

   void unmarkSquare(Point pt);

/*
 * Method: isMarked
 * Usage: if (maze.isMarked(pt)) . . .
 * -----------------------------------
 * Returns true if the specified square is marked.
 */

   bool isMarked(Point pt);

};

#endif
```

图 9-2 （续）

一旦你能够访问 Maze 类，编写一个程序来解决迷宫问题就变得更简单了。这道习题的目的是编写这样一个函数：

```
bool solveMaze(Maze & maze, Point pt);
```

solveMaze 的参数是：（1）一个保存数据结构的 Maze 对象；（2）起始位置，对于每一个递归子问题，它会改变。由于确保当找到一个解决方案时递归可以终止，故当解决方案被找到时，solveMaze 函数返回 true，否则返回 false。

给出 solveMaze 的定义，主程序如下所示：

```
int main() {
   initGraphics();
   Maze maze("SampleMaze.txt");
   maze.showInGraphicsWindow();
```

```
   if (solveMaze(maze, maze.getStartPosition())) {
      cout << "The marked path is a solution." << endl;
   } else {
      cout << "No solution exists." << endl;
   }
   return 0;
}
```

394 : 396

solveMaze 和 adjacentPoint(start,dir)函数的代码都显示在图 9-3 中，如果你从起始位置向一个特定的方向移动，后者会返回你到达的位置。

```
/*
 * Function: solveMaze
 * Usage: solveMaze(maze, start);
 * ------------------------------
 * Attempts to generate a solution to the current maze from the specified
 * start point.  The solveMaze function returns true if the maze has a
 * solution and false otherwise.  The implementation uses recursion
 * to solve the submazes that result from marking the current square
 * and moving one step along each open passage.
 */

bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjacentPoint(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}

/*
 * Function: adjacentPoint
 * Usage: Point finish = adjacentPoint(start, dir);
 * ------------------------------------------------
 * Returns the point that results from moving one square from start
 * in the direction specified by dir.  For example, if pt is the
 * point (2, 3), calling adjacentPoint(pt, EAST) returns the
 * point (3, 3).  To maintain consistency with the graphics package,
 * the y coordinates increase as you move downward on the screen.  Thus,
 * moving NORTH decreases the y component, and moving SOUTH increases it.
 */

Point adjacentPoint(Point start, Direction dir) {
   switch (dir) {
    case NORTH: return Point(start.getX(), start.getY() - 1);
    case EAST:  return Point(start.getX() + 1, start.getY());
    case SOUTH: return Point(start.getX(), start.getY() + 1);
    case WEST:  return Point(start.getX() - 1, start.getY());
   }
   return start;
}
```

图 9-3 solveMaze 函数的实现

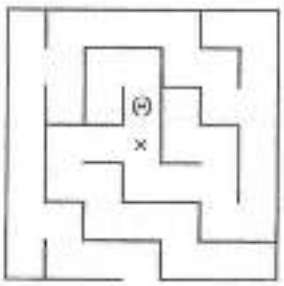
397

9.1.5 说服你自己那个方案可行

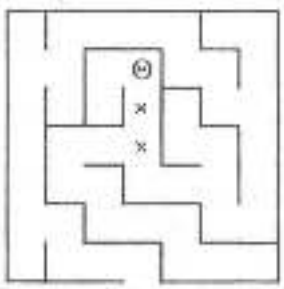
为了有效地使用递归，在某些情况下你必须能够看到如图 9-3 所示的 solveMaze 递归函数，并对自己说："我明白它如何运行。问题会变得越来越简单，因为每一次递归都会有更多的方格被标记，而且简单情况显然是正确的。这段代码足以完成这项工作。"然而，对于大多数人来说，并不会简单地确信递归的力量。人与生俱来的怀疑态度使得你想弄清楚这

解法的步骤。问题是，即使是一个本章开始展示的简单迷宫，其解法设计的完整的步骤也会有很多，以至于不容易想象。例如，当解决那个迷宫问题的方案最终被发现时，需要调用 `solveMaze` 函数 66 次，而 `solveMaze` 函数的嵌套又多达 27 层。如果你想去尝试跟踪代码运行的细节，你几乎肯定会迷失在其中。

如果还没有准备好采纳递归的稳步跳跃，那最好在更一般的意义上跟踪代码的运行。你知道，程序第一次尝试从方格向北移动来寻找解法，因为 `for` 循环是按照 `Direction` 枚举所定义的顺序来遍历所有方向的。因此，解决方案的第一步是从下面的位置开始进行一个递归调用：

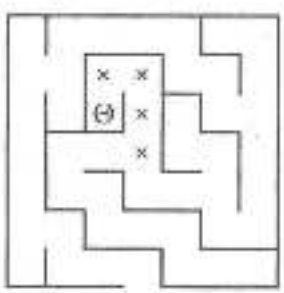


此时，相同的过程再次发生了。程序再一次尝试向北移动，并且在这个位置做了一次新的递归调用：



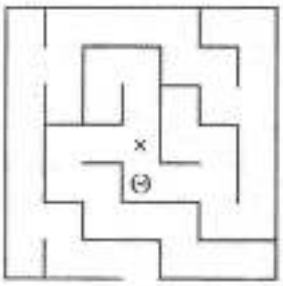
398

在这层递归中，不可能再向北移动了，所以 `for` 循环遍历了其他方向。在试图向南移动时，程序遇到一个标记过的方格，所以改为向西寻找出口并产生了一次新的递归调用。相同的过程发生在这个新的方格中，进而就导致了如下局面：

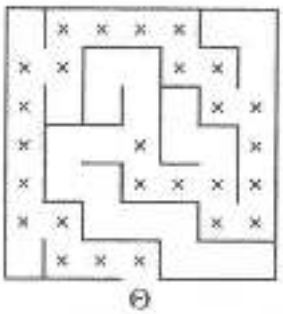


在这个位置上，`for` 循环中的任何一个方向都不行；每一个方格要么被墙堵住，要么已经被标记过。因此当这一级的 `for` 循环从底部退出时，程序会解除当前方格的标记，并且返回到前一级。结果，这个位置上的所有路径也都被探索过一遍。所以程序再次解除当前

方格的标记，并回溯到递归的更高一级上。最终，程序又回溯到初始的递归调用上。此时，for 循环已经穷尽了向北移动的所有可能性。for 循环尝试向东移动，发现被堵了，又继续向南探索，在以下局面下开始递归调用：



从这里开始，相同的过程又接踵而至。递归系统地遍历了该条路上的每一条通道，一旦遇到死胡同就沿递归调用栈原路返回。这条路上唯一的不同是，最终（对于路径中的每一步，降低一个额外的递归层次后）程序在下面的位置进行了一次递归调用：



在这种情况下，特休斯站在迷宫外面，所以简单情况出现了，并返回 true 给它的调用者。然后这个值会通过所有 27 级递归调用传递回去，最终返回到主程序。

9.2 回溯与游戏

尽管回溯在迷宫情景中最容易阐明，这种策略却相当的普遍。例如，你可以将回溯应用到大多数双人策略游戏中。一开始，第一个玩家在移动时有多种选择，根据其选择的移动，然后第二个玩家有特定的一系列响应。每一个响应又会导致第一个玩家进行新的选择，这个过程将一直持续到游戏结束。在游戏中，每一轮不同的可能的位置形成了一个分支结构，其中每一次选择都产生越来越多的可能性。

如果你想让程序把计算机作为双人游戏的一方，一种方法是让计算机跟踪所有可能的分支。在做出第一次移动之前，计算机会尝试进行每一种可能的选择。对于每一次选择，计算机会紧接着决定它的对手将如何响应。为此计算机将遵循相同的逻辑：尝试每一种可能性，并且评估对手可能做出的反击。如果计算机能够有远见知晓自己做出某一步移动可以置对手于死地，那么将走出这一步。

理论上，这个策略可以应用到所有双人策略游戏中。而实际上，即使是现代的计算机，前瞻所有可能的移动、对手潜在的响应以及对这些响应做出的响应等等，这一过程也是既耗时又耗内存的。然而，还是有那么几个游戏很简单，它们可以通过前瞻所有的可能性来求解，然而对于人类玩家来说太复杂，不能立刻找到解法。

9.2.1 拿子游戏

为了理解递归回溯如何应用到双人游戏上，我们考虑一个简单的例子，例如拿子游戏（Nim），它是这一大类游戏的泛称。在这类游戏中，玩家依次从初始的布局中将目标拿走。在这个特定的版本中，游戏从桌子上的13枚硬币开始。每一轮玩家从中任意拾取一枚、两枚或三枚硬币，并将其放置一边。游戏的目的是避免使自己成为拿到最后一枚硬币。图9-4显示了一个人机对战的示例。

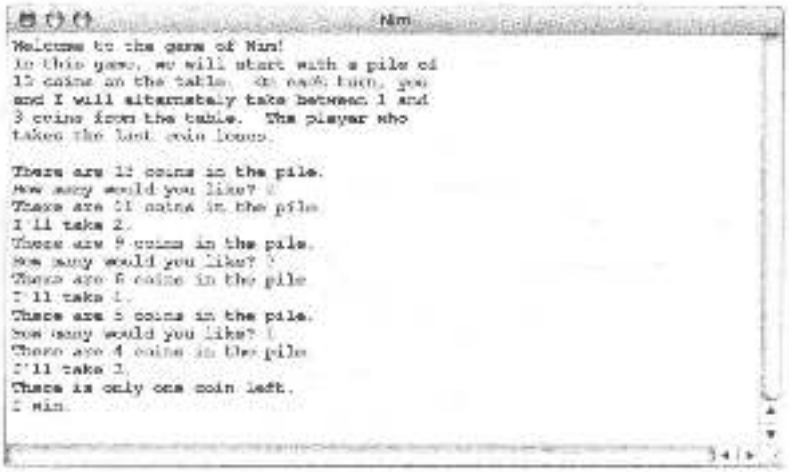


图9-4 拿子游戏的示例运行

你该怎样编写程序去玩这个拿子游戏呢？这个游戏的机械方面（记录硬币数目，请求玩家合法地走子，决定游戏的结束等）组成了一个简单的编程任务。程序中有意思的部分包括如何给计算机指明一个策略，使其演示一个可能最好的游戏。

为拿子游戏寻找一个成功的策略并不特别困难，尤其是，如果你从游戏的结束点往回考虑的话。拿子游戏的潜规则是拿到最后一枚硬币的玩家是输家。因此，如果你发现桌子上仅剩一枚硬币在等着你，那可就糟啦：你不得不取走它并且认输。反之，如果还有两枚、三枚或者四枚，那将是极好的，如果你遇到其中任何一种情况，你往往可以拿走一部分硬币，仅剩一枚硬币，让你的对手不得不拿走最后一枚硬币。

401

但是假设桌子上还有五枚硬币呢？那么你该怎么办？稍加思索，就很容易发现在这种情况下，你也是要输的。无论你怎么做，你都不得不留给对手两枚、三枚或四枚硬币——从对手角度考虑，你会发现局势是有利的。如果对手还算机智的话，下一轮桌子上肯定只剩一枚硬币等着你。既然你没有好棋可走，那么面对五枚硬币也是一种明显不利的局势。

这种非正式的分析揭示了拿子游戏的一个重要规则。在每一轮游戏中，你要去寻找一步好棋。好棋的意思就意味着置对手于不利处境。但是什么又是不利处境呢？那就是没有好棋可走的局面。

即使对于好棋和不利处境的定义看起来好像是个循环，但是它们仍然组成了玩一个完美的双人游戏的完整策略。只是你必须得借助递归的力量。如果你有一个函数findGoodMove，该函数将硬币数作为参数，它必须做的是尝试每一种可能性，寻找置对手于不利处境的走法。你可以将决定一个特定的情形是否是不利的工作分派给判断函数isBadPosition，该函数调用findGoodMove来观察是否有一个不利的处境。这两个方法

一来一往相互调用，评估游戏过程中所有可能的分支。

findGoodMove 和 isBadPosition 这两个相互递归的函数提供了拿子游戏程序进行完美游戏的全部策略。为了完成这个程序，你所要做的就是编码和人类玩家进行游戏的机制。这部分代码对于创建游戏、输出指令、记录该谁走了、请求玩家走子、检查走子是否合法、更新金币数目、判断游戏何时结束、通知玩家输赢情况都是至关重要的。

尽管这些任务概念上并不是很难，但是拿子游戏这个应用程序非常之大，并且采用了6.5节所描述的实现策略，其中，程序定义为一个类而不是一个自由函数的集合。图9-5展示了拿子游戏采用这种设计的一个实现。游戏的代码被封装在一个称之为 SimpleNim 的类中，其中有两个实例变量来记录玩游戏的过程：

- 一个记录桌子上剩余硬币数目的整数变量 nCoins。
- 变量 whoseTurn 指出哪个玩家该走子了。变量的值存储在枚举类型 Player 中，它定义了常量 HUMAN 和 COMPUTER。在每一轮的结束，play 方法的代码通过将 whoseTurn 的值赋给 opponent(whoseTurn) 将顺序传给下一位玩家。 402

```
/*
 * File: Nim.cpp
 * -------------
 * This program simulates a simple variant of the game of Nim.  In this
 * version, the game starts with a pile of 13 coins on a table.  Players
 * then take turns removing 1, 2, or 3 coins from the pile.  The player
 * who takes the last coin loses.
 */

#include <iostream>
#include <string>
#include "error.h"
#include "simpio.h"
#include "strlib.h"
using namespace std;

/* Constants */

const int N_COINS = 13;       /* Initial number of coins                */
const int MAX_MOVE = 3;       /* Number of coins a player may take      */
const int NO_GOOD_MOVE = -1;  /* Marker indicating there is no good move */

/*
 * Type: Player
 * ------------
 * This enumerated type differentiates the human and computer players.
 */

enum Player { HUMAN, COMPUTER };

/*
 * Method: opponent
 * Usage: Player other = opponent(player);
 * ---------------------------------------
 * Returns the opponent of the player.  The opponent of the computer
 * is the human player and vice versa.
 */

Player opponent(Player player) {
   return (player == HUMAN) ? COMPUTER : HUMAN;
}

/*
 * Constant: STARTING_PLAYER
 * -------------------------
 * Indicates which player should start the game.
 */

const Player STARTING_PLAYER = HUMAN;
```

图9-5 Nim.cpp的实现

```
/*
 * Class: SimpleNim
 * ----------------
 * The SimpleNim class implements the simple version of Nim.
 */

class SimpleNim {

public:

/*
 * Method: play
 * Usage: game.play();
 * -------------------
 * Plays one game of Nim with the human player.
 */

   void play() {
      nCoins = N_COINS;
      whoseTurn = STARTING_PLAYER;
      while (nCoins > 1) {
         cout << "There are " << nCoins << " coins in the pile." << endl;
         if (whoseTurn == HUMAN) {
            nCoins -= getUserMove();
         } else {
            int nTaken = getComputerMove();
            cout << "I'll take " << nTaken << "." << endl;
            nCoins -= nTaken;
         }
         whoseTurn = opponent(whoseTurn);
      }
      announceResult();
   }

/*
 * Method: printInstructions
 * Usage: game.printInstructions();
 * --------------------------------
 * Explains the rules of the game to the user.
 */

   void printInstructions() {
      cout << "Welcome to the game of Nim!"  << endl;
      cout << "In this game, we will start with a pile of" << endl;
      cout << N_COINS << " coins on the table.  On each turn, you" << endl;
      cout << "and I will alternately take between 1 and" << endl;
      cout << MAX_MOVE << " coins from the table.  The player who" << endl;
      cout << "takes the last coin loses." << endl << endl;
   }

private:

/*
 * Method: getComputerMove
 * Usage: int nTaken = getComputerMove();
 * --------------------------------------
 * Figures out what move is best for the computer player and returns
 * the number of coins taken.  The method first calls findGoodMove
 * to see if a winning move exists.  If none does, the program takes
 * only one coin to give the human player more chances to make a mistake.
 */

   int getComputerMove() {
      int nTaken = findGoodMove(nCoins);
      return (nTaken == NO_GOOD_MOVE) ? 1 : nTaken;
   }

/*
 * Method: findGoodMove
 * Usage: int nTaken = findGoodMove(nCoins);
 * -----------------------------------------
 * Looks for a winning move, given the specified number of coins.
 * If there is a winning move in the current position, the method
 * returns that value; if not, the method returns the constant
 * NO_GOOD_MOVE.  This method depends on the recursive insight that
```

图 9-5 (续)

```
 * a good move is one that leaves your opponent in a bad position and
 * a bad position is one that offers no good moves
 */

   int findGoodMove(int nCoins) {
      int limit = (nCoins < MAX_MOVE) ? nCoins : MAX_MOVE;
      for (int nTaken = 1; nTaken <= limit; nTaken++) {
         if (isBadPosition(nCoins - nTaken)) return nTaken;
      }
      return NO_GOOD_MOVE;
   }

/*
 * Method: isBadPosition
 * Usage: if (isBadPosition(nCoins))
 * ---------------------------------
 * Returns true if nCoins represents a bad position.
 * A bad position is one in which there is no good move.
 * Being left with a single coin is clearly a bad position
 * and represents the simple case of the recursion.
 */

   bool isBadPosition(int nCoins) {
      if (nCoins == 1) return true;
      return findGoodMove(nCoins) == NO_GOOD_MOVE;
   }

/*
 * Method: getUserMove
 * Usage: int nTaken = getUserMove();
 * ----------------------------------
 * Asks the user to enter a move and returns the number of coins taken.
 * If the move is not legal, the user is asked to reenter a valid move.
 */

   int getUserMove() {
      while (true) {
         int nTaken = getInteger("How many would you like? ");
         int limit = (nCoins < MAX_MOVE) ? nCoins : MAX_MOVE;
         if (nTaken > 0 && nTaken <= limit) return nTaken;
         cout << "That's cheating!  Please choose a number";
         cout << " between 1 and " << limit << "." << endl;
         cout << "There are " << nCoins << " coins in the pile." << endl;
      }
   }

/*
 * Method: announceResult
 * Usage: announceResult();
 * ------------------------
 * Announces the final result of the game.
 */

   void announceResult() {
      if (nCoins == 0) {
         cout << "You took the last coin.  You lose." << endl;
      } else {
         cout << "There is only one coin left." << endl;
         if (whoseTurn == HUMAN) {
            cout << "I win." << endl;
         } else {
            cout << "I lose." << endl;
         }
      }
   }

/* Instance variables */

   int nCoins;                /* Number of coins left on the table */
```

图 9-5 （续）

```
   Player whoseTurn;          /* Marker showing whose turn it is  */
};

/* Main program */

int main() {
   SimpleNim game;
   game.printInstructions();
   game.play();
   return 0;
}
```

403
:
406

图9.5（续）

9.2.2 一个通用的双人游戏程序

图9-5中的代码是特别针对拿子游戏的。例如，play方法直接关系到键立变量nCoins，并在每一次玩家走了后对其进行更新。然而，双人游戏的一般结构有着更为广泛的应用。即使不同的游戏要求不同的实现去完善细节，许多游戏仍可以通过使用全局策略得到解决。

本书的关键概念就是抽象（abstraction），它是将一个问题的一般性质分离出来的过程，从而使人不至于迷失在特定领域的诸多细节当中。你可能对编写一个拿子游戏的程序并不是很感兴趣；毕竟，这确实是一个十足无聊的游戏。你真正感兴趣的可能是编写一个更加普遍的程序，它可以应用于拿子游戏、三联棋游戏，或者其他任何一种你选择的双人策略游戏。

创建这样一种通用机制的动机来自于这样一个事实：大多数游戏共享一部分基础概念。第一个这样的概念就是状态（state）。对于任何一种游戏，都会有一些数据值准确记录任何时间点发生了什么。例如在拿子游戏中，其状态包括它的两个实例变量nCoins和whoseTurn的值。而对于象棋这种游戏，其状态需要包括棋子当前被放在那个格子里，尽管它大概也包括whoseTurn变量，或者其他实现相同功能的一些东西。然而，对于任何一种双人游戏，它都应该存储实现这个游戏的类的变量中的相关数据。

第二个重要的概念是走子（move）。在拿子游戏中，每次走子都会产生一个代表被拿走的硬币数目的整数。在国际象棋中，每一次走子都会产生表示移动棋子的起点和终点的坐标对，尽管这种方法事实上会因“王车易位”或“兵牛变”的高招而变得复杂。然而，对于任何游戏，都可以定义一个Move类型，该类型用来封装在游戏中需要呈现走子的任何信息。

尽管第一眼看Move应该被定义成一个类，但这样做的话会带来额外的开销，降低了代码的可读性。随之而来的问题是类中的成员默认的能见度为私有，它意味着实现这个游戏的类不能访问这些声明为私有的成员。

你可以采用以下几种策略来解决这个问题。一种策略是将Move定义成一个类，但是将其实例变量声明为公有的。然而，这种策略违背了本书使用的一条规则，即实例变量不能出现在公有部分。第二种策略是以第6章类的风格来定义getter和setter方法。这种策略符合现代面向对象编程，但同时使得类Move变得更难使用。对于一个类而言，复杂性无法保障用户对类的有效使用，其用户可能是仅为实现游戏的类。第三种策略是将Move类声明为游戏类的友元。第四种策略是将Move定义为一个结构类型而不是一个类。从某种意义上说，这样做将会使其实例变量声明为公有的，但是事实上Move声明为结构类型允许了一

407

个警告，即在整个游戏代码的上下文中，它只应该被用到与游戏有关的地方。

本书采用了最后一种策略，也就意味着 Move 类型应该如下所示：

```
struct Move {
   int nTaken;
};
```

一旦你有了一个 Move 类型，你就可以定义一些额外的辅助方法，它允许你可像以下这样重写 play 方法：

```
void play() {
   initGame();
   while (!gameIsOver()) {
      displayGame();
      if (getCurrentPlayer() == HUMAN) {
         makeMove(getUserMove());
      } else {
         Move move = getComputerMove();
         displayMove(move);
         makeMove(move);
      }
      switchTurn();
   }
   announceResult();
}
```

最重要的一点是，play 方法的实现代码并没有给出玩的是什么游戏。它也许是拿子游戏，但是也可能仅仅是其他较容易的游戏。每一个游戏需要自己定义一个 Move 类型，以及各种基于游戏的方法，例如 initGame 和 makeMove 的特定实现。尽管如此，play 方法的结构一般足以适用于许多不同的双人游戏。

如果你把 play 的一般实现和图 9-5 的代码进行比较，你会注意到转换顺序的代码通过辅助方法 getCurrentPlayer 和 switchTurn 已经包含在一般结构中。做出这种改变意味着 play 方法不再直接涉及实例变量，而是通过调用方法来完成工作。这个策略允许底层实现更加灵活。 408

然而，play 方法以及实现轮流的机制并不是编写一个双人游戏最有趣的部分。算法上最有趣的部分其实已经嵌入到了方法 getComputerMove 中，它负责为计算机选择最好的走子策略。图 9-5 的拿子游戏版本利用 findGoodMove 和 isBadPosition 两个方法的相互递归调用实现了这种策略，这两个方法在当前状态会遍历所有可能的选择来找到一个最佳的走子。既然此策略也是独立于任何特定游戏的细节之外，因此我们应该可以使用一种更一般的方式编写这些方法。然而，在进一步深入之前，将问题变得更一般化是很有帮助的，这样也能使程序适用于更多的游戏。

9.3 最小最大算法

之前章节所描述的技巧，对于像拿子游戏这样简单、完全可解的游戏非常有用。但是随着游戏变得更加复杂，程序不可能去检验每一种可能的输出。例如，如果你想尝试国际象棋每一种可能的走法，即使以现代计算机的运算速度，也要花上几十亿年。然而，不顾这些限制的话，计算机还是很擅长国际象棋的。1997 年，IBM 的“深蓝”超级计算机就曾打败了当时的世界冠军——加里·卡斯帕罗夫。“深蓝”并没有对所有可能游戏走法进行穷尽的分

析，而是仅仅前瞻了有限数量的走法，这在很大程度上与人类相似。

即使对于游戏来说，遍历所有可能的一系列走法也是不可行的，但是在拿子游戏中，关于好的走法和坏的处境的递归概念仍然能派上用场。尽管也许不可能像万无一失的赢家一样确定某一步走法，但是一个仍正确的事实是：任何处境下最好的走法就是能够将你的对手置于最坏处境的走法。类似地，最坏的处境也就是使你的对手使不出高招。这种策略（它包括寻找一种情况，能够让对手难使高招）被称为**最小最大**（minimax）算法，因为其目的就是寻找一种走法将对手最大的机会最小化。

9.3.1 游戏树

可视化最小最大策略的最好方法就是形成每一轮的分支图来考虑游戏中的可能走法。由于这种分支结构，这样的图被称为**游戏树**（game tree）。初始状态由游戏树顶端的点来表示。例如，如果存在从这个位置开始的三种可能的走法，那么将会从当前状态向新状态引出三条
409 线，如下图所示：



从每一个新位置，你的对手也会有多种选择。如果每一个位置又可以有三种选择，那么下一阶段的游戏树如下图所示：



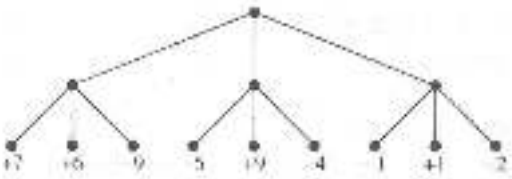
从初始位置开始你会选择那种走法呢？显然，你的目标是取得最好的结果。遗憾的是，你只能控制游戏的一半。如果你能够像选择自己走法一样也能选择对手的走法，你就可以在两轮之后使自己处于最佳位置。鉴于你的对手也想赢，所以最好的方法是尽量选择让对手赢的机会最小的那步棋。

9.3.2 评价位置和走法

为了理解你如何能从一个特定的位置寻找最佳走法，加入一些定量的数据进行分析是很有帮助的。如果可以为每种可能的走法分配一个数值得分，那么判断一种走法比另一种是否更好将会更简单。分值越高，走法越好。因此得分为+7的走法比得分为-4的走法要好。除了为每一种可能的走法评分，也有必要为游戏中的每个位置进行评分。因此，一个得分为+9的位置比一个得分为+2的位置要好。

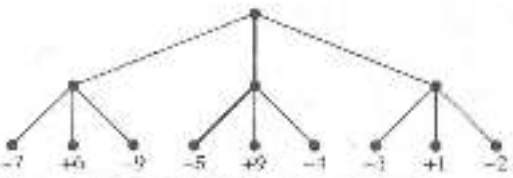
位置和走法都是从走棋玩家的角度进行评分的。而且，评级系统是关于以0对称来设计的，也就是说，当前玩家得分为+9的位置在其对手眼里则是-9的位置。这个评分的解释正是抓住了一种思想，即一个玩家好的位置对另一个玩家来说则是坏的位置。正如在拿子游戏中的情形一样。更重要的是，用这种方法定义一个评估系统很容易表达位置得分和走法得分之间的关系。任何一步走法的得分都是从对手眼里看到的相反数。类似地，任何位置的得分都可以定义为最佳走法的得分。

为了使这种论述更具体，考虑一个简单的例子是很有帮助的。假设你在游戏中可以能瞻两步棋，包括你的一步走法和对手相应的可能的一步走法。在计算机科学中，为了避免和单词“move”以及“turn”产生二义性，单个游戏者的单个动作被称作 ply，它有时表示两个玩家都有玩的机会。如果在评估了两个 ply 之后的位置，那么游戏树可能如下图所示： 410



因为树底部的处境是树顶部的处境的一种可能——你不得不走了，因此这些位置的评分都是从你的角度确定的。给出这些潜在位置的评分，从最初的布局开始，你该如何走子呢？

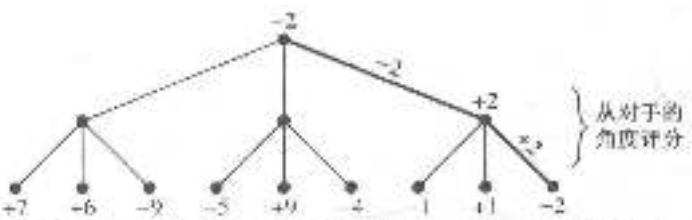
乍一看，你可能被中间那个 +9 的分支所吸引，它对于你来说是一个很好的结果。遗憾的是，虽然中间的分支给出了如此不错的结果，但这并不是真的重要。如果你的对手很理智，那么游戏不可能到达 +9 的位置。例如，假如你真的选择了中间的分支，给出所有可能的选择，那么你的对手将会选择最左边那个 −5 的分支，也就是下面游戏树中粗线标记的分支：



你最初的选择使你处于这样一个位置（从你的角度），得分为 −5。你最好选择最右边的分支，这样你的对手最好的策略也就使你的处境得分变为 −2：



正如在这一节前面所提到的，从对手的角度来看，走法的评分是自身位置的相反数。游戏树中粗线标记的最后一步的评分是 −2，因为它会使对手处于一个 −2 的位置。负号表示角度的转换。能够使对手处于不利位置的走法对你来说是很好的。反之亦然。每一个位置的评分都是它提供的最好走法的评分。因此，关于位置和走法的评分以及粗线标记的路径如下图所示： 411



因此，初始位置的评分是 −2。尽管这个位置并不那么理想，但假设你的对手思路理智，这比起其他可能的结果对你来说算好的了。

接下来会在本章概述最小最大算法的实现，其中表示评分的值是整数，它必须落在以下两个常量之间：

```
const int WINNING_POSITION  = 1000;
const int LOSING_POSITION   = -WINNING_POSITION;
```

在游戏的结尾，位置的得分可以根据谁获胜来确定。任何位置的评分结果并不一定要规定是一个处于这两个常量之间的整数。

9.3.3 限制递归搜索的深度

如果你能从一个游戏一开始搜索并得到每一种可能的结果，基本上就可以利用之前拿子游戏的结构来实现最小最大算法了。你需要两个相互递归的函数，一个用来寻找最好的走法，另一个则用来评估位置。对于非常复杂的游戏，程序是不可能在合理的时间范围内搜索整棵游戏树的。因此，最小最大算法一个比较实际的实现必须规定搜索在某个确定点结束。

约束搜索常见的策略是为递归深度设置某个最大值。例如，你可以规定当每个玩家走五步之后终止递归，两个玩家就是十步。如果游戏在约束条件到达之前结束，你可以通过
412 检测观察谁赢得比赛来评估最后的位置，然后视情况而定返回 WINNING_POSITION 或 LOSING_POSITION。

但是如果在游戏结束之前，你已经到达递归约束条件，会发生什么呢？此时，你需要借助其他一些不会额外进行递归调用的方法来评估位置。鉴于这种分析方法只依赖于当前游戏的状态，它通常被称为**静态分析**（static analysis）。例如，在国际象棋的程序中，静态分析则表现在会根据双方棋盘中的棋子进行简单的计算。如果玩家的走法不在计算范围之内，那么他的位置有一个正的评分，如果不是，评分是负的。

尽管任何简单的计算都免不了要忽视一些重要的因素，但是牢记静态分析只能在达到递归终止条件时进行是很重要的。例如，如果有一种玩法能在没走几步之后取胜，那么这和静态分析无关，因为递归分析会在进入静态分析阶段之前找到赢得比赛的玩法。

在最小最大算法的实现中加入一个深度约束最简单的方法是让每一个递归函数都有一个名为 depth 的参数，它用来记录到目前为止进行过多少级分析，并在对下一个处境进行评分之前就其值加 1。如果参数值大于定义的常量 MAX_DEPTH，那么必须用静态分析来继续进行更深层次的评估。

9.3.4 实现最小最大算法

最小最大算法可以通过使用两个相互递归调用的函数来实现：findBestMove 和 evaluatePosition，它们显示在图 9-6 中。findBestMove 方法考虑每一种可能的走法，然后在产生的位置处调用 evaluatePosition，寻找一个在对手角度看来评分最低的位置。evaluatePosition 方法用 findBestMove 来确定最好的走法，然后返回该走法的评分，除非递归约束条件或者游戏状态需要用到静态分析。

正如你从图 9-6 中的代码所看到的，findBestMove 函数以两种形式存在。第一种形式无参数，并且由用户调用来寻找当前位置最好的走法。findBestMove 的递归调用采用第二种形式，它有两个参数。第一个参数表示递归的深度，正如前面章节所描述的，它使得算法能够在一个确定数量的走法之后终止计算。第二个参数是一个引用参数 rating，它允许
413 findMove 函数将最佳走法的评分返回给 evaluatePosition 函数。

```
/*
 * Method: findBestMove
 * Usage: Move move = findBestMove();
 *        Move move = findBestMove(depth, rating);
 * ----------------------------------------------
 * Finds the best move for the current player and returns that move as the
 * value of the function.  The second form is used for later recursive calls
 * and includes two parameters.  The depth parameter is used to limit the
 * depth of the search for games that are too difficult to analyze.  The
 * reference parameter rating is used to store the rating of the best move.
 */

   Move findBestMove() {
      int rating;
      return findBestMove(0, rating);
   }

   Move findBestMove(int depth, int & rating) {
      Vector<Move> moveList;
      Move bestMove;
      int minRating = WINNING_POSITION + 1;
      generateMoveList(moveList);
      if (moveList.isEmpty()) error("No moves available");
      for (Move move : moveList) {
         makeMove(move);
         int moveRating = evaluatePosition(depth + 1);
         if (moveRating < minRating) {
            bestMove = move;
            minRating = moveRating;
         }
         retractMove(move);
      }
      rating = -minRating;
      return bestMove;
   }

/*
 * Method: evaluatePosition
 * Usage: int rating = evaluatePosition(depth);
 * --------------------------------------------
 * Evaluates a position by finding the rating of the best move starting at
 * that point.  The depth parameter is used to limit the search depth.
 */

   int evaluatePosition(int depth) {
      if (gameIsOver() || depth >= MAX_DEPTH) {
         return evaluateStaticPosition();
      }
      int rating;
      findBestMove(depth, rating);
      return rating;
   }
```

图 9-6 最小最大算法的一段实现

414

图 9-6 中的代码调用了几个方法（每个方法都独立于一个特定的游戏而进行编码），所以值得更深入的解释：

- generateMoveList 方法使用当前状态的合法走法填充矢量 moveList。
- makeMove 和 retractMove 方法分别用来实现走法和撤销一个特定的走法。这两种方法允许程序尝试一种潜在的走法，评估产生的位置，然后返回到原来的状态。
- isGameOver 方法用来检测游戏是否达到最后的状态，在这个状态，程序不可能进行更深的分析。
- evaluateStaticPosition 方法用来在不进行任何进一步递归调用的情况下评估一个特定的状态。

本章小结

本章通过迷宫游戏和双人对战游戏，你已经学会了当寻求某一目标时，如何作出一系列的选择来解决问题。基本的策略是编写一个程序，当某种选择导致无路可走时，它可以回溯到上一个决策点。然而，利用递归的力量，你可以避免对回溯过程的细节进行编码，并且能开发出对很多问题领域都适用的一般的解决策略。

本章的重点包括：

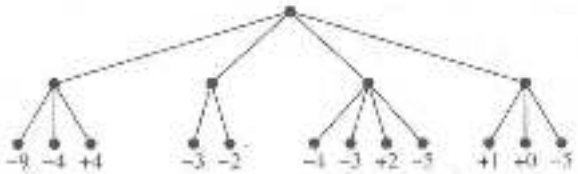
- 采用下面的递归方法，你可以解决大多数需要回溯的问题：

```
如果你已经求得解，报告成功。
for（在当前位置对于每一种可能的选择）{
    做出一种选择，并沿着该路径执行一步。
    从新的位置采用递归来求解这个问题。
    如果递归调用成功，报告下一个更高层次的递归调用的成功。
    否则，回退到当前的选择以恢复之前的程序状态。
}
报告失败。
```

- 一个回溯问题中完整的递归调用的历史（即使是相对简单的应用程序）在细节上也是
415 难以理解。对于涉及多次回溯的问题，接受递归的跳步跳跃是很重要的。
- 在双人游戏中，通过采用一种递归回溯的方法，你总能找到一条取胜的策略。因为这类游戏的目标涉及最小化对手取胜的机会，这种传统的策略被称为最小最大算法。

复习题

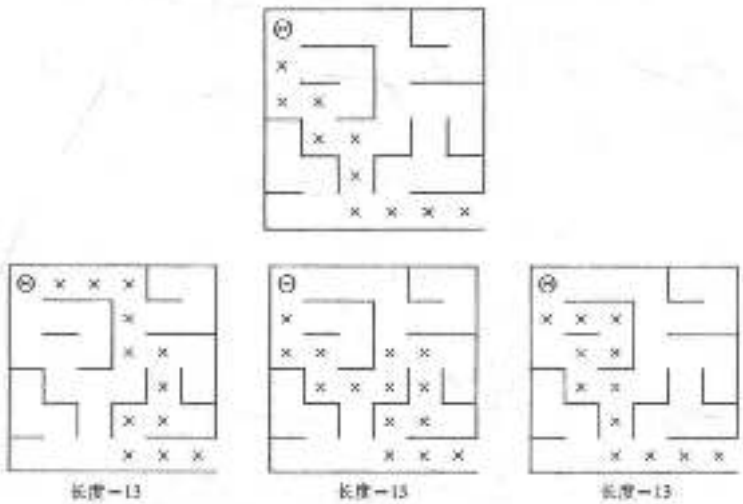
1. 回溯算法的主要特征是什么？
2. 用自己的话来陈述用于迷宫法的右手法则。左手法则也能达到相同的效果吗？
3. 用递归回溯算法解决迷宫问题的核心是什么？
4. 在 `solveMaze` 的函数实现中，简单情况是什么？
5. 当你穿越迷宫时，为什么标记方格很重要？如果你不标记任何方格，那么 `solveMaze` 函数会出现什么情况？
6. 在 `solveMaze` 实现的 `for` 循环结尾调用 `unmarkSquare` 的目的是什么？这条语句对于算法重要吗？
7. `solveMaze` 返回的布尔结果的作用是什么？
8. 用自己的话解释一下，在 `solveMaze` 的递归实现中，回溯过程是如何发生的？
9. 在简单的拿子游戏中，开始时共 13 枚硬币，人类玩家先走第一步。这是一个有利的位置还是不利的位置，为什么？
10. 编写一个基于 `nCoins` 值的简单的 C++ 程序，位置对当前玩家有利时，`nCoins` 返回 `true`，否则返回 `false`。
11. 什么是最小最大算法？它的名字有什么意义？
12. 为什么开发一个最小最大算法的抽象实现，并且不依赖于特定游戏细节很有用？
416 13. 函数 `findBestMove` 和 `evaluatePosition` 中参数 `depth` 的作用分别是什么？
14. 解释一下 `evaluateStaticPosition` 方法在最小最大算法实现中的作用。
15. 假设你处在一个位置，其中从你的角度出发，关于之后两步棋的分析展示了下面的评分结果：



如果你采用最小最大策略，那么这种情况下，最好的走法是什么？从你的角度看，这一步的评分是多少？

习题

1. 在许多迷宫中，有多条路径。例如，图 9-7 展示了同一迷宫的三种解法。然而，三种解法没有一种是最佳的。通过迷宫的最短路径的长度是 11：



417

图 9-7 通过迷宫的多条路径

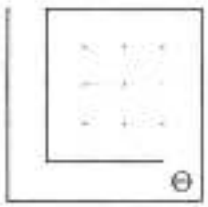
编写一个函数：

```
int shortestPathLength(Maze & maze, Point start);
```

该函数返回从指定地点出发到达出口的最短路径的长度。如果该迷宫问题不存在解法，shortestPathLength 应该返回 -1。

2. 正如图 9-3 所实现的，solveMaze 函数在发现从一点出发没有解法时，会擦除每个方格的标记。尽管这种设计策略有一个优点，即迷宫的最终解法的布局由一系列带标记的点表示，但是对于整个算法的效率而言，回溯时擦除标记的方格会付出很大的代价。如果你已经标记了一个方格并且回溯经过它，你就已经搜索了从那个方格出发的所有可能情况。如果从其他某条路径又回到该点，你也要借助前面的分析，而不是再进行一次相同的探索。

就效率而言，为了理解这些擦除标记的操作花费多大，扩展 solveMaze 程序使它能记录处理过程中递归调用的次数。使用该程序来计算，如果 unmarkSquare 调用仍作为程序的一部分，解决下面的迷宫问题需要多少次递归：



再次运行程序，这次不调用 unmarkSquare 函数，递归调用的次数有何变化？

3. 上一题的结果清楚地表明：通过在 Maze 类中使用 markSquare 机制来记录迷宫的路径需要花费很大的代价。一种更实际的方法是改变递归函数的定义，使得它只记录当前路径。遵循 solveMaze 的逻辑，编写一个函数：

```
bool findSolutionPath(Maze & maze, Point start,
                      Vector<Point> & path);
```

418

该方法的参数除了起始位置之外，还有一个 Point 值的矢量 path 和 solveMaze、findSolutionPath 一样，通过返回一个布尔值来表示迷宫是否有解。另外，findSolutionPath 函数以一系列的坐标来初始化 path 数组的元素，坐标从起点位置开始，并以迷宫出口外第一个方格的坐标结束。对于本题，findPath 可以寻找任何路径，不一定要找最短的那条。

4. 大多数个人计算机的画图程序都可以用纯色来填充一个封闭区域。典型地，通过选择“颜色填充”工具，然后使用你图形中的光标点击鼠标，你可以调用这个操作。当你这样做之后，颜色就会布满它能到达的图形的每一部分，而不是通过线。

例如，假设你已经画了下面这样一幢房屋图：



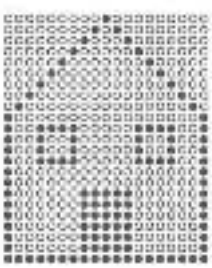
如果你选择颜色填充工具并在门内点击，画图程序就会填充以门为边界的区域，如以下左图所示。如果你点在房屋前面墙上某个地方，程序就会将窗户以及门以外的全部区域填充，如以下右图所示：

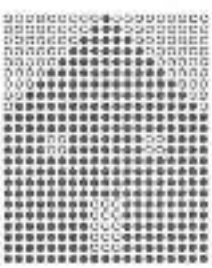




419

为了理解这个过程是如何工作的，需要理解计算机屏幕实际上是被分解成称之为像素（pixel）的点阵。在黑白显示器上，像素点要么是黑的，要么是白的。颜色填充操作就是从点击的点开始涂黑连通的白色像素点。因此，前两个图在屏幕上的像素点如下图所示：



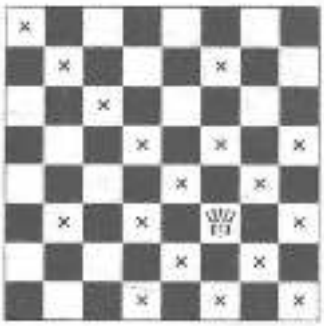


用 Grid<bool> 类型来表示一个像素网格是非常简单的。网格中白色的像素点的值为 false，黑色的则为 true。给出这种表示方法，编写一个函数：

```
void fillRegion(Grid<bool> & pixels, int row, int col)
```

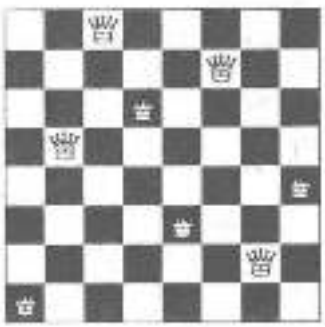
它通过从某个具体的行和列开始在绕过存在的黑色像素点的情况下，将能接触到的白色像素点涂黑来模拟颜色填充工具的操作。

5. 在国际象棋中，最有威力的棋子就是皇后，该棋子可以在任意方向移动任意数目的方格，要么水平移动，要么垂直移动，要么沿对角线移动。例如，下面的棋盘显示的皇后棋可以移动到标记的任何一个方格：



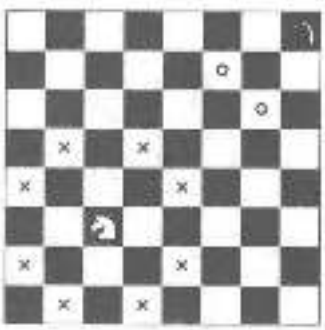
420

即使皇后被可以纵横很多方格，还是可以在 8×8 的棋盘上放置 8 个皇后使它们之间不能相互攻击，如下图所示：



编写一个程序，解决更普遍的问题，即是否能在 N×N 的棋盘中放置 N 个皇后，使它们在一步之内无法攻击对方。你的程序应该能显示一种解决方案或者报告无解。

6. 在国际象棋中，骑士走的是“L”形路：沿横向或纵向移动两个方格，再垂直移动一个方格。例如，下图中白骑士可以移动到任何以 × 标记的方格：

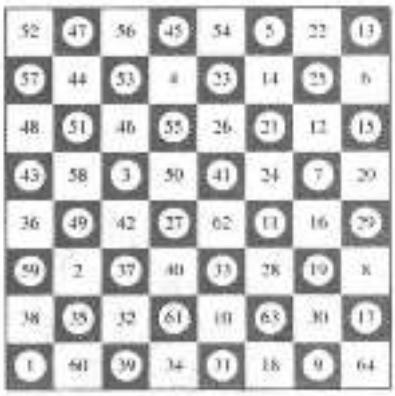


接近棋盘边缘会降低骑士的机动性，正如角落中的黑骑士所示，它只能到达两个以 o 为标记的方格。

421

事实证明：一个骑士在不重复踏入相同方格的前提下，可以访问棋盘上 64 个方格。骑士不重复走过

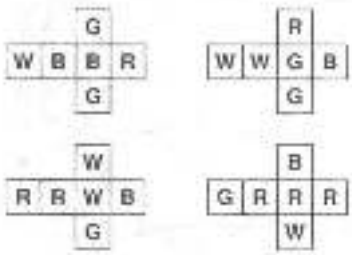
所有这些方格的路径被称作骑士之旅（knight's tour）。下图就展示了这样的一次旅行，其中，方格中的数字表示它们被访问的先后顺序：



编写一个程序，要求使用回溯递归找到一条骑士之旅。

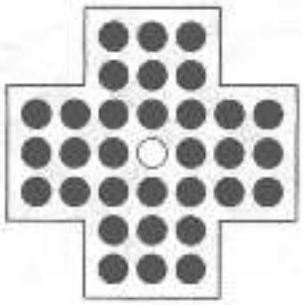
7. 在20世纪60年代，一种称作四色方柱（Instant Insanity）的谜题曾经风靡了好多年。这个谜题包含四个方块，它们分别被涂上了红色、蓝色、绿色以及白色。在下面的图表中，分别用它们的英文首字母来表示。谜题的目标就是将这些方块排成一行，无论你从任何一条边看一条直线，你都看不到重复的颜色。

在二维平面中很难画出方块，但是如果你将它们展开并放在平面上，下图展示了它们的形状：

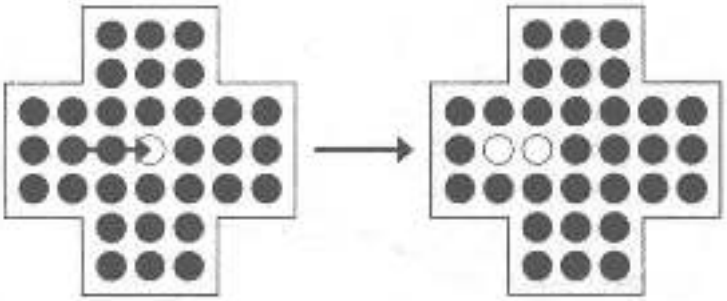


422 编写一个程序，要求使用回溯来解决四色方柱谜题。

8. 理论上，本章描述的递归回溯策略应该足以解决涉及一系列连续移动达到目标状态的谜题。然而，实际中许多这样的谜题太复杂而不能在合理的时间内完成。有一种谜题仅仅会在达到递归约束条件时完成，而没有其他捷径可寻，它就是起始于17世纪的孔明棋（peg solitaire）。孔明棋通常会在这样一个棋盘上来玩：

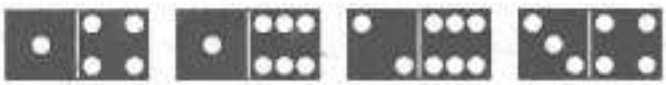


图中的黑点表示棋子，除了中间的洞，棋盘都被黑点填满了。每一轮你需要跳过一个棋子然后拿走它，如下图所示，其中，黑棋跳到了中间的空洞里面，并且中间的棋子被移走：



该游戏的目标是进行一系列的跳跃，使得只剩一个棋子留在中间的洞里。编写程序解决此谜题。

9. 多米诺骨牌是一个由两个相连方格组成的矩形块，其中每一个都由一些点进行标记。例如，下面四个矩形块都代表一个多米诺骨牌：



423

多米诺骨牌可以通过首尾相接形成链，其前提条件是：只有当两个多米诺骨牌接触的地方点数匹配时才可以连接。例如，你可以形成一个链，这个链包含四张多米诺骨牌，这四张牌以下面的顺序连接：



在传统的游戏中，多米诺骨牌可以通过旋转 180 度来反转数字。例如，在这条链中，牌 1-6 和牌 3-4 都被反转使得它们适用这条链。

假设你能使用 Domino 类（见第 6 章习题 11），它提供了 getLeftData 和 getRightData 方法。利用这个类，编写一个递归函数：

```
bool formsDominoChain(Vector<Domino> & dominos);
```

如果可以将矢量中每一个多米诺骨牌连接成一条链，函数返回 true。

10. 假设你被分派了一项工作，要去为一个建筑工程购买管道。上司给了你一个清单，上面有需要的一些不同长度的管子，但是零售店只有一种固定规格的管子。不过，你可以根据需要切断管子。你的工作是计算出最小数目的库存管来满足清单的要求，这样也能省钱并使开销最小。

编写一个递归函数：

```
int cutStock(Vector<int> & requests, int stockLength);
```

该函数有两个参数（需求的矢量的长度和零售店的库存管长度），并返回满足需求矢量数组需要的库存管的最小数目。例如，如果数组中是 [4，3，4，1，7，8]，并且库存管的长度为 10，那么你可以买一个库存管并将其分割如下：

管子 1：4,4,1

管子 2：3,7

管子 3：8

这样做会剩余两小节管子。还有其他的分配方法也使用到三个库存管，但是任务完成时，剩余量不会更少。

424

11. 大多数操作系统和许多应用程序都允许用户使用文件支持通配符模式（wildcard pattern）。其中特殊的字符被用来创建文件名模式来匹配许多不同的文件。在通配符中最常见的字符就是“?”，它可以匹配任意单个字符，还有“*”，它用来匹配任意的字符序列。一个文件名模式中

的其他字符必须匹配文件名中相应的字符。例如，模式“*.*”可匹配任何形式的文件名，例如 EnglishWords.dat 以及 HelloWorld.cpp，但是不能匹配不包含点号的文件名。类似地，模式 test.? 匹配任何包含名字 test、一个点号以及一个单一字符的文件名。因此，test.? 匹配 test.h 但不匹配 test.cpp。这些模式可以用你喜欢的任何方式组合。例如，模式“??*”匹配任何至少包含两个字符的文件名。

编写一个函数：

```
bool wildcardMatch(string filename, string pattern);
```

该函数有两个字符串参数，分别表示文件名和通配符，并且如果文件名和模式匹配时，函数返回 true。因此，

```
wildcardMatch("US.txt", "*.*")          返回   true
wildcardMatch("test", "*.*")            返回   false
wildcardMatch("test.h", "test.?")       返回   true
wildcardMatch("test.cpp", "test.?")     返回   false
wildcardMatch("x", "??*")               返回   false
wildcardMatch("yy", "??*")              返回   true
wildcardMatch("zzz", "??*")             返回   true
```

12. 重写简单的拿子游戏，要求它使用图 9-6 所示的一般化的最小最大算法。你的程序不能改变 findBestMove 以及 evaluatePosition 的实现代码。你的工作就是提出一个合适的关于 Move 类型和游戏具体实现方法的定义，使得程序仍可以完成一个完美的游戏。
13. 修改你在习题 12 中编写的拿子游戏的代码，使得程序可以玩一个不同的拿子游戏。在这个版本中，桌子上一开始有 17 枚硬币。每一轮玩家可以选择从桌子上拿走一枚、两枚、三枚、或者四枚硬币。在简单的拿子游戏中，玩家拿走的硬币直接被忽视了，在这个游戏版本中，拿走的硬币积累
[425] 交给对个玩家。在最后一枚硬币被拿走以后，手中有偶数个硬币的玩家赢得游戏。
14. 在大多数拿子游戏中，硬币并不是放成一堆，而是如下图所示被排列成三行：



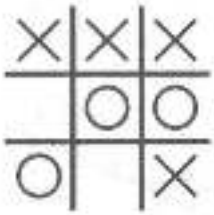
游戏的每一步包括拿走任意数目硬币，前提是所有的硬币必须来自于同一行。拿走最后一枚硬币的玩家为输。

编写一个程序，要求使用最小最大算法来完成一个完美的三堆拿子游戏。这里展示的开始布局是一个非常典型的示例，但是你的程序应该足够一般化，这样你很容易改变行数以及每一行的硬币数。

15. **三联棋**（tic-tac-toe）（或者 naught and cross）游戏的玩法是两个玩家轮流在如下的 3×3 的网格中放置 × 和 ○：



游戏目的是将你自己的三个符号放在一行中，垂直、水平、或者对角线放置。例如，在下面的游戏中，顶部的 × 因为三个在同一行中，所以已经赢得了游戏：



如果棋盘被占满，却没有任何人完成一行，本次游戏就算平局，在三联棋中它被称作猫的游戏（cat's game）。

编写一个程序，要求使用最小最大算法来实现一个完美的三联棋游戏。图 9-8 展示了对战一个特别笨的玩家的运行示例。

426

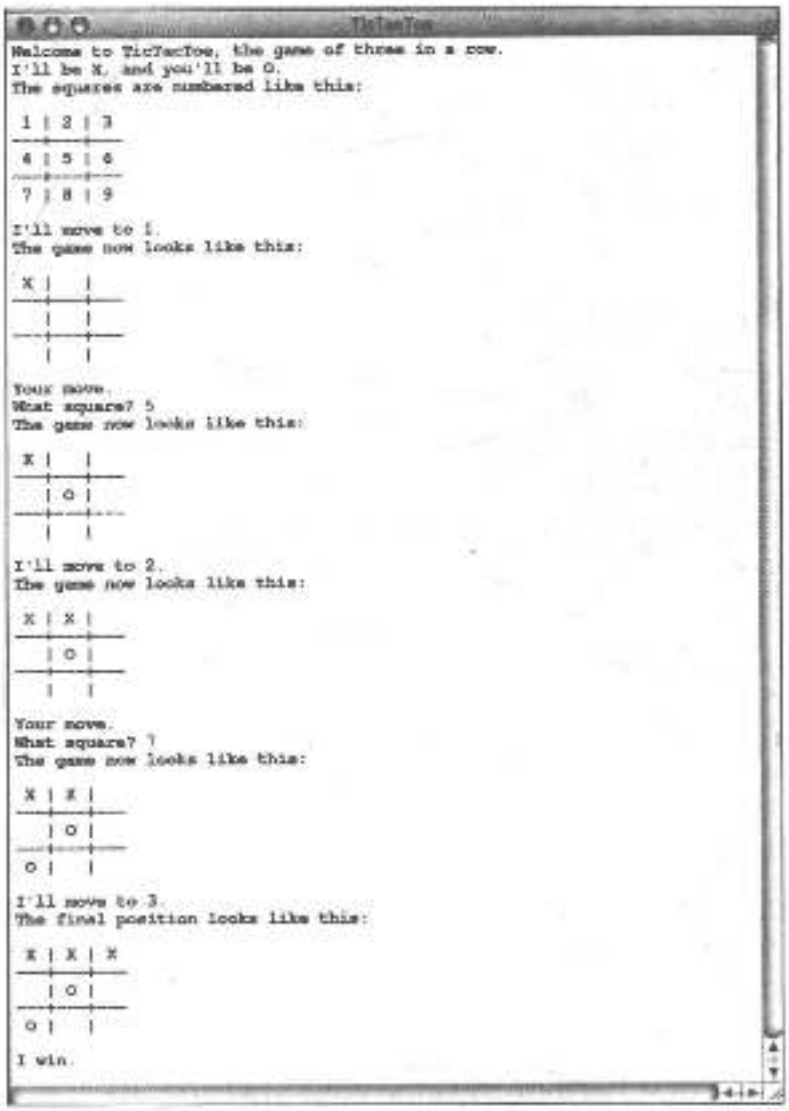


图 9-8　三联棋游戏的运行示例

427

16. 拼字游戏是在一个4×4的网状方块上玩的，每一个网格表面都有一个字母。目标是尽可能多地连接四个或更多的字母。连接仅限于水平、垂直，或者沿对角线邻接的字母方体中，并且不能使用一个方格超过一次。图9-9展示了拼字一种可能的布局以及可以在该布局中连接的单词，这些单词都在EnglishWords.dat字典中。举个例子，你可以利用下面的方块序列来组成单词"programming"：



编写一个函数：

```
void findBoggleWords(const Grid<char> & board,
                     const Lexicon & english,
                     Vector<string> & wordsFound);
```

它的功能是找到棋盘中在英语大字典中出现的所有合法单词，并且将这些单词添加到矢量wordsFound中。



图9-9 拼字游戏的示例配置及它所包含的单词

算法分析

没有分析，就没有综合。

——弗里德里希·恩格斯，《反杜林论》，1878

在第 7 章，你学习了函数 fib(n) 的两种不同的递归实现，这个函数用来计算斐波那契数列中的第 n 项。第一种递归实现直接以下面的数学定义为基础：

$$\text{fib}(n) = \begin{cases} n & n=0,\ 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{其他} \end{cases}$$

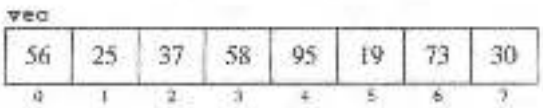
它非常低效。第二种实现使用了可加序列的概念，从而产生了一个新的 fib(n) 函数版本，它与传统的迭代方法相比更高效，这说明递归本身并不是产生问题的原因。即使如此，由于第一个版本的斐波那契数列函数需要如此高的执行代价，以致递归常常获得了一个坏名声。

正如本章将要阐述的，递归地思考一个问题的能力经常会产生出新的策略，这些新策略远比那些由一个迭代设计过程产生的任何策略都要高效。分治算法的强大功能对很多实际问题都有着意义深远的影响。通过使用这种形式的递归算法，其求解效率得到了显著的提升，其提升的效率不是 2~3 个数量级，而是成千上至更高的数量级。

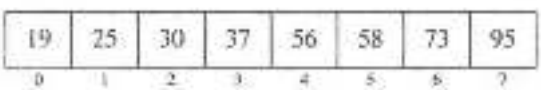
然而，在你了解这些算法之前，询问几个问题非常重要。对一个算法而言，术语效率意味着什么？如何评估一个算法的效率？这些问题形成计算机科学的一个子领域，即**算法分析**（analysis of algorithm）。尽管对算法分析的详细理解需要一个合理的数学工具以及大量的思考，但是你可以通过研究几个简单算法的性能对算法分析有一个感性认识。

10.1 排序问题

领会算法分析重要性的最好方法就是考虑一个问题域，其中不同算法的性能有很大的差别。当然，这些问题域中一个最有趣的问题就是**排序**（sorting）问题，排序就是将一个数组或者一个矢量中的元素重新进行排列，以使它们以一个既定的顺序排列。例如，假设你能对存储在 Vector<int> 类的变量 vec 中的以下元素进行排序：



你的任务就是编写一个函数 sort(vec)，它能以升序方式重新排列这些元素，如下图所示：



10.1.1 选择排序算法

有很多可供选择的算法可使你以升序方式重新排列一个矢量中的整型元素。其中最简单

的一种算法称为**选择排序**（selecting sort）算法。给定一个大小为 N 的矢量，选择排序算法会仔细检查其中每个元素的位置并找到该元素值在最终排好序的矢量中应占据的位置。如果算法发现了一个合适的元素，它就将该元素与先前占据这个位置的元素进行交换以确保没有丢失元素。因此，在第一次循环中，算法找出最小的元素并将其与 C++ 中当成在索引位置 0 处的第一个元素进行交换。在第二次循环中，算法找出次小元素并将其与数组中的第二个元素进行交换。之后，算法以此策略继续执行，直到矢量中所有位置上的元素都正确排序为止。图 10-1 展示了一种使用选择排序的 sort 实现。

```
/*
 * Implementation notes: sort
 * --------------------------
 * This implementation uses an algorithm called selection sort, which can
 * be described as follows.  With your left hand (lh), point at each element
 * in the vector in turn, starting at index 0.  At each step in the cycle:
 *
 * 1. Find the smallest element in the range between your left hand and the
 *    end of the vector, and point at that element with your right hand (rh).
 *
 * 2. Move that element into its correct position by exchanging the elements
 *    indicated by your left and right hands.
 */

void sort(Vector<int> & vec) {
   int n = vec.size();
   for (int lh = 0; lh < n; lh++) {
      int rh = lh;
      for (int i = lh + 1; i < n; i++) {
         if (vec[i] < vec[rh]) rh = i;
      }
      int tmp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = tmp;
   }
}
```

431

图 10-1 选择排序算法的实现

例如，如果矢量中的初始内容如下：

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

在外层 for 循环的第一次循环中，算法会将位于索引位置 5 的元素 19 标识为整个矢量中最小的元素，然后将其与位于索引位置 0 的 56 进行交换，排序结果如下：

19	25	37	58	95	56	73	30
0	1	2	3	4	5	6	7

在第二次循环中，算法在索引位置 1 与 7 之间寻找最小元素，其结果为位于位置 1 的 25。程序向前执行并且不进行元素的交换操作。在其后的每次循环中，算法执行一个交换操作从而将下一个最小的元素移到合适的最终位置。当 for 循环结束时，整个矢量中的元素就已经排序完毕。

10.1.2 性能的经验评估

作为一种排序策略，选择排序算法的效率如何？为了回答这个问题，收集不同规模的元素排序所耗费的计算时间的实际数据对我们会很有帮助。例如，当在我的 MacBook Pro 笔

记本上运行排序算法时，我观测到以下的运行时间，这里 N 代表矢量中的元素个数：

N	运行时间
10	0.000 002 4 s
50	0.000 0448 s
100	0.000 169 s
500	0.004 02 s
1000	0.015 9 s
5000	0.395 s
10 000	1.58 s
50 000	39.6 s
100 000	158.7 s

对于一个包含 10 个整数的矢量而言，选择排序算法只用几微秒的时间就完成了工作。即使对于 5000 个整数来说，函数 sort 的这个实现也只花费了不超过一秒的时间，就我们人类对时间的感觉来说，这段时间无疑是非常短的。然而，随着矢量规模的不断增大，选择排序算法的性能开始逐渐衰退。对于一个含有 100 000 个整数的矢量而言，该算法需要超 432
过两分半的时间才能完成。如果你坐在你的计算机前等待结果，那么这个时间就显得非常长了。

更令人不安的是，选择排序算法的性能随着矢量规模的增大而迅速地恶化。正如你可以从以上计时数据中看到的一样，每次你将需要排序的值的数量扩大 10 倍，则其排序所需的时间将增长百倍。因此，对一个包含有百万个数字的序列进行排序将会花费大约四个半小时。如果你在工作中需要处理这种规模的数据，那你只能去寻找一种更为有效的方法。

10.1.3 分析选择排序算法的性能

随着所需排序数据规模的增大，为什么选择排序算法的性能会变得如此之差呢？为了回答这个问题，我们需要思考算法在每一个外层循环中都做了些什么。为了正确地确定矢量中的第一个最小值，选择排序算法必须考虑在 N 个元素中进行查找。因此，第一次循环所需的时间大概是与 N 成正比的。对于矢量中的其他元素，算法执行相同的基本操作，但是每次都考虑更少的元素。第二次循环考虑 $N-1$ 个元素，第三次循环考虑 $N-2$ 个元素，依此类推。因此，总的运行时间大约正比于以下式子：

$$N + N-1 + N-2 + \ldots + 3 + 2 + 1$$

由于很难处理这样一个展开形式的表达式，所以通过运用一点数学知识来简化这个表达式是很有用的。你可能已经在代数课上学过前 N 个整数的求和公式为：

$$\frac{N \times (N+1)}{2}$$

或者，将其分子相乘并展开后为：

$$\frac{N^2+N}{2}$$

你将会在 10.6 节中学到如何证明这个公式的正确性。目前，你所要知道的就是前 N 个整数之和可以使用这种更加简洁的形式来表达。

如果写出以下函数的值：

$$\frac{N^2+N}{2}$$ 433

对于 N 的不同取值，如果你根据以上公式计算对应的值，你会得到如下所示的表格：

N	$\frac{N^2+N}{2}$
10	55
50	1275
100	5050
500	125 250
1000	500 500
5000	12 502 500
10 000	50 005 000
50 000	1 250 025 000
100 000	5 000 050 000

由于选择排序算法的运行时间可能与算法所需要做的工作总量有关，因此，表格中的数值应该大致与算法运行时间的观测值成比例，它也得到了证实。例如，如果你观察图 10-2 中选择排序所测得的计时数据，你会发现该算法需要 1.58 s 来处理 10 000 个数据。在这段时间内，选择排序算法内部执行了 50 005 000 次操作。假设在这两个值之间确实存在一个比例关系，用执行时间除以总的操作次数得到下面这个近似的比例常数：

$$\frac{1.58\text{ s}}{50\,005\,000} = 3.16\times10^{-8}\text{ s}$$

如果你对于表中的其他记录应用相同的比例常数，至少在 N 取较大值的时候，那么你会发现以下公式：

$$3.16\times10^{-8}\times\frac{N^2+N}{2}$$

提供了运行时间的一个合理的近似值。图 10-2 中表示的是观测的时间和使用这个公式计算的估计时间，以及两者之间的相对误差。

N	观测时间	估计时间	误差
10	0.000 002 4 s	0.000 001 7 s	28%
50	0.000 044 8 s	0.000 040 3 s	10%
100	0.000 169 s	0.000 159 s	6%
500	0.004 02 s	0.003 95 s	2%
1000	0.015 9 s	0.015 9 s	<1%
5000	0.395 s	0.395 s	<1%
10 000	1.58 s	1.58 s	<1%
50 000	39.6 s	39.5 s	<1%
100 000	158.7 s	158.0 s	<1%

图 10-2 选择排序算法的观测时间与估计时间

10.2 时间复杂度

像图 10-2 所示的那样详细地分析算法所存在的问题是：你最终总是得到过多的信息。尽管有一个公式能够准确地预测一个程序执行将要花费多长时间有时是有用的，但同时得到的大量定性指标也会让你偏离主题。当 N 较大时，选择排序算法不可行的原因在于：它与运行在笔记本电脑上的一个特定算法实现的精确计时特性没有太多关系。这个问题现在变得更简单并且更基础了。从本质上讲，选择排序算法的问题是：如果待排序的元素数目翻倍，则选择排序

算法的运行时间将增大四倍，这意味着其运行时间的增长要比待排序元素数目的增长快得多。

关于算法效率你所能获得的最有价值的定性观察，通常有助于你理解因问题规模变化而引发的算法性能变化。问题规模通常很容易量化。例如，如果算法操作的是数字，则通常用数字本身的规模作为问题规模。就大多数对数组或者矢量进行操作的算法而言，你可以使用数组中的元素个数表示问题的规模。当计算算法的效率时，计算机科学家无论如何进行计算都习惯上使用字母 N 来表明问题的规模。随着 N 的变大，N 与一个算法性能之间的关系称为该算法的时间复杂度（computational complexity）。通常，尽管也可能对其他因素进行所需的内存空间总量进行评估，但是对算法性能最重要的度量是其执行时间。除非特殊声明，否则本书中所有的算法复杂度均指的是其执行时间。

10.2.1 大 O 符号

计算机科学家采用一种特殊的助记符来表示算法的时间复杂度，这个符号称为大 O 符号（big-O notation）。大 O 符号由德国数学家保罗·巴赫曼在 1892 年引入——这远在计算机出现之前。这个符号本身非常简单，由字母 O 后跟一对圆括号括起来的公式组成。采用此符号表示算法复杂度时，其中的公式通常是一个涉及问题规模 N 的简单函数。例如，本章中你很快会遇到大 O 表示法：

434
435

$$O(N^2)$$

可读作“N 平方的大 O”。

大 O 符号常用于表明算法的定性近似值，因此，它是一个算法时间复杂度的理想表示。由于该表示法从数学中引入，因此，大 O 符号有一个精确的定义，该定义参见 10.2.6 节。然而，此时，无论你自认为是程序员还是计算机科学家，直观地理解大 O 的含义对你来说是至关重要的。

10.2.2 大 O 的标准简化

当你使用大 O 符号来估计一个算法的时间复杂度时，其目的就是提供一个定性的认识：当 N 变大时，N 的变化是如何影响算法的性能的。因为大 O 符号不是用来定量度量的，所以它不仅适合而且有助于减少括号内的公式，以便能用最简洁的形式来捕获一个算法的行为。当使用大 O 符号时，可使用的最常用的简化如下：

1. 随着 N 变大，删除那些对总的结果影响不大的项。当一个公式包含几个相加在一起的项时，随着 N 的变大，其中一项总比其他项增加得快，即这一项是支配表达式结果的项。当 N 的值较大时，由于这个最支配作用的单独项控制着算法的运行时间，因此，可以完全忽略这个公式中的其他项。

2. 删除任何常量因子。当你计算算法复杂度时，主要关注点是算法的运行时间是如何作为问题规模 N 的函数而变化的。常量因子总体上对算法时间性能不会产生任何影响。如果你购买了一台比你原来的计算机快 2 倍的计算机，对于每一个 N 值，在你的机器上执行的任何算法将会比在原来机器上执行时快 2 倍。无论如何，这个增长的模式将保持完全相同。因此，当你使用大 O 符号时，可以省略掉常量因子。

10.2.3 选择排序算法的时间复杂度

可以采用上节所述的规则来推导选择排序算法时间复杂度的大 O 表达式。从 10.1.3 节

436 中的分析，你已知道对 N 个元素的选择排序算法的运行时间与以下公式成比例：

$$\frac{N^2+N}{2}$$

尽管在大 O 表达式中直接使用这个公式从数学上来说是正确的，如下所示：

$$O\left(\frac{N^2+N}{2}\right)$$



但在实际中，你完全不能这么做，因为括号里的公式没有采用最简单的形式。

简化这个表达式的第一步是识别出这个公式实际上为两项之和，如下所示：

$$\frac{N^2}{2} + \frac{N}{2}$$

然后，你需要考虑随着 N 的增大，其中的每一项对总公式的贡献，其贡献如下表所示：

N	$\frac{N^2}{2}$	$\frac{N}{2}$	$\frac{N^2+N}{2}$
10	50	5	55
100	5 000	50	5 050
1 000	500 000	500	500 500
10 000	50 000 000	5000	50 005 000
100 000	5 000 000 000	50 000	5 000 050 000

随着 N 的增加，含有 N^2 的项迅速占主导地位。因此，根据简化规则，可删除较小的项。即使这样，也不能把选择排序算法的时间复杂度写成以下形式：

$$O\left(\frac{N^2}{2}\right)$$



由于可以删除常量因子，因此，可以把选择排序的算法复杂度用如下最简单的表达式表示：

$$O(N^2)$$

437 这个表达式抓住了选择排序算法性能的本质。随着问题规模的增大，算法的运行时间趋向于问题规模 N 的平方倍。因此，如果你将矢量中待排序元素的数目 N 扩大 2 倍，那么算法的运行时间将增大 4 倍。如果 N 增大了 10 倍，那么算法的运行时间将猛增 100 倍。

10.2.4 从代码中降低时间复杂度

常常需要在粗略地浏览一遍代码后就能计算出其时间复杂度，以下函数计算一个矢量中的元素平均值：

```
double average(Vector<double> & vec) {
   int n = vec.size();
   double total = 0;
   for (int i = 0; i < n; i++) {
      total += vec[i];
   }
   return total / n;
}
```

当调用这个函数时，这段代码中的部分代码只执行了一次，例如将 `total` 变量初始化为 0 以及 `return` 语句中的除法操作。这些操作都消耗一个确定的时间，且这些时间严格来说

是常量，它们不会随着矢量大小的改变而改变。可以认为那些执行时间不依赖于问题规模的代码在常量时间（constant time）内运行，用 $O(1)$ 表示。

$O(1)$ 看起来可能令一些人困惑，因为括号里面的表达式不依赖于 N。事实上，对 N 不产生任何依赖正是 $O(1)$ 符号的全部要点。当问题规模增大时，那些运行时间为 $O(1)$ 的代码所需的执行时间正与 1 增加的方式完全相同：换句话说，也就是代码的运行时间不变。

然而，average 函数的其他部分执行了 n 次，对每一个 for 循环，它们都执行了一次。包括在 for 循环中的表达式 i++ 以及语句

```
total += vec[i];
```

构成了循环体。尽管这部分计算的任何一次执行都耗费了一个固定的时间，但事实上，这些语句执行了 n 次意味着它们总的执行时间直接与矢量的大小成正比。average 函数中这部分的算法复杂度是 $O(N)$，它经常称为**线性时间**（linear time）。

因此，函数 average 总的运行时间就等于算法的常量部分与线性部分所需的时间之和。然而，随着问题规模的增大，常量项变得越来越不重要了。通过采用简化规则允许你忽略那些随着 N 的增大而变得不重要的项，就可估算出 average 函数的总体运行时间为 $O(N)$。 428

尽管可以通过观察循环结构的代码来预测算法的复杂度，但大多数情况下，单个表达式和语句（除非涉及必须单独计数的函数调用）的运行时间为常数。就算法复杂度来说，最重要的就是这些语句执行了多少次。对于大多数程序而言，可以通过找出最经常执行的代码来确定算法的复杂度，并且将它的运行时间确定为一个关于 N 的函数。在 average 函数这个例子中，函数体共执行了 n 次。由于没有其他部分的代码执行超过 n 次，因此可以预测这个函数的算法复杂度为 $O(N)$。

选择排序函数也可以用相似的方法进行分析。这个函数中最经常执行的代码就是比较语句：

```
if (vec[i] < vec[sh]) sh = i;
```

该语句嵌套在两个 for 循环中，其执行次数取决于 N 的值。每运行一次外部循环，内部循环就运行 N 次。这也意味着内部循环体执行了 $O(N^2)$ 次。和选择排序一样，具有 $O(N^2)$ 性能的算法称为以平方时间（quadratic time）运行。

10.2.5 最坏情况以及平均情况下的复杂度

某些情况下，一个算法的运行时间不仅取决于问题的规模，还取决于数据的具体特点。例如，考虑以下函数：

```
int linearSearch(int key, Vector<int> & vec) {
   int n = vec.size();
   for (int i = 0; i < n; i++) {
      if (key == vec[i]) return i;
   }
   return -1;
}
```

它将返回 vec 中 key 第一次出现的索引位置，或者当 key 没有出现时，则返回 -1。由于该实现中 for 循环执行了 n 次，因此，顾名思义，可以预测函数 linearSearch 的性能

439 是 $O(N)$。

另一方面，对于 linearSearch 函数的某些调用会很快地执行。例如，假设你正在寻找的 key 元素恰好出现在这个矢量中的第一个位置。此时，for 循环体内仅执行了一次。如果你非常幸运，总能在矢量的开始位置找到所需要的 key 值，那么 linearSearch 函数将以常数时间运行。

当你分析一个程序的算法复杂度时，总是对最小的可能时间不感兴趣。一般情况下，计算机科学家往往关心的是以下两类复杂度分析：

- **最坏情况下的复杂度**。复杂度分析的最常见类型由确定一个算法在最坏的可能情况下的性能。这种分析是很有用的，因为它允许你给算法复杂度设立一个上界。如果你分析最坏的情况，你可以确保算法的性能将至少比你分析表明的性能要好。你有时候可能会很幸运，但是当算法性能降低时，你可能就不那么自信了。
- **平均情况下的复杂度**。从实际观点来看，如果你算出一个算法在其所有可能的输入数据集上行为的平均值，那么考虑这个算法执行的好坏是有用的。特别是当你的具体输入并不是一个典型的数据集时，平均情况分析对于实际的执行提供了最好的统计估计。然而，问题在于，平均分析经常是难以实施的，且需要相当复杂的数学分析过程。

对 linearSearch 函数而言，最坏的情况发生在矢量中没有 key 时。当 key 不在其中时，函数必须完成 n 次 for 循环。这也就意味着它的性能是 $O(N)$。如果该 key 已知在矢量中，那么 for 循环的平均执行时间是总时间的一半，这也就意味着其平均性能也是 $O(N)$。正如你将会在 10.5 节中所观察到的一样，一个算法的平均情况和最坏情况的性能经常在定性方式上有所不同。这也就意味着：在实际中，同时考虑这两种性能的特征是很重要的。

10.2.6 大 O 符号的正式定义

对于现代计算机科学来说，理解大 O 符号是非常重要的。因此，提供一个更加正式的定义来帮助你理解为什么大 O 的直观模型可以奏效，以及为什么对大 O 公式的简化在实际中是可调整的，将是非常重要的。然而，做这些不可避免地需要一些数学运算。如果你很害
440 怕数学，请不要担心。理解大 O 的实际意义远比遵循本节所呈现的所有步骤要重要。

在计算机科学中，大 O 符号用于表达两个函数之间的关系，通常以下面的表达式表示：

$$t(N) = O(f(N))$$

这个表达式的正式含义是 $f(N)$ 是 $t(N)$ 的一个近似值，并且具有以下特征：一定能够找出一个常量 N_0 以及一个正的常数 C，使得对于每一个大于 N_0 的值，以下条件均成立：

$$t(N) \leq C \times f(N)$$

换句话说，只要 N 足够大，函数 $t(N)$ 的范围总是在一个常量与函数 $f(N)$ 相乘得到的结果范围内。

当大 O 符号用来表示计算复杂度时，函数 $t(N)$ 代表算法的实际运行时间，通常它难以计算。函数 $f(N)$ 是一个更加简单的公式，但随着函数 N 值的变化，它提供运行时间是如何变化的一个合理的量化估值。由于大 O 的数学定义所表达的条件，它确保了实际的运行时间不能够增长得比 $f(N)$ 快。

为了查看正式定义的应用，返回去重新查看选择排序是很有用的。分析选择排序的结构表明了最内层的循环操作执行的次数为：

$$\frac{N^2+N}{2}$$

并且运行时间大概是和这个公式成正比的。当这个复杂度以大 O 符号形式表达时，常量项和低阶项都被省略了，只留下执行时间为 $O(N^2)$ 的断言，事实上这个断言是：

$$\frac{N^2+N}{2} = O(N^2)$$

为了证明这个表达式在大 O 符号正式定义下确实是正确的，你所要做的就是求出常量 C 和 N_0 的值，使得 $N \geqslant N_0$ 时，满足以下条件：

$$\frac{N^2+N}{2} \leqslant C \times N^2$$

441

这个特殊的例子很简单，因为当设定常量 C 和 N_0 均为 1 时，不等式总是成立的。毕竟，只要 N 不小于 1，就可以得到 $N^2 \geqslant N$。因此，这一定会得到以下情况：

$$\frac{N^2+N}{2} \leqslant \frac{N^2+N^2}{2}$$

由于不等式右边的值仅仅是 N^2，这也就意味着：

$$\frac{N^2+N}{2} \leqslant N^2$$

对于所有的 $N \geqslant 1$ 来说都成立，并且 $N \geqslant 1$ 是定义所要求的。

可以使用一个类似的参数来表示任何 k 次多项式，这个多项式通常可以表达成：

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \cdots + a_2 N^2 + a_1 N + a_0$$

它为 $O(N^k)$。和上次一样，你的目标是求出常量 C 以及 N_0，使得在所有的 $N_0 \geqslant N$ 时，都满足以下条件：

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \cdots + a_2 N^2 + a_1 N + a_0 \leqslant C \times N^k$$

和以前的例子一样，可以首先将常量 N_0 的值选择为 1。对于所有的 $N \geqslant 1$，N 的每一个连续的幂总比它的前驱的幂要大，因此，满足：

$$N^k \geqslant N^{k-1} \geqslant N^{k-2} \geqslant \cdots \geqslant N \geqslant 1$$

这个性质反过来意味着：

$$\begin{aligned} & a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \cdots + a_1 N + a_0 \\ \leqslant\ & a_k |N^k + |a_{k-1}| N^k + |a_{k-2}| N^k + \cdots + |a_1| N^k + |a_0| N^k \end{aligned}$$

其中，不等式右边用竖线括起来的系数为其绝对值。通过提取公因子 N^k，可以将上述不等式的右边化简为：

$$(|a_k| + |a_{k-1}| + |a_{k-2}| + \cdots + |a_1| + |a_0|) N^k$$

因此，如果将常量 C 定义为：

$$|a_k| + |a_{k-1}| + |a_{k-2}| + \cdots + |a_1| + |a_0|$$

442

那么，

$$a_k N^k + a_{k-1} N^{k-1} + a_{k-2} N^{k-2} + \cdots + a_2 N^2 + a_1 N + a_0 \leqslant C \times N^k$$

这个结果证明了完整的多项式是 $O(N^2)$。

10.3 递归的终止

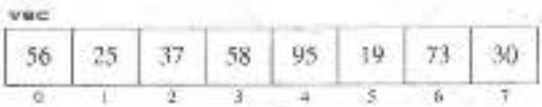
此时，和刚开始学习本章的时候相比，你已经相当了解复杂性分析了。然而，对于一个大的矢量而言，在如何编写一个更加高效的排序算法这一问题上，你并非更进一步。选择排序算法很明显不适合这个任务，因为它的运行时间的增加是和问题规模的平方成正比的。对于大多数以线性顺序处理矢量中元素的排序算法来说，这都是相同的。为了开发出一个更好的排序算法，你需要采取另一种完全不同的方法。

10.3.1 分治策略的作用

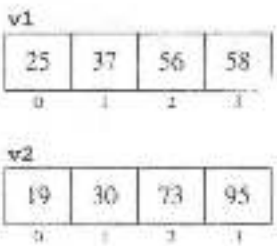
说来奇怪，找到一个更好的排序策略的关键在于认识到像选择排序类算法的平方级行为有一个隐藏的优点。平方级复杂度的基本特征是随着问题规模的加倍，其运行时间将增加 4 倍，反之亦然。如果你将一个平方级的问题划分成两个问题，那么运行的时间同样将减少 1/4。这个事实暗示了：将一个矢量一分为二，然后再运用一个递归的分治算法，可能会减少所需的处理时间。

为了使这个想法更具体，假设你有一个较大的矢量需要对其元素进行排序。如果你将这个矢量一分为二，并且对其每一部分使用选择排序算法，那么将会发生什么？由于选择排序算法所需的时间是平方级的，每一个小的矢量需要原来运行时间的 1/4。当然，你需要对这两部分分别排序，但是处理这两个小的矢量总共所需的时间仍然只有对原来矢量进行排序所需时间的一半。如果结果证明对一分为二的矢量进行排序简化了对一个完整的矢量的排序问题，你将能够大幅度地减少总的执行时间。更重要的是，一旦你发现在一个层面上如何提高其性能，你就可以使用相同的算法来对每部分进行递归排序。

为了确定一个分治策略是否适合于排序问题，你需要回答这样一个问题：将一个矢量划分成两个小的矢量，然后对每个小的矢量分别进行排序是否会对解决一般问题有所帮助。作为一
443 种能够获得对这个问题的透彻了解的方法，假设刚开始你有一个包含了以下 8 个元素的矢量：



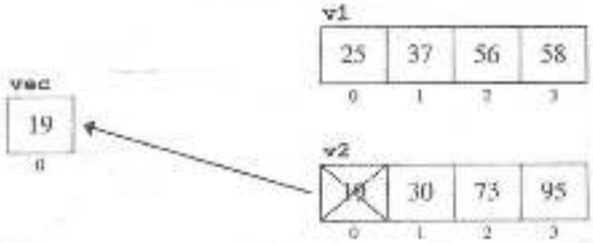
如果将这个矢量中的 8 个元素划分成两个长度均为 4 的矢量，然后再对这两个小的矢量进行排序（记住：递归的稳步跳跃意味着你可以假设递归调用能正确地工作）例如以下情况，每一个更小的矢量已排序：



上述分解有什么作用呢？记住：你的目的是从这些更小的矢量中取值，并将它们放到原来矢量中的正确位置上。这些更小的排序好的矢量是如何帮助你实现这个目标的？

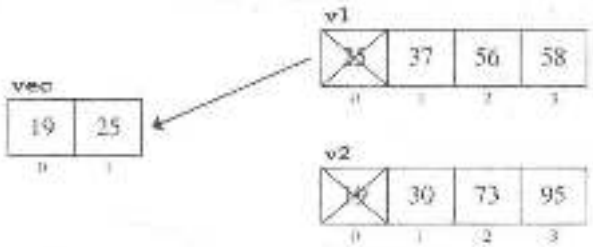
10.3.2 合并两个矢量

正如所发生的一样，基于较小的已经排序好的矢量重新构建完整的矢量是一个比原问题更简单的问题。其中所需的技术称为归并（merging）。归并依赖于这样一个事实：最终排序后的首元素一定是 v1 和 v2 首元素中的那个较小的元素。在这个例子中，所创建的新的矢量中的首元素是 v2 中的 19。如果把这个元素添加进一个空的 vec 中，就要在 v2 中将其删去，并得到以下结果：



下一次，vec 中的下一个元素只能是两个小的矢量中未使用过的第一个元素。这次，将 v1 中的 25 与 v2 中的 30 进行比较后，选择了前者：

444



可以通过从 v1 或者 v2 中挑选出较小的值来简单地继续这个过程，直到重新构建了完整的矢量为止。

10.3.3 归并排序算法

归并操作与递归分解共同产生了一种称为**归并排序**（merge sort）的排序算法，已证明它比选择排序算法具有更高的效率。归并排序算法的基本框架如下：

1. 查看矢量是否为空，或者是否其仅有一个元素。如果为空或者仅有一个元素，那么它一定已经排序好了。这种情况定义了递归的简单情况。
2. 否则，将矢量划分成两个更小的矢量，每一个小的矢量大小只有原矢量对象的一半。
3. 对每个较小的矢量进行递归排序。
4. 清空原矢量对象使它为空。
5. 将两个排序好的小矢量进行合并，形成一个排序好的原始大小的矢量。

图 10-3 展示了归并排序算法的代码，其功能由被巧妙地划分成的两个函数 sort 和 merge 完成。sort 函数的代码直接遵循算法框架。在检查完特殊情况之后，算法将原始的矢量划分成两个较小的矢量：v1 和 v2。sort 函数代码将原矢量中的所有的元素分别复制给 v1 和 v2 后，然后对它们分别进行递归排序，清空原始的矢量，再调用 merge 函数最终在原始矢量中构造出所有的元素排序。

函数 merge 以待排序的原始矢量及划分之后更小的 v1 和 v2 为参数，完成了大部分

的算法。索引 p1 以及 p2 标记了程序在每一个划分的矢量子对象中的执行位置。对于每一次循环，函数从 v1 或者 v2 中挑选出一个元素（这个元素是它们中最小的），并且将这个值添加进 vec 的末尾。当两个较小的矢量中的任意一个元素为空时，函数无须测试即可将另外一个矢量中的剩余元素拷贝到目标 vec 对象中。事实上，由于当第一个 while 循环结束时，其中一个矢量的元素已经耗尽，函数仅将其他的每个矢量中剩余的元素拷贝到目标对象。这些矢量中的任意一个为空，因此所对应的 while 循环将不会执行。

445

```
/*
 * Implementation notes: sort
 * --------------------------
 * This function sorts the elements of the vector into increasing order
 * using the merge sort algorithm, which consists of the following steps:
 *
 * 1. Divide the vector into two halves.
 * 2. Sort each of these smaller vectors recursively.
 * 3. Merge the two vectors back into the original one.
 */

void sort(Vector<int> & vec) {
   int n = vec.size();
   if (n <= 1) return;
   Vector<int> v1;
   Vector<int> v2;
   for (int i = 0; i < n; i++) {
      if (i < n / 2) {
         v1.add(vec[i]);
      } else {
         v2.add(vec[i]);
      }
   }
   sort(v1);
   sort(v2);
   vec.clear();
   merge(vec, v1, v2);
}

/*
 * Implementation notes: merge
 * ---------------------------
 * This function merges two sorted vectors, v1 and v2, into the vector
 * vec, which should be empty before this operation.  Because the input
 * vectors are sorted, the implementation can always select the first
 * unused element in one of the input vectors to fill the next position.
 */

void merge(Vector<int> & vec, Vector<int> & v1, Vector<int> & v2) {
   int n1 = v1.size();
   int n2 = v2.size();
   int p1 = 0;
   int p2 = 0;
   while (p1 < n1 && p2 < n2) {
      if (v1[p1] < v2[p2]) {
         vec.add(v1[p1++]);
      } else {
         vec.add(v2[p2++]);
      }
   }
   while (p1 < n1) vec.add(v1[p1++]);
   while (p2 < n2) vec.add(v2[p2++]);
}
```

446

图 10-3 归并排序算法的实现

10.3.4 归并排序的计算复杂度

你现在已经有了一个基于分治策略实现的 sort 函数。它的效率如何？可以通过对矢量中的数据进行排序并测试其执行时间来衡量它的效率，就计算复杂度来说，以考虑这个算法作为开始是很有帮助的。

当对一个包含 N 个数的列表调用以归并排序实现的 sort 函数时，其运行时间可以划分为以下两部分：

1. 执行当前层的递归分解中的操作所需的总时间。

2. 执行递归调用所需的时间。

在递归分解的顶层，执行非递归操作所消耗的时间与 N 成正比。在矢量子对象中添加元素的循环执行了 N 次，然后调用函数 merge 以便在矢量中原始的 N 个位置重新添加元素。如果你添加这些操作并且忽略常量因素，会发现对 sort 函数的任何单次调用的复杂度（不包括计算其内部的递归调用）需要 $O(N)$ 次操作。

但是递归操作需要花费多长时间呢？为了对一个长度为 N 的矢量中的元素进行排序，必须对两个大小为 $N/2$ 的矢量中的元素进行递归地排序，这须花费一些时间。如果运用相同的逻辑，在当前递归分解层次上，可以迅速确定对每一个小的矢量进行排序所需的时间与 $N/2$ 成正比，再加上任何更下层的递归调用所需的时间。相同的过程继续执行，直到你遇到了简单情况，即矢量仅包含一个元素或者其中没有元素。

解决这个问题所需的总时间为每层递归调用所需的时间之和。通常，图 10-4 展示了分解的结构。当逐渐降低递归层次时，矢量的长度会变得越来越小，同时矢量的个数也会变得越来越多。然而，在每层上所需做的总工作量总是直接与 N 成正比。因此，确定算法总的工作量就是一个求算法分解到底有多少层的问题。

在层次结构中的每一层中，把 N 的值除以 2。因此，层的总数等于在 N 减少到 1 之前它能被 2 整除的总次数。将这个问题表述成数学形式，你需要求出一个 k 值，使得下式成立：

$$N=2^k$$ [447]

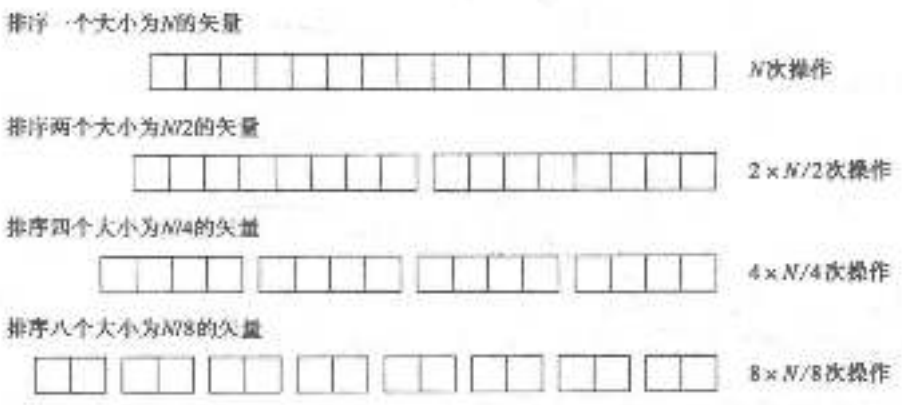


图 10-4 归并排序的递归分解

解这个关于 k 的方程，得到：

$$k=\log_2 N$$

由于 $\log_2 N$ 层中每一层的工作总量都与 N 成正比，所以，工作总量与 $N\log_2 N$ 成正比。

不像其他学科，对数可表示成以 10 为底（常用对数）或者是以数学常量 e 为底（自然对数），计算机科学倾向于使用二进制对数（binary logarithm），它以 2 为底。对数计算使用不同的底，区别仅在于一个常量因子，因此，在讨论关于时间复杂度时，传统的方法是忽略对数的底。因此，归并排序的算法复杂度通常表示为：

$O(N \log N)$

10.3.5 比较 N^2 与 $N \log_2 N$ 的性能

与 $O(N \log N)$ 运行时间的算法相比，需要 $O(N^2)$ 运行时间的算法哪一个更好？评估算法性能改进水平的一种方法是查看实验数据，通过比较选择排序算法与归并排序算法的运行时间来得到直觉上的认识。图 10-5 显示两种算法执行时间的比较实验数据。当排序元素个数 N 为 10 时，归并排序的实现比选择排序的实现慢了 5 倍。当 N 为 100 时，选择排序仍然比归并排序快，但是快得并不多。当将 N 增加到 100 000 时，归并排序比选择排序快了几乎
448 500 倍。在我的计算机上，选择排序算法需要超过两分半钟的时间来对这 100 000 个数进行排序，而归并排序不到半秒就完成了。对于大的矢量来说，归并排序显然标志着一个重大的算法性能改进。

N	选择排序算法	归并排序算法
10	0.000 002 4 s	0.000 012 8 s
50	0.000 044 8 s	0.000 088 7 s
100	0.000 169 s	0.000 196 s
500	0.004 02 s	0.001 10 s
1000	0.015 9 s	0.002 36 s
5000	0.395 s	0.012 9 s
10 000	1.58 s	0.027 s
50 000	39.6 s	0.156 s
100 000	158.7 s	0.324 s

图 10-5 选择和归并排序算法的执行时间实验数据比较

可以通过比较两个算法的时间复杂度公式得到更多的相同信息，如下所示：

N	N^2	$N \log N$
10	100	33
100	10 000	664
1000	1 000 000	9965
10 000	100 000 000	132 877

随着 N 变大，每一列的数字都变大，但是 N^2 这列要比 $N \log N$ 这列增长得快得多。因此，对于长度范围更大的矢量而言，基于 $N \log_2 N$ 的排序算法将更有用。

10.4 标准的算法复杂度类别

在程序设计中，大多数算法属于一些常见的复杂度类别中的一种。最重要的复杂度类别显示在图 10-6 中，这些算法复杂度类别都有统称，并且有相对应的大 O 表达式以及一个代表该类复杂度的典型算法。

图 10-6 中的类别是严格按照复杂度的递增次序呈现的。如果你需要在一个 $O(\log N)$ 时间的算法与另外一个 $O(N)$ 时间的算法之间做出选择，随着 N 的变大，第一个算法总是
449 优于第二个算法。当 N 较小时，大 O 计算也许会使得一个理论上低效的算法表现出低复杂度。另一方面，随着 N 变大，总会存在这样一个点：效率的理论区别会成为决定性的因素。

常量	$O(1)$	在一个矢量中返回第一个元素
对数	$O(\log N)$	在一个排序好的矢量中进行二分查找
线性	$O(N)$	在一个矢量中进行线性查找
$N \log N$	$O(N \log N)$	归并排序
平方	$O(N^2)$	选择排序
立方	$O(N^3)$	传统的矩阵相乘算法
指数	$O(2^N)$	汉诺塔

图 10-6 标准的算法类别

这些复杂度类别效率之间的差别实际上是深奥的。你可以通过查看图 10-7 中的曲线图来认识这些不同的复杂度函数之间的相互关系，这个图将这些复杂度函数绘制在一个传统的线性坐标上。遗憾的是，由于 N 的值都很小，因此这个图表示的是一个不完整并且在有些部分是误导的情况。毕竟，复杂度的分析与 N 值变大根本上是密切相关的。图 10-8 展示了绘制在一个对数尺度上的相同数据，这对于你理解这些函数是如何在一个更广阔范围内的值增长是非常有帮助。

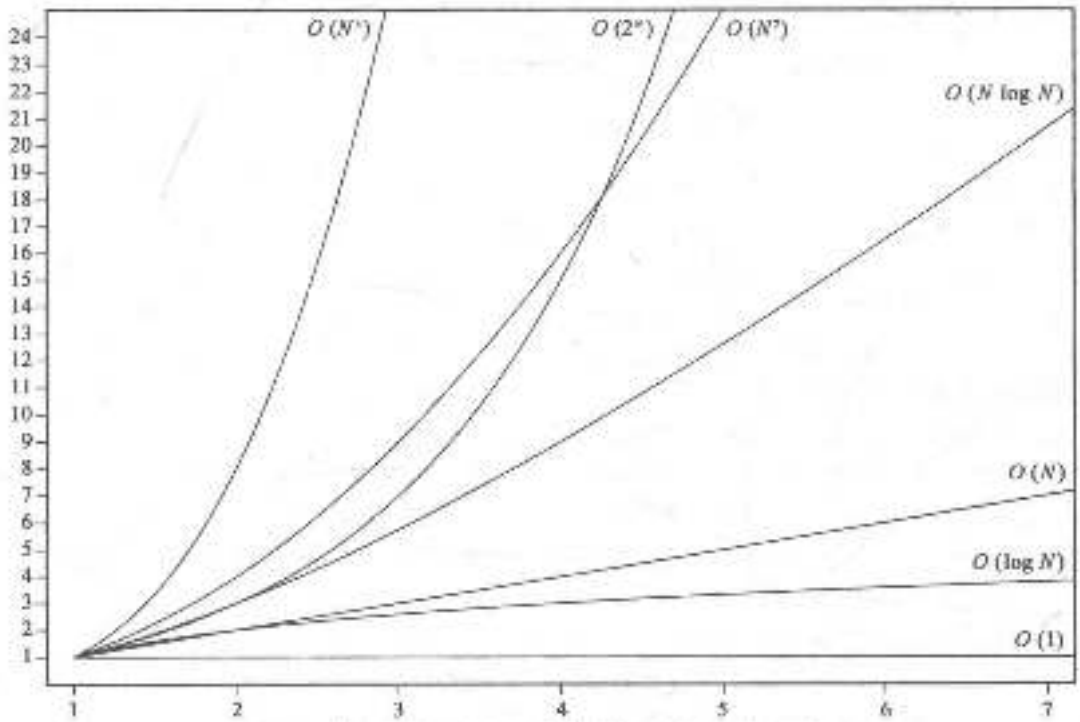


图 10-7 标准的算法复杂度类别的增长特性：线性平面图

算法可分为常量、线性、平方、立方复杂度类别，并且这些复杂度类别均属于一个称为**多项式算法**（polynomial algorithm）的系列一部分，对于某个常量 k 而言，该类算法执行时间为 N^k。其中一个有用的性质是：在图 10-8 的对数坐标图中，任何 N^k 函数的图形总是一条直线，且其斜率与 k 成正比。观察图 10-8，你会发现 N^k 的函数（无论 k 有多大）总是比以 2^N 为代表的指数函数增长得缓慢，而指数函数会随着 N 的增长持续向上激增。这个性质对于为现实问题寻找一个可行的算法有着重要的意义。尽管选择排序展示了平方算法对于大 N 有着严重的性能问题，而复杂度为 $O(2^N)$ 的算法效率则更加低下。作为一个经验法则，

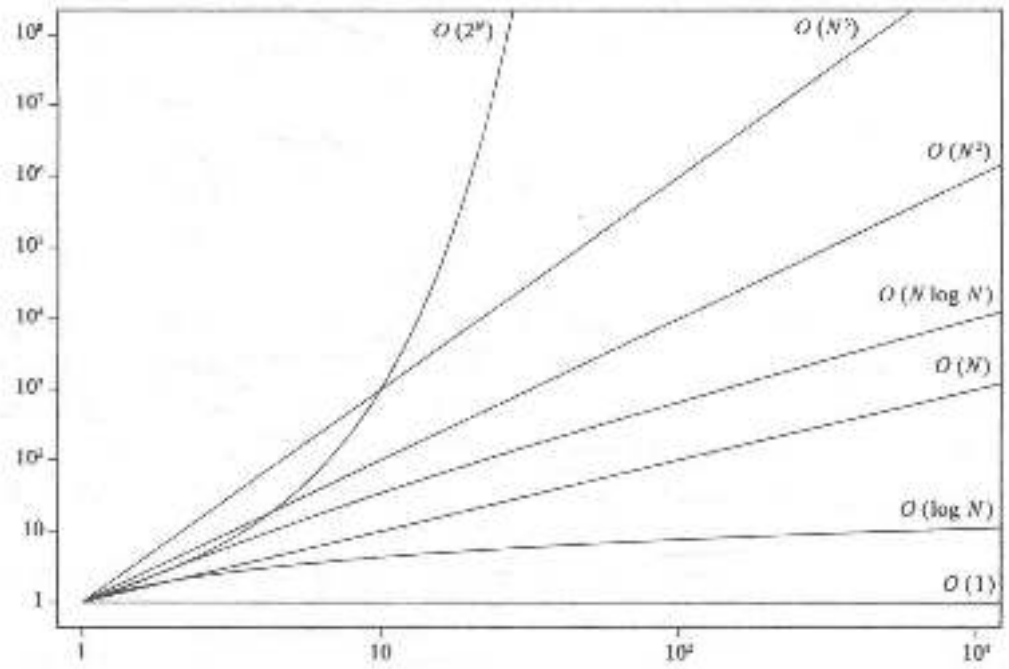


图 10-8 标准的算法复杂度类别的增长特性：对数平面图

计算机科学家将是否存在一个在多项式时间内可解的算法问题分为**易于处理的**（tractable）和**不易于处理的**（intractable）。易于处理的问题意味着它们适合在计算机上实现。

遗憾的是，商业上许多重要的问题采用我们所知的算法都需要指数级的时间，其中一个就是第 8 章介绍的子集求和问题，它来源于几个实际应用问题。另外一个就是**旅行商问题**（traveling salesman problem），该问题是寻找这样一条最短路线：一个旅行者由起点出发，能够访问由多个码头连接的 N 个城市，最后再回到原点。正如大家都知道的，子集求和问题与旅行商问题二者都不可能在多项式时间内求解。所有最著名的算法在最坏情况下都具有指数级性能，并且在生成所有可能路径或者比较其代价时其效率都是相等的。通常，对上述每一个问题的求解方法是尝试每一种可能，这需要指数级的时间。另外，没有人能够确切地证明对这个问题的求解存在非多项式的算法。也许存在某些能够解决这些问题的更聪明的算法。若如此，现在认为困难的许多问题将会变为可解的。

像子集求和问题或者旅行商问题是否可以在多项式时间内求解这一问题，是计算机科学乃至在数学中的一个最重要的开放性问题。这个问题称为 **P 与 NP 问题**（P versus NP problem），解决这类难题的悬赏奖金高达一百万美元。

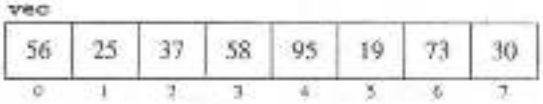
10.5 快速排序算法

即使本章早些时候出现的归并排序在理论上效果很好，并且有一个最坏情况的复杂度 $O(N \log N)$，但该算法在实际中并不常用。取而代之的是，今天所使用的大多数排序程序都是基于一种称为快速排序的算法，它是由英国计算机科学家 C.A.R（托尼）霍尔发明的。

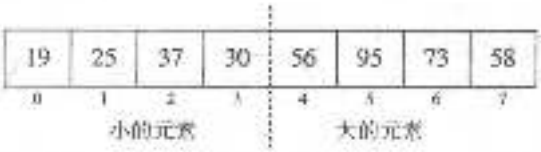
快速排序算法以及归并排序算法都采用了分治策略。在归并排序算法中，原始的矢量划分成两部分，每一部分单独地排序。之后，将它们进行归并以最终完成矢量的排序。然而，假设你采取一种不同的方法来划分这个矢量，如果你通过对这个矢量做一个初始的遍历作为

这个过程的开始，并且修改元素的位置，以使在某种定义上“小”的元素出现在矢量的开始，“大”的元素出现在矢量的末尾，那么将会发生什么？ 450 ~ 452

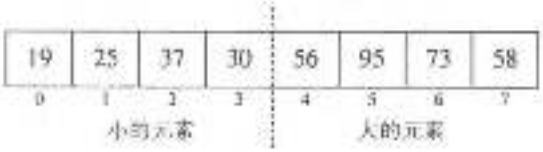
例如，假设你想要排序的矢量的初始状态正如之前讨论的归并排序的元素一样：



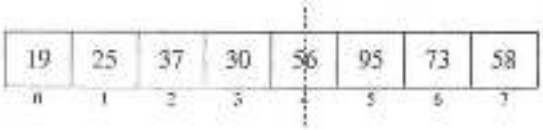
由于这些元素中有一半大于50，而另一半比50小得多，此时，定义那些小于50的元素为小，定义那些大于等于50的元素为大是有意义的。如果可以找出一种方法来重新排列这些元素，以便所有小的元素出现在矢量的开头而所有的大的元素出现在它的末尾，那么就能把这个矢量划分为下图所示的形式，该划分显示了一种可能的划分，其中小的元素以及大的元素分别出现在划分边界的两侧：



当矢量中的元素以这种方式被划分成两部分后，剩下要做的就是排序每一部分的元素，这可以递归调用排序算法。由于在分界左边的所有的元素都小于其右边的元素，所以最终的结果将会是一个完全排好序的矢量：



如果在每一次循环中，你总能在“小”元素与“大”元素之间选择一个最优的边界，那么这个算法每一次都会将矢量划分成一半，并且具有和归并排序算法等价的性能。事实上，快速排序算法会挑选某个矢量中已经存在的元素，并且用它的值代表在“大”元素和“小”元素之间的分界线。然而在这个练习中，你有机会探索更高效的策略，策略之一就是挑选第一个元素（原始矢量中的56），并且使用其值作为边界值。当这个矢量被重新排序时，这个边界值将取一个特定的索引值，而不是处于分界线两边的位置之间，如下图所示：



453

就此看来，这个递归调用一定在矢量的索引位置0与3之间，以及在索引位置5与7之间对其元素进行排序，其中，索引位置4是矢量的分界线。

正如归并排序，快速排序的简单情况是已排序好的大小为0或1的矢量。快速排序算法的递归部分由下面的步骤构成：

1. 选择一个元素作为“大”元素和“小”元素之间的边界。这个元素在传统上被称为基准（pivot）。到目前为止，选择任何元素充当这个基准都是充分的，但是最简单的策略就是选择矢量中的首元素作为基准。

2. 对矢量中的元素重新排序，以使“大”的元素移动到矢量的末尾，而“小”的元素移动到矢量的开头。更正式地说，这一步的目的是：将元素划分到基准的周围，从而使得在边界线左边的所有元素都小于基准，而在边界线右边的所有元素都大于或等于基准，这个过程被称为划分（partitioning）矢量，它将会在下节进行详细讨论。

3. 将划分的每一部分的矢量元素进行排序。由于所有出现在基准边界左边的元素都要严格地比其右边的元素小，因此，对基准边界划分的每一部分的矢量元素进行排序，将使得整个矢量元素处于有序。此外，由于算法使用了分治策略，这些被划分为更小的矢量可以采用递归的快速排序来进行排序。

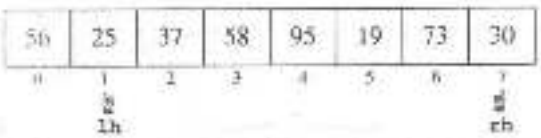
10.5.1 划分矢量

在快速排序算法的划分步骤中，其目的是对元素重新排序，从而使它们可被划分成三种类型：那些比基准小的元素；位于边界位置的基准元素本身；那些大于等于基准元素的元素。划分的棘手部分在于：在不能使用任何额外的存储空间的前提下，对这些元素重新排序，实现它的典型方法是通过交换一对元素来完成的。

托尼·霍尔初始的划分方法正好可以用英语做简单的解释。和之前章节一样，后面的讨论假设基准元素为矢量的首元素。由于在算法划分阶段开始之前，基准元素的值已被选择，因此你可以立即区分一个元素值相对于基准元素是“大”还是“小”。托尼的划分算法以如下步骤执行：

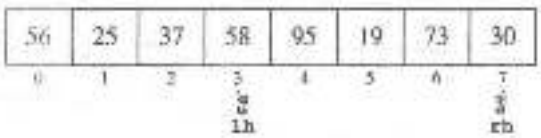
1. 此时，忽略处于索引位置0的基准元素，把精力放在剩下的元素中。使用两个索引值 lh 和 rh 来记录矢量中剩余元素的开始索引位置与末尾索引位置，如下图所示：

454

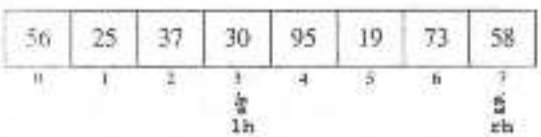


2. 将 rh 向左移动，直到与 lh 一致，或者指向一个所包含的值小于基准元素值的元素。在此例中，索引位置7的元素值30已是一个小值，所以 rh 索引不需要移动。

3. 将 lh 向右移动，直到与 rh 相同，或者指向一个所包含的值大于基准元素值的元素。在此例中，lh 索引必须向右移动，直到它指向一个其值大于56的元素，得到了如下所示的结构：

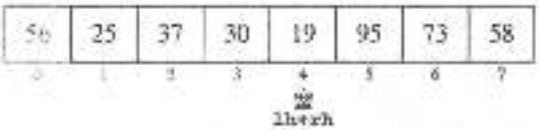


4. 如果 lh 和 rh 索引没有达到相同的位置，将 lh 与 rh 位置所指的元素进行交换，得到矢量中的内容变为下图所示的情形：



5. 重复步骤2～4，直到 lh 和 rh 相遇为止。例如，在下一次索引位置移动过程中，第

4 步的交换操作将 19 与 95 交换。当这种情况发生时，下一次的步骤 2 会将 rh 向左移动，直到与 lh 匹配为止，如下图所示：



6. 除非选择的基准元素恰好是整个矢量中最小的元素（这段代码中应该包含对这种情况的一个特殊检查），否则，lh 与 rh 索引位置所指的元素将会是矢量右边元素的最小值。剩下的一步就是将这个值与出现在矢量开始的基准元素相交换，如下图所示： 455



注意：至此矢量中这个结构满足划分步骤的需求。基准值被标记在边界位置，边界位置左边的每一个元素都比基准元素值小，而其右边的每一个元素都比基准元素值大。

图 10-9 展示的是采用快速排序算法实现的一个 sort 函数。

```
/*
 * Implementation notes: sort
 * --------------------------
 * This function sorts the elements of the vector into
 * increasing numerical order using the Quicksort algorithm.
 * In this implementation, sort is a wrapper function that
 * calls quicksort to do all the work.
 */

void sort(Vector<int> & vec) {
   quicksort(vec, 0, vec.size() - 1);
}

/*
 * Implementation notes: quicksort
 * -------------------------------
 * This function sorts the elements in the vector between index
 * positions start and finish, inclusive.  The Quicksort algorithm
 * begins by "partitioning" the vector so that all elements smaller
 * than a designated pivot element appear to the left of a
 * boundary and all equal or larger values appear to the right.
 * Sorting the subsidiary vectors to the left and right of the
 * boundary ensures that the entire vector is sorted.
 */

void quicksort(Vector<int> & vec, int start, int finish) {
   if (start >= finish) return;
   int boundary = partition(vec, start, finish);
   quicksort(vec, start, boundary - 1);
   quicksort(vec, boundary + 1, finish);
}

/*
 * Implementation notes: partition
 * -------------------------------
 * This function rearranges the elements of the vector so that the
 * small elements are grouped at the left end of the vector and the
 * large elements are grouped at the right end.  The distinction
 * between small and large is made by comparing each element to the
 * pivot value, which is initially taken from vec[start].  When the
 * partitioning is done, the function returns a boundary index such
 * that vec[i] < pivot for all i < boundary, vec[i] == pivot
 * for i == boundary, and vec[i] >= pivot for all i > boundary.
 */
```

图 10-9 快速排序算法的实现

```
int partition(Vector<int> & vec, int start, int finish) {
   int pivot = vec[start];
   int lh = start + 1;
   int rh = finish;
   while (true) {
      while (lh < rh && vec[rh] >= pivot) rh--;
      while (lh < rh && vec[lh] < pivot) lh++;
      if (lh == rh) break;
      int tmp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = tmp;
   }
   if (vec[lh] >= pivot) return start;
   vec[start] = vec[lh];
   vec[lh] = pivot;
   return lh;
}
```

图 10-9 （续）

10.5.2 快速排序算法的性能分析

图 10-10 中出现的是归并排序与快速排序算法实际运行时间的一个直接比较。正如你所看到的那样，快速排序的实现常常比图 10-3 中的归并排序运行速度快几倍。这也是程序员在实际中更加频繁使用快速排序算法的原因之一。此外，两种算法的运行时间以大致相同的方式增长。

然而，图 10-10 中所显示的实验结果隐藏了一个要点。如果快速排序算法选择的基准接近于矢量的中间值，那么划分步骤将会把矢量划分成大小差不多相等的两部分。否则，所划分的两部分中的一个部分可能要比另外一部分大得多，这也就违背了分治策略的目标。在一个矢量中，随机地选择一个元素作为其基准元素，则快速排序往往表现得很好，具有 $O(N \log N)$ 的平均复杂度。最差的情况（也就是一个矢量中的元素早已有序），算法的性能会退化到 $O(N^2)$。除最坏情况，快速排序实际上要比其他大多数的排序算法快得多，因此对于大多数排序程序来说，快速排序已成为通用排序过程的一个标准选择。

456
/
457

N	归并排序	快速排序
10	0.000 012 8 s	0.000 001 4 s
50	0.000 088 7 s	0.000 012 0 s
100	0.000 196 s	0.000 028 8 s
500	0.001 10 s	0.000 200 s
1000	0.002 36 s	0.000 456 s
5000	0.012 9 s	0.002 84 s
10 000	0.027 s	0.006 08 s
50 000	0.156 s	0.036 5 s
100 000	0.324 s	0.077 4 s

图 10-10 归并排序与快速排序的实验性能比较

这里有几种可供你使用的策略，以提高基准在实际中接近矢量中间值的可能性。一种最简单的方法是在快速排序的实现中随机地选择基准。尽管在随机选择基准过程中仍然有可能选择了一个次优的基准值，但是在递归分解中的每层，每次不可能重复地产生这一相同的错误。此外，原始矢量中的分布并不总是最坏情况。对于任何输入来说，随机地选择基准确保了算法的平均复杂度将会为 $O(N \log N)$。另外一个可能的策略就是在矢量中挑选出一些值，典型的是 3 个或 5 个，然后挑选出这些值的中间值作为基准，你可以在习题 6 中仔细地研究这个策略。

当你试图用这种方式改进算法的时候你要小心。挑选出一个好的基准会提高算法性能，但同时也需花费一些时间。如果算法在选择基准时所花费的时间比其所提高算法性能的时间要长，你应该结束这种基准选择方法的实现，而不应为加速该实现为目标。

10.6 数学归纳法

在本章的前面小节，我要求你相信这个事实，以下的求和

$$N+N-1+N-2+\cdots+3+2+1$$

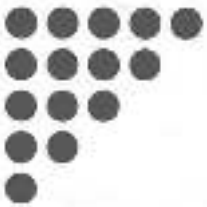
458

可以简化成更便于处理的公式：

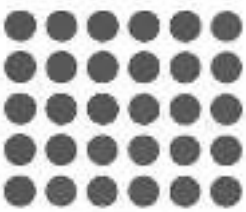
$$\frac{N^2+N}{2}$$

如果你对这个简化有所怀疑，你如何来证明这个简化公式在实际中是正确的呢？

事实上，这里有几种不同的你可以尝试使用的证明方法。其中一种可能就是以几何形式来表达这个原始的求和。例如，假设 N 等于 5。然后，如果你将求和表达式中的每一项表示成一行圆点，这些圆点形成了以下三角形：



如果你拷贝这个三角形并对其进行翻转，那么这两个三角形的组合形成了一个矩形，下图用灰色表示了下面的三角形：



由于这个图案现在是一个矩形，其总的圆点数（包括黑色以及灰色）很容易计算。在这张图中，总共有五行，并且每一行都有六个圆点，所以两种颜色的圆点总数就是 5×6，也就是 30。由于这两个三角形是完全相同的，并且这些圆点中的一半恰好是黑色的，因此，黑色圆点的数目就是 30/2，即 15。在更普遍的情况下，这里有 N 行，并且每一行包含了 $N+1$ 个圆点，则原始三角形中黑色圆点的数目就是：

$$\frac{N\times(N+1)}{2}$$

然而，用这种方式证明一个公式是正确的有一些潜在的缺陷。其中一个缺陷就是：在这公式中的几何参数并非是许多计算机科学家想要的形式。此外，构建这种类型的参数需要你想出正确的几何透视，而这对于每一个问题都是不同的。采用一个更普遍的证明策略，从而能应用于许多不同的问题会更好。

459

计算机科学家们通常采用证明以下这一命题的方法被称为**数学归纳法**（mathematical induction）：

$$N + N\text{-}1 + N\text{-}2 + \cdots + 3 + 2 + 1 = \frac{N \times (N+1)}{2}$$

数学归纳法应用于：当你想要展示一个命题对于一个整数 N 从某个初始点开始的所有值都是正确的。这个初始起点被称为归纳的基（basis），典型的是 0 或 1。这个过程包含以下步骤：

- 证明基本情况。第一步是证明当 N 为基值时，这个命题是正确的。在大多数情况中，这一步仅仅是将基值带入一个公式，并且查看所期望的关系是否成立。
- 证明归纳情况。第二步是证明如果你假设这个公式对于 N 是正确的，那么它对 $N+1$ 也是正确的。

作为一个例子，这里向你展示如何使用数学归纳法来证明以下公式：

$$N + N\text{-}1 + N\text{-}2 + \cdots + 3 + 2 + 1 = \frac{N \times (N+1)}{2}$$

对于所有 N 大于等于 1 时，它确实是正确的。第一步就是证明基本情况，此时 N 等于 1。这步证明是非常容易的，你所要做的就是在公式的两边用 1 代替 N，然后对等式两边求值：

$$1 = \frac{1 \times (1+1)}{2} = \frac{2}{2} = 1$$

为了证明归纳情况，你首先假设以下命题对于 N 是正确的：

$$N + N\text{-}1 + N\text{-}2 + \cdots + 3 + 2 + 1 = \frac{N \times (N+1)}{2}$$

这个假设被称为归纳假设（inductive hypothesis）。你现在的目标是证明：对于 $N+1$ 来说，命题也是成立的。换句话说，为了证明当前公式的正确性，你需要做的就是证明以下公式：

460

$$N\text{-}1 + N + N\text{-}1 + N\text{-}2 + \cdots + 3 + 2 + 1 = \frac{(N+1) \times (N+2)}{2}$$

如果观察等式的左边，应该注意到：以 N 开始的项的序列和你的归纳假设左边的部分是完全相同的。由于你假设了归纳假设是正确的，你可以用解析表达式等价地替换，因此你试图证明的命题的左边如下所示：

$$N+1 + \frac{N \times (N+1)}{2}$$

从这里开始，剩余的证明就是简单的代数运算：

$$N+1 + \frac{N \times (N+1)}{2}$$

$$= \frac{2N+2}{2} + \frac{N^2+N}{2}$$

$$= \frac{N^2+3N+2}{2}$$

$$= \frac{(N+1) \times (N+2)}{2}$$

这个推导的最后一行恰恰是你正在寻找的结果，因此整个证明完毕。

许多学生需要花时间来适应数学归纳法的思想。乍一看，归纳假设在某种程度上看起来像是“欺骗”；毕竟，你的假设正是你想要证明的命题。实际上，数学归纳法的过程只不

过是证明的一个无限族，无限族中的每一个都以相同的逻辑进行。一个典型的例子就是基本情况证明了对于 $N=1$ 命题是正确的。一旦你已经证明了基本情况，你可以采取下面的一串推理：

既然我已经知道了 $N=1$ 对于公式是正确的，可以证明 $N=2$ 对于公式也是正确的。
既然我已经知道了 $N=2$ 对于公式是正确的，可以证明 $N=3$ 对于公式也是正确的。
既然我已经知道了 $N=3$ 对于公式是正确的，可以证明 $N=4$ 对于公式也是正确的。
既然我已经知道了 $N=4$ 对于公式是正确的，可以证明 $N=5$ 对于公式也是正确的。
……

上述过程的每一步，你都可以通过运用证明归纳情况的逻辑写出一个完整的证明。数学归纳法的力量来自于这样一个事实：你实际上并不需要单独写出每一步的细节。 461

在某种程度上，数学归纳法的过程看起来像是反方向的递归过程。如果你试图详细地解释一个典型的递归分解，这个过程经常听起来像这样：

为了计算 $N=5$ 时的函数值，我需要知道 $N=4$ 时的函数值。
为了计算 $N=4$ 时的函数值，我需要知道 $N=3$ 时的函数值。
为了计算 $N=3$ 时的函数值，我需要知道 $N=2$ 时的函数值。
为了计算 $N=2$ 时的函数值，我需要知道 $N=1$ 时的函数值。
$N=1$ 代表了一种简单情况，所以我可以直接地返回结果。

归纳法以及递归都需要你进行一个信心的飞跃。当你编写一个递归函数时，这个转变要求你相信函数调用的所有简单情况都可以工作，而无须关注其细节。做出归纳假设需要很多相同的心智训练。在两种情况下，你必须将你的思维一直限制在一个解决方案的层面上，而不能在求解过程中陷入到细节中。

本章小结

本章你所获得的最有价值的概念就是：处理一个问题的不同算法在其各自的性能上有很大的变化。选择一个复杂度性能好的算法可以减少对许多排序问题处理所需的时间。通过本章中所呈现的表格，显著地说明了不同排序算法的实际运行的不同的时间性能。例如，当对一个含有 10 000 个整数的矢量进行排序时，快速排序比选择排序快 250 倍；随着矢量尺寸的增大，这些算法在效率上的不同将变得更加明显。

本章中的其他要点包括：

- 大多数算法问题可以通过一个代表该问题大小的整数 N 来刻画。对于那些大的整数，整数的大小提供了一种有效描述问题规模的方法；且该类算法是对数组或者矢量进行操作的，问题规模常用元素的数目来定义。
- 对效率最有效的测量方法就是时间复杂度，它被定义为问题的规模与随着问题规模增加的算法性能之间的关系。
- 大 O 表示法提供了一个直观的表示时间复杂度的方法，因为它允许将复杂度关系最重要的方面简化成最简单的形式。 462
- 当你使用大 O 表示法时，你可以通过删除公式中那些随着 N 的变大变得无关紧要的项以及所有的常量项来简化公式。
- 你可以通过观察一个程序所包含的循环的内部结构来预测这个程序的时间复杂度。
- 测量复杂度的两种有效的测量方法就是最坏情况分析以及平均情况分析。平均情况

分析通常很难进行。

- 分治策略能够将排序算法的复杂度从 $O(N^2)$ 降低到 $O(N \log N)$，这是一种重要的性能改进。
- 大多数算法都属于几种常见复杂度算法类别中的一个，这些常见的复杂度类别算法包括：常量、对数、线性、$N \log N$、二次方、三次方，以及指数这些类别。至少当问题被考虑成足够大时，出现在这个列表中较前的算法的复杂度类别要比出现在这个列表中后面的时间复杂度类别更高效。
- 如果问题可以在多项式时间内求解，会被定义成关于一些常量 k 的多项式 $O(N^k)$，这是易于处理的。不存在多项式时间算法的问题被认为是不易于处理的，因为即使问题的规模相对来说较小，处理那样的问题也需要大量的时间。
- 由于快速排序算法在实际中往往表现得很好，因此大多数排序问题都是基于由托尼·霍尔开发的快速排序算法的，虽然它的最差复杂度是 $O(N^2)$。
- 数学归纳法提供了一种通用的技术，以证明一个适用于所有大于或等于一些基值的 N 值的性质。为了应用这种技术，第一步是在基本情况下证明这个性质。第二步，你必须证明如果这个公式对于一个特殊值 N 是成立的，那么它对于 $N+1$ 也是成立的。

复习题

1. 斐波那契函数最简单的递归实现与其迭代版本的实现相比，被认为是低效的。这个事实是否可让你得出一些关于递归解决方案的效率以及迭代解决方案效率的一般性结论呢？
2. 什么是排序问题？
3. 图 10-1 展示的 sort 函数的实现，即使在 lh 与 rh 的位置值相等的情况下也能运行代码来交换 lh 与 rh 位置的元素值。如果你修改代码以使代码在做出交换前进行检查，确保 lh 与 rh 位置值是不同的，那么很可能比原始的算法运行得更快。为什么会出现这种情况？ 463
4. 假如你正在使用选择排序算法对一个含有 250 个值的矢量进行排序，并且发现该算法花费了 50 毫秒的计算时间。如果在相同的机器上使用相同的算法对一个含有 1000 个值的矢量进行排序，你期望的运行时间是多少？
5. 对于计算以下序列和，它的解析表达式是什么？

$$N-N-1+N-2+\cdots+3+2+1$$

6. 用你自己的语言来定义时间复杂度的概念。
7. 判断题：大 O 符号是作为表达时间复杂度的一种方式被发明的。
8. 本章中简化大 O 表示所提出的两种规则是什么？
9. 选择排序算法的运行时间是 $O\left(\frac{N^2+N}{2}\right)$，这种说法在理论上是正确的吗？如果不是的话，那么以这种形式表达的时间复杂度有什么问题？
10. 选择排序的运行时间是 $O(N^2)$，这种说法在理论上是正确的吗？如果不正确，那么以这种形式表达的选择排序算法有什么问题？
11. 为什么通常在大 O 符号中省略对数的底，例如 $O(N \log N)$？
12. 下面函数的时间复杂度是多少？

