

国外计算机科学教材系列

C++ 编程

——从问题分析到程序设计

C++ Programming:
- From Problem Analysis To Program Design

[美] D. S. Malik 著

钟书毅 高志刚 陈雷 等译

THOMSON
COURSE TECHNOLOGY



电子工业出版社
Publishing House of Electronics Industry
<http://www.phei.com.cn>

经典教材

C++编程

——从问题分析到程序设计

C++ Programming:

From Problem Analysis To Program Design

本书是一本无需任何程序设计基础、内容涵盖CS1课程的权威教材。全书的编写与设计充分考虑到学生的学习方法,借助于详尽的解释和例子对C++进行全面描述。书中还包含了输入/输出、自定义函数、数组、结构、类等C++程序设计方面的内容,并通过大量的实例加以阐述。

本书特点:

- 本书所有程序源代码都以ANSI/ISO标准C++和标准C++形式提供,并可以使用Microsoft Visual C++, Borland C++ Builder 或 Metrowerks CodeWarrior 等开发工具编译这些程序
- 每章后面的程序实例都是完整的程序代码。这些程序实例中包括输入、输出、问题分析与算法设计等步骤,并提供了完整的程序代码清单
- “小结”部分总结了每章的重要概念
- “练习”测试学生标识声明和语句正误的能力
- “编程练习”培养学生亲自动手编写C++程序的能力
- “注意”强调了各章节中重要概念的注意事项
- 每章中统一编号的例题通过相应代码来帮助读者理解程序设计的概念

D. S. Malik

D. S. Malik 现任 Creighton 大学数学与计算机科学系教授。他 1985 年获得 Ohio 大学博士学位。自从在 Creighton 大学任教以来,他一直在讲授计算机程序设计课程。D. S. Malik 在抽象代数,模糊自动机理论及语言,模糊逻辑及应用和信息科学领域发表了超过 45 篇论文并出版了 6 本图书。

ISBN 7-5053-8241-1



9 787505 382411 >

THOMSON
COURSE TECHNOLOGY



责任编辑:李爱华
熊健
封面设计:毛惠庚



本书贴有激光防伪标志,凡没有防伪标志者,属盗版图书
ISBN 7-5053-8241-1/TP·4800 定价:59.00 元

国外计算机科学教材系列

C++ 编程

——从问题分析到程序设计

C++ Programming:

From Problem Analysis To Program Design

[美] D. S. Malik 著

钟书毅 高志刚 陈 雷 等译

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书循序渐进地介绍了 C++ 程序设计的各个方面, 包括 C++ 语言的基本概念、结构化程序设计、面向对象编程以及 C++ 的一些高级主题。在内容和结构的安排上, 本书强调编程实践的重要性, 许多简洁易懂的实例贯穿始终, 每一章的最后包含丰富的编程练习。此外, 本书还注重对 C++ 概念的深入理解, 作者通过论述与简单实例相结合的形式, 生动地阐明了 C++ 中很多关键的概念。通过本书的系统学习, 读者能够很快地掌握 C++ 程序设计的知识并有效地提高编程技术。

本书适合作为大学计算机科学 C++ 课程的教材, 也可供教师、学生或程序开发人员用做参考书。

981-243-396-1

Simplified Chinese edition Copyright © 2003 by Thomson Learning and Publishing House of Electronics Industry.

C++ Programming: From Problem Analysis To Program Design by D. S. Malik, Copyright © 2002. First published by Course Technology, a division of Thomson Learning, Inc(www.thomsonlearningasia.com).

All Rights Reserved.

Authorized simplified Chinese edition by Thomson Learning and Publishing House of Electronics Industry. No part of this book may be reproduced in any form without the express written permission of Thomson Learning and Publishing House of Electronics Industry.

本书中文简体字翻译版由电子工业出版社和汤姆森学习出版集团合作出版。未经出版者预先书面许可, 不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号: 图字: 01-2002-3798

图书在版编目 (CIP) 数据

C++ 编程——从问题分析到程序设计 / (美) 马力克 (Malik, D. S.) 著; 钟书毅等译.

-北京: 电子工业出版社, 2003.7

(国外计算机科学教材系列)

书名原文: C++ Programming: From Problem Analysis To Program Design

ISBN 7-5053-8241-1

I. C... II. ①马... ②钟... III. C 语言 - 程序设计 - 教材 IV. TP312

中国版本图书馆 CIP 数据核字 (2003) 第 049843 号

责任编辑: 李秦华 熊 健

印刷者: 北京兴华印刷厂

出版发行: 电子工业出版社 <http://www.phei.com.cn>

北京市海淀区万寿路 173 信箱 邮编: 100036

经 销: 各地新华书店

开 本: 787 × 1092 1/16 印张: 46.25 字数: 1332 千字

版 次: 2003 年 7 月第 1 版 2003 年 7 月第 1 次印刷

定 价: 59.00 元

凡购买电子工业出版社的图书, 如有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系。联系电话: (010) 68279077

出版说明

21 世纪初的 5 至 10 年是我国国民经济和社会发展的关键时期,也是信息产业快速发展的关键时期。在我国加入 WTO 后的今天,培养一支适应国际化竞争的一流 IT 人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡,是我国面对国际竞争时成败的关键因素。

当前,正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期,为使我国教育体制与国际化接轨,有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材,以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验,翻译出版了“国外计算机科学教材系列”丛书,这套教材覆盖学科范围广、领域宽、层次多,既有本科专业课程教材,也有研究生课程教材,以适应不同院系、不同专业、不同层次的师生对教材的需求,广大师生可自由选择 and 自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时,我们也适当引进了一些优秀英文原版教材,本着翻译版本和英文原版并重的原则,对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上,我们大都选择国外著名出版公司出版的高校教材,如 Pearson Education 培生教育出版集团、麦格劳-希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者,如道格拉斯·科默(Douglas E. Comer)、威廉·斯托林斯(William Stallings)、哈维·戴特尔(Harvey M. Deitel)、尤利斯·布莱克(Uyless Black)等。

为确保教材的选题质量和翻译质量,我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士,也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中,为提高教材质量,我们做了大量细致的工作,包括对所选教材进行全面论证;选择编辑时力求达到专业对口;对排版、印制质量进行严格把关。对于英文教材中出现的错误,我们通过与作者联络和网上下载勘误表等方式,逐一进行了修订。

此外,我们还将与国外著名出版公司合作,提供一些教材的教学支持资料,希望能为授课老师提供帮助。今后,我们将继续加强与各高校教师的密切联系,为广大师生引进更多的国外优秀教材和参考书,为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

教材出版委员会

主 任	杨芙清	北京大学教授 中国科学院院士 北京大学信息与工程学部主任 北京大学软件工程研究所所长
委 员	王 珊	中国人民大学信息学院院长、教授
	胡道元	清华大学计算机科学与技术系教授 国际信息处理联合会通信系统中国代表
	钟玉琢	清华大学计算机科学与技术系教授 中国计算机学会多媒体专业委员会主任
	谢希仁	中国人民解放军理工大学教授 全军网络技术研究中心主任、博士生导师
	尤晋元	上海交通大学计算机科学与工程系教授 上海分布计算技术中心主任
	施伯乐	上海国际数据库研究中心主任、复旦大学教授 中国计算机学会常务理事、上海市计算机学会理事长
	邹 鹏	国防科学技术大学计算机学院教授、博士生导师 教育部计算机基础课程教学指导委员会副主任委员
	张昆藏	青岛大学信息工程学院教授

译者序

C++编程语言是一门面向对象的程序设计语言，它提供了C语言所包含的各种功能，同时还改进了C语言自身所包含的某些缺陷。从C语言转换到C++语言是比较方便的，作为C语言的超集，C++语言非面向对象程序设计部分的功能、特性及语法和C语言基本上是相同的。所以，利用C++语言同样可以进行结构化编程，尽管很少人这么做。当然，C++语言的总体特性是面向对象的（虽然有些人认为C++语言不是纯面向对象的编程语言，如Java）。所以，在程序设计过程中，C++程序员要利用面向对象的思想来进行问题分析和程序设计。

对于一名编程初学者而言，面向对象的C++编程语言是庞大而复杂的，类、继承、多态、虚函数、构造函数和析构函数等诸多概念往往让人望而却步。许多初学者在学习C++语言时往往过于理论化，在短时间内灌输大量的C++概念，然而却不能很好地理解和应用，以至于编程水平得不到迅速的提高。事实上，初学者迅速掌握和领会C++语言的最佳途经就是进行大量的、系统性的编码练习，同时理解解决问题的有效方法。

本书是为C++初学者和大学计算机科学C++课程的教学而设计的，书中内容丰富并具有较强的系统性和实用性。对于有一定C或C++编程经验的读者而言，此书也是一本很有用的参考书。作者通过大量的例子深入剖析和阐述了几乎所有的关键性概念，并且给出了解决专门问题的案例研究，同时在每一章末都留有一定数量的编程练习。本书的这些特色有助于读者深入理解C++语言并提高程序设计的能力。

读者需要注意的是，本书的程序是系统化和用户友好的，每个编程实例都开始于问题分析，随后是算法设计。算法的每一步都利用C++语言实现了编码。除了讲授解决问题的技巧之外，这些程序还给用户展示了如何在实际的C++程序设计中实现所讲授的概念。建议读者仔细地研究书中的编程实例，以便有效地学习C++语言。

参加本书翻译的人员包括：高志刚和姜君（第1章～第10章）、钟书毅（序言和第14章～第17章）和陈雷（第11章～第13章、第18章和附录），全书由钟书毅统稿。

限于译者的水平，本书翻译过程中不足和错误之处在所难免，敬请广大读者批评指正，译者将不胜感激！

前 言

欢迎您阅读《C++ 编程——从问题分析到程序设计》。此教程是为初级计算机科学 C++ 课程而设计的，它将给老师和学生耳目一新的感觉。该课程是计算机科学全部课程的基础，作者的主要目标是激发学生的学习动力，无论他们水平高或低。学生的学习动力促成学习的兴奋性，而动力和兴奋性则是引领学生走向成功的关键因素。此教程源自于作者 50 多个学期以来授课过程中课堂笔记的积累，以及在此基础之上的进一步扩展。

本书开始于一系列的简单例子、练习和较复杂的编程，并用于作者所在大学的其他教程当中。事实上，本书所采用的内容组织方式是为了满足学生对清晰和可读性的要求。书中内容经过了多次重写直到学生觉得易于接受为止。书中的大部分例子来自于课堂上和学生的交流。

实践对于任何专业都是必不可少的。烹饪学生练习他们的菜谱；崭露头角的小提琴手练习他们的乐谱；编程新手也必须进行解决问题和编写代码的实践。这不是一本 C++ 大全，我们并不只简单地罗列 C++ 语法和例子，而是深入剖析所有概念背后的“为什么”。每一次介绍的主题，其关键性疑问“为什么”都得到了解答。这种方法构筑了学习 C++ 的桥梁。学生必须理解“为什么”以得到足够的学习动力。

传统上，C++ 初学者一般需要掌握一门其他编程语言。本书并不要求任何编程经验。不过，足够的数学背景知识如代数是必需的。

学习方法

从 C 语言发展而来的 C++ 编程语言不再认为是仅适用于工业界的语言。许多学院和大学都使用 C++ 作为第一门编程语言课程。C++ 是结构化编程和面向对象编程的结合，本书对两个方面都给予了论述。

本书可以分为两部分：结构化编程和面向对象编程。前 12 章构成了结构化编程部分，第 13 章到第 18 章构成了面向对象部分。然而，前 12 章（除了第 8 章）的内容对于学习面向对象部分是必不可少的。

ANSI/ISO 标准 C++ 于 1998 年 7 月得到了正式批准。本书主要讨论 ANSI/ISO 标准 C++，同时也说明了如何使用标准 C++ 编程。标准 C++ 和 ANSI/ISO 标准 C++ 的语法几乎是相同的。在第 8 章中，讨论了标准 C++ 中没有的一些 ANSI/ISO 标准 C++ 特征。

第 1 章简要回顾了计算机和编程语言的历史。读者通过阅读该章可以快速地了解计算机系统的硬件和软件组成。该章还叙述了结构化和面向对象编程。

完成第 2 章的学习后，学生就掌握了 C++ 的基础知识，并能够编写一些具有一定计算功能的、较复杂的程序了。输入/输出对任何一种编程语言来说都是最基本的，第 3 章对这部分内容首先做了简单的介绍，然后进行了详细的讨论。

第 4 章和第 5 章介绍了改变程序执行流程的控制结构。第 6 章和第 7 章研究用户定义的函数，没有编程经验的读者应该详细阅读这两章。这两章提供了一些例子以帮助读者加深对参数传递和标识符作用范围的理解。

第 8 章讨论了用户定义的简单数据类型（枚举类型）、ANSI/ISO 标准 C++ 的命名空间机制和

string 类型。早期的 C 版本并不包括枚举类型, 这种类型的作用很有限, 使用它的主要目的是增强程序的可读性。本书的组织方式让读者在第一次阅读时可以跳过枚举类型这一节, 而不影响学习的连续性, 可以在稍后再阅读这一节中的内容。

第 9 章和第 10 章详细地描述了数组。查找和排序算法在第 10 章中研究。第 11 章介绍并讨论递归。

第 12 章介绍记录 (结构)。本书中对结构的介绍类似于 C 结构。实际上, 读者完全可以跳过该章, 因为其后章节的学习并不要求以该章知识为前提。

第 13 章开始了面向对象编程 (OOP) 的学习, 还介绍了类的基本概念。这一章的前半部分展示了在程序中类是如何定义和使用的, 后半部分介绍抽象数据类型 (ADT)。同时还展示了在 C++ 中类是实现 ADT 的自然方法。第 14 章继续介绍面向对象设计 (OOD) 和 OOP 的基本概念, 还讨论了继承和组合。该章解释了 C++ 中的类是如何为 OOD 提供一种自然的机制, 以及 C++ 是如何支持 OOP 的。第 14 章还讨论了如何在给定的问题中确定对象。

第 15 章详细地学习指针。在介绍了指针和如何在程序中使用指针后, 还强调了带指针成员变量类的特异性以及如何避免一些错误操作。第 15 章还讨论了通过虚函数实现的多态类型。

第 16 章继续 OOD 和 OOP 的学习, 重点研究了 C++ 中的多态。其中特别讨论了两种类型的多态——重载和模板。

第 17 章和第 18 章主要研究数据结构。第 17 章详细讨论了链表, 而第 18 章详细讨论了栈和队列。这两章中的程序代码是通用的。这些章节有效地利用了 OOD 的基本概念。

附录 A 列举了 C++ 中的保留字。附录 B 展示了 C++ 中运算符的优先级和结合性。附录 C 列出了 ASCII 码 (美国标准信息交换码) 和 EBCDIC (扩充的二进制编码的十进制交换码) 字符集。附录 D 列出了可以被重载的 C++ 运算符。附录 E 描述了 ANSI/ISO 标准 C++ 以及标准 C++ 的头文件命名约定。附录 F 介绍了一些最常用的库函数以及标准 C++ 头文件的名称。附录 G 中的程序用于打印系统内部数据类型的内存尺寸。附录 H 给出了课本中精选练习的答案。

特色

本书的每一章都具有下列特色, 它们有助于知识的学习, 让学生能够按照自己的步调学习各种资料。

- 提供每章中 C++ 编程概念的要点。
- 超过 275 张图表, 广泛而详尽并阐述难以理解的概念。
- 书中的例子通过相关代码阐述关键的概念。这些例子中的程序代码是按照易于使用的方式进行编号的, 附带有运行实例。随后是详细的注释, 描述每一行代码所起的作用。
- 编程实例是在每一章的末尾推出的完整程序。这些例子包括准确和具体的输入、输出、问题分析和算法设计以及完整的程序清单。
- 小结提供每章中所包含概念的总结。
- 练习进一步加强学习, 保证学生学到实际的知识。
- 编程练习考查学生编写产生特定结果的 C++ 程序的能力。

概念的介绍是以有助于学习的步骤进行的。本书的书写风格是简单和直观的, 这和课堂的教学风格相似。在介绍每一种关键概念之前, 我们将解释为什么某些特定的因素是必需的。然后使用例子和小程序描述所要介绍的概念。

每章都有两种类型的程序。第一种类型的程序是作为书中的例子(如例4.1),用来解释关键概念。在这些例子中,程序代码的每一行都做了编号。程序通过运行实例进行阐述并逐行解释。然后详细地讨论每一行后面的原理。

本书的特色还包括众多称为编程实例的案例研究。这些编程实例构成了本书的支柱。

本书的程序是系统化和用户友好的。每个编程实例都开始于问题分析,随之是算法设计。算法中的每一步都以C++进行编码。除了讲授解决问题的技巧之外,这些详细的程序还展示了如何在实际的C++程序中实现相应的概念。作者强烈建议学生仔细地研究这些编程实例,以便有效地学习C++。

每一章末的小结部分起到加强学习的作用。在阅读完该章后,学生可以快速地浏览一遍本章的要点,然后可以用随后的练习进行自我测试。许多学生把小结看做是考试前复习该章的一种方便途径。

所有的源代码和解答都是用ANSI/ISO标准C++和标准C++进行代码编写、编译和测试的。程序可以使用Microsoft Visual C++ 6.0、Borland C++ Builder 5开发工具或Metrowerks Code Warrior进行编译。

目 录

第 1 章 计算机和编程语言回顾	1
1.1 导言	1
1.2 计算机历史的简要回顾	1
1.3 计算机系统的组成	2
1.4 计算机语言	3
1.5 编程语言的发展	4
1.6 处理高级语言程序	5
1.7 分析问题 - 编码 - 执行	6
1.8 面向对象的编程	9
1.9 ANSI/ISO 标准 C++	10
1.10 小结	10
1.11 练习	11
第 2 章 C++ 的基本要素	13
2.1 C++ 程序的基础知识	14
2.2 数据类型	16
2.3 算术运算符和运算符优先级	20
2.4 表达式	22
2.5 输入	25
2.6 自增和自减运算符	32
2.7 输出	34
2.8 预处理器指令	39
2.9 编程风格和形式	41
2.10 赋值语句的更多知识	45
2.11 编程实例: 长度转换	46
2.12 编程实例: 找零钱	48
2.13 小结	51
2.14 练习	52
2.15 编程练习	58
第 3 章 输入 / 输出	60
3.1 I/O 数据流和标准 I/O 设备	60
3.2 在程序中使用预定义函数	64
3.3 输入失败	69
3.4 输出和格式化输出	72
3.5 其他输出格式化工具	77

3.6	文件输入/输出	82
3.7	编程实例: 电影票销售和对慈善机构的捐助	84
3.8	编程实例: 学生分数	88
3.9	小结	91
3.10	练习	92
3.11	编程练习	93
第4章	控制结构 I (选择结构)	95
4.1	控制结构	95
4.2	关系运算符	96
4.3	逻辑(布尔)运算符和逻辑表达式	98
4.4	选择: if 和 if-else	104
4.5	switch 结构	117
4.6	用 assert() 函数来终止一个程序	122
4.7	编程实例: 有线电视公司账单	123
4.8	小结	127
4.9	练习	128
4.10	编程练习	132
第5章	控制结构 II (重复)	134
5.1	为什么需要重复结构	134
5.2	while 循环(重复)结构	135
5.3	编程实例: 检查账户余额	143
5.4	编程实例: 斐波那契数列	150
5.5	for 循环(重复)结构	153
5.6	编程实例: 数字分类	157
5.7	do-while 循环(重复)结构	159
5.8	break 和 continue 语句	161
5.9	嵌套控制结构	163
5.10	小结	164
5.11	练习	165
5.12	编程练习	172
第6章	用户自定义函数 I	174
6.1	标准(预定义)函数	174
6.2	用户定义函数	176
6.3	具有返回值的函数	176
6.4	编程实例: 最大数	184
6.5	编程实例: 有线电视公司账单问题	185
6.6	小结	190
6.7	练习	191
6.8	编程练习	194

第7章 用户自定义函数 II	196
7.1 void 函数	196
7.2 值参数、引用参数和内存分配	204
7.3 引用参数和返回值函数	213
7.4 变量的范围	213
7.5 全局变量的副作用	216
7.6 静态和自动变量	216
7.7 函数重载：介绍	218
7.8 有默认参数的函数	218
7.9 编程实例：对数字分类	220
7.10 编程实例：数据比较	224
7.11 小结	231
7.12 练习	232
7.13 编程练习	237
第8章 用户定义的简单数据类型、命名空间和 string 类型	240
8.1 枚举类型	240
8.2 编程实例：石头、剪子和布的游戏	247
8.3 命名空间	255
8.4 string 类型	259
8.5 编程实例：Pig Latin 字符串	266
8.6 小结	270
8.7 练习	271
8.8 编程练习	274
第9章 数组和字符串	276
9.1 数组	277
9.2 C 字符串（字符数组）	288
9.3 关联数组	292
9.4 编程实例：代码检测	293
9.5 编程实例：文本处理	298
9.6 小结	302
9.7 练习	303
9.8 编程练习	306
第10章 数组的应用和扩展	308
10.1 顺序表处理	308
10.2 二维和多维数组	320
10.3 多维数组	330
10.4 编程实例：选举结果	331
10.5 小结	344
10.6 练习	344

10.7 编程练习	346
第 11 章 递归	349
11.1 递归的定义	349
11.2 利用递归解决问题	351
11.3 递归还是迭代	359
11.4 编程实例: 将一个数字从二进制表示转换为十进制表示	360
11.5 编程实例: 将一个数字从十进制转换为二进制	363
11.6 小结	365
11.7 练习	365
11.8 编程练习	367
第 12 章 记录 (结构体)	369
12.1 记录 (结构体)	369
12.2 编程实例: 销售数据分析	379
12.3 小结	391
12.4 练习	391
12.5 编程练习	393
第 13 章 类和数据抽象	395
13.1 类	395
13.2 数据抽象、类和抽象数据类型	414
13.3 结构体与类	415
13.4 信息隐藏	416
13.5 可执行代码	418
13.6 编程实例: 糖果机	421
13.7 小结	431
13.8 练习	432
13.9 编程练习	434
第 14 章 继承和组合	437
14.1 继承	437
14.2 组合	452
14.3 面向对象设计 (OOD) 和面向对象编程 (OOP)	456
14.4 编程实例: 分数报告	457
14.5 小结	473
14.6 练习	474
14.7 编程练习	479
第 15 章 指针、类和虚函数	482
15.1 指针数据类型和指针变量	482
15.2 取址运算符 (&)	483
15.3 取值运算符 (*)	483

15.4	类、结构和指针变量	487
15.5	初始化指针变量	489
15.6	动态变量	490
15.7	指针变量的运算	491
15.8	动态数组	492
15.9	指针的浅 (shallow) 复制与深 (deep) 复制	494
15.10	类和指针: 一些特性	496
15.11	继承、指针和虚函数	504
15.12	取址运算符和类	508
15.13	小结	511
15.14	练习	512
15.15	编程练习	517
第 16 章	重载和模板	518
16.1	为什么需要运算符重载	518
16.2	运算符重载	519
16.3	编程练习: clockType	546
16.4	编程练习: 复数	551
16.5	重载数组索引 (下标) 运算符 ([])	554
16.6	编程练习: newString	556
16.7	函数重载	560
16.8	模板	561
16.9	小结	568
16.10	练习	570
16.11	编程练习	573
第 17 章	链表	578
17.1	链表	578
17.2	作为 ADT 的链表	587
17.3	有序链表	597
17.4	双向链表	606
17.5	编程实例: 音像商店	612
17.6	小结	627
17.7	练习	627
17.8	编程练习	630
第 18 章	栈和队列	632
18.1	栈	632
18.2	栈的数组实现	635
18.3	程序实例: 最大 GPA	645
18.4	栈的链表实现	648
18.5	栈的应用: 后缀表达式计算器	658

18.6 消除递归：非递归逆序打印链表的算法	664
18.7 队列	668
18.8 队列的应用：仿真	680
18.9 小结	697
18.10 练习	697
18.11 编程练习	700
附录 A 保留字	703
附录 B 运算符的优先级	704
附录 C 字符集	705
附录 D 运算符重载	707
附录 E 在 ANSI/ISO 标准 C++ 和 标准 C++ 中头文件的名称约定	708
附录 F 头文件	709
附录 G 系统中数据类型的内存尺寸	714
附录 H 部分练习答案	715

第1章 计算机和编程语言回顾

本章要点:

- 学习不同类型的计算机
- 了解计算机系统的软件和硬件
- 学习计算机语言
- 了解编程语言的发展
- 了解高级编程语言
- 了解编译器及其功能
- 了解如何执行高级语言
- 学习算法并了解解决问题的技能
- 学习结构化设计和面向对象设计的编程方法
- 了解标准 C++ 和 ANSI/ISO 标准 C++

1.1 引言

在信息社会中,信息处理的速度已经变得非常迅速。通过计算机,技术革命彻底地改变了人们的生活和交流方式。前些年提到“Internet”这个词时可能还会觉得陌生,但是现在却是家喻户晓了。计算机可以帮助用户在几秒钟之内收发好友的信件;用户不必再到邮局去邮递简历以申请工作,而只需要通过Internet就可以递交简历了;股民可以了解实时的股市信息,以决定买入还是抛售股票;孩子们可以定期地进行网上冲浪,还可以使用计算机设计他们的教室。学生们再也不必用打字机或手写方式来完成家庭作业了,只要用功能强大的字处理软件就可以完成作业;很多人还使用计算机来管理支票簿。

从上面这些例子可以看出,计算机对人们日常生活的影响是巨大的。而这些要成为可能则需要借助不同的软件,也就是计算机程序。例如,字处理软件可以帮助完成家庭作业、制作精美的简历甚至撰写书籍,本书就是在功能强大的字处理软件的帮助下制作完成的。没有软件,计算机也就失去了作用。而软件则是由编程语言开发的。C++ 语言特别适合于开发软件以完成特定的任务。

在20世纪90年代以前,教师在讲授程序课程之前往往要花几个星期的时间教授学生如何使用计算机。而现在,学生们中学毕业的时候就已经能够熟练地操作计算机了。本书的主要内容不是讲述如何使用计算机,而是教授读者如何使用C++来编制程序。但是在开始编程之前,了解一些基本术语和计算机的不同组成部件是很有好处的。本章简要介绍一下计算机系统的主要部件、计算机语言的历史和发展以及如何解决编程过程中遇到的问题。

1.2 计算机历史的简要回顾

20世纪50年代,计算机是一种体积很大的设备,只有少数人能接触到。所有工作,包括外科手术、会计、字处理以及其他任务中的计算都是在没有计算机的帮助下完成的。到20世纪60年代,出现了价值上百万美元的计算机,只有大公司才能买得起。这些计算机体积仍旧很大,只有计算机

专家才能使用。20世纪70年代中期,计算机的价格已经很便宜而且体积也变得很小。到了20世纪90年代中期,各行各业的人们都能买得起计算机了。20世纪90年代末期,小型计算机更加便宜而且运行速度也更快。

1.3 计算机系统的组成

计算机是能执行一系列命令的电子设备。计算机执行的基本命令是输入(得到数据)、输出(显示结果)、存储、执行算术和逻辑运算。

在市场中,销售计算机时都这样描述: Pentium 4 1.4 GHz 处理器、128 MB 内存、40 GB 硬盘、19英寸SVGA显示器,同时还有预装的软件,比如操作系统、游戏、百科全书以及字处理和货币管理等应用程序软件。这段描述涉及两类内容:硬件和软件。Pentium 4 1.4 GHz 处理器、128 MB 内存、40 GB 硬盘、19英寸显示器这一类归为硬件;而操作系统、游戏、百科全书以及字处理和货币管理等应用程序软件则属于软件一类。下面先看一下硬件。

1.3.1 硬件

硬件主要包括下面几组部件:中央处理单元(CPU)、主存(也称为随机访问存储器)、输入输出设备和辅助存储器。键盘、鼠标和辅助存储器等都是输入设备,而输出设备则包括显示器、打印机和辅助存储器等。下面更详细地介绍一下这些部件。

中央处理单元

中央处理单元(CPU)是计算机的中枢。CPU的功能越强大,运行速度就越快。CPU包括几个部分:控制单元(CU)、程序计数器(PC)、指令寄存器(IR)、算术逻辑单元(ALU)和累加器(ACC)。图1.1显示了这几个部分是如何结合在一起的。

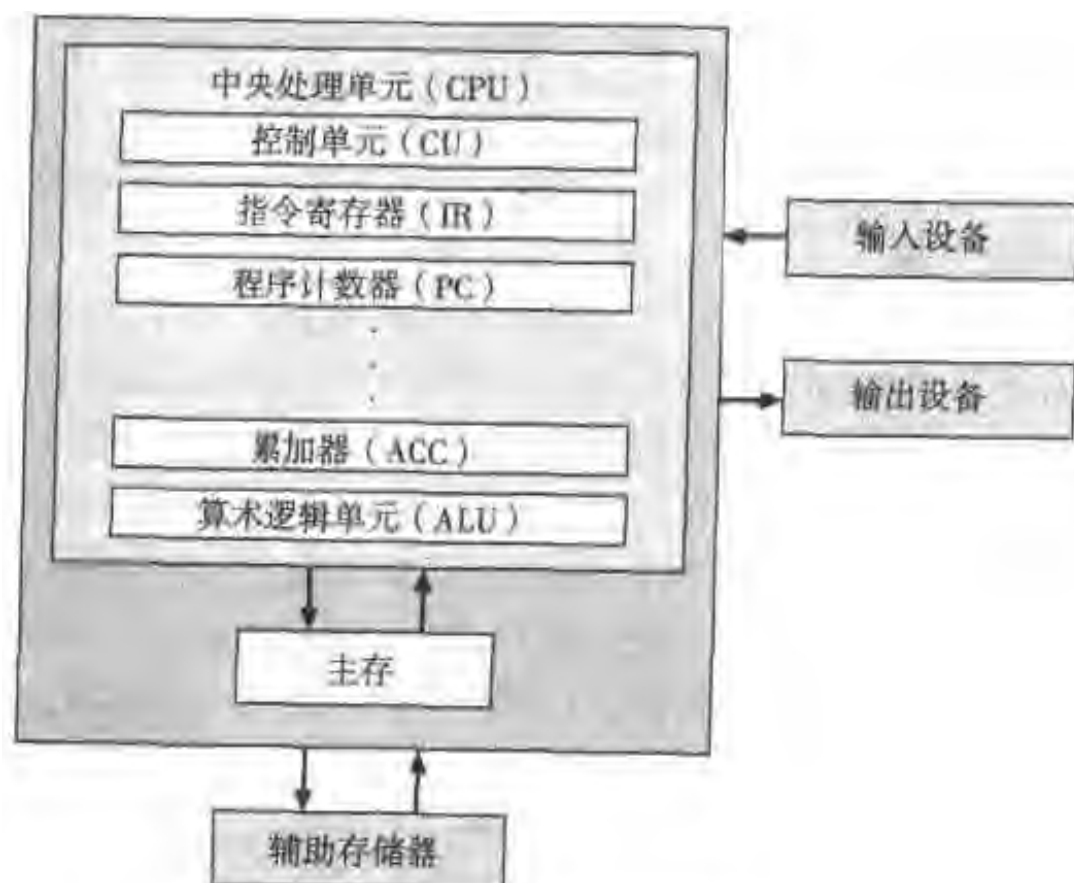


图 1.1 计算机硬件的组成

- 与CPU相似,控制单元(CU)是CPU的大脑。它有三个主要功能:取指令并翻译指令,控制信息流(指令或数据)进出主存储器,控制CPU内部组件的操作。
- 程序计数器(PC)指向下一条要执行的指令。

- 指令寄存器 (IR) 保存当前正在被执行的指令。
- 算术逻辑单元 (ALU) 执行所有的算术和逻辑操作。
- 累加器 (ACC) 保存算术逻辑单元执行后的运算结果。

主存

主存直接与 CPU 相连。所有的程序在执行前必须被装载到主存中。同样, 所有的数据在程序进行操作之前必须调入主存。当计算机关机后, 主存中的所有数据会永久地丢失。

主存是由一系列存储单元组成的。每一个单元在主存里有唯一的位置, 称为单元地址。这些地址可以帮助访问存储在单元里的信息。图 1.2 显示了包含 100 个存储单元的主存。

现代计算机的主存包含几百兆甚至更多的存储单元。尽管在图 1.2 中显示了单元中存储的数据, 但是单元中的内容既可以是程序指令, 也可以是数据。

0	54
1	A
...	...
98	Hi
99	12.5

图 1.2 有 100 个存储单元的主存

辅助存储器

因为处理程序和数据时, 程序和数据必须存储在主存中。因为当计算机关机时, 主存中的所有数据会丢失, 所以主存中的数据必须被转移到其他设备中以进行永久保存。能永久存储信息的设备 (除非设备不能用或通过重写而改变了信息) 称为辅助存储器。为了能从主存向辅助存储器传输数据, 各个组成部分应该被正确地相互连接。硬盘、软盘、Zip 盘、CD-ROM 以及磁带等都属于辅助存储器。

输入/输出设备

对于执行任务的计算机来说, 它必须能够接收数据和程序并且显示计算的结果。能把数据和程序输入到计算机中的设备称为输入设备。键盘、鼠标和辅助存储器都属于输入设备。计算机用来显示结果的设备称为输出设备。显示器、打印机以及辅助存储器都是输出设备。

1.3.2 软件

用来执行特定任务的程序称为软件。例如, 字处理程序能用来撰写书信、论文甚至书籍。有两种类型的程序: 系统程序和应用程序。

系统程序控制计算机。当打开计算机时, 首先被装载的系统程序称为操作系统。没有操作系统, 计算机就不能使用。操作系统监测计算机所有的活动情况并且提供相应的服务。其中的一些服务包括: 内存管理、输入/输出活动以及存储管理。操作系统有一个特殊的程序来管理辅助存储器, 这样就可以很容易地访问信息。而且, 操作系统是使用编程语言实现的。

应用程序执行特定的任务。字处理软件、电子表格以及游戏等都属于应用程序。所有的应用程序都是使用编程语言实现的。操作系统是能够运行应用程序的程序。

1.4 计算机语言

当在键盘上按下字母 A、屏幕上显示 A 的时候, 到底何种信息被存储在计算机的主存里? 而什么又是计算机语言? 当敲击键盘时, 字母是如何被存储的?

应该知道, 计算机是由电子器件组成的, 电子信号在计算机中传输。有两种类型的电子信号: 模拟信号和数字信号。模拟信号是用连续不断的波形来表示像声音之类的信号。比如音乐磁带存储

的数据就是模拟信号。而数字信号是用一系列的0或1代表信息，0代表低电压，1代表高电压。与模拟信号相比，数字信号是更可靠的信息载体，并且可以准确地从一个设备复制到另一个载体。读者是否注意到，复制磁带中的声音效果不如母带中的声音效果呢？

由于数字信号在计算机中进行处理，计算机语言最终（称为机器语言）是由一系列0和1组成的。数字0和1称为二进制数或位（bit）。有时，一系列的0和1也称为二进制码。

位（bit）：二进制的数字0或1。

8位的序列称为字节（byte）。键盘上的每个字母、数字或特殊的符号（比如*或{）被编码为一系列的二进制数，每一个有其惟一的含义。

计算机中最常用的编码方案是7位的美国信息交换标准码（ASCII）。ASCII数据集包含128个编号为0到127的字符。也就是说，在ASCII数据集中，第一个字符的位置是0，第二个字符的位置是1，等等。在该方案中，A编码为1000001。实际上，在ASCII码中，A是第66个字符，但它的位置是65，因为第一个字符的位置是0。而且，1000001是65的二进制表示。字符3被编码为0110011。对于完整的ASCII数据集，可以参考附录C。

在计算机内，每一个字符是由8位的序列表示的，也就是一个字节。因为ASCII码是7位编码代码，所以必须在ASCII编码的左侧添加0。这样，在计算机内，字符A由01000001表示，字符3由00110011来表示。

也有其他编码方案，比如EBCDIC（IBM使用）和Unicode（它是最近才得到发展的）。EBCDIC包含256个字符；Unicode包含65 536个字符。为了存储一个Unicode的字符，需要两个字节。

1.5 编程语言的发展

计算机最基本的语言，即机器语言，是以位来提供程序指令。即使大部分计算机执行相同类型的操作，计算机的设计者仍然可以选择不同集的二进制代码来执行操作。所以，一台计算机的机器语言不必与另一台计算机的机器语言相同。计算机惟一的相同在于：在任何一台现代的计算机中，所有的数据是以二进制代码进行存储和操作的。

早期的计算机是用机器语言进行编程的。为了查看计算机指令是如何用机器语言编写的，可以假定用公式：

$$\text{工资} = \text{价格} * \text{时间}$$

来计算周薪。而且假定二进制代码100100代表载入指令，100110代表乘法指令，100010代表存储指令。在机器语言中，可能需要下列的指令来计算周薪：

```
100100 0000 010001
100110 0000 010010
100010 0000 010011
```

为了用机器语言代替周薪公式，编程人员不得不记住各种操作的机器语言。而且，为了对数据进行操作，编程人员还需记住主存中数据的位置。这需要记住特定的代码，从而使编程变得不仅困难，而且容易出错。

随着汇编语言的出现，编程人员的工作变得容易起来。在汇编语言中，指令用一种称为助记符的易记形式来表示。例如，表1.1显示了指令用汇编语言表示的一些例子及其相应的机器语言代码。

表 1.1 用汇编语言和机器语言表示指令的一些例子

汇编语言	机器语言
LOAD	100100
STOR	100010
MULT	100110
ADD	100101
SUB	100011

使用汇编语言指令，可以用下面的方案来计算周薪：

```
load rate
mult hours
stor wages
```

正如所看到的，用汇编语言编写指令要容易得多。然而，计算机并不能直接执行汇编语言指令。指令首先必须翻译成机器语言。称为汇编器（assembler）的程序把汇编语言翻译成机器语言。

汇编器：汇编器是一个程序，能把汇编语言编写的程序翻译成一个等效的机器语言表示的程序。

从机器语言发展到汇编语言使编程变得更为容易，但是编程人员仍然必须根据个别机器指令来考虑整个程序。使编程变得更为简单的是高级语言，它更接近实际语言，比如英语、法语、德语或西班牙语。Basic, FORTRAN, COBOL, Pascal, C, C++ 和 Java 都是高级语言。在本书中，读者将会学习 C++ 这种高级语言。

可以使用 C++ 来编写下面的周薪公式：

```
wages = rate* hours;
```

用 C++ 编写的指令更加易懂，并且对一名熟悉基本算术的初学者来说具有自解释性。然而对于汇编语言来说，计算机不能直接执行用高级语言编写的指令。为了在计算机中运行这些 C++ 指令，首先需要将它们翻译成机器语言。称为编译器的程序把这些高级语言编写的指令翻译成机器代码。

编译器：编译器是一个程序，把用高级语言编写的程序翻译成等效的机器语言。

1.6 处理高级语言程序

通过上面的介绍，读者对什么是机器语言和什么是高级语言应该有了一定的了解，而且应该认识到计算机只能读懂机器语言，现在回顾一下为了执行高级语言程序所需要的步骤。需要执行下面的 5 个步骤，如图 1.3 所示。

1. 使用编辑器来创建（也就是输入）一个高级语言程序，在这一过程中应该遵循高级语言的编程规则和语法。该程序称为源程序。

源程序：用高级语言编写的程序。

2. 必须保证程序遵守编程语言的规则——也就是说程序在语法上讲是正确的，同时能保证把程序翻译成等价的机器语言。编译器检查语法错误并且把程序翻译成机器语言。编译器检查源程序的语法错误，如果没有发现错误，就会把程序翻译成等价的机器语言。等价的机器语言称为目标程序。

目标程序：高级语言程序的机器语言版本。

3. 高级语言程序是用软件开发包（SDK）来开发的。SDK 包含很多在创建程序过程中有用的程序。例如，它包含必要的代码来显示程序的结果，还包含一些具体的函数，从而使程序员的工作变得更容易一些。所以，如果特定的代码已经可用，程序员就可以使用这些特定的代

码，而不必来写自己的代码。即使程序开发完成并且成功地被编译，仍然必须使用 SDK 中的一些代码，以产生可执行的程序。预写代码驻留的地方称为库。称为连接程序的程序将目标程序和库中的程序结合起来。

连接程序：能把目标程序和 SDK 提供的其他程序连接起来并创建可执行代码的程序。

4. 必须把可执行程序装载到主存中以执行程序。称为装载程序的程序完成该任务。

装载程序：能把可执行程序装载到主存中的程序。

5. 最后一个步骤是执行程序。

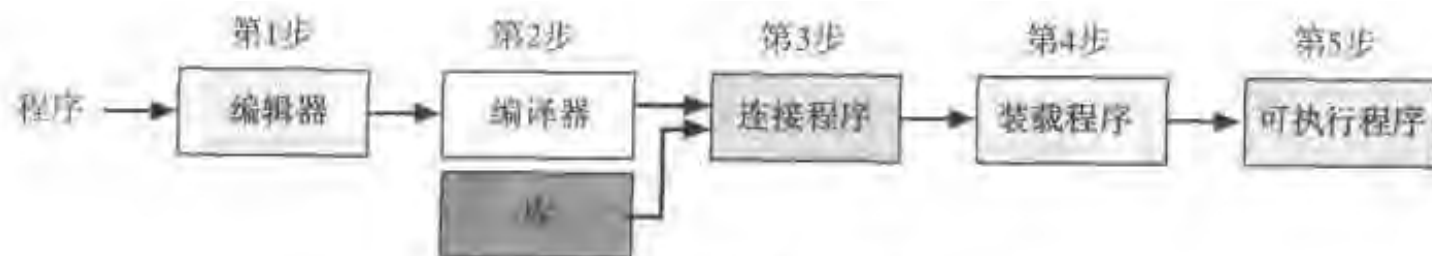


图 1.3 执行高级语言程序

作为一名程序员，只需要把精力放在第一步。也就是说，读者必须学习、了解并且掌握编程语言的规则以创建源程序。前面说过，程序是通过 SDK 进行开发的。在 C++ 中，比较流行的 SDK 包括 Visual C++（来自于微软）、C++ Builder（来自于 Borland）和 Code Warrior（来自于 Metrowerks）。这些 SDK 包含创建程序的编辑器、用来检查程序语法错误的编译器、连接目标程序与 SDK 源代码的程序以及用来执行程序的程序。而且，对于用户来说，这些 SDK 是很友好的。

当编译一个程序时，编译器不仅标记出语法错误，而且还会给出如何改正错误的一般性建议。而且用一条简单的命令就可以把目标代码与 SDK 中所使用的源代码连接起来。对于 Visual C++ 来说，进行连接的命令是 Build 或 Rebuild；对于 C++ Builder 来说，该命令是 Build 或 Make；而 Code Warrior 的连接命令是 Make。（如果想得到关于使用这些命令的详细分类，可以查看这些 SDK 中的文档。）如果程序还没有被编译，那么这些命令中的每一条命令首先都会编译程序，然后连接和产生可执行代码。

注意：通常，高级语言开发要经历上面所阐述的 5 个步骤。然而一个 C++ 程序却要经历 6 个步骤。额外的步骤发生在第一步创建源程序之后、第二步编译源程序之前。当然，SDK 会自动处理这一额外的步骤。在第 2 章中，在介绍 C++ 基本要素之后，读者将会看到 C++ 程序是如何被处理的。作为程序员，正如上面提到的，仍然应该把精力放在第一步。也就是说，读者必须掌握创建 C++ 源程序的规则。

1.7 分析问题 – 编码 – 执行

编程是解决问题的过程，不同的用户使用不同的技术来解决问题。一些技术被很好地概括并且容易操作。它们不仅解决问题而且能给出如何解决问题的直觉。如果问题的范围发生变化，可以很容易地修改这些解决问题的技术。