```
int mystery1(int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < i; j++) {
```

```
            sum += i * j;
         }
      }
      return sum;
   }
```

464

13. 下面函数的时间复杂度是多少？

```
int mystery2(int n) {
   int sum = 0;
   for (int i = 0; i < 10; i++) {
      for (int j = 0; j < i; j++) {
         sum += j * n;
      }
   }
   return sum;
}
```

14. 解释最坏情况下时间复杂度和平均情况下时间复杂度之间的不同。通常情况下，这些度量中哪个较难以计算？
15. 在大 O 符号的形式化定义中，解释常量 C 以及 N_0 的作用。
16. 用你自己的语言解释为什么 merge 函数的运行时间是线性时间。
17. merge 函数的最后两行是：

```
while (p1 < n1) vec.add(v1[p1++]);
while (p2 < n2) vec.add(v2[p2++]);
```

如果这两行代码互换位置会发生什么？解释为什么会发生或为什么不会发生？

18. 本章标识的七种在实际中最常见的复杂度类别是什么？
19. 多项式算法这个术语的含义是什么？
20. 计算机科学家区分易于处理和不易于处理的问题所使用的标准是什么？
21. 在快速排序算法中，划分步骤的结果必须满足什么条件？
22. 快速排序算法的最坏情况及平均情况的时间复杂度分别是多少？
23. 描述采用数学归纳法所涉及的两步证明。
24. 用你自己的语言描述递归与数学归纳法之间的关系。

465

习题

1. 编写一个递归函数：

```
double raiseToPower(double x, int n)
```

通过依赖以下递归公式来计算 x^n：

$$x^n = x \times x^{n-1}$$

这样的一个策略产生了一个以线性时间运行的实现。然而，你可以采取一个递归的分治策略，这个策略利用了以下公式的优点：

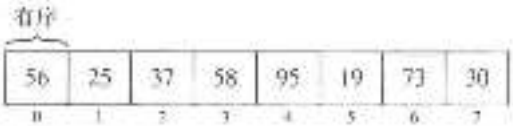
$$x^{2n} = x^n \times x^n$$

使用公式编写一个以 $O(\log N)$ 时间运行的递归版本的函数 raiseToPower。

2. 其他一些排序算法也具有选择排序的行为。其中一个最重要的排序算法就是**插入排序**（inserting sort）算法，这个算法进行以下操作：首先对矢量中的每一个元素依次进行检查，就像在选择排序算法中那样。然而，在这个过程中，你的目标不是找出矢量中剩余元素的最小值并将其放入到正确的位置，而是确保到目前为止所有考虑的元素都处在正确的位置。尽管这些值可能会随着处理元素的

增加而移动，但是它们本身形成了一个有序的序列。

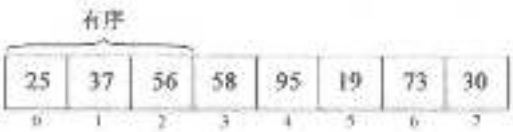
例如，如果你再次考虑本章中用来排序的示例数据，插入排序算法的第一次循环没有做任何工作，因为只含有一个元素的矢量总是有序的：



在下一次循环中，你需要将25放在你已经看见过的元素中的正确的位置上。这也就意味着你需要交换56和25的位置以便使矢量达到以下状态：



466 在第三次循环中，你需要找出37应该放在哪里。为此，你需要将前面的元素向后移动（你知道这些元素它们彼此是有序的），用来寻找37这个元素应该所处的正确位置。随着算法的进行，你需要将每一个大的元素向右移动一个位置，最终产生了一个欲插入元素的空间。此时，56向右移动一个位置，同时37向前移动一个位置。因此，经过第二次循环之后，矢量的状态如下图所示：



每次循环之后，矢量起始部分总是有序的。这也就意味着用这种方法循环访过的所有位置整个矢量都将被排序。

插入排序在实际中并非常重要的，因为对于那些已经或多或少以正确顺序排序的矢量来说，它的运行时间是线性的。因此，对于那些只有很少的元素且不是有序的大的矢量而言，采用插入排序对其元素进行重新排序是很有意义的。

编写一个使用插入排序算法实现的 sort 函数。构建一个非正式的参数来展示插入排序最坏情况下的时间复杂度是 $O(N^2)$。

3. 编写一个函数，跟踪 sort 函数处理一个随机选择的矢量的运行时间。并使用该函数编写一个程序，它产生一个预先确定大小的集合程序运行时间的表格，如以下示例的运行结果所示：



测量这个排序程序运行的系统时间的最好方式是使用 ANSI clock 函数，这个函数可由接口 ctime 导出。函数 clock 不需要参数，并且返回计算机处理器用来执行当前程序所花费的时间总量。度量单位以及 clock 函数运行结果的存储类型都取决于机器的类型，但是你总可以使用下面的表达式将依赖于系统的时钟单位转换成秒：

467

```
double(clock()) / CLOCKS_PER_SEC
```

如果你用变量 start 以及 finish 分别记录程序起始和结束时间，你可以使用下面的代码来计算：

```
double start = double(clock()) / CLOCKS_PER_SEC;
...执行一些计算...
```

```
double finish = double(clock()) / CLOCKS_PER_SEC;
double elapsed = finish - start;
```

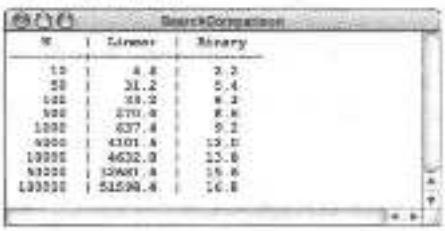
遗憾的是，因为不能确保系统时钟单位足够精确以至于能够测量出运行时间，所以计算一个运行很快的程序所需的时间需要一些巧妙的方法。例如，如果你使用这种策略来计算排序 10 个整数的时间，那么代码片段最后的 `elapsed` 值很可能是 0。产生这种现象的原因是：大多数机器上的处理单元可以在单个时钟周期范围内执行许多指令——几乎可以对一个含有 10 个元素的矢量进行完整的排序。由于系统内部时钟在此期间可能是相同的，因此 `start` 以及 `finish` 记录的值可能是相同的。

避开这个问题最好的方法是在两个对 `clock` 函数的调用之间多次重复这个计算。例如，如果你想要确定排序 10 个数需要花费多长时间，你可以连续对 10 个数执行 1000 次排序，然后将总的运行时间除以 1000。这种策略提供了一种更加精确的计时方法。

4. 假设你已经知道一个整数数组中元素值的范围为 0 到 9999。证明编写一个复杂度为 $O(N)$ 的算法来对这个数组进行排序是可能的。实现你的算法，并且通过使用习题 3 中概括的策略来取实际测量的方式来评估该算法的性能。当 N 取较小值时，解释该算法与选择排序算法相比，为什么缺乏效率。

5. 编写一个能够产生比较两个算法（线性查找和二分查找）性能的表格程序，它们都在一个有序的 `Vector<int>` 类对象中随机地查找一个整数关键字。线性查找算法仅仅是依次地检查矢量中的每一个元素以确定在矢量中是否存在待查找的关键字。图 7-5 中的二分查找算法应用于元素类型为字符串的矢量，它使用了一种分治策略，通过检查矢量的中间元素，然后再决定在剩下元素中的哪一部分进行查找。 468

对这个问题以表格形式产生两种算法的性能比较，不像习题 3 那样计算时间，而是计算矢量中元素比较的次数。为了确保结果不是完全的随机，你的程序应该进行多次独立的试验，然后再算出结果的平均数。程序运行的一个示例如下图所示：



6. 修改快速排序算法的实现方法，不同于挑选矢量中的第一个元素作为基准，划分（partition）函数选择第一个元素、中间元素、以及最后元素的中位数作为基准。

7. 对于大的矢量来说，尽管时间复杂度为 $O(N \log N)$ 的排序算法明显要比时间复杂度为的 $O(N^2)$ 算法更加高效，当 N 取较小值时，像选择排序一样的简单二次算法往往意味着它们有更好的性能。这一事实提出了可开发一种将两种算法相结合的策略：对于大的矢量使用快速排序算法，而当矢量中元素数目小于某个被称为**分界点**（crossover point）的阈值时，使用选择排序算法。将两种不同的算法结合在一起，并利用各自最好的特性的方法被称为**混合策略**（hybrid strategy）。

采用综合快速排序算法和选择排序算法的混合策略重新实现 `sort` 函数。对分界点的每一个不同值分别进行试验，当小于分界点值时，函数实现选用选择排序，并且确定了哪些值提供了最佳性能。分界点的选取决于你电脑特定的时间特性，并且随着系统的修改而修改。

8. 对排序问题另一个有趣的混合策略是：以一个快速排序的递归实现开始，当矢量的大小低于某个特定的阈值时，只简单地返回。当该函数返回时，矢量并没有排序，但是所有的元素都相对地接近于它们最终的位置。此时，你可以对整个矢量使用习题 2 中所呈现的插入排序算法来处理剩下整个矢量元素以固定剩余的问题。在矢量中的大多数元素已经有序的情况下，插入排序的运行时间是线性的，因此以上两步过程比每一个单独的算法运行得都要快。编写一个以这种混合方法实现的 `sort` 函数。 469

9. 假设你有两个函数 f 和 g，当 N 取不同的值时，都有 $f(N)$ 小于 $g(N)$。使用大 O 符号的正式定义证明：

$$15f(N) + 6g(N)$$

的时间复杂度为 $O(g(N))$。

10. 使用大 O 符号的形式化定义证明 N^2 的时间复杂度为 $O(N^2)$。

11. 使用数学归纳法证明以下性质对于所有的整数 N 都是正确的。

a) $1 + 3 + 5 + 7 + \ldots + 2N-1 = N^2$

b) $1^2 + 2^2 + 3^2 + 4^2 + \ldots + N^2 = \frac{N \times (N+1) \times (2N+1)}{6}$

c) $1^3 + 2^3 + 3^3 + 4^3 + \ldots + N^3 = (1 + 2 + 3 + 4 + \ldots + N)^2$

d) $2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^N = 2^{N+1} - 1$

12. 习题 1 说明了能够在 $O(\log N)$ 时间内计算 x^n。这一事实反过来可以编写运行时间为 $O(\log N)$ 的函数 `fib(n)` 的实现，它比传统的迭代版本的函数实现运行得要快得多。为此，你需要依靠有点令人惊讶的事实：斐波那契函数与被称为黄金比例（golden ratio）的值有关。黄金比例自古希腊数学时代就闻名遐迩。黄金比例通常被设计为用特定的希腊字母 phi（φ）表示，并被定义为其值满足以下等式：

$$\varphi^2 - \varphi - 1 = 0$$

由于这是一个二次等式，因此该等式实际上有两个解，如果你运用二次方程公式，你将会发现这两个根为：

$$\varphi = \frac{1+\sqrt{5}}{2}$$

$$\hat{\varphi} = \frac{1-\sqrt{5}}{2}$$

470

在 1718 年，法国数学家亚伯拉罕·棣·莫弗（Abraham de Moivre）发现了第 n 项斐波那契数字可以被表示成以下近似形式：

$$\frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}$$

此外，由于 $\hat{\varphi}^n$ 总是很小，所以该公式可以被简化为：

$$\frac{\varphi^n}{\sqrt{5}}$$

四舍五入为最接近的整数。

使用这个公式以及习题 1 中的 `raiseToPower` 函数，编写一个运行时间为 $O(\log N)$ 的函数 `fib(n)` 的实现。一旦你实验验证了公式对于斐波那契数列中的前几项是正确的，使用数学归纳法证明公式：

$$\frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}$$

实际上可以计算斐波那契数列的第 n 项。

13. 如果你准备好挑战真正的算法，编写函数：

```
int findMajorityElement(Vector<int> & vec);
```

它以一个非负的整型矢量为参数，并且返回**多数元素**（majority element）。该返回值被定义为一个绝大多数元素（超过百分之五十）出现的位置。如果多数元素不存在，则函数返回 -1 以表示这一事实。你编写的函数必须满足以下条件：

- 运行时间必须是 $O(N)$。

- 必须使用 $O(1)$ 的额外空间。换句话说，该函数可能使用个别的临时变量，但不会为任何数组或者矢量分配额外的存储空间。此外，这个条件排除了递归的解决方案，因为存储栈帧所需的空间将会随着递归深度的增加而增加。
- 不能修改矢量中的任何值。

这个问题的难点在于想出算法，而不是怎样去实现它。以几个矢量例子做实验，并且看看你是否能够想出一个满足这些条件的高效策略。

14. 如果你喜欢前面的问题，这里有一个更为挑战的问题，它曾经作为被欢的面试题。假设你有一个含有 N 个元素的矢量，并且这个矢量中的每一个元素都有一个在 1 到 $N-1$ 范围内的值。假设这个矢量有 N 个元素，而只有 $N-1$ 个可能的值存储在其中。当然，这里一定有一个重复的值，但是你知道这里有一个数学家所说的鸽巢原理（pigeonhole principle）：如果鸽子的数目比鸽笼的数目要多，那么某个鸽笼中的鸽子数目肯定超过一个。 471

在这个问题中，你的任务就是编写一个函数：

```
int findDuplicate(Vector<int> vec);
```

它以元素值范围为 1 到 $N-1$ 的矢量作为参数，并且返回其中的一个重复值。这个问题的难点在于设计一个算法以使你的实现方法遵循先前习题中的条件：

- 运行的时间必须是 $O(N)$。
- 必须使用 $O(1)$ 额外的空间。换句话说，该函数可能使用个别的临时变量但是不会为任何数组或者矢量分配额外的存储空间。此外，这个条件排除了递归的解决方案，因为存储栈帧所需的空间将会随着递归深度的增加而增加。
- 不能修改矢量中的任何值。

例如，对于这个问题可以很容易写出一个运行二次方时间的解决方案，如下所示：

```
int findDuplicate(Vector<int> & vec) {
   for (int i = 0; i < vec.size(); i++) {
      for (int j = 0; j < i; j++) {
         if (vec[i] == vec[j]) return vec[i];
      }
   }
   error("Vector has no duplicates");
   return -1;
}
```

难点在于将其优化成运行时间为线性时间的算法。 472

第 11 章
Programming Abstractions in C++

指针和数组

奥兰多迅速浏览了一遍，然后用右手食指点着，念出与这件事有密切关系的下列事实。
473 ——弗吉尼亚·伍尔夫，《奥兰多》(*Orlando*)，1928

本书大多数程序都借助于抽象数据类型来表示复合对象。实际上，这个策略很明显是非常正确的。当你采用面向对象语言（例如 C++）来编写程序时，你应该尽可能地采用类库提供的抽象类型，并且尽可能规避复杂的底层细节。同时，多了解一下 C++ 是如何表示数据的也很有帮助。有了这方面的知识，你就能更好地理解这些抽象类型到底是如何工作的，并且帮助你理解 C++ 语言的行为机制。

写到这里，其实有一个引人注目的原因促使你需要学习 C++ 的内存机制。在第 5 章，你已经了解到使用 C++ 中不可思议的集合类可以使得编程更加容易。在后续的各章中，你的首要目标就是理解如何高效地实现这些结构。在评估各种不同算法的效率时，你需要明白每一种算法的选择代价。如果不深入了解 C++ 语言用来实现这些算法的底层结构（最值得注意的指针和数组），那么也无法对代价进行评估。

11.1 内存结构

在你理解 C++ 内存模型的细节之前，需要知道信息是如何被存放在计算机中的。每一台现代计算机都拥有一些作为信息主要存储库的高速内存。在一台典型的机器中，由特殊的集成电路芯片构成的内存被称作 RAM，它代表随机存取存储器（random-access memory）。RAM 允许程序在任何时候可以访问任意内存单元。对于大多数程序员，了解 RAM 芯片工作的技术细节并不重要。重要的是如何组织管理内存。

11.1.1 位、字节和字

在计算机中，所有的数值（不管它多么复杂）都以信息的基本单元的组合进行存储，信息的基本单元是位（bit）。每一个位只能取两种可能的状态中的一种。如果你把机器中的电流想象成小电灯的开关，那么就可以把这两个状态称为开和关。如果你把每一个位想象成一个布尔值，则可用 `true` 和 `false` 来表示。然而，因为单词“bit”最初来源于“binary digit”的缩写，所以经常用 0 和 1 来标记这两种状态，0 和 1 这两个数就是计算机运算中基于二进制系统所使用的数字。

由于一个位所能存储的信息太少，因此，用几个位来存储数据并不是最合适的机制。为了更易于将这些传统类型的信息存储为数字或字符，将单个的位连接组合起来组成更大的
474 可整体对待的存储单元。这种最小组合单元称为一个字节（byte），它由 8 个位构成，并且足以存储一个 `char` 类型的数据。在大多数机器中，字节被集成为更大的称为字（word）的结构，其中，一个字通常可以存储一个 `int` 类型的数据。今天，大多数机器要么使用四字节的字，要么使用更长的八字节的字（32 位或 64 位）。

对于一台特定的计算机，其可用内存量可在很大的范围内变化。早期的机器只提供 KB

（kilobytes）级别的内存，二十世纪八九十年代开始有 MB（megabytes）级别的内存，到今天大多数机器都具有 GB（gigabytes）级别的内存。在大多数自然科学学科中，前缀 kilo、mega 和 giga 分别代表一千、一百万和十亿。然而，在计算机世界中，这些基于 10 的数值并不适合于机器的内存结构表示。因此，按照传统惯例，这些前缀被用来表示 2 的多次幂得到的与传统解释相近的值。因此，在编程中，前缀 kilo、mega 和 giga 分别代表如下意义：

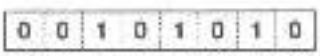
$$\text{kilo (K)} = 2^{10} = 1\,024$$
$$\text{mega (M)} = 2^{20} = 1\,048\,576$$
$$\text{giga (G)} = 2^{30} = 1\,073\,741\,824$$

20 世纪 70 年代早期一台 64KB 的计算机的内存为 64×1024，即 65 536 字节的内存。同样，一台现代 4GB 机器的内存为 4×1 037 741 824，即 4 294 967 296 字节的内存。

11.1.2 二进制和十六进制表示

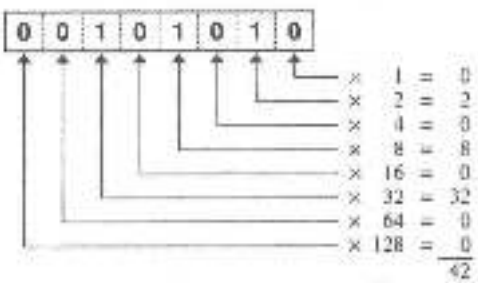
一台计算机中的每一个字节存储的数据的意义取决于系统如何解释这些单个位中的数。根据计算机的硬件指令，一个特定的位序列可以表示一个整数、一个字符，或者一个浮点数，它们要求具有不同的编码模式。无符号整数的编码模式最简单。一个无符号数中的各位可用**二进制计数法**（binary notation）表示，它的合法取值只有 0 和 1，正如计算机底层位中数值的真实表示形式一样。二进制表示在结构上与我们最熟悉的十进制表示类似，但是，二进制表示是以 2 为基数而不是以 10 为基数的。二进制数的最大好处是其值仅取决于它在这个数中的位置。最右边的一个位表示一个单位字段，其他位置上数值的大小总是其右边数值的两倍。

例如，让我们考虑一个包含以下位二进制数的字节：



475

这个位序列代表数值 42，你可以将每一位的值计算和加进行验证，如下图所示：

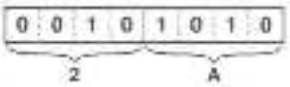


此图说明了如何将一个整数用二进制数表示出来，但是它也帮助我们例证了这样一个事实，即用二进制表示数并不方便。二进制表示显得很笨拙，更多的是因为它们往往很长。十进制表示更直观而且被熟知，但是却很难让我们理解一个数的计算机底层位的表示形式。

为了在应用程序中更好地理解如何将一个数用二进制表示，而不用纠结于过长甚至跨页的二进制数，计算机科学家往往采用十六（hexadecimal）（基数 16）进制表示。十六进制计数法用十六个数表示，从 0 到 15。十进制数字 0 到 9 已经足以代表前十个数字，但是传统的算术并没有为额外的其他六个数字定义符号。计算机科学因此采用字母 A 到 F 来表示它们。这些字母分别代表的数值如下：

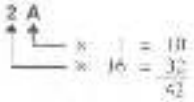
A = 10
B = 11
C = 12
D = 13
E = 14
F = 15

十六进制计数法之所以如此吸引人，就在于你可以立即在十六进制数和二进制数之间进行转换。你所要做的就是把所有的二进制数的四个位数划为一组。例如，数 42 可以如下图所示从二进制转换到十六进制：



476

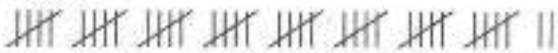
前四位代表数字 2，后四位代表数字 10。这两个数字按照十六进制格式转化成对应的十六进制数得到 2A。接着你可以以下述方法通过数值的相加来验证 2A 的值仍是 42：



为了保证可读性，本书大多数的数据都采用十进制表示。如果不能从上下文中得知数的基数，可根据本书所遵循的采用下标来代表基数这一惯用策略来推导出其基数。因此，对于数 42，它的三种最常见表示（十进制、二进制和十六进制）形式如下所示：

$$42_{10} = 00101010_2 = 2A_{16}$$

一个数无论采用哪种进制表示，关键在于数值本身不变，数的基数只影响其表示形式。42 有一个与基数无关的真实解释。这个真实解释可能是小学生使用的最简单的表示，毕竟也只是书写数字的另一种方式：



这一行表示的就是数字 42。事实上，一个数字写成二进制、十进制或者其他进制仅仅是其表示形式不同，与数字本身无关。

11.1.3 表示其他数据类型

在很多方面，现代计算的基本理念就是任何数值都可以用位的集合来表示。例如，很容易看出如何仅用一位来表示一个布尔值。你所要做的就是给每一个位的状态分配两个布尔值中的一个。一般地，0 代表 false，1 代表 true。正如最后一节所确展示的：你可以用一连串的位来存储一个二进制的无符号数，因此，八位二进制序列 00101010 表示数字 42。采用八位，可以表示 0 到 2^8-1，即 255 个数。16 位可以表示 0 到 $2^{16}-1$，即 65 535 个数。32 位足以表示 0 到 $2^{32}-1$，即 4 294 967 295 个数。

477

事实上，内存的每个字节可以存储 0 到 255 中的任何一个数值，这意味着这个字节足以存储 ASCII 字符。追溯到 C++ 的前趋语言，由于历史的原因，C++ 定义 char 类型为一个字节长度。这个设计决策使得 C++ 程序在处理那些需要采用扩展的字符集编码但又要与 ASCII 字符模型相兼容的语言时，显得更为艰难。因此，C++ 标准库定义了 wchar_t 类型来表示“宽字符”以扩展 ASCII 编码范围。然而，介绍这些机制已超出了本书的范围。

通过编码上一个很小的改变就能使带符号整数以位序列进行存储。主要因为这样做简化了硬件设计，大多数计算机使用被称为 2 进制补码运算（two's complement arithmetic）来表

示带符号整数。如果你想用 2 的补码运算表示一个非负数，只需要简单地使用传统的二进制表示即可。但是如果想表示一个负数，就要用 2^N 减去该负数的绝对值，N 表示所需的位个数。例如，32 位的 -1 的补码表示就可以以如下减法运算求得：

```
 100000000000000000000000000000000
-000000000000000000000000000000001
 111111111111111111111111111111111
```

在 C++ 中，浮点数也可以以固定长度的位序列来表示。尽管表示浮点数的细节已经超出了本书所讨论的范围，但是不难想象硬件开发将会使用一个字的一些位的子集来表示浮点值，用另一些位子集表示该数的指数。重要的是，需要牢记每一个数在计算机内部都是简单地以位的形式进行存储的。

在 C++ 中，不同的数据类型需要不同大小的内存。对于基本类型，以下存储值是很典型的（尽管 C++ 标准为编译器的编写者提供了灵活的机制来根据特定的硬件类型选择不同的大小）：

char	1字节（根据定义）
bool	1字节
short	2字节
int	4字节
float	4字节
long	8字节
double	8字节
long double	16字节

在 C++ 中，一个对象的大小往往是它所包含的实例变量所占存储空间大小的总和。例如，如果你定义了如第 6 章所示的 Point 类，它的私有部分包含以下实例变量： 478

```
int x;
int y;
```

上述的每一个实例变量都需要四个字节，因此在大多数机器上存储这个对象的数据总共需要八个字节的内存。然而，编译器允许给一个对象的内部表示增加内存空间，这样做允许编译器产生更高效的机器码。因此，Point 对象所占内存的大小至少为 8 字节，使之足以存储变量 x 和 y，也可能更大一些。

在一个 C++ 程序中，你可以用 sizeof 操作符来求一个变量在特定的平台上会被分配多少字节的内存。sizeof 操作需要一个操作数，该操作数要么是一对括号内的类型名，要么是一个表达式。如果操作数是一个类型名，sizeof 操作将返回该类型所分配的字节数。如果操作数是一个表达式，sizeof 返回的是存储表达式的值所需要的字节数。例如，表达式

```
sizeof(int)
```

返回需要存储一个 int 类型的值所需的字节数。而表达式

```
sizeof x
```

返回需要存储变量 x 的字节数。

11.1.4 内存地址

在一个典型的计算机内存系统中，每个字节都由一个数字地址（address）所标识。计算机的第一个字节编址为 0，第二个是 1，以此类推，直到机器可表示的最大字节数减 1。例

如，一个内存为64KB的小型计算机的内存地址从字节0开始到字节65 535结束。然而，这些数都以十进制表示，这并不是大多数程序员想要的地址。给定地址与硬件的内存结构密不可分，一般会想到内存地址如果用上一小节介绍的十六进制表示法，应该以0000开始编址到FFFF结束。然而，重要的是需牢记地址仅仅是一些简单的数，它的基数仅决定这些数如何被表示出来。

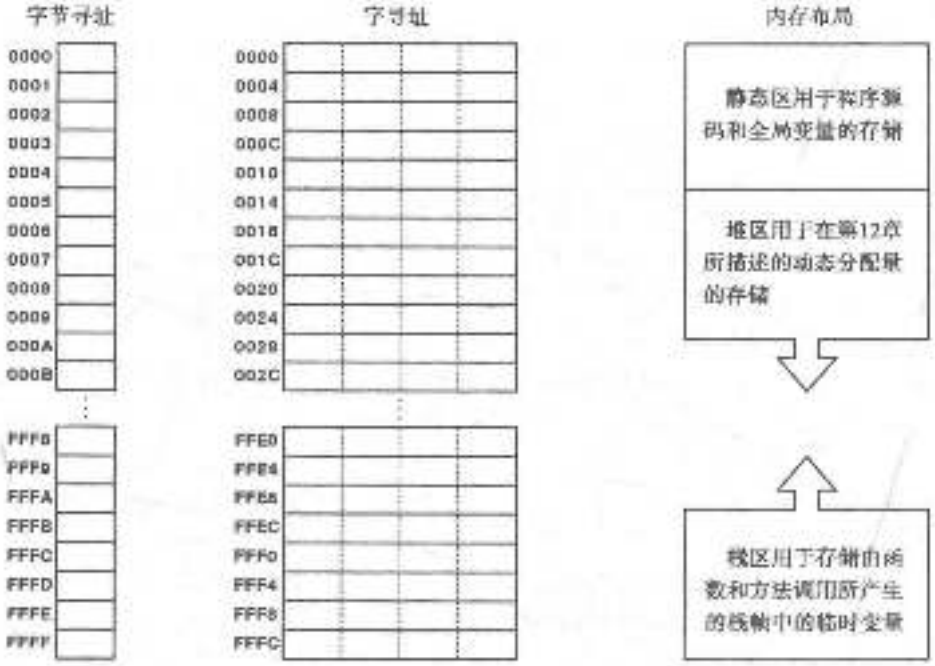
尽管地址也可以被表示成十进制，但是本书使用十六进制表示地址，具体原因如下：

- 地址传统的表示法就是十六进制，C++的调试器以及运行环境往往都以此形式表示地址。

479

- 采用sans-serif字体的十六进制形式表示地址，能更容易识别出一个特殊的数代表的是一个地址而不是某些身份不明的整数。在本文中，如果你看到数字65 536，你可以假定它代表一个整数。如果你看到数FFFF，你可以很自信地确定它代表一个地址。
- 采用十六进制表示法能更容易看出为什么会有一些特殊的约束。如果你把它写成十进制数，数65 535更像是一个随机数。如果你把同样的数用十六进制FFFF表示，很容易识别这个数是能用16位表示的数的最大值。

尽管内存地址往往以字节为单位，但大多数计算机也支持更大的如字的操作单元。在一台典型的机器中，一个字有四个字节，因此可以用四个字节一组来表示一个字。然而，那样的话，连续的字的地址以四递增。字节和字编址的区别如图11-1所示。



480

图11-1 C++程序典型的内存布局

图11-1的右边给出了在一个典型的C++程序中如何组织内存的框架。程序中的指令（在底层都是按位存储的）和全局变量往往被存储在静态区（static area），该区域位于地址编址号较小的接近机器地址空间的开始处。该区域所分配的内存量在程序运行期间不会发生改变。

内存中的最高地址区表示栈区（stack area）。当你的程序每调用一个函数或者方法，计算机都会在这个内存区创建一个新的栈帧。当函数返回时，所创建的栈帧会被撤销，以为后续的函数调用所需的栈帧释放内存。下节将对栈帧的结构进行更为详细的描述。

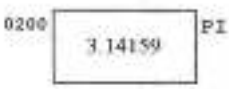
处于栈区和静态区之间的内存区域被称为堆区（heap area）。该区域会在程序运行时请求更多内存的时候发挥作用。该技术在抽象数据类型的设计以及实现中是非常重要的，具体描述请见第 12 章的内容。

11.1.5 为变量分配内存

当你在一个 C++ 程序中声明一个变量时，编译器必须保证给声明的变量分配足够的内存来存储该类型变量的值。所分配的内存大小取决于变量是如何被声明的。本书唯一用到的一种全局变量是常量（它一般都被分配在内存中的同一区域），该区域在今天大多数机器架构下处于地址相对较小的内存区。因此，如果编译器看到以下声明：

```
const double PI = 3.14159;
```

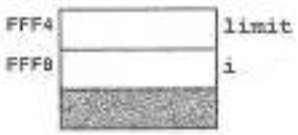
则编译器会在低地址区域分配八个字节内存，并将 3.141 59 存储到常量 PI 中。作为一名程序员，你不知道编译器会选择什么内存地址，但是如果你构造一个地址并以图示之，它会帮助你呈现机器内部发生了什么。这里给出一个示例，你可以想象常量 PI 被存储在地址 0200 中，如下图所示：



然而，大多数变量都是局部变量。局部变量被分配在内存高端处的称之为栈帧（stack frame）连续的地址块中。从第 2 章开始你已经了解了栈帧，但这些框架当时被抽象描述为盒子。在底层，这些变量被分配一块内存空间，并在每次函数调用时被压入栈顶。 [481]

为了使这个表述更为具体，让我们追溯第 1 章中的程序 PowersOfTwo 的运行过程来看一下栈中到底发生了什么。这样你也不用一直回溯前面的图了（忽略程序中的注释和函数原型声明行），该程序的代码再次输出，如图 11-2 所示。

当你运行程序 PowersOfTwo 时，操作系统首先会产生一个向 main 函数的调用。该 main 函数没有参数，但有两个局部变量，limit 和 i。因此，栈帧必须为这两个整型局部变量分配内存，如下图所示：

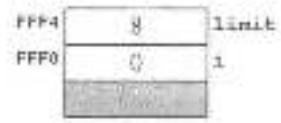


在该图中，变量 limit 被分配到地址 FFF4，变量 i 被分配到地址 FFF8。这些地址都是随机分配的，因此不可能准确预测编译器将会分配哪些地址或者先给哪个变量分配内存。你能指望的就是这些变量都会被分配到指定为栈帧的区域。图中底部灰色的矩形表明计算机需要追踪除了每一个局部变量之外的函数调用的附加信息。如果没有什么信息可追踪，每一个栈帧也需要追踪程序返回的位置。信息的格式取决于机器的结构，而且对于理解数据类型并不是至关重要的。本书的栈图在每一个栈帧中都包含一个灰色的矩形来提醒你额外的信息是存在的，并且可视化地理解每个栈帧的扩展会更容易。

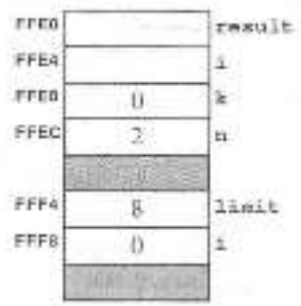
```
/*
 * File: PowersOfTwo.cpp
 * ---------------------
 * This figure contains only the function definitions from the
 * PowersOfTwo program from Chapter 1.
 */

int main() {
   int limit;
   cout << "This program lists powers of two." << endl;
   cout << "Enter exponent limit: ";
   cin >> limit;
   for (int i = 0; i <= limit; i++) {
      cout << "2 to the " << i << " = "
           << raiseToPower(2, i) << endl;
   }
   return 0;
}

int raiseToPower(int n, int k) {
   int result = 1;
   for (int i = 0; i < k; i++) {
      result *= n;
   }
   return result;
}
```

图 11-2 PowersOfTwo 程序代码

当该程序运行时，每个函数都访问它自己的栈帧，并且在栈帧变化时更新局部变量的值。假设用户输入 8 作为 limit 的值，那么在第一次调用 raiseToPower 之前的那一刻，其栈帧中的内容如下图所示：



调用 raiseToPower(2,i) 时，会在已存在的栈帧顶部创建一个新的栈帧。该栈帧中包含形参变量 n 和 k 的值，以及局部变量 result 和 i。形参变量被初始化为实参的值，这意味着现在的栈如下图所示：



482
|
483

当函数 raiseToPower 返回时，它的栈帧会被撤销，返回到它被调用之前的状态。

这个例子很简单，而且没有涉及创建更多精确内存图的复杂度问题。然而，这个例子确实足以使你掌握了为理解将在下节介绍的指针主题中所涉及的如何为变量分配内存所需的基本知识。在第 12 章，你将有机会返回到内存图，并且学到更多有关内存分配的策略。

11.2 指针

C++ 的一个设计原则是：程序员应该尽可能多地访问到由底层硬件提供的机制。因此，

C++ 语言使得内存位置的地址对程序员可见。一个数据项的值是内存中的一个地址，该数据项被称为一个指针（pointer）。在许多高级编程语言中，指针用得很保守，因为这些语言提供了其他机制来限制对于指针的需求。例如，Java 编程语言对程序员隐藏了所有指针。在 C++ 中，指针使用得很普遍，而且如果不了解指针如何工作就不可能理解大多数专业的 C++ 程序。

在 C++ 中，指针有很多用途，下面几条是其中最重要的：

- 指针允许以一种压缩的方式引用一个大的数据结构。一个程序中的数据结构可以变得任意大。然而，无论它多大，数据结构始终会在计算机内存中占有一席之地，并因此获得一个地址。指针允许你使用这个地址作为那个数据结构所表示的值的一个缩写。由于一个内存地址一般占用四个字节的存储空间，因此，数据结构本身很大时，这个策略节省了相当多的内存空间。
- 指针使得在程序运行时能够预订新的内存。到目前为止，程序中唯一能用的内存是分配给你明确声明的变量的内存。在许多应用程序中，程序运行时能请求新的内存以及用指针指向这些内存都是很方便的。这个策略将会在 12.1 节中讨论。
- 指针可以用来记录数据项之间的关系。在高级的程序应用中，指针被广泛地用于建模单个数据值之间的关联。例如，程序员通常通过在第一个数据项的内部表示中用一个指针指向第二个数据项来指明一个数据项跟随另外一个的概念顺序。使用指针来创建各个组件之间联系的数据结构被称为链接结构（linked structure）。链接结构在实现许多抽象数据类型时扮演着重要角色，你会在本书的后半部分遇到这些抽象数据类型。 484

11.2.1 把地址当作数值

在 C++ 语言中，任何引用内存中能够存储数据的内存单元的表达式被称为左值（lvalue）。之所以被称为左值，是因为这些标识符都出现在 C++ 赋值语句的左边。例如，简单的变量就是一个左值，因为你可以编写如下这条语句：

```
x = 1.0;
```

然而 C++ 中的许多值都不是左值。例如常量就不是左值，因为常量不能被改变。同样，尽管数学表达式的结果是一个值，但它也不是左值，因为不能给一个表达式的结果赋一个新值。

下面是 C++ 中左值的一些属性：

- 任何一个左值都存储在内存中，所以都有一个地址。
- 一旦声明了一个左值，它的地址将不会改变，尽管地址中存储的内容会发生改变。
- 一个左值的地址是一个指针值，它可以被存储在内存中，并且可以和数据一样被操作。

11.2.2 声明指针变量

就像 C++ 中的其他变量一样，在使用指针变量之前对它必须进行声明。为了声明一个变量为指针变量，你所需要做的就是在声明的变量名前加一个星号（*）即可。例如，下行代码：

```
int *p;
```

声明 p 是一个指向 int 型的指针变量。类似地，下行代码：

```
char *cptr;
```

声明 cptr 为指向 char 类型变量的指针。这两种类型（指向 int 类型的指针和指向 char 类型的指针）在 C++ 中是不同的，尽管它们在内部都表示为地址。为了使用指针地址中的数据，编译器需要知道如何解释它，因此要求必须明确地指明指针所指对象的类型。指针所指对象的类型被称为指针的**基类型**（base type）。因此，指向 int 的指针类型是以 int 作为其基类型的。

485 值得注意的是，星号用来指明一个变量是一个指针变量，在语法上它属于变量名，但具有基类型。因此，当你在同一个声明语句中声明两个指向相同类型的指针时，需要给每个变量标记一个星号，如下所示：

```
int *p1, *p2;
```

而以下声明：

```
int *p1, p2;
```

则声明 p1 为指向一个整数的指针，而 p2 被声明为一个整型变量。

11.2.3 基本的指针运算

C++ 定义了两个操作符，允许你在指针和目标数据之间进行运算：

```
&          取地址
*          取指针所指对象的值
```

& 操作符取一个左值作为操作数，返回左值所在的内存地址。* 操作符可以是任何类型的指针变量，并返回指针变量所指对象的左值。这种运算被称作**解析引用**（dereferencing）。* 操作返回一个左值，这意味着可以给解析引用的指针赋值。

说明这些操作符最简单的方式是举例说明，考虑如下声明：

```
int x, y;
int *p1, *p2;
```

这些声明语句共分配了四个字的内存，两个是 int 类型，两个是指向 int 类型的指针类型。具体而言，让我们假设下面这些值存储在栈中，其机内地址如下图所示：



给出这些声明，你就可以像以往一样使用赋值语句为 x 和 y 赋值。例如，执行下面的赋值语句：

```
x = 42;
y = 163;
```

486

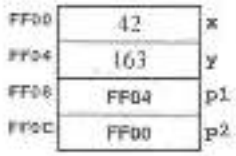
得到以下内存状态：



为了初始化指针变量 p1 和 p2，你需要用代表某个整型对象地址的值赋给 p1 或 p2。在 C++ 语言中，产生地址的操作符是 &，你可以利用赋值和 & 操作符使 p1 指向 y，p2 指向 x：

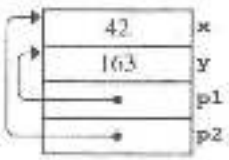
```
p1 = &y;
p2 = &x;
```

这些赋值语句执行之后内存状态如下图所示：



p1 的值为 FF04，它是变量 y 的地址。类似地，p2 的值为 FF00，它是变量 x 的地址。

使用明确的地址来表示指针强调了这一事实：地址在内部以数字进行存储。然而，它并没有使你直观地认识到指针的意义。为了实现这一目标，最好使用箭头来表明每个指针的所指对象。如果你除去了整个地址值，并且用箭头表示指针，则如下图所示：



图中使用箭头使我们很明显地看出变量 p1 和 p2 指向了其箭头所指单元。箭头使得更容易理解指针是如何工作的，因此在本书中大多数都使用内存图来表示指针。同时，要牢记指针只是简单的数字地址，它在机器内部没有箭头。 487

为了从指针中取出它所指的对象值，可以使用 * 操作符。例如，以下表达式：

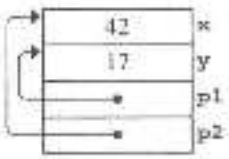
```
*p1
```

表明取出 p1 所指的内存单元的数据。而且，由于 p1 声明指针为一个整数，所以编译器知道 *p1 表达式必然指向一个整数类型的对象。因此，假设内存中布局和图示一样，那么 *p1 就是变量 y 的一个别名。

就像简单变量 y 一样，表达式 *p1 也是一个左值，可以给它赋值。执行以下的赋值语句：

```
*p1 = 17;
```

会改变变量 y 的值，因为指针变量 p1 指向 y。这个赋值操作执行后，内存格局如下：

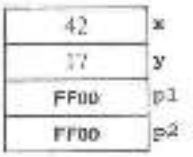


可以看出，变量 p1 的值本身没有因为赋值语句而改变，p1 始终指向变量 y。

还可以给指针变量自身赋新值。例如，语句

```
p1 = p2;
```

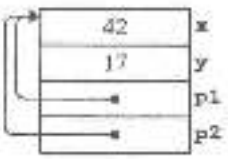
告诉计算机把变量 p2 的值复制到变量 p1 中。变量 p2 中的值是一个指针值 FF0C。如果把该值复制到变量 p1 中，那么两个指针变量的值就相同，如下图所示：



只要你牢记指针的底层表示是一个整数，那么复制指针值就一点也不显得神秘了。它仅
488 将指针值复制到目的地。如果你用箭头画出内存图，必须牢记复制一个指针代替目标指针，只需要用一个新的箭头和原来的指针指向相同的位置。因此，以下赋值语句

```
p1 = p2;
```

的效果就是改变从 p1 引出的箭头，以使它也指向来自 p2 的箭头所指向的相同的内存地址，如下图所示：



区分指针赋值和数值赋值很重要。**指针赋值**（pointer assignment），例如

```
p1 = p2;
```

使得 p1 和 p2 指向相同的地址。相反地，**数值赋值**（value assignment），如以下语句：

```
*p1 = *p2;
```

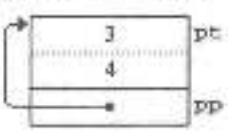
是将 p2 所指向的地址中的数据复制到 p1 所指向的地址中。

11.2.4 指向结构和对象的指针

上节中的例子仅声明了指向一些基本类型的指针。在 C++ 中，对结构体和对象使用指针是很常见的事。例如，下列声明：

```
Point pt(3, 4);
Point *pp = &pt;
```

声明了两个局部变量。变量 pt 是一个存有坐标值 3 和 4 的 Point 类型对象。变量 pp 指向
489 pt。使用基于指针的格式，得到的内存图如下图所示：



基于指针 pp，你可以使用 * 操作符得到它所指对象的内容，因此，*pp 和 pt 的效果相同。

然而，如果你将一个指针指向一个对象，并且需要索引到该对象的区域和方法时，你确

实需要练一些细心。例如，你不能用指针进行如下调用：

```
*pp.getX()
```



尽管代码看起来没有问题，但这个表达式中有一个优先级问题。在C++语言中，点操作符的优先级比星操作符高，这就意味着编译器想要如下所示解释该表达式

```
*(pp.getX())
```

这显然没有意义。你想让它做的是先将指针解析，然后再调用方法，这就意味着表达式应该加上如下所示的括号：

```
(*pp).getX()
```

该表达式得到了预期值，但每次这样使用会显得非常烦琐。当你编写一些更复杂的应用时，会发现自己总是在使用指向对象的指针。迫使程序员在每一次操作时加上这些括号会使得指针指向对象相当不方便。为了消除这种不便，C++语言定义了操作符 ->，它是由解析与选择操作符组合而形成的一个单一的操作符。因此，采用指针 pp 来调用 getX 方法的惯用语法为：

```
pp->getX()
```

11.2.5 关键字 this

当你实现一个类时，C++语言定义关键字 this 作为指向当前对象的指针。这种定义有若干重要的应用，你将会在后面的示例中发现。其中，最常见的一种就是你可以使用关键字 this 来选择一个对象的实例变量，尽管它们的名字都被形参变量或者局部变量隐藏了。 [490]

隐藏问题在第6章讲 Point 类的构造函数时曾介绍过，当时的构造函数是这样的：

```
Point(int cx, int cy) {
   x = cx;
   y = cy;
}
```

该构造函数的参数必须命名为除 x 和 y 之外的其他名称，以避免与实例变量产生命名冲突。然而，用户很可能觉得新命名有一点难懂。从用户的角度看，x 和 y 才是构造函数最恰当的命名。但是从实现者的角度来看，x 和 y 却是实例变量的完美命名。

只有使用关键字 this 来选择实例变量才能使用户和实现者同时满意，如下所示：

```
Point(int x, int y) {
   this->x = x;
   this->y = y;
}
```

一些程序员建议：使用 this 引用当前对象的成员使得代码更易于阅读。JavaScript 语言到目前为止对于所有的引用均使用关键字 this。然而本书遵循 C++ 语言传统，只会在解决二义性的时候使用 this。

11.2.6 特殊的指针 NULL

在许多指针应用中，有一个特殊指针，它并不指向任何实际内存地址，至少目前不指

向。这个特殊的值被称作空指针（null pointer），并且在内部表示为地址值 0。在 C++ 语言中，表示一个空指针最好的方式就是使用常量 NULL，该常量已在 <cstddef> 接口中定义。

用 * 操作符取一个空指针是不合法的。今天使用的很多流行的编程环境一般都会识别出该错误并中止程序的运行，但是这种情况不是总会发生。在一些机器中试图读取一个空指针所指向的目标数值时，会返回存储在地址 0000 中的数据。这种情况也适用于未初始化的指针。如果你声明但未初始化的一个指针变量，计算机会把指针的数据解析为一个地址值，并
491 且尝试读取那块内存空间的数值。此时，程序就非常容易崩溃。

空指针的使用会在本书的具体应用中介绍，现在只需知道存在这个常量即可。

11.2.7 指针和引用调用

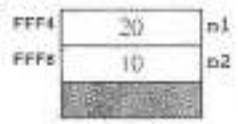
C++ 内部通过使用指针来实现引用调用。当一个参数通过引用传递时，栈帧会在调用时存储一个指针指向该值的内存单元。该值的任何改变都会影响指针的目标数据。这也意味着这些改变直到函数返回一直有效。

图 11-3 的程序提供了一个简单的示例来阐明 C++ 如何实现引用调用。程序从用户端读取两个整数并且验证它们是否依照递增顺序排序。如果没有，程序会调用函数 swap 来交换它们的值。

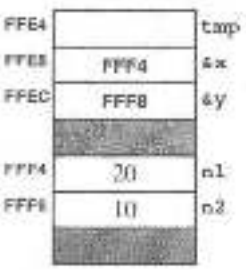
假设你运行了这个程序，并且输入了 20 和 10，如下图所示：



这两个值没有递增排序，所以主函数将会调用 swap。调用之前栈中的数据表示如下：

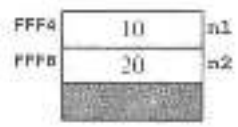


函数 swap 取引用作参数，这就意味着 swap 的栈帧给出的是地址值而不是数值。调用之后，栈中的数据如下：



492

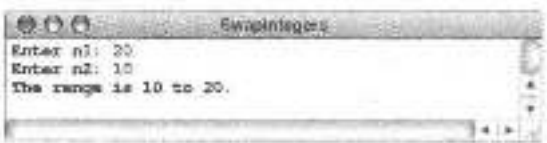
swap 的所有指向 x 和 y 的引用被传递给变量 n1 和 n2，作为指针的目标数据。交换这两个值意味着在 swap 调用返回之后函数产生的效果还在延续，如下图所示：



```
/*
 * File: SwapIntegers.cpp
 * ----------------------
 * This program illustrates the use of call by reference to exchange
 * the values of two integers.
 */

#include <iostream>
#include "simpio.h"
using namespace std;

/* Function prototype */

void swap(int & x, int & y);

/* Main program */

int main() {
   int n1 = getInteger("Enter n1: ");
   int n2 = getInteger("Enter n2: ");
   if (n1 > n2) swap(n1, n2);
   cout << "The range is " << n1 << " to " << n2 << "." << endl;
   return 0;
}

/*
 * Function: swap
 * Usage: swap(x, y);
 * ------------------
 * Exchanges the values of x and y.  The arguments are passed by
 * reference and can therefore be modified.
 */

void swap(int & x, int & y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```

图 11-3 确保两个整数是顺序的程序 493

程序会继续使用更新了的数据，这就得到了下面的输出：



尽管引用调用十分方便，但这并不是 C++ 语言一个至关重要的特性。你可以通过明确地调用指针来替代引用调用的效果。在该程序中，你所要做的就是改变 swap 的实现如下：

```
void swap(int *px, int *py) {
   int tmp = *px;
   *px = *py;
   *py = tmp;
}
```

在主程序中的调用变为：

```
swap(&n1, &n2);
```

在习题中，你会有机会练习将使用引用调用的程序转换成基于指针的等价程序。

11.3 数组

当 Vector 类在第 5 章第一次出现时，那一小节介绍了矢量和数组，注意你很可能已经从前面的编程经验中有了一些数组的概念。C++ 语言提供了一个内置的数组类型，它基于从 C 语言继承而来的语言模型。尽管在很多应用程序中还是能看到数组，但考虑到已有的 Vector 集合类更加灵活方便，所以在新的代码中没什么理由再使用数组了。

在 C++ 语言中，数组（array）是一个较低级的多个数据值的集合，它具有以下两个显著特性：

1. 数组是有序的。你肯定能够按照数组元素的索引值依次地读写它们：第一个、第二个，以此类推。
2. 数组是同质的。数组中的每个元素值必须是同类型的。因此，你可以定义一个整型
494 数组或一个浮点型数组，但是数组不允许两种类型的元素混合存在。

既然你已经熟悉了矢量类，很容易想到将数组作为实现矢量思想的原始实现。正如矢量一样，数组包含一些由整型索引选择的基类型的单个元素。画出一个表示矢量的图同样也适用于数组。虽然语义上有些不同，但却易于掌握。真正的不同在于数组的以下约束使得在实际应用中矢量显得更好用：

- 数组被分配为固定大小的内存，之后不能再改变。
- 尽管数组有固定大小内存，C++ 语言并没有使得其容量对程序员可用。结果是处理数组的程序需要一个额外的变量来追踪元素的个数。
- 数组不支持插入和删除元素。
- C++ 没有提供边界检查来确保你选择的元素存在于数组中。

尽管有这些明显的缺点，数组仍然可以在其上建立更有威力的集合类的框架。为了理解这些类的实现，你需要熟悉数组的实现机制。

11.3.1 声明数组

就像 C++ 语言中的其他变量一样，数组在使用之前必须声明。声明数组的一般形式为：

```
type name[size];
```

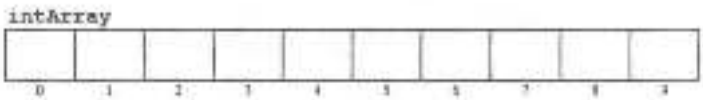
其中，type 表示数组中元素的类型，name 是数组的名称，size 是一个表示数组元素个数的常量。例如以下声明：

```
int intArray[10];
```

声明了一个其元素类型为整型，名为 intArray 的数组，它包含了 10 个元素。然而，在多数情况下，你应该使用符号常量而不是确定的整数值来指定数组的大小，这样可方便地对数组大小进行修改。因此，可以用一个更加传统的方法声明：

```
const int N_ELEMENTS = 10;
int intArray[N_ELEMENTS];
```

495 你可以用图示化的方法来表示这个表明：



和一个矢量中的元素一样，一个数组的索引值从 0 开始到数组容量减一为止。因此，一个有 10 个元素的数组，其索引值为 0、1、2、3、4、5、6、7、8 和 9。

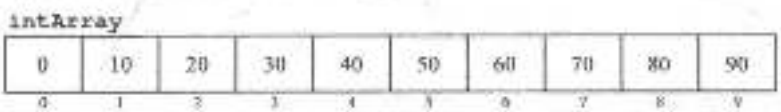
11.3.2 数组元素的选择

为了引用数组中的某个元素，需要给出数组名和数组中相应元素的索引位置。这种在数组中识别特定元素的过程被称为**选择**（selection）。在 C++ 语言中，通过数组名加上方括号内的元素索引值表示和你在矢量中选择一个元素一样。

选择表达式的结果为一个左值，这就意味着你可以给它赋值。例如，如果你执行 for 循环：

```
for (int i = 0; i < N_ELEMENTS; i++) {
   intArray[i] = 10 * i;
}
```

数组 intArray 会被初始化为如下图所示的情况：



当你从数组中选择一个元素时，C++ 不进行数组边界的检查。如果索引越界，C++ 语言会试图在内存中寻找该索引所指向的内存单元的值并使用它，这会导致不可预期的结果。更糟的是，如果你给该元素赋一个新值，你可能会覆盖了程序其他部分的内存内容。数组越界是黑客攻击计算机系统的一个致命点。

11.3.3 数组的静态初始化

数组可以在声明时赋初值。在这种情况下，等号后面是由一对花括号括起来的一行初始值。例如，以下声明：

```
const int DIGITS[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

496

声明了一个常量数组 DIGITS，其中 10 个元素分别以下标数进行初始化。可以从该例子中看出，明确初始化允许你忽略数组容量，但可以从初始化的数值的数目中推断得出。

在 DIGITS 一例中，你知道数组中有 10 个数。然而，在许多情况下，程序需要在一个静态声明的数组中判断元素数目，使得程序员不用在每一次程序改变时再依次数元素数目。例如，想象你在编写一个程序，该程序要求逐个包含美国所有人口超过 1 000 000 的城市名。从 2010 年的人口普查中获取数据，你可以通过以下方式声明和初始化 BIG_CITIES 为一个全局常量数组：

```
const string BIG_CITIES[] = {
   "New York",
   "Los Angeles",
   "Chicago",
   "Houston",
   "Philadelphia",
   "Phoenix",
   "San Antonio",
   "San Diego",
   "Dallas",
};
```

但是，这个清单数据是随时间变化的。在 1990 年和 2000 年的人口调查结果中，底特律从清单中消失，而菲尼克斯和圣安东尼奥加入到该清单。若取来自于 2020 年的人口普查结果的话，圣何塞很有可能会加进来。如果你负责维护包含这段代码的程序，永远不要想着清单中有多少个城市，也不要想着程序能判断有多少个元素。反之，你可能需要更新清单中的城市，并且使编译器了解到其数量。

幸运的是，C++ 语言提供了一个标准的习语来确定静态声明得到的数组的容量，然后根据其设置元素数目。给定一个静态初始化的数组 MY_ARRAY，MY_ARRAY 的元素个数可以这样计算出来：

```
sizeof MY_ARRAY / sizeof MY_ARRAY[0]
```

该表达式获取整个数组的容量除以数组中每个元素大小。因为数组的同质性，运算结果就是
497 数组元素的数目，与元素类型无关。因此你可以如下初始化一个变量 N_BIG_CITIES 来存储 bigCities 数组的城市数目：

```
const int N_BIG_CITIES = sizeof BIG_CITIES /
                         sizeof BIG_CITIES[0];
```

11.3.4 有效容量和分配容量

尽管 sizeof 允许你确定静态数组的大小，但是当你编写代码时，有许多应用你没办法知道一个数组该多大，因为元素实际的数目取决于用户数据。解决选择一个合适的数组容量问题的策略是：声明一个比你需求大的数组，然后只使用其中的一部分。因此，定义一个常量表示数组元素数目有最大值并以此来声明数组，而不是定义一个可以存储元素实际数目的数组。在任何给定用途的程序中，元素的实际数目都要小于其边界。当你使用这个策略时，你需要一个单独的整型变量来跟踪实际用到的元素的数目。在声明中，指定数组容量的被称为**分配容量**（allocated size）。实际使用到的元素数目被称为**有效容量**（effective size）。

例如，假设你想定义一个数组来存储体操运动员的分数，裁判会在一个记分牌上给每一个选手打分，打分范围为 0.0 到 10.0，到 2005 年之前的奥林匹克运动会都采用这个打分规则。如果你想让程序最多有一百个裁判（尽管实际数目会更小），你可能需要这样定义这个数组：

```
const int MAX_JUDGES = 100;

double scores[MAX_JUDGES];
```

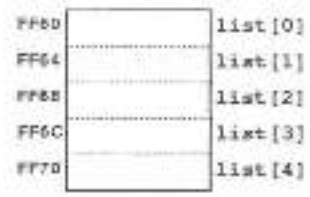
为了跟踪有效容量，你需要声明另一个称为 nJudges 的变量，并且保证它能够跟踪实际裁判的数量。

11.3.5 指针和数组的关系

在 C++ 中，数组名和指向其第一个元素的指针同义。其同一性最好用示例说明。声明

```
int list[5];
```

498 给数组分配了五个整型数据的内存，并且将内存存储到当前的线程中，如下图所示：



数组名 list 表示一个数组，但是同时被用作一个指针值。当它被用作一个指针时，list 被定义成数组中首元素的地址。因此，如果编译器遇到变量名 list 没带下标，它会将该数组名解释为一个指向数组开始内存的指针变量。

C++ 将数组视为指针的一个最重要的原因是数组形参和实参共享，尽管并没有涉及任何明确调用。例如，你可以如下实现一个基于数组的选择排序算法版本：

```
void sort(int array[], int n) {
   for (int lh = 0; lh < n; lh++) {
      int rh = lh;
      for (int i = lh + 1; i < n; i++) {
         if (array[i] < array[rh]) rh = i;
      }
      swap(array[lh], array[rh]);
   }
}
```

这个函数会将调用的数组排序，因为函数通过复制正在调用的参数地址来初始化数组参数。函数接着用该地址选择数组元素，这就意味着这些元素是正在调用参数数组的元素。

如果你将函数原型写成以下形式，sort 函数将会实现相同的功能：

```
void sort(int *array, int n)
```

在该函数中，第一个参数被声明为一个指向数组的指针，但其效果与原来的将其声明为一个数组的实现相同。在前一种情况下，在数组名 array 下的栈帧中存储的数值是正在调用的参数首元素的地址。在机器内部，这两种声明是等价的。不管在声明中使用哪种形式，都能对变量 array 进行相同的操作。

499

作为一个普遍规则，你需要用能反映它们用途的方式来声明参数。如果你打算将一个数组作为参数并且从中选择元素，那就将其参数声明为一个数组。如果你打算将一个指针或者引用作为参数，那就将其声明为指针。

在 C++ 中，指针和数组最关键的区别是：当变量被声明时就会进入角色，而不是在这些变量被作为参数传递时。声明

```
int array[5];
```

和

```
int *p;
```

基本的区别是内存分配。第一个声明会得到五个连续字的内存，它足以存储数组元素。第二个声明仅获得一个字的内存，它足以存储一个机器地址。将上述声明的区别牢记在心是非常重要的。如果你声明一个数组，你必须要有存储空间来处理它。如果你声明一个指针变量，除非你对它初始化，否则它就不能存储。

将一个指针初始化为数组的最简单的方式是：将存在的数组的首地址复制给该指针变量。例如，如果在前面进行了声明，那么你就要编写以下语句：

```
p = array;
```

它将变量 p 指向 array 的地址，之后你就可以交替使用这两个名字了。

将一个已存在的数组的首地址赋给指针有严格的限制。毕竟，如果你已经有一个数组名，你最好使用它，把它赋给一个指针并不是上策。将指针作为数组使用的真正优势是你可

以将一个指针初始化为之前未分配的新内存首址，以允许你在程序运行时创建一个新数组。该技术将在第 12 章进行讲述。

11.4 指针运算

在 C++ 语言中，指针可以使用操作符 + 和 -。这种计算和普通算术运算有相似之处，
500 但又不完全一样。这种对指针进行加减运算的运算被称为**指针运算**（pointer arithmetic）。

指针运算定义的规则很简单。如果 p 指向命名为 array 数组的首元素，k 是一个整数，以下等价总是成立的：

```
p + k        定义为        &array[k]
```

换句话说，如果你将一个指针值与一个整数 k 相加，其返回结果就是取索引 k 处数组元素的地址。

11.4.1 数组索引和内存地址

为了更好地理解指针运算如何实现，考虑一个涉及数组存储如何分配到内存例子。例如，假设一个函数包含如下声明：

```
double list[3] = { 1.61803, 2.71828, 3.14159 };
double *p = list;
```

这些变量都会在该函数的栈帧内分配空间。对于数组变量 list，编译器给数组中的三个元素分配内存，每一个都足以存储一个双精度浮点数。对指针 p 而言，编译器给指针分配足够的内存，用来表示类型为 double 的左值的地址。

在该示例中，也给出了数组的初始元素。数组 list 的元素被初始化为 1.618 03、2.718 28 和 3.141 59。声明

```
double *p = list;
```

初始化 p 来使它能够存储数组的起始地址。如果栈帧从 FFA0 单元开始，内存分配情况如下图所示：



在图中，p 当前指向数组 list 的初始地址。如果将整数 k 和指针 p 相加，返回结果为
501 k 位置索引所对应的地址。例如，如果一个程序包含以下表达式：

```
p + 2
```

该表达式的计算结果是一个新的指向包含 list[2] 的地址的指针。因此，在前面的图中，p 包含地址 FFA0，p+2 指向数组中第三个元素的地址，即 FFB0。

注意到指针加法运算和传统的加法运算并不完全等价，因为编译器必须考虑基类型的空间大小。在该例中，每一个加到指针的整数相当于 double 类型的大小，即 8 个字节。

C++ 编译器以同样的方式处理指针相减。如果 p 是一个指针变量，k 是一个整数，下列表达式：

```
p - k
```

计算出指针 p 指向的当前地址之前 k 个位置的数组元素的地址。因此，你如果要使用以下语句设置 p 指向 list[1] 的地址：

```
p = &list[1];
```

那么 p+1 和 p-1 分别对应的地址为 list[0] 和 list[2]。

算术运算 *、/ 和 % 在指针运算中没有意义，且其运算结果不能作为指针操作数使用。而且，运算 + 和 - 也是有约束的。在 C 语言中，指针可以加或减一个整数，但是不能进行指针相加，减是唯一可以进行指针相减运算的算术操作符。以下表达式：

```
p1 - p2
```

返回当前两个指向同一数组的指针 p1 和 p2 之间的数组元素的个数。例如，若 p1 指向 list[2]，p2 指向 list[0]，那么表达式

```
p1 - p2
```

的值为 2，因为在当前两个指针之间有两个元素。 502

11.4.2 指针的自增自减运算

了解了指针运算的规则，我们就能更好地理解 C++ 语言一个最为常见的语法结构，如以下表达式所示：

```
*p++
```

在该表达式中，* 操作符和 ++ 操作符均要对操作数 p 进行运算。但是 C++ 中的一元操作符的运算顺序是从右向左的，所以 ++ 先运算，接下来是 *，所以编译器就会这样解释这个表达式：

```
*(p++)
```

我们在第 1 章学过，后缀 ++ 操作符使 p 的值加一，然后返回 p 增加前的值。因为 p 是一个指针，所以自增运算满足指针运算规则。因此，p 值加 1 产生了指向数组中下一个元素的指针。如果 p 原先指向 arr[0]，那么自增操作以后会使其指向 arr[1]。因此，表达式

```
*p++
```

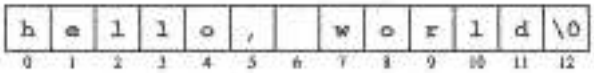
有如下的解释：

解析指针 p，并且返回一个当前指针所指的右值。因此，p 值自增运算的效果使得如果原先的左值是数组中的一个元素，那么 p 的新值就是指向那个元素的下一个元素的指针。

11.4.3 C 风格字符串

解释 *p++ 这一习惯用法的最佳方式是在一个处理 C 风格字符串代码的上下文中看它最有可能的使用方式。从第 3 章你已经知道，C++ 使用两种不同的字符串类型。到目前为止，最容易使用的就是 <string> 接口输出的类 string，该类定义了一个高层的操作集合以使得字符串操作相对容易。然而，由于历史的原因，C++ 支持一个更基本的，继承自 C 语

言的字符串模型。在C语言中，字符串就是一个以\0结尾的字符数组。\0被称为空字符（null character）。例如，如果你在一个C++程序中使用字符串常量"hello, world"，编译器将会在内存中产生一个字符数组，该数组包含字符串中的所有元素外加一个空字符，如下图所示：



503

如图所示，C风格的字符串并不是明确存储为数据结构的一部分，而是以一个空字符来表示结束。如果你需要知道C风格字符串的长度，需要从开始数到结尾。在标准C库中，这个操作可由strlen函数来完成，该函数支持许多不同的使用方式。例如，下面的实现声明了一个字符数组作为参数，并且使用数组选择来按顺序查看每一个字符：

```
int strlen(char str[]) {
   int n = 0;
   while (str[n] != '\0') {
      n++;
   }
   return n;
}
```

然而，该实现可以使用如下代码进行替换，该函数将一个字符指针作为参数：

```
int strlen(char *cp) {
   int n = 0;
   while (*cp++ != '\0') {
      n++;
   }
   return n;
}
```

在这个版本中，while循环按顺序检测每一个字符是否为空字符。同时表达式*cp++使字符指针自动向前检查字符串的下一个字符。然而，使指针向前直到结尾，然后使用指针下标来确定字符数目，这会显得更为高效：

```
int strlen(char *str) {
   char *cp;
   for (cp = str; *cp != '\0'; cp++);
   return cp - str;
}
```

注意for循环由一个分号终止，表明循环体为空，而同时出现一些初始化、测试和步长表达式。

504

如果你在一个应用中遇到一些类似于对字符串进行操作的代码，其实现一般会省去通过显示的比较来判定当前字符不为空。在C和C++中，如果一个整数不为零，那就代表true。表达式*cp++可以说明这一规则，当该表达式为true时，其值一定不等于空字符，空字符对应着ASCII的码值0。

涉及指针运算的代码会比之前的例子显得更加模糊。C标准库中strcpy函数的实现就是将src中C风格的字符串拷贝到字符数组dst中，如下所示：

```
void strcpy(char *dst, char *src) {
   while (*dst++ = *src++);
}
```



while 循环的主体是空的，所有的工作都在下面这个高度线性化的表达式中完成：

```
*dst++ = *src++
```

该表达式的作用是将当前 src 指向的字符复制到地址 dst 中，并且在该过程中两指针同时自增 1。只有当代码将字符串结尾的空字符复制时，表达式结果才变成 0，也就是 false。

正如你所看到的，尽管代码从技术上说没什么问题，但 strcpy 的定义被标记上错误图标。strcpy 的实现精确地表达了 C++ 详细说明的内容。问题（同时也是做错误标记的原因）是由于 C++ 继承自旧的 C 语言，strcpy 的具体实现使得该函数使用起来极度危险。主要是因为 strcpy 并不检查目的指针数组是否有足够的空间来存储源字符串的一个复制。如果内存不足以存储完整的源字符串，strcpy 会继续向前并为有其他分配目的的内存复制额外的字符。这种问题被称为**缓冲区溢出错误**（buffer overflow error）。

缓冲区溢出错误经常会以一种极难调试的形式导致程序莫名其妙地崩溃。然而，这种类型的错误引起的问题会比错误本身更加严重。没有检查缓冲区溢出的应用程序会在调用 strcpy 时更有可能陷入严重的安全风险。通过允许 strcpy 向不足的内存空间复制精心挑选的字符串，攻击者能够用这些恶意的代码重写某些应用来获得计算机的控制权。 505

11.4.4 指针运算和程序风格

由于历史的原因，许多 C++ 程序员会在使用数组更简单的地方使用指针运算来处理问题。除了本章为了阐明概念使用的代码外，本文的示例都尽量避免指针运算，而借助于数组索引来提高可靠性。通常情况下，一般建议在自己的代码中也这样做。

如果你曾经负责维护过代码，肯定遇到过 *p++ 这样的情况，可能还有很多更晦涩的指针运算示例。既然维护代码是编程过程中的一个关键部分，就需了解指针运算是如何进行的。标准模板库正是使用 *p++ 的语法模式来实现迭代器的，这将在第 20 章进行介绍。到那时，记住 *p++ 是 C++ 中检索一个数组的当前元素并且将索引指向下一个元素的简写是非常有帮助的。

本章小结

本书的目标之一就是鼓励你使用高级结构，这能让你独立于底层实现以抽象思维来思考数据。抽象数据类型和类能够帮助实现这一全局观点。同时，想要高效地使用 C++ 语言需要你对数据结构如何在内存中表示要心中有数。在本章，你有机会看到了这些数据结构如何存储的，并且了解了当你编写程序时其底层是如何运行的。

本章的主点包括：

- 现代计算机信息的基本存储单元是位，它有两种状态。内存图中一般用二进制数 0 和 1 来表示位的状态，但是就应用而言，将其想象成 false 和 true，或者 off 和 on，二者是等价的。
- 在硬件中，位序列的组合构成了更大的数据结构，包括字节（一个字节有 8 位）和字（一个字有 32 位或 64 位，这取决于机器结构）。
- 计算机的内存是按字节序列分配的，每个字节由它在序列中的索引位置区分，该位

置又被称为地址。

- 计算机科学家倾向于以十六进制来表示地址值以及内存单元的数据，因为这样做使得辨别单个位变得简单。
- C++ 的基本数据类型要求不同的内存容量。一个 char 类型的值需要一个字节，而 int 类型的值一般要求四个字节，而 double 类型的值则要求八个字节。
- 内存中的地址本身也是由数字表示的，也可以像数字一样被操作。表示其他数据地址的数据称为指针。在 C++ 中，指针变量的声明是在声明变量时在变量名前加上星号。
- 指针的基本操作符是 & 和 *，分别表示获取存储数据的地址和获取存于某个地址中的数据。
- 在 C++ 程序中创建的数据值会根据情况被分配在不同的内存区域。静态变量以及常量被分配在专门存储程序代码和静态数据的内存区。局部变量被分配在栈区，一个方法或函数的所有局部变量都被分配在一个栈帧中。在第 12 章将会看到，程序在运行时会动态分配额外的内存。
- 关键字 this 表示指向当前对象的指针。
- C++ 使用操作符 -> 来选择某个结构或对象的成员，当然该操作符的左操作数必须为指针变量。
- 常量 NULL 用来表示不指向任何对象的空指针。
- 在 C++ 中，通过在栈帧中存储一个指向调用参数的指针来形成引用参数。
- 和大多数编程语言一样，C++ 包含一个内置的用于存储一个有序的、同质元素的集合的数组。正如在一个矢量中，数组中的元素都以 0 开始的整型索引。
- 数组声明是在数组名后面的方括号内以常量说明其容量。声明的容量就是该数组的分配容量，一般大于数组中元素实际使用的有效容量。
- C++ 中数组名在内部被解释为一个指向其首元素的地址。这种设计的一个重要的作用是将数组作为参数传递而不复制其元素，且由函数将数组的地址存储在调用方。最后，如果一个函数改变了作为参数传递的数组的任何一个元素，这些改变对于调用者是可见的。
- C++ 定义了指针运算，当整数和指针相加时，得到的结果是距离指针指向元素向下数一定数目后的元素的地址。因此，如果指针 p 指向 array[0]，则表达式 p+2 的结果是指向 array[2]。
- *p++ 的惯用模式是返回 p 当前所指对象的值，然后 p 加 1 指向数组的下一个元素。

复习题

1. 定义以下几个概念：位、字节和字。
2. 单词“bit”的词源是什么？
3. 2GB 内存的机器有多少字节？
4. 将下面十进制数转换成十六进制数：
   a) 17　　c) 1729
   b) 256　　d) 2766
5. 将下列十六进制数转换成十进制数：

a) 17　　c) CC

b) 64　　d) FADE

6. C—给一个 char 类型值分配多少字节？那 double 类型呢？

7. 判断题：在 C++ 中，char 类型的值总是需要一个字节内存。

8. 判断题：在 C++ 中，int 类型的值总是需要四个字节内存。

9. 如果一个机器使用补码来表示负数，那么 -7 在 32 位的整型格式中的内部表示是多少？

10. 内存的三个区域是什么，分别可以存储哪些类型的数据？

11. sizeof 操作的作用是什么？如何使用？

12. 什么叫做地址？

13. 什么叫做走值？

14. 本章阐明了那些使用指针的原因？

15. 下面声明的变量类型分别是什么：

```
int * p1, p2;
```

16. 指针的两个基本运算是什么？哪一个是取值运算？

17. 解释指针赋值和看到赋值的差异。 508

18. 假定 int 类型变量和所有的指针都需要四个字节内存，画图表示包含下列声明的栈帧：

```
int v1 = 10;
int v2 = 25;
int *p1 = &v1;
int *p2 = &v2;
```

在你的图中，追踪以下语句的操作：

```
*p1 += *p2;
p2 = p1;
*p2 = *p1 + *p2;
```

19. 判断题：对任何变量 x，表达式 *&x 与 x 同义。

20. 判断题：对任何变量 p，表达式 &*p 与 p 同义。

21. 如何使用指针实现引用调用？

22. 写出下面数组变量的声明：

a) 数组 realArray 包含 100 个浮点数。

b) 数组 inUse 包含 16 个布尔值。

c) 数组 lines 可容纳 1000 个字符串。

理记声明常量以指定这些数组所分配的容量。

23. 写出变量声明和必要的 for 循环来创建和初始化下面的整型数组：

序列

0	1	4	9	16	25	36	49	64	81	100
0	1	2	3	4	5	6	7	8	9	10

24. 分配容量与有效容量的区别是什么？

25. 假设 intArray 声明为：

```
int intArray[10];
```

509

j 是一个整型变量，描述计算机执行下面表达式的步骤：

```
&intArray[j + 3];
```

26. 如果 array 被声明为一个数组，描述表达式

```
array[2]
```

与表达式

```
array + 2
```

的区别。

27. 假定在你所用的系统上，一个 double 型变量占用八个字节内存。如果 doubleArray 地址为 FF00，那么 doubleArray+5 的地址值是多少？

28. 判断题：如果 p 是一个指针变量，表达式 p++ 表示 p 的内部表示加 1。

29. 描述表达式 *p++ 的作用。

```
*p++
```

30. 在表达式 *p++ 中，哪个操作符（* 或 ++）先运算？通常，C 语言一元操作符的运算规则是什么？

习题

1. 从本章你了解到，整数在计算机内部由一连串位表示，每一个位是二进制计数系统的一个数字，即 0 或 1。N 个位可以表示 2N 个不同整数。例如，3 个位足以表示 0 到 7 之间的 8 个整数，如下所示：

```
000 → 0
001 → 1
010 → 2
011 → 3
100 → 4
101 → 5
110 → 6
111 → 7
```

510

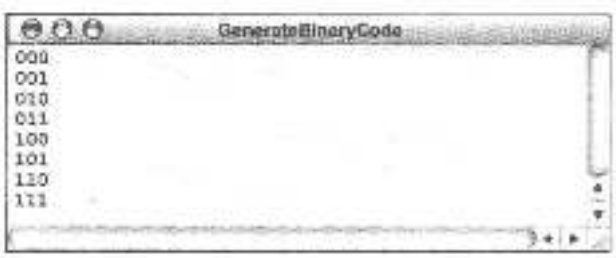
这些整数的位表示遵循一个递归模式。N 位二进制数包含下面两种数据集合：

- 以 0 开头，后面有 N-1 位的二进制数；
- 以 1 开头，后面有 N-1 位的二进制数。

编写递归函数：

```
void generateBinaryCode(int nBits);
```

它可以生成用特定位数所表示的所有的二进制数。例如，调用 generateBinaryCode(3); 应该产生如下输出：



2. 尽管对于大多数应用程序而言，习题 1 的二进制编码显得非常理想，但是它还是有一定的缺陷。从标准二进式可以看出：在某些点上，程序列会同时改变好几位。例如，在三位二进制编码中，从 3（011）到 4（100）的每位都发生了变化。

在一些应用程序中，使用这种位模式去表示相邻数字会引发问题。想象一下你在使用一个硬件度量设备，该设备恰好要产生三位二进制数表示 3 和 4。有时，设备要产生 011 来表示 3，有时则要

以 011 表示 4。为了使该设备正确工作，每位所进行的转变必须同步。如果第一位快于其他位，设备就会额外产生一些中间状态，可能是 111，这将是一个很不精确的结果。

事实证明，你可以通过改变数字系统来避免此问题。你可以在表示中控制相邻两数的二进制表示使其只有一位变化以给 0 到 7 赋 3- 位二进制数。这种编码方式被称为格雷码（Gray code）（以它 511
的发明者数学家弗兰克·格雷命名），表示如下：

```
000 → 0
001 → 1
011 → 2
010 → 3
110 → 4
111 → 5
101 → 6
100 → 7
```

在格雷码表示中，3 和 4 只有一位不同。如果该设备采用格雷码，值在 3 和 4 之间振荡时，不同的位会立刻变换状态，不会出现同步的问题。

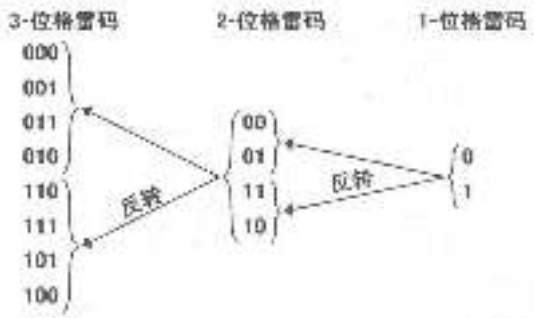
创建 N 位格雷码的递归思想如以下总结的步骤所示：

1）写出 $N-1$ 位的格雷码。

2）按相反顺序复制序列，并将其放在原始序列的下面。

3）在原先编码的序列前加 0，再在反转后的序列前加 1。

下图说明了 3- 位格雷码的形成过程：



编写一个递归函数 generateGrayCode(nBits) 来生成给定位数的格雷码。例如，如果调用以 512
下函数：

```
generateGrayCode(3)
```

程序将会产生如下输出：



3. 编写 IntegerToString 和 stringToInteger 重载函数版本，取第二个表示基数的参数，该基数是一个 2 到 36 的整数（十进制数加上 26 个字母）。例如，调用

```
integerToString(42, 16)
```

应该返回字符串 "2A"。同样地，调用

```
stringToInteger("111111", 2)
```

将会返回整数 63。你的函数应能处理负数，并且当 stringToInteger 的第一个参数对于一个特定的基数来说越界时应能够报错。

4. 重写第 6 章习题 11 的表达式计算器，使其输入输出值均能用十六进制表示。下面是该程序一个运行示例：

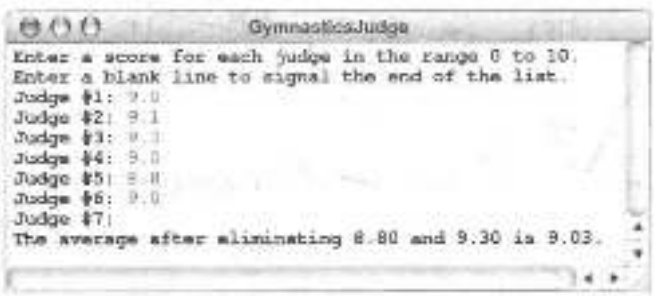


513

在编写该程序时，最简单的方法是将表达式按字处理而不参数数据处理，然后调用你在习题 3 中编写的函数来实现该转换。

5. 重写图 2-3 所示的 Quadratic 程序，使其使用明确的指针而不是引用调用来得到函数 getCoefficients 和 solveQuadratic 的返回值。

6. 使用 MAX_JUDGES 的定义，并且将 11.3.4 节中的 scores 作为一个初始指针，编写一个程序读取裁判的分数（0 到 10），然后去掉最高分和最低分计算其平均值。你的程序应该能不断接受输入，直到达到裁判给的最高分，或者有用户输入一个字符来终止程序。该程序的一个运行示例如下图所示：



7. 重写图 10-3 所示的归并排序算法的实现，使其对一个数组而不是矢量进行排序。正如在 11.3.5 节中所示的选择排序算法的实现，你的函数原型如下：

```
void sort(int array[], int n)
```

8. 尽管在 C++ 语言中复制流对象是不合法的，但是你可以在数据结构中存储一个指向一个流的指针。例如，可以如下所示实现此方法：

```
void setInput(istream & infile)
```

该方法在 6.4 节的 TokenScanner 类中介绍过。你所要做的就是以指针变量存储 infile 的地址，然后解析这个指针从流中读取值。

扩展图 6-10 和图 6-11 所示的简化 TokenScanner 类，使其能够读取输入流中的记号及字符串数据。

514

第 12 章
Programming Abstractions in C++

动态内存管理

爆炸的系统和无用的命名会导致你的存储过载。

——玛丽·雪莱，《弗莱肯斯坦》(*Frankenstein*)，1818

515

截至目前，你已经接触过两种为变量分配内存的机制。当你声明一个全局常量或变量，编译器会给这个全局常量在整个程序生命周期内分配一块持久的内存空间。这种分配模式被称为**静态分配**（static allocation），因为变量在整个程序生命周期被分配到固定的内存地址。另一种是当你在一个函数内声明一个局部变量时，该变量的存储空间被分配在栈中。在调用这个函数时，将为该变量分配存储空间；当函数返回时，该变量所占用的存储空间就会被自动释放。这种内存分配模式被称为**自动分配**（automatic allocation）。然而，还有第三种内存分配方式，即当程序运行时，允许你获得新的内存空间。这种模式叫做**动态分配**（dynamical location）。

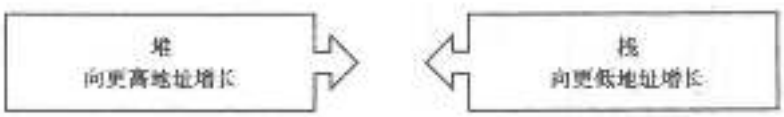
动态分配是你可以自认为精通 C++ 之前必须掌握的最重要的技巧之一。部分原因是动态分配内存允许在程序运行时扩展所需的数据结构。例如在第 5 章介绍过的集合类就依赖于这一能力。当然对于 `Vector` 或者 `Map` 的大小并没有什么限制。如果这些类需要更多的内存，它们只需要直接向系统申请。

在 C++ 中，动态分配内存特别重要，因为 C++ 与大多数现代语言相比，赋予了编程者更多的职责。在 C++ 中，知道如何分配内存是不够的，你还必须学习当不再需要这块内存空间时应如何释放它。按照一定原则分配和释放内存的过程被称为**内存管理**（memory management）。

12.1 动态分配和堆

当程序加载到内存时，通常只会占用可用存储空间的一小部分，和大部分编程语言一样，无论你的应用什么时候需要更多的内存，C++ 都允许你给程序分配一些闲置的存储空间。例如，如果在程序运行时需要给一个数组分配空间，你可以保留一部分未分配的内存，留下剩余部分给随后的分配行为。程序中一组未分配的可用内存池被称为**堆**（heap）。

在现代计算机体系结构中，整个内存空间被设置为栈和堆，它们以相反的方向增长，如下图所示：



516

这种策略的优点在于任一区域都可以依照需要增长，直到所有可用内存填满为止。

当你需要内存时，从堆中分配内存的能力是已被普遍应用于编程的一种非常有效的方法。例如，所有的集合类用堆存储它们的元素，因为动态分配对于构建数据结构，使其依照需求扩展来说是必不可少的。在这一章的后面，你会有机会搭建一个简易集合类版本。然

而，在此之前，学习动态分配的基本机制并了解其过程如何进行是非常重要的。

12.1.1 new操作符

C++使用new操作符从堆中分配内存。new操作符最简单的一种形式是：取一种类型，并在堆中分配一块空间给所指定类型的变量。例如，如果你想在堆中分配整型量的内存空间，你可以调用：

```
int *ip = new int;
```

这个new操作符的调用将返回在堆中已经被留出用来存储整数的存储位置的地址。如果堆中第一个空闲的字位于地址1000，当前栈帧中的变量ip会被赋予这个地址空间，如下图所示：



从概念上来说，栈帧中的局部变量ip指向新分配的堆空间。为清楚起见，你可用箭头代替地址来表示这种关系，如下图所示：



地址空间的内容未变，唯一变化的就是你如何去画这个图。

一旦你在堆中为整型数分配了空间，那么你可以通过解析指针来引用该整数。例如，通过执行下述语句将整数42存储在新分配的字中：

```
*ip = 42;
```

这将使内存如下发生变化：



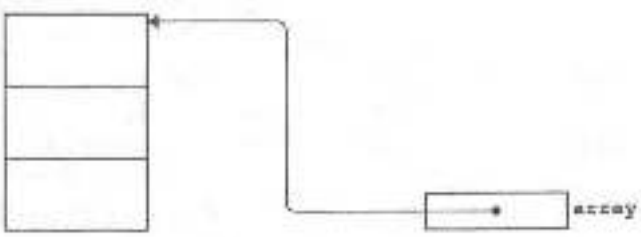
517

12.1.2 动态数组

new操作符也能做到在堆上给一个数组分配空间，它被称为动态数组（dynamic array）。为一个动态数组分配空间，你要在类型名之后跟一对方括号，并在其中指明要求所分配的元素数目。因此，声明语句：

```
double *array = new double[3];
```

初始化数组array，以便于其指向连续的足够大的一块内存来存储三个double类型的变量，如下图所示：



变量array现在是一个全功能的数组，它的存储现在位于堆而不是栈。你可以给数组array的每一个元素赋值，接着这些元素会被存储在堆上合适的内存位置。

尽管动态数组在大量数据结构的底层实现中很有用，但在大多数应用中却很少使用。第

5 章中介绍过的 Vector 类在实际应用中是一个更好的选择，主要是因为 Vector 类执行自己的内存管理，从而减轻你的责任。

12.1.3 动态对象

new 操作符也用于在堆上分配对象和结构体。如果你仅提供类名，就像如下语句：

```
Rational *rp = new Rational;
```

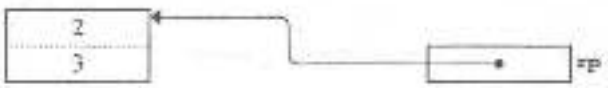
C++ 在堆中为 Rational 对象分配空间，并调用默认构造函数，在内存中构造出以下情形：



如果你在类型名后提供了参数，C++ 会调用对应的构造函数。以下声明： 518

```
Rational *rp = new Rational(2, 3);
```

因而创建了以下内存状态：



12.2 链表

指针能够在一个大型数据结构中记录不同值之间的联系。当一个数据结构包含了另一个数据结构的地址时，这种结构关系被称为**链接**（link）。在接下来的章节，你会看到更多的有关链接结构的例子。为了让你对即将展现的链接的吸引力有一个预览，同时为了提供更多在堆中对结构使用指针的例子，这一节接下来会介绍被称为**链表**（linked list）的基本数据结构。在一个线性链表中，指针将一个个数据值像一个链一样连接起来。

12.2.1 刚铎灯塔

我最喜欢的链表的例子是从下面这一段由 J.R.R. 托尔金所著的《王者归来》的文章中获取的灵感：

> 甘道夫大声地向它的马匹戏道："快，幻影！我们必须加速。时间很紧迫。看！刚铎的灯塔已经点亮，寻求帮助。战争已经爆发。看，阿诺丁峰已经燃起了火焰，伊瑟纳基亚也已泛红，火焰正在向西蔓延：纳多尔、艾瑞哥、伦－里被卡兰奇依、洛汗直至界河到费力安。"

改编《指环王》三部曲的这一段情节时，彼得·杰克逊为这一场景创作了一个启发性的解释。在米那斯提力斯的第一座灯塔点亮之后，我们看到信号在每个灯塔的看守人的操作下，从一个山顶传递到另一个山顶，看守者总是十分警惕，只有看到前一站的灯火被点亮才会点燃他们自己的灯火。刚铎的险情因此迅速地在洛汗众多被分离的联盟国之间传递，如图 12-1 所示。

为了使用 C++ 来模拟刚铎灯塔这一情景，你需要定义一个结构类型来表示链中的每一座灯塔。这个结构必须包含每一座灯塔的名字以及链中指向下一座灯塔的指针。因此，这个表示米那斯提力斯的结构包含一个指向 Amon Din 的指针，而它按顺序又包含一个 Eilenach 结构的指针，等等，直到指向一个标志着这条链结束的空指针为止。

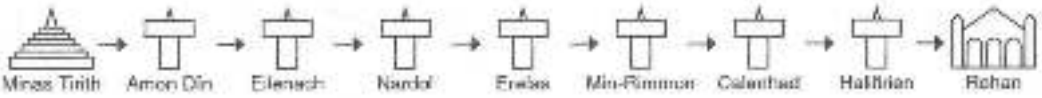


图 12-1 托尔金刚铎灯塔的语义图

如果你采用这种方法，灯塔 Tower 的结构定义就如下所示：

```
struct Tower {
   string name;
   Tower *link;
};
```

name 域记录了灯塔的名字，link 域指向链中下一个信号灯塔。图 12-2 阐明了这些结构的内存表示法。每一个独立的 Tower 结构代表了链表中的一个结点，其内部的指针叫做链接。结点可能出现在内存的任何地方；其顺序是由每一个结点链接其后继单元的链接所决定的。

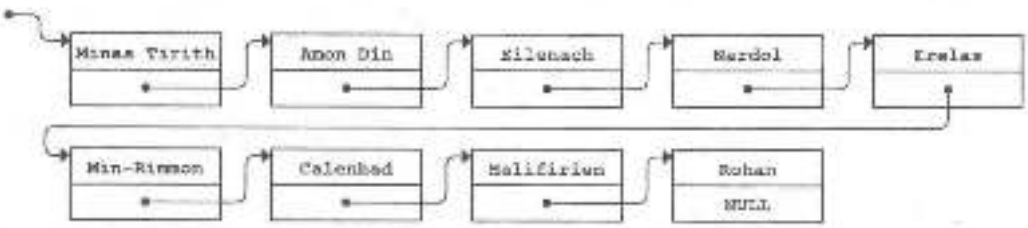


图 12-2 链表表示刚铎灯塔

图 12-3 中的程序模拟了刚铎灯塔的点亮过程。该程序以调用 createBeaconsOfGondor 函数开始，它构建该链表并返回指向链表中第一座灯塔的指针。每一座灯塔都是由 createTower 函数创建，它为新的 Tower 值分配空间，然后通过参数传入其 name 和 link 域值。createBeaconsOfGondor 的实现逆序构建链表，即首先在链表的尾部创建了第一个灯塔 Rohan，然后继续向前，一次一座灯塔，直到它到达链表的开始处 Minas Tirith 为止。在一个更通用的应用中，更倾向于从数据文件中读取每一个存储单元的值，你在习题 3 中会有机会扩展这个实现。

```
/*
 * File: BeaconsOfGondor.cpp
 * -------------------------
 * This program illustrates the concept of a linked list by simulating the
 * Beacons of Gondor story from J. R. R. Tolkien's Return of the King.
 */

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

/*
 * Type: Tower
 * -----------
 * This structure contains the name of the tower and a link to the next one
 */

struct Tower {
   string name;            /* The name of this tower                    */
   Tower *link;            /* Pointer to the next tower in the chain */
};

/* Function prototypes */
```

图 12-3 刚铎灯塔的仿真程序

```
Tower *createBeaconsOfGondor();
Tower *createTower(string name, Tower *link);
void signal(Tower *start);

/* Main program */

int main() {
   Tower *list = createBeaconsOfGondor();
   signal(list);
   return 0;
}

/*
 * Function: createBeaconsOfGondor
 * Usage: Tower *list = createBeaconsOfGondor();
 * ---------------------------------------------
 * Creates a linked list of the towers described by Tolkien.  The function
 * builds the link backwards and returns a pointer to the first tower.
 */

Tower *createBeaconsOfGondor() {
   Tower *tp = createTower("Rohan", NULL);
   tp = createTower("Halifirien", tp);
   tp = createTower("Calenhad", tp);
   tp = createTower("Min-Rimmon", tp);
   tp = createTower("Erelas", tp);
   tp = createTower("Nardol", tp);
   tp = createTower("Eilenach", tp);
   tp = createTower("Amon Din", tp);
   return createTower("Minas Tirith", tp);
}

/*
 * Function: createTower
 * Usage: Tower *chain = createTower(name, link);
 * ----------------------------------------------
 * Creates a new Tower structure with the specified components.
 */

Tower *createTower(string name, Tower *link) {
   Tower *tp = new Tower;
   tp->name = name;
   tp->link = link;
   return tp;
}

/*
 * Function: signal
 * Usage: signal(start);
 * ---------------------
 * Generates a signal starting at the current tower and proceeding
 * through the end of the chain.
 */

void signal(Tower *start) {
   for (Tower *tp = start; tp != NULL; tp = tp->link) {
      cout << "Lighting " << tp->name << endl;
   }
}
```

图 12-3 （续）

这个链表被初始化之后，主程序调用 signal 函数显示出灯塔的名字。如果按照图 12-2 所示的链表开始，程序的输出结果如下：



519 / 522 尽管这个结果不如杰克逊的更新卡获奖影片中的场景那么激动人心，但你可以扩展这个 BeaconsOfGondor 程序，以便它使用图形库展示出这条链上灯塔之间的烽火传递。

12.2.2 链表内的迭代

signal 这段代码表明了关于链表的一个基本编程模式，其具体表现在 for 循环：

```
for (Tower *tp = start; tp != NULL; tp = tp->link)
```

在 signal 函数体中，for 循环模式的效果与经典的 for 循环遍历数组中的每一个元素一样，这个循环也是遍历链表中的每一个元素。初始化表达式声明了指针变量 tp，并将其初始化，以便它指向链表中第一个元素的指针。条件表达式确保了只要 tp 变量不为空，即它的值若没有指向链表的末尾，循环就会继续。for 循环中的步长表达式为：

```
tp = tp->link;
```

这改变了 tp 的值，从而改变了当前 Tower 结构的 link 指针的值，使 tp 指向链表中的下一座灯塔。

12.2.3 递归和列表

尽管用来处理链表的大多数代码都如之前小节所描述的迭代方式进行，但链表的递归特性在实际中很有用。简单情况就是它为空列表（empty list），在 C++ 中，用空指针 NULL 表示。一般情况下，一个链表由一个数据单元后跟一个链表构成。这种形式导致了自然的递归分解，在你第一次检查链表是否为空时，如果不为空，在第一个存储单元上执行一些操作，用该链表的剩余了链表作参数递归地进行调用。例如，你可以像下段代码一样递归地实现 signal 函数：

```
void signal(Tower *start) {
   if (start != NULL) {
      cout << "Lighting " << start->name << endl;
      signal(start->link);
   }
}
```

12.3 释放内存

尽管计算机的内存一直在增大，但它总是有限的。因此，堆空间最终会被耗尽。当这种情况发生时，new 操作符将不能分配所需要的内存空间。对分配内存空间的请求失败是非常 523 严重的错误，以致通常程序没有任何办法将其恢复。

12.3.1 delete 操作符

为了确保你的内存不会被耗尽，更好的策略是每当你使用完堆空间后便释放它。为了实现此策略，C++ 提供了 delete 操作符，它取 new 操作符事先分配内存的一个指针，然后释放由该指针所指的内存空间。

例如，假定你已经声明了一个如下所示指向 int 型的指针：

```
int *ip = new int;
```

在程序的后面，你发现不再需要这个指针变量所指的空间。此时，该做的事情就是通过调用

以下语句释放该指针变量所指的空间：

```
delete ip;
```

如果堆内存是一个数组，你需要在 delete 关键字后面加上一对方括号。因此，如果通过以下语句：

```
double *array = new double[3];
```

分配了一个动态数组，你可以通过执行以下语句来释放已分配的那块内存：

```
delete[] array;
```

如果你需要释放一个链表，那你必须对每一个存储单元进行迭代，释放你所涉及的每一个单元的空间。然而，你这样做需要多加小心，因为你不能在释放一个对象后再访问它的 link 域。因此，在你释放指针当前所指的存储单元之前，你的循环结构必须存储指向下一个存储单元的指针，就像下面 while 循环那样：

```
while (list != null) {
   Tower *next = list->link;
   delete list;
   list = next;
}
```

如果你使用下面的递归算法，那么删除链表中的每一个指针就相对简单了：

```
void freeList(Tower *list) {
   if (list != NULL) {
      freeList(list->link);
      delete list;
   }
}
```

[524]

12.3.2 释放内存策略

知道何时释放一块内存并不容易，尤其是当程序比较大时。如果程序的某些部分共享一些已经在堆中被分配的数据结构，或许就不太可能辨别任一部分的内存是否需要。对于简单的程序，一直运行直到产生预期的结果，你可以分配你想要的内存，不用为再次释放它而烦恼，这也不失为一个合理的策略。然而这样做很可能使你养成一个危险的习惯。如果另一个程序员试图在长时间运行的应用中使用你的代码，那么你对内存的忽视会导致一个严重的问题。因此好的做法就是确保在某一时刻释放你所分配的堆内存。当一个程序未能释放它不用的堆内存时，这个程序就被称为存在**内存泄漏**（memory leak）。

某些编程语言，包括带有大量脚本语言的 Java 语言，支持一种动态分配的策略，它会自动查找不再使用的内存，然后释放之。这种策略被称为**垃圾回收**（garbage collection）。尽管垃圾回收增加了一些开销，但它也使得程序员对于内存的管理非常简单。快速查找整个堆，计算出堆中哪些部分正在使用是需要时间开销的。更糟糕的是，垃圾回收经常使我们难以预测执行一个特定任务所用的时间。如果一个特定的函数调用被垃圾回收的运行所中断，这通常会使得运行很快的函数可能会突然需要大量的运行时间。如果这个函数负责某些实时任务，则该应用可能就不能满足其响应时间。

然而，大多数新的编程语言还是采用垃圾回收作为它们的内存管理策略。作为一个普遍原理，用牺牲一小部分处理时间来换取大量的编程时间是值得的。毕竟处理器便宜，而程序

员的代价高。然而，由于C++起源于较早的年代，无论如何，C++的设计者还是决定将释放堆内存的职责交到程序员的手中，而不是将这个任务托付给程序自动处理。

幸运的是，设计者们通过给程序员提供一种与众不同的内存管理策略极大地简化了这个问题，用以弥补垃圾回收的不足。在C++中，每一个类都允许指定它的一个对象消失时会发生什么。在良好设计的C++应用中，每个类都要对它自己的堆内存负责，从而把用户需要准确地记忆当前那些堆是活动的这一几乎不可能的任务中释放了出来。学习有效地使用这个策略是你作为一个C++程序员必须掌握的重要技能之一，因此，在下节详细地讲述该方
525 法的细节上多花点心思是很值得的。

12.3.3 析构函数

就像你已经看过的例子那样，C++类典型地定义了一个或多个用来初始化对象的构造函数。每一个类也可以定义一个**析构函数**（destructor），当一个类的对象消亡时，析构函数被自动调用。这个析构函数可以完成各种清理操作。例如，它可以关闭一个对象打开的任何文件。然而，析构函数最重要的一个作用是释放对象所创建的所有堆内存。

在C++中，析构函数的名字就是简单地在类名前加一个被称为波浪符（tilde）的~符号。因此，如果你需要为一个叫做MyClass的类定义一个析构函数，其接口原型应如下所示：

```
~MyClass();
```

和构造函数一样，析构函数没有返回类型。和构造函数不同的是析构函数不能重载。每一个类只能有一个无参的析构函数。

在C++中，对象会以几种不同的方式消失。大多数情况下，对象会像函数体中的局部变量那样被声明，意味着它被分配到栈中。这些对象在函数返回时自动消亡。这也就意味着它们的析构函数同时被自动调用，这恰恰为类定义释放在其对象生命周期所分配的堆内存提供了机会。大多数C++文档中，将一个函数返回时其局部变量消亡的现象称为超出范围（go out of scope）。

当计算表达式时，即使它们的值不会存储在局部变量中，对象也可以作为临时变量被创建。例如，第6章的Rational类中的测试程序包含了以下代码：

```
Rational a(1, 2);
Rational b(1, 3);
Rational c(1, 6);
Rational sum = a + b + c;
```

当函数返回时，局部变量a、b、c和sum将会销毁。然而，上述最后一行代码在计算最终结果之前，会将表达式a+b+c的值作为Rational类型的临时对象而暂存。一旦该表达式的值赋给了对象sum，则上述临时对象就会超出作用域范围。如果Rational类有一个析
526 构函数，此时系统就会自动调用其析构函数来释放该临时对象所占用的内存。

12.4 定义CharStack类

为了对如何编写一个使用动态内存分配的类有更深入的认识，一个最简单的方法是实现第5章的某个容器类。最易于实现的容器类就是Stack类，主要原因在于它只有几个公有方法。然而，为了使该实现更简单，有必要约束该栈中的元素类型（基类型）均为同一种数

据类型——这种情况下，元素类型为字符的栈会在第 13 章看到是非常有用的。当你学完了第 14 章介绍的模板机制后，你就有机会去编写一个多态版本的 Stack 类。现在，你的目标是了解像 Stack 类是如何采用动态分配来管理内存的。为此，这个栈的基类型并不重要，字符栈将足以阐明动态内存分配的一些通用原理。

12.4.1 charstack.h 接口

图 12-4 所示的是接口 charstack.h 的内容。这个接口对外提供了包括默认构造函数，析构函数和各种方法，即 size、isEmpty、clear、push、pop 和 peek 方法，它们定义了栈的抽象行为。唯一不同于上述方法行为的是其析构函数，它的原型如下：

```
~CharStack();
```

CharStack 类的析构函数永远也不会被明确地调用。在接口中出现的这一函数原型只是让编译器知道 CharStack 类中定义了一个只要 CharStack 的对象超出作用域范围就需要调用的析构函数。该析构函数负责释放该类中已分配的堆空间，并对用户隐藏了内存管理细节。

如果你读完了图 12-4 所示的代码，你将发现 CharStack 类的程序清单中没有出现其私有部分内容。该私有部分的位置处于一个稍后会被填充的盒中。从概念上来说，一个类的私有部分并不是公有接口的一部分。遗憾的是，C++ 的语法规则要求私有部分要在类的内部定义。任何私有部分的执行代码要在 .cpp 文件中重写，但是对于这些方法的原型和实例变量的声明必须包含在 .h 文件中。当你作为一个用户使用类时，你应该养成一个习惯，即忽视私有部分的细节。这些细节只有在理解了类的公有特性后才可能了解。本书中忽略了私有部分的接口清单，便隐藏这些细节更加简单。 527

```
/*
 * File: charstack.h
 * -----------------
 * This interface defines the CharStack class, which implements
 * the stack abstraction for characters.
 */

#ifndef _charstack_h
#define _charstack_h

/*
 * Class: CharStack
 * ----------------
 * This class models a stack of characters.  The fundamental operations
 * are the same as those for the Stack<char> class.
 */

class CharStack {

public:

/*
 * Constructor: CharStack
 * Usage: CharStack cstk;
 * ----------------------
 * Initializes a new empty stack that can contain characters.
 */

   CharStack();

/*
 * Destructor: ~CharStack
 * Usage: (usually implicit)
```

图 12-4 CharStack 类基于数组的接口

```
 * ------------------------
 * Frees any heap storage associated with this character stack.
 */

   ~CharStack();

/*
 * Method: size
 * Usage: int nElems = cstk.size();
 * --------------------------------
 * Returns the number of characters in this stack.
 */

   int size();

/*
 * Method: isEmpty
 * Usage: if (cstk.isEmpty()) . . .
 * --------------------------------
 * Returns true if this stack contains no characters.
 */

   bool isEmpty();

/*
 * Method: clear
 * Usage: cstk.clear();
 * --------------------
 * Removes all characters from this stack.
 */

   void clear();

/*
 * Method: push
 * Usage: cstk.push(ch);
 * ---------------------
 * Pushes the character ch onto this stack.
 */

   void push(char ch);

/*
 * Method: pop
 * Usage: char ch = cstk.pop();
 * ----------------------------
 * Removes the top character from this stack and returns it.
 */

   char pop();

/*
 * Method: peek
 * Usage: char ch = cstk.peek();
 * -----------------------------
 * Returns the value of the top character from this stack without
 * removing it.  Raises an error if called on an empty stack.
 */

   char peek();

   The private section of the class goes here.