为了成为熟练解决问题的人并且成为一名好的程序员，必须遵守好的解决问题的技术。解决问题的技术包括分析问题、概括解决问题需要的条件、设计步骤并运用算法来解决问题。

算法：一步步解决问题的过程。在这一过程中使用有限的时间来得到解决方法。

在编程环境中，问题解决过程需要下面三个步骤：

1. 分析问题、概括问题以及解决问题的必要条件、设计解决问题的算法。
2. 用编程语言（例如 C++）执行算法并且证明该算法是可行的。
3. 维护使用的程序，如果问题范围发生改变则修改该程序。

图 1.4 概括了这三个编程过程。

为了开发能解决问题的软件，首先应该分析问题，然后设计算法并用高级语言编写程序指令，或者编写程序代码把程序应用于计算机系统。分析问题是第一步，也是最重要的步骤。这一步骤需要下面的工作：

1. 完全理解需要解决的问题。
2. 了解解决问题需要的必要条件。这些必要条件包括：程序是否需要与用户建立联系，程序是否处理数据，程序是否有输出，结果看起来应该是什么样子。如果程序要对数据进行操作，那么程序员必须知道数据是什么以及这些数据代表什么。也就是说，程序员需要查看样本数据。如果程序产生输出结果，程序员应该知道怎样产生结果以及以何种形式来输出结果。
3. 如果问题比较复杂，应该把程序分为若干个子问题并且重复步骤 1 和 2。也就是说，对于复杂问题来说，需要分析每一个子程序并且理解解决每一个子问题的必要条件。

把问题分解为若干个子问题的方法称为结构化设计。结构化设计方法也称为自上而下的设计、逐步细化和模块化编程。在结构化设计中，问题被分解为若干个较小的问题。然后把所有子问题的解决方法结合在一起来解决整个问题。执行结构化设计的过程称为结构化编程。

在仔细分析问题以后，下一步是设计算法来解决问题。如果把问题分成许多个子问题，则需要为每一个子问题设计算法。一旦设计完算法，就需要对它进行错误检查。有时可能需要一些样本数据来测试算法的正确性，也可能需要进行一些具体的分析来测试算法的正确性。

在设计完算法并对其进行正确性测试之后，下一步骤是把等价的代码转换成相应的高级语言。程序员可以使用文本编辑器把程序输入到计算机中。必须保证程序没有语法错误。为了验证语法的正确性，需要通过编译器来运行代码。如果编译器产生了错误信息，则必须识别出代码中的错误并改正这些错误，然后再使用编译器来运行代码。当所有的语法错误被改正后，编译器会生成等价的机器代码，连接程序把机器代码与系统源代码连接起来，装载器把程序载入主存加以执行。

最后一个步骤是执行程序。编译器只保证程序没有语法错误，并不能保证程序没有错误而正确运行。在程序执行期间，程序可能因为逻辑错误而异常终止，例如 0 作为除数。即使程序正常结束，它仍然可能产生错误结果。在这些情况下，可以重新检查代码、算法甚至是程序分析。

正如所看到的，在试图编写程序指令之前需要做大量的工作。通常，可以在书面上完成这些工作。采用这种仔细的方法进行编程有很多好处。这样，在一个经过良好分析、设计的程序中能更容

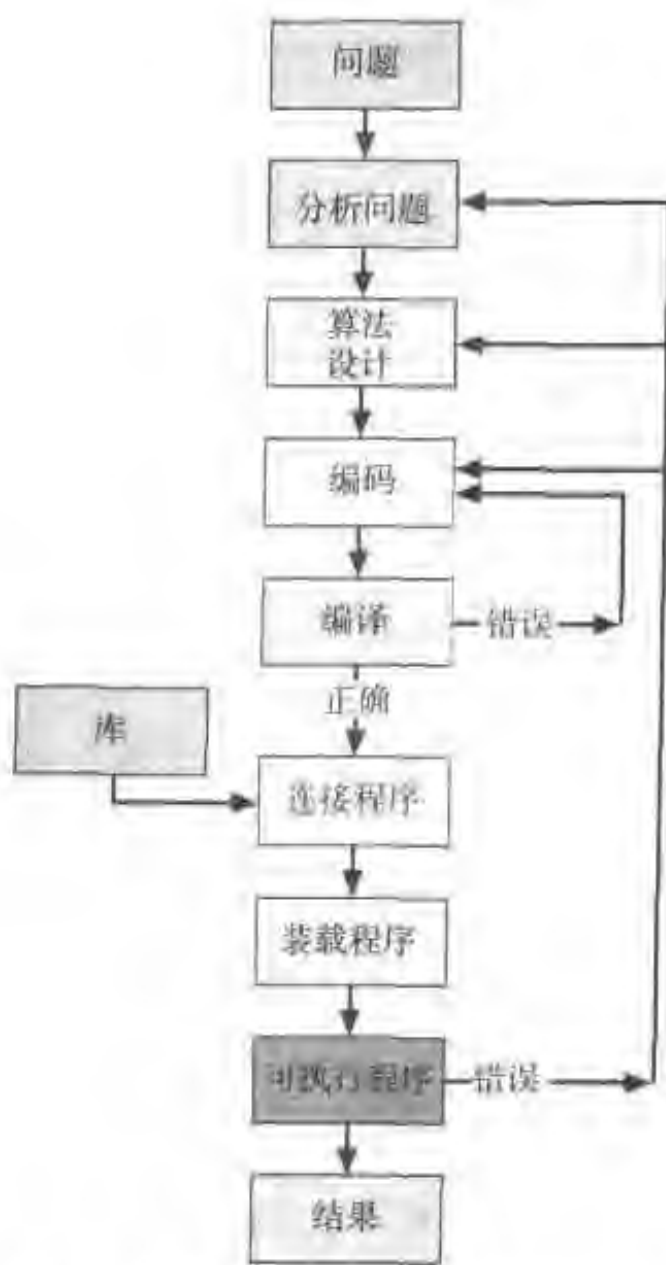


图 1.4 问题分析 - 编码 - 执行循环结构图

易发现错误。而且,一个经过仔细分析和设计的程序更容易理解和修改。即使是一名有着丰富编程经验的程序员,也会花费很多的时间来分析和设计算法。

通过对本书的学习,不仅可以学会编写C++的规则,还可以了解解决问题的方法。每一章中都会讨论几个编程问题,每一个问题都会有一些例子。这些编程例子不仅给出了分析和解决问题的方法,也会帮助理解这些章节中所讨论的概念。为了能最大限度的利用本书,推荐读者在每一章的末尾能解决这些编程实例。

例 1.1

在本例中,设计算法来计算矩形的周长与面积。

为了计算矩形的周长与面积,需要知道矩形的长度和宽度。矩形的周长和面积则由下面的算法给出:

```
perimeter = 2 * (length + width)
area = length * width
```

算法计算出了矩形的周长和面积,所以:

1. 得到了矩形的长度。
2. 得到了矩形的宽度。
3. 用下面的算法计算周长:

```
perimeter = 2 * (length + width)
```

4. 用下面的算法计算面积:

```
area = length * width
```

例 1.2

在本例中,设计一个算法来计算当地百货公司一位售货员的月薪。

每一位售货员拥有底薪。在每个月底,售货员还可以按照下面的标准得到奖金:如果售货员在百货公司已经工作5年或5年以下,奖金每年是10美元;如果售货员工作5年以上,奖金是每年20美元。售货员还可以得到一份额外的奖金:如果售货员该月的销售额在5000美元以上、10000美元以下,可以得到3%的回扣。如果该月的销售额在10000美元以上,可以得到6%的回扣。

为了计算售货员的月薪,需要知道底薪、售货员在百货公司工作的年数以及售货员该月的销售总额。假定baseSalary代表基本工资,noOfServiceYears代表该售货员在公司工作的年数,bonus代表奖金,totalSale代表售货员该月的销售总额,additionalBonus代表额外的奖金。

奖金可以由下面的算法给出:

```
if(noOfServiceYears is less than or equal to five)
    bonus = 10*noOfServiceYears
otherwise
    bonus = 20*noOfServiceYears
```

下一步,可以由下面的算法计算出售货员的奖金:

```
if(totalSale is less than 5000)
    additionalBonus = 0
otherwise
    if(totalSale is greater than or equal to 5000 and totalSale is
        less than 10000)
        additionalBonus = totalSale*(0.03)
```

```
otherwise
    additionalBonus = totalSale * (0.06)
```

通过上面的讨论,现在可以设计算法来计算售货员的月薪。

1. 得到 baseSalary
2. 得到 noOfServiceYears
3. 使用下面的算法计算 bonus:

```
if(noOfServiceYears is less than or equal to five)
    bonus = 10 * noOfServiceYears
otherwise
    bonus = 20 * noOfServiceYears
```

4. 得到 totalSale
5. 使用下面的算法计算额外奖金:

```
if(totalSale is less than 5000)
    additionalBonus = 0
otherwise
    if(totalSale is greater than or equal to 5000 and totalSale is
        less than 10000)
        additionalBonus = totalSale * (0.03)
    otherwise
        additionalBonus = totalSale * (0.06)
```

6. 使用下面的算法计算月薪:

```
payCheck = baseSalary + bonus + additionalBonus
```

1.8 面向对象的编程

前面讨论了结构化设计的编程方法论。本节将简要介绍称为面向对象设计(OOD)的编程方法论。

在OOD中,解决问题的第一个步骤是识别称为对象的组件,这些对象组成了解决方法的基础,并且决定了这些组件之间是如何发生相互作用的。例如,假定编写一个程序使当地一家音像店的出租业务实现自动化。在这一问题中主要的两个对象是影碟和顾客。

在确定了对象之后,下一个步骤是为每一个对象指定相应的数据以及对数据进行处理的方法。例如,对于影碟对象来说,数据可能包括电影的名字、著名影星、制片人、制片公司以及库存的拷贝数量,等等。对影碟对象的操作可能包括检查电影的名字、拷贝出租之后对于库存量的减法操作(以及顾客归还影碟后对库存量的加法操作)。

这里举例说明了每一个对象包含的数据以及对数据进行的操作。对象把数据以及对数据进行的操作结合成一个单元。在OOD中,最终的程序是相互作用的对象。执行OOD的编程语言称为面向对象的编程语言(OOP)。在后面的章节中,读者可以了解到OOD的许多优点。

因为对象包含数据以及对数据进行的操作,所以在设计和使用对象之前需要了解如何在内存中描述数据、如何对数据进行操作以及如何执行操作。在第2章中,读者将会了解到C++的基本类型,并会发现如何描述数据以及对数据进行操作。第3章将讨论如何向C++程序中输入数据以及输出C++程序的结果。

为了创建操作,需要使用编程语言编写算法并执行该算法。因为复杂程序中的数据元素通常包含许多操作,所以为了把每一个操作相互分开并且有效便捷地使用它们,需要使用函数来执行算法。

在第2章和第3章中简短地对函数进行介绍之后,将在第6章和第7章中对函数进行详细地介绍。需要进行决策的算法称为选择结构。其他算法可能需要特定的语句重复执行直到符合条件为止,这称为循环结构。而另一些算法可能既要用到选择也要用到循环。在第4章和第5章中,将会了解到选择和循环机制,称为控制结构。在第9章和第10章中,通过使用称为数组的机制,将会了解到当数据项类型相同(例如销售表中的项目)时如何对数据进行操作。

最后,为了能使用这些对象,需要知道怎样把数据和对数据进行的操作结合成一个单元。在C++中,允许把数据和对数据进行的操作结合成单元的机制称为类。在第13章中,将会了解到类是如何工作、如何使用以及如何创建一个类。

正如上面看到的,在学会使用OOD方法论之前需要了解很多事情。为了使学习更容易、更有效,本书将控制结构分为两章(第4章和第5章),用户定义函数分为两章(第6章和第7章),数组分为两章(第9章和第10章)。

对于一些问题来说,编程采用结构化方法将会变得更有效。用OOD的方法将会使问题得到很好地阐述。例如,如果问题需要用数学函数对一组数字进行操作,则可以使用结构化设计方法并且概括出为了解决问题需要的步骤。C++库提供了大量的函数,这样可以有效地对数据进行操作。另一方面,如果需要编写程序来进行糖果机的操作,OOD方法会更有效。C++是特别设计用来执行OOD的。而且,OOD十分有效并且能与结构化设计联合使用。

结构化设计和OOD方法都需要掌握编程语言的基本组成,这样才能成为一名更有效率的程序员。在下面几章中,将会了解到两种类型中C++的基本组件。在介绍基本概念之后,第2章到第12章中将会使用结构化编程语言的方法对特定问题进行解决和编程。从第13章以后,本书主要将重点放在如何使用OOD来解决问题和编程上。

1.9 ANSI/ISO 标准 C++

20世纪80年代早期,编程语言C++由C发展而成,它是由贝尔实验室的Bjarne Stroustrup设计的。从20世纪80年代早期到20世纪90年代早期,出现了几种有效的C++编译器。尽管所有的C++编译器的基本特征通常都相同,但是本书所讲的标准C++语言在不同的编译器中进行编译时,其结果会有一些不同,所以C++程序从一个编译器到另一个编译器并不总是可以移植的。

为了解决这一问题,在20世纪90年代早期,美国国家标准化组织(ANSI)和国际标准化组织(ISO)成立了一个联合委员会来标准化C++语法。在1998年中,批准通过了ANSI/ISO C++语言标准。现在的大多数编译器都符合这一新标准。

本书主要集中讲述ANSI/ISO批准通过的C++的语法,该C++也称为ANSI/ISO标准C++。然而,可能偶然遇到用标准C++编写的代码(或者老式编译器),所以在后面少数几章中会说明在ANSI/ISO标准C++程序中需要进行的修改。尽管标准C++和ANSI/ISO标准C++大部分相同,但是ANSI/ISO标准C++有自己的特征。在本书中,这些特征将会显式地表现出来并且给予讨论。

1.10 小结

1. 计算机是一种电子器件,能够执行算术和逻辑运算。
2. 计算机系统包含两种组件:硬件和软件。
3. 中央处理单元(CPU)和主存是硬件组件的例子。
4. 控制单元(CU)控制程序的全部执行。它是CPU组件的几个部分之一。
5. 算术逻辑单元(ALU)是执行算术和逻辑运算的组件。

6. 在执行程序之前, 所有程序必须进入内存。
7. 当电源关掉后, 内存中的所有数据都会丢失。
8. 辅助存储设备对信息提供永久的保存。
9. 向计算机输入信息是通过输入设备完成的。常用的两种输入设备是键盘和鼠标。
10. 计算机将输出结果输出到输出设备中, 例如计算机显示器。
11. 软件是计算机运行的程序。
12. 操作系统监视计算机所有的活动并且提供相应服务。
13. 计算机中最基本的语言是由一系列0和1组成的机器语言。
14. 一位是一个二进制数字: 0或1。
15. 一个字节是一个8位序列。
16. 汇编语言使用易记的助记符指令。
17. 汇编器是把汇编语言翻译成机器语言的程序。
18. 编译器是把高级语言程序翻译成机器代码的程序。
19. 连接程序把目标代码与其他软件开发包提供的程序连接起来, 以用来产生可执行代码。
20. 在高级语言中, 需要5个步骤执行程序: 编辑、编译、连接、装载和执行。
21. 装载器把可执行代码传送到主存中。
22. 算法是一步步解决问题的过程。在这一过程中, 将在有限的时间内解决问题。
23. 解决问题的过程包含三个步骤: 分析问题和设计算法, 用编程语言执行算法, 维护程序。
24. 使用结构化设计方法编写的程序更容易理解、测试和调试, 同时也更容易修改。
25. 在结构化设计中, 问题被分解成更小的子问题。每一个子问题被解决后, 然后把所有的解决方法结合起来。
26. 在面向对象设计(OOD)中, 程序是相互作用的对象集合。
27. 对象包含数据和对数据进行的操作。
28. ANSI/ISO 标准C++语法在1998年中期通过批准。

1.11 练习

1. 判断下面句子的对错。
 - a. 汇编语言是使用助记符作为指令的语言。
 - b. 编译器把汇编语言程序翻译成机器代码。
 - c. 算术逻辑单元执行算术操作, 如果发现错误, 它会输出逻辑错误。
 - d. 装载器把目标代码从主存装载到CPU加以执行。
 - e. 高级语言程序开发包括6个步骤。
 - f. CPU在控制单元的控制下运行。
 - g. RAM代表保存信息的存储器。
 - h. 用高级编程语言编写的程序称为源程序。
 - i. 当打开电源时, 操作系统是第一个被装载到计算机的程序。
 - j. 解决问题过程的第一个步骤是分析问题。
2. 指出中央处理单元中一些组件的名字。
3. 控制单元的功能是什么?
4. 指出两种输入设备。

5. 指出两种输出设备。
6. 为什么需要辅助存储器?
7. 为什么需要把高级语言程序翻译成机器语言?
8. 为什么要用高级语言编写程序, 而不用机器语言进行编写?
9. 在使用高级语言编程过程中, 问题分析和算法设计的优点是什么?
10. 设计一个算法, 计算 4 个测试成绩的加权平均值。分数和权值以下面的方式给出:

```
testscore1 weight1
...
```

样本数据如下所示:

```
75 0.20
95 0.35
85 0.15
65 0.30
```

11. 推销员每周一离开家, 周五返回并乘坐公司的汽车。每天的路上, 推销员会记录加到汽车中的汽油数量。里程表的起始和最后读数如下所示, 设计一个算法计算每加仑汽油行驶的公里数。

样本数据如下:

```
68723 71289 15.75 16.30 10.95 20.65 30.00
```

第2章 C++ 的基本要素

本章要点:

- 熟悉 C++ 程序的基本组成部分, 包括函数、特殊符号和标识符
- 研究简单的数据类型和字符串数据类型
- 学习怎样使用算术运算符
- 分析程序如何求解算术表达式
- 了解什么是赋值语句及其功能
- 学习如何使用输入语句
- 熟悉自增和自减运算符
- 学习如何使用输出语句输出结果
- 学习使用预处理器指令
- 研究如何正确地构造一个程序, 包括使用注释来说明一个程序
- 学习如何编写 C++ 程序

在本章中读者将会学习 C++ 的基础知识。由于目标是学习 C++ 编程语言, 自然地产生了两个问题。首先, 什么是计算机程序? 第二, 什么是程序设计? 计算机程序是由一系列的语句组成的, 其目的是为了完成一项任务。程序设计是计划并创建程序的过程。这两个定义讲述了程序设计的事实, 当然这并不是全部, 甚至可以写一本书来给出一个好的并且满意的程序设计定义。我们可以用推理的方法很好地掌握程序设计的特性, 所以现在把话题转向众人皆知的知识——烹饪。菜谱也是一个程序, 有烹饪经历的每一个人都会同意下面的观点:

1. 通常, 按照菜谱做菜比创建一个菜谱容易。
2. 有好菜谱, 也有差的菜谱。
3. 一些菜谱很容易照着做, 而一些菜谱却很难。
4. 一些菜谱产生可靠的结果, 而一些却不是。
5. 为了按照一个菜谱烹饪, 必须要有怎样使用烹饪工具的知识。
6. 为了创建新的菜谱, 必须有一定的烹饪知识。

上面6点对编程也同样适用, 让我们用烹饪来做进一步的类比。假定要使某人成为一名厨师, 该如何进行呢? 会首先向他介绍好的食品, 希望通过品尝之后能有所进展吗? 会让他按照菜谱来做, 以学会多种菜谱吗? 还是首先教他使用工具, 了解配料、食品和调料的特性, 解释怎样把它们调和在一起呢? 正如对于如何教授烹饪有不同的意见一样, 对于如何进行编程也有不同的意见。

学习编程语言就像学习烹饪或学习演奏乐器一样。这三种技术都需要与工具发生直接的相互作用。不能只通过读菜谱就能成为一个优秀的厨师。同样, 通过阅读关于乐器方面的书籍也不能成为一名演奏家。这些同样适用于编程。读者必须对语言有一定的基本了解, 必须在计算机中测试程序以保证每个程序能完成所设想的任务。

2.1 C++ 程序的基础知识

C++ 程序是一个或多个子程序（称为函数）的集合。一般地讲，子程序或函数是语句的集合，当它被调用或执行时，会完成一定的任务。一些称为预定义或标准函数的函数已经被写成，并且作为系统的一部分。但是为了完成大部分的任务，程序员必须学会写自己的函数。

每一个 C++ 程序都有一个 `main()` 函数。这样，如果 C++ 程序只有一个函数，那么它必须是 `main()` 函数。在第 6 章之前，除了使用一些预定义的函数之外，主要讲述 `main()` 函数。在本章末尾，将会了解怎样编写一个 `main()` 函数。

下面是 C++ 程序的一个例子。在这个例子中不必关心程序的细节。

例 2.1

下面显示了一段 C++ 程序。

```
#include <iostream>
using namespace std;
int main()
{
    cout<<"Welcome to C++ Programming"<<endl;
    return 0;
}
```

如果执行这个程序，它会在屏幕上打印下面一行：

```
Welcome to c++ Programming
```

如果读者从来没有接触过高级语言程序，那么上面例 2.1 的 C++ 程序看起来像外语。为了使语句更有意义，必须学习它的字母、单词和语法，而编程也是如此。为了编写有意义的程序，必须学习编程语言的特殊符号、单词和语法规则。语法规则（syntax rules）让系统知道哪些语句（指令）是合法的、可被编程语言接受的，哪些是不行的。同时也必须学会语义规则（semantic rules），它决定了指令的意义。编程语言的规则、符号和关键字使用户能够写出程序以解决问题，语法规则决定哪些指令是有效的。

编程语言：一系列规则、字符和关键字的集合。

可以用几种方法来描述一种编程语言的语法规则。用来描述语法规则的语言称为元语言。本书使用称为语法模板（syntax template）的元语言来描述语法规则。模板比较形象化，可以很容易地理解。例如，`main()` 函数的语法模板为：

```
int main()
{
    statement1
    .
    .
    .
    statementn
    return 0;
}
```

注意：在上面的语法模板中，阴影部分表示的定义是可选的。

在本部分的剩余部分，将会学习 C++ 程序的一些特殊符号。在后面部分讲述其他概念时，也会介绍另一些特殊符号。同样，也会介绍语法和语义规则并加以讨论。

任何语言的程序最小个体称为语言符号。C++ 的语言符号分为特殊字符、字符和标识符。

特殊符号

下面是一些特殊符号：

```
+ - * /  
. ; ? ,  
<= != == >=
```

第一行为加、减、乘、除数学符号。第二行包含从英语语法中选取的标点符号。需要指出，逗号（comma）也是一个特殊的符号。在C++中，逗号用来分开列表中的项。分号用来结束一条C++语句。需要指出，上面没有显示的空格也是一个特殊符号。可以按键盘上的空格键来创建空格符。第三行包含了由两个字符组成的记号。在两个字符之间不允许插入字符，即使空格也不可以。

字符号

另一类记号是字符号。下面列出了一些字符号：

```
int, float, double, char, void, return
```

字符号也称为保留字或关键字。组成保留字的字符总是小写字母。与特殊字符一样，每一个都认为是一个单独的符号。而且，字符号不能在任何程序中重新定义，这意味着除了固定用法之外不能用做任何定义。可以参看附录A来查看保留字。

标识符

第三类记号是标识符。标识符是程序中一些语句的名字，例如变量、常量和函数。一些标识符是预定义的，而其他一些标识符则是由用户定义的。所有的标识符必须符合C++的规则。

标识符：C++标识符包含字母、数字和下划线，而且必须用字母或下划线开头。

标识符只能由字母、数字和下划线组成，不允许使用其他的符号组成标识符。

注意：C++区分大小写字母，也就是说C++认为大写字母和小写字母是不同的。这样，标识符NUMBER和标识符number是不同的。同样标识符X和标识符x也是不同的。

在C++中，标识符可以为任意长度。然而在实际情况中，一些编译器也许不能识别出超过系统最大字符数量的标识符。（检查系统文档以获得标识符长度的限制。）例如，如果只能识别下面两个标识符的前7个字符，那么编译器就会认为它们是相同的：

```
program1  
program2
```

两个常用到的标识符是cout和cin，其中cout在产生输出时使用，而cin则用来输入数据。与保留字不同，预定义的标识符可以被重定义，但是这样做是不明智的。

例 2.2

在C++中，下面的标识符是合法的：

```
first  
conversion  
payRate  
counter1
```

表2.1显示了一些不合法的标识符并解释了其原因。

表 2.1 不合法标识符的例子

不合法标识符	描述
employee salary	在 employee 和 salary 之间不能有空格
Hello!	在标识符中使用了!号
one+two	在标识符中使用了+号
2nd	标识符不能以数字开头

注意：编译器厂商通常以下划线开始特定的标识符。当连接程序将目标程序与软件开发包提供的系统资源连接起来时，可能会产生错误。所以，建议标识符不要使用下划线作为开始。

2.2 数据类型

C++ 程序的目标是处理数据，不同程序处理不同的数据。用来计算雇员工资的程序需要对数字进行加、减、乘、除等运算，其中的一些数字可能代表工作时间和薪水。同样，设计用来对班级成员按字母顺序进行列表的程序则需要对名字进行处理。不能使用执行算术运算的语句对字母符号进行操作。而且，也不能对名字进行乘法和减法操作。为了反映这些数据的潜在不同点，C++ 把数据分成不同的类型，而且只有特定的操作才能执行特定的数据类型。虽然一开始看起来有点让人迷惑，但通过 C++ 内建的检查机制可以检查出现的类型错误。

数据类型：与一组操作相联系的一组值。

C++ 数据类型分成下面三类，如图 2.1 所示。

1. 简单数据类型
2. 复合数据类型
3. 指针

在后面几章中只讲述简单数据类型。

2.2.1 简单数据类型

简单数据类型是 C++ 中的基本数据类型，这是因为它是复合数据类型的组成元素，在第 9 章中将讲述复合数据类型。有三类简单数据类型：

1. **整型** 处理整数或没有小数部分的数值数据类型
2. **浮点型** 用来处理小数的数据类型
3. **枚举类型 (Enumeration)** 用户自定义的一种数据类型

图 2.2 显示了这三种数据类型。

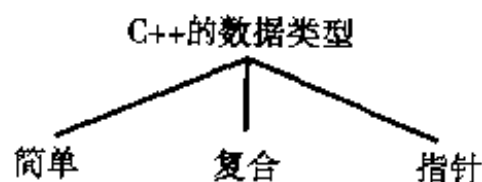


图 2.1 C++ 数据类型

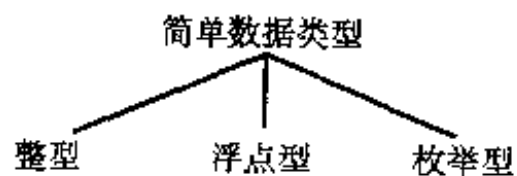


图 2.2 简单数据类型

注意：枚举类型是 C++ 允许程序员创建自定义简单数据类型的一种方法，这种数据类型将在第 8 章进行讨论。

整型数据类型进一步分成 9 类，如图 2.3 所示。

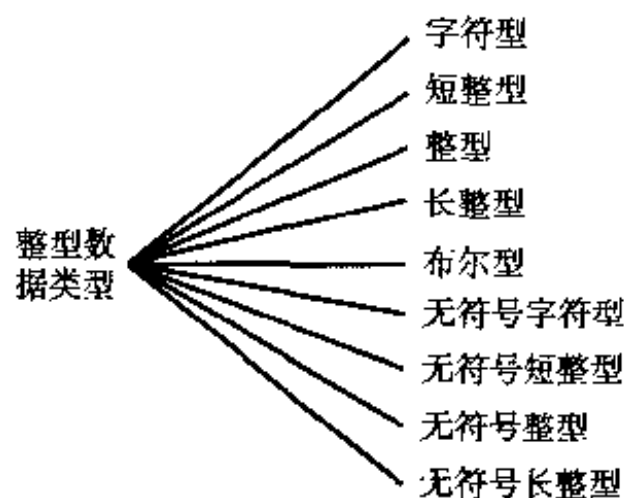


图 2.3 整型数据类型

为什么相同的数据类型分为这么多的种类？这是因为每一种数据类型有与之对应的、不同的数据集。例如，char 数据类型用来表示 -128 到 127 之间的整数，int 数据类型用来表示 -2147483648 到 2147483647 之间的整数，而数据类型 short 表示 -32768 到 32767 之间的整数。

使用哪种数据类型取决于程序中所使用数值的大小。在早期编程中，计算机和主存的价格非常昂贵，只有一小部分是用来执行程序和对数据进行操作的。所以，程序员不得不对存储器的使用进行优化。因为编写程序以及让程序运行已经是一个非常复杂的过程，其中可能没有考虑一些小问题所占用的存储器大小，所以，为了能有效地使用存储器，程序员可以查看程序中的数据类型并考虑使用哪种数据类型。

现代编程语言只有 5 类简单数据类型：integer, real, char, bool 和枚举（enumeration）类型。在本书中使用的整型数据类型是 int, bool 和 char。

表 2.2 给出了这三种数据类型的取值范围以及存储这些值所占用的内存大小。

表 2.2 三种简单类型的值和内存分配大小

数据类型	值	占用内存大小
int	-2147483648~2147483647	4
bool	true, false	1
char	-128~127	1

注意：表 2.2 仅供参考。不同的编译器可能允许不同范围的值，用户可以参阅所使用的编译器手册。为了找到特定系统中准确的整型数据类型大小，可以运行附录 G 所给出的程序。而且，为了找到这些数据类型的最大值和最小值，可以运行附录 F（头文件 climits）给出的另一个程序。

int 数据类型

本部分讲述 int 数据类型。实际上，这些讨论也能应用到其他整型数据类型。

在 C++ 中，整数就像数学中的整数一样，如下面的一些数字：

-6728, -67, 0, 78, 36782, 763

从上面的例子中可以看出如下两种规则：

1. 正整数不必在前面加 + 号。
2. 在一个整数中没有逗号。在 C++ 中，逗号是用来分开列表中的项。所以 36,782 会认为是两个整数：36 和 782。

布尔数据类型

布尔数据类型只有两个值：true 和 false。true 和 false 也称为逻辑（布尔）值。该数据类型的主要目的是对逻辑表达式进行操作。返回值为 true 或 false 的表达式称为逻辑表达式。在第 4 章中将对

逻辑 (Boolean) 表达式进行探讨。在 C++ 中, bool, true 和 false 是保留字。

注意: 数据类型 bool 是 C++ 最新的特征之一。为了完全保证可以使用, 需要检查相应的系统文件, 了解这种数据类型在系统中是否可用。

char 数据类型

char 数据类型是最小的整型数据类型。除了处理较小的数字 (-128 ~ 127), char 数据类型还用来代表字符, 也就是字母、数字和特殊字符。这样, char 数据类型可以代表键盘的每一个键。当使用 char 数据类型时, 需要在每个字符两边用单引号括起来。下面是使用 char 数据类型值的一些例子:

'A', 'a', '0', '*', '+', '\$', '&'

需要注意空格也是一个字符, 它被写为 ' ', 在两个单引号之间是一个空格符。

char 数据类型只允许单引号间有一个字符, 所以 'abc' 不属于 char 数据类型。而且即使 '!=' 和类似的特殊符号被认为是一个符号, 也不能认为它们是 char 数据类型的一个可能值。所有位于键盘上的可打印字符可以被认为是 char 数据类型的一个可能值。

目前有几种不同的字符数据集被使用。最常用的是美国标准信息交换码 (ASCII) 和扩展 BCD 交换码 (EBCDIC)。ASCII 字符集包含 128 个值, 而 EBCDIC 包含 256 个值, 它是由 IBM 提出的。在附录 C 中描述了这两种字符集。

ASCII 字符集中 128 个字符的每一个值代表不同的字符。例如, 值 65 代表 'A', 值 43 代表 '+'. 这样, 该集中的每一个字符都有一个预定义的顺序, 这称为对照序列。在进行字符比较时就会用到对照序列。例如, 值 'B' 代表 66, 所以 'A' 比 'B' 小。同样, '+' 比 'A' 小, 因为 43 小于 65。

注意: ASCII 字符集的第 14 个字符称为换行符, 用 '\n' 来表示。(注意在 ASCII 字符集中换行符的位置是 13, 这是因为第一个字符的位置是 0。)尽管换行符是由两个字符组成的, 但是只被当做一个字符。同样, 在 C++ 中, 制表符用 '\t' 表示而 null 用 '\0' 来表示。而且, ASCII 字符集中的前 32 个字符是不可打印的。(附录 C 对这些字符进行了描述。)

2.2.2 浮点数据类型

为了处理带小数的数字, C++ 提供了浮点数据类型, 在本节中将其进行讨论。为了方便讨论, 现在回顾一下高中或大学代数课中的概念。

读者可能比较熟悉科学计数法。例如:

43872918=4.3872918 * 10 ⁷	{ 10 的 7 次方 }
.0000265=2.65 * 10 ⁽⁻⁵⁾	{ 10 的负 5 次方 }
47.9832=4.7983 * 10 ¹	{ 10 的 1 次方 }

为了表示实数, C++ 使用了一种科学计数法, 称为浮点型计数法。表 2.3 显示了 C++ 使用某台机器并采用浮点型计数法打印出的实数。在 C++ 浮点计数法中, 字母 E 代表指数。

表 2.3 用 C++ 浮点计数法打印实数的例子

实数	C++ 浮点计数法
75.924	7.592400E1
0.18	1.800000E-1
0.0000453	4.530000E-5
-1.482	-1.482000E0
7800.0	7.800000E3

C++ 提供了三种数据类型对小数进行操作: float, double 和 long double。与整型数据类型的情况一样, 数据类型 float, double 和 long double 有不同的值集合。图 2.4 定义了这些数据类型。

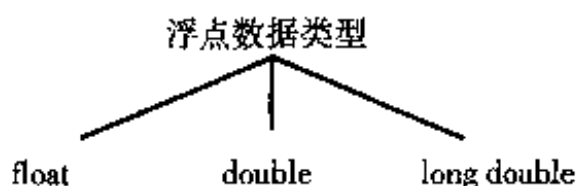


图 2.4 浮点数据类型

注意: 在最新的编译器中, 数据类型 double 和 long double 是相同的。所以本书只讲述 float 和 double 类型。

float: 在 C++ 中, float 数据类型代表在 $-3.4E+38$ 到 $3.4E+38$ 之间的任何实数。存储器分配给 float 数据类型 4 个字节。

double: C++ 中, double 数据类型代表在 $-1.7E+308$ 到 $1.7E+308$ 之间的任何实数。存储器分配给 double 数据类型 8 个字节。

float 和 double 数据类型的最大值和最小值依赖于系统。为了找到特定系统中的这些值, 可以查阅编译器的文档, 也可以运行在附录 F 中所给出的程序 (头文件 cfloat)。

除了值的范围不同外, float 和 double 数据类型还有显著的不同, 就是有效数字的最大位数, 也就是十进制的位数。float 型的最大位数为 6 或 7, 而 double 型有效数字的最大位数为 15。最大有效数字的位数称为精度。有时称 float 值为单精度, 把 double 类型的值称为双精度。如果要处理十进制数字, 大部分情况下只需要单精度类型; 如果需要比 6 位或 7 位有效数字更准确, 则可以使用 double 类型。

注意: 在程序中使用 float 数据类型来处理浮点型数字时, 一些特定的编译器可能给出一个警告信息, 例如“把双精度截短为单精度”。为了避免在编程例子中出现这样的警告信息, 本书主要使用 double 数据类型来处理浮点型数字。

在讨论完数据类型之前, 下面再讨论一种数据类型——string。

2.2.3 string 型

string 数据类型是一种程序员定义的数据类型。在程序中它不像前面介绍的简单数据类型那样是直接可用的。为了使用这种数据类型, 需要从库中访问程序文件, 在本章后面将对库进行讨论。string 数据类型是 ANSI/ISO 的一个新特征。

注意: 在 ANSI/ISO C++ 语言标准之前, 标准 C++ 库没有提供 string 数据类型。编译器开发厂家经常提供他们自己的程序员定义的 string 类型, 不同厂家的 string 操作语法和语义也不相同。

string 是一个字符序列或者不含字符。在 C++ 中, 字符串用双引号括起来。不包含字符的字符串称为 null 或空字符串。下面是字符串的例子。注意 "" 是空字符串。

```
"William Jacob"
"Mickey "
""
```

字符串中的每一个字符在字符串中有一个相对的位置。第一个字符的位置为 0, 第二个字符的位置为 1, 依次类推。字符串的长度是字符串中字符的个数。

例 2.3

字符串	字符串中字符的位置	字符串的长度
"William Jacob"	字符 'W' 的位置: 0 字符 'J' 首次出现的位置: 1 空格的位置: 7 字符 'J' 的位置: 8 字符 'b' 的位置: 12	13
"Mickey"	字符 'M' 的位置: 0 字符 'i' 的位置: 1 字符 'c' 的位置: 2 字符 'k' 的位置: 3 字符 'e' 的位置: 4 字符 'y' 的位置: 5	6

当判断字符串的长度时，字符串中的空格也必须计算在内。例如，下面字符串的长度是 22。

"It is a beautiful day."

注意：string 数据类型在标准 C++ 中是不可用的。

2.3 算术运算符和运算符优先级

计算机最重要的一个用途是它的计算能力。可以使用标准算术运算符对整型和浮点数据类型进行操作。C++ 中包含 5 种算术运算符。

1. + 加
2. - 减
3. * 乘
4. / 除
5. % 求余

无论整型还是浮点型数据类型都可以使用 +、-、*、/ 运算符。而 % 运算符只能应用于整型数据类型。对整型数据类型可以使用 / 运算符，其结果是除法的商，即整数除法截断任何小数部分并且没有四舍五入。

读者已经熟悉使用下列数学表达式：

```
3+4
2+3*5
5.6+6.2*3
x+2*5+6/y
```

一般说来，一个数学表达式是由数学运算符和数字组成的。这些出现在表达式中的数字称为操作数。此外，用来对运算符求值的数字称为该运算符的操作数。

在表达式 "3 + 4" 中，3 和 4 是运算符 + 的操作数。所以，当计算运算符 + 的值时，表达式 "3 + 4" 的结果是 7。需要注意在表达式 "3 + 4" 中，运算符 + 有两个操作数。有两个操作数的运算符称为二元运算符。

只有一个操作数的运算符称为一元运算符。在表达式 -5 中 "-" 只有一个操作数，所以 "-" 称为一元运算符。前面讲过在一个正数前面不必加 + 号，但在表达式 +27 中，+ 是一个一元运算符。

一元运算符：只有一个操作数的运算符。

二元运算符：有两个操作数的运算符。

在下面的表达式中，+ 和 - 都是二元运算符。这样，- 和 + 既是一元运算符，也是二元算术运算符。然而，由于算术运算符 *、/ 和 % 都是二元运算符，所以必须有两个操作数。

3+4
23-45

下面的例子显示了算术运算符，尤其是 / 和 % 是怎样对整型数据类型进行操作的。从这些例子中可以看出，当与整型数据类型结合使用时，运算符 / 表示除法的商。

例 2.4

算术表达式	结果	描述
2+5	7	
13+89	102	
34-20	14	
45-90	-45	
2*7	14	
5/2	2	除法 5/2 的商是 2，余数为 1。所以，5/2 的值是整型操作数的商，即 2。
14/7	2	
34%5	4	在除法 34/5 中，商是 6，余数是 4。所以 34%5 的值是余数 4。
-34%5	-4	在除法 -34/5 中，商是 -6，余数是 -4。所以 -34%5 的值是余数 -4。
34%-5	4	在除法 34/-5 中，商是 -6，余数是 4。所以 34%-5 的值是余数 4。
-34%-5	-4	在除法 -34/-5 中，商是 6，余数是 -4。所以 -34%-5 的值是余数 -4。
4%6	4	在除法 4/6 中，商是 0，余数是 4。所以 4%6 的值是余数 4。

需要指出，在除法 34/5 和 -34/-5 中，商同为 6，但是余数是不同的。在除法 34/5 中，余数是 4；而在在除法 -34/-5 中，余数是 -4。

下面的表达式显示了算术运算符是怎样对浮点数进行操作的。

例 2.5

表达式	结果
5.0+3.5	8.5
3.0+9.4	12.4
16.3-5.2	11.1
4.2*2.5	10.50
5.0/2.0	2.5

2.3.1 优先级顺序

当表达式中有不止一个算术运算符时，C++ 使用运算符优先级规则来计算一个表达式的值。算术运算符 *、/、% 的优先级高于 +、- 的优先级。* 和 /、% 的优先级相同，同样 + 和 - 的优先级也相同。

当运算符的优先级相同时，操作从左至右执行。为了避免冲突，可以使用括号来对算术表达式进行分组。

例如，使用优先级规则时， $3*7-6+2*5/4+6$ 与下面的表达式等价：

```

(3*7) - 6 + ((2*5)/4) + 6
=21-6 + (10/4) + 6      (求*的值)
=21-6+2+6              (求/的值)
=15+2+6                (求-的值)
=17+6                  (求第一个+的值)
=23                    (求+的值)

```

在第二个例子中,使用括号确定了优先级顺序。也可以使用括号来改写优先级规则(例如例2.6)

例 2.6

在表达式 $3+4*5$ 中, $*$ 在 $+$ 之前被计算。所以,表达式的结果是 23。而在表达式 $(3+4)*5$ 中, $+$ 在 $*$ 之前被计算,其结果是 35。

因为算术运算符从左至右进行计算,除非使用括号,否则算术运算符的结合方向默认为从左至右。

注意: (字符运算) 因为 `char` 数据类型也是整型数据类型,所以 C++ 允许对 `char` 数据进行算术运算。应该慎重地使用这种功能。字符 `'8'` 和整数 8 是有差别的。8 的整数值是 8,而字符 `'8'` 在 ASCII 对照序列中的整数值是 56。所以计算算术表达式时, $8+7=15$, $'8'+7'=56+55$, 结果为 111, $'8'+7=56+7$, 结果是 63。而且,因为 $'8'*7'=56*55=3080$,而 ASCII 字符集只有 128 个值,所以 $'8'*7'$ 在 ASCII 字符数据集中是没有定义的。

这些例子说明当执行字符运算操作时可能产生很多错误。如果必须应用它们,那么应该慎重使用 `char` 数据类型。

2.4 表达式

前面只讨论了算术运算符,本节将详细地讨论算术表达式。

如果表达式中所有的操作数(即数字)为整数,则该表达式称为整型表达式。如果表达式中全部是浮点型数字,则表达式称为浮点型或小数表达式。整型表达式的结果为整数,浮点型表达式会得到一个浮点型的结果。下面的例子将会帮助读者明确这些定义。

例 2.7

考虑下面的 C++ 整型表达式:

```
2+3*5
3+x-y/7
x+2*(y-z)+18
```

在表达式中, x 、 y 和 z 表示整型变量,也就是说它们能保存整数值。

例 2.8

考虑下面的 C++ 浮点型表达式:

```
12.8*17.5-34.50
x*10.5+y-16.2
```

其中 x 和 y 表示浮点型变量,即它们能保存浮点型值。在本章后面部分将会讨论变量。

整型或浮点型表达式的值是可以直接计算的。前面讲过,当运算符具有相同优先级时,表达式是从左至右计算的。可以一直使用括号来对操作数和运算符进行分组以避免出现混乱。

2.4.1 混合表达式

具有不同数据类型操作数的表达式称为混合表达式。混合表达式既包含整型也包含浮点型数字。下面是混合表达式的例子:

```
2+3.5
6/4+3.9
5.4*2-13.6+18/2
```

在第一个表达式中,运算符+有一个整型操作数、一个浮点型操作数。在第二个表达式中,运算符/的两个操作数是整型的,运算符+的第一个操作数是6/4的结果,第二个操作数是浮点型数字。第三个例子是一个更复杂的整型和浮点型相结合的混合表达式。一个直接的问题是: C++ 怎样计算混合表达式的值?