};

#endif
```

图 12-4 （续）

然而，在类接口清单中忽略私有部分的细节还有另一个重要原因。之后几章我们主要关注类在效率和实现策略之间的关系上。清楚地理解这种关系要求能够比较同一个类的多种实现。即使这些实现使用不同的数据结构，类的公有部分仍完全一致。在接下来的几节，你会有机会探索字符栈类的几种可能的实现，每一种实现都需要你用不同并且合适的代码块代替

图 12-4 中的盒子。

12.4.2 选择字符栈的表示

当你打算为一个类设计其基础的数据结构时，你应该询问你自己的第一个问题就是：该类的每个对象中都需要存储什么信息。一个字符栈必须记录字符进栈的次序。对于任何集合类，没有理由对栈可以包含任意的字符数目加以限制。因此，作为一个实现者，你需要为其选择一个可以在程序运行时动态扩展的数据结构。

当你思考这样的数据结构应该是什么样子时，你的一个想法可能是采用 Vector<char> 去存储栈中的元素。Vector 可以动态变化，这正是你在这个应用中所需要的。事实上，采用 Vector<char> 作为类的基本表示会使得其实现极其简单，正如以下的图 12-5 和图 12-6 所示。

```
/* Private section */

/*
 * Implementation notes
 * --------------------
 * This version of the CharStack class uses a Vector<char> as its
 * underlying representation.  Characters are always added and
 * removed from the end, which gives rise to the last-in/first-out
 * behavior that is characteristic of stacks.
 */

private:

/* Instance variables */

   Vector<char> elements;     /* Data structure to hold the stack elements */

};
```

图 12-5 使用 Vector 作为 CharStack 类的私有部分 530

```
/*
 * File: charstack.cpp
 * -------------------
 * This file implements the CharStack class using a Vector<char> as the
 * underlying representation.  The Vector class already implements most
 * of the essential operations for the CharStack class, which can simply
 * forward the request to the underlying structure.  The methods are
 * short enough to require no detailed documentation.
 */

#include "charstack.h"
#include "error.h"
#include "vector.h"
using namespace std;

CharStack::CharStack() {
   /* Empty */
}

CharStack::~CharStack() {
   /* Empty */
}

int CharStack::size() {
   return elements.size();
}

bool CharStack::isEmpty() {
   return elements.isEmpty();
}
```

图 12-6 使用 Vector 的 CharStack 类的实现

```
void CharStack::clear() {
   elements.clear();
}

void CharStack::push(char ch) {
   elements.add(ch);
}

char CharStack::pop() {
   if (isEmpty()) error("pop: Attempting to pop an empty stack");
   char result = elements[elements.size() - 1];
   elements.remove(elements.size() - 1);
   return result;
}

char CharStack::peek() {
   if (isEmpty()) error("peek: Attempting to peek at an empty stack");
   return elements[elements.size() - 1];
}
```

531

图 12-6 （续）

选择 Vector<char> 作为字符栈类的底层表示是一个正确的做法。你应该总是注意寻找根据已解决的问题能够构造出解决你新问题的方法。此外，作为一种软件工程策略，采用 Vector 类来实现栈类绝对没错。Stack 类的库版本正是这样做的。唯一的问题是采用 Vector 会减弱示例的指导价值。实现 Vector 类很明显比实现栈类更复杂。采用 Vector 类作为底层表示对搞清楚 CharStack 类的操作并无帮助，它仅能把其实现细节隐藏起来。

或许更重要的是，依赖 Vector 类会使得分析 CharStack 类的性能变为困难，因为 Vector 类隐藏了很多复杂性。由于你还不知道 Vector 类的细节，你就无法知道涉及在添加或删除其中一个元素时需要多少工作，如方法 push 和 pop 所需要的工作。以下几章主要的目的是分析数据表示如何影响算法的效率。如果所耗费的所有代价都是可见的，那么分析就很容易进行。

确保没有隐藏耗费代价的方法之一就是限制实现，以便它只能依靠编程语言支持的最简单的操作。在字符栈类的例子中，采用内置的数组类型来存储元素的优势使数组不会隐藏任何东西。从一个数组中选取一个元素需要几个机器指令，这在现代计算机上只需耗费很少的时间。在堆中分配数组空间或是不再使用它时回收其空间，通常都比选择元素耗费更多的时间，然而这些操作仍然是以常量时间运行的。在一个典型的内存管理系统中，分配一个 1000 字节的内存块和分配一个 10 字节的内存块花费的时间一样。

尽管上述操作提供常量时间的运行性能，但是这也不能立即确定数组应作为集合类的底层表示。正如在本节一开始提到的，不管为了在栈中存储字符你采用什么样的表示，都必须允许其存储容量可以扩充。而数组却做不到。一旦你为数组分配了空间，那么它的大小就不能再改变。

然而，你可以做的是首先给数组分配某个固定大小的空间，一旦其空间越界时就可以用一个更大的数组替换它。在这个过程中，你必须把原来数组的所有元素复制到新的数组中，并且确保原来数组所占用的内存已被回收。一旦你完成了这些工作，新数组就会拥有你所期望的存储空间。

532 为了存储栈元素，私有部分的新版本需要记录某些变量值。最为重要的是在 CharStack 类中需要一个指向包含所有字符的动态数组指针。同时也需要记录这个动态数组的大小（即可存储多少个元素），以便程序能分辨出该栈何时用完了存储空间，并需要重新给它分配新

的空间。因为这个值表示当数组在当前情况下可以存储多少个字符，所以该值通常被称为动态数组的容量（capacity）。最后，记住这种字符数组只能包含比它限定容量少1个的字符。因此这个数据结构需要包含一个记录该数组当前有效元素个数的变量。在私有部分包含这三个实例变量使得实现这个堆操作和数组操作的复杂度相当。CharStack类中更新的私有部分显示在图12-7中，相关的实现部分显示在图12-8中。

```
/* Private section */

/*
 * Implementation notes
 * --------------------
 * In this version of CharStack, the characters are stored in a dynamic
 * array that doubles in size whenever the stack runs out of space.
 */

private:

/* Private constants */

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   char *array;           /* Dynamic array of characters   */
   int capacity;          /* Allocated size of that array  */
   int count;             /* Current count of chars pushed */

/* Private function prototype */

   void expandCapacity();
```

图12-7 用动态数组实现的CharStack类的私有部分

```
/*
 * File: charstack.cpp
 * -------------------
 * This file implements the CharStack class.
 */

#include "charstack.h"
#include "error.h"
using namespace std;

/*
 * Implementation notes: constructor and destructor
 * ------------------------------------------------
 * The constructor allocates the array storage for the stack elements and
 * initializes the fields of the object.  The destructor frees any heap
 * memory allocated by the class, which is just the array of elements.
 */

CharStack::CharStack() {
   capacity = INITIAL_CAPACITY;
   array = new char[capacity];
   count = 0;
}

/*
 * Implementation notes: ~CharStack
 * --------------------------------
 * The destructor frees any heap memory allocated by the class, which
 * is just the dynamic array of elements.
 */

CharStack::~CharStack() {
   delete[] array;
}

/*
 * Implementation notes: size, isEmpty, clear
 * ------------------------------------------
 * These methods are each a single line and need no detailed documentation.
 */
```

图12-8 基于数组的CharStack类的实现

```
int CharStack::size() {
   return count;
}

bool CharStack::isEmpty() {
   return count == 0;
}

void CharStack::clear() {
   count = 0;
}

/*
 * Implementation notes: push
 * --------------------------
 * This function first checks to see whether there is enough room for
 * the character and then expands the array storage if necessary
 */

void CharStack::push(char ch) {
   if (count == capacity) expandCapacity();
   array[count++] = ch;
}

/*
 * Implementation notes: pop, peek
 * -------------------------------
 * These functions check for an empty stack and report an error if
 * there is no top element.
 */

char CharStack::pop() {
   if (isEmpty()) error("pop: Attempting to pop an empty stack");
   return array[--count];
}

char CharStack::peek() {
   if (isEmpty()) error("peek: Attempting to peek at an empty stack");
   return array[count - 1];
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This method doubles the capacity of the elements array whenever it runs
 * out of space.  To do so, the method must copy the pointer to the old
 * array, allocate a new array with twice the capacity, copy the characters
 * from the old array to the new one, and finally free the old storage.
 */

void CharStack::expandCapacity() {
   char *oldArray = array;
   capacity *= 2;
   array = new char[capacity];
   for (int i = 0; i < count; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}
```

图 12-8 （续）

尽管在 charstack.cpp 中的大多数代码跟你在本书中较早看到的类很类似，一小部分方法包含了特别的注释。例如，构造函数必须初始化表示一个空栈的内部数据结构。count 变量必须为 0，但没有理由不提供一些含有某些初始容量的栈。在该实现中，构造函数提供了以常量 INITIAL_CAPACITY 值为 10 的字符空间。这个常量值没有什么神奇之处，任何整数都可以作为其值。选择一个大一点的值会减小栈需要扩充的频率，因此节省了执行时间；而选择小一点的值可以节省内存。决定一个适当的值是时间—空间平衡的一个实例，这个话题将在第 13 章更详细地加以讨论。

533
|
535

CharStack 类的下一个方法是其析构函数，如果没有什么特别的原因，这应该是你第一次看到的析构函数。大部分析构函数的职责是释放为类的对象所分配的堆空间。当

CharStack 准备回收空间时，唯一指向所分配的堆空间的指针是动态数组，它的地址存储在实例变量 array 中。因此，析构函数的实现：

```
CharStack::~CharStack() {
   delete[] array;
}
```

delete 关键字后的一对空方括号是必需的，因为它的目标是释放整个动态数组而非该动态数组的某个单元。

在 push 方法中最大的改变来自于基于矢量的实现，push 方法的功能是在栈顶向栈内增加一个新字符。只要该字符栈有空间能存储一个字符，push 方法就会在它第一个不用的数组位置处存储它，并将 count 域值加 1。反之，若数组中没有剩余的空间，事情就会变得复杂。此时，push 方法的实现必须分配有额外更多空间的一个新数组，然后将原来数组中的元素拷贝到新数组。除了包括 push 方法本身的代码外，图 12-8 中的代码还委托其任务给一个私有的 expandCapacity 辅助方法。和所有辅助方法一样，expandCapacity 的原型出现在类定义的私有部分，其实现代码在 .cpp 文件中。正如你在代码中所看到的那样，expandCapacity 方法开始存储一个指向原数组的指针。然后它分配一个具有原数组两倍容量的一个新数组，再把原数组的所有字符拷贝到新数组，最后释放旧数组的存储空间。

C++ 使用析构函数来解决内存释放问题并不是一个神奇的解决方案。charstack.cpp 的实现使得这点更清楚，你仍然要密切关注内存管理并确保你的实现释放了其所分配的堆空间。析构函数的优势在于向用户隐藏了内存管理的复杂性。一个用户可以声明一个 CharStack 对象，使用它一次，接着忘掉它。只要字符栈的堆空间溢出，CharStack 类的实现就负责释放其空间。

12.5 堆 – 栈图

不画大量的图很难理解内存分配是如何工作的。以我的经验，可视化内存分配过程的最有效工具就是**堆 – 栈图**（heap-stack diagram），在这类图中，你可以同时画出堆和栈的内存状态。通过 new 操作符动态分配的内存画在图的左边，它表示堆。对于每个函数调用的栈 536
帧画在图的右边。

画堆 – 栈图不像编写代码的过程（总是要求创造性），它是必不可少的机械活动。然而，事实并不意味着这个过程很琐碎，或者你从这些图中获得的洞察是不重要的。当我帮助学生调试他们的代码时，发现画这些图是有助于学生解决代码令人丧气的阻塞点的最好办法。如果花几分钟画几幅图节省了几个小时的沮丧，那么画图所花费的时间肯定是非常值得的。

理解如何去绘制堆 – 栈图的最好方法就是通过实例学习。假设你已经在图 12-8 中定义了 CharStack 类，然后想通过运行下面的主程序去测试它。测试程序将字母表的每个字母压进一个新声明的 CharStack 对象 cstk 中：

```
int main() {
   CharStack cstk;
   for (int i = 0; i < 26; i++) {
      cstk.push(char('A' + i));
   }
   return 0;
}
```

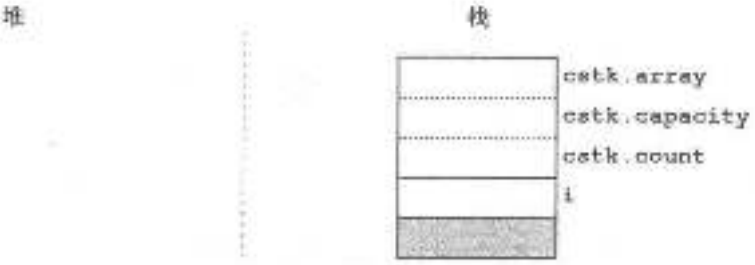
你应该忽视程序实际上并没有产生任何输出这一事实。这个例子的关键不是程序做了什么，而是它如何在栈和堆中分配内存的。这节的剩余部分将以一系列堆－栈图来记录该测试程序的执行过程。图 12-9 概述了堆－栈图的构建过程。我确信在你理解了编译器分配内存的规则之前，这个堆－栈图通过几次这个过程会对你有所帮助。一旦你掌握了其思想，就不用每次执行上述的整个过程了。

1. 以空闲开始。在开始前堆－栈图之前，首先在页面上画一条垂直线将堆空间与栈空间分隔开。一开始，分隔线两边的图都是空的。在一台典型的机器上，堆是朝着高存储地址方向进行扩展，因此在页面上堆是自顶向下延伸的。而栈是朝着相反的方向扩展，因此，栈在页面上是自底向上延伸的。在本书的堆－栈图中，分配的首地址为 1000，栈的末字节地址为 FFFF，这种地址选择更便于堆与栈的扩展。
2. 手动模拟程序执行时的内存分配。内存分配发生在程序运行时，它是一个动态过程。为了方便查看特定时间点的内存情况，你需要从一开始就跟踪程序。当你这样做了之后，其余的规则将适用于合适的时间点。
3. 为每个函数或方法调用添加一个新的栈帧。程序每次开始一个函数调用时（包括 main 函数的初始调用），在图中栈的这边就分配一块新的内存以存储该调用的栈帧内容。一步步地画出栈帧内容以描述其产生过程是非常值得的。
   3a) 添加一个用灰色矩形表示的首字单元。正如本章所指出的，该灰色区域的内容与机器相关，但是画这个灰色矩形有助于我们在视觉上分隔栈帧。
   3b) 将所有在函数中声明的局部变量包含进栈帧中。所创建的栈帧的大小依赖于在函数中所声明的变量个数。仔细阅读代码并找到所有在函数中所声明的局部变量，包括函数参数，它也属于局部变量。在栈帧中分配你找到的所有局部变量所需要的存储空间，然后用变量名标记该空间。通过引用传递的参数仅占用一个指针空间而非其参数值所占用的空间。如果该调用是一个方法调用，栈帧还应另外包含一个标记为 this 的单元用来指向当前对象。栈帧中变量的顺序是任意的，因此你可以按照你想要的顺序来重新排序这些变量。
   3c) 通过拷贝实参的值来初始化形参。当画出了栈帧中的所有局部变量后，你需要将实参值赋值给相应的形参变量。需要注意的是，C++ 中形参是值传递的，除非形参变量声明为引用类型，否则赋值顺序是按照形参顺序而非它们的名字来决定的。当某个参数使用引用传递时，你无需赋实参的值给形参，仅需将实参的地址赋给栈帧中相应的形参即可。
   3d) 继续手动模拟函数体的执行过程。一旦你初始化了形参，你就准备好了执行函数体代码步骤的过程。这个过程很可能涉及赋值（规则 4）、动态分配（规则 5）和函数嵌套调用（规则 3 的递归调用）。
   3e) 每当函数返回时，释放整个栈帧。当你执行完一个函数调用后，正在使用的栈帧就会被自动释放。在栈帧图中，你可以轻易地删除这块被释放的空间。下一个函数调用会重用这些相对的内存空间。
4. 以从右到左的方向通过拷贝值来执行每一条赋值语句。赋值有种类取决于所赋值的类型。如果是简单的值属于基本类型或者枚举类型，则你仅需简单地将该值拷贝给赋值号左边的变量即可。如果你将一个指针值赋给一个指针变量，则是该指针被拷贝赋值，而非指针所指向的值被拷贝赋值。而且，由于 C++ 将数组名看作是指向数组起始元素的指针的同义词，因此，将一个数组名赋值给一个变量，则赋值的是数组指针，即数组起始元素的地址，而非数组中的元素。如果你将一个对象赋值给另外一个对象，则赋值的语义取决于该对象所属的类中是如何定义这个赋值操作的。
5. 当程序正式动态申请内存时，将分配新的堆内存空间。仅当一个 new 操作符明确地出现在一个表达式中，C++ 程序才会在堆中创建新的内存。一旦你看到关键字 new，就需要在堆中开辟足够的空间以容纳动态分配的值。new 操作符的值为一个指向所动态申请分配的堆空间的首地址指针值，正如其他类型的指针值一样。

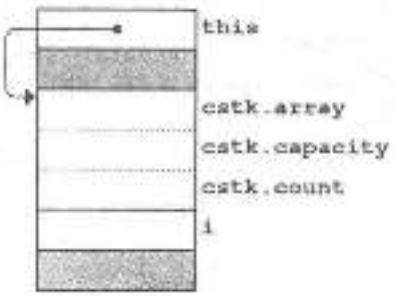
图 12-9 创建堆－栈图需遵循的步骤

对于任何 C++ 程序，发生的第一件事情就是操作系统调用主函数。在这个例子中，主函数声明了两个局部变量：CharStack 类型的 cstk 变量和 for 循环的索引变量 i。CharStack 对象要求三个字的内存：一个是动态数组的地址，另外是整型域 capacity 和 count。在初始化之前，栈帧结构如下图所示（因堆中此时还没有分配任何东西，所以图的左边现在是空的）：

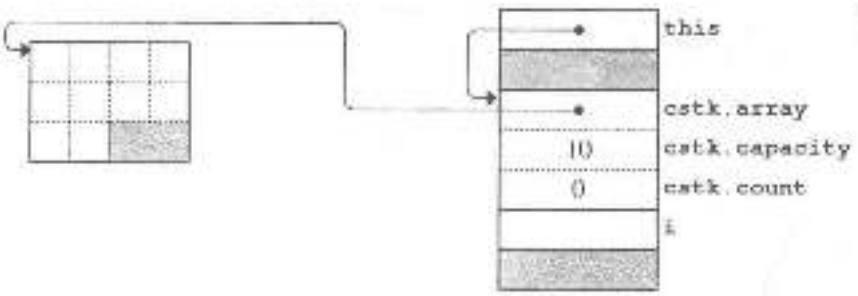
537 / 538



声明一个CharStack类的对象时，C++会自动调用CharStack类的构造函数。即使这个构造函数没有参数，也没有声明局部变量，其栈帧结构仍然包含一个被称为this的指向当前对象的系统指针，由于每一个方法调用都将this作为隐含参数，因此，在这种情况下，主程序中的this指向对象cstk，如下图所示：



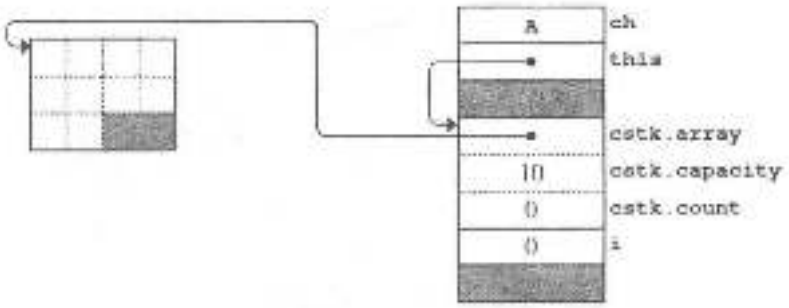
构造函数中的步骤相当易懂。在构造函数中的每一个变量都是当前对象中的一个数据域。第一行将cstk对象的容量设为常量INITIAL_CAPACITY，它的值为10。第二行分配了一个可容纳10个字符的动态数组。使用操作符new可分配任意大小的堆空间，故这个动态数组的空间被分配到堆上。在C++中，char类型占据一个字节，因此该数组在堆内存中需要10个字节，由于计算机分配内存是以机器字的整数倍来分配，因此，在堆中为数组分配了3个字（即12个字节）的空间。最后一行将cstk对象的count初始化为0，表明该栈为空。堆和栈的内容现在如下图所示：



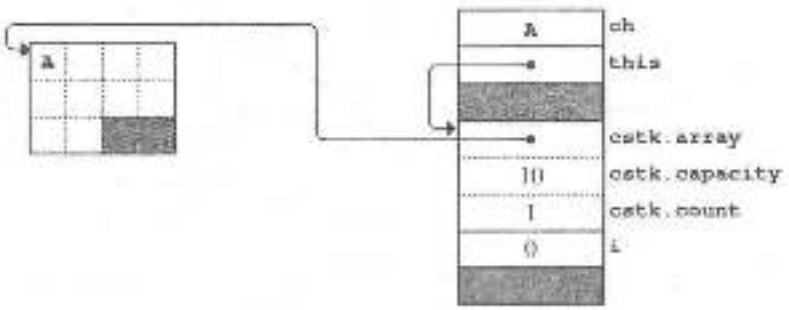
539

构造函数返回，cstk对象初始化的内容如上图所示。

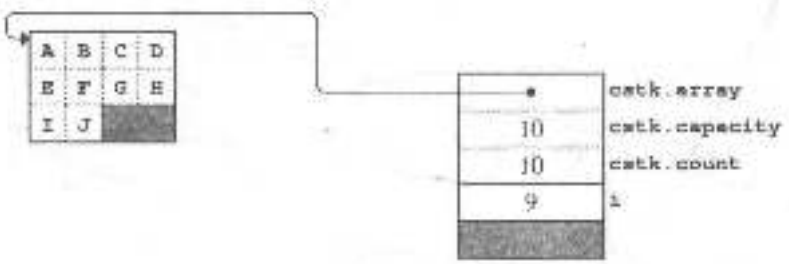
当第一次for循环开始时，即i值为0时，开始方法的调用。该循环产生了一个以参数字符'A'的cstk.push的调用。再说一次，push也是一个方法调用，因此它包含一个指针变量this及参数ch，如下图所示：



若count不等于capacity，则这个调用会执行下去。此时，字符ch被复制到动态数组中，count值会自增1，如下图所示：

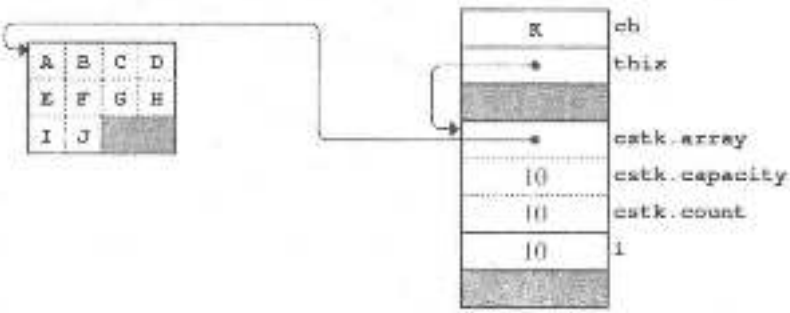


for循环接下来的几个循环都以上述同样的方式进行，在可用的栈空间中填充了如下内容：



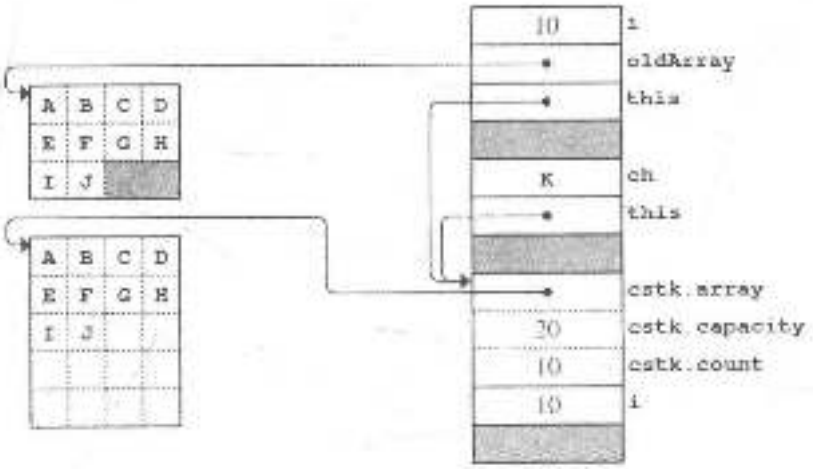
540

此时，下一个cstk.push调用将创建一个新的栈帧以准备将字符'K'压入到该栈中：



这次方法调用与之前调用的不同在于count等于capacity，它表明数组中已有的字符已

经填满了。这种情况引发了类中的一个私有方法 expandCapacity 的调用，这导致创建了另一个栈帧。expandCapacity 方法中声明了局部变量 oldArray 和 i，这意味着这些局部变量现在会出现在栈帧中，并伴随着一个指向当前对象的指针。所产生的结果栈帧显示在下图的顶部：

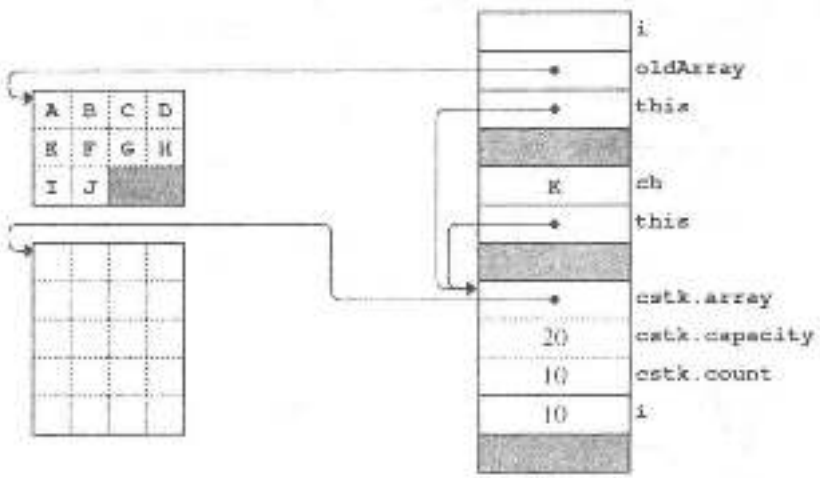


expandCapacity 方法中的操作很有趣，它使得浏览该过程的细节更有意义。以下代码行：

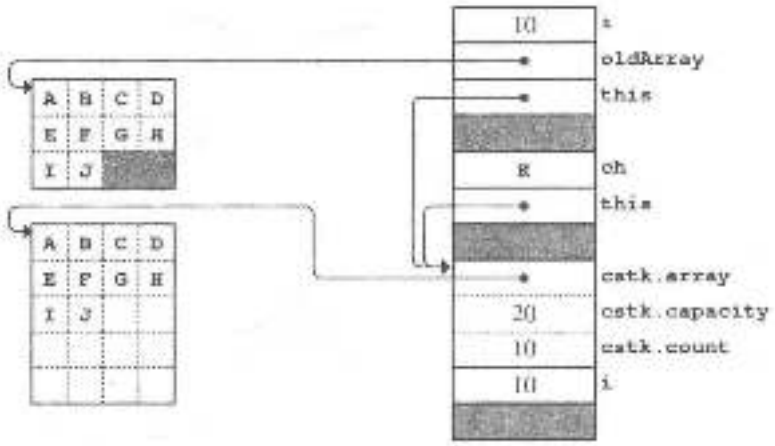
```
char *oldArray = array;
capacity *= 2;
char *array = new char[capacity];
```

341

它的功能是新分配一个为原数组两倍容量的动态数组，然后将新分配的动态数组的指针拷贝到旧的数组指针中。如下图所示：



for 循环将旧数组中字符拷贝给新的数组，使得产生了以下的堆和栈帧内容：

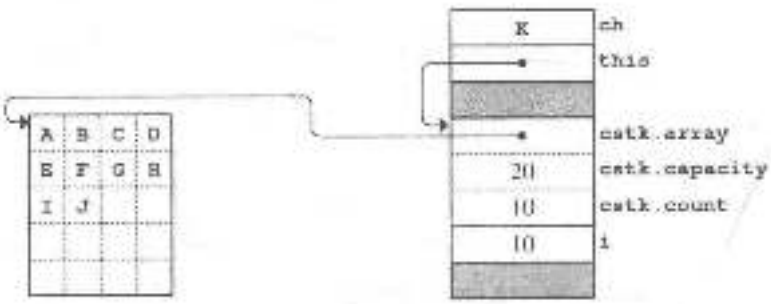


expandCapacity方法的最后一条语句是：

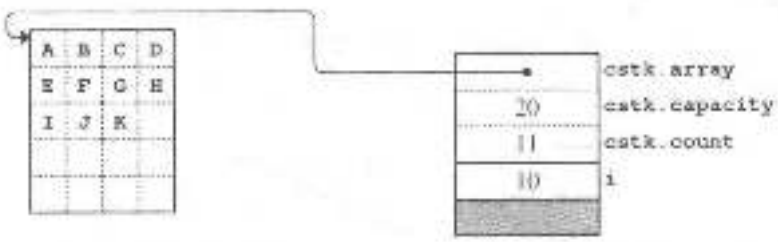
```
delete[] oldArray;
```

它释放了旧数组的内存，使得在expandCapacity返回后其堆及栈帧中的内容如下图所示：

542



既然在新的数组中还有空间，push方法就可以像它之前那样操作。push返回后的状态如下图所示：



当count变为20时，然后主程序继续通过字母表的其他字符再次对数组的容量扩充加倍。

这个例子中唯一要注意的是main函数的返回。此时，对象cstk已超出了其作用域范围，因此，它会引起~CharStack析构函数的调用。析构函数确保在字符栈的生命周期结束时所分配的堆上的动态数组内存被回收。

12.6 单元测试

上一节的主程序作为堆－栈图的说明是很有用的，但通过弹出这些字母并以逆序输出

它们来展示栈的细节，并没有真正构成对 CharStack 类的一个充分的测试。当你为用户定义一个新类时，很有必要尽可能彻底地去测试你所涉及的程序实现。未经测试的程序总会存在很多问题。作为一个实现者，你的职责就是把像 CharStack 这样的类按照其功能进行测试，检查在预期模式的近似使用的条件下接口输出的每一个方法。例如，在 CharStack 类的对象中，压入足够多的字符以确保 expandCapacity 像以前那样理想化地被调用是很重要的。

543

为每一个类或库接口开发一个独立的测试程序也是一个很好的实践。如果你设计的测试程序能正确地依赖几个类的功能，那么在运行出错的时候很难确定故障在哪儿。在与其他模块独立的情况下，分别检查每一个类或接口的策略被称为单元测试（unit testing）。图 12-10 展示了 CharStack 类的一个可能的单元测试。它包含了存入字母表中字母的代码，接着确保它们以相反的顺序出栈，还包含了对类中方法是否按它们的功能运行的检测。

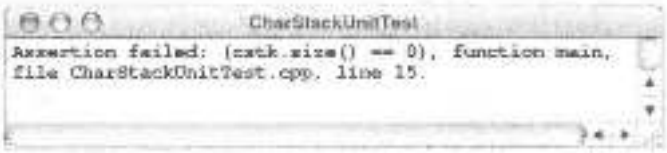
```
/*
 * File: CharStackUnitTest.cpp
 * ---------------------------
 * This file contains a unit test of the CharStack class that uses the
 * C++ assert macro to check that each operation performs as it should.
 */

#include <iostream>
#include <cassert>
#include "charstack.h"
using namespace std;

int main() {
   CharStack cstk;                          /* Declare an empty CharStack  */
   assert(cstk.size() == 0);                /* Make sure its size is 0     */
   assert(cstk.isEmpty());                  /* And that isEmpty is true    */
   cstk.push('A');                          /* Push the character 'A'      */
   assert(!cstk.isEmpty());                 /* The stack is now not empty  */
   assert(cstk.size() == 1);                /* And has size 1              */
   assert(cstk.peek() == 'A');              /* Check that peek returns 'A' */
   cstk.push('B');                          /* Push the character 'B'      */
   assert(cstk.peek() == 'B');              /* Make sure peek returns it   */
   assert(cstk.size() == 2);                /* And that the size is now 2  */
   assert(cstk.pop() == 'B');               /* Pop and test for the 'B'    */
   assert(cstk.size() == 1);                /* Recheck the size            */
   assert(cstk.peek() == 'A');              /* And make sure 'A' is on top */
   cstk.push('C');                          /* Test a push after a pop     */
   assert(cstk.size() == 2);                /* Make sure size is correct   */
   assert(cstk.pop() == 'C');               /* And that pop returns a 'C'  */
   assert(cstk.peek() == 'A');              /* The 'A' is now back on top  */
   assert(cstk.pop() == 'A');               /* Pop and test for the 'A'    */
   assert(cstk.size() == 0);                /* And make sure size is 0     */
   for (char ch = 'A'; ch <= 'Z'; ch++) {   /* Push the entire alphabet    */
      cstk.push(ch);                        /*   one character at a time   */
   }                                        /*   to test stack expansion   */
   assert(cstk.size() == 26);               /* Make sure the size is 26    */
   for (char ch = 'Z'; ch >= 'A'; ch--) {   /* Pop the characters in       */
      assert(cstk.pop() == ch);             /*   reverse order to make     */
   }                                        /*   sure they're all there    */
   assert(cstk.isEmpty());                  /* Ensure the stack is empty   */
   for (char ch = 'A'; ch <= 'Z'; ch++) {   /* Push the alphabet again to  */
      cstk.push(ch);                        /*   test that it works after  */
   }                                        /*   expansion                 */
   assert(cstk.size() == 26);               /* Check that size is again 26 */
   cstk.clear();                            /* Check the clear method      */
   assert(cstk.size() == 0);                /* And check if stack is empty */
   cstk.clear();                            /* Test clear with empty stack */
   assert(cstk.size() == 0);
   cout << "CharStack unit test succeeded" << endl;
   return 0;
}
```

图 12-10 CharStack 类的单元测试

在 CharStackUnitTest.cpp 文件中独立的测试都是用 <cassert> 库输出的 assert 机制编码的。assert 调用（它使用 C++ 的宏机制实现，宏已超出了本书的范围）具有以下形式：

```
assert(test);
```

只要测试表达式 *test* 的值为 true，则 assert 宏就不起作用。然而，如果测试结果为 false，那么 assert 宏会发出故障信息，并且以一个表示程序失效的状态码退出程序。assert 消息的格式是与系统相关的，但典型的如下图所示：



这个消息包括了测试失效的文本，它使得寻找故障的来源更加简便。

尽管测试对于软件开发非常重要，但它不能抵消细心实现代码的必要，因为用户有各种各样的使用类库的方式。已故的艾兹格·W·迪科斯彻（Edsger W. Dijkstra）在 1972 发表的论文《结构化程序设计》（*Notes on Structured Programming*）中定义了测试的重要性：

> 程序测试是说明程序最初的存在性，而不是证明程序的正确性！

作为一个实现者，你需要掌握减少故障的不同技巧。认真的设计有利于简化程序的总体结构，使得能够更加容易找到那里出错。手动跟踪你的代码（使用堆一栈图或任何有用的简略）可以在正式测试开始之前发现漏洞。大多数情况下，让其他程序员检查你的代码是发现你所忽视的问题的最好方法之一。在软件行业内，软件开发周期中的这个过程常由一系列预定的代码审查（code review）来形式化。

544
~
545

12.7 拷贝对象

正如图 12-7 和图 12-8 所示，CharStack 类的定义并没有完成。只要你通过引用传递每一个 CharStack 的对象，并且从不将 CharStack 对象的值赋给另一个对象，就不会有什么问题。但是，如果你的代码想通过值传递一个 CharStack 对象，或者试图复制一个已经存在的 CharStack 对象时，你的程序很可能会以无法预料的方式崩溃。

12.7.1 深拷贝和浅拷贝

当前对于 CharStack 类的实现的一个关键问题是：C++ 在对象之间赋值的方法往往都行得通，但是当对象包含动态分配的内存时就会出现问题。默认情况下，C++ 通过拷贝每一个对象的实例变量值来给另外一个对象赋值。如果这些值是常用的数据类型（如数字、字符及其他类似的类型），那么拷贝操作会像你想的那样做。然而，如果该值是一个指针，则拷贝该指针并不能实际地拷贝它所指的值。C++ 默认的拷贝为浅拷贝（shallow copying），即它不能实现真正的拷贝。当你拷贝一个包含动态内存的对象时，你通常是想拷贝它所指的对象中的数据。这样的拷贝被称为深拷贝（deep copying）。

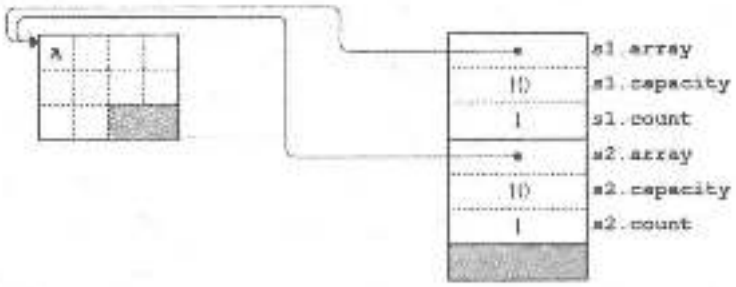
为了对这个问题的重要性有更多的理解，考虑一下，如果你执行下面的代码会发生什么：

```
CharStack s1, s2;
s1.push('A');
s2 = s1;
```

你想让这个程序做的是通过拷贝 s1 来初始化 s2，这意味着将在两个独立的栈顶都会包含字符“A”。如果 C++ 使用深拷贝来初始化 s2，这就是执行的结果。然而，如果你使用 C++ 默认的浅拷贝，执行这些语句肯定会在程序的某个时刻造成问题。

了解程序代码工作最简单的方法就是画一个堆 - 栈图，用来表示在一段语句之后的内存状况。这个图看起来如下图所示：

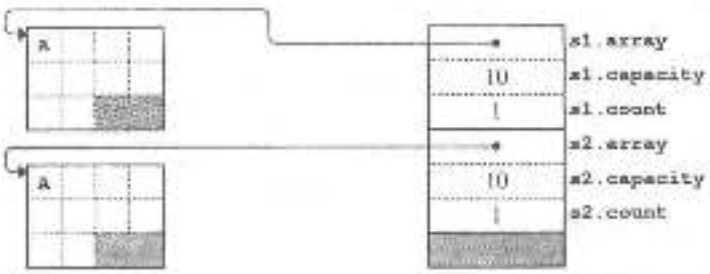
546



浅拷贝已经正确地拷贝了 count 和 capacity 的域值，但是 s1 和 s2 的 array 值一样，即它们指向同一个动态数组。如果你从栈 s2 中弹出了栈顶的字符，再压入一些其他字符，这个操作也会同样改变 s1 的内容。这不是你想要的将 s1 真正拷贝到 s2 中的结果。

更糟糕的是，当声明了变量 s1 和 s2 的函数返回时，整个程序很有可能崩溃。当此情况发生时，由于这两个变量都超出了其作用域范围，因此会引发它们各自 CharStack 析构函数的调用。析构函数的第一次调用会释放 array，第二次调用会试着做同样的事情。两次释放同一内存是非法的，但不能保证 C++ 能检测并报告这个错误。在某些机器上，第二次调用 free 会损害堆的内部结构，它会导致程序在某点失效。

如果你想要拷贝一个有别于原来的独立的 CharStack 对象，那么你就需要做一个深拷贝，如下图所示：



编写一个深拷贝的代码并不难，其挑战在于：当你把一个 CharStack 对象赋值给另一个时，让 C++ 来调用这段深拷贝的代码。完成这个任务的编程模式会在下一节描述。

547

12.7.2 赋值和拷贝构造函数

在 C++ 中，你可以通过定义两种方法来改变默认的浅拷贝行为。一种方法是过载赋值操作符。第二种方法是定义一种构造函数的特殊形式，该构造函数被称为拷贝构造函数（copy constructor），用于从一个已存在的同类对象复制并初始化另一个对象。在 C++ 中，仅当你用一个已经存在的对象进行赋值时，拷贝构造函数才会被调用。当一个对象首次被初始化时（包括一个带有初始化器的声明时），C++ 也会调用拷贝构造函数。

默认情况下，赋值操作符和无代码实现的拷贝构造函数都会像上一节所描述的那样创建一个浅拷贝。如果你想要你的类支持深拷贝，就需要为这两个方法提供一个新的定义，使它

们能拷贝动态分配的数据。

在C++中，如果你尝试东拼西凑写出赋值操作和拷贝构造函数，那么其中大量的细节很容易混乱。大多数情况下，最好的办法是从标准模式中拷贝这些类函数，接着做出必要的修改以支持你的类。图12-11表明了实现CharStack类的标准技术。这段代码重新定义了拷贝构造函数和赋值操作符，它们的实现是将大部分工作委托给了deepCopy私有方法。

```
/*
 * Implementation notes: copy constructor and assignment operator
 * ---------------------------------------------------------------
 * These methods make it possible to pass a CharStack by value or
 * assign one CharStack to another.  The actual work is done by the
 * private deepCopy method, which represents a useful pattern
 * for designing other classes that need to implement deep copying.
 */

CharStack::CharStack(const CharStack & src) {
   deepCopy(src);
}

CharStack & CharStack::operator=(const CharStack & src) {
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}

/*
 * Implementation notes: deepCopy
 * ------------------------------
 * This method copies the data from the src parameter into the current
 * object.  All dynamic memory is reallocated to create a "deep copy"
 * in which the current object and the source object are independent.
 */

void CharStack::deepCopy(const CharStack & src) {
   array = new char[src.count];
   for (int i = 0; i < src.count; i++) {
      array[i] = src.array[i];
   }
   count = src.count;
   capacity = src.capacity;
}
```

图12-11 CharStack类深拷贝的实现

当你查看图12-11所示的代码时，很可能注意到的第一件事情就是三个方法的参数声明都包含了关键字const，就像deepCopy方法的头文件说明一样：

```
void deepCopy(const CharStack & src);
```

const关键字保证了即使deepCopy通过引用来传递也不会改变src的值。这种参数传递方式被称为常量引用调用（constant call by reference）。常量引用调用的细节将在本章的稍后部分介绍。

拷贝构造函数只是简单地调用deepCopy方法，将原来CharStack类的对象内部数据拷贝给当前对象的内部数据中。更有趣的方法是重载赋值操作符，如以下代码所示：

```
CharStack & operator=(const CharStack & src) {
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}
```

上述操作符重载代码开始处的 if 语句用于检查赋值号的左边和右边是否实际上指向了同一对象。这个测试是重载赋值操作的标准模式中的一个关键部分。这个测试的目的不只是消除不必要的拷贝操作。没有这个测试，在调用 DeepCopy 拷贝数据到目的栈之前，CharStack 对象进行的自身拷贝并不会释放其所占用的数组空间。重载赋值操作符的最后一行代码也须特别注意。在 C++ 中，赋值操作符被定义为返回它的左值。关键字 this 指向当前对象的指针，因此 *this 表达式代表了对象本身。 548 / 549

如果觉得拷贝构造函数和重载赋值操作符的过程令人困扰，你还有另外一种选择。图 12-12 中以更简单的模式定义了私有的拷贝构造函数和重载赋值操作符。这些定义重置了 C++ 提供的默认构造函数。同时，出现在私有部分的这些类中的方法事实上对用户是不可见，这能有效防止用户在任何环境下拷贝 CharStack 对象。例如，C++ 的标准库用这种方法以避免拷贝流。

然而，理解这些复杂的事情并不像知道如何将这些模式融入你自己设计的类那么重要。如果你动态分配一块内存作为类的一部分，那么有责任重新定义拷贝构造函数和赋值操作符，除非重载拷贝构造函数和赋值操作符，否则编译器会自动提供可能会进行误操作的这些方法。因此，对于你设计的每个类，可以从上述策略中选择其一（不论是执行深拷贝或完全禁止拷贝）。你也可以接着使用图 12-11 或图 12-12 中的代码作为模型，用来编写你的类的必要部分。

```
private:
/*
 * Standard methods: copy constructor and assignment operator
 * ----------------------------------------------------------
 * The following lines make it illegal to copy a CharStack, by defining
 * private versions of the copy constructor and assignment operator.
 */

   CharStack(const CharStack & src) { }
   CharStack & operator=(const CharStack & src) { return *this; }
```

图 12-12 使拷贝不合规定的必要定义

12.8 关键字 const 的使用

到本章为止，唯一一次你遇到 const 关键字的地方是在常量值的定义中，如第 1 章中的这个例子：

```
const double PI = 3.14159265358979323846;
```

上一节在常量引用传递的例子中介绍了 const 的另一种新应用。可是专业的 C++ 程序员也在其他一些环境中使用 const。很遗憾的是，正确地使用 const 很困难，特别是对于初学者。如果你打算成为一名专业的 C++ 程序员，你必须掌握很多 const 关键字的微妙之处。 550
为了帮助你接近这个目标，下一节将概述 const 最普遍使用和你可能会遇到的某些陷阱。

12.8.1 常量定义

从第 1 章开始，你就已经使用 const 关键字来命名一个常量值。在第 2 章，你学会了如何从库接口中输出常量，即在 .h 和 .cpp 文件的声明中增加 extern 关键字。你需要理解的唯一的其他微妙之处是：定义常量值时，你需要给声明的常量加上关键字 static，以此作为类的一部分。例如，在 charstack.h 文件中，常量 INITIAL_CAPACITY 在私有部分是这样被声明的：

```
static const int INITIAL_CAPACITY = 10;
```

这个声明包含的 static 确保了 CharStack 类的所有对象共享一个 INITIAL_CAPACITY 常量的拷贝，而不是 CharStack 类的每个对象都有这个常量值。

12.8.2 常量引用调用

本章之前，仅通过使用值调用和引用调用这些基本形式就会使简化参数传递有意义。然而，为了实现深拷贝，就像 12.6 节说明的一样，C++ 编译器要求对拷贝构造函数和重载赋值操作符函数使用常量引用调用。但是常量引用调用绝非仅限于这些方法。事实上，如果你正在传递一个对象，那么常量引用调用通常优于传统的引用调用和值调用。在很多方面，常量引用调用结合了每种参数传递的传统模式的优点，它提供了引用传递的高效性和值传递的安全性。

尽管 C++ 的语法表明常量引用调用似乎比你以前看到的例子简单，但如果你试图将它应用到一个指针参数时，就会变得比较有技巧性。例如，考虑以下函数原型：

```
int strlen(const char *cptr);
```

它是一个重要的库函数，用于返回 C 风格的字符串的长度。如果你类推到常量引用调用的早期例子，可能以为这个原型中声明的形参 cptr 是一个指向字符的常量指针。然而 C++ 解释略微不同。由于此时 const 后是类型名，意味着所声明的这个 cptr 是一个指向 const char 的指针。基于这样的解释，它能完全改变 strlen 函数内的 cptr 的值，而不能改变它所指向的字符串的内容。若想防止改变 cptr 变量自身的值，则需要将 const 放在星号的后面。

[551]

用常量引用调用作为通用的替换值调用方式所存在的唯一问题就是：在参数前加入 const 关键字以使 C++ 编译器保证这个值不被接受参数的函数所修改。通常，让编译器执行检测是一件好事。可是，为此编译器必须要确定类中哪些方法可以修改对象，哪些方法不能修改对象。在 C++ 中，负责提供这些信息职责的是程序员。使用常量引用调用的代价是类的设计者必须提供更多的关于类中定义方法的信息，正如下节所描述的一样。

12.8.3 const 方法

在一个典型的类定义中，某些方法改变了对象的值，而另一些方法并不改变对象的值。例如，在 CharStack 类这个例子中，push、pop 和 clear 这些方法都改变了其对象栈中的内容。相反，size、isEmpty 和 peek 这些方法仅返回对象的值。C++ 允许程序员通过在方法参数表后面增加关键字 const 来指定一个方法，它不改变其对象的状态。例如，在 charstack.h 文件中的 size 方法原型应该用关键字 const 约束，以通知编译器该方法不能改变对象的状态，如下所示：

```
int size() const;
```

const 关键字在方法的实现中也必须出现，如下所示：

```
int CharStack::size() const {
   return count;
}
```

如果一个类使用关键字 const 来指明参数不能发生改变，其方法不能改变对象的内容，则这个类就被称为常量正确的（const correct）。在 STL 和 Stanford 类库中的类都是常量正

确的。编写常量正确的类需要更多努力，但却能够写出尚较易读的代码。图 12-13 和图 12-14 展示了一个常量正确的 CharStack 类。 552

```
/*
 * File: charstack.h
 * -----------------
 * This interface defines the CharStack class, which implements
 * the stack abstraction for characters.
 */

#ifndef _charstack_h
#define _charstack_h

/*
 * Class: CharStack
 * ----------------
 * This class models a stack of characters.  The fundamental operations
 * are the same as those for the Stack<char> class.
 */

class CharStack {

public:

/*
 * Constructor: CharStack
 * Usage: CharStack cstk;
 * ----------------------
 * Initializes a new empty stack that can contain characters.
 */

   CharStack();

/*
 * Destructor: ~CharStack
 * Usage: (usually implicit)
 * -------------------------
 * Frees any heap storage associated with this character stack.
 */

   ~CharStack();

/*
 * Method: size
 * Usage: int nElems = cstk.size();
 * --------------------------------
 * Returns the number of characters in this stack.
 */

   int size() const;

/*
 * Method: isEmpty
 * Usage: if (cstk.isEmpty())
 * --------------------------
 * Returns true if this stack contains no characters.
 */

   bool isEmpty() const;

/*
 * Method: clear
 * Usage: cstk.clear();
 * --------------------
 * Removes all characters from this stack.
 */

   void clear();

/*
 * Method: push
 * Usage: cstk.push(ch);
 * ---------------------
 * Pushes the character ch onto this stack.
 */
```

图 12-13 charstack.h 接口的常量正确版本

```
   void push(char ch);

/*
 * Method: pop
 * Usage: char ch = cstk.pop();
 * ----------------------------
 * Removes the top character from this stack and returns it.
 */

   char pop();

/*
 * Method: peek
 * Usage: char ch = cstk.peek();
 * -----------------------------
 * Returns the value of the top character from this stack without
 * removing it.  Raises an error if called on an empty stack.
 */

   char peek() const;

/*
 * Copy constructor: CharStack
 * Usage: (usually implicit)
 * -------------------------
 * Initializes the current object to be a deep copy of the specified source.
 */

   CharStack(const CharStack & src);

/*
 * Operator: =
 * Usage: dst = src;
 * -----------------
 * Assigns src to dst so that the two stacks are independent copies.
 */

   CharStack & operator=(const CharStack & src);

/* Private section */

private:

/* Private constants */

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   char *array;           /* Dynamic array of characters   */
   int capacity;          /* Allocated size of that array  */
   int count;             /* Current count of chars pushed */

/* Private method prototypes */

   void deepCopy(const CharStack & src);
   void expandCapacity();

};

#endif
```

图 12-13 （续）

```
/*
 * File: charstack.cpp
 * -------------------
 * This file implements the CharStack class.
 */

#include "charstack.h"
#include "error.h"
using namespace std;
```

图 12-14 charstack.cpp 实现的常量正确版本

```
/*
 * Implementation notes: constructor and destructor
 * ------------------------------------------------
 * The constructor allocates the array storage for the stack elements and
 * initializes the fields of the object.  The destructor frees any heap
 * memory allocated by the class, which is just the array of elements.
 */

CharStack::CharStack() {
   capacity = INITIAL_CAPACITY;
   array = new char[capacity];
   count = 0;
}

CharStack::~CharStack() {
   delete[] array;
}

/*
 * Implementation notes: size, isEmpty, clear
 * ------------------------------------------
 * These methods are each a single line and need no detailed documentation.
 * Note that size and isEmpty leave the stack unchanged and are therefore
 * marked as const.
 */

int CharStack::size() const {
   return count;
}

bool CharStack::isEmpty() const {
   return count == 0;
}

void CharStack::clear() {
   count = 0;
}

/*
 * Implementation notes: push
 * --------------------------
 * This function first checks to see whether there is enough room for
 * the character and then expands the array storage if necessary.
 */

void CharStack::push(char ch) {
   if (count == capacity) expandCapacity();
   array[count++] = ch;
}

/*
 * Implementation notes: pop, peek
 * -------------------------------
 * These functions check for an empty stack and report an error if
 * there is no top element.
 */

char CharStack::pop() {
   if (isEmpty()) error("pop: Attempting to pop an empty stack");
   return array[--count];
}

char CharStack::peek() const {
   if (isEmpty()) error("peek: Attempting to peek at an empty stack");
   return array[count - 1];
}

/*
 * Implementation notes: copy constructor and assignment operator
 * ---------------------------------------------------------------
 * These methods make it possible to pass a CharStack by value or
 * assign one CharStack to another.  The actual work is done by the
 * private deepCopy method, which represents a useful pattern
 * for designing other classes that need to implement deep copying.
 */
```

图 12-14 （续）

```
CharStack::CharStack(const CharStack & src) {
   deepCopy(src);
}

CharStack & CharStack::operator=(const CharStack & src) {
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}

/*
 * Implementation notes: deepCopy
 * ------------------------------
 * This method copies the data from the src parameter into the current
 * object.  All dynamic memory is reallocated to create a "deep copy"
 * in which the current object and the source object are independent.
 */

void CharStack::deepCopy(const CharStack & src) {
   array = new char[src.count];
   for (int i = 0; i < src.count; i++) {
      array[i] = src.array[i];
   }
   count = src.count;
   capacity = src.capacity;
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This method doubles the capacity of the elements array whenever it runs
 * out of space.  To do so, the method must copy the pointer to the old
 * array, allocate a new array with twice the capacity, copy the characters
 * from the old array to the new one, and finally free the old storage.
 */

void CharStack::expandCapacity() {
   char *oldArray = array;
   capacity *= 2;
   array = new char[capacity];
   for (int i = 0; i < count; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}
```

图 12-14 （续）

12.9 CharStack 类的效率

第 13 章用 CharStack 类作为其中一种可能的实现策略来创建文本编辑器。由于下一章的主要目标是评估这些不同策略的相对效率，因此就需要对 CharStack 类本身的效率有一个理解。从第 10 章中就已知道，一个算法的效率通常以它的时间复杂度表示，它测试随着一个函数问题规模的变化，该函数的运行时间如何变化。

556
|
558

对于 CharStack 类，其中大部分方法在栈当前大小情况下是以常量时间运行的。事实上，类内仅有一个方法会改变其栈的大小。通常，push 方法仅将一个字符压入到其数组的下个空槽中，这仅需要一个常量时间。然而，若这个数组已满，则 expandCapacity 方法就不得不复制数组的内容给新分配的内存，这会使得随着栈的增大，程序会运行得越来越慢。调用 expandCapacity 要求线性时间，它表明最坏情况下 push 方法的时间复杂度为 $O(N)$。

至此，复杂度分析集中于：在最坏情况下，一个特定的算法是如何执行的。然而，这里有一个重要的特征使得对 push 操作的复杂度分析不同于传统的对其他操作的复杂度分

析：最坏情况不可能每次都发生，特别是，如果压栈触发了一个expandCapacity方法的调用，而该调用以$O(N)$的时间运行，则push操作压入下一项的花费保证为$O(1)$，因为栈的容量已经被扩展了。因此在整个push操作中，有必要平均分配这个栈扩展的时间代价。这种复杂度度量类型被称为分步分析（amortized analysis）。

为了使这个过程更易于理解，计算push操作重复N次所花费的全部时间是很有用的，这里N可以是某个较大值，不管栈的容量是否被扩充，每一个push操作都会花费一些时间。如果你用希腊字母α表示固定成本，那么压入N项的总成本为αN。然而，push操作常常会要扩充栈中数组的容量，这是一个花费正比时间的线性操作（用希腊字母β表示），以栈中字符数量的点数。

关于N次push操作的总运行时间，最坏情况是当循环后测时才要求栈扩充容量。此时，最后的push操作会导致βN的额外开销。假定每次expandCapacity方法的调用都将使底向数组容量翻一番，当栈的容量为N的一半、N的1/4，等等时，也必须被扩展，则push操作N次的总代价为下式公式所示：

$$\text{总时间} = \alpha N + \beta\left(N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \cdots\right)$$

平均时间就是总时间除以N，如下式所示：

$$\text{平均时间} = \alpha + \beta\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)$$

559

尽管圆括号内的数字之和取决于N，但其和绝不可能超过2，这意味着平均时间以常数$\alpha + 2\beta$为界，它的时间复杂度为$O(1)$。

本章小结

本书的目标之一就是鼓励你使用高级数据结构，它允许你以独立于底层表示的抽象方式来思考数据。抽象数据类型和类的使用使得我们能坚持这一观点。同时，高效地使用C++要求你在头脑中有一个数据结构在内存中是如何表示的模型。本章，你有机会去了解这些结构是如何被存储的，并获得你编写程序时对"处于底层"的认识。

本章的重点如下：

- C++使用new操作符和一个类型名从堆中分配内存，new操作返回一个指向所分配的这块足以存放该类型值的内存块的首指针。
- 基本类型、对象和结构都可以通过在类型名前添写new操作符从而在堆中被分配内存，如以下语句所示：

```
int *ip = new int;
```

  在动态分配对象的情况下，你也可以在类型名之后指定其构造函数的参数，如以下语句所示：

```
Rational *pointerToOneHalf = new Rational(1, 2);
```

- 指针经常被用以指明一个数据结构中各个元素之间的关系。该技术一个特别重要的应用就是链表，在链表中的指针形成了一个线性链。链表中的每一个元素被称为一个结点（cell）。链表中的最后一个结点包含了一个NULL空指针，它标识着链表的结束。
- 采用new操作符，并在类型名后的方括号内指明所期望分配的该类型量的个数，便

可分配一个所期望容量的动态数组的内存空间，就像以下声明语句，它分配了一个可容纳10个字符大小的动态数组：

```
char *cp = new char[10];
```

- 不像大部分现代程序设计语言，C++要求程序员负责内存管理。程序员所面临的最大挑战就是释放程序所分配的任意堆内存。在底层中，C++使用delete操作符释放单一的堆空间，delete[]操作符释放动态分配的一块内存空间。
- C++的内存管理任务通过析构函数得到了相当大的简化。当前栈帧所包含的一个对象在调用方法返回时应消失，为此，此时系统会自动调用对象所属类的析构函数。析构函数的主要任务是释放对象所分配的任意堆内存。
- 析构函数名是类名前加一个波浪符~。每一个类只能有一个无参的析构函数。
- 正如本章所述的CharStack类，可以使用数组来实现可动态扩展其容量的抽象数据类型。
- 堆-栈图有助于理解C++是如何分配内存的。每一个函数或方法调用都会创建一个栈帧，包含了自身声明的局部变量只有程序执行new操作才会把内存分配到堆上。仅当程序执行delete类作时其所分配的堆内存才被回收。当一个函数返回时，栈内空间被自动释放。
- 无论何时你实现一个供他人使用的类，你都有责任尽可能彻底地测试它。一个有效的技术就是编写一个能自动测试类中每一个方法的测试程序，它应独立于一个应用中的其他模块。这样的测试程序被称为单元测试，它是好的软件工程实践的一个重要部分。
- 当你将一个对象赋值给另一个对象，或者将其值传递给另一个对象时，C++默认的规则是进行对象的浅拷贝。当一个对象包含动态分配的内存时，通常需要做深拷贝，深拷贝会将指针所指向的对象进行拷贝。
- 你可以通过重载赋值操作符和拷贝构造函数来改变C++拷贝特定类的对象拷贝方法。正确地定义这些方法的规则是十分精妙的，你应该从一个已存在的类中拷贝其基本结构，然后再修改必要的代码。
- 你可以通过在类的私有部分重载赋值操作符和拷贝构造函数来共同禁止对象拷贝。
- const关键字在C++程序中有不同的应用。除了定义常量值，你可以使用const指定一个方法，它不能改变特定的参数值或者是对象的值。
- 即使CharStack类中的push方法有时要求以$O(N)$的时间来扩展动态数组的容量，但这个时间代价不会连续多次发生。事实上，这个时间代价被分配给方法的每一次调用中。使用这种分步分析的方法，CharStack类中的每个方法都以$O(1)$时间运行。

复习题

1. 本章所描述的三种内存分配机制是什么？
2. 什么是栈？
3. 为什么堆和栈在内存的末端以相反的方向扩展？
4. 为了创建和初始化下面的变量，你应该使用什么声明？
   a) 一个指向布尔值的bp指针。

b) 一个名为 pp 的指针变量，指向坐标为 (3, 4) 的一个 Point 对象。

c) 能容纳 100 个 C++ 字符串的动态数组 names。

5. 你应该使用什么语句来释放在前述习题中分配的内存？

6. 定义在数据结构中使用的链表中的术语结点和链接。

7. 什么是标记链表结束的标准方法？

8. 什么样的数据结构可使你定义整型链表？

9. 假设你已有了一个整型链表的定义，应该如何编写 for 循环，以单步调试链表 list 中的每一个元素？

10. 什么是内存泄漏？

11. 判断题：C++ 使用垃圾回收来管理内存。

12. 什么是析构函数？它最重要的角色是什么？

13. 如果你创建一个名为 IntArray 的类，如何编写它的析构函数原型？

14. 变量越界意味着什么？

15. 判断题：析构函数甚至可以被没有赋给局部变量的临时变量调用。

16. 如何使动态扩展 CharStack 类的容量成为可能，即使它使用的数组大小在分配时已经确定？ 562

17. 描述 CharStack 类中每一个实例变量的目的。

18. 解释图 12-8 中 expandCapacity 类实现中的每一条语句。

19. 假设，expandCapacity 类只需要给其数组增加一个元素，而不是将数组的容量翻一番。push 方法的平均时间复杂度仍是 $O(1)$ 吗？为什么是或者不是？

20. 新的内存什么时候被加到堆－栈图中栈的那一边？内存什么时候被回收？

21. 新的内存什么时候被加到堆－栈图中堆的那一边？内存什么时候被回收？

22. 本章在堆－栈图中给出开销 (overhead) 这一术语的原因是什么？

23. 在堆－栈图中，你如何表示引用参数？

24. 当你调用类的一个方法而不是调用一个普通函数时，哪些额外的信息会加入到当前的栈帧结构中？

25. 在术语单元测试中单元确指什么？

26. 浅拷贝和深拷贝有何不同？C++ 默认使用这两种策略中的哪一个？

27. 为了改变 C++ 拷贝对象的方式，你必须重载哪些方法？

28. 常量引用说明为什么与大部分之前所熟悉的值调用和引用调用的参数传递方式不同？

29. 对类而言，常量正确的意味着什么？

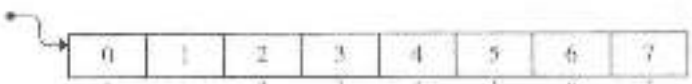
30. push 操作的分步复杂度是 $O(1)$ 的论据依赖于这一系列数字之和：

$$1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots$$

无论你包含多少步，和不会超过 2。试着用你自己的话解释为什么。(如果遇到困难，可以试着在网上查找芝诺悖论 (Zeno's Paradox)，然后基于此理论给出你的解释。) 563

习题

1. 编写一个 createIndexArray() 函数，它动态分配可存储 n 个整数的数组，数组中的每个元素值被初始化为其下标值。例如，调用 createIndexArray(8) 应该返回一个指向堆中数组的指针，如下图所示：



2. 在 11.4.3 小节的 strcpy 定义中，用粘脸头图标标记就是为了提醒你这个函数多么危险。事实上，这个危险项就是 strcpy 没有检查字符数组收到拷贝后是否还有有效的存储空间。因此它增加了数组越界故障的概率。然而，如果可能的话，采用动态分配以创建用于拷贝字符串所需的内存空间以

避免上述危险。

编写一个函数：

```
char *copyCString(char *str);
```

它可以为 C 风格的 str 字符串分配足够大的内存，并且将结束字符在内的所有字符复制到新分配的内存中。

3. 在图 12-3 创建的灯塔程序中，灯塔的名字被引诱地列在 CreateBeaconsOfGondor 函数中。一种更灵活的方式是从数据文件中读取灯塔的名字。修改 BeaconsOfGondor 程序，以便在主程序的第一条语句中调用以下函数：

```
Tower *readBeaconsFromFile(string filename);
```

该函数从指定的文件中读取灯塔列表。例如，如果文件 BeaconsOfGondor.txt 包含下图所示的灯塔列表名，则这个程序会像本章之前描述的那样运行。

```
Minas Tirith
Amon Din
Eilenach
Nardol
Erelas
Min-Rimmon
Calenhad
Halifirien
Rohan
```

564

4. 设计和实现一个名为 IntArray 的类，并实现以下类中的方法：
   - 创建一个包含 n 个元素对象的构造函数 IntArray(n)，并且将每个元素都初始化为 0。
   - 释放由 IntArray 类所分配的任意堆内存的析构函数。
   - 一个返回 IntArray 中元素个数的方法 size()。
   - 一个返回下标 k 的元素方法 get(k)。如果 k 已在动态数组中越界，则 get 方法会调用 error 方法，并产生相应错误提示消息。
   - 一个给下标为 k 的元素赋值的方法 put(k,value)。和 get 方法一样，若 k 越界，则方法 put 就会调用 error 方法。

   你的解决办法应该用与本章 CharStack 例子相似的方式，分开接口和实现文件。在初始的代码版本中，你应该给 intarray.h 文件增加必要的定义，以防止用户拷贝 IntArray 对象。设计和实现一个单元测试以检测类的方法。

   通过跟踪数组的大小即检查数组边界内的数组索引值，这个简单的 IntArray 类已经解决了内置数组类型的两个最严重的缺陷。

5. 你可以通过重载括号选择操作符，会使前面习题中的 IntArray 类有点像普通的数组。该重载函数具有以下原型：

```
int & operator[](int k);
```

   类似 get 和 put 方法，你的 operator[] 实现应该进行越界检查，以确保索引 k 是有效的。若 k 有效，operator[] 方法应该返回 k 所指的元素，便于用户可以将一个新值放入到下标为 k 的数组元素位置上。

6. 为出现在习题 4 和习题 5 中的 IntArray 实现深拷贝。
7. 假设你有一个包含图 12-15 中代码的文件。在 initFair 函数返回之前，画一个表示内存内容的堆—栈图。做一个额外的练习，画这个图的两个版本：一个用明确的地址；另一个用箭头表示指针。
8. 就像前面的习题，图 12-16 画了一个表示内存现状的堆—栈图，它要求你画出第二次调用构造函数期间的内存变化。

565

```
struct Domino {
   int leftDots;
   int rightDots;
};

void initPair(Domino list[], Domino & dom);

int main() {
   Domino onetwo;
   onetwo.leftDots = 1;
   onetwo.rightDots = 2;
   Domino *array = new Domino[2];
   initPair(array, onetwo);
   return 0;
}

void initPair(Domino list[], Domino & dom) {
   list[0] = dom;
   list[1].leftDots = dom.rightDots;
   list[1].rightDots = dom.leftDots;
   dom = list[1];  <- Diagram memory at this point in the execution
}
```

图 12-15 习题 7 中使用的多米诺骨牌程序的代码

```
class Student {
public:

   Student() {
      id = 0;
      gpa = 4.0;
   }

   Student(int id, double gpa) {
      this->id = id;
      this->gpa = gpa;  <- Diagram at this point on the second call to the constructor
   }

private:

   int id;
   double gpa;

};

int main() {
   Student *advisees = new Student[2];
   advisees[0] = Student(2718281, 3.61);
   advisees[1] = Student(3141592, 4.2);
   return 0;
}
```

图 12-16 习题 8 中使用 Student 类的代码 566

9. 尽管程序员倾向于把字符串看作为相对简单的实体，但它们的实现涉及了本章中你所见过的所有方法。在本题中，你的任务是定义一个叫做 MyString 的类，它近似于标准 C++ 类库中的 string 类。你的类应具有以下公有方法：

- 构造函数 MyString(str) 从 C++ 的 string 类中创建了一个 MyString 对象。
- 一个释放 MyString 分配的任一堆空间的析构函数。
- 一个将 MyString 类型转变为 C++ 中的 string 类型的 toString() 方法。
- 一个返回字符串中字符数量的方法 length()。
- 一个返回当前字符串对象的子串的方法 substr(start,n)。无论哪种情况先发生，它的功能和 string 类的库版本一样，子串要么是以下标 start 处开始的 n 个字符，或者是到字符串末尾的字符串。参数 n 应该是可选的；如果没有参数 n，子串应该总是从 start 至字符串的尾端。
- 重新定义 + 操作符以连接两个 MyString 对象。重载操作符 += 也很有意义，它将一个字符或字符串附加到另一个字符串的后面。

- 重新定义 << 操作符以将 MyString 类的对象写到输出流中。
- 重新定义括号选择操作符（与习题 5 中描述的一样），通过引用将 str 中下标 i 的字符返回。随着 C++ 类库中 string 类的完善，如果在下标选择符的实现中出现了下标越界，那么应该调用 error 方法。
- 重新定义关系操作符 ==、!=、<、<=、> 和 >=，用于比较字符串的字母顺序。
- 为 MyString 类重新定义一个赋值操作符和拷贝构造函数，以便任何拷贝操作都能对一个新创建字符数组的实行深拷贝。

你的代码应该采用你自己的内在表示，而不能调用 C++ 语言 string 类中的任何方法。你的接口和实现也应该是常量正确的，以便用户和 C++ 编译器都能确切地知道哪些方法能修改字符串的
367 内容。

10. 习题 9 作为 MyString 类的初步测试，重写 3.6 节图 3-2 中的儿童黑话程序，以便它使用 MyString 类代替 string 类。
11. 前面习题中的儿童黑话程序不是 MyString 类的充分测试。设计和实现一个 MyString 类的更彻底的单元测试。
12. 为 6.3 节介绍的 Rational 类编写一个单元测试。如果你实现了第 6 章习题 8 中描述的 Rational 类的扩展版本，那你应该在你的单元测试中也应包含对这些扩展的测试。
568 13. 重写 rational.h 和 rational.cpp 文件，使得 Rational 类是常量正确的。

效率和表示

完成一件事所允许的时间与可充分利用的时间并非一定吻合。

——蕾切尔·卡森，《沉默》(*Silences*)，1965

569

本章要将数据结构的设计以及算法效率这两个看似毫不相关的概念联系到一起。迄今为止，对效率的讨论主要聚焦于算法。如果选择一个更高效的算法，你可以大大减少程序的运行时间，尤其是在较为复杂的类中使用新的算法。然而，在某些例子中，为类选择一个不同的底层表示可达到同样奇妙的效果。为了阐述这一观点，本章将展示一个特定类的几种不同表示，然后对比这些表示的效率。

13.1 编辑文本的软件模式

在这个手机广泛使用的时代，文本已经变成了最主流的交流形式。你可以在手机按键上编辑信息并发送给你的一个或多个好友，然后他们就可以在自己的手机上阅读信息。现在的手机需要大量的软件，具有代表性的是那些几百万行代码的软件。为了管理这种级别的程序复杂性，将软件的实现分为几个可以独立开发和管理的模块是非常必要的。此外，这对于使用已经完备的软件模式来简化实现的过程也是很有用的。

为了对模式的有效性有一个认知，可以考虑我们平时在手机上发信息时会发生什么事情。首先你要在手机键盘上输入你要输入的文字，当然也包括编辑键。根据不同的手机类型，手机键盘也可以分为不同形式。在较老式的手机上，键盘主要由一系列的数字键组成。在这些数字键上，可以通过按同一键不同次数来选择你所需要的字母。智能手机可能就没有所谓的物理学意义上的键盘了，它们用触摸屏上的一些虚拟的键代替了键盘。在任何一种情况下，都存在一种概念意义上的键盘允许你编辑信息。手机还有一个显示屏，允许我们查看所写入的信息。然而，在任何现代设计中，还有一个对作为用户的你来说不可见的第三个构件。在手机键盘和显示屏之间有一个抽象的数据结构，我们用它来记录当前的信息内容。手机键盘会更新这个数据结构的内容，其内容会在显示屏上显示出来。

上一段讲的三部件分解方法是一个很重要的设计策略实例。这个分解策略称为**模型－视图－控制器**（model-view-controller，MVC）模式。在手机这个例子中，键盘代表控制器，显示屏代表视图，底层的数据结构代表模型。这种手机示例的实现如图 13-1 所示，这个图跟踪了不同模块之间的信息流向。

570

如图 13-1 所示，当你用手机发送短信时，你正在使用**编辑器**（editor）。编辑器是一个支持创建和修改文本数据的软件模块。在很多应用中，我们都要用到编辑器。当你需要在一个基于 Web 的表单中输入信息或者在你的开发环境中创建一个 C++ 程序时，你就在使用一个编辑器。现在大多数编辑器都采用模型－视图－控制器模式。在模型内部，一个编辑器可以包含一系列的字符，这通常称为**缓冲区**（buffer）。控制器允许用户在缓冲区的内容里进行各种操作，其中很多操作仅限于缓冲区的当前位置。这个位置在显示屏上用一个**光标**（cursor）符号表示，它主要在两个字中间以垂直线的形式出现。

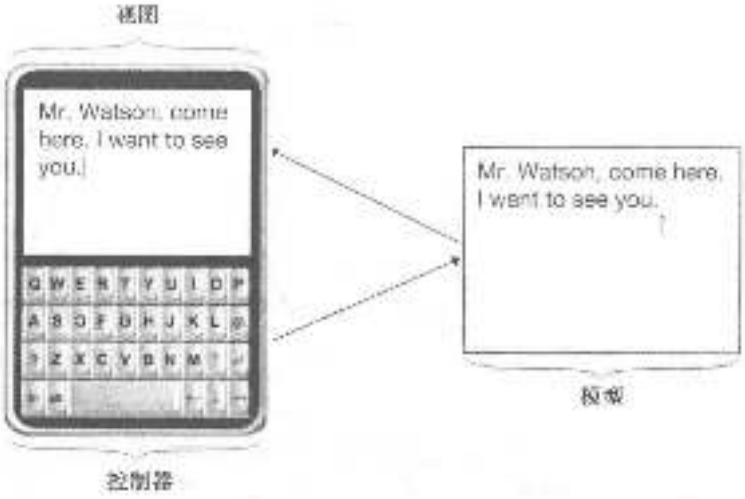


图 13-1 使用模型 – 视图 – 控制器模式分解的手机

显然编辑器应用的控制器和视图构件面临比较有趣的编程挑战，但本章重点研究构成模型的编辑器缓冲区。编辑器应用的效率对你选择用来表现缓冲区的数据结构是特别敏感的。本章会用三种不同的底层表示（一个字符数组、一对字符栈，以及一个字符链表）实现对编辑器缓冲区的抽象，并且评估它们各自的优缺点。

[571]

13.2 设计简单的文本编辑器

现代的编辑器提供了一种高度完备的编辑环境，而且有一些别致的特性使其更加完善。比如使用鼠标来定位光标，或者用命令搜寻特定字符串文本。此外，它们趋向于使所有的编辑命令的操作结果可以和期望的相同。那些在编辑过程中可以始终显示当前缓冲区内容的编辑器称为所见即所得（wysiwyg，英文读作“wizzy-wig”）编辑器，它是“what you see is what you get”的首字母缩写词。这种编辑器易于使用，但是所有那些高级特性也让我们很难看到编辑器内部是如何工作的。

在计算的早期时代，编辑器相对简单。没有鼠标也没有完备的图形显示器，编辑器用来对在键盘上输入的指令做出回应。例如，对于一个基于键盘的典型编辑器，你需要敲击命令字母 I 外加一系列有序文字来插入新的文本。附加的命令完成其他的编辑功能，例如将光标在缓冲区中移动。通过输入正确的命令，你可以实现你所期望的任何文本编辑。考虑到本章的重点是编辑器缓冲区的表示，而不是支持一个更完备的编辑环境的必要高级特性，我们有必要以这种命令驱动形式的方法探究缓冲区的抽象。一旦你完成了对编辑器缓冲区的抽象实现，你就可以返回并将它合并为一个基于模型 – 视图 – 控制器模式的更完备的应用。

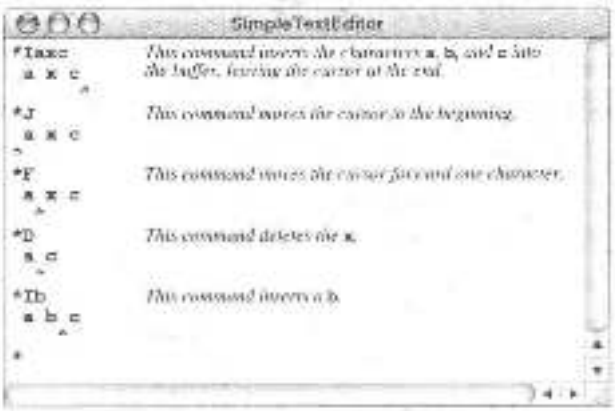
13.2.1 编辑器命令

接下来的几小节将展示一个极其简单的编辑器的开发过程。这个编辑器可以执行表 13-1 中所示的命令。除了可以获取所要插入字符的 I 命令之外，每一条编辑器命令都由一行上的单个字母组成。

表 13-1 一个基于命令的简单编辑器的可用命令

F	将编辑光标向前移动一个字符
B	将编辑光标向后移动一个字符
J	跳转至缓冲区的开始
E	将光标移至缓冲区的末尾
Ixxx	在当前光标位置插入字符串 xxx
D	删除当前光标位置之后的字符
H	输出所列出命令的帮助信息
Q	退出编辑器程序

下面运行的例子简述了这个基于命令的编辑器的操作，以及用来描述每个操作的文字注释。在这种情况下，用户首先插入字符 axc，然后将缓冲区的内容改为 abc。



编辑器程序显示了执行完每个命令之后缓冲区的状态。从运行的程序结果来看，程序用下一行的插入符号（^）标识了光标的位置。在真实的编辑器中这种行为是你不想看到的，但它可以让我们很清晰地看到程序到底是如何运行的。

在像 C++ 这样的面向对象语言中，定义一个类来表示编辑器缓冲区是很合理的。在这里使用类的优点就是它允许你将行为和表示分开。因为你理解了它必须响应的操作，所以你就早已经知道了一个编辑器缓冲区的行为。在 buffer.h 接口中，可以定义一个 EditorBuffer 类，它的公有接口提供了所需要的操作集，而它的数据表示是私有的。用户可完全通过 EditorBuffer 对象的公有接口而不必访问其底层的数据表示就对其进行操作。这样反过来可以让我们自由地改变数据的表示形式而无须修改用户程序。

13.2.2 EditorBuffer 类的公有接口

公有接口由一系列原型化的方法构成，这些方法实现了编辑器缓冲区的基本操作。你需要定义哪些操作？如果没有特别的要求，你需要为下面的 6 个编辑器命令定义方法。然而，就像类一样，你需要定义一个构造函数以允许你初始化一个新的缓冲区。由于这个类名为 EditorBuffer，因此它的构造函数的原型为：

572 ↓ 573

```
EditorBuffer();
```

该类还需要提供一个析构函数：

```
~EditorBuffer();
```

它可以撤消分配给 EditorBuffer 对象的任何堆存储空间。

处理完这些细节以后，下一步就是定义与编辑器命令相对应的方法。例如，将光标向前移动，可以定义如下方法：

```
void moveCursorForward();
```

在设计一个接口时，你不用关注这个操作是如何实现的，或者缓冲区和它的光标是如何表示的，牢记这一点十分重要。moveCursorForward 方法完全是由其抽象效果定义的。

除了实现编辑器命令的方法之外，编辑器应用程序必须能够显示缓冲区的内容，这也包括光标的位置。为了使这些操作可行，EditorBuffer 类具有以下公有方法：

```
string toString() const;
```

它以 C++ 字符串形式返回当前整个缓冲区的内容。同时，以下方法：

```
int getCursor() const;
```

以数字 0 与缓冲区长度之间的一个整数形式返回当前光标的位置。这些方法并不改变缓冲区的内容，因此用 const 关键字来标记它们是一个很好的实践。

editorbuffer.h 接口的内容显示在图 13-2 中。正如在第 12 章所列出的接口，EditorBuffer 类的私有部分看起来像一个盛色的空盒子。它将基于不同的缓冲区数据表示策略而用不同的定义填满。

524

```
/*
 * File: buffer.h
 * --------------
 * This file defines the interface for the EditorBuffer class.
 */

#ifndef _buffer_h
#define _buffer_h

/*
 * Class: EditorBuffer
 * -------------------
 * This class represents an editor buffer, which maintains an ordered
 * sequence of characters along with an insertion point called the cursor.
 */

class EditorBuffer {

public:

/*
 * Constructor: EditorBuffer
 * Usage: EditorBuffer buffer;
 * ---------------------------
 * Creates an empty editor buffer.
 */

   EditorBuffer();

/*
 * Destructor: ~EditorBuffer
 * -------------------------
 * Frees any heap storage associated with this buffer.
 */

   ~EditorBuffer();
```

图 13-2 编辑器缓冲区抽象的接口

```
/*
 * Methods: moveCursorForward, moveCursorBackward
 * Usage: buffer.moveCursorForward();
 *        buffer.moveCursorBackward();
 * ----------------------------------
 * Moves the cursor forward or backward one character.  If the command
 * would shift the cursor beyond either end of the buffer, this method
 * has no effect.
 */

   void moveCursorForward();
   void moveCursorBackward();

/*
 * Methods: moveCursorToStart, moveCursorToEnd
 * Usage: buffer.moveCursorToStart();
 *        buffer.moveCursorToEnd();
 * ---------------------------------
 * Moves the cursor to the start or the end of this buffer.
 */

   void moveCursorToStart();
   void moveCursorToEnd();

/*
 * Method: insertCharacter
 * Usage: buffer.insertCharacter(ch);
 * ----------------------------------
 * Inserts the character ch into this buffer at the cursor position,
 * leaving the cursor after the inserted character.
 */

   void insertCharacter(char ch);

/*
 * Method: deleteCharacter
 * Usage: buffer.deleteCharacter();
 * --------------------------------
 * Deletes the character immediately after the cursor, if any.
 */

   void deleteCharacter();

/*
 * Method: getText
 * Usage: string str = buffer.getText();
 * -------------------------------------
 * Returns the contents of the buffer as a string.
 */

   std::string getText() const;

/*
 * Method: getCursor
 * Usage: int cursor = buffer.getCursor();
 * ---------------------------------------
 * Returns the index of the cursor.
 */

   int getCursor() const;

   The private section of the class goes here.

};

   The implementation of the class goes here.

#endif
```

图 13-2 （续）

575
|
576

13.2.3 选择一种底层表示

甚至到这一步，你可能对哪种内部数据结构表示更适合这个类有自己的想法。因为这个缓冲区包含一个有序的字符序列，一个更有可能的主观选择就是采用 string 或者

Vector<char>类作为它的底层表示。只要你的运行环境提供这些类，那么选择它们中的任一个都是一种不错的底层表示。然而，本章的目的是研究表示方式的选择如何影响应用程序的效率。如果程序使用了string或者Vector<char>这些高级的数据结构，理解这一点就会更困难。因为这些类的内部工作对于用户来说是不可见的。如果你采用相反的方式使用内置的数据结构来限制你的实现，那么每个操作就都变为可见了。因此，你就可以比较容易鉴别不同的底层表示方式的效率。逻辑表明使用一个字符数组作为底层表示，因为数组操作不会隐藏其时空开销。

尽管使用数组表达缓冲区是一个相对合理的方法，但同时也存在其他的表示，并且它们可能更有意思。本章的基础出发点（甚至是本书的重点）是你不能草率地确定一种特定的表示方式。在这个编辑器缓冲区例子中，数组只是其中一种可选方案。每一种可选方案都有它的优缺点。通过权衡各种方案，你可能会在一系列特定的环境下选择某种策略，然后在其他环境中选择别的方案。与此同时，你要注意到：不管选择哪种表示，编辑器必须能够实现完全相同的命令集。因此，即使底层表示有所改变，编辑器缓冲区的外部行为也必须保持一致。

13.2.4 编写编辑器应用代码

一旦定义完公有接口，即使你还没有实现缓冲区类或者还没有确定一个适合的内部表示，你也可以编写编辑器应用程序了。编写编辑器应用时，你最需要考虑的就是每个操作都干了什么。从这一层面上来看，实现细节并不重要。

只要你使用表13-1所示的命令，编写编辑器程序就会相对简单。这个程序简单地创建了一个新的EditorBuffer对象，然后进入一个循环，在该循环中，程序读取一系列的编辑器命令。只要用户输入一个命令，程序就会查看这个命令的第一个字母，然后通过调用缓冲区接口的适当方法执行请求的命令。基于命令的编辑器的代码如图13-3所示。

577

```
/*
 * File: SimpleTextEditor.cpp
 * --------------------------
 * This program implements a simple command-driven text editor, which is
 * used to test the EditorBuffer class.
 */

#include <cctype>
#include <iostream>
#include "buffer.h"
#include "foreach.h"
#include "simpio.h"
using namespace std;

/* Function prototypes */

void executeCommand(EditorBuffer & buffer, string line);
void displayBuffer(EditorBuffer & buffer);
void printHelpText();

int main() {
   EditorBuffer buffer;
   while (true) {
      string cmd = getLine("*");
      if (cmd != "") executeCommand(buffer, cmd);
   }
   return 0;
}
```

图13-3 测试EditorBuffer类的简单文本编辑器

```
/*
 * Function: executeCommand
 * Usage: executeCommand(buffer, line);
 * ------------------------------------
 * Executes the command specified by line on the editor buffer.
 */

void executeCommand(EditorBuffer & buffer, string line) {
   switch (toupper(line[0])) {
    case 'I': for (int i = 1; i < line.length(); i++) {
                 buffer.insertCharacter(line[i]);
              }
              displayBuffer(buffer);
              break;
    case 'D': buffer.deleteCharacter(); displayBuffer(buffer); break;
    case 'F': buffer.moveCursorForward(); displayBuffer(buffer); break;
    case 'B': buffer.moveCursorBackward(); displayBuffer(buffer); break;
    case 'J': buffer.moveCursorToStart(); displayBuffer(buffer); break;
    case 'E': buffer.moveCursorToEnd(); displayBuffer(buffer); break;
    case 'H': printHelpText(); break;
    case 'Q': exit(0);
    default: cout << "Illegal command" << endl; break;
   }
}

/*
 * Function: displayBuffer
 * Usage: displayBuffer(buffer);
 * -----------------------------
 * Displays the state of the buffer including the position of the cursor.
 */

void displayBuffer(EditorBuffer & buffer) {
   string str = buffer.getText();
   for (int i = 0; i < str.length(); i++) {
      cout << " " << str[i];
   }
   cout << endl;
   cout << string(2 * buffer.getCursor(), ' ') << "^" << endl;
}

/*
 * Function: printHelpText
 * Usage: printHelpText();
 * -----------------------
 * Displays a message showing the legal commands.
 */

void printHelpText() {
   cout << "Editor commands:" << endl;
   cout << "  Iabc  Inserts the characters abc at the cursor position" << endl;
   cout << "  F     Moves the cursor forward one character" << endl;
   cout << "  B     Moves the cursor backward one character" << endl;
   cout << "  D     Deletes the character after the cursor" << endl;
   cout << "  J     Jumps to the beginning of the buffer" << endl;
   cout << "  E     Jumps to the end of the buffer" << endl;
   cout << "  H     Prints this message" << endl;
   cout << "  Q     Exits from the editor program" << endl;
}
```

图 13-3 （续）

13.3 基于数组的类实现

正如 13.2.3 一节所指出的，缓冲区的一种可能表示就是采用一个字符数组。尽管这不是表示缓冲区的唯一可选方案，但它仍然是一个很有用的出发点。毕竟，缓冲区的字符形成了一个有序的元素序列，这和传统的数组使用方法是相一致的。然而，用来实现缓冲区的数组必须动态分配，从而使它能够随着缓冲区字符数目的增加而变大。

578
579

13.3.1 定义私有数据结构

在很多方面，基于数组的编辑器缓冲区的底层表示看起来与第12章的CharStack类很像。CharStack类定义了三个实例变量：一个用来指向存储元素的动态数组的指针、数组的容量及其中字符的数目。对于一个基于数组的缓冲区来说，需要相同的实例变量，但将变量名从count改为length是很有意义的，因为这样对于讨论缓冲区的长度来说更方便一些。除了实例变量之外，EditorBuffer类的私有数据也必须包含一个指示当前光标位置的数字。这些实例变量外加私有方法原型，以及一些标准定义，使得不用通过拷贝就可实现EditorBuffer类。EditorBuffer类的私有部分如图13-4所示。

```
/* Private section */

private:

/*
 * Implementation notes: Buffer data structure
 * -------------------------------------------
 * In the array-based implementation of the buffer, the characters in the
 * buffer are stored in a dynamic array.  In addition to the array, the
 * structure keeps track of the capacity of the buffer, the length of the
 * buffer, and the cursor position.  The cursor position is the index of
 * the character that follows the cursor on the screen.
 */

/* Constants */

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   char *array;           /* Dynamic array of characters      */
   int capacity;          /* Allocated size of that array     */
   int length;            /* Number of characters in buffer   */
   int cursor;            /* Index of character after cursor */

/* Make it illegal to copy editor buffers */

   EditorBuffer(const EditorBuffer & value) { }
   const EditorBuffer & operator=(const EditorBuffer & rhs) { return *this; }

/* Private method prototype */

   void expandCapacity();
```

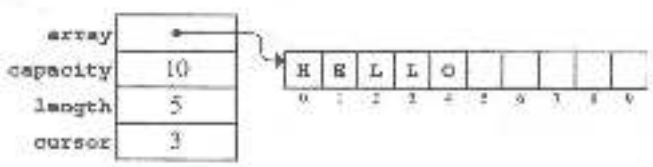
580

图13-4 基于数组实现的编辑器的私有部分

给定这种数据结构设计，一个包含以下内容的缓冲区：

H E L L O

如下图所示：



13.3.2 缓冲区操作的实现

基于数组的编辑器的大多数操作还是很容易实现的。四个光标移动操作中的任何一个都可以通过给cursor域的实例变量赋一个新值来实现。例如，将光标移动到缓冲区开始，仅仅需要给cursor赋0值；将光标移动到缓冲区末尾，仅需要将length域拷贝给

cursor域。同样，将光标前后移动也仅仅是对光标值的加减，然而确保光标值没有越界是非常重要的。你可以在图13-5中EditorBuffer类的实现中看到这些简单方法的实现代码。

图13-5中唯一需要进一步讨论的就是构造函数、析构函数，insertCharacter和deleteCharacter方法。因为这些方法可能看起来比较有技巧性，尤其是对于那些初次接触编码实现的人。所以在代码中包含其操作的注释是非常重要的。例如，在图13-5所示的代码中，对那些以“Implementation notes”注释的特殊的方法提供了其他文档，诸如光标移动的简单方法就没有单独的文档。

构造函数负责初始化那些表示空缓冲区的实例变量，因此构造函数的注释是描述类的实例变量及其含义的最佳位置。析构函数主要用于释放在对象生命周期中对其所分配的动态内存。对EditorBuffer这一基于数组的类的实现来说，唯一分配的动态内存就是用来存储文本的数组。因此，析构函数的代码由以下代码构成：

```
delete[] array;
```

它可以删除给数组array分配的动态内存。 581

```
/*
 * File: buffer.cpp (array version)
 * --------------------------------
 * This file implements the buffer.h interface using an array representation.
 */

#include <iostream>
#include "buffer.h"
using namespace std;

/*
 * Implementation notes: Constructor and destructor
 * ------------------------------------------------
 * The constructor initializes the private fields.  The destructor
 * frees the heap-allocated memory, which is the dynamic array.
 */

EditorBuffer::EditorBuffer() {
   capacity = INITIAL_CAPACITY;
   array = new char[capacity];
   length = 0;
   cursor = 0;
}

EditorBuffer::~EditorBuffer() {
   delete[] array;
}

/*
 * Implementation notes: moveCursor methods
 * ----------------------------------------
 * The four moveCursor methods simply adjust the value of cursor.
 */

void EditorBuffer::moveCursorForward() {
   if (cursor < length) cursor++;
}

void EditorBuffer::moveCursorBackward() {
   if (cursor > 0) cursor--;
}

void EditorBuffer::moveCursorToStart() {
   cursor = 0;
}

void EditorBuffer::moveCursorToEnd() {
   cursor = length;
}
```

图13-5 基于数组的编辑器缓冲区的实现

```
/*
 * Implementation notes: character insertion and deletion
 * ------------------------------------------------------
 * Each of the functions that inserts or deletes characters must shift
 * all subsequent characters in the array, either to make room for new
 * insertions or to close up space left by deletions.
 */

void EditorBuffer::insertCharacter(char ch) {
   if (length == capacity) expandCapacity();
   for (int i = length; i > cursor; i--) {
      array[i] = array[i - 1];
   }
   array[cursor] = ch;
   length++;
   cursor++;
}

void EditorBuffer::deleteCharacter() {
   if (cursor < length) {
      for (int i = cursor+1; i < length; i++) {
         array[i - 1] = array[i];
      }
      length--;
   }
}

/* Simple getter methods: getText, getCursor */

string EditorBuffer::getText() const {
   return string(array, length);
}

int EditorBuffer::getCursor() const {
   return cursor;
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This private method doubles the size of the array whenever the old one
 * runs out of space.  To do so, expandCapacity allocates a new array,
 * copies the old characters to the new array, and then frees the old array.
 */

void EditorBuffer::expandCapacity() {
   char *oldArray = array;
   capacity *= 2;
   array = new char[capacity];
   for (int i = 0; i < length; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}
```

582 ~ 583

图 13-5 （续）

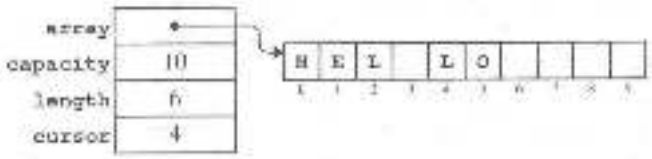
insertCharacter 和 deleteCharacter 方法是很有趣的，因为它们都需要在数组中移动字符，要么为待插入的字符提供空间，要么释放一个已删除字符遗留的空间。例如，假设你想在下面的缓冲区中将字符 x 插入到光标位置：

H E L_ L O

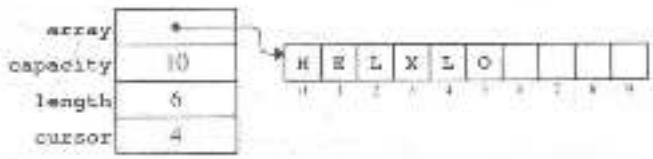
在以数组表示的缓冲区中要进行这样的操作，要首先确保数组中有空闲空间。如果数组的 length 域值与它的 capacity 域值相等，则在当前分配的存储空间中就没有容纳新字符

的空间了。这样的话，就有必要像在第 12 章中 CharStack 类的实现一样，用同样的方法扩展数组的容量。

然而，若数组中有多余的空间，且全部都处于数组的最后。若在中间插入字符，就需要在当前光标所在的位置为将要插入的字符提供一个空间。获得空间唯一的方法就是将光标右侧的字符都向右移动一个位置，因此缓冲区结构会变为如下形式：



在数组中有一个空间可供我们插入字符 X。此后光标向前移动，指向新插入的字符后面，从而得到如下图所示的结构：



deleteCharacter 操作也是类似的，只是它需要一个循环来去除删除了元素之后所遗留下的数组空间中的空隙。

13.3.3 基于数组的编辑器的时间复杂度

为了建立一个可以与其他方式进行比较的基准，确定一个基于数组实现的编辑器的时间复杂度是很有帮助的。通常，复杂度分析的目的是理解编辑操作所需的执行时间随着问题规模的变化是如何改变的。在编辑器例子中，缓冲区中字符的数目是对问题规模的最好度量。 [584]
因此，对于编辑器缓冲区，你就需要确定缓冲区的大小是如何影响每个编辑操作的运行时间的。

对于基于数组的实现，最容易理解的操作就是移动光标的操作。例如，moveCursorForward 方法有如下实现：

```
void EditorBuffer::moveCursorForward() {
   if (cursor < length) cursor++;
}
```

尽管这个方法检查了缓冲区的长度，但是过不了多久我们就知道方法的执行时间与缓冲区的长度是无关的。无论缓冲区的长度是多少，这个函数执行的都是相同的操作：进行一个测试，并对 cursor 进行一个递增操作。因为执行时间与 N 是无关的，所以 moveCursorForward 操作运行的时间是 $O(1)$。相同的分析方法适用于其他移动光标的操作，这些操作都与缓冲区的长度无关。

但是 insertCharacter 方法怎样呢？在基于数组实现的 EditorBuffer 类中，insertCharacter 方法的内容包含下面的 for 循环：

```
for (int i = length; i > cursor; i--) {
   array[i] = array[i - 1];
}
```

如果在缓冲区的末尾插入一个字符，那么这个方法运行得特别快，因为这样就不需要移动别的字符以便为新的字符留出空间。换一个角度来看，如果在缓冲区的开头插入一个字符，缓冲区数组的每一个元素都需要右移。于是，在最坏情况下，insertCharacter 方法的运行时间与字符的个数成正比，因此时间复杂度是 $O(N)$。由于 deleteCharacter 方法与 insertCharacter 方法的情况类似，因此，它的时间复杂度也是 $O(N)$。每个编辑器操作的时间复杂度见表 13-2。

表 13-2 基于数组的缓冲区时间复杂度

操 作	数 组
moveCursorForward	$O(1)$
moveCursorBackward	$O(1)$
moveCursorToStart	$O(1)$
moveCursorToEnd	$O(1)$
insertCharacter	$O(N)$
deleteCharacter	$O(N)$

对于编辑器程序来说，表中最后两个需要线性时间的操作具有很重要的性能。如果一个编辑器使用数组来表示其内部的缓冲区，它就会随着缓冲区中字符数目的增加而运行得越来越慢。由于这个问题看起来比较严重，所以探索其他的表示是合乎情理的。

13.4 基于栈的类实现

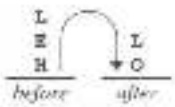
用数组实现缓冲区的问题在于：当插入和删除操作发生在数组的开头部分时，程序会运行得很慢。当上述相同的操作在数组的尾部进行时，程序的运行速度就相对较快，因为这不需要移动数组内部的字符。这种性质暗示了一种加快运行的方法：迫使所有的插入和删除字符操作都发生在缓冲区的尾部。尽管这种方法从用户的角度来看是完全不可行的，但是它确实是一种可行方法的萌芽。

使插入和删除操作变快的必要技术就是：可以以光标位置为分界线，将缓冲区分为光标前和光标后两个相互独立的结构。因为所有对缓冲区的修改都出现在光标的位置，所以这些结构都像栈一样，并且可以用第 12 章介绍的 CharStack 类来表示。在光标之前的元素被压入一个栈中，因此缓冲区的首元素就在栈底，而光标之前的那个元素则在栈顶。在光标之后的元素的存储与此恰恰相反，将缓冲区的最后一个元素放在栈底，然后将光标后面的那个元素放在栈顶。

说明这种结构的最好方法就是使用一个图。如果这个缓冲区包含以下内容：

H E L L O

则该缓冲区的两个栈的表示如下图所示：



585 ~ 586

为了读取缓冲区的内容，有必要如图中箭头所示的那样先读取 before 栈的内容，然后再读取 after 栈的内容。

13.4.1 定义私有数据结构

使用这种策略，即一个缓冲区对象的实例变量就是一对栈，它们分别保存光标前后两部分缓冲区的内容。对基于栈的缓冲区而言，类的私有部分的声明仅有如图 13-6 所示的两个实例变量。切记，光标在这个模型中不是明确表示的，它仅仅是两个栈的边界。

```
/* Private section */

private:

/*
 * Implementation notes: Buffer data structure
 * -------------------------------------------
 * In the stack-based buffer model, the characters are stored in two
 * stacks.  Characters before the cursor are stored in a stack named
 * "before"; characters after the cursor are stored in a stack named
 * "after".  In each case, the characters closest to the cursor are
 * closer to the top of the stack.  The advantage of this
 * representation is that insertion and deletion at the current
 * cursor position occurs in constant time.
 */

/* Instance variables */

   CharStack before;      /* Stack of characters before the cursor */
   CharStack after;       /* Stack of characters after the cursor  */

/* Make it illegal to copy editor buffers */

   EditorBuffer(const EditorBuffer & value) { }
   const EditorBuffer & operator=(const EditorBuffer & rhs) { return *this; }
```

图 13-6 基于栈的编辑器的私有部分

13.4.2 缓冲区操作的实现

在栈模型中，编辑器的大部分操作的实现还是很简单的。例如，后移元素只需要将 before 栈的元素弹出后压入 after 栈即可。前移元素也是与此相对称的。插入字符需要将该字符压入 before 栈。删除元素就包括从 after 栈将字符弹出栈并扔掉字符。

这种概念上的操作框架使得编写基于栈的编辑器代码更容易，其实现代码如图 13-7 所示。insertCharacter、deleteCharacter、moveCursorForward 和 moveCursorBackward 这四个命令的执行时间是常数时间，因为它们调用的栈操作的时间复杂度为 $O(1)$。 587

```
/*
 * File: buffer.cpp (stack version)
 * --------------------------------
 * This file implements the EditorBuffer class using a pair of character
 * stacks to represent the buffer.
 */

#include <iostream>
#include "buffer.h"
#include "charstack.h"
using namespace std;

/*
 * Implementation notes: Constructor and destructor
 * ------------------------------------------------
 * In this implementation, all dynamic allocation is managed by the
 * CharStack class, which means there is no work for EditorBuffer to do.
 */

EditorBuffer::EditorBuffer() { }
EditorBuffer::~EditorBuffer() { }

/*
 * Implementation notes: moveCursor methods
 * ----------------------------------------
 * The four moveCursor methods use push and pop to transfer values
 * between the two stacks.
 */
```

图 13-7 基于栈的编辑器的实现

```
void EditorBuffer::moveCursorForward() {
   if (!after.isEmpty()) {
      before.push(after.pop());
   }
}

void EditorBuffer::moveCursorBackward() {
   if (!before.isEmpty()) {
      after.push(before.pop());
   }
}

void EditorBuffer::moveCursorToStart() {
   while (!before.isEmpty()) {
      after.push(before.pop());
   }
}

void EditorBuffer::moveCursorToEnd() {
   while (!after.isEmpty()) {
      before.push(after.pop());
   }
}

/*
 * Implementation notes: character insertion and deletion
 * ------------------------------------------------------
 * Each of the functions that inserts or deletes characters can do so
 * with a single push or pop operation.
 */

void EditorBuffer::insertCharacter(char ch) {
   before.push(ch);
}

void EditorBuffer::deleteCharacter() {
   if (!after.isEmpty()) {
      after.pop();
   }
}

/*
 * Implementation notes: getText and getCursor
 * -------------------------------------------
 * The only difficult part of implementing these operations is making
 * sure that the state of the buffer is restored after copying the
 * characters from the two stacks.
 */

string EditorBuffer::getText() const {
   CharStack beforeCopy = before;
   CharStack afterCopy = after;
   string str = "";
   while (!beforeCopy.isEmpty()) {
      str = beforeCopy.pop() + str;
   }
   while (!afterCopy.isEmpty()) {
      str += afterCopy.pop();
   }
   return str;
}

int EditorBuffer::getCursor() const {
   return before.size();
}
```

588 / 589

图 13-7 （续）

但是剩余的两个操作moveCursorToStart和moveCursorToEnd又会怎样呢？上述每一种操作都需要将某个栈内的全部内容移动到另外一个栈里。假设这个操作由类CharStack提供，完成此操作唯一的方法就是将这些字符一个个从一个栈弹出并压入到另外一个栈中，直到最开始的栈为空栈。例如，moveCursorToEnd操作就有如下实现：

```
void EditorBuffer::moveCursorToEnd() {
   while (!after.isEmpty()) {
      before.push(after.pop());
   }
}
```

这些实现可以达到预期的结果，不过在最坏情况下，它的时间复杂度为 $O(N)$。

13.4.3 时间复杂度的比较

表 13-3 列出了基于数组和基于栈的编辑器实现的操作的时间复杂度。哪种实现更好呢？如果没有一些使用模式的知识，就很难回答这个问题。然而，如果对人们使用编辑器的方式有所了解的话，就会觉得基于栈的表示更高效。因为基于数组实现的较慢的操作（插入和删除）使用的频率要远远高于基于栈实现的比较耗时的操作（将光标移动很长一段距离）。

如果涉及操作的使用频率的话，这种权衡是相对合理的。但是也有必要再问一下有没有更好的方案。毕竟，我们可以确定这六个基本编辑操作中至少有一个在上述两个编辑器的实现中的运行时间是常数时间。插入运算在数组实现中比较慢，但是在使用栈实现时是很快的。相反，将光标移到缓冲区开头对数组来说特别快，但对于栈那就比较慢了。然而，没有任何一种操作，从本质上来说是慢的。因为总有某种实现可以让该操作变快。那有可能开发出一种使得所有的操作都比较快的实现吗？答案是肯定的，但是解决谜题的关键就需要你学习一种新的方法，以便在一个数据结构中表示元素的有序关系。

表 13-3 基于数组和基于栈的缓冲区的时间复杂度

操　作	数　组	栈
moveCursorForward	$O(1)$	$O(1)$
moveCursorBackward	$O(1)$	$O(1)$
moveCursorToStart	$O(1)$	$O(N)$
moveCursorToEnd	$O(1)$	$O(N)$
insertCharacter	$O(N)$	$O(1)$
deleteCharacter	$O(N)$	$O(1)$

13.5 基于列表的类实现

作为寻找一个更加高效的编辑器缓冲区表示的第一步，检查之前的方法为何不能对特定操作提供有效服务是很有意义的。在数组实现的例子中，答案是显而易见的：当我们需要在缓冲区的开头插入一些文本时，需要移动大量的字符。例如，假设你准备输入字母表却输成了如下情况：

```
A C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

当你发现遗漏了字母 B 时，就需要将接下来的 24 个字符统一右移一位，从而为遗漏的字母腾出位置。只要缓冲区不是特别大，一台现代计算机可以相对快捷地完成此操作。虽然如此，如果缓冲区里面字符的数目已足够大的话，这种操作所需的时间还是相当可观的。

然而，假设你写信的时候现代计算机还没有发明。想象你是托马斯·杰斐逊，正在忙于起草独立宣言。在对乔治沙皇的抱怨中，你很认真地写下了下面这句话：

Our repeated Petitions have been answered by repeated injury.

遗憾的是，在最后一分钟，有人觉得在这个句子中的第二个 injury 前面加上 only 一词更为恰当。对这个问题思考一会之后，你可能决定（就像有人在真实的独立宣言中做的那样）拿出你的笔，像下面这样加上漏掉的单词：

Our repeated Petitions have been answered only by repeated injury.

590
~
591

如果用同样的策略给字母表添加遗漏的字母，你可能会做如下的编辑：

```
    B
A   C D E F G H I J K L M N O P Q R S T U V W X Y Z
^
```

这个结果可能看起来不太优雅，但在这种特殊情况下也是可以接受的。

使用这种过时的编辑策略的优点在于：它允许你打破所有字母都必须按照它们像输出时以特定顺序排序这一规则。下一行的插入符告诉你的眼睛在读完 A 以后，需要上移读 B，接着下移读 C，然后再按照顺序继续读。也需要注意到使用这种插入策略的另一个优点。不管这一行有多长，你要做的仅仅是写一个新的字符和插入符号。当使用纸笔时，插入所用的时间是常数时间。

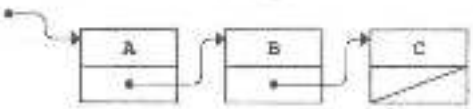
链表可以允许你达到近乎相同的效果。如果字符是存储在一个链表中而不是字符数组中，插入一个遗漏的字符你所要做的仅仅是修改几个指针。如果缓冲区的原始内容用以下链表存储：

```
A→C→D→E→F→G→H→I→J→K→L→M→N→O→P→Q→R→S→T→U→V→W→X→Y→Z
```

你所需要做的就是：（1）把 B 写在某处；（2）从 B 开始画一个箭头让它指向 A 原来所指向的地方（也就是现在的 C）；（3）改变 A 箭头所指的位置让它指向 B，如下图所示：

```
  B
A→C→D→E→F→G→H→I→J→K→L→M→N→O→P→Q→R→S→T→U→V→W→X→Y→Z
```

这个结构和第 12 章的链表具有相同的形式。为了表示这个字符链，你需要将这些字符存入链表的结点中。例如，字符串 ABC 的链表如下图所示：



然而在链表结构图中，通常可以在结点中画一条斜线以表示它是空值（NULL），如上面例子中的 C 结点所示。

乍一看，链表是你要表达缓冲区内容的全部。唯一的问题就是你还需要表示光标位置。如果用整数的形式存储光标，寻找光标的当前位置就需要数这个链表结点的个数直到光标所在的位置为止。这个策略需要的时间是线性的。一个更好的方法就是定义一个 EditorBuffer 类，并且它包含两个指向结点类 Cell 对象的指针：一个 start 指针说明
592 链表从何处开始；一个 cursor 指针标记当前光标的位置。

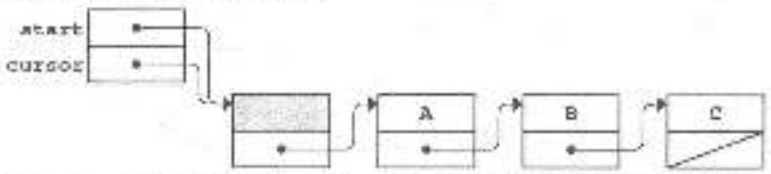
在没有了解光标的工作细节前，你可能会觉得这个设计似乎比较合理。如果一个缓冲区有三个字符，你的第一反应肯定是使用一个有三个结点的链表。遗憾的是，这里还有一些问题。假设一个缓冲区有三个字符，而对于光标而言却有四个可能的位置，如下图所示：

```
^A B C      A^B C      A B^C      A B C^
```

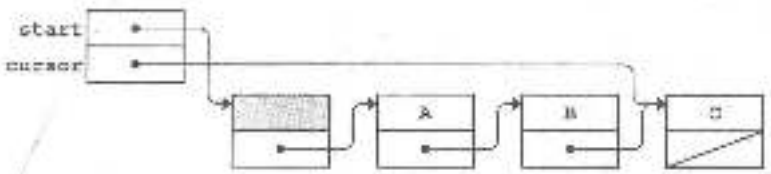
如果 cursor 域在这里只有三个结点可以存放其指向，你就不清楚怎样才能表示每个可能的光标位置。

解决这个问题有很多巧妙的方法，但最常用的就是额外分配一个结点，以便让该链表对每个可能的插入点都有一个存储单元。通常而言，这个额外分配的结点处于链表首部，被称为空结点（dummy cell）。空结点中 ch 域的值是无关的，在图上用灰色的背景表示。

当使用空结点方法时，cursor 域指向逻辑插入点的前面。例如，一个包含 ABC 且光标在最前面的缓冲区看起来如下图所示：



start 指针和 cursor 指针均指向空结点，并在空结点后面进行插入操作。如果 cursor 域指向了缓冲区的末尾，其链表图就该具有如下形态：



出现在类的私有部分的仅有的实例变量就是 start 指针和 cursor 指针。尽管这个结构的剩余部分还没有正式成为该对象的一部分，但如果在 buffer.h 文件的私有部分中将这种结构文档化，它可以为之后使用这种结构的程序员提供帮助，如图 13-8 所示。 [593]

```
/* Private section */

private:

/*
 * Implementation notes: Buffer data structure
 * -------------------------------------------
 * In the linked-list implementation of the buffer, the characters
 * in the buffer are stored in a list of Cell structures, each of
 * which contains a character and a pointer to the next cell in the
 * chain.  To simplify the code used to maintain the cursor, this
 * implementation adds an extra "dummy" cell at the beginning of the
 * list.  The character in this cell is not used, but having it in
 * the data structure provides a cell for the cursor to point to
 * when the cursor is at the beginning of the buffer.
 *
 * The following diagram shows the structure of the list-based buffer
 * containing "ABC" with the cursor at the beginning:
 *
 *         +-------+      +------+      +------+      +------+      +------+
 *  start  |   o---+----->|      |  --->|  A   |  --->|  B   |  --->|  C   |
 *         +-------+    / +------+    / +------+    / +------+    / +------+
 * cursor  |   o---+---   |   o--+---   |   o--+---   |   o--+---   |  /   |
 *         +-------+      +------+      +------+      +------+      +------+
 */

/*
 * Type: Cell
 * ----------
 * This structure type is used locally within the implementation to
 * store each cell in the linked-list representation.  Each cell
 * contains one character and a pointer to the next cell in the chain.
 */
```

图 13-8 基于链表的编辑器的私有部分

```
    struct Cell {
      char ch;
      Cell *link;
    };

  /* Instance variables */

    Cell *start;          /* Pointer to the dummy cell      */
    Cell *cursor;         /* Pointer to cell before cursor  */

  /* Make it illegal to copy editor buffers */

    EditorBuffer(const EditorBuffer & value) { }
    const EditorBuffer & operator=(const EditorBuffer & rhs) { return *this; }
```

图 13-8 （续）

13.5.1 链表缓冲区的插入操作

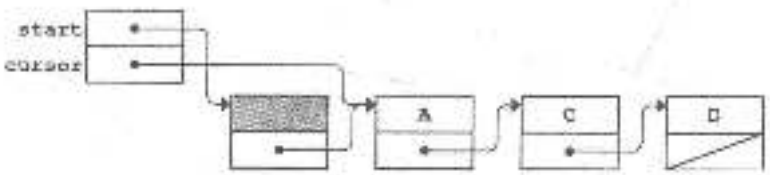
无论光标的位置在那里，链表的插入操作由下面几步构成：

1. 分配一个新的结点，让临时变量 cp 指向这个新的结点。
2. 将要插入的字符的值复制到新的结点。
3. 找到 cursor 域所指向的结点，让新分配的结点的指针指向该指针的指针。这个操作可以保证你没有丢失当前光标所指向的结点中的字符。
4. 改变当前光标所指向的结点的指针，让它指向新的结点。
5. 改变缓冲区中的 cursor 域的指针，让它指向新的结点。这个操作可以保证插入一个字符后，下一个字符也可以通过同样的方法插入。

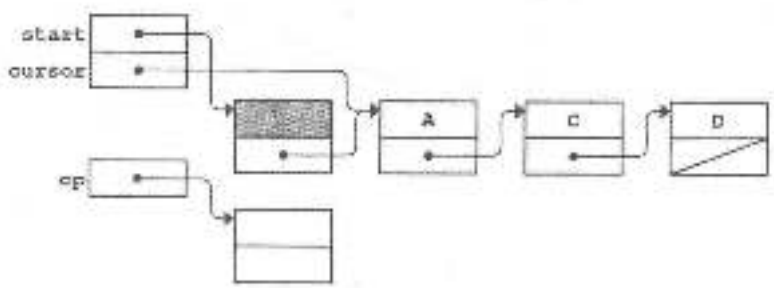
为了解释这个过程，假设你想把字母 B 插入到如下的缓冲区中：

A C D
 ^

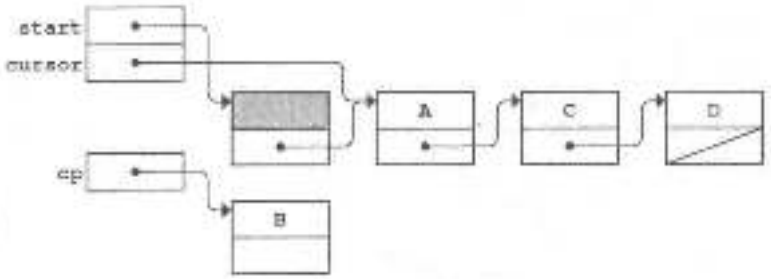
如上图所示，光标在字母 A 和 C 之间。在插入之前链表状况如下图所示：



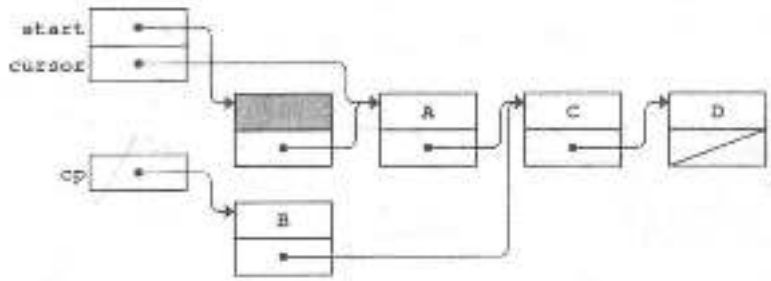
第一步：分配一个新的结点，并且让 cp 变量指向该结点，如下图所示：



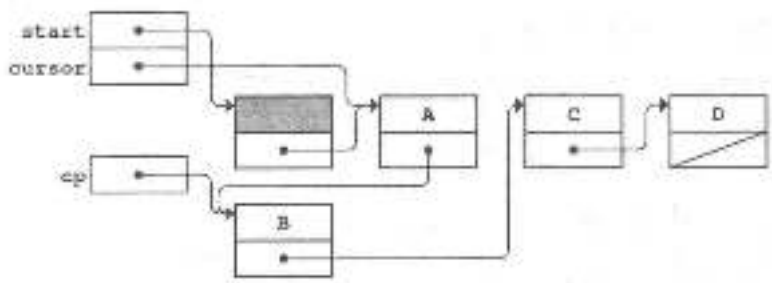
第二步：将字符 B 存入新的结点中的 ch 域，如下图所示：



第三步：将光标 cursor 所指位置的结点的指针赋给新的结点的指针。指针指向的结点就包含字符 C。结果如下图所示：

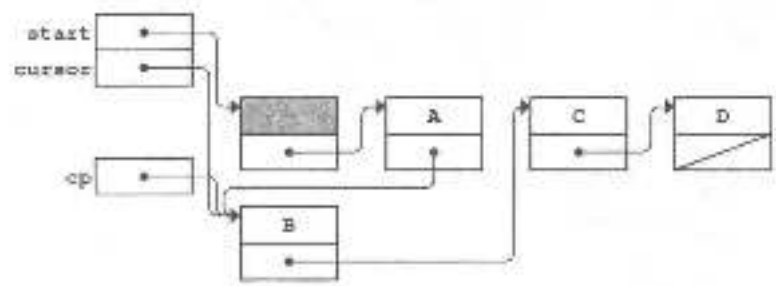


第四步：改变当前光标指针所指向的结点的指针，使其指向那个新分配的结点，如下图所示：

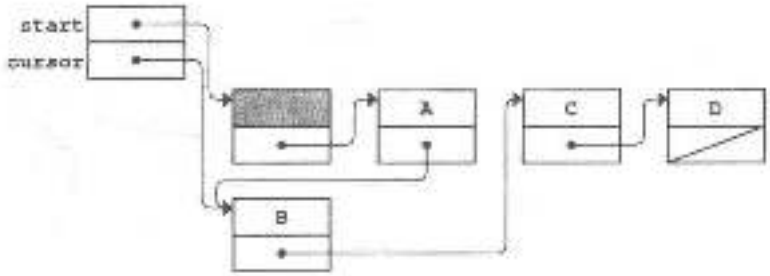


注意到缓冲区现在已经有了正确的内容。如果从缓冲区起始的空结点开始，依照箭头指示，我们会依次发现包含 A、B、C 和 D 这四个字母的结点。 596

最后一步就是修改缓冲区结构中的光标 cursor 指针域，使其指向新分配的结点。完成之后其链表内容如下图所示：



当这个程序从 insertCharacter 方法返回后，临时变量 cp 就会被释放。最终缓冲区的状态如下图所示：



这个缓冲区的内容包括：

A B C D
^

下面是用C++语言对insertCharacter方法实现的一个简单版本，它可以分为几个步骤实现：

```
void EditorBuffer::insertCharacter(char ch) {
   Cell *cp = new Cell;
   cp->ch = ch;
   cp->link = cursor->link;
   cursor->link = cp;
   cursor = cp;
}
```

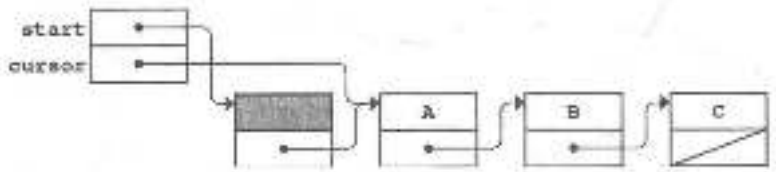
597 由于在这个方法里没有循环，所以insertCharacter方法的运行时间是常数时间。

13.5.2 链表缓冲区的删除操作

在链表中删除一个结点，你要做的只是将它从指针链中移除。假设当前缓冲区的内容为：

A B C
^

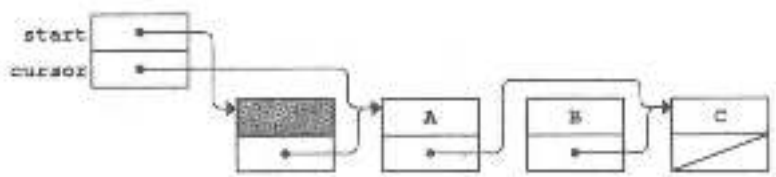
采用图形表示法，它的状态如下所示：



删除光标之后的字符需要你通过改变存储字母A的那个结点的指针，使其指向存储字母B的结点，从而删除存储字母B的结点。为了找到想要删除的字符，你需要使当前指针指向它所指向结点的指针所指向的结点。因此必要的语句为：

```
cursor->link = cursor->link->link;
```

执行这条语句就会让缓冲区变为以下状态：



由于存储字母 s 的结点对于链表而言是不可访问的，因此，通过调用 delete 函数将它释放是一个好办法。正如以下 deleteCharacter 方法的实现：

```
void EditorBuffer::deleteCharacter() {
   if (cursor->link != NULL) {
      Cell *oldCell = cursor->link;
      cursor->link = cursor->link->link;
      delete oldCell;
   }
}
```

598

注意：当你调整链表指针时，你需要一个像 oldCell 的变量来保存将要被释放的指针所指向的结点。如果你没有保存这个值，当你调用 delete 函数时，将无法获得该结点。

13.5.3 链表表示法中光标的移动

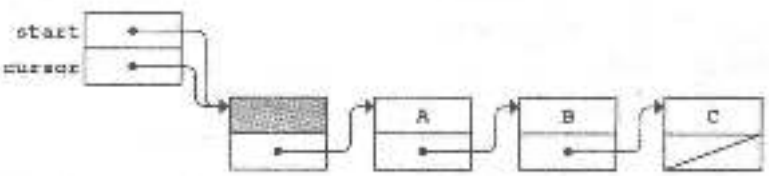
在 EditorBuffer 类中剩下的操作就是对光标的简单移动。你该如何在链表的缓冲区中实现这些操作呢？其中的 moveCursorForward 和 moveCursorToStart 这两种操作在链表模型中是很容易实现的。例如，要将光标前移，你只需要将缓冲区中指向当前结点的 link 域指针值取出来，并使 cursor 指针指向该结点的 link 域指针所指向的结点。实现该操作的必要语句如下：

```
cursor = cursor->link;
```

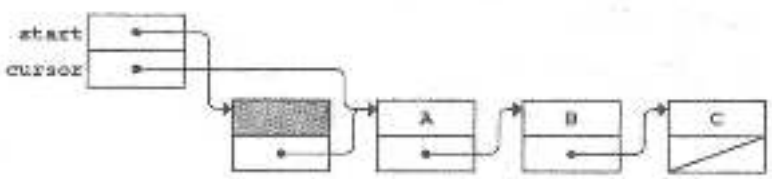
例如，假设编辑器缓冲区包含如下图所示在开始位置的光标：

A B C

那么缓冲区的链表结构图为：



moveCursorForward 操作执行完以后，其链表结构如下图所示：



当然，当到达缓冲区的末尾时，你就不能再前移光标了。moveCursorForward 操作的实现必须检查这种情况，所以该方法的完整定义如下：

599

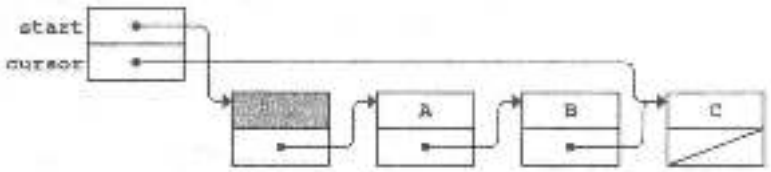
```
void EditorBuffer::moveCursorForward() {
   if (cursor->link != NULL) {
      cursor = cursor->link;
   }
}
```

将光标移动到缓冲区的起始处是非常简单的。无论光标处在哪个位置，我们都可

以通过把start指针复制给cursor指针，从而将光标移动到缓冲区的开头。因此，moveCursorToStart的实现仅仅是以下代码：

```
void EditorBuffer::moveCursorToStart() {
   cursor = start;
}
```

然而，moveCursorBackward和moveCursorToEnd操作更加复杂。例如，假设光标在包含了字符ABC的缓冲区的末尾，你想将光标后移一个位置。若以图形表示，这个缓冲区如下图所示：



假设在EditorBuffer类中，回溯指针操作不为常数时间。问题是你没有简单的方法（从你看到的信息中）找出当前结点的前一个结点。指针可以允许你按照其指向对象的一条链继续向前查找，但是指针的指向是不能逆转的。仅给定一个结点的地址是无法找出其结点之前的结点的。关于指针图，这种约束的效果就是我们可以从箭头的尾部移动到它所指向的结点中，但我们不可以从箭头的顶部返回指向该箭头的结点。

在由链表结构表示的缓冲区中，你需要以你在缓冲区结构中看到的数据来实现每一个操作，这些数据包含start指针和cursor指针。仅仅关注cursor指针并且按照光标当前位置可以到达的位置是没希望的。因为在那条指针链上可到达的结点都是那些缓冲区中特别靠后的单元。然而，start指针就为你提供了完整的指针链表。同时，你也需要关注
600 cursor指针的值，因为你需要回到那个位置。

在放弃希望之前，你要意识到：找出当前结点之前的那个结点是可能的。不过在常量时间内完成是不可行的。当你从缓冲区的开头开始按照链表顺序遍历它的每个结点时，最终会发现一个结点，这个结点的link指针刚好和EditorBuffer自身的cursor指针指向的是同一个结点。这个结点就是光标所指向的结点前面的那个结点。只要找到这个结点，你就可以将EditorBuffer的cursor指针指向该结点，这样做和将光标后移的效果是相同的。

你可以通过第12章介绍的传统的for循环来编写寻找光标的程序。然而，这种方法有两个问题。首先，当循环结束后，你需要使用指针变量的值，这也就意味着你要在循环外对这个变量进行声明。其次，如果你用标准的for循环，你会发现这个变量与循环体完全没有任何联系。因为你所关注的仅仅是指针变量的值。包含空循环体的循环语句会给读者一种缺少某种东西的感觉，从而使代码难以阅读。

鉴于这些原因，用while语句编写循环代码就会比较简单，如下所示：

```
Cell *cp = start;
while (cp->link != cursor) {
   cp = cp->link;
}
```

当while循环结束后，cp就指向光标前面的那个结点。随着光标向前移动，你需要防止这个循环试图超出缓冲区的范围，所以moveCursorBackward的完整代码如下所示：

```
void EditorBuffer::moveCursorBackward() {
   if (cursor != start) {
      Cell *cp = start;
      while (cp->link != cursor) {
         cp = cp->link;
      }
      cursor = cp;
   }
}
```

基于同样的原因，你可以仅仅通过将光标前移直至它指向链表的最后一个结点，即发现空值 NULL 为止，因此，完整的 moveCursorToEnd 方法的实现如下所示：

```
void EditorBuffer::moveCursorToEnd() {
   while (cursor->link != NULL) {
      cursor = cursor->link;
   }
}
```

601

13.5.4 缓冲区实现的完善

完整的 EditBuffer 类包含了一些到现在还没有实现的方法：构造函数、析构函数，以及 getText 和 getCursor 方法。在构造函数中，你唯一需要注意的就是要记住空结点的存在。在编码的时候，即使对于空的缓冲区，也必须分配空结点所需的存储空间。然而，一旦你记住这个细节之后，编码就相对容易了：

```
EditorBuffer::EditorBuffer() {
   start = cursor = new Cell;
   start->link = NULL;
}
```

析构函数的实现更为微妙。当析构函数被调用之后，它就有职责释放该类所分配的所有内存。这些内存也包括在链表中分配的各个结点。和之前讨论的 for 循环一样，你可能会编写以下的循环代码：

```
for (Cell *cp = start; cp != NULL; cp = cp->link) {
   delete cp;
}
```



这里的问题是当结点空间被释放以后，这段代码会尝试使用所释放的结点的 link 指针。一旦执行 delete 操作，就不允许再次访问所删除的 cp 所指结点中 link 指针所指向的内容了。这样做很可能会导致错误。为了解决这个问题，你就需要在释放每个结点后，用一个独立的变量记录你当前的位置。本质上，你需要维持一个位置。因此，~EditorBuffer 的正确编码会更为复杂，并且具有如下形式：

```
EditorBuffer::~EditorBuffer() {
   Cell *cp = start;
   while (cp != NULL) {
      Cell *next = cp->link;
      delete cp;
      cp = next;
   }
}
```