当计算混合表达式的值时有两条规则:

1. 当计算混合表达式一个运算符的值时:

- a. 如果运算符有相同类型的操作数(即同为整型或浮点型数字),运算符根据操作数的类型来计算值。这样整型操作数的结果是整数,而浮点型数字的结果是浮点型的。
- b. 如果运算符包含两种类型(即一个整型数字,另一个是浮点型数字),在计算期间,整数小数部分补0以转换为浮点型数字。最后结果是浮点型的。

2. 整个表达式根据优先级规则来计算结果。乘、除和求余运算符优先于加和减运算符。相同优先级的运算符从左至右计算。可以对运算符分组以避免混乱。

从这些规则中可以看出,当计算一个混合表达式的值时,可以使用优先级规则每次计算一个运算符。如果运算符有相同数据类型的操作数,则使用规则1(a)来计算值。即具有整型操作数的运算符会产生一个整型的操作数,而具有浮点型操作数的运算符会产生浮点型的结果。如果被计算的运算符有一个是整型操作数,另一个是浮点型操作数,那么在计算运算符的值之前,会先把整型操作数转换为浮点型,即在整数的小数点后加一组零。下面的例子说明了怎样计算混合表达式的值:

例 2.9

混合表达式	值	应用规则
3/2+5.0	=1+5.0 =6.0	3/2=1 (整数除法;规则 1(a)) (1+5.0=1.0+5.0 (规则 1(b)) =6.0)
15.6/2+5	=7.8+5 =12.8	15.6/2=15.6/2.0 (规则 1(b)) =7.8 7.8+5=7.8+5.0 (规则 1(b)) =12.8
4*3+7/5-25.6	=12+7/5-25.6 =12+1-25.6 =13-25.6 =-12.6	4*3=12; (规则 1(a)) 7/5=1 (整数除法; 规则 1(a)) 12+1=13 (规则 1(a)) 13-25.6=13.0-25.6 (规则 1(b)) =-12.6

这些例子表明,除非运算符有一个整数和一个浮点操作数,否则整数是不会被转化为浮点型数字的。

现在看一下下面的例子。这些表达式的值是从一个C++程序中获得的。在这些表达式中, x=15, y=23, z=3.75。

表达式	值
x+y/z	21.1333
7/2+7.45	10.45
15/2+4*5-3.50	23.5
7/2*3.5+6	16.5
7.0/2+7.45	10.95

2.4.2 类型转换 (强制转型)

在前面的部分中,学习了如何计算算术表达式的值,如果运算符包含混合的操作数,那么整数会转换为小数点后带有零的浮点数。当一种数据类型自动地转换为另一种数据类型时,就会产生隐式类型转换 (implicit type coercion)。正如前面的例子所看到的,如果不注意数据类型,那么隐式

类型转换可能不会产生期望值。

为了避免隐式类型转换, C++ 通过转型运算符提供了显式类型转换, 转型运算符也称为类型转换或类型强制转换并采取下面的形式:

```
static_cast<dataTypeName> (expression)
```

expression 先被计算, 然后其值被转化为由 dataTypeName 指定的数据类型。在 C++ 中, static_cast 是一个保留字。

当使用转型运算符把一个浮点数转换为一个整数时, 只是简单地把浮点数的小数部分去掉, 也就是说, 把浮点数字截断。下面的例子显示了转型运算符是如何起作用的。注意, 读者应该理解最后两个表达式的计算结果。

例 2.10

表达式	值
static_cast<int>(7.9)	7
static_cast<int>(3.3)	3
static_cast<double>(25)	25.0
static_cast<double>(5+3)	= static_cast<double>(8) = 8.0
static_cast<double>(15/2)	= 15.0/2 (static_cast<double>(15) = 15.0) = 15.0/2.0 = 7.5
static_cast<double>(15/2)	= static_cast<double>(7) (15/2 = 7) = 7.0
static_cast<int>(7.8 + static_cast<double>(15/2))	= static_cast<int>(7.8 + 7.5) = static_cast<int>(15.3) = 15
static_cast<int>(7.8 + static_cast<double>(15/2))	= static_cast<int>(7.8 + 7.0) = static_cast<int>(14.8) = 14

考虑另外的一系列例子, 例子中 $x=15$, $y=23$, $z=3.75$ 。对于 C++ 来说, 下面表达式的结果如下:

表达式	值
static_cast<int>(7.9 + 6.7)	14
static_cast<int>(7.9) + static_cast<int>(6.7)	13
static_cast<double>(y / x) + z	4.75
static_cast<double>(y) / x + z	5.28333

注意: 在 C++ 中, 转型运算符也能采用 dataType (表达式) 的形式。该形式称为类 C 转型。然而, static_cast 比类 C 转型更稳定。

也可以使用转型运算符明确地将 char 数据值转换为 int 数据值, 将 int 数据值转换为 char 数据值。为了将 char 数据值转换为 int 数据值, 可以使用单引号。例如, 在 ASCII 字符集中, static_cast<int>('A') 是 65, static_cast<int>('8') 是 56。同样 static_cast<char>(65) 是 'A', static_cast<char>(56) 是 '8'。

在本章前面的部分, 我们学习了在 C++ 中怎样形成一个算术表达式并计算它的值。如果想在另一个表达式中使用该表达式的值, 必须先保留该表达式的值。有很多原因需要保存该表达式的值。一些表达式很复杂, 可能需要非常多的计算时间来求值。通过计算并保存值以进一步使用它们, 不仅可以节省计算时间和创建一个执行更快的程序, 而且可以避免可能出现的打字错误。在 C++ 中,

表达式求值之后如果不被保存,那么它就会丢失。也就是说,除非表达式的值被保存,否则它不能在后面的计算中使用。在下一部分中,将学习怎样保存表达式的值以及怎样在后面的计算中使用它们。

2.5 输入

前面讲过,C++的主要目标是执行计算并处理数据。回想一下,数据在操作之前必须被装入主存。在本节中,将会学习如何把数据输入到计算机的内存中。在计算机的内存中装载数据包含两个步骤:

1. 命令计算机分配内存。
2. 在程序中编写语句把数据置入分配的内存中。

2.5.1 使用常量和变量来分配内存

当命令计算机分配内存时,不仅要告诉计算机每一个内存位置使用什么名字,还要明确存储的数据类型。知道数据的位置是很重要的,因为存储在内存位置中的数据可能要在程序中的几个地方用到。前面看到,数据类型对于执行准确的计算是非常重要的,而且在整个程序执行过程中数据应该保持固定还是可以改变也是非常重要的。

有一些数据是特殊的。例如,单位工资通常对所有的兼职雇员是相同的。将英寸转换为厘米的转换方程式是固定的,因为1英寸总是等于2.54厘米。当存储在内存中时,这种类型的数据应该被保护起来,以避免在程序执行过程中出现意外的改变。在C++中,可以使用常量命令程序为那些在程序执行过程中数据保持不变的内存位置做出标识。

常量: 在程序执行过程中内容保持不变的内存位置。

一些特定的数据在程序执行过程中需要改变。例如,在每次测试以后,平均分数和测试的次数发生了改变。同样,在每次工资增长之后,雇员的薪水发生了改变。这种数据类型必须被存储在程序执行期间内容可以改变的内存单元中。在C++中,程序执行期间内容可以改变的内存单元称为变量。更正式地说:

变量: 程序执行期间内容可以改变的内存位置。

为了分配内存,可以使用C++声明语句。下面的语法声明一个常量。

```
const dataType identifier = value;
```

在C++中,const是一个保留字。

例 2.11

考虑下面的C++语句:

```
const double conversion=2.54;  
const int noOfstudents=20;  
const char blank='';  
const double payRate=15.75;
```

第一条语句告诉编译器去分配相应内存来存储double类型的值,内存空间称为conversion并在内存中存储2.54。使用该语句的整个程序中,无论什么时候需要该变量,内存空间conversion都可以被访问。其他语句的含义也是类似的。

使用常量来存储固定数据,而不是使用数据值本身有一个主要的好处。当固定值改变时,不需要编辑整个程序把旧值改变为新值。相反,只需要在一处改变该值,然后使用新值重新编译、执行

程序。另外,通过存储一个值并在需要使用该值时引用其内存位置,可以避免一次次敲入相同的值并且防止出现意外的拼写错误。如果拼错了该位置的名字,计算机将会通过一条错误信息来警告用户,但是如果值被拼错则不会发出警告信息。

声明一个或多个变量的语法:

```
dataType identifier, identifier, ...;
```

例 2.12

考虑下面的语句:

```
double amountDue;  
int counter;  
char ch;  
int x, y;  
string name;
```

第1条语句告诉编译器分配足够的内存来存储一个double数据类型的值,称为amountDue。语句2和语句3有相同的标准。第4条语句告诉编译器分配两个不同的内存空间存储int数据类型的值,分别命名内存空间为x, y。第5条语句告诉编译器分配内存空间来存储一个字符串,其名字为name。

从现在起,当说“变量”时,是指一个变量内存的位置。

注意:在C++中,必须在使用标识符之前声明所有的标识符。如果使用一个没有被声明的标识符,编译器会产生一条错误信息表明标识符没有被声明。所以,为了使用一个常量或变量,首先必须对其进行声明。

注意:既然数据类型、变量和常量已经被定义和讨论过,那么提供一个简单数据类型的定义是可能的。如果变量或常量数据类型一次只能存储一个值,那么该数据类型称为简单数据类型。例如,如果x是一个int型变量,那么x每次只能存储一个值。

2.5.2 向变量中赋值

现在知道了怎样去声明一个变量,下一个问题是:怎样向这些变量中放入数据呢?在C++中,可以使用两种方法向变量中放入数据:

1. 使用C++的赋值语句。
2. 使用输入(读)语句。

赋值语句

赋值语句采用下面的形式:

```
variable=expression;
```

在赋值语句中,表达式的值应该与变量的数据类型相匹配。右边的表达式被计算之后赋给左边的变量(内存位置)。

如果变量中第一次赋值,那么称变量被初始化。

在C++中,=称为赋值运算符。

例 2.13

假定有下面的变量声明:

```
int i, j;
```

```
double sale;  
char first;  
string str;
```

现在,考虑下面的赋值语句:

```
i=4  
j=4*5-11;  
sale=0.02*1000;  
first='D';  
str="it is a sunny day.";
```

对于每一条语句,计算机首先计算右边表达式的值,然后把这些值存储到以左边变量命名的内存位置中。第1条语句把值4存储到*i*中,第2条语句把值9存储到*j*中,第3条语句把值20.00存储到*sale*中,第4条语句把字符'D'存储到*first*中,第5条语句把字符串"it is a sunny day."存储到变量*str*中。

下面的C++语句:

```
i=i+2;
```

表示“无论*i*为何值,将*i*加2,然后再将新值赋给内存位置*i*。”右边的表达式必须先被计算,然后再把计算的值赋给左边变量指定的内存位置。这样,下列C++语句:

```
i=6;  
i=i+2;
```

和语句:

```
i=8;
```

都是将8赋值给*i*。另外,如果*i*没有被初始化,那么语句“*i*=*i*+2;”是没有意义的。

对于赋值语句“*i*=5;”,其意义就像“*i*成为5”或“*i*得到5”。每次一个新值赋给*i*时,旧值就会被擦除。

假定*i*, *j*和*k*是int型变量。下列语句:

```
i=12;  
i=i+6;  
j=i;  
k=j/2;  
k=k/3;
```

结果是在*k*中存储了值3。可以利用一个用于跟踪检查的序列来跟踪值,这对于学习和实践来说是非常有价值的。在本章后面将会学习到更多如何跟踪检查一系列C++语句的知识。

注意:假定*x*, *y*和*z*是int型变量,则下面是一条合法的C++语句。

```
x=y=z;
```

在这条语句中,*z*的值首先被赋值给*y*,然后*y*的新值被赋值给*x*。因为赋值运算符=是从右至左求值,所以赋值运算符的结合性被认为是从右至左。

在知道了如何声明变量以及如何向变量中输入数据之后,下面将学习如何保存表达式的值。然后就可以在后面的语句中使用该值,而不必使用该表达式本身,这样就回答了本章前面提出的问题。为了保存一个表达式的值并在以后使用它,需要进行下面的步骤:

1. 声明适当数据类型的变量。例如,如果表达式的结果是整数,则声明int型变量。

2. 把表达式的值赋给已声明的变量。
3. 无论什么时候需要该表达式的值，都可以使用变量中保存的值。

下面的例子进一步阐述了这一概念。

例 2.14

假定有下面的语句声明：

```
int a,b,c,d;  
int x,y;
```

计算表达式 $-b + (b^2 - 4ac)$ 和 $-b - (b^2 - 4ac)$ 的值，然后把这些表达式的值分别赋给 x 和 y 。因为表达式 $b^2 - 4ac$ 在两个表达式中都出现，所以可以先计算该表达式的值并且将值存储在 d 中，然后使用 d 的值来计算表达式的值，如下面语句所示：

```
d=b*b-4*a*c;  
x=-b+d;  
y=-b-d;
```

前面讲到：如果变量在一个表达式中使用，那么只有该变量已被初始化后表达式才能产生一个有意义的结果。而且还讲到：在声明一个变量后，可以使用赋值语句对变量进行初始化。同时初始化并声明变量是可能的，在讨论怎样使用输入（读）语句之前，先讲述一下这一重要的问题。

声明并初始化变量

当变量被声明时，C++ 不能自动地为变量放入一个值。换言之，C++ 不能自动地初始化变量。例如，`int` 和 `double` 变量并不是初始化为 0，这就像其他编程语言一样。然而这并不意味着在变量声明之后它就没有值。当变量被声明的时候，会为该变量分配内存。

第 1 章中讲过，主存是有顺序的一系列的单元，每一个单元能存储一个值。而且，机器语言是由 0 和 1 组成的序列。所以，内存单元中的数据是一系列的二进制数。通常状态下，一些位是 1，一些位是 0。这些位的状态依赖于系统是怎样工作的。然而，当命令计算机在一个内存单元中存储特定值的时候，这些位是根据存储的数据来设置的。

在处理数据期间，计算机取出特定单元中存储的值并执行计算。如果声明一个变量但并没有在变量中存储一个值，该内存单元仍然有一个值——通常是上次使用设定的值——但无法知道该值是什么。

如果仅声明一个变量而不命令计算机把数据放到变量中，那么变量的值是无意义的。然而，计算机并不警告用户，无论内存中有什么值都认为是合法的，并且使用那些内存中的值来进行计算。使用表达式中没有初始化的变量会产生错误的结果。为了避免这些错误，C++ 允许在变量声明的时候对它们进行初始化。例如，考虑下面的 C++ 语句，在这些语句中变量首先被声明然后初始化：

```
int first,second;  
char ch;  
double x,y;  
  
first=13;  
second =10;  
ch='';  
x=12.6;  
y=123.456;
```

可以使用下面的 C++ 语句同时声明并初始化变量：

```
int first=13, second=10;
char ch='';
double x=12.6, y=123.456;
```

第一条 C++ 语句声明了两个变量: first 和 second, 然后在 first 中存储 13, 在 second 中存储 10。

注意: 实际上, 并不是所有的变量都需要在声明期间被初始化。应该由程序的特性或者程序员的选择来决定哪一个变量在声明期间需要被初始化。但是, 所有的变量在使用之前必须被初始化。

输入 (读) 语句

在前面的部分中学习了怎样使用赋值语句将数据置入到变量中。在本部分中, 将学习怎样从标准输入设备中使用 C++ 的输入 (或读) 语句将数据置入变量中。

注意: 在大多数情况下, 标准输入设备是键盘。

当计算机从键盘中得到数据时, 称用户正在和计算机进行交互。

从标准输入设备中把数据放到变量中是通过使用 cin 和运算符 >> 来完成的。cin 和 >> 的语法是:

```
cin>>variable>>variable...;
```

这称为输入语句, 有时也称为 cin 语句。在 C++ 中, >> 称为流析取运算符。

输入或 (cin) 语句按如下的过程来工作。假定 miles 是 double 数据类型的变量, 语句:

```
cin>>miles;
```

会让计算机从标准输入设备中得到一个 double 数据类型的值, 然后把它放到名字为 miles 的内存单元中。

通过 cin 语句并使用多于一个的变量, 则可以一次读入多个值。假定 feet 和 inch 是 int 数据类型的变量, 下面的语句:

```
cin>>feet>>inch;
```

从键盘得到两个整数并且把它们放到内存位置分别为 feet 和 inch 的内存单元中。

注意: 程序执行期间, 如果要在同一行输入多个值, 那么这些值必须被至少一个空格或 Tab 键分隔开。也可以一行输入一个值。

记住, 有两种方法初始化变量: 通过使用赋值语句和输入语句。考虑下面的声明:

```
int feet;
```

可以使用赋值语句初始化变量为 35:

```
feet=35;
```

也可以执行下面的语句在程序执行期间输入:

```
cin>>feet;
```

如果使用赋值语句初始化 feet, 那么每次程序运行的时候都是使用相同的值, 除非编辑源代码、修改值、重新编译和运行程序。每次程序运行时通过使用输入语句, 可以输入一个值, 该值被存储到 feet 中, 所以, 输入语句比赋值语句更具灵活性。

有时通过使用赋值语句来初始化变量是很必要的, 尤其是如果变量只用于内部计算而不用于输入并存储数据的时候。

注意：回忆一下，C++ 在声明变量时并不能自动初始化变量。一些变量在声明时可以被初始化，而一些无论使用赋值语句还是输入语句都需要被初始化。

注意：假定想使用输入语句向 char 变量存储一个字符。在程序执行期间，当输入字符时，不能使用单引号。例如，假定 ch 是变量。考虑下面的输入语句：

```
cin>>ch;
```

如果想使用该语句向 ch 中存储 K，那么在程序执行期间，只能输入 K。同样，如果想使用一条输入语句向一个 string 类型的变量存储一个字符串，那么在程序执行期间，只能输入没有双引号的字符串。

例 2.15

下面的例子进一步展示了赋值语句和输入语句如何对变量进行操作。考虑下面的声明：

```
int one,two;
double z;
char ch;
string name;
```

而且假定下面的语句按照给定的顺序执行。

```
1. one=4;
2. two=2*one+6;
3. z=(one+1)/2.0;
4. ch='A';
5. cin>>two;
6. cin>>z;
7. one=2*two+static_cast<int>(z);
8. cin>>name;
9. two=two+1;
10. cin>>ch;
11. one=one+static_cast<int>(ch);
12. z=one-z;
```

另外假定输入是：

```
8 16.3 Goofy G
```

该行有 4 个值：每一个值都使用一个空格与其他值分开。

现在确定在最后一语句执行后所声明变量的值。为了明确地显示一条特定的语句如何改变一个变量的值，在每一条语句执行后变量的值被显示出来。（在图 2.5 到图 2.17 中，框中的问号表示数据值是未知的。）

在语句 1 执行之前，所有的变量都没有初始化，如图 2.5 所示。

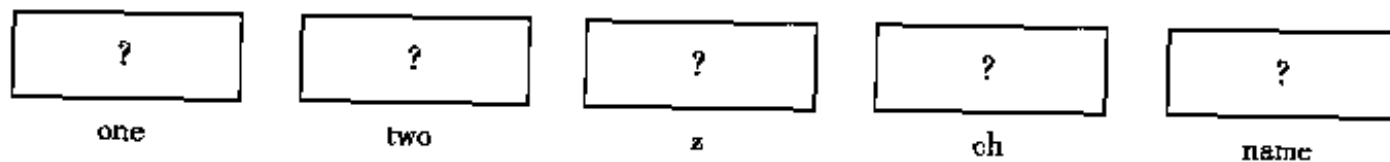


图 2.5 语句 1 执行前的变量

语句 1 将 4 存储到名字为 one 的变量中。在语句 1 执行后，各变量值如图 2.6 所示。

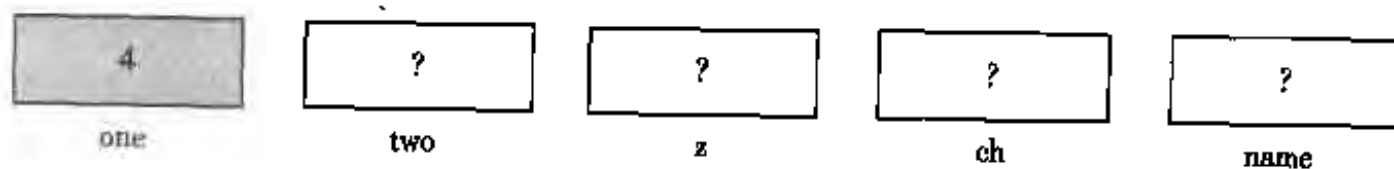


图 2.6 语句 1 执行后 one、two、z、ch 和 name 的值

语句2首先计算表达式 $2*one+6$ (值为14) 的值, 然后向 `two` 中存储表达式的值。在语句2执行后, 变量的值如图2.7所示。

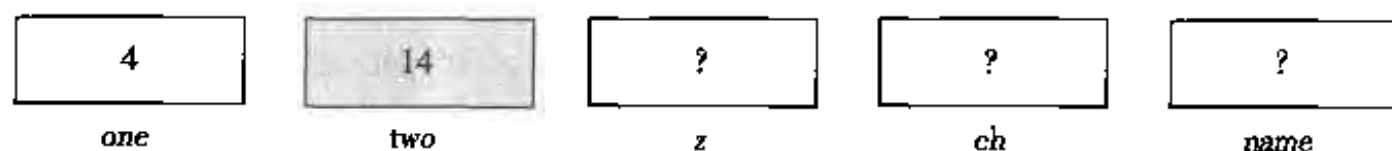


图2.7 语句2执行后 `one`、`two`、`z`、`ch` 和 `name` 的值

语句3首先计算表达式 $(one+1)/2.0$ (值为2.5) 的值, 然后将表达式的值存储到 `z` 中。在语句3执行后, 变量的值如图2.8所示。



图2.8 语句3执行后 `one`、`two`、`z`、`ch` 和 `name` 的值

语句4把字符 'A' 存储到 `ch` 中。在语句4执行后, 变量的值如图2.9所示。

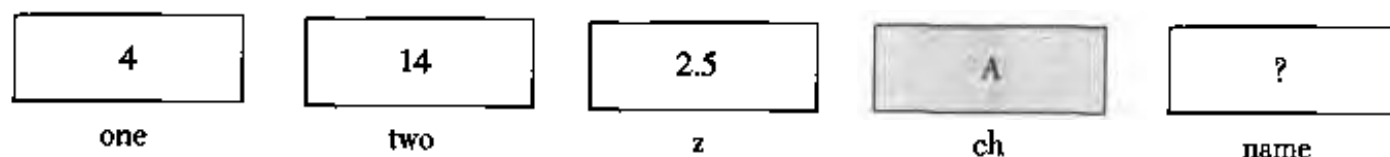


图2.9 语句4执行后 `one`、`two`、`z`、`ch` 和 `name` 的值

语句5从键盘读一个数字(8)并且把数字8存储到 `two` 中。该语句用这一新值替换了原来的旧值。在语句5执行后, 变量的值如图2.10所示。

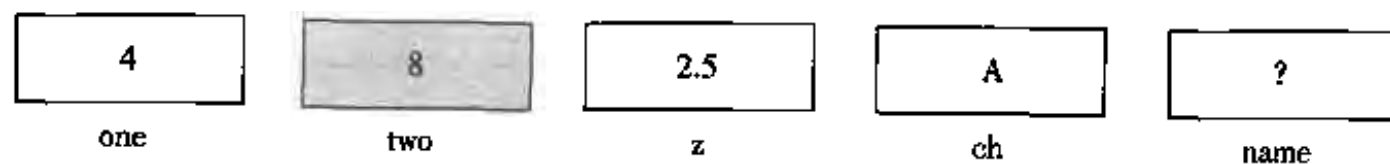


图2.10 语句5执行后 `one`、`two`、`z`、`ch` 和 `name` 的值

语句6从键盘读一个数字(16.3)并且把数字存储到 `z` 中。该语句用这一新值替换了原来的旧值。在语句6执行后, 变量的值如图2.11所示。



图2.11 语句6执行后 `one`、`two`、`z`、`ch` 和 `name` 的值

注意变量 `name` 仍没有定义。语句7首先计算表达式 $2*two+static\_cast<int>(z)$ (值为32) 的值, 然后将表达式的值存储到 `one` 中。该语句用这一新值替换了原来的旧值。在语句7执行后, 变量的值如图2.12所示。

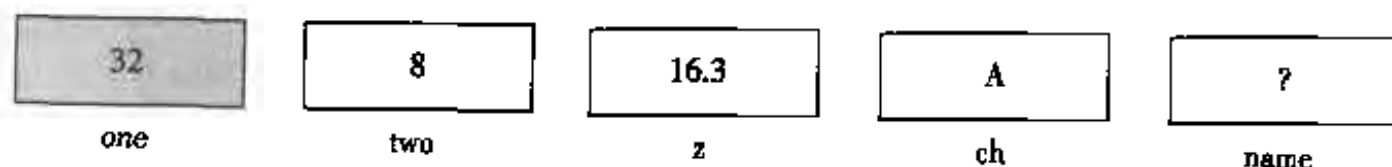


图2.12 语句7执行后 `one`、`two`、`z`、`ch` 的值

语句8从键盘得到下一个输入(Goofy)并且把它存储到 `name` 中。在语句8执行后, 变量的值如图2.13所示。

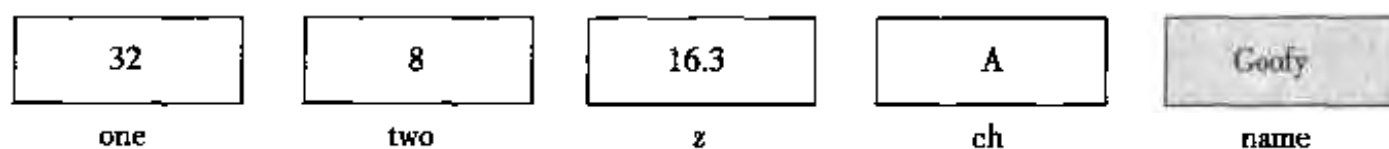


图 2.13 语句 8 执行后 one、two、z、ch 和 name 的值

语句 9 首先计算表达式 $two+1$ (值为 9) 的值,然后将表达式的值存储到 two 中。该语句通过对旧值加 1 更新了 two 的值。在语句 9 执行后,变量的值如图 2.14 所示。

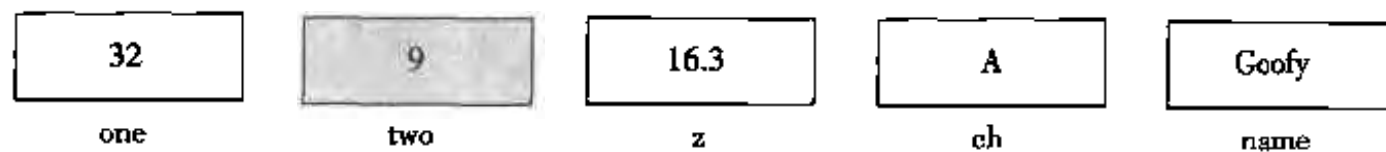


图 2.14 语句 9 执行后 one、two、z、ch 和 name 的值

语句 10 从键盘读下一个输入 (D) 并且把它存储到 ch 中。该语句用这一新值替换了原来的旧值。在语句 10 执行后,变量的值如图 2.15 所示。

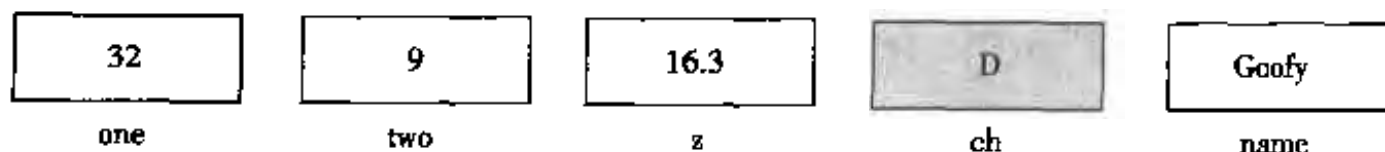


图 2.15 语句 10 执行后 one、two、z、ch 和 name 的值

语句 11 首先计算表达式 $one+static\_cast<int>(ch)$,也就是 $32+static\_cast<int>('D')$,其值等于 $32+68=100$ 。这里 $static\_cast<int>('D')$ 给出了 ASCII 字符数据集中字符 'D' 的对照结果,即 68。然后语句 11 将表达式的值存储到 one 中。该语句用这一新值替换了原来的旧值。在语句 11 执行后,变量的值如图 2.16 所示。

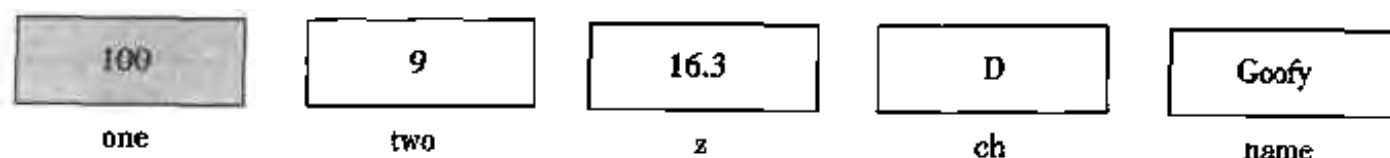


图 2.16 语句 11 执行后 one、two、z、ch 和 name 的值

最后,语句 12 首先计算表达式 $one-z$ (值等于 $100-16.3=83.7$) 的值,然后将表达式的值存储到 z 中。在最后一行语句执行后,变量的值如图 2.17 所示。



图 2.17 语句 12 执行后 one、two、z、ch 和 name 的值

注意: 当程序出现错误并且所产生的结果并不是所期望的值时,可以对变量赋值语句做跟踪检查。例 2.15 显示了怎样进行一个程序跟踪检查。跟踪检查是一种很有效的除错技术。

注意: 如果对一个计算浮点值的表达式求值,而对 int 型的变量没有使用转型运算符,那么小数部分会被截掉。在这种情况下,编译器有可能发出一个关于隐式类型转换的信息。

2.6 自增和自减运算符

前面学习了如何声明一个变量以及如何向变量中输入值,本节将学习另外两个运算符:自增和自减运算符。这些运算符经常被 C++ 程序员使用,而且它是一个很有用的编程工具。

假定 count 是一个 int 变量,语句:

```
count=count+1;
```

使 count 的值加 1。为了执行该赋值语句，计算机首先计算右边表达式的值，即 count+1，然后把该值赋给左边的变量 count。

在后面将会看到，这样的语句经常用来跟踪特定事情发生的次数。为了加速这些语句的执行，C++ 提供了自增运算符 ++（它每次使变量的值加 1）和自减运算符 --（它每次使变量的值减 1）。自增和自减运算符都包含两种形式：前自增和后自增。自增运算符的语法是：

```
前自增  ++variable
后自增  variable++
```

自减运算符的语法是：

```
前自减  --variable
后自减  variable--
```

下面看一些例子：

语句：

```
++count;
```

或

```
count++;
```

使 count 的值每次增加 1。同样，语句：

```
--count;
```

```
count--;
```

使 count 的值每次减 1。

因为自增和自减都被集成到 C++ 中，所以变量的值可以很快地加 1 或减 1，而不必采用赋值语句的形式。

由上面的这些例子可以看出，前或后自增运算符都是使变量的值加 1。同样，前或后自减运算符都是使变量的值减 1。这些不同形式的运算符有什么区别呢？当使用这些运算符的变量被应用到表达式中的时候，区别就明显了。

假定 x 是 int 型的变量。如果 ++x 用在一个表达式中，x 的值先被加 1，然后 x 的新值被用来计算表达式。另一方面，如果 x++ 用在表达式中，表达式首先使用 x 的当前值，然后 x 的值再加 1。下面的例子阐明了前、后自增运算符的不同。

假定 x 和 y 是 int 型变量。考虑下面的语句：

```
x=5;
y=++x;
```

第一条语句将 5 赋值给 x。为了计算第二条语句的值（它使用前自增运算符），首先 x 的值加 1 等于 6，然后将该值 6 赋值给 y。第二条语句执行之后，x 和 y 的值都为 6。

现在考虑下面的语句：

```
x=5;
y=x++;
```

同前面一样，第一条语句将 5 赋值给 x。在第二条语句中，后自增运算符应用于 x，为了执行第二条语句，首先 x 的值 5 用来计算表达式的值，然后 x 的值增加 1 变为 6。最后，表达式的值（5）

存储到 y 中。在第二条语句执行之后, x 的值是 6 而 y 的值是 5。

下面的例子进一步阐述了自增运算符是如何进行计算的。

例 2.16

假定 a 和 b 是 int 型变量, 并且:

```
a=5;  
b=2+(++a);
```

第一条语句将 5 赋值给 a。为了执行第二条语句, 首先计算表达式 2+(++a)。因为前自增运算符应用于 a, 所以首先 a 的值增加 1 变为 6, 然后 2 加上 6 得到 8, 然后赋值给 b。这样, 在第二条语句执行之后, a 是 6, b 是 8。

注意: 本书在单独的语句中将经常对变量使用自增和自减运算符。即使用自增和自减运算符的变量不属于任何一个表达式。

2.7 输出

如果一个程序不能在输出设备上输出结果, 那么它就没有太大的用处。尽管前面已学习过如何把数据放到计算机的内存中以及怎样对数据进行操作, 但是没有学习怎样使程序显示结果。

注意: 标准输出设备通常是屏幕。

在 C++ 中, 向标准输出设备输出结果通常是通过 cout 和运算符 << 来完成的。cout 与 << 一起使用的语法是:

```
cout<<expression or manipulator<<expression or manipulator...;
```

这称为输出语句, 有时也称为 cout 语句。在 C++ 中, << 称为流插入运算符。

用 cout 语句产生输出需要遵守下面两条规则:

1. 计算表达式的值, 该值被打印到输出设备的当前光标位置。
2. 操纵符用来规定输出的格式。最简单的操纵符是 endl (最后一个字符是字母 l), 它可使光标移动到下一行的开始位置。

下面的例子演示了 cout 语句是如何工作的。在 cout 语句中, 当一个字符串只包含一个变量或单值的表达式时, 其值等于自身。

注意: 当输出语句输出 char 值时, 它只输出没有单引号的字符 (除非单引号是输出语句的一部分)。

例如, 假定 ch 是一个 char 变量并且 ch='A', 则语句:

```
cout<<ch;
```

或

```
cout<<'A';
```

输出:

A

同样, 当使用一条输出语句输出字符串时, 它只输出没有双引号的字符串 (除非双引号是输出的一部分)。

例 2.17

考虑下面的语句。每条语句的右边显示输出。

语句	输出
1 cout<<29/4;	7
2 cout<<"Hello there. ";	Hello there.
3 cout<<12;	12
4 cout<<"4+7";	4+7
5 cout<<4+7;	11
6 cout<<'A';	A
7 cout<<"4 + 7 = "<<4 + 7;	4 + 7 = 11
8 cout<<2+3*5;	17
9 cout<<"Hello \nthere. ";	Hello there.

看一下语句 9 的输出。前面讲过换行符是 '\n'，它使光标在打印之前移动到下一行的开始。所以，当 '\n' 出现在一个输出语句的字符串中时，它使光标在输出设备上移动到下一行的开始。这就解释了为什么“Hello”和“there.”被打印到不同的行上。

注意：在 C++ 中，\ 称为转义字符，\n 称为换行符。

所有的变量必须被正确地初始化，否则变量存储的值可能没有意义，同时，C++ 不能自动地初始化变量。

C++ 语句的输出：

```
cout<<a<<endl;
```

是有意义的，但前提是变量 a 包含一个给定的值。例如，下面的 C++ 语句：

```
a=45;
cout<<a<<endl;
```

将会产生输出 45。

例 2.18

假定下列语句是按给定的顺序执行的。

```
int a, b, c, d;

a = 65 ;                               //Line 1
b = 78 ;                               //Line 2

cout<<29/4<<endl;                       //Line 3
cout<<3.0/2<<endl;                       //Line 4
cout<<"Hello there.\n";                   //Line 5
cout<<7<<endl;                           //Line 6
cout<<3+5<<endl;                         //Line 7
cout<<"3+5";                             //Line 8
cout<<endl;                             //Line 9
cout<<2+3*6<<endl;                       //Line 10
cout<<"a"<<endl;                         //Line 11
cout<<a<<endl;                           //Line 12
cout<<b<<endl;                           //Line 13
cout<<c<<'\n';                           //Line 14
cout<<d;                                 //Line 15
cout<<endl;                             //Line 16
```

在下面的输出中，标有“Output of statement at”的列和行标号不是输出的一部分。该列显示的行数字是为了更容易明白输出所对应的语句。

	Output of Statement at
7	Line 3
1.5	Line 4
Hello there.	Line 5
7	Line 6
8	Line 7
3+5	Line 8
20	Line 10
a	Line 11
65	Line 12
78	Line 13
6749684	Line 14
4203005	Line 15

大多数情况下，输出是直观的。看一下 7, 8, 9, 10 行语句的输出。第 7 行语句输出 $3 + 5$ 的结果（是 8）并且将光标移到下一行的开始。第 8 行语句输出字符串 $3 + 5$ 。注意第 8 行语句只包含字符串 $3 + 5$ 。所以，在打印 $3 + 5$ 之后，光标停留在 5 的位置，而并不移到下一行的开始。

第 9 行输出语句只包含 `endl` 操纵符，它把光标移动到下一行的开始。所以，当第 10 行语句执行时，输出开始于该行的开始。注意在该输出中，列“output of Statement at”不包含第 9 行。这是因为第 9 行语句不能产生任何可打印的输出。它只是把光标移动到下一行的开始。然后，第 10 行语句输出表达式 $2 + 3 * 6$ 的结果，即 20。然后运算符 `endl` 将光标移动到下一行的开始。

仔细看一下第 14 和第 15 行语句的输出。注意变量 `c` 和 `d` 不能被正确地初始化。所以，`c` 和 `d` 的打印值（第 14 行为 6749684，第 15 行为 4203005）是没有意义的。如果在用户的系统中执行这些语句，可以得到 `c` 和 `d` 的不同值。

注意：输出或访问表达式中变量的值不会破坏或修改变量的内容。

现在看一下换行符 `\n`。考虑下面的 C++ 语句：

```
cout<<"Hello there.";
cout<<"My name is Goofy.";
```

如果这些语句按顺序执行，输出是：

```
Hello there.My name is Goofy.
```

现在看一下下面的语句：

```
cout<<"Hello there.\n";
cout<<"My name is Goofy.";
```

则这些 C++ 语句的输出是：

```
Hello there.
My name is Goofy.
```

当字符串与 `\n` 搭配时，光标会定位到下一行的开始。注意，`\n` 可能出现在字符串中的任何位置。例如下面的语句：

```
cout<<"Hello\nthere.\nMy name is Goofy.";
```

其输出是：

```
Hello
there.
My name is Goofy.
```

而且, 需要指出语句:

```
cout<<'\\n';
```

的输出与语句:

```
cout<<"\\n";
```

的输出是相同的。

它们都等价于语句:

```
cout<<endl;
```

的输出。

这样, 下面语句的输出

```
cout<<"Hello there.\\n";
cout<<"My name is Goofy.";
```

与语句:

```
cout<<"Hello there."<<endl;
cout<<"My name is Goofy.";
```

的输出是等价的。

例 2.19

考虑下面的 C++ 语句:

```
cout<<"Hello there.\\nMy name is Goofy.";
```

或

```
cout<<"Hello there.";
cout<<"\\nMy name is Goofy.";
```

或

```
cout<<"Hello there.";
cout<<endl<<"My name is Goofy.";
```

在每一个例子中, 语句输出都是:

```
Hello there.
My name is Goofy.
```

例 2.20

C++ 语句的输出:

```
cout<<"Count...\\n....1\\n.....2\\n.....3";
```

或

```
cout<<"Count..."<<endl<<"....1"<<endl
<<".....2"<<endl<<".....3";
```

其结果为:


```
Count...
....1
.....2
.....3
```

例 2.21

假定想要在一行内输出下面的句子作为一部分信息：

It is a sunny, warm, and not a windy day. Let us go golfing.