602 基于链表的缓冲区类的实现的完整代码如图 13-9 所示。

```
/*
 * File: buffer.cpp (list version)
 * -------------------------------
 * This file implements the EditorBuffer class using a linked
 * list to represent the buffer.
 */

#include <iostream>
#include "buffer.h"
using namespace std;

/*
 * Implementation notes: EditorBuffer constructor
 * ----------------------------------------------
 * The constructor initializes an empty editor buffer represented as
 * a linked list.  In this representation, the empty buffer contains
 * a "dummy" cell whose ch field is never used.  The constructor
 * allocates this dummy cell and then sets the internal pointers.
 */

EditorBuffer::EditorBuffer() {
   start = cursor = new Cell;
   start->link = NULL;
}

/*
 * Implementation notes: EditorBuffer destructor
 * ---------------------------------------------
 * The destructor deletes every cell in the buffer.  Note that the loop
 * structure is not exactly the standard for loop pattern for processing
 * every cell within a linked list.  The complication that forces this
 * change is that the body of the loop can't free the current cell and
 * later have the for loop use the link field of that cell to move to
 * the next one.  To avoid this problem, this implementation copies the
 * link pointer before calling delete.
 */

EditorBuffer::~EditorBuffer() {
   Cell *cp = start;
   while (cp != NULL) {
      Cell *next = cp->link;
      delete cp;
      cp = next;
   }
}

/*
 * Implementation notes: moveCursor methods
 * ----------------------------------------
 * The four methods that move the cursor have different time complexities
 * because the structure of a linked list is asymmetrical with respect to
 * moving backward and forward.  The moveCursorForward and moveCursorToStart
 * methods operate in constant time.  By contrast, the moveCursorBackward
 * and moveCursorToEnd methods each require a loop that runs in linear time.
 */

void EditorBuffer::moveCursorForward() {
   if (cursor->link != NULL) {
      cursor = cursor->link;
   }
}

void EditorBuffer::moveCursorBackward() {
   if (cursor != start) {
      Cell *cp = start;
      while (cp->link != cursor) {
         cp = cp->link;
      }
      cursor = cp;
   }
}
```

图 13-9 基于链表的缓冲区的实现

```
void EditorBuffer::moveCursorToStart() {
   cursor = start;
}

void EditorBuffer::moveCursorToEnd() {
   while (cursor->link != NULL) {
      cursor = cursor->link;
   }
}

/*
 * Implementation notes: insertCharacter
 * -------------------------------------
 * The steps required to insert a new character are:
 *
 * 1. Allocate a new cell and put the new character in it.
 * 2. Copy the pointer indicating the rest of the list into the link.
 * 3. Update the link in the current cell to point to the new one.
 * 4. Move the cursor forward over the inserted character.
 */

void EditorBuffer::insertCharacter(char ch) {
   Cell *cp = new Cell;
   cp->ch = ch;
   cp->link = cursor->link;
   cursor->link = cp;
   cursor = cp;
}

/*
 * Implementation notes: deleteCharacter
 * -------------------------------------
 * The steps necessary to delete the character after the cursor are:
 *
 * 1. Remove the current cell by pointing to its successor.
 * 2. Free the cell to reclaim the memory.
 */

void EditorBuffer::deleteCharacter() {
   if (cursor->link != NULL) {
      Cell *oldCell = cursor->link;
      cursor->link = cursor->link->link;
      delete oldCell;
   }
}

/*
 * Implementation notes: getText and getCursor
 * -------------------------------------------
 * The getText method uses the standard linked-list pattern to loop
 * through the cells in the linked list.  The getCursor method counts
 * the characters in the list until it reaches the cursor.
 */

string EditorBuffer::getText() const {
   string str = "";
   for (Cell *cp = start->link; cp != NULL; cp = cp->link) {
      str += cp->ch;
   }
   return str;
}

int EditorBuffer::getCursor() const {
   int nChars = 0;
   for (Cell *cp = start; cp != cursor; cp = cp->link) {
      nChars++;
   }
   return nChars;
}
```

图 13.9 （续）

13.5.5 基于链表的缓冲区的时间复杂度

从前面的讨论可以看出，在复杂度那张表里加一列是很容易的。可以用这列以缓冲区中

603 ~ 605 字符数目来表示基本编辑操作的代价。这个包含缓冲区三种实现的时间复杂度数据的新表如表 13-4 所示。

表 13-4 三种缓冲区模型的时间复杂度表

操 作	数 组	栈	链 表
moveCursorForward	$O(1)$	$O(1)$	$O(1)$
moveCursorBackward	$O(1)$	$O(1)$	$O(N)$
moveCursorToStart	$O(1)$	$O(N)$	$O(1)$
moveCursorToEnd	$O(1)$	$O(N)$	$O(N)$
insertCharacter	$O(N)$	$O(1)$	$O(1)$
deleteCharacter	$O(N)$	$O(1)$	$O(1)$

遗憾的是，在链表结构的表示中，仍然有 moveCursorBackward 和 moveCursorToEnd 这两个复杂度为 $O(N)$ 的操作。这种表示的问题是在实现的时候链表指针的方向是强制规定的：光标前移比较容易，因为指针的指向就是向前的。

13.5.6 双向链表

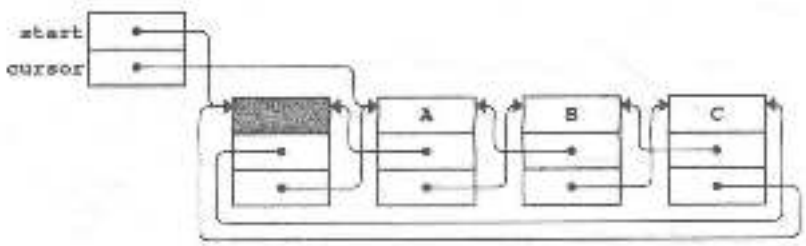
好消息是这个问题比较容易解决。为了解决链表只能向一个方向前进的问题，你需要使指针可以双向地移动。除了让每个结点有一个可以指向下一个结点的指针，你也应该包含一个指向它前一个结点的指针。这种结构被称为**双向链表**（double linked list）。

双向链表的每个结点都有两个指针，一个 prev 指针域指向当前结点的前一个结点，另一个 next 指针域指向其下一个结点。因此当你实现之前的操作时，就会比较清晰。如果空结点的 prev 指针指向缓冲区的最后，并且最后一个结点的 next 指针指向空结点，那么就能简化链表的操作。

如果你采用这种设计，使用双链表实现的缓冲区中若包含以下内容：

A B C

看起来就如下图所示：



606 在这个图中有很多指针，这令人困惑。另一方面，这个结构包含了你所需要的所有信息，这些信息可以让每个基本编辑操作都可以在常数时间内完成。然而，最终的实现就留作一道习题，这样可以强化你对双向链表的理解。

13.5.7 时空权衡

你可以实现 EditorBuffer 类，并且使得标准的编辑操作都可以在常数时间内运行，这是一个很重要的理论性成果。遗憾的是，这个结果在实际使用中可能并不是很有用。至少在编辑器这个应用环境中是这样。当你开始考虑在双向链表的每一个结点都增加一个 prev

指针的时候，你就要至少多使用几个字节的内存来表示每个字符。你可能能够很快地执行编辑操作，但你消耗内存的速度也是惊人的。此时，你就遇到了被计算机科学家称为时空权衡（time-space tradeoff）的问题。你可以提高算法的计算效率，但是这样做是以耗费内存为代价的。耗费内存是很严重的问题，例如，它意味着你在机器上可以编辑的最大文件的大小只是你使用数组表示所占用的内存空间的十分之一。

在现实环境中出现这种情况时，你可能就需要采用一种混合策略。这样可以让你在时空权衡中选择一个中间点。例如，你可以通过将缓冲区按行划分成双向链表，然后每一行又用数组来表示的方式将数组与双向链表策略相结合。这样的话，插入操作在每一行的开始会比较慢，但仅仅是在每一行而不是整个缓冲区域。另一方面，这个策略的每一行需要一个指针而不是每个字符都需要指针。因为一行通常由很多字符构成，使用这种方法就会减少相当多的内存。在混合策略中把所有细节都权衡好是很有挑战的，但是知道存在这种方法也是很重要的。并且也存在既可以提高算法的时间效率也没有使用大量内存的方法。

本章小结

尽管本章的主要内容是对编辑器缓冲区实现一个类的表示，但编辑器本身并不是重点。包含一个光标位置的文本缓冲区只在一个相对较小的应用领域比较有用。用来提高缓冲区表示方法的独特的技巧才是你日后会反复使用的东西。 607

本章的重点内容包括：

- 表示一个类所使用的策略对它操作的时间复杂度有很大影响。
- 尽管数组为编辑器缓冲区提供了一个可用的表示方法，但你可以通过使用其他的表示策略来提高它的性能。例如，使用一对栈，通过以光标移动很长距离为代价从而简化了插入与删除操作。
- 你可以通过给每个结点存储一个指针让它指向该结点的下一个结点的方法来表示结点的先后次序。在编程中，这样设计的结构称为链表。连接下一个值的指针被称为链，用来存储值和指引的单个记录称为结点。
- 标记链表末尾的传统方法是将最后一个结点的指针设为常量 NULL。
- 如果在一个链表中插入和删除值，在链表的开头分配一个空结点是非常方便的。这种方法的优点就在于：空结点的存在减少了我们在代码中要考虑的特殊情况的数目。
- 在链表的一个特定的位置进行插入和删除操作所用的时间是常数时间。
- 你可以通过如下的方式对链表的结点进行迭代操作：

```
for (Cell *cp = list; cp != NULL; cp = cp->link) {
    . . . 代码使用 cp . . .
}
```

- 双向链表可以使我们在前后两个方向有效地遍历链表。
- 链表在执行时间上比较高效，但比较浪费内存。在有些情况下，你可能需要采用混合策略进行设计。这个策略允许你将节约内存的数组策略和执行比较高效的链表策略相结合。

复习题

1. 判断题：程序的时间复杂度仅仅取决于算法结构，与数据的表示结构无关。

608 2. wysiwyg 是什么意思？

3. 用自己的语言描述本章使用缓冲区抽象的目的。

4. 编辑器应用实现的六个命令是什么？在 EditorBuffer 类中相对应的公有方法是什么？

5. 除了和编辑器命令中相对应的方法以外，EditorBuffer 类还提供哪些其他的公有操作？

6. 在用数组表示的编辑器缓冲区中，哪种编辑操作所用的时间是线性的？导致这些操作缓慢的原因是什么？

7. 如下图所示，有一块缓冲区域，它的内容如下：

A B C D E F G H I J

光标位置如图所示。用两个栈表示这块缓冲区域。画图表示 before 和 after 栈中的内容。

8. 用两个栈表示的编辑器缓冲区中的光标位置是如何表示的？

9. 在两个栈表示的编辑器缓冲区中，哪种编辑操作所用的时间是线性的？

10. 在用链表表示编辑器缓冲区时，使用辛结点的目的是什么？

11. 为什么辛结点会处于链表的开始或末尾？

12. 将一个新的字符插入到用链表表示的缓冲区中，需要进行哪五步？

13. 一块缓冲区域包含如下内容：

H E L L O

光标位置如图所示，且该缓冲区用链表表示。画一个图表示该链表的每一个结点。

14. 如果在光标的位置插入字母 x，修改之前画的图，看看有何变化？

15. 遍历链表是什么意思？

16. 用 C++ 遍历链表的标准模式是什么？

17. 在编辑器缓冲区的链表表示中，哪种编辑操作所用的时间是线性的？导致这些操作比较慢的原因是什么？

609 18. 时空权衡是什么意思？

19. 你可以对链表结构做何种修改，才能使得其执行六个操作的时间都为常数时间？

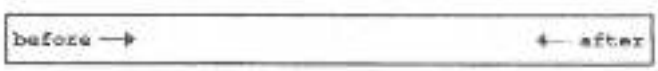
20. 上一题所给的答案有什么主要的缺点？你可以如何对其进行改进？

习题

1. 尽管 SimpleTextEditor 应用在说明编辑器的工作原理时比较有用，但它不是一个理想的测试程序，因为它依赖于用户明确的输入。为 EditorBuffer 类设计和实现一个单元测试，并且要求该单元测试能测试实现过程中的所有可能错误。

2. 尽管基于两个栈实现的 EditorBuffer 类中的栈是动态扩展的，但是栈中字符所需的空间是相应数组实现的两倍。导致这个问题的主要原因是两个栈都要能够放入缓冲区中的所有字符。例如，假设你在使用一个总有 N 个字符的缓冲区。如果光标在缓冲区开头，那么所有字符都要放入 after 栈；如果光标移动到缓冲区最末尾，这些字符又需要移动到 before 栈。因此，每个栈的容量至少为 N。

   你可以通过将两个栈存入一个数组的相反方向来减少内存的使用。before 栈放在数组的开头，after 栈放在数组的末尾。然后两个栈就会按照图中箭头所示的方向增长：



   用这种表示方法重新实现 EditorBuffer 类（实际上，现在很多编辑器都使用这种设计方法）。确保你的程序和文中两个栈的实现有相同的计算效率，并且缓冲区空间是随着需要动态增长的。

3. 如果使用一个实际的编辑器应用，你可能希望程序展示缓冲区的内容，而不是每一个命令。改变 SimpleTextEditor 应用的实现，使它不再展示每个命令之后的缓冲区，而是提供一个 t 命令输
610 出内容。与图 13-3 中包含的 displayBuffer 函数相反，t 命令仅仅以字符串的形式输出缓冲区

的内容，不需要显示光标的位置。新的编辑器应用的一个运行示例如下图所示：

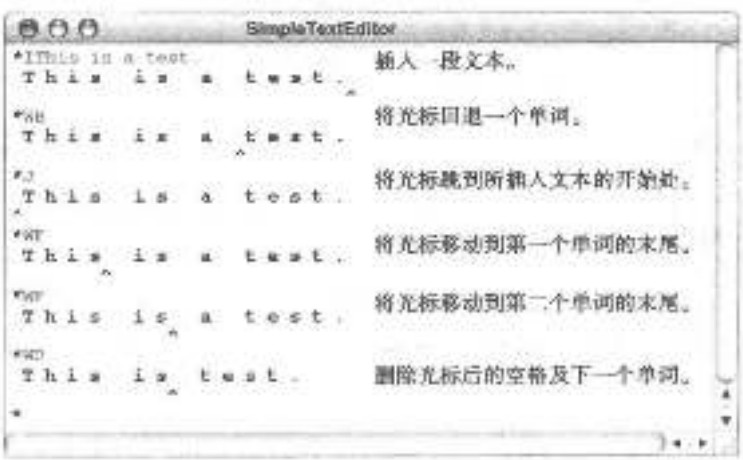


4. SimpleTextEditor 应用中一个很重要的限制就是不能在缓冲区里插入换行符。因此就不能写入多于一行的数据。从习题 3 开始，编辑器中增加了一个 A 命令，它可以读取随后多行的文本，和 UNIX 系统中的 ed 编辑器一样，当用户输入一个单独句号表示结束。这种版本的编辑器的一个示例运行可能如下图所示：



5. 重写图 13-3 所示的编辑器应用。使得在 F、B 和 D 命令之前可以添加一个数字，以便这些命令重复执行由该数字指定的次数。例如，命令 17F 将会把光标向前移动 17 个字符位置。

6. 扩展编辑器应用，使得 F、B 和 D 命令可以出现在字母 W 之前表示他们是移动操作。因此，WF 命令 611
就是将光标移动到下一个单词的末尾，WB 表示将光标返回到上一个单词的开头，WD 表示删除下一个单词。为了实现这个练习，假设一个单词由连续的字母或数字序列组成，并且在光标和单词之间可包含任意的非字母字符。用下面的例子就能很容易理解：



实现这些命令最直接的方法就是扩充 EditorBuffer 类，使它可以导出那些完成基于单词操作的方法。这些方法是由你设计的。用本章介绍的三种缓冲区表示法分别实现这些扩展。

7. 大多数现代编辑器提供了一种机制，它允许用户把缓冲区文本的一部分复制到内存，然后再把它粘贴到其他地方。本章所给的三种表示缓冲区的方法，以下实现方法：

```
void EditorBuffer::copy(int count);
```

这个方法保存了缓冲区内部结构中某处的字符，并且以下方法：

```
void EditorBuffer::paste();
```

可以将那些保存的字符插入到当前光标的位置。调用 paste 方法不会影响保存的文本，这就意味着你可以通过多次调用 paste 将其插入多次。通过在编辑器中添加 C 和 P 命令来表示复制和粘贴
612 操作，然后利用这两个操作来测试你的实现。同样，和习题 5 中使用的方法相同，C 命令需要获取一个指定字符数目的数值参数。

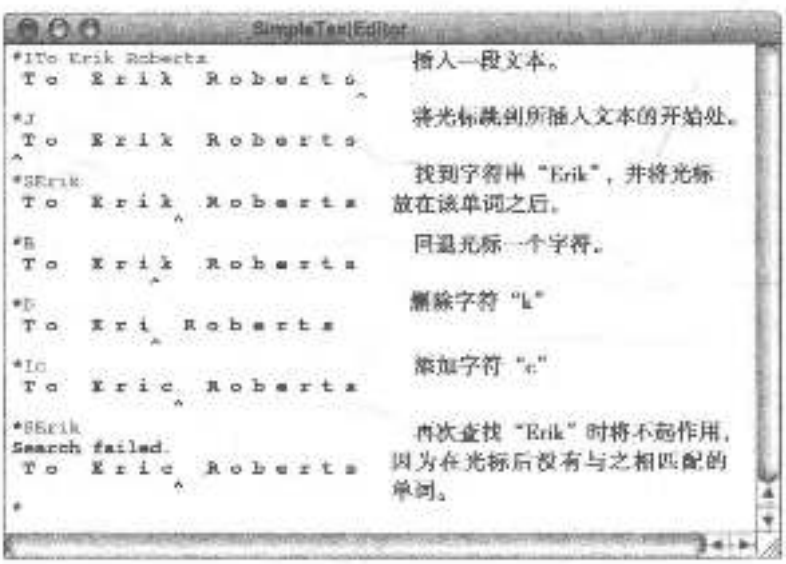
8. 之前习题描述的支持复制－粘贴机制的编辑器通常会提供第三种名为剪切的操作。剪切操作可以对缓冲区的内容先进行复制然后再删除。实现一个新的编辑器命令 X，在对习题 7 中的 EditorBuffer 类的接口不做任何修改的情况下实现剪切操作。

9. 用本章所给的一种表示缓冲区方法中的任一种实现以下方法：

```
bool EditorBuffer::search(string str);
```

当这个方法被调用时，它就从当前的光标位置开始搜索，寻找下一个字符串 str 的位置。如果找到，光标就会放在字符串 str 的最后一个字符后面，并且返回 true，否则光标位置保持不变，返回 false。

为了说明 search 操作，假设你需要在 editor.cpp 程序中添加命令 S 来调用 search 方法，就会将它传入到输入行的剩余部分。你的程序运行结果要和下面例子中程序运行的结果相符：



613

10. 在对习题 9 中的 EditorBuffer 类的接口不做任何改变的前提下，在编辑器的应用中添加一个 R 命令。当两个字符串中间用斜线隔开时，该命令可以用斜线后面的字符串代替斜线前面的字符串。如下图所示：



11. 书中介绍的空结点策略是非常有用的，因为它减少了代码中特殊情况发生的数目，但是严格来说并不必要的。编写一个新的链表版本的 buffer.cpp 实现，要做到如下改变：
   - 链表中不存在空结点，每个结点只允许存储一个字符。
   - 光标处在第一个字符之前的缓冲区表明了 cursor 指针存储的是 NULL。
   - 每个检查光标位置的方法在当前情况下的任何操作都要对 NULL 做一次特殊的检查。
12. 用 13.5.6 节中所说的双向链表方法实现 EditorBuffer 类。尽可能彻底地测试你的程序，尤其要保证你在当前进行插入和删除的地方可以让光标向两个方向移动。 614

第 14 章
Programming Abstractions in C++

线性结构

我从来都没有遇到过像那样很高的线；它有一两个弯，但是这没什么大不了的。
615 ——简·奥斯汀，《劝导》(*Persuasion*)，1818

在第 5 章介绍的 Stack、Queue 和 Vector 类都被称为**线性结构**（linear structure）的一类抽象数据类型的实例，线性结构中的元素都以线性的顺序排列。本章着眼于这些类型的几种可能表示方法，并思考表示方法的选择对效率的影响。

由于一个线性结构中的元素是按像数组顺序排列的，使用数组表示它们似乎是一个显而易见的选择。的确，第 12 章展示的 CharStack 类是通过使用一个数组作为基本表示而实现的。然而，数组并不是唯一的选择。栈、队列和矢量也可以使用一种类似第 13 章用于实现编辑器缓冲区的链表来实现。通过学习这些结构的链表实现方法，你不仅能提高对链表如何工作的理解，也能提高对如何能将它们应用到实际编程情况中的理解。

本章还有另外一个目的。正如你从第 5 章了解到的，容器类（不像第 12 章的 CharStack 类）并不局限于单一的数据类型。实际的 Stack 类允许用户通过提供一个类型参数来指定基类型，正如 Stack<int> 和 Stack<Point> 一样。然而，到目前为止，作为用户的你只有使用参数化类型的机会。而在本章，你将学习如何实现它们。

14.1 模板

在计算机科学中，能将相同的代码应用到多种数据类型上的方式称为**多态**（polymorphism）。程序语言可以用多种方式实现多态。C++ 使用了一种被称为**模板**（template）的方法，使用这种方法，程序员定义了一个共同的代码模式之后，这个模式就可以被用于许多不同的类型。第 5 章的容器类依赖于 C++ 模板的功能，这意味着在你能够领会容器类的底层实现方法之前，需要懂得模板是如何工作的。

在详细地理解模板之前，重温第 2 章介绍的**重载**概念是很有帮助的。只要这些函数可以通过它们的参数加以区分，重载就允许你用相同的名字定义不同的函数。给定一个特定的函数调用，编译器通过观察实参的数目和类型以选择和签名相匹配的函数版本。

作为一个例子，你可以使用下面的代码来定义名为 max 函数的两个版本（一个用于整
616 数值，另一个用于浮点数值），它返回两个参数中较大者。

```
int max(int x, int y) {
   return (x > y) ? x : y;
}

double max(double x, double y) {
   return (x > y) ? x : y;
}
```

这两个函数体是完全相同的。唯一不同的是参数签名。如果用户编写了函数调用：

```
max(17, 42)
```

则编译器注意到它的两个实参都是整数，因此会发出调用 max 形参为整数的函数版本，它将返回一个 int 类型的结果。相比之下，调用以下函数：

```
max(3.14159, 2.71828)
```

将产生一个浮点数版本 max 函数的调用，它将返回一个 double 类型的结果。

当只有函数参数的数据类型不同时，模板才使得自动函数重载变为可能。在 C++ 中，你可以将前面的 max 函数定义合并成一个单独的模板定义，如下所示：

```
template <typename ValueType>
ValueType max(ValueType x, ValueType y) {
   return (x > y) ? x : y;
}
```

在上述定义中，标识符 ValueType 对于 max 调用使用的参数类型来说是一个占位符。关键字 typename 告诉 C++ 编译器这个占位符表示一个类型的名字，它使得编译器能够正确地解释该标识符。

一旦你定义了该函数模板，就可以将它应用到任何的基本类型中。例如，如果编译器遇到以下函数调用：

```
max('A', 'Z')
```

它会自动地产生 max 的一个字符版本，如下所示：

```
char max(char x, char y) {
   return (x > y) ? x : y;
}
```

617

然后编译器就能将一个调用插入到这个新创建的 max 版本中，它会正确地返回字符 'Z' 作为表达式 max('A', 'Z') 的值。

max 函数模板版本适用于任何定义了 > 操作符的数据类型。例如，如果你扩展第 6 章习题 7 描述的 Rational 类，使得它包含比较操作符，你就可以使用 max 函数选择两个 Rational 对象中较大的那个。模板意味着在新数据类型上使用 max 函数是无需编写更多代码的。

然而，有一些地方需要小心。假设运行在机器上的编译器调用：

```
max("cat", "dog")
```



正好返回 "cat"，这看起来和逻辑结果相反。这里的问题是字面值 "cat" 和 "dog" 都是 C 字符串而不是 C++ 字符串。这意味着它们是指向字符的指针。因此，C++ 编译器产生了或多或少无用的函数：

```
char *max(char *x, char *y) {
   return (x > y) ? x : y;
}
```

该函数将返回存储在内存中更高地址的 C 字符串的地址。相比于 "dog" 中的字符，编译器将 "cat" 中的字符存储在更高的地址中，然后 "cat" 将作为最大值返回。为了使用关于 C++ 字符串定义的字符串比较操作符，你需要编写如下的函数调用：

```
max(string("cat"), string("dog"))
```

它会正确地返回 C++ 字符串 "dog"。

值得注意的一个有趣事实是：实际上，相比于单独定义函数的重载版本的选择策略，模板的功能并没有节省任何空间。编译器无论何时遇到一个它没有见过的类型函数模板的应用，都会产生一个全新的相应数据类型的函数拷贝。因此，如果你使用相同的程序将 max 函数应用到 int、double、char、string 和 Rational 类型中，编译器将产生 5 个函数代码的拷贝，每种类型一个。这种实现策略强调了为什么单词 "template" 是如此的合适。在 C++ 中，你不能定义单一的函数用于多种数据类型，而是应该提供一个模板，从中编译器会产生它们需要的专门定制的版本。

编译器需要创建模板代码的多种拷贝，作为一个程序员，这个事实对你有重要的意义。
618 在 C++ 中，当编译器遇到一个函数模板调用时，该模板必须能够实现。原型本身是不够的。这个约束意味着你不能将接口和一个函数模板的实现分开，本书中的程序通常将原型放在 .h 文件中，而将相应的实现放在 .cpp 文件中。如果你想输出一个函数模板作为库的一部分，对于编译器来说，当 .h 文件被读取时，函数实现必须是有效的。

14.2 栈的实现

第 12 章中的 CharStack 类定义了各种类型的栈所需要的相关操作，但是它有这样一个限制：它只能存储 char 类型的元素。为了获得 Stack 类库版本的灵活性，有必要将 Stack 类作为一个模板类（template class）重新实现，模板类是一个使用 C++ 模板机制的类，它可以适用于任意数据类型。

创建一个已有的类模板形式涉及一些简单的语法变换。例如，如果你想使用一个通用的 Stack 模板来更新第 12 章的 CharStack 类，你首先需要将名字 CharStack 用 Stack 替换，然后在类定义前添加以下一行语句：

```
template <typename ValueType>
```

template 关键字表示这一行后面的整个语法单位（在这种情况下，类的定义）是模板模式的一部分。模板模式可以被用于 ValueType 参数的许多不同值。例如，在 Stack 类的代码中，涉及存储元素类型的地方需要使用名为 ValueType 的占位符。因此，当你将 CharStack 类定义转换成它的更一般的模板形式时，你必须将每一个出现的特定类型 char 用通用的占位符 ValueType 替换。

stack.h 接口更新后的版本显示在图 14-1 中，它只包括一些独立于策略实现之外的公共定义。这些定义毕竟只是用户感兴趣的接口文件的一部分。正如第 13 章 buffer.h 接口列举的，Stack 类私有部分是作为一个框出现的，之后它将被适合于特定表示的定义所代替。然而，Stack 是一个模板类的事实也意味着编译器必须使用该实现以及类定义本身。因此，图 14-1 是以第二个框结束的，它最终将被相对应的选择表示的实现所替换。在下面的章节中，这些框将被栈的两种不同的基本表示所替换：一个使用动态数组表示；另一个则
619 使用链表。

14.2.1 使用动态数组实现栈

实现栈类模板的最简单方法是采用第 12 章 CharStack 类使用的动态数组模型。正如前

```
/*
 * File: stack.h
 * -------------
 * This interface exports a template version of the Stack class.
 */

#ifndef _stack_h
#define _stack_h

#include "error.h"

/*
 * Class: Stack<ValueType>
 * -----------------------
 * This class implements a stack of the specified value type.
 */

template <typename ValueType>
class Stack {

public:

/*
 * Constructor: Stack
 * Usage: Stack<ValueType> stack;
 * ------------------------------
 * Initializes a new empty stack.
 */

   Stack();

/*
 * Destructor: ~Stack
 * Usage: (usually implicit)
 * -------------------------
 * Frees any heap storage associated with this stack.
 */

   ~Stack();

/*
 * Method: size
 * Usage: int n = stack.size();
 * ----------------------------
 * Returns the number of values in this stack.
 */

   int size() const;

/*
 * Method: isEmpty
 * Usage: if (stack.isEmpty()) . . .
 * ---------------------------------
 * Returns true if this stack contains no elements.
 */

   bool isEmpty() const;

/*
 * Method: clear
 * Usage: stack.clear();
 * ---------------------
 * Removes all elements from this stack.
 */

   void clear();

/*
 * Method: push
 * Usage: stack.push(value);
 * -------------------------
 * Pushes the specified value onto this stack.
 */

   void push(ValueType value);

/*
 * Method: pop
 * Usage: ValueType top = stack.pop();
```

图 14-1 关于多态栈抽象的接口

```
 * ------------------------------------
 * Removes the top element from this stack and returns it.  This
 * method signals an error if called on an empty stack.
 */

   ValueType pop();

/*
 * Method: peek
 * Usage: ValueType top = stack.peek();
 * ------------------------------------
 * Returns the value of top element from this stack without removing
 * it.  This method signals an error if called on an empty stack.
 */

   ValueType peek() const;

/*
 * Copy constructor and assignment operator
 * ----------------------------------------
 * These methods implement deep copying for stacks.
 */

   Stack(const Stack<ValueType> & src);
   Stack<ValueType> & operator=(const Stack<ValueType> & src);

   The private section of the class goes here.

};

   The implementation of the class goes here.

#endif
```

图 14-1 （续）

面的例子所示，Stack 的底层实现必须跟踪动态数组的元素、数组的大小和当前元素的个数。类的私有部分包括这些变量的声明，以及一个私有的定义初始大小的常量和私有方法的原型。关于基于数组实现的栈的私有部分显示在图 14-2 中。

```
/* Private section */

/*
 * Implementation notes
 * --------------------
 * This version of the stack.h interface uses a dynamic array to store
 * the elements of the stack.  The array begins with INITIAL_CAPACITY
 * elements and doubles the size whenever it runs out of space.  This
 * discipline guarantees that the push method has O(1) amortized cost.
 */

private:

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   ValueType *array;                    /* A dynamic array of the elements */
   int capacity;                        /* The allocated size of the array */
   int count;                           /* The number of stack elements    */

/* Private method prototypes */

   void deepCopy(const Stack<ValueType> & src);
   void expandCapacity();
```

图 14-2 基于数组的栈私有部分

模板类对一般类型更加适用，用 ValueType 替换 char 和变换变量名在类型转换过程中是一个很大的问题，例如，在 CharStack 实现中，将原有变量名 ch 处替换为名为

value 将更有意义。在 Stack 类的模板版本中唯一实质的变化是：实现了拷贝构造函数和赋值操作符的重载版本，从而确保了用户可以拷贝 Stack 类型的值。stack.h 的实现部分显示在图 14-3 中。

620 ~ 622

```
/*
 * Implementation section
 * ----------------------
 * C++ requires that the implementation for a template class be availabl
 * to the compiler whenever that type is used.  The effect of this
 * restriction is that header files must include the implementation.
 * Clients should not need to look at any of the code beyond this point.
 */

/*
 * Implementation notes: Stack constructor
 * ---------------------------------------
 * The constructor allocates the array storage for the stack elements
 * and initializes the fields of the object.
 */

template <typename ValueType>
Stack<ValueType>::Stack() {
   capacity = INITIAL_CAPACITY;
   array = new ValueType[capacity];
   count = 0;
}

/*
 * Implementation notes: ~Stack
 * ----------------------------
 * The destructor frees any heap memory allocated by the class, which
 * is just the dynamic array of elements.
 */

template <typename ValueType>
Stack<ValueType>::~Stack() {
   delete[] array;
}

template <typename ValueType>
int Stack<ValueType>::size() const {
   return count;
}

template <typename ValueType>
bool Stack<ValueType>::isEmpty() const {
   return count == 0;
}

template <typename ValueType>
void Stack<ValueType>::clear() {
   count = 0;
}

/*
 * Implementation notes: push
 * --------------------------
 * This function first checks to see whether there is enough room for
 * the value and then expands the array storage if necessary.
 */

template <typename ValueType>
void Stack<ValueType>::push(ValueType ch) {
   if (count == capacity) expandCapacity();
   array[count++] = ch;
}

/*
 * Implementation notes: pop, peek
 * -------------------------------
 * These functions checks for an empty stack and reports an error
 * if there is no top element.
 */
```

图 14-3 基于数组的栈的实现

```
template <typename ValueType>
ValueType Stack<ValueType>::pop() {
   if (isEmpty()) error("pop: Attempting to pop an empty stack");
   return array[--count];
}

template <typename ValueType>
ValueType Stack<ValueType>::peek() const {
   if (isEmpty()) error("peek: Attempting to peek at an empty stack");
   return array[count - 1];
}

/*
 * Implementation notes: copy constructor and assignment operator
 * --------------------------------------------------------------
 * These methods follow the standard template, leaving the work to deepCopy.
 */

template <typename ValueType>
Stack<ValueType>::Stack(const Stack<ValueType> & src) {
   deepCopy(src);
}

template <typename ValueType>
Stack<ValueType> & Stack<ValueType>::operator=(const Stack<ValueType> & src) {
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}

/*
 * Implementation notes: deepCopy
 * ------------------------------
 * This function copies the data from the src parameter into the current
 * object.  All dynamic memory is reallocated to create a "deep copy" in
 * which the current object and the source object are independent.
 * The capacity is set so that the stack has some room to expand.
 */

template <typename ValueType>
void Stack<ValueType>::deepCopy(const Stack<ValueType> & src) {
   capacity = src.count + INITIAL_CAPACITY;
   this->array = new ValueType[capacity];
   for (int i = 0; i < src.count; i++) {
      array[i] = src.array[i];
   }
   count = src.count;
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This private method doubles the capacity of the elements array whenever
 * it runs out of space.  To do so, it copies the pointer to the old array,
 * allocates a new array with twice the capacity, copies the values from
 * the old array to the new one, and finally frees the old storage.
 */

template <typename ValueType>
void Stack<ValueType>::expandCapacity() {
   ValueType *oldArray = array;
   capacity *= 2;
   array = new ValueType[capacity];
   for (int i = 0; i < count; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}
```

图 14-3 （续）

14.2.2 使用链表实现栈

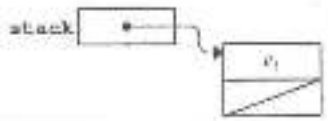
尽管数组是栈最常见的底层表示，但是也可以使用链表来实现 Stack 类。如果你这样

做。空栈的概念表示则仅仅是一个 NULL 指针：

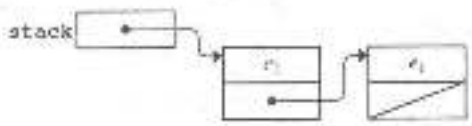
623
/
625



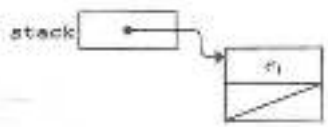
当你将一个新元素压入栈中时，元素仅仅是被添加到链表链的前面。因此，如果你将元素 e_1 压入进一个空栈中，该元素将被存储在一个新的节点中，并成为这个链表中的唯一链接。



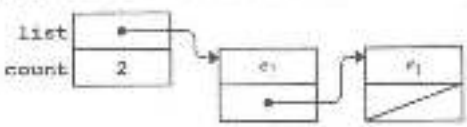
将一个新元素压入栈，该元素会被添加到链首部。这个步骤所涉及的与将一个字符插入到链表缓存区所需的操作一样。你首先需分配一个新的空间，然后输入数据，最后更新链表指针。这样新的空间会变成链表的第一个元素。因此，如果你将元素 e_2 压入栈中，将获得以下结构：



在链表表示中，pop 操作包含删除链表中的第一个元素并返回其值。因此，上图所示的一个 pop 操作将返回 e_2，并回到栈之前的状态，如下图所示：



尽管在栈的底层实现中可以使用一个指针指向链表头部，但是这样会对 size 方法的效率产生不良影响。如果栈类中只存储该链表，唯一确定链表长度的方法便是遍历链表中的元素，直到你发现链表尾的 NULL 指针为止。这个过程需要 $O(N)$ 时间复杂度。为了确保 size 方法在常量时间内运行，最简单的方法是使用一个单独的实例变量来跟踪其元素的个数。采用这种设计意味着类的私有部分声明了两个实例变量：一个表头指针和一个元素个数计数器。一个包含两个元素的栈的更完整图如下图所示：



626

基于链表的栈接口修改的私有部分的内容显示在图 14-4 中。一旦你定义了这种数据结构，就可以继续重写 Stack 类方法，使得它们在新的数据表示方法中起作用。stack.h 接口的基于链表版本的实现显示在图 14-5 中。

```
/* Private section */

private:

/*
 * Implementation notes
 * --------------------
 * This version of the stack.h interface uses a linked list to store
```

图 14-4 基于链表的栈的私有部分

```
 * the elements of the stack.  The top item is always at the front of
 * the linked list and is therefore always accessible without searching.
 * The instance variable count keeps track of the number of elements so
 * that the size method runs in constant time.
 */

/* Type for linked list cell */

   struct Cell {
      ValueType data;                  /* The data value                */
      Cell *link;                      /* Link to the next cell         */
   };

/* Instance variables */

   Cell *list;                         /* Initial pointer in the list   */
   int count;                          /* Number of elements            */

/* Private method prototypes */

   void deepCopy(const Stack<ValueType> & src);
```

图 14-4 （续）

关于该实现，有几个方面值得特别提及。一如既往，构造函数负责建立对象的初始状态。初始状态包括一个空链表和一个值为 0 的元素计数器。图 14-5 中的构造函数明确地初始化了这些方面，如下所示：

```
template <typename ValueType>
Stack<ValueType>::Stack() {
   list = NULL;
   count = 0;
}
```

627

```
/*
 * Implementation section
 * ----------------------
 * C++ requires that the implementation for a template class be available
 * to the compiler whenever that type is used.  Clients should not need
 * to look at any of the code beyond this point.
 */

/*
 * Implementation notes: Stack constructor
 * ---------------------------------------
 * The constructor creates an empty linked list and initializes the count.
 */

template <typename ValueType>
Stack<ValueType>::Stack() {
   list = NULL;
   count = 0;
}

/*
 * Implementation notes: Stack destructor
 * --------------------------------------
 * The destructor frees any heap memory that is allocated by the
 * implementation.  Because clear frees each element it processes,
 * this implementation of the destructor simply calls that method.
 */

template <typename ValueType>
Stack<ValueType>::~Stack() {
   clear();
}

/*
```

图 14-5 基于链表的栈的实现

```
/*
 * Implementation notes: size, isEmpty
 * -----------------------------------
 * These methods use the count variable and therefore run in constant time.
 */

template <typename ValueType>
int Stack<ValueType>::size() const {
   return count;
}

template <typename ValueType>
bool Stack<ValueType>::isEmpty() const {
   return count == 0;
}

/*
 * Implementation notes: clear
 * ---------------------------
 * This method pops the stack until it is empty, thereby freeing each cell.
 */

template <typename ValueType>
void Stack<ValueType>::clear() {
   while (count > 0) {
      pop();
   }
}

/*
 * Implementation notes: push
 * --------------------------
 * This method chains a new element onto the front of the list where it
 * becomes the top of the stack.
 */

template <typename ValueType>
void Stack<ValueType>::push(ValueType value) {
   Cell *cp = new Cell;
   cp->data = value;
   cp->link = list;
   list = cp;
   count++;
}

/*
 * Implementation notes: pop, peek
 * -------------------------------
 * These methods check for an empty stack and report an error if
 * there is no top element.  The pop method frees the cell to ensure
 * that the implementation does not leak memory as it executes.
 */

template <typename ValueType>
ValueType Stack<ValueType>::pop() {
   if (isEmpty()) error("pop: Attempting to pop an empty stack");
   Cell *cp = list;
   ValueType result = cp->data;
   list = list->link;
   count--;
   delete cp;
   return result;
}

template <typename ValueType>
ValueType Stack<ValueType>::peek() const {
   if (isEmpty()) error("peek: Attempting to peek at an empty stack");
   return list->data;
}

/*
 * Implementation notes: copy constructor and assignment operator
 * --------------------------------------------------------------
 * These methods follow the standard template, leaving the work to deepCopy.
 */
```

图 14-5 （续）

```
template <typename ValueType>
Stack<ValueType>::Stack(const Stack<ValueType> & src) {
   deepCopy(src);
}

template <typename ValueType>
Stack<ValueType> & Stack<ValueType>::operator=(const Stack<ValueType> & src) {
   if (this != &src) {
      clear();
      deepCopy(src);
   }
   return *this;
}

/*
 * Implementation notes: deepCopy
 * ------------------------------
 * The deepCopy method creates a copy of the cells in the linked list.
 * The variable tail keeps track of the last cell in the chain.
 */

template <typename ValueType>
void Stack<ValueType>::deepCopy(const Stack<ValueType> & src) {
   count = src.count;
   Cell *tail = NULL;
   for (Cell *cp = src.list; cp != NULL; cp = cp->link) {
      Cell *ncp = new Cell;
      ncp->data = cp->data;
      if (tail == NULL) {
         list = ncp;
      } else {
         tail->link = ncp;
      }
      tail = ncp;
   }
   if (tail != NULL) tail->link = NULL;
}
```

628
}
630

图 14-5 （续）

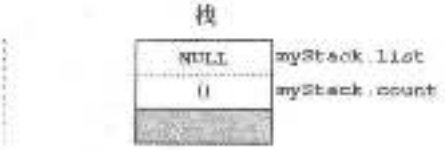
push 方法展示了将一个新结点添加到链表头部的标准模式：

```
template <typename ValueType>
void Stack<ValueType>::push(ValueType value) {
   Cell *cp = new Cell;
   cp->data = value;
   cp->link = list;
   list = cp;
   count++;
}
```

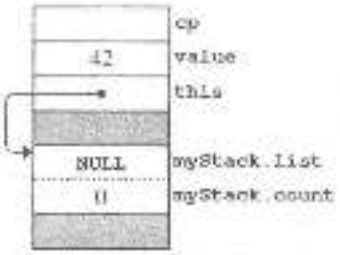
这个模式十分重要，它值得采用堆－栈图来追历上述步骤。假设你正在执行以下主程序：

```
int main() {
   Stack<int> myStack;
   myStack.push(42);
   cout << myStack.pop() << endl;
   return 0;
}
```

当执行过程到达 push 调用时，堆中还没有被分配任何内存。堆－栈图看起来如下图所示：

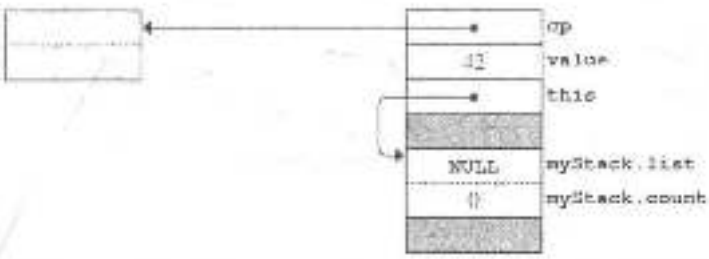


通过指向当前对象的 this 关键字，参数变量 value 和局部变量 cp，调用 myStack.push(42) 将创建一个新的栈帧。这些参数被初始化后，堆-栈图看起来如下图所示：



一旦新的栈帧准备就绪，push 方法必须分配一个新的空间存储值。push 方法中的第一行在堆中创建了一个新的 Cell 结构空间，并将它的地址赋给变量 cp，得到了以下图：

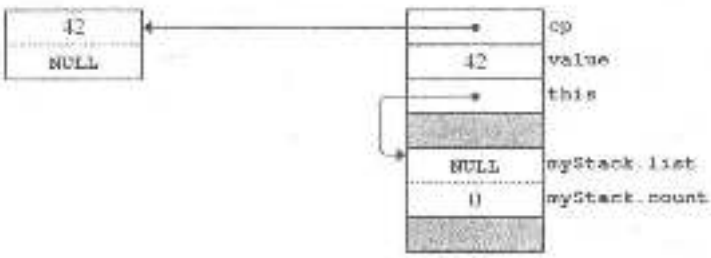
631



接下来的两行填写这个新分配空间的内容。value 域仅仅给调用者提供值。link 域是指向这个结点的后继结点的初始化指针。在本例中链表为空，但是在 push 调用时依然会使用 Stack 对象中的 list 域。因此，语句

```
cp->link = list;
```

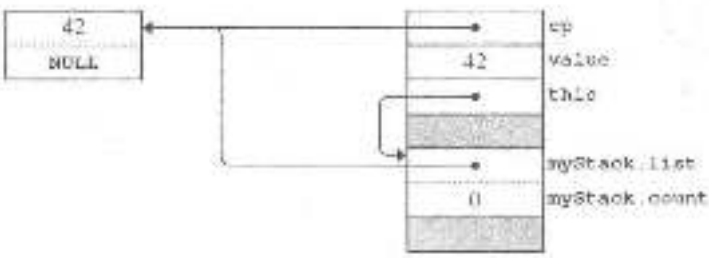
确保了这个结点出现在现有链表的头部。如下图所示：



下一步包括更新 Stack 对象的 list 域，这样链表就以新分配的空间开始。语句

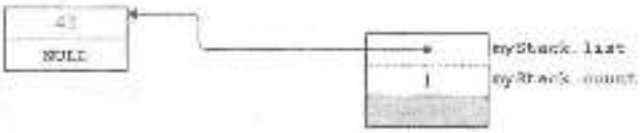
```
list = cp;
```

使内存空间看起来如下图所示：

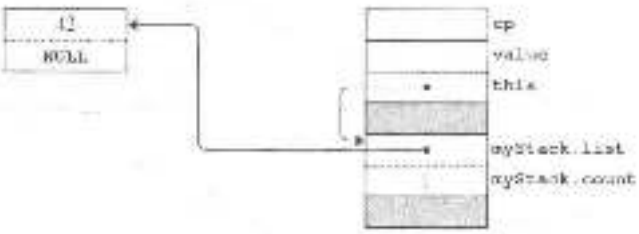


632

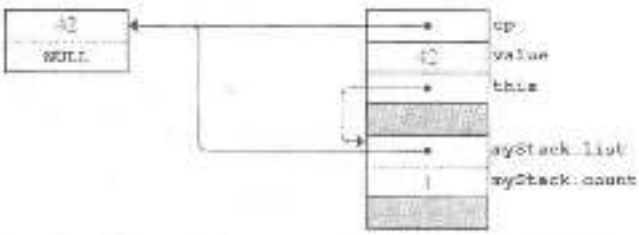
从那里，push 方法增加 count 域并返回至主程序，这导致了以下内存图：



pop 方法刚好和这个过程相反。在当前状态调用 myStack.pop() 创建一个新的栈帧，它和 push 方法对应的栈帧基本上是相同的，只是 value 当前是一个局部变量而不是一个参数：



在确保栈不为空后，pop 方法开始的两条赋值语句将指向栈顶的指针和栈顶的值复制给栈帧的局部变量，如下图所示：



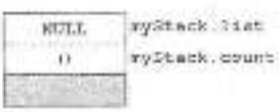
由于 pop 方法必须释放结点所占用的内存空间，因此需要局部变量保存结点的值，这样，当 Stack 对象运行时，它不会消耗越来越多的内存。因为在结点内存空间被释放后再查看其中的内容是不合法的，故弹出值必须存储在局部变量中，这样方法可以返回它。

pop 实现方法的核心语句是：

```
list = list->link;
```

它使用当前结点后的子链表代替 Stack 对象的链表。此例中子链为空。将这个值赋给栈的 list 部分，释放空间，从方法中返回，从而得到以下的最终状态，它对应于一个空栈：

633



14.3 队列的实现

正如你从第 8 章中所了解到的，栈和队列的结构非常类似。它们之间的唯一差别是元素的处理顺序不同。栈使用一种被称为后进先出（LIFO）的原则来处理元素，即最后入栈的元素总是第一个出栈。而队列采用了一种称为先进先出（FIFO）的模式来处理元素，即它更像是一条等待队列。栈和队列的接口也较为相似。两个接口的公有部分的唯一变化是定义类行为的两个方法的名字。来自 Stack 类的 push 方法现在被称为 enqueue，pop 方法被称为

dequeue。这些方法的行为也是不同的，这也反映在图 14-6 所示的 queue.h 接口的注释中。

鉴于这些结构和它们的接口概念上的相似性，栈和队列都能使用基于数组或者基于链表的策略来实现，我们对此不应感到惊讶。然而，使用这些模型时，一个队列的实现方法有微妙的区别，它们并不出现在栈的实例中。这些不同点基于这样一个事实：栈中的所有操作发生在内部数据结构的末尾。在一个队列中，enqueue 操作发生在内部结构的末尾，而 dequeue 操作则发生在另一端。

14.3.1 基于数组的队列实现

一个队列的行为不再限制于数组的末尾，根据这个事实，你还需要两个参数跟踪队列的头部和末尾。因此，队列类的私有的实例变量如下所示：

```
ValueType *array;
int capacity;
int head;
int tail;
```

在上述表示中，head 域拥有下一个将出队的队列中头元素的索引，而 tail 域拥有队列末尾元素的索引。很显然，在一个空队列中 tail 域应该为 0，它表示数组的初始位置，但是 head 域的值为多少？为了方便，通常的策略也是设置 head 域的值也为 0。当队列采用这种方法定义时，head 和 tail 相等，并表示队列为空。 634

```
/*
 * File: queue.h
 * -------------
 * This interface exports a template version of the Queue class.
 */

#ifndef _queue_h
#define _queue_h

#include "error.h"

/*
 * Class: Queue<ValueType>
 * -----------------------
 * This class implements a queue of the specified value type.
 */

template <typename ValueType>
class Queue {

public:

/*
 * Constructor: Queue
 * Usage: Queue<ValueType> queue;
 * ------------------------------
 * Initializes a new empty queue.
 */
   Queue();

/*
 * Destructor: ~Queue
 * Usage: (usually implicit)
 * -------------------------
 * Frees any heap storage associated with this queue.
 */
   ~Queue();

/*
 * Method: size
 * Usage: int n = queue.size();
```

图 14-6 关于多态队列抽象的接口

```
 * ------------------------------
 * Returns the number of values in the queue.
 */
   int size() const;

/*
 * Method: isEmpty
 * Usage: if (queue.isEmpty()) . . .
 * ---------------------------------
 * Returns true if the queue contains no elements.
 */
   bool isEmpty() const;

/*
 * Method: clear
 * Usage: queue.clear();
 * ---------------------
 * Removes all elements from this queue.
 */
   void clear();

/*
 * Method: enqueue
 * Usage: queue.enqueue(value);
 * ----------------------------
 * Adds value to the end of the queue.
 */
   void enqueue(ValueType value);

/*
 * Method: dequeue
 * Usage: ValueType first = queue.dequeue();
 * -----------------------------------------
 * Removes and returns the first item in the queue.  This method
 * signals an error if called on an empty queue.
 */
   ValueType dequeue();

/*
 * Method: peek
 * Usage: ValueType first = queue.peek();
 * --------------------------------------
 * Returns the first value in the queue without removing it.  This
 * method signals an error if called on an empty queue.
 */
   ValueType peek() const;

/*
 * Copy constructor and assignment operator
 * ----------------------------------------
 * These methods implement deep copying for queues.
 */
   Queue(const Queue<ValueType> & src);
   Queue<ValueType> & operator=(const Queue<ValueType> & src);

   The private section of the class goes here.

};

   The implementation of the class goes here.

#endif
```

635
}
636

图 14-6 （续）

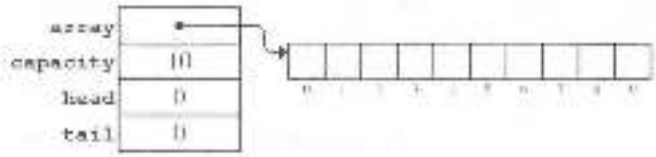
如果你使用这种表示策略，Queue 的构造函数如下图所示：

```
template <typename ValueType>
Queue<ValueType>::Queue() {
    head = tail = 0;
}
```

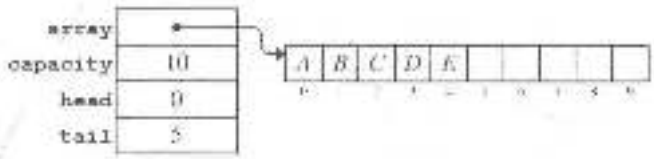
enqueue 和 dequeue 方法看起来几乎和 Stack 类中对应的 push 和 pop 方法完全一样，尽管这种想法十分诱人，但如果你尝试复制现存的代码，将会遇到几个问题。在你实现队列之前，用画图的方式保证你能够完全理解队列的操作往往更有帮助。

为了理解队列的这种表示法是如何工作的，想象一下队列代表一条等待队列，它和第 5 章模拟的图形相似。一个新的顾客时不时到达并被添加到队列中。队列中的顾客周期性地在队列的头部接受服务，之后他们将完全离开队列。队列的数据结构是如何响应这些操作的？

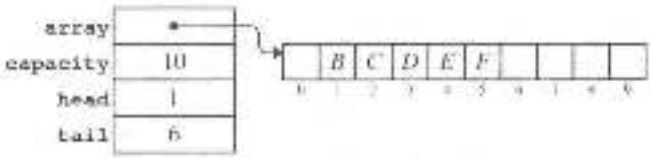
假设刚开始时队列为空，它的内部结构看起来如下图所示：



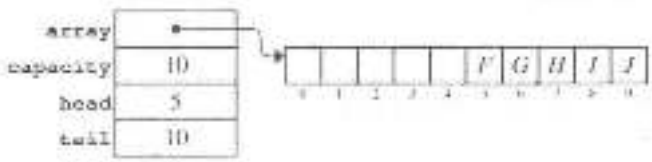
现在假设五个顾客到达了，用字母 A 到 E 表示。这些顾客按顺序依次入队，这产生了以下的结构图形：



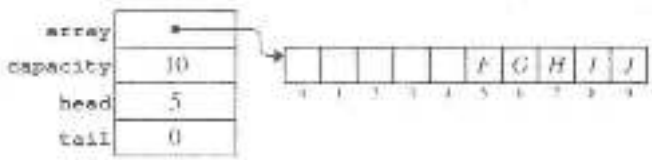
head 域的值为 0，表示队列中的第一个顾客被存储在数组的位置 0 处；tail 域的值为 5，表示下一个顾客将被放置在数组位置 5 处。到目前为止，一切顺利。此时，假设你轮流服务队列开始位置的顾客，然后将一个新的顾客添加到队尾。例如，顾客 A 出列，顾客 F 到达，这得到了下面的情形：



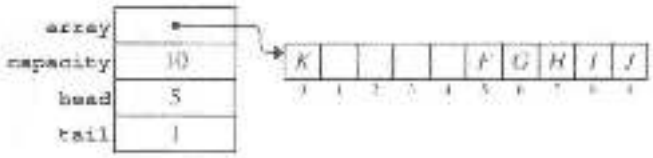
想象一下，在下一个顾客到来之前，你每次服务一位顾客，这种趋势一直持续到顾客 J 到达。然后，队列的内部结构看起来如下图所示：



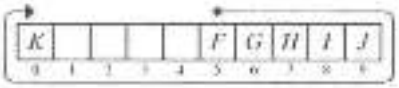
此时，你遇到一点问题。队列中只有五个顾客，但你已经用完了所有可用的空间。tail 域正指向超出数组末尾的位置。另一方面，在数组的头部尚有未使用的空间。因此，不增加 tail，因为它表示不存在的位置 10，取而代之的是，你可以从数组的尾部绕回到位置 0 处，如下图所示：



从这个位置开始，你有了可使用的空间，顾客K入列，并存储在数组位置0处，这得到了下面的图结构：



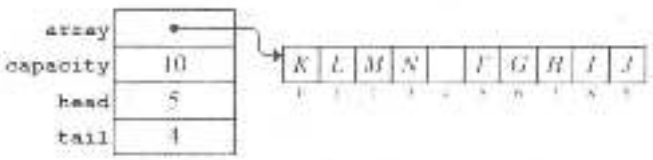
如果你允许队列中的元素从数组的尾部绕回到头部，活跃的元素总是从head索引处一直延续到tail索引前面的位置，如下图所示：



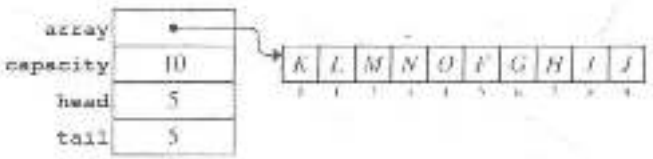
638

因为数组的尾部和头部表现得好像被连接在一起，因此，程序员称这种表示法为环形缓冲区（ring buffer）。

在你编写enqueue和dequeue的代码之前，剩下唯一需要考虑的问题是如何检测一个队列是否满了。队列为满的情况检测可能比你预期的要更加复杂。为了理解哪些地方会产生复杂问题，假设在其他额外的顾客被服务之前，有二个以上的顾客到达。如果你将顾客L、M和N入队，数据结构看起来如下图所示：



此时，上图中好像只有一个额外的空间。如果此时顾客O到达，将会发生什么？如果你遵循前面入队操作的逻辑，最终的结果将具有如下结构：



队列数组现在是完全满的。遗憾的是，无论何时，head和tail域都有相同的值，正如上图所示，因此队列被认为是空的。没有办法能够区分队列结构本身是空还是满，因为在两种情形下，head和tail域值看起来一样。尽管你可以通过为空队列采用一个不同的定义并写一些特殊情况的代码来解决这个问题，最简单的方法是限制队列元素的个数比队列的容量小1，并且在达到限制条件时，扩充数组。

Queue类模板的环形缓冲区的实现代码显示在图14-7和图14-8中。一个重要的观察结果是：代码并不能明确检测数组索引以观察元素是否从数组尾部绕回到数组头部。取而代之的是，代码能够利用%操作符自动计算正确的索引位置。通过求一个结果的余数，使其结果值落在一个整数循环范围内的技术是一个重要的数学方法，被称为**模运算**（modular

639 arithmetic）。

```
/* Private section */

/*
 * Implementation notes
 * --------------------
 * The array based queue stores the elements in successive index
 * positions in an array, just as a stack does.  What makes the
 * queue structure more complex is the need to avoid shifting
 * elements as the queue expands and contracts.  In the array
 * model, this goal is achieved by keeping track of both the
 * head and tail indices.  The tail index increases by one each
 * time an element is enqueued, and the head index increases by
 * one each time an element is dequeued.  Each index therefore
 * marches toward the end of the allocated array and will
 * eventually reach the end.  Rather than allocate new memory,
 * this implementation lets each index wrap around back to the
 * beginning as if the ends of the array of elements were joined
 * to form a circle.  This representation is called a ring buffer.
 *
 * The elements of the queue are stored in a dynamic array of
 * the specified element type.  If the space in the array is ever
 * exhausted, the implementation doubles the array capacity.
 * Note that the queue capacity is reached when there is still
 * one unused element in the array.  If the queue is allowed to
 * fill completely, the head and tail indices have the same
 * value, and the queue appears empty.
 */

private:

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   ValueType *array;                    /* A dynamic array of the elements */
   int capacity;                        /* The allocated size of the array */
   int head;                            /* The index of the head element   */
   int tail;                            /* The index of the tail element   */

/* Private method prototypes */

   void deepCopy(const Queue<ValueType> & src);
   void expandCapacity();
```

图 14-7 基于数组的队列的私有部分

640

```
/*
 * Implementation section
 * ----------------------
 * Clients should not need to look at any of the code beyond this point.
 */

/*
 * Implementation notes: Queue constructor
 * ---------------------------------------
 * The constructor allocates the array storage and initializes the fields.
 */

template <typename ValueType>
Queue<ValueType>::Queue() {
   capacity = INITIAL_CAPACITY;
   array = new ValueType[capacity];
   head = 0;
   tail = 0;
}

/*
 * Implementation notes: ~Queue
 * ----------------------------
 * The destructor frees any memory that is allocated by the implementation.
 */
```

图 14-8 基于数组的队列的实现

```
template <typename ValueType>
Queue<ValueType>::~Queue() {
   delete[] array;
}

/*
 * Implementation notes: size
 * --------------------------
 * The size is calculated from head and tail using modular arithmetic.
 */

template <typename ValueType>
int Queue<ValueType>::size() const {
   return (tail + capacity - head) % capacity;
}

/*
 * Implementation notes: isEmpty
 * -----------------------------
 * The queue is empty whenever the head and tail pointers are equal.  This
 * interpretation means that the queue must always leave one unused space.
 */

template <typename ValueType>
bool Queue<ValueType>::isEmpty() const {
   return head == tail;
}

/*
 * Implementation notes: clear
 * ---------------------------
 * The clear method need not take account of where in the ring buffer the
 * existing values are stored and can simply reset the head and tail indices.
 */

template <typename ValueType>
void Queue<ValueType>::clear() {
   head = tail = 0;
}

/*
 * Implementation notes: enqueue
 * -----------------------------
 * This method first checks to see whether there is enough room for the
 * element and then expands the array storage if necessary.  Because it
 * is otherwise impossible to differentiate the case when a queue is
 * empty from when it is completely full, this implementation expands
 * the queue when the size is one less than the capacity.
 */

template <typename ValueType>
void Queue<ValueType>::enqueue(ValueType value) {
   if (size() == capacity - 1) expandCapacity();
   array[tail] = value;
   tail = (tail + 1) % capacity;
}

/*
 * Implementation notes: dequeue, peek
 * -----------------------------------
 * These methods check for an empty queue and report an error if
 * there is no first element.
 */

template <typename ValueType>
ValueType Queue<ValueType>::dequeue() {
   if (isEmpty()) error("dequeue: Attempting to dequeue an empty queue");
   ValueType result = array[head];
   head = (head + 1) % capacity;
   return result;
}

template <typename ValueType>
ValueType Queue<ValueType>::peek() const {
   if (isEmpty()) error("peek: Attempting to peek at an empty queue");
   return array[head];
}
```

图 14-8 (续)

```
/*
 * Implementation notes: Deep copying support
 * ------------------------------------------
 * These methods implement deep copying for queues.
 */

template <typename ValueType>
Queue<ValueType>::Queue(const Queue<ValueType> & src) {
   deepCopy(src);
}

template <typename ValueType>
Queue<ValueType> & Queue<ValueType>::operator=(const Queue<ValueType> & src) {
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}

template <typename ValueType>
void Queue<ValueType>::deepCopy(const Queue<ValueType> & src) {
   int count = src.size();
   capacity = count + INITIAL_CAPACITY;
   array = new ValueType[capacity];
   for (int i = 0; i < count; i++) {
      array[i] = src.array[(src.head + i) % src.capacity];
   }
   head = 0;
   tail = count;
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This private method doubles the capacity of the dynamic array whenever
 * it runs out of space.  For simplicity, this implementation also shifts
 * all the elements back to the beginning of the array.
 */

template <typename ValueType>
void Queue<ValueType>::expandCapacity() {
   int count = size();
   ValueType *oldArray = array;
   array = new ValueType[2 * capacity];
   for (int i = 0; i < count; i++) {
      array[i] = oldArray[(head + i) % capacity];
   }
   capacity *= 2;
   head = 0;
   tail = count;
   delete[] oldArray;
}
```

图 14-8 （续）

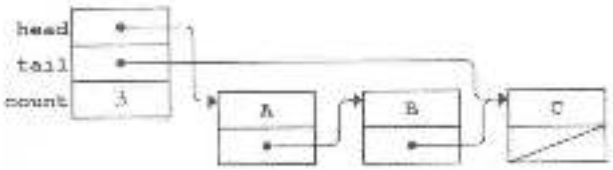
641 | 643

14.3.2 队列的链表表示

队列 queue 类也有一个简单地使用链表结构的表示方法。如果你采用这种方法，队列中的元素将被存储在一个链表中，链表以队头开始并在队尾结束。为了允许 enqueue 和 dequeue 操作能在一个常量时间里运行，Queue 对象必须拥有一个指向队尾的指针。因此，私有的实例变量被定义成如图 14-9 私有部分的修订版所示一样。在注释中，相比周围的文字，ASCII 数据图可能给实现者传递了更多的信息。有时候产生这样的图是单调乏味的，但是它们给读者提供了大量的信息。

给出一个现代的文字处理器和一个绘图程序，相比于你单独使用 ASCII 字符，可能会产生更多的细节图形。如果你正在为一个大而复杂的系统设计数据结构，创建这些图并将它们作为一个程序包的扩展文件的一部分，理想地是包含在一个网页中。例如，这里是包含原

容A、B和C的队列的一个可读性更高的图：

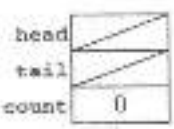



```
/* Private section */

/*
 * Implementation notes: Queue data structure
 * ------------------------------------------
 * The list-based queue uses a linked list to store the elements
 * of the queue.  To ensure that adding a new element to the tail
 * of the queue is fast, the data structure maintains a pointer to
 * the last cell in the queue as well as the first.  If the queue is
 * empty, both the head pointer and the tail pointer are set to NULL.
 *
 * The following diagram illustrates the structure of a queue
 * containing two elements, A and B.
 *
 *       +--------+          +--------+          +--------+
 *  head |   o----+--------->|   A    |  +---+-->|   B    |
 *       +--------+          +--------+  |   |   +--------+
 *  tail |   o----+---+      |   o----+--+   |   |  NULL  |
 *       +--------+   |      +--------+      |   +--------+
 *                    |                      |
 *                    +----------------------+
 */

private:

/* Type for linked list cell */

   struct Cell {
      ValueType data;                  /* The data value                */
      Cell *link;                      /* Link to the next cell         */
   };

/* Instance variables */

   Cell *head;                         /* Pointer to the cell at the head */
   Cell *tail;                         /* Pointer to the cell at the tail */
   int count;                          /* Number of elements in the queue */

/* Private method prototypes */

   void deepCopy(const Queue<ValueType> & src);
```

图 14.9 基于链表的队列的私有部分

关于基于链表的队列实现的代码显示在图 14-10 中。总的来说，这个代码是相当简单的，尤其是如果你使用栈的链表实现方法作为模板。内部的结构图表供了你所需的重要视角，它可以帮助你理解如何实现这些队列操作。例如，enqueue 操作在 tail 指针标记的地方添加一个新的空间，然后更新 tail 指针使它继续表示链表的结尾。dequeue 操作包括 head 指针标记的空间，并返回该空间处的值。

实现方法中唯一复杂的地方是空队列的表示。表示一个空队列的最简单的方法是在 head 指针处存储 NULL，如下图所示：



644 ↓ 645

```
/*
 * Implementation section
 * ----------------------
 * It is required that the implementation for a template class be available
 * to the compiler whenever that type is used.  The effect of this
 * restriction is that header files must include the implementation.
 * Clients should not need to look at any of the code beyond this point.
 */

/*
 * Implementation notes: Queue constructor
 * ---------------------------------------
 * The constructor creates an empty linked list and sets the count to 0.
 */

template <typename ValueType>
Queue<ValueType>::Queue() {
   head = tail = NULL;
   count = 0;
}

/*
 * Implementation notes: ~Queue destructor
 * ---------------------------------------
 * The destructor frees any heap memory allocated by the queue.
 */

template <typename ValueType>
Queue<ValueType>::~Queue() {
   clear();
}

/*
 * Implementation notes: size, isEmpty, clear
 * ------------------------------------------
 * These methods use the count variable and therefore run in constant time.
 */

template <typename ValueType>
int Queue<ValueType>::size() const {
   return count;
}

template <typename ValueType>
bool Queue<ValueType>::isEmpty() const {
   return count == 0;
}

template <typename ValueType>
void Queue<ValueType>::clear() {
   while (count > 0) {
      dequeue();
   }
}

/*
 * Implementation notes: enqueue
 * -----------------------------
 * This method allocates a new list cell and chains it in at the tail of
 * the queue.  If the queue is currently empty, the new cell also becomes
 * the head pointer in the queue.
 */

template <typename ValueType>
void Queue<ValueType>::enqueue(ValueType value) {
   Cell *cp = new Cell;
   cp->data = value;
   cp->link = NULL;
   if (head == NULL) {
      head = cp;
   } else {
      tail->link = cp;
   }
   tail = cp;
   count++;
}
```

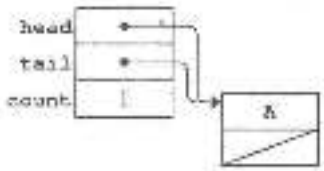
图 14-10　基于链表的队列的实现

```
/*
 * Implementation notes: dequeue, peek
 * -----------------------------------
 * These methods check for an empty queue and report an error if
 * there is no first element.  The dequeue method also checks for
 * the case in which the queue becomes empty and sets both the head
 * and tail pointers to NULL.
 */

template <typename ValueType>
ValueType Queue<ValueType>::dequeue() {
   if (isEmpty()) error("dequeue: Attempting to dequeue an empty queue");
   Cell *cp = head;
   ValueType result = cp->data;
   head = cp->link;
   if (head == NULL) tail = NULL;
   delete cp;
   count--;
   return result;
}

template <typename ValueType>
ValueType Queue<ValueType>::peek() const {
   if (isEmpty()) error("peek: Attempting to peek at an empty queue");
   return head->data;
}

/*
 * Implementation notes: copy constructor and assignment operator
 * ---------------------------------------------------------------
 * These methods follow the standard template, leaving the work to deepCopy.
 */

template <typename ValueType>
Queue<ValueType>::Queue(const Queue<ValueType> & src) {
   deepCopy(src);
}

template <typename ValueType>
Queue<ValueType> & Queue<ValueType>::operator=(const Queue<ValueType> & src) {
   if (this != &src) {
      clear();
      deepCopy(src);
   }
   return *this;
}

/*
 * Implementation notes: deepCopy
 * ------------------------------
 * This function copies the data from the src parameter into the current
 * object.  This implementation simply walks down the linked list in the
 * source object and enqueues each value in the destination.
 */

template <typename ValueType>
void Queue<ValueType>::deepCopy(const Queue<ValueType> & src) {
   head = NULL;
   tail = NULL;
   count = 0;
   for (Cell *cp = src.head; cp != NULL; cp = cp->link) {
      enqueue(cp->data);
   }
}
```

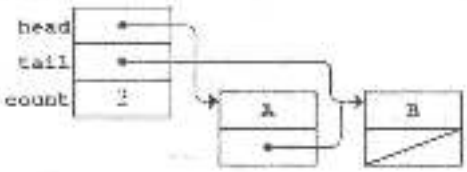
图 14-10 （续）

enqueue 的实现代码必须将空队列作为一个特殊情况进行检测。如果 head 指针为 NULL，enqueue 必须设置 head 和 tail 指针的值，使得它们指向包含新元素的空间。因此，如果你想要将顾客 A 放入一个空队列中，在 enqueue 操作的最后，指针的内部结构将看起来如下图所示：

646 ↓ 648



如果再次调用 enqueue，head 指针将不再为 NULL。这意味着实现的代码不再需要执行关于空队列的特殊情况的动作。取而代之的是，enqueue 实现方法使用 tail 指针找到链表的结尾，并在该处添加新的空间。例如，如果你在顾客 A 之后将顾客 B 入队，该队列的最终结构看起来如下图所示：



14.4 实现矢量类

第 5 章介绍的 vector 类是线性结构的另一个例子。在很多方面，Vector 类的实现方法是第 13 章的编辑器缓冲区和本章中你所见过的栈和队列抽象的结合。Vector 类类似于 EditorBuffer 类，它允许用户插入或删除任何位置上的元素。同时，Vector 也类似于 Stack 和 Queue 类，因为它实现了深拷贝，并且使用模板来支持多态。

由于你见过和 Vector 类似的类，所以代码需要解释的方面更少。在前面模型中的 Vector 类唯一没有出现的方法是用于选择的方括号。正如你从前面几章所了解到的，C++ 可以扩展基本运算，使它们可以应用到新的数据类型中。使用大致相同的方式，通过重载 operator[] 的定义，C++ 允许类重新定义选择的方法。重载 operator[] 方法的函数原型如下所示：

```
ValueType & operator[](int index);
```

与插入操作符一样，选择操作符必须通过引用返回值，以便它可以将一个新值赋给一个元素位置。

多态的 vector.h 接口的完整文本描述包括其私有部分和实现部分，如图 14-11 所示。 649

```
/*
 * File: vector.h
 * --------------
 * This interface exports the Vector template class, which provides an
 * efficient, safe, convenient replacement for the array type in C++.
 */

#ifndef _vector_h
#define _vector_h

#include "error.h"

/*
 * Class: Vector<ValueType>
 * ------------------------
 * This class stores an ordered list of values similar to an array.  It
 * supports traditional array selection using square brackets, but also
 * supports the insertion and deletion of elements.
 */
```

图 14-11 vector.h 接口

```
template <typename ValueType>
class Vector {

public:

/*
 * Constructor: Vector
 * Usage: Vector<ValueType> vec;
 *        Vector<ValueType> vec(n, value);
 * ----------------------------------------
 * Initializes a new Vector object.  The first form creates an empty vector;
 * the second creates a vector of size n in which each element is initialized
 * to the specified value or the default value for the element type.
 */

   Vector();
   Vector(int n, ValueType value = ValueType());

/*
 * Destructor: ~Vector
 * Usage: (usually implicit)
 * -------------------------
 * Frees any heap storage allocated by this vector.
 */

   ~Vector();

/*
 * Method: size
 * Usage: int n = vec.size();
 * --------------------------
 * Returns the number of values in this vector.
 */

   int size() const;

/*
 * Method: isEmpty
 * Usage: if (vec.isEmpty()) . . .
 * -------------------------------
 * Returns true if this vector contains no elements.
 */

   bool isEmpty() const;

/*
 * Method: clear
 * Usage: vec.clear();
 * -------------------
 * Removes all elements from this vector.
 */

   void clear();

/*
 * Method: get
 * Usage: ValueType value = vec.get(index);
 * ----------------------------------------
 * Returns the element at the specified index in this vector.  This method
 * signals an error if the index is not in the array range.
 */

   ValueType get(int index) const;

/*
 * Method: set
 * Usage: vec.set(index, value);
 * -----------------------------
 * Replaces the element at the specified index in this vector with a new
 * value.  The previous value at that index is overwritten.  This method
 * signals an error if the index is not in the array range.
 */

   void set(int index, ValueType value);

/*
 * Method: insert
```

图14-1 (续)

```
 * Usage: vec.insert(0, value);
 * ----------------------------
 * Inserts the element into this vector before the specified index.  All
 * subsequent elements are shifted one position to the right.  This method
 * signals an error if the index is outside the range from 0 up to and
 * including the length of the vector.
 */

   void insert(int index, ValueType value);

/*
 * Method: remove
 * Usage: vec.remove(index);
 * -------------------------
 * Removes the element at the specified index from this vector.  All
 * subsequent elements are shifted one position to the left.  This method
 * signals an error if the index is outside the array range.
 */

   void remove(int index);

/*
 * Method: add
 * Usage: vec.add(value);
 * ----------------------
 * Adds a new value to the end of this vector.
 */

   void add(ValueType value);

/*
 * Operator: []
 * Usage: vec[index]
 * -----------------
 * Overloads [] to select elements from this vector.  This extension
 * enables the use of traditional array notation to get or set individual
 * elements.  This method signals an error if the index is outside the
 * array range.
 */

   ValueType & operator[](int index);

/*
 * Copy constructor and assignment operator
 * ----------------------------------------
 * These methods implement deep copying for vectors.
 */

   Vector(const Vector<ValueType> & src);
   Vector<ValueType> & operator=(const Vector<ValueType> & src);

/* Private section */

/*
 * Notes on the representation
 * ---------------------------
 * This version of the vector.h interface stores the elements in a
 * dynamic array of the specified element type.  If the space in the
 * array is ever exhausted, the implementation doubles the array capacity.
 */

private:

   static const int INITIAL_CAPACITY = 10;

/* Instance variables */

   ValueType *array;                    /* A dynamic array of the elements */
   int capacity;                        /* The allocated size of the array */
   int count;                           /* The number of elements in use   */

/* Private method prototypes */

   void deepCopy(const Vector<ValueType> & src);
   void expandCapacity();
};
```

图 14-11 （续）

```
/*
 * Implementation notes: Vector constructor and destructor
 * -------------------------------------------------------
 * The two implementations of the constructor each allocate storage for
 * the dynamic array and then initialize the other fields of the object.
 * The destructor frees the heap memory used by the dynamic array.
 */

template <typename ValueType>
Vector<ValueType>::Vector() {
   capacity = INITIAL_CAPACITY;
   count = 0;
   array = new ValueType[capacity];
}

template <typename ValueType>
Vector<ValueType>::Vector(int n, ValueType value) {
   capacity = (n > INITIAL_CAPACITY) ? n : INITIAL_CAPACITY;
   array = new ValueType[capacity];
   count = n;
   for (int i = 0; i < n; i++) {
      array[i] = value;
   }
}

template <typename ValueType>
Vector<ValueType>::~Vector() {
   delete[] array;
}

/*
 * Implementation notes: size, isEmpty, clear
 * ------------------------------------------
 * These methods require only the count field and do not look at the data.
 */

template <typename ValueType>
int Vector<ValueType>::size() const {
   return count;
}

template <typename ValueType>
bool Vector<ValueType>::isEmpty() const {
   return count == 0;
}

template <typename ValueType>
void Vector<ValueType>::clear() {
   count = 0;
}

/*
 * Implementation notes: get, set
 * ------------------------------
 * These methods first check that the index is in range and then get or set
 * the appropriate index position in the dynamic array.
 */

template <typename ValueType>
ValueType Vector<ValueType>::get(int index) const {
   if (index < 0 || index >= count) error("get: index out of range");
   return array[index];
}

template <typename ValueType>
void Vector<ValueType>::set(int index, ValueType value) {
   if (index < 0 || index >= count) error("set: index out of range");
   array[index] = value;
}

/*
 * Implementation notes: Vector selection
 * --------------------------------------
 * The following code implements traditional array selection using square
 * brackets for the index.  To ensure that clients can assign to array
 * elements, this method uses an & to return the result by reference.
 */
```

图 14-11 (续)

```
template <typename ValueType>
ValueType & Vector<ValueType>::operator[](int index) {
   if (index < 0 || index >= count) error("Vector index out of range");
   return array[index];
}

/*
 * Implementation notes: add, insert, remove
 * -----------------------------------------
 * These methods shift the existing elements in the array to make room
 * for a new element or to close up the space left by a deleted one.
 */

template <typename ValueType>
void Vector<ValueType>::add(ValueType value) {
   insert(count, value);
}

template <typename ValueType>
void Vector<ValueType>::insert(int index, ValueType value) {
   if (count == capacity) expandCapacity();
   if (index < 0 || index > count) error("insert: Index out of range");
   for (int i = count; i > index; i--) {
      array[i] = array[i - 1];
   }
   array[index] = value;
   count++;
}

template <typename ValueType>
void Vector<ValueType>::remove(int index) {
   if (index < 0 || index >= count) error("remove: Index out of range");
   for (int i = index; i < count - 1; i++) {
      array[i] = array[i + 1];
   }
   count--;
}

/*
 * Implementation notes: copy constructor and assignment operator
 * --------------------------------------------------------------
 * These methods follow the standard template, leaving the work to deepCopy
 */

template <typename ValueType>
Vector<ValueType>::Vector(const Vector<ValueType> & src) {
   deepCopy(src);
}

template <typename ValueType>
Vector<ValueType> & Vector<ValueType>::operator=(const Vector<ValueType> & src)
   if (this != &src) {
      delete[] array;
      deepCopy(src);
   }
   return *this;
}

/*
 * Implementation notes: deepCopy
 * ------------------------------
 * This function copies the data from the src parameter into the current
 * object.  All dynamic memory is reallocated to create a 'deep copy' in
 * which the current object and the source object are independent.
 * The capacity is set so that the vector has some room to expand.
 */

template <typename ValueType>
void Vector<ValueType>::deepCopy(const Vector<ValueType> & src) {
   capacity = src.count + INITIAL_CAPACITY;
   this->array = new ValueType[capacity];
   for (int i = 0; i < src.count; i++) {
      array[i] = src.array[i];
   }
   count = src.count;
```

图 14-11 (续)

```
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------------
 * This method doubles the array capacity whenever it runs out of space.
 */

template <typename ValueType>
void Vector<ValueType>::expandCapacity() {
   ValueType *oldArray = array;
   capacity *= 2;
   array = new ValueType[capacity];
   for (int i = 0; i < count; i++) {
      array[i] = oldArray[i];
   }
   delete[] oldArray;
}

#endif
```

图 14-11 （续）

14.5 集成原型和代码

650 / 656

使用接口的一个主要原因是向用户隐藏实现的复杂性。对于第 2 章你所见到的排序的简单接口，接口和实现的界限是通过将它们放在单独的文件中实现的。用户所需要看的是定义接口的 .h 文件，实现的繁琐细节被归入到相应的 .cpp 文件中。

遗憾的是，当接口变得更加复杂，C++ 就很难维持那样的分离级别。一个类的私有部分必须被包含在类定义中，这个要求意味着这些细节地方必须是 .h 文件的一部分。对于模板类而言，相关情况更加糟糕，因为 C++ 要求无论何时模板类想要扩展一个模板，它完整的实现必须可用。这种约束的作用是使模板类的接口和定义一样，必须包含所有的代码。

假设一个模板类的实现无论如何都将是 .h 文件的一部分，一些专业的 C++ 程序员一起放弃了物理分离的概念，并直接在类中包含方法的主体。采用这种策略能够显著简化类的语法结构。template 关键字只在类的开头出现一次，并且不需要在每一个方法的实现代码中重复它。每一个方法的实现都在类内，这个事实意味着你可以省略 ;: 这个标签。

然而，尽管就语法的简化而言，它存在优势，但本书中的例子仍继续把方法原型和实现代码放在 .h 文件的不同部分。类的公有部分只包含原型。相应的实现代码出现在文件的末尾，在一个注释后提醒用户除了 .h 文件中的其他东西都是为了实现。至少维持一些分离意味着：对于用户而言，他们仍可以看到整理好的实现细节的部分接口文件。更重要的是，同时在 .h 文件的不同部分包含原型和实现意味着和完整定义联系的每一部分的注释都能有相应的受众。在 .h 文件开头书写的原型是为了用户，并且它不包含任何的实现细节。相比之下，和实际代码相关的注释是为实现者准备的。将这两个部分放在一起迫使注释面向两类受众，这使得它们对于用户而言作用更小。

本章小结

在本章，你已经学会了如何将 C++ 的模板机制用于一般的容器类中。就一个专门用于特殊用户数据类型的占位符而言，模板允许你定义类。你也有机会观察 Stack 和 Queue 类，以及多态的 Vector 类的公共接口，这些基于数组和基于链表的实现方法。

本章的重点包括：

- 模板用于定义一般的容器类。 657
- 除了较传统的基于数组的表示方法，我们还可以使用链表结构实现栈。
- 基于数组的队列实现在某些地方比基于栈的实现更为复杂。传统的实现方法使用一个被称为环形缓存区的结构，其中，元素逻辑性地从数组结尾绕回到开头。模运算容易实现环形缓存区的概念。
- 在本章使用的环形缓存区的实现方法中，当一个队列的头和尾索引值相同时，它被认为是空的。这种表示策略意味着队列的最大容量比数组分配的空间少一个。尝试将所有的元素都填入数组使得一个满队列和一个空队列几乎难以区分。
- 也可以使用两个指针标记的链表来表示队列，一个指向队列头部，另一个指向队列尾部。
- 使用动态数组很容易表示 Vector 类。插入和删除元素要求在数组中移动元素，这意味着这些操作通常需要 $O(N)$ 时间。
- 你可以通过定义方法来重新定义类的操作。方法名包含关键字 operator，后面接操作符符号。尤其是你可以通过定义 operator[] 方法来重新定义选择操作。

复习题

1. C++ 模板能给适用容器的设计者带来什么优势？
2. 作为用户，当你实例化一个类模板时，如何确定什么类型将被用于填入模板的占位符位置中？
3. 使用链表的实现方法，在下面的操作执行完成后，画一个表示 myStack 类对在执行完下述操作后的栈元素图：

```
Stack<char> myStack;
myStack.push('A');
myStack.push('B');
myStack.push('C');
```

4. 如果你使用一个数组来存储一个队列潜在的元素，Queue 类需要哪些私有的实例变量？
5. 什么是一个环形缓存区？环形缓存区的概念是如何应用到队列中的？ 658
6. 你怎样识别一个基于数组的队列是否为空？你怎样识别元素个数是否到达它的最大容量？
7. 假设 INITIAL_CAPACITY 有人为的小的数值 3，在下面的操作序列执行完后，画图表示基于数组的队列 myQueue：

```
Queue<char> myQueue;
myQueue.enqueue('A');
myQueue.enqueue('B');
myQueue.enqueue('C');
myQueue.dequeue();
myQueue.dequeue();
myQueue.enqueue('D');
myQueue.enqueue('E');
myQueue.dequeue();
myQueue.enqueue('F');
```

8. 解释一下模运算在基于数组的队列实现中是如何起作用的。
9. 描述一下，关于基于数组的队列实现，下面的 size 实现代码有哪些错误？

```
template <typename ValueType>
int Queue<ValueType>::size() const {
    return (tail - head) % capacity;
}
```



10. 在计算机完成了复习题7中的一组操作后，画图表示一个链表队列的内部结构。
11. 你如何识别一个链表队列是否为空？
12. 你需要哪些重载方法来重新定义一个类的选择 [] 运算？

习题

1. 标准的C++头文件 <algorithm>（你将会在第20章学到更多内容）包含一个模板函数 swap(x,y)。
659 函数交换了 x 和 y 的值。编写并测试你自己的 swap 函数的实现代码。
2. 编写一个 sort.h 接口，该接口提供了一个 sort 函数的多态版本。sort 函数适用于实现了标准关系运算的任何基类型。你的函数应该具有以下原型：

```
template <typename ValueType>
void sort(Vector<ValueType> & vec);
```

使用图10-9出现的快速排序算法实现 sort 函数。
3. 设计并实现一个模板类 Pair<T1,T2>，它表示一对值，第一个类型为 T1，第二个类型为 T2。Pair 类应该提供以下方法：
   - 一个默认的构造函数，它产生一对默认类型 T1 和 T2 的值。
   - 一个构造函数 Pair(v1,v2)，它有两个明确的类型值。
   - 获取 first 和 second 方法，它们返回所存储的这一对值的值拷贝。
4. 开发一个关于 stack.h 接口的相当全面的单元测试，使用几种不同基本类型的栈来测试 Stack 类提供的操作。使用你的单元测试程序来证明 Stack 类基于数组和基于链表这两种实现方法。
5. 为 queue.h 接口设计一个类似的单元测试。
6. 因为队列的环形缓冲区实现方法使得我们可以辨别一个空对列和一个满队列之间的不同，所以当动态数组中还有一个未使用的空间时，所实现的方法就必须为其增加容量。通过改变队列的内部表示，你可以避免这种限制，这样队列的固定结构就可跟踪队列的元素个数，而不是队尾元素的索引。给出队头元素的索引和队列中元素的个数，你往容易就能算出队尾元素索引，这意味着你不需要明确存储这个值。重写基于数组的队列的表示方法，使得它可以使用这种表示方法。
7. 在第5章的习题13中，你有机会编写以下函数：

```
void reverseQueue(Queue<string> & queue);
```

该函数完全从用户角度逆向排序了队列中的元素。然而，如果你是一名类的设计者，可以将这个功
660 能添加到 queue.h 接口，并作为其中一种方法提供给用户。对于队列基于数组和基于链表这两种实现方法，编写方法：

```
void reverse();
```

这个方法逆向排序队列中的元素。在基于数组与基于链表实现的两种情况下，编写这个函数使得它们使用原来的内存空间，而不需要分配额外的存储空间。
8. 在本章展示的队列抽象中，新的项总是被添加到队列的末尾，并依次等待处理。对于某些程序应用，将简单的队列抽象扩展为优先级队列是很有用的。其中，项的顺序是由一个数字化优先值决定的。一个项在优先级队列中入列后，它被插入到所有比它低优先级项的前面。如果队列中有两个项的优先级相同，它们将按照标准的先进先出顺序处理。

使用队列的链表实现方法作为一个模型，设计并实现一个 pqueue.h 接口，要求接口提供一个被称为 PriorityQueue 的类，和传统的 Queue 类一样。除了 enqueue 方法，PriorityQueue 的类应和 Queue 类一样提供相同的方法，enqueue 方法现在有一个额外的参数，如下所示：

```
void enqueue(ValueType value, double priority);
```

参数 value 和传统的 enqueue 版本中的参数相同，priority 参数是一个表示优先级的数值。

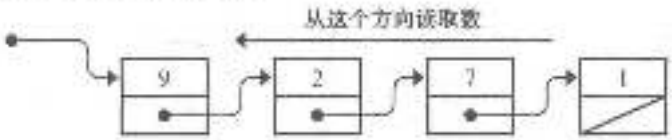
正如在传统的英语用法中一样，小的整数对应高优先级，因此优先级 1 在优先级 2 的前面，以此类推。

9. 为 Vector 类设计一个可用不同类型值进行测试的单元测试。
10. 编写 vector.h，它使用链表作为其基本表示。确保它通过你为前面的习题设计的单元测试。
11. 使用 12.4 节中 Vector 实现方法的技术来实现 Grid 类。除了门选择操作，它对于一个二维结构的编码也算手。
12. 在第 12 章的习题 9 中，要求你使用一个被称为 MyString 的类，该类要尽可能地复制 C++ 类库中 string 类的行为。对于本题而言，重新实现 MyString，它使用一个字符链表来代替一个动态数组作为它的基本表示。 661
13. 在新型机器上，使用 64 位来存储整数值类型 long 的值，这意味着类型 long 的最大正值是 9 223 372 036 854 775 807，也就是 $2^{63}-1$。尽管这个数看起来很大，但仍有一些应用需要更大的整数。例如，如果要求你计算有 52 张牌的一副牌所有可能的排列情况，你需要计算 52!，它的结果是：

80658175170943878571660636856403766975289505440883277824000000000000

如果你正在解决的问题涉及的整数值在这个范围内（例如，它经常出现在密码系统中），那么需要一个提供**扩展精度运算**（extended-precision arithmetic）的软件包，其中整数被表示在一个能够允许它们自动扩展的表格中。

尽管有更多有效的技术可以这样做，一种实现扩展精度运算的策略是在一个链表中存储单独的数字。在这种表示中，很很方便（大部分是因为这样做使得运算操作符更容易实现）排列链表，使得个位数首先出现，后面跟着十位数，然后是百位数，以此类推。因此，为了将数 1729 表示为一个链表，你要按照下面的顺序排列表：



设计并实现一个被称为 BigInt 的类，至少对于非负数，要求它使用这种表示方法实现扩展精度运算。你的 BigInt 类至少应该支持以下操作：

- 一个构造函数，该函数从一个 int 或一个数字字符串创建一个 BigInt 对象。
- 一个 toString 方法，该方法将一个 BigInt 对象转化成一个字符串。
- 操作符 + 和 *，分别表示加法和乘法。

如果你正自动手实现这些运算，通过模拟你所做的工作，可以实现这个算术运算。例如，加法要求你跟踪从一个数字位置到下一个数字位置的移动过程。乘法更困难一些，但是如果你找到正确的迭代分解，它仍然很容易实现。

使用你的 BigInt 类产生一张表，表展示了 $n!$ 的值，n 是 0 到 52 之间所有可能的值，包括 0 和 52。 662

第 15 章
Programming Abstractions in C++

映　射

当你愿意思考一些事情时，研究地图是一件不错的事。

663

——乔治·爱略特，《米德尔马奇》(*Middlemarch*)，1874

本书中你遇到的最有用的数据结构之一是映射，它实现了键与值的关联关系。第 5 章介绍了两个类（Map 和 HashMap），它们都实现了映射的概念。这两个类实现了同样的方法，并且经常能够相互替换使用。其主要区别是在遍历其中的元素时键的处理顺序。HashMap 更高效，但以一个看似随机的顺序遍历其键。Map 的效率稍低，但是它的优点是按照键的自然顺序遍历其元素。

下面两章的目标是看看这两个类是如何实现的。本章的重点在于 HashMap 类，它使得在常数时间内找到一个键所关联的值成为可能。之后的第 16 章将介绍树的概念。虽然树还有很多其他应用，但是它为 Map 提供了基础框架。这个框架在提供了对数时间操作的同时保留了按顺序处理键的能力。

正如你在阅读第 14 章中的较长代码时所发现的：使用模板定义泛型集合类会引起大量的复杂性。虽然模板在集合类的库版本中很重要，但是它所需要的额外复杂性很容易妨碍你理解用于实现映射思想的算法结构。由于这个原因，下面的几节实现了一个相对简单的类，称为 StringMap 类，在该类中键和值都是字符串类型。为了进一步简化，StringMap 类的公有部分仅导出那些对映射抽象具有重要性的 put 和 get 方法。StringMap 的公共接口如图 15-1 所示。

```
/*
 * File: stringmap.h
 * -----------------
 * This interface exports a simplified version of the Map class in which
 * the keys and values are always strings.
 */

#ifndef _stringmap_h
#define _stringmap_h

#include <string>

#include "vector.h"
class StringMap {

public:

/*
 * Constructor: StringMap
 * Usage: StringMap map;
 * ---------------------
 * Initializes a new empty map that uses strings as both keys and values.
 */

   StringMap();

/*
 * Destructor: ~StringMap
 * ----------------------
 * Frees any heap storage associated with this map.
 */
```

图 15-1　为映射抽象简化的接口

```
   ~StringMap();

/*
 * Method: get
 * Usage: string value = map.get(key);
 * -----------------------------------
 * Returns the value for key or the empty string if key is unbound.
 */

   std::string get(const std::string & key) const;

/*
 * Method: put
 * Usage: map.put(key, value);
 * ---------------------------
 * Associates key with value in this map.
 */

   void put(const std::string & key, const std::string & value);

   The private section of the class goes here.

#endif
```

图 15-1 （续）

15.1 使用矢量实现映射

在考虑更多有效的策略之前，首先实现基于矢量的简单 StringMap 类。这对于理解 StringMap 类是如何工作的将很有帮助。一个特别直接的方法就是跟踪矢量中的键－值对，其中每个元素都是一个结构体成员，定义如下：

```
struct KeyValuePair {
   string key;
   string value;
};
```

考虑到这个类型是 StringMap 类的一部分，key 和 value 域的类型都是 string 类型。基于模板的实现都会使用类似的结构，其中两个 string 类的实例将被模板参数 KeyType 和 ValueType 所代替，稍后你将会在本章中看到。

664 : 665

基于矢量版本的 StringMap 类的私有部分如图 15-2 所示。对键的绑定保存在一个称为 bindings 的类 KeyValuePair 矢量中，bindings 是作为该类的一个实例变量被存储的。

基于矢量版本的 StringMap 类的实现如图 15-3 所示。这个实现大部分都很简单。构造函数和析构函数都没有代码，原因在于 Vector 类执行它自己的存储管理。get 和 put 方法必须搜索已存在的键，因此把搜索矢量这一过程交给一个叫做 findKey 的私有方法，这对于 get 和 put 这两种方法是很有意义的。findKey 看起来如下所示：

```
int findKey(string key) {
   for (int i = 0; i < bindings.size(); i++) {
      if (bindings.get(i).key == key) return i;
   }
   return -1;
}
```

这个方法返回特定的已包含在 bindings 矢量中的键值在键列表中所处的索引值。如果这个键值不存在，findKey 返回 -1。使用线性查找算法意味着 get 和 put 方法所需的时间都是 $O(N)$。

```
/*
 * Notes on the representation
 * ---------------------------
 * This version of the StringMap class stores key-value pairs in a Vector.
 */

private:

/*
 * Type: KeyValuePair
 * ------------------
 * This type combines a key and a value into a single structure.
 */

   struct KeyValuePair {
      std::string key;
      std::string value;
   };

/* Instance variables */

   Vector<KeyValuePair> bindings;

/* Private function prototypes */

   int findKey(const std::string & key) const;
```

666

图 15-2 基于矢量实现的 StringMap 类的私有部分

```
/*
 * File: stringmap.cpp
 * -------------------
 * This file implements the stringmap.h interface.
 */

#include <string>
#include "stringmap.h"
using namespace std;

/*
 * Implementation notes: StringMap constructor and destructor
 * ----------------------------------------------------------
 * All dynamic allocation is handled by the Vector class.
 */

StringMap::StringMap() { }
StringMap::~StringMap() { }

/*
 * Implementation notes: put, get
 * ------------------------------
 * These methods use findKey to search for the specified key.
 */

string StringMap::get(const string & key) const {
   int index = findKey(key);
   return (index == -1) ? "" : bindings.get(index).value;
}

void StringMap::put(const string & key, const string & value) {
   int index = findKey(key);
   if (index == -1) {
      KeyValuePair entry;
      entry.key = key;
      index = bindings.size();
      bindings.add(entry);
   }
   bindings[index].value = value;
}

/*
 * Private method: findKey
 * -----------------------
 * Returns the index at which the key appears, or -1 if the key is not found.
 */
```

图 15-3 基于矢量实现的 StringMap 类的代码

```
int StringMap::findKey(const string & key) const {
   for (int i = 0; i < bindings.size(); i++) {
      if (bindings.get(i).key == key) return i;
   }
   return -1;
}
```

图 15-3 （续）

667

通过将键按顺序排序并应用 7.5 节所介绍的二分查找算法可以改善 get 方法的性能。二分查找可以将查找时间降低到 $O(\log N)$。这表明和比线性查找所需要的时间 $O(N)$ 而言，查找性能有了明显提高。遗憾的是，并没有一个显而易见的方法能将同样的优化应用于 put 方法。尽管在 $O(\log N)$ 时间内检查特定的键是否已存在于一个 map 中（甚至确定一个新的键需要被插入的确切位置）是可能的，但在某个位置插入一个新的键－值对，需要向前移动插入位置后的每一个元素。因此，即使在一个排好序的列表中，put 方法也需要 $O(N)$ 的时间。

15.2 查找表

编程时经常会遇到映射抽象问题，因此有必要投入巨大的努力提高它的性能。前面章节描述的实现策略（将排好序的键－值对存储在矢量中）对 get 操作给出了 $O(\log N)$ 的时间性能。put 操作的时间性能为 $O(N)$。我们可以把它们做得更好。

当你努力优化一个数据结构的性能时，首先选择一些特殊的情况使得性能得到改进，继而寻找一些方法来使算法更加通用。本节引入了一个特定的问题，我们可以很容易地找到一种对 get 和 put 操作的常量时间的实现方法。之后继续探究一种类似的技术如何在更普遍的情况下也会有所帮助。

1963 年，美国邮政服务提出用一组两个字母的代码表示美国各个州和地区的方案。50 个州的代码表示如图 15-4 所示。尽管你想要从相反的方向翻译，但是本节仅考虑把两个字母的代码翻译成州名的问题。因此你选择的数据结构必须能够表示从两个字母的缩写到州名的映射。

AK Alaska	HI Hawaii	ME Maine	NJ New Jersey	SD South Dakota
AL Alabama	IA Iowa	MI Michigan	NM New Mexico	TN Tennessee
AR Arkansas	ID Idaho	MN Minnesota	NV Nevada	TX Texas
AZ Arizona	IL Illinois	MO Missouri	NY New York	UT Utah
CA California	IN Indiana	MS Mississippi	OH Ohio	VA Virginia
CO Colorado	KS Kansas	MT Montana	OK Oklahoma	VT Vermont
CT Connecticut	KY Kentucky	NC North Carolina	OR Oregon	WA Washington
DE Delaware	LA Louisiana	ND North Dakota	PA Pennsylvania	WI Wisconsin
FL Florida	MA Massachusetts	NE Nebraska	RI Rhode Island	WV West Virginia
GA Georgia	MD Maryland	NH New Hampshire	SC South Carolina	WY Wyoming

图 15-4 用于美国邮政服务的 50 个州的缩写

668

当然，你可以将上述翻译表编码在一个 StringMap 中，或更一般的 Map<String, String> 中。然而，如果你严格地从用户的角度看这个问题，那么实现的细节就不是特别重要。本章的目的是确定一种使映射操作更有效的新实现方法。在这个例子中，要问一个重要的问题：把两个字母的字符串作为键是否比使用基于矢量的策略的实现方法具有更高的效率。

事实证明，两个字符的键的限制降低了查找操作的复杂性，并把时间降到了常量时间。你需要把州的名字存到一个二维的表格中，而在这个表格中，行和列的索引用州名缩写中的字母来计算。为了从表格中选出某个特定的元素，你可以简单地把州的缩写分解成它所包含的两个字符，每个字符减去 'A' 的 ASCII 码值，得到一个介于 0 到 25 之间的索引，然后使用这两个索引选择行和列。在图 15-5 中，给出了一个已经被初始化的包含州名的表格。你可以使用下面的函数把一个州的缩写转换成对应的州名：

```
string getStateName(string key, Grid<string> & grid) {
   char row = key[0] - 'A';
   char col = key[1] - 'A';
   if (!grid.inBounds(row, col) || grid[row][col] == "") {
      error("No state name for " + abbr);
   }
   return grid[row][col];
}
```

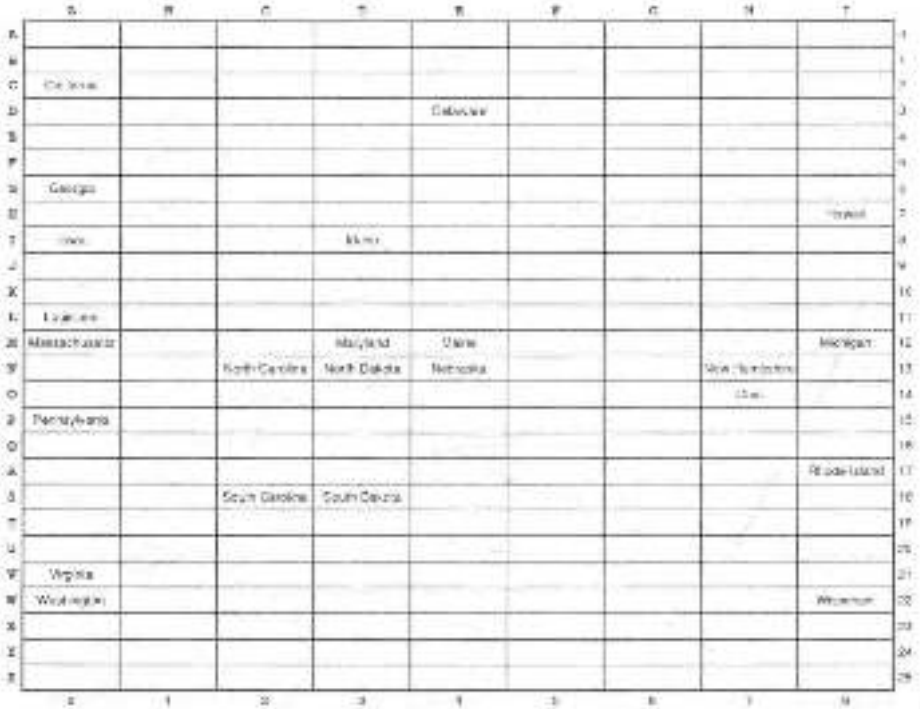


图 15-5 州查找表的前 9 列

该函数不包括任何看起来像查找数组元素的传统过程。取而代之的是：函数对字符的码值执行简单的算术运算，然后在表格里寻找答案。在其实现中，既没有循环也没有任何依赖于映射中的键的数量代码。因此，在这个表中，查找一个缩写花费的时间是 $O(1)$。

函数 getStateName 使用的表格是查找表（lookup table）的一个例子。查找表是一种程序结构，它可以通过计算表中适当的索引获得一个想要的值，最典型的就是矢量或者表格。查找表高效的原因在于其键可以立即告诉你去哪里找答案。然而，在当前的应用程序中，表的组织依赖于这样一个事实：键总是包含两个大写字母。如果键可以是任意的字符串（就像标准库版本中的 Map 类），那么至少以它当前的形式，查找表将不再适用。问题的关键在于是否能够推广这种策略以把它应用到更一般的情况。

如果你思考一下如何把这个问题运用到现实生活中，或许会发现：事实上，你在字典里查找单词的过程，就使用了与查找表类似的策略。如果你打算把基于矢量的映射应用于字典的查找问题，你会从第一个条目开始，接着是第二个、第三个，直到找到那个单词为止。当然没有人会把这个算法应用到一个真正任意大小的字典中。但是你不可能应用 $O(\log N)$ 二分查找算法（它包括准确地打开字典的中间部分）确定你要找的单词是在第一部分还是在第二部分，然后重复地应用这个算法到字典越来越小的部分。极有可能，你会利用很多字典侧面都有拇指标签这个事实，拇指标签表明每个字母出现的条目在哪里。你在 A 区域找以 A 开头的单词，在 B 区域找以 B 开头的单词，以此类推。这些拇指标签就代表了一个查找表，它使得你能找到正确的区域，从而降低需要查找的单词数量。

669 ↑ 670

至少对于像 StringMap 这种使用字符串作为键类型的映射，我们可以使用同样的策略。在这种类型的映射中，即使那个字符不一定是一个字母，但每一个键仍以某个字符值开始。如果你想为每个可能的首字符模拟使用拇指标签的策略，可以把映射分成 256 个单独由键－值对构成的列表（为每个首字符提供一个）。用户无论何时使用某个键调用 put 和 get 方法，代码都可以依据第一个字符选择出合适的列表。如果用于构成键的字符是均匀分布的，则该策略就能够将平均查找时间降低到原来的 1/256。

遗憾的是，一个映射中的键（类似于字典中的单词）并不是均匀分布的。例如，在字典中，以 C 开头的单词就要比以 X 开头的单词多得多。如果你在一个应用程序中使用映射，有可能 256 个字符中大多数都不会以首字符的身份出现。因此，一些键列表会是空的，而另外一些键列表会很长。因此，通过使用首字符策略得到的效率层面的提升依赖于键首字符可能出现的普遍程度。

另一方面，你没有理由仅使用键的首字符去优化映射的性能。首字符策略是对真实字典的一个粗劣模拟。你需要这样一个策略：其中键的值可以表明其对应的值的位置，就像查找表那样。该想法的最好实现就是使用称为哈希（hashing）的技术，该技术将在下节描述。

15.3 哈希

提高映射实现效率的最好方法就是提出一个使用键确定（或至少去接近）对应位置值的方法。选择键的任意一个明显的特征，例如它的首字符，甚至是它的前两个字符，都会遇到键特性分布不均匀的问题。

然而，既然你在使用计算机，那么就没有理由要求定位键的特征必须是人容易识别的东西。为了维护实现的效率，对计算机而言，唯一重要的是该特征是否容易确定。由于计算机比人更善于计算，因此，允许算法开辟更广阔的范围。 671

称为哈希操作的计算策略如下：

1. 选择一个函数 f，它把键转换成一个整数值，该整数值被称为键的哈希码（hash code）。计算哈希码的这个函数自然地被称为哈希函数（hash function）。使用该策略的映射抽象的实现习惯上被称为哈希表（hash table）。

2. 在表中查找匹配的键时，以键的哈希码作为起点。

15.3.1 设计数据结构

将 StringMap 类实现为哈希表的第一步是设计其数据结构。虽然其他的表示也是可行

的，但是一个通用的策略应该是使用哈希码计算一个列链表数组的索引，每个链表包含所有和该哈希码相匹配的键－值对。这些链表习惯上被称为桶（bucket）。为了找到你要查找的键，需要在桶中遍历键－值对链表。

在绝大多数哈希实现中，可能的哈希码的数量都大于桶的数量。但是，你可以把一个任意大的非负哈希码转换成一个桶数，通过计算哈希码除以桶数来得到其余数。因此，如果桶数被存储在变量 nBuckets 中，函数 hashCode 对给出的键将返回其哈希码，你可以按照如下方式计算桶数：

```
int bucket = hashCode(key) % nBuckets;
```

桶数代表了一个数组的索引，数组中的每个元素都是键－值对列表中第一个单元的地址。通俗地讲，如果一个关于键的哈希函数返回了经过取余操作的桶数，计算机科学家就会说一个键哈希到一个桶里（hashes to a bucket）。因此，同一个链表中所有键的共同特征是它们都能哈希到同一个桶里。两个或多个不同的键哈希到同一个桶里被称为**冲突**（collision）。

哈希有效的原因在于：对于任意特定的键，哈希函数总是返回相同的值。如果一个键被哈希到 17 号桶中，当你调用 put 函数把该键插入哈希表中时，它就被哈希到 17 号桶中，当你调用 get 函数查找键对应的值时，该键将仍然被哈希到 17 号桶中。

15.3.2 为字符串定义哈希函数

实现的下一步就是定义 hashCode 函数。在 StringMap 类这个例子中，hashCode
672 函数必须得到一个 string 类型的键，然后返回一个非负整数。为了达到哈希表的高效性，hashCode 函数必须具有以下两个特性：

1. 函数的计算代价不能太高。如果你设计的哈希函数太复杂，那就要花费很多时间去计算。哈希函数涉及的操作应相对容易实现。

2. 函数应该将键尽量均匀地分布在一个整数区间内。如果将冲突的数目降到最低，则哈希函数的效率最高。如果常用的键返回相同的哈希码，这些桶的链表会变得很长，而且需要很长的查找时间。

虽然哈希函数通常都很短，但是哈希函数特别敏感，而且经常依赖于复杂的数学。一般情况下，编写哈希函数的工作最好留给专家。Stanford 类库对 HashMap 类的实现采用了如图 15-6 所示的基于字符串的哈希函数，它是由芝加哥伊利诺伊大学数学系的教授丹尼尔·斯蒂文杨（Daniel J. Bernstein）设计的。

```
/*
 * Implementation notes: hashCode
 * ------------------------------
 * This function takes a string key and uses it to derive a hash code,
 * which is nonnegative integer related to the key by a deterministic
 * function that distributes keys well across the space of integers.
 * The specific algorithm used here is called djb2 after the initials
 * of its inventor, Daniel J. Bernstein, Professor of Mathematics at
 * the University of Illinois at Chicago.
 */

const int HASH_SEED = 5381;                 /* Starting point for first cycle */
const int HASH_MULTIPLIER = 33;             /* Multiplier for each cycle      */
const int HASH_MASK = unsigned(-1) >> 1;    /* The largest positive integer   */
```

图 15-6 字符串 hashcode 函数的实现

```
int hashCode(const string & str) {
   unsigned hash = HASH_SEED;
   int n = str.length();
   for (int i = 0; i < n; i++) {
      hash = HASH_MULTIPLIER * hash + str[i];
   }
   return int(hash & HASH_MASK);
}
```

图 15-6（续）

在任何特定的库中，用于字符串的哈希函数或许和这个完全不一样，但是大多数的实现都有相似的结构。在这个实现中，代码循环遍历键中的每一个字符，更新存储在局部变量 hash 中的值，该局部变量 hash 被声明为一个 unsigned 整数，并且初始化为一个看 673
似随机的常数 5381。在每次循环中，hashCode 函数把之前的 hash 值和一个称为 HASH_MULTIPLIER 的常量相乘，然后加上当前字符的 ASCII 码值。在循环的最后，结果不是 hash 值，而是用奇怪的表达式计算出来的：

```
int(hash & HASH_MASK)
```

考虑到在如此短的函数中有许多令人困惑的代码，你应该认同没有必要对复杂的 hashCode 函数进行详细理解。所有复杂性的关键是确保 hashCode 函数的结果均匀地分布在一个非负的整数区间。用户并不关心函数如何实现这个目标的细节，他们在意的是其本身作为一个理论问题的结果。

你选择哪个 hashCode 函数对实现的效率影响很大。例如，如果你用下面相对简单的实现，想一下会发生什么：

```
int hashCode(const string & str) {
   int hash = 0;
   int n = str.length();
   for (int i = 0; i < n; i++) {
      hash += str[i];
   }
   return hash;
}
```



这段代码很难理解。它的所有功能就是将字符串中字符的 ASCII 码值加起来，除非字符串特别长，否则这个值是一个非负整数。遗憾的是，除了长字符串会引起整数溢出并导致负数结果外，按这种方法编写 hashCode，如果哈希表的键刚好陷入某个模式，将很有可能会导致哈希表冲突。加 ASCII 码值的策略意味着任何彼此字母是置换关系的键都会有冲突。因此，cat 和 act 会哈希到同一个桶中。关键码 a3、b2 和 c1 也是如此。如果你将该哈希表应用于一个编译程序中，与这个模式相适应的变量名最后将全都哈希到相同的桶中。

以 hashCode 函数的实现代码更晦涩难懂为代价，你可以降低相似键冲突的可能性。但是，理解如何设计一个哈希函数需要相当多计算机科学理论的高深知识。大多数 674
hashCode 函数的实现使用了如第 2 章所述的与产生伪随机数相似的技术。在一个值域区间中，结果难以预测是非常重要的。在哈希表中，这种不可预测的结果正是程序员所选择的键，它们是不可能呈显出出偶然的期望更高的冲突的。

虽然仔细地设计哈希函数可以降低冲突的数目并提高其性能，但重要的是必须认识到算法的正确性不能被冲突率所影响。使用设计很差的哈希函数的实现运行很慢，但能给出正确

的结果。

15.3.3 实现哈希表

一旦你有了哈希函数的代码，实现的其余部分就相对容易编写了。StringMap 类的私有部分如图 15-7 所示，相应的实现展示在图 15-8 中。

```
/*
 * Notes on the representation
 * ---------------------------
 * This version of the StringMap class uses a hash table that keeps the
 * key-value pairs in an array of buckets, each of which is a linked list
 * of keys that hash to that bucket.
 */

private:

/* Type definition for cells in the bucket chain */

   struct Call {
      std::string key;
      std::string value;
      Call *link;
   };

/* Constant definitions */

   static const int INITIAL_BUCKET_COUNT = 13;

/* Instance variables */

   Call **buckets;              /* Dynamic array of pointers to cells */
   int nBuckets;                /* The number of buckets in the array */

/* Private methods */

   Call *findCall(int bucket, const std::string & key) const;

/* Make copying illegal */

   StringMap(const StringMap & src) { }
   StringMap & operator=(const StringMap & src) { return *this; }
```

675

图 15-7 StringMap 类哈希表实现的私有部分

```
/*
 * File: stringmap.cpp
 * -------------------
 * This file implements the stringmap.h interface using a hash table
 * as the underlying representation.
 */

#include <string>
#include "stringmap.h"
using namespace std;

/*
 * Implementation notes: HashMap constructor and destructor
 * ---------------------------------------------------------
 * The constructor allocates the array of buckets and initializes each
 * bucket to the empty list.  The destructor frees the allocated cells.
 */

StringMap::StringMap() {
   nBuckets = INITIAL_BUCKET_COUNT;
   buckets = new Call*[nBuckets];
   for (int i = 0; i < nBuckets; i++) {
      buckets[i] = NULL;
   }
}

StringMap::~StringMap() {
```

图 15-8 StringMap 类哈希表的实现代码

```
   for (int i = 0; i < nBuckets; i++) {
      Cell *cp = buckets[i];
      while (cp != NULL) {
         Cell *oldCell = cp;
         cp = cp->link;
         delete oldCell;
      }
   }
}

/*
 * Implementation notes: get
 * -------------------------
 * The get method calls findCell to search the linked list for the
 * matching key.  If no key is found, get returns the empty string.
 */

string StringMap::get(const string & key) const {
   int bucket = hashCode(key) % nBuckets;
   Cell *cp = findCell(bucket, key);
   return (cp == NULL) ? "" : cp->value;
}

/*
 * Implementation notes: put
 * -------------------------
 * The put method calls findCell to search the linked list for the
 * matching key.  If a cell already exists, put simply resets the
 * value field.  If no matching key is found, put adds a new cell
 * to the beginning of the list for that chain.
 */

void StringMap::put(const string & key, const string & value) {
   int bucket = hashCode(key) % nBuckets;
   Cell *cp = findCell(bucket, key);
   if (cp == NULL) {
      cp = new Cell;
      cp->key = key;
      cp->link = buckets[bucket];
      buckets[bucket] = cp;
   }
   cp->value = value;
}

/*
 * Private method: findCell
 * Usage: Cell *cp = findCell(bucket, key);
 * ----------------------------------------
 * Finds a cell in the chain for the specified bucket that matches key.
 * If a match is found, the return value is a pointer to the cell
 * containing the matching key.  If no match is found, findCell
 * returns NULL.
 */

StringMap::Cell *StringMap::findCell(int bucket, const string & key) const {
   Cell *cp = buckets[bucket];
   while (cp != NULL && key != cp->key) {
      cp = cp->link;
   }
   return cp;
}
```

图 15-8 （续）

StringMap 类的私有部分定义了两个实例变量：一个动态数组 buckets 和一个存储这个数组大小的整型变量 nBuckets。buckets 中的每个元素都是一个键－值对链表，它根据键哈希到相应的桶中。每个链中的单元都和你在前述示例中看到的链表队列类似，区别是它的每个单元都包含一个键－值对。在这个程序中，桶的数目是个常数，但之后的实现中将会对其进行改变。

676 | 677

如果看图 15-7 中实例变量 buckets 的声明，可能觉得这个语法初看起来有点奇怪。

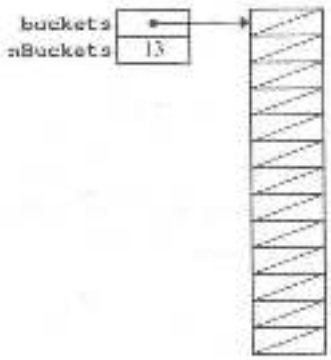
到现在为止，你为各种集合类创造的动态数组被声明为指向基本类型的指针。在这种情况下，其声明为：

```
Cell **buckets;
```

如之前的例子，buckets 是一个动态数组，在 C++ 中被描述为一个指向数组初始元素的指针。而且每个元素都是一个指向键－值对链表中第一个单元的指针。因此变量 buckets 是一个指向每个单元指针的指针，这就是上述声明语句中双星号的含义。

15.3.4 跟踪哈希表的实现

理解图 15-8 中哈希表的实现最简单的办法是完成一个简单的例子。构造函数创建一个动态数组，并且把 buckets 数组中的每个元素设置为 NULL，它表明这是一个空链表。这个结构如下图所示：



假设程序之后执行以下调用：

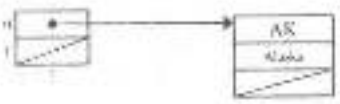
```
stateMap.put("AK", "Alaska");
```

put 函数代码的第一步是计算键 "AK" 的桶数，这需要计算 hashCode("AK") 的值。虽然 hashCode 函数的代码复杂，但至少按步骤运行一次还是很有必要的。如果你跟踪如图 15-6 所示的 hashCode 中 for 循环的每次执行，会看到执行涉及以下步骤：

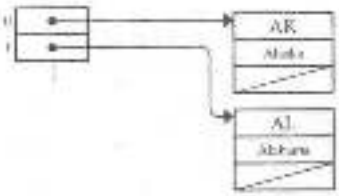
678

- 在 for 循环开始之前，变量 hash 被设置为常量 HASH_SEED，其值为 5381。
- 第一次循环通过将 hash 的初始值乘以 33，并加上字符 'A' 的 ASCII 码值 65 来更新变量 hash。因此更新后的 hash 值是 33×5381＋65，即结果为 177 638。
- 最后一次循环再一次用把 hash 乘以 33，并加上 'K' 的 ASCII 码值 75。hashCode 函数返回的值是 33×177 638＋75，即 5 862 129。