显然，可以使用 cout 语句来产生输出。然而，在程序代码中，该语句作为 cout 语句的一部分可能不能在一行中输出。当然，可以使用多个 cout 语句，如下所示：

```
cout<<"It is a sunny, warm, and not a windy day.";
cout<<"Let us go golfing."<<endl;
```

注意第一条语句末尾的分号和第二条语句开始的标识符。而且还要注意在第一条语句的末尾没有操纵符 endl。这里，两条 cout 语句用来在一行输出一个句子。同样，可以使用下面的 cout 语句来输出该句子：

```
cout<<"It is a sunny, warm, and not a windy day."
<<"Let us go golfing."<<endl;
```

在这条语句中，注意第一行的末尾没有分号，第二行的开始没有标识符 cout。因为第一行末尾没有分号，cout 语句在第二行继续。而且，注意每一行中语句的开始和末尾都标有双引号。字符串被拆分成两个字符串，但是两个字符串都是同一个 cout 语句的一部分。

如果出现在 cout 语句中的字符串很长，而且想在一行输出该字符串，那么可以使用上面的两种方法中的任何一种把 cout 语句的字符串拆分。然而，下面的语句是不正确的：

```
cout<<"It is a sunny, warm, and not a windy day.
Let us go golfing."<<endl;
```

换言之，键盘中的 Enter 键不是字符串的一部分。即在代码中，一个字符串不能使用键盘上的 Enter 键被拆分成几行。

前面讲过，换行符是 \n，它使光标移到下一行的开始。在 C++ 中有很多的转义序列符，它们允许对输出进行控制。

表 2.4 列出了一些常用的转义序列符。

表 2.4 常用转义序列符

	转义序列	描述
\n	换行	光标移到下一行的开始
\t	制表符	光标移到下一个 Tab 停留处
\b	退格	光标向左移动一个空格
\r	回车	光标移到当前行（不是下一行）的开始
\\	反斜杠	打印反斜杠字符
\'	单引号	打印单引号字符
\"	双引号	打印双引号字符

下面的例子显示了一些转义序列符的功能。

例 2.22

下面语句：

```
cout<<"The newline escape sequence is\\n." <<endl;
```


准 C++ 头文件的名称是 `math.h`。这样，为了在标准 C++ 程序中包含头文件 `iostream.h`，需要使用下面的语句：

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

注意：关于 ANSI/ISO 标准 C++ 和标准 C++ 头文件名称的转换，可以参看附录 E。

预处理器命令在程序编译之前处理。这样，C++ 程序需要 6 个步骤，如图 2.18 所示：

从图 2.18 中可以总结出，C++ 系统有三个基本的组成部分：程序开发环境、C++ 语言和 C++ 库。所有三个组成部分是 C++ 系统的集成部分。程序开发环境包含图 2.18 所示的 6 个步骤。整本书都会讲述 C++ 语言，需要的时候将会讨论 C++ 库的组成部分。

2.8.1 在程序和命名空间中使用 `cin` 和 `cout`

前面讲过，`cin` 和 `cout` 都是预定义标识符。在 ANSI/ISO 标准 C++ 中，这些标识符在头文件 `iostream` 中而且是在一个命名空间中声明。该命名空间是 `std`。在第 8 章中将会正式地定义和详细地讨论命名空间机制，到现在为止，只需知道如何运用头文件 `iostream` 中的 `cin` 和 `cout` 以及其他标识符。

有几种方法使用在 `std` 命名空间中声明的标识符。使用 `cin` 和 `cout` 的一种方法是在整个程序中把它们当做 `std::cin` 和 `std::cout`。另一种方法是在程序中包含下面的语句：

```
using namespace std;
```

该语句在 `#include<iostream>` 指令之后。然后可以使用 `cin` 和 `cout`，而不必使用前缀 `std::`。为了简化 `cin` 和 `cout` 的使用，本书使用了第二种形式。即在程序中使用 `cin` 和 `cout`，该程序将会包含下面的两条语句：

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

在 C++ 中，`namespace` 和 `using` 是保留字。

注意：命名空间机制是 ANSI/ISO 标准 C++ 的一个特征。随着学习更多的 C++ 程序，可以逐渐了解其他头文件。例如，头文件 `cmath` 包含很多数学函数的详细说明。同样，头文件 `iomanip` 也包含了很多有用的函数和运算符，这些函数和运算符有助于以特定的方式来确定输出格式。然而，就像头文件 `iostream` 中的标识符一样，ANSI/ISO 标准 C++ 头文件的标识符也是在一个命名空间中被声明的。

每一个头文件中命名空间的名称是 `std`。所以，无论什么时候讨论 ANSI/ISO 标准 C++ 中一个头文件的特定特征时，本书将会参照没有标注 `std::` 的标识符。而且，在程序中，为了简化对标识符的访问，语句“`using namespace std;`”将会被包括进来。而且，如果一个程序使用多个头文件，则只需要一条这样的 `using` 语句。该 `using` 语句将会出现在所有头文件的后面。

在标准 C++ 中，`cin` 和 `cout` 在头文件 `iostream.h` 中声明。所以，在标准 C++ 中，为了在程序中使用 `cin` 和 `cout`，必须包含下面的语句：

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



图 2.18 处理一个 C++ 程序

2.8.2 在程序中使用 string 数据类型

前面讲过, string 数据类型是一种程序员自定义的数据类型, 并且不能在程序中直接使用。为了使用 string 数据类型, 需要从头文件 string 中访问它的定义。所以, 为了在程序中使用 string 数据类型, 必须包含下面的预处理器指令:

```
#include <string>
```

一些编译器不允许 string 数据类型与标准 C++ 类型头文件 (也就是有扩展名.h 的头文件) 一同使用。换言之, 如果在程序中使用 string 数据类型, 那么使用的头文件必须是 ANSI/ISO 标准头文件, 即没有扩展名.h 的头文件。在这种情况下, 为了使用 cin, cout 和 string 数据类型, 程序必须包含下面的处理器指令和语句 (查看用户的编译器文档):

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

注意: 在本章后面的编程例子中, 将会看到既使用标准 C++ 又使用 ANSI/ISO 标准 C++ 的 C++ 程序代码。在除了第8章的所有剩余章节中, 编程代码将会使用 ANSI/ISO 标准 C++ 风格的头文件。下面几章显示了相应的标准 C++ 头文件。在与本书所对应的网页上, 显示了既使用标准 C++ 格式又使用 ANSI/ISO 标准 C++ 格式的编程代码。

2.9 编程风格和形式

在前面的部分中学习了很多关于编写程序的 C++ 概念。然而在开始编写程序之前, 还需要学习适当的程序结构。在 C++ 程序中使用合适的结构会使程序更易读并更易修改。理解或修改语法正确但没有良好结构的程序是非常让人伤脑筋的事。

每一个 C++ 程序分为两个部分: 预处理器指令和程序。预处理器指令是在编译之前指导预处理器修改 C++ 程序的命令。程序包含完成一些有意义结果的语句。将预处理器指令和程序语句组织在一起就组成了 C++ 源代码。为了使这些代码可用, 它们必须保存在以扩展名为.cpp 的文件中。

当程序编译时, 编译器产生了目标代码, 它保存在扩展名为.obj 的文件中。当目标码与系统资源连接时, 就会产生可执行文件, 该文件保存在扩展名为.exe 的文件中。目标码的文件名以及可执行代码的文件名与源代码的文件名是相同的。例如, 如果源代码在名字为 firstProg.cpp 的文件中, 那么包含目标代码的文件名为 firstProg.obj, 包含可执行代码的名字为 firstProg.exe。

注意: 上面给定的扩展名, 即.cpp, .obj 和 .exe, 是系统独立的。为了确信这一点, 可以查看系统的 SDK 文档。

另外, 每一个 C++ 程序必须满足特定的语言规则。一个 C++ 程序必须包含 main() 函数。也必须遵守语法规则, 即区分程序中什么是对的, 什么是错的, 什么是合法的, 什么是不合法的。其他规则是为了给语言提供精确的意义, 即它们支持语言的语义。下面的部分将把学过的 C++ 编程元素放在一起创建一个有意义的程序。这些部分包含了 main() 函数、语法、空格的使用、分号、括号和逗号的使用、语义、形式和风格、文档 (包含注释和命名标识符) 和命令行。

2.9.1 main() 函数

前面讲过, 每一个 C++ 程序必须有一个 main() 函数。main() 函数的基本部分是函数头和函数体。

函数头的形式如下：

```
typeofFunction main(argument list)
```

例如语句：

```
int main(void)
```

表示 `main()` 函数返回 `int` 数据类型的值，它没有参数。不必把 `void` 放到括号中，但是括号仍然是需要的。这样，前面的语句等价成为下面的语句：

```
int main()
```

在 C++ 中，`void` 是一个保留字。

函数体必须用花括号括起来，它包含两种类型的语句：

- 声明语句
- 可执行语句

声明语句用来声明变量。在 C++ 中，变量或标识符能在程序中的任何位置声明，但是在使用之前必须被声明。

例 2.23

下面的语句是变量声明的例子：

```
int a,b,c;
double x,y;
```

可执行语句执行计算、数据操作、产生输出、接受输入，等等。一些可执行语句就是前面讲过的赋值、输入和输出语句。

例 2.24

下面的语句是可执行语句的例子：

```
a = 4;           //assignment statement
cin>>b;          //input statement
cout<<a<<endl<<b<<endl; //output statement
```

本书中的 `main()` 函数包含下面的语法：

```
int main()
{
    statement1
    .
    .
    statementn
    return 0;
}
```

在 `main()` 函数的语法中，通常每一条语句既可以是声明语句也可以是可执行语句。语句 “`return 0;`” 必须包含在 `main()` 函数中并且必须是最后一条语句。如果语句 “`return 0;`” 错放在函数体其他位置，则产生的程序结果可能不是所希望的。语句 “`return 0;`” 的意义将在第 6 章进行讨论。在 C++ 中，`return` 是一个保留字。

2.9.2 语法

语言的语法规则告诉什么是合法、什么是不合法的。编译期间会检测语法错误。例如下面的一段C++语句：

```
int x;           //Line 1
int y           //Line 2:syntax error
double z;       //Line 3
y = w + x;      //Line 4:syntax error
```

当这些语句编译时，第二行将会产生编译错误，因为在声明变量y后没有分号。第4行也会产生编译错误，因为标识符w没有声明就被使用了。

在第1章中讲过，可以通过编辑器向程序输入程序。当程序键入时，错误几乎是不可避免的。所以，当程序编译时，很有可能发现语法错误。一个特定位置的语法错误很可能导致随后几条语句的错误。因为缺少一个字符而导致4个或5个错误是很常见的。然而，当改掉第一个语法错误后再编译程序时，由该语法错误引起的后面的语法错误就可能消失。随着越来越熟悉C++以及经验的增加，将会学到怎样快速地检查和定位语法错误。而且，编译时不仅能发现错误，还能显示错误，有时甚至能告诉用户错误在哪里以及怎样修改这些错误。

2.9.3 空格的使用

在C++中，当数据输入时可以使用一个或多个空格来分开输入的数字。空格也可以用来将保留字和标识符互相分开，或与其他符号分开。不允许空格出现在保留字或标识符中。

2.9.4 分号、花括号和逗号的使用

所有C++的语句必须以分号结束。分号也称为语句终止符。

注意花括号{}不是C++语句本身的一部分，尽管它们经常出现在空代码行中。可以把花括号作为分隔符，因为它包含了函数体并且将其与其他部分的程序分开。花括号还有其他的用处，这将在后面进行介绍。

前面讲过，逗号用来隔开一个列表项。例如，当声明同一种类型的变量不止一个时，可以使用逗号。

2.9.5 语义

一组关于语言意义的规则称为语义。例如，算术运算符的优先级顺序是语义规则。

如果一个程序包含语法错误，编译器将会发出警告。当程序含有语义错误时会出现什么问题呢？虽然消除了所有语法错误，但程序仍无法运行。即使它能够运行，结果可能也不是期望得到的。例如，下面的两行代码都是语法正确的表达式，但是它们有不同的意思：

$2+3*5$

和

$(2+3)*5$

如果在一个程序中用其中的一行替换另一行，就不会得到相同的结果。尽管数字是相同的，但是语义却是不同的。

2.9.6 形式和风格

读者可能会认为C++有太多的规则,但实际上它有很大的自由度。例如:考虑下面两种声明变量的方法:

```
int feet, inch;  
double x, y;
```

和

```
int feet, inches; double x, y;
```

计算机很容易理解上面的两种格式,但是第一种格式更容易读和遵守。当然,无论哪种格式如果丢掉一个逗号或分号,都可能导致各种各样的错误信息。

那么空格呢?什么地方有用,什么地方没有用呢?请看下面的两条语句:

```
int a,b,c;
```

和

```
int a,  b,  c;
```

这两种声明是一样的。这里,第二行标识符之间的空格是没有意义的。再来看一下下面的语句:

```
inta,b,c;
```

语句包含一个语法错误。int中的t和标识符a之间缺少空格,这样改变了保留字int,标识符a成为了一个新的标识符inta。

清晰的语法和语义规则能使用户自由地采纳所喜欢的方式,并且容易被理解。

2.9.7 文档

用户所编写的程序不仅自己能清楚,对其他任何人也应该很容易理解。所以,用户应该为自己的程序编写一些文档。一个具有很好文档的程序更容易理解和修改,即使程序已经编写了很长时间。可以使用注释来归档程序。出现在程序中的注释应该解释程序的目的、记录程序的作者以及解释特定语句的目的。

注释

C++有两种类型的注释:单行注释和多行注释。单行注释以//开始,它可以出现在程序中的任何地方。在该行//后出现的任何代码都会被编译器忽略。多行注释用/*和*/括起来,编译器会忽略所有出现在/*和*/之间的代码。

可以在程序的前面插入特定的注释,来给出程序的简明解释和程序员的信息。也可以在每一个重要步骤之前做出注释并给出本步骤功能的简单描述。

命名标识符

当命名标识符时,可以选择那些能自解释的标识符。例如,考虑下面两组语句:

```
const double a = 2.54;    //conversion constant  
double x;                //variable to hold centimeters  
double y;                //variable to hold inches  
  
x = y * a;
```

以及

```
const double conversion = 2.54;
```

```
double centimeters;  
double inches;  
  
centimeters = inches * conversion;
```

可以看到，自解释标识符使注释显得不是很必要。

考虑自解释标识符 `inchperfoot`，该标识符称为组合字。在使用自解释的标识符时，可能经常包括组合字，它看起来会使文档变得不清晰。通过大写每个单词的开始字母或在每一个新词前加下划线的办法，可以使组合字更容易明白，例如，可以使用 `inchPerFoot` 或 `inch_per_foot` 来创建一个更清晰的标识符。

提示行

有一部分文档是为了方便使用而编写的命令行，这样当用户使用程序的时候会知道去做什么。在一个运行的程序前，用户可能不知道输入什么数据，这是很令人伤脑筋的。提示行是通知用户去做什么的语句。例如下面的 C++ 语句，其中 `num` 是一个 `int` 型变量：

```
cout<<"Please enter a number between 1 and 10 and"  
    <<" press the return key"<<endl;  
cin>>num;
```

当这两条语句按给定的顺序执行时，首先 `cout` 语句会使屏幕上出现下面一行文本：

```
Please enter a number between 1 and 10 and press the return key
```

在看到该行后，用户知道他们必须输入一个数字并且按 Enter 键。如果程序只包含第二条语句，用户不知道必须输入一个数字，计算机就会永远等待下去。前面的 `cout` 语句是提示行的一个例子。

在一个程序中，无论什么时候需要用户输入，都必须包含必要的提示行。而且，这些提示行应该尽可能多地包含哪种输入是可接受的信息。例如，前面的提示行不仅告诉用户输入一个数字，还告诉用户所输入的数字应该在 1 和 10 之间。

2.10 赋值语句的更多知识

目前所看到的赋值语句称为简单赋值语句。在特定情况下，可以使用称为复合赋值语句的特殊赋值语句编写简单的赋值语句，这样会更简洁。对于算术运算符来说，C++ 定义了复合赋值语句，如下所示：这里 `op` 是任何一种算术运算符。

使用复合赋值语句，可以重新编写简单赋值语句：

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

为如下形式：

```
variable op = expression;
```

这样，通过把算术运算符和赋值运算符联系起来的办法，复合赋值语句允许使用一种简洁的方式来编写简单赋值语句。

例 2.25

该例显示了几个与简单赋值语句等价的复合赋值语句。

简单赋值语句

```
I = I + 5;  
counter = counter + 1;  
sum = sum + number;
```

复合赋值语句

```
I += 5;  
counter += 1;  
sum += number;
```

```
amount = amount * (interest + 1);
x = x / (y + 5);
```

```
amount *= interest + 1;
x /= y + 5;
```

注意：任何一条赋值语句都可以被转换为简单赋值语句。然而，一条简单赋值语句不一定能够（容易地）转换成复合赋值语句。例如下面的简单赋值语句：

```
x = x * y + z - 5;
```

为了将该语句转换为一条复合赋值语句，变量 x 必须是右边的公因子，而实际未必如此。所以，不能立刻将该语句转换为一条复合赋值语句。实际上，等价的复合语句是：

```
x = y + (z - 5) / x;
```

而这比简单赋值语句更为复杂。

注意：在程序代码中，本书只使用复合运算符 $+=$ 。使用 $+=$ 复合运算符后，语句 “ $a=a+b;$ ” 就可以写为 “ $a+=b;$ ”

2.11 编程实例：长度转换

编写一个程序，输入是用英尺和英寸表示的长度。然后程序应该转换并输出用厘米表示的长度。假定用英尺和英寸表示的给定长度是整数。

输入：用英尺和英寸表示的长度。

输出：厘米表示的等效长度。

问题分析和算法设计

长度由英寸和英尺给出，需要求得用厘米表示的等效长度。1 英寸等于 2.54 厘米。程序的第一步应该将英尺和英寸表示的长度全部转换为英寸表示的长度。然后再使用转换公式：1 英寸 = 2.54 厘米，求得等效的厘米长度。为了将英尺转换为英寸，应该将英尺乘以 12，即 1 英尺等于 12 英寸，然后再加上给定的英寸数。

例如：如果输入是 5 英尺 7 英寸，那么可以用下面的方法求得总英寸数：

```
totalInches = (12 * feet) + inches
             = 12 * 5 + 7
             = 67
```

然后应用转换公式：1 英寸 = 2.54 厘米，计算用厘米表示的长度。

```
centimeters = totalInches * 2.54
             = 67 * 2.54
             = 170.18
```

通过上面的分析，可以设计出下面的算法：

1. 得到英寸和英尺表示的长度。
2. 将长度转化为英寸。
3. 将英寸转换为厘米。
4. 输出厘米长度。

变量 程序的输入有两项：一项是英尺，一项是英寸。这样，就需要两个变量：分别存储英尺数和英寸数。因为程序首先将给定的长度转换为英寸，所以需要另一个变量来存储总的英寸数。

还需要一个变量来存储等效的厘米表示的长度。因此，需要下面的变量：

```
int feet;           //variable to hold given feet
int inches;         //variable to hold given inches
int totalInches;    //variable to hold total inches
double centimeters; //variable to hold length in centimeters
```

常量 为了计算等效的厘米表示的长度，需要将总英寸数乘以 2.54。不是直接在程序中使用 2.54，而是用一个常量来声明该值。同样，为了求得总的英寸数，需要将英尺数乘以 12 并且加上英寸数。

```
const double conversion = 2.54;
const int inchesPerFoot = 12;
```

主要算法

1. 提示用户输入。（否则，用户将会盯着屏幕而不知道做什么。）
2. 得到数据。
3. 回显输入，即输出程序读入的输入数据。（没有这一步骤，在程序执行之后，将不会知道输入人是什么。）
4. 求出用英寸表示的长度。
5. 输出用英寸表示的长度。
6. 将长度转换为厘米。
7. 输出用厘米表示的长度。

集合在一起 在分析了问题和设计了算法之后，下一步骤就是把算法翻译成 C++ 代码。因为这是编写的第一个完整的 C++ 程序，所以回顾一下必要的步骤。

程序以注释开始，来说明它的目的和功能。由于既有输入（用英尺和英寸表示的长度）又有输出（用厘米表示的等效长度），所以将会使用系统的输入/输出资源。换言之，程序将会使用输入语句来得到数据，用输出语句来打印结果。因为数据从键盘输入并显示在屏幕上，所以程序必须包含头文件 `iostream`。这样，在上面描述的注释后，程序的第一条语句是使用预处理指令来包含该头文件。

程序需要两种类型的内存变量来对数据进行操作：常量和变量。常量通常放在 `main()` 函数之前，这样它们可以在整个程序中使用。

该程序只有一个函数，即 `main()` 函数，它包含了函数体中所有的程序指令。另外，程序需要变量来对数据进行操作，这些变量应该在 `main()` 函数的函数体中进行声明。在函数体中声明这些变量的原因将在第 7 章进行介绍。`main()` 函数的函数体也将包含执行算法的 C++ 语句。所以，`main()` 函数的函数体如下所示：

```
int main()
{
    declare variables
    statements
    return 0;
}
```

为了编写完整的长度转换程序，应该遵循下面这些步骤：

1. 用文档注释开始部分。
2. 包含相关的头文件。

3. 如果需要的话, 可以声明变量。
4. 编写 main() 函数的定义。

ANSI/ISO 标准 C++ 中的完整的程序清单

```

/*****
//   Program Convert: this program converts measurements
//   in feet and inches into centimeters using the
//   approximation that 1 inch is equal to
//   2.54 centimeters.
*****/

//header file
#include <iostream>
using namespace std;

//named constants
const double conversion = 2.54;
const int inchesPerFoot = 12;

int main()
{
    //declare variables
    int feet;
    int inches;
    int totalInches;
    double centimeter;

    //Statements: Step 1 ~ Step 7
    cout<<"Enter two integers, one for feet, "
        <<"one for inches: ";           //Step 1
    cin>>feet>>inches;                 //Step 2
    cout<<endl;
    cout<<"The numbers you entered are "<<feet
        <<" for feet "<<"and" <<inches
        <<" for inches. "<<endl;       //Step 3

    totalInches = inchesPerFoot * feet + inches; //Step 4
    cout<<endl;
    cout<<"The total number of inches = "
        <<totalInches<<endl;           //Step 5
    centimeter = conversion * totalInches;       //Step 6
    cout<<"The number of centimeters = "
        <<centimeter<<endl;           //Step 7
    return 0;
}

```

运行实例: 在本运行实例中, 用户输入用阴影表示。

```

Enter two integers, one for feet, one for inches: 15 7
The numbers you entered are 15 for feet and 7 for inches.
The total number of inches = 187
The number of centimeters = 474.98

```

2.12 编程实例: 找零钱

编写一个程序, 输入是用 cent 表示的零钱。然后计算用 half-dollar, quarter, dime, nickel 和 penny 各个单位表示的数量, 应尽可能多的用 half-dollar 表示, 然后再依次用 quarter, dime, nickel 和 penny 来表示。例如, 483 cent 应该计算为 9 个 half-dollar, 1 个 quarter, 1 个 nickel 和 3 个 penny。

输入 用 cent 表示的零钱。

输出 用 quarter, dime, nickel 和 penny 表示等价的零钱。

问题分析和算法设计

假定有 646 cent, 为了求得 half-dollar 的数量, 应该使 646 除以 50, 即 half-dollar 的值, 商为 12, 余数为 46。商 12 就是 half-dollar 的数目, 余数 46 是剩余的零钱。

下一步用 25 除上面所得的余数, 就可以得到 quarter 的数目。因为余数为 46, 被 25 除后商为 1, 即 quarter 的数目为 1, 余数为 21, 即剩余的零钱。对 dime, nickel 应用同样的方法。为了计算整除后的余数, 应该使用 % 操作符。

上述讨论可以得到下面的计算:

1. 零钱 = 646
2. half-dollar 数量 = $646 / 50 = 12$
3. 剩余零钱 = $646 \% 50 = 46$
4. quarter 数量 = $46 / 25 = 1$
5. 剩余零钱 = $46 \% 25 = 21$
6. dime 数量 = $21 / 10 = 2$
7. 剩余零钱 = $21 \% 10 = 1$
8. nickel 数量 = $1 / 5 = 0$
9. penny 数量 = 剩余零钱 = $1 \% 5 = 1$

该讨论可以翻译成下面的算法:

1. 得到用 cent 表示的零钱。
2. 计算 half-dollar 的数目。
3. 计算剩余的零钱。
4. 计算 quarter 的数目。
5. 计算剩余的零钱。
6. 计算 dime 的数目。
7. 计算剩余的零钱。
8. 计算 nickel 的数目。
9. 计算剩余的零钱。
10. 剩余的零钱是 penny 的数目。

变量 从前面的讨论和算法可以看出, 程序需要变量来存储 half-dollar, quarter 等变量。然而, half-dollar, quarter 等变量在后面的计算中并不使用, 所以程序只需要简单地输出这些值, 而不必保存其中的每一个变量。惟一需要保持变化的是零钱, 所以程序实际上只需要一个变量:

```
int change;
```

常量 为了计算等价的零钱, 程序需要使用 half-dollar 的值 50 进行计算。同样, quarter 值为 25; dime 值为 10; nickel 值为 5。因为这些数据比较特别, 而且程序不止一次使用它们, 所以声明为常量是比较合适的。使用常量也简化了后面程序的修改。

```
const int Halfdollar = 50;  
const int Quarter = 25;
```



```
const int Dime = 10;
const int Nickel = 5;
```

主要算法

1. 提示用户以进行输入。
2. 得到输入。
3. 回显输入, 显示用户输入的零钱。
4. 计算和打印 half-dollar 的数量。
5. 计算剩余的零钱。
6. 计算和打印 quarter 的数量。
7. 计算剩余的零钱。
8. 计算和打印 dime 的数量。
9. 计算剩余的零钱。
10. 计算和打印 nickel 的数量。
11. 计算剩余的零钱。
12. 打印剩余的零钱。

完整的程序清单

```
// *****
// Program Make Change: Given any amount of change
// expressed in cents, this program computes the number
// of half-dollars, quarters, dimes, nickels, and
// pennies to be returned, returning as many
// half-dollars as possible, then quarters, dimes,
// nickels, and pennies in that order.
// *****

//header file
#include <iostream>
using namespace std;

//named constants
const int Halfdollar = 50;
const int Quarter = 25;
const int Dime = 10;
const int Nickel = 5;

int main()
{
    //declare variable
    int change;

    //Statements: Step 1 - Step 12
    cout<<"Enter change in cents: "; //Step 1
    cin>>change; //Step 2
    cout<<endl;
    cout<<"The change you entered is "<<change<<endl; //Step 3
    cout<<"The number of half-dollars to be returned "
        <<"are "<<change / Halfdollar<<endl; //Step 4
    change = change % Halfdollar; //Step 5
    cout<<"The number of quarters to be returned are "
        <<change / Quarter<<endl; //Step 6
    change = change % Quarter; //Step 7
    cout<<"The number of dimes to be returned are "
```

```

        <<change / Dime<<endl;           //Step 8
        change = change % Dime;          //Step 9

        cout<<"The number of nickels to be returned are "
            <<change / Nickel<<endl;      //Step 10
        change = change % Nickel;        //Step 11

        cout<<"The number of pennies to be returned are "
            <<change<<endl;              //Step 12
        return 0;
    }

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Enter change in cents: 583
The change you entered is 583
The number of half-dollars to be returned are 11
The number of quarters to be returned are 1
The number of dimes to be returned are 0
The number of nickels to be returned are 1
The number of pennies to be returned are 3

```

注意：为了使用标准 C++ 头文件编写该程序，需要把语句：

```

#include<iostream>
using namespace std;

```

替换为：

```

#include<iostream.h>

```

2.13 小结

1. C++ 程序是函数的集合。
2. 每一个 C++ 程序都有一个称为 `main()` 的函数。
3. 在 C++ 中，标识符是一些事物的名字。
4. C++ 标识符包含字母、数字和下划线，而且必须以字母和下划线开头。
5. 保留字在程序中不能用做标识符。
6. 所有 C++ 中的保留字由小写字母构成（参看附录 A）。
7. 最常用的字符集是 ASCII 码（有 128 个值）和 EBCDIC 码（有 256 个值）。
8. 字符对照序列是字符数据集中的预制数字。
9. C++ 中的算术运算符是加（+）、减（-）、乘（*）、除（/）和求余（%）。
10. 求余运算符只适用于整数操作数。
11. 算术表达式使用优先级规则和运算符的结合性进行求值。
12. 整型表达式或整数表达式中的所有操作数都是整数，浮点型表达式中的所有操作数都是小数数字。
13. 混合表达式是一种包含了整数和小数的表达式。
14. 当计算表达式中一个运算符的值时，如果运算符包含混合操作数，那么整数会转换为后面补零的浮点型数。
15. 可以使用强制转换运算符强制性地将一种数据类型转换为另一种数据类型。
16. 程序执行期间，常量的内容是不会被改变的。
17. 常量是通过保留字 `const` 加以声明的。

18. 常量应在声明时初始化。
19. 所有变量在使用前必须被声明。
20. C++ 不能自动初始化变量。
21. 每一个变量有一个名字、值、数据类型和数据值。
22. 当变量赋予新值时, 旧值就被覆盖了。
23. 只有赋值语句或输入(读)语句能修改一个变量的值。
24. 输出或访问表达式中一个变量的值不会修改该变量中的内容。
25. 在 C++ 中, >> 称为流析取运算符。
26. 标准输入设备的输入是通过 cin 和流析取运算符 >> 来完成的。
27. 当向程序中输入数据时, 数据项, 例如数字, 通常由空格, 行或 Tab 键分隔开。
28. 在 C++ 中, << 称为流插入运算符。
29. 标准输入设备的输出是通过 cout 和流插入运算符来完成的。
30. 在 ANSI/ISO 标准 C++ 中, 为了使用 cin 和 cout, 程序必须包含头文件 iostream 以及语句 “using namespace std;” 或指向这些标识符 std::cin 和 std::cout。
31. 操纵符 endl 将光标定位于下一行的开始。
32. 字符 \ 称为转义字符。
33. \n 称为换行符。
34. 所有预处理器命令都以 # 符号开始。
35. 预处理命令由预处理器在程序编译之前进行处理。
36. 预处理命令 #include<iostream> 指示预处理器在程序中包含 iostream 头文件。
37. 所有 C++ 语句都以分号结束。C++ 中的分号称为语句结束符。
38. C++ 系统有三个组成部分: 环境、语言 and 标准库。
39. 标准库不是 C++ 语言的一部分。它们包含了执行诸如算术操作的函数。
40. 包含 C++ 程序的文件通常以 .cpp 为扩展名。
41. 单行注释以符号 // 开始, 它可以在该行的任何位置。
42. 多行注释用 /* 和 */ 括起来。
43. 编译器会忽略注释。
44. 提示行是告诉用户去做什么的可执行语句。
45. 算术复合赋值运算符具有 op= 的形式, op 是任何一种算术运算符。
46. 对于算术运算符, 复合赋值语句有下面的形式:
variable op=expression;
它等价于语句:
variable=variable op(expression);
op 是任何一种算术运算符。

2.14 练习

1. 判断下面语句的对错。
 - a. 标识符可以是数字和字母的任意序列。
 - b. 在 C++ 中, 保留字和预定义标识符没有区别。
 - c. C++ 标识符可以以数字开头。

- d. 求余运算符的操作数必须是整数。
- e. 如果 $a=4$, $b=3$, 那么在执行语句 “ $a=b$;” 之后, b 的值仍为 3。
- f. 在语句 “ $\text{cin} >> y$;” 中, y 只能是 `int` 或 `double` 类型的变量。
- g. 在一条 `cout` 语句中, 换行符可以是字符串的一部分。
- h. 下面是一个合法的 C++ 程序:

```
int main()  
{  
    return 0;  
}
```

- i. 在一个混合表达式中, 所有的操作数都被转换为浮点数字。
 - j. 假定 $x=5$ 。在执行语句 “ $y=x++$;” 后, y 是 5, x 是 6。
 - k. 假定 $a=5$ 。在执行语句 “ $++a$;” 之后, a 的值仍然为 5, 因为表达式的值没有保存在另一个变量中。
2. 下面哪一项是正确的 C++ 标识符。
- a. RS6S6
 - b. MIX-UP
 - c. STOP!
 - d. exam1
 - e. September1Lecture
 - f. 2May
 - g. Mike's
 - h. First Exam
 - i. J
 - j. Three
3. 下面哪一项是 C++ 的保留字。
- a. `int`
 - b. `long`
 - c. `Char`
 - d. `CHAR`
 - e. `Float`
 - f. `Double`
4. 选择最好的答案。
- a. $15/2$ 的值为:
 - (i) 7 (ii) 7.5 (iii) $7\frac{1}{2}$ (iv) 0.75 (v) 以上全不是
 - b. $18/3$ 的值为:
 - (i) 6 (ii) 0.167 (iii) 6.0 (iv) 以上全不是
 - c. $22 \% 7$ 的值为:
 - (i) 3 (ii) 1 (iii) 3.142 (iv) $22/7$
 - d. $5 \% 7$ 的值为:
 - (i) 0 (ii) 2 (iii) 5 (iv) 未定义
 - e. $17.0/4$ 的值为:
 - (i) 4 (ii) 4.25 (iii) $4\frac{1}{4}$ (iv) 未定义

f. $5 - 3.0 + 2$ 的值为:

(i) 0 (ii) 0.0 (iii) 4 (iv) 4.0

g. $7 - 5 * 2$ 的值为:

(i) -2 (ii) 5 (iii) 6 (iv) 以上全不是

h. $15.0 / 3.0 + 2.0$ 的值为:

(i) 3 (ii) 3.0 (iii) 5 (iv) 以上全不是

5. $x=5, y=6, z=4, w=3.5$, 如果能计算值, 则给出结果, 否则说明原因。

a. $(x + z) \% y$

b. $(x + y) \% w$

c. $(y + w) \% x$

d. $(x + y) * w$

e. $(x \% y) \% z$

f. $(y \% z) \% x$

g. $(x * z) \% y$

h. $((x * y) * w) * z$

6. 假定:

```
int n, m, l;
```

```
double x, y;
```

那么下面的哪一项语句是有效的? 如果赋值无效, 说明原因。

a. $n = m = 5;$

b. $m = l = 2 * n;$

c. $n = 5; m = 2 + 6; n = 6 / 3;$

d. $m + n = 1;$

e. $x = 2 * n + 5.3;$

f. $l + 1 = n;$

g. $x / y = x * y;$

h. $m = n \% 1;$

i. $n = x \% 5;$

j. $x = x + 5;$

k. $n = 3 + 4.6$

7. 逐步求出 e 的值。假定所有变量已正确声明。

```
a = 3;
```

```
b = 4;
```

```
c = (a \% b) * 6;
```

```
d = c / b;
```

```
e = (a+b+c+d) / 4;
```

8. 下面哪一个变量声明是正确的? 如果变量声明不正确, 给出原因并且给出正确的变量声明。

```
n = 12; //Line 1
```

```
char letter = ; //Line 2
```

```
int one = 5, two; //Line 3
```

```
double x, y, z; //Line 4
```

9. 下面哪一项是有效的 C++ 赋值语句? 假定 x 和 $percent$ 是 $double$ 变量。

- a. $I = I + 5;$
- b. $x + 2 = x;$
- c. $x = 2.5 * x;$
- d. $\text{percent} = 10\%;$

10. 编写 C++ 语句完成下面的任务。

- a. 声明 int 变量 x,y。
- b. 初始化 int 变量 x 为 10, char 变量 ch 为 'B'。
- c. 通过向 int 变量 x 加 5 来更新 x 的值
- d. 设置 double 变量 z 的值为 25.3。
- e. 将 int 变量 y 的内容复制给 int 变量 z。
- f. 交换 int 变量 x 和 y 的值。(如果需要则可声明额外的变量。)
- g. 输出变量 x 和表达式 $2*x+5-y$, 其中 x,y 是 double 变量。
- h. 声明 char 变量 grade 并将 grade 的值设置为 'A'。
- i. 声明 int 变量来存储 4 个整数。
- j. 将最近似于 double 变量 z 的整数值赋值给 int 变量 x。

11. 编写下面每一个 C++ 表达式。

- a. -10 乘以 a
- b. 值为 '8' 的字符
- c. $(b^2-4ac)/2a$
- d. $(-b + (b^2-4ac))/2a$

12. 假定 x,y,z 和 w 是 int 变量。在最后一语句执行后每一个变量的值是什么?

```
x = 5; z = 3;
y = x - z;
z = 2 * y + 3;
w = x - 2 * y + z;
z = w - x;
w++;
```

13. 假定 x,y 和 z 是 int 变量, w 和 t 是 double 变量。在最后一语句后每一个变量的值是什么?

```
x = 17;
y = 15;
x = x + y / 4;
z = x % 3 + 4;
w = 17 / 3 + 6.5;
t = x / 4.0 + 15 % 4 - 3.5;
```

14. 假定 x,y 和 z 是 int 变量且 $x=2, y=5, z=6$ 。则下面每一项语句的输出是什么?

- a. `cout<<"x = "<<x<<" , y = "<<y<<" , z = "<<z<<endl;`
- b. `cout<<"x + y = "<<x + y<<endl;`
- c. `cout<<"Sum of "<<x<<" and "<<z<<" is "<<x+z<<endl;`
- d. `cout<<"z / x = "<<z / x<<endl;`
- e. `cout<<" 2 times "<<x<<" = "<<2*x<<endl;`

15. 下面语句的输出是什么? 假定 a 和 b 是 int 变量, c 是 double 变量。且 $a=13, b=5, c=17.5$ 。

- a. `cout<<a + b - c<<endl;`


```

b. cout<< 15 / 2 + c <<endl;
c. cout<<a / static_cast<double>(b) + 2 * c
   <<endl;
d. cout<< 14 % 3 + 6.3 + b / a<<endl;
e. cout<<static_cast<int>(c)%5 + a ~ b<<endl;
f. cout<< 13.5 / 2 + 4.0 * 3.5 + 18<<endl;

```

16. 怎样输出换行?

17. 下面哪一项是正确的 C++ 语句?

```

a. cout<<"Hello There!"<<endl;
b. cout<<"Hello";
   <<" There!"<<endl;
c. cout<<"Hello"
   <<" There!"<<endl;
d. cout<<'Hello There! '<<endl;

```

18. 下面两个程序有语法错误, 请改正其中的错误。在每一行后都假定前面的错误已经被改正了。

a.

```

#include <iostream>
const int prime=11,213;
const rate = 15.6
int main ()
{
    int i, x, y, w;
    x = 7;
    y = 3;
    x = x + w;
    prime = x + prime;
    cout<<prime<<endl;
    wages = rate * 36.75;
    cout<<"Wages = "<<wages<<endl;
    return 0;
}

```

b.

```

const char = Blank = ' ';
const int one 5;

int main (
{
    int a, b, cc;

    a = one + 5;
    b = a + blank;
    cc := a + one * 2;
    a + cc = b;
    one = b + c;
}

```

```

    cout<<"a = "<<a<<"", b = "<<b<<"", cc = "<<cc<<endl;

    return 0;
}

```

19. 如果可能, 写出下面的复合语句。

```

a. x = 2 * x
b. x = x + y - 2;
c. sum = sum + num;
d. z = z * x + 2 * z;
e. y = y / (x + 5);

```

20. 将下面的复合语句写成等效的简单语句。

```

a. x += 5 - z;
b. y *= 2 * x + 5 - z;
c. w += 2 * z + 4;
d. x -= z + y - t;
e. sum += num;

```

21. 假定 a, b 和 c 是 int 变量且 a=5, b=6。在每一条语句执行后, 每一个变量的值是什么? 如果变量在一个特定的语句没有定义, 显示 UND (undefined)。

	a	b	c
a = (b++) + 3;	_____	_____	_____
c = 2 * a + (++b);	_____	_____	_____
b = 2 * (++c) - (a++);	_____	_____	_____

22. 假定 a, b 和 sum 是 int 变量, c 是 double 变量。在每一条语句执行后, 每一个变量的值是什么? 假定 a=3, b=5, c=14.1。

	a	b	c	sum
sum = a + b + c;	_____	_____	_____	_____
c /= a;	_____	_____	_____	_____
b += c - a;	_____	_____	_____	_____
a *= 2 * b + c;	_____	_____	_____	_____

23. 下面的程序会打印什么结果? 假定输入是:

```

20 15
#include <iostream>
using namespace std;
const int NUM = 10;
const double x = 20.5;
int main()
{
    int a, b;
    double z;
    char grade;

    a = 25;
    cout<<"a = "<<a<<endl;
    cout<<"Enter two integers : ";

```

```

    cin>>a>>b;
    cout<<endl;
    cout<<"The numbers you entered are "
        <<a<<" and "<<b<<endl;
    z = x + 2 * a - b;
    cout<<"z = "<<z<<endl;
    grade = 'A';
    cout<<"Your grade is "<<grade<<endl;

    a = 2 * NUM + z;
    cout<<"The value of a = "<<a<<endl;
    return 0;
}

```

24. 下面的程序需要什么类型的输入？输入应该按照什么顺序？

```

#include <iostream>
using namespace std;
int main()
{
    int x,y;
    char ch;
    cin>>x;
    cin>>ch>>y;
    return 0;
}

```

2.15 编程练习

1. 编写一个产生下面结果的程序：

```

*****
*   Programming Assignment 1   *
*   Computer Programming I     *
*   Author: Duffy Ducky       *
*   Due Date: Thursday, Jan. 24 *
*****

```

2. 编写一个产生下面结果的程序：

```

CCCCCCCCC      ++      ++
CC              ++      ++
CC      ++++++      ++++++
CC      ++++++      ++++++
CC              ++      ++
CCCCCCCCC      ++      ++

```

3. 编写一个程序，用户输入一个小数，输出是一个四舍五入后的整数。
4. 编写一个程序，输入矩形的长和宽，然后打印矩形的面积和周长。
5. 编写一个程序，用户输入5个测试得分，然后打印平均分。

6. 编写一个程序打印下面的形式。

```
*
* *
* * *
```

7. 编写一个程序打印下面的标题。

```
*****
*****
***** WELCOME *****
=====
=====
***** HOME *****
*****
*****
```

8. 编写一个程序，提示用户输入5个小数。然后程序将这5个分数相加，把结果转换成最近似的整数并打印结果。

9. 编写一个程序完成下面的要求：

- 提示用户输入5个小数。
- 打印这5个小数。
- 将每一个小数转换为最近似的整数。
- 将这5个数字相加。
- 打印和以及5个整数的平均值。

10. 编写一个程序，提示用户输入一个用厘米表示的长度。然后程序将长度转换为英寸，并且输出用yard（码），feet（英尺）和inch（英寸）依次表示的长度。例如，123英寸应该输出：

3 yard(s), 1 feet (foot), and 3 inch(es)

下列为错误输出：

2 yard(s) 4 feet (foot) 3 inch(es),

10 feet (foot) 3 inch(es),

10.25 feet (foot).