桶数就是 5 862 129 除以 13 的余数，这里刚好等于 0。put 方法因此将 "AK" 关联到 0 号桶中，而这个桶原本是空的。所以结果是一个仅包含了 Alaska 这个单元的链表，看起来如下图所示：



通过相同的步骤，你可以发现 "AL" 在 1 号桶中。



最后，特别是在一个有13个桶的哈希表中，会发生一个冲突。例如，关键码"KS"也会哈希到0号桶。put方法的代码必须遍历索引为0的链表以找到一个相匹配的键。因为"KS"没有出现，所以put方法在链表的开头加入一个新单元，如下图所示：

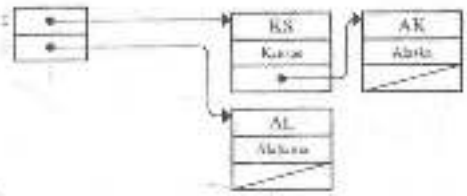


图15-9显示了50个州的缩写如何纳入到有13个桶的表中。缩写AK、KS、ME、RI和VT全部都哈希到0号桶，AL、MS、NY、OR、SC和WA全部哈希到1号桶，以此类推。通过把键分配到不同的桶中，get和put方法就可以搜索一个比较小的列表。同时，依据桶数而不是键的自然顺序来排列键很难按升序遍历键。为了能按键的升序遍历键，需要一种新的称为树（tree）的数据结构，它将在第16章描述。 679

15.3.5 调整桶的数目

虽然哈希函数的设计很重要，但是要清楚冲突的可能性也依赖于桶的数目。如果桶数很小，冲突发生将会很频繁。特别是，如果哈希表条目比桶数多，冲突将不可避免。冲突影响到哈希表的效率，因为put和get方法必须搜索更长的链表。当哈希表被填满时，冲突率会上升从而降低其性能。

重要的是记住这一点：使用哈希表的目的是优化put和get方法以便它们至少在平均情况下的常量时间内运行。实现这个目标需要每个桶的链表要短，反过来也就意味着桶数必须大于表的条目数。假设哈希函数能够很好地将键均匀地分配到各个桶中，则每个桶链的平均长度可用下面的公式计算：

$$\lambda = \frac{N_{\text{entries}}}{N_{\text{buckets}}}$$

例如，如果表中条目总数是桶数的3倍，每个链平均包含3个条目，也就是说为了找到一个键平均需要3个字符串间的比较。这个比率通常都可用希腊字母λ表示，它被称为哈希表的**负荷系数**（load factor）。

为了获得更好的哈希表性能，你要确保λ很小。虽然其数学细节已超出了本书的范围，但是保持负荷系数为0.7或更小，意味着在映射中查找一个键的平均时间是$O(1)$。更小的负荷系数表明哈希表中会有很多空桶，这浪费了一定的空间。哈希表是一个很好的时空权衡的实例，时空权衡这一概念在第13章中已作了介绍。通过增加哈希表所用到的空间，你可以提高其性能，但是把负荷系数降低到阈值0.7以下几乎没有任何益处。

除非哈希算法是为某个特定的已提前告知键数的应用程序而设计的，否则我们不可能为所有的用户都选择一个固定的桶数nBuckets值，并期望它能很好地为所有用户工作。如

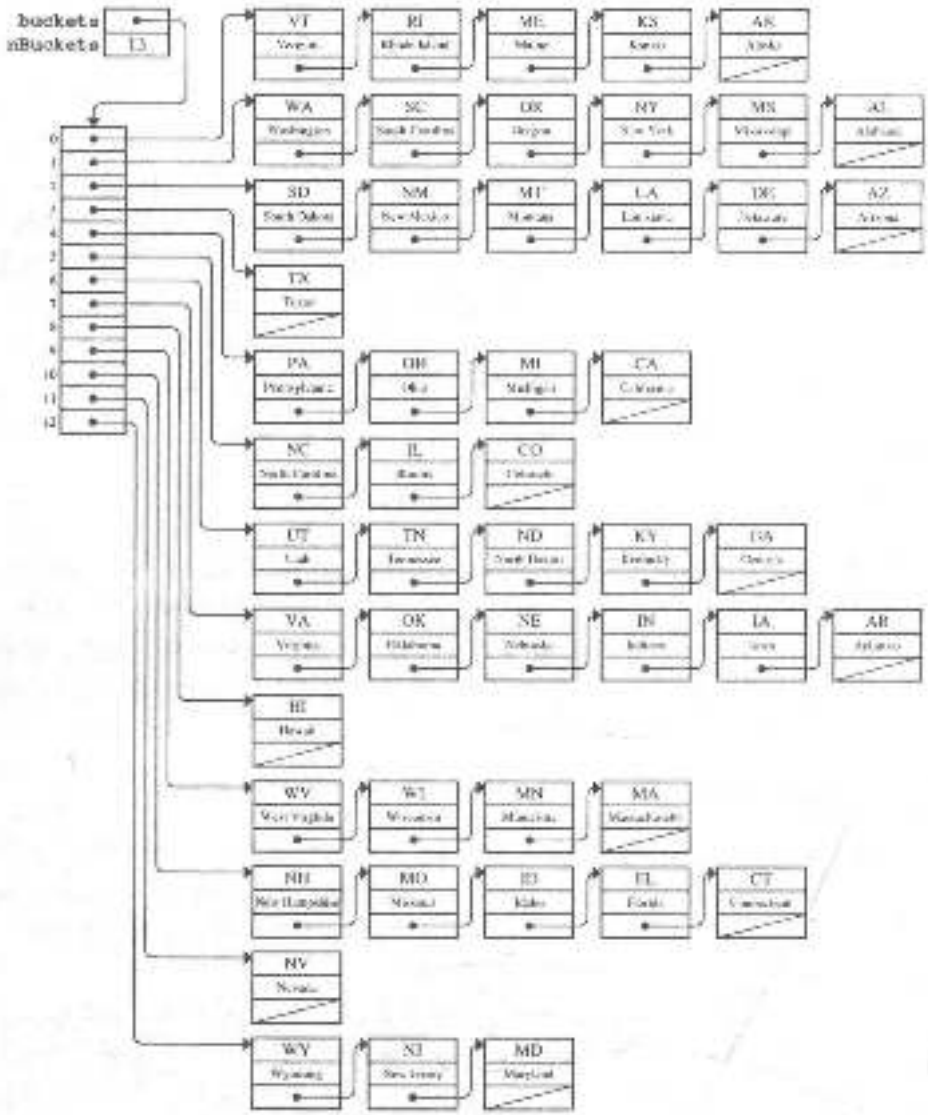


图 15-9 包含州缩写的哈希表

680
681

果用户不断往映射里添加条目，其性能最终会下降。如果你想要保持良好的性能，最好的办法就是实现桶数的动态增加。例如，你可以设计一种实现，当负荷系数达到某个阈值时，它可以重新分配一个更大的哈希表。遗憾的是，如果你增加桶的数目，桶数中的内容将全部改变，这意味着扩大哈希表的代码必须从旧表重新输入每一个键到新表。这个过程称为**重哈希**（rehashing）。虽然重哈希是非常耗时的，但是它不经常被执行，所以对应用程序总的运行时间影响很小。在习题 5 中，你将有机会实现这个重哈希策略。

15.4 实现 HashMap 类

到目前为止，本章的代码示例已实现了 StringMap 接口，而不是第 5 章介绍的普通的 HashMap 类的接口。完成 HashMap 类的实现，需要在代码上做以下改变：

- 添加遍历的方法。除了最简化的 StringMap 类提供的 put 和 get 方法外，HashMap

类还提供了 size、isEmpty、containsKey、remove 和 clear 方法，以及能够把映射当作关联数组的下标选择运算和一些必要的对深度拷贝提供支持的拷贝构造函数和重载的赋值操作符。

- 泛化键和值类型。HashMap 类使用模板参数 KeyType 和 ValueType 给用户更大的灵活性。

在很大程度上，上述变化都是显而易见的。只有一个变化有一些微妙之处，它使用模板功能支持用户定制的键类型。

实现哈希表的算法对键的类型有以下几点要求：

- 键的类型必须是可赋值的，以便代码能够存储哈希表单元中键的副本。
- 键的类型必须支持 == 比较操作符，以便代码可以区别两个键是否相同。
- HashMap 类模板扩展的时候，编译器必须使用 hashCode 函数的一个版本，为键类型的每一个值产生一个非负整数。内置类型如 string 和 int，其函数定义在 hashmap.h 接口中。对那些针对特定应用的类型而言，其函数必须由用户指定。

在很多情况下，用于其他类型的 hashCode 函数可能很简单。例如，用于整数的哈希函数可以简单地表示为：

```
int hashCode(int key) {
   return key & HASH_MASK;
}
```

[682]

常量 HASH_MASK 与 15.3.2 小节中定义的一样，包括除符号位为 0 其他位都为 1 的一个字。操作符 & 将在第 18 章进行详细讨论，在此的作用是从 key 中去除符号位以确保其 hashCode 值不为负。

为复合类型编写较好的哈希函数，通常需要一定程度的数学复杂性以确保哈希代码均匀地分布在某个非负整数区间。但是，有一个简单的权宜之计，就是引入 toString 方法为任何类型编写出一个合理的哈希函数。你需要把数值转变成字符串，然后使用哈希的字符串版本得到结果。使用这种方法，你可以编写一个针对 Rational 类的 hashCode 函数，如下所示：

```
int hashCode(const Rational & r) {
   return hashCode(r.toString());
}
```

虽然计算这个函数主执行除法算术运算需要更多的时间，但是这个代码更容易编写。

本章小结

本章主要讨论各种实现基本操作的策略，它们由 HashMap 类的库版本提供。Map 类能够按升序遍历键，它需要一个更复杂的数据结构树，这是第 16 章的主题。

本章的要点包括：

- 通过把键－值对存储在一个矢量中，可以实现基本的映射操作。保持矢量的排序顺序使得 get 方法的运行时间为 $O(\log N)$，但 put 方法仍为 $O(N)$。
- 特定的应用通过使用查找表来实现映射操作，get 和 put 方法的运行时间均为 $O(1)$。

- 采用称为哈希的策略可以高效地实现Map类。在其Map类的实现中，键被转化为一个整数，它可以确定在哪里找到其对应的值结果。
- 一个通用的哈希算法的实现是分配一个动态的桶数组，每个桶都包含一个哈希到这个桶的键的链表。只要Map对象中条目数与桶数的比例不超过0.7，get和put方法的平均执行时间均为$O(1)$。当条目数增长时，想要保持这样的性能需要定期重哈希以增大桶数。
- 哈希函数的详细设计是精妙的，为了获得最优的性能需要数学分析。虽然如此，任何输出非负整数的哈希函数都可以产生正确的结果。

复习题

1. 对于基于矢量的映射实现，为了把get方法的时间降低到$O(\log N)$，本章建议用什么样的算法策略？
2. 如果你实现了前面问题中建议的策略，为什么put方法仍然需要$O(N)$的时间？
3. 什么是查找表？查找表在什么情况下使用？
4. 使用键的首字符的ASCII码值作为其哈希码有何缺点？
5. 在哈希表的实现中，术语桶是什么含义？
6. 冲突表示什么？
7. 在图15-7中，stringmap.h接口的私有部分包括拷贝构造函数和赋值操作符的定义，它们使得StringMap对象不能复制。在较早的使用矢量作为底层表示的stringmap.h版本中，这些定义是不存在的。如果你复制一个基于矢量的StringMap会发生什么？
8. 图15-8显示了StringMap的哈希表版本的实现，解释这个实现中findCell方法的操作。
9. 本书中提及的字符串的hashCode函数有一个结构类似于随机数产生器。如果假设这种相似性，那么你会倾向于编写下面的hash函数：

```
int hashCode(const string & str) {
   return randomInteger(0, HASH_MASK);
}
```



684

为什么这种方法不行呢？

10. 如果你提供下面的hashCode函数，HashMap类还能正确运行吗？

```
int hashCode(const string & str) {
   return 42;
}
```

11. 在跟踪把州名缩写加入一个有13个桶的映射代码时，注意到条目"AK"和"XS"在0号桶发生冲突。假设新的条目按州名缩写的字母顺序被添加，第一个发生的冲突是什么？通过查看图15-9中的图表你应该能够猜出答案。
12. 在实现哈希表时有什么样的时空权衡？
13. 术语负荷系数是什么含义？
14. 为了确保HashMap类的平均时间性能为$O(1)$，负荷系数合适的阈值是什么？
15. 术语重哈希表示什么含义？
16. 每种键类型必须实现什么操作？
17. 假设你想要使用第6章的Point类作为HashMap类的键类型。为了实现必要的hashCode函数，本章提供了什么样的简单策略？

习题

1. 修改图 15-3 中的代码，以使 put 方法总会使数组中的键保持排序顺序。改变私有方法 findKey 的实现，使用二分查找在 $O(\log N)$ 时间内找到其键值。
2. 重写一个基于矢量的 StringMap 类的实现，它使用了一个 KeyValuePair 的动态数组，从而保证了 StringMap 类不再依赖于 Stanford 类库中的 Vector 类。
3. 从前面习题中描述的基于数组的 StringMap 类开始，增加方法 size、isEmpty、containsKey 和 clear 到 stringmap.h 接口及相应的实现中。
4. 虽然这使得数学运算更难，但罗马人使用不同的字母代表 5 和 10 的倍数。用来编码罗马数字的字符有以下值：

685

```
I  →    1
V  →    5
X  →   10
L  →   50
C  →  100
D  →  500
M  → 1000
```

设计一个查找表，使其能够确定每个字母所对应的数值。使用这个表实现一个函数：

```
int romanToDecimal(const string & str);
```

它把一个包含罗马数字的字符串翻译为它的数字形式。

为了计算罗马数字的值，一般把每个字母对应的值相加。然而，这个规则有一个例外：如果字母的值小于后面的字母值，它的值应该从总数里面减而不是加。例如，罗马数字串MCMLXIX相当于

$$1000-100+1000+50+10-1+10$$

即 1969。字母 C 和 I 因为它们之后的字母值较大而被减去。

5. 扩展图 15-7 和图 15-8 中 StringMap 类的实现，使得桶数可动态扩展。你的实现应该记录哈希表的负荷系数，并且在负荷系数超过阈值时执行重哈希操作。这个阈值用以下定义的一个常数表示：

```
static const double REHASH_THRESHOLD = 0.7;
```

6. 在特定的应用中，扩展 HashMap 类使得你可以为某个特定的键插入一个临时值，隐藏之前与键相关联的任何值。在程序的后面，你可以删除这个临时值，重新存储接下来最近的那一个键－值对。例如，你可以使用这样一种机制来获得局部变量的影响，当调用函数时这个变量值存在，当函数返回时该值消失。

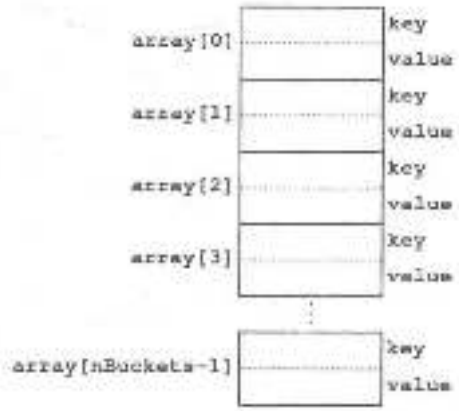
686

通过加入以下方法实现这个功能：

```
void add(const string & key, const string & value);
```

该方法会加入使用哈希表存储条目的 StringMap 类的实现中。因为 get 和 put 方法总是找到链表中的第一个条目，你可以通过在链表的开始处为特定的哈希桶增加新条目，以保证 add 方法隐藏了之前的定义。此外，只要 remove 的实现只删除了在哈希链中一个符号的第一次出现，你可以使用 remove 删除最近的新插入的键的定义，并同时将它后继的定义重新存储。

7. 按照 15.4 节中描述的策略实现 HashMap 类，就像你从第 5 章中了解的一样。
8. 为每个基本类型编写 hashCode 函数。
9. 虽然本书描述的桶链方法在实际中工作得很好，但是还存在其他的策略可解决哈希表中的冲突。在计算的早期（内存很小、额外的指针必须被认真对待），哈希表常常使用一种更节省内存的被称为开放寻址（open addressing）的策略，其中的键－值对被直接存储在数组中，如下图所示：



例如，如果一个键被哈希到 2 号桶中，开放寻址策略会试着将键和它的值直接放到 array[2] 中的条目内。

687 采用这种方法的问题是：array[3] 可能已经被分配给其他哈希到相同桶的键。解决这类冲突最简单的方法就是把每一个新的键存放在第一个自由单元或在它期望的哈希位置之后。因此，如果一个键哈希到 2 号桶，put 和 get 方法第一次会在 array[2] 中找到或者插入这个键。如果该位置已经有了一个不同的键，那么这些函数就会继续移到 array[3] 处，继续这一过程，直到找到一个空的位置或者一个匹配的键。正如第 14 章中环形缓冲队列的实现，如果索引越过指向数组的末端，它会绕回到开始处。这个解决冲突的策略叫做线性探测（linear probing）。

重新实现 StringMap 类，使用具有线性探测的开放寻址。对于这个练习，如果用户想在一个满的哈希表中增加一个键，你的实现应该显示错误。

10. 扩展你对习题 9 的解决方法，使得无论何时负荷系数超过常量 REHASH_THRESHOLD 时，它都能动态地扩充容易，正如习题 5 中所定义的。在那个练习中，你需要重建条目表，因为针对键的桶

688 数在你给 nBuckets 赋新值时会改变。

第16章
Programming Abstractions in C++

树

我喜欢树，因为它们比其他事物看起来更顺从于生活。

——薇拉·凯瑟，《啊，拓荒者！》(*O Pioneers!*)，1913 689

正如前几章所述，在不使用数组的情况下，链表可用来表示一个有序的数据集合。由指针链接起来的每个单元构成线性链表，该链表定义了底层的顺序。尽管链表相对于数组要求更多的空间，而且在选取指定索引位置的值时显得更为低效，但是它的优点是能在常量时间内进行插入和删除操作。

在数据集合中，用指针来定义它们的关系比用链表所带来的优势要强大得多，它不仅仅限制于建立线性结构。在本章，你将学习一种使用指针来模拟层次关系的数据结构。该数据结构被称为树（tree），它被定义为一个节点（node）的集合，并具有以下性质：

- 只要树中有节点，一定存在着一个特定的节点，被称之为根（root）节点，它处于树的最顶层。
- 树中的每个节点与根节点只有唯一的一条通路。

树结构层次也出现在计算机科学以外的其他许多领域中。我们最熟悉的一个实例就是家谱，将在下一节讨论，其他的示例还包括：

- 游戏树（game tree）。第9章“最小最大算法”一节中曾提到过的分支模式是一种很典型的树。当前位置是树的根，各个分支通向游戏中接下来可能出现的场景。
- 生物分类（biological classification）。生物的分类系统是18世纪出瑞典的植物学家洛鲁斯·林奈发明的一种树状结构。该树的根节点是所有生物。然后分类系统的不同分支构建起了各自的王国，最常见的就是植物和动物。这样的层次继续下去，直到一个独立的物种为止。
- 组织图（organizational chart）。许多商业团体都是每个雇员隶属于一个主管，这样就建立了一棵树，最顶部是公司的总裁，代表根节点。
- 目录层次（directory hierarchy）。在大多数现代计算机中，文件存储的目录构成一棵树。顶部的目录代表根节点，它包含其他文件和目录。这些目录又有自己的子目录，以此构成了树的层次结构。 690

16.1 家谱

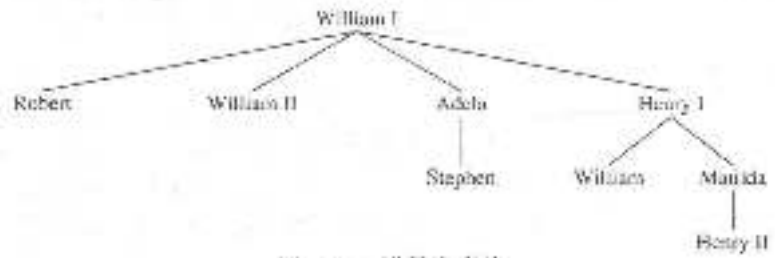
家谱提供了一种便捷的方式来表示从单个个体到若干代血缘关系的方法。例如，图16-1表示了诺曼底家族的家谱，该家族在1066年的哈斯丁战役后统治了英国。该图的结构符合上节的树定义。威廉一世是树根，其他人都以唯一的下降路径连接到威廉一世。

16.1.1 用来描述树的术语

图16-1的家谱使得介绍计算机科学中用以描述树结构的术语变得简单了。书中的每一个节点都可以有好几个孩子节点（children node），但是只有一个父节点（parent node），在树

中，**祖先**（ancestor）和**子孙**（descendant）的含义与日常语言中的含义完全一致。从 Henry I 到 Matilda 的下降路线表示 Henry II 是 William I 的一个子孙，反过来讲，Henry I 是 Henry II 的一个祖先。类似地，两个节点如果共享一个共同的父节点，例如 Robert 和 Adela，则这两个节点就被称为**兄弟**（sibling）。

尽管用来描述树的大部分术语都直接来自于家谱的比拟，但是某些术语（例如根）来自于植物学。与根节点相反的没有孩子节点的节点被称为**叶子节点**（leaf node）。既不是根节点也不是叶子节点的节点被称为**内部节点**（interior node）。例如，在图 16-1 中，Robert、William II、Stephen、William 和 Henry II 都是内部节点。一个非空树从根节点到叶子节点的最长路径被称为树的**高度**（height）。因此，图 16-1 中树的高度为 3。因为从 William I 到 Henry II 的路径长于其他任何一条路径。按照惯例，空树的高度一般定义为 -1。



691 图 16-1 诺曼底家族

16.1.2 树的递归特性

最值得注意的是：树的分支结构也适用于任意层次的分解。如果你去除树中任意一个节点以及相应的子孙，其结果仍然满足树的定义。例如，如果你从图 16-1 中抽取 Henry I 及其子孙，那么得到以下的树：



从树中取出一个节点及其子孙所构成的树被称为原来树的**子树**（subtree）。例如，上图的树就是以 Henry I 为根节点的一棵子树。

树中的每个节点都可以看成是它自己子树的根，这就强调了树结构的递归特性。如果你以递归的视角考察树，那么树是一个节点和一个其附着子树的集合——当为叶子节点时，该集合为空。树的递归特性是其底层表示和大部分针对树操作的算法基础。

16.1.3 用 C++ 语言表示家谱

为了用 C++ 表示一棵树，需要以某种方式模拟数值之间的层次关系。在大多数情况下，表示父子关系最简单的方式就是在父方包含一个指向子方的指针。如果使用这个策略，每个节点除了存储自己的数据之外，还需要存储指向其每个孩子节点的指针。通常，将节点定义为结构，将树定义为指向该结构的一个指针，就能很好地实现树的表示。这种树的定义即使是以自然语言表述也是递归的，因为它们拥有如下关系：

- 树是指向节点的指针。

- 节点是包含树的结构。

你可以使用这种递归思想来设计一个适合存储如图 16-1 所示家谱中的数据结构。每个节点包含一个人名以及指向他们孩子节点指针的集合。如果将孩子指针存储在一个矢量中，那么节点的结构如下所示：

```
struct FamilyTreeNode {
   string name;
   Vector<FamilyTreeNode *> children;
};
```

一棵家谱树简单来说就是一个指向这些节点的指针。

图 16-2 展示了该皇族家谱的内部实现。为使家谱图整齐有序，图 16-2 表示的孩子节点看起来像是一个五元数组；事实上，children 域是一个随着孩子数量增加的矢量。你将有机会寻找其他策略来存储孩子节点，例如在本章结尾的习题中，可以使用链表而不是矢量。

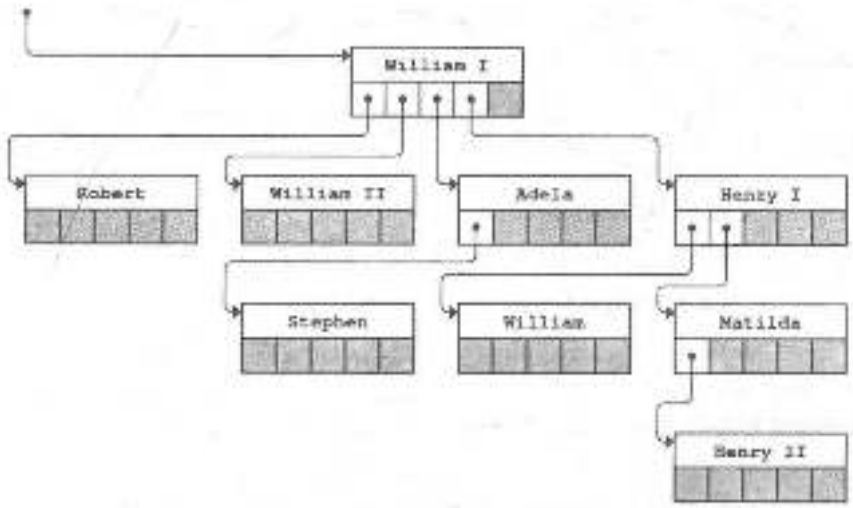


图 16-2 诺曼底家族的基于指针的树表示

16.2 二叉搜索树

尽管可以使用家谱树来说明树的算法，但在简单情境下进行直接编程将是更有效的方法。虽然家谱树的例子为我们描述树的术语提供了一个框架，但是每个节点可以有任意数量的孩子节点使得实际实现变得复杂。在许多情况下，限制孩子节点的数目使得树更易于实现是很合理的。

树的一棵最重要的（且有很多实际应用的）子树就是二叉树（binary tree），它是一棵树并具有以下额外特性：

- 树中每个节点至多有两个孩子节点。
- 除了根节点外，每个节点被指定为要么是父节点的左孩子（left child），要么是父节点的右孩子（right child）。

上述第二个条件强调了在二叉树中的孩子节点相对于其父节点而言是有序的。例如以下二叉树：



尽管它们包含了相同的节点，但它们是两棵不同的树。在这两种情况下，B 节点是根节点 A 的孩子节点，但是在左边的树中，B 是左孩子节点，而在右边的树中，它是右孩子节点。

事实上，在二叉树中的节点之间存在已定义的几何关系，这使得使用二叉树表示有序的数据集合变得更加方便。大多数应用都使用一种称为**二叉搜索树**（binary search tree）的具有特殊类的二叉树。该树一般缩写成 BST，它具有以下特性：

1. 每个节点都包含（也可能包含其他的数据）一个特定的被称为键的值，它定义了节点的顺序。

2. 键值是唯一的，也就是说任何键值在树中只能出现一次。

3. 树中的每个节点，其键值必须大于以它左孩子节点为根节点的子树的所有节点的键值，一定小于以它右孩子节点为根节点的子树的所有节点的键值。

尽管该定义形式上是正确的，但是一开始看确实很令人困惑。方清楚树的定义和理解树满足的上述条件是非常有用的，它能够帮助我们评判关于用二叉树作为解决策略的典型问题。

694

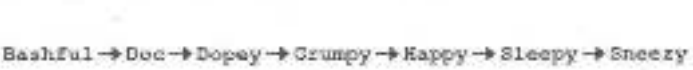
16.2.1 使用二叉搜索树的动机

在第 15 章哈希算法之前所提出的表示映射的一个策略是将键-值对存放在一个矢量中。这个策略有一个有用的计算特性：如果你保持键值的排列顺序，可以编写一个运行时间为 $O(\log N)$ 的 get 函数的实现。你所要做的就是采用第 7 章介绍的二分查找算法。遗憾的是，数组表示无法编写出具有相同效率的 put 函数。尽管 put 方法能够使用二叉搜索树来决定新的键值要往哪里插，但是维护所存储的顺序则需要 $O(N)$ 时间，因为数组中在欲插入新值元素位置后的每一个后续元素都必须向后移动以为新元素提供空间。

这个问题使我们想起第 13 章中的一个类似情景。当使用数组表示编辑器缓冲区时，插入一个新字符为线性时间操作。在那个示例中，解决方案是采用链表代替数组。那么在映射中可以采用相似的策略来改进 put 方法的性能吗？毕竟，只要你有一个指向插入点的前一个元素的指针，在链表中插入一个新元素就是一个常量时间的操作。

用链表表示的问题在于它们不支持任何有效的二分查找算法。二分查找依赖于能以常量时间找到元素集合中的中间元素。在一个数组中，找中间值很简单。在链表中找中值唯一的方法是遍历链表前半部分所有的链接指针。

为了更具体地理解链表的这些限制，假设有一个包含沃特·迪斯尼的七个矮人名的链表：



该链表中的元素以字典顺序排序，即按照内部字符码排列的顺序。

给定一个这种排序的链表，可以很容易地找到其中的第一个元素，因为内部指针给出了它的地址。然后，你可以顺着链接指针找到第二个元素。另一方面，没有方便的方法可定位序列的中间元素。为此，不得不遍历链表中的每个指针，直到计数 $N/2$ 处为止。在链表中找中值的操作需要线性时间，它会完全抵消二分查找的效率优势。如果二分查找旨在改进效

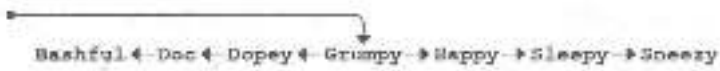
率，那么该数据结构必须能够快速找到中间元素。 695

尽管这样做第一眼看起来很傻，但是想象如果简单地将指针指向链表中间而不是开头将会发生什么是很有用的：



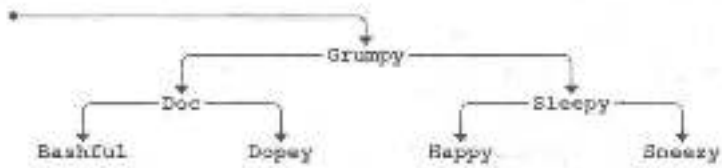
在这个图中，你能很容易找到中间元素。通过链表指针可以立即找到这个链表中的中间元素。然而，问题是你已经将链表一半的元素丢掉了。该结构中的指针据仅了访问第人 Grumpy 及其之后的所有的名字的方法，但是却不能再访问 Bashful、Doc 和 Dopey 了。

如果你站在 Grumpy 的角度来思考，一个一般的问题解决方案就变得清晰起来。你需要从 Grumpy 节点散射出两个链表：一个链表包含 Grumpy 前面的元素，另一个链表则包含 Grumpy 后面的元素。在概念图中，你所要做的就是翻转箭头：



现在链表中的每个字符串都能访问。你可以很容易地把整个链表一分为二。

此时，你需要递归地应用此策略。二分查找算法不仅仅需要你找出原链表的中间元素，还有子链表的中间元素。因此，你需要使用相同的分解策略重构 Grumpy 前面和后面的链表。每一个元素都指向两个方向：指向前面链表的中间点和后面链表的中间点。应用此过程将原链表转化成以下的二叉树：



这种特定风格的二叉树的一个最重要的特点是元素为有序的。对于树中任何特定节点，它所包含的字符串必须在其左子树中所有元素的后面，而在其右子树中所有元素的前面。在该例中，Grumpy 跟在 Doc、Bashful 和 Dopey 的后面，但却在 Sleepy、Happy 和 Sneezy 的前面。同样的规则应用到树中的各个层次，因此包含 Doc 的节点在 Bashful 节点的后面，而在 Dopey 节点的前面。在前面小节结尾的关于二叉搜索树的正式定义保证了 696
树中的每个节点都遵循有序原则。

16.2.2 在二叉搜索树中寻找节点

二叉搜索树一个最基本的优点就是你可以使用二分查找算法来寻找一个特定的节点。例如，假如你要在前面一节最后的树中寻找包含字符串 Happy 的节点。第一步是将 Happy 和树根节点 Grumpy 作比较。在字典中，Happy 位于 Grumpy 之后，所以你应该清楚：如果树中字符串 Happy 存在，那么一定在 Grumpy 的右子树中。接下来，就比较 Happy 和 Sleepy。在这种情况下，Happy 在 Sleepy 前面，因此它必定在 Sleepy 的左子树中。该子树只有一个节点，它正好就是我们要找的。

因为树是一种递归结构，所以很容易以递归的形式编写搜索算法。为了具体说明，让我们假设 BSTNode 的类型定义如下：

```
struct BSTNode {
   string key;
   BSTNode *left, *right;
};
```

给定此定义，很容易编写出以下 findNode 函数来实现二分查找算法：

```
BSTNode *findNode(BSTNode *t, const string & key) {
   if (t == NULL) return NULL;
   if (key == t->key) return t;
   if (key < t->key) {
      return findNode(t->left, key);
   } else {
      return findNode(t->right, key);
   }
}
```

如果树为空，所求节点必然不在树中，findNode 函数就会返回 NULL，表示找不到该关键字。如果树不为空，函数会检验所求节点与当前节点是否匹配。如果匹配，findNode 函数返回指向当前节点的指针。如果不匹配，findNode 函数就向前递归，至于是向左子树还
697 是向右子树中寻找，取决于键值比较的结果。

16.2.3 在二叉搜索树中插入一个新节点

接下来首先要考虑的问题是如何建立一个二叉搜索树。最简单的方法是以空树开始，然后调用 insertNode 函数来向树中每次插入一个新的键值。当所有新的键值被插入后，维持树中节点的有序关系很重要。为了确保 findNode 函数继续工作，insertNode 函数必须使用二叉搜索树来辨别正确的插入点。

结合 findNode，insertNode 函数可以从树根开始递归向前插入节点。在每个节点上，insertNode 必须将新的键值与当前键值进行比较。如果新键值小于当前键值，则将新键值插入到其左子树中，相反，如果新键值大于当前键值，就将其插入到右子树中。最后，程序会遇到一个空子树，代表树中需要插入新节点的地方。在该点上，insertNode 函数用包含了那个键值的已初始化的新节点代替 NULL 指针。

然而，insertNode 函数的代码有点复杂。其难点是 insertNode 必须能够通过增加一个新键值来改变二叉搜索树的键值。既然 insertNode 函数需要改变参数值，因此必须采用引用方式传递参数，而不能像 findNode 函数那样取 BSTNode * 类型的参数。因此，insertNode 函数的原型如下所示：

```
void insertNode(BSTNode * & t, const string & key);
```

一旦你理解了 insertNode 函数的原型，编写它就不会太困难。函数实现如下所示：

```
void insertNode(BSTNode * & t, const string & key)
   if (t == NULL) {
      t = new BSTNode;
      t->key = key;
      t->left = t->right = NULL;
   } else {
      if (key != t->key) {
         if (key < t->key) {
            insertNode(t->left, key);
         } else {
```

```
            insertNode(t->right, key);
        }
    }
}
```

698

如果 t 为 NULL，则 insertNode 函数创建一个新节点，并初始化其数据域，然后用一个指向新节点的指针代替当前节点的 NULL 指针。如果 t 不为 NULL，insertNode 函数会使用树 t 的根节点的键值与新键值比较。如果匹配，说明该键值已在树中，不需要进行比较操作了。如果不匹配，insertNode 使用比较结果来决定是插入到左子树还是右子树，然后执行相应的递归调用。

因为 insertNode 的代码看起来有点复杂，所以跟踪插入几个键值的过程细节很有必要。例如，假如你已经声明并初始化了一棵空树，如下所示：

```
BSTNode *dwarfTree = NULL;
```

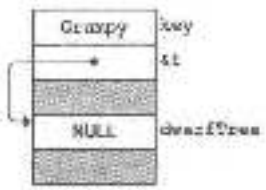
该语句在包含该声明的函数的栈帧中创建了一个局部变量 dwarfTree，如下图所示：



如果你调用以下语句，会发生什么：

```
insertNode(dwarfTree, "Grumpy");
```

如果 dwarfTree 为空指针，会出现什么情况？在 insertNode 的栈帧中，引用变量 t 是指向变量 dwarfTree 的指针。因此该调用开始的栈帧如下图所示：

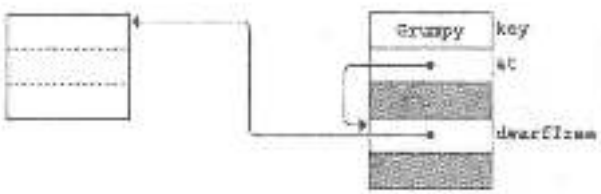


代码首先检查 t 是否为 NULL，此时符合该情况。程序继续以下行语句开始的并带有 if 语句函数体执行：

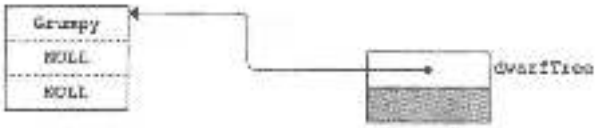
699

```
t = new BSTNode;
```

该行代码在堆存储区给新节点分配内存，并且将其赋值给引用参数 t，因此指针单元发生改变，如下图所示：

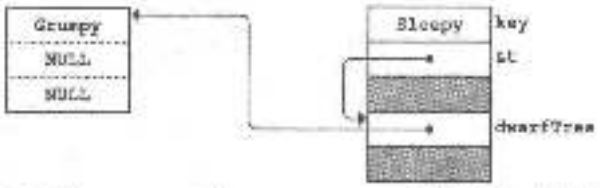


接下来的语句将键值 Grumpy 复制到新节点的域中，并且将每一个子树的指针初始化为空指针。当 insertNode 函数返回时，树的状态如下图所示：



该结构正确地表示了包含单个节点 Grumpy 的二叉搜索树。

如果再使用 insertNode 向树中插入 Sleepy 将会怎样呢？和之前一样，初始调用会产生一个栈帧，其中引用参数指向 dwarfTree：

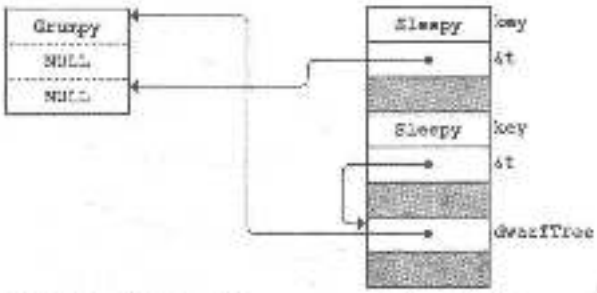


然而，这次树 t 的值不再为 NULL，因为 dwarfTree 变量现在已经包含了 Grumpy 节点的地址。在字典中，Sleepy 在 Grumpy 的后面，故 insertNode 函数继续如下递归调用：

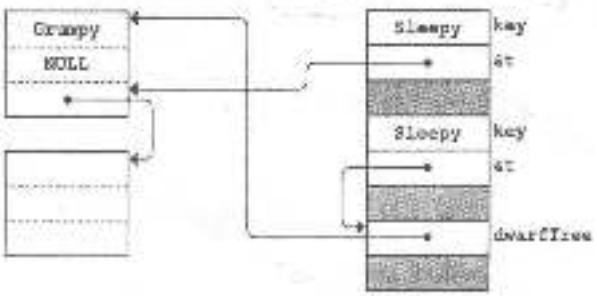
```
insertNode(t->right, key);
```

此时，递归调用看起来像是在给原来空树插入 Grumpy 一样，唯一的不同是引用参数 t 现在指向一个已经存在的节点，如下图所示：

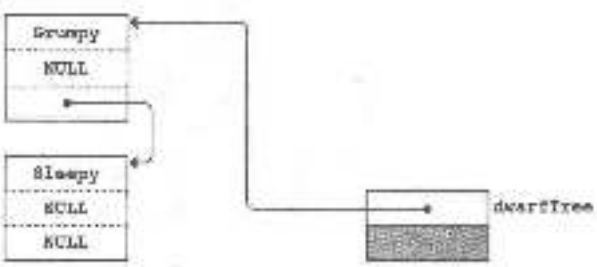
700



由 insertNode 分配的新节点代替了节点 Grumpy 的右孩子节点，如下图所示：



在新节点中填入数据后，对 insertNode 的调用将返回下图所示的树：



其他对 InsertNode 的调用又会创建新的节点，并且将它们以二叉搜索树要求的顺序插入到该树结构中。例如，如果将剩下的其他五个小矮人按照 Doc、Bashful、Dopey、Happy 和 Sneezy 的顺序插入，最后得到的二叉搜索树如图 16-3 所示。 701

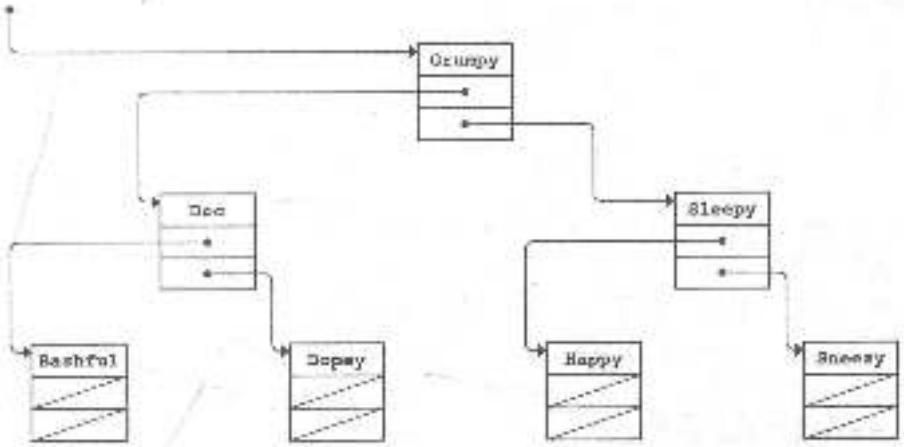
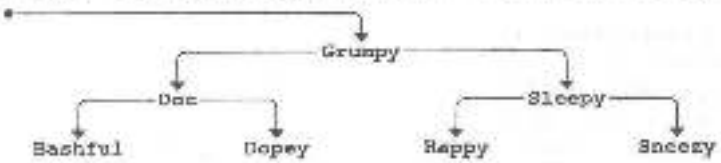


图 16-3 包含七个小矮人的二叉搜索树的结构图

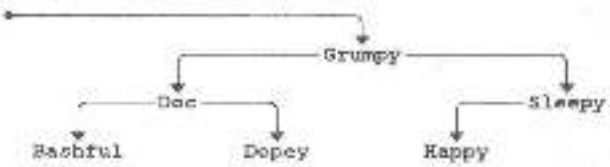
16.2.4 删除节点

从二叉搜索树中删除节点比插入一个节点要更复杂一些。找出要被删除的节点比较简单。你只需采用同样的二分查找策略来定位一个特定键所在的位置。一旦找到匹配的节点，就必须在不违反二叉搜索树顺序关系的前提下从树中将其删除。由于删除节点取决于该节点在树中的位置，所以难度很大。

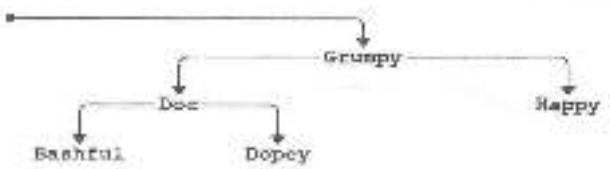
为了理解这个问题，假设你在处理包含下面七个小矮人名字的二叉搜索树：



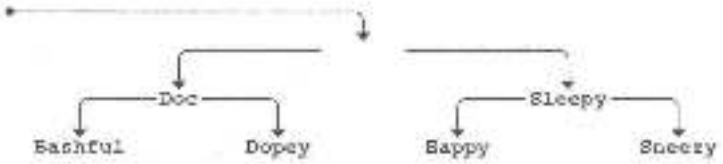
删除 Sneezy（大概是创建了一种不好的工作环境）是容易的。你需要用一个 NULL 指针代替指向 Sneezy 的指针，这就形成了下面的树： 702



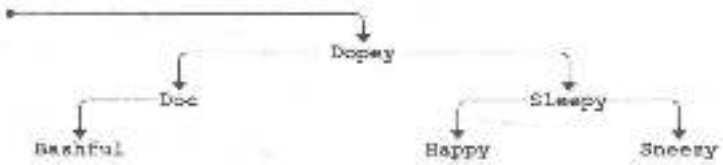
从此开始，删除 Sleepy 也相对容易一些。如果你想删除的节点的某个孩子节点为空，你所要做的就是用非空的那个子节点来替换它，如下图所示：



然而，如果你尝试删除既有左孩子节点又有右孩子节点的节点，就会出现一个问题。例如，假如你想从包含所有的七个小矮人的原始树中删除 Grumpy（为了避免错误）。如果简单地将其移除，就会形成两部分查找树，一个以 Doc 为根节点，一个则以 Sleepy 为根节点，如下图所示：



此时，你想做的可能就是找一个节点来插入移除 Grumpy 节点留下的空间。为了确保删除结果也是一个二叉搜索树，只有两个节点可以使用：左子树最右边的节点或右子树最左边的节点。这两个节点作用等价。例如，如果你选择了左子树最右边的节点，即 Dopey 节点，该节点键值大于左子树其他键值，小于右子树所有键值。为了完成删除，你需要以 Dopey 的左孩子节点来代替 Dopey，当然，可能像示例中是个空指针。然后将 Dopey 移到删除的节点上。最终结果如下图所示：



703

16.2.5 树的遍历

二叉搜索树的结构使得遍历按键值顺序排列的节点变得很容易。例如，你可以使用如下方法以字典顺序展示二叉搜索树的键值：

```
void displayTree(BSTNode *t) {
   if (t != NULL) {
      displayTree(t->left);
      cout << t->key << endl;
      displayTree(t->right);
   }
}
```

因此，如果调用 displayTree 处理图 16-3 所示的树，你将会得到以下输出：

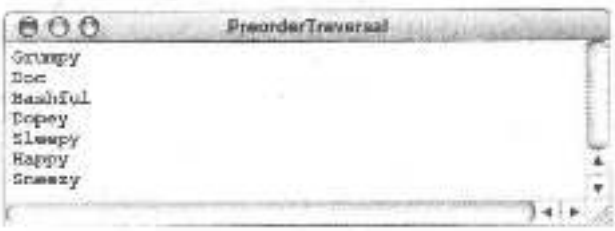


在每个递归层次上，displayTree 都会检查树是否为空。如果为空，则不再执行。否则，递归调用的顺序确保了以正确的顺序输出结果。第一次递归调用显示当前节点前的键值，它们都必须出现在左子树中。因此，在当前节点前显示左子树的键值，就维持了正确的显示顺序。类似地，在执行最后一个调用前显示当前节点的键值就非常重要，该递归调用将显示在 ASCII 序列中靠后出现并因此出现在右子树中的键值。

遍历树中的节点，并且在每个节点上进行某种操作的过程被称为**树的遍历**（traversing / walking the tree）。在很多情况下，你总是想以键的顺序遍历一棵树，正如在 displayTree 中示例的那样。该方法是首先处理当前节点的左子树，然后是该节点，之后是该节点的右子树的一个递归调用过程。这种遍历称为**中序遍历**（inorder traversal）。然而，在下文中还有其他两种类型的树的遍历方式，分别被称为**先序遍历**（preorder traversal）和**后序遍历**（postorder traversal）。在先序遍历中，当前节点在它的两个子树前处理，如以下代码所示： 704

```
void preorderTraversal(BSTNode *t) {
   if (t != NULL) {
      cout << t->key << endl;
      preorderTraversal(t->left);
      preorderTraversal(t->right);
   }
}
```

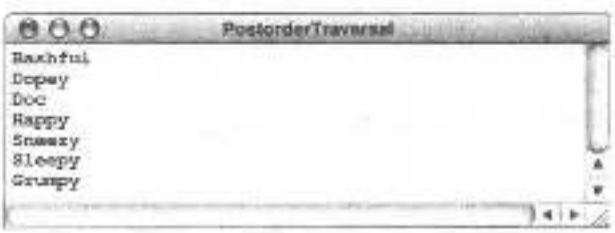
假定有图 16-3 所示的树，在以下示例输出中，先序遍历输出各节点：



在后序遍历中，先处理左右子树，然后处理当前节点。后序遍历的代码如下所示：

```
void postorderTraversal(BSTNode *t) {
   if (t != NULL) {
      postorderTraversal(t->left);
      postorderTraversal(t->right);
      cout << t->key << endl;
   }
}
```

用包含七个小矮人的二叉搜索树执行上述函数，得到以下结果：



705

16.3 平衡树

尽管实现 insertNode 用到的递归策略保证了将节点组织为一棵合法的二叉搜索树，但是树的结构取决于节点要插入的位置。例如图 16-3 的树，由按以下顺序插入的名字序列产生：

```
Grumpy, Sleepy, Doc, Bashful, Dopey, Happy, Sneezy
```

假设你按照字母表顺序输入上述这些名字。那么第一次调用 insertNode 时会在根节点插

入 Bashful。连续的 insertNode 调用会在 Bashful 之后插入 Doc，在 Doc 之后插入 Dopey，以此类推。上述每个新插入的节点都附加在先前节点的右子树上。

最后所形成的图 16-4 的树看起来更像是一个链表。无论如何，图 16-4 的树保持着这样一个特性：任何节点的键值都大于其左子树的所有节点的键值，而小于其右子树的所有节点的键值。因此它符合二叉搜索树的定义，使得函数 findNode 将会被正确调用。然而，findNode 函数算法的运行时间与树的高度成正比，这也就意味着树的结构对算法行为有很大影响。如果一个二叉搜索树如图 16-3 所示，那么在树中找出键值的时间为 $O(\log N)$。另一方面，如果树如图 16-4 所示，运行时间将会恶化为 $O(N)$。

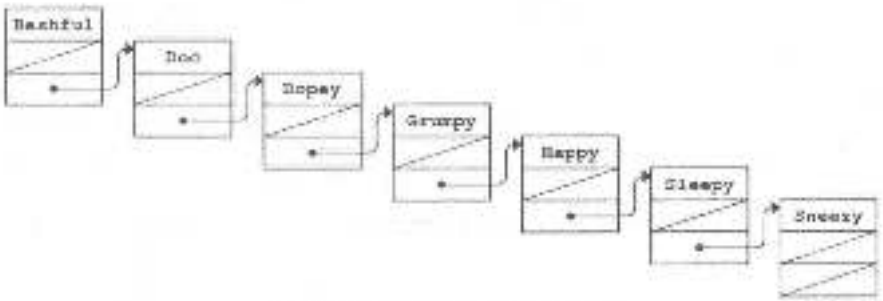


图 16-4 非平衡的二叉搜索树

只有当左子树和右子树大体上高度相同时，用以实现 findNode 函数的二叉搜索算法才会达到理想的性能。具有上述特性的树，即如图 16-3 所示的树，被称为是平衡的
706 (balanced)。更正式地讲，如果对于树中的每一个节点，其左子树和右子树的高度相差不超过 1，那么一个二叉树被定义为平衡树。为了说明平衡二叉树的定义，图 16-5 所示的上面一行的各棵树均为具有 7 个节点的平衡树，而下面一行的各个树均为非平衡树。在上述各棵树中，不满足平衡树定义的节点用空心圆表示。例如，在非平衡树最左边的那棵树中，其根节点的左子树高度为 2，而右子树高度为 0。在另外两个非平衡树中，由于根节点有不平衡的孩子节点，故它们也是不平衡的。

在图 16-5 中的第一棵树是一棵最佳的平衡二叉树，因为树中每个节点左右子树的高度都相等。然而，只有当树中节点数等于 2^k-1 时，才有可能产生这种最佳平衡二叉树。如果树中的节点数不满足上述条件，则树中肯定会出现某些节点的左右子树的高度不同的情况。由于平衡二叉树允许节点左右子树的高度可相差 1，因此，在不影响计算性能的情况下，平衡二叉树在树的结构上提供了某种灵活性。

16.3.1 平衡树策略

只有避免与非平衡树相关的最差情况出现，二叉搜索树在实际应用中才有用。当树不平衡时，findNode 和 insertNode 操作的运行时间就会变为线性。如果二叉树的运行时间恶化到 $O(N)$，你最好使用一个有序数组来存储节点。有序数组需要 $O(\log N)$ 时间实现 findNode，$O(N)$ 时间实现 insertNode。从计算角度来看，一个基于数组的表示更有可能优于基于平衡树的表示，而且数组表示更易于编码。

二叉搜索树之所以能作为一种有用的编程工具，其原因在于你可以在建树的时候保持其平衡。它的基本思想就是扩展 insertNode 的实现，使之在插入新节点时追踪检查树是否平衡。一旦树不平衡，insertNode 在不打乱二叉搜索树顺序的情况下，必须重新分布树

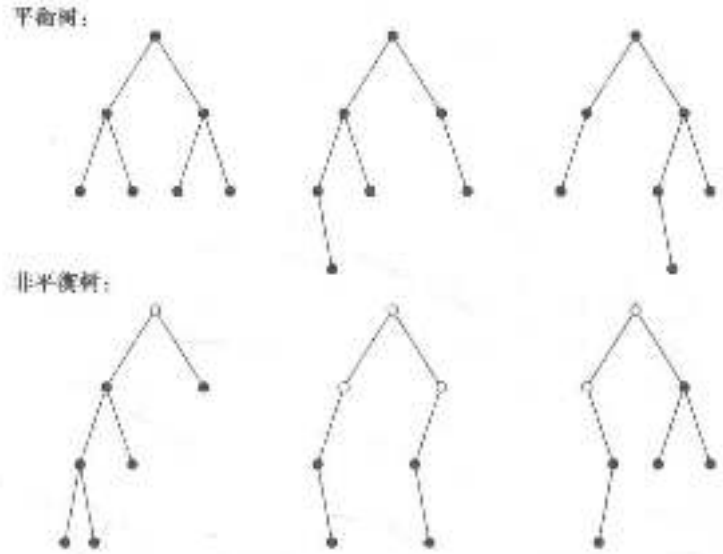


图 16-5 平衡和非平衡二叉树的示例

中的节点使之保持平衡。假设可以重新分布树中节点的时间与树的高度使之成正比，则 findNode 与 insertNode 可以在 $O(\log N)$ 时间内实现。

维持二叉搜索树平衡的算法在计算机科学领域已得到深入研究。现在所使用的实现平衡二叉树的算法都是计算机科学领域广泛的理论研究结果。然而，其中的大多数算法，如果不复习已超出了本书范围的相关数学基础是很难解释清楚的。为了说明这些算法确实可行，我们在接下来几节中将展示第一代平衡树算法，它由俄国数学家乔吉·安德森·维斯基（Georgii Adelson-Velskii）和艾维基尼·兰蒂斯（Evgenii Landis）于 1962 年提出，被称为 AVL（取两人名字的首字母）算法。尽管 AVL 算法已在很大程度上被现代算法所取代，但是它比现代算法更容易解释。而且，由于用于实现 AVL 算法操作的基本策略也用于其他的算法中，因此，这使得 AVL 算法仍是很多现代技术的一个很好模型。

16.3.2 可视化 AVL 算法

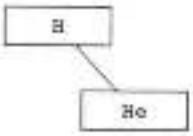
在你试图理解 AVL 算法的实现细节之前，回顾一下向二叉搜索树中插入节点的过程，并看一下会出现什么错误。如果可能的话，请再想象一下你将要如何处理所出现的问题。让我们想象一下：你要创建一个包含化学元素符号的二叉搜索树。例如，树中的前六个元素是：

707 / 708

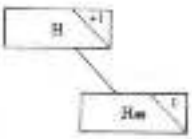
```
B   (硼)
Be  (铍)
Li  (锂)
Ba  (钡)
B   (硼)
C   (碳)
```

如果你以默认顺序（即以这些元素在周期表上的顺序）插入这些元素符号，会出现什么情况呢？因为树初始为空，所以插入第一个元素很简单，即创建了一个仅包含符号 B 的根节

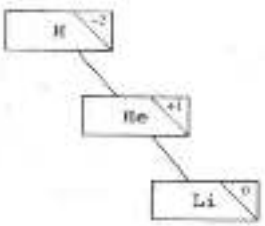
点。如果调用 InsertNode，由于 He 以字典序排在 H 之后，所以它将加入到 H 节点之后。因此，树中的头两个节点的分布如下图所示：



为了追踪树是否平衡，AVL 算法给树中的每个节点关联一个表示其右子树与其左子树高度之差的整数值，它被称为该节点的平衡因子（balance factor）。在上述仅含前两个元素符号的简单树中，平衡因子显示在每个节点的右上角，如下图所示：



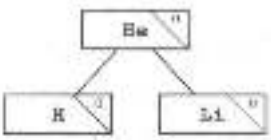
到目前为止，该树是平衡的，因为没有任何节点的平衡因子的绝对值大于 1。然而，当你加入下一个元素时，情况发生了变化。如果你遵循标准插入算法，插入 Li 元素后树的布局如下图所示：



709

此时，根节点不再平衡，因为它的右子树高度为 1，左子树高度为 -1（由定义知），两者之差大于 1。

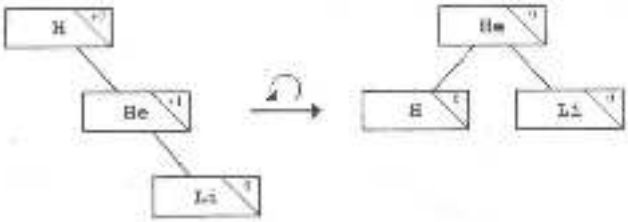
为了消除不平衡，你需要重构这棵树。对于此给定的节点集，只有一种平衡状态使得节点具有正确的相对顺序。该树以 He 为根，H 和 Li 分别是它的左右子树，如下图所示：



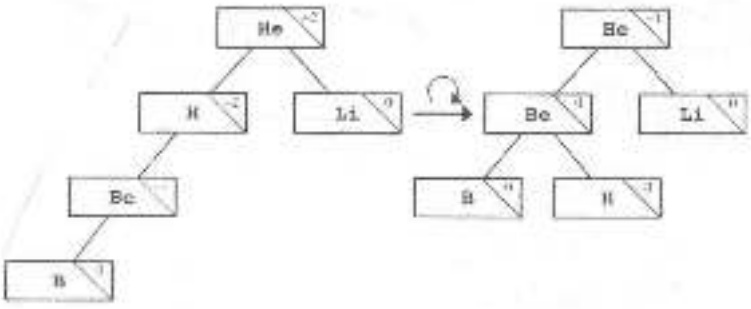
该树每一次达到平衡，但是还有一个重要的问题：如何知道为了重新得到一棵平衡树要执行那些操作呢？

16.3.3 单旋转

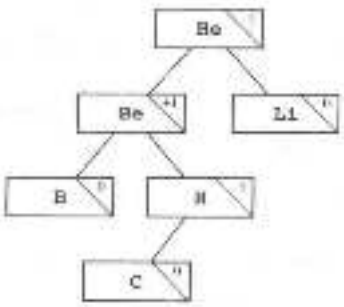
AVL 策略的基本思想是：你总是能够通过一次简单的节点互换来使树达到平衡。想象一下破除此树中不平衡所需的步骤。显然，将 He 节点向上移使其成为树根，而将 H 向下移使其成为 He 的孩子。在某种程度上，上述转换具有将节点 H 和 He 向左旋转的特点，如下图所示：



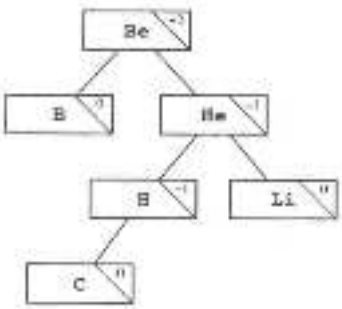
在旋转操作中涉及的两个节点被称为一次旋转的轴（axis）。在包含 H、Be 和 Li 三个元素的示例中，旋转是绕着 H-Be 轴进行的。由于该操作向左移动节点，因此图中所示的操作被称为**左旋转**（left rotation）。如果树在相反的方向上失去平衡，那么可以使用对称的**右旋转**（right rotation）操作，它只是简单地将左旋转进行反向操作。例如，接下来的两个元素符号 Be 和 B，每一个都加在树的左边。为了使树重新平衡，需要绕着 Be-H 轴执行一次右旋转，如下图所示：



遗憾的是，简单地旋转操作并不总能保证树平衡。例如，考虑将 C 加入树中会出现什么情况。在执行任何平衡操作之前，树的状态如下图所示：



根节点 He 是不平衡的。如果你尝试通过绕 Be-He 轴向右旋转树来获得平衡，那么将得到如下图所示的树状态：



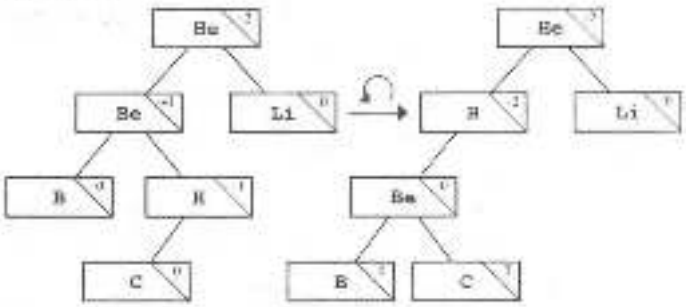


711 旋转之后，树依然不平衡。它和上图唯一的不同在于：此时，根节点在其右子树上不平衡。

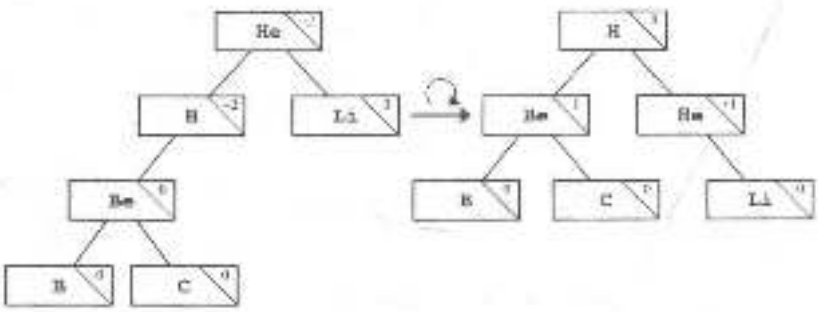
16.3.4 双旋转

在上述最后的示例中，出现问题的原因是：涉及旋转的节点的平衡因子具有相反的符号。此时，仅单向旋转是不够的。为了解决该问题，你需要进行两次旋转。在旋转不平衡节点之前，先将其孩子节点以相反的方向进行旋转，使得父节点与孩子节点的平衡因子具有相同符号，这也意味着第二次旋转将会成功。这样一对旋转操作被称为**双旋转**（double rotation）。

为了说明双旋转操作，让我们考虑之前加入 C 元素后的不平衡树。第一步是绕 Be-H 轴向左旋转树，如下图所示：



虽然 H 和 Be 节点的平衡因子同号，但是得到的树的根节点仍然是不平衡的。此时，绕 H-He 轴向右旋转使树重新平衡，如下图所示：



712 在安德森·维斯基和兰蒂斯的论文中，阐述了它们的树平衡算法具有如下特性：

- 如果向 AVL 树中插入一个新节点，总可以通过至多一次操作来重新获得平衡，该操作要么是一个单旋转，要么是一个双旋转。
- 在完成旋转操作之后，旋转轴子树的高度总是和在插入新节点前一致。该特性保证了不会改变树中更高层次节点的平衡因子。

16.3.5 实现 AVL 算法

尽管 AVL 树在实现 InsertNode 时涉及一些细节，但是并没有你想的那么困难。首先需要改变的是，在节点结构中应包括一个新的域来跟踪平衡因子，如下所示：

```
struct BSTNode {
   string key;
   BSTNode *left, *right;
```

```
   int bf;
};
```

insertNode 函数的代码如图 16-6 所示。从这段代码中可以看到，insertNode 被实现为函数 insertAVL 的一个包装器，乍看起来这两个函数原型貌似相同。事实上，这两个函数的参数确实相同。唯一的不同是函数 insertAVL 返回一个表示插入新节点后树高度变化的整数值。该返回值总是 0 或者 1，使得当前代码沿递归调用的层次修复树的结构变得容易。递归的简单情况如下：

1. 在空树中加入一个节点，使其树高增加 1。
2. 若遇到已存在的键值节点，则树高不变。

在递归情况下，图 16-6 中的代码首先将新节点插入到树中适当的子树中，用局部变量 delta 追踪其高度的变化。如果插入节点未改变子树的高度，则当前节点的平衡因子必须保持不变。然而，如果插入节点使子树的高度增加，则有以下三种可能性：

1. 被插入新节点的子树应比另一个子树矮。此时，插入一个新节点实际上使得树比之前更加平衡。当前节点的平衡因子变为 0，该节点的子树的高度不变。 713
2. 当前节点的两棵子树的规模相同。此时，增大其中一棵树会使当前节点稍微不平衡，但是并不需要矫正措施。视不同的情况，其节点平衡系数变为 -1 或者 +1，函数返回 1 则表示该节点的子树高度已增加。
3. 较高的子树比另一棵子树更高。此时，由于一棵子树比另一棵子树的高度大 2，因此，树不再平衡。这时，程序代码必须进行合适的旋转操作以纠正其不平衡。如果当前节点的平衡因子和扩展的子树的平衡因子的符号相同，那么仅需单旋转操作。如果平衡因子的符号不同，则必须进行双旋转操作。执行完旋转操作后，程序代码必须修改位置已改变的节点的平衡因子值。节点平衡因子的单旋转与双旋转操作如图 16-7 所示。

```
/*
 * Function: InsertNode
 * Usage: insertNode(t, key);
 * ---------------------------
 * Inserts a node with the specified key into the correct position in the
 * binary search tree.  If key already exists in the tree, this call has
 * no effect.
 */

void insertNode(BSTNode * & t, const string & key) {
   insertAVL(t, key);
}

/*
 * Function: insertAVL
 * Usage: delta = insertAVL(t, key);
 * ---------------------------------
 * Enters the key into the tree that is passed by reference as the first
 * argument.  The return value is the change in depth in the tree, which
 * is used to correct the balance factors in ancestor nodes.
 */

int insertAVL(BSTNode * & t, const string & key) {
   if (t == NULL) {
      t = new BSTNode;
      t->key = key;
      t->bf = 0;
      t->left = t->right = NULL;
```

图 16-6　在 AVL 树中插入一个节点的代码

```
      return +1;
   }
   if (key == t->key) return 0;
   if (key < t->key) {
      int delta = insertAVL(t->left, key);
      if (delta == 0) return 0;
      switch (t->bf) {
       case +1: t->bf =  0; return 0;
       case  0: t->bf = -1; return +1;
       case -1: fixLeftImbalance(t); return 0;
      }
   } else {
      int delta = insertAVL(t->right, key);
      if (delta == 0) return 0;
      switch (t->bf) {
       case -1: t->bf =  0; return 0;
       case  0: t->bf = +1; return +1;
       case +1: fixRightImbalance(t); return 0;
      }
   }
}

/*
 * Function: fixLeftImbalance
 * Usage: fixLeftImbalance(t);
 * ---------------------------
 * This function is called when a node has been found that is out of
 * balance with the longer subtree on the left.  Depending on the balance
 * factor of the left child, the code performs a single or double rotation.
 */

void fixLeftImbalance(ASTNode * & t) {
   ASTNode *child = t->left;
   if (child->bf != t->bf) {
      int oldBF = child->right->bf;
      rotateLeft(t->left);
      rotateRight(t);
      t->bf = 0;
      switch (oldBF) {
       case -1: t->left->bf = 0; t->right->bf = +1; break;
       case  0: t->left->bf = t->right->bf = 0; break;
       case +1: t->left->bf = -1; t->right->bf = 0; break;
      }
   } else {
      rotateRight(t);
      t->right->bf = t->bf = 0;
   }
}

/*
 * Function: rotateLeft
 * Usage: rotateLeft(t);
 * ---------------------
 * Performs a single left rotation of the tree passed by reference as the
 * argument t.  The balance factors are unchanged by this function and must
 * be corrected at a higher level of the algorithm.
 */

void rotateLeft(ASTNode * & t) {
   ASTNode *child = t->right;
   if (DEBUG) {
      cout << "rotateLeft[" << t->key << "-" << child->key << "]" << endl;
   }
   t->right = child->left;
   child->left = t;
   t = child;
}

/*
 * Function: fixRightImbalance
 * Usage: fixRightImbalance(t);
 * ----------------------------
 * This function is called when a node has been found that is out of
 * balance with the longer subtree on the right.  Depending on the balance
```

图16-6 (续)

```
 * factor of the right child, the node performs a single or double rotation.
 */

void fixRightImbalance(BSTNode * & t) {
   BSTNode *child = t->right;
   if (child->bf != t->bf) {
      int oldBF = child->left->bf;
      rotateRight(t->right);
      rotateLeft(t);
      t->bf = 0;
      switch (oldBF) {
       case -1: t->left->bf = 0; t->right->bf = +1; break;
       case  0: t->left->bf = t->right->bf = 0; break;
       case +1: t->left->bf = -1; t->right->bf = 0; break;
      }
   } else {
      rotateLeft(t);
      t->left->bf = t->bf = 0;
   }
}

/*
 * Function: rotateRight
 * Usage: rotateRight(t);
 * ----------------------
 * Performs a single right rotation of the tree passed by reference as the
 * argument t.  The balance factors are unchanged by this function and must
 * be corrected at a higher level of the algorithm.
 */

void rotateRight(BSTNode * & t) {
   BSTNode *child = t->left;
   if (DEBUG) {
      cout << "rotateRight(" << t->key << "-" << child->key << ")" << endl;
   }
   t->left = child->right;
   child->right = t;
   t = child;
}
```

图 16-6 （续）

使用图 16-6 的 AVL 算法代码保证了在新节点插入后仍能保持树的平衡。这样，findNode 与 insertNode 函数就都可以以 $O(\log N)$ 的时间执行。然而，即使不对 AVL 策略进行扩展，上述代码也能正常运行。采用 AVL 策略的优点就是它以代码的一些复杂性为代价保证了其 $O(\log N)$ 的时间性能。

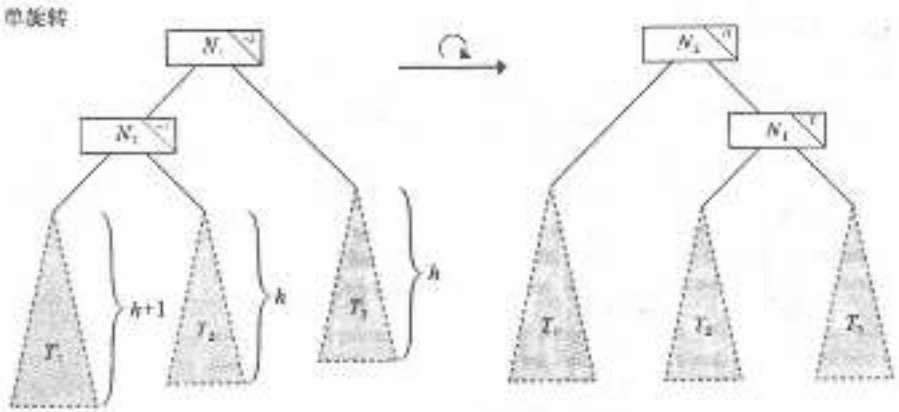


图 16-7 在 AVL 树中的旋转操作

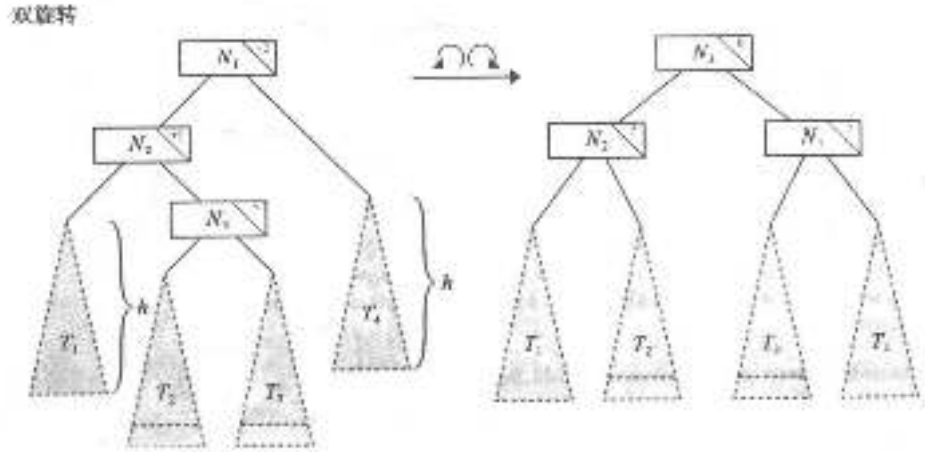


图 16-7 （续）

注：至少子树 T_2 和 T_3 中的一个树高为 h，其他子树的树高为 h 或 $h-1$。树中最终节点的平衡因子需要调整，以考虑其高度之差。

16.4 使用 BST 实现映射

在第 15 章中曾指出：标准模板库采用二叉搜索树来实现映射的抽象。该实现策略意味着 get 和 put 方法的运行时间为 O (log N)，它比哈希表策略的平均运行时间 O (1) 效率稍微低一些。在实际应用中，两者的时间性能差别并不重要。O (log N) 的时间性能图增长得十分缓慢，而且与时间性能为 O (N) 相比，它与 O (1) 的时间性能图更接近。C++ 的设计者们认为有序地处理键值的能力足以补偿一些额外的时间开销。

使用二叉搜索树实现映射抽象的困难之处在于几乎整个二叉搜索树本身的实现代码，这一点你已看到了。为了将此思想应用到 Map 类中，还有以下任务有待完成：

- 节点结构必须包含一个与键对应的值域。
- 代码必须使用模板来参数化不同类型的键和值。
- 操作树的代码必须嵌入 Map 类中。

713
:
718

上述每个改变都是一个很好的练习，它会巩固并加强你对于类的理解。

16.5 偏序数

树出现在很多应用程序中。特别有用的一个应用是优先级队列（priority queue），其中，元素出队的顺序取决于其优先级。Stanford C++ 类库通过 pqueue.h 接口实现了这一概念，该接口导出了一个 PriorityQueue 类。除 enqueue 方法外，PriorityQueue 类与标准类库中的 Queue 类的操作相同，enqueue 方法以其第二个参数指定其优先级，其原型如下所示：

```
void enqueue(ValueType element, double priority);
```

按照传统英语习惯，低优先级的元素先入队，因此，在队列中优先级为 1 的元素排在优先级为 2 的元素之前。

在本书第 14 章习题 8 中曾提及过一次优先级队列。如果你使用了该习题所建议的策略，则 enqueue 方法需要 O (N) 时间。你可以使用称为偏序数（partially ordered tree）的数据结构来改善优先级队列的时间性能至 O (log N)。偏序数满足以下特性：

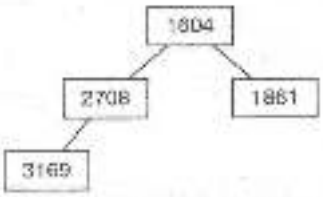
1. 该树是一棵二叉树，在树中每个节点最多有两个孩子节点。然而，该树不是一棵二叉

搜索树，二叉搜索树与偏序数有着不同的排序规则。

2. 树中的节点尽量被分配成对称的。因此，堆树的任何路径上的节点数之间相差不超过1。而且，叶子节点必须严格从左到右填充。

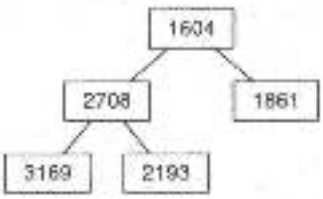
3. 树中每个节点的键值都小于等于它孩子节点的键值。因此，树中最小的键值往往都在根节点。

例如，下图展示了有四个节点的偏序数，每个节点都包含了一个数字键值：

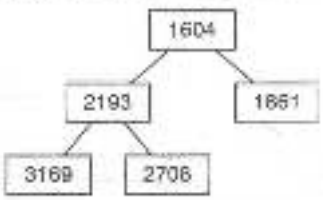


树的第二层被完全填充，第三层正处于从左向右填充的过程，满足偏序数的第二个特性。第三个特性也满足，因为树中任意节点的键值都小于其孩子节点的键值。 719

假如你想添加一个键值为2193的节点。很显然能看出该节点应添加到哪里。要求树的最底层从左至右进行填充意味着新的节点应添加到以下位置：

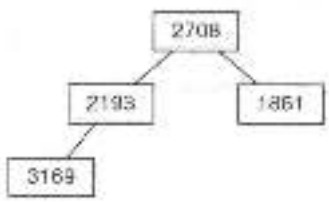


然而，该图违反了偏序数的第三个特性，因为键值2193小于它的父节点的键值2708。为了解决该问题，应交换这两个节点的键值，如下图所示：

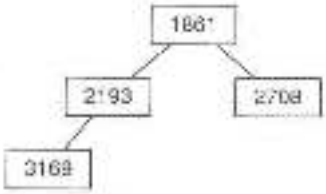


通常，新插入的节点都不得不通过一连串向上的移动来和它的父节点进行交换。在该实例中，交换过程终止在2193，因为它已比其父节点1604大。在任何情况下，树的结构保证了这些交换的时间复杂度永远不会超过 $O(\log N)$。

偏序数的结构意味着树中最小的键值往往都处于根节点处。然而，删除根节点需要花费些功夫，因为你必须重新排列那些不是树的最底层的最右边的所有节点。标准的做法是：用要删除的节点的键值来替换根节点，然后顺着树向下交换键值直到满足有序特性为止。例如，如果你想删除上图中树的根节点，第一步是用树中最底层最右边的节点2708来替换根节点，如下图所示： 720



然后，由于树的节点顺序并不符合要求，你需要用2708同个孩子节点中较小的那个替换2708，如下图所示：

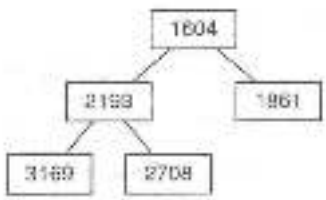


尽管在本例中，一次简单的交换足以使得树保持有序性，但通常情况下，寻找移动到根节点元素的正确位置可能需要经过树的每一层来进行元素交换。和插入一样，删除最小的键值需要的时间复杂度为 $O(\log N)$。

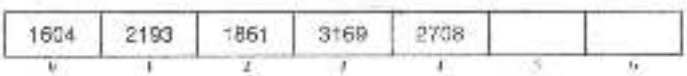
定义偏序数的操作正好是你实现优先级队列所需要的操作。enqueue 操作包括向偏序数插入一个新节点。dequeue 操作则是删除树中的最小值。因此，如果你将偏序数作为底层实现，你就可以以 $O(\log N)$ 的时间复杂度来实现优先级队列。

尽管你可以使用基于指针的结构来实现偏序数，但是优先级队列的大多数实现采用的是一个基于数组的结构，该结构被称为堆（heap），它用来模拟偏序数的操作。（堆这个术语听起来令人困惑，因为堆数据结构和动态分配内存时的可用内存池没多大关系，两者同名但不同义。）堆的实现策略取决于这一特性：你可以在一个数组的前 N 个元素中，一层接一层地
721 从左向右存储容量为 N 的偏序数的节点。

例如，下面的偏序数：



可以用以下的堆表示：



堆的组织结构使得实现树的操作变得简单，因为父节点和孩子节点都处于易于计算的位置。例如，给定索引位置为 n 的节点，你可以使用以下表达式来找出其父节点和孩子节点的索引：

```
parentIndex(n)          节点 (n - 1) / 2 给定
leftChildIndex(n)       节点 2 * n + 1 给定
rightChildIndex(n)      节点 2 * n + 2 给定
```

parentIndex 的除法运算采用 C++ 标准整数除法运算。因此，索引位置为 4 的节点的父节点计算结果显示为数组中索引为 1 的节点，因为 (4-1)/2 的运算结果为 1。

实现基于堆的优先级队列是一个很好的练习，它会提高你的编程技巧并且给你更多的关于如何使用本文中所见到的数据结构的体验。在习题 13 中你将会得到这样一个机会。

本章小结

在本章中，你接触到了树的概念，它是节点的层次化的集合，满足如下特性：

- 在树的顶部有一个唯一节点构成了其层次化的根。
- 树中的每一个节点都有一条唯一的通向根节点的路径。

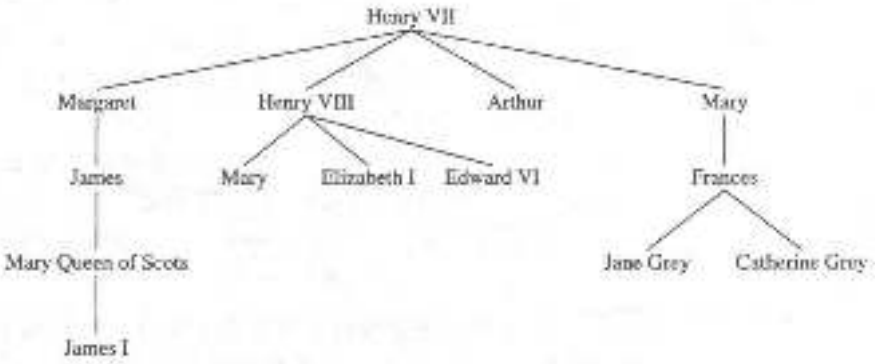
本章的重点内容包括：

- 许多用来描述树的术语，例如“父节点”“孩子节点”“祖先节点”“子孙节点”“兄弟节点”，都直接来自于家谱树。另外一些术语，包括“根节点”和“叶子节点”，则直接来源于自然界中的树。这些比喻使得用于树的专业术语易于理解，因为这些词语在计算机科学中的解释与通常的解释相同。
- 树具有一种良好定义的递归结构，因为树中的每个节点都是其子树的根节点。因此，一棵树是由一个节点以及其孩子节点的集合构成，树中又有树。这种递归结构反映在树的底层表示上，该结构被定义为指向一个节点的指针；一个节点继而成为包含树的结构。
- 二叉树是树的一个子类，其中每个节点最多拥有两个孩子节点，并且除了根节点之外，每个节点要么是其父节点的左孩子节点，要么是右孩子节点。
- 如果一个二叉树中的每一个节点都包含一个键值域，其键值总是小于其右子树的所有键值，大于其左子树的所有键值，那么这个二叉树被称为二叉搜索树。正如其名字所暗示的，二叉搜索树的结构采用了二分查找算法，这使得寻找单个键值变得更加高效。因为在树中键值是有序的，因此，往往能够确定你所搜索的键值是在某个节点的左子树上还是在右子树上。
- 使用递归使得访问二叉搜索树中的节点更加容易，这种操作被称为树的遍历（traversing 或 walking）。有几种类型的遍历，遍历类型取决于处理节点的顺序。如果每个节点的键值在递归调用处理其子树前被处理，那么它就称为先序遍历。如果对于每个节点的处理是在处理其左右子树的两个递归调用之后，就称为后序遍历。而在处理某个节点的左右子树的两个递归调用之间来处理当前节点，则称为中序遍历。在一棵二叉搜索树中，中序遍历由于其键值按顺序处理这一特性而更为有用。
- 给定相同的节点集，根据不同的顺序插入节点，二叉搜索树的结构也会完全不同。如果树的分支在高度上有很大不同，则该树是不平衡的，这也会降低其效率。使用AVL算法，可以在加入新节点的同时维持树的平衡。
- 使用堆数据结构可以高效实现优先级队列，该数据结构基于二叉树的一个特殊的子类——偏序数。如果使用堆表示，enqueue 和 dequeue 操作的时间复杂度均为 $O(\log N)$。

复习题

1. 一个节点集要组成树，必须满足的两个条件是什么？
2. 给出四个现实世界有关树结构的例子。
3. 定义用于树的这些术语：父节点、孩子节点、祖先节点、子孙节点和兄弟节点。
4. 莎士比亚时代统治英国都铎家族的家谱树如图 16-8 所示。找出根节点、叶子节点和内部节点。该树的树高是多少？
5. 有关树的什么特征使得树是递归的？

6. 画出将图 16-8 中的树表示为一个 FamilyTreeNode 时的内部结构。
7. 二叉搜索树的定义特性是什么？
8. 为什么 findNode 和 insertNode 中的第一个参数使用不同类型声明？
9. 在英国作家托尔金的小说《霍比特人》中，13 个小矮人以如下的顺序来到比尔博·巴金斯的房子：Dwalin、Balin、Kili、Fili、Dori、Nori、Ori、Oin、Gloin、Bifur、Bofur、Bombur 和 Thorin。画出将这些小矮人的名字插入到一棵空树中所产生的二叉搜索树。



724

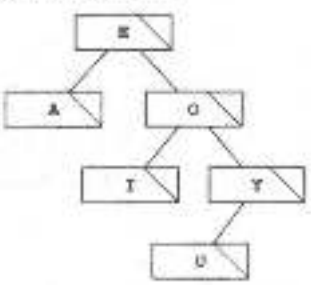
图 16-8 都铎家族树

10. 给定你在第 9 题中创建的树，回答问题：如果你在 Bombur 节点上调用 findNode，需要与哪些键值进行比较？
11. 分别写出第 9 题中你所创建的树的前序遍历、中序遍历和后序遍历。
12. 在树的三种遍历方式中，有一种是不依赖于树中插入节点的顺序的，请问这是哪一种遍历方式？
13. 一棵二叉树是平衡的，其含义是什么？
14. 对于以下每一种树的结构，判断其是否平衡？



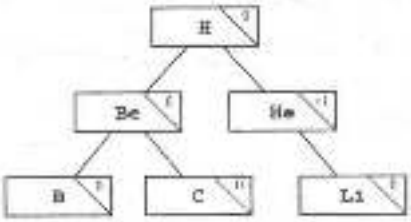
对于不平衡的树，指出哪个节点不平衡。

15. 判断题：如果一棵二叉搜索树不平衡，那么函数 findNode 和 insertNode 的算法将不能正常运行。
16. 如何计算一个节点的平衡因子？
17. 在下面的二叉搜索树中，填充各节点的平衡因子。



725

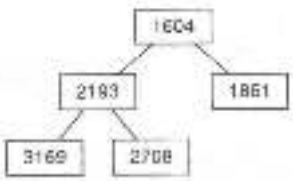
18. 如果你使用 AVL 平衡策略，对于前一题中的树需要什么旋转操作来使它重新平衡？包含更新的平衡因子的结果树的结构是什么？
19. 判断题："如你向一棵平衡二叉树插入新节点时，你总能通过执行一次操作矫正带来的不平衡，该操作要么是一次单旋转，要么是一次双旋转。
20. 正如在 16.3.2 一节中所展示的那样，向一个 AVL 树插入前六个化学元素符号得到如下布局：



"继续插入以下六个元素符号时，树的状态会发生什么变化：

N （氮）
O （氧）
F （氟）
Ne （氖）
Na （钠）
Mg （镁）

21. 详细描述 `insertNode` 的调用过程。
22. 为了避免在删除二叉搜索树中的一个中间节点时树不相连的问题，本章给出了什么策略？
23. 假如你在处理一棵偏序数，该树包含如下数据：



726

给出插入键值为 1521 的节点后的树状态。

24. 堆和偏序数之间的关系是什么？

习题

1. 在 16.1.3 一节中，给出了 `FamilyTreeNode` 的定义，试编写一个函数：

```
FamilyTreeNode *readFamilyTree(string filename);
```

该函数从一个数据文件中读取一棵家谱树，数据文件名作为参数传递给函数。文件的第一行是树的根节点名。数据文件接下来的所有行按以下格式排列：

孩子节点:父节点

其中，孩子节点是一个新输入的节点名，父节点则是孩子节点的父节点名，父节点名一定在数据文件的前面。例如，如果文件 `Normandy.txt` 如下：

`Normandy.txt`

```
William I
Robert:William I
William II:William I
Adela:William I
Henry I:William I
Stephen:Adela
William:Henry I
Matilda:Henry I
Henry II:Matilda
```

调用 readFamilyTree("Normandy.txt") 将返回图 16-2 所示的结构。

2. 编写函数：

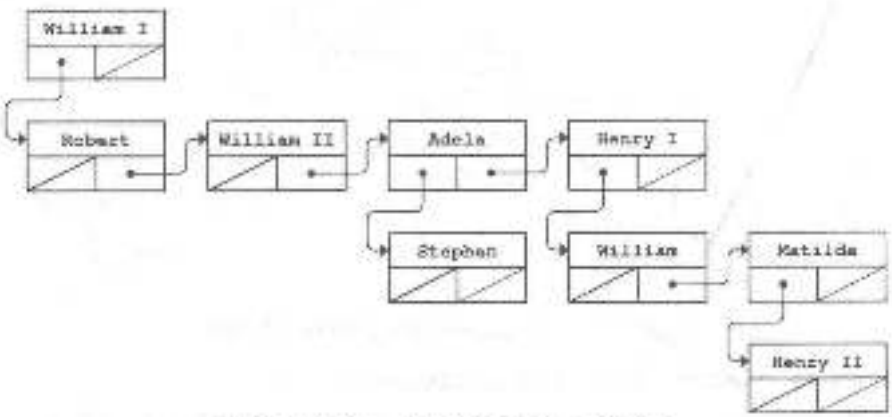
```
void displayFamilyTree(FamilyTreeNode *tree);
```

该函数显示家谱树中所有的个体。为了记录树的层次结构，程序的输出应该将每一代分开，使得每
727 个孩子节点名相对其父节点名向右缩进两个空格，运行示例如下图所示：



3. 正如本章中定义的，FamilyTreeNode 结构使用一个矢量来存储孩子节点。另一种方案是在这些节点中包含一个指针来构建一个孩子节点的链表。在这个设计中，每一个节点需要两个指针：一个指向最年长的孩子节点；一个指向它的下一个年轻的兄弟节点。使用这种表示，诺曼底家族的家谱树如图 16-9 所示。在每个节点上，左指针总是指向一个孩子节点；右指针则指向同代的兄弟节点。因此，William I 最年长的孩子节点是 Robert，可以顺着图的左边找到。剩下的孩子节点都通过一个链连接在一起。孩子节点以 Henry I 终止，该节点的兄弟节点指针为空。

使用下图所示的链表策略，写出 FamilyTreeNode、readFamilyTree 和 displayFamilyTree 的定义。



728 图 16-9 使用一个兄弟列表的诺曼底家族

4. 在习题 3 中，改变 FamilyTreeNode 结构使得你必须重写函数 readFamilyTree 和 displayFamilyTree，因为这两个函数依赖于其内部实现。如果用一个类来实现就会维护接口不受变化影响，也会避免过多的重复编码工作。图 16-10 给出了这样一个接口。编写该接口相应的实现代码，使用一个矢量来存储孩子节点。

```
/*
 * File: familytree.h
 * ------------------
 * This file is an interface to a simple class that represents an individual
 * person in a family tree.
 */
```

图 16-10 FamilyTreeNode 类的接口

```
#ifndef _familytree_h
#define _familytree_h

#include <string>
#include "vector.h"

/*
 * Class: FamilyTreeNode
 * ---------------------
 * This class defines the structure of an individual in the family
 * tree, which consists of a name and a vector of children.
 */

class FamilyTreeNode {

public:

/*
 * Constructor: FamilyTreeNode
 * Usage: FamilyTreeNode *person = new FamilyTreeNode(name);
 * ---------------------------------------------------------
 * Constructs a new FamilyTreeNode with the specified name.  The
 * newly constructed entry has no children, but clients can add
 * children by calling the addChild method.
 */

   FamilyTreeNode(const std::string & name);

/*
 * Method: getName
 * Usage: string name = person->getName();
 * ---------------------------------------
 * Returns the name of the person.
 */

   string getName() const;

/*
 * Method: addChild
 * Usage: person->addChild(child);
 * -------------------------------
 * Adds child to the end of the list of children for person, and
 * makes person the parent of child.
 */

   void addChild(FamilyTreeNode *child);

/*
 * Method: getParent
 * Usage: FamilyTreeNode *parent = person->getParent();
 * ----------------------------------------------------
 * Returns the parent of the specified person.
 */

   FamilyTreeNode *getParent() const;

/*
 * Method: getChildren
 * Usage: Vector<FamilyTreeNode *> children = person->getChildren();
 * -----------------------------------------------------------------
 * Returns a vector of the children of the specified person.
 * Note that this vector is a copy of the one in the node, so
 * that the client cannot change the tree by adding or removing
 * children from this vector.
 */

   Vector<FamilyTreeNode *> getChildren() const;
```

类的私有部分在这。

```
};

#endif
```

图 16-10 （续）

5. 使用图 16-10 所定义的 familytree.h 接口，编写一个函数：

```
FamilyTreeNode *commonAncestor(FamilyTreeNode *p1,
                               FamilyTreeNode *p2);
```

它返回 p1 和 p2 共同的最近的祖先节点。

6. 使用 16.2 节中 BSTNode 的定义，编写一个函数：

729
?
730

```
int height(BSTNode *tree);
```

该函数取一棵二叉搜索树为参数，并返回其高度。

7. 编写一个函数：

```
bool isBalanced(BSTNode *tree);
```

它根据平衡树的定义判断树是否平衡。为了解决该问题，你所要做的就是将平衡树的定义直接翻译成代码。然而，这样做最后的实现可能相当低效，因为这需要在树中遍历好几遍。该问题的真正难点在于：在对节点进行不多于一次检查的情况下，isBalanced 函数的实现就能判断出结果。

8. 编写一个函数：

```
bool hasBinarySearchProperty(BSTNode *tree);
```

该函数接受一棵树并判断它是否具有二叉搜索树的基本特性：每个节点的键值必须大于其左子树所有节点的键值，而小于其右子树所有节点的键值。

9. 本书对于 AVL 算法的讨论为插入节点提供了一个策略，但并不适用于具有对称过程的删除操作。该操作同样要求能树保持平衡。其实，这两个算法特别相近。删除节点有可能对于树的高度没什么影响，但也有可能使其高度减 1。如果一棵树变矮了，那么它的父节点平衡因子也会发生改变。如果父节点不再平衡，可以通过单旋转或双旋转来使其恢复平衡。

实现下面这个函数：

```
void removeNode(BSTNode * & t, const string & key);
```

该函数在保持底层的 AVL 树平衡的条件下，从树中删除包含键值为 key 的节点。思考一下会出现的各种问题，并且确保你的实现可以正确处理这些不同情况。

10. 以 16.4 小节的讨论为指导，使用二叉搜索树作为底层实现来实现 map.h 接口。以图 15-1 展示的简单版的 StringMap 作为开始。一旦你开始进行，就去实现那些遗漏的操作。

11. 在第 5 章习题 19 中，你有机会编写一个程序，将摩尔斯码信息翻译成等价的英语字母。该练习题鼓励你使用一个映射来存储翻译表，但是也有其他方法。例如，你可以将摩尔斯码想象成一棵二叉
731 树，点代表向左，破折号代表向右。用这种形式，得到的摩尔斯码表的结构如图 16-11 所示。例如，你可以从根节点开始，以左－右－左－左的顺序沿着箭向下找到字母 L，L 的摩尔斯码就是·－··。

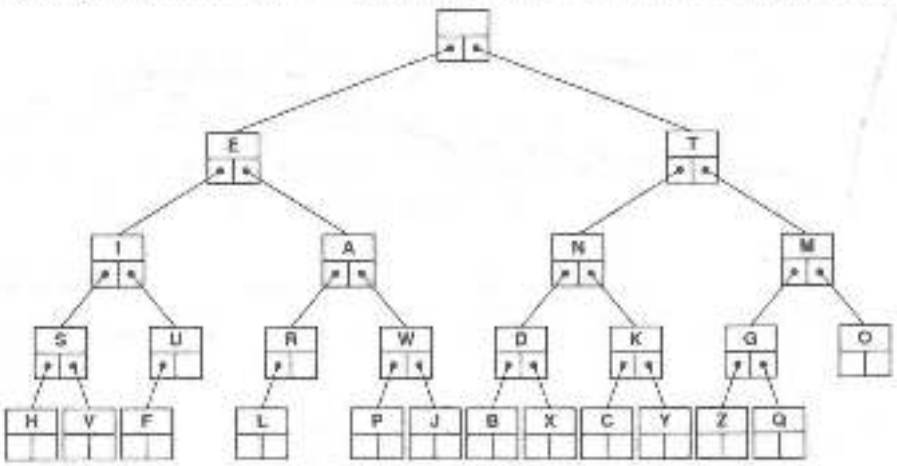


图 16-11 摩尔斯码树

设计一个数据结构存储图 16-11 的树，之后编写函数 getMorseCodeLetter(code)，该函数查找与字符串参数 code 代表的摩尔斯码对应的字母。

12. 从实际观点看，AVL 算法过于严苛。因为它要求每个节点的子树高度差永远不能超过 1，在新节点插入时，AVL 算法花费了一点时间执行旋转操作来保持平衡。如果你允许树有些不平衡，但是仍旧使其左右子树尽可能相似，那么你可以在很大意义上减少一部分操作开销。

二叉搜索树的一种数据结构提供了更好的性能，该结构被称为红黑树（red-black tree）。树名来源于树中每个节点都被染色，要么是红色，要么是黑色。如果一棵二叉树全部满足以下三个特性，它就是一棵红黑树：

1. 根节点是黑色的。
2. 每一个节点的父节点是黑色的。
3. 从根节点到叶子节点的所有路径上，黑色节点的数目相等。 732

这几个特性保证了从根节点到叶子节点的最长路径不会比最短路径长两倍以上。从给定的这些规则中，你已经知道每一条路径都含有相同数目的黑色节点，这也就意味着最短的路径可能全都由黑色节点组成，而最长的路径则由黑色节点和红色节点交替出现组成。尽管这些条件相比于使用 AVL 算法的平衡树的定义而言并不那么苛刻，但是也足以保证在 $O(\log N)$ 的时间复杂度级别上进行寻找和插入节点操作。

使红黑树得以正确工作的关键是找到一个插入算法，使得在保持定义红黑树的前提下加入新的节点。该算法和 AVL 算法有很多相同之处。并使用同样的旋转操作。第一步是使用不带平衡功能的标准插入操作插入新节点。新节点总是替代树中某个位置的 NULL 项。如果该节点是插入树中的第一个节点，它就成为树根而被赋予黑色。在其他所有情况下，新节点必须初始化为红色，以避免违反从根节点到叶子节点的每一条路径上都包含相同数量黑色节点的规则。

只要新节点的父节点是黑色的，那么该树仍旧是一颗合法的红黑树。如果父节点也是红色的，就会出现问题，这意味着该树违反了第二条规则，要求每个红色节点的双亲为黑色节点。在这种情况下，需要对树重新进行构造以满足红黑树的条件。根据红－红对和树中其他节点的关系，可以通过执行下列其中一个操作来解决这个问题：

1. 一次单旋转，同时进行一个重新着色，让顶部节点为黑色。
2. 一次双旋转，同时进行一个重新着色，让顶部节点为黑色。
3. 颜色的简单变化，让顶部节点为红色，接着在更高的层次上对树进行进一步重新构造。

这三个操作在图 16-12 中进行了说明。图中只显示了不平衡发生在左边的情况。发生在右边的情况可以对称地进行处理。

重新实现 Map 类，用红黑树作为其底层表示。当你调试你的程序时，会发现实现一个函数来展示树结构并包含节点颜色（对用户不可见）是很有用的。该方法应该能在树发生变化时检查是否符合构建红黑树的规则。 733

13. 使用 16.5 一节中的算法来实现 PriorityQueue 类，用堆作为底层表示。为了消除一些复杂性，使用一个矢量要比一个动态数组更好。

14. 堆数据结构为排序算法提供了基础。它总是以 $O(N \log N)$ 时间复杂度运行。在**堆排序**（heapsort）算法中，你需要将每个值输入到堆中，并且按从最小到大的顺序取出这些项。使用该策略写出一个函数模板来实现堆排序：

```
template <typename ValueType>
void sort(Vector<ValueType> & vec);
```

15. 除了本章介绍的树的应用，还有很多其他树的应用。例如，可以用树实现一个字典，这在第 5 章曾介绍过。这种方式实现的结构，由爱德华·弗雷德在 1960 年首次提出，并被命名为**字典树**（trie）。基于字典树对字典的实现，尽管对空间的使用有些低效，但是可以比使用哈希表更快地

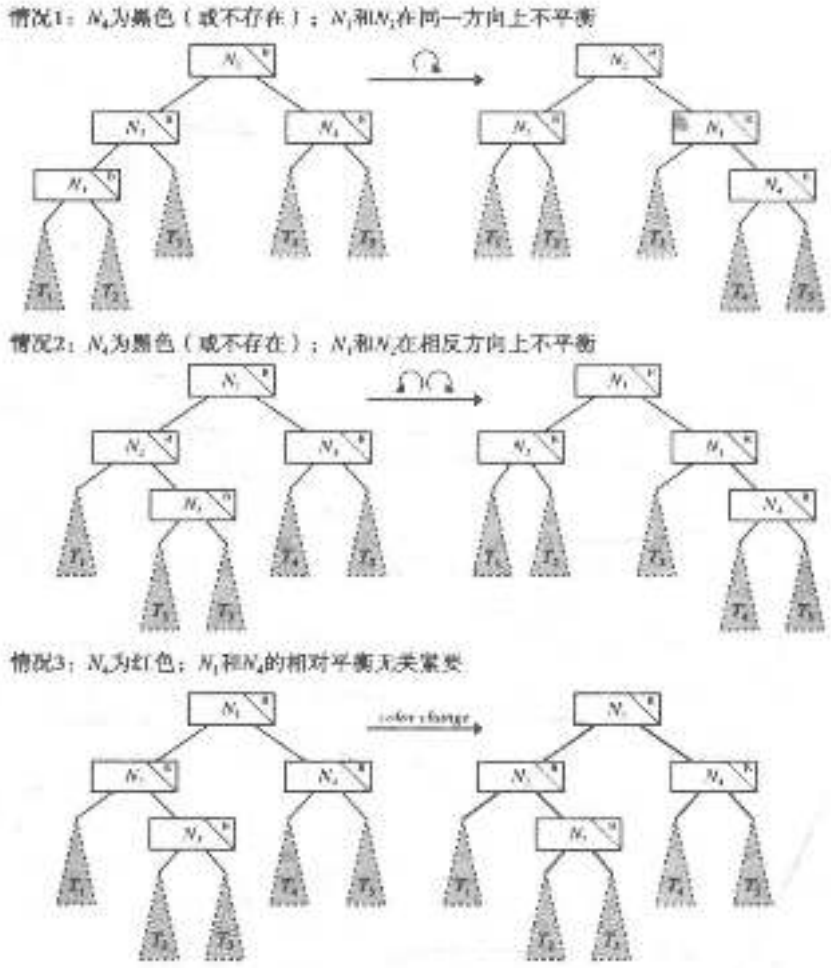


图 16-12 在一棵红黑树上的旋转操作

找到单词。

在字典树中的每一层，每一个节点都有 26 个分支，每个分支代表字母表中每个可能的字母。当使用字典树来表示字典时，单词被隐含地存储在树结构中，它们被表示为从根节点向下的一个序列。树根对应于空字符串，每一个随后的层次对应于在其父节点表示的字符串后加入了一个字母构成的单词的子集。例如，根节点下的 A 链接的子树包含一个所有以 A 开头的单词，该 A 节点下一层次的 B 链接的子树包含了所有以 AB 开头的单词，以此类推。每个节点有一个标记指明到该节点为止的子串是否是一个合理的单词。

举例理解字典树结构比用定义来理解要容易得多。图 16-13 展示了包含六个元素符号 H、He、Li、Be、B 和 C 的字典树。该树根对应于空字符串，就像该结构最右边域中的符号 no 所指明的那样：它是一个不合法的符号。从该字典树的根节点中的标记为 B 出发所下降的节点对应字符串 "B"。该节点最右边的域为 yes，表明 "B" 本身是一个合法的符号。从这个节点，标记为 E 的链接导致了一个新的节点，它代表字符串 "BE"，也是一个合法的符号。字典树中的 NULL 指针说明没有合法的符号出现在该子串开始的子树中，因此可以结束搜索过程了。

734 / 735

以字典树作为内部表示重新实现 Lexicon 类。该实现要能够读取文本文件，但不能读取类似 EnglishWords.dat 的二进制数据文件。

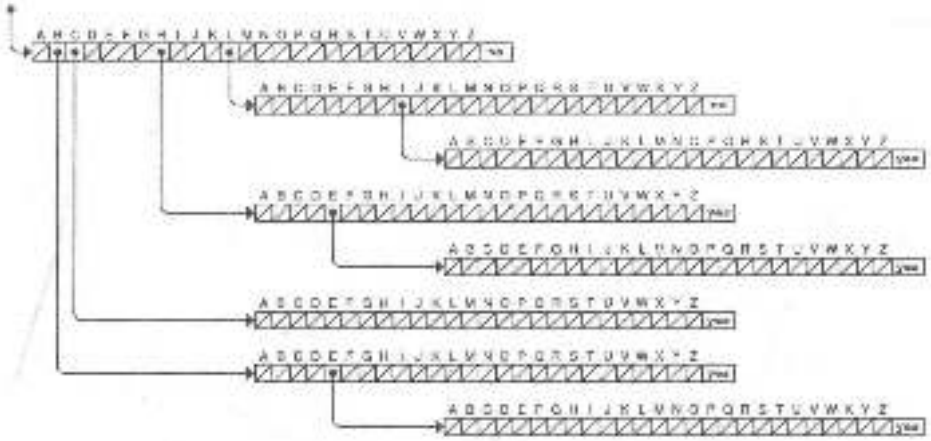


图 16-13 包含元素符号 H、He、Li、Be、B 和 C 的字典树 736

第 17 章
Programming Abstractions in C++

集　合

我们是一个雄心勃勃的集合，不是吗？

737

——路易莎·梅·奥尔科特，《小妇人》，1868

Set 和 HashSet 类在第 5 章都出现过。和本书剩余章节一样，本章的目标就是学习如何实现这些类。讨论这些类如何实现费不了多少口舌，参考 Map 和 HashMap 类的实现，相应的 Set 类是很容易实现的。因此本章将接受另一项挑战，使用一种理论上更精确的方法定义集合类 Set。集合对于计算机科学理论和实践这两方面都非常重要。理解这个原理能使你更容易在程序中有效使用集合。

17.1　集合作为一种数学抽象

在你学习数学时，肯定经常会遇到集合。尽管集合的定义并不完全准确，最好是将集合（set）看作是一个不同元素组成的无序集合。例如，一周的天数组成了一个具有七个元素的集合，它可以写成下面的形式：

{ Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }

上述集合中的元素以这种顺序书写只是因为这样比较符合习惯。如果你用别的顺序书写这些元素，仍会得到同样的集合。然而，一个集合绝不能包含相同的元素。

工作日集合是一个**有限集**（finite set），因为它包含有限个元素。在数学中，也存在**无限集**（infinite set），例如所有整数的集合。在一个计算机系统中，集合通常是有限的，即使它们对应数学中的无限集。例如，整数集合就是一个有限集，因为计算机可以用 int 类型的变量来表示集合中的元素，并且计算机硬件对整数范围施加了限制。

为了说明集合的基本操作，使用几个集合作为基础是很重要的。为了和数学惯例保持一致，本教材使用以下符号来代表所要表示的集合：

∅	**空集**（empty set），不包含任何元素
Z	所有整数的集合
N	**自然数**（natural number）集合，在计算机科学中通常被定义为：0, 1, 2, 3, …
R	所有实数的集合

遵循数学惯例，本书使用大写字母表示集合。集合的全体成员（如 **N**、**Z** 和 **R**）使用黑
738 体字母表示。某些未详细说明的集合名用斜体字母表示，例如，集合 S 和集合 T。

17.1.1　隶属关系

定义一个集合的基本属性是**隶属关系**（membership），它在数学和英语中有着相同直观的意义。数学家使用符号 $x \in S$ 来象征性地表示隶属关系，它表示 x 是集合 S 的一个元素。例如，根据上节中集合的定义，下面的语句是正确的：

$$17 \in \mathbf{N} \qquad -4 \in \mathbf{Z} \qquad \pi \in \mathbf{R}$$

相反，符号 $x \notin S$ 表示 x 不是集合 S 中的元素。例如，$-4 \notin \mathbf{N}$，因为自然数集合不包括负数。

一个集合的成员通常通过以下两种方法来指定：

- 枚举法。通过枚举定义一个集合，简单来说就是列出集合中的所有元素。按照惯例，在集合列表中的元素都放在大括号中并以逗号相隔。例如，单个数字的自然数集合可以使用枚举法定义如下：

$$\mathbf{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

- 规则法。你可以指定一个规则来定义一个集合，该规则可以区分集合中的成员。在大多数情况下，规则包含两部分含义：作为一个更大的集合提供潜在的候选元素；作为一些条件表达式，这些表达式要求选择的集合元素必须满足这些条件。例如，前面例子中的集合 D 可以用这种方法定义：

$$\mathbf{D} = \{x \mid x \in \mathbf{N} \text{ 且 } x < 10\}$$

如果你大声读出这个定义，结果听起来应该像这样："D 被定义成所有元素 x 的集合，条件是 x 为一个自然数并且 x 小于 10。"

17.1.2　集合运算

数学集合理论定义了集合中的几种运算，其中，下面是几种最重要的运算：

- 并（union）。两个集合的并记为 $A \cup B$，其结果为由所有属于 A 或 B 的元素或者同时属于 A 和 B 的元素构成的集合。

$$\{1, 3, 5, 7, 9\} \cup \{2, 4, 6, 8\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$\{1, 2, 4, 8\} \cup \{2, 3, 5, 7\} = \{1, 2, 3, 4, 5, 7, 8\}$$
$$\{2, 3\} \cup \{1, 2, 3, 4\} = \{1, 2, 3, 4\}$$

- 交（intersection）。两个集合的交记为 $A \cap B$，它包含同时属于 A 和 B 的元素。 739

$$\{1, 3, 5, 7, 9\} \cap \{2, 4, 6, 8\} = \varnothing$$
$$\{1, 2, 4, 8\} \cap \{2, 3, 5, 7\} = \{2\}$$
$$\{2, 3\} \cap \{1, 2, 3, 4\} = \{2, 3\}$$

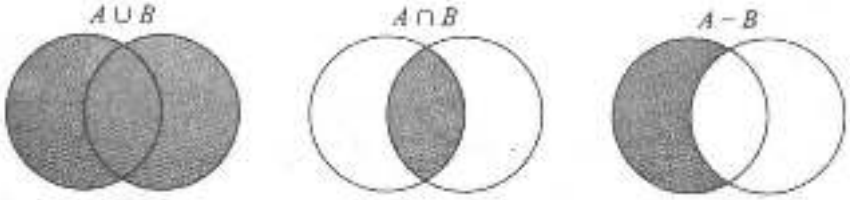
- 差（set difference）。两个集合的差记为 $A - B$，它包含属于 A 但不属于 B 的元素。

$$\{1, 3, 5, 7, 9\} - \{2, 4, 6, 8\} = \{1, 3, 5, 7, 9\}$$
$$\{1, 2, 4, 8\} - \{2, 3, 5, 7\} = \{1, 4, 8\}$$
$$\{2, 3\} - \{1, 2, 3, 4\} = \varnothing$$

除了像并和交这样的集合运算外，数学集合理论也定义了用于判定两个集合是否存在某种性质的操作。测试这种特定性质的操作往往采用数学等式的判定函数，它们通常被称为关系（relation）。集合中最重要的关系如下：

- 相等（equality）。如果集合 A 和 B 有相同的元素，则它们相等。集合的相等关系是用标准的等号表示的。因此，记号 $A = B$ 表示集合 A 和 B 包含相同的元素。
- 子集（subset）。子集关系写作 $A \subset B$，如果 A 中所有的元素都是 B 中的元素，则它是正确的。例如，集合 {2, 3, 5, 7} 是集合 {1, 2, 3, 4, 5, 6, 7, 8, 9} 的一个子集。类似地，自然数集合 N 是整数集合 Z 的子集。从定义可以得出：每个集合显然都是它自身的子集。数学家使用记号 $A \subset B$ 表示 A 是 B 的真子集（proper subset），它意味着子集关系存在，但两个集合不相等。

集合关系通常用韦恩图（Venn diagram）表示，它是以英国逻辑学家约翰·韦恩（1834~1923）的名字命名的。在一个韦恩图中，单个集合用几何图形表示，重叠的几何图形表示它们共同元素所在的区域。例如，集合运算并、交和差的结果用下面韦恩图中的阴影区域表示：



740

17.1.3 集合恒等式

从数学集合理论，你会了解到交、并和差操作在很多方面相互关联。这些关系通常表述为恒等式（identity），它是表示两个表达式恒等的规则。在本书中，恒等式采用以下书写符号：

$$lhs = rhs$$

它意味着根据定义，集合表达式 lhs 和 rhs 是相等的，因此，它们可以相互替换。最常见的集合恒等式列在表 17-1 中。

你可以意识到：当使用画韦恩图的方法表示计算的各个阶段时，这些恒等式是如何起作用的。例如，图 17-1 证明了表 17-1 中的德·摩根定律的第一部分，这个定律是以英国数学家奥古斯都·德·摩根的名字命名的，他第一个使这些性质形式化。阴影区域表示性质中每个子表达式的值。图 17-1 右侧的韦恩图有相同的阴影区域，这个事实证明集合 $A-(B\cup C)$ 和集合 $(A-B)\cap(A-C)$ 是一样的。

表 17-1 基本的集合恒等式

$S\cup S=S$ $S\cap S=S$	幂等性
$A\cup(A\cap B)=A$ $A\cap(A\cup B)=A$	吸收律
$A\cup B=B\cup A$ $A\cap B=B\cap A$	交换律
$A\cup(B\cup C)=(A\cup B)\cup C$ $A\cap(B\cap C)=(A\cap B)\cap C$	结合律
$A\cup(B\cap C)=(A\cup B)\cap(A\cup C)$ $A\cap(B\cup C)=(A\cap B)\cup(A\cap C)$	分配律
$A-(B\cap C)=(A-B)\cup(A-C)$ $A-(B\cup C)=(A-B)\cap(A-C)$	德摩根定律

可能你们不明白，作为一个程序员，为什么需要学习的规则一眼看上去是如此复杂和神秘。由于某些原因，数学知识对于计算机科学是很重要的。首先，理论知识自身确实是有用的，因为它能加深你对计算基础知识的理解。另外，这种理论知识通常已经直接应用到实际编程中了。

741 正是有了这些数学性质完善的数据结构，你才可以适当使用这些结构的理论基础。例如，如果你使用集合作为一种抽象类型编写一个程序，通过应用表 17-1 中的标准集合恒等

式，你可以基于集合抽象理论来简化代码。选择使用集合作为一种编程抽象，而不是设计一些属于你自己的、但不那么正式的结构，可使得你很容易将这些理论应用到实践。

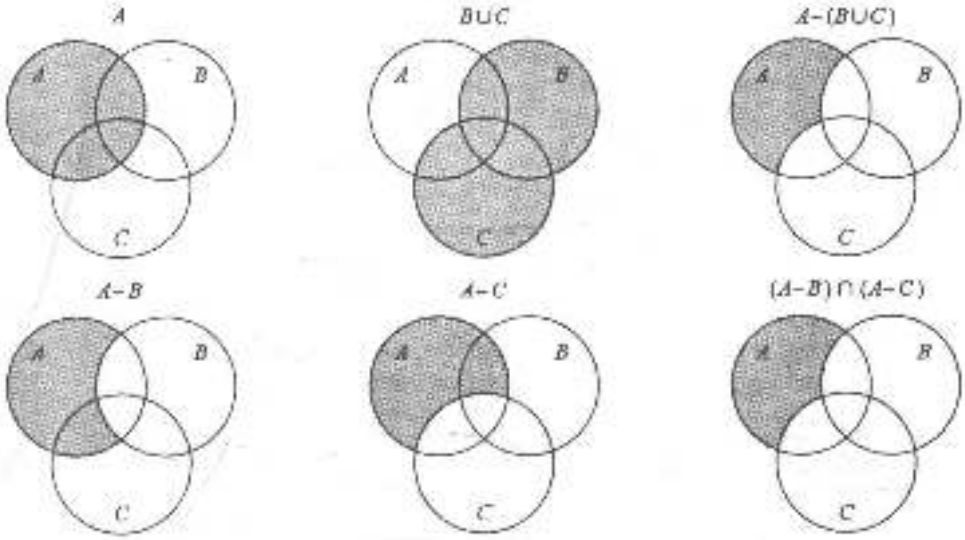


图 17-1 用韦恩图来展示部分德摩根定律

17.2 集合接口的扩展

尽管你已经在本书集合类的论述中遇到过 Set 类，但 Stanford 类库中的 set.h 接口提供了一组更丰富的方法，以及第 5 章云描述的一些扩展的操作符。或许更重要的是，Stanford 类库中关于 Set 类的实现包含了几种方法（它们在 Stanford 模板库中是不可使用的）来实现高级的集合运算，例如，求子集、相等判断、并、交和差。该接口的扩展版本显示在图 17-2 中。 742

```
/*
 * File: set.h
 * -----------
 * This interface exports the Set class, a collection for storing a set
 * of distinct elements.
 */

#ifndef _set_h
#define _set_h

#include "map.h"
#include "vector.h"

/*
 * Class: Set<ValueType>
 * ---------------------
 * This template class stores a collection of distinct elements.
 */

template <typename ValueType>
class Set {

public:

/*
 * Constructor: Set
 * Usage: Set<ValueType> set;
 * --------------------------
```

图 17-2 Set 类的接口

```
 * Initializes an empty set of the specified value type
 */

   Set();

/*
 * Destructor: ~Set
 * ----------------
 * Frees any heap storage associated with set.
 */

   ~Set();

/*
 * Method: size
 * Usage: int count = set.size();
 * ------------------------------
 * Returns the number of elements in this set.
 */

   int size() const;

/*
 * Method: isEmpty
 * Usage: if (set.isEmpty())
 * -------------------------
 * Returns true if this set contains no elements.
 */

   bool isEmpty() const;

/*
 * Method: add
 * Usage: set.add(value);
 * ----------------------
 * Adds an element to this set if it is not already there.
 */

   void add(const ValueType & value);

/*
 * Method: remove
 * Usage: set.remove(value);
 * -------------------------
 * Removes an element from this set.  If the value was not contained in the
 * set, the set remains unchanged.
 */

   void remove(const ValueType & value);

/*
 * Method: contains
 * Usage: if (set.contains(value)) . . .
 * -------------------------------------
 * Returns true if the specified value is in this set.
 */

   bool contains(const ValueType & value) const;

/*
 * Method: clear
 * Usage: set.clear();
 * -------------------
 * Removes all elements from this set.
 */

   void clear();

/*
 * Method: isSubsetOf
 * Usage: if (set.isSubsetOf(set2))
 * --------------------------------
 * Implements the subset relation for sets.  This method returns true
 * if every element of this set is contained in set2.
 */

   bool isSubsetOf(const Set & set2) const;
```

图17-2 （续）

```
/*
 * Operator: ==
 * Usage: set1 == set2
 * -------------------
 * Returns true if set1 and set2 contain the same elements.
 */

   bool operator==(const Set & set2) const;

/*
 * Operator: !=
 * Usage: set1 != set2
 * -------------------
 * Returns true if set1 and set2 are different.
 */

   bool operator!=(const Set & set2) const;

/*
 * Operator: +
 * Usage: set1 + set2
 *        set1 + value
 * -------------------
 * Returns the union of sets set1 and set2, which is the set of elements
 * that appear in at least one of the two sets.  The second form returns
 * the set formed by adding a single element.
 */

   Set operator+(const Set & set2) const;
   Set operator+(const ValueType & value) const;

/*
 * Operator: *
 * Usage: set1 * set2
 * -------------------
 * Returns the intersection of sets set1 and set2, which is the set of all
 * elements that appear in both.
 */

   Set operator*(const Set & set2) const;

/*
 * Operator: -
 * Usage: set1 - set2
 *        set1 - value
 * -------------------
 * Returns the difference of sets set1 and set2, which is all the
 * elements that appear in set1 but not set2.  The second form returns
 * the set formed by removing a single element.
 */

   Set operator-(const Set & set2) const;
   Set operator-(const ValueType & value) const;

/*
 * Operator: +=
 * Usage: set1 += set2;
 *        set1 += value;
 * ----------------------
 * Adds all elements from set2 (or the single specified value) to set1.
 */

   Set & operator+=(const Set & set2);
   Set & operator+=(const ValueType & value);

/*
 * Operator: *=
 * Usage: set1 *= set2;
 * ----------------------
 * Removes any elements from set1 that are not present in set2.
 */

   Set & operator*=(const Set & set2);
```