第3章 输入/输出

本章要点:

- 学习什么是流以及输入和输出流
- 学习怎样从标准输入设备读取数据
- 学习怎样在程序中使用预定义函数
- 学习怎样使用输入流函数 `get()`, `ignore()`, `fill()`, `putback()` 以及 `peek()`
- 了解输入错误
- 学习怎样向标准输出设备写数据
- 学习怎样在程序中使用操纵符来确定输出格式
- 了解怎样对字符串数据类型执行输入和输出操作
- 了解文件输入和输出

在第2章中,介绍了一些C++的输入输出(I/O)指令,这些指令可以把数据输入到程序中并将结果打印在屏幕上。可以使用 `cin` 和析取运算符 `>>` 从键盘得到数据,使用 `cout` 和插入运算符向屏幕发送输出。因为I/O操作对于任何编程语言都是很基础的,所以本章将会详细地讲述C++的I/O操作。首先,将会学习从标准输入设备取得输入的函数以及向标准输出设备发送输出的函数。读者将会学习怎样使用操纵符确定输出的格式。另外,还会了解与标准输入输出设备相关的运算符以及与其他设备联系的扩展运算符的限制。

3.1 I/O 数据流和标准 I/O 设备

程序执行三个基本的操作:得到数据并输入给程序,对数据进行处理,输出结果。在第2章中学习了怎样使用算术运算符对数值型数据进行操作。在后面的章节中,将会学习怎样处理非数值型数据。因为编写I/O程序是很复杂的,所以C++通过丰富的I/O操作从而实现了其可扩展性。在本章中,将会学习多种I/O操作,这样可以提供程序的自由度。

在C++中,I/O是一系列由源到目的字节流,字节通常是字符,除非程序需要其他类型的信息,例如图形和数字语音。所以,数据流是一系列从源到目的字符。有两种类型的数据流:输入数据流和输出数据流。

输入数据流:从输入设备到计算机的序列字符。

输出数据流:从计算机到输出设备的序列字符。

前面讲过,标准输入设备通常是键盘,标准输出设备通常是屏幕。为了从键盘接收数据并输出到屏幕,每一个C++程序必须使用头文件 `iostream`。该头文件包含了两种数据类型的定义: `istream` (输入数据流) 和 `ostream` (输出数据流)。同时还声明了两个变量,一个是 `cin`,它代表输入(发音为 see-in);另一个是 `cout` (发音为 see-out),它代表输出。这些变量声明与下面的C++语句相似:

```
istream cin;  
ostream cout;
```

为了使用 `cin` 和 `cout`，每一个 C++ 程序必须包含预处理器指令：

```
#include<iostream>
```

注意：在第2章中，除了包含头文件 `iostream` 使用 `cin` 和 `cout` 外，还一直使用语句 “`using namespace std;`”。否则可以使用标识符 `std::cin` 和 `std::cout`。在第8章中，将会详细地讲述语句 “`using namespace std;`”。

`istream` 类型的变量称为输入数据流变量；`ostream` 类型的变量称为输出数据流变量。数据流变量既可以是输入数据流变量也可以是输出数据流变量。

因为 `cin` 和 `cout` 已经被定义并且有特定的含义，所以为了避免混淆，不允许在程序中对它们进行重新定义。

变量 `cin` 具有可访问的操作符及函数，从而可以从标准输入设备获取数据。可以使用简单的析取运算符 `>>` 从标准输入设备中输入数据。下一部分会详细地描述运算符 `>>` 是如何工作的。在下面的部分中，将会学习怎样使用函数 `get()`、`ignore()`、`peek()` 和 `putback()` 以一种特定的方式来输入数据。

3.1.1 `cin` 和析取运算符 `>>`

在第2章中，已经学习讲过怎样使用 `cin` 和析取运算符从标准输入设备输入数据。考虑下面的 C++ 语句：

```
cin>>payRate;
```

当计算机执行该语句时，它输入从键盘输入的一个数字并且将该数字存储到 `payRate` 变量中。所以，如果 `payRate` 是 `double` 数据类型并且用户输入的值是 15.50，那么在语句执行后 `payRate` 中存储的值是 15.50。

析取运算符 `>>` 是一个二元操作符，所以必须有两个操作数。左操作数必须是一个输入数据流变量，例如 `cin`。因为输入语句的目的是读值并且在内存中存储该值，所以右操作数是变量。

注意：析取运算符 `>>` 是为了向简单数据类型输入数据而定义的。所以，析取运算符右边的操作数是简单数据类型的变量。然而，C++ 允许程序员扩展析取运算符的定义，这样通过一条输入语句，数据可以放入其他数据类型的变量中。第16章将会讲述这种机制。

使用 `cin` 和析取运算符 `>>` 输入语句的语法是：

```
cin>>variable>>variable.变量;
```

从上面的语句中可以看出，`cin` 语句可以通过使用几次运算符 `>>` 来读入不止一个数据。每一个 `>>` 的出现都会从输入数据流中提取下一个数据项。例如，可以通过 `cin` 语句使用下面的代码来读入 `payRate` 和 `hoursWorked`。

```
cin>>payRate>>hoursWorked;
```

前一条 `cin` 语句和后两条 `cin` 语句之间没有区别。使用哪种 `cin` 语句只是一种习惯和风格。

```
cin>>payRate;  
cin>>hoursWorked;
```

析取运算符是怎样工作的呢？当检测到下一个输入时，`>>` 会跳过所有的空白字符。空白字符包含空格和特定的非打印字符，例如 Tab 键和换行符。这样，无论使用换行符还是空格，析取运算符 `>>` 都会很容易地在输入数据流中找到下一个输入数据。例如，假定 `payRate` 和 `hoursWorked` 是 `double` 数据类型的变量。考虑下面的输入语句：


```
cin>>payRate>>hoursWorked;
```

无论输入:

```
15.50 48.30
```

或:

```
15.50    48.30
```

还是:

```
15.50
48.30
```

48.30 前面的输入语句都会将 15.50 存储到 payRate 中, 而将 48.30 存储到 hoursWorked 中。注意第一项输入是使用空格隔开的, 第二项输入是用 Tab 键隔开的, 第三项输入是用行隔开的。

现在假设输入是 2。析取运算符怎样区别是字符 2 还是数字 2 呢? 析取运算符右边的操作数可以区分开来。如果右边的操作数是 char 数据类型的变量, 那么输入 2 就会被当做字符 2, 在这种情况下, 2 的 ASCII 值被保存。如果右边的操作数是 int 或 double 数据类型的变量, 那么输入 2 就会被当做数字 2。

再来考虑输入 25 时的情况:

```
cin>>a;
```

这里 a 是一些简单数据类型的变量。如果 a 是 char 数据类型, 那么只有单独的字符 2 被存储到 a 中。如果 a 是 double 数据类型, 输入 25 就会被转换为小数 25.0。表 3.1 通过显示简单数据类型变量的有效输入归纳了这些讨论。

表 3.1 简单数据类型变量的有效输入

数据类型	有效输入
char	可打印字符
int	整数, 前面可能有 ± 号
double	小数数字, 前面可能有 ± 号。如果实际输入是整数, 那么输入转换为小数点后补零的小数数字

当向一个 char 变量输入数据时, 在跳过前面的空格后, 析取运算符会找到并且只存储下一个字符, 在读入一个单字符后会停止输入。为了向 int 或 double 变量读入数据, 在跳过所有的空格并且读入正负号后 (如果有的话), 析取运算符 >> 会读入输入的数字, 包括浮点型变量的小数点, 当发现空格或者除了数字之外的字符时读入就会停止。

例 3.1

假设下面的变量声明:

```
int a,b;
double z;
char ch,ch1,ch2;
```

下面的语句显示了析取运算符是怎样工作的。

	语句	输入	内存中存储的值
1	cin>>ch;	A	ch='A'
2	cin>>ch;	AB	ch='A', 'B' 被缓存
3	cin>>a;	48	a=48
4	cin>>a;	46.35	a=46.35 被缓存
5	cin>>z;	74.35	z=74.35

6	<code>cin>>z;</code>	39	<code>z=39.0</code>
7	<code>cin>>z>>a;</code>	65.78 38	<code>z=65.78, a=38</code>
8	<code>cin>>a>>b;</code>	4 60	<code>a=4, b=60</code>
9	<code>cin>>a>>ch>>z;</code>	57 A 26.9	<code>a=57, ch='A', z=26.9</code>
10	<code>cin>>a>>ch>>z;</code>	57 A 26.9	<code>a=57, ch='A', z=26.9</code>
11	<code>cin>>a>>ch>>z;</code>	57 A 26.9	<code>a=57, ch='A', z=26.9</code>
12	<code>cin>>a>>ch>>z;</code>	57A26.9	<code>a=57, ch='A', z=26.9</code>
13	<code>cin>>z>>ch>>a;</code>	36.78B34	<code>z=36.78, ch='B', a=34</code>
14	<code>cin>>z>>ch>>a;</code>	36.78 B34	<code>z=36.78, ch='B', a=34</code>
15	<code>cin>>a>>b>>z;</code>	11 34	<code>a=11, b=34, 计算机等待输入下一数字</code>
16	<code>cin>>a>>z;</code>	46 32.4 68	<code>a=46, z=32.4, 68 被缓存</code>
17	<code>cin>>ch>>a;</code>	256	<code>ch='2', a=56</code>
18	<code>cin>>a>>ch;</code>	256	<code>a=256, 计算机等待输入 ch 值</code>
19	<code>cin>>ch1>>ch2;</code>	A B	<code>ch1='A', ch2='B'</code>

在语句1中,析取运算符从输入数据流提取并且存储字符'A'并将它存储到变量ch中。在语句2中,析取运算符>>从输入数据流提取字符'A'并且把它存储到变量ch中,值'B'保存在输入数据流中以便后面的输入。

同样在语句16中,值68被保存以备后面的输入。在语句15中,11存储在a中,34存储在b中,但是输入数据流没有足够多的输入数据去填充每一个变量。在这种情况下,计算机等待下一个输入。直到下一个值被键入,计算机才会继续执行。

在语句4中,析取运算符>>从输入数据流中提取46并且将其保存到a中。注意,因为a是int变量而且在46之后是“.”(它是非数值类型),所以只有46从数据流中被提取。这样,读操作停在“.”处,35被保存以备后面输入。在语句5中,析取运算符>>从输入数据流中提取74.35并且将该值保存到z中。

在语句6中,z是double数据类型的变量,输入39是整数。所以,39被转换为小数39.0并且保存到z中。

在语句7中,析取运算符>>右边的第一个操作数是double变量,第二个操作数是int变量。所以在语句7中,首先65.78从输入数据流中提取并且存储在z中。然后析取运算符跳过65.78后面的空格,值38被提取并且存储在a中。语句8的原理类似。

注意,从语句9到语句12,输入语句都是相同的,但是数据的输入是不同的。对于语句9,数据在同一行,只是用空格分开。对于语句10,数据分两行输入,前两项输入使用两个空格隔开,第三项输入在下一行。对于语句11,所有的三项输入值都是由行隔开的。而在语句12中,所有三项输入值都在同一行,但是彼此之间没有空格。注意第二项输入是非数值字符。这些语句的运行机制如下:

在语句9中,首先析取运算符>>从输入数据流提取57并且保存在a中,然后跳过57后面的空格,提取字符A并且存储到ch中。接着析取运算符>>跳过A后面的空格,从输入数据流中提取并存储值26.9。

在语句10中,析取运算符>>首先从输入数据流提取57并且保存在a中,然后跳过57后面的两个空格,从输入数据流提取字符A并且存储到ch中。接着析取运算符>>跳过A后面的换行符,从输入数据流中提取值26.9并存储到z中。

在语句11中,析取运算符>>首先从输入数据流提取57并且保存在a中,然后跳过57后面的换

行符，从输入数据流提取字符 A 并且存储到 ch 中。接着析取运算符 >> 跳过 A 后面的换行符，从输入数据流中提取值 26.9 并存储到 z 中。

语句 9, 10 和 11 表明，尽管输入被空格或行分隔开，析取运算符 >> 总是能找到下一项输入。

在语句 12 中，析取运算符 >> 首先从输入数据流提取 57 并且保存在 a 中，然后从输入数据流提取字符 A 并且存储到 ch 中。接着提取 26.9 并存储到 z 中。

在语句 13 中，因为 >> 右边的第一个操作数是 z，它是 double 变量，所以 36.78 从输入数据流中提取并且保存到 z 中。接着 B 被提取并被保存到 ch 中。最后 34 被提取并被保存到 a 中。语句 14 的原理与此类似。

在语句 17 中，析取运算符 >> 右边第一个操作数是 char 变量，所以第一个非空白字符 2 从输入数据流中提取并保存在变量 ch 中。析取运算符的下一个操作数是 int 变量，所以值 56 被提取并被保存到 a 中。

在语句 18 中，析取运算符 >> 右边第一个操作数是 int 变量，所以第一个数据项 256 从输入数据流中提取并保存到 a 中。现在计算机为变量 ch 等待下一个数据项。

在语句 19 中，A 存储在 ch1 中，然后析取运算符 >> 跳过空格，B 被保存到 ch2 中。

注意：前面讲过，在程序执行期间，当输入如字母之类的字符时，不应该用单引号把字符括起来。

如果输入数据流比程序需要的数据项多，会发生什么情况呢？在程序结束之后，输入数据流中任何剩余的值都会被忽略。当输入数据进行处理时，数据值应该与输入数据中变量的类型值保持一致。前面讲过，当输入一个 double 变量的数字时，不必一定要输入一个小数数字。如果输入数字是一个整数，没有小数部分，那么它会转换为一个小数值。然而计算机不能接受任何其他不匹配的类型。例如，向 int 或 double 变量输入一个 char 类型，会产生严重的错误，称为输入错误。输入错误会在后面的章节中进行讨论。

当尝试向一个 int 变量读入一个非数值型字符时，会发生什么情况呢？例 3.5（本章后面给出）显示了这种情况。

当在输入数据流中检测到下一个输入时，析取运算符跳过空白字符（例如空格和换行符）。然而，当这些字符必须被存储并且处理时，例如，如果用逐行的方法来处理文本时，必须知道输入数据流中换行符在什么地方。如果鉴别不出换行符的位置，程序就不会知道一行在哪里结束，另一行在哪里开始。下面几个部分将会介绍怎样使用输入函数（例如 get(), ignore(), putback() 和 peek()）向一个程序中输入数据。这些函数与 istream 数据类型相联系并且称为 istream 成员函数。I/O 函数（例如 get()）一般称为 stream 成员函数或者 stream 函数。

在学习本章使用的 get(), ignore(), putback(), peek() 和其他 I/O 函数之前，需要知道函数是什么以及它是怎样工作的。

在第 6 章和第 7 章中，将会详细地讨论函数并学习怎样编写自己的函数。

3.2 在程序中使用预定义函数

在第 2 章中讲过，函数（也称为子程序）是指令的集合。当一个函数执行时，它会完成一些任务。当程序运行时，main 函数()会自动执行。其他函数只有被激活也就是被调用时才会执行。C++ 提供了大量的预定义函数，即它们已经被定义了。在下面的部分中，将会学习怎样使用数据流函数来执行一个特定的 I/O 操作。

在本部分中，将不会学习怎样编写自定义的函数，但是将学习如何使用作为 C++ 系统一部分的预定义函数。

在第2章中讲过,预定义函数是作为一个集合的库被提供的,这称为头文件。一个特定的头文件可能包含几个函数。所以,为了使用一个特定的函数,需要知道函数的名字以及其他一些简要信息。

`pow()`函数是一个很有用的函数,称为幂函数。它可以计算程序中的 x^y 值。即 $\text{pow}(x,y)=x^y$ 。例如 $\text{pow}(2,3)=2^3=8$, $\text{pow}(4,0.5)=4^{0.5}=2$ 。`pow()`函数中使用的数字 x 和 y 称为`pow()`函数的参数。例如,在函数`pow(2,3)`中,参数是2和3。

形如`pow(2,3)`的表达式称为函数调用,在这种情况下,它与函数`pow()`相联系的代码被执行并计算 2^3 。`cmath`头文件中包含了`pow()`函数的声明。

为了在程序中使用预定义函数,需要知道包含该特定函数的头文件名字,同时要在程序中包含该头文件。另外还要知道函数的名字、函数所包含的参数数量以及每个参数的类型。同时应该了解该函数的功能。例如,为了使用函数`pow()`,必须包含头文件`cmath`。函数`pow()`有两个参数,每一个参数都是数值类型。函数计算第一个参数的第二个参数次幂。

因为I/O对于任何一种编程语言都是很基本的,同时编写指令执行特定的I/O操作不是每个人都要做的工作,所以每一种编程语言都提供了一组有用的函数来执行特定的I/O操作。在本章的剩余部分,将会学习在程序中怎样使用这一类函数。作为一名程序员,必须仔细注意这些函数是怎样使用的,这样才能充分地利用这些函数。这里,读者将要学到的第一个函数是`get()`函数。

3.2.1 `cin` 和 `get()` 函数

可以看到,当检测到下一个输入值时,析取运算符会跳过所有前面的空白字符。考虑变量声明:

```
char ch1, ch2;  
int num;
```

和输入:

A 25

现在考虑下面的语句:

```
cin >> ch1 >> ch2 >> num;
```

当计算机执行该语句时, 'A' 存储到 `ch1` 中, 然后析取运算符跳过空格, 字符 '2' 保存到 `ch2` 中, 5 被保存到 `num` 中。然而, 如果试图把 'A' 存储到 `ch1` 中, 空格存储到 `ch2` 中, 25 存储到 `num` 中时, 应该怎么办呢? 很明显, 不能使用析取运算符 `>>` 来输入数据。

前面讲过, 有时需要处理完整的输入, 包括空白字符, 比如空格和换行符。例如, 假想逐行地处理完整的数据。因为析取运算符跳过换行符, 所以除非程序能捕获换行符, 否则计算机不能知道一行在哪里结束, 下一行在哪里开始。

变量 `cin` 能访问流函数 `get()`, `get()` 用来读取字符数据。`get()` 函数从输入数据流中输入下一个字符 (包括空格) 并保存在其参数所指示的内存位置。函数 `get()` 有很多种形式。下面讨论一种用来读入字符的 `get()` 函数。

`cin` 与 `get()` 函数联合使用的语法如下:

```
cin.get(varChar);
```

在 `cin.get()` 语句中, `varChar` 是一个 `char` 变量。`varChar` 出现在函数后面的参数中, 称为实参。前面语句的结果将会存储到 `varChar` 变量中。

考虑下面的输入:

A 25

为了将 'A' 存储到 ch1, 空格存储到 ch2, 25 存储到 num 中, 可以有效地使用下面的 get() 函数:

```
cin.get(ch1);  
cin.get(ch2);  
cin>>num;
```

因为 get() 函数的这种形式只有一个参数而且每次只读入一个字符, 所以需要从输入数据流中读入两个字符, 也就是需要调用两次该函数。注意到不能使用 get() 函数将数据读入到变量 num 中, 因为 num 是一个 int 变量。get() 函数前面的形式只读入 char 数据类型的变量。

前面的一组 cin.get() 语句等价于下面的语句:

```
cin>>ch1;  
cin.get(ch2);  
cin>>num;
```

注意: 函数 get() 还有其他的形式, 其中的一种将在第 9 章学到。在下面几章中, 只需要使用这里所介绍的 get() 函数形式。

3.2.2 cin 和 ignore() 函数

当想处理部分数据时 (比如一行内的数据), 可以使用数据流函数 ignore() 来忽略输入的一部分。使用 ignore() 函数的语法是:

```
cin.ignore(intExp, chExp);
```

这里, intExp 是一个产生整数值的整形表达式, chExp 是一个产生 char 值的 char 表达式。实际上, 表达式 intExp 详细规定了在一行中被忽略字符的最大数量。

假设 intExp 产生一个值, 例如为 100, 该语句就会忽略下 100 个字符, 或者忽略输入数据直到它遇到由 chExp 指定的首个字符。为了详细地说明, 考虑下面的语句:

```
cin.ignore(100, '\n');
```

当该语句执行时, 它忽略下 100 个字符或者所有的字符, 直到遇到第一个换行符。例如, 如果 120 个字符中不包含换行符, 那么只有前 100 个字符被忽略, 下一个输入数据是第 101 个字符。然而, 如果第 75 个字符是换行符, 那么前 75 个字符被忽略掉, 下一个字符是第 76 个字符。同样, 执行下面的语句:

```
cin.ignore(100, 'A');
```

结果会忽略前 100 个字符或者所有字符, 直到遇到第一个字符 'A'。

例 3.2

考虑声明:

```
int a, b;
```

以及输入:

```
25 67 89 43 72  
12 78 34
```

考虑下面的语句:

```
cin>>a;
```

```
cin.ignore(100, '\n');  
cin>>b;
```

第一行语句“cin>>a;”将25存储到a中，第二行语句“cin.ignore(100, '\n');”忽略第一行剩余的字符。第三行语句“cin>>b;”将12（下一行）保存到b中。

例 3.3

考虑声明：

```
char ch1, ch2;
```

和输入：

```
Hello there. My name is Mickey.
```

现在考虑下面的语句：

```
cin>>ch1;  
cin.ignore(100, '.');  
cin>>ch2;
```

第一行语句“cin>>ch1;”将'H'保存到ch1中，第二行语句“cin.ignore(100, '.');”忽略所有字符直到遇到“.”号。第三行语句“cin>>ch2;”将字符'M'（相同行）存储到ch2中。（析取运算符>>跳过所有前面的空白字符。这样提取操作符跳过“.”后面的空格并且将'M'保存到ch2中。）

3.2.3 putback()和peek()函数

假设正在处理数字与字符混合的数据，而且数字必须作为数值被读入并进行处理。同时，程序还包含了很多组的取样数据而且不能决定先输入一个字符还是数字。可以逐字符地读入整个数据集，并且检查特定的字符是否是数字。如果找到数字，就可以读入数字的剩余部分并且将这些字符转化为数字。因此，程序代码会有点复杂，幸运的是，C++提供了两个非常有用的数据流函数，可以有效地用于这些情况。

数据流函数putback()允许把通过get()函数从输入数据流提取的数据重新放回到输入数据流中。数据流函数peek()检查输入数据流并且告知下一个字符是什么，而不必把它从输入数据流中移除。通过使用这些函数，在确定下一个输入是数字后，可以把它作为一个数字来读取，而不必把数字的每一位读做字符并将这些字符转化为数字。

使用函数putback()的语法为：

```
istreamVar.putback(ch);
```

这里istreamVar是一个输入数据流变量，例如cin。ch是一个char变量。

peek()函数从输入数据流中返回下一个字符，但是并不会从数据流中将其移除。换言之，函数peek()检查输入数据流并且核对下一个输入字符的有效性。而且，在检查输入数据流中的下一个输入字符后，它能在一个指定的内存位置存储该字符，而不会将它从输入数据流中删除。也就是当使用peek()函数时，下一个输入字符不会被移除，即使已经查看过该字符。

peek()函数的语法是：

```
ch=istreamVar.peek();
```

这里istreamVar是一个输入数据流变量，例如cin。而ch是一个char变量。

注意函数peek()是怎样被使用的。首先，函数peek()用于一条赋值语句中。它不像get(), ignore()和putback()函数是一条单独的语句。第二，peek()函数不含有参数。在熟悉使用一个函数和学会

怎样编写函数之前，应该仔细注意怎样使用一个预定义的函数。

下面的例子演示了怎样使用 `peek()` 和 `putback()` 函数。

例 3.4

```
//Functions peek and putback

#include <iostream>
using namespace std;

int main()
{
    char ch;
    cout<<"Line 1: Enter a string: ";           //Line 1
    cin.get(ch);                                //Line 2
    cout<<endl;                                //Line 3
    cout<<"Line 4: After first cin.get(ch); "
        <<"ch = "<<ch<<endl;                    //Line 4

    cin.get(ch);                                //Line 5
    cout<<"Line 6: After second cin.get(ch); "
        <<"ch = "<<ch<<endl;                    //Line 6

    cin.putback(ch);                            //Line 7
    cin.get(ch);                                //Line 8
    cout<<"Line 9: After putback and then "
        <<"cin.get(ch); ch = "<<ch<<endl;        //Line 9

    ch = cin.peek();                            //Line 10
    cout<<"Line 11: After cin.peek(); ch = "
        <<ch<<endl;                            //Line 11

    cin.get(ch);                                //Line 12
    cout<<"Line 13: After cin.get(ch); ch = "
        <<ch<<endl;                            //Line 13
    return 0;
}
```

运行实例：在本运行实例中，用户输入用阴影表示。

```
Line 1: Enter a string: abcd
Line 4: After first cin.get(ch); ch = a
Line 6: After second cin.get(ch); ch = b
Line 9: After putback and then cin.get(ch); ch = b
Line 11: After cin.peek(); ch = c
Line 13: After cin.get(ch); ch = c
```

用户输入 `abcd`，第 1 行语句提示用户输入一个字符串。第 2 行语句“`cin.get(ch);`”从输入数据流提取第一个字符并且存储在变量 `ch` 中。所以在第 2 行执行之后，`ch` 的值是 `'a'`。

第 4 行的 `cout` 语句输出 `ch` 的值。第 5 行语句“`cin.get(ch);`”从输入数据流提取下一个字符 `'b'` 并且将它存储在 `ch` 中。这时 `ch` 的值是 `'b'`。

第 6 行的 `cout` 语句输出 `ch` 的值。第 7 行语句将前面由 `get()` 函数提取的字符 `'b'` 重新放回到输入数据流中。所以，从输入数据流中提取的下一个字符是 `'b'`。

第 8 行语句“`cin.get(ch);`”从输入数据流提取下一个字符 `'b'` 并且将它存储在 `ch` 中。这时 `ch` 的值是 `'b'`。第 9 行的 `cout` 语句输出 `ch` 的值 `'b'`。

在第 10 行，语句“`ch=cin.peek();`”检查输入数据流中的下一个字符 `'c'` 并且将它保存在 `ch` 中。`ch` 的值现在是 `'c'`。第 11 行的 `cout` 语句输出 `ch` 的值。第 12 行语句“`cin.get(ch);`”从输入数据流提取

下一个字符并且将它存储在 `ch` 中。第 13 行的 `cout` 语句输出 `ch` 的值，它仍然是 'c'。

注意，第 10 行语句 “`ch=cin.peek();`” 没有从输入数据流中删除字符 'c'，它只是查看输入数据流。第 11 行和第 13 行展示了这种功能。

3.2.4 I/O 数据流变量和 I/O 函数之间的点号

在前面的部分中，学习了怎样操纵输入数据流向程序写入数据。也学习了怎样使用 `get()`，`ignore()`，`peek()` 和 `putback()` 函数。

按照所显示的例子来使用这些函数是很重要的。例如，为了使用 `get()` 函数，显示的语句文本如下：

```
cin.get(ch);
```

忽略 “.” 号，也就是在变量 `cin` 和函数名字 `get()` 之间的 “.” 号，将会导致语法错误。例如语句：

```
cin.get(ch);
```

`cin` 和 `get` 是使用 “.” 隔开的两个独立的标识符。在语句：

```
cinget(ch);
```

中，`cinget()` 成为一个新的标识符。如果在一个程序中使用 “`cinget(ch);`”，编译器将会试图解析一个没有声明的标识符，这将会产生一个错误。同样，漏掉括号，例如在 “`cin.getch;`” 中，将会导致一个语法错误。记住，必须将输入函数与输入数据流变量一起使用。如果试图单独使用任何一个输入函数，即没有输入数据流变量，编译器可能会产生一条错误信息 “undeclared identifier”。例如，语句 “`get(ch);`” 将会导致一个语法错误。

正如所看到的，几个函数与一个输入数据流变量结合使用时，每一个函数具有特定的功能。前面讲过函数 `get()`，`ignore()`，等等是 `istream` 数据类型的成员，“.” 把输入数据流变量与成员或函数、名字分隔开来。实际在 C++ 中，“.” 是一个称为成员访问运算符的变量。

注意：C++ 中，数据类型 `istream` 和 `ostream` 有特殊的名字。数据类型 `istream` 和 `ostream` 称为类。`cin` 和 `cout` 也有特殊的名字，称为对象。所以，`cin` 称为输入数据流对象，`cout` 称为输出数据流对象。实际上，流变量称为数据流对象。在第 13 章中将会学到这些概念。

3.3 输入失败

程序执行期间可能发生很多错误。语法正确的程序可能产生不正确的结果，例如，假设一个兼职雇员的薪金可以使用下面的公式进行计算：

```
wages=payRate*hoursWorked;
```

如果不小心用 `+` 代替了 `*`，那么计算的薪水可能是不正确的，虽然包含 `+` 操作符的语句在语法上是正确的。

试图读入一个无效数据会发生什么结果呢？例如，将一个字母输入给一个 `int` 变量。如果输入数据与相应的变量不匹配，程序就可能出现错误。例如，试图向一个 `int` 或 `double` 变量读入一个字母时，将会产生一个输入错误。考虑下面的语句：

```
int a,b,c;  
double x;
```

如果输入是：

w 54

那么语句:

```
cin>>a>>b;
```

将会导致一个输入错误, 因为试图向一个 int 变量 a 输入一个字符 'w'。如果输入是:

```
35 67.93 48 78
```

那么输入语句:

```
cin>>a>>x>>b;
```

会将 35 保存到 a 中, 67.93 保存到 x 中, 48 保存到 b 中。

现在与前面的输入 (有三个值的输入) 结合起来, 考虑下面的输入语句:

```
cin>>a>>b>>c;
```

该语句将 35 保存到 a 中, 将 67 保存到 b 中。读入在 "." (小数点) 停止。因为下一个变量是 int 数据类型, 所以计算机试图将 "." 读给 c, 这会产生一个错误。然后输入数据流进入了一个称为 fail 的状态。

当输入数据流进入 fail 状态到底会发生什么呢? 一旦输入数据流进入 fail 状态, 所有使用该数据流后面的 I/O 语句都将被忽略。但是程序还会平缓地继续进行, 而不管变量中存储的是什么值, 这样就产生了不正确的结果。在例 3.5 的程序中演示了输入错误。用户系统中的程序可能产生不同的结果。

例 3.5

```
//Input Failure program
#include <iostream>
using namespace std;

int main()
{
    int a = 10; //Line 1
    int b = 20; //Line 2
    int c = 30; //Line 3
    int d = 40; //Line 4

    cout<<"Line 5: Enter four integers: "; //Line 5
    cin>>a>>b>>c>>d; //Line 6
    cout<<endl; //Line 7
    cout<<"Line 8: The numbers you entered are:"
        <<endl; //Line 8
    cout<<"Line 9: a = "<<a<<", b = "<<b
        <<"", c = "<<c<<", d = "<<d<<endl; //Line 9
    return 0;
}
```

运行实例 在本运行实例中, 用户输入用阴影表示。

运行实例 1:

```
Line 5: Enter four integers: 34 K 67 28
Line 8: The numbers you entered are:
Line 9: a = 34, b = 20, c = 30, d = 40
```

第 1, 2, 3, 4 行语句声明并且分别初始化 a,b,c,d 的值为 10, 20, 30 和 40。第 5 行语句提示用户输入 4 个整数, 第 6 行语句将这 4 个整数输入给变量 a,b,c 和 d。

在运行实例中，第二项输入值是字符'K'。cin语句试图将该字符输入到变量b中。然而，因为b是一个int变量，所以输入数据流进入fail状态。所以b,c和d的值没有改变，如第9行语句的输出所示。（如果使用标准C++风格的头文件运行该程序，在第9行会得到不同的b,c,d的值。）

运行实例2

```
Line 5: Enter four integers: 37 653.89 23 76
Line 8: The numbers you entered are:
Line 9: a = 37, b = 653, c = 30, d = 40
```

在示例程序运行后，第6行的cin语句将37存储到a中，653存储到b中，然后试图把小数点输入给c。因为c是一个int变量，而小数点被认为是一个字符，这样输入数据流进入fail状态。在运行实例中，c和d的值没有改变，如第9行语句的输出所示。（如果使用标准C++风格的头文件运行该程序，在第9行会得到不同的c和d的值。）

3.3.1 clear()函数

当输入数据流进入fail状态时，系统忽略所有使用该数据流后面的I/O输入。可以使用数据流函数clear()来恢复输入数据流进入工作状态。

使用clear()函数的语法是：

```
istreamVar.clear();
```

这里istreamVar是输入数据流变量，例如cin。

在使用函数clear()将数据流返回到工作状态之后，需要从数据流中清除剩余的无用数据。可以通过使用ignore()函数来完成。例3.6演示了这种情况。

例3.6

```
//Input failure and the clear function
#include <iostream>
using namespace std;

int main()
{
    int a = 23; //Line 1
    int b = 34; //Line 2

    cout<<"Line 3: Enter a number followed"
         <<" by a character: "; //Line 3
    cin>>a>>b; //Line 4
    cout<<endl<<"Line 5: a = "<<a
         <<" , b = "<<b<<endl; //Line 5

    cin.clear(); //Restore input stream; Line 6

    cin.ignore(200, '\n'); //Clear the buffer; Line 7
    cout<<"Line 8: Enter two numbers: "; //Line 8
    cin>>a>>b; //Line 9
    cout<<endl<<"Line 10: a = "<<a
         <<" , b = "<<b<<endl; //Line 10

    return 0;
}
```

运行实例：在本运行实例中，用户输入用阴影表示。

```
Line 3: Enter a number followed by a character: 23 d
```

```
Line 5: a = 78, b = 34
Line 8: Enter two numbers: 65 88
Line 10: a = 65, b = 88
```

第1, 2行语句声明并且分别初始化a,b的值为23和34。第3行语句提示用户在数字后面输入一个字符。第4行语句将数字输入给变量a, 然后试图将字符输入给变量b。因为b是一个int变量, 所以试图向b输入一个字符将会使输入数据流进入fail状态。b的值没有被改变, 如第5行语句的输出所示。

第6行语句通过使用clear()函数来恢复输入数据流, 第7行语句忽略输入数据剩余部分。第8行语句再一次提示用户输入两个数字, 第9行将这两个数字输入给a和b。接着, 第10行语句输出a和b的值。(如果使用标准C++风格的头文件运行该程序, 在第5行会得到不同的b值。)

3.4 输出和格式化输出

除了编写有效率的程序之外, 产生相应的输出也是程序员的主要的目标之一。第2章简要介绍了与标准输出设备有关的处理过程, 而且还详细地学习了怎样使用插入运算符和操纵符endl在标准输出设备上显示结果。

然而, 除了显示结果外, 输出还有很多功能。有时浮点型数字必须以一种特定的方式输出。例如, 薪水必须以保留2位小数的形式输出, 而科学实验的结果可能需要有6位、7位甚至可能是10位的浮点数输出。而且, 用户可能喜欢用特定的列来排列数字, 或者在字符串与数字之间用一个字符而非空格来填充。例如, 在输出报表内容的过程中, 可能需要在表头部分与页码之间的空白处填充逗号或分号。在这一部分中, 读者将学习不同的输出函数和运算符, 它们允许用户以需求方式来格式化输出。

前面学过, cout与插入运算符<<一起使用的语法是:

```
cout<<expression or manipulator<<expression or manipulator...;
```

其中expression是求值表达式, manipulator用来格式化输出。到目前为止使用的最简单的操纵符是endl, 它用来将光标移动到下一行的开始处。

其他我们感兴趣的操纵符包括: setprecision(), fixed, showpoint和setw()。下面的几个部分将阐述这些操纵符。

3.4.1 setprecision()

可以使用操纵符setprecision()来控制浮点数的输出。浮点数的默认输出是科学技术法。一些软件开发包(SDK)可能使用最多6位小数作为浮点数默认输出。然而, 当打印一名雇员的薪水时, 只希望输出两位小数。为了将浮点数输出到两个小数位, 可以使用setprecision()操纵符将精度设置为2。

通常setprecision()操纵符的语法是:

```
setprecision(n)
```

其中n是小数位数。

可以将setprecision()操纵符与cout和析取运算符一起使用。例如, 语句:

```
cout<<setprecision(2);
```

将小数格式化输出并保留两位小数, 直到下一条语句改变了精度。注意, 小数位数(精度值)作为一个参数被传送到setprecision()。

为了使用操纵符 `setprecision()`, 程序必须包含头文件 `iomanip`。因此, 下面包含的语句是必要的:

```
#include<iomanip>
```

3.4.2 fixed

为了进一步控制浮点数的输出, 可以使用其他的操纵符。为了以一个固定小数格式输出浮点型数字, 可以使用操纵符 `fixed`。下面的语句在标准输出设备上以一种固定的小数点格式设置浮点型数字的输出:

```
cout<<fixed;
```

在执行完该语句后, 所有浮点型数字以固定的小数格式显示, 直到操纵符 `fixed` 失效。可以通过使用流成员函数 `unsetf()` 使操纵符 `fixed` 失效。例如, 为了使操纵符在标准输出设备上失效, 可以使用下面的语句:

```
cout.unsetf(ios::fixed);
```

在操纵符 `fixed` 失效后, 浮点型数字的输出返回到默认的设置。操纵符 `scientific` 用来以科学计数方式输出浮点型数字。

3.4.3 showpoint

假定一个数的小数部分是零, 在这种情况下, 当指示计算机以一种固定小数位数输出小数时, 输出可能不会显示小数点和小数部分。为了强制输出显示小数点和后面的零, 可以使用操纵符 `showpoint`。下面的语句在标准输入设备上将显示小数点和补零后的小数:

```
cout<<showpoint;
```

当然, 下面的语句在标准输出设备上以一种固定的小数格式来设置一个有小数点和补零的浮点型数字的输出:

```
cout<<fixed<<showpoint;
```

例 3.7

下面的语句演示了如何使用操纵符 `setprecision()`, `fixed` 和 `showpoint`。

```
//Example: setprecision, fixed, showpoint
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double x,y,z;
    x = 15.674; //Line 1
    y = 235.73; //Line 2
    z = 9525.9864; //Line 3
    cout<<fixed<<showpoint; //Line 4
    cout<<setprecision(2)
        <<"Line 5: setprecision(2)"<<endl; //Line 5
    cout<<"Line 6: x = "<<x<<endl; //Line 6
    cout<<"Line 7: y = "<<y<<endl; //Line 7
```

```

    cout<<"Line 8: z = "<<z<<endl;           //Line 8

    cout<<setprecision(3)
        <<"Line 9: setprecision(3)"<<endl;     //Line 9
    cout<<"Line 10: x = "<<x<<endl;           //Line 10
    cout<<"Line 11: y = "<<y<<endl;           //Line 11
    cout<<"Line 12: z = "<<z<<endl;           //Line 12

    cout<<setprecision(4)
        <<"Line 13: setprecision(4)"<<endl;     //Line 13
    cout<<"Line 14: x = "<<x<<endl;           //Line 14
    cout<<"Line 15: y = "<<y<<endl;           //Line 15
    cout<<"Line 16: z = "<<z<<endl;           //Line 16

    cout<<"Line 17: "
        <<setprecision(3)<<x<<"  "
        <<setprecision(2)<<y<<"  "
        <<setprecision(4)<<z<<endl;           //Line 17

    return 0;
}

```

输出:

```

Line 5: setprecision(2)
Line 6: x = 15.67
Line 7: y = 235.73
Line 8: z = 9525.99
Line 9: setprecision(3)
Line 10: x = 15.674
Line 11: y = 235.730
Line 12: z = 9525.986
Line 13: setprecision(4)
Line 14: x = 15.6740
Line 15: y = 235.7300
Line 16: z = 9525.9864
Line 17: 15.674 235.73 9525.9864

```

第1, 2, 3行语句分别对 x, y 和 z 初始化为 15.674, 235.73 和 9525.9864。第4行语句以固定小数位的格式输出浮点数。第5行语句以两位小数格式输出浮点数。

第6, 7和8行以两位小数格式输出 x, y 和 z 的值。注意第8行 z 的输出是四舍五入后的数值。第9行语句设置小数点后三位小数的输出, 第10, 11和12行则保留三位小数并输出 x, y 和 z 的值。注意第11行 y 的值是输出到小数点后面三位。因为存储在 y 中的数字只到小数点后两位, 所以小数点后第3位是0。

在第13行将浮点数输出设置到小数点第4位。第14, 15和16行则以4位小数输出 x, y 和 z 的值。注意第14行打印的 x 值在第4个小数位是0。并且第15行 y 的值在小数点后第3, 4位分别包括两个0。

在第17行上首先将浮点数的输出设置为三位小数, 并以3位小数格式输出 x。在打印 x 值以后, 第17行语句再将浮点数的输出设置为两位小数, 并以两位小数格式输出 y。接着将浮点数的输出设置到小数点后第4个位置, 然后以4位小数格式输出 z。

也可以使用流函数 `setf()` 来设置 `fixed`, `scientific` 和 `showpoint`。在这种情况下, `fixed`, `scientific` 和 `showpoint` 分别指定为 `ios::fixed`, `ios::scientific` 和 `ios::showpoint` 并且称做格式化标志。标志 `ios::fixed` 和 `ios::scientific` 都是 `ios::floatfield` 的一部分, `ios::floatfield` 在 C++ 中是一种数据类型。当设置固定或科学计数标志时, 为了保证只有 `ios::fixed` 或 `ios::scientific` 标志被设置, 必须将 `ios::floatfield` 作为第二个

参数传递给函数 `setf()`。下面的语句演示了怎样设置标志 `ios::fixed` 并在标准输出设备上用一种固定小数格式来输出浮点数:

```
cout.setf(ios::fixed, ios::floatfield);
```

同样, 下面的语句设置标志 `ios::showpoint` 并在标准输出设备上输出带有小数点和尾零的浮点数:

```
cout.setf(ios::showpoint);
```

C++ 也提供了操纵符 `setiosflags()`, 可以使用它来设置标志 `ios::fixed`, `ios::scientific`, 和 `ios::showpoint`。为了使用操纵符 `setiosflags()`, 程序必须包含头文件 `iomanip`。

下面的语句演示了怎样使用操纵符 `setiosflags()` 在标准输出设备上设置标志 `ios::fixed` 和 `ios::showpoint`:

```
cout<<setiosflags(ios::fixed);  
cout<<setiosflags(ios::showpoint);
```

可以用符号 `|` 通过分隔标志的方法, 在操纵符 `setiosflags()` 中指定多种标志。下面的语句在标准输出设备中设置了 `ios::fixed` 和 `ios::showpoint`:

```
cout<<setiosflags(ios::fixed|ios::showpoint);
```

注意: 在 ANSI/ISO 标准 C++ 中, `fixed`, `scientific` 和 `showpoint` 既可以作为操纵符, 也可以作为标志来使用。然而, 在一些标准 C++ 的编译器中, 它们只能用作标志。如果是这种情况, 并且用户使用的是标准 C++, 那么需要通过使用函数 `setf()` 或操纵符 `setiosflags()` 来设置它们。

3.4.4 setw()

操纵符 `setw()` 用来以特定的列输出一个表达式的值。表达式的值既可以是字符串, 也可以是数字。语句 `setw(n)` 以 `n` 列输出表达式的值。输出是右对齐的。这样, 如果指定了列的数目是 8, 例如, 输出只需要 4 列, 那么前 4 列将会是空格。而且, 如果指定列的数目小于输出需要的列数, 输出会自动地扩展为所需要的列数, 输出不会被截断。例如, 如果 `x` 是一个 `int` 变量, 下面的语句在标准输出设备上以 5 列来输出 `x` 的值:

```
cout<<setw(5)<<x<<endl;
```

为了使用操纵符 `setw()`, 程序必须包含头文件 `iomanip`。因此必需包含下面的语句:

```
#include<iomanip>
```

不像 `setprecision()`, 它控制所有浮点数的输出, 直到被重新设置。 `setw()` 只控制下一个表达式的输出。

例 3.8

下面的程序演示了如何使用运算符 `setw()`。

```
//Example: setw  
  
#include <iostream>  
#include <iomanip>  
  
using namespace std;  
  
int main()  
{  
    int x = 19;                                     //Line 1
```



```

int a = 345; //Line 2
double y = 76.384; //Line 3

cout<<fixed<<showpoint; //Line 4
cout<<"12345678901234567890"<<endl; //Line 5
cout<<setw(5)<<x<<endl; //Line 6
cout<<setw(5)<<a<<setw(5)<<"Hi"
    <<setw(5)<<x<<endl<<endl; //Line 7

cout<<setprecision(2); //Line 8
cout<<setw(6)<<a<<setw(6)<<y
    <<setw(6)<<x<<endl; //Line 9
cout<<setw(6)<<x<<setw(6)<<a
    <<setw(6)<<y<<endl<<endl; //Line 10

cout<<setw(5)<<a<<x<<endl; //Line 11
cout<<setw(2)<<a<<setw(4)<<x<<endl; //Line 12
return 0;
}

```

输出:

```

12345678901234567890
19
345   Hi   19

345 76.38   19
19   345 76.38

34519
345 19

```

第1, 2, 3行的语句声明了变量x, a和y, 并分别将这些变量的值初始化为19, 345和76.384。第4行语句用小数点和补0这种固定格式来设置浮点数的输出, 第5行语句的输出显示了当打印特定值时列的位置, 它是第1行的输出。

第6行语句用5列来输出x, 因为x只有两位数字, 所以只需要两列来输出它的值, 因此, 在第2行的输出中前三列是空格, 第7行语句在前5列输出了a的值, 在下5列输出字符串“Hi”, 然后在下5列输出x的值, 因为字符串“Hi”值包含了两个字符, 并且用5列来设置这两个字符的输出, 所以前3列是空格, 参看第3行的输出, 第4行的输出是空行, 这是因为操纵符endl在第7行语句中出现了两次。

第8行语句将浮点数的输出设置到小数点后第二位, 第9行语句用6列来输出a的值, 相邻的6列来输出y的值, 在后面的6列输出x的值, 这就产生了第5行的输出。第10行语句的输出(它是第6行的输出), 类似于第9行, 注意在第9行和第10行的语句输出中, 数字是如何排列, 第7行的输出是空行, 是因为操纵符endl在第10行语句中出现了两次。

第11行语句首先用5列输出了a的值, 然后输出x的值。注意第11行语句中的操纵符setw()控制a的输出, 这样a的值打印之后, x的值被打印在当前光标的位置(参看第8行输出)。

在第12行cout的输出中只指定了两个列来输出a的值, 然而, a的值包含三个数字, 所以输出扩展为三列, 然后在第4列打印x的值, 因为x的值包含两个数字, 所以需要两列来输出x的值。前两列是空格(参看第9行的输出)。

3.5 其他输出格式化工具

在前面章节中,已经了解了如何使用操纵符 `setprecision()`, `fixed` 和 `showpoint` 来控制浮点数的输出,以及如何使用操纵符 `setw()` 来显示特定列的输出。尽管这些操纵符对于产生完美的报表来说是足够的,但是有时用户希望能做一些更多的事情。在本部分中将学习其他格式化工具并对输出做更多的控制。

3.5.1 `fill()` 和 `setfill()`

前面讲过,在操纵符 `setw()` 中,如果指定列的数目超过了表达式需要的列数,那么表达式的输出是右对齐的,并且不被使用的列用空格来填充。输出流变量可以访问一个叫做 `fill()` 的函数和一个叫做 `setfill()` 的操纵符,它们允许用户使用其他字符来填充空列,而非空格。

使用函数 `fill()` 的语法是:

```
ostreamVar.fill(ch);
```

这里 `ostreamVar` 是输出流变量, `ch` 是一个字符。例如语句:

```
cout.fill('*');
```

在标准输出设备中设置 `*` 作为填充字符。

使用操纵符 `setfill()` 的语法是:

```
ostreamVar<<setfill(ch);
```

其中 `ostreamVar` 是一个输出流变量, `ch` 是一个字符。例如语句:

```
cout<<setfill('#');
```

在标准输出设备上设置填充字符为 `#`。

为了使用操纵符 `setfill()`, 程序必须包含头文件 `iomanip`。

如果使用 `fill()` 函数来设置填充字符,它必须是单独的语句。而操纵符 `setfill()` 是输出语句的一部分。下面的例子演示了函数 `fill()` 和操纵符 `setfill()` 的用法。

例 3.9

下面的程序演示了在程序中使用 `fill()` 和 `setfill()` 的输出结果。

```
//Example: fill and setfill
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int x = 15; //Line 1
    int y = 7634; //Line 2
    cout<<"12345678901234567890"<<endl; //Line 3
    cout<<setw(5)<<x<<setw(7)<<y
        <<setw(8)<<"Warm"<<endl; //Line 4
    cout.fill('*'); //Line 5
    cout<<setw(5)<<x<<setw(7)<<y
        <<setw(8)<<"Warm"<<endl; //Line 6
```

```

    cout<<setw(5)<<x<<setw(7)<<setfill('#')
        <<y<<setw(8)<<"Warm"<<endl;           //Line 7

    cout<<setw(5)<<setfill('@')<<x
        <<setw(7)<<setfill('#')<<y
        <<setw(8)<<setfill('^')<<"Warm"
        <<endl;                               //Line 8

    cout.fill(' ');                           //Line 9
    cout<<setw(5)<<x<<setw(7)<<y
        <<setw(8)<<"Warm"<<endl;             //Line 10
    return 0;
}