图 17-2 （续）

```
/*
 * Operator: -=
 * Usage: set1 -= set2;
 *        set1 -= value;
 * ------------------------
 * Removes all elements from set2 (or a single value) from set1.
 */

   Set & operator-=(const Set & set2);
   Set & operator-=(const ValueType & value);

   The private section of the class goes here.

};

   The implementation of the class goes here.

#endif
```

图 17-2 （续）

包含高级方法和操作符的 Set 类使得基于集合的算法非常容易理解，主要是因为这些算法的实现最终看起来与它们的数学公式很像。这个事实尤其与第 18 章将论述的图有关。图算法是最重要的，而且对于聪明的程序员是最有吸引力的，你将从本书中学到该算法。我们在斯坦福的教学经验表明，如果代码使用扩展的 Set 类的高级操作符，那么学习这些算法将会非常容易。

743 ~ 746

在集合类中增加并、交和差操作符需要在设计中深入思考。∪和∩符号没有显示在标准键盘上，这个事实暗示着它将会聪明地使用更多的习惯性符号用于这些运算。虽然 C++ 允许程序员扩展操作符集合，但 C++ 限制使用重载现有的操作符。这意味着有必要从 C++ 已经定义的操作符中选择合适的符号来表示集合类的各种高级运算。尽管并和差运算已经用操作符 + 和 - 直观地表示，但是选择一个操作符表示交运算仍有点困难。

尽管 Stanford 类库中 set.h 接口的设计者考虑了其他的可能性，在该类库关于 Set 类的实现中，它采用 * 表示集合交运算。* 操作符在布尔代数的论述中经常被用于这种目的，主要因为值 0 和 1 乘法运算的结果表示交运算更有说服性：

*	0	1
0	0	0
1	0	1

正如二进制乘法表所示的，只有当输入的两个值都是 1 时，产生的值才是 1。使用一种类似的方法，只有当一个元素同时是两个集合的成员时，它才在集合的交集中。

重新定义操作符 +、* 和 -，显然会使用户认为他们也可以使用简写的赋值操作符 +=、*= 和 -=。因此，扩展的 set.h 接口也包含这些操作符。而且，这些操作符将一个集合或单个元素作为其右操作数，例如，通过书写下面的表达式将值 v 添加到集合 s 中：

```
s += v;
```

17.3 集合的实现策略

和映射的情况相似，实现 Set 类通常有两种策略。Stanford 模板库的设计者选择的方

法是使用一棵平衡二叉树作为基本表示。然而，其他编程语言通常采用哈希策略来实现 Set 类，它的效率稍微高一点。使用平衡二叉树最主要的优点是：我们很容易以事先排列好的顺序遍历集合中的元素。

Java 语言通过提供 TreeSet 和 HashSet 类以尽量满足尽可能多的用户需求。Stanford 类库中的 Set 类对应着 STL 库的 TreeSet 类的实现方法，但是 STL 库也提供了 HashSet 类。在习题 5 中，你将有机会实现它。 747

好消息是只要你利用已经拥有的类，TreeSet 和 HashSet 类用 C++ 语言都很容易实现。开发一个简单的实现的根本点在于集合和映射本质上是相同的。你可以使用 Map 类很容易地创建 Set 类。如果你采用这种策略，Set 类除了一个包含 Map 类的单个的实例变量外，不需要任何东西，如图 17-3 所示。映射中的值域被忽略了，通过检测一个关键字是否存在于 Map 中可以确定键与值的关系。然而，Set 类需要一个值域。图 17-3 中的注释表示：这种实现使用 bool 作为值类型表示一个特定的元素在或不在集合中。

```
/*
 * Notes on the representation
 * ---------------------------
 * This implementation of the Set class uses a map as its underlying
 * data structure.  The value field in the map is ignored, but is
 * declared as a bool to suggest the presence or absence of a value.
 * The fact that this class is layered on top of an existing collection
 * makes it substantially easier to implement.
 */

private:

/* Instance variables */

   Map<ValueType,bool> map;           /* Map used to store the elements   */
```

图 17-3　Set 类的私有部分

就其他方面而言，当你定义一个抽象时（正如在当前使用映射实现集合的建议中），最终的抽象被认为是分层的（layered）。分层抽象有许多优点。首先，它们容易实现，因为许多工作可以和已有的低级接口联系起来。

基于映射而分层地实现集合的策略，就其本身而言，对编写 Set 类的相关代码是不够的，Set 类提供了各种各样的高级操作，而它们不属于 Map 类的一部分。然而，这种策略确实提供了一个好的开始。另外，高级的运算容易使用已有的 Map 类的功能来实现。Set 类中的操作符代码见图 17-4。 748

```
/*
 * Implementation notes: Set constructor and destructor
 * ----------------------------------------------------
 * The constructor and destructor are empty because the Map class manages
 * the underlying representation.
 */

template <typename ValueType>
Set<ValueType>::Set() {
   /* Empty */
}

template <typename ValueType>
Set<ValueType>::~Set() {
   /* Empty */
}
```

图 17-4　Set 类的实现

```
/*
 * Implementation notes: size, isEmpty, add, remove, contains, clear
 * ------------------------------------------------------------------
 * These methods forward their operation to the underlying Map object
 */

template <typename ValueType>
int Set<ValueType>::size() const {
   return map.size();
}

template <typename ValueType>
bool Set<ValueType>::isEmpty() const {
   return map.isEmpty();
}

template <typename ValueType>
void Set<ValueType>::add(const ValueType & value) {
   map.put(value, true);
}

template <typename ValueType>
void Set<ValueType>::remove(const ValueType & value) {
   map.remove(value);
}

template <typename ValueType>
bool Set<ValueType>::contains(const ValueType & value) const {
   return map.containsKey(value);
}

template <typename ValueType>
void Set<ValueType>::clear() {
   map.clear();
}

/*
 * Implementation notes: isSubset
 * ------------------------------
 * This method simply checks to see whether each element of the current
 * set is an element of set2.
 */

template <typename ValueType>
bool Set<ValueType>::isSubsetOf(const Set & set2) const {
   for (ValueType value : map) {
      if (!set2.contains(value)) return false;
   }
   return true;
}

/*
 * Implementation notes: operator==, operator!=
 * --------------------------------------------
 * These operators make use of the fact that two sets are equal only
 * if each set is a subset of the other.
 */

template <typename ValueType>
bool Set<ValueType>::operator==(const Set & set2) const {
   return isSubsetOf(set2) && set2.isSubsetOf(*this);
}

template <typename ValueType>
bool Set<ValueType>::operator!=(const Set & set2) const {
   return !(*this == set2);
}

/*
 * Implementation notes: operator+
 * -------------------------------
 * The union operator copies the current set and then adds the elements
 * from set2 to the result.
 */
```

图 17-4 （续）

```
template <typename ValueType>
Set<ValueType> Set<ValueType>::operator+(const Set & set2) const {
   Set<ValueType> set = *this;
   for (ValueType value : set2.map) {
      set.add(value);
   }
   return set;
}

Set<ValueType> Set<ValueType>::operator+(const ValueType & value) const {
   Set<ValueType> set = *this;
   set.add(value);
   return set;
}

/*
 * Implementation notes: operator*
 * -------------------------------
 * The intersection operator adds elements to an empty set only if they
 * appear in both sets.
 */

template <typename ValueType>
Set<ValueType> Set<ValueType>::operator*(const Set & set2) const {
   Set<ValueType> set;
   for (ValueType value : map) {
      if (set2.contains(value)) set.add(value);
   }
   return set;
}

/*
 * Implementation notes: operator-
 * -------------------------------
 * The set difference returns a new set consisting of the elements in
 * the current set that do not appear in set2.
 */

template <typename ValueType>
Set<ValueType> Set<ValueType>::operator-(const Set & set2) const {
   Set<ValueType> set;
   for (ValueType value : map) {
      if (!set2.contains(value)) set.add(value);
   }
   return set;
}

template <typename ValueType>
Set<ValueType> Set<ValueType>::operator-(const ValueType & value) const {
   Set<ValueType> set = *this;
   set.remove(value);
   return set;
}

/*
 * Implementation notes: shorthand assignment operators
 * ----------------------------------------------------
 * These operators modify the current set but are otherwise similar to
 * the operators that create new sets.  The only subtlety is that the
 * intersection operator must create a vector of elements that need to be
 * removed to avoid changing the set while cycling through its elements.
 */

template <typename ValueType>
Set<ValueType> & Set<ValueType>::operator+=(const Set & set2) {
   for (ValueType value : set2.map) {
      add(value);
   }
   return *this;
}

template <typename ValueType>
Set<ValueType> & Set<ValueType>::operator+=(const ValueType & value) {
   add(value);
```

图 17-4 （续）

```
    return *this;
}

template <typename ValueType>
Set<ValueType> & Set<ValueType>::operator*=(const Set & set2) {
    Vector<ValueType> toRemove;
    for (ValueType value : map) {
        if (!set2.contains(value)) toRemove.add(value);
    }
    for (ValueType value : toRemove) {
        remove(value);
    }
    return *this;
}

template <typename ValueType>
Set<ValueType> & Set<ValueType>::operator-=(const Set & set2) {
    for (ValueType value : set2.map) {
        remove(value);
    }
    return *this;
}

template <typename ValueType>
Set<ValueType> & Set<ValueType>::operator-=(const ValueType & value) {
    remove(value);
    return *this;
}
```

749 / 752

图 17-4 （续）

17.4 优化小整数的集合

上一节的 Set 类实现策略适用于任何值类型。然而，这个实现策略可以进行非常大的改进，集合的元素值可采用小整数类型，例如枚举类型或字符类型。

17.4.1 特征向量

暂且假设你正在使用一个集合，其中的元素值总是在 0 和 RANGE_SIZE-1 之间，RANGE_SIZE 是一个常量，以表明元素值的约束范围。你可以使用拥有布尔值的数组来有效地表示这样的集合。数组索引位置 k 处的值表示整数 k 是否在集合中。例如，如果 elements[4] 的值为 true，那么 4 在集合中通过布尔数组 elements 表示。类似地，如果 elements[5] 的值为 false，5 就不是该集合的元素。

若在一个布尔数组中，每个元素的索引与某个集合的元素值存在着一一对应的关系，则这样的布尔数组被称为**特征向量**（characteristic vector）。下面的例子展示了采用这种特征向量策略来表示的若干集合，假设 RANGE_SIZE 的值为 10：



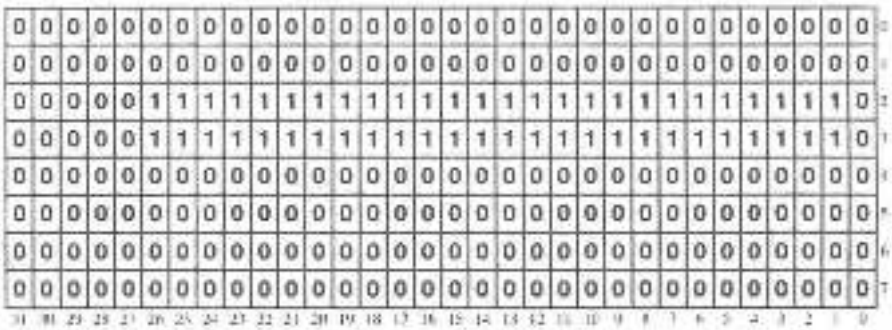
使用特征向量策略的优点是我们能够以常量时间实现 add、remove 和 contains 操作。例如，为了将元素 k 添加到集合中，需要将特征向量中索引位置 k 处的元素值设置为 true。类似地，测试全体成员就是简单地选择集合中合适元素的问题。

17.4.2 压缩的位数组

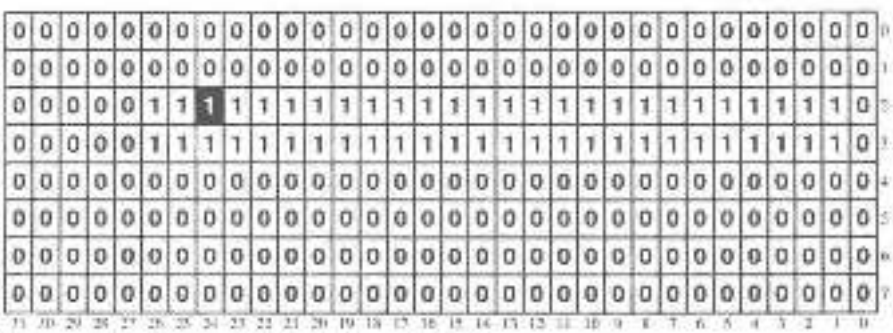
虽然采用特征向量策略可使算法获得高效的运行时间，但将特征向量用数组表示仍需要大量的内存，特别是当 RANGE_SIZE 较大时。为了降低所需的存储空间，你可以将特征向量元素压缩到机器字中，使机器字中的每一位与特征向量的每一位相对应。例如，假设类型 unsigned long 在你机器上是 32 位。既然特征向量中的每个元素只需要一比特信息，那么你就可以用一个单独的 unsigned long 类型的值存储一个有 32 个元素的特征向量。另外，如果 RANGE_SIZE 的值是 256，你可以将一个特征向量所需的所有的 256 个比特存储在一个含有 8 个 unsigned long 值的数组中。

753

为了理解特征向量如何被压缩成一个具有机器字的数组，想象一下你想表示包含关于字母表字符的 ASCII 码的整数集合。该集合包含 26 个大写字母，ASCII 码值在 65～90 之间；包含 26 个小写字母，ASCII 码值在 97～122 之间。因此，它可以编码成下面的特征向量：



如果你想找到对应于特定整数值的比特，最简单的方法是使用整数除法和取模运算。例如，假设你想定位对应于字符 'X' 的比特，它的 ASCII 码值为 88。所求比特的行数是 2，因为每一行有 32 个比特，并且根据标准的整数除法定义，88/32 的结果为 2。类似地，在第 2 行，你找到了 'X' 记录在 24 比特处，它是 88 除以 32 的余数。因此，在特征向量中，对应于字符 'X' 的比特在下面图中加黑的位置：



加黑的位是 1，这表明 'X' 是集合中的一个成员。

754

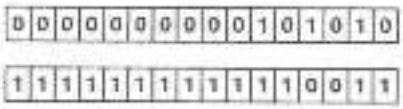
17.4.3 位操作符

为了编写能处理这种压缩的位数组数据的代码，你必须学习如何使用低级的由 C++ 提供的用于操作内存位的**位操作符**（bitwise operator），如表 17-2 所示。这些操作符读取任意的标量类型值并将它们翻译成与底层硬件相对应的比特序列表示。

为了说明位操作符的功能，让我们考虑一个特定的例子。假设在一台机器上，变量 x 和 y 已经被声明为具有 16 个比特的 short 类型，如下所示：

```
unsigned short x = 0x002A;
unsigned short y = 0xFFF3;
```

正如第 11 章所描述的，如果你将初始值从十六进制转换为二进制，很容易就能确定变量 x 和 y 的比特模式，其比特模式如下图所示：



&、| 和 ^ 操作符的语义如表 17-2 所示，其中的每一个操作符都作用于操作数的所有比特位。例如，& 操作符产生了一个结果，即只有当两个操作数对应的比特位都为 1 时，其对应的结果数的比特位为 1。因此，如果将 & 操作符应用到 x 和 y 的比特模式中，你将得到以下结果：

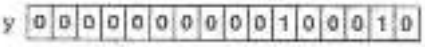
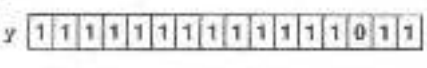


表 17-2 C++ 的位操作符

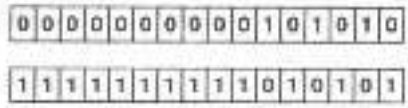
x & y	逻辑与。当 x 和 y 对应的比特位均为 1 时，结果比特位为 1
x | y	逻辑或。当 x 和 y 对应的比特位只要有一个为 1 时，结果比特位为 1
x ^ y	逻辑异或。当 x 和 y 对应的比特位值不同时，其结果比特位为 1
~x	逻辑非。将 x 的各比特位取反，若为 0 时，结果为 1，反之亦然
x << n	左移。将 x 向左移 n 个比特位
x >> n	右移。将 x 向右移 n 个比特位

755

| 和 ^ 操作符产生了以下结果：



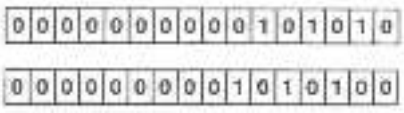
~ 操作符为一元操作符，它将操作数的每一位取反。例如，如果你将 ~ 操作符应用到 x 的比特模式中，结果看起来如下图所示：



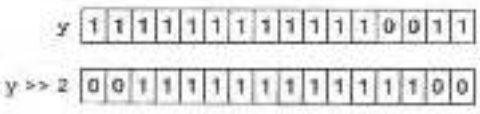
在编程中，~ 操作是取操作符后面的单个操作数的反码，因此，这一操作被称为**取反**（taking the complement）。

操作符 << 和 >> 是移动其左操作数的比特位，移动的位数由右操作数指定。这两个

操作的唯一区别是移动的方向。<< 操作符将比特串左移；而 >> 操作符将比特串右移。因此，表达式 x<<1 产生了一个新的值，其中，值 x 的每一比特向左移动了一个位置，如下图所示：



类似地，表达式 y>>2 产生了一个值，其中，值 y 的每一比特向右移动了两个位置，如下图所示：



只要被移动的值是无符号的，显示在字结尾的被移动的比特数字消失，并且在另一端用比特 0 代替。如果被移动的值是有符号的，移动操作符的行为依赖于硬件的基本特征。由于这个原因，在实际中你要约束对无符号类型值使用移动操作符，从而提高代码的可移植性。 [756]

17.4.4　实现特征向量

在上节介绍的位操作符使我们可以使用一种极其有效的方式来实现特征向量上的操作。如果要测试特征向量上一个单独比特的状态，你需要创建一个值，在期望的位置设比特为 1，其余的位置是比特 0。这样的值被称为掩码（mask），因为它掩盖了字中其他所有的比特位而得名。如果你将 & 操作符应用到表示特征向量的字中，其中包含了你正在试图寻找的比特位和对应于正确比特位置的掩码，字中其他所有的比特都被去除了，只留下反映预期比特状态的值。

为了使这个策略更具体，更深入地思考一个特征向量的底层表示是很有帮助的。下面的代码将特征向量 CharacteristicVector 定义为一个结构，它包含了一个被翻译成比特序列的字数组。

```
struct CharacteristicVector {
   unsigned long words[CVEC_WORDS];
};
```

其中，CVEC_WORDS 是一个常量，定义如下：

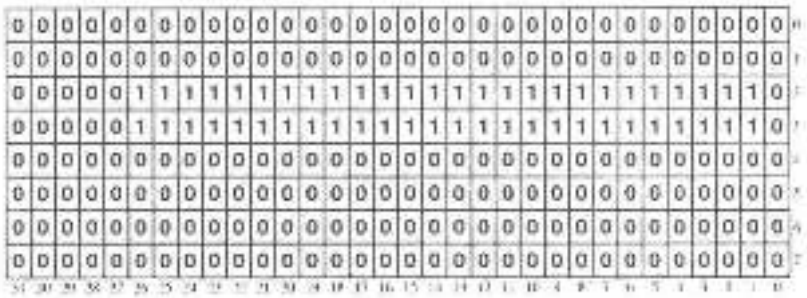
```
const int BITS_PER_BYTE = 8;
const int BITS_PER_LONG = BITS_PER_BYTE * sizeof(long);
const int CVEC_WORDS = (RANGE_SIZE + BITS_PER_LONG - 1)
                       / BITS_PER_LONG;
```

给定这个结构，你可以使用函数 testBit 来测试一个特征向量中的特定比特位，该函数的实现如下所示：

```
bool testBit(CharacteristicVector & cv, int k) {
   if (k < 0 || k >= RANGE_SIZE) {
      error("testBit: Bit index is out of range");
   }
   return cv.words[k / BITS_PER_LONG] & createMask(k);
}
```

```
}

unsigned long createMask(int k) {
   return unsigned long(1) << k % BITS_PER_LONG;
}
```

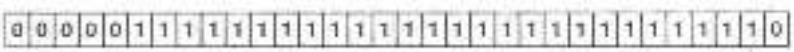
[757] 例如，假设你调用 testBit(cv,'X')，其中 cv 是对应于字母表的字符集合的特征向量。正如在本章前一节中所讲述的，cv 特征向量看起来如下图所示：



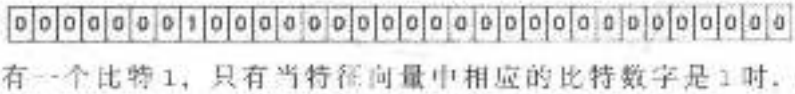
函数 testBit 开始通过计算以下表达式来选择查询所对应的特征向量中合适的字：

```
cv.words[k / BITS_PER_LONG];
```

下标表达式 k/BITS_PRE_LONG 对应着特征向量的第 k 个比特位所在的字的位置。由于字符 'X' 的 ASCII 码值是 88，且 BITS_PER_LONG 是 32，上述下标表达式的结果为 2，即选择索引位置为 2 的字，该字包含以下比特：



函数 createMask(k) 产生一个在合适位置包含比特 1 的掩码。例如，如果 k 的值为 88，k%BITS_PER_LONG 是 24，它意味着掩码包含的值 1 左移了 24 个比特位置，如下图所示：



因为掩码只有一个比特 1，只有当特征向量中相应的比特数字是 1 时，代码中关于 testBit 的 & 操作符才会返回一个非零值。如果特征向量在那个位置包含一个 0，将没有比特是向量和掩码所共有的，这意味 & 操作符将会返回只包含比特 0 的字。一个完全由比特 0 组成的字的整数值为 0。

使用掩码策略也使我们很容易操作特征向量中单独比特的状态。根据惯例，将值 1 赋给一个指定的比特被称为**设置**（setting）比特；赋值 0 被称为**清空**（clearing）比特。你可以
[758] 通过使用逻辑 OR 操作符同时将掩码某位置 1 使一个字的旧值指定位置 1。你也可以通过将逻辑 AND 操作符应用到字的旧值和掩码的补码中。下面的关于两数 setBit 和 clearBit 的定义展示了这些操作：

```
void setBit(CharacteristicVector & cv, int k) {
   if (k < 0 || k >= RANGE_SIZE) {
      error("setBit: Bit index is out of range");
   }
   cv.words[k / BITS_PER_LONG] |= createMask(k);
}

void clearBit(CharacteristicVector & cv, int k) {
```

```
    if (k < 0 || k >= RANGE_SIZE) {
       error("setBit: Bit index is out of range");
    }
    cv.words[k / BITS_PER_LONG] |= createMask(k);
}
```

17.4.5 实现高级集合运算

将特征向量压缩成机器字中的比特位节省了大量的内存空间。同时，这种策略也能改善并、交和集合差这些高级的集合操作的效率。其技巧是通过使用一个单独的合适位操作符来计算新的特征向量中的每个字。

例如，两个集合的并集包含属于任何两个集合之一的所有元素。如果你把这个想法应用到特征向量，就很容易明白：通过采用逻辑 OR 操作符，特征向量集合 A∪B 中的每个字都进行了或运算。进行逻辑 OR 运算使得集合 A 和 B 所对应的比特位若有一个为 1，则相应比特位结果为 1，这正是你所需要计算的并集。

17.4.6 模板特化

就空间和时间而言，特征向量模型允许字符集合比一般的集合实现起来效率更高。尽管在效率上有所区别，但是它对于用户学习两种不同的集合模型（一种是字符集合，另一种是元素类型为任意的集合）是没有影响的。你需要定义两个模板类的实现，然后让编译器来确定集合元素的类型。 759

C++ 允许定义类模板。你可以指定一个带有一个或者多个具有更通用类型的模板参数的类模板。这种技术被称为**模板特化**（template specialization）。如果用户提供的实参类型与模板参数类型相匹配，则编译器就会使用模板的实参类型从类模板中特化出一个具体的模板类。

作为一个例子，假设你想定义（你将有机会在习题 8 中遇到）一个采用字符特征向量来存储数据的类 `Set<char>` 的特化实现，`Set<char>` 类采用字符特征向量来存储数据。该类的接口如下所示：

```
template <>
class Set<char> {
   class body for character sets
};
```

模板后的一对空的尖括号告诉编译器 `Set` 类的这个版本仍被定义为一个模板，但是不允许用户指定模板参数类型。`Set` 类的这个版本的值类型总是 `char` 类型。同样的语法应用于具体化类的方法的实现中。因此，`Set` 类的构造函数有以下结构：

```
template <>
Set<char>::Set {
   implementation of the character set constructor
};
```

`Set<char>` 类的私有部分和实现与更一般的 `Set<ValueType>` 肯定是不同的，因为它们使用不同的数据模型。然而，理想的情况是它们的接口应该是相同的。

17.4.7 使用一种混合的实现

预定义的 `char` 类型只有 256 个可能的值。这个事实使得我们很容易使用一个特征向量

作为底层表示，因为该向量只需要256个比特的存储空间。但是如果你想借用类bitset<int>来实现字符特征向量，将会发生什么呢？在一个使用32位整数的机器上，一个整数特征向量将需要2^{32}个比特，这对于字符类型的特征向量是不合适的。

即使特征向量不能表示所有的整数集合，但只要集合中的值在一个有限的范围内，它们仍然很有用。你必须设计一种数据结构，要求只要整数较小，数据结构就使用特征向量，但是如果一个用户尝试存储一个超出范围的值，数据结构就应转到更通用的
760 二叉树实现方法上。最后，用户只使用在约束范围内的整数就能得到增强性能的特征向量策略。然而，若用户需要定义包含最佳范围之外的整数的集合，也可以使用相同的接口。

本章小结

在本章，你已经学习了关于集合的知识，作为一个理论和实践的抽象，这对于计算机科学是很重要的。集合有着坚实的数学基础（完全不会使它们太抽象以至于没有作用），使它作为一个编程工具有更多的效用。因为集合深厚的理论基础，你可以依靠集合所表现的特定性质并遵守其特定的规则。通过编写关于集合的算法，你可以在集合理论的基础上编写出更易用于理解的程序。

本章的重点包括：

- 一个集合是一个无序的不同元素的容器。本书使用的集合运算以及它们的数学符号见表17-3。
- 如果你牢记集合运算的恒等式，那么在遇到各种各样的集合运算时一般很容易理解。使用这些恒等式也能提高你的编程实践能力，因为它们给你提供了用来简化代码中集合运算的工具。
- 集合类很容易实现，因为它们大多在Map类之上分层，使用基于树或基于哈希表的表示方式。
- 使用称为对称向量的布尔数组能够有效实现整数集合。如果你采用C++提供的位操作符，可以将特征向量压缩成少量的机器字，并且同时在向量中的许多元素上实现如并和交这样的集合运算。

表17-3 有关集合的数学记号汇总

空集	$\emptyset$	集合不包含任何元素
集合成员	$x \in S$	如果x是集合S的一个元素，为真
非集合成员	$x \notin S$	如果x不是S的一个元素，为真
相等	$A = B$	如果A和B包含完全相同的元素，为真
子集	$A \subseteq B$	如果A中所有的元素都在B中，为真
真子集	$A \subset B$	如果A是B的子集并且两集合不相等，为真
并集	$A \cup B$	要么在A中，要么在B中，要么同时在A和B中
交集	$A \cap B$	集合中的元素同时在集合A和B中
集合差集	$A - B$	集合中的元素来自于A集合但不能包含B中的元素

复习题

1. 判断题：集合中的元素是无序的，所以集合 {3, 2, 1} 和集合 {1, 2, 3} 表示相同的集合。
2. 判断题：一个集合可以包含多个同一元素的拷贝。
3. 符号：∅、**Z**、**N** 和 **R** 指的是什么集合？
4. 符号 ∈ 和 ∉ 分别表示什么含义？
5. 使用枚举法来描述下面集合中的元素：

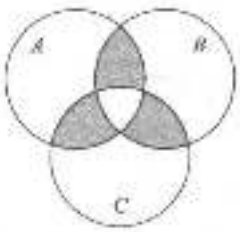
$$\{x \mid x \in \mathbf{N} \text{ 且 } x \leq 100 \text{ 且 } \sqrt{x} \in \mathbf{N}\}$$

6. 写一个关于下面集合的基于规则的定义：

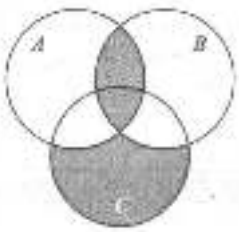
$$\{0, 9, 18, 27, 36, 45, 54, 63, 72, 81\}$$

7. 什么数学符号用于表示运算并、交和差？
8. 计算下面的集合表达式：
   a) $\{a, b, c\} \cup \{a, e, e\}$　　b) $\{a, b, c\} \cap \{a, e, e\}$
   c) $\{a, b, c\} - \{a, e, e\}$　　d) $(\{a, b, c\} - \{a, e, e\}) \cup (\{a, b, c\} - \{a, e, e\})$
9. 子集和真子集的区别是什么？
10. 给出一个有限集合的例子，要求它是另一个有限集的真子集。
11. 对于以下每一个集合运算，画出它们各自的韦恩图。图中的阴影区域表示集合表达式的计算结果：
    a) $A \cup (B \cap C)$　　b) $(A - C) \cap (B - C)$
    c) $(A - B) \cup (B - A)$　　d) $(A \cup B) - (A \cap B)$
12. 写出表示以下每个韦恩图的阴影区域所表示的集合表达式：

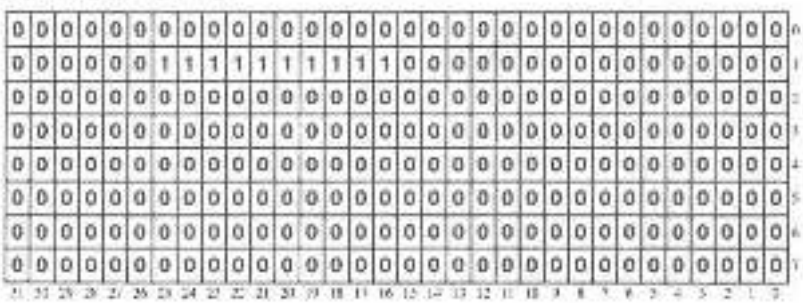
a)



b)



13. 画出表示表 17-1 中每个恒等式的韦恩图。
14. 什么是一个集合的基数？
15. `Set` 类的一般实现方法是采用一个前面章节介绍的数据结构来表示集合元素。那种结构是什么？那种结构的什么性质使得它对于这个目的有用？
16. 什么是特征向量？
17. 为了使用特征向量作为一种实现策略，集合必须要有什么限制？
18. 假设 `RANGE_SIZE` 的值为 10，画出集合 {1, 4, 9} 的特征向量表示。
19. 以下的特征向量表示什么集合：



761
762

通过查询表1-2所示的ASCII码表，确定在 <cctype> 中与该集合相对应的函数。

20. 在特征向量图中，一个 unsigned long 类型为32位。假设你有一台机器，其中64位的 long 类型数代替了原来32位的 unsigned long 类型数，那么本章给出的代码还能在你的机器上继续工作吗？

763

21. 假设变量 x 和 y 的类型都是 unsigned short，并且包含以下比特：

x 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 1

y 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

以比特序列的形式给出以下表达式的值：

a) x & y
b) x | y
c) x ^ y
d) x ^ x
e) ~x
f) x & ~y
g) ~x & ~y
h) y >> 4
i) x << 3
j) (x >> 8) & y

22. 使用八进制和十六进制，将上题中的变量 x 和 y 作为常数计算它们的值。

23. 假设变量 x 和 mask 都被声明为 unsigned 类型，并且 mask 的值在某些位置上包含一个单独的比特 1。你将用什么表达式来实现下面的操作：

a. 测试在 x 与 mask 相对应的比特，观察它是否非零。

b. 设置在 x 中和 mask 中的比特相对应的比特

c. 清除在 x 中和 mask 中的比特相对应的比特

d. 在 x 中和 mask 中的比特相对应的比特的补码

24. 编写一个表达式，构建一个 unsigned 类型的掩码，其中，在比特位置 k 处，有一个单独的比特 1，比特是从 0 开始计数，并且从右边结尾处开始。例如，如果 k 为 2，表达式应该产生下面的掩码：

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

习题

1. 为了使我们更容易编写使用 Set 类的程序，set.h 接口提供一个 >> 操作符的重载版本，从一个输入流中读取集合，这将很有用。在可能的情况下，>> 操作符应该和前面章节中描述的 << 操作符保持对称。特别地，>> 操作符应该要求集合中元素都在花括号中，并用逗号隔开。

764

2. 编写一个简单的测试程序，要求该程序使用前面习题中的 >> 操作符来读取两个字符串集合，然后展示在这些集合上调用并、交和差操作符的结果。这个程序的一个示例运行结果可能如下图所示：



3. 编写一个函数：

```
Set<int> createPrimeSet(int max)
```

该函数返回一个包含 2 和 max 之间所有素数的集合。一个数 N 如果是素数则当且仅当它只有两个公约数：1 和数 N 本身。然而，素数的检测不需要你尝试每一个可能的公约数。你只需要检测的数是

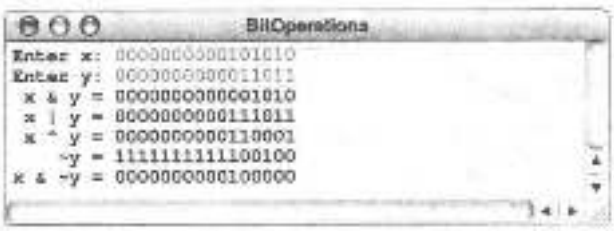
2 和 N 的平方根之间的素数。当它检测一个数是合为素数时，你应该利用这个事实，即所有潜在的因数一定在你已经构建的素数集合中。

4. 本书只简述了 Set 类的实现方法。通过补充私有的部分来完成 Set 类。私有部分定义了集合的结构和实现 Set 类的代码以完成 Set 类，其中，将代码作为一个映射的基于树的版本之上的分层抽象。

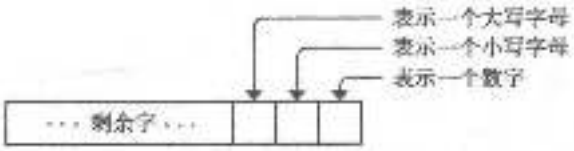
5. 正如本章中所论述的，集合的类库实现使用了平衡二叉搜索树来确保集合的迭代按排列的顺序产生键。如果集合的迭代不成问题，你可以通过使用哈希表作为基本表示以获得更好的性能。采用这个策略，写出 HashSet 类的接口和实现。

6. 编写一个程序，要求实现下面的过程：
   - 读取两个字符串，每一个代表一个比特序列。这些字符串必须只包含字符 0 和 1，并且必须都是 16 个字符的长度。
   - 使用比特相同的内在模式，将这些字符串转化为 unsigned short 类型的值。假设用于存储转换后的结果的变量名为 x 和 y。
   - 将下面表达式表示为一个 16 比特的序列：x & y、x | y、x ^ y、~y、x & ~y。 765

   下面的示例运行结果展示了这个程序的操作：



7. 在大部分的计算机系统中，第 3 章介绍的 ANSI<cctype> 接口是采用位操作符实现的。其实现策略是使用一个字中的特定比特位位置来表示一个下标所对应的字符。例如，想象一下，如下图所示的在一个字右边结尾处的三个比特位用来指出一个字符是否是一个数字、一个小写字母，或者是一个大写字母？



   如果你创建了由 256 个这种字组成的数组（一个字用于表示一个字符），那么你可以实现 <cctype> 中的函数，这样每个函数需要选择数组中的合适元素，应用某个位操作符并测试结果。

   使用这种策略实现 <cctype> 接口的一个简化版本，要求该接口提供函数 isdigit、islower、isalpha 和 isalnum。重要的是要确保函数 isalpha 和 isalnum 的实现不能比其他函数有更多的操作符。

8. 实现一个 Set<char> 模板类，它采用字符矢量表示而不是通常的二叉树表示。

9. 实现一个 Set<int> 模板类，若其中的元素值在 0~255 的范围内，要求使用特征向量表示。但是如果用户添加了范围之外的元素，必须使用传统的表示方法。 766

图

因此我无法靠链路将世界连在一起，从 Delos 到 Limerick 再返回。

——鲁德亚德·吉卜林（Rudyard Kipling），
"班卓琴之歌"（The Song of the Banjo），1894

现实世界中很多结构都是由一系列用链相连接的值构成的。这样的结构就是图（graph）。图的常见例子包括由高速公路所连接的城市，由超链接连接的网页，由各种预备知识连接的大学课程。尽管在数学中分别用顶点（vertex）和边（edge）来表示它们，但程序员一般将独立的元素（如城市网页和课程等）称为节点（node）；而将像高速公路、超链接以及先决条件这些相互的联系称为弧（arc）。

图是由一系列链连接的节点构成的，它和第 16 章介绍过的树比较相像。事实上，它们的唯一不同点在于：对图中的连接结构的约束要比树少。例如，图中的弧通常会有回路。而在树中，由于每一个节点都要通过一个特殊的线与它的根部相连，所以这种结构是不允许出现的。因为树中的约束不适合图，所以图是一种更通用的类型，而树是图的一个子集。因此，每棵树都是一个图，而某些图不是树。

在本章，你将从理论和实践两个方面学习图。学会将图作为一种编程工具是很有用的，因为它在大量的环境中都会出现。掌握这种理论也很重要，它为我们对那些实践性较强的问题找出更有效的解决方案提供了可能。

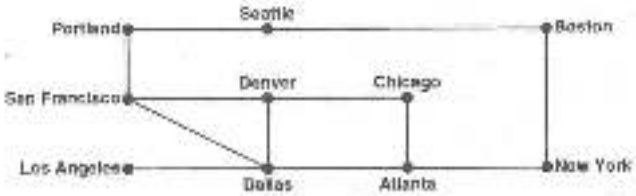
18.1 图的结构

理解图结构最简单的方法就是考虑一个简单的例子。假如你在一家小型航空公司工作，它为美国 10 个主要城市提供航班，航线如图 18-1 所示。标出的圆点代表城市，它构成了图的节点。城市间的线条代表航线，构成了图的弧。



图 18-1 小型的服务于 10 个城市的航线图

图通常用来表示地理关系，但你要牢记图仅仅是由节点和连接边定义的。从图的抽象概念来看，它的布局并不重要。例如，下图和图 18-1 所表示的图是一样的：



节点代表城市的位置，而不是它地理上的正确位置，但是它们之间的关系是相同的。

你可以更进一步忽略它们之间的几何关系。数学家使用一系列的理论将图定义为两个集合的组合，即 V 和 E 这两个在数学中被称为顶点（vertex）和边（edge）的集合。例如，航线图由下列集合构成：

V = { Atlanta, Boston, Chicago, Dallas, Denver, Los Angeles,
New York, Portland, San Francisco, Seattle }

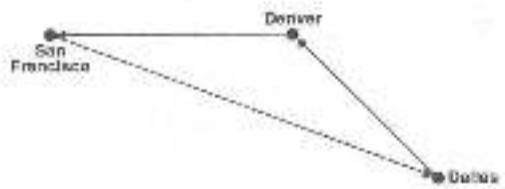
E = { Atlanta↔Chicago, Atlanta↔Dallas, Atlanta↔New York,
Boston↔New York, Boston↔Seattle, Chicago↔Denver,
Dallas↔Denver, Dallas↔Los Angeles, Dallas↔San Francisco,
Denver↔San Francisco, Portland↔San Francisco,
Portland↔Seattle }

图除了在数学理论上的重大意义外，用集合方式来定义图简化了它的实现。因为 Set 类已经实现了很多必要的操作。

768 ~ 769

18.1.1 有向图和无向图

由于图中的弧没有标示方向，因此，在图 18-1 中的弧线就表示飞机在两个方向上都是有航班的，例如，Atalanta 和 Chicago 之间有连线意味着也存在 Chicago 到 Atalanta 的航班。如果在一个图中所有节点的连接都是双向的，则称该图为无向图（undiercted graph）。很多情况下，应需要用有向图（diercted graph）的，其中的每个弧都是有方向的。例如，如果你的航线从 San Francisco 到 Dallas 有直航航班，且返航时在 Denver 有一次停留，那么这样的路线在有向图中看起来如下图所示：



本书中的有向图必须是用带箭头的弧以指明其方向的图。如果没有箭头的话，可以认为它是无向图，如在图 18-1 中看到的航线图一样。

有向图中的弧可以用记号 start → finish 标注，start 和 finish 是有向弧的两个节点。因此，上图中那个三角路线图可以由下面的弧组成：

San Francisco→Dallas
Dallas→Denver
Denver→San Francisco

尽管无向图中的弧经常用双箭头表示，但实际上你不需要一个独立的符号。如果一个图中包含无向弧，你可以用一对有向弧来表示它。例如，如果一个图中包含双向弧 Portl ↔ Seattle，你可以通过 Portl → Seattle 和 Seattle → Portland 这对弧来表示这个事实。因为经常可以用有向图来表示无向图，所以包括本章介绍的大多数图都支持有向图。如果你想定义一个无向图，需要为每一对有联系节点在两个方向上都分别定义一个有向弧。

18.1.2 路径和回路

在图中，弧表示直接的联系，这与航班例子中不停靠的航班是一致的。在航线图中，不
770 存在 San Francisco → New York 的航班，这并不意味着你不可以坐飞机在那些城市之间旅行。如果你想从 San Francisco 飞往 New York，你可以使用如下任意一条路线：

San Francisco → Dallas → Atlanta → New York
San Francisco → Denver → Chicago → Atlanta → New York
San Francisco → Portland → Seattle → Boston → New York

允许你从一个节点到达另外一个节点的弧序列被称为**路径**（path）。如果该路径的起点和终点是相同的，例如以下路径

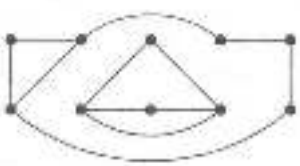
Dallas → Atlanta → Chicago → Denver → Dallas

则称它为**回路**（cycle）。**简单路径**（simple path）是不包含重复节点的路径。同理，**简单回路**中除了起点和终点相同之外，其余各节点也是互不相同的。

在图中，如果两个节点之间是用一条弧连接在一起的，则称它们为**相邻节点**（neighbors）。如果你计算了某个节点的相邻节点数，则称该数是这个节点的**度**（degree）。例如，在航线图中，Dallas 的度数为 4，因为它与 4 个城市直接相连，即它与 Atlanta、Denver、Los Angeles 和 San Francisco 相连。另外，Los Angeles 的度数为 1，因为它仅仅和 Dallas 相连。在有向图中，区别**入度**（in-degree）和**出度**（out-degree）是很有用的。入度是以该节点为终点的弧的数量，而出度是以该节点为初始点的弧的数量。

18.1.3 连通性

在一个无向图中，如果每个节点都存在与其他剩余节点的路径，则称该图为**连通的**（connected）。例如，图 18-1 所示的航线图就是按照这种规则连接的。然而，定义图时并不需要将所有的节点连接在一个单一的单元中。如下图所示：



这是一个非连通图。因为不能将图内部的两个节点组成的链与其他的节点相连。

对于任意一个非连通图，你可以将它分解为一系列连通的子图，但是这些子图之间是不存在连接关系的。这些子图称为图的**连通分支**（connected component）。上图的连通分支如
771 下图所示：



对于有向图来说，连通的概念就更为复杂了。如果一个连通图中每对节点都有连接，则称该图为强连通的（strongly connected）。如果一个有向图在去除了弧的方向后是连通图，则称之为弱连通的（weakly connected）。如下图所示：



它不是强连通的，因为你不能按照图中弧所示的方向从右下的节点移动到左上的节点。换句话说，它是弱连通的，这个图在去除了箭头以后是一个连通图，因此它是弱连通的。如果你改变了图中最上面弧的箭头方向，那么所得到的图就是一个强连通图。如下图所示



18.2 表示策略

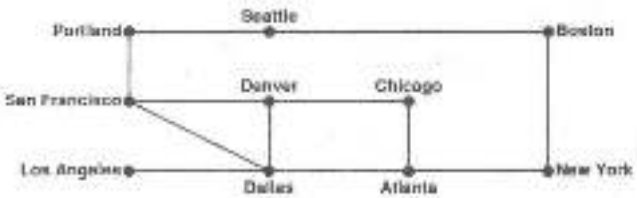
像很多的抽象结构一样，有很多方法来实现图。这些实现最主要的不同点就是用来表达节点之间关系的策略不同。实际上，最常用的策略有：

- 把每个节点的关系存储在邻接表中。
- 将整个图中所有的关系存储在一个邻接矩阵中。
- 将每个节点的关系存在一个集合内。

本节接下来的部分将会进一步详细描述这些表示策略。 772

18.2.1 用邻接表表示连接关系

图中最简单的表示关系的方法就是将每一个节点及与它有联系的节点的数据结构用一个链表示。这种结构被称为**邻接表**（adjacency list）。例如，在当前熟悉的航线图例子中：



每个节点的邻接表如下所示：

```
Atlanta      → (Chicago, Dallas, New York)
Boston       → (New York, Seattle)
Chicago      → (Atlanta, Denver)
Dallas       → (Atlanta, Denver, Los Angeles)
Denver       → (Chicago, Dallas, San Francisco)
Los Angeles  → (Dallas)
```

```
Los Angeles  →  {Dallas}
New York     →  {Atlanta, Boston}
Portland     →  {San Francisco, Seattle}
San Francisco →  {Dallas, Denver, Portland}
Seattle      →  {Boston, Portland}
```

18.2.2 用邻接矩阵表示连接关系

尽管在图的表示中，链表提供了一个简单的表示连接的方法。然而在寻找与某个节点有关的弧序列时，这样的操作就不太高效了。例如，在使用邻接表来表示图时，判断两个节点是否连接需要的时间复杂度为 $O(D)$，其中 D 表示源节点的度数。如果图中所有节点的度数都很小，搜寻这个邻接表的代价就会比较小。然而，若图中每个节点都有较高的度数，那么搜索代价就会相当大。

考虑到效率问题，你可以将图中的弧存入一个被称为邻接矩阵（adjacency matrix）的二维数组中。它存储了节点之间的关系，这样可使检查节点间的关系操作变为常量时间。航线
773 图的邻接矩阵如下图所示：

	Atlanta	Boston	Chicago	Dallas	Denver	Los Angeles	New York	Portland	San Francisco	Seattle
Atlanta			×	×			×			
Boston							×			×
Chicago	×				×					
Dallas	×				×	×			×	
Denver			×	×					×	
Los Angeles				×						
New York	×	×								
Portland									×	×
San Francisco				×	×			×		
Seattle		×						×		

对于像这样的无向图而言，它的邻接矩阵是对称的（symmetric），说明这个矩阵主对角线两侧的内容完全一样，主对角线如图中的虚线所示。

为了使用邻接矩阵存储图，表中的行号或列号都表示图中的一个节点。作为该图具体结构的一部分，实现中需要分配一个由行号和列号组成的表示图中节点的二维网格。数组中元素的值为布尔值。如果 `matrix[start][finish]` 中的值为真，则在图中存在一条 start → finish 的弧。

从执行时间来看，使用邻接矩阵方法明显比使用邻接表快。另外，邻接矩阵需要的存储空间为 $O(N^2)$，其中 N 表示图中节点的个数。虽然在某些图中并非如此，但对于大部分图来说，从空间角度来看使用邻接表的表示更为高效。在邻接表表示中，每个节点和其他节点都有一个连接列表，在最坏的情况下，节点的连接数可能达到 D_{max}，其中 D_{max} 是图中各个节点中最大的度数，即从一个单独节点所发出的最大弧数。因此，邻接表的空间复杂度为 $O(N \times D_{max})$。如果大多数的节点和其他节点是相互连通的，D_{max} 的值就与 N 值相对比较接近，这也就意味着用这两种方法表示的代价是相当的。另一方面，如果一个图有很多节点，但各个节点之间的联系相对较少，使用邻接表表示就可以节约大量的空间。

尽管图的分类界线还没有严格的定义，但那些 D_{max} 的值与 N 值相比而言较小的图被称为稀疏图（sparse）。反之，D_{max} 的值与 N 值接近的图被称为稠密图（dense）。通常，你采用的图算法及表示策略取决于你是否想让此图成为稀疏图或稠密图。例如，之前的分析表明对于稀疏图用邻接表表示更适合；而处理稠密图时，采用邻接矩阵是一个更好的选择。 774

18.2.3 用弧集合表示关系

表示图关系的第三种策略的动机来源于数学公式化地将一个图表示为一个节点和一系列弧的集合。如果你满足于除了一个节点的名字之外不存储别的有关节点的信息，你就可以将图定义为节点与弧的集合，如下所示：

```
struct StringBasedGraph {
   Set<string> nodes;
   Set<string> arcs;
};
```

节点集合包括图中的所有节点。弧集合包括一对以某种方式连接的节点的名字，用这种方式表示可以更容易地区分哪个节点是弧的开始或者结束。

这种图表示方式的最主要优点就是它在概念上的简单性以及它能准确地反映了图的数学定义。然而，这种表示也有两个重要的局限性。首先，找出特定节点的邻接节点需要遍历图中的每条边。其次，最主要的应用是需要将独立的节点和弧的信息与其他附加的信息联系到一起。例如，很多图算法会对每一条弧赋一个数值来表示遍历弧所需要的代价（cost），它可能在实际的货币成本中提及或未提及。例如，图 18-2 中的航线图上的每个弧旁表示它们两个节点之间距离的数字。你可以使用这个信息来实现一个优惠程序，节点表示一个频繁飞行的旅客，而弧值表示所航行的距离。



图 18-2 带有相关旅行数据的飞行线路图

幸运的是，解决上述问题并非特别困难。如果你采用支持迭代的集合类来表示一个图中的结点和弧，则对它们进行迭代会很容易。而且，通过采用特定的数据结构来表示图中的结点和弧，你还可以将额外的数据添加到结点和弧中。

鉴于C++是面向对象语言，你可以将图、节点和弧均用对象来表示，并把每一层定义为一个新类。这种设计肯定适合于这种图的表示问题，可能的实现方法将会在 18.5 节中讲述。然而，下一节将把图用结构表示而不是用类表示。这样做有两个原因。首先，使用结构实现更简单，因为这样会重点关注于高层的操作而不是对象表示的细节。其次，底层的数据

结构在实际中使用得很频繁，由于存在差异很大的图，很难采用一个通用的框架来解决各种以图表示的应用问题。因此，通常将图抽象的相关部分加到一些其他数据结构的实现中是合乎情理的。如果这样做，那么代码与基于数据结构的实现更相像，而非本章后面将要讲的基于对象的实现。

18.3 一种低层的图抽象

这部分将会概述一个低级图包的设计，这里使用C++的结构类型来表示图，其中，每个图、图中的每个节点，连接各个节点的弧是图中三种不同层次的实元素。作为最终设计的第一步，这节首先介绍了一个基于结构类型的接口，它定义了以下三种结构：

- 一个SimpleGraph结构类型。使用这个名字是为了特意将这种类型与本章之后所要介绍的Graph类区分开。和图的数学定义一样，一个SimpleGraph包含两个集合：图中节点的集合；图中弧的集合。并且，由于节点在这个公式化的描述中是有名字的，因此，对SimpleGraph这个结构来说，包含一个允许用户将节点的名字和节点相对应起来的图是很有用的。
- 一个Node结构类型。它包含了节点的名字以及可以指出图中该节点与其他节点的连接关系的集合。
- 一个Arc结构类型。它可以确定弧的结束点，以及一个表示该弧所需代价的数值。

775 ? 776

这种数据类型的非正式描述已经为你提供了几乎足够的信息以编写出图的必要定义，但是还存在一个你在设计过程中必须考虑的问题。SimpleGraph结构概念上所"包含"的Node值不仅仅是它的节点集合的一部分，也是弧集合中元素的一个构件。同理，因为SimpleGraph和Node这两个结构都可以确定一条弧，Arc值会出现在两个地方。在任何情况下，这个抽象结构中涉及同一个实体时，那些结构的不同部分出现的节点和弧必须完全相同。例如，与Atlanta所对应的Node必须和它在顶层的节点集或者内部弧中出现的Node表示的是同一实体。这些节点不是对某个节点的简单复制，因为如果是那样的话，修改一个节点值就不会在其复制的另外一个节点中反映出来。

这个观察的重要含义在于：表示图的集合和结构不能直接包含Node和Arc的值。共享共同结构的需求就意味着所有对Node和Arc的内部引用必须指明具体的Node和Arc值。因此，在SimpleGraph结构中的集合，必须使用Node*和Arc*作为元素类型，而不能用Node和Arc作为元素类型。Node结构体中弧集合和指向节点的指针也是如此。图18-3表示了一个低级的图形接口，称为graphtypes.h，以和18.5节中更高级的graph.h接口进行区分。

图18-3也说明了C++的一个需要进一步解释的新特征。定义的Node和Arc这两个结构类型是互相引用的，Node结构体中包含指向Arc的指针，而Arc结构体中也包含指向Node的指针。这种类型定义上的相互递归使得我们不能像C++要求的那样用已经定义的数据类型来声明这两者中的任何一个类型。

C++允许你声明一个类或结构但无具体内容作为标识符，以此解决这个问题。你要用一个分号代替这个类或者结构体的内容。下面用几乎同样的方法编写一个递归函数。

777

```
struct Node;
struct Arc;
```

以上两行代码告诉编译器Node和Arc是结构体的名称。这样就可以在它们的定义之前声

明指向它们的指针。这样的定义称为**前向引用**（forward reference）。

```
/*
 * File: graphtypes.h
 * ------------------
 * This file defines low-level data structures that represent graphs.
 */

#ifndef _graphtypes_h
#define _graphtypes_h

#include <string>
#include "map.h"
#include "set.h"

struct Node;      /* Forward references to these two types so  */
struct Arc;       /* that the C++ compiler can recognize them. */

/*
 * Type: SimpleGraph
 * -----------------
 * This type represents a graph and consists of a set of nodes, a set of
 * arcs, and a map that creates an association between names and nodes.
 */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};

/*
 * Type: Node
 * ----------
 * This type represents an individual node and consists of the
 * name of the node and the set of arcs from this node.
 */

struct Node {
   std::string name;
   Set<Arc *> arcs;
};

/*
 * Type: Arc
 * ---------
 * This type represents an individual arc and consists of pointers
 * to the endpoints, along with the cost of traversing the arc.
 */

struct Arc {
   Node *start;
   Node *finish;
   double cost;
};

#endif
```

778

图 18-3　基于结构类型的图形抽象

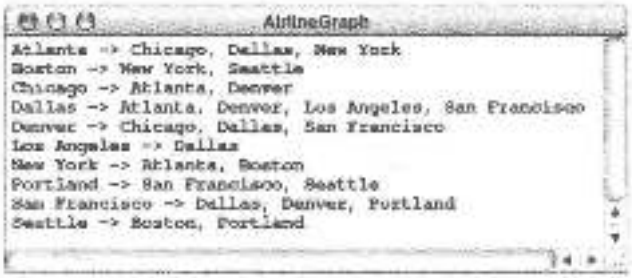
用集合定义图有很多优点。尤其是这种策略意味着这种数据结构与用集合定义图的数学公式化表示是相似的。分层的方法在简化实现上也有很大优点。例如，用集合定义图不需要对图定义一个独立的迭代机制。因此，如果你想在图 g 中遍历其节点，你需要编写以下语句：

```
for (Node *node : g.nodes) {
   处理单个节点的代码
}
```

除了简化迭代过程以外，用集合定义图可以实现并和交这样的高级操作。计算机理论科学家就会使用这些操作来构成图的算法。这些操作对用户也是可用的，使得算法编写也更容易。

graphtypes.h 接口与本书之前的接口不同，它介绍了三种结构类型而不是类和方

法。因此，该接口不需要实现。所以 graphtypes.cpp 文件就没有存在的必要性。由于这个接口没有给用户提供适合于图、节点和弧操作的方法，因此就迫使用户定义自己所需的数据结构。例如，图 18-4 中的代码使用了几个辅助函数以创建本章前面所提到的航线图，然后调用 printAdjacencyLists 函数来遍历每个城市，并且显示可以一步直达的城市名称，如下图所示：



379

```
/*
 * File: AirlineGraph.cpp
 * ----------------------
 * This program initializes the graph for the airline example and then
 * prints the adjacency lists for each of the cities.
 */

#include <iostream>
#include <string>
#include "graphtypes.h"
#include "set.h"
using namespace std;

/* Function prototypes */

void printAdjacencyLists(SimpleGraph & g);
void initAirlineGraph(SimpleGraph & airline);
void addFlight(SimpleGraph & airline, string c1, string c2, int miles);
void addNode(SimpleGraph & g, string name);
void addArc(SimpleGraph & g, Node *n1, Node *n2, double cost);

/* Main program */

int main() {
   SimpleGraph airline;
   initAirlineGraph(airline);
   printAdjacencyLists(airline);
   return 0;
}

/*
 * Function: printAdjacencyLists
 * Usage: printAdjacencyLists(g);
 * ------------------------------
 * Prints the adjacency list for each city in the graph.
 */

void printAdjacencyLists(SimpleGraph & g) {
   for (Node *node : g.nodes) {
      cout << node->name << " -> ";
      bool first = true;
      for (Arc *arc : node->arcs) {
         if (!first) cout << ", ";
         cout << arc->finish->name;
         first = false;
      }
      cout << endl;
   }
}
/*
 * Function: initAirlineGraph
 * Usage: initAirlineGraph(airline);
```

图 18-4 创建航线图的程序

```
 * ------------------------------------
 * Initializes the airline graph to hold the flight data from Figure 18-2.
 * In a real application, the program would almost certainly read this
 * information from a data file.
 */

void initAirlineGraph(SimpleGraph & airline) {
   addNode(airline, "Atlanta");
   addNode(airline, "Boston");
   addNode(airline, "Chicago");
   addNode(airline, "Dallas");
   addNode(airline, "Denver");
   addNode(airline, "Los Angeles");
   addNode(airline, "New York");
   addNode(airline, "Portland");
   addNode(airline, "San Francisco");
   addNode(airline, "Seattle");
   addFlight(airline, "Atlanta", "Chicago", 599);
   addFlight(airline, "Atlanta", "Dallas", 725);
   addFlight(airline, "Atlanta", "New York", 756);
   addFlight(airline, "Boston", "New York", 191);
   addFlight(airline, "Boston", "Seattle", 2489);
   addFlight(airline, "Chicago", "Denver", 907);
   addFlight(airline, "Dallas", "Denver", 650);
   addFlight(airline, "Dallas", "Los Angeles", 1240);
   addFlight(airline, "Dallas", "San Francisco", 1468);
   addFlight(airline, "Denver", "San Francisco", 954);
   addFlight(airline, "Portland", "San Francisco", 550);
   addFlight(airline, "Portland", "Seattle", 130);
}

/*
 * Function: addFlight
 * Usage: addFlight(airline, c1, c2, miles);
 * -----------------------------------------
 * Adds an arc in each direction between the cities c1 and c2.
 */

void addFlight(SimpleGraph & airline, string c1, string c2, int miles) {
   Node *n1 = airline.nodeMap[c1];
   Node *n2 = airline.nodeMap[c2];
   addArc(airline, n1, n2, miles);
   addArc(airline, n2, n1, miles);
}

/*
 * Function: addNode
 * Usage: addNode(g, name);
 * ------------------------
 * Adds a new node with the specified name to the graph.
 */

void addNode(SimpleGraph & g, string name) {
   Node *node = new Node;
   node->name = name;
   g.nodes.add(node);
   g.nodeMap[name] = node;
}

/*
 * Function: addArc
 * Usage: addArc(g, n1, n2, cost);
 * -------------------------------
 * Adds a directed arc to the graph connecting n1 to n2.
 */

void addArc(SimpleGraph & g, Node *n1, Node *n2, double cost) {
   Arc *arc = new Arc;
   arc->start = n1;
   arc->finish = n2;
   arc->cost = cost;
   g.arcs.add(arc);
   n1->arcs.add(arc);
}
```

图 18-4 （续）

如果你仔细思考示例程序的运行结果，会发现所有的城市名称是按字母顺序出现的，这可能会让你比较惊讶。这些图元素集合在建立的时候，最起码在数学形式上是无序的集合。算法中的 for 循环让 Node 和 Arc 指针在内存中出现的顺序和它们在结构体中出现的顺序是相同的。既然内存和城市名称是没有联系的，那么输出结果按照字母序列排序就有点奇怪了。

导致这个看似奇怪的行为的原因就是：C++ 运行时间系统按照指令出现的先后顺序分配堆内存。表 18-4 的初始化操作是按照字母表顺序建立城市以及它们之间联系的。这也就意味着与第一个节点的城市相比，第二个节点的城市存入了更高的内存地址中。因此，输出结果如此有序仅仅是个巧合，在不同的平台上也不一定总是如此。你会在第 20 章看到可以扩展 Set 类的定义，因此它的用户就可以定义元素出现的次序，例如这个例子中的 Graph 类。

780 ⋮ 782

18.4 图的遍历

正如之前的例子，遍历图中的节点是很容易的，只要依照抽象集合规定的顺序来按序处理节点即可。然而，许多图算法要求你在处理节点时把它们之间的联系考虑进去。这种算法通常开始于一些节点，然后沿弧移动，从一个节点前进到另一个节点，并在每个节点上执行某种操作。这个操作的确切性取决于算法，但执行该操作的过程（无论何种操作）被称为访问（visiting）节点。沿着弧访问图中每一个节点的过程被称为图的遍历（traversing）。

在第 16 章你已经学过几种树的遍历方法，最主要的是先序遍历、中序遍历和后序遍历。像树一样，图也提供了不止一种遍历方法。图有两种基本的遍历算法，即**深度优先**（depth-first）搜索和**广度优先**（breadth-first）搜索，它们会在接下来的两节中介绍。

为了使算法机制更易于理解，实现深度优先和广度优先搜索时，假设用户已经提供了一个可对图中的节点进行操作的 visit 函数。遍历的目标就是对每一个节点按照关系决定的顺序调用有且仅有一次的 visit 函数。因为图的遍历经常有不同的路径返回到相同的节点，为了确保遍历算法没有多次访问相同的节点就需要额外的记忆。以下两节的遍历算法实现定义了一个叫做 visited 的节点集合，它用来标记那些已经被处理过的节点。如果遍历算法遇到一个已经在 visited 节点集合中的节点，那么这个节点一定在之前被访问过了。

18.4.1 深度优先搜索

深度优先搜索遍历类似于树的先序遍历，并且有相同的递归结构。唯一复杂的是图中可以包含回路。因此，标记已被访问过的节点是很必要的。实现深度优先遍历的代码开始于一个特定的节点，如图 18-5 所示。

```
/*
 * Function: depthFirstSearch
 * Usage: depthFirstSearch(node);
 * ------------------------------
 * Initiates a depth-first search beginning at the specified node.
 */

void depthFirstSearch(Node *node) {
   Set<Node *> visited;
   visitUsingDFS(node, visited);
}
```

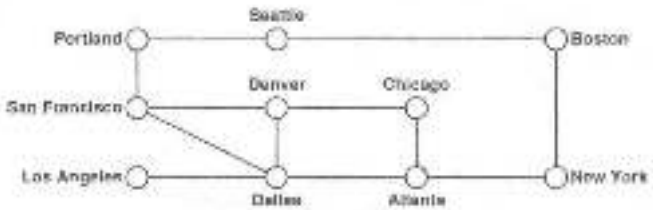
图 18-5 执行深度优先搜索的代码

```
/*
 * Function: visitUsingDFS
 * Usage: visitUsingDFS(node, visited);
 * ------------------------------------
 * Executes a depth-first search beginning at the specified node that
 * avoids revisiting any nodes in the visited set.
 */

void visitUsingDFS(Node *node, Set<Node *> & visited) {
   if (visited.contains(node)) return;
   visit(node);
   visited.add(node);
   for (Arc *arc : node->arcs) {
      visitUsingDFS(arc->finish, visited);
   }
}
```

图 18-5 （续）

在这个实现中，depthFirstSearch 是一个封装的函数，它的唯一功能是产生那些已经被处理过的节点的 visited 集合。函数 visitUsingDFS 访问当前节点，然后对每一个最接近当前节点的节点直接递归调用函数自己。

在一个简单的例子中，通过追踪它的操作是很容易理解深度优先算法的。例如本章开始介绍的航线图：



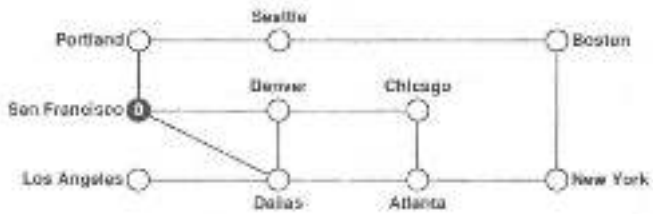
如上图所示，画成空心圆的节点表示它还没被访问过。按照算法的执行步骤，这些圆被逐渐标记，其标记的数字代表节点被处理的顺序。

783 / 784

假设你通过调用以下函数开始对图进行深度优先搜索：

```
depthFirstSearch(airline.nodeMap["San Francisco"]);
```

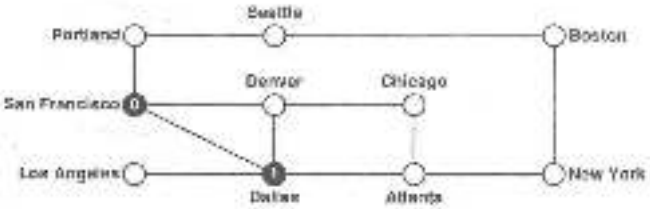
depthFirstSearch 函数的调用首先创建了一个空的 visited 节点集，然后将控制转向对 visitUsingDFS 函数的递归调用。第一次调用访问了 San Francisco 节点，该节点被标记如下图所示：



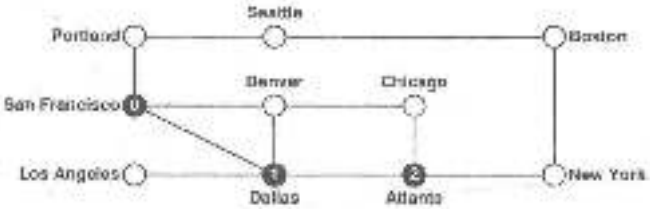
接下来的代码在以下循环语句的每一次循环中对 visitUsingDFS 进行递归调用：

```
for (Arc *arc : node->arcs) {
   visitUsingDFS(arc->finish, visited);
}
```

函数调用的顺序取决于 for 循环语句通过弧的顺序。假设 for 语句按照字母表顺序处理节点，第一次循环对 Dallas 节点调用 visitUsingDFS，从而得到如下的图状态：

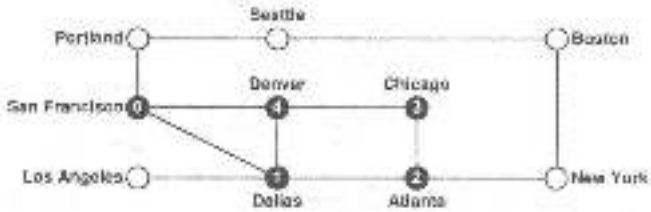


考虑到代码的实现方式，程序必须在考虑了所有以 San Francisco 节点出发的其他可能路径后，才能完成全部涉及 Dallas 节点的函数调用。下一个被访问的节点是首先出现在字母表中从 San Francisco 可达的城市，它就是 Atlanta，如下图所示：

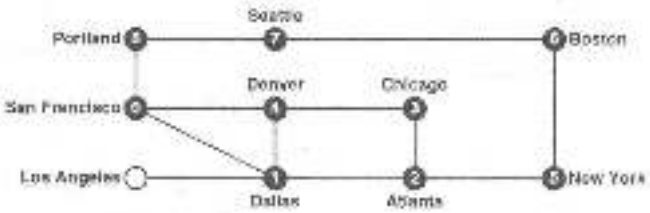


[785]

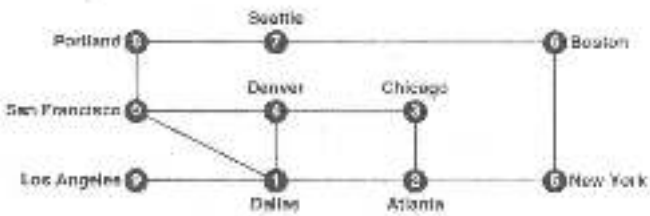
深度优先搜索算法的最终结果是在回溯之前在图中探索一条尽可能长的路径，来完成高层次的路径探索。从 Atlanta 节点，程序通过选择首先出现在字母表中的相邻节点沿着路径持续下去。因此按照深度优先搜索算法，接下来应为 Chicago 和 Denver 节点，得到的图具有以下状态：



然而，在 Denver 节点处不可能再向前了，每一个与 Denver 相邻的节点都已被访问过，因此此次函数调用立即返回。递归过程返回到 Chicago 节点，仍然发现没有相邻节点在未被探索的范围内。递归回溯返回到 Atlanta 节点，现在可以找到并且探索 New York 节点。如此下去，深度优先算法探索到了尽可能长的一条路径，如下图所示：



至此，算法过程将倒退到 Dallas 节点，并从该节点找到 Los Angeles 节点，如下图所示：



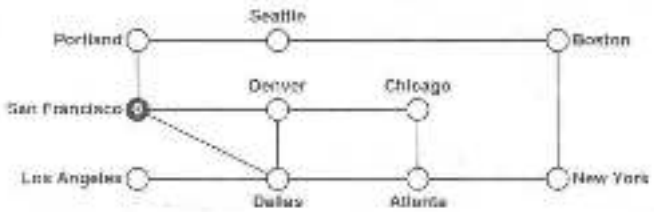
如果你考虑到深度优先算法和其他已知算法之间的关系，就会意识到它的操作和第 9 章讲的迷宫问题的算法特别类似。在那个算法中，为了避免在迷宫中永远地绕着一个圈转，标记正方形是很必要的。在迷宫中的标记类似于深度优先搜索算法中 visited 集合中的节点。 786

18.4.2 广度优先搜索

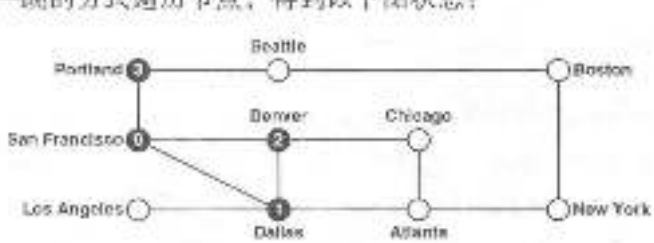
即使深度优先搜索有许多重要的应用，但这种方法的缺点是它对某些应用是不适合的。深度优先算法最大的问题是在它回溯和寻找到其他相邻节点之前，探索了从一个相邻节点开始的整个路径。如果你在一个大图中设法寻找两个节点之间的最短路径，采用深度优先算法会带你到图中最远的地方，即使你的目的地沿着另一条路线只有一步之遥。

广度优先搜索算法以确定的顺序来访问图中的每个节点，从而解决了深度优先搜索的问题，它通过度量一个节点距离起始节点的远近来决定是否访问该节点，以沿着可能的最短路径弧的数目作为度量标准。当你通过弧的数目测量距离时，每一条弧构成一个**跳跃**（hop）。因此，广度优先搜索的本质是：你首先访问起始节点，然后以跳跃方式离开刚访问过的节点，接着再进行第二次跳跃，以此类推。

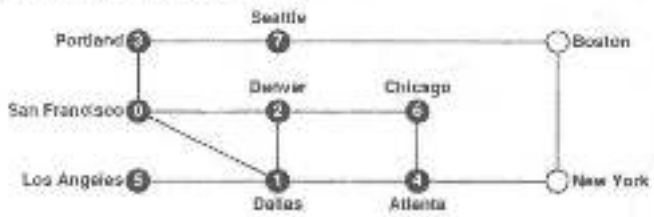
为了对这个算法有一个更加准确的认识，假设你想在航线图上再次从 San Francisco 节点开始采用广度优先遍历。算法首先仅简单地访问起始节点，如下图所示：



下一步以每次一跳的方式遍历节点，得到以下图状态：



到此，算法开始探索有两个跳跃的节点： 787



最后一步，算法通过访问从开始数三个跳跃的节点来完成对图的探索，如下图所示：



实现广度优先算法最简单的方法是使用未处理节点的队列，程序的每一步将与当前节点相邻的节点入队。因为队列是按顺序处理的，所有从开始节点只有一个跳跃的节点将比具有两个跳跃的节点较早的出现在队列中，如此等等。在图 18-6 中展示了这个方法的实现。

```
/*
 * Function: breadthFirstSearch
 * Usage: breadthFirstSearch(node);
 * ------------------------------
 * Initiates a breadth first search beginning at the specified node.
 */

void breadthFirstSearch(Node *node) {
   Set<Node *> visited;
   Queue<Node *> queue;
   queue.enqueue(node);
   while (!queue.isEmpty()) {
      node = queue.dequeue();
      if (!visited.contains(node)) {
         visit(node);
         visited.add(node);
         for (Arc *arc : node->arcs) {
            queue.enqueue(arc->finish);
         }
      }
   }
}
```

788

图 18-6 执行广度优先搜索的代码

18.5 定义图类

图 18-3 所示的 graphtypes.h 接口还有许多待完善之处。尤其是现有的接口使用了底层的数据结构来表示图，因此就不能使用 C++ 的面向对象特性。而且，使用 C 风格的结构类型意味着不存在和图相绑定的类的方法，这就迫使用户开发自己的工具。以下两节将会阐述两种更高级的策略来替代低级的图表示。

18.5.1 用类表示图、节点和弧

如果设计 graph.h 接口的最主要目的是最大化地利用面向对象的优势的话，那么，显然应采用类来代替原有的表示图的每一个低级的结构。采用这种策略，接口就会输出一个 Graph 类，它代替了原有的 SimpleGraph 结构，以及与 Node 和 Arc 结构类型相匹配的类。那些类的私有部分就可以采用上述原有的结构类型。然而，用户要访问这些域的话就需要调用类中的方法而不能直接访问或引用。

尽管这种设计可行，但它在实际应用中比较麻烦。为了了解其原因，注意到图的使用方式与我们所熟悉的容器类是不同的，这一点很重要，例如数组、栈、队列和集合。这些更方便的容器类包含了某种用户定义的类型值，例如，在本书中，你会看到程序中声明的 Stack<double> 和 Set<string> 类型的对象。尖括号里面的模板参数类型是一个值类型。然而，值类型对类本身的实现不会产生很大影响。

而图的情况就不同。图的元素是节点和弧，这些节点和弧结构是图不可或缺的部分，它们包含了维护整个数据结构所需的信息。例如，节点需要保存它和其他节点相连的弧集，弧需要记录它们的端点。同时，用户可能基于应用希望给每个节点或者每条弧增加附加的数据信息。因此，节点和弧是混合的结构体，它们保存了用户以及实现所需要的数据。

大多数面向对象语言在这种情况下最常使用的策略就是子类。在这种模型中，图被定义

为节点类和弧类的父类（或称基类），它包含了表示一个图结构所需的信息。用户通过对图 789
类添加数据域和方法将其扩展为子类。这种方法对C++并不适用，其原因在第19章将进行更深入的探索。最主要的问题是动态内存分配和继承在C++中不能一起使用，这一点与其他语言不同。因此，有必要选择另一种不同的策略。

18.5.2 用参数化的类实现图

幸运的是，使用C++模板可以设计一个Graph类，这样既可以使用面向对象设计的优点，又能保留底层数据结构的简单性。这种设计的基本理念就是：产生一个参数化的Graph类，用户可以选择图中节点和弧的类型。然而，Graph的模板参数类型不能任意选择，它必须是能提供对图进行基本操作的类型，而底层的数据结构刚好满足这一需要。因此，用户选择表示节点的类型必须包含：

- 一个name的字符串，用它来指明节点名。
- 一个arcs域，用它来指明开始于该节点的弧的集合。

选择表示弧的类型必须包含：

- 名为start和finish的域，用来表示弧开始和结束节点。

除了所需要的域，用来表示节点和弧的类型还应该包含用户应用所需要的附加信息。

基于用户定义的Node和Arc域类型在图中必须保持一致，因为所有指向节点和弧的指针都需要使用用户自定义的类型。因此，每个节点所包含的弧集合的元素必须是指向用户定义的Arc类型的指针类型。同样，在Arc结构中的两个节点指针也必须声明为用户定义的Node类型。

Graph类必须具有对Node和Arc类型特定部分的访问权限。一个提供访问权限的策略就是使那些域成为公有部分，就像在任何数据结构中的那样。一个支持封装的更好方法就是将它们在类的私有部分中声明，但是可以通过使用友元（friend）声明使Graph类访问它们。

用一个简单的例子来说明那些规则是很有效的。图18-7的代码定义了两个类，City类和Flight类，二者都没有公有变量。然而，这两个类都包含了所需要的私有部分，并且将Graph<City,Flight>定义为友元类。 790

```
class City;          /* Forward references to these two types so  */
class Flight;        /* that the C++ compiler can recognize them. */

/*
 * Class: City
 * -----------
 * This class defines the node type for the airport graph
 */

class City {

public:
   string getName() {
      return name;
   }

private:
   string name;
   Set<Flight *> arcs;
   string airportCode;
   friend class Graph<City,Flight>;
};
```

图18-7 再次定义的航线图所使用的类

```
/*
 * Class: Flight
 * -------------
 * This class defines the arc type for the airport graph
 */

class Flight {

public:
   City *getStart() {
      return start;
   }

   City *getFinish() {
      return finish;
   }

   int getDistance() {
      return distance;
   }

   void setDistance(int miles) {
      distance = miles;
   }

private:
   City *start;
   City *finish;
   int distance;
   friend class Graph<City,Flight>;
};
```

791

图 18-7 （续）

鉴于 City 和 Flight 类的定义，航线图自身就变成了一个用适当的参数节点和弧类型由 Graph 类模板所生成的一个模板类的一个实例，如下所示：

```
Graph<City,Flight> airlineGraph;
```

表 18-1 以表格形式列出了 graph.h 接口中的方法。图 18-8 所示为该接口中的实际内容。正如你从表 18-1 或图 18-8 所看到的，graph.h 的接口通常对每个方法都提供了多个重载函数版本，从而确保类中的方法能方便地供用户使用。

表 18-1 graph.h 接口中的内容

构造函数	
Graph<*nodetype,arctype*>()	创建一个没有节点和弧的空图
方法	
size()	返回图中的节点数
isEmpty()	如果图中没有节点，则返回 true
clear()	将图中所有的节点和弧删除
addNode(*name*) addNode(*node*)	给图中增加一个节点。第一种形式创建一个名为参数 name 的新节点并加入到图中；第二种形式将用户创建的节点 node 加入到图中
removeNode(*name*) removeNode(*node*)	从图中删除一个节点以及所有和它相连的弧
getNode(*name*)	返回图中参数名的节点，如果该节点不存在，返回空值
addArc(s_1, s_2) addArc(n_1, n_2) addArc(*arc*)	在图中增加一条包含两个参数节点的弧。前两种形式是增加一个包含特定节点的弧；第三种形式是用户增加的节点

（续）

removeArc(s1, s2) removeArc(n1, n2) removeArc(arc)	删除连接特定节点的各个弧
isConnected(s1, s2) isConnected(n1, n2)	如果两个节点间存在弧，则返回 true
getNodeSet()	返回图中所有节点的集合
getArcSet()	返回图中所有弧的集合
getArcSet(name) getArcSet(node)	返回特定节点的弧集合
getNeighbors(name) getNeighbors(node)	返回当前节点的所有相邻节点的集合，在某种意义上存在一个特定节点到它相邻节点间的一条弧

792

```
/*
 * File: graph.h
 * -------------
 * This file is the interface for a flexible graph package that exports
 * a parameterized Graph class.
 */

#ifndef _graph_h
#define _graph_h

#include <string>
#include "map.h"
#include "set.h"

/*
 * Class: Graph<NodeType,ArcType>
 * ------------------------------
 * This class represents a graph with the specified node and arc types.
 * The NodeType and ArcType parameters indicate the record or object types
 * used for nodes and arcs, respectively.  These types can contain any
 * fields or methods required by the client, but must contain the following
 * fields required by the Graph package itself:
 *
 * The NodeType definition must include:
 *   - A string field called name
 *   - A Set<ArcType *> field called arcs
 *
 * The ArcType definition must include:
 *   - A NodeType * field called start
 *   - A NodeType * field called finish
 */

template <typename NodeType,typename ArcType>
class Graph {

public:

/*
 * Constructor: Graph
 * Usage: Graph<NodeType,ArcType> g;
 * ---------------------------------
 * Creates an empty Graph object.
 */

   Graph();

/*
 * Destructor: ~Graph
 * ------------------
 * Frees the internal storage allocated to represent the graph.
 */

   ~Graph();
```

图 18-8 参数化的 Graph 类接口

```
/*
 * Method: size
 * Usage: int size = g.size();
 * ---------------------------
 * Returns the number of nodes in the graph.
 */

   int size() const;

/*
 * Method: isEmpty
 * Usage: if (g.isEmpty()) . . .
 * -----------------------------
 * Returns true if the graph is empty.
 */

   bool isEmpty() const;

/*
 * Method: clear
 * Usage: g.clear();
 * -----------------
 * Reinitializes the graph to be empty, freeing any heap storage.
 */

   void clear();

/*
 * Method: addNode
 * Usage: g.addNode(name);
 *        g.addNode(node);
 * ------------------------
 * Adds a node to the graph.  The first form creates the node from
 * the name.  The second form takes a node pointer created by the client.
 * Both forms return a pointer to the added node, although that value is
 * typically ignored.
 */

   NodeType *addNode(std::string name);
   NodeType *addNode(NodeType *node);

/*
 * Method: removeNode
 * Usage: g.removeNode(name);
 *        g.removeNode(node);
 * ---------------------------
 * Removes a node from the graph, where the node can be specified
 * either by its name or as a pointer value.  Removing a node also
 * removes all arcs that contain that node.
 */

   void removeNode(std::string name);
   void removeNode(NodeType *node);

/*
 * Method: getNode
 * Usage: NodeType *node = g.getNode(name);
 * ----------------------------------------
 * Looks up a node in the name table attached to the graph and
 * returns a pointer to that node.  If no node with the specified
 * name exists, getNode returns NULL.
 */

   NodeType *getNode(std::string name) const;

/*
 * Method: addArc
 * Usage: g.addArc(s1, s2);
 *        g.addArc(n1, n2);
 *        g.addArc(arc);
 * ----------------------
 * Adds an arc to the graph.  The endpoints of the arc can be specified
 * either as strings indicating the names of the nodes or as pointers to
 * the node structures.  All versions return a pointer to the added arc,
 * although that value is typically ignored.
```

图 18-8 （续）

```
 */

   ArcType *addArc(std::string s1, std::string s2);
   ArcType *addArc(NodeType *n1, NodeType *n2);
   ArcType *addArc(ArcType *arc);

/*
 * Method: removeArc
 * Usage: g.removeArc(s1, s2);
 *        g.removeArc(n1, n2);
 *        g.removeArc(arc);
 * ----------------------------
 * Removes an arc from the graph, where the arc can be specified in any
 * of three ways: by the names of its endpoints, by the node pointers
 * at its endpoints, or as an arc pointer.  If more than one arc
 * connects the specified endpoints, all of them are removed.
 */

   void removeArc(std::string s1, std::string s2);
   void removeArc(NodeType *n1, NodeType *n2);
   void removeArc(ArcType *arc);

/*
 * Method: isConnected
 * Usage: if (g.isConnected(s1, s2)) . . .
 *        if (g.isConnected(n1, n2)) . . .
 * -----------------------------------
 * Returns true if the graph contains an arc between the specified nodes.
 * Nodes can be specified either by name or as pointers to node objects.
 */

   bool isConnected(std::string s1, std::string s2) const;
   bool isConnected(NodeType *n1, NodeType *n2) const;

/*
 * Method: getNodeSet
 * Usage: for (NodeType *node : g.getNodeSet()) . . .
 * --------------------------------------------------
 * Returns the set of all nodes in the graph.
 */

   Set<NodeType *> & getNodeSet();

/*
 * Method: getArcSet
 * Usage: for (ArcType *arc : g.getArcSet()) . . .
 *        for (ArcType *arc : g.getArcSet(node1)) . . .
 *        for (ArcType *arc : g.getArcSet(name1)) . . .
 * ---------------------------------------------------
 * Returns the set of all arcs in the graph or, in the second and
 * third forms, the arcs that start at the specified node, which
 * can be indicated either as a pointer or by name.
 */

   Set<ArcType *> & getArcSet();
   Set<ArcType *> & getArcSet(NodeType *node);
   Set<ArcType *> & getArcSet(std::string name);

/*
 * Method: getNeighbors
 * Usage: for (NodeType *node : g.getNeighbors(node)) . . .
 *        for (NodeType *node : g.getNeighbors(name)) . . .
 * -------------------------------------------------------
 * Returns the set of nodes that are neighbors of the specified
 * node, which can be indicated either as a pointer or by name.
 */

   Set<NodeType *> getNeighbors(NodeType *node);
   Set<NodeType *> getNeighbors(std::string node);

/*
 * Methods: copy constructor and assignment operator
 * -------------------------------------------------
 * These methods implement deep copying for graphs.
 */
```

图 18-8 （续）

```
    Graph(const Graph & src);
    const Graph & operator=(const Graph & src);

  类的私有部分。

};

  成员实现。

#endif
```

图 18-8 （续）

Graph 类的私有部分需要节点集合、弧集合，以及一种能将节点名映射到相应节点的数据结构。该类中的私有部分内容如图 18-9 所示。Graph 类的实现比其他集合类实现更简单一些，因为它将很多的复杂性都转移给了基于图的 Set 类和 Map 类。Graph 类的代码如图 18-10 所示。

```
/*
 * Notes on the representation
 * ---------------------------
 * The Graph class is built as a layered abstraction on top of the Set
 * and Map classes.  Most of the complexity appears in the underlying
 * implementations.
 */

private:

/* Instance variables */

   Set<NodeType *> nodes;                  /* The set of nodes in the graph */
   Set<ArcType *> arcs;                    /* The set of arcs in the graph  */
   Map<std::string,NodeType *> nodeMap;    /* A map from names and nodes    */

/* Private methods */

   void deepCopy(const Graph & src);
   NodeType *getExistingNode(std::string name) const;
```

793
⋮
797

图 18-9 Graph 类的私有部分

```
/*
 * Implementation notes: Graph constructor and destructor
 * ------------------------------------------------------
 * The only initialization required at this level is creating empty data
 * structures, which is performed automatically by the underlying classes.
 * The destructor, however, must free the individual arc and node
 * structures as well.  Calling clear is sufficient to accomplish this task.
 */

template <typename NodeType,typename ArcType>
Graph<NodeType,ArcType>::Graph() {
   /* Empty */
}

template <typename NodeType,typename ArcType>
Graph<NodeType,ArcType>::~Graph() {
   clear();
}

/*
 * Implementation notes: size, isEmpty
 * -----------------------------------
 * These methods are defined in terms of the node set, so the Graph
 * class simply forwards the requests to the Set class.
 */
```

图 18-10 Graph 类的实现

```
template <typename NodeType,typename ArcType>
int Graph<NodeType,ArcType>::size() const {
   return nodes.size();
}

template <typename NodeType,typename ArcType>
bool Graph<NodeType,ArcType>::isEmpty() const {
   return nodes.isEmpty();
}

/*
 * Implementation notes: clear
 * ---------------------------
 * The implementation of clear frees all nodes and arcs.
 */

template <typename NodeType,typename ArcType>
void Graph<NodeType,ArcType>::clear() {
   for (NodeType *node : nodes) {
      delete node;
   }
   for (ArcType *arc : arcs) {
      delete arc;
   }
   arcs.clear();
   nodes.clear();
   nodeMap.clear();
}

/*
 * Implementation notes: addNode
 * -----------------------------
 * The addNode method adds the node to the set of nodes for the graph and
 * to the map from names to nodes.
 */

template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::addNode(std::string name) {
   if (nodeMap.containsKey(name)) {
      error("addNode: Node " + name + " already exists");
   }
   NodeType *node = new NodeType();
   node->name = name;
   return addNode(node);
}

template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::addNode(NodeType *node) {
   nodes.add(node);
   nodeMap[node->name] = node;
   return node;
}

/*
 * Implementation notes: removeNode
 * --------------------------------
 * The removeNode method removes the specified node but must also
 * remove any arcs in the graph containing the node.  To avoid
 * changing the node set during iteration, this implementation
 * creates a vector of arcs that require deletion.
 */

template <typename NodeType,typename ArcType>
void Graph<NodeType,ArcType>::removeNode(std::string name) {
   removeNode(getExistingNode(name));
}

template <typename NodeType,typename ArcType>
void Graph<NodeType,ArcType>::removeNode(NodeType *node) {
   Vector<ArcType *> toRemove;
   for (ArcType *arc : arcs) {
      if (arc->start == node || arc->finish == node) {
         toRemove.add(arc);
      }
   }
```

图 18-10 (续)

```
   for (ArcType *arc : toRemove) {
      removeArc(arc);
   }
   nodes.remove(node);
}

/*
 * Implementation notes: getNode, getExistingNode
 * -----------------------------------------------
 * The getNode method simply looks up the name in the map, which correctly
 * returns NULL if the name is not found.  Other methods in the
 * implementation call the private method getExistingNode instead,
 * which checks for a NULL value and signals an error.
 */

template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::getNode(std::string name) const {
   return nodeMap.get(name);
}

template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::getExistingNode(std::string name) const {
   NodeType *node = nodeMap.get(name);
   if (node == NULL) error("No node named " + name);
   return node;
}

/*
 * Implementation notes: addArc
 * ----------------------------
 * The addArc method appears in three forms, as described in the interface.
 */

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(std::string s1, std::string s2) {
   return addArc(getExistingNode(s1), getExistingNode(s2));
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(NodeType *n1, NodeType *n2) {
   ArcType *arc = new ArcType();
   arc->start = n1;
   arc->finish = n2;
   return addArc(arc);
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(ArcType *arc) {
   arc->start->arcs.add(arc);
   arcs.add(arc);
   return arc;
}

/*
 * Implementation notes: removeArc
 * -------------------------------
 * These methods remove arcs from the graph, which is ordinarily a simple
 * matter of removing the arc from two sets: the set of arcs in the graph
 * as a whole and the set of arcs in the starting node.  The methods that
 * remove an arc specified by its endpoints, however, must take account of
 * the possibility that there is more than one arc and remove all of them.
 */

template <typename NodeType,typename ArcType>
void Graph<NodeType,ArcType>::removeArc(std::string s1, std::string s2) {
   removeArc(getExistingNode(s1), getExistingNode(s2));
}

template <typename NodeType,typename ArcType>
void Graph<NodeType,ArcType>::removeArc(NodeType *n1, NodeType *n2) {
   Vector<ArcType *> toRemove;
   for (ArcType *arc : arcs) {
      if (arc->start == n1 && arc->finish == n2) {
         toRemove.add(arc);
```

图 18-10 (续)

```
        }
    }
    for (ArcType *arc : toRemove) {
        removeArc(arc);
    }
}

template <typename NodeType,typename ArcType>
void Graph<NodeType,ArcType>::removeArc(ArcType *arc) {
    arc->start->arcs.remove(arc);
    arcs.remove(arc);
}

/*
 * Implementation notes: isConnected
 * ---------------------------------
 * Node n1 is connected to n2 if any of the arcs leaving n1 finish at n2
 */

template <typename NodeType,typename ArcType>
bool Graph<NodeType,ArcType>::isConnected(std::string s1,
                                          std::string s2) const {
    return isConnected(getExistingNode(s1), getExistingNode(s2));
}

template <typename NodeType,typename ArcType>
bool Graph<NodeType,ArcType>::isConnected(NodeType *n1, NodeType *n2) const
    for (ArcType *arc : n1->arcs) {
        if (arc->finish == n2) return true;
    }
    return false;
}

/*
 * Implementation notes: getNodeSet, getArcSet
 * --------------------------------------------
 * These methods simply return the set requested by the client.  For
 * efficiency, the sets are returned by reference, because doing so
 * eliminates the need to copy the set.
 */

template <typename NodeType,typename ArcType>
Set<NodeType *> & Graph<NodeType,ArcType>::getNodeSet() {
    return nodes;
}

template <typename NodeType,typename ArcType>
Set<ArcType *> & Graph<NodeType,ArcType>::getArcSet() {
    return arcs;
}

template <typename NodeType,typename ArcType>
Set<ArcType *> & Graph<NodeType,ArcType>::getArcSet(NodeType *node) {
    return node->arcs;
}

template <typename NodeType,typename ArcType>
Set<ArcType *> & Graph<NodeType,ArcType>::getArcSet(std::string name) {
    return getArcSet(getExistingNode(name));
}

/*
 * Implementation notes: getNeighbors
 * ----------------------------------
 * This implementation recomputes the set each time, which is reasonably
 * efficient if the degree of the node is small.
 */

template <typename NodeType,typename ArcType>
Set<NodeType *> Graph<NodeType,ArcType>::getNeighbors(NodeType *node) {
    Set<NodeType *> nodes;
    for (ArcType *arc : node->arcs) {
        nodes.add(arc->finish);
    }
    return nodes;
```

图 18-10 （续）

```
}

template <typename NodeType,typename ArcType>
Set<NodeType *> Graph<NodeType,ArcType>::getNeighbors(std::string name) {
   return getNeighbors(getExistingNode(name));
}

/*
 * Implementation notes: copy constructor and assignment operator
 * ---------------------------------------------------------------
 * These methods ensure that copying a graph creates an entirely new
 * parallel structure of nodes and arcs.
 */

template <typename NodeType,typename ArcType>
const Graph<NodeType,ArcType> &
      Graph<NodeType,ArcType>::operator=(const Graph & src) {
   if (this != &src) {
      clear();
      deepCopy(src);
   }
   return *this;
}

template <typename NodeType,typename ArcType>
Graph<NodeType,ArcType>::Graph(const Graph & src) {
   deepCopy(src);
}

/*
 * Private method: deepCopy
 * ------------------------
 * This method reallocates all the nodes and arcs to ensure that the
 * structures are disjoint.
 */

template <typename NodeType,typename ArcType>
void Graph<NodeType,ArcType>::deepCopy(const Graph & other) {
   for (NodeType *oldNode : other.nodes) {
      NodeType *newNode = new NodeType();
      *newNode = *oldNode;
      newNode->arcs.clear();
      addNode(newNode);
   }
   for (ArcType *oldArc : other.arcs) {
      ArcType *newArc = new ArcType();
      *newArc = *oldArc;
      newArc->start = getExistingNode(oldArc->start->name);
      newArc->finish = getExistingNode(oldArc->finish->name);
      addArc(newArc);
   }
}
```

798 ~ 801

图 18-10 （续）

18.6 寻找最短路径

很多重要的商业应用都会涉及图，因此，人们花费了巨大的努力来寻找图相关问题的高效算法。在这些问题中，一个特别有意思的问题就是在图中寻找一条路径，使得按照某种标准评估时其路径上的两个节点之间有最小的代价。这个标准不一定是经济学上的。尽管对于特定的应用而言，你可能对寻找两个节点之间最低的化费路径感兴趣。你可以使用同样的算法找到一个最短距离的路径、最少的跳跃次数或者最短的遍历时间。

作为一个具体的实例，假设你想找到 San Francisco 与 Boston 之间总距离最短的路径，可以用图 18-2 在弧上所示的里程数值计算。它应该经过 Portland 和 Seattle，还是经过 Dallas，Atlanta 和 New York 呢？或是可能还存在一些不明显但确实距离更短的路径呢？

对于像航线图一样简单的路线图，通过沿着可能的路径把所有弧的距离相加就能很容易

途计算出结果。然而，随着图的增大，这种方法就不可行了。通常，图中两个节点之间的路径数是以指数级增长的，这也意味着探索所有路径方法的时间复杂度为 $O(2^n)$。正如第 10 章中复杂度计算所讨论的那样，指数级的问题算法是不可行的。如果你想在一个合理的时间内找出图中的最短路径，使用更高效的算法非常必要。

在图中寻找最短路径最常用的算法是 Edsger W. Dijkstra 于 1959 年提出的 Dijkstra 算法，它是一个**贪心算法**（greedy algorithm）的特例。贪心算法可以让你通过一系列逻辑可行的选择找出所有的解决方法。贪心算法并非对每个问题都适用，但是在解决最短路径问题上还是相当有用的。

就本质而言，Dijkstra 算法的核心就是寻找最短路径，或者更通俗地说，就是找出那些具有最小代价的弧的路径，可以对其做如下定义：从起始节点开始探索沿着路径总长度增加的方向前进，直到到达目的节点。这个路径一定是最优的，因为你已经探索了起始节点到目的节点的代价较小的所有路径。在寻找最短路径的问题中，Dijkstra 算法可以用图 18-11 中的方式实现。

```
/*
 * Function: findShortestPath
 * Usage: Vector<Arc *> path = findShortestPath(start, finish);
 * ------------------------------------------------------------
 * Finds the shortest path between the nodes start and finish using
 * Dijkstra's algorithm, which keeps track of the shortest paths in
 * a priority queue.  The function returns a vector of arcs, which is
 * empty if start and finish are the same node or if no path exists.
 */

Vector<Arc *> findShortestPath(Node *start, Node *finish) {
   Vector<Arc *> path;
   PriorityQueue< Vector<Arc *> > queue;
   Map<string,double> fixed;
   while (start != finish) {
      if (!fixed.containsKey(start->name)) {
         fixed.put(start->name, getPathCost(path));
         for (Arc *arc : start->arcs) {
            if (!fixed.containsKey(arc->finish->name)) {
               path.add(arc);
               queue.enqueue(path, getPathCost(path));
               path.remove(path.size() - 1);
            }
         }
      }
      if (queue.isEmpty()) {
         path.clear();
         return path;
      }
      path = queue.dequeue();
      start = path[path.size() - 1]->finish;
   }
   return path;
}

/*
 * Function: getPathCost
 * Usage: double cost = getPathCost(path);
 * ---------------------------------------
 * Returns the total cost of the path, which is just the sum of the
 * costs of the arcs.
 */

double getPathCost(const Vector<Arc *> & path) {
   double cost = 0;
   for (Arc *arc : path) {
      cost += arc->cost;
   }
   return cost;
}
```

图 18-11　寻找最短路径的 Dijkstra 算法实现

如果你认真考虑 findShortestPath 算法所使用的数据结构，对该算法的理解就会更深入一些。在该算法实现中定义了以下三个局部变量：

- 变量 path 用来追踪最短路径，它由弧矢量构成。矢量中的第一个弧由开始节点指向第一个中间节点。每一个子路径都在前一个路径结束时开始，然后继续前进，直到它最后一个弧的终点到达目的节点。如果两个节点之间不存在路径，findShortestPath 返回空矢量来表示这一事实。
- 变量 queue 是弧的一个有序队列，因此队列中的弧是按照代价递增的顺序排列的。因此，该队列不同于传统的先进先出的规则，它是一个优先队列（priority queue）。这个队列允许用户定义每个元素的优先级。findShortestPath 的代码假定这个功能已经在 PriorityQueue 类中实现了，如第 16 章描述的那样。除了 enqueue 方法，PriorityQueue 与标准 Queue 类完全相同。enqueue 方法用其第二个参数指明优先级，其方法原型如下：

```
void enqueue(ValueType element, double priority);
```

  与传统的英语含义相同，在优先队列中，优先级小的数字会首先出现在队列中。因此优先级为 1 的元素会在优先级为 2 的元素之前。因为所有的路径都按照距离进入了优先序列，每次 dequeue 的调用都会返回队列中存在的最短路径。标准的 C++ 类库通过 pqueue.h 接口输出 PriorityQueue 类。
- 变量 fixed 是一个映射，它将每个城市名与到达那个城市的已知的最短距离相关联。当你从优先队列中删除一条路径时，你就能知道抵达该路径终节点的最短路径，除非你已经发现了最短路径，并且你知道该路径是目前已知的最短距离的路径。

findShortestPath 的操作显示在图 18-12 中，它展示了在图 18-2 的航线图中从 San Francisco 到 Boston 的最短路径的计算步骤。

806

```
确定到 San Francisco 的距离为 0
处理从 San Francisco 射出的弧的顶点的集合 (Dallas, Denver, Portland)
    将路径 San Francisco → Dallas (1468) 入队
    将路径 San Francisco → Denver (954) 入队
    将路径 San Francisco → Portland (550) 入队
将最短路径 San Francisco → Portland (550) 出队
确定到 Portland 的距离为 550
处理从 Portland 射出的弧的顶点的集合 (San Francisco, Seattle)
    忽略 San Francisco，因为 Portland 到它的距离已知
    将路径 San Francisco → Portland → Seattle(680) 入队
将最短路径 San Francisco → Portland → Seattle (680) 出队
确定到 Seattle 的距离为 680
处理从 Seattle 射出的弧的顶点的集合 (Boston, Portland)
    将路径 San Francisco → Portland → Seattle → Boston (3169) 入队
    忽略 Portland，因为 Seattle 到它的距离已知
将最短路径 San Francisco → Denver (954) 出队
确定到 Denver 的距离为 954
```

图 18-12 Dijkstra 算法的执行步骤

处理从 Denver 射出的弧的顶点的集合 (Chicago, Dallas, San Francisco)
    忽略 San Francisco, 因为 Denver 到它的距离已知
    考虑径 San Francisco → Denver → Chicago (1861) 入队
    考虑径 San Francisco → Denver → Dallas (1604) 入队
将最短路径：San Francisco → Dallas (1468) 出队
确定到 Dallas 的距离为 1468
处理从 Dallas 发出的弧的顶点的集合 (Atlanta, Denver, Los Angeles, San Francisco)
    忽略 Denver 和 San Francisco, 因为 Dallas 到它们的距离已知
    考虑径 San Francisco → Dallas → Atlanta (2193) 入队
    考虑径 San Francisco → Dallas → Los Angeles (2708) 入队
将最短路径 San Francisco → Denver → Dallas (1604) 出队
忽略 Dallas, 因为它的距离已知
将最短路径 San Francisco → Denver → Chicago (1861) 出队
确定到 Chicago 的距离为 1861
处理从 Chicago 发出的弧的顶点的集合 (Atlanta, Denver)
    忽略 Denver, 因为 Chicago 到它的距离已知
    考虑径 San Francisco → Denver → Chicago → Atlanta (2460) 入队
将最短路径 San Francisco → Dallas → Atlanta (2193) 出队
确定到 Atlanta 的距离为 2193
处理从 Atlanta 发出的弧的顶点的集合 (Chicago, Dallas, New York)
    忽略 Chicago 和 Dallas, 因为 Atlanta 到它们的距离已知
    考虑径 San Francisco → Dallas → Atlanta → New York (2949) 入队
将最短路径 San Francisco → Denver → Chicago → Atlanta (2460) 出队
忽略 Atlanta, 因为它的距离已知
将最短路径 San Francisco → Dallas → Los Angeles (2708) 出队
确定到 Los Angeles 的距离为 2708
处理从 Los Angeles 发出的弧的顶点的集合 (Dallas)
    忽略 Dallas, 因为 Los Angeles 到它的距离已知
将最短路径 San Francisco → Dallas → Atlanta → New York (2949) 出队
确定到 New York 的距离为 2949
处理从 New York 发出的弧的顶点的集合 (Atlanta, Boston)
    忽略 Atlanta, 因为 New York 到它的距离已知
    考虑径 San Francisco → Dallas → Atlanta → New York → Boston (3140) 入队
将最短路径 San Francisco → Dallas → Atlanta → New York → Boston (3140) 出队

图 18-12 （续）

807

在你阅读 Dijkstra 算法的实现时，牢记以下几点是非常有用的：

- 路径是按照距离的远近而不是按照跳跃的次数进行探索的。因此，San Francisco → Portland → Seattle 的探索会在 San Francisco → Denver 或者 San Francisco → Dallas 之前，因为它的总距离更短。
- 当一条路径出队列而不是进队列时，到达一个节点的距离是固定的。在优先队列中到达 Boston 的第一条路径为通过 Portland 和 Seattle 的路径，这不是可能的最短路径。San Francisco → Portland → Seattle → Boston 的总距离为 3169。由于最短距离仅为 3140，因此，算法执行完后，San Francisco → Portland → Seattle → Boston 这条路径仍然会在优先队列中。

- 每个节点的弧最多遍历一次。当到达某个节点的距离确定时，算法的内部循环才会执行。对于每个节点而言，这个只会发生一次。因此，最终执行内部循环的次数就是从之前节点到最后节点的节点和弧的最小个数。Dijkstra 算法的完整分析超越了本书的范围，但是它的运行时间为 $O(M \log N)$，其中，N 代表节点点数，M 的值为 N 和弧数中的最大值。

18.7 搜索网页的算法

正如本章引言部分所述，网页也是一个图，其中节点就是一个网页，而弧是可以从一个网页跳转到另一个网页的超链接。但是和你在本章所见到的图不同的是，网页图是巨大的。网络中网页的个数是数十亿的，链接的数量也是如此。

为了在大量的网页集合中找到有用的东西，很多人会使用搜索引擎来产生你可能感兴趣的网页的序列。典型的搜索引擎是遍历整个 Web 上的网页，这个过程称为**爬取**（crawling），它可以通过多台计算机并行执行，然后通过这些信息创建一个索引确定哪个网页包含一个特定的单词或短语。然而，鉴于网页的规模，单靠索引是不够的。除非查询关键词是非常准确的，否则包含查询关键词的网页数量是惊人的。因此，搜索引擎必须对搜索结果进行排序，以使得分较高的网页出现在查询结果的较前面。设计一个有效的搜索引擎最大的挑战就是设
SD8 计一个可以对每个网页的重要性进行评估打分的算法。

18.7.1 谷歌的网页排名算法

对网页进行排序的最著名的算法就是谷歌的**网页排名算法**（PageRank algorithm），它基于 Web 网页的链接结构而将整个 Web 看成为一张图，每个网页都被赋予一个分值以反映它在图中的重要性。尽管它的名称暗示了对网页排序的思法，PageRank 算法其实是以拉里·佩奇命名的，他和谷歌的共同创始人谢尔盖·布林一起设计了这个算法，并且两人都毕业于斯坦福大学。

从某种程度上说，PageRank 算法背后的思想很简单，如果其他的网页与某个网页相链接时，这个网页就会变得更重要。因此，依据网页间的相互链接，Web 中的每个网页都得到一个分值。然而，所有的链接并不处于同一层次。从一个较为权威的网页来的链接的权值要高于非权威网页的链接值。这形成了对网页重要性更具体的刻画：如果一个网页和重要的网页间有链接，则该网页变得更重要。

网页的重要性会随着和它链接的网页的重要性而变化，这一事实意味着网页的得分会随着其他网页得分的波动而上下起伏。因此，PageRank 算法按照一系列的逐次逼近进行网页得分的计算。首先，所有的网页都赋予相同的权值。在算法后期的迭代中，每个网页的得分会随着和它链接的网页的得分而进行调整。最后，这个过程就会达到一个平衡点，此时每个网页都会按网络的链接结构计算出反映它的重要性的得分。

另外一个描述 PageRank 算法效果的方法为：每个网页的最后得分代表随机链接到达那个网页的可能性。采用随机选择而不考虑之前的决策结果的过程被称为马尔可夫过程（Markov processes），它以俄国数学家安德烈·马尔可夫命名的。马尔可夫（1856~1922）是率先分析这种过程的数学性质的数学家之一。

18.7.2 网页排名算法的一个简例

鉴于真实的网络太大而不能作为一个有效的示例，因此，用一个较小的例子开始是很合理的。图 18-13 所示的图就代表了由五个网页构成的一个简单网络，图中的网页用 A、B、C、D 和 E 标记。例如，A 网页和其他网页都有链接，而网页 B 仅仅和网页 E 有链接。

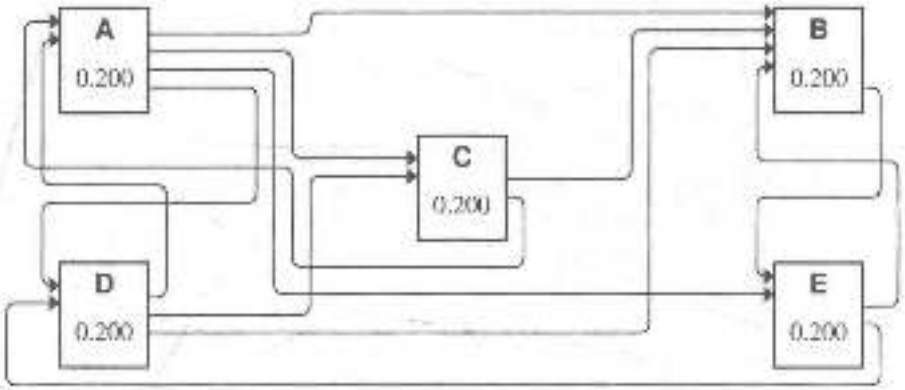


图 18-13 具有等概率的初始的五个网页的 Web 图

PageRank 算法的第一步就是给每个网页赋一个初始的分值，它是一个简单的概率值，代表在整个网页中随机地选择某个网页的可能性。例如，在本例中有五个网页，因此每个特定网页被随机选中的概率均为五分之一。用数学方式来描述就是 0.2，这个概率值出现在每个网页名的底部。

在算法的每次迭代中，PageRank 算法都会通过计算用户从先前循环按照随机链接更新给各个网页所赋的可能性值。例如，你恰好在 A 节点上，你就可以选择访问其他任意四个节点，因为 A 和它们都有链接。如果你随机选择一个链接，则你去 B 网页的概率为四分之一，去 C 也为四分之一，去 D 和 E 也是四分之一。然而，如果你在节点 B 呢？由于节点 B 只有一个链接，任何在 B 网页中的用户都会无一例外的到达 E 网页。

你可以使用这种计算方法确定从某个特定节点开始到达其他任意节点的概率。例如，有两种通过链接到 A 网页的方法。你可以从 C 网页开始，然后在 C 网页的两个链接中选择那个返回 A 节点的链接。此外，你也可以从 D 节点开始，但这样的话只有你足够幸运才能从 D 的三个链接中选出到 A 的链接，而不是其他节点的链接。如果你将这种计算用公式表示，用有标示的字母表示到下一个节点的可能性，A 节点的计算公式如下：

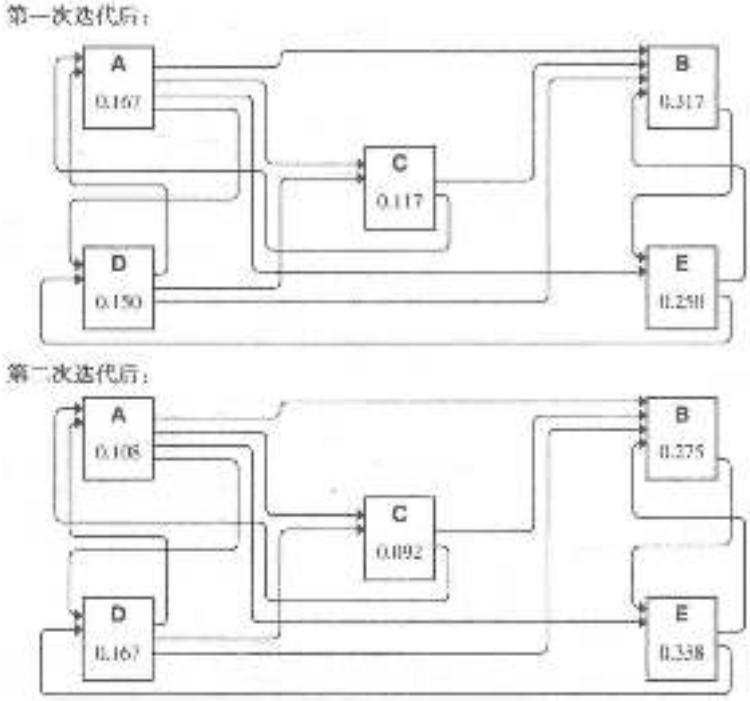
809
↓
810

$$A' = \frac{1}{2}C + \frac{1}{3}D$$

采用相同的分析，可以写出以下其余节点的得分计算公式：

$$B' = \frac{1}{4}A + \frac{1}{2}C + \frac{1}{3}D + \frac{1}{2}E$$
$$C' = \frac{1}{4}A + \frac{1}{3}D$$
$$D' = \frac{1}{4}A + \frac{1}{2}E$$
$$E' = \frac{1}{4}A + B$$

算法的每一次迭代都会用公式计算出的 A'、B'、C'、D'、E' 来代替 A、B、C、D、E。网页排序算法执行了两次迭代的结果如图 18-14 所示。



811

图 18-14 PageRank 算法头两次迭代后各个节点的概率

现实生活中，马尔可夫过程最有趣的就是在适当的迭代后结果会趋于稳定。图 18-15 展示了经过 16 次迭代之后五个节点的概率。此时概率的前三个小数位不会再发生变化。因此，那些值就能表示随机浏览时到达该特定网页的可能性，这也是 PageRank 算法的本质。

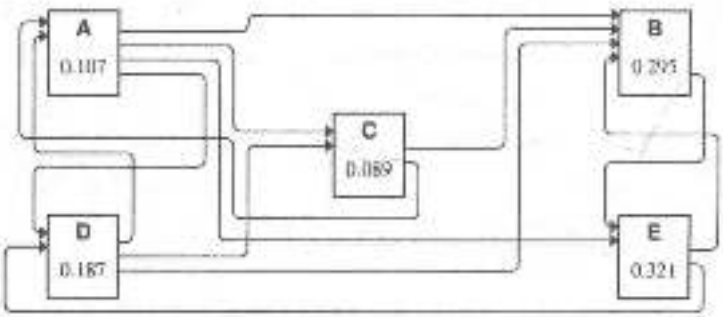


图 18-15 达到稳定的各个节点的概率

本章小结

本章向你介绍了图的概念，它被定义为由一系列弧相连的节点的集合。与集合一样，图不仅是一种重要的抽象理论，也是一个解决很多应用领域中出现的实际问题的工具。例如，图算法在研究因特网到大型的交流系统这些相互联系结构的性质问题时特别有用。

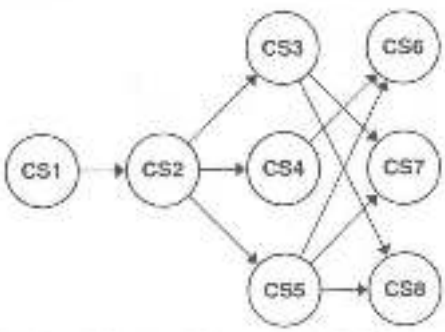
本章的重点包括：

- 图可以是有向的或无向的。有向图的弧只能有一个方向，因此，若存在 $n_1 \rightarrow n_2$ 这条弧，并不能说明也存在 $n_2 \rightarrow n_1$ 这条弧。你可以用有向图来表示无向图，即在有向图中用两个反向弧来表示一对节点之间的双向关系。

- 你可以采取任一种策略来表示图中的关系。一个常用的方法就是构建一个邻接链表，此时，每个节点的数据结构是一个相互连接的节点的列表。你也可以使用一个邻接矩阵，它将节点间的联系存入到一个二维布尔数组中。其中，矩阵的行或列表示图中的各节点；若在图中两个节点相连，则矩阵中节点对应的行与列的值为 true。 812
- 采用集合分层的思想，graph.h 接口可以被较容易地实现。虽然我们可能用完全低层的基于结构的方式，或者高层的完全面向对象的方式来定义这个接口，但是我们最好采用一种折中的方法来定义 Graph 类，但应将 Graph 类中所使用的节点和弧的结构留给用户去定义。
- 图最重要的两个遍历顺序是深度优先搜索和广度优先搜索。深度优先算法从起始节点选择弧，然后从这个弧开始逐个地对所有路径进行遍历操作，遍历操作持续进行直到没有其他的节点。只有到这时，算法才会返回到起始节点，再对其他弧进行遍历操作。广度优先算法是按照节点与起始节点的距离的顺序来进行遍历，这个距离是通过计算最短路径中弧的数量来确定的。对初始节点进行处理后，广度优先搜索在跳到下两个节点之前将会对其所有的邻接节点进行处理。
- 采用 Dijkstra 算法，你可以找到图中两个节点之间的最短路径。这个算法比起比较所有可能路径的长短的指数级策略其效率要高得多。Dijksta 算法是一类采用贪心算法的实例之一，贪心算法的思想是在任何决策点选择局部最优。

复习题

1. 图是什么？
2. 判断题：树是图的子集。
3. 有向图和无向图的区别是什么？
4. 假如你正在使用某个仅支持有向图的图形包，你怎样将无向图表示出来？
5. 给出下列术语在图中的定义：路径、回路、简单路径、简单回路。
6. 相邻节点和度的关系是什么？
7. 强连接图和弱连接图的区别是什么？ 813
8. 判断题：强连接和无向图没有实际相关性，因为所有这样的图都是只要有连接就自然是强连接的。
9. 数学中典型的代表节点和弧的术语是什么？
10. 假设某大学开设了八门计算机课，这些课程的前趋后继关系如下图所示：



采用本章所描述的图的数学形式，将该图定义为一对节点的集合。

11. 用一个邻接链表图来表示复习题 10 中的图。
12. 复习题 10 中的图，其对应的邻接矩阵中的内容是什么？
13. 稀疏图和稠密图的区别是什么？

14. 假如在一个特定的应用中，要求选择图的底层表示，是什么因素决定你用邻接表还是邻接矩阵来实现这个需求？
15. 为什么为一个图形包实现一个用单独的迭代器是不必要的？
16. 在 graph.h 接口的各个版本中，为什么使用的集合都使用指向节点或者弧的指针作为它的元素类型？
814 17. 图的两种基本的遍历操作是什么？
18. 写出图 18-1 中的飞机航线图从 Atlanta 开始的深度优先遍历和广度优先遍历的结果。假设节点和弧的顺序和字典顺序相同。
19. 本章把哪个问题列成了包含类中 graph.h 接口的 Node 和 Arc 定义的最重要的问题？
20. graph.h 接口在定义用于表示节点和弧用户自定义类型时需遵循什么法则？
21. 什么是贪心算法？
22. 解释 Dijkstra 算法寻找最短路径的操作流程。
23. 给出对图 18-11 执行 Dijkstra 算法时，每一执行步骤所对应的优先队列中的值。
24. 把图 18-12 作为一个模型，追踪 Dijkstra 算法的执行过程，找出 Portland 到 Atlanta 的最短路径。

习题

1. 用低层的、基于结构的 graph.h 接口版本设计并实现函数：

```
void readGraph(SimpleGraph & g, istream & infile);
```

函数从 infile 读取一个描述图的文本到用户传入的图 g 中。已经打开的输入流中的内容可以由以下三种形式中的任一种组成：

x	定义一个名为 *x* 的节点
x -- *y*	定义一个双向弧 *x* <-> *y*
x -> *y*	定义有向弧 *x* → *y*

x 和 *y* 的名称是任意不包含连接符的字符串。上述任意两种连接格式也要允许用户在行末用中括号包围数字指定弧数。如果括号里没有值，弧数默认为 1。图和文件的定义用一个空白行结束。

当在数据文件中出现新名字时就定义一个新节点。这样，如果所有的节点都和其他节点相连，
815 在数据文件中仅仅包含弧就足够了，因为定义弧自然就定义了节点的末端。如果你需要表示一个包含孤立节点的图，必须注明这些节点在数据文件的不同行里的名字。

当读取弧的描述时，你的执行程序应该去除节点名中的首部空格，但要保留节点名中间的空格。以下这行文本：

```
San Francisco -- Denver (954)
```

因此，它定义了节点名为 "San Francisco" 和 "Denver"，然后在两个节点之间建立双向连接，并将两个弧长被初始化为具有代价 954。

作为例子，对下列数据文件调用 readGraph 会产生在本章中图 18-2 所示的航线图：

AirlineGraph.txt

```
Atlanta -- Chicago (599)
Atlanta -- Dallas (725)
Atlanta -- New York (756)
Boston -- New York (191)
Boston -- Seattle (2489)
Chicago -- Denver (907)
Dallas -- Denver (650)
Dallas -- Los Angeles (1240)
Dallas -- San Francisco (1468)
Denver -- San Francisco (954)
Portland -- Seattle (130)
Portland -- San Francisco (550)
```

2. 实现一个函数，使它和上一题中的 readGraph 函数的功能一样：

```
void writeGraph(SimpleGraph & g, ostream & outfile);
```

这个函数要求输出指定的输出文件的图的文本说明。你可以假设图中每个节点中的数据区包含节点名，就如同 readGraph 生成了图。writeGraph 函数的输出结果必须在 readGraph 中可读。

3. 用一个栈来存储未遍历的节点的方法替代在图 18-5 中的深度优先搜索的递归实现。算法开始时，你仅需把开始节点压入栈中。然后，重复下列操作直到栈为空：

   1. 弹出栈顶元素。
   2. 访问节点。
   3. 把邻接节点压入栈。 816

4. 把你上一题中的栈改成用队列表示。描述结果代码实现的遍历顺序。

5. 本章给出的 depthFirstSearch 和 breadthFirstSearch 遍历函数是用来强调隐含的算法结构。如果你想要兼插这些图的遍历策略，必须重新实现函数使其不再依赖于用户自定义的 visit 函数。一种方法是通过给 graph.h 加入下列类方法以实现上述两种算法：

```
void mapDFS(void (*fn)(NodeType *), NodeType *start);
void mapBFS(void (*fn)(NodeType *), NodeType *start);
```

在上述任一种遍历顺序中，方法要为每个可达节点从 start 指针调用 fn(node)。

6. 本章给出的广度优先搜索的实现算法得到了正确的遍历，但在队列中产生了冗余的遍历路径。问题在于：即使结点已经被访问，代码依然将新路径加到队列中。这意味着一旦这条路径从队列中移除，它将很容易被忽略。你可以在加入队列之前通过检测终节点是否被访问来修复这个问题。

   编写一个检测程序来评估使用和没有使用这个检测对于函数实现的相对效率。你的检测程序应该能够在数个大图中读取，这些大图的平均度数不同，并且这些算法都可在任一个图中的随机一个节点开始运行。你的检测程序还要能够在算法执行中记录平均队列长度，以及访问其每一个节点所需要的总的运行时间。

7. 编写一个函数：

```
bool pathExists(Node *n1, Node *n2);
```

如果图中的节点 n1 和 n2 中有路径，则返回 true。实现采用深度优先搜索从 n1 节点开始遍历图的函数；如果途中遇到 n2，则路径存在。采用广度优先搜索算法重新实现该函数。在大图中，那种实现方式可能更高效？

8. 编写一个函数：

```
int hopCount(Node *n1, Node *n2);
```
817

函数返回节点 n1 和 n2 之间最短路径的跳跃次数。如果 n1 和 n2 相等，hopCount 返回值为 0；如果不存在路径，hopCount 返回值为 -1。这个函数易于用广度优先搜索实现。

9. 通过编写文件 graphpriv.h 和 graphimpl.cpp，完成图 18-7 中的 Graph 类的实现。

10. 定义并实现 graphio.h 接口，使之输出习题 1 和习题 2 中的 readGraph 和 writeGraph 方法，通过修改 Graph 类模板版本完成。

11. 尽管本章包含了 Dijkstra 算法的实现，然而，缺乏环境基础使该算法能得以应用。编写 C++ 程序来创建一个有实际应用的 Dijkstra 算法，它执行以下操作：

    - 从图中读取文件。
    - 允许用户输入两个城市的名称
    - 用 Dijkstra 算法找到并输出最短路径。

12. 数个重要的图算法可作用于一类特殊的图中，图中的节点可以这种方式被划分为两个集合，使得所有的弧和不同集合中的节点相连，同一集合中没有节点相连。这种图被称为二部图（bipartite）。编

写一个函数模板：

```
template <NodeType,ArcType>
bool isBipartite(Graph<NodeType,ArcType> & g);
```

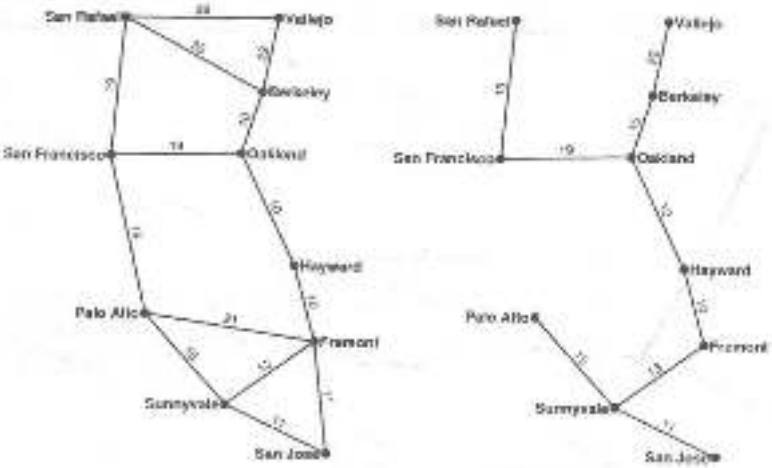
函数读取任意一个图，如果具有二部图性质，则返回 true。

13. 尽管 Dijkstra 算法对于寻找最小代价路径具有极大实际意义，但还有其他一些图算法也具有相当大的商业价值。在很多情况下，寻找特定两个节点间的最小代价路径并不如最小化整个网络的代价更重要。

例如，假设你在一家新建的电缆系统的公司里工作，它连接着 10 个大城市。初步研究为你提供了沿着不同的可能路径架设新电缆系统的预算。这些路径及其成本如图 18-16 中的左图所示。你
818 的任务是找到最经济的架设新电缆系统的方案，使得所有城市都能够通过路径相连。

为了节约成本，要避免把架设的电缆在图中构成一个循环。这样的电缆是不必要的，因为城市已经可以通过其他的路径相连。如果你的目标是找出图中节点之间的最小代价路径的一组弧，你可能会避免这样的弧。剩下的没有循环的图形成了树。一个连接图中所有节点的树被称为生成树（spaning tree）。生成树相连弧的总路径最短的树称为最小生成树（minimum spaning tree）。上一题提及的电缆网络问题因此等同于找出图的最小生成树。最小生成树在图 18-16 中的右边。

文献中已有许多寻找最小生成树的算法。其中最简单的一个方法是由约瑟夫·克鲁斯卡（Joseph Kruskal）于 1956 年设计的。在 Kruskal 算法中，你要把图中的弧按照递增次序进行排序。如果节点末端的弧是未连接的，这部分弧就成为生成树的一部分。然而，如果这些节点已经通过路径相连，则可以完全忽略弧。构建图 18-16 中的最小生成树的步骤在下列示例中展示：



819 图 18-16 图和它的最小生成树

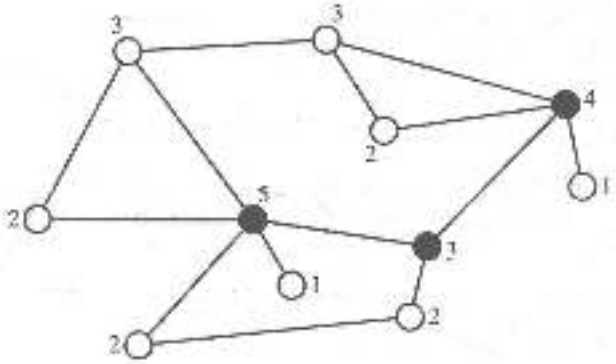


编写一个函数：

```
Graph<Node,Arc>
  findMinimumSpanningTree(Graph<Node,Arc> & g);
```

实现 Kruskal 算法，找出最小生成树。函数要返回一个新图，它的节点按照原图相连，但仅包含最小生成树部分的弧。

14. 图的支配集（dominating set）是由节点与其相邻节点组成全图节点的集合的子集。也就是说，图中所有的节点都是在控制集中，或是支配集中的节点的邻节点。在下面的图中，每一个节点都标注了它们的邻节点数量。以便于追踪算法——有内容的节点组成图的支配集。其他的支配集也可以。



820

理想状态下，你很可能找到尽量小的支配集。但是，众所周知，这是一个计算难度极大的任务——对大多数图来说计算成本太高。下列算法即使不能总是输出最理想的结果，通常也能够找到相对小的支配集：

1）从空集 S 开始。

2）将图中的节点按照度数递减顺序排序。换句话说，你想要从相邻节点最多的节点开始，然后按照相邻节点递减的次序遍历。如果两个或更多的节点具有相同的度数，你可以按任意次序遍历。

3）如果你在第 2 步选的节点不是冗余节点，将其并入 S。当一个节点自身以及它的所有相邻节点已经在 S 里了，则称该节点是冗余节点。

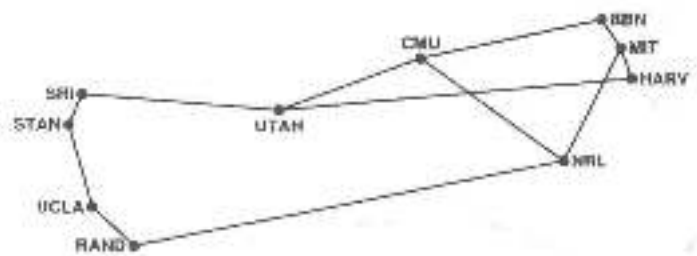
4）继续操作直到 S 支配整个图。

编写一个函数模板：

```
template <NodeType,ArcType>
Set<NodeType *>
   findDominatingSet(Graph<NodeType,ArcType> & g);
```

用这个算法找到图 g 的小支配集。

15. 图算法经常适合于分布式处理，其中，对图中的每个节点进行处理。尤其是这种算法通常用来寻找计算机网络中最优传输路线。举例说明，下面的图展示了 ARPANET 中的前 10 个节点，网络由美国国防部，即今天复杂的因特网的先驱 ARPA 创建：



ARPANET 早期的节点由被称为接口通信处理机（interface message processor，IMP）的小型

821 计算机组成。作为网络操作的一部分，每个 IMP 都会给它的相邻节点发信息，用来指出从那个节点到其他任意节点的跳跃次数，以及 IMP 拥有那个信息的程度。通过监控信息的传入，每个 IMP 会迅速从整个网络找出有效的路径信息。

为了使想法更加具体，设想一下每个 IMP 维持一个数组，这个数组的每个索引位置对应着一个节点。当整个网络运行时，Stanford IMP（STAN）中的数组包含以下内容：

4	3	3	4	3	2	1	0	1	2
BBN	CMU	HARV	MIT	NRL	RAND	SRI	STAN	UCLA	UTAH

然而有趣的问题是，数组包含的内容并不太重要，更重要的是网络如何运行和维护。当一个节点重启时，它根本不懂得完整的网络是什么。实际上，Stanford 节点所能自己决定的仅有信息是它的入口为 0 跳跃。因此，在启动时，STAN 节点中的数组看起来如下所示：

?	?	?	?	?	?	?	0	?	?
BBN	CMU	HARV	MIT	NRL	RAND	SRI	STAN	UCLA	UTAH

路由算法通过让每个节点沿着自己的相邻节点向前推进。举个例子，Stanford IMP 把其数组传送至 SRI 和 UCLA。它也从相邻节点收到类似的信息。如果在 UCLA 的 IMP 刚刚启动，它可能传送包含以下数组的信息：

?	?	?	?	?	?	?	?	0	?
BBN	CMU	HARV	MIT	NRL	RAND	SRI	STAN	UCLA	UTAH

这条信息为 Stanford 节点提供了一些有趣的信息。如果它的相邻节点能够以 0 跳跃到达 UCLA，那么 Stanford 节点就能在 1 以内到达。结果，Stanford 节点会校正其路径数组，如下所示：

?	?	?	?	?	?	?	0	1	?
BBN	CMU	HARV	MIT	NRL	RAND	SRI	STAN	UCLA	UTAH

总之，无论何时任意节点从其相邻节点得到路径数组，除非它的入口已经是最小的，否则需要检查每一个即将到来的数组的已知入口，用已知值加一代替对应的自己的数组。在非常短的时间内，贯穿整个网络的路径数组会得到正确的信息。

822 编写一个程序，用图来模拟这个网络中节点的路由算法的计算过程。

第 19 章
Programming Abstractions in C++

继　承

请当心不要玩弄你宝贵的遗产。

——亨利·卡伯特·洛奇，“国家联盟”，1919

通过第 4 章对流类层次结构的介绍，你已经知道了类似 C++ 和 Java 这样的面向对象语言的一个特性：允许你在类之间定义继承关系。如果你的类提供了某些你所需要且可以应用于其他特定场合的功能，那么你可以考虑定义一个从该类派生的新子类。该子类以某种形式特化了父类的行为。每一个子类继承了其父类的行为，而这些父类的行为也是从它们自身的父类继承而来。虽然你已经在流类库中接触过继承，但是并没有将这一特性应用于你自己定义的类。本章我们将引导你定义类层次，让继承在这些类的联系中扮演重要作用。

同时，你必须认识到：在 C++ 语言中使用继承特性比在其他语言中更容易产生问题。特别是在你已经具备 Java 语言编程经验的情况下，你将受先前学习的继承概念影响，但实际上这些概念并不完全适合于 C++ 语言。因此学习 C++ 语言中对继承特性使用的约束与学习使用该特性所带来的优势同样重要。

19.1　简单的继承

在考虑复杂的继承应用之前，我们先通过几个简单的例子来了解继承特性。为了了解继承的最基本形式，先来看 C++ 语言中对于子类的定义：

```
class subclass : public superclass {
   new entries for the subclass
};
```

在上述定义模式中，子类 subclass 继承了父类 superclass 中所有的公有成员。父类中的私有部分仍保持了其私有特性。因此，子类不能直接访问父类中的私有方法和私有实例变量。

19.1.1　指定模板类中的类型

在 C++ 语言中，我们可以创建一个除了类头之外不包含任何其他代码的实用子类，尤其在父类是一个模板类的情况下。例如，你可以像下面的代码一样定义一个 StringMap 类来映射字符串对：

```
class StringMap : public Map<string,string> { };
```

使用简单的类名 StringMap 可以让程序变得更加短小和易读，因为在使用类名的时候不需要写出模板参数。

指定模板参数类型的优势在涉及复杂类型时将变得更加明显。特别是当应用程序已经定义了自己的 City 类和 Flight 类时，你可能想定义一个新的 Graph 子类来表示第 18 章中的飞机航线图。下面的定义创建了一个新的 AirlineGraph 类，该类将节点和弧的类型分别指定为 City 和 Flight：

```
class AirlineGraph : public Graph<City,Flight> { };
```

19.1.2 定义 Employee 类

假设你的当前任务是为公司设计一个面向对象的工资结算系统。在起始阶段，你可能需要设计一个名为 Employee 的类，该类封装了一名雇员的信息和工资结算系统运行所需的部分方法。这些操作可能包括类似 getName 这样的用来返回员工姓名的简单方法，还有类似 getPay 这样根据每一个 Employee 对象中存储的数据来计算员工工资的复杂方法。然而，在许多公司中，员工可能被分为不同的类别，它们在某些方面是类似的，但却各不相同。例如，一个公司可能有钟点工、计件工和薪水工，它们可能会使用同一个工资结算系统。在这样的公司里，像图 19-1 中 UML 图所示的为每种类别的员工定义一个子类将会非常有用。

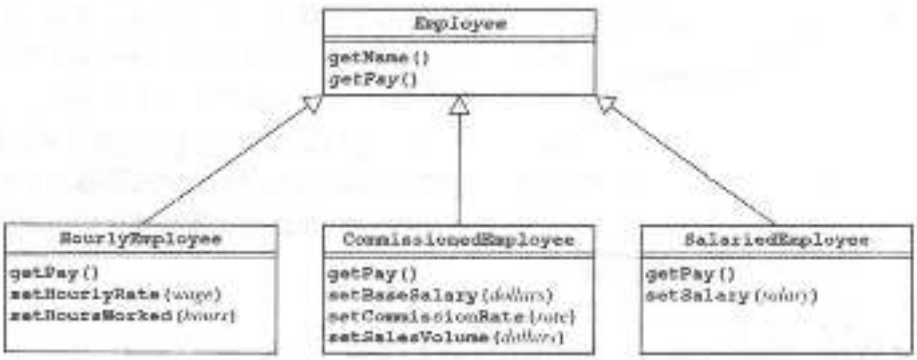


图 19-1 一个简单的 Employee 类层次

该继承层次的根是 Employee 类，该类定义了所有员工的通用方法。Employee 类提
825 供了类似 getName 这样的方法，它被其他类简单地继承。毕竟，所有的员工都有自己的名字。另一方面，为每一个子类编写自己的 getPay 方法是非常重要的，因为每一类员工的工资结算方法不同。小时工的最后工资取决于每小时的工资数和员工工作的总时间。计件工的最后工资取决于底薪加上员工负责的委托销售额的佣金。同时，必须很清楚地意识到，即每个子类的 getPay 方法实现各不相同，每一类员工也都有自己的 getPay 方法。所以我们必须在 Employee 类中定义这个方法，并在子类中重置（override）该方法定义。

如果你仔细观察图 19-1 的版式，会发现 Employee 类和该类的 getPay 方法都以斜体书写。在 UML 图中，斜体字用来说明一个类或者一个方法是抽象的（abstract），这表明在该继承层次中仅提供了那些将出现子类中的方法的规格说明。例如，我们知道不存在基本的 Employee 类型的对象。每一个 Employee 类型的对象必须属于其子类 HourlyEmployee、CommissionedEmployee，或者 SalariedEmployee。每个子类对象仍然是 Employee 类型，因此这些对象在继承了 getName 方法的同时也继承了虚方法 getPay 的原型。

图 19-2 向我们展示了基于 Employee 类层次的一组类的定义框架。正如你在本书中看到的 C++ 类接口，Employee 类与其子类的实现细节被组织在 .cpp 文件中。即使考虑到了上述事实，由于它们缺少类的私有部分的内容，同时缺少构造该类对象的构造函数和类接口所必需的相关注释，因此图 19-2 中的类定义依旧显得过于残缺而不利于实际应用。当然，

该图的作用只是向我们说明 C++ 语言中对子类的应用，为此目标，该示例肯定已经足够了。

构成整个继承层次的根 Employee 类的定义，与之前其他类定义的结构是相似的。Employee 类的公有部分声明了两个方法：一个是返回 string 类型的 getName 方法；另一个是返回 double 类型的 getPay 方法。getName 方法的原型根据你的期望定义。但是抽象方法 getPay 的原型声明却略有不同：

```
virtual double getPay() = 0;
```

这一声明引入了 C++ 语言的两个新特性。该声明与以往相比，第一个不同点是该方法原型由关键字 virtual 开头，这一关键字向编译器表明该方法的实际代码由其可构建对象的子类提供。第二个不同点是原型以 =0 结尾。C++ 语言使用这一语法来标记这一方法为纯虚方法（pure virtual method），它在基类中没有定义，因此，其实现只能由其子类提供。然 [826]
而，并不是所有的虚方法都是纯虚方法。在许多继承层次中，父类提供了某一方法的一种默认定义，而这些父类的子类在需要改变这些方法的定义时可以重置这些方法的定义。

```
/*
 * Class: Employee
 * ---------------
 * This class defines the root of the Employee hierarchy.  Employee is
 * an abstract class, which means that there are no objects whose primary
 * type is Employee.  Every object is constructed as one of the subclasses.
 * The getPay method is declared using the virtual keyword, which means
 * that it can be overridden by its subclasses.  The "= 0" notation at the
 * end of the prototype marks getPay as a "pure virtual" method, which
 * is implemented only in the subclasses.
 */

class Employee {
public:
   std::string getName();
   virtual double getPay() = 0;
};

/*
 * Subclasses: HourlyEmployee, CommissionedEmployee, SalariedEmployee
 * ------------------------------------------------------------------
 * These classes represent the concrete manifestations of the abstract
 * Employee class.  Each subclass inherits the getName method from
 * Employee, but defines its own version of the getPay method.
 */

class HourlyEmployee : public Employee {
public:
   virtual double getPay();
   void setHourlyRate(double rate);
   void setHoursWorked(double hours);
};

class CommissionedEmployee : public Employee {
public:
   virtual double getPay();
   void setBaseSalary(double dollars);
   void setCommissionRate(double rate);
   void setSalesVolume(double dollars);
};

class SalariedEmployee : public Employee {
public:
   virtual double getPay();
   void setSalary(double salary);
};
```

图 19-2 Employee 类层次的简化定义 [827]

如果你习惯了其他语言的继承机制，会觉得使用关键字 virtual 标记一个方法没有必

要。在大多数支持继承的语言中，所有的虚方法都需标识，只不过C++语言用virtual这一术语标记虚方法而已。如果子类重置了其父类的一个方法，程序员总是习惯于传统的继承模式，期待在子类中该方法的新定义会被应用到该类型的对象中。但是这一规则在C++语言中不会被自动遵循。如果你缺少了virtual关键字，编译器将会根据对象的声明类型来判断调用哪一个版本的方法，而不是根据对象实际构造的类型来判断。例如，即使你在SalariedEmployee子类中重置了getName方法，在使用一个声明为Employee类型的对象来调用getName方法时，依旧会调用Employee类中定义的原始版本getName方法。新版本的getName方法仅用于显式声明它是SalariedEmployee类的一个方法。

继承自Employee类的三个子类定义的结构相同。子类的头部通过在其类名之后增加一个冒号、关键字public和父类名来表明其继承关系。因此HourlyEmployee类的定义头部如下所示：

```
class HourlyEmployee : public Employee
```

HourlyEmployee类将自动继承其父类Employee类的公有方法，因此该类包含了一个父类中定义的getName方法。然而，getPay方法的定义必须在子类中实现，因为该方法在Employee类中被定义成纯虚方法。每一个子类定义必须包括以下虚方法原型，向编译器告知该类重置了getPay方法：

```
virtual double getPay();
```

getPay方法的实现在.cpp文件中，并使用了类名来标记其所属类。因此，HourlyEmployee类中的getPay方法实现如下，这里我们假设类中包含私有实例变量hoursWorked和hourlyRate：

```
double HourlyEmployee::getPay() {
   return hoursWorked * hourlyRate;
}
```

19.1.3 C++中子类的局限性

面向对象程序设计的一个基本原则就是子类对象是它们所属父类的一个实例。因此，在
828 前面介绍的Employee类继承层次中，HourlyEmployee对象也是更泛化的Employee类的实例。然而，这一事实可能会由于你接触过类似Java语言这类与C++语言机制大不相同的编程语言，而误导你在操作继承的数据结构实例时做出错误的假设。

C++语言继承机制中最容易让编程人员产生混乱的是赋值操作的实现。例如，由于每一个HourlyEmployee对象仍然是一个Employee对象，像在Java语言中一样，将一个HourlyEmployee类型的对象赋值给一个Employee类型对象是合乎逻辑的，如以下代码所示：

```
HourlyEmployee bobCratchit;
Employee clerk;
clerk = bobCratchit;
```



上述程序的第一行将bobCratchit声明为HourlyEmployee类型的对象；第二行将clerk声明为Employee类型。但是错误发生在第三行，程序试图使用bobCratchit进行赋值。虽然这一段代码在C++语言中是完全合法的，但是会弹出一段警告，告诉你代码

很可能不会以你所期望的模式运行。

在C++语言中，即使局部变量是某个类的实例，也总是被分配在栈中。为了创建一个栈帧结构，编译器必须提前知道每一个局部变量所需分配的空间。在这个例子中，编译器为bobCratchit变量分配了足够的空间来存储一个HourlyEmployee类型的对象，也给clerk变量分配了足够的空间来存储Employee类型的对象。其中，HourlyEmployee类扩展了Employee类，这意味着该类中继承了父类已定义的实例变量。每一个子类最少需要与其父类同等大小的存储空间，并且通常所需要的空间会更大。因此，存储这两个变量的栈帧结构图如下图所示：



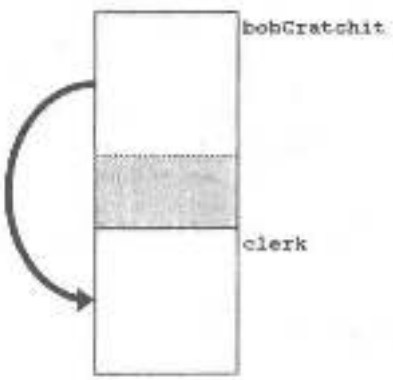
829

C++语言在没有对赋值操作进行重载定义情况下，将一个对象赋值给另一个对象是通过将源对象的所有变量域复制到目的对象相应的变量域来实现的。因此，以下赋值语句：

```
clerk = bobCratchit;
```

将试图拷贝bobCratchit中的数据到clerk中的空间，这就像试图将一颗大钉子嵌入到一个较小的孔中。

在这种情况下，C++语言只会拷贝对象间重合部分的变量域中的内容，在这个例子中就是HourlyEmployee类从Employee继承的部分。因此对象赋值操作后的数据转换如下图所示：



图中HourlyEmployee类的灰色部分变量域的内容将被丢弃。这种赋值时只拷贝对象一部分的行为被称为**切片**（slicing）。

禁止将一个子类实例拷贝到其父类所创建的存储空间会同时导致其他问题。例如，

你可能想要将雇员列表存储在一个矢量中。不幸的是，如果你采用以下声明，将不能达到目的：

```
Vector<Employee> payroll;
```



如同上述赋值例子，payroll 矢量的元素只能储存 Employee 类的实例。如果你试图执行以下语句：

```
payroll.add(bobCratchit);
```



[830] add 方法的实现将切除 bobCratchit 中所有不是 Employee 继承来的数据。

C++ 语言中解决该问题的传统方法就是使用指针来操作对象而不是使用对象本身。例如，最好使用如下方法声明 payroll 矢量：

```
Vector<Employee *> payroll;
```

所有指向 Employee 对象的指针都占有相同的存储空间，因此，这一指针适用于矢量中的所有 Employee 类型对象。除此之外，如果你采用 payroll 中的指针调用 Employee 类中的虚方法，C++ 将会调用与子类中的对应方法相绑定。因此，如果你执行以下代码：

```
for (Employee *ep : payroll) {
   cout << ep->getName() << ": " << ep->getPay() << endl;
}
```

将会得到 payroll 矢量中的所有雇员列表，同时还有每个雇员应得的报酬。对 getPay 方法的调用将为其选择适合实际 Employee 子类方法的版本。

可惜，使用指针使内存管理的过程变得复杂，这是 C++ 语言编程所面临的重大挑战。每个类都定义了一个析构函数用以释放为该类对象所分配的内存空间。因此，我们可以让内存管理置于我们的控制之下。但是当你的指针越界时，就不能保证析构函数会在合适的时间被调用，此时类的整体复杂度迅速增大。

当你用 C++ 语言设计一个类时，最重要的就是要保持这一平衡。在大多数情况下，最好的方法是避免使用继承并创建独立的类来管理自身的堆内存。但是，如果你认为需要使用继承机制，将面临一个困难的选择。一种方法是定义私有的拷贝构造函数和重载赋值操作符函数，使得拷贝对象在该继承层次中是被禁止的。虽然这一做法消除了切片导致数据丢失的可能，但是禁止拷贝使得将对象嵌入到大型数据结构变得困难。但是，如果你不禁用赋值，用户就必须承担内存管理的职责。从不管理到完全管理，这种情况会给用户带来可怕的工作量。

C++ 语言的类库设计者根据不同的情况做出了不同的选择。集合类实现为独立的类，并不包含继承关系。相反，流类构成了一个复杂的继承层次但不允许用户进行赋值。设计者对每一种情况都选择了向用户隐藏内存管理的细节。当你设计自己的类时，这么做是明智的。

[831]

19.2 图形对象的继承层次

本书的图形接口设计例子中，我们发现对继承层次的应用所带来的好处使得复杂性增加这一弊端可以被忽略。图形用户接口在当今的系统中普遍存在，并通常依赖继承层次来定义

相关的应用程序接口（application programming interface，API），实现者使用这一接口来编写必要的代码。在一个特定的 API 中，继承关系分为几个层次。用于应用显示的窗口、对话框和面板构成了一个类层次。在这个类层次中，大多数窗口系统允许程序显示文本、图像和各种几何形状，从而形成了它们自己的一个继承层次。

Stanford 库包括了一个 gobject.h 接口，该接口允许用户在图形窗口中显示对象。本节将实现一个简化版本的 gobjects.h，该接口提供了图 19-3 中展示的类，类限制在直线、矩形和椭圆类上。

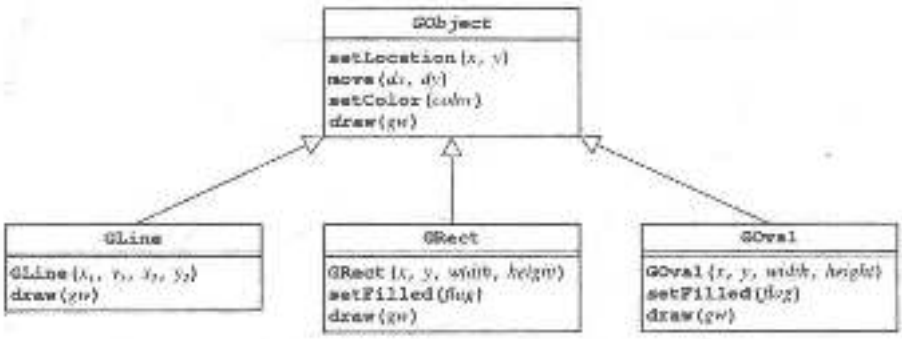


图 19-3 简化的 Gobject 类继承

正如在图 19-3 中所看到的那样，gobjects.h 中的类自己组成了一个继承层次。抽象类 GObject 是继承树中的根。GObject 类中提供的方法可以应用于所有图形对象。给定任何 GObject 对象，你可以设置它在图形窗口中的位置，也可以定义它的颜色。GObject 类还定义了一个 draw 方法，它可以在图形窗口中用来显示一个对象。在 GObject 类中，draw 被声明为一个纯虚函数，这意味着在这一类层次中它没有被实现。draw 函数的实现由 GLine、Grect 和 Goval 类提供。图 19-4 展示了简化版本 gobjects.h 接口的内容。 R32

```
/*
 * File: gobjects.h
 * ----------------
 * This file defines a simple hierarchy of graphical objects.
 */

#ifndef _gobjects_h
#define _gobjects_h

#include <string>
#include "gwindow.h"

/*
 * Class: GObject
 * --------------
 * This class is the root of the hierarchy and encompasses all objects
 * that can be displayed in a window.  Clients typically use a pointer
 * to a GObject rather than the GObject itself.
 */

class GObject {

public:

/*
```

图 19-4 gobjects.h 接口

```
 * Method: setLocation
 * Usage: gobj->setLocation(x, y);
 * -------------------------------
 * Sets the x and y coordinates of gobj to the specified values.
 */

   void setLocation(double x, double y);

/*
 * Method: move
 * Usage: gobj->move(dx, dy);
 * --------------------------
 * Adds dx and dy to the coordinates of gobj.
 */

   void move(double x, double y);

/*
 * Method: setColor
 * Usage: gobj->setColor(color);
 * -----------------------------
 * Sets the color of gobj.
 */

   void setColor(std::string color);

/*
 * Abstract method: draw
 * Usage: gobj->draw(gw);
 * ----------------------
 * Draws the graphical object on the GraphicsWindow specified by gw.
 * This method is implemented by the specific GObject subclasses.
 */

   virtual void draw(GWindow & gw) = 0;

protected:

/* The following methods and fields are available to the subclasses */

   GObject();                          /* Superclass constructor          */
   std::string color;                  /* The color of the object         */
   double x, y;                        /* The coordinates of the object   */

};

/*
 * Subclass: GLine
 * ---------------
 * The GLine subclass represents a line segment on the window.
 */

class GLine : public GObject {

public:

/*
 * Constructor: GLine
 * Usage: GLine *lp = new GLine(x1, y1, x2, y2);
 * ---------------------------------------------
 * Creates a line segment that extends from (x1, y1) to (x2, y2).
 */

   GLine(double x1, double y1, double x2, double y2);

/* Prototypes for the overridden virtual methods */

   virtual void draw(GWindow & gw);

private:
   double dx;                          /* Horizontal distance from x1 to x2
   double dy;                          /* Vertical distance from y1 to y2

};

/*
 * Subclass: GRect
 * ---------------
```

图 19-4 （续）

```
 * The GRect subclass represents a rectangle.
 */

class GRect : public GObject {

public:

/*
 * Constructor: GRect
 * Usage: GRect *rp = new GRect(x, y, width, height);
 * --------------------------------------------------
 * Creates a rectangle of the specified size whose upper left corner is (x, y).
 */

   GRect(double x, double y, double width, double height);

/*
 * Method: setFilled
 * Usage: rp->setFilled(flag);
 * ---------------------------
 * Indicates whether the rectangle is filled.
 */

   void setFilled(bool flag);

/* Prototypes for the overridden virtual methods */

   virtual void draw(GWindow & gw);

private:
   double width, height;          /* Dimensions of the rectangle          */
   bool filled;                   /* True if the rectangle is filled      */

};

/*
 * Subclass: GOval
 * ---------------
 * The GOval subclass represents an oval defined by a bounding rectangle.
 */

class GOval : public GObject {

public:

/*
 * Constructor: GOval
 * Usage: GOval *op = new GOval(x, y, width, height);
 * --------------------------------------------------
 * Creates an oval inscribed in the specified rectangle.
 */

   GOval(double x, double y, double width, double height);

/*
 * Method: setFilled
 * Usage: op->setFilled(flag);
 * ---------------------------
 * Indicates whether the oval is filled.
 */

   void setFilled(bool flag);

/* Prototypes for the overridden virtual methods */

   virtual void draw(GWindow & gw);

private:
   double width, height;          /* Dimensions of the bounding rectangle */
   bool filled;                   /* True if the oval is filled           */

};

#endif
```

图 19-4 （续）

gobjects.h接口中的每一个子类都定义了一个构造函数，该构造函数的参数与

GWindow 类的构造函数参数一致，并且子类与父类将显示相同的图形。例如，GLine 构造函数传入直线两个端点的坐标值，其原型如下：

```
GLine(double x1, double y1, double x2, double y2);
```

这些参数与 GWindow 类中的 drawLine 方法相同。

图 19-4 中的代码引入了 C++ 语言一个重要的新特性。在许多接口层次中，让子类可以访问父类的实例变量是很实用的，因为这样做可以简化代码。GObject 类通过使用关键字 protected 来标识这些实例变量达成了这一目标。protected 部分的各项对于其所有的子类来说都是可访问的，但用户不可以访问。

833 / 836

实现 GObject 类层次所需的代码显示在图 19-5 中。

```
/*
 * File: gobjects.cpp
 * ------------------
 * This file implements the gobjects.h interface
 */

#include <string>
#include "gwindow.h"
#include "gobjects.h"
using namespace std;
/*
 * Implementation notes: GObject class
 * -----------------------------------
 * The constructor for the superclass sets the default color (BLACK).
 */

GObject::GObject() {
   setColor("BLACK");
}

void GObject::setLocation(double x, double y) {
   this->x = x;
   this->y = y;
}

void GObject::move(double dx, double dy) {
   x += dx;
   y += dy;
}

void GObject::setColor(string color) {
   this->color = color;
}

/*
 * Implementation notes: GLine class
 * ---------------------------------
 * The constructor for the GLine class has to change the specification
 * of the line from the endpoints passed to the constructor to the
 * representation that uses a starting point along with dx/dy values.
 */

GLine::GLine(double x1, double y1, double x2, double y2) {
   this->x = x1;
   this->y = y1;
   this->dx = x2 - x1;
   this->dy = y2 - y1;
}

void GLine::draw(GWindow & gw) {
   gw.setColor(color);
   gw.drawLine(x, y, x + dx, y + dy);
}

/*
 * Implementation notes: GRect and GOval classes
 * ---------------------------------------------
```

图 19-5 GObject 类层次的实现

```
 * The constructors for these classes store their arguments in the
 * corresponding instance variables.  The draw method forwards the
 * appropriate request to the GWindow class.
 */

GRect::GRect(double x, double y, double width, double height) {
   this->x = x;
   this->y = y;
   this->width = width;
   this->height = height;
   filled = false;
}

void GRect::setFilled(bool flag) {
   filled = flag;
}

void GRect::draw(GWindow & gw) {
   gw.setColor(color);
   if (filled) {
      gw.fillRect(x, y, width, height);
   } else {
      gw.drawRect(x, y, width, height);
   }
}

GOval::GOval(double x, double y, double width, double height) {
   this->x = x;
   this->y = y;
   this->width = width;
   this->height = height;
   filled = false;
}

void GOval::setFilled(bool flag) {
   filled = flag;
}

void GOval::draw(GWindow & gw) {
   gw.setColor(color);
   if (filled) {
      gw.fillOval(x, y, width, height);
   } else {
      gw.drawOval(x, y, width, height);
   }
}
```

图 19-5 （续）

837
|
838

19.2.1 调用父类的构造函数

构造函数负责初始化一个对象的数据域以确保所创建的对象有一个一致状态。为了维护整个继承层次的一致性，每一个子类的构造函数必须调用其父类的某个构造函数。在缺乏其他相关定义的情况下，这一责任由父类的默认构造函数承担。因此，在初始化对象自身的数据域之前，GLine、GRect 和 GOval 对象构造函数将调用父类 GObject 的默认构造函数。

将 GObject 类的构造函数声明为 protected 部分意味着该类的子类可以访问 GObject 的构造函数，但是用户不可以这么做。采取这一机制可禁止用户声明 GObject 类的对象，或者将 GObject 作为任何集合类的模板参数。在 19.13 这一节中，所有试图以上述方式使用 GObject 类的用户将使自己陷入到麻烦之中。因此，若想避免麻烦，应将构造函数声明为 public 部分。当然，像本书中所介绍的，现实中用户可以使用指针来操作 GObject 类对象。

然而，在某些情况下，调用带有参数的构造函数相比调用不带参数的默认构造函数更加有用。C++ 语言允许你在子类的构造函数代码中增加额外的定义，这一定义称

为**初始化列表**（initializer list）。初始化列表放在构造函数体开始的花括号之前，与函数的参数列表处于同一行并在其之后。初始化列表的元素应该符合以下其中一条标准：

- 在父类名后用括号括起来的参数列表，参数列表必须与父类中某一构造函数的函数原型相匹配。
- 在父类的数据域名后用括号括起来的该数据域的初始化值。

上述两种形式都可以在专业 C++ 语言编程中使用，但本书只采用第一种风格。

说明初始化列表使用方式最简单的方法就是通过一个简单的例子进行实践。假设你想要增加一个 GOval 的子类 GCircle 来扩展 GObject 继承层次。按常理说，每一个圆通过指定半径和圆心坐标来进行定义。因此，对用户来说，GCircle 类的构造函数获取这些参数值来构造圆将比使用 GOval 类的长方形边界构造圆更加简便。图 19-6 展示了采用这一方法
839 的 GCircle 类的接口内容。

```
/*
 * Subclass: GCircle
 * -----------------
 * The GCircle subclass represents a circle.
 */

class GCircle : public GOval {

public:

/*
 * Constructor: GCircle
 * Usage: GCircle circle(x, y, r);
 *        GCircle *cp = new GCircle(x, y, r);
 * ------------------------------------------
 * Creates a circle of radius r centered at the point (x, y).
 */

   GCircle(double x, double y, double r);

};
```

图 19-6 GCircle 类的接口内容

GCircle 类实现的唯一难点在于怎样初始化自 GOval 继承的 GCircle 类的对象。GCircle 类不能使用之前的方法调用 GOval 类的默认构造函数，因为 GOval 类并没有定义默认构造函数。GCircle 类的构造函数必须传入 GOval 所需的参数来调用 GOval 类的构造函数。幸运的是，这些参数值可以很容易地通过计算有关 x、y 和 r 的表达式得到。图 19-7 展示了 GCircle 类的实现。这一实现代码只需定义初始化列表即可。

```
/*
 * Implementation notes: GCircle
 * -----------------------------
 * The GCircle class is a subclass of GOval for which the constructor
 * interprets its arguments in a different way.  This constructor uses
 * an initialization list to call the GOval constructor with the
 * correct arguments.
 */

GCircle::GCircle(double x, double y, double r)
          : GOval(x - r, y - r, 2 * r, 2 * r) {
   /* Empty */
}
```

840 图 19-7 GCircle 类实现中对初始化列表的应用

19.2.2 将 GObject 类指针存储在矢量中

定义 GObject 类继承层次的一个优势就是这么做允许我们将图形对象都存储在一个集合中，当然存储的时候你必须记住使用指向 GObject 对象的指针而不是使用对象。图 19-8 向我们展示了将一个 GObject 类集合组织为一个矢量，之后在图形窗口中绘制这些对象，它产生了与第 2 章中 GraphicsExample 程序一样的输出结果。这个程序在使用完成后释放了所分配的堆内存。

```
/*
 * File: TestDisplayList.cpp
 * -------------------------
 * This program tests the GObject classes by storing pointers to several
 * graphical objects in a vector and then drawing them all at once.  The
 * picture is the same as the GraphicsExample.cpp program from Chapter 2.
 */

#include <iostream>
#include "gwindow.h"
#include "gobjects.h"
#include "vector.h"
using namespace std;

int main() {
   GWindow gw;
   double width = gw.getWidth();
   double height = gw.getHeight();
   GRect *rp = new GRect(width / 4, height / 4, width / 2, height / 2);
   GOval *op = new GOval(width / 4, height / 4, width / 2, height / 2);
   rp->setColor("BLUE");
   op->setColor("GRAY");
   Vector<GObject *> displayList;
   displayList.add(new GLine(0, height / 2, width / 2, 0));
   displayList.add(new GLine(width / 2, 0, width, height / 2));
   displayList.add(new GLine(width, height / 2, width / 2, height));
   displayList.add(new GLine(width / 2, height, 0, height / 2));
   displayList.add(rp);
   displayList.add(op);
   for (GObject *sp : displayList) {
      sp->draw(gw);
   }
   for (GObject *sp : displayList) {
      delete sp;
   }
   displayList.clear();
   return 0;
}
```

图 19-8 将图形对象存储在一个矢量中的程序

841

19.3 表达式的类层次

应用类继承特性的另一个场合是在编程语言中算术表达式的表示上。在编译程序时，C++ 编译器必须分析表达式来获取其中的信息，包括操作符应该应用到哪个操作数上，以及选择这些操作符所表示的功能。一般来说，编译器会以树结构来保存这些信息，在树结构中，每一个独立节点用来表示不同表达式类型，这些节点属于类继承的一部分。

本章的目的在于通过实现一个简单的应用程序介绍算术表达式的表示机制，这一应用程序不断执行以下步骤：

1. 读取用户输入的表达式并将其存储到内部的树结构中。
2. 检查树结构并计算表达式的值。
3. 在控制台中输出表达式的计算结果。

这种迭代过程称为读取－求值－输出循环（read-eval-print loop）。读取－求值－输出循环是解释器的重要特性。在程序没有被翻译成机器语言之前，我们使用解释器执行并计算程序中的重要操作步骤。虽然解释一个程序的效率低于编译一个程序，但相比之下，解释性语言更加容易编写和理解。

读取一个表达式并将其转换为内部格式这一操作由以下三部分组成：

1. 输入。输入部分将从用户端读取一行文本，这一操作可以简单地通过调用 getline 函数来实现。
2. 词法分析。词法分析阶段负责将输入的文本行分解为独立单元，这些独立单元称为记号（tokens）。每一个记号都表示一个单独的逻辑实体，例如一个整型常量、操作符，或者变量名。第6章的 TokenScanner 类提供了执行这一阶段操作的理想工具。
3. 语法分析。当一行文本被分解为组成该行文本的一组记号之后，将执行语法分析阶段，这一阶段将检测每一个独立记号是否构成了一个合法的表达式，如果表达式合法，则检测这些表达式具有什么结构。为了实现这一目标，语法分析器必须能够使用输入的独立记号构成正确的语法解析树。

342 图19-9列出了一个可以执行这些阶段的简单解释器主程序。

```
/*
 * File: Interpreter.cpp
 * ---------------------
 * This program simulates the top level of an expression interpreter.  The
 * program reads an expression, evaluates it, and then displays the result.
 */

#include <iostream>
#include <string>
#include "error.h"
#include "exp.h"
#include "parser.h"
#include "tokenscanner.h"
using namespace std;

int main() {
   EvaluationContext context;
   TokenScanner scanner;
   Expression *exp;
   scanner.ignoreWhitespace();
   scanner.scanNumbers();
   while (true) {
      exp = NULL;
      try {
         string line;
         cout << "=> ";
         getline(cin, line);
         if (line == "quit") break;
         scanner.setInput(line);
         Expression *exp = parseExp(scanner);
         int value = exp->eval(context);
         cout << value << endl;
      } catch (ErrorException ex) {
         cerr << "Error: " << ex.getMessage() << endl;
      }
      if (exp != NULL) delete exp;
   }
   return 0;
}
```

图19-9 解释器的 main 函数部分

程序的运行示例如下：



运行示例向我们清楚地展示了解释器允许向变量赋值，并且遵循 C++ 的运算优先级规则：先计算乘法，然后计算加法。

该实现的关键在于 Expression 类，它代表一个算术表达式。Expression 类在 exp.h 接口文件中定义。在你学习这一接口之前，可以尝试通过观察解释器代码来推断该接口的构成。从变量 exp 的声明已知，代码使用指向 Expression 对象的指针来工作，而不是直接使用对象本身。这一设计也暗示着所有需要操作类似 exp 这样的表达式变量的方法必须使用 -> 操作符。并且，虽然你不知道 Expression 类中具体方法实现的操作，但单从代码中我们依旧可以推断该类提供了一个名为 eval 的方法。当然，这一操作的目的与其名称应该是相符的。作为 expression 包的用户，你应更多地关注怎样使用表达式，而不是表达式是如何实现的。作为一个用户，你需要将 Expression 类想象成一个抽象数据类型。底层细节只有在你想要修改类的实现时才需要去关注。

在图 19-9 中，你可能还会注意到 eval 方法需要传入一个名为 context 的参数，这个参数是类 EvaluationContext 的对象。EvaluationContext 类的设计初衷是用于**符号表**（symbol table）的维护，该表记录了每一个变量被赋予的值。正如你所期望的那样，EvaluationContext 类的代码使用了一个 Map 以实现变量名与变量值之间的映射关系。实际上，这是属于实现细节方面的问题。EvaluationContext 类中的方法构成了对符号表的操作，这一操作让我们能更好地理解编程语言的语法。

19.3.1 异常处理

类似你在本书中看到的其他程序，解释器中的模块通过调用 error 函数来报告错误，这一机制已经在第 2 章中介绍过。error 函数的用法与之前类似，函数会输出错误信息，并终止程序的执行。如果解释器在你输入一个冗长复杂的表达式时发生终止，你会对此感到非常不便。只需设想一下应用卡死导致你之前的所有输入消失无踪的情况，你就会明白这种感受。与大多数交互程序一样，解释器最好被设计成可以反馈给用户详细的错误信息，并且允许用户对输入进行修正。

图 19-9 中的解释器程序是依靠 C++ 语言中一种称为**异常处理**（exception handler）的特性向用户准确而人性化地报告错误信息，这一特性允许编程人员对超出程序正常执行范围事件的发生采取不同的措施。与 Stanford 库中的实现一样，error 函数产生信号来报告某种错误已经发生，并通知其他函数对该错误采取对应的措施，即使该错误发生点位于实现错误处理策略的函数之中，也不会影响其错误处理过程。实现这一机制的唯一要求是响应异常的代码必须出现在函数调用链中错误出现的位置之前。标记异常情况的代码被称为**抛出异常**（throw an exception）。为了与这一名称相匹配，处理异常的代码称为**捕获**（catch）异常。

C++ 语言的异常处理采用 try 语句，其形式如下所示：

```
try {
   code under the control of the try statement
} catch (type var) {
   code to respond to an exception with the specified value type
}
```

用于抛出异常的语句语法模式为：

```
throw value;
```

当这条语句出现在嵌套在 try 语句块的函数中时，程序将会暂停执行正在执行的函数，并且沿着函数调用链进行回溯，回退并弹出栈帧结构，直到到达包含 try 语句的栈点为止。假设 throw 语句中的值与 catch 子句中的类型相匹配，该值 value 将会被赋值给变量 var，并将控制权移交给 catch 子句。在更复杂的应用中，一个 try 语句会拥有多个 catch 子句，不同的 catch 子句的区别在于其接受的参数类型不同。本书中的 try 语句只用于捕获 error 函数抛出的 ErrorException，因此只需要一个 catch 子句。

19.3.2 表达式结构

在完成解释器的实现之前，你需要理解什么是表达式，以及怎样用对象这一概念去表示一个表达式。类似于考虑怎样进行编程抽象，以一个 C++ 程序员的经验来分析表达式的本质是非常有必要的。举例来说，你知道以下表达式：

```
0
2 * 11
3 * (a + b + c)
x = x + 1
```

845

这在 C++ 语言中是合法的。同时，你也知道以下文本行：

```
2 * (x - y
17 k
```

这并不是表达式。第一行的括号不配对，第二行则缺失了操作符。理解表达式本质最重要的一点就是弄清表达式是由哪些部分组成的，并且这些部分可以让你分辨出合法和不合法的表达式。

19.3.3 表达式的递归定义

就像你所看到的，定义合法表达式的最佳方式就是使用递归的方法。一个符号序列如果符合以下的任意一条，则该符号序列就是一个表达式：

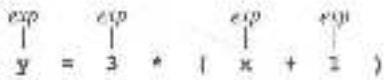
1. 符号序列是一个整型常量。
2. 符号序列是一个变量名。
3. 符号序列是被圆括号括起来的一个表达式。
4. 符号序列是被一个操作符分割成两部分的两个表达式序列。

前两个标准表示单个符号的场合。后两个标准递归地定义了由一组独立表达式符号组成新表达式的情况。

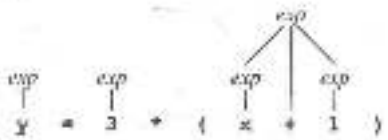
为了理解怎样应用这一递归的标准，我们考虑以下符号序列：

```
y = 3 * (x + 1)
```

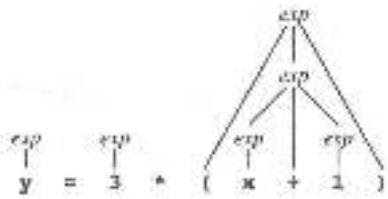
这一序列构成了一个表达式吗？根据经验进行判断，你得到的答案是肯定的，但是你可以通过使用表达式的递归定义来检验你的答案。根据第一个标准，整型常量 3 和 1 是两个表达式。同样的，根据第二个标准，变量名 x 和 y 也是表达式。因此，现在你已经知道下图中使用 exp 进行标记的符号是表达式：



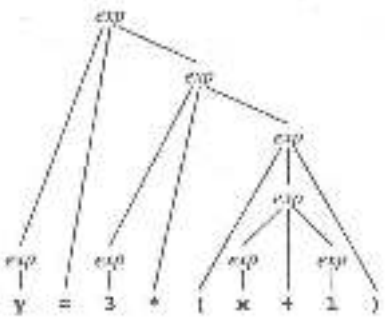
此时，你可以开始调用它的递归定义了。给定 x 和 1 是表达式，你可以通过第四个标准来判定符号 x+1 是一个合法的表达式，因为这一字符串包含了两个被操作符分割开的独立表达式。你可以在图中将这一观察结果进行标记，像下面这样增加一个新的表达式标记连接到表达式中符合该标准的部分：



现在，根据第三个标准，圆括号中及其包含的字符可以被定义为一个单独的表达式，这一结果如下图所示：



再调用第四个标准至少两次，考虑到剩下的操作符，你可以知道整个这些字符像下面这样组成了一个完整的表达式：



正如你所看到的，这个图组成了一个树结构。树结构证明了输入符号序列刚好符合一种编程语言的语法规则，这棵树也被称为解析树（parse tree）。

19.3.4 二义性

从一个符号序列中生成一棵解析树时有几点需要注意。根据前一节总结出的四个表达式判定标准，一个表达式可能生成不止一棵解析树，如以下表达式：

```
y = 3 * (x + 1)
```

虽然两结构在上一节的最后与程序员预期的大致相同，但是根据第四个标准将 y=3 看成一
847 个表达式也是合法的，因此整个表达式将包含子表达式 y=3，并在其后连接一个乘法符号和另一个表达式（x+1）。这种操作最终也达到了将输入序列转换为表达式的目的，但是却生成了另外一棵不同的解析树。图 19-10 向我们展示了这两棵不同的解析树。左边的解析树是由前一节生成的，与表达式表达的语义一致。右边的解析树向我们展示了对表达式判断标准的一次正确运用，但得到的结果与程序员的预期设想相异。

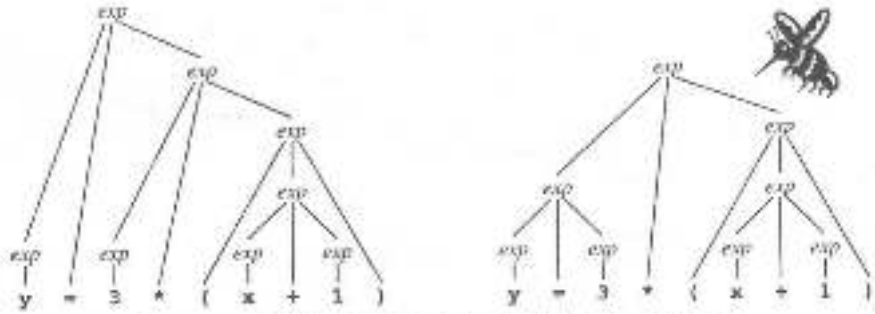


图 19-10 预期的解析树以及合法却不符合语义的解析树

第二棵解析树中存在的问题就是树的生成过程忽略了乘法必须在赋值之前执行这一运算优先级顺序。递归定义只规定了被一个操作符分割的两个独立表达式可以组成一个表达式，但是却没有说明操作符之间的优先级关系，因此这一语法定义允许同时生成两个不同的结果。由于该套标准处理同一个字符串产生的结果不具有唯一性，所以我们认为上一节给出的表达式定义具有二义性（ambiguous）。为了解决定义的二义性，语法分析算法必须包含某些机制以确保操作符执行正确的操作符优先顺序。

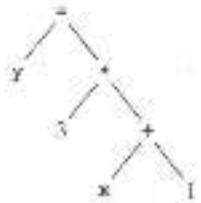
怎样解决表达式二义性这一问题在 19.4 节中将进行详细介绍。现在要解决的有关解析树的问题是：你要怎样将一个表达式表示为某种数据结构。为此，我们必须假设图 19-10 所示的解析树并不是二义的。每一棵树的结构都清晰地表示了一个合法的表达式的结构。二义性只存在于将已输入字符串转换为解析树的阶段。在得到正确的解析树之后，这棵树的结构将包含理解操作符执行顺序所需的全部信息。

19.3.5 表达式树

实际上，解析树中所包含的信息远比数值计算阶段所需的信息要多。圆括号可以在生
848 成解析树阶段起到重要作用，但是在表达式的结构确定之后的数值计算阶段却毫无意义。如果你只关心如何得到表达式的值，那么并不需要在树结构中包括圆括号。这一结果允许你将一棵解析树简化为一个更抽象的结构。该结构称为表达式树（expression tree），这一结构更适合数值计算。在表达式树中，用圆括号括起来的子表达式的节点被删除。除此之外，取消树上的 exp 标记，并使用合适的操作符符号来标记每个节点将更加简便。例如，以下的表达式：

```
y = 3 * (x + 1)
```

表示为以下表达式树：



表达式树的结构与第 16 章讲解的二叉搜索树在很多方面很类似，但是它们也有某些重要的差别。在二叉搜索树中，每个节点都有相同的结构。在表达式树中，存在以下三种不同类型的节点：

1. 常量节点（constant node）表示整型常量，比如上例表达式树中的 3 和 1。
2. 标识符节点（identifier node）表示了变量名，比如 x 和 y。
3. 复合节点（compound node）表示了将一个操作符应用到两个操作数上，其中每一个操作数都是另外一个任意类型的表达式。

每一种节点类型都对应着表达式递归定义中的某条规则。Expression 类的定义必须让用户可以操作全部三种类型的表达式节点。与此类似，该类底层实现必须确保树中可以同时存在多种不同的表达式。

为了表示这一结构，你必须将表达式定义为可以根据自身不同类型来构成不同的结构。例如，一个整型表达式必须将整型数当做内部结构的一部分进行存储。一个标识符表达式必须包括一个同时拥有左表达子式和右表达子式的操作符。定义一个允许表达式拥有这些底层结构的抽象类型需要你实现类的继承，这一继承层次使得 Expression 类成为三个子类的父类，每一个子类都代表了一种表达式类型。 849

创建继承层次是表示表达式树的合理方法。这一继承层次的顶层是 Expression 类，该类定义了所有表达式类型共有的特性。Expression 类拥有三个子类，每个子类都表示了一种表达式类型。上层的 Expression 类，下层的 ConstantExp 类、IdentifierExp 类和 CompoundExp 类，这四个类都在 exp.h 文件中定义。

与典型的类继承层次类似，大多数共有的方法定义在 Expression 类中，然后在子类中单独实现。每一个 Expression 对象都实现了如下方法：

- eval 方法计算出表达式的值，在该应用中该值是一个整型值。对于常量表达式来说，eval 只是简单地返回常量值。对于标识符表达式来说，eval 方法通过查询符号表中的标识符名来计算表达式的值。对于组合表达式来说，eval 方法通过递归地调用左子式和右子式并执行正确的操作符来实现。
- toString 方法将表达式转换成一个字符串，并在表达式中无差别地增加圆括号来明确地表示出表达式的结构。虽然 toString 方法并不用于解释器中，但这一方法在程序的调试查错步骤中将起到重要的作用。如果你不清楚表达式结构是否正确，可以通过调用 toString 方法进行确认。
- getType 方法用于计算表达式的类型，其返回类型是 expressionType 类型中定义的 CONSTANT、IDENTIFIER 或 COMPOUND 枚举常量中的一种。
- Expression 类提供了一系列取值方法，这些方法返回每个表达式结构中的某些部分。在 exp.h 文件中，取值方法定义在抽象类中，因为这样做将使该类层次更易于使用。

在 Expression 类中，上述方法都定义为虚函数。当程序执行时，计算机将通过检测表达式的实际类型来决定调用哪个类的方法。

19.3.6 exp.h 接口

图 19-11 展示了 Expression 类及其子类的接口。每一个 Expression 子类都必须实现其父类中定义的三个纯虚方法：eval、toString 和 getType。父类中定义了这些方法
850 的原型，每个子类负责使用不同的方式独立实现这些方法。

```
/*
 * File: exp.h
 * -----------
 * This interface defines a class hierarchy for arithmetic expressions.
 */

#ifndef _exp_h
#define _exp_h

#include <string>
#include "exp.h"
#include "tokenscanner.h"

/* Forward reference */

class EvaluationContext;

/*
 * Type: ExpressionType
 * --------------------
 * This enumerated type is used to differentiate the three different
 * expression types: CONSTANT, IDENTIFIER, and COMPOUND.
 */

enum ExpressionType { CONSTANT, IDENTIFIER, COMPOUND };

/*
 * Class: Expression
 * -----------------
 * This class is used to represent a node in an expression tree.
 * Expression itself is an abstract class, which means that there are
 * never any objects whose primary type is Expression.  All objects are
 * instead created using one of the three concrete subclasses:
 *
 *   1. ConstantExp   -- an integer constant
 *   2. IdentifierExp -- a string representing an identifier
 *   3. CompoundExp   -- two expressions combined by an operator
 *
 * The Expression class defines the interface common to all expressions;
 * each subclass provides its own implementation of the common interface.
 */

class Expression {

public:

/*
 * Constructor: Expression
 * -----------------------
 * Specifies the constructor for the base Expression class.  Each subclass
 * defines its own constructor as well.
 */

   Expression();

/*
 * Destructor: ~Expression
 * Usage: delete exp;
 * ------------------
 * Deallocates the storage for this expression.  This method must be
 * declared virtual to ensure that the correct subclass destructor
 * is called when deleting an expression.
 */
```

图 19-11 exp.h 接口

```
   virtual ~Expression();

/*
 * Method: eval
 * Usage: int value = exp->eval(context);
 * ------------------------------------
 * Evaluates this expression and returns its value in the context of
 * the specified EvaluationContext object.
 */

   virtual int eval(EvaluationContext & context) = 0;

/*
 * Method: toString
 * Usage: string str = exp->toString();
 * ------------------------------------
 * Returns a string representation of this expression.
 */

   virtual std::string toString() = 0;

/*
 * Method: getType
 * Usage: ExpressionType type = exp->getType();
 * --------------------------------------------
 * Returns the type of the expression, which must be one of the constants
 * CONSTANT, IDENTIFIER, or COMPOUND.
 */

   virtual ExpressionType getType() = 0;

/*
 * Getter methods for convenience
 * ------------------------------
 * The following methods get the fields of the appropriate subclass.  Calling
 * these methods on an object of the wrong subclass generates an error.
 */

   virtual int getConstantValue();
   virtual std::string getIdentifierName();
   virtual std::string getOperator();
   virtual Expression *getLHS();
   virtual Expression *getRHS();

};

/*
 * Subclass: ConstantExp
 * ---------------------
 * This subclass represents an integer constant.
 */

class ConstantExp : public Expression {

public:

/*
 * Constructor: ConstantExp
 * Usage: Expression *exp = new ConstantExp(value);
 * ------------------------------------------------
 * Creates a new integer constant expression.
 */

   ConstantExp(int value);

/* Prototypes for the virtual methods overridden by this class */

   virtual int eval(EvaluationContext & context);
   virtual std::string toString();
   virtual ExpressionType getType();
   virtual int getConstantValue();

private:
   int value;                   /* The value of the integer constant */

};

/*
 * Subclass: IdentifierExp
 * -----------------------
```

图 19-11 （续）

```
 * This subclass represents an identifier used as a variable name.
 */

class IdentifierExp : public Expression {

public:

/*
 * Constructor: IdentifierExp
 * Usage: Expression *exp = new IdentifierExp(name);
 * ------------------------------------------------
 * Creates an identifier expression with the specified name.
 */

   IdentifierExp(std::string name);

/* Prototypes for the virtual methods overridden by this class */

   virtual int eval(EvaluationContext & context);
   virtual std::string toString();
   virtual ExpressionType getType();
   virtual std::string getIdentifierName();

private:
   std::string name;            /* The name of the identifier */

};

/*
 * Subclass: CompoundExp
 * ---------------------
 * This subclass represents a compound expression consisting of
 * two subexpressions joined by an operator.
 */

class CompoundExp : public Expression {

public:

/*
 * Constructor: CompoundExp
 * Usage: Expression *exp = new CompoundExp(op, lhs, rhs);
 * -------------------------------------------------------
 * Creates a new compound expression composed of the operator (op)
 * and the left and right subexpressions (lhs and rhs).
 */

   CompoundExp(std::string op, Expression *lhs, Expression *rhs);

/* Prototypes for the virtual methods overridden by this class */

   virtual ~CompoundExp();
   virtual int eval(EvaluationContext & context);
   virtual std::string toString();
   virtual ExpressionType getType();
   virtual std::string getOperator();
   virtual Expression *getLHS();
   virtual Expression *getRHS();

private:

   std::string op;              /* The operator string (+, -, *, /) */
   Expression *lhs, *rhs;       /* The left and right subexpression */

};

/*
 * Class: EvaluationContext
 * ------------------------
 * This class encapsulates the information that the evaluator needs to
 * know in order to evaluate an expression.
 */

class EvaluationContext {

public:

/*
```

图 19-11 （续）

```
   * Method: setValue
   * Usage: context.setValue(var, value);
   * ------------------------------------
   * Sets the value associated with the specified var.
   */

      void setValue(std::string var, int value);

   /*
   * Method: getValue
   * Usage: int value = context.getValue(var);
   * -----------------------------------------
   * Returns the value associated with the specified variable.
   */

      int getValue(std::string var);

   /*
   * Method: isDefined
   * Usage: if (context.isDefined(var)) . . .
   * ----------------------------------------
   * Returns true if the specified variable is defined.
   */

      bool isDefined(std::string var);

   private:

      Map<std::string,int> symbolTable;

   };

   #endif
```

图 19-11 （续）

851 / 855

19.3.7 Expression 子类的表示

抽象类 Expression 并没有声明实例变量。因为在该类层次中并不存在所有节点类型共有的数据值，所以这一设计是非常合理的。每一个子类都有其独立的数据存储需求。一个整型节点需要存储一个整型常量，一个复合节点需要存储其子表达式的指针，以此类推。每一个子类都声明了自身表达式类型所需的实例变量。每一个子类也同时定义了自己的构造函数，这些构造函数都需要传入构成对应类型表达式所需的参数。例如，为了创建一个常量表达式，你需要定义一个整型数值。为了构造一个复合表达式，你需要提供操作符、左子式和右子式。

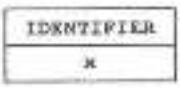
所有的表达式对象都是不可变的，这也意味着每一个 Expression 对象一旦创建就不能改变。虽然允许用户将一个表达式嵌入到一个规模更庞大的表达式中，但是接口并不提供任何用于改变已存在表达式组成部分的机制。使用不可变类型来表示表达式使得 Expression 类的实现与其用户分离开来。因为用户不再被允许对底层实现做出任何改变，所以用户不能使用违反表达式树要求的方法去改变其内部结构。

19.3.8 表达式图解

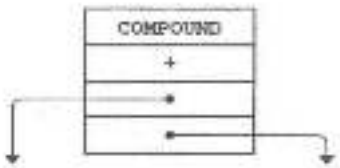
为了让你更深入地理解 Expression 对象是怎样存储的，我们建模了计算机内存中这一结构的表示。Expression 对象的表示取决于它的特定子类。你可以通过将三个子类分开单独考虑的方式画出表达式树的结构。一个 ConstantExp 对象只存储一个整型值，下图表示了其中存储着整型值 3：

CONSTANT
3

一个 IdentifierExp 对象存储一个代表变量名的字符串，该变量名用变量 x 表示：



856 一个 CompoundExp 对象存储了一个二元操作符和两个分别表示左子式和右子式的指针：



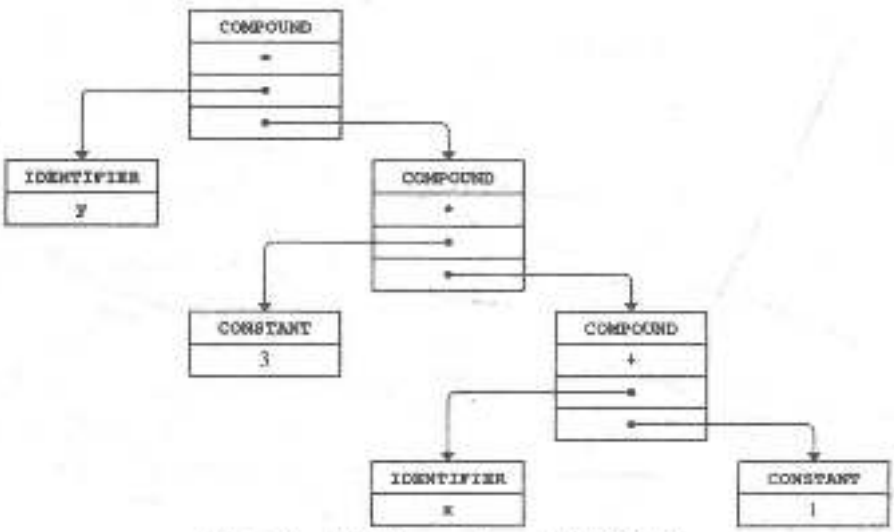
因为复合节点包括了自身可以成为复合节点的子表达式，所以一个表达式树既可以变得极其复杂，也可以变得极其简单。图 19-12 说明了以下表达式的内部数据结构：

```
y = 3 * (x + 1)
```

这一结构包含了三个操作符，也因此需要三个复合节点。在表达式树中并不显式地出现圆括号，因为树结构中已经清楚地说明了计算的次序。

19.3.9 方法的实现

几个表达式类共有的方法已经被实现。图 19-13 中是 Expression 类层次的实现。



857 图 19-12 表达式 y=3*（x+1）的内部结构

```
/*
 * File: exp.cpp
 * -------------
 * This file implements the exp.h interface.
 */

#include <string>
#include "error.h"
#include "exp.h"
#include "strlib.h"
```

图 19-13 exp.h 接口的实现

```
using namespace std;

/*
 * Implementation notes: Expression
 * --------------------------------
 * The Expression class itself implements only those methods that
 * are not designated as pure virtual methods.
 */

Expression::Expression() {
   /* Empty */
}

Expression::~Expression() {
   /* Empty */
}

int Expression::getConstantValue() {
   error("getConstantValue: Illegal expression type");
   return 0;
}

std::string Expression::getIdentifierName() {
   error("getIdentifierName: Illegal expression type");
   return "";
}

std::string Expression::getOperator() {
   error("getOperator: Illegal expression type");
   return "";
}

Expression *Expression::getLHS() {
   error("getLHS: Illegal expression type");
   return NULL;
}

Expression *Expression::getRHS() {
   error("getRHS: Illegal expression type");
   return NULL;
}


 * Implementation notes: ConstantExp
 * ---------------------------------
 * The ConstantExp subclass represents an integer constant.  The eval
 * method simply returns that value.
 */

ConstantExp::ConstantExp(int value) {
   this->value = value;
}

int ConstantExp::eval(EvaluationContext & context) {
   return value;
}

string ConstantExp::toString() {
   return integerToString(value);
}

ExpressionType ConstantExp::getType() {
   return CONSTANT;
}

int ConstantExp::getConstantValue() {
   return value;
}

/*
 * Implementation notes: IdentifierExp
 * -----------------------------------
 * The IdentifierExp subclass represents a variable name.  The
 * implementation of eval looks up that name in the evaluation context.
 */

IdentifierExp::IdentifierExp(string name) {
```

图 19-13 （续）

```
   this->name = name;
}

int IdentifierExp::eval(EvaluationContext & context) {
   if (!context.isDefined(name)) error(name + " is undefined");
   return context.getValue(name);
}

string IdentifierExp::toString() {
   return name;
}

ExpressionType IdentifierExp::getType() {
   return IDENTIFIER;
}

string IdentifierExp::getIdentifierName() {
   return name;
}

/*
 * Implementation notes: CompoundExp
 * ---------------------------------
 * The implementation of eval for CompoundExp evaluates the left and right
 * subexpressions recursively and then applies the operator.  Assignment is
 * treated as a special case because it does not evaluate the left operand.
 */

CompoundExp::CompoundExp(string op, Expression *lhs, Expression *rhs) {
   this->op = op;
   this->lhs = lhs;
   this->rhs = rhs;
}

CompoundExp::~CompoundExp() {
   delete lhs;
   delete rhs;
}

int CompoundExp::eval(EvaluationContext & context) {
   int right = rhs->eval(context);
   if (op == "=") {
      context.setValue(lhs->getIdentifierName(), right);
      return right;
   }
   int left = lhs->eval(context);
   if (op == "+") return left + right;
   if (op == "-") return left - right;
   if (op == "*") return left * right;
   if (op == "/") {
      if (right == 0) error("Division by 0");
      return left / right;
   }
   error("Illegal operator in expression");
   return 0;
}

string CompoundExp::toString() {
   return '(' + lhs->toString() + ' ' + op + ' ' + rhs->toString() + ')';
}

ExpressionType CompoundExp::getType() {
   return COMPOUND;
}

string CompoundExp::getOperator() {
   return op;
}

Expression *CompoundExp::getLHS() {
   return lhs;
}

Expression *CompoundExp::getRHS() {
   return rhs;
}

/*
 * Implementation notes: EvaluationContext
```

图 19-13 （续）

```
 * ---------------------------------------
 * The methods in the EvaluationContext class simply call the appropriate
 * method on the map used to represent the symbol table.
 */

void EvaluationContext::setValue(string var, int value) {
   symbolTable.put(var, value);
}

int EvaluationContext::getValue(string var) {
   return symbolTable.get(var);
}

bool EvaluationContext::isDefined(string var) {
   return symbolTable.containsKey(var);
}
```

图 19-13 （续）

Expression 类的实现包括了一个空构造函数、一个空析构函数和默认实现的取值方法。在 exp.h 文件中，析构函数使用 virtual 关键字进行标记，这一语法应该被应用在所有其他使用了动态内存分配的继承层次类中。将一个析构函数标记为虚函数确保了子类可以提供自己的析构函数，并且自身的析构函数与父类的析构函数会被同时调用。如果用户调用了取值方法，它会报告一个错误。因此每一个子类都可以重置取值方法来应用到自身数据域，并从 Expression 类中继承其他取值方法。

图 19-13 中其他子类的实现采用了同一种模式。每个子类都定义了需要传入接口中声明参数的构造函数，这些构造函数使用传入参数来初始化对应的实例变量。剩下大多数子类方法的实现遵循了前一个类的结构。

eval 方法的实现在每个表达式类型中都不一样。常量表达式的值是其节点中存储的整型数值。标识符表达式的值是通过数值计算阶段生成的符号表得来的。复合表达式的值需要递归运算得到。每一个复合表达式由操作符和两个子表达式组成。对于四则操作符号（+、-、* 和 /）来说，eval 使用递归运算分别计算出左子式和右子式的值，之后再将这些值应用到对应的操作符上。但是对于赋值操作符（=），eval 通过将标识符右边的值赋值给标识符左边的变量，并更新符号表。

858 | 861

19.4 解析表达式

从一个记号输入流中提取出一棵正确的解析树并不是一件简单的事。建立一个有效的语法解释器在很大程度上已经超出了本书讨论的范围。即使这样，我们依旧可以在这一问题上取得一些进展，并且将其应用到有限的四则运算中。

19.4.1 语句解析和语法

早期的编程语言中，程序员在实现编译器的解释器部分时，并没有过多地考虑实现过程的本质思想。所以，早期的解释器程序很难编写，也更难进行编译。但是在 20 世纪 60 年代，计算机科学家从一个更加理论化的层面对解释器进行了研究，这一研究简化了解释器编写的难度。现在，一个学习过编译原理的计算机科学家可以很容易地为一个编程语言创建一个解释器。实际上，我们可以根据一个编程语言的定义和需求自动生成一个解释器。在计算机科学领域，语言解析这一课题最容易看到理论对实践产生的深远影响。没有理论研究来简化这一问题，编程语言就不会取得如此大的进步。

简化语言解析的基础理论实际上借鉴于传统的语言学。与人类语言类似，编程语言也具有定义语言语法结构的语法规则。除此之外，与人类语言相比，编程语言更依赖于语言的结构，所以，通常来说，我们更容易做到将编程语言的语法结构描述为特定的形式，而这一形式也称为**语法**（grammar）。在编程语言中，语法规则向我们描述了怎样从简单的语句出发，构造出更加复杂的语言结构。

如果你从英语语法出发研究表达式形式，很容易就可以写出本章所要求的简单表达式语法规则。部分原因是在这里我们可以部分简化这一解释器，简化将体现在项（term）这一概念上，我们将其看成是一个独立单元，并且可以在一个更长的表达式中以操作数的形式出现。例如常量或者变量都是项。此外，括号中的表达式会看作是一个独立单元，因此该表达式也可以看成一个项。所以，项表示了以下的其中一种含义：

- 一个整型常量
- 一个变量
- 一个括号中的表达式

862

之后，我们可以将表达式看成以下的其中一项：

- 一个项
- 被操作符分割的两个表达式

这一非正式定义可以被直接转化为以下语法，并表示成**巴克斯－诺尔范式**（Backus-Naur Form），英文简写为 BNF，这由它的发明者约翰·巴科斯和彼得·诺尔命名得来，其形式如下：

```
E → T                    T → integer
E → E op E               T → identifier
                         T → ( E )
```

在这一语法中，像 E 或 T 这样的大写字母称为**非终止符**（nonterminal symbol），这些符号代表了抽象的语言学类型，如一个表达式或者一个项。特殊的标点符号和斜体字表示一个**终结符**（terminal symbol），这些符号出现在记号流中。类似最后一条语法规则中的圆括号这样的明确终结符必须像规则中一样确切地出现在输入流中。斜体字表示的是对应记号的占位符。因此，符号 integer 表示了扫描器以记号形式返回的数字字符串。每一个终结符都对应着扫描器输入流中的一个记号。每一个非终结符都对应着一连串特定顺序的记号。

19.4.2 考虑运算的优先级

与 19.3.3 一节中介绍的非正式表达式定义类似，语法可以用来生成解析树。与这些规则相同，这一语法在书写的时候带有二义性，并且可以生成多个带有相同序列记号但结构不同的解析树。我们发现，语法并没有解决操作符与操作数之间的结合性问题。从带二义性的语法中生成正确的解析树需要解释器拥有优先级信息。

定义优先级的最简单方式就是赋予每个操作符一个表示优先级的数值，较高的优先级权值代表着操作符与操作数之间的联系更加紧密。对于四则操作符以及赋值操作符来说，这一优先级信息可以通过以下代码进行处理：

```
int precedence(string token) {
   if (token == "=") return 1;
   if (token == "+" || token == "-") return 2;
   if (token == "*" || token == "/") return 3;
   return 0;
}
```

863

如果调用 precedence 函数并传入一个不与任何一个合法操作符相匹配的记号，函数将返回数值 0。因此你可以调用 precedence 函数并通过其返回值来判断一个记号是否为合法的操作符。

19.4.3 递归下降语法分析器

当今大多数针对编程语言语法的语法分析器都采用解析生成器（parser generator）而自动生成。但是对简单语法的处理，人工编程实现一个语法分析器并不困难。最适用的策略是编写一个负责读取语法中所有非终结符的函数。表达式语法使用非终结符 E 和 T，所以我们的语法分析器中必须包含函数 readE() 和 readT()。这些函数都需要传入一个记号扫描器类型的参数来完成对输入资源中记号的读取。在检查记号是否违反语法规则的时候，我们通常需要判断该记号所适用的规则，至少对于简单的语法检查来说是这样。这一选择操作在 readE 函数需要获得记号的当前优先级信息的时候将起到关键作用。

readE 函数和 readT 函数是相互递归的。当 readE 函数需要获取一个项时，它将调用 readT 函数。同样，readT 函数通过调用 readE 函数来完成读取括号中表达式的任务，使用相互递归函数实现的语法分析器称为递归下降语法分析器（recursive-descent parser）。

在相互递归过程中，readE 函数和 readT 函数通过调用合适的表达式类构造器生成表达式树。例如，如果 readT 函数发现一个整型记号，该函数将分配一个包含该值的 ConstantExp 类型节点。递归函数调用链中各函数的返回值将组成一个表达式树并将此树返回。

语法分析器模块的实现如图 19-14 所示。其中比较复杂的部分是 readE 函数的代码实现，因为该函数需要将优先级纳入函数操作范围。只要在新操作符优先级比 readE 函数调用者提供的当前优先级高的情况下，readE 函数可以创建一个组合表达式节点，并将子表达式放置在该操作符的左子树和右子树中，之后函数将返回并继续检查下一个操作符。当 readE 函数检测到输入到达末尾或者操作符优先级小于等于当前优先级的情况时，该函数将返回到 readE 函数调用链中下一个更高层的位置，该位置优先级将比之前更低。在这样做之前，readE 函数必须将刚才还未处理的操作符记号放回到扫描器输入流中，使得在到达合适优先级层次时该记号可以被再次读取。这一工作是通过调用 TokenScanner 类中的 saveToken 函数来完成的。 864

```
/*
 * File: parser.cpp
 * ----------------
 * This file implements the parser.h interface.
 */

#include <iostream>
#include <string>
#include "error.h"
#include "exp.h"
#include "parser.h"
#include "strlib.h"
#include "tokenscanner.h"
using namespace std;

/*
 * Implementation notes: parseExp
 * ------------------------------
 * This code just reads an expression and then checks for extra tokens.
 */
```

图 19-14 表达式语法分析器的实现

```
Expression *parseExp(TokenScanner & scanner) {
   Expression *exp = readE(scanner, 0);
   if (scanner.hasMoreTokens()) {
      error("Unexpected token \"" + scanner.nextToken() + "\"");
   }
   return exp;
}

/*
 * Implementation notes: readE
 * Usage: exp = readE(scanner, prec);
 * ----------------------------------
 * The implementation of readE uses precedence to resolve the ambiguity in
 * the grammar.  At each level, the parser reads operators and subexpressions
 * until it finds an operator whose precedence is greater than that of the
 * prevailing one.  When a higher-precedence operator is found, readE calls
 * itself recursively to read that subexpression as a unit.
 */

Expression *readE(TokenScanner & scanner, int prec) {
   Expression *exp = readT(scanner);
   string token;
   while (true) {
      token = scanner.nextToken();
      int tprec = precedence(token);
      if (tprec <= prec) break;
      Expression *rhs = readE(scanner, tprec);
      exp = new CompoundExp(token, exp, rhs);
   }
   scanner.saveToken(token);
   return exp;
}

/*
 * Implementation notes: readT
 * ---------------------------
 * This function scans a term, which is either an integer, an identifier,
 * or a parenthesized subexpression.
 */

Expression *readT(TokenScanner & scanner) {
   string token = scanner.nextToken();
   TokenType type = scanner.getTokenType(token);
   if (type == WORD) return new IdentifierExp(token);
   if (type == NUMBER) return new ConstantExp(stringToInteger(token));
   if (token != "(") error("Unexpected token \"" + token + "\"");
   Expression *exp = readE(scanner, 0);
   if (scanner.nextToken() != ")") {
      error("Unbalanced parentheses");
   }
   return exp;
}

/*
 * Implementation notes: precedence
 * --------------------------------
 * This function checks the token against each of the defined operators
 * and returns the appropriate precedence value.
 */

int precedence(string token) {
   if (token == "=") return 1;
   if (token == "+" || token == "-") return 2;
   if (token == "*" || token == "/") return 3;
   return 0;
}
```

图 19-14 （续）

以我的教学经验，想要在不观察 readE 函数运行实例的情况下理解该函数的代码几乎是不可能的。在这一节的剩余部分将分析扫描器中调用 parseExp 函数时，程序后台处理的细节：

```
odd = 2 * n + 1
```

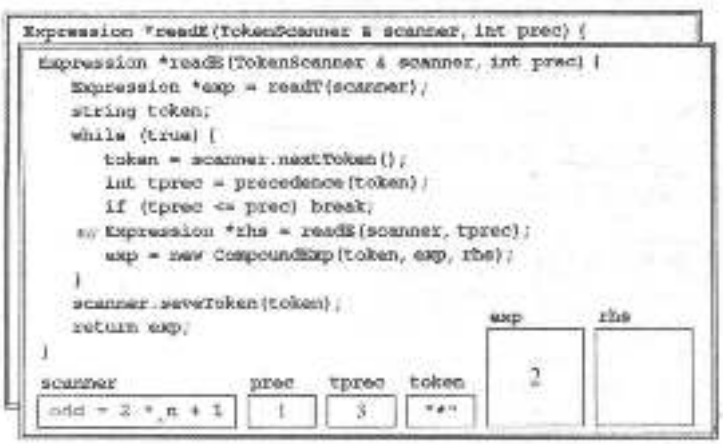
在这个表达式中，首先计算乘法，之后是加法，最后是对变量进行赋值。其中，我们感兴趣的是：语法分析器是怎样生成这一运算顺序并构造正确的表达式树的。

一次一行地分析这一表达式的处理过程将显得太过复杂。我们采用一个更加实际有效的方法来进行分析，这一方法只列出处理过程中的几个关键点。更特别的是，我们必须了解 nextToken 函数和 readT 函数的实现细节，并对这两个函数每一次的调用结果进行精确预测。 865 - 866

在第一次调用 readE 函数时，代码将读取第一项和在其之后的记号，在这里，读取到的记号是赋值操作符。－操作符对应的优先级为 1，这比当前优先级 0 更大。因此第一次调用 readE 函数后程序状态如下图所示：

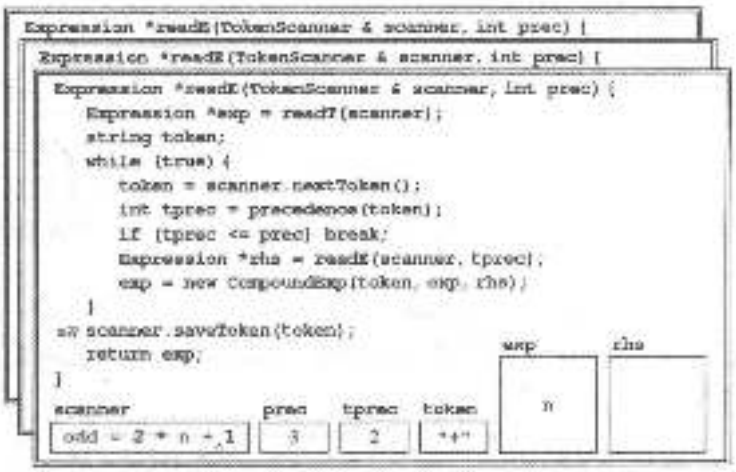


接下来，语法分析器必须读取赋值操作符的右操作数，这一操作需要通过递归调用 readE 函数来完成。该次调用与之前类似，但是调用的当前优先级不同，是 1。运算完成后程序处于如下状态：



这一优先级别的处理过程只负责读取以记号 2 开始的子表达式。在该层调用之下的栈帧结构依旧保留着之前语法分析器的工作状态使得递归调用得以实现。

在这里，语法分析器将执行另一次对 readE 函数的递归调用，并传入 * 操作符的优先级，该优先级的值为 3。然而在该次调用之后，记号流中的下一个是 + 操作符，其优先级比该次调用的优先级要低，所以程序将退出这一递归循环进入以下状态： 867

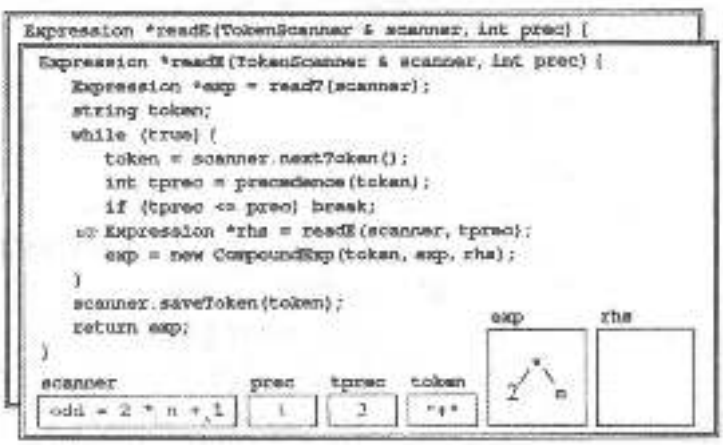


语法分析器将在记号流中保存 + 操作符，并将表达式标记 n 返回到最近一次函数调用点。readE 函数的调用结果给变量 rhs 进行了赋值如下图所示：



执行完这一步，语法分析器将合并 exp 和 rhs 表达式，使之成为一个新的组合表达式。在这一阶段计算产生的值还不足以作为程序运行的最终值加以返回，但是该值将被赋值给变量 exp。之后语法分析器再次执行 while 循环，并第二次读取记号 +。然而在这一次循环中，+ 记号将比赋值操作符拥有更高的优先级，所以程序的状态将产生如下改变：

868

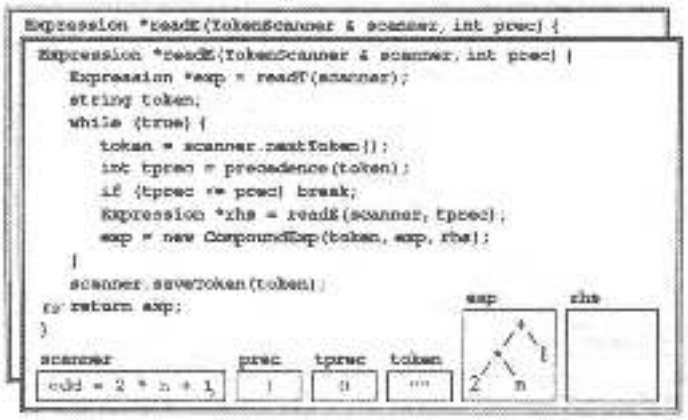


如果需要，虽然你可以认真核查上述步骤，但是现在你可以应用递归的跳步跳跃这一机制。此时，扫描器仅包含一个单一的记号，即整数 1。假定你已经看到了语法分析器读取了整数 2，那么你就能够跳到下一行继续执行：



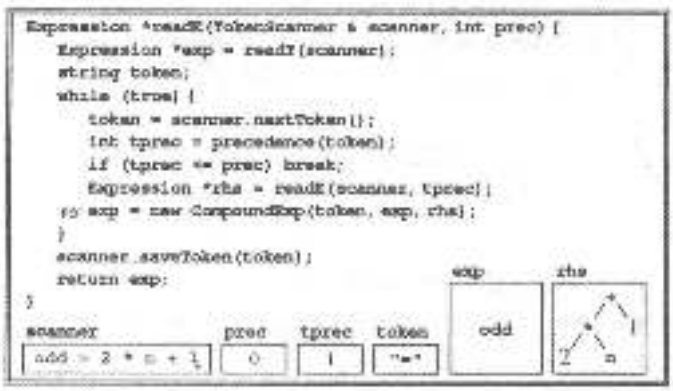
语法分析器将 exp 和 rhs 的值组合成一个新的复合表达式，然后开始 while 的下一个循环。

在下一个循环，token 为空字符串以此标志记号流的结束。由于空字符串并不是一个合法的操作符，因此，precedence 函数的返回值为 0，它表明该操作的优先级比普通的优先级更低。因此语法分析器从 while 循环中退出，并得到了如下的程序执行状态：

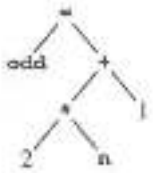


869

当程序的控制权回到我们第一次调用的 readE 函数上时，完成最后计算的所有必要信息已经准备完毕：



readR 函数的任务就是再次读取一个空记号，创建一个新的复合表达式，同时向 parseExp 函数返回表达式树的最终版本：



870

19.5 多重继承

C++ 语言与其他大多数面向对象语言的最大区别在于：C++ 语言中的类可以继承自多个父类。这一特性被称为**多重继承**（multiple inheritance）。虽然多重继承使得类继承变得更加难以理解，但是这一特性在 C++ 类库中被多次应用，所以本书将对其进行介绍。

19.5.1 stream 类库中的多重继承

虽然第 4 章已经对流类进行了介绍，但是在涉及多重继承这一特性时，我们只是一笔带过。在流类库中包含了同时处理输入流和输出流的类，我们在这些类中应用了多重继承。图 19-15 是之前图 4-7 中 UML 图的一个新版本，该图向我们说明了整个流类层次。最底层的 fstream 类同时也是一个 iostream 类，可以看到，iostream 类的左箭头指向了 istream 类。所以 fstream 类将继承 istream 类中的所有方法。与此同时，我们可以追溯 iostream 类的右箭头，发现 fstream 类同时也是 ostream 类型，这也意味着 fstream 类继承了 ostream 类的所有方法。

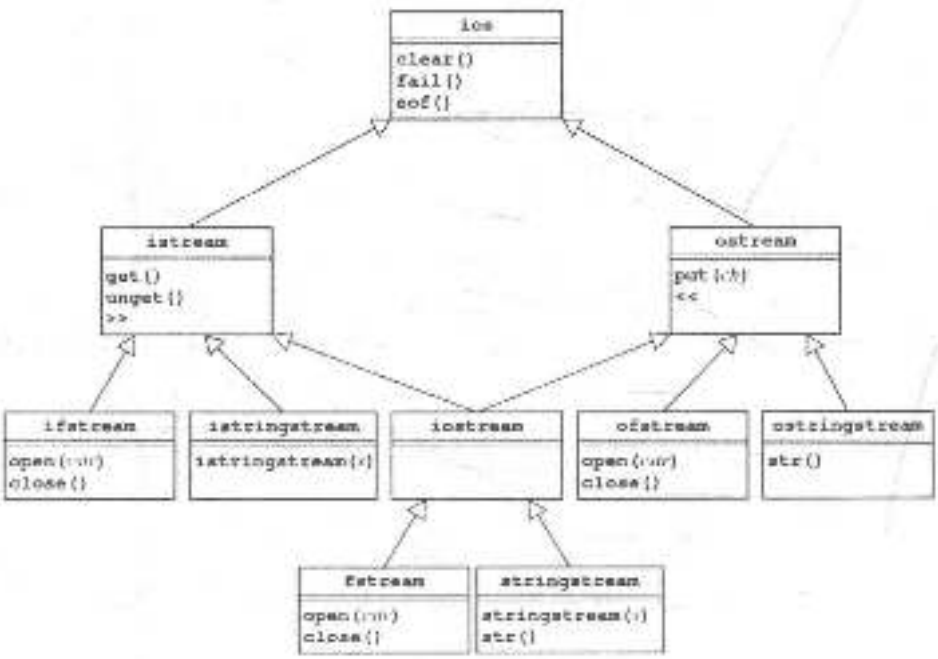


图 19-15 流类的多重继承层次

作为一个例子，我们来了解一下如何使用双向数据流。假设你需要编写一个 roundToSignificantDigits 函数，该函数为变量 x 传入一个 float 类型的数值，并将其转换

为一个具有指定位数有效数字的小数。例如，如果你这样调用函数：

```
roundToSignificantDigits(3.14159265, 5)
```

函数将返回一个只保留五位有效数字的圆周率 π 值，该值为 3.1416。虽然我们可以通过其他方式来编写这一函数，但是最简单的实现方法是使用流类中提供的函数。你需要将变量 x 写入到设定了接受数字有效位数的输出流中，之后将该值从输入流中读出并重新转换为数值。下面的代码就运用了这一策略，并使用了既继承自 istream 类又继承自 ostream 类的 stringstream 类对象来保存中间字符串：

```
double roundToSignificantDigits(double x, int nDigits) {
   stringstream ss;
   ss << setprecision(nDigits) << x;
   ss >> x;
   return x;
}
```

19.5.2 在 GObject 继承层次中添加 GFillable 类

为了证明多重继承在编程应用中具有重要的作用，我们必须回顾 19.2 节介绍的 GObject 类。之前我们早已完成该类的继承层次，其中 GRect 和 GOval 类都提供了 setFilled 方法，该方法允许用户对图形对象是空心还是实心进行设置。默认情况下，绘制出的矩形和椭圆形是空心的，但是用户可以通过调用 setFilled(true) 方法对这一设置进行更改。

图 19-4 中的代码实现了 GRect 类和 GOval 类的 setFilled 方法，图中该方法的不同版本实现代码是一样的。通常，有经验的程序员如非必要，会尽量回避重复代码。而多重继承则提供了用于削减重复代码的机制。在上述例子中，GRect 类和 GOval 类已经继承了 GObject 类，如果它们再继承一个称为 GFillable 的类，则 setFilled 方法可以选择定义在 GFillable 类中。图 19-17 向我们展示了描绘图 19-16 中 GObject 类层次的 UML 图和 GFillabel 类的实现代码。

871 ↓ 872

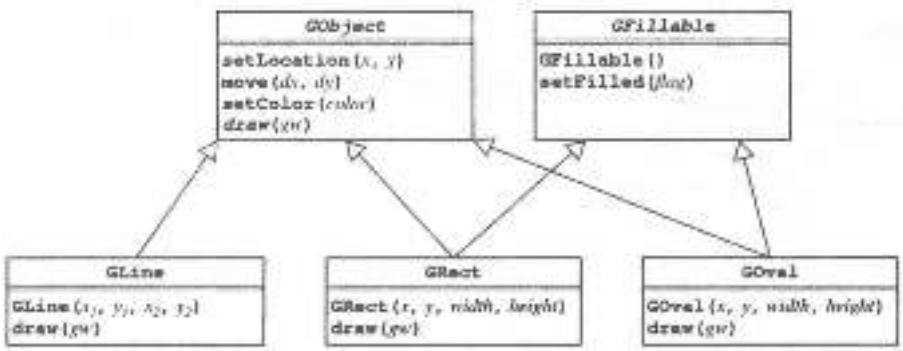


图 19-16 包含 GFillable 类的图形类层次

```
class GFillable {
public:
/*
```

图 19-17 GFillable 类

```
 * Constructor: GFillable
 * ----------------------
 * Ensures that fillable shapes are created as outlines by default.
 */

   GFillable() {
      fillFlag = false;
   }

/*
 * Method: setFilled
 * Usage: shape.setFilled(flag);
 * -----------------------------
 * Sets the fill status for shape, where false is outlined and true is filled.
 */

   void setFilled(bool flag) {
      fillFlag = flag;
   }

protected:

   bool fillFlag;          /* Flag is false for outline, true for solid fill */

};
```

873

图 19-17 （续）

19.5.3 多重继承的危险性

虽然多重继承的概念在抽象层面上并不难理解，但是将其应用到实际编程中却可能导致一些程序变得复杂和模糊不清。例如，多重继承可能导致多个父类中拥有相同的方法名和数据域名，这一结果使得父类被继承时名称的使用发生混乱，我们很难判断一个方法或者数据域到底继承自哪一个父类。这一问题应该引起足够的重视，所以 Java 语言的发明者在设计编程语言时决定不采用多重继承机制。

虽然多重继承机制已经应用在 C++ 语言编程中的部分场合，考虑到在 C++ 语言中单重继承在应用时已经足够复杂，采用多重继承机制将进一步增加 C++ 语言的复杂性和编程时错误概率，所以我们最好避免使用多重继承，与此同时，又应能够完全理解应用这一机制的代码。

本章小结

在这一章，你已经学习了如何使用 C++ 语言中的继承，也接触了对这一概念的多个实际应用。特别是 19.3 节和 19.4 节，我们了解了编译器编写人员是如何运用继承层次结构来表示算术表达式的。

本章的重点包括：

- C++ 语言允许子类继承父类的公有和保护部分。在这一继承方式中，定义子类的 C++ 语法如下所示：

```
class subclass : public superclass {
   new entries for the subclass
};
```

- 与大多数面向对象编程语言相反，C++ 语言并不能将子类对象赋值给父类对象，这样做将导致数据丢失。实际上，将对象指针而不是对象本身应用到继承中是很有用的。不幸的是，指针的直接访问和操作要求用户在内存管理上担负更多责任，这一特点也使得 C++ 语言的继承机制变得更加难以使用。

- 类中的方法并不会因为子类中的方法定义而被自动重置。在C++语言中，只有使用关键字 virtual 进行标记的方法会被重置。
- 只在子类中有实现的函数称为纯虚（pure virtual）函数。这类函数在接口声明文件中的函数原型之后使用 =0 进行标记。 874
- C++语言中的类同时包含了 protected 部分、public 部分和 private 部分。protected 部分中的声明只在类本身以及子类中起作用，而且用户不能访问。
- 调用子类的构造函数时，将同时调用父类的构造函数。在缺少其他定义的情况下，即使用户提供的初始化列表所包含的信息足以调用其他版本构造函数，C++也将调用默认构造函数。
- C++语言中包含了 try 语句，该语句允许编程人员对程序运行过程中发生的异常状态作出响应。try 语句的最简化形式如下所示：

```
try {
   code under the control of the try statement
} catch (type var) {
   code to respond to an exception with the specified value type
}
```

为了抛出一个异常，我们使用关键字 throw，并在后面附上一个值。

- 编程语言中的表达式具有递归结构。简单的表达式包括了常量和变量名。更复杂的表达式将简单的子表达式组合起来构成一个更大的单元，该单元具有层次结构，可以表示为一棵树。
- 继承机制使得定义一组用来表示表达式树节点，并具有特定结构层次的类变得更加简单。
- 从用户处读取表达式的过程可以被分为以下几个阶段：输入、词法分析和语法分析。输入阶段最简单，只是从用户处读取一个字符串。词法分析将字符串划分为记号形式，这些记号与我们第6章中介绍的 TokenScanner 类处理的记号具有相似模式。语法分析阶段将词法分析阶段返回的记号转换为内部表现形式，并在其中附上称为语法句型表达规则。
- 对于大多数语法来说，我们可以使用递归向下策略来处理转换问题。在一个递归向下语法分析器中，语法规则被编码成一系列相互递归的函数。
- 一旦转换完毕，我们可以在表达式树上进行与第16章相似的递归操作。在解释器执行过程中，最重要的一个操作就是处理表达式树，该处理过程包括递归地遍历树并计算其最终值。
- C++语言允许类继承自多个父类，这一技术称为多重继承。虽然多重继承在某些情况下能起到重要的作用，但是这一机制增加了程序的复杂性，这一复杂性体现在C++语言自身设计以及应用这一机制的程序编写中。 875

复习题

1. 在C++语言中，你会如何定义 Sub 类的首部，使该类继承了 Super 类所有公有部分？
2. 判断题：在定义一个新类时，该类的父类可以是不带有实例化模板类型的模板类。
3. 判断题：与大多数面向对象编程语言类似，C++子类中方法的定义将自动重置父类中的方法。
4. 使用你自己的语言描述关键词 virtual 的作用。

5. 什么是纯虚方法？这一结构具有什么作用？
6. C++ 语言中使用什么语法来标记一个纯虚方法？
7. 什么是抽象类？抽象类可以提供自身方法的定义吗？
8. 术语切片是什么意思？
9. 当你将继承层次中某个类的对象存储到集合中时，是否有必要使用指向分配在存储空间别处的该对象的指针，还是直接存储该对象本身？
10. 图 19-3 中 GObject 类继承层次中有哪些类和方法是虚拟方法？
11. 类中 protected 部分的访问度与 public 和 private 部分有何区别？
12. 什么是初始化列表？该列表出现在 C++ 语言编程的什么地方？
13. 解释器和编译器之间有什么区别？
14. 什么是读取－求值－输出循环？
15. 读取表达式包含了哪几个阶段？
16. 什么是异常？
17. 在只捕获一个异常类的最简单情况下，C++ 语言中 try 语句的语法是什么样的？
876 18. 陈述本章介绍的表达式递归定义。
19. 根据本章的表达式定义，判断以下每一行是合法表达式：

a. ((10)))
b. 2x + 3y
c. x - (y * (x / y))
d. -y
e. x = (y = 2 * x - 3 * y)
f. 10 - 9 + 8 / 7 * 6 - 5 + 4 * 3 / 2 - 1

20. 给上一道题中所有合法表达式画一棵语法解析树，该树必须反映出数学中的标准操作符优先级。
21. 对于 19 题中合法表达式来说，哪一个与简单的表达式递归定义原则相违背。
22. 分析树和表达式树之间有什么不同？
23. 表达式树中可以出现哪三种类型的表达式？
24. 判断题：exp.h 接口中的方法并不直接获取 Expression 对象，而是使用指针指向 Expression 对象。
25. Expression 类中有哪些公共方法？
26. 使用图 19-12 作为模型，画出以下表达式的结构图：

y = (x + 1) / (x - 2)

27. 为什么说语法在编程语言中具有重要的作用？
28. BNF 这一缩写中的每个字母都有什么含义？
29. 在语法中，终结符和非终结符有什么不同？
30. 什么是递归下降语法分析器？
31. 在已经实现的语法分析器中，readE 函数的第二个参数有什么意义？
32. 观察图 19-14 中的 readT 函数定义，你会发现函数体中不包含任何对 readT 函数本身的调用。请
877 问 readT 函数是否是一个递归函数？
33. 在 CompoundExp 子类的实现中，为何在运算操作符中使用不同的方法处理 - 操作符。
34. 什么是多重继承？
35. 判断题：多重继承在 C++ 中起到了重要的作用，所以 Java 语言的设计者在设计 Java 语言时也采用了这一机制。

习题

1. 通过在 Employee 类的私有部分中加入必要的实例变量，并实现类中的方法补全 Employee 类继承层次的定义。设计一个简单的程序来测试你的代码。
2. 在 GObject 继承层次中增加一个新的 Square 子类，该子类的构造函数需要传入图形左上角坐标和图形的大小。
3. 在图 19-5 的 GObject 类中增加纯虚方法：

```
virtual bool contains(double x, double y) = 0;
```

之后在各子类中实现这一方法。对于子类 GRect 类和 GOval 类来说，如果定义的点在对象图形中，则 contains 方法必须返回 true，如果在图形外，则返回 false。对于 GLine 类来说，当点线距离在半个像素以内时，contains 方法返回 true。如果你并不确定怎样判断一个点是否在 GOval 对象中，或者不知道怎样计算点到线的距离，你应该像一个专业程序员一样：在网上查找答案。

4. 图 19-8 中的 TestGObjects 程序使用一个矢量来保存图形对象。这一技术有效地将多个对象封装在一个类中。请实现一个新的 DisplayList 类，该类继承自 Vector<GObject *> 类，同时也提供了更多像图 19-18 中一样可以更好处理图像的方法。
5. 从第 4 题中 DisplayList 类的实现开始，在类中增加一个 getElementAt(x,y) 方法，返回一个最接近图形窗口前端图形大小的 GObject 对象指针，该对象包含了点 (x,y)。为了完成这一目标，你需要使用习题 3 中的 gobjects.h 接口，并应用其中的 contains 方法。
6. 将图 19-17 中的 GFillable 类的代码集成到 GObject 继承层次中。编写一个测试程序显示所有实心和空心的图形。 878

```
/*
 * File: displaylist.h
 * -------------------
 * This file defines a DisplayList class that maintains a list of graphical
 * objects.
 */

#ifndef _displaylist_h
#define _displaylist_h

#include "gobjects.h"
#include "gwindow.h"

/*
 * Class: DisplayList
 * ------------------
 * This class is a vector of graphical objects arranged from back to front.
 * The individual elements of the DisplayList are pointers to GObjects.
 */

class DisplayList : public Vector<GObject *> {

public:

/*
 * Methods: moveToFront, moveToBack, moveForward, moveBackward
 * Usage: list.moveToFront(obj);
 *        list.moveToBack(obj);
 *        list.moveForward(obj);
 *        list.moveBackward(obj);
 * ----------------------------------
 * These methods change the position of obj in the DisplayList.  The first
 * two methods move the object all the way to the specified end.  The last
 * two move it one position in the indicated direction.  If possible.  Each
 * of these method signals an error if obj is not in the DisplayList.
 */

   void moveToFront(GObject *obj);
   void moveToBack(GObject *obj);
```

图 19-18 displaylist.h 接口

```
   void moveForward(GObject *obj);
   void moveBackward(GObject *obj);

/*
 * Method: draw
 * Usage: list.draw(gw);
 * ---------------------
 * Draws the GObjects in the DisplayList on the graphics window.  The
 * objects are drawn from back to front, so that objects closer to the
 * front seem to cover those further back.
 */

   void draw(GWindow & gw) const;

};

#endif
```

879

图 19-18 （续）

7. 在有关后台追踪的讨论中，第 9 章介绍了一个基于最小最大算法的两人游戏实现大纲。在之前的章节中，要将最小最大算法进行封装，同时又允许代码被许多不同游戏共享是很困难的。模板和继承的结合使得为两人游戏定义一个易于被其他游戏进行扩展的基类变得更加简单。

   设计并实现一个叫做 TwoPlayerGame 的模板类，该类获取一个特定类型的参数，该类型是一个模板类，表示游戏中的一步移动。其他类可以通过重置方法来扩展该类并应用于特定的游戏，其中最小最大算法只在函数中单独实现。举例来说，第 9 章中介绍的拿子游戏类定义如下：

```
class NimGame : public TwoPlayerGame<NimMove> {
   code specific to the Nim game
};
```

   NimMove 类型与拿子游戏中实现的 Move 类型定义方式一致。在第 9 章介绍最大最小算法时，该类型被命名为 Move。声明模板类使得类型的名称变得更加具体。

   通过完成对 Nim 游戏的定义来测试 TwoPlayerGame 类的实现。当你的 Nim 游戏定义完成并成功运行后，实现第 9 章习题中另一个两人游戏来证明该类的灵活性。

8. 对 19.3 节中介绍的解释器进行必要的改造，使得表达式中可以包括操作符 %，这一操作符与 * 和 / 具有相同的优先级。

9. 改造你的解释器，使得程序可以计算 double 类型的数值，而不是 int 类型的数值。

10. 使用 exp.h 接口中提供的 Expression 类继承层次，编写一个函数：

```
void listVariables(Expression *exp);
```

   它输出表达式中的变量名称，变量名每行输出一个，并且按字母顺序排列。例如，如果你对以下表达式进行转换：

880

```
3 * x * x - 4 * x - 2 * a + y
```

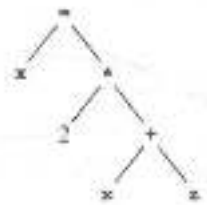
   调用 listVariables 函数将产生如下输出：



11. 在数学中，一些应用场合需要你将公式中所有实例变量使用另外的变量进行替换。以 exp.h 用户的身份，编写函数：

```
Expression *changeVariable(Expression *exp,
                           string oldName,
                           string newName);
```

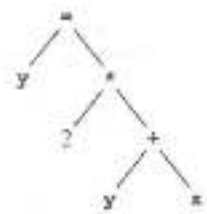
它返回一个与 exp 相同的表达式，但是其中所有 oldName 标识符被替换为 newName。举例来说，如果 exp 是如下表达式：



调用以下语句：

```
Expression *newExp = changeVariable(exp, "x", "y");
```

将为 newExp 赋值如下表达式：



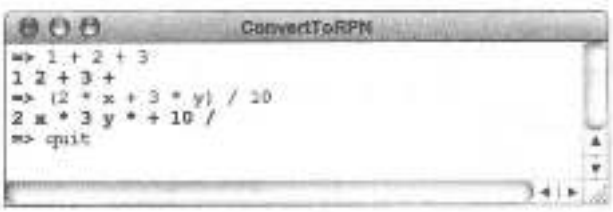
当你实现该函数时，你必须做到新的表达式树与原来的树没有公共节点。如果节点是共享的，将不能随时调用 delete 操作来释放堆内存，因为这么做将使得另一棵树的节点被释放。 881

12. 在 19.3 节介绍的表达式解释器中，每一个操作符都是一个二元操作符，需要作用于两个操作数。大多数编程语言允许使用只传入一个操作数的一元操作符。改进解释器使得解释器支持一元操作符 -。
13. 编写函数：

```
bool expMatch(Expression *e1, Expression *e2);
```

该函数在 e1 与 e2 这两个表达式拥有相同结构、相同操作符、相同常量和相同标识符名，并且都拥有相同顺序的情况下，返回 true。如果表达式树中有任何一个层级是不相同的，函数将返回 false。

14. 编写一个从用户处读取具有标准数学输入形式表达式的程序，该程序读入表达式后使用逆波兰式形式输出输入的表达式，在逆波兰式表达式中，操作符在其应用的操作数之后显示（逆波兰式表达式在第 4 章的运算应用中进行了介绍）。你的程序必须可以实现下面这一执行实例：



15. 在转换一个表达式之后，商业编译器通常要找到一个简化表达式的方法以提高表达式的计算效率。这一过程是通用技术里的一个部分，称为优化（optimization），编译器在这一阶段将尽量使生成的

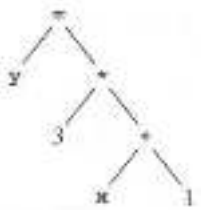
代码效率更高。优化过程中最常用的一个技术就是常数合并（constant folding）。常数合并将检查组成每个表达式的子表达式，并使用该子表达式的计算结果代替表达式本身。例如，如果编译器处理以下表达式：

```
days = 24 * 60 * 60 * sec
```

在程序运行时，我们不需要执行计算前面两个乘法的代码。子表达式 24*60*60 的值是一个常量，
882 并且可以在编译期生成实际代码之前使用数值（86400）进行代替。

编写函数 foldConstants（exp），该函数获取一个表达式指针，并且返回另一个指向个新表达式的指针。在新表达式中，所有由常量组成的表达式已经被计算结果代替。

16. 将表达式由内部表示形式转换为文本模式这一过程被称为**反解析**（unparsing）。编写一个 unparse（exp）函数，该函数将表达式 exp 用标准数学形式显示在屏幕上。同时，只有在需要考虑优先级的情况下才会给表达式加上括号。因此以下表达式树：



应该被反解析为：

```
y = 3 * (x + 1)
```

17. 虽然本章介绍的解释器程序比一完整个编译器要更容易实现，但是我们还是必须通过定义一个简单地称为**堆栈机**（stack machine）的电脑系统来了解编译器的工作细节。一个堆栈机在内部栈中进行操作，该栈由硬件进行维护，并且其运行特征与第 5 章中的逆波兰式计算相类似。根据以下图 19-19 中的指令，假设我们要使用堆栈机来解决这一问题。

LOAD #n	将常量n压入栈中
LOAD var	将变量var的值压入栈中
STORE var	在没有出栈的情况下将栈顶的值赋值给变量var
DISPLAY	弹出栈顶并显示结果
ADD SUB MUL DIV	这些指令弹出栈顶的两个元素并对这两个值进行特定的运算，将运算结果再压回到栈中。其中，第一个弹出的值为右操作数，后弹出的值为左操作数

883 图 19-19 堆栈机中实现的指令

首先编写一个函数：

```
void compile(istream & infile, ostream & outfile);
```

该函数从 infile 中读取表达式，并向 outfile 中写入一系列堆栈机指令，这些指令可以执行与输入表达式相同的表达式运算操作，并且显示到输出结果中。例如，如果 infile 文件中内容如下：

```
x = 7
y = 5
2 * x + 3 * y
```

调用 compile（infile，outfile）将向文件写入如下代码：

```
LOAD #7
STORZ x
DISPLAY
LOAD #5
STORZ y
DISPLAY
LOAD #2
LOAD x
MUL
LOAD #3
LOAD y
MUL
ADD
DISPLAY
```

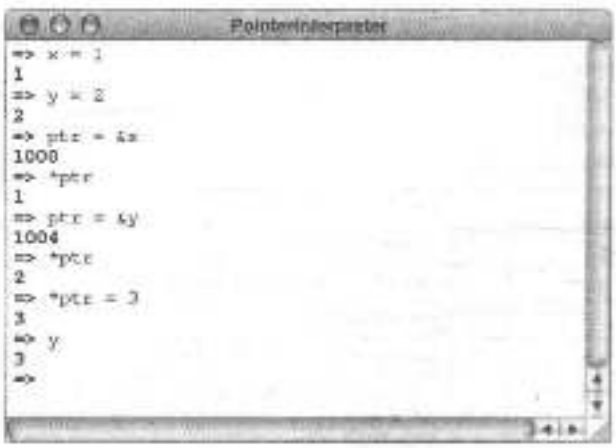
18. EvaluationContext 类中的符号表与标识符名称代表的值相对应。在实际的编译器中，符号表映射到标识符名所代表的地址上，该地址存有相应的值。你可以模仿这一过程，改造 EvaluationContext 类，使得该类提供以下函数：

```
int getAddress(string name);
int getValue(int addr);
void setValue(int addr, int value);
```

第一个函数将在内部符号表中查找标识符名，返回一个包含变量值的数字地址。如果标识符名在之前没有出现过，getAddress 应该生成一个新的地址，并且返回该地址的值。getValue 方法和 setValue 方法获取地址值而不是地址名称。但是其执行结果应该与获取地址名时的执行结果一样。

请将这一设计运用到 19.3 节中的解释器上。

19. 从习题 18 中基于地址实现的解释器开始，加入一元操作符 & 和 *，用于操作指针类型值。在增加这些操作符之后，你的解释器必须能够生成以下运行实例：



地址值的生成是任意的，取决于你的 getAddress 函数实现中如何赋值一个新地址。在该实现中，我们使用 4 位整数作为地址，使得这些值更容易辨认。

20. 用树结构来表示表达式使复杂数学运算成为可能，我们在运算中将复杂数学表达式转换为树形结构。例如，编写一个函数，应据表达式导数运算规则对表达式进行求导。最常用的数学表达式求导规则显示在图 19-20 中。

$x' = 1$	其中：
$c' = 0$	x是用于求导某准的变量
$(u+v)' = u' + v'$	c是一个常量或变量，它与x无关
$(u-v)' = u' - v'$	u和v是任意表达式
$(uv)' = uv' + vu'$	
$(u/v)' = \frac{uv' - vu'}{v^2}$	

885

图 19-20 导数标准公式

编写一个应用了图 19-20 中规则的递归函数 `differentiate(exp,var)`。该函数将计算表达式 `exp` 的 `var` 所导数。`differentiate` 函数的返回值是一个可以用于任何其他地方的 `Expression` 指针。例如，你可以计算该表达式，或者将其传入 `differentiate` 函数中进行二次求导。

21. 扩展解释器，使得解释器可以应用于 BASIC 语言的简单语句处理。该语言在 20 世纪 60 年代由约翰·科姆尼（John Kemeny）和托马斯·克茨（Thomas Kurtz）开发。在 BASIC 语言中，程序的每一行以一个数字开头。该数字决定了程序语句的执行顺序。每一行都包括了图 19-21 中展示的语句。其中使用 `RUN` 命令来代替使程序运行的语句，程序将从数字最小的行开始。一直运行到 `END` 语句，该语句标志着程序的结束。例如，以下例子将输出 500 以内 2 的乘方。



`LET` *var* = *exp*	将*exp*的值赋给变量*var*
`INPUT` *var*	从用户输入取一个输入数值，并将该值赋给变量*var*
`PRINT` *exp*	在控制台上输出*exp*的值
`GOTO` *line*	跳转到指定的行数，并从该行继续执行
`IF` e_1 *op* e_2 `THEN` *line*	如果运算的值为真，将控制权移交到程序指定的行数。其中操作符可以为任何标准关系操作符
`END`	标记程序的结束

886

图 19-21 BASIC 语言中的简单语句

第 20 章
Programming Abstractions in C++

迭代策略

何物需要迭代？

—威廉·莎士比亚，《奥赛罗》，1603 887

集合中最重要的一个操作就是迭代访问它的每一个元素。现在，你已经使用过基于范围的 for 循环来实现该目的，它使代码十分简明，可读性强。然而这个特性是 C++ 2011 标准新增的。在介绍 for 循环扩展版本之前，我们可以使用迭代器（iterator）来遍历访问 C++ 集合中的每一个元素，该迭代器指向一个集合中的一个特定元素，并且每次可以通过单步递进的方式来访问其他元素。

由于 C++ 的大部分历史原因，基于范围的 for 循环是无法使用的，迭代器在以前的代码中十分普遍。此外，大部分用来处理集合类有用的库函数使用迭代器来指定该函数运用在一个集合类的某一具体部分。基于这些原因，为了使用 C++ 高效的编程，理解迭代器的使用是很有必要的。然而，理解底层的实现细节并不重要。因此，这章集中讨论迭代器如何解决用户的问题，以及在应用中如何使用它们。本章的最后一节介绍实现的细节，它只与你需要执行并支持迭代器的类有关。