```

输出:

```

12345678901234567890
  15   7634   Warm
***15***7634***Warm
***15###7634###Warm
@@@15###7634^^^Warm
  15   7634   Warm

```

第1行和第2行语句分别声明并初始化了变量x和y为15和7634,第3行语句——输出的第1行——显示了当后面的语句输出了变量值时列的位置。第4行语句用5列输出了x的值,用7列输出了y的值,用8列输出了字符串“Warm”,语句中填充字符是空格字符,如第2行输出所示。

第5行语句设置填充字符为*,第6行语句用5列输出了x的值,用7列输出了y的值,用8列输出了字符串“Warm”,因为x是一个包含两位的数字并且指定了5列来输出它的值,所以x没有使用前3列,则用字符*来填充。为了打印y的值,指定了7列,然而y是一个包含4位的数字。所以填充字符填充了前3列,同样,为了打印字符串“Warm”的值,指定了8列,然而字符串“Warm”只有4个字符,所以填充字符填充了前4列,参看第3行的输出。

第7行语句(第4行的输出)类似于第6行语句,除了y和字符串“Warm”的填充字符是#,在第8行语句中(第5行输出)用于x的填充字符是@,用于y的填充字符是#,用于字符串“Warm”的填充字符是^。操纵符setfill()设置了这些填充字符。

第9行语句设置填充字符为空格,第10行语句使用填充字符空格来输出x,y和字符串“Warm”的值,如第6行输出所示。

3.5.2 left 和 right 操纵符

前面讲过,如果在setw()操纵符中指定的列数超过了表达式中所需要的列数,那么输出是右对齐的。有时,需要使输出左对齐时,可以使用操纵符left,设置操纵符left的语法为:

```
ostreamVar<<left;
```

其中ostreamVar是一个输出流变量。例如,下面的语句在标准输出设备上设置输出是左对齐的:

```
cout<<left;
```

可以使用流函数unsetf()使操纵符left失效。其语法为:

```
ostreamVar.unsetf(ios::left);
```

其中ostreamVar是一个输出流变量。使操纵符left失效并返回默认输出格式设置的输出。例如,下面的语句使标准输出设备上的操纵符left失效:

```
cout.unsetf(ios::left);
```

设置操纵符 right 的语法为:

```
ostreamVar<<right;
```

其中 ostreamVar 是输出流变量。例如, 下面的语句将标准输出设备上的输出设置为右对齐:

```
cout<<right;
```

例 3.10

下面的程序演示了操纵符 left 和 right 的使用。

```
//Example: left justification
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int x = 15; //Line 1
    int y = 7634; //Line 2
    cout<<left; //Line 3
    cout<<"12345678901234567890"<<endl; //Line 4
    cout<<setw(5)<<x<<setw(7)<<y
        <<setw(8)<<"Warm"<<endl; //Line 5
    cout.fill('*'); //Line 6
    cout<<setw(5)<<x<<setw(7)<<y
        <<setw(8)<<"Warm"<<endl; //Line 7
    cout<<setw(5)<<x<<setw(7)<<setfill('#')<<y
        <<setw(8)<<"Warm"<<endl; //Line 8
    cout<<setw(5)<<setfill('@')<<x
        <<setw(7)<<setfill('#')<<y
        <<setw(8)<<setfill('^')<<"Warm"<<endl; //Line 9
    cout.unsetf(ios::left); //Line 10
    cout.fill(' '); //Line 11
    cout<<setw(5)<<x<<setw(7)<<y
        <<setw(8)<<"Warm"<<endl; //Line 12
    return 0;
}
```

输出:

```
12345678901234567890
15 7634 Warm
15***7634***Warm***
15***7634###Warm###
15@@@7634###Warm^^^
15 7634 Warm
```

程序的输出与例 3.9 的输出是相同的。惟一不同在于第 4 行到第 9 行的语句, 输出是左对齐的。希望读者能一步步地仔细阅读该程序。

也可以使用流函数 `setf()` 来设置 `left` 和 `right`。在这种情况下, `left` 和 `right` 分别指定为 `ios::left` 和 `ios::right` 并称为格式化标志。标志 `ios::left` 和 `ios::right` 都是 `ios::adjustfield` 的一部分, `ios::adjustfield` 在 C++ 中是一种数据类型。为了保证只有一个标志 `ios::left` 或 `ios::right` 被设置, `ios::adjustfield` 需要作为第二个参数传递给函数 `setf()`。

设置标志 `left` 的语法是:

```
ostreamVar.setf(ios::left, ios::adjustfield);
```

其中 `ostreamVar` 是一个输出流变量。例如, 下面的语句在标准输出设备上设置输出是左对齐的:

```
cout.setf(ios::left, ios::adjustfield);
```

流函数 `unsetf()` 使标志 `ios::left` 失效。使标志 `ios::left` 失效的语法是:

```
ostreamVar.unsetf(ios::left);
```

其中 `ostreamVar` 是一个输出流变量。使标志 `ios::left` 失效并返回默认设置的输出。

也可以使用流操纵符 `setiosflags` 来设置标志 `ios::left` 和 `ios::right`。例如, 下面的语句在标准输出设备上设置标志 `ios::left`:

```
cout<<setiosflags(ios::left);
```

注意: 下面的语句使用操纵符 `setiosflags()` 在标准输出设备上设置标志 `ios::fixed`, `ios::showpoint` 和 `ios::left`: “`cout<<setiosflags(ios::fixed|ios::showpoint|ios::left);`”。

注意: 在标准 C++ 的一些编译器中, 操纵符 `left` 和 `right` 只能被用做格式化标志。所以, 必须使用函数 `setf()` 或操纵符 `setiosflags()` 来设置 `left` 和 `right`。检查编译器文档以获得更多的信息。

3.5.3 flush() 函数

操纵符 `endl` 和换行转义序列符 `\n` 都可以将光标定位于输出设备下一行的开始, 然而操纵符 `endl` 还有另一个用途。

当一个程序向输出设备输出时, 输出首先进入计算机的缓冲区中, 只要缓冲区已满, 输出就会发送到输出设备上, 然而, 只要一遇到操纵符 `endl`, 缓冲区中的输出就会立刻送入输出设备, 即使缓冲区未满, 所以, 操纵符 `endl` 将光标定位于输出设备下一行的开始, 并且帮助清除缓冲区。

有时可能看不到整个输出, 这是因为当程序终止时, 缓冲区可能未满。在 C++ 中可以使用函数 `flush()` 来清除缓冲区, 即使缓冲区未满, 与操纵符 `endl` 相比, 函数 `flush()` 不会将光标移到下一行的开始。

使用函数 `flush()` 的语法是:

```
ostreamVar.flush();
```

其中 `ostreamVar` 是一个输出流变量, 例如: `cout`。

类似于 `endl`, 函数 `flush()` 也可以用作操纵符, 在这种情况下, `flush` 直接用于输出语句中, 例如下面的语句, 将输出从缓冲区发送到标准输出设备中:

```
cout<<flush;
```

例 3.11

考虑下面的语句, 其中 `num` 是一个 `int` 变量:

```
cout<<"Enter an integer: " ;           //Line 1
cin>>num;                               //Line 2
cout<<endl;                             //Line 3
```

第一行语句输出下面的文本: Enter an integer:。在输出该行后,光标停留在冒号后。前面讲过,第一行语句的输出首先进入到缓冲区。如果缓冲区未满,该行文本可能不被显示,在这种情况下用户不知道下一步要做什么。可以在第一行语句的后面加入操纵符 endl。然而,这样做之后,在打印文本行之后,光标定位在下一行的开始。这样用户在下一行输入数字,有时可能不会令人满意。另一方面,假设第一行语句替换为下面一行语句:

```
cout<<"Enter an integer:"<<flush;      //Line 1
```

在这种情况下,文本行 Enter an integer: 显示在标准输出设备上,即使缓冲区未满。而且,在输出文本行之后,光标停留在冒号之后,然后用户可以在冒号后面输出数字。

注意: 本章主要讨论几个流函数和流操纵符。为了在程序里使用流函数,比如 get(), ignore(), clear() 和 setf(), 程序必须包括头文件 iostream。有两种类型的操纵符: 有参数的操纵符和无参数的操纵符。有参数的操纵符被称为参数化流操纵符。

例如,像 setprecision(), setw(), setfill() 和 setiosflags() 的操纵符是参数化的。另一方面,像 endl, fixed, scientific, showpoint 和 left 是没有参数的操纵符。因为 flush 也可以用做一个没有任何参数的操纵符,所以 flush 也是一个没有参数的操纵符。

为了在程序中使用参数化流操纵符(即有参数的流操纵符),必须包含头文件 iomanip。没有参数的操纵符是 iostream 头文件的一部分,所以,不需要包含头文件 iomanip。

3.5.4 输入/输出和字符串类型

可以使用一个输入流变量,例如 cin, 以及析取运算符 >> 向一个 string 类型的数据变量读入数据。例如,如果输入是字符串 "shelly", 下面的代码将该字符串输入到变量 name 中:

```
string name;           //declaration
cin>>name;             //input statement
```

前面讲过,析取运算符会跳过前面的空格字符,并且读操作会停在一个空格字符处。结果,则不能使用析取运算符来读入包含空格的字符。例如,假设变量 name 如上述所定义,如果输入是:

```
Alice Wonderland
```

那么下面的语句执行后变量 name 的值是 "Alice":

```
cin>>name;
```

为了读入一个包含空格的字符串,可以使用函数 getline()。使用函数 getline() 的语法是:

```
getline(istreamVar, strVar);
```

其中 istreamVar 是一个输入数据流变量, strVar 是 string 数据类型的变量。读入由换行符 '\n' 来限制。

函数 getline() 会读到当前行的末尾。换行符也会被读入,但是不会存储到 string 变量中。

考虑下面的语句:

```
string myString;
```

如果输入 29 个字符,

```
bbbbHello there.How are you?
```

其中 `b` 代表空格, 在语句:

```
getline(cin, mystring);
```

执行后, `myString` 的值是:

```
myString="    Hello there.How are you? "
```

所有 29 个字符, 包括前 4 个空格, 都被存储到 `myString` 中。

同样, 可以使用输出流变量, 例如 `cout`, 以及插入操纵符来输出 `string` 数据类型变量的内容。

3.6 文件输入/输出

前面已经详细地讨论了如何从键盘(标准输入设备)获得输入以及如何将输出发送到屏幕上(标准输出设备)。然而, 从键盘获得输入和将输出发送到屏幕是有一定限制的。在一个程序中从键盘上接受数据是很容易的, 只要输入的数量比较少。如果这些数据比较少(不超过屏幕的尺寸), 并且不想用一种打印的方式向其他设备发送输出, 那么输出到屏幕上的工作是很容易的, 然而, 如果输入数据量很大并且每次运行程序时将其输入, 那么效率是很低的。除了输入大量数据不方便外, 输入可能还会产生错误, 并且无目的的输入还可能引起错误。因此, 需要有一些方法从其他数据源向程序输入数据。通过使用可选数据源, 可以在程序运行之前准备数据, 并且程序每次运行时能访问数据。

假设在会议上需要提供一个程序的输出材料, 分发程序输出的打印材料比在屏幕上显示输出效果会更好, 例如: 在一个重要会议之前, 可以将打印报告发放给每一个参加会议的成员, 而且有时输出必须被保存, 这样一个程序产生的输出可以作为另一个程序的输入。

该部分讨论了怎样从其他输入设备例如磁盘(即辅存)来获得数据, 以及怎样将输出保存到磁盘上, C++ 允许程序直接从辅存获得数据, 或者直接将结果保存到辅存上。程序可以使用文件 I/O 并且能从文件读入数据或向文件写入数据。文件的正式定义如下:

file: 在辅存中用来保存信息的一个区域。

标准 I/O 头文件 `iostream` 包含了只能用于从标准输入设备输入以及向标准输出设备输出的数据类型变量, 另外, C++ 提供了一个名为 `fstream` 的头文件, 它用于文件 I/O, 其中, `fstream` 头文件包含了两个数据类型的定义: `ifstream` 表示输入文件流, 它与 `istream` 相似; `ofstream` 表示输出文件流, 与 `ostream` 相似。

变量 `cin` 和 `cout` 前面已经定义过, 它们与标准输入/输出设备是相关联的。另外, `>>`、`get()`、`ignore()`、`putback()`、`peek()`, 等等, 能与 `cin` 一起使用; `<<`、`fill()`、`setfill()`, 等等, 能与 `cout` 一起使用。这些相同的运算符和函数对于文件 I/O 也是可用的, 但是头文件没有声明并使用它们。必须声明一个名为文件流变量的变量, 它包括用于输入的 `ifstream` 变量和用于输出的 `ofstream` 变量。然后将这些变量与 `>>`、`<<` 或其他用于 I/O 的函数一起使用。前面讲过, C++ 不能自动初始化用户定义的变量。一旦声明了 `fstream` 变量, 需将这些变量与输入/输出源联系起来。

文件 I/O 包括 5 个步骤:

1. 在程序中包含头文件 `fstream`。
2. 声明文件流变量。
3. 将文件流变量与输入/输出源联系起来。
4. 将文件流变量与 `>>`、`<<` 或其他输入/输出函数一起使用。
5. 关闭文件。

现在详细地描述这5个步骤。一个简要的程序显示了这些步骤如何应用在程序中。

第1步需要在程序中包含头文件 `fstream`。下面的语句完成该任务：

```
#include<fstream>
```

第2步需要声明文件流变量。考虑下面的语句：

```
ifstream inData;
ofstream outData;
```

第一条语句声明 `inData` 是一个输入文件流变量。第二条语句声明 `outData` 是一个输出文件流变量。

第3步需要将文件流变量与输入/输出源联系起来，该步骤称为打开文件。流成员函数 `open()` 用来打开文件。打开文件的语法是：

```
fileStreamVariable.open(sourceName,fileOpeningMode);
```

这里 `fileStreamVariable` 是一个文件流变量，`sourceName` 是一个输入/输出文件的名称，`fileOpeningMode` 指定了文件打开的方式。文件打开方式用阴影表示，表示了它是语法定义的可选部分。对于 `ifstream` 变量，默认的文件打开方式是用于输入数据。对于 `ofstream` 变量，默认的文件打开方式是用于输出。表 3.2 列出了不同的文件打开方式。

表 3.2 文件打开方式

文件打开方式	描述
<code>ios::in</code>	打开文件用以输入数据。 <code>ifstream</code> 变量的默认方式。
<code>ios::out</code>	打开文件用以输出数据。 <code>ofstream</code> 变量的默认方式。
<code>ios::nocreate</code>	如果文件不存在，打开操作失败。
<code>ios::app</code>	追加输出至文件尾。如果文件不存在，则创建空文件。
<code>ios::ate</code>	打开文件用以输入数据并移至文件尾。输出数据可以写入文件中任一位置。
<code>ios::trunc</code>	如果文件已存在，其内容将被删除。
<code>ios::noreplace</code>	如果文件存在，打开操作失败。

假设在程序中从第2步包含了声明。假设输入数据存储在驱动器 A 的软盘上且名为 `prog.dat` 的文件中，并想将输出保存到驱动器 A 的软盘上名为 `prog.out` 的文件中，下面的语句将 `inData` 和 `prog.dat` 联系起来，将 `outData` 和 `prog.out` 联系起来。即打开文件 `prog.dat` 用于输入数据，打开文件 `prog.out` 用于输出数据。

```
inData.open("A:prog.dat"); //open input file
outData.open("A:prog.out"); //open output file
```

第4步的工作原理如下：使用文件流变量和 `>>`, `<<` 或其他输入/输出函数。使用 `>>` 或 `<<` 等文件流变量的语法与使用 `cin` 和 `cout` 的语法是完全相同的。然而，与使用 `cin` 和 `cout` 不同的是，必须使用声明的文件流变量名字。

例如，语句：

```
inData>>payRate;
```

从文件 `prog.dat` 中读入数据并且将这些数据存储在变量 `payRate` 中。语句：

```
outData<<"The paycheck is:$"<<pay<<endl;
```

在文件 `prog.out` 中存储输出 `The paycheck is:$565.78`。语句假定了计算后的工资是 565.78。

一旦 I/O 完成，第5步需要关闭文件。关闭文件表示文件流变量与存储域分开，文件流变量被释放。一旦这些流变量被释放，它们能重新用于其他文件 I/O。使用流函数 `close()` 关闭文件。例如，假定程序包含第2步和第3步所列的声明，关闭文件的语句是：


```
inData.close();
outData.close();
```

注意：在一些系统中，关闭文件这一步骤是必需的。虽然当程序终止时，文件会自动关闭。但是，自己关闭文件是一个很好的习惯。而且，如果想使用同一个文件流变量打开另一个文件，必须关闭用该文件流变量打开的第一个文件。

综上所述，使用文件 I/O 的程序通常有下面的形式：

```
#include <fstream>
//Add additional header files you use
using namespace std;

int main()
{
    //Declare file stream variables such as the following
    ifstream inData;
    ofstream outData;
    ...
    //Open files
    inData.open("A:prog.dat"); //open input file
    outData.open("A:prog.out"); //open output file
    //Code for data manipulation

    //Close files
    inData.close();
    outData.close();
    return 0;
}
```

第3步需要打开文件用于文件 I/O。打开一个文件会将程序中声明的文件流变量与源（例如磁盘）中的物理文件联系起来。文件输入时，`open()` 语句执行前文件必须存在。如果文件不存在，`open()` 语句就会失败并且输入流进入错误状态。输出文件在打开前不必一定存在，如果输出文件不存在，计算机将为输出准备一个空文件。如果指定的输出文件已经存在，默认情况下，当该文件打开时，文件中原来的内容会被删除。

注意：在前面的部分中，学习了怎样使用 `open()` 函数来打开一个文件以向一个程序中输入数据，然后存储程序的输出。假定程序试图打开一个文件以进行输入数据，但是输入文件不存在。那么 `open()` 语句就会失败，而且输入流进入错误状态。在这种情况下，一些系统会自动地创建一个空文件以供输入数据，这样避免了 `open()` 语句进入错误状态。

为了阻止系统创建一个空输入文件，必须在 `open()` 语句中包含 `ios::nocreate` 作为第二个参数。第二个参数 `ios::nocreate` 保证了如果输入文件不存在，打开操作将会失败。例如，假设已经包括前面的声明，下面的语句打开了文件 `prog.dat` 以进行输入数据。如果文件不存在，打开操作将会失败。

```
inData.open("A:prog.dat", ios::nocreate);
```

3.7 编程实例：电影票销售和对慈善机构的捐助

为了帮助一家本地的慈善机构，电影院业主决定将从电影收入的总额中拿出一部分捐助给慈善机构。该例子设计并执行了一个程序，即提示用户输入电影名字、成人票价格、儿童票价格、销售的成人票数量、销售的儿童票数量和要捐助给慈善机构的总额百分比。程序的输入如下：

```

--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*
Movie Name: ..... Ducky Goes to Mars
Number of Tickets Sold: ..... 2650
Gross Amount: ..... $ 9150.00
Percentage of Gross Amount Donated: 10.00%
Amount Donated: ..... $ 915.00
Net Sale: ..... $ 8235.00

```

注意第一列是左对齐的，右边列的数字是右对齐的，小数输出保留小数点后两位。

输入 程序的输入包括电影名字、成人票价格、儿童票价格、销售的成人票数量、销售的儿童票数量和要捐助给慈善机构的总额百分比。

输出 输出如上述显示。

问题分析和算法设计

为了计算本地慈善机构的捐款和净销售额，首先需要计算总额。为了计算总额，需要将销售的成人票数量乘以成人票价格、儿童票数量乘以儿童票价格，然后将两个数字相加。即：

```

grossAmount = adultTicketPrice * noOfAdultTicketsSold
              + childTicketPrice * noOfChildTicketsSold;

```

接着，需要决定捐献给本地慈善机构数额的百分比，然后从总额中减掉捐赠的数额得到净销售额。计算捐赠的数额和净销售额的公式由下面给出。该分析导致了下面的算法：

1. 得到电影名字
2. 得到成人票价格
3. 得到儿童票价格
4. 得到销售的成人票数量
5. 得到销售的儿童票数量
6. 得到捐助给慈善机构的总额百分比
7. 使用下面的公式计算总销售额：

```

grossAmount = adultTicketPrice * noOfAdultTicketsSold
              + childTicketPrice * noOfChildTicketsSold;

```

8. 使用下面的公式计算捐助给慈善机构的款量：

```

amountDonated = grossAmount * percentDonation / 100;

```

9. 使用下面的公式计算净销售额：

```

netSale = grossAmount - amountDonated;

```

变量 从前面的讨论中可以看出，需要变量来存储电影名字、成人票价格、儿童票价格、销售的成人票数量、销售的儿童票数量和要捐助给慈善机构的总额百分比、总销售额、捐赠的款额和净销售额。所以，需要下面的变量：

```

string    movieName;
double    adultTicketPrice;
double    childTicketPrice;
int       noOfAdultTicketsSold;
int       noOfChildTicketsSold;
double    percentDonation;
double    grossAmount;
double    amountDonated;

```

```
double    netSaleAmount;
```

因为 movieName 声明为一个 string 类型的变量,所以需要包含头文件 string。所以程序还需要包含下面的语句:

```
#include<string>
```

格式化输出 在输出中,第一列是左对齐的,第二列的数字是右对齐的。所以,当在第一列打印一个值时,需要使用操纵符 left; 在第二列打印值之前,需要使用 right 操纵符。第一列和第二列之间的空白用逗号来填充,程序使用了操纵符 setfill() 来达到这一目的。在显示总销售额、捐赠数额和净销售数额的行中,在 \$ 符号和数字之间用空格来填充。所以,在打印美元符之前,程序使用操纵符 setfill() 来设置填充字符。下面的语句完成了输出:

```
cout<<"*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*"
    <<"*-*-*-*-*-*-*-*-*-*-*-*-*-*-*" <<endl;
cout<<setfill('.')<<left<<setw(35)<<"Movie Name: "
    <<right<<" "<<movieName<<endl;
cout<<left<<setw(35)<<"Number of Tickets Sold: "
    <<setfill(' ')<<right<<setw(10)
    <<noOfAdultTicketsSold + noOfChildTicketsSold
    <<endl;
cout<<setfill('.')<<left<<setw(35)<<"Gross Amount: "
    <<setfill(' ')<<right<<" $"
    <<setw(8)<<grossAmount<<endl;
cout<<setfill('.')<<left<<setw(35)
    <<"Percentage of Gross Amount Donated: "
    <<setfill(' ')<<right
    <<setw(9)<<percentDonation<<"%"<<endl;
cout<<setfill('.')<<left<<setw(35)<<"Amount Donated: "
    <<setfill(' ')<<right<<" $"
    <<setw(8)<<amountDonated<<endl;
cout<<setfill('.')<<left<<setw(35)<<"Net Sale: "
    <<setfill(' ')<<right<<" $"
    <<setw(8)<<netSaleAmount<<endl;
```

主要算法 通过前面的讨论,程序的主要算法如下:

1. 声明变量。
2. 用包含小数点和尾零的固定方式将浮点数的输出设置到小数点后面两位。所以,需要包含头文件 iomanip。
3. 提示用户输入电影名字。
4. 输入(读入)电影名字。因为电影的名字可能包含不止一个单词(可能包含空格),所以程序使用函数 getline() 输入电影的名字。而且,因为函数 getline() 也读入换行符,所以在一行中输入电影名字之后,需要按两次 Enter 键。
5. 提示用户输入成人票的价格。
6. 输入(读入)成人票的价格。
7. 提示用户输入儿童票的价格。
8. 输入(读入)儿童票的价格。
9. 提示用户输入销售的成人票数量。
10. 输入(读入)销售的成人票数量。
11. 提示用户输入销售的儿童票数量。

12. 输入 (读入) 销售的儿童票数量。
13. 提示用户输入要捐助给慈善机构的总额百分比。
14. 输入 (读入) 要捐助给慈善机构的总额百分比。
15. 计算总销售额。
16. 计算捐赠的款额。
17. 计算净销售额。
18. 输出结果。

完整程序清单

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main()
{
    //Step 1
    string movieName;
    double adultTicketPrice;
    double childTicketPrice;
    int noOfAdultTicketsSold;
    int noOfChildTicketsSold;
    double percentDonation;
    double grossAmount;
    double amountDonated;
    double netSaleAmount;
    cout<<fixed<<showpoint<<setprecision(2); //Step 2

    cout<<"Enter movie name: "<<flush; //Step 3
    getline(cin, movieName); //Step 4
    cout<<endl;

    cout<<"Enter the price of an adult ticket: "<<flush; //Step 5
    cin>>adultTicketPrice; //Step 6
    cout<<endl;

    cout<<"Enter the price of a child ticket: "<<flush; //Step 7
    cin>>childTicketPrice; //Step 8
    cout<<endl;

    cout<<"Enter number of adult tickets sold: "<<flush; //Step 9
    cin>>noOfAdultTicketsSold; //Step 10
    cout<<endl;

    cout<<"Enter number of child tickets sold: "<<flush; //Step 11
    cin>>noOfChildTicketsSold; //Step 12
    cout<<endl;

    cout<<"Enter the percentage of donation: "<<flush; //Step 13
    cin>>percentDonation; //Step 14
    cout<<endl<<endl;

    grossAmount = adultTicketPrice * noOfAdultTicketsSold + //Step 15
                  childTicketPrice * noOfChildTicketsSold;

    amountDonated = grossAmount * percentDonation / 100; //Step 16
    netSaleAmount = grossAmount - amountDonated; //Step 17

    //Step 18: Output results
```

```

cout<<"-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*"
    <<"-*-*-*-*-*-*-*-*-*-*-*-*-*-*" <<endl;
cout<<setfill('.')<<left<<setw(35)<<"Movie Name: "
    <<right<<" "<<movieName<<endl;
cout<<left<<setw(35)<<"Number of Tickets Sold: "
    <<setfill(' ')<<right<<setw(10)
    <<noOfAdultTicketsSold + noOfChildTicketsSold
    <<endl;
cout<<setfill('.')<<left<<setw(35)<<"Gross Amount: "
    <<setfill(' ')<<right<<" $"
    <<setw(8)<<grossAmount<<endl;
cout<<setfill('.')<<left<<setw(35)
    <<"Percentage of Gross Amount Donated: "
    <<setfill(' ')<<right
    <<setw(9)<<percentDonation<<"%"<<endl;
cout<<setfill('.')<<left<<setw(35)<<"Amount Donated: "
    <<setfill(' ')<<right<<" $"
    <<setw(8)<<amountDonated<<endl;
cout<<setfill('.')<<left<<setw(35)<<"Net Sale: "
    <<setfill(' ')<<right<<" $"
    <<setw(8)<<netSaleAmount<<endl;

return 0;
}

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Enter movie name: Duckey Goes to Mars

Enter the price of an adult ticket: 4.50
Enter the price of a child ticket: 3.00
Enter number of adult tickets sold: 800
Enter number of child tickets sold: 1850
Enter the percentage of donation: 10

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Movie Name: ..... Duckey Goes to Mars
Number of Tickets Sold: ..... 2650
Gross Amount: ..... $ 9150.00
Percentage of Gross Amount Donated: 10.00%
Amount Donated: ..... $ 915.00
Net Sale: ..... $ 8235.00

```

3.8 编程实例：学生分数

编写一个程序，读入学生的ID号以及5个测验成绩。程序应该输出学生ID、5个测验成绩和平均测验成绩。平均分保留小数点后两位。假定学生ID是一个字符。

读入的数据存储在一个名为test.txt的文件中，该文件存储在驱动器A的软盘中。输出应该保存在一个名字为testavg.out的文件中，输出文件也应该保存在驱动器A的软盘中。

输入 包含学生ID和5个测验成绩的文件。

输出 学生ID、5个测验成绩和5个测验成绩的平均分保存到一个文件中。

问题分析和算法设计

为了计算5个测验成绩的平均分，需要将5个测验成绩相加，然后将和除以5。输入数据使用

下面的形式：学生 ID，然后是 5 个测验成绩。所以，必须读入学生 ID，然后读入 5 个测验成绩。这一分析过程可以翻译成下面的算法：

1. 读入学生 ID 和 5 个测验成绩。
2. 输出学生 ID 和 5 个测验成绩。
3. 计算平均分。
4. 输出平均分。

保留两位小数输出测验平均分。

变量 程序需要读入学生 ID 和 5 个测验成绩。所以需要有一个变量来存储学生 ID，5 个变量来存储测验成绩。为了求得平均分，必须将 5 个测验成绩相加，然后将和除以 5。这样，需要一个变量来存储平均测验成绩。而且，因为输入数据在一个文件中，所以需要有一个 `ifstream` 变量打开输入文件。因为程序输出将保存在一个文件中，所以需要有一个 `ofstream` 变量打开输出文件。这样，程序至少需要下面的变量：

```
ifstream inFile;    //input file stream variable
ofstream outFile;   //output file stream variable

int test1, test2, test3, test4, test5; //variables to
                                     //read five test scores
double average;     //variable to store average test score
char studentId;     //variable to store student ID
```

通过前面的讨论，现在可以编写主要的算法。

主要算法

1. 声明变量。
2. 打开输入文件。
3. 打开输出文件。
4. 为了以小数点和尾零的固定方式来输出浮点数，需要设置操纵符 `fixed` 和 `showpoint`。而且，为了将浮点数输出设置到小数点后面两位，需要将精度设置到小数点后第二个位置。
5. 读入学生 ID。
6. 输出学生 ID。
7. 读入 5 个测验成绩。
8. 输出 5 个测验成绩。
9. 计算平均测验成绩。
10. 输出平均测验成绩。
11. 关闭输入和输出文件。

因为该程序从一个文件中读入数据并且将数据输出到一个文件，所以必须包含头文件 `fstream`。因为程序输出平均分保留两位小数，所以需要将精度设置到小数点后第二个位置。这样，程序要使用操纵符 `setprecision()`，所以需要包含头文件 `iomanip`。程序也需要包含头文件 `iostream` 在屏幕上打印信息，这样在程序执行时，不会出现空白屏幕。

完整程序清单

```
//Program to calculate average test score
#include <iostream>
#include <fstream>
```



```

#include <iomanip>
using namespace std;

int main()
{
    //Declare variables; Step 1
    ifstream inFile; //input file stream variable
    ofstream outFile; //output file stream variable

    int test1, test2, test3, test4, test5;
    double average;
    char studentId;
    inFile.open("a:test.txt"); //Step 2
    outFile.open("a:testavg.out"); //Step 3

    outFile<<fixed<<showpoint; //Step 4
    outFile<<setprecision(2); //Step 4

    cout<<"Processing data"<<endl;
    inFile>>studentId; //Step 5
    outFile<<"Student ID: "<<studentId
        <<endl; //Step 6
    inFile>>test1>>test2>>test3
        >>test4>>test5; //Step 7
    outFile<<"Test scores: "<<setw(4)<<test1
        <<setw(4)<<test2<<setw(4)<<test3
        <<setw(4)<<test4
        <<setw(4)<<test5<<endl; //Step 8
    average = static_cast<double>(test1+test2+test3+
        test4+test5)/5.0; //Step 9
    outFile<<"Average test score: "<<setw(6)
        <<average<<endl; //Step 10

    inFile.close(); //Step 11
    outFile.close(); //Step 11
    return 0;
}

```

运行实例

输入文件 (文件 a:test.txt 内容):

T 87 89 65 37 98

输出文件 (文件 a:testavg.out 内容):

Student ID: T
 Test scores: 87 89 65 37 98
 Average test score: 75.20

注意: 为了用标准 C++ 编写程序, 需要将下面的语句;

```

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

```

替换为:

```

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>

```

也许需要替换语句:


```
outFile<<fixed<<showpoint;
```

为:

```
outFile.setf(ios::fixed,ios::floatfield);  
outFile.setf(ios::showpoint);
```

3.9 小结

1. C++ 中的数据流是一种从源到目标的序列字符。
2. 输入数据流是从源到计算机的数据流。
3. 输出流是从计算机到目标的数据流。
4. cin 代表常用输入设备,它是一个输入数据流对象,通常初始化为标准输出设备即键盘。
5. cout 代表常用输出设备,它是一个输出数据流对象,通常初始化为标准输出设备即屏幕。
6. 当二元操作符>>与输入流对象(例如 cin)使用时,它称为析取运算符。操作符左边的操作数必须是一个输入数据流变量,例如 cin,右边的运算符必须是一个变量。
7. 当二元操作符<<与输出流对象(例如 cout)使用时,它称为插入操作符。操作符左边的操作数必须是一个输出数据流变量,例如 cout,右边的操作数必须是一个表达式或一个操纵符。
8. 当向一个变量输入数据时,操作符>>跳过所有前面的空白字符。
9. 为了使用 cin 和 cout,程序必须包含头文件 iostream。
10. 函数 get() 逐个字符地读入数据,并跳过任何空白字符。
11. 函数 ignore() 用来跳过一行数据。
12. 函数 putback() 将函数 get() 接收的最后一个字符重新放回输入数据流。
13. 函数 peek() 从输入数据流返回下一个字符,但是并不删除输入数据流中的字符。
14. 试图将一个无效数据读入变量会使输入数据流进入 fail 状态。
15. 一旦输入产生失败,可以使用函数 clear() 将输入数据流恢复到工作状态。
16. 操纵符 setprecision() 将浮点数的输出格式化到一个指定的小数位置。
17. 操纵符 fixed 用固定的小数格式来输出浮点数。
18. 操纵符 showpoint 输出包含小数点和尾零的浮点数。
19. 操纵符 setw() 用特定的列数来格式化一个表达式的输出,默认输出是右对齐的。
20. 如果在 setw() 的参数中指定的列数小于打印表达式的值所需要的列数,输出不会被截断,表达式的输出会扩展为相应的列数。
21. 为了使用数据流函数 get(), ignore(), putback(), peek(), fill(), clear(), setf() 和 unsetf() 进行标准 I/O 操作,程序必须包含头文件 iostream。
22. 为了使用操纵符 setprecision(), setw(), setfill() 和 setiosflags(), 程序必须包含头文件 iomanip。
23. 头文件 fstream 包含 ifstream 和 ofstream 的定义。
24. 对于文件 I/O, 必须使用命令 #include<fstream> 在程序中包含头文件 fstream。同时需要: 声明 ifstream 类型的变量以备文件输入, 声明 ofstream 类型的变量以备文件输出。使用 open() 函数打开输入和输出文件。可以将<<, >>, get(), ignore(), peek(), putback() 或 clear() 与文件流变量一起使用。
25. 为了关闭一个由 ifstream 变量 inFile 标识的文件, 需要使用语句 "inFile.close();"。为了关闭一个由 ofstream 变量 outFile 标识的文件, 需要使用语句 "outFile.close();"。

3.10 练习

1. 判断下面语句的对错。

- 当检查输入数据流中下一个数据时，析取运算符>>跳过所有前面的空白字符。
- 在语句“cin>>x;”中，x必须是一个变量。
- 语句“cin>>x>>y;”需要x,y的输入值出现在同一行。
- 语句“cin>>num;”等价于语句“num>>cin;”
- 通过按下键盘上的Enter (return 键)可以产生一个换行符。
- 函数ignore()用来跳过一行中特定的输入。

2. 假设x和y是int变量，ch是char变量。考虑下面的输入：

5 28 36

在下面每一行语句执行后，x,y和ch分别为何值？（每一条语句使用相同的输入。）

- cin>>x>>y>>ch;
 - cin>>ch>>x>>y;
 - cin>>x>>ch>>y;
 - cin>>x>>y;
- cin.get(ch);

3. 假设x和y是int变量，ch是一个char变量。假定下面的输入数据：

13 28 D

14 E 98

A B 56

在下面每一行语句执行后，x,y和ch分别为何值？（每一条语句使用相同的输入。）

- cin>>x>>y;
cin.ignore(50, '\n');
cin>>ch;
- cin>>x;
cin.ignore(50, '\n');
cin>>y;
cin.ignore(50, '\n');
cin.get(ch);
- cin>>y;
cin.ignore(50, '\n');
cin>>x>>ch;
- cin.get(ch);
cin.ignore(50, '\n');
cin>>x;
cin.ignore(50, 'E');
cin>>y;

4. 给定输入：

46 A 49

和C++代码：

```
int x = 10, y = 18;
char z = 'A';
cin>>x>>y>>z;
cout<<x<<" "<<y<<" "<<z;
```

那么输出是什么?

5. 编写 C++ 语句, 使用操纵符 `setfill()` 在一行中输出 35 个星号。如下所示:

```
*****
```

6. 下面的程序假定从名为 `input.dat` 的文件中读入数据, 向名为 `output.dat` 的文件写入数据的和。然而, 如果它不能完成这些任务, 修改该程序以能达到相应要求。

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    int num1, num2;
    ifstream infile;

    outfile.open("output.dat");
    infile>>num1>>num2;
    outfile<<"Sum = "<<num1+num2<<endl;
    return 0;
}
```

7. 什么能导致输入数据流进入 fail 状态? 当输入数据流进入 fail 状态会发生什么?
8. 程序从名为 `inputFile.dat` 的文件中读入数据, 在做完一些计算后, 将结果写到名为 `outFile.dat` 的文件中。回答下面的问题:
- 在程序执行后, 文件 `inputFile.dat` 的内容是什么?
 - 如果文件 `outFile.dat` 的内容为空, 在程序执行后该文件的内容是什么?
 - 如果文件 `outFile.dat` 的内容包含 100 个数字, 程序执行后该文件的内容是什么?
 - 如果在程序执行之前, `outFile.dat` 文件不存在将会发生什么?

3.11 编程练习

1. 考虑下面不完整的 C++ 程序。

```
#include <iostream>

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

- 编写一条语句在该程序中包含头文件 `fstream`。
- 编写一条语句声明 `inFile` 是一个 `ifstream` 类型的变量, `outFile` 是一个 `ofstream` 类型的变量。
- 程序将会从文件 `inData.txt` 中读数据, 向 `outData.dat` 文件写数据。编写语句打开这两个文件, 将 `inFile` 与 `inData.txt` 连接起来, `outFile` 与 `outData.dat` 连接起来。
- 假设文件 `inData.txt` 包含下面的数据:

```
56    38
A
7      8
```

编写一条语句, 在程序执行后, 文件 `outData.dat` 的内容如下所示。如果需要, 可以声明额外的变量。语句应该足够通用, 这样如果输入文件的内容修改并且程序再次运行时(没有编辑和重新编译), 它还可以输出适当的结果。

```
Sum of 56 and 38 = 94.
The character that comes after A in the ASCII set is B.
The product of 7 and 8 = 56.
```

e. 编写语句关闭输入和输出文件。

f. 编写一个 C++ 程序来测试 a~e 语句部分。

2. 编写程序提示用户输入小数, 然后输出该小数四舍五入到小数点后第二位的数字。

3. 体育场的经理需要编写一个程序来计算每一场比赛后销售的总票数。有4种类型的票: 包厢, sideline, premium 和普通票。在每一场比赛后, 数据用下面的格式存储在一个文件中:

```
ticketPrice  numberOfTicketsSold
```

...

样本数据如下所示:

```
250      5750
100      28000
 50      35750
 25      18750
```

第一行显示票价是250美元, 该价格的票销售了5750张。输出销售的总票数以及总销售额。将输出格式化到小数点后两位。

4. 编写一个程序计算财产税。财产税以总评估财产的92%计算。例如, 如果评估资产是100000美元, 那么其中92000美元要上缴财产税。假设财产税率是1.05%。程序应该提示用户输入评估的财产值。用下面的格式在文件中存储输出。(这里是一个样本输出。)

```
Assessed Value:      100000.00
Taxable Amount:      92000.00
Tax Rate for each $100.00:      1.05
Property Tax:      966.00
```

格式化输出到小数点后面两位。(注意左边的列是左对齐的, 右边的列是右对齐的。)

5. 编写一个程序, 将华氏温度转化为摄氏温度。华氏温度转化为摄氏温度的公式为:

$$c = (5/9)(F - 32)$$

程序应该提示用户输入华氏温度表示的气温。程序应该输出用华氏温度和摄氏温度表示的气温。

6. 编写一个程序来计算和打印一名雇员每月的薪水。扣除下列款项并计算纯收入:

```
Federal Income Tax: 15%
State Tax:      3.5%
Social Security Tax: 5.75%
Medicare/Medicaid Tax:      2.75%
Pension Plan: 5%
Health Insurance:      $75.00
```

程序应该提示用户输入总收入和雇员名字。输出保存在一个文件中。将输出格式化到小数点后面两位。输出的例子如下所示:

```
Bill Robinson
Gross Amount: ..... $3575.00
Federal Tax: ..... $ 536.25
State Tax: ..... $ 125.13
Social Security Tax: ..... $ 205.56
Medicare/Medicaid Tax: ... $ 98.31
Pension Plan: ..... $ 178.75
Health Insurance: ..... $ 75.00
Net Pay: ..... $2356.00
```

注意第一列是左对齐的, 右边的列是右对齐的。

第4章 控制结构 I (选择结构)

本章要点:

- 学习控制结构
- 了解关系和逻辑运算符
- 学习如何创建和计算逻辑(布尔)表达式的值
- 学习如何在一个程序中使用选择结构 if, if...else 和 switch 语句
- 学习如何使用 assert() 函数来终止程序

在第2章中讲过,程序是由一系列的语句构成的,其目的是为了完成一些任务。到目前为止,所学到的程序都是很简单并且是很直接的。在执行程序期间,计算机从第一条可执行语句开始,然后按顺序执行,直到结束。在本章和第5章中,将会学习如何指示计算机不必简单地按照序列顺序来执行语句,它也可以做出决定并且能多次重复执行特定的语句,直到满足特定的条件为止。

4.1 控制结构

计算机可以使用下面三种方法中的一种来执行一个程序:顺序执行;选择执行,也称为分支;循环执行,它使用一种称为循环(如图4.1所示)的结构重复地执行语句。在第2章和第3章中包含了简单的顺序执行程序。在这些程序中,计算机从开始并且按照顺序执行语句。不存在选择和循环结构。控制结构为顺序执行提供了选择方法,并且可以用来改变顺序结构的执行流程。两种最常用的控制结构是选择结构和循环结构。

在选择结构中,程序根据一些条件来执行特定的语句;在循环结构中,程序重复执行由一些条件所规定的语句。

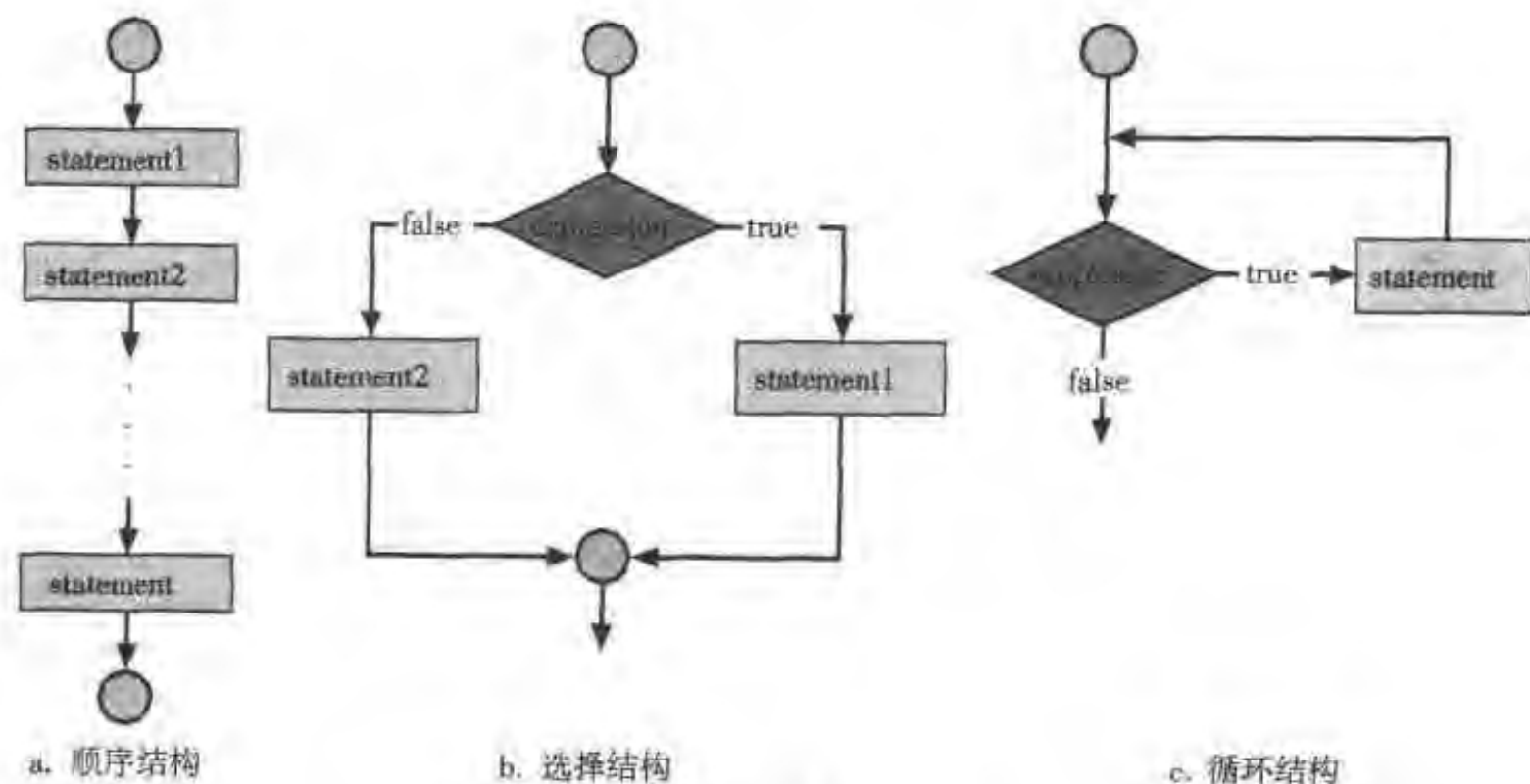


图 4.1 执行流程

在学习选择结构和循环结构之前,必须先了解条件语句的特性以及如何使用它们。考虑下面的语句:

```
1. if(score is greater than or equal to 90)
    grade is A
2. if(hours worked are less than or equal to 40)
    wages = rate * hours
    otherwise
    wages = (rate * 40) + 1.5 * (rate * (hours - 40))
3. if(temperature is greater than 70 degrees and it is not
    raining)
    recommended activity is golfing
```

这些语句是条件语句的一些例子。可以看出只有满足特定的条件,相应的语句才被执行。如果条件的值为 true,那么条件才被满足。例如,在语句 1 中:

```
score is greater than or equal to 90
```

如果 score 的值大于或等于 90,那么该语句的值为 true;否则为 false。例如,如果 score 的值为 95,那么语句的值为 true。同样,如果 score 的值为 86,那么语句的值为 false。

如果计算机能将这些类型的语句识别为正确的值,那么这将是很有用的。而且,在特定的情况下,一条语句的正确或错误可能依赖于不止一个条件。例如,在语句 3 中,temperature is greater than 70 degrees 和 it is not raining 必须都是正确的情况下,推荐的运动才是打高尔夫球。

从这些例子可以看出,为了执行选择或循环中的语句,计算机必须能对程序执行过程中存在的条件做出判断。下面的几个部分将讨论怎样在 C++ 中表示和计算条件语句的值。

4.2 关系运算符

为了做出选择,必须对相应条件进行比较。例如,某账户的利率和服务费可能依赖于月底的余额。如果余额小于最低余额限度,那么不仅利率比较低,而且通常可能需要服务费。所以,为了决定利率,必须声明最低余额限度(条件)并且将账目余额与最低余额限度进行比较。保险政策中,保险费也是由声明的条件和所进行的比较来决定的。例如,为了确定保险费,必须检查投保人的吸烟情况。非吸烟(条件)者比吸烟者得到更少的保险费。这两个例子都包含了比较项。将某项与特定条件进行比较,并判断是否相等、大于或小于该条件。

在 C++ 中,条件由逻辑(布尔)表达式表示。值为 true 或 false 的表达式被称为逻辑表达式(布尔)。而且, true 和 false 是逻辑(布尔)值。假设 I 和 J 是整数,考虑下面的表达式:

```
I > J
```

如果该表达式是一个逻辑表达式,那么如果 I 的值大于 J,则表达式的值为 true;否则,它的值为 false。可以将符号 > 认为是一个运算符,在这种情况下,其操作数为整数,产生的结果为逻辑值。这里,符号 > 类似于能产生整数或实数结果的 + 和 - 运算符。实际上,符号 > 称为关系运算符,因为 I > J 的值只有当 I 和 J 之间的关系保持为大于时,它的值才为 true。关系运算符允许在程序中进行比较。

C++ 包含了 6 种关系运算符,从而可以构造条件并进行比较。表 4.1 列出了关系运算符。

表 4.1 C++ 中的关系运算符

操作符	描述
==	相等
!=	不等
<	小于
<=	小于等于
>	大于
>=	大于等于

注意：在 C++ 中，符号 == 称为等于运算符，它包含了两个等号。前面讲过，= 称为赋值运算符。记住，等于运算符 == 确定两个表达式是否相等，而赋值操作符 = 将一个表达式的值赋值给变量。

每一种关系运算符都是二元运算符，即它需要两个操作数。因为比较的结果是 true 或 false，所以表达式使用这些运算符来计算值为 true 或 false。

4.2.1 关系运算符和简单数据类型

可以对简单数据类型使用关系运算符。例如，下面的表达式使用了整数和实数。

表达式	描述	值
8 < 15	8 小于 15	true
6 != 6	6 不等于 6	false
2.5 > 5.8	2.5 大于 5.8	false
5.9 <= 7.5	5.9 小于等于 7.5	true

注意：实数相等的比较通常是依赖于计算机的，记住这一点很重要。在一台计算机中， $6.8+3.1==2.7+7.2$ 的值为 false 是很可能的。

对于 char 类型的值，使用关系运算符时，表达式的值是 true 还是 false 取决于计算机的对照序列。表 4.2 显示了使用 ASCII 数据集的表达式是如何计算值的。

表 4.2 使用关系运算符和 ASCII 数据集计算表达式的值

表达式	值	描述
' ' < 'a'	true	' ' 和 'a' 的 ASCII 码值分别为 32 和 97。32 < 97，所以值为 true
'R' > 'T'	false	'R' 和 'T' 的 ASCII 码值分别为 82, 84。82 > 84，所以表达式值为 false
'+' < '*'	false	'+' 和 '*' 的 ASCII 码值分别为 43, 42。43 < 42，所以表达式值为 false
'6' <= '>'	true	'6' 和 '>' 的 ASCII 码值分别为 54, 62。54 <= 62，所以表达式值为 true

比较不同数据类型的值会产生无法预测的结果。例如，下面的表达式比较一个整数和一个字符：

8 < '5'

因为结果是无法预测的，所以在程序中不应该使用包含不同数据类型的比较。在前面的表达式中，对于某一台特定的计算机，8 将会与 '5' 的对照序列，即 53 进行比较。形如 4 < 6 和 'R' > 'T' 的表达式是逻辑（布尔）表达式的一些例子。当 C++ 计算一个逻辑表达式的值时，如果逻辑表达式的值为 true，那么它返回整数值 1，否则返回整数值 0。在 C++ 中，任何非零值都被认为是 true。

注意：在第 2 章中介绍了 bool 数据类型。前面讲过，bool 数据类型有两个值：true 和 false。在 C++ 中，true 和 false 是保留字。标识符 true 设置为 1，标识符 false 设置为 0。如果表达式的值为 true 或非零，那么该逻辑表达式的值为 1，尽管如此，为了易读，无论什么时候用到逻辑表达式，在本书中标识符 true 和 false 都将用来作为该逻辑表达式的值。

4.2.2 关系运算符和 string 类型

关系运算符能应用于 string 类型的变量。string 类型变量间的比较是逐个字符进行的，从第一个字符开始并且使用 ASCII 对照序列，直到发现不匹配或最后一个字符已经被比较并且相等，逐个字符的比较才会停止。考虑下面的声明：

```
string str1 = "Hello";
string str2 = "Hi";
string str3 = "Air";
string str4 = "Bill";
string str5 = "Big";
```

使用这些变量声明，表 4.3 显示了不同的逻辑表达式是怎样被计算值的。

表 4.3 计算使用字符串变量的表达式的值

表达式	值
str1 < str2	true
str1 > "Hen"	false
str3 < "An"	true
str1 == "hello"	false
str3 <= str4	true

如果比较两个不同长度的字符串，并且直到较短字符串的最后，字符的比较仍是相等的，那么较短字符串的值会小于较长字符串的值，例如：

表达式	值
str4 >= "Billy"	false
str5 <= "Bigger"	true

4.3 逻辑（布尔）运算符和逻辑表达式

该部分讨论怎样来组成并且计算逻辑表达式的值。逻辑（布尔）运算符允许将逻辑表达式结合起来。C++ 有三种逻辑（布尔）运算符，如表 4.4 所示。

表 4.4 C++ 中的逻辑（布尔）运算符

运算符	描述
!	取反
&&	逻辑与
	逻辑或

逻辑运算符只将逻辑值作为操作数并且只产生逻辑值。运算符!是取反操作，所以它只有一个操作数。运算符&&和||是二元运算符。表 4.5，表 4.6 和表 4.7 定义了这些运算符。

表 4.5 定义了运算符!。表 4.5 显示了当使用!运算符时,!true 是 false,!false 是 true。将!放在逻辑表达式的前面会将该表达式的值取反。

表 4.5 !（取反）运算符

表达式	!(表达式)
true (非 0 值)	false (0)
false (0)	true (1)

例 4.1

表达式	值	描述
!('A' > 'B')	true	'A' > 'B' 为 false，所以表达式值为 true
!(6 <= 7)	false	6 <= 7 为 true，所以表达式值为 false

表4.6定义了运算符&&(与)。从该表中可以看出,其计算值的方法为:只有当表达式1和表达式2的值同为true时,表达式1&&表达式2的值才为true,否则表达式1&&表达式2的值为false。

表 4.6 && (与) 运算符

表达式 1	表达式 2	表达式 1 && 表达式 2
true (非 0 值)	true (非 0 值)	true (1)
true (非 0 值)	false (0)	false (0)
false (0)	true (非 0 值)	false (0)
false (0)	false (0)	false (0)

例 4.2

表达式	值	描述
(14 >= 5) && ('A' < 'B')	true	14 >= 5 和 'A' < 'B' 值分别为 true, 所以, 表达式值为 true
(24 >= 35) && ('A' < 'B')	false	24 >= 35 值为 false, 'A' < 'B' 值为 true, 所以, 表达式值为 false

表4.7定义了运算符|| (或)。从该表可以看出,如果至少有一个表达式的值为true,那么表达式1||表达式2的值为true。否则表达式1||表达式2的值为false。

表 4.7 || (或) 运算符

表达式 1	表达式 2	表达式 1 表达式 2
true (非 0 值)	true (非 0 值)	true (1)
true (非 0 值)	false (0)	true (1)
false (0)	true (非 0 值)	true (1)
false (0)	false (0)	false (0)

例 4.3

表达式	值	描述
(14 >= 5) ('A' > 'B')	true	14 >= 5 值为 true, 'A' > 'B' 值为 false, 所以表达式为 true
(24 >= 35) ('A' > 'B')	false	24 >= 35 值为 false, 'A' > 'B' 值为 false, 所以表达式值为 false
('A' <= 'a') (7 != 7)	true	'A' <= 'a' 值为 true, 7 != 7 值为 false, 所以表达式值为 true

4.3.1 优先级顺序

复合表达式求值时,情况将会复杂一些。考虑下面的逻辑表达式:

11 > 5 || 6 < 15 && 7 >= 8

逻辑表达式可能会产生不同的结果,这取决于先计算||还是先计算&&。如果首先计算||,那么它的值为false;如果首先计算&&,那么它的值为true。

为了使复合逻辑表达式能有效工作,必须有一个优先级方案来决定首先计算哪个运算符的值。因为表达式可能包含算术、关系和逻辑运算符(例如在表达式5+3<=9&&2>3中),所以需要为所有的运算符建立一个优先级的顺序。表4.8显示了一些C++运算符的优先级顺序,包括算术、关系和逻辑运算符。(参看附录B以得到所有C++运算符的优先级。)

表 4.8 运算符优先级

运算符	优先级
!, +, - (一元运算符)	1
*, /, %	2
+, -	3

(续表)

运算符	优先级

<, <=, >=, >	4
==, !=	5
&&	6
	7
= (赋值运算符)	最低

注意：在 C++ 中，& 和 | 也是运算符。这些运算符的含义与 && 和 || 的含义很不相同。可能由于输入的错误，在 && 的地方使用 & 或在 || 的地方使用 | 将会产生很奇怪的结果。

在表达式中使用表 4.8 给出的优先级规则，关系和逻辑运算符将会从左至右被计算，所以可以说这些运算符的结合性是从左至右的。

例 4.4 演示了包含变量的逻辑表达式是如何计算值的。

例 4.4

假定有下面的声明：

```
bool found = true;
bool flag = false;
int num = 1;
double x = 5.2;
double y = 3.4;
int a = 5, b = 8;
int n = 20;
char ch = 'B';
```

考虑下面的表达式：

表达式	值	描述
!found	false	found 值为 true，所以 !found 为 false
x > 4.0	true	x = 5.2，并且 5.2 > 4.0 值为 true，所以表达式值为 true
!num	false	num 为非 0 值 1，所以表达式值为 false
!found && (x >= 0)	false	表达式中，!found 值为 false，5.2 >= 0 值为 true，所以表达式值为 false
!(found && (x >= 0))	false	表达式中，found && (x >= 0) 值为 true，所以原表达式值为 false
x + y <= 20.5	true	x + y = 5.2 + 3.4 = 8.6，同时 8.6 <= 20.5 值为 true，所以表达式值为 true
(n >= 0) && (n <= 100)	true	n = 20 时，n >= 0 和 n <= 100 值分别为 true，所以表达式值为 true
('A' <= ch && ch <= 'Z')	true	ch = 'B' 时，'A' <= ch 和 'A' <= 'Z' 值分别为 true，所以表达式值为 true
(a + 2 <= b) && !flag	true	a = 5, b = 8, flag = false 时，a + 2 <= b 与 flag 表达式值分别为 true，所以原表达式值为 true

也可以编写 C++ 程序来计算例 4.4 给出的逻辑表达式的值。如例 4.5 所示。

例 4.5

下面的程序计算并输出了例 4.4 给出的逻辑表达式的值。注意如果逻辑表达式的值为 true，那么相应的输出为 1；如果逻辑表达式的值为 false，那么相应的输出为 0，如程序最后的输出所示。前面讲过如果逻辑表达式的值为 true，那么它的值为 1，如果逻辑表达式的值为 false，那么它的值为 0。

```
//Chapter 4: Logical operators
#include <iostream>
using namespace std;
int main()
{
    bool found = true;
    bool flag = false;
    int num = 1;
    double x = 5.2;
```

```

double y = 3.4;
int a = 5, b = 8;
int n = 20;
char ch = 'B';

cout<<"Line 1: !found evaluates to "
    <<!found<<endl; //Line 1
cout<<"Line 2: x > 4.0 evaluates to "
    <<(x > 4.0)<<endl; //Line 2
cout<<"Line 3: !num evaluates to "
    <<!num<<endl; //Line 3
cout<<"Line 4: !found && (x >= 0) evaluates to "
    <<(!found && (x >= 0))<<endl; //Line 4
cout<<"Line 5: !(found && (x >= 0)) evaluates to "
    <<(!(found && (x >= 0)))<<endl; //Line 5
cout<<"Line 6: x + y <= 20.5 evaluates to "
    <<(x + y <= 20.5)<<endl; //Line 6
cout<<"Line 7: (n >= 0) && (n <= 100) evaluates to "
    <<((n >= 0) && (n <= 100))<<endl; //Line 7
cout<<"Line 8: ('A' <= ch && ch <= 'Z') evaluates to "
    <<('A' <= ch && ch <= 'Z')<<endl; //Line 8
cout<<"Line 9: (a + 2 <= b) && !flag evaluates to "
    <<((a + 2 <= b) && !flag)<<endl; //Line 9

return 0;
}

```

输出:

```

Line 1: !found evaluates to 0
Line 2: x > 4.0 evaluates to 1
Line 3: !num evaluates to 0
Line 4: !found && (x >= 0) evaluates to 0
Line 5: !(found && (x >= 0)) evaluates to 0
Line 6: x + y <= 20.5 evaluates to 1
Line 7: (n >= 0) && (n <= 100) evaluates to 1
Line 8: ('A' <= ch && ch <= 'Z') evaluates to 1
Line 9: (a + 2 <= b) && !flag evaluates to 1

```

可以在表达式中运用括号以使它的意思更明确。也可以使用括号来重新更改运算符的优先级。使用标准的优先级顺序, 表达式:

```
11 > 5 || 6 < 15 && 7 >= 8
```

等价于:

```
11 > 5 || (6 < 15 && 7 >= 8)
```

表达式中, 11>5 值是 true, 6<15 值是 true, 7>=8 值是 false。在表达式 11>5||(6<15&&7>=8) 中替换这些值, 可以得到 true||(true&&false)=true||false=true。所以, 表达式 11>5||(6<15&&7>=8) 的值是 true。

例 4.6

计算下面表达式的值:

```
(17 < 4 * 3 + 5) || (8 * 2 == 4 * 4) && !(3 + 3 == 6)
```

计算过程如下:

```

(17 < 4*3+5) || (8*2 == 4*4) && !(3+3 == 6)
= (17 < 12+5) || (16 == 16) && !(6 == 6)
= (17 < 17) || true && !(true)

```

```
= false || true && false  
= false || false  
= false
```

所以, 原始逻辑表达式的值是 false, 即为 0。

4.3.2 简化求值

在C++的逻辑表达式中, 可以使用一种高效的算法来计算表达式的值。该算法可以通过下面语句加以演示:

1. $(x > y) \parallel (x == 5)$
2. $(a == b) \&\& (x >= 7)$

在第一条语句中, 运算符 $\parallel$ 的两个操作数是表达式 $(x > y)$ 和 $(x == 5)$ 。二者之中, 如果某一操作数值为 true, 该表达式的值就为 true。通过短路方法求值, 计算机从左至右计算逻辑表达式的值。只要知道整个逻辑表达式的值, 求值就会停止。例如, 在语句 1 中, 如果操作数 $(x > y)$ 的值为 true, 那么整个表达式的值就为 true, 因为 $\text{true} \parallel \text{true}$ 为 true, $\text{true} \parallel \text{false}$ 也为 true。所以, 不必计算操作数 $(x == 5)$ 的值。

同样在语句 2 中, 运算符 $\&\&$ 的两个操作数为 $(a == b)$ 和 $(x >= 7)$ 。如果操作数 $(a == b)$ 的值为 false, 那么整个表达式的值就为 false, 因为 $\text{false} \&\& \text{true}$ 为 false, $\text{false} \&\& \text{false}$ 也是 false。

简化求值 (逻辑表达式): 在该过程中, 计算机从左至右计算逻辑表达式的值, 一旦知道表达式的值, 它就会停止。

例 4.7

考虑下面的表达式:

1. $(5 >= 3) \parallel (x == 5)$
2. $(2 == 3) \&\& (x >= 7)$

在语句 1 中, 因为 $(5 >= 3)$ 是 true, 而且在表达式中使用的逻辑运算符是 $\parallel$, 所以表达式的值为 true。计算机不会计算 $(x == 5)$ 的值。同样在语句 2 中, 因为 $(2 == 3)$ 是 false, 并且在表达式中使用的逻辑运算符是 $\&\&$, 所以表达式的值为 false, 计算机并不计算 $(x >= 7)$ 的值。

在C++中, 逻辑 (布尔) 表达式可以使用两种方法中的一种来进行操纵或处理: 通过使用 int 变量或 bool 变量。下面的部分阐述了这些方法。

4.3.3 int 数据类型和逻辑 (布尔) 表达式

早期版本的C++没有提供包含逻辑 (或布尔) 值的内建数据类型, 即 true 和 false。因为逻辑表达式的值为 1 或 0, 所以逻辑表达式的值存储在 int 数据类型的变量中。所以, 可以使用 int 数据类型来操纵逻辑 (布尔) 表达式。

前面讲过, 非零值被当做 true。现在考虑下面的声明:

```
int legalAge;  
int age;
```

和赋值语句:

```
legalAge=21;
```

如果将 legalAge 作为一个逻辑变量,那么由该语句赋给 legalAge 的值为 true。

赋值语句:

```
legalAge=(age>=21);
```

如果 age 的值大于或等于 21,那么将值 1 赋给 legalAge; 如果 age 的值小于 21,那么语句将赋值为 0。

4.3.4 bool 数据类型和逻辑 (布尔) 表达式

新版本的 C++ 包含一种内建的数据类型 bool, 它的逻辑值为 true 和 false。所以, 可以使用 bool 数据类型来操纵逻辑 (布尔) 表达式。前面讲过, 在 C++ 中, bool, true 和 false 是保留字。另外, 标识符 true 的值为 1, 标识符 false 的值为 0。现在考虑下面的声明:

```
bool legalAge;
int age;
```

语句:

```
legalAge=true;
```

将变量 legalAge 的值设置为 true。如果 age 的值大于或等于 21, 那么语句:

```
legalAge=(age>=21);
```

将 legalAge 的值设置为 true; 如果 age 的值小于 21, 那么该语句将 legalAge 的值设置为 0。例如, 如果 age 的值是 25, 那么赋给 legalAge 的值是 true, 即 1。同样, 如果 age 的值为 16, 那么赋给 legalAge 的值将为 false, 即 0。

注意: 从前面的部分可以看出, 既可以使用 int 变量也可以使用 bool 变量来存储逻辑表达式的值。为了使目的更加明确, 本书使用 bool 变量来存储逻辑表达式的值。

有时逻辑表达式不能达到预期的目的。例如, 假设 num 是一个 int 变量。编写一个逻辑表达式来求值, 如果 num 的值在 0 和 10 之间, 包括 0 和 10, 那么该值为 true, 否则为 false。下面的表达式提供了一种形式用以比较 0, num 和 10:

```
0<=num<=10
```

尽管该语句是一个合法的 C++ 表达式, 但是可能不会得到预期的结果。如果 num 的值在 0 和 10 之间, 该表达式的值就是 true 吗? 如果表达式的值大于 10 又会发生什么情况呢? 假设 num=5, 那么:

```
0 <= num <= 10
= 0 <= 5 <= 10
= (0 <= 5) <= 10   (关系运算符为左结合)
= 1 <= 10           (0 <= 5 布尔值为 true, 即 1)
= 1 (true)
```

现在假设 num=20, 那么:

```
0 <= num <= 10
= 0 <= 20 <= 10
= (0 <= 20) <= 10  (关系运算符为左结合)
= 1 <= 10           (0 <= 20 布尔值为 true, 即 1)
= 1 (true)
```

显然, 答案是不正确的。因为 num 等于 20, 不在 0 和 10 之间, 所以 0<=20<=10 的值不会为 true。

注意该表达式的值总是为 true，而不论 num 的值是什么。这主要由于表达式 $0 \leq \text{num}$ 的值可以是 0，也可以是 1， $0 \leq 10$ 的值为 true， $1 \leq 10$ 的值为 true。那么表达式 $0 \leq \text{num} \leq 10$ 到底有什么错误呢？它丢失了逻辑运算符 &&。在 C++ 中编写该表达式正确的方法是：

```
0 <= num && num <= 10
```

当编写逻辑表达式时，必须注意这一问题。当创建一个复杂的逻辑表达式时，必须使用正确的逻辑运算符。

4.4 选择：if 和 if...else

尽管只有两个逻辑值 true 和 false，但是它们也被证明是极其有用的，因为这允许程序包含决策处理来改变程序执行的流程。本章的剩余部分讨论将判断包含到程序中的方法。在 C++ 中，有两种选择结构，或称为分支控制结构：if 语句和 switch 结构。下面将讨论如何使用 if 和 if...else 语句来创建单分支选择、双分支选择和多重选择。switch 结构将在本章的后面进行讨论。

4.4.1 单分支选择

如果顾客的账户余额低于所需的最小余额限度，银行将会向顾客发送一个通知。同样，保险公司条例中，如果投保人是一个非吸烟者，那么公司将会对保险费给予 10% 的优惠。这两个例子都包含了单分支循环。在 C++ 中，程序中包含的单分支选择结构使用 if 语句。单分支选择结果的语法是：

```
if (表达式)
    statement
```

注意该语法的基本元素。它从保留字 if 开始，然后是括号中的表达式，最后是语句。表达式有时被称为决策者，因为它决定了是否要执行后面的语句。表达式通常是一个逻辑表达式。如果表达式的值为 true，那么相应的语句得以执行。如果值为 false，那么语句不会执行，计算机会继续执行程序中的下一条语句。表达式后面的 statement 有时称为执行语句。图 4.2 显示了 if 语句的执行流程（单分支选择）。

例 4.8

```
if (score >= 90)
    grade = 'A';
```

在该代码中，如果表达式 $(\text{score} \geq 90)$ 的值为 true，那么赋值语句 “grade='A';” 就会执行。如果表达式的值为 false，那么直接执行 if 结构后面的语句（如果有的话）。例如，如果 score 的值为 95，那么赋给变量 grade 的值是 'A'。

例 4.9

下面的 C++ 程序计算一个整数的绝对值：

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

int main ()
{
    int number;

    cout << "Please enter an integer--> ";    //Line 1
    cin >> number;                            //Line 2
```

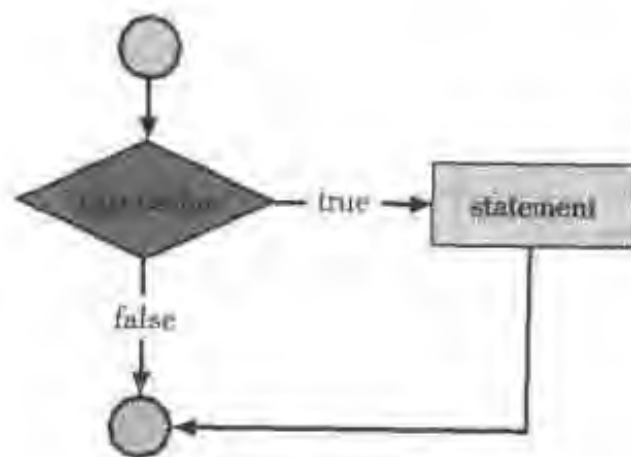


图4.2 单分支选择


```

    if(number < 0)                //Line 3
        number = -number;        //Line 4

    cout<<endl<<"The absolute value is "
        <<number<<endl;        //Line 5
    return 0;
}

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Please enter an integer --> -6734
The absolute value is 6734

```

第1行语句提示用户输入一个整数；第2行语句将数字输入给变量 `number`。第3行语句检查 `number` 是否为负数，第4行语句将 `number` 转换为正数。第5行语句输出 `number`，它是原数据的绝对值。

例 4.10

考虑下面的语句：

```

if score >= 90    //syntax error
    grade = 'A';

```

该语句演示了 `if` 语句的错误版本。缺少逻辑表达式的括号将会导致一个语法错误。

`if` 语句（即在 `statement` 前）中，在表达式的括号后面放入分号是一种语法错误。如果分号跟在括号的后面，那么 `if` 语句将会执行空语句。

例 4.11

考虑下面的 C++ 语句：

```

if(score>=90);    //Line 1
    grade='A';    //Line 2

```

该语句提供了单分支选择。因为在表达式（参看第1行）后面有一个分号，所以 `if` 语句在第1行终止，`if` 语句中的执行动作是空的，第2行的语句不是第1行 `if` 语句的一部分。这样无论 `if` 语句的值如何被计算，第2行语句都会执行。

4.4.2 双分支选择

在前面部分中学习了怎样在一个程序中执行单分支选择。在两个判断之间进行选择有很多种情况。例如，如果一名兼职雇员加班工作，那么应该使用加班报酬公式来计算工资，否则应该使用常规公式来计算工资。这是双分支选择的一个例子。为了在两者中进行选择，即为了执行双分支选择，C++ 提供了 `if...else` 语句。双分支选择使用下面的语法：

```

if(expression)
    statement1
else
    statement2

```

花一点功夫来研究一下这种语法。它从保留字 `if` 开始，然后是括号内的逻辑表达式，接着是 `statement1`，然后是保留字 `else`，最后是 `statement2`。`statement1` 和 `statement2` 可以是任何有效的 C++ 语句。在双分支选择中，如果表达式的值为 `true`，那么 `statement1` 执行。如果表达式的值为 `false`，那么 `statement2` 执行。图 4.3 显示了 `if...else` 语句（双分支选择）的执行流程。

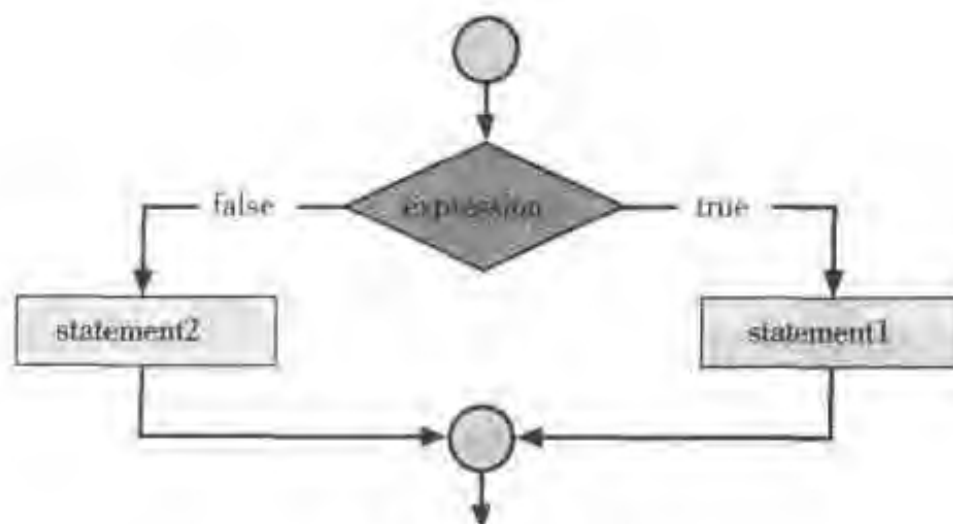


图 4.3 双分支选择

例 4.12

考虑下面的语句：

```

if(hours > 40.0)                //Line 1
    wages = 40.0 * rate +
        1.5 * rate * (hours - 40.0); //Line 2
else                             //Line 3
    wages = hours * rate;        //Line 4
  
```

如果变量 `hours` 的值大于 40.0，那么 `wages` 包含加班报酬。假设 `hours` 是 50。第 1 行 `if` 语句的表达式值为 `true`，所以第 2 行语句执行。另一方面，如果 `hours` 是 30，或者任何小于 40 的数字，那么第 1 行 `if` 语句表达式的值为 `false`。在这种情况下，程序跳过第 2 行语句，执行第 4 行的语句，即保留 `else` 后面的语句执行。

在双分支选择语句中，在表达式后和 `statement1` 前面添加分号会产生语法错误。如果 `if` 语句以分号结尾，那么 `statement1` 不再是 `if` 语句的一部分，`if-else` 语句的 `else` 部分被分离出来。在 C++ 中没有单独的 `else` 语句，即它不能与 `if` 语句分开。

例 4.13

下面的语句显示了语法错误的一个例子：

```

if(hours > 40.0);                //Line 1
    wages = 40.0 * rate +
        1.5 * rate * (hours - 40.0); //Line 2
else                             //Line 3
    wages = hours * rate;        //Line 4
  
```

因为分号跟在 `if` 语句（第 1 行）括号的后面，所以 `else` 语句是单独的。在 `if` 语句末尾的分号结束了 `if` 语句，所以第 2 行语句将 `else` 语句与 `if` 语句分隔开，即 `else` 语句总是自己本身。由于在 C++ 中没有 `else` 语句，所以该代码产生一个语法错误。

例 4.14

下面的程序将计算一名雇员的周薪：

```

//Program: Weekly wages
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double wages, rate, hours;
  
```

```

cout<<fixed<<showpoint<<setprecision(2);           //Line 1
cout<<"Line 2: Enter working hours and rate: ";       //Line 2
cin>>hours>>rate;                                     //Line 3

if(hours > 40.0)                                       //Line 4
    wages = 40.0 * rate +
        1.5 * rate * (hours - 40.0);                 //Line 5
else                                                  //Line 6
    wages = hours * rate;                             //Line 7

cout<<endl;                                           //Line 8
cout<<"Line 9: The wages are $"<< wages<<endl;       //Line 9

return 0;
}

```

运行实例：在本运行实例结果中，用户输入用阴影表示。

Line 2: Enter working hours and rate: 56.45 12.50

Line 9: The wages are \$808.44

第1行语句用固定的小数格式，即使用小数点、尾零和两个小数位设置了浮点数的输出。第2行语句提示用户输入工作小时数和薪水率。第3行语句分别将这些值输入到变量hours和rate中。第4行语句检查变量hours的值是否大于40.0。如果hours大于40.0，那么工资由第5行语句来计算，它包含了加班报酬。否则，工资由第7行语句来计算。第9行语句输出工资。

现在考虑if语句的其他例子并且检查可能产生的语义错误。

例 4.15

考虑下面的语句：

```

if(score >= 90)
    grade = 'A';
cout<<"The grade is "<<grade<<endl;

```

这些语句包含了一个语义错误。if语句只有一条语句可用，即“grade='A';”。无论(score>=90)的值是true还是false，cout语句都会执行。

例4.16演示了另一种常见的错误。

例 4.16

考虑下面的语句：

```

if(score >= 60)
    cout<<"Passing"<<endl;
    cout<<"Failing"<<endl;

```

若表达式score>=60的值为false，那么输出应该是Failing。也就是说，该集合的语句执行与else语句相同的动作。它将会执行第二条cout语句，而不是第一条语句。例如，如果score的值是50，这些语句将会输出下面的结果：

Failing

然而，如果表达式score>=60的值为true，程序将会输出两条语句，从而给出一个很不能令人满意的结果。例如，如果score的值是70，这些语句将会输出下面的结果：

Passing

Failing

输出 Passing 或 Failing 的正确代码（依赖于 score 的值）是：

```
if(score >= 60)
    cout<<"Passing"<<endl;
else
    cout<<"Failing"<<endl;
```

4.4.3 复合（块）语句

if 和 if...else 结构每次只控制一条语句。然而，假设在 if 或 if...else 语句中，表达式值为 true 的情况下希望执行多条语句。为了允许执行更复杂的语句，C++ 提供了一种称为复合语句或块语句的结构。复合语句具有下面的形式：

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

即复合语句包含花括号中的一系列语句。在 if 或 if...else 结构中，复合语句的功能就好像是单个语句。这样，简单的双分支选择类似于下面的代码：

```
if(age > 18)
    cout<<"Eligible to vote."<<endl;
else
    cout<<"Not eligible to vote."<<endl;
```

也可以包含复合语句，类似于下面的代码：

```
if(age > 18)
{
    cout<<" Eligible to vote."<<endl;
    cout<<" No longer a minor."<<endl;
}
else
{
    cout<<"Not eligible to vote."<<endl;
    cout<<"Still a minor."<<endl;
}
```

复合语句是很有用的并在本章后面大多数的结构化语句中用到。

4.4.4 多重选择：嵌套 if

前面的部分学习了怎样在一个程序中执行单/双选择。有一些问题可能需要执行多于两个的选择。例如，假设如果账户余额大于 50 000 美元时，利率是 7%；如果账户余额在 25 000 美元和 49 999.99 美元之间时利率是 5%；如果账户余额在 1000 美元和 24 999.99 美元之间时利率是 3%；其他情况时利率是 0%。特定的问题有 4 种选择，即多重选择结构。如果执行语句本身是一条 if 或 if...else 语句，那么可以在程序中使用 if...else 结构来包含多重选择。当一条控制语句在另一条控制语句内，就称为嵌套。

假定所有的变量被正确地声明，考虑下面的语句：

```
if(score >= 90)                                     //Line 1
```

```

        cout<<"The grade is A"<<endl;           //Line 2
    else                                         //Line 3
        if(score >= 80)                         //Line 4
            cout<<"The grade is B"<<endl;       //Line 5
        else                                     //Line 6
            if(score >= 70)                     //Line 7
                cout<<"The grade is C"<<endl;   //Line 8
            else                                 //Line 9
                if(score >= 60)                 //Line 10
                    cout<<"The grade is D"<<endl; //Line 11
                else                             //Line 12
                    cout<<"The grade is F"<<endl; //Line 13

```

这些语句演示了怎样使用嵌套 if...else 结构来包含多重选择。

嵌套 if...else 结构需要一个重要问题的答案: else 语句与哪个 if 配对? 前面讲过, C++ 中没有独立的 else 语句。每一个 else 必须与一个 if 配对。if 与 else 配对的规则如下:

将 else 与 if 配对: 在一个嵌套 if 语句中, C++ 将 else 与最近的、独立的 if 联系起来, 即将 else 与没有配对的、最近的 if 进行匹配。

在前面的例子中, 第 3 行的 else 与第 1 行的 if 相配对。第 6 行的 else 与第 4 行的 if 相配对。第 9 行的 else 与第 7 行的 if 相配对。第 12 行的 else 与第 10 行的 if 相配对。

为了避免过多的缩进, 一些程序员宁愿将上面的代码写成下面的形式:

```

if(score >= 90)
    cout<<"The grade is A"<<endl;
else if(score >= 80)
    cout<<"The grade is B"<<endl;
else if(score >= 70)
    cout<<"The grade is C"<<endl;
else if(score >= 60)
    cout<<"The grade is D"<<endl;
else
    cout<<"The grade is F"<<endl;

```

下面的例子将会演示各种方法并使用嵌套 if 结构来执行多重选择。

例 4.17

假设所有的变量已被正确地声明, 考虑下面的语句:

```

if(temperature >= 50)           //Line 1
    if(temperature >= 80)       //Line 2
        cout<<"Good day for swimming."<<endl; //Line 3
    else                         //Line 4
        cout<<"Good day for golfing."<<endl;  //Line 5
else                             //Line 6
    cout<<"Good day to play tennis."<<endl;  //Line 7

```

在这些程序代码中, 第 4 行的 else 与第 2 行的 if 相配对。第 6 行的 else 与第 1 行的 if 相配对。注意第 4 行的 else 不能与第 1 行的 if 相配对。否则, 第 2 行的 if 就会成为第 1 行 if 的执行语句部分, 这样就会将第 6 行的 else 语句独立起来。而且, 第 2 行到第 5 行的语句形成了第 1 行 if 的语句部分。

例 4.18

假设所有的变量已被正确地声明, 考虑下面的语句:

```

if(temperature >= 70)           //Line 1

```

```

    if(temperature >= 80)                //Line 2
        cout<<"Good day for swimming."<<endl; //Line 3
    else                                  //Line 4
        cout<<"Good day for golfing."<<endl; //Line 5

```

在该代码中，第4行的else与第2行的if相配对。注意对于第4行的else来说，最近的、独立的if是第2行的if。在这段代码中，第1行的if没有else语句并且是单分支选择。

例 4.19

假设所有的变量已被正确地声明，考虑下面的语句：

```

    if(GPA >= 2.0)                        //Line 1
        if(GPA >= 3.9)                    //Line 2
            cout<<"Dean\'s Honor List."<<endl; //Line 3
    else                                  //Line 4
        cout<<"Current GPA below graduation requirement. "
        <<"\nSee your academic advisor."<<endl; //Line 5

```

该代码显得很别扭。按照else与if相对应的原则，第4行的else与第2行的if相配对。然而，该配对产生了一个不能令人满意的结果。假设GPA是3.8。第1行if中的表达式值是true，这样该if中的部分语句，即一个if...else结构将得到执行。因为GPA是3.8，所以第2行的表达式值是false，这样与该if（第2行）相联系的else语句执行，产生下面的输出：

```

Current GPA below graduation requirement.
see you academic advisor.

```

然而，GPA为3.8的学生将会毕业而且还会获得某项荣誉。实际上，当GPA小于2.0时，代码试图输出下面的信息：

```

Current GPA below graduation requirement.
see you academic advisor.

```

如果GPA大于或等于3.9，则输出信息：

```

Dean's Honor List.