在关于迭代器是以用户为本还是以实现为本的讨论中，本章介绍了一种不同的模型，该模型为一个集合类中的每一个元素运用一种操作。为了代替使用迭代器或基于范围的 for 循环去遍历其中的每一个元素，一种可选的策略是允许用户依次地对集合类中的每个元素运用一个函数。以这种方式使用的函数被称为映射函数（mapping function）。映射函数没有迭代器方便，因此很少使用。然而它们更易于实现。此外，映射函数对于计算机科学（特别是随着大规模并行应用程序的开发）愈加重要。

20.1 使用迭代器

在 STL 和 Stanford 类库中，每一个集合类导出了一个名为 iterator 的类，这个类为该集合类提供了一个迭代器。从语法上讲，C++ 的迭代器类似于指针。例如，迭代器使用 * 操作符获取当前元素的值，++ 操作符让迭代器指向下一个元素。如果你对这些操作感觉生疏，有必要复习一下第 11 章，它向你介绍了使用迭代器所需要的惯用模式。

20.1.1 简单的迭代器例子

在进行 iterator 类及其操作的详细讨论之前，考虑一个使用迭代器来单步访问集合类中各元素的简单例子是很有用的。第 5 章介绍的第一个基于范围的 for 循环的例子使用 888
了以下代码，它列出了存储在 EnglishWords.dat 字典中两个字母构成的单词：

```
int main() {
   Lexicon english("EnglishWords.dat");
   for (string word : english) {
      if (word.length() == 2) {
         cout << word << endl;
```

```
            }
        }
        return 0;
    }
```

使用了迭代器，TwoLetterWords 程序就如下所示：

```
int main() {
    Lexicon english("EnglishWords.dat");
    for (Lexicon::iterator it = english.begin();
                           it != english.end(); it++) {
        string word = *it;
        if (word.length() == 2) {
            cout << word << endl;
        }
    }
    return 0;
}
```

这个实现的核心是用来声明循环索引变量 it 的 iterator 类型。迭代器 it 从 english 字典的开头开始遍历，一次一个单词，直至到达终点。

不像你之前见到的大部分类型，iterator 并不是一个独立的类型，它是集合类的一部分而被导出的一个类型，用这种方式定义的类型称为嵌套类型（nested type）。每个集合类都有自己定义的 iterator 版本作为一个嵌套类型。因为这个 iterator 名字并不是唯一标识它所属的集合类，用户必须使用完整的名称。因此，Lexicon 类的迭代器命名为 Lexicon::iterator。同样，Vector<int>类的迭代器命名为 Vector<int>::iterator。

除了导出 iterator 类来指定该集合类之外，每一个集合类导出两个返回迭代值的方法。begin 方法返回一个初始化的迭代器，以便它指向集合类对象的第一个元素。end 方法返回一个指向其对象的最后一个元素后面不存在的元素的迭代器。因此 begin 和 end 迭代器描述了一个范围，这个范围让人联想到第 2 章介绍过的半开区间。集合类中的元素从 begin 指向的元素开始，然后递进，但是不包括 end 指向的元素。

在 C++ 中，用于迭代器的操作符和用于指针的操作符是相同的。给定一个迭代器，你可以通过使用 * 操作符来查找它指向的值，正如你用一个指针解析它所指的对象内容一样。因此，语句

```
string word = *it;
```

初始化变量 word，使它包含迭代器当前位置在字典中的字符串。

for 循环末尾的表达式 it++ 使迭代器递进，以使它指向字典中的下一个元素。同样对于指针来说，我们可以使用 ++ 操作符来增加指针值，从而使其指向数组的下一个元素。

迭代器支持 C++ 中指针实现的多种操作。例如，一些程序员可能选择使用下面的表达方式（尽管从习惯上来看这样做不是一个较好的选择）将自增和解析操作结合：

```
string word = *it++;
```

然后在 for 循环的初始位置省略自增操作。另一些程序员可能在 if 测试语句和输出语句中，通过解析迭代器的值来消除局部变量 word，如下所示：

```
for (Lexicon::iterator it = english.begin();
                       it != english.end(); it++) {
   if (it->length() == 2) {
      cout << *it << endl;
   }
}
```

与指针一样，常用模式 *it++ 意味着自增指针，但是在自增操作发生之前解析当前的值。同样地，表达式 it->length() 是 (*it).length() 的简化，对字典中当前的元素调用 length 方法。

20.1.2 迭代器的层次结构

在最低限度下，C++ 中的每一个迭代器都支持上节所描述的 * 和 ++ 操作，以及相关的 == 和 != 操作。这些操作符对于实现基于范围的 for 循环很有用，每次可以在集合中遍历一个元素。然而，一些集合类定义的迭代器支持更多的通用操作。 R90

图 20-1 展示的是迭代器在不同类层次上所支持的扩展操作，它们遵循继承层次。迭代器支持的最基本的操作为 InputIterator，它允许读值，以及 OutputIterator，它允许给解析的迭代器赋一个新值。ForwardIterator 类结合了这些功能，因此支持读值和写值。BidirectionIterator 模型增加了 -- 操作符，使得向前或向后移动迭代器成为可能。RandomAccessIterator 是非常通用的模式，它既包含了将迭代器向前迭代 n 个元素，也包括了全部的关系操作符。

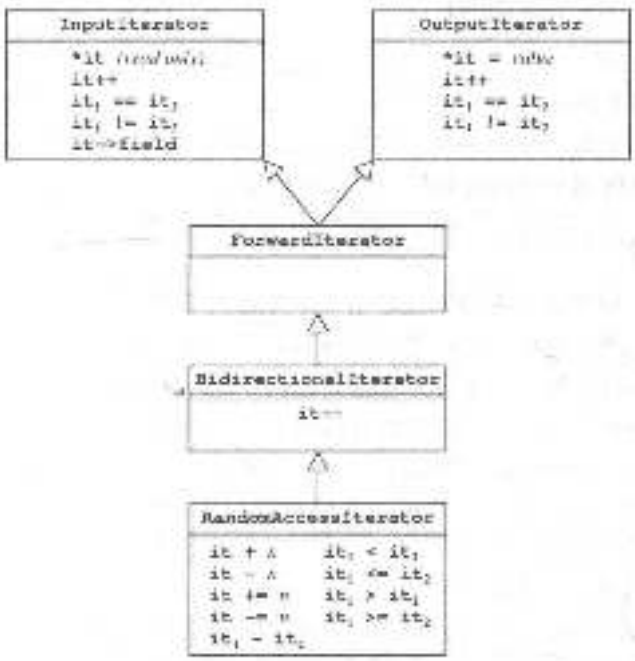


图 20-1 C++ 迭代器类层次

Lexicon 类只支持 InputIterator 层的功能，这意味着在不将其放入一些更通用的结构时，从 Lexicon 的最后开始，迭代访问该字典中的所有单词是难以实现的。相比之下，Vector 类的迭代器是 RandomAccessIterator，因此，使用下面的代码从矢量对 R91

象 v 的最后开始，迭代访问 v 中的元素是可行的，该代码在控制台上逆序输出元素：

```
Vector<int>::iterator it = v.end();
while (it != v.begin()) {
   cout << *--it << endl;
}
```

在这段代码中，在对迭代器解析引用之前自减迭代器是很重要的。it 的初始值指向矢量末尾不存在的元素。自减操作符将迭代器向前移动一位，因此指向最后一个元素。类似地，你可以使用下面的代码输出 v 中的任一元素：

```
for (Vector<int>::iterator it = v.begin();
                           it < v.end(); it += 2) {
   cout << *it << endl;
}
```

和那些尝试充分使用指针操作的代码一样，以这种不常见的方式使用迭代器最终经常会使程序难以维护。指定迭代器的最好方法是无论何时都使用基于范围的 for 循环，因为这样做能完全地隐藏迭代操作。

然而，迭代器在 C++ 中还因其他原因显得非常重要。标准模板库中的很多函数和方法都使用迭代器作为参数，或者返回结果为一个迭代器。20.4 节将会更细致地探索这些机制。然而，在此之前，它帮助引用一个更通用的编程概念，这一概念也集成进 STL 的设计中，即能够使用函数作为数据结构的一部分。

20.2 使用函数作为数据值

直到现在，你一直认为应该将函数和数据结构的概念保持相对独立。函数提供了表示一种算法的方式；数据结构允许你组织用于算法的信息。函数是算法结构的一部分，而不是数据结构的一部分。然而，能够将函数作为数据值使用，常常会使设计有效的接口变得很容
892 易，因为这种机制允许用户像指定数据一样指定操作。

20.2.1 函数指针

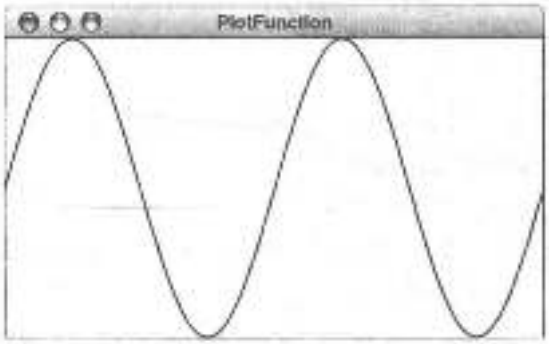
在早期计算中，程序是以代码和数据完全分开的形式表示的。典型地，代码被记录在打孔纸带上，接着输入到机器中，然后依次执行指令。如果你需要改变这个程序，就必须重打一条新的纸带。现代计算机的一个重要特征就是同一内存既被用来存储数据值，也被用来存储硬件执行的机器指令。将指令存储在内存地址中作为数据值使用的这种技术被称为**冯·诺伊曼体系结构**（von Neumann architecture）。尽管现在计算机历史学家认为冯·诺伊曼不是创造这个想法的人，但他还是第一个实现这种技术的人，因此这个概念体系被冠以他的名字。

冯·诺伊曼体系结构的一个重要理念就是程序中每一个机器指令在内存中都有一个地址。这个事实使得创建一个指向函数的指针（pointer to a function）成为可能，函数指针即为函数的第一条指令的地址。大多数现代编程语言使用指向函数的指针并对程序员隐藏其细节。与这些语言相反，C++ 允许程序员声明指向函数的指针，然后在应用中使用这些函数指针作为数据值。

20.2.2 简单的画图应用

研究如何在 C++ 语法中加入指向函数的指针细节之前，思考一个例子来展示这种技术

是如何用于实践的会很有帮助。最早解释该技术的例子是为用户指定的函数产生一个函数图形的问题。例如，假设你想要编写一个绘制函数 $f(x)$ 在给定范围 x 上的函数值的图形。例如，如果 f 是正弦函数，程序应该产生如下图所示的运行结果：



893

这个图形化输出只是展示了图的形状，并没有指明任何沿着 x 轴和 y 轴的单位。在该图中，x 的值从 -2π 变化到 2π，而 y 的值从 -1 变到 1。用户调用 plot 函数需要将范围作为参数，它意味着假定给常量 PI 合适的定义，这个调用能产生如下所示的输出：

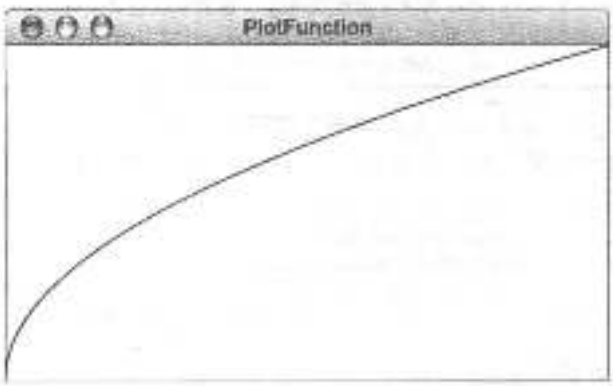
```
plot(gw, sin, -2 * PI, 2 * PI, -1, 1);
```

在这里，有意思的参数就是图像化窗口之后的参数，也就是你想要绘制的函数的名字。在这个例子中，函数是来自 <cmath> 库中的三角函数 sin。

然而，如果 plot 以一种更通用的方式来设计的话，应该可以通过改变第二个参数来绘制一个不同的函数。例如，函数调用：

```
plot(gw, sqrt, 0, 4, 0, 2);
```

应该绘制出 sqrt 函数的图形，该函数图形在 x 轴的 0 到 4 区间内，以及 y 轴的 0 到 2 区间内，如下图所示：



同时，传递给 plot 的函数不能是任意的已有函数。例如，使用字符串函数就没有任何意义，因为在 x-y 坐标系中的图形只对数字函数有意义。此外，若传递给 plot 的函数若是带有几个参数的数值函数也是没有意义的。函数只以一个实数作为参数（可能是一个 double 类型），并返回一个实数才有意义。因此，你可以说函数 plot 的第二个参数必须在能够将一个 double 类型映射为另一个 double 类型的这类函数中选择。

894

此外，这个参数必须是某种 C++ 类型的一个数据值。尽管函数本身不是数据值，而是

指向函数的指针。plot 的第二个变量因此是一个指向函数的指针，该函数以 double 类型作为它的参数，并返回一个 double 类型值。用户提供了 plot 函数的地址。plot 的实现中接着调用用户提供的函数来计算图中每一个新的 y 值。由于 plot 函数的实现与其调用者分离，因此用户提供的这个函数被称为回调函数（call function）。

20.2.3 声明函数指针

在编写 plot 函数之前，你唯一需要学习的语言新特征是其第二个参数的声明语法。你已经知道它是一个指向参数值和返回值都是 double 类型的函数指针。当你第一次遇到它时，即使它跟 C++ 其他地方使用的声明类型一样，声明一个函数指针的语法看起来也很令人困惑。理解函数指针声明的关键在于分辨一个变量的声明是反映了它的用法而不是它的结构。表 20-1 的例子阐述了这个原则。

表 20-1 读 C++ 的声明

`double x;`	在这个简单声明中，变量 x 是 double 类型
`double list[n];`	这个声明指出当 list 后面跟着一个被方括号包围的数字时，其结果是 double 类型。变量 list 因此是一个 double 类型的数组
`double *dp;`	表达式 *dp 是 double 类型，这意味着 dp 必须是一个指向 double 类型的指针
`double **dpp;`	表达式 **dpp 是 double 类型，这意味着 dpp 必须是一个指向 double 类型指针的指针
`double f(double);`	如果 f(exp) 以一个 double 类型的参数出现，那么函数的返回结果为 double 类型。因此，这个声明是以一个 double 类型作为参数，并返回一个 double 类型的结果的函数 f 的原型
`double *g(double);`	如果表达式 *g(exp) 出现在代码中，那么其结果为 double 类型。因为圆括号的优先级高于星号，该声明是返回一个指向 double 类型指针的函数 g 的原型
`double (*fn)(double);`	与上一个声明相反，解析引用操作在函数调用之前就被应用。因此变量 fn 是一个返回值为 double 类型函数的指针。由于 C++ 对函数指针自动解析引用，因此调用该函数通常被写为 fn(exp)

895

表 20-1 的最后一行声明了一个指向函数的指针 fn，它指向一个以 double 类型为参数并返回一个 double 类型的函数，它也是完成 plot 函数原型所需要的参数，如下所示：

```
void plot(GWindow & gw, double (*fn)(double),
                        double minX, double maxX,
                        double minY, double maxY);
```

这些参数分别是图形化窗口，被绘制的函数以及图形在 x 和 y 轴方向上的界限。

20.2.4 实现 plot 函数

一旦你定义了函数原型，就可以使用第 2 章中介绍的图形库编写一个如图 20-2 所示的简单 plot 实现。这个实现对图形窗口中的每一个像素循环遍历，将每一个在 minX 和 maxX 间隔位置上的 x 坐标转变为相应的 y 坐标。

例如，在窗口的中间点对应着 minX 和 maxX 的中间值。程序接着调用函数 fn 去计算 y 的值，如下所示：

```
double y = fn(x);
```

```
/*
 * Function: plot
 * Usage: plot(gw, fn, minX, maxX, minY, maxY);
 * ------------------------------------------------
 * Plots the specified function (which must map one double to another
 * double) on the screen.  The remaining arguments indicate the range
 * of values in the x and y directions, respectively.
 */

template <typename FunctionClass>
void plot(GWindow & gw, FunctionClass fn,
                        double minX, double maxX,
                        double minY, double maxY) {
   double width = gw.getWidth();
   double height = gw.getHeight();
   double nSteps = int(width);
   double dx = (maxX - minX) / nSteps;
   double sx0 = 0;
   double sy0 = height - (fn(minX) - minY) / (maxY - minY) * height;
   for (int i = 1; i < nSteps; i++) {
      double x = minX + i * dx;
      double y = fn(x);
      double sx1 = (x - minX) / (maxX - minX) * width;
      double sy1 = height - (y - minY) / (maxY - minY) * height;
      gw.drawLine(sx0, sy0, sx1, sy1);
      sx0 = sx1;
      sy0 = sy1;
   }
}
```

图 20-2 plot 函数的实现

最后一步通过将 y 值与 minY 和 maxY 进行比较，从而将 y 值转换成屏幕垂直方向上合适位置上的像素点。这个操作本质上是将 x 的值进行反转。唯一的不同是屏幕上的 y 轴被转化为传统的笛卡尔积坐标平面，它使得从图形视窗的高度中减去计算值十分必要。

plot 的执行开始于计算窗口左边的点的坐标，将结果存储在变量 sx0 和 sy0 中，从中进一步正确地计算曲线中的每一个像素的坐标，将这些坐标存储在变量 sx1 和 sy1 中。图形库接着调用连接这些点的一个函数：

```
gw.drawLine(sx0, sy0, sx1, sy1);
```

下一次循环将连接当前点与它的前驱点。这个过程与通过连接一系列的线段有同样的效果，每一线段在 x 方向延长一个像素。

图 20-2 中的 plot 函数几乎肯定太原始，不能应用于实际的应用程序。虽然如此，它展示了一个如何将函数看作数据值的有用的示例，这在应用中十分有效。

20.2.5 映射函数

回调函数提供了另一种策略来迭代集合类对象中的元素。如果一个类提供了一种允许用户对一个集合类对象中的每一个元素调用一个函数的方法，用户可以使用这个方法作为一个可选的方案去使用一个迭代器或是一个基于范围的 for 循环。允许你在集合类对象的每个元素上调用一个被称为**映射函数**（mapping function）的方法。

例如，Lexicon 类中提供：

```
void mapAll(void (*fn)(string));
```

它是以一个形参类型为 string 类型的函数指针作为其参数的映射函数。mapAll 的作用是对字典上每一个单词调用特定方法，使其和 Lexicon 迭代器处理单词有着相同的顺序。

896 / 897

在 Lexicon 类中映射函数 mapAll 的存在使得可以重新编写 TwoLetterWords 程序，如下所示：

```
int main() {
   Lexicon english("EnglishWords.dat");
   english.mapAll(printTwoLetterWords);
   return 0;
}
```

这里的 printTwoLetterWords 定义为：

```
void printTwoLetterWords(string word) {
   if (word.length() == 2) {
      cout << word << endl;
   }
}
```

mapAll 映射函数为字典中的每一个元素调用 printTwoLetterWords。回调函数检查单词的长度是否为两个字母，如果是则输出。

在 Stanford 类库中，每个集合类都定义了一个称为 mapAll 的映射函数，它为每一个元素调用一个回调函数。当使用基于范围的 for 循环迭代集合类对象中的元素时，调用的顺序与规定的顺序一致。因此，对于一个矢量对象而言，mapAll 按照元素的索引顺序来调用回调函数，对于一个 Grid 对象来说，mapAll 按照行优先的顺序来调用回调函数，对于一个 Lexicon 对象，按照字母表的顺序来调用回调函数，对于一个 Map 对象，按照键值递增的顺序来调用回调函数，对于一个 Set 对象，按照元素值递增的顺序来调用回调函数。

回调函数本身通常选取一个与其参数的类型相匹配的类作为其参数。这个规则的一个例外就是键－值匹配函数。对于 Map 类和 HashMap 类，回调函数需要两个参数，第一个参数代表了一个键，另外一个参数则是相对应的值。例如，你可以使用图 20-3 中的 listMap 函数来列出 Map<string, int> 中所有的键－值对。

除了图 20-2 中使用的传值参数以外，在 Stanford 集合类库中的 mapAll 函数也接受使用常量引用作为实参的映射函数。因此，如果你想要通过消除对键的拷贝来增加代码的效率，你可以使用下面 listMapEntry 的原型：

```
void listMapEntry(const string & key, const int & value);
```

898 这两个函数类型在 C++ 中有不同的特征，mapAll 方法设计成可以使用任意类型的参数。

```
/*
 * Function: listMap
 * Usage: listMap(map);
 * ------------------------
 * Displays the key-value pairs in the map.  The output appears in
 * lexicographic order because the Map class uses the ordering of
 * the key type.
 */

void listMap(const Map<string,int> & map) {
   map.mapAll(listMapEntry);
}

/*
 * Function: listMapEntry
 * Usage: listMap(key, value);
 * ---------------------------
 * Prints a single key-value pair.  This function is designed to be
 * used as a callback function for the mapAll method in the Map class.
 */

void listMapEntry(string key, int value) {
   cout << key << " = " << value << endl;
}
```

图 20-3 Map<string,int> 的功能方法列表

20.2.6 实现映射函数

和迭代器相比，20.6 节中你将有机会进行更加详细的思考，映射函数相对容易实现。假如你使用图 14-11 中基于矢量实现，可以像图 20-4 所示的那样来实现 mapAll 函数。只要你定义了一个递归的辅助方法，对于 Map 类来说，实现 mapAll 就相当容易了。图 20-5 中的代码对 Map 类的实现做了如下的假设：像第 16 章中描述的那样，Map 类基于二叉搜索树，二叉搜索树的根存储在实例变量 root 中，BSTNode 类型包含了键和值的字段。

```
/*
 * Implementation notes: mapAll
 * ----------------------------
 * This method uses a for loop to call fn on every element.
 */

template <typename ValueType>
void Vector<ValueType>::mapAll(void (*fn)(ValueType)) const {
   for (int i = 0; i < count; i++) {
      fn(array[i]);
   }
}
```

图 20-4 Vector 类的 mapAll 函数实现

899

```
/*
 * Implementation notes: mapAll
 * ----------------------------
 * The exported version of mapAll uses a private helper method that takes
 * the tree as an argument and performs a standard inorder traversal,
 * calling fn(key, value) for every key-value pair.
 */

template <typename KeyType,typename ValueType>
void Map<KeyType,ValueType>::mapAll(void (*fn)(KeyType, ValueType)) const {
   mapAll(root, fn);
}

template <typename KeyType,typename ValueType>
void Map<KeyType,ValueType>::mapAll(BSTNode *t,
                                    void (*fn)(KeyType, ValueType)) const {
   if (t != NULL) {
      mapAll(t->left, fn);
      fn(t->key, t->value);
      mapAll(t->right, fn);
   }
}
```

图 20-5 Map 类的 mapAll 函数实现

20.2.7 回调函数的限制

到目前为止，你见过的最简单形式的回调函数并不难理解。事实上，就程序结构而言，将对一个集合类对象中的每一个元素进行遍历的任务与每个循环中执行的代码隔离是很有用的。然而，这个策略也有严格的限制。基本问题是用户通常向回调函数中所传递的信息超过了集合类所能提供的参数。使用回调函数使得该过程很困难。

为了说明这个问题，同时使 TwoLetterWords 问题一般化，以利于程序列出一个指定长度的所有单词，而并不一定是两个字母长度的单词。更具体地说，这个例子的目的是编写一个函数：

```
void listWordsOfLengthK(const Lexicon & lex, int k)
```

900 就像函数名所暗示的一样，该函数列出字典中长度为 k 的所有单词。

如果你使用基于范围的 for 循环，这个函数就很容易编写：

```
void listWordsOfLength(const Lexicon & lex, int k) {
   for (string word : lex) {
      if (word.length() == k) {
         cout << word << endl;
      }
   }
}
```

如果你尝试使用 mapAll 函数来代替，会遇到大量问题。回调函数需要获取 k 的值，但是使用 mapAll 并没有明显的方式来传递该信息。为了确保回调函数有其所需要的信息，用户必须以某种方式向映射函数传递 k。反过来函数返回长度与传递的值相同的所有单词。在某种程度上，这种情况使人想起路易斯·卡罗尔的《走到镜子里》(*Through the Looking Class*) 中的红心皇后的观察："它选取你可以做的任何喜欢的事情，保持在同一地点。"用户需要提供回调函数所需的数据，但是必须依赖于集合类向用户的回调函数提供数据。幸运的是，C++ 提供了一个解决这个问题的合理方法，我们将在下一节中描述。

20.3 用函数封装数据

函数指针在它们的效用中被限制，因为它们无法将函数与用户提供的数据一起提供。对于大多数应用来说，你需要这样一种策略：它将回调函数与用户提供的数据封装成一个单独的单元。在计算机科学中，一个函数与其相关的数据的组合被称为闭包 (closure)。

一些语言通过允许在函数定义的同时使用函数内部的变量来支持闭包。但是 C++ 并不支持这种模式。为了在 C++ 中使用闭包，你需要自己创建必要的数据结构。尽管这个过程主传递一个简单的函数指针要更为复杂，但是闭包还是很重要的，因此，花费一些时间理解闭包如何工作是很值得的。

20.3.1 使用对象模拟闭包

在展现 C++ 程序员通常创建闭包的策略之前，告知 C++ 已经提供了一种将数据与代码封装在一个单独的实体里的机制是很值得的。这个机制称为对象。对于大多数情况而言，提
901 供的这个封装恰好就是对象。类中的变量存储数据，同时方法提供代码。

为了解决列出所有 k 个字母的单词问题，你需要定义一个新的类，它将 k 存储在一个变量中，但也要提供一个方法，如果单词的长度与存储的 k 值相匹配，该方法就从字典中输出当前的单词。在 listWordsOfLength 的实现中，你可以初始化创建一个包含期望值 k 的对象。接下来如果可以将这个对象传递给映射函数，你就完全解决了这个问题。

这个解决办法的唯一难点在于：映射函数需要知道对于集合类中的每个值应该调用的方法名。为了确保映射函数尽可能地一般化，为此定义一个一致的方法名是很有意义的。尽管任何方法名都可以，但最简单的策略是通过重载 operator() 将对象本身作为方法。这个定义意味着像函数那样"调用"一个对象。在 C++ 中，重载这个操作符的类叫做函数类 (function class)。这些类的实例称为函数对象 (function object) 或函子 (functor)。

20.3.2 函数对象的简单例子

为了确保你理解函数对象是如何工作的，用一个与映射函数无关的简单例子作为开始是有意义的。假定你允许用户创建并调用一个类似于函数对象的对象，该对象以一个整型作为参数，并且在加上用户选中的增量后返回该参数的值。对于给定的增量，这个函数在C++中很容易编写。例如，函数对其参数值加1：

```
int add1(int x) {
   return x + 1;
}
```

这个例子的目的可能与最初的目的有一点不同。给定一个整型常量k，你允许用户定义函数：

```
int addk(int x) {
   return x + k;
}
```

你不可能实现这种形式的所有可能的函数，因为这里会有与参数数目相同的函数。你需要创建一个封装两个构件的函数类：一个变量记录k的值；另一个重载operator()，以便该操作符向其参数增加存储的k值。图20-6中是AddKFunction类的实现。构造函数创建了一个指定增量的AddKFunction的新实例；重载operator()给用户提供的参数增加了存储的值。

902

```
/*
 * Class: AddKFunction
 * -------------------
 * This class defines a function object that takes a single integer x and
 * computes the value x + k, where k is a constant specified by the client.
 */

class AddKFunction {

public:

/*
 * Constructor: AddKFunction
 * Usage: AddKFunction addk = AddKFunction(k);
 * -------------------------------------------
 * Creates a function object that adds k to its argument.
 */

   AddKFunction(int k) {
      this->k = k;
   }

/*
 * Operator: ()
 * ------------
 * Defines the behavior of an AddKFunction object when it is called
 * as a function.
 */

   int operator()(int x) {
      return x + k;
   }

private:

   int k;      /* Instance variable that keeps track of the increment value */

};
```

图20-6 向参数中增加一个常量的函数类

下面的主程序通过以下操作提供了一个对 AddKFunction 类的简单说明：

```
int main() {
   AddKFunction add1 = AddKFunction(1);
   AddKFunction add17 = AddKFunction(17);
   cout << "add1(100) -> " << add1(100) << endl;
   cout << "add17(25) -> " << add17(25) << endl;
   return 0;
}
```

903

用图 20-6 中 AddKFunction 的定义运行这个程序，在控制窗口产生以下输出：



C++ 中的函数对象十分有用，对于函数调用来说，用户可以使用常规的语法来调用它们。例如，在 AddKFunction 类的测试程序中，局部变量 add1 和 add17 是 AddKFunction 类的不同实例。然而调用这些函数对象看起来就像传统的函数调用。表达式 add1(100) 调用 100 次 add1，正如它已经被定义成一个常见函数。

20.3.3 向映射函数传递函数对象

使用函数对象的策略可以解决向回调函数传递额外信息的问题。除了函数指针之外，Stanford 集合类中的 mapAll 函数（在标准模板库中有与它对应的部分）允许其参数为一个函数对象，只要该函数对象的 operator() 方法被重载为与你想要传递给映射函数相同的参数。例如，你可以对一个以一个字符串作为参数的函数对象的 Lexicon 类调用 mapAll 函数，这意味着对应的函数类必须重载方法：

```
void operator()(string);
```

图 20-7 定义了你需要编写的 listWordsOfLengthK 的函数类，它本身只要一行代码：

```
void listWordsOfLengthK(const Lexicon & lex, int k) {
   lex.mapAll(ListKLetterWords(k));
}
```

```
/*
 * Class: ListKLetterWords
 * -----------------------
 * This class defines a function object that takes a word and prints it
 * on the console if it has length k, where k is specified by the client.
 */

class ListKLetterWords {

public:

/*
 * Constructor: ListKLetterWords
 * Usage: ListKLetterWords fn = ListKLetterWords(k);
 * ------------------------------------------------
 * Creates a function object that prints its argument only if it has
 * length k.  This function object is used as the argument to the mapAll
 * method in the Lexicon class.
 */
```

图 20-7 只显示 k 个字母的单词的函数类

```
   ListKLetterWords(int k) {
      this->k = k;
   }

/*
 * Operator: ()
 * ------------
 * Defines the behavior of a ListKLetterWords object when it is called
 * as a function.
 */

   int operator()(string word) {
      if (word.length() == k) {
         cout << word << endl;
      }
   }

private:

/* Instance variables */

   int k;      /* Length of desired words */

};

/*
 * Function: listWordsOfLengthK
 * Usage: listWordsOfLengthK(lex, k);
 * ----------------------------------
 * Lists all words in the specified lexicon whose length is equal to k.
 */

void listWordsOfLengthK(const Lexicon & lex, int k) {
   lex.mapAll(ListKLetterWords(k));
}
```

图 20-7 （续）

20.3.4 编写以函数作为参数的函数

从本章之前大量映射函数的实现中，如果你要传递一个函数指针，那么在 Lexicon 类中的 mapAll 函数的头部必须写成以下形式：

```
void mapAll(void (*fn)(string));
```

这个原型指出 mapAll 的参数必须是一个以字符串作为参数并且返回值为空的函数指针。

904
:
905

有趣的问题是在这种情况下应如何声明一个 mapAll 函数，它选取一个函数对象来代替函数指针。C++ 提供了一种简明的（即使有时会复杂一些）语法来声明一个函数指针类型。如果你想让 mapAll 以一个函数对象作为参数，你应该如何声明参数的类型呢？这个问题很难，因为一个函数对象可以是任意重载函数调用操作符的类的一个实例。考虑到你可以在任意类中重载这个操作符，这里似乎没有任何明确的方法声明它的类型。

C++ 通过使用模板函数来实现任何以函数对象作为参数的函数的方式来解决这个问题。因此，mapAll 的这个版本的原型如下所示：

```
template <typename FunctionClass>
void mapAll(FunctionClass fn)
```

传递给 mapAll 的值可以是任何类型。当编译器试图展开 mapAll 模板函数时，如果该类型不能重载函数调用操作符以至于不能获得期望的参数，那么编译器会产生错误信息。

20.4 STL 算法库

尽管迭代器对于其最初单步遍历集合类中的元素的目的相当有效，但是在 C++ 中它们

更加重要，因为 STL 中的大部分函数以迭代器作为参数。例如，假定你需要对一个矢量对象的元素进行排序，并且充分利用大量的 C++ 类库的设计者发明创建的一个通用的排序算法库。为此，STL 库中的确包含了一个名为 sort 的函数。然而，在第 10 章阅读过了大量的 sort 函数的实现之后，这个函数并不像你期望的那样以一个矢量对象作为参数。库中的 sort 函数版本以两个迭代器作为参数，它确定了你想要排序的矢量对象的范围。为了对矢量对象 v 的所有元素排序，需要调用：

```
sort(v.begin(), v.end());
```

如果你只想对 v 中的前 k 个元素排序，你可以调用：

```
sort(v.begin(), v.begin() + k);
```

矢量的 iterator 类实现了 RandomAccessIterator 模型，这意味着给一个迭代器增加
906 k，将会得到一个指向第 k 个元素的迭代器。

STL 库中 sort 函数是 <algorithm> 接口提供的最有用的函数之一。尽管这个接口的范围相当广泛，你可以使用表 20-2 列出的函数来获得合适的范围。正如你在表中第一行看到的使用模式，与 sort 函数的使用方法相同，这些函数中的大部分都以一对迭代器作为其参数。例如，你可以通过调用以下语句随机打乱矢量对象 v 中的元素顺序：

```
random_shuffle(v.begin(), v.end());
```

表 20-2 底部的函数以函数作为参数从而对一个集合进行操作。for_each 函数推广了映射函数的理念，并支持迭代器的任意集合类。例如，你可以通过调用：

```
for_each(lex.begin(), lex.end(), printTwoLetterWords);
```

列出字典 lex 中所有两个字母的单词。其中，这里的 printTwoLetterWords 是 20.2.5 一节中定义的回调函数。回调函数也可以是一个函数对象，这意味着你可以通过调用：

```
for_each(lex.begin(), lex.end(), ListKLetterWords(k));
```

列出 k 个字母的单词。

你可以使用 <algorithm> 接口中的函数作为更复杂的操作的构件。例如，图 20-8 中的代码以 <algorithm> 库中提供的一些高层函数作为工具重新实现了第 10 章中的选择排序算法。

```
/*
 * Function: sort
 * Usage: sort(vec);
 * -----------------
 * Sorts the elements in the vector by combining high-level operations
 * from the <algorithm> interface.  The selection sort algorithm is
 * described in Chapter 10.
 */

void sort(Vector<int> & vec) {
   for (Vector<int>::iterator lh = vec.begin(); lh != vec.end(); lh++) {
      Vector<int>::iterator rh = min_element(lh, vec.end());
      iter_swap(lh, rh);
   }
}
```

907 图 20-8 使用 <algorithm> 库实现的选择排序函数

表 20-2 <algorithm>库中的选择函数

简单的多态函数	
max(x, y)	返回 x 和 y 中的较大者
min(x, y)	返回 x 和 y 中的较小者
swap(x, y) iter_swap(i, j)	交换参数 x 和 y 的值或是交换迭代器 i 和 j 指向的值
迭代范围内操作的函数	
binary_search(*begin*, *end*, *value*)	如果在迭代 *begin* 到 *end* 的范围中包含指定值 *value*，则返回 true
copy(*begin*, *end*, *out*)	将指定迭代范围中的值拷贝给以 *out* 开始的迭代器
count(*begin*, *end*, *value*)	返回迭代范围中与指定的 *value* 相等的值的数目
fill(*begin*, *end*, *value*)	将指定迭代范围中的每一个元素置为 *value*
find(*begin*, *end*, *value*)	返回指定迭代范围中第一个与 *value* 相等的元素的迭代器，如果不存在则结束
merge(*begin*$_1$, *end*$_1$, *begin*$_2$, *end*$_2$, *out*) inplace_merge(*begin*, *middle*, *end*)	将两个已经有序的子序列合并成一个以 *out* 开始的完整的有序序列。inplace_merge 版本合并将同一集合中的两个子序列，使用 *middle* 指明第二个序列的开始
min_element(*begin*, *end*) max_element(*begin*, *end*)	返回一个指向迭代范围中最小或最大的元素的迭代器
random_shuffle(*begin*, *end*)	将迭代范围中的元素
replace(*begin*, *end*, *old*, *new*)	将迭代范围中的所有 *old* 实例变量用 *new* 替换
reverse(*begin*, *end*)	逆序指定迭代范围中的元素
sort(*begin*, *end*)	将迭代范围中的元素以升序排列
以函数式作为参数的函数，该类函数参数可以是函数对象或函数指针	
for_each(*begin*, *end*, *fn*)	对迭代范围内的每一个元素调用 *fn*
count_if(*begin*, *end*, *pred*)	返回迭代范围内调用 *pred* 返回 true 的值的数目
replace_if(*begin*, *end*, *pred*, *new*)	将迭代范围内调用 *pred* 返回 true 的所有值替换为 *new*
partition(*begin*, *end*, *pred*)	重新排列迭代范围内的元素，以至于调用 *pred* 返回 true 的所有元素排列在最开始。该函数返回一个指向边界的迭代器

908

20.5 C++ 的函数式编程

与你在第 1 章所学的一样，C++ 的设计人员在 C 语言的基础上加入了面向对象的特性。鉴于这样的历史，有人就期望 C++ 既包含面向对象的特性又包含底层的特征。相比之下，C++ 并没有设计支持另一种主流的范型，即**函数式编程**（functional programming），该范型有以下的特征：

- 程序用紧密的函数调用来表示，这些函数可以进行必要的计算，但是不会执行任何改变程序状态（例如赋值）的操作。
- 函数就是数据值，程序员可以像对待其他的数据值一样对其进行操作。

尽管函数式编程不是设计语言的一个目标，事实上 C++ 包含的模板以及函数对象使得

采用一种极其类似于函数式编程模型的编程风格变为可能。

20.5.1 STL 库 <functional> 的接口

标准模板库通过 <functional> 接口为函数式编程范型提供了基本的支持。如表 20-3 所示，<functional> 接口提供了大量的类以及方法。与表的第一部分一样，类通常分为两种。模板类 binary_function<arg1, arg2, result> 是 <functional> 库中含有两个参数的函数的共同父类，其中第一个参数的类型为 arg1 类型，第二个参数为 arg2 类型，并且返回一个 result 类型的值。类 unary_function<arg, result> 为那些只含有一个参数的函数类扮演了同样的角色。

下面二部分中的类扮演了 C++ 中为面向对象所提供的标准算术、关系、逻辑操作符的角色。记住这些实体指的是类而不是对象这一事实是非常重要的。例如，如果你想构造一个将两个整数值相加的函数对象，你就需要调用 plus<int> 类中的构造函数，如下所示：

```
plus<int>()
```

漏掉圆括号是一个易犯的错误，但是这又是在编译器中很易产生的隐式错误。

bind1st 和 bind2nd 函数可以让我们在一个函数对象中包含常量。例如，以下语句：

909
```
bind2nd(plus<int>(), 1)
```

表 20-3 <functional> 接口中的部分类和函数

基类	
binary_function<*arg1 type*, *arg2 type*, *result type*>	包含两个指定类型的参数，并且返回一个指定的结果类型的函数类的父类
unary_function<*argument type*, *result type*>	包含一个指定类型的参数，并且返回一个指定的结果类型的函数类的父类
实现算术操作符的类	
plus<*argument type*>	实现相加操作符 + 的二元函数类
minus<*argument type*>	实现相减操作符 - 的二元函数类
multiplies<*argument type*>	实现相乘操作符 * 的二元函数类
divides<*argument type*>	实现相除操作符 / 的二元函数类
modulus<*argument type*>	实现取模操作符 % 的二元函数类
negate<*argument type*>	实现取反操作符 - 的一元函数类
实现比较操作的类	
equal_to<*argument type*>	实现关系操作符 == 的函数类
not_equal_to<*argument type*>	实现关系操作符 != 的函数类
less<*argument type*>	实现关系操作符 < 的函数类
less_equal<*argument type*>	实现关系操作符 <= 的函数类
greater<*argument type*>	实现关系操作符 > 的函数类
greater_equal<*argument type*>	实现关系操作符 >= 的函数类

（续）

实现逻辑关系的类	
logical_and<argument type>	实现操作符 && 的函数类
logical_or<argument type>	实现操作符 || 的函数类
logical_not<argument type>	实现操作符! 的函数类
产生函数对象的函数	
bind1st(fn, value) bind2nd(fn, value)	返回一个新的一元函数对象。该对象用其与 value 绑定的第一个参数（bind1st）或者其第二个参数（bind2nd）调用二元函数对象 fn
not1(fn) not2(fn)	返回一个新的函数对象，且该函数对象为一元函数对象 not1 时返回 true，反之为二元函数对象 not2 时返回 false
ptr_fun(fnptr)	返回一个调用特定函数指针的新函数对象，它可能带有一个或两个同类型的参数

910

返回一个一元函数对象，并且该函数对象可以向自己的参数增加常量 1。因此这个笨与通过调用你在本章前面见到的 AddKFunction(1) 产生的函数对象很相似。函数形式的优点就是你可以通过组合不同 <functional> 接口所提供的不同的函数，从而不需要再定义一个 AddKFunction 类。

bind1st 和 bind2nd 函数在与 <algorithm> 库的高层的方法结合使用时是很有用的。例如，你可以通过下面的调用计算一个元素类型为整型的矢量对象 v 中的负数的个数：

```
count_if(v.begin(), v.end(), bind2nd(less<int>(), 0))
```

ptr_fun 函数可以使函数指针和函数对象的概念一体化。如果 fnptr 是一个指向函数的指针，并且需要一个或者两个相同类型的参数，那么 ptr_fun(fnptr) 返回一个具有相同效果的函数对象。与 <algorithm> 库中函数所做的一样，当你定义一个以一个函数指针或者一个函数对象作为参数的函数时，你可以使用这个函数来避免代码重复。

为了让你了解如何应用这种技术，假设你想强化 20.2 节中 plot 函数的定义，使得其第二个参数既可以是一个函数对象，也可以是一个函数指针。为此，你要复制图 20-2 的代码，并且将该函数的头部修改为：

```
template <typename FunctionClass>
void plot(GWindow & gw, FunctionClass fn,
                        double minX, double maxX,
                        double minY, double maxY)
```

函数体不需要做任何变化。此外，plot 函数的两个版本可以共存，因为编译器可以通过调用者提供的参数类别使用哪个版本。

然而，在两个版本中使用完全相同的代码是不完美的。为了避免这样，你可以使用下面的代码重新实现函数－指针版本的 plot：

```
void plot(GWindow & gw, double (*fn)(double),
                        double minX, double maxX,
                        double minY, double maxY) {
   plot(gw, ptr_fun(fn), minX, maxX, minY, maxY);
}
```

调用 fun_ptr 从函数指针中产生了一个函数对象。这里的函数指针可以传递给另一个 plot
911 函数版本。

20.5.2 比较函数

在 <functional> 库中，比较函数尤为重要。因为它们可以让用户定义自己的排序关系。在 <algorithm> 库中的函数包括排序函数，其中就有表 20-2 中列出的 sort、merge、inplace_merge、binary_search、min_element 以及 max_element。这些函数可以获取一个可选的函数式参数来定义顺序。按照系统默认，这个参数通过调用与值类型相匹配的 less 类的构造函数而生成。为了使用不同的顺序，用户可以自己提供函数指针和函数对象。这类函数被称为比较函数（comparision function），它需要两个变量，返回一个布尔类型的值，并且当第一个值在第二个值之前出现时返回 true。

作为一个简单的例子，你可以通过下面的调用对元素类型为整型的矢量对象 vec 逆序排序：

```
sort(vec.begin(), vec.end(), greater<int>());
```

在这个调用中，比较函数是 greater<int> 的一个实例，而不是默认的 less<int> 的一个实例，这也就意味着顺序是相反的。

你可以使用函数指针或者函数对象作为比较函数。例如，如果你定义了以下函数：

```
bool lessIgnoringCase(string s1, string s2) {
   return toLowerCase(s1) < toLowerCase(s2);
}
```

在忽略大小写的情况下，你可以调用以下语句对元素类型为字符串类型的矢量对象 names 进行排序：

```
sort(names.begin(), names.end(), lessIgnoringCase());
```

用近乎同样的方法，你可以通过定义函数：

```
bool isShorter(string s1, string s2) {
   return s1.length() < s2.length();
}
```

并且调用：

```
sort(words.begin(), names.end(), isShorter());
```

对元素类型为字符串类型的矢量对象 words 按由短到长的顺序排序。

你也可以将一个比较函数传给 Map 类和 Set 类的构造函数，从而确定元素的出现次
912 序。Graph 类依靠这种机制确保了所有的节点和弧的出现次序依赖于节点名称。

20.6 迭代器的实现

上一节，你已经学到如何在 STL 库中使用迭代器。为了完备性，观察这些类型是如何实现的非常重要。因此你必须了解它在底层表示。

20.6.1 为矢量类实现迭代器

为矢量类实现迭代器提供了一个相对直接的挑战。矢量类 Vector 的底层结构定义为

一个简单的动态数组，而迭代器需要保存的唯一状态信息是当前的索引值以及一个返回矢量类对象的指针。因此，iterator 类的私有变量可以定义以下：

```
const Vector *vp;
int index;
```

变量 vp 是一个指向 const Vector 的指针，让编译器知道迭代器的操作不能改变 Vector 对象本身。Vector 类中的 begin 函数需要返回一个迭代器，该迭代器中变量 vp 指向 Vector 本身，而变量 index 置为 0。end 函数必须返回一个迭代器，该迭代器中 vp 同样被初始化指向 Vector 对象本身，而 index 被置为 Vector 对象中以 count 变量存储的元素数目。如果 iterator 类中包含一个构造函数，该构造函数获取两个参数，并且使用它们初始化相对应的实例变量，如下所示：

```
iterator(const Vector *vp, int index) {
   this->vp = vp;
   this->index = index;
}
```

那么这些函数都可以很容易实现。

Vector 类需要访问该构造函数，但是对用户来说这是不可能实现的。实现这个目标的最简单方法就是将 Vector 类声明为 iterator 类的友元。那么 begin 和 end 的实现如下所示：

```
iterator begin() const {
   return iterator(this, 0);
}

iterator end() const {
   return iterator(this, count);
}
```

从这一点来看，所有剩余的操作就是实现不同的操作符。图 20-9 表示的是为 Vector 类实现的 iterator 类的代码。 913

```
/*
 * Nested class: iterator
 * ----------------------
 * This nested class implements a standard iterator for the Vector class.
 */

   class iterator {

   public:

/*
 * Implementation notes: iterator constructor
 * ------------------------------------------
 * The default constructor for the iterator returns an invalid iterator
 * in which the vector pointer vp is set to NULL.  Iterators created by
 * the client are initialized by the constructor iterator(vp, k), which
 * appears in the private section.
 */

      iterator() {
         this->vp = NULL;
      }
```

图 20-9 Vector 在 iterator 类中的实现

```
/*
 * Implementation notes: dereference operator
 * ------------------------------------------
 * The * dereference operator returns the appropriate index position in
 * the internal array by reference.
 */

      ValueType & operator*() {
         if (vp == NULL) error("Iterator is uninitialized");
         if (index < 0 || index >= vp->count) error("Iterator out of range");
         return vp->array[index];
      }

/*
 * Implementation notes: -> operator
 * ---------------------------------
 * Overrides of the -> operator in C++ follow a special idiomatic pattern.
 * The operator takes no arguments and returns a pointer to the value.
 * The compiler then takes care of applying the -> operator to retrieve
 * the desired field.
 */

      ValueType *operator->() {
         if (vp == NULL) error("Iterator is uninitialized");
         if (index < 0 || index >= vp->count) error("Iterator out of range");
         return &vp->array[index];
      }

/*
 * Implementation notes: selection operator
 * ----------------------------------------
 * The selection operator returns the appropriate index position in
 * the internal array by reference.
 */

      ValueType & operator[](int k) {
         if (vp == NULL) error("Iterator is uninitialized");
         if (index + k < 0 || index + k >= vp->count) {
            error("Iterator out of range");
         }
         return vp->array[index + k];
      }

/*
 * Implementation notes: relational operators
 * ------------------------------------------
 * These operators compare the index field of the iterators after making
 * sure that the iterators refer to the same vector.
 */

      bool operator==(const iterator & rhs) {
         if (vp != rhs.vp) error("Iterators are in different vectors");
         return vp == rhs.vp && index == rhs.index;
      }

      bool operator!=(const iterator & rhs) {
         if (vp != rhs.vp) error("Iterators are in different vectors");
         return !(*this == rhs);
      }

      bool operator<(const iterator & rhs) {
         if (vp != rhs.vp) error("Iterators are in different vectors");
         return index < rhs.index;
      }

      bool operator<=(const iterator & rhs) {
         if (vp != rhs.vp) error("Iterators are in different vectors");
         return index <= rhs.index;
      }

      bool operator>(const iterator & rhs) {
         if (vp != rhs.vp) error("Iterators are in different vectors");
         return index > rhs.index;
      }
```

图 20-9 （续）

```
      bool operator>=(const iterator & rhs) {
         if (vp != rhs.vp) error("Iterators are in different vectors");
         return index >= rhs.index;
      }

/*
 * Implementation notes: ++ and -- operators
 * -----------------------------------------
 * These operators increment or decrement the index.  The suffix versions
 * of the operators, which are identified by taking a parameter of type
 * int that is never used, are more complicated and must copy the original
 * iterator to return the value prior to changing the count.
 */
      iterator & operator++() {
         if (vp == NULL) error("Iterator is uninitialized");
         index++;
         return *this;
      }

      iterator operator++(int) {
         iterator copy(*this);
         operator++();
         return copy;
      }

      iterator & operator--() {
         if (vp == NULL) error("Iterator is uninitialized");
         index--;
         return *this;
      }

      iterator operator--(int) {
         iterator copy(*this);
         operator--();
         return copy;
      }

/*
 * Implementation notes: arithmetic operators
 * ------------------------------------------
 * These operators update the index field by the increment value k.
 */
      iterator operator+(const int & k) {
         if (vp == NULL) error("Iterator is uninitialized");
         return iterator(vp, index + k);
      }

      iterator operator-(const int & k) {
         if (vp == NULL) error("Iterator is uninitialized");
         return iterator(vp, index - k);
      }

      int operator-(const iterator & rhs) {
         if (vp == NULL) error("Iterator is uninitialized");
         if (vp != rhs.vp) error("Iterators are in different vectors");
         return index - rhs.index;
      }

/* Private section */

   private:
      const Vector *vp;                    /* Pointer to the Vector object */
      int index;                           /* Index for this iterator      */

/*
 * Implementation notes: private constructor
 * -----------------------------------------
 * The begin and end methods use the private constructor to create iterators
 * initialized to a particular position.  The Vector class must therefore be
 * declared as a friend so that begin and end can call this constructor.
 */
      iterator(const Vector *vp, int index) {
```

图 20-9 （续）

```
        this->vp = vp;
        this->index = index;
    }

    friend class Vector;
};
```

图 20-9 （续）

图 20-9 中的函数包含了大量的检查，从而确保迭代器的合理使用。如果用户试图使用一个未初始化的迭代器，或者对一个超出范围的迭代器进行解析引用，或是对元素类型不同的 Vector 对象的迭代器进行比较，那么程序会调用 error 来报告错误。尽管 STL 库中的一些实现也是这样，但是大多数实现都不是如此。使用一个很少执行错误检查或不执行错误检查的实现会使得纠错过程很困难。

另一方面，如果你愿意放弃一些错误检查来简化代码，与图 20-9 中的代码相比较，你可以用少量的代码为 Vector 类实现其迭代器。了解如何做到这一点需要用一种不同的方式来观察 C++ 迭代器的需求，我们将在下节中讨论。

20.6.2 将指针作为迭代器

图 20-9 中代码如此之长的大部分原因是 RandomAccessIterator 提供的服务需要定义大量的操作符。为了确保迭代器的操作像用户所期望的那样，迭代器的实现必须定义下面每一个操作符：

914
|
917

```
*  ->  []  ==  !=  <  <=  >  >=  ++  --  +  -
```

然而，这里没有通过在一个类中定义方法从而实现提供这些操作符的需求。如果你已经有一个以合适的方式实现这些操作符的类型，就可以使用该类型作为一个迭代器。在 C++ 中，指针类型对所有这些操作符都能正确实现，这就意味着你可以使用指针值作为迭代器。

迭代器定义的一种重新解释就是：你可以使用 C++ 传统的数组以及 <algorithm> 接口提供的函数。例如，如果你有一个名为 array 的数组，其实际的大小存储在变量 n 中，你可以通过以下调用对该数组进行排序：

```
sort(array, array + n);
```

数组名被认为是指向该数组第一个元素的指针，指针增量 array+n 指向第 n+1 个元素。

迭代器可以是指针。这一事实提供了另一种为 Vector 类实现迭代器的一种策略。如果迭代器仅仅是指向数组中适当元素的指针，你需要像下面这样定义 begin 和 end 函数：

```
ValueType *begin() const {
   return array;
}

ValueType *end() const {
   return array + count;
}
```

那么，所有操作符都不需要定义。

然而，你必须在定义中包含一种额外的定义，为了使基于范围的 for 语句能够对 Vector 类型作用。C++ 编译器将基于范围的 for 循环：

```
for (type var : collection) {
   body of the loop
}
```

转换为下面传统的 for 循环，其中 ctype 代表集合的类型，_it 是在别处不使用的私有的迭代器变量：

```
for (ctype::iterator it = collection.begin();
                      it != collection.end(); it++) {
   type var = *it;
   body of the loop
}
```

为了保证这种转换能够起作用，这里必须有一个嵌套在 Vector 类中名为 iterator 的嵌套类型。在实现中，该类型应该简化成一个指向值类型的指针，但是必须命名为 iterator，因为这是编译器所期望的。幸运的是，C++ 可以对已经存在的类型名重新命名，这将会在下一节中进行介绍。

20.6.3 typedef 关键字

C++ 包含了一种从 C 语言继承而来的机制，这种机制允许程序员给已经存在的类型重新命名。如果你在任意的变量声明前面加关键字 typedef，编译器会定义该声明中的每一个名字为该类型名的同义词，并且该名字将作为一个变量。正如该准则说明的那样，声明：

```
char *cstring;
```

定义变量 cstring 为一个指向 char 类型的指针。typedef 声明：

```
typedef char *cstring;
```

定义类型名 cstring 为“指向 char 类型的指针类型”，因此是 char* 类型的同义词。

尽管很容易过度使用关键字 typedef，但它在某些情况下会非常有用。一种常见的应用是为函数指针类型类供一个简洁的名字。你在阅读本章的过程中可能会注意到：函数指针参数的声明通常很长并且很复杂，因此在实际中为这些参数提供一个简洁的类型名意义重大。为了减少复杂性，例如，通过在 Map 类的定义中包含以下的一行，你可以定义类型名 mapCallback，使得它是类型“一个参数类型为 KeyType 和 ValueType，并且没有返回值的函数指针”的同义词：

```
typedef void (*mapCallback)(KeyType, ValueType);
```

那么，mapAll 函数的原型就变为：

```
void mapAll(mapCallback fn)
```

从而取代了下面长的版本（尽管可以认为更有益的）：

```
void mapAll(void (*fn)(KeyType, ValueType))
```

typedef 关键词是你在完成 Vector 类的迭代器类型的改进实现中所需要的。你需要

在 Vector 类的公共部分提供下面的定义：

```
typedef ValueType *iterator;
```

20.6.4 为其他集合类实现迭代器

尽管为 Vector 类实现迭代器解决了大多数复杂问题，但是为 Vector 类实现迭代器比为大多数其他集合函数实现迭代器更加简单。为 Grid 类和 HashMap 类定义迭代器并不是太困难，你会在习题中体会到这一点。然而，为像 Map 一样具有树结构的类定义一个迭代器会变得很麻烦，主要是因为在实现中必须将递归结构转换成迭代。通常，将迭代器的实现交给专家是很明智的，就像依靠专家去编写随机数产生器、哈希函数以及排序算法会更好。

本章小结

本章使用迭代遍历一个集合类中的元素这一问题作为一个框架，从而介绍许多有趣的主题，这些主题包含**迭代器**（iterator）、**函数指针**（function point）、**函数对象**（function object）、STL<algorithm>库以及在 C++ 中使用函数编程范式的技巧。

本章包含的重要概念有：

- Stanford 类库和 Stanford 模板库中的所有集合类都提供了一个其内部嵌套的 iterator 类，它支持循环遍历一个集合类中的元素。
- 在 C++ 中，iterator 的语法基于与指针相同的操作集的计算。然而，不是所有的迭代器都支持应用于指针上的所有操作集。因此，迭代器提供不同层次的服务取决于它们定义的集合类的能力。图 20-1 中表示的是迭代器功能的层次结构。
- 除了提供本身的 iterator 类型，每个类也都提供 begin 和 end 方法，它们分别返回指向首元素和最后一个元素之后的迭代器。
- C++ 编译器将基于范围的 for 循环：

```
for (type var : collection) {
   body of the loop
}
```

转换为以下的 for 循环，其中 ctype 表示集合类元素的类型，it 是一个私有的迭代变量：

```
for (ctype::iterator it = collection.begin();
                      it != collection.end(); it++)
   type var = *it;
   body of the loop
}
```

- 在大部分现代计算机中使用的冯·诺依曼体系结构中，每一个函数存储在内存里，因此拥有一个地址。在 C++ 中，指向函数的指针是合法的数据值。
- 映射函数对一个集合类中的每一个元素调用一个用户指定的回调函数。因此，映射函数可以作为迭代器用于许多相同的上下文中，或者是基于范围的 for 循环中。
- 传统的映射函数给回调函数传递额外的数据是很困难的。为了解决这个问题，C++ 支持函数对象，它在一个实现函数调用操作符的类实例中封装必要的数据和操作。
- STL<algorithm>库中包含了对集合操作的通用算法的实现。这些函数中的大部分

需要迭代器来指定元素的范围。

- 即使 C++ 被设计使用命令式范型和面向对象的范型，它也可以被用于——尤其是在 <functional> 的辅助下——支持函数编程模型的特定方面。
- 比较函数能够对 <algorithm> 库中函数以及对像 Map 和 Set 这样的集合类指定新的顺序关系，它需要确定元素的相对顺序。
- C++ 包含关键字 typedef，它可以为已经存在的类型名定义新的名字。理解 typedef 是如何工作的最简单的方式是记住：输入一条 typedef 语句之后，如果你去掉关键字 typedef，那么该语句则声明一个新的变量。
- 为一个新的类实现迭代器需要仔细的考虑和大量的代码。通常，把这个工作交给专家可能更明智。 921

复习题

1. 假设你有一个名为 primes 的 Set<int> 变量。如何在不使用基于范围的 for 循环的情况下，使用迭代器将 primes 中的所有元素按照升序输出。
2. 假设变量 it 是一个迭代器，描述表达式 *it++ 的作用。
3. 判断题：如果 c 是一个非空集合，调用 c.begin() 返回一个指向该集合第一个元素的迭代器。
4. 判断题：如果 c 是一个非空集合，调用 c.end() 返回一个指向该集合最后一个元素的迭代器。
5. 图 20-1 中表示的迭代器继承层次的 UML 图，ForwardIterator 中的方法列表为空。为什么会存在这种层次？
6. 冯·诺依曼结构的哪个特性使得可以定义指向函数的指针？
7. 描述下面两个声明的区别：

```
char *f(string);
char (*f)(string);
```

8. 如何将变量 fn 声明为：一个以两个整数作为形参并且返回值为布尔类型的函数的指针。
9. 什么是回调函数？
10. 什么是比较函数？
11. 用自己的语言描述函数指针和函数对象的区别。
12. 根据定义，每个函数类都要实现一个特定的操作符。这个操作符是什么？
13. 由 STL<algorithm> 库提供的 sort 函数的两个参数是什么？ 922
14. 本章中描述的函数式编程范型的主要性质是什么？
15. 判断题：从 binary_function 继承的一个函数类中，两个参数的类型必须相同。
16. <functional> 库中包含函数 bind1st 和 bind2nd 的目的是什么？
17. 使用 <functional> 库中的功能编写一个调用 count_if，使其返回元素类型为整型的 Vector 对象 vec 中偶数的个数。
18. 比较函数是什么？
19. 描述下面 typedef 语句产生的类型：

```
typedef void (*proc)();
```

20. 列出实现一个能够提供 RandomAccessIterator 层服务的迭代器所需要的完整操作集。
21. 判断题：C++ 中的指针定义了实现一个 RandomAccessIterator 的所有必需的操作符。

习题

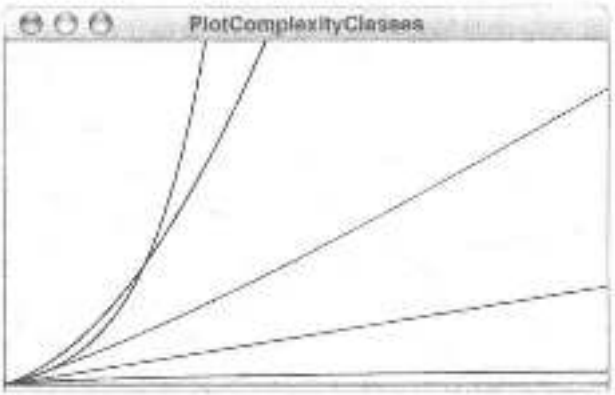
1. 重新编写图 5-11 中的程序 WordFrequency，使它使用迭代器代替在实现中出现 3 次基于范围的

for 循环。

2. 在图 20-2 中，plot 函数作为独立函数出现。为了使用户易于使用，将这个函数放入库中会更好。创建必要的 plot.h 和 plot.cpp 文件，从而通过这个接口导出 plot 函数的两个版本，其中一个以函数指针为参数，另一个以函数对象为参数。虽然你早已经完成了该代码，本题的难点就在于理解 plot.h 需要哪部分代码以及 plot.cpp 需要哪部分代码。

使用新的 plot.h 接口绘制一个展示大部分常见复杂类的增长曲线图：这些复杂类包含常量、对数、线性、$N \log N$、二次方程式和指数。如果你使用 x 的范围为 1 到 15，并且使用 y 的范围为 0 到 50，那么，这个图应如下所示：

923



3. 在当前设计中，函数 plot 有 6 个参数：图形窗口、要绘制的函数，以及两对指引 x 和 y 方向绘制范围的值。你可以通过 plot 函数计算出 y 方向上绘制的范围的方式来消除最后两个参数。你需要对这个计算执行两次，一次是找出函数的最大值和最小值；另一次是使用这些值作为绘制函数的界限范围。以你在习题 2 中编写的 plot 库为起点，增加一个新的能够自动计算 y 方向界限的 plot 版本。

4. 你可以使用函数指针以函数名的方式保存一个数学函数 Map 对象。例如，如果你采用以下的声明语句：

```
typedef double (*doubleFn)(double);
Map<string,doubleFn> functionTable;
```

然后，你可以通过给 functionTable 添加新条目的方式按照传统的函数名来存储这些函数。例如，下面的两行代码从 <cmath> 库中增加 sin 和 cos 函数：

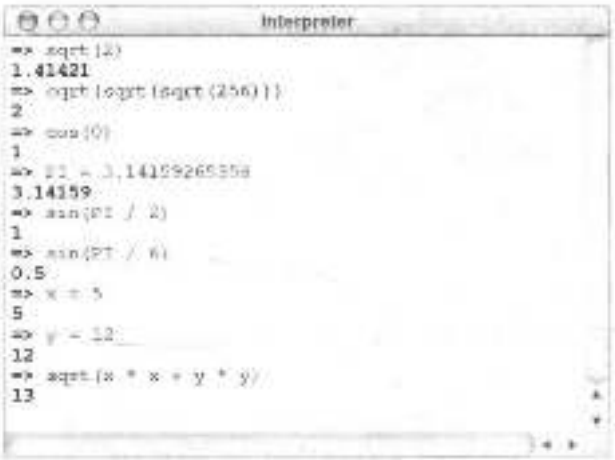
```
functionTable["sin"] = sin;
functionTable["cos"] = cos;
```

使用这种方法给第 19 章中的表达式解释器增加标准数学函数。这个改变要求你给原有的结构做如下一些扩展：

924

- 解释器在计算中必须使用实数而非整数。你在第 19 章中的习题 8 已经做过这个改变。
- 函数表需要赋入 EvaluationContext 类，以使解释器可以通过名字访问函数。
- 语法分析器需要包含一条对于表达式的新的语法规则，使得该表达式表示的是只含一个参数的函数调用。
- 对于新的函数类来说，eval 方法必须能够在函数表中查询函数名，并将该函数应用于计算参数的结果中。

你的实现代码应该像 C++ 那样允许函数的结合和嵌套。例如，如果你的解释器定义了 sqrt、sin 和 cos 函数，程序应该能够产生下面简单的运行结果：



5. 以习题4实现的Expression类的扩展版本作为起点，创建一个名为ExpressionFunction的函数类，它将一个表达式字符串转化为一个函数对象，该函数对象将变量x的值代入它的计算结果中。例如，如果你调用以下构造函数：

```
ExpressionFunction("2 * x + 3")
```

那么，返回的结果应该是一个你可以使用单个参数调用的函数。因此，如果你调用：

```
ExpressionFunction("2 * x + 3")(7)
```

那么，结果应该是表达式2*7+3的值，即14。

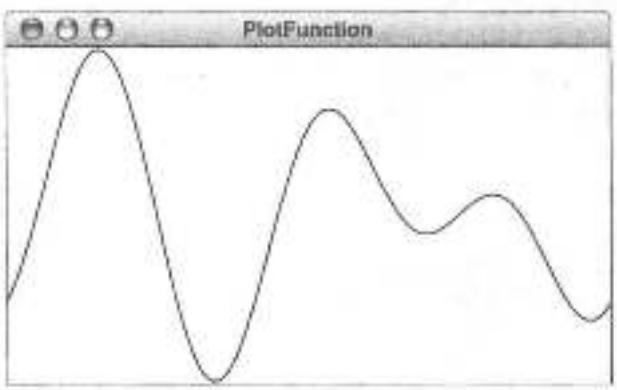
6. 将习题5的ExpressionFunction机制和习题3的plot函数相结合，使得plot函数参数可以是一个表示含有变量x的表达式的字符串。例如，在对plot.h接口做出这些扩展后，你应该能够调用以下语句生成20.2.2一节中展示的正弦波形： 925

```
plot(gw, "sin(x)", -2 * PI, 2 * PI, 1, 1)
```

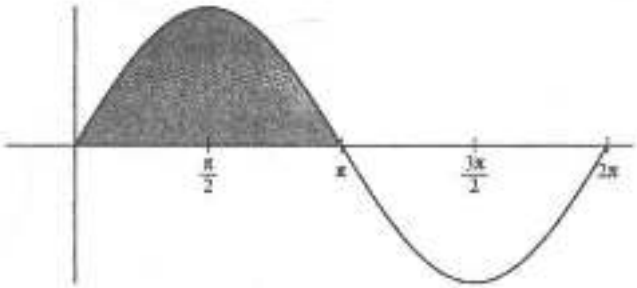
然而，表达式字符串可以使用任何能够被表达式语法分析器识别的机制，因此你可以使用它来绘制更复杂的函数，就像下面的调用说明的一样：

```
plot(gw, "sin(2 * x) + cos(3 * x)", -PI, PI)
```

这条语句使用ExpressionFunction生成一个计算表达式sin(2*x)+cos(3*x)的函数对象，接着使用该函数对象产生下面的图形：

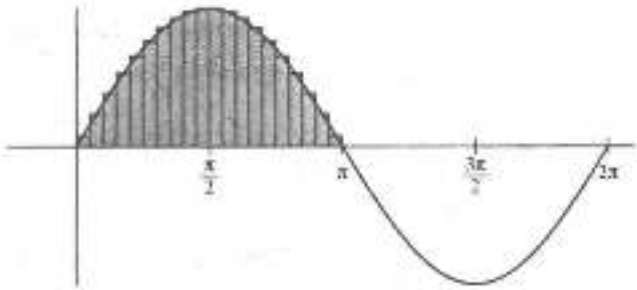


7. 在数积分中，一个函数的积分定义为水平方向上边界与由水平方向上的边界以及函数所确定的垂直方向的值所围成的面积。例如，0到π范围内的正弦函数的积分就是下图中的阴影面积：



926

你可以通过将这些固定宽度的小矩形的面积相加起来，从而计算出这个区域的近似面积。这些小矩形的高度为函数在该矩形中点处的函数值：



设计该原型并为一个名为 integrate 的函数编写代码，它通过求这些矩形的面积之和来估算该区域的面积。例如，为了计算前面例子中阴影区域的面积，用户要编写以下语句：

```
double value = integrate(sin, 0, PI, 20);
```

最后一个参数是该区域被分成的矩形的数量；这个值越大，估算值就越精确。

注意到在 x 轴下方的任何区域都被看作是负面积。因此，如果你计算 sin 从 0 到 2π 的面积，那么结果会是 0，因为坐标轴上方和下方的面积会相互抵消。

8. 实现 <algorithm> 库中的函数模板：

```
template <typename IterType>
IterType max_element(IterType begin, IterType end);
```

它返回一个指向迭代范围内最大的元素的迭代器。由于大部分标准库中都包含 <algorithm>，你可以为该函数起一个不同的名字。

9. 编写 <algorithm> 库中的函数模板 count_if 原型，然后再实现它。

10. 20.5.2 一节中的 lessIgnoringCase 代码是低效的，因为即使它能够通过查找第一个字符来确定相对的顺序也还是一直将字符串转化为小写字母。编写一个更加高效的 lessIntegringCase

927 版本，使其能够尽可能少看一些字符来确定结果。

11. 编写一个比较函数 titleComesBefore，它以两个字符串作为形参并比较它们，并且符合下面的规则：

- 这个比较应该忽略大小写。
- 除了空格以外的标点符号都应该被忽略。
- 出现在标题开头的单词 a、an 和 the 都应被忽略。

12. 在图 17-4 中表示的 Set 类的实现中，集合 == 操作符的代码是基于两个集合相等当且仅当每一个集合是对方的子集这一数学原则。尽管依赖该机制导致实现很简洁，但并不是特别高效。一种检测两个集合是否相等的更迅速方式是：如果你对这两个集合分别迭代遍历，你得到的值是否是相同的。

在尝试解决这个问题之前，唯一一个需要解决的问题是：第 17 章中实现的 Set 类不包括一个

迭代器。然而，这个实现位于 Stanford 类库中的 Map 类的顶层，它提供一个迭代器类。因此，你可以在每个集合中创建必要的 map 实例变量的迭代器。鉴于 C++ 中涉及模板类的声明规则确实难懂，因此，Set 类中 Map 迭代器的声明需要使用关键字 typedef 标记，这就迫使你像下面这样编写声明：

```
typename Map<ValueType,bool> it = map.begin();
```

使用这个策略重新在 set.h 接口中实现 == 操作符。

13. 参考例 15-8，为 StringMap 类实现一个迭代器。这个类是 HashMap 的一个特化版本，它意味着迭代器可以按照任意顺序处理元素。
14. 为 Grid 类实现一个迭代器，使其按照行优先的顺序处理元素。
15. 为了给你自己一个更大的挑战，为第 16 章中的 Map 类实现一个迭代器，它基于二叉搜索树。Map 迭代器总是按照升序来处理元素，它对应树的一个中序遍历。然而，当你用迭代器实现这个行为时，不能再依赖于递归，因为迭代器需要顺序操作。因此，当你递归地实现中序遍历时，Map 迭代器的代码必须自动跟踪其元素的状态。这要求你维持一个还没被访问的节点的栈。 928

索　引

索引中的页码为英文原版书的页码，与书中页边标注的页码一致。

#define directive (#define 指令), 79
#ifndef directive (#ifndef 指令), 79
--operator (-- 操作符), 33
-=operator (-= 操作符), 32
->operator (-> 操作符), 490
:: qualifier (:: 限定符), 268
!operator (! 操作符), 35
?: operator (?: 操作符), 36
.operator (点操作符), 262
[]operator ([] 操作符), 649
operator (操作符), 486
*p++idiom (*p++ 习语), 503
/*…*/ comment (/*…*/ 注释), 8
//… comment (//… 注释), 8
&operator (& 操作符), 486
&&operator (&& 操作符), 35
|operator (| 操作符), 486
||operator (|| 操作符), 35
++operator (++ 操作符), 33
+=operator (+= 操作符), 32
<<operator (<< 操作符), 11, 160, 170
>>operator (>> 操作符), 165

A

abs function (abs 函数), 61, 63
absorption (吸收律), 741
abstract class (抽象类), 826
abstract data type (抽象数据类型), 196
abstraction (抽象), 85, 407
abstraction boundary (抽象边界), 87
accessor (访问器), 265
acronym (首字母缩略词), 150
ado method (ado 方法), 199, 232, 236
AddIntegerList.cpp (AddIntegerList.cpp C++ 源程序), 44
additive sequence (可加序列), 330
additiveSequence function (additiveSequence 函数), 332
AddSFunction class (AddSFunction 类), 902
addOperator method (addOperator 方法), 294
address (地址), 479
AddThreeNumbers.cpp, 15
addWordCharacters method (addWordCharacters 方法), 294
addWordsFromFile method (addWordsFromFile 方法), 236
Adelson-Velskii, Georgii (乔治·安德森·维斯基), 708
adjacency list (邻接表), 773
adjacency matrix (邻接矩阵), 773
adjacentPoint function (adjacentPoint 函数), 397
ADT, 196
Aesop (伊索), 86
Airlinegraph class (Airlinegraph 类), 825
Airlinegraph.cpp (Airlinegraph.cpp C++ 源程序), 780
al-Khwārizmī (花拉子米), 58
Alcott, Louisa May (路易莎·梅·奥尔科特), 737
algebra (代数), 58
algorithm (算法), 58
<algorithm> library (<algorithm> 库), 907
allocated size (分配容量), 498
allocation (分配), 500
alphanumeric (字母数字混合的), 137
ambiguity (二义性), 847
ambiguous expression (具有二义性的表达式), 848
American National Standards Institute (美国国家标准协会), 4
amortized analysis (均摊分析), 559

anagram（回文构词法），364
analysis of algorithms（算法分析），430
ancestor（祖先），691
ANSI，4
application programming interface（应用编程接口），832
arc，768
area code（代码段），258
argument（参数），57
Ariadne（阿里阿德涅），390
ARPANET，18，821
array（数组），494
array capacity（数组容量），533
array selection（数组元素的选择），496
array-based editor（基于数组的编辑器），579
ASCII，22
assignment operator（赋值操作符），30
assignment statement（赋值语句），31
associative array（关联数组），232
associative law（优先级规则），741
associativity（优先级），28
astronomical unit（天文单位），345
at function（at 函数），130
AT&T Bell Laboratories（AT & T 贝尔实验室），5
atan function（atan 函数），61
atan2 function（atan2 函数），61
atomic type（原子类型），19
Austen, Jane（简·奥斯汀），615
automatic allocation（自动分配），516
average function（average 函数），438
average-case complexity（平均复杂度），440
AVL tree（AVL 树），708
axis of rotation（旋转轴），710

B

Bachmann, Paul（保罗·巴赫曼），435
backtracking algorithm（回溯算法），390
Backus-Naur Form (BNF)（BNF 巴斯范式），863
Backus, John（约翰·巴克斯），4，863
balance factor（平衡因子），709
balanced tree（平衡树），706
Ball, W. W. R.（鲍尔），350
base type（基类型），198，485
BASIC language（BASIC 语言），886
basis（基础），460
Beacons of Gondor（刚铎灯塔），519
BeaconsOfGondor.cpp（BeaconsOfGondor.cpp C++ 源程序），521
begin method（begin 方法），889
Bell Laboratories（贝尔实验室），4
Bernstein, Daniel J.（丹尼尔·斯蒂文杨），673
Bidirectionalliterator，891
big-O notation（大 O 助记符），435
BigInt class（BigInt 类），662
binary logarithm（以 2 为底的对数），448
binary notation（二元助记符），475
binary operator（二元操作符），27
binary search（二分查找），335
binary search tree（二分查找树），694
binary tree（二叉树），694
binary_function class（binary_function 类），910
binary_search function（binary_search 函数），908
bind1st function（bind1st 函数），910
bind2nd function（bind2nd 函数），910
biological hierarchy（生物层次），181
bipartite graph（二分图），818
bit（位），474
bitwise operator（位操作符），755
block（块），37
Bode, Johann Elert（约翰·埃勒特·巴德），344
body（体），12，57
Boggle game（Boggle 游戏），428
boilerplate（接口模板），78
bool type（bool 类型），21
boolalpha manipulator（boolalpha 操纵符），162
Boole, George（乔治·布尔），21
Boolean data（布尔数据），21
Boolean operators（布尔操作符），34
bounds-checking（边界检查），201
Brando, Marlon（马龙·白兰度），261
Braque, Georges（乔治·布拉克），368
breadth-first search（广度优先搜索），787
break statement（break 语句），39，42
Brin, Sergey（谢尔盖·布林），809
BST，694
bucket（桶），672
buffer（缓冲区），571

buffer overflow error (缓冲区溢出错误), 505
buffer.h interface (buffer.h 接口), 575
byte (字节), 475

C

C programming language (C 编程语言), 2
c_str function (c_str 函数), 130
C-style string (C 风格的字符串), 139
C(n, k) function [C(n, k) 函数], 66
C++11, 5
Caesar cipher (凯撒密码), 155
Caesar, Julius (尤利乌斯·凯撒), 155
calculator (计算器), 213
call by reference (引用调用), 74
callback function (回调函数), 895
calling a function (调用一个函数), 57
capacity (容量), 533
capitalize function (capitalize 函数), 150
Card class (Card 类), 307
cardinal number (首卡尔数), 152
Carl Linnaeus (卡尔·林奈), 690
Carmichael, Stokely (斯托·卡迈克尔), 1
Carroll, Lewis (刘易斯·卡罗尔), 2, 167, 901
case clause (case 子句) 38
case constant (case 常量), 39
catch clause (catch 子句), 845
catching an exception (捕获一个异常), 845
Cather, Willa (薇拉·凯瑟), 689
<cctype> library (<cctype> 库), 137, 233
ceil function (ceil 函数), 61
Cell type (Cell 类型), 520
cells in a linked list (链表中的一个元素), 520
cerr stream (cerr 流), 75
char type (char 类型), 22
characteristic vector (特征向量), 753
CharStack class (CharStack 类), 527
charstack.cpp (charstack.cpp C++ 源程序), 531
charstack.h interface (charstack.h 接口), 528
CheckoutLine.cpp (CheckoutLine.cpp C++ 源程序), 222
CheckoutLineClass.cpp (CheckoutLineClass.cpp C++ 源程序), 302
chess (国际象棋), 409
child (孩子), 691
Churchill, Winston (温斯顿·丘吉尔), 88
cin stream (cin 流), 12
Circle class (Circle 类), 839
class (类), 129
clear method (clear 方法), 170, 212, 218, 227, 232
clearing a bit (清除一位), 758
Clemens, Samuel (塞缪尔·克莱门斯), 195
client (用户), 61
close method (close 方法), 170
closing a file (关闭一个文件), 170
closure (闭包), 901
<cmath> library (<cmath> 库), 60
code review (代码审查), 544
Coin type (Coin 类型), 25
collectContributions function (collectContributions 函数), 317
collection class (集合类), 196
collision (冲突), 672
combinations (组合), 347
combinations function (combinations 函数), 66
Combinations.cpp (Combinations.cpp C++ 源程序), 68
combinatorics (组合学), 120
comment (注释), 8
common logarithm (常用对数), 448
commutative law (交换律), 741
compare function (compare 函数) 130
comparison function (比较函数), 912
compiler (编译器), 6
complement (取补), 756
complexity class (复杂类), 449
compound node (组合节点), 849
compound statement (组合语句), 37
computational complexity (时间复杂度), 435
concatenation (连接), 131
conditional execution (条件执行), 37
connected component (连接分量), 771
connected graph (连通图), 771
console input stream (控制台输入流), 12
console.h interface (console.h 接口), 108
console output stream (控制台输出流), 11

const correctness (const 正确性), 552
const keyword (const 关键字), 17
constant (常量), 17, 83
constant call by reference (常量引用调用), 548
constant folding (常量合并), 882
constant node (常量结点), 849
constant time (常量时间), 438, 450
constructor (构造函数), 207, 265
contains method (contains 方法), 232, 236
containsKey method (containsKey 方法), 227
containsPrefix method (containsPrefix 方法), 236
control expression (控制表达式), 38
convert (转换), 99
copy constructor (拷贝构造函数), 548
copy function (copy 函数), 908
copying objects (拷贝对象), 546
cos function (cos 函数), 61
cosDegrees function (cosDegree 函数), 84
cost (代价), 775
count function (count 函数), 908
count_if function (count_if 函数), 908
cout stream (cout 流), 11
Craps.cpp (Craps.cpp C++ 源程序), 97
crawling the web (爬取 web), 808
Cribbage (克里比奇), 379
crossover point (交叉点), 469
<cstddef> library (<cstddef> 库), 491
<cstdlib> library (<cstdlib> 库), 75, 91
cubic time (3 次方时间) 450
Cubism (立体派), 368
cursor (游标), 11, 571
cycle (周期), 41, 771
cyclic cypher (循环码), 155

D

Daedalus (代达鲁斯), 390
Dahl, Ole-Johan (奥利·约翰·达尔), 4
data members (数据成员), 264
data structure (数据结构), 196
data type (数据类型), 14, 19
daysInMonth function (daysInMonth 函数), 40
de Moivre, Abraham (亚伯拉罕·棣·美弗), 471
De Morgan, Augustus (奥古斯都·德·摩根), 741
De Morgan's law (德·摩根定理), 741
declaring a variable (声明一个变量), 11, 14
decomposition (分解), 58
decrement operator (自减运算符), 33
Deep Blue (深蓝), 409
deep copy (深拷贝), 546
default clause (default 从句), 39
default constructor (默认构造函数), 207, 265
default parameter (默认参数), 64
default value (默认值), 207
defaultExtension function (defaultExtension 函数), 191
definite integral (定积分), 926
degree (度), 771
delete operator (delete 操作符), 524
deleteCharacter method (deleteCharacter 方法), 576
denominator (分母), 284
dense graph (稠密图), 775
depth-first search (深度优先搜索), 783
dequeue method (dequeue 方法), 218
dereferencing a pointer (解析一个指针), 486
descendant (后代), 691
destructor (析构函数), 526
Dürer, Albrecht (阿尔布雷特·丢勒), 250
Dickens, Charles (查尔斯·狄更斯), 378
digital root (数字根), 346
Dijkstra, Edsger W. (艾兹格·迪杰斯特拉), 544, 804
directed graph (有向图), 770
Direction type (Direction 类型), 24, 80
direction.h interface (direction.h 接口), 81
directionToString function (directionToString 函数), 40
discrete time (离散时间), 220
Disney, Walt (沃特·迪斯尼), 257, 695
displayTree function (displayTree 函数), 704
distribution (分布), 247
distributive law (分配律), 741
divide-and-conquer algorithm (分而治之算法),

318
divides class (divides 类), 910
DNA, 157
domain (值域), 19, 126
dominating set (支配集), 820
domino (多米诺骨牌), 306, 423
Domino class (Domino 类), 306
dot operator (点操作符), 263
double roration (双旋转), 712
double type (double 类型), 14, 21
double-precision (双精度), 15
doubly linked list (双链表), 606
draw method (draw 方法), 832
drawPolarLine function (drawPolarLine 函数), 373
dummy cell (空元素), 593
Dylan, Bob (鲍勃·迪伦), 255
dynamic allocation (动态分配), 516
dynamic array (动态数组), 518

E

Easter date algorithm (复因斯坦日期算法), 118
edge (边), 768
editor (编辑器), 571
EditorBuffer class (EditorBuffer 类), 573
EditorBuffer constructor (EditorBuffer 构造函数), 575
effective size (有效容量), 498
Einstein, Albert (阿尔伯特·爱因斯坦), 88, 120
Eliot, George (乔治·艾略特), 191, 206, 663
Eliot, T. S. (托马斯·斯特尔那斯·艾略特), 125
embedded assignment (嵌套赋值), 31
Employee class (Employee 类), 825
empty list (空列表), 523
empty set (空集合), 738
empty vector (空向量), 199
encapsulation (封装), 181
end method (end 方法), 889
endl manipulator (endl 操作符), 11, 162
endsWith function (endsWith 函数), 136, 146
Engels, Friedrich (马·恩格斯), 429
EnglishWords.dat, 235
enqueue method (enqueue 方法), 218
enumerated type (枚举类型), 24
enumeration (枚举符), 739
EOF constant (EOF 常量), 171
eof method (eof 方法), 170
equal_to class (equal_to 类), 910
equalsIgnoreCase function (equalsIgnoreCase 函数), 136, 146
erase method (erase 方法), 130
Eratosthenes (埃拉托色尼), 249
error function (error 函数), 75
error.h interface (error.h 接口), 78
error.cpp (error.cpp C++ 源程序), 79
escape sequence (转义字符), 23
Euclid (欧几里得), 58, 345
eval method (eval 方法), 850
evaluatePosition method (evaluatePosition 方法), 413
exception handling (异常处理), 844
executable file (可执行文件), 6
exit function (exit 函数), 75
EXIT_FALLURE constant (EXIT_FALLURE 常量), 75
exp function (exp 函数), 61
exp.cpp (exp.cpp C++ 源程序), 858
exp.h interface (exp.h 接口), 851
exponential time (指数时间), 450
exporting constants (导出的常量), 83
expression tree (表达式树), 849
expression (表达式), 26
ExpressionFunction class (ExpressionFunction 类), 925
ExpressionType type (ExpressionType 类型), 850
extended-precision arithmetic (扩展精度运算), 662
extending an interface (扩展一个接口), 90
extension (扩展), 191
extern keyword (extern 关键字), 83
extraction operator (抽取运算符), 165

F

fact function (fact 函数), 45, 319
factorial (阶乘), 45, 318
fail method (fail 方法), 170
false constant (false 常量), 21
fib function (fib 函数), 331
Fib.cpp (Fib.cpp C++ 源程序), 328

Fibonacci sequence（斐波纳契数列），326
Fibonacci, Leonardo（列奥纳多·斐波那契），325
field（域），262
FIFO（先进先出），217
file（文件），167
fileio.h interface（fileio.h 接口），186
filename extension（文件扩展名），191
filename root（文件名），191
fill function（fill 函数），908
FillableShape class（FillableShape 类），872
final state（终态），314
find method（find 方法），130，134，908
findBestMove method（findBestMove 方法），413
findGoodMove method（findGoodMove 方法），402
findNode method（findNode 方法），697
finite set（有限集），738
finite-state machine（有限状态机），314
first method（first 方法），232
fixed manipulator（定点操作符），162
FlipCoin.cpp（FlipCoin.cpp C++ 源文件），95
float type（float 类型），21
floating-point（浮点类型），15，21
floor function（floor 函数），61
for statement（for 语句），43
for-each function（for-each 函数），908
foreach macro（foreach 宏），238
formatted input（格式化输入），165
formatted output（格式化输出），160
FORTRAN，4
forward reference（前向引用），779
ForwardIterator，891
fractal（分形），371
fractal snowflake（雪花分形），371
fractal tree（分形树），386
fraction（分数），281
Fredkin, Edward（爱德华·弗雷德），735
free function（free 函数），129
friend keyword（friend 关键字），275
fstream class（fstream 类），871
<fstream>library（<fstream> 库），168
function（函数），10，56，61
function class（函数类），902
function object（函数对象），902
function pointer（函数指针），893
function prototype（函数原型），62
functional programming（函数编程），909
functor（函子），902

G

game trees（游戏树），409
garbage collection（垃圾回收器），525
GATTACA，157
Gauss, Carl Friedrich（卡尔·弗里德里希·高斯），50，118
gcd function（gcd 函数），59，345
generateMoveList method（generateMoveList 方法），415
get method（get 方法），170，200，210，227
getExtension function（getExtension 函数），191
getInteger function（getInteger 函数），108，180
getLine function（getLine 函数），108
getPosition method（getPosition 方法），294
getReal function（getReal 函数），108
getRoot function（getReal 函数），191
getters（读取器），265
getTokenType method（getTokenType 方法），294
getType method（getType 方法），850
getX method（getX 方法），266
getY method（getY 方法），266
giga（千兆），475
global variable（全局变量），17
gmath.cpp（gmath.cpp C++ 源程序），107
gmath.h interface（gmath.h 接口），84
go out of scope（超出范围），526
GObject class（GObject 类），832
gobjects.cpp（gobjects.cpp C++ 源程序），837
gobjects.h interface（gobjects.h 接口），833
Goldberg, Adele（阿黛尔·戈德堡），5
golden ratio（黄金分割比率），470
Google（谷歌），809

GPoint class (GPoint 类), 308
grammar (语法), 862
graph (图), 768
graph.h interface (graph.h 接口), 793, 798
graphical recursion (图的递归), 368
GraphicsExample.cpp (GraphicsExample.cpp C++ 源程序), 100
Gray code (格雷码), 512
Gray, Frank (弗兰克·格雷), 512
greater class (greater 类), 910
greater_equal class (greater_equal 类), 910
greatest common divisor (最大公约数), 58, 345
Grectangle class (Grectangle 类), 308
greedy algorithms (贪心算法), 804
Grid class (Grid 类), 210
Grid constructor (Grid 构造函数), 210
Gödel, Escher, Bach (哥德尔、艾舍尔、巴赫), 52

H

H-fractal (H-分形), 384
hailstone sequence (冰雹序列), 53
half-life (半生命期), 21
half-open interval (半开区间), 95
Hamilton, Charles V. (查尔斯汉·密尔顿), 1
Hanoi.cpp (Hanoi.cpp C++ 源程序), 356
Harry Potter (哈利·波特), 51
hash code (哈希码), 672
hash function (hash 函数), 672
hash table (哈希表), 672
hashing (哈希), 672
HashMap class (HashMap 类), 682
HashSet class (HashSet 类), 748
hasMoreTokens method (hasMoreTokens 方法), 294
head of a queue (队列头), 217
header file (头文件), 9, 78
header line (首行), 12
heap (堆), 516, 721
heap area (堆区), 481
heap-stack diagram (堆-栈图), 536
heapsort algorithm (堆排序算法), 735
height of a tree (树的高度), 691
Heilman, Lillian (温斯顿·丘吉尔), 89
HelloWorld.cpp (HelloWorld.cpp C++ 源程序), 2
hexadecimal (十六进制), 20, 476
higher-level language (高级语言), 4
histogram (直方图), 248
Hoare, C. A. R. (托尼·霍尔), 452
Hofstadter, Douglas (道格拉斯·霍夫斯塔特), 52
holism (整体论), 338
hop (一跳), 787
hybrid strategy (混合策略), 469

I

IATA, 229
IBM, 409
idempotence (幂等性), 741
identifier (标识符), 16
identifier node (标识符结点), 849
identity (恒等式), 741
if statement (if 语句), 37
ifstream class (ifstream 类), 168
ignoreComments method (ignoreComments 方法), 294
ignoreWhitespace method (ignoreWhitespace 方法), 294
immutable class (不变类), 267
implementation (实现), 78
implementor (实现者), 61
in-degree (入度), 771
inBounds method (inBounds 方法), 210
inclusion/exclusion pattern (包含/排斥模式), 363
increment operator (自增运算符), 33
index (索引), 131
index variable (索引变量), 45
inductive hypothesis (归纳假设), 460
infinite set (无限集合), 738
information hiding (信息隐藏), 87
inheritance (继承), 181
initializer (初始化), 16
initializer list (初始化列表), 839
inorder traversal (中序遍历), 704
inplace_merge function (inplace_merge 函数), 908
input manipulators (输入操纵符), 166
InputIterator, 891
insert method (insert 方法), 130, 199

insertCharacter method (insertCharacter 方法), 576
insertion operator (插入操作符), 160
insertion sort (插入排序), 466
insertNode function (insertNode 函数), 698
instance (实例), 129
instance variables (实例变量), 264
Instant Insanity (四色方柱), 422
int type (int 类型), 14, 19
integer division (整除), 29
integerToString function (integerToString 函数), 146
integral (积分), 926
interface boilerplate (接口模板), 78
interface entry (接口条目), 78
Interface Message Processor (IMP)[接口消息处理器 (IMP)], 821
interface (接口) 78, 85
interior node (中间结点), 691
International Standards Organization (国际标准组织), 5
Internet (互联网), 18, 821
interpreter (解释器), 842
Interpreter.cpp (Interpreter.cpp C++ 源程序), 843
intersection (交), 232, 739
intractable (棘手的), 452
<iomanip> library (<iomanip> 库), 161
ios class (ios 类), 183
iostream class (iostream 类), 871
<iostream> library (<iostream> 库), 9, 160
isalnum function (isalnum 函数), 137
isalpha function (isalpha 函数), 137
isBadPosition method (isBadPosition 方法), 402
isdigit function (isdigit 函数), 137
isDigitString function (isDigitString 方法), 138
isEmpty method (isEmpty 方法), 212, 218, 227, 232
isEven function (isEven 函数), 337
isLeapYear function (isLeapYear 函数), 41
islower function (islower 函数), 137
isOdd function (isOdd 函数), 337
isPalindrome function (isPalindrome 函数), 141, 333
isprint function (isprint 函数), 137
ispunct function (ispunct 函数), 137
isspace function (isspace 函数), 137
isSubsetOf method (isSubsetOf 方法), 232
istream class (istream 类), 184
istringstream class (istringstream 类), 178
isupper function (isupper 函数), 137
isVowel function (isVowel 函数), 40
isWordCharacter method (isWordCharacter 方法), 294
isxdigit function (isxdigit 函数), 137
iter_swap function (iter_swap 函数), 908
iteration order (迭代顺序), 238
iterative (迭代的), 319
iterative statement (迭代语句), 41
iterator (迭代器), 236, 237, 506, 888

J

Jabberwocky (无聊的话), 167
Jackson, Peter (皮特·杰克逊), 519
James, William (威廉·詹姆斯), 315
John von Neumann (约翰·冯·诺依曼), 893
Juster, Norton (诺顿·贾斯特), 313

K

Kasparov, Garry (加里·卡斯帕罗夫), 409
Kemeny, John (约翰·科迷尼), 886
Kernighan, Brian (布莱恩·柯林汉), 2
key (键), 226, 694
KeyValuePair type (KeyValuePair 类型), 664
kilo (公里), 475
King, Martin Luther, Jr. (马丁·路德·金), 159
Kipling, Rudyard (鲁德亚德·吉卜林), 767
knight's tour (骑士之旅), 422
Koch snowflake (科赫雪花分形), 371
Koch, Helge von (海里格·冯·科赫), 371
Kruskal, Joseph (约瑟夫·克鲁斯卡), 819
Kulu, Thomas (托马斯·库恩), 4
Kurtz, Thomas (托马斯·克茨), 886

L

Landis, Evgenii (艾维基尼·兰蒂斯), 708
Lao-tzu (老子), 87

layered abstraction（分层抽象），748
leaf node（叶子节点），691
left child（左孩子），694
left manipulator（左操纵符），162
left rotation（左旋转），710
left-associative（左优先），28
leftFrom function（leftFrom 函数），80
Leibniz, Gottfried Wilhelm（莱布尼茨），53
length method（length 方法），130
less class（less 类），910，912
less_equal class（less_equal 类），910
letter-substitution cipher（字母替换密文），156
LetterFrequency.cpp（LetterFrequency.cpp C++ 源程序），205
lexical analysis（词法分析），842
lexicographic order（字典次序），130，239，335，695
lexicon（字典），235
Lexicon class（Lexicon 类），235
library（库），6
LIFO，211
Line class（Line 类），832
linear probing（线性探测），688
linear search（线性搜索），335
linear structures（线性结构），616
linear time（线性时间），438，450
link（链接），520
linked list（链表），519
linked structure（链接结构），484，519
linking（链接），6
Linnaeus, Carl（卡尔·林奈），181
list-based editor（基于列表的编辑器），591
ListBuffer.cpp（ListBuffer.cpp C++ 源程序），603
load factor（加载因子），681
local variable（局部变量），17
Lodge, Henry Cabot（亨利·卡伯特·洛奇），823
log function（log 函数），61
log10 function（log10 函数），61
logarithm（对数），448
logarithmic time（对数时间），450
logical and（逻辑与），35
logical not（逻辑非），35
logical operator（逻辑操作符），35
logical or（逻辑或），35
logical_and class（logic_and 类），910
logical_not class（logic_not 类），910
logical_or class（logic_or 类），910
long double type（long double 类型），21
long type（long 类型），19
lookup table（查找表），668，669
loop（循环），41
loop-and-a-half problem（循环一半问题），42
Lucas, Édouard（爱德华·卢卡斯），350
lvalue（左值），485

M

machine language（机器语言），3
Magdalen College（莫德林学院），385
magic square（魔方），250
main function（main 函数），11
majority element（主要元素），471
makeMove method（makeMove 方法），415
Mandelbrot, Benoit（本华·曼德博），371
manipulator（操纵符），161
map（映射），226
Map class（Map 类），226，664
map.h interface（map.h 接口），665
mapAll method（mapAll 方法），898
mapping function（映射函数），888，897
Markov process（马尔克夫过程），809
Markov, Andrei（安德烈·马尔可夫），809
mask（面具），757
mathematical induction（数学推导），460
max function（max 函数），617，908
max_element function（max_element 函数），908
Maze class（Maze 类），394
maze.h interface（maze.h 接口），395
mean（平均值），247
mega（一百万），475
member（成员），262
membership（隶属关系），739
memory allocation（内存分配），500
memory leak（内存泄漏），525
memory management（内存管理），516
merge function（merge 函数），908
merge sort algorithm（归并排序算法），445
merging（归并），444

message（消息），129
method（方法），129
metric（度量标准），339
min function（min 函数），908
min_element function（min_element 函数），908
Minesweeper game（扫雷游戏），252
minimax algorithm（求最小最大值算法），409
minimum spanning tree（最小生成树），819
Minotaur（克塔洛斯），390
minus class（minus 类），910
mnemonic（助记符），381
model（模型），219
model-view-controller pattern（模型－视图－控制器模型），570
modular arithmetic（模块化算法），639
modulus class（modulus 类），910
Mondrian, Piet（彼特·蒙德里安），368
Mondrian.cpp（Mondrian.cpp C++ 源程序），370
Monte Carlo integration（蒙特卡洛积分），122
Month type（Month 类型），25
Morse code（摩尔斯密码），257，731
Morse, Samuel F. B.（萨缪尔·摩尔斯），257
move in a game（在游戏中移动），407
moveCursorBackward method（moveCursorBackward 方法），575
moveCursorForward method（moveCursorForward 方法），575
moveCursorToEnd method（moveCursorToEnd 方法），576
moveCursorToStart method（moveCursorToStart 方法），576
moveStringToDisk function（moveStringToDisk 函数），355
moveTower function（moveTower 函数），355
multiple assignment（多重赋值），32
multiple inheritance（多重继承），871
multiplies class（multiplies 类），910
mutator（设值方法），267
mutual recursion（间接递归），337
MVC pattern（MVC 模式），570

N

N log *N* time（*N* log *N* 时间），450
N-queens problem（*N* 皇后问题），421
name of a variable（变量名），14
named constant（命名常量），17
namespace（名字空间），9
natural logarithm（自然对数），448
natural number（自然数），738
naughts and crosses（十字游戏），426
Naur, Peter（彼得·诺尔），863
negate class（negate 类），910
neighbor（邻居），771
nested type（嵌套类型），889
new operator（new 操作符），517
nextToken method（nextToken 方法），294
Nim game（拿子游戏），401，426
Nim.cpp（Nim.cpp C++ 源程序），403
noboolalpha manipulator（noboolalpha 操纵符），162
node（结点），690，768
nondeterministic programs（非终结程序），90
nonterminal symbol（非终结符），863
nonterminating recursion（非终结递归），339
normalization（规范化），99
noshowpoint manipulator（noshowpoint 操纵符），162
noshowpos manipulator（noshowpos 操纵符），162
noskipws manipulator（noskipws 操纵符），166
not_equal_to class（not_equal_to 类），910
not1 function（not1 函数），910
not2 function（not2 函数），910
nouppercase manipulator（nouppercase 操纵符），162
npos constant（npos 常量），134
null character（空字符），23，503
NULL constant（NULL 常量），491
null pointer（空指针），491
numCols method（numCols 方法），210
numerator（分子），284
numRows method（numRows 方法），210
Nygaard, Kristen（克利斯登·奈加特），4

O

Obenglobish（Obenglobish），154
object（对象），129
object file（对象文件），6
object-oriented paradigm（面向对象范型），4

octal（十进制），20
ofstream class（ofstream 类），168
Olsen, Tillie（蒂莉·奥尔森），569
open addressing（开放寻址），687
open method（open 方法），170
opening a file（打开文件），168
operator overloading（操作符重载），131，268
operator（操作符），26，165
operator!=，275
operator+，287
operator<<，272
operator++，273
operator|，649
optimization（优化），882
ordinal number（序数），152
origin（原点），111
ostream class（ostream 类），184
ostringstream class（ostringstream 类），178
out-degree（出度），771
output manipulators（输出操纵符），162
OutputIterator，891
Oval class（Oval 类），832
overhead word（开销词），538
overloading（过载），63，616
Oxford University（牛津大学），385

P

P versus NP problem（P 与 NP 问题），452
Page, Larry（拉里·佩奇），809
PageRank algorithm（PageRank 算法），809
palindrome（回文），141，332
paradigm shift（范型转移），4
parameter（参数），61
parameterized classes（模板类），198
parent（父亲），691
parse tree（解析树），847
parser generator（语法生成器），864
parser.cpp（parser.cpp C++ 源程序），865
parsing（解析），842
parsing an expression（解析表达式），862
partially ordered tree（偏序树），719
partition function（partition 函数），908
partitioning（分割），454
Parville, Henri de（亨利·德·巴微），350
Pascal, Blaise（布莱斯·帕斯卡），347
Pascal's Triangle（帕斯卡三角形），347
path（路径），771
pattern（模式），134
peek method（peek 方法），212，218
peg solitaire（peg 纸牌），423
perfect number（完全数），117
permutation（排列），119，364
Permutations.cpp（Permutations.cpp C++ 源程序），366
persistent property（持久性），161
PI constant（PI 常数），17，83，84
Picasso, Pablo（巴勃罗·毕加索），368
Pig Latin（儿童黑话），142
pigeonhole principle（鸽巢原理），472
PigLatin.cpp（PigLatin.cpp C++ 源程序），143
pivot（枢纽），454
pixel（像素），111，420
plot function（plot 函数），894，897
plotting a function（绘制一个函数），893
plus class（plus 类），910
ply（玩），411
pocket calculator（小型计算器），213
Point class（Point 类），266，276
Point type（Point 类型），262
point.cpp（point.cpp C++ 源程序），278
point.h interface（point.h 接口），276
pointer（指针），484
pointer arithmetic（指针运算），500
pointer assignment（指针赋值），489
pointer to a function（指向函数的指针），893
Poisson distribution（泊松分布），221
Poisson, Siméon（西莫恩·泊松），221
polar coordinates（极坐标），373
polymorphism（多态），616
polynomial algorithm（多项式算法），451
pop method（pop 方法），212
portability（可移植性），20
postorder traversal（后序遍历），704
PostScript（下标），255
pow function（pow 函数），61
PowersOfTwo.cpp（PowerOfTwo.cpp C++ 源程序），7

precedence（前趋），27
PrecisionExample.cpp（PrecisionExample.cpp C++源程序），164
predicate function（谓词函数），41，62
prefix operator（前缀操作符），333
preorder traversal（前序遍历），704
prime factorization（素因数分解），52
prime number（素数），117
primitive type（基本类型），19
priority queue（优先队列），661，719，806
private keyword（private关键字），264
procedural paradigm（过程程序设计范型），4
procedure（过程），62
programming abstraction（编程抽象），85
prompt（提示符），11
promptUserForFile function（promptUserForFile函数），172
proper divisor（公约数），117
proper subset（真子集），740
protected keyword（protected关键字），836
prototype（原型），10，62
pseudorandom number（伪随机数），91
Ptolemy I（托勒密一世），58
ptr_fun function（ptr_fun函数），910
public keyword（public关键字），264
pure virtual method（纯虚方法），826
push method（push方法），212
put method（put方法），170，172，227

Q

quadratic equation（二次方程式），74
quadratic time（二次方时间），439，450
Quadratic.cpp（Quadratic.cpp C++源程序），76
qualifier（限定符），268
question-mark colon（问号冒号），36
queue（队列），217
Queue class（Queue类），217，634
queue.h interface（queue.h接口），635
Quicksort（快速排序），452

R

radioactive decay（放射衰减），121
raiseIntToPower function（raiseIntToPower函数），344
raiseToPower function（raiseToPower函数），7
rand function（rand函数），91
RAND_MAX constant（RAND_MAX常量），91
random number（随机数），90
random.cpp（random.cpp C++源程序），105
random access memory（随机存储器，RAM），434
RandomAccessIterator（RandomAccessIterator），891
RandTest.cpp（RandTest.cpp C++源程序），92
range-based for loop（基于范围的for循环），238
rational arithmetic（有理数运算），282
Rational class（Rational类），288
rational number（有理数），281
rational.cpp（rational.cpp C++源程序），290
rational.h interface（rational.h接口），288
read-eval-print loop（读写循环），842
read-until-sentinel pattern（读直到哨兵模式），43
real number（实数），738
realToString function（realToString函数），146
receiver（接收器），129
record（记录），262
Rect class（Rect类），832
recurrence relation（递归关系），326
recursion（递归），316
recursive decomposition（递归分解），318
recursive descent（递归下降），864
recursive leap of faith（递归的稳步跳跃），324
recursive paradigm（递归范型），318
red-black tree（红黑树），732
reductionism（简化论），338
reference parameter（引用参数），73
rehashing（重新培养），682
relation（关系），740
relational operator（关系运算符），34
remove method（remove方法），200，227，232
repeatChar function（repeatChar函数），136
replace function（replace函数），130
replace_if function（replace_if函数），908
replaceAll function（replaceAll函数），151
reserved word（保留关键字），16
resize method（resize方法），210

retractMove method (retractMove 方法), 415
return address (返回地址), 113
return by reference (引用返回), 272
RETURN key (RETURN 键), 12
return statement (return 语句), 57
reverse function (reverse 函数), 137
Reverse Polish Notation (RPN) [逆波兰注记符 (RPN)], 213
ReverseFile.cpp (ReverseFile.cpp C++ 源程序), 204
right child (右孩子), 694
right manipulator (右操纵符), 162
right rotation (右旋转), 710
right-associative (右结合), 28
right-hand rule (右手规则), 390
rightFrom function (rightFrom 函数), 80
ring buffer (环形缓冲区), 639
Ritchie, Dennis (丹尼斯·里奇), 2, 4
Robson, David (大卫·罗伯森), 5
Roman numerals (罗马数字), 685
root (根), 191, 690
roundToSignificantDigits, 872
row-major order (行优先次序), 239
Rowling, J. K. (乔安娜·凯瑟琳·罗琳), 51
RPNCalculator.cpp (RPNCalculator.cpp C++ 源程序), 215

S

sample run (示例运行), 7
saveToken method (saveToken 方法), 294
scalar type (标量类型), 39
scaling (缩放), 99
scanNumber method (scanNumber 方法), 294
scanStrings method (scanStrings 方法), 294
scientific manipulator (科学操纵符), 162
scope (范围), 14, 526
Scrabble (游戏), 150
searching (搜索), 335
seed (种子), 103
selection sort algorithm (选择排序算法), 431
selection (选择), 496
sender (发送器), 129
sentinel (哨兵), 42
set (集合), 232, 738
Set class (Set 类), 232
set difference (集合差), 232, 740
set equality (集合相等), 740
set method (set 方法), 200, 210
setfill manipulator (setfill 操纵符), 162
setInput method (setInput 方法), 294
setprecision manipulator (setprecision 操纵符), 162
setter (设置器), 267
setting a bit (设置一位), 758
setw manipulator (setw 操纵符), 162
shadowing (遮蔽), 267
Shakespeare, William (威廉·莎士比亚), 887
shallow copy (浅拷贝), 546
Shaw, George Bernard (乔治·萧伯纳), 151
Shelley, Mary (玛丽·雪莱), 515
short type (short 类型), 19
short-circuit evaluation (短路求值), 35
shorthand assignment (缩写赋值), 32
showContents method (showContents 方法), 576
ShowFileContents.cpp (ShowFileContents.cpp C++ 源程序), 173
showpoint manipulator (showpoint 操纵符), 162
showpos manipulator (showpos 操纵符), 162
sibling (兄弟), 691
Sierpinski Triangle (谢尔宾斯基三角形), 387
Sierpiński, Waclaw (瓦斯瓦·谢尔宾斯基), 387
sieve of Eratosthenes (爱拉托逊斯筛法), 249
signature (签名), 63
simple.h interface (simple.h 接口), 186
simple case (简单情况), 318
simple cycle (简单回路), 771
simple path (简单路), 771
simple statement (简单语句), 36
simplifications of big-O (大 O 的简化), 436
SIMULA, 4
simulation (仿真), 219
sin function (sin 函数), 61, 893
sinDegrees function (sinDegrees 函数), 84
single rotation (单向旋转), 710
size method (size 方法), 212, 218, 227, 232, 236

size_t type (size_t 类型), 132
sizeof operator (sizeof 操作符), 479
skipws manipulator (skipws 操纵符), 166
slicing (切片), 830
Smalltalk, 5
Snowflake.cpp (Snowflake.cpp C++ 源程序), 374
solveMaze (solveMaze 函数), 397
sort function (sort 函数), 908
sorting (排序), 430
source file (源文件), 6
spanning tree (扫描树), 819
sparse graph (稀疏图), 774
sqrt function (sqrt 函数), 61
stand function (stand 函数), 103
<sstream> library (<sstream>库), 177
stack (栈), 211
stack area (栈区), 481
Stack class (Stack 类), 211, 619
Stack constructor (Stack 构造函数), 212
stack frame (栈帧), 65, 481
stack machine (栈机), 883
stack-based editor (基于栈的编辑器), 586
stack.h interface (stack.h 接口), 620
standard deviation (标准差), 247
Standard Template Library (标准模板库), 197
Stanford libraries (Stanford 库), 107, 186
startsWith function (startsWith 函数), 135, 146
state (状态), 407
statement (语句), 36
static allocation (静态分配), 516
static analysis (静态分析), 413
static area (静态区), 481
static initialization (静态初始化), 496
static keyword (static 关键字), 104
static local variable (静态局部变量), 104
std namespace (std 名字空间), 9, 79
stepwise refinement (步长精化), 58
STL, 197
stock-cutting problem (下料问题), 424
Stoppard, Tom (汤姆·斯托帕德), 122
str function (str 函数), 179
stream (流), 23, 126
string class (string 类), 24, 126
string comparison (字符串比较), 130
string constructor (字符串构造函数), 130
<string> library (<string>库), 24
string literal (字符串字面值), 126
string methods (字符串方法), 130
string stream (字符串流), 177
StringMap class (StringMap 类), 664, 824
stringToInteger function (stringToInteger 函数), 146
stringToReal function (stringToReal 函数), 146
strlib.h function (strlib.h 接口), 146, 186
strongly connected (强连通), 772
Stroustrup, Bjarne (本贾尼·斯特劳斯特卢普), 5
structure (结构), 262
subclass (子类), 184
subset (子集), 740
subset-sum problem (子集合问题), 361
subsetSumExists function (subsetSumExists 函数), 363
substr (字符串方法), 130, 133
substring (子串), 133
subtree (子树), 692
successive approximation (逐次逼近法), 118
Sudoku, 209, 251
Suetonius (苏维托尼乌斯), 155
suffix operator (后缀操作符), 33
Sun Tzu (孙子), 349
superclass (超类), 184
swap function (swap 函数), 492, 908
SwapIntegers.cpp (SwapIntegers.cpp C++ 源程序), 493
switch statement (switch 语句), 38
symbol table (符号表), 226, 844
symmetric matrix (对称矩阵), 774

T

tail of a queue (队尾), 217
taking the complement (取补), 756
target (目标), 485
template (模板), 198, 616
template class (模板类), 619
template specialization (模板实例化), 760
term (术语), 26

terminal symbol（终结符号），863
termination condition（终止条件），41
text data（文本数据），126
text file（文本文件），167
The Phantom Tollbooth（《神奇的收费亭》），313
Theseus（提休斯），390
this keyword（this 关键字），490
three-pile Nim（三堆牛子），426
throw statement（throw 语句），845
throwing an exception（抛出一个异常），845
Thurber, James（詹姆斯·瑟伯），193
tic-tac-toe（十字棋），426
time-space tradeoff（时－空平衡），607
time function（time 函数），103
Titius-Bode law（提丢斯－波得定律），344
Titius, Johann Daniel（尤翰·丹尼尔·提丢斯），344
toDegrees function（toDegrees 函数），84
token（记号），292，842
TokenScanner class（TokenScanner 类），292
TokenScanner constructor（TokenScanner 构造函数），294
tokenscanner.cpp（tokenscanner.cpp C++ 源程序），300
tokenscanner.h interface（tokenscanner.h 接口），298
Tolkien, J. R. R.（J.R.R. 托尔金），519，724
tolower function（tolower 函数），137
toLowerCase function（toLowerCase 函数），146
top-down design（自顶向下的设计），58
toRadians function（toRadians 函数），84
toupper function（toupper 函数），137
toUpperCase function（toUpperCase 函数），146
Towers of Hanoi puzzle（汉诺塔游戏），350
tractable（易于处理的），452
transient property（透明性），161
translation（翻译），99
traveling salesman problem（旅行商问题），452
traversing a graph（遍历一个图），783
traversing a tree（遍历一棵树），704
tree（树），690
trie（字典树），735
trim function（trim 函数），146
true constant（true 常量），21
truncation（截断），29
truth table（真值表），35
try statement（try 语句），845
Twain, Mark（马克·吐温），195
two's complement arithmetic（二进制补码），478
TwoLetterWords.cpp（TwoLetterWords.cpp C++ 源程序），237
TwoPlayerGame class（TwoPlayerGame 类），880
type cast（类型转型），30
type conversion hierarchy（类型转换层次），28
type of a variable（变量的类型），14
typename keyword（typename 关键字），617

U

UML diagram（UML 图），184
unary operator（一元运算符），27
unary_function class（unary_function 类），910
undirected graph（无向图），770
unget method（unget 方法），170，174
union（并），232，739
unit testing（单元测试），544
universal modeling language（统一建模语言），184
unparsing（无解析），883
unsigned keyword（unsigned 关键字），20
uppercase manipulator（uppercase 操纵符），162

V

value assignment（值赋值），489
value parameter（值参），74
variable（变量），14
variable declaration（变量声明），11
variable name（变量名），14
Vector class（Vector 类），197，649
Vector constructor（Vector 构造函数），206
vector.h interface（vector.h 接口），650
Venerable Bede（圣比德），118
Venn diagram（文氏图），740
Venn, John（约翰·维恩），740
verifyToken method（verifyToken 方法），294
vertex（顶点），768

visiting a node（访问一个结点），783
von Neumann architecture（冯·诺依曼体系结构），893
von Neumann, John（约翰·冯·诺依曼），893

W

walking a tree（遍历树），704
Wallinger, Mark（马克·谢林格），385
wchar_t type（wchar_t 类型），478
weakly connected（弱连接），772
Weil, Simone（西蒙娜·韦伊），389
while statement（while 语句），41
whitespace character（空白字符），127，137
wildcard pattern（通配符模式），425
Woolf, Virginia（弗吉尼亚·伍尔夫），473
word（单词），475
WordFrequency.cpp（WordFrequency.cpp C++ 源程序），243
worst-case complexity（最坏的复杂度），440
wrapper（包装器），331
ws manipulator（ws 操作符），166
wysiwyg（所见即所得），572

X

Xerox PARC（施乐 PARC），5

推荐阅读



C程序设计语言（第2版·新版）

作者：Brian W. Kernighan 等 译者：李忠
ISBN：978-7-111-12806-0 定价：30.00元



Java语言程序设计（基础篇）（原书第10版）

作者：梁勇（Y. Daniel Liang） 译者：戴开宁
ISBN：978-7-111-50690-4 定价：85.00元



Haskell函数式程序设计

作者：理查德·伯德 译者：乔海燕
ISBN：978-7-111-52932-3 定价：69.00元



Scala编程思想（原书第2版）

作者：布鲁斯·埃克尔 等 译者：陈昊鹏
ISBN：978-7-111-51740-5 定价：69.00元

推荐阅读













C程序设计语言（第2版·新版）

作者：Brian W. Kernighan等 ISBN：978-7-111-12806-0 定价：30.00元

C语言的科学和艺术

作者：Eric S. Roberts ISBN：978-7-111-34775-0 定价：79.00元

C程序设计导引

作者：尹宝林 ISBN：978-7-111-41891-7 定价：35.00元

C程序设计思想与方法

作者：尹宝林 ISBN：978-7-111-25495-9 定价：36.00元

从问题到程序——程序设计与C语言引论 第2版

作者：诸宗红 ISBN：978-7-111-33715-7 定价：39.00元

C语言解惑

作者：刘振安等 ISBN：978-7-111-47965-7 定价：79.00元

推荐阅读













算法导论（原书第3版）

作者：Thomas H.Cormen 等 ISBN：978-7-111-40701-0 定价：128.00元

算法基础：打开算法之门

作者：Thomas H. Cormen ISBN：978-7-111-52076-4 定价：59.00元

算法心得：高效算法的奥秘（原书第2版）

作者：Henry S.Warren ISBN：978-7-111-45356-7 定价：99.00元

算法设计编程实验：大学程序设计课程与竞赛训练教材

作者：吴永辉 ISBN：978-7-111-42383-6 定价：69.00元

算法与数据结构考研试题精析 第3版

作者：陈守孔 ISBN：978-7-111-50067-4 定价：69.00元

数据结构编程实验：大学程序设计课程与竞赛训练教材

作者：吴永辉 ISBN：978-7-111-37335-7 定价：59.00元

C++程序设计 基础、编程抽象与算法策略

Programming Abstractions in C++

本书是一本风格独特的C++语言教材，内容源自作者在斯坦福大学多年成功的教学实践。它突破了一般C++编程教材注重介绍C++语法特性的局限，不仅全面讲解了C++语言的基本概念，而且将重点放在深入剖析编程思路上，并以循序渐进的方式教授读者正确编写可行高效的C++程序。本书内容遵循ACM CS2013关于程序设计课程的要求，既适合作为高校计算机及相关专业学生的教材或教学参考书，也适合希望学习C++语言的初学者和中高级程序员使用。

本书特色

- 面向C++语言的初学者，从内容安排到讲授都遵循化繁为简、通俗易懂的特色，并安排大量案例，理论联系实际，使读者轻松进入C++编程的大门。
- 突出C++语言的特点，以面向对象概念和编程抽象为核心，使读者了解并掌握优秀软件开发人员应具备的编程思维与实践能力。
- 努力跨越传统的程序设计语法与算法策略之间的鸿沟，通过独具匠心的内容安排，将数据结构、算法的相关内容与语法基础巧妙结合，将众多经典、实用的算法策略传授给读者，为后续课程或读者的深入学习奠定基础。



书号：978-7-111-34775-0
定价：79.00元



书号：978-7-111-38074-0
定价：99.00元

Pearson
www.pearson.com

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指南：计算机/程序设计
ISBN 978-7-111-54696-2
9 787111 546962 >
定价：129.00元

[illegible]
书名吼[illegible]程序设计　基础、编程抽象与算法策略
作者吼（英）埃里克·[illegible]·罗伯茨（[illegible]）著
页数吼[illegible]
[illegible]
[illegible]
出版日期吼[illegible]
出版社吼北京机械工业出版社