```

为了取得这样的结果，第4行的else需要与第1行的if配对并使用下面的复合语句：

```

    if(GPA >= 2.0)                        //Line 1
    {
        if(GPA >= 3.9)                    //Line 2
            cout<<"Dean\'s Honor List."<<endl; //Line 3
    }
    else                                  //Line 4
        cout<<"Current GPA below graduation requirement. "
        <<"\nSee your academic advisor."<<endl; //Line 5

```

这种情况下，通常不能在一个块中（即花括号中）查看else与if是否配对。第4行的else不能与第2行的if配对，因为第2行的if语句位于花括号中，并且第4行的语句不能匹配花括号中的内容。所以，第4行的else与第1行的if配对。

4.4.5 多个if语句和if...else语句

考虑下面的C++程序段，所有的程序段都是完成同样的任务：

```

(a)
    if(month == 1)                        //Line 1

```



```

    cout<<"January"<<endl;           //Line 2
else if(month == 2)                 //Line 3
    cout<<"February"<<endl;         //Line 4
else if(month == 3)                 //Line 5
    cout<<"March"<<endl;           //Line 6
else if(month == 4)                 //Line 7
    cout<<"April"<<endl;           //Line 8
else if(month == 5)                 //Line 9
    cout<<"May"<<endl;             //Line 10
else if(month == 6)                 //Line 11
    cout<<"June"<<endl;           //Line 12

```

(b)

```

if(month == 1)
    cout<<"January"<<endl;
if(month == 2)
    cout<<"February"<<endl;
if(month == 3)
    cout<<"March"<<endl;
if(month == 4)
    cout<<"April"<<endl;
if(month == 5)
    cout<<"May"<<endl;
if(month == 6)
    cout<<"June"<<endl;

```

程序段(a)是由一系列的 if-else 语句编写的；程序段(b)是由一系列的 if 语句编写的。这两个程序段完成同样的任务。如果 month 等于 3，那么两个程序段都输出 March。如果 month 为 1，那么在程序段(a)中第 1 行 if 语句的值是 true。与该 if 相联系的语句（第 2 行）就会执行；其他结构中的语句，即该 if 语句的 else 将被忽略，剩余的 if 语句也就不会被求值。在程序段(b)中，计算机必须计算每一条 if 语句的值，因为没有 else 语句。所以，程序段(b)比程序段(a)执行得慢。

4.4.6 使用伪代码来开发、测试和调试程序

有几种方法来开发一个程序。一种方法是使用 C++ 和自然语言的非正式混合体，称为伪代码。有时在将伪代码转化为正式的 C++ 代码之前，它提供了一种很有效的手段来概括和精炼一个程序。当创建一段包含复合嵌套的程序时，使用伪代码可以快速地编写出正确的结构并可以避免一般性错误：

```

a. if (x > y) then
    x is larger
b. if (y > x) then
    y is larger

```

如果(a)中的语句是 true，那么 x 值较大；如果(b)中的语句是 true，那么 y 值较大。然而，对于协同两个条件并计算两个整数较大值的代码，即使第一条语句是 true，计算机也需要计算两个表达式的值：

```
(x>y) and (y>x)
```

计算两个表达式的值会浪费计算机时间。可以将该伪代码重写成如下形式：

```

if(x > y) then
    x is larger
else
    y is larger

```


这只需要计算一个条件的值。该伪代码看起来没问题，将它转换为 C++ 代码。

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

int main()
{
    if (x > y)
```

一旦开始将伪代码转换成一个 C++ 程序，就应该立刻注意到没有定义变量存储 x 或 y 的值。变量没有声明，这是一种很常见的错误，特别是对于一名新的程序员。如果检查该伪代码，就会看到该程序需要三个变量，最好为它们编写出自解释文档。

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

int main()
{
    int num1, num2, larger;           //Line 1
    if(num1 > num2);                   //Line 2; error
        larger = num1;                //Line 3
    else                               //Line 4
        larger = num2;                //Line 5

    return 0;
}
```

编辑该程序将会产生一个常见的语法错误（第 2 行）。前面讲过，分号不能出现在 if...else 语句中表达式的后面。然而，即使改正了这一语法错误，程序仍然不会给出正确的结果，因为它所使用的变量没有值。变量没有被初始化，这是另一种常见的错误。另外，因为没有输出语句，用户不能看到程序的结果。

因为在程序中可能有很多的错误，要一步步检查它到底是否能工作。在此过程中，应该尝试在各种情况下使用不同的值。例如，如果一个数字是 0，如果一个数字是负数，另一个数字是正数，如果两个数字都是负数，或者如果两个数字相同，该程序会有效吗？通过检查可以看出，该程序不能判断出这两个数字是否相等。考虑所有的情况，可以重写出下面的程序：

```
//Program: Compare Numbers
//This program compares two integers and finds the largest.

#include <iostream>
using namespace std;

int main()
{
    int num1, num2, larger;

    cout<<"Enter any two integers:
    cin>>num1>>num2;
    cout<<endl;

    cout<<"The two integers entered are "<<num1
        <<" and "<<num2<<endl;

    if(num1 > num2)
    {
        larger = num1;
        cout<<"The larger number is "<<larger<<endl;
    }
}
```

```
    else if(num2 > num1)
    {
        larger = num2;
        cout<<"The larger number is "<<larger<<endl;
    }
    else
        cout<<"Both numbers are equal."<<endl;
    return 0;
}
```

运行实例：在运行实例中，用户输入用阴影表示。

```
Enter any two integers: 78 90
The two integers entered are 78 and 90.
The larger number is 90.
```

从前面程序中可了解到：首先要使用笔和纸来开发程序。尽管在一张纸上首先写出的程序在第一次运行时不能保证成功，但是这一步骤却是一个好的开始。在纸上，可以很容易地检查出错误并且改进程序，特别是对于一个复杂程序。

4.4.7 输入错误和 if 语句

在第3章中，我们知道试图读无效的数据会使输入数据流进入失败状态。一旦输入流进入失败状态，所有与该输入流联系的后续输入语句都会被忽略，而计算机将继续工作，这样就会产生错误的结果。可以使用if语句来检查一个输入流变量的状态，如果输入流进入失败状态，可以包含指令来终止程序执行。

除了读入无效数据，其他事件也可能引起输入流进入失败状态。两种其他的常见原因如下所示：

- 试图打开不存在的输入文件
- 试图读文件结尾后的内容

解决这些输入错误的一种方法是检查输入流变量的状态。可以通过在一条if语句中将它作为一个逻辑表达式来检查输入流变量的状态。当输入流变量在一条if语句中作为一个表达式使用时，如果输入成功，它的值就为 true，如果发生输入错误，它的值就为 false。

如果输入成功，那么语句：

```
if(cin)
    cout<<"Input is OK."<<endl;
```

就会输出：

```
Input is OK.
```

同样，如果与流变量相关的输入失败，且 infile 是一个 ifstream 变量，那么语句：

```
if(!infile)
    cout<<"Input failed."<<endl;
```

将会输出：

```
Input failed.
```

假设一个输入流变量试图打开一个文件以向程序中输入数据。如果输入文件不存在，可以使用与 return 语句相连的输入流变量的值来终止程序，下面会对此进行讨论。

前面讲过，包含在 main() 函数中的最后一条语句是：

```
return 0;
```

当程序终止时，该语句向操作系统返回一个0值。0值表示程序正常终止，在程序执行期间没有错误产生。除了0之外的其他int类型的值也可以通过return语句返回给操作系统。然而，除了0之外的任何返回值都表示程序执行期间有错误产生。

return语句可以出现在程序中的任何地方。无论何时执行一条return语句，它都会立刻从相应的函数中返回。对于main()函数来说，当return语句执行时程序会终止。这样，无论输入流何时出现失败，都可以使用return语句的这些性质来终止main()函数。当一个程序试图打开一个输入文件时，这是特别有用的。考虑下面的语句：

```
ifstream infile;
infile.open("a:inputdat.dat"); //open inputdat.dat file
if(!infile)
{
    cout<<"Cannot open input file. "
        <<"The program terminates."<<endl;
    return 1;
}
```

假设文件inputdat.dat不存在。打开文件的操作失败，会引起输入流进入失败状态。作为一个逻辑表达式，文件流变量infile的值为false。因为infile的值为false，表达式!infile (if语句中)的值为true，if语句体执行。信息：

```
Cannot open input file.The program terminates.
```

被打印在屏幕上。return语句通过向操作系统返回值1从而终止了程序。

注意：前面讲过，一些系统可能需要将ios::nocreate作为第二个参数传递给打开语句，这样如果输入文件不存在，那么打开操作就会失败。在这种情况下，打开语句有下面的形式：

```
infile.open("a:inputdat.dat",ios::nocreate);
```

现在通过在第3章中的学生分数编程实例来使用与输入操作相应的代码。前面讲过，该程序在输入文件数据的基础上计算平均测试分数，然后将结果输入另一个文件。下面的程序代码与第3章的代码是相同的，但是它包含了“如果输入文件不存在，终止程序”的语句。

```
//Program: Average test score
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
int main()
{
    ifstream inFile; //input file stream variable
    ofstream outFile; //output file stream variable

    int test1, test2, test3, test4, test5;
    double average;
    char studentId;

    inFile.open("a:test.txt"); //open input file
    if(!inFile)
    {
        cout<<"Cannot open input file. "
```

```

        <<"The program terminates."<<endl;
        return 1;
    }

    outFile.open("a:testavg.out"); //open output file
    outFile<<fixed<<showpoint;
    outFile<<setprecision(2);

    cout<<"Processing data"<<endl;
    inFile>>studentId;
    outFile<<"Student Id: "<<studentId<<endl;
    inFile>>test1>>test2>>test3>>test4>>test5;
    outFile<<"Test scores: "<<setw(4)<<test1
        <<setw(4)<<test2<<setw(4)<<test3
        <<setw(4)<<test4<<setw(4)<<test5<<endl;

    average = static_cast<double>(test1 + test2 +
        test3 + test4 + test5) / 5.0;

    outFile<<"Average test score: "<<setw(6)
        <<average<<endl;

    inFile.close();
    outFile.close();
    return 0;
}

```

注意：为了使用标准 C++ 风格头文件重写该程序，将下面的语句：

```

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

```

替换为：

```

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>

```

而且必须使用 `setf()` 函数或 `setiosflags()` 操纵符来设置操纵符 `fixed` 和 `showpoint`。（参看第3章可以获得详细资料。）

4.4.8 等于运算符与赋值运算符之间的混淆

前面讲过，如果 `if` 语句中判断表达式的值为 `true`，那么 `if` 语句中的 `statement` 部分会执行。另外，`expression` 通常是一个逻辑表达式。然而，C++ 允许使用任何值为 `true` 或 `false` 的表达式作为 `if` 语句的表达式。考虑下面的语句：

```

if(x=5)
    cout<<"The value is five."<<endl;

```

在 `if` 语句中的表达式，即判断部分是 `x=5`。表达式 `x=5` 称为赋值表达式，因为运算符 `=` 出现在表达式中并且在表达式的末尾没有分号。

该表达式由下面的方法来计算值。首先，运算符 `=` 的右边被计算值，它等于 5。然后值 5 赋给

x。而且值 5，即 x 的新值也成为 if 语句中表达式的值，也就是赋值表达式的值。因为 5 是非零的，所以 if 语句中的表达式的值为 true，这样 if 语句的 statement 部分会输出：The value is five.

无论程序员阅历如何，几乎每一个人都犯过用 = 代替 == 的错误。这两种运算符经常被混淆的原因是很多编程语言用 = 作为等号运算符。这样，其他编程语言的经验可能会导致混淆。使用赋值运算符代替等于运算符的另一个原因是在输入程序时不小心造成的。

尽管赋值表达式可以用做表达式，但是在程序中使用赋值运算符代替等于运算符会产生很严重的结果。例如，假设汽车保险的折扣以驾驶记录为基础。驾驶记录 1 表示驾驶员没有出现事故，可以得到 25% 的折扣。只有 drivingcode 的值为 1 时，语句：

```
if(drivingCode == 1)
    cout<<"The discount on the policy is 25%."<<endl;
```

输出为：

```
The discount on the policy is 25%.
```

然而，语句：

```
if(drivingCode = 1)
    cout<<"The discount on the policy is 25%."<<endl;
```

总是输出：

```
The discount on the policy is 25%.
```

因为赋值表达式的右边等于 1，它是非零的，所以值为 true。这样，if 语句中表达式的值为 true，会输出下面一行文字：The discount on the policy is 25%。而且，值 1 赋给变量 drivingCode。假设在 if 语句执行前，变量 drivingCode 的值为 4，在 if 语句执行后，不仅输出是错误的，而且新值也会代替旧的驾驶代码。

= 代替 == 就像一个无声杀手。它不是一种语法错误，所以编译器不会警告出现错误。它是一种逻辑错误。

注意：使用 = 代替 == 能引起严重的后果，特别是在一条 if 循环语句中使用。第 5 章将会讨论循环结构。

在程序中，用等于运算符代替赋值运算符也可能产生错误。例如，假设 x, y 和 z 是 int 变量。语句：

```
x=y+z;
```

将表达式 y+z 的值赋给 x。语句：

```
x==y+z;
```

将表达式 y+z 的值与 x 的值进行比较，然而 x 的值仍然保持不变。如果程序中的其他地方计算的 x 值是 y+z，那么会产生逻辑错误，程序输出就不会正确，编译器将会对这种情况发出警告。编译器只对语法错误而不是逻辑错误提供了反馈。由于这个原因，当使用等于运算符和赋值运算符时必须额外当心。

4.4.9 条件运算符 (?:)

注意：读者可以直接跳过该部分。

特定的 if-else 语句可以使用 C++ 的条件运算符——一种更简洁的方式来编写。条件运算符写作?:, 它是一个三元运算符, 即表示它有三个参量。使用条件运算符的语法是:

```
expression1 ? expression2 : expression3
```

该类型的语句称为条件表达式。条件表达式由下面的方法来计算值: 如果 expression1 的值是一个非零整数 (即是 true), 那么条件表达式的结果是 expression2。否则条件表达式的结果是 expression3。

考虑下面的语句:

```
if(a >= b)
    max = a;
else
    max = b;
```

可以使用条件运算符来简化该 if-else 语句, 如下所示:

```
max = (a >= b) ? a : b;
```

4.5 switch 结构

前面讲过, 在 C++ 中有两种选择或分支结构。第一种选择结构是使用 if 或 if-else 语句执行的, 通常需要一个 (逻辑) 表达式的值。第二种选择结构不需要一个逻辑表达式的值, 称为 switch 结构。C++ 的 switch 结构可以在多个条件中进行选择。

switch 语句的常用语法是:

```
switch(expression)
{
    case value1: statements1
                break;
    case value2: statements2
                break;
    ...
    case valuen: statementsn
                break;
    default: statements
}
```

在 C++ 中, switch, case, break 和 default 是保留字。在 switch 结构中, 首先计算 expression 的值, 然后 expression 的值用来执行在保留字 case 后面指定语句中的动作。前面讲过在一个语法模板中, 阴影表示定义的可选部分。

尽管可以不需要, 但是 expression 通常是一个标识符。无论它是一个标识符还是一个表达式, 其值只能是整数。

expression 有时称为选择器。它的值决定了选择哪条语句来执行。一个特定的 case 值只能应用一次。一条或更多的语句可以跟在 case 标记后面, 所以不需要使用花括号将多条语句转化为单个的复合语句。break 语句也可以不必出现在每一条语句的后面。图 4.4 显示了 switch 语句的执行流程。

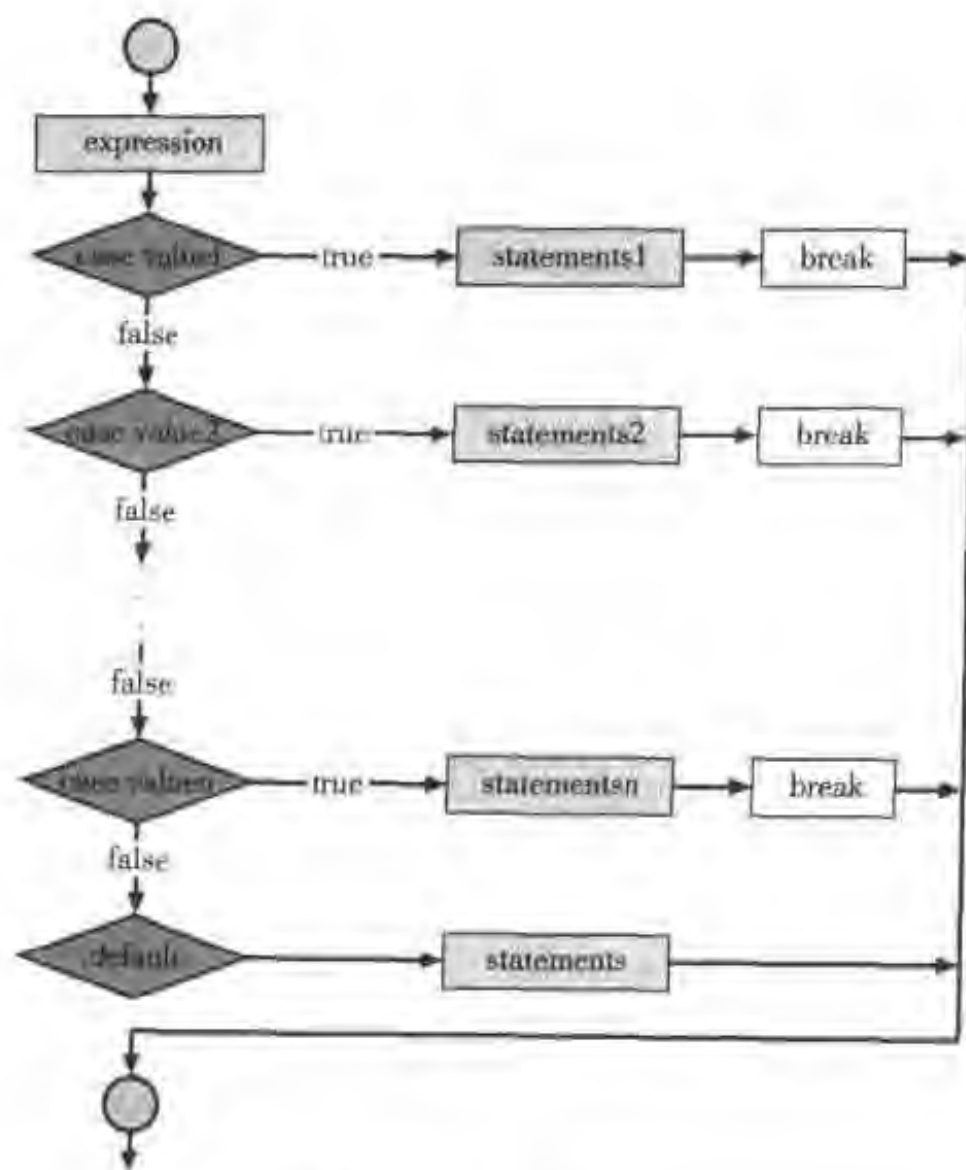


图 4.4 switch 语句

switch 语句根据下面的规则来执行：

1. 当 expression 的值与一个 case 值相匹配时，语句就会执行，直到遇到一条 break 语句或者到达 switch 结构的末尾。
2. 如果 expression 的值没有与任何 case 值相匹配，会执行跟在 default 后的语句。如果 switch 结构没有 default 标记，并且如果 expression 的值没有与任何的 case 值相匹配，那么会跳过整个 switch 语句。
3. break 语句会使程序立刻跳出 switch 结构。

例 4.20

考虑下面的语句：

```

switch(grade)
{
    case 'A': cout<<"The grade is A.";
              break;
    case 'B': cout<<"The grade is B.";
              break;
    case 'C': cout<<"The grade is C.";
              break;
    case 'D': cout<<"The grade is D.";
              break;
    case 'F': cout<<"The grade is F.";
              break;
    default:  cout<<"The grade is invalid.";
}
  
```


在该例中, switch 语句的表达式是一个变量标识符。变量 `grade` 是一个 `char` 类型的变量, 即一个整型变量。`grade` 的可能值是 'A', 'B', 'C', 'D' 和 'F'。每一个 `case` 标识都指定了要执行的相应语句, 这取决于 `grade` 的值。如果 `grade` 的值是 'A', 那么输出是:

The grade is A.

例 4.21

下面的程序演示了 `break` 语句的功能。它要求读者输入 0 到 10 之间的数字。

```
//Program: Effect of break statements in a switch structure
#include <iostream>
using namespace std;
int main()
{
    int a;

    cout<<"Enter an integer between 0 and 10: "; //Line 1
    cin>>a; //Line 2

    cout<<"\nThe number you entered is "<<a<<endl; //Line 3

    switch(a) //Line 4
    {
        case 0: //Line 5
        case 1: cout<<"Hello "; //Line 6
        case 2: cout<<"there. "; //Line 7
        case 3: cout<<"I am "; //Line 8
        case 4: cout<<"Mickey."<<endl; //Line 9
                break; //Line 10
        case 5: cout<<"How "; //Line 11
        case 6: //Line 12
        case 7: //Line 13
        case 8: cout<<"are you?"<<endl; //Line 14
                break; //Line 15
        case 9: break; //Line 16
        case 10: cout<<"Have a nice day."<<endl; //Line 17
                break; //Line 18
        default: cout<<"Sorry number is out of "
                <<"range."<<endl; //Line 19
    }
    cout<<"Out of switch structure."<<endl; //Line 20
    return 0;
}
```

运行实例

这些输出是通过多次执行上述程序得到的。在每一次输出中, 用户输入用阴影表示。

运行实例 1:

```
Enter an integer between 0 and 10: 0
The number you entered is 0
Hello there. I am Mickey.
Out of switch structure.
```

运行实例 2:

```
Enter an integer between 0 and 10: 1
The number you entered is 1
```

```
Hello there. I am Mickey.  
Out of switch structure.
```

运行实例 3:

```
Enter an integer between 0 and 10: 3  
The number you entered is 3  
I am Mickey.  
Out of switch structure.
```

运行实例 4:

```
Enter an integer between 0 and 10: 4  
The number you entered is 4  
Mickey.  
Out of switch structure.
```

运行实例 5:

```
Enter an integer between 0 and 10: 5  
The number you entered is 5  
How are you?  
Out of switch structure.
```

运行实例 6:

```
Enter an integer between 0 and 10: 7  
The number you entered is 7  
are you?  
Out of switch structure.
```

运行实例 7:

```
Enter an integer between 0 and 10: 9  
The number you entered is 9  
Out of switch structure.
```

运行实例 8:

```
Enter an integer between 0 and 10: 10  
The number you entered is 10  
Have a nice day.  
Out of switch structure.
```

运行实例 9:

```
Enter an integer between 0 and 10: 11  
The number you entered is 11  
Sorry number is out of range.  
Out of switch structure.
```

逐步浏览该程序将有助于理解 break 语句是怎样工作的。如果 a 的值是 0, 那么 switch 表达式的值应该与 case 值 0 相匹配。所有“case:0”后面的语句都被执行, 直到遇到 break 语句。

第一条 break 语句出现在第 10 行, 正好在 case 值等于 5 的前面。尽管 switch 表达式的值没有与任何 case 值相匹配 (即 1, 2, 3 或 4), 这些值后面的语句都被执行。

当 switch 表达式的值与一个 case 值相匹配, 其中所有语句都会执行, 直到遇到一条 break 语句, 程序会跳过其间所有的 case 标识。同样, 如果 a 的值是 3, 它将与 case 值 3 相匹配, 所有该标识后

的语句都会执行,直到遇到第10行的break语句。如果a的值是9,它与case值9相匹配。在这种情况下,执行语句是空的,因为在第16行,case值9后面只有break语句。

例 4.22

尽管switch结构的case值是有限的,但是switch语句中expression的复杂程度可以根据需要加以构造。例如,考虑下面的switch语句:

```
switch(score / 10)
{
case 0: case 1: case 2:
case 3: case 4: case 5:  grade = 'F';
                        break;

case 6: grade = 'D';
      break;

case 7: grade = 'C';
      break;

case 8: grade = 'B';
      break;

case 9: case 10: grade = 'A';
           break;

default: cout<<" Invalid test score."<<endl;
}
```

假定score是一个int变量,其值在0到100之间。如果score是75,那么 $score/10=75/10=7$,即grade赋值为'C'。如果score在0到59之间,那么grade是'F'。现在如果score在0和59之间,那么 $score/10$ 是0, 1, 2, 3, 4或5,这些值的每一个都对应“grade='F'”。

所以,在switch结构中,case0, case1, case2, case3, case4和case5的执行语句都是相同的。在case标识为0, 1, 2, 3, 4或5的语句中,不必分别采取“grade='F'”和break语句这种结构。通过首先指定所有的case值(如前面的代码所示)及其相应执行语句,可以简化编程代码。case值为9和10也是同样的道理。

除了可以是一个变量标识符或复合表达式值外,switch表达式的值也可以是一个逻辑值。考虑下面的语句:

```
switch(age >= 18)
{
case 1: cout<<"Old enough to be drafted."<<endl;
      cout<<"Old enough to vote."<<endl;
      break;

case 0: cout<<"Not old enough to be drafted."<<endl;
      cout<<"Not old enough to vote."<<endl;
}
```

如果age的值是25,那么表达式 $age \geq 18$ 的值是1,即是true。如果表达式的值是1,那么case标识为1中的语句执行。如果age的值是14,那么表达式 $age \geq 18$ 的值为0,即false,case标识为0的语句执行。

在case标识处可以分别使用true和false,而不是1和0,来重写前面的程序,如下所示:

```
switch(age >= 18)
{
case true: cout<<"Old enough to be drafted."<<endl;
          cout<<"Old enough to vote."<<endl;
          break;

case false: cout<<"Not old enough to be drafted."<<endl;
}
```

```
        cout<<"Not old enough to vote."<<endl;  
    }
```

从前面的例子可以看出, switch 语句是执行多重选择的较好方法。在本章的编程实例中将会看到 switch 语句的用法。尽管没有固定的应用规则来决定是使用 if...else 结构还是 switch 结构来执行多重选择,但是需要考虑下面一些问题:如果多重选择包含一个范围的值,那么既可以使用 if...else 结构,也可以使用 switch 结构,这样可以将每一个范围转化为一个有限集的值。

例如,在例 4.22 中, grade 的值取决于 score 的值。如果 score 在 0 到 59 之间, grade 是 'F'。因为 score 是一个 int 变量,所以 60 个值对应于分数 'F'。如果列出所有的 60 个值作为 case 值,那么 switch 语句将会很长。然而,被 10 除后这 60 个值只包含 6 个值: 0, 1, 2, 3, 4, 5。

如果范围值包含无限多的值,并且不能把它们减少为包含有限数字的集合,那么必须使用 if...else 结构。例如,如果分数是 double 变量,那么 0 到 60 之间的数字值是无限的。然而,可以使用表达式 `static_cast<int>(score)/10`,还是可以将这些无限多的值减少为 6 个值。

4.6 用 assert() 函数来终止一个程序

程序中有一些特定类型的错误是很难捕获的。例如,使用目前学过的任何编程技术都很难捕获被零除产生的错误。C++ 包含了一个预定义函数 `assert()`,当某一隐藏错误产生时,用它来终止程序执行是很有用的。对于被零除的情况,可以使用 `assert()` 函数来保证程序终止,并且用适当的错误信息来显示错误类型和程序中出现错误的位置。

考虑下面的语句:

```
int numerator;  
int denominator;  
int quotient;  
double hours;  
double rate;  
double wages;  
char ch;  
  
1. quotient = numerator / denominator;  
2. if(hours > 0 && ( 0 < rate && rate <= 15.50))  
    wages = rate * hours;  
3. if('A' <= ch && ch <= 'B')
```

在第一条语句中,如果 denominator 是 0,逻辑上就不能执行除法。然而,程序执行期间,计算机试图执行除法。如果 denominator 是 0,那么程序将会终止,并且产生错误信息说明出现了一个非法操作。

只有 hours 大于 0、rate 是正数并且小于或等于 15.50,第二条语句才设计用来计算工资。如果 ch 是一个大写字母,第三条语句设计用来执行特定的语句。

对于所有的这些语句(在任何情况下,特定的条件必须被满足),如果条件不满足,应该提供错误信息来显示程序中哪里出现了错误。可以通过在程序中包含输出和返回语句来处理这些情况。然而,如果必要的条件没有被满足,C++ 提供了一种更有效的方法来终止程序,即使用 `assert()` 函数。

使用 `assert()` 函数的语法是:

```
assert(expression);
```

其中 expression 是任何一个逻辑表达式。如果 expression 的值为 true,那么下一条语句执行。如

果 expression 的值为 false, 那么程序会终止并且显示程序中什么地方出现了错误。

assert() 函数的详细规范可以在头文件 cassert 中找到。所以, 对于使用 assert() 函数的程序来说, 它必须包含下面的语句:

```
#include<cassert>
```

注意: 在标准 C++ 中, 包含 assert() 函数详细规范的头文件名是 assert.h。

使用 assert() 函数的语句有时称为断言语句。

回到前面的语句中, 可以使用 assert() 函数重写语句 1 “quotient=numerator/denominator;”。因为 quotient 只有在 denominator 为非零时才能被计算, 所以应该在赋值语句之前包含 assert 语句, 如下所示:

```
assert(denominator);  
quotient = numerator / denominator;
```

现在, 如果 denominator 是 0, 断言语句将会终止程序并且出现错误信息, 如下所示:

```
Assertion failed: denominator, file c:\temp\assert  
function\assertfunction.cpp, line 20
```

错误信息显示 denominator 的断言失败。错误信息也给出了包含源代码的文件名和断言失败所在的行数。

也可以使用断言语句来重写语句 2, 如下所示:

```
assert(hours > 0 && (0 < rate && rate <= 15.50));  
if(hours > 0 && (0 < rate && rate <= 15.50))  
    wages = rate * hours;
```

如果断言语句中的表达式失败, 那么程序就会终止并出现类似于下面的错误信息:

```
Assertion failed: hours > 0 && (0 < rate && rate <= 15.50), file  
c:\temp\assertfunction\assertfunction.cpp, line 26
```

程序开发和测试期间, 断言语句对于加强编程约束是很有用的。正如所看到的, 断言语句不仅终止了程序, 还标识了断言失败所在的表达式、包含源代码的文件名以及断言失败所在的行数。

尽管断言语句在程序开发期间是很有用的, 但是在开发完程序并且投入使用后, 如果断言语句由于某些原因失败, 那么终端用户将不知道错误是什么意思。所以, 在开发和测试完程序之后, 应该删除断言语句, 或使断言语句无效。在一个大型程序中, 删除所有开发期间的断言语句是很乏味的, 甚至是不可能的。另外, 如果打算在将来修改程序, 最好保留断言语句。所以, 合乎逻辑的选择是保留这些语句, 而不是使它们无效。可以使用预处理器指令使断言语句无效。

```
#define NDEBUG
```

该预处理器指令 #define NDEBUG 必须放在指令 #include<cassert>之前。

4.7 编程实例: 有线电视公司账单

该编程实例演示了一个为当地公司计算顾客账单的程序。有两种比率来计算有线电视账单: 一种用于家用客户, 一种用于商用客户。对于家用客户来说, 应用下面的比率:

- 账单处理费: 4.50 美元
- 基本服务费: 20.45 美元

- 附加频道：每频道 7.50 美元

对于商用客户来说，应用下面的比率：

- 账单处理费：15.00 美元
- 基本服务费：前 10 个连接 75.00 美元，每增加一个连接收费 5.00 美元
- 附加频道：每频道对于任何数量的连接都是 50.00 美元

程序应该要求用户输入账号（一个整数）和顾客代码。假设 R 或 r 代表一位家用顾客，B 或 b 代表商用顾客。

输入 顾客的账号、顾客代码、用户订阅附加频道的数量。对于商用顾客来说，还有服务连接的数量。

输出 顾客的账号和账单。

问题分析和算法设计

该程序的目的是计算和打印账单。为了计算账单，需要知道要计算账单的顾客（无论顾客是家用还是商用）和顾客订阅的额外频道的数量。对于商用顾客来说，需要知道服务连接的数量和附加频道的数量。其他需要计算的数据，例如账单处理费和每个附加频道的费用，是已知的量。程序应该将账单以标准货币格式打印到小数点后第二位。问题分析转化为下面的算法：

1. 设置精度到小数点第二位。
2. 提示用户输入账号和顾客类型。
3. 确定附加频道的数量和服务连接、计算账单、打印基于顾客类型的账单：
 - a. 如果顾客类型是 R 或 r，那么：
 - (i) 提示用户输入附加频道的数量
 - (ii) 计算账单
 - (iii) 打印账单
 - b. 如果顾客类型是 B 或 b，那么：
 - (i) 提示用户输入服务连接的数量和附加频道的数量
 - (ii) 计算账单
 - (iii) 打印账单

变量 因为程序要求用户输入顾客账号、顾客代码、附加频道的数量和服务连接的数量，所以需要变量来存储所有这些信息。而且，因为程序要计算账单，所以需要有一个变量来存储账单费用。这样，程序至少需要下面的变量来计算和打印账单：

```
int    accountNumber; //variable to store the customer's
                        //account number
char   customerType;  //variable to store the customer code
int    numberOfPremiumChannels; //variable to store the number
                        //of premium channels to which the
                        //customer subscribes
int    numberOfBasicServiceConnections; //variable to store the
                        //number of basic service connections
                        //to which the customer subscribes
double amountDue;     //variable to store the billing amount
```

常量 可以看到，账单处理费、服务连接的费用和附加频道的费用是固定的，需要这些值来计算账单。尽管可以在程序中直接使用常量值并加以修改，但为了简化后面修改程序的过程，在程序

中不应直接使用这些值,应该将它们声明为常量。在问题分析的基础上,需要声明下面的常量:

```
//Named constants - residential customers
const double rBillProcessingFee = 4.50;
const double rBasicServiceCost = 20.50;
const double rCostOfaPremiumChannel = 7.50;
//Named constants - business customers
const double bBillProcessingFee = 15.00;
const double bBasicServiceCost = 75.00;
const double bBasicConnectionCost = 5.00;
const double bCostOfaPremiumChannel = 50.00;
```

公式 程序使用了大量的公式来计算账单。为了计算家用顾客账单,只需要知道用户订阅的附加频道的数量。下面的语句计算了家用顾客的账单。

```
amountDue = rBillProcessingFee + rBasicServiceCost +
            numberOfPremiumChannels * rCostOfaPremiumChannel;
```

为了计算商用顾客账单,需要知道服务连接的数量和用户订阅的附加频道的数量。如果服务连接的数量小于或等于10,服务连接的费用是固定的。如果服务连接的数量超过10,必须为超过10的每一个连接增加成本。下面的语句计算了商用账单费用:

```
if(numberOfBasicServiceConnections <= 10)
    amountDue = bBillProcessingFee + bBasicServiceCost +
                numberOfPremiumChannels * bCostOfaPremiumChannel;
else
    amountDue = bBillProcessingFee + bBasicServiceCost +
                (numberOfBasicServiceConnections - 10) *
                bBasicConnectionCost +
                numberOfPremiumChannels * bCostOfaPremiumChannel;
```

主要算法

在前面讨论的基础上,现在可以写出下面的主要算法。

1. 为了用小数点和尾零的固定方式输出浮点数,需要设置操纵符fixed和showpoint。而且为了将浮点数输出到小数点第2位,应该将精度设置到小数点第2位。为了使用这些运算符,程序必须包含头文件iomanip。
2. 提示用户输入账号。
3. 获取账号。
4. 提示用户输入顾客代码。
5. 获取顾客代码。
6. 如果顾客代码是r或R,那么:
 - a. 提示用户输入附加频道的数量
 - b. 得到附加频道的数量
 - c. 计算账单
 - d. 打印账号和账单费用
7. 如果顾客类型是b或B,那么:
 - a. 提示用户输入基本服务连接的数量
 - b. 获取服务连接的数量
 - c. 提示用户输入附加频道的数量
 - d. 获取附加频道的数量

e. 计算账单

f. 打印账号和账单费用

8. 如果顾客代码是除了 r,R,b,B 的其他字母, 则输出错误信息。

对于第 6 步和第 7 步, 程序使用了 switch 语句计算对应顾客的账单。

完整程序清单

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

//Named constants - residential customers
const double rBillProcessingFee = 4.50;
const double rBasicServiceCost = 20.50;
const double rCostOfaPremiumChannel = 7.50;

//Named constants - business customers
const double bBillProcessingFee = 15.00;
const double bBasicServiceCost = 75.00;
const double bBasicConnectionCost = 5.00;
const double bCostOfaPremiumChannel = 50.00;

int main()
{
    //Variable declaration
    int    accountNumber;
    char   customerType;
    int    numberOfPremiumChannels;
    int    noOfBasicServiceConnections;
    double amountDue;
    cout<<fixed<<showpoint;           //Step 1
    cout<<setprecision(2);           //Step 1

    cout<<"This program computes a cable bill."<<endl;

    cout<<"Enter account number: ";   //Step 2
    cin>>accountNumber;               //Step 3
    cout<<endl;

    cout<<"Enter customer type: R or r (Residential), "
        <<"B or b(Business): ";       //Step 4
    cin>>customerType;               //Step 5
    cout<<endl;

    switch(customerType)
    {
    case 'r':                          //Step 6
    case 'R': cout<<"Enter the number"
                <<" of premium channels: "; //Step 6a
                cin>>numberOfPremiumChannels; //Step 6b
                cout<<endl;
                amountDue = rBillProcessingFee + //Step 6c
                    rBasicServiceCost +
                    numberOfPremiumChannels *
                    rCostOfaPremiumChannel;

                cout<<"Account number = "<<accountNumber
                    <<endl;           //Step 6d
                cout<<"Amount due = $"<<amountDue
                    <<endl;           //Step 6d
                break;
```

```

case 'b': //Step 7
case 'B': cout<<"Enter the number of basic "
          <<"service connections: "; //Step 7a
cin>>noOfBasicServiceConnections; //Step 7b
cout<<endl;
cout<<"Enter the number"
      <<" of premium channels: "; //Step 7c
cin>>numberOfPremiumChannels; //Step 7d
cout<<endl;

if(noOfBasicServiceConnections <= 10) //Step 7e
    amountDue = bBillProcessingFee +
                bBasicServiceCost +
                numberOfPremiumChannels *
                bCostOfaPremiumChannel;
else
    amountDue = bBillProcessingFee +
                bBasicServiceCost +
                (noOfBasicServiceConnections - 10)
                * bBasicConnectionCost +
                numberOfPremiumChannels *
                bCostOfaPremiumChannel;

cout<<"Account number = "
      <<accountNumber<<endl; //Step 7f
cout<<"Amount due = $"<<amountDue
      <<endl;
break;
default: cout<<"Invalid customer type." <<endl; //Step 8
} //end switch
return 0;
}

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

This program computes a cable bill.
Enter account number: 12345

Enter customer type: R or r (Residential), B or b (Business): B
Enter the number of basic service connections: 10
Enter the number of premium channels: 2

Account number = 12345
Amount due = $520.00

```

4.8 小结

1. 控制结构改变程序正常的控制流程。
2. 两种最常用的控制结构是选择结构和循环结构。
3. 选择结构在一个程序中加入了判断条件。
4. 关系运算符是== (等于)、< (小于)、<= (小于或等于)、> (大于)、>= (大于或等于)以及!= (不等于)。
5. 在关系运算符==、<=、>=和!=中包含空格会产生语法错误。
6. 字符通过计算机的对照序列来进行比较。
7. 逻辑表达式的值为1 (非零值)或0。逻辑值1 (或任何非零值)被视为true, 逻辑值0被视为false。

8. 在 C++ 中, int 变量能用来存储逻辑表达式的值。
9. 在 C++ 中, bool 变量能用来保存一个逻辑表达式的值。
10. 在 C++ 中, 逻辑运算符是!(非), &&(与)和|| (或)。
11. 在 C++ 中, 有两种选择结构。
12. 单分支选择结构有下面的形式:

```
if (expression)
    statement
```

如果 expression 是 true, 那么执行 statement, 否则执行 if 语句后面的语句。
13. 双分支选择结构采用下面的形式:

```
if (expression)
    statement1
else
    statement2
```

如果 expression 的值是 true, 那么执行 statement1, 否则执行 statement2。
14. 在 if 或 if...else 结构中的表达式通常是一个逻辑表达式。
15. 在单分支选择的 statement 前面包含分号会产生一个语义错误。在这种情况下, if 语句的执行语句为空。
16. 在双分支选择的 statement1 前面包含分号会产生一个语法错误。
17. 在 C++ 中没有 else 语句, 每一个 else 都有一个相关联的 if 语句。
18. else 语句与最近的、没有与其他 else 配对的 if 语句配对。
19. 包含在括号之间的一系列语句称为复合语句或块语句。复合语句可被看做是一条单语句。
20. 可以在 if 语句中使用输入流变量来决定输入流的状态。
21. 使用赋值运算符代替等于运算符会产生语义错误。在程序中这可能产生严重的后果。
22. switch 结构用来处理多重选择。
23. 在 switch 语句中, break 语句的执行会使程序立刻跳出 switch 结构。
24. 如果特定的条件在程序中不能满足, 那么可以使用 assert() 函数终止程序。

4.9 练习

1. 判断下面语句的对错。
 - a. 逻辑表达式的结果不能赋给一个 int 变量。
 - b. 在单分支选择中, 如果分号放在 if 语句中的表达式之后, if 语句中的表达式总是 true。
 - c. 每一条 if 语句必须有一条相对应的 else 语句。
 - d. if 语句中的表达式:

```
if (score=30)
    grade='A';
```

总是正确的。
 - e. 如果 $ch < 'A'$ 或 $ch \geq 'z'$, 那么表达式:
 $(ch \geq 'A' \&\& ch \leq 'z')$
的值是 false。
 - f. 假设输入值是 5。下面代码的输出是: Num is zero

```
cin >> num;
if (num > 5)
    cout << num;
num = 0;
```

```
else
```

```
cout<<"Num is zero";
```

g. switch 语句中的表达式应该是一个简单数据类型的值。

h. 只有当 x 是一个负数时, 表达式!(x>0)的值才为 true.

i. 在 C++ 中,!和!= 都是逻辑运算符。

j. 在一个程序中语句执行的顺序叫做控制流。

2. 选出最佳答案。

a. if (6 < 2*5)

```
cout<<"Hello ";
```

```
cout<<" There";
```

代码段输出为:

(i) Hello There

(ii) Hello

(iii) Hello

(iv) There

There

b. if ('a' > 'b' || 66 > static_cast<int>('A'))

```
cout<<"*#*";
```

代码段输出为:

(i) #*#

(ii) #

(iii) *

(iv) 以上全不是

*

#

c. if (7 <= 7)

```
cout<<6-9*2/6;
```

代码段输出为:

(i) -1

(ii) 3

(iii) 3.0

(iv) 以上全不是

d. if (7 < 8)

```
{
```

```
cout<<"2 4 6 8"<<endl;
```

```
cout<<"1 3 5 7"<<endl;
```

```
}
```

代码段输出为:

(i) 2 4 6 8

(ii) 1 3 5 7

(iii) 以上全不是

1 3 5 7

e. if (5 < 3)

```
cout<<"*";
```

```
else
```

```
if (7 == 8)
```

```
cout<<"&";
```

```
else
```

```
cout<<"$";
```

代码段输出为:

(i) *

(ii) &

(iii) \$

(iv) 以上全不是

3. 下面语句的输出是什么?

```
x = 100;
```

```
y = 200;
```

```
if (x > 100 && y <= 200)
```

```
cout<<x<<" "<<y<<" "<<x+y<<endl;
```

```
else
```

```
cout<<x<<" "<<y<<" "<<2*x-y<<endl;
```

4. 编写 C++ 语句, 如果 gender 是 'M', 那么输出 Male, 如果 gender 是 'F', 那么输出 Female, 否则输出 invalid gender。

5. 改正下面的代码并输出正确的信息。

```
if(score >= 60)
    cout<<"You pass."<<endl;
else;
    cout<<"You fail."<<endl;
```

6. 判断下面的语句是否是正确的 switch 语句。如果不是, 解释原因。假设 n 和 digit 是 int 变量。

a. switch(n <= 2)

```
{
    case 0: cout<<"Draw.";
            break;
    case 1: cout<<"Win.";
            break;
    case 2: cout<<"Lose.";
            break;
}
```

b. switch (digit / 4)

```
{
    case 0, case 1: cout<<"low.";
                    break;
    case 1, case 2: cout<<"middle.";
                    break;
    case 3: cout<<"high.";
}
```

c. switch(n % 6)

```
{
    case 1: case 2: case 3: case 4: case 5: cout<< n;
                                                break;
    case 0: cout<<endl;
            break;
}
```

d.

```
switch (n % 10)
{
    case 2: case 4: case 6: case 8: cout<<"Even";
                                    break;
    case 1: case 3: case 5: case 7: cout<<"Odd";
                                    break;
}
```

7. 假设输入是 5, 在下面的 C++ 代码执行后, alpha 的值是什么?

```
cin>>alpha;
switch(alpha)
{
    case 1:
    case 2: alpha = alpha + 2;
            break;
    case 4: alpha++;
    case 5: alpha = 2 * alpha;
    case 6: alpha = alpha + 5;
            break;
    default: alpha--;
}
```

8. 假设输入是 3, 在下面的 C++ 代码执行后, beta 的值是什么?

```

cin>>beta;
switch(beta)
{
case 3: beta = beta + 3;
case 1: beta++; break;
case 5: beta = beta + 5;
case 4: beta = beta + 4;
}

```

9. 假设输入是6, 在下面的 C++ 代码执行后, a 的值是什么?

```

cin>>a;
if(a > 0)
    switch(a)
    {
        case 1: a = a + 3;
        case 3: a++;
                break;
        case 6: a = a + 6;
        case 8: a = a * 8;
                break;
        default: a--;
    }
else
    a = a + 2;

```

10. 在下面的代码中, 改正任何可能导致程序编译和运行时的错误。

```

include <iostream>

main ()
{
    int a,b;
    bool found;
    cout<<"Enter two integers: ";
    cin>>a>>b;

    if a > a*b && 10 < b
        found = 2* a > b;
    else
    {
        found = 2 * a < b;
        if found
            a = 3;
            c = 15;
        if b
        {
            b = 0;
            a = 1;
        }
    }
}

```

11. 下面的程序包含错误。改正这些错误, 令程序可以运行并且输出 w=21。

```

#include <iostream>
using namespace std;
const int one = 5
main ()
{
    int x, y, w, z;
    z = 9;
}

```



```

    if z > 10
        x = 12; y = 5, w = x + y + one;
    else
        x = 12; y = 4, w = x + y + one;
    cout<<"w = "<<w;
}

```

4.10 编程练习

1. 编写一个程序，提示用户输入一个数字。然后程序应该输出该数字，并且输出一条信息告诉用户该数字是正数、负数还是零。
2. 编写一个程序，提示用户输入三个数字。然后程序按升序输出这三个数字。
3. 编写一个程序，提示用户输入一个 0 到 35 之间的数字。如果该数字小于或等于 9，那么程序应该输出该数字；否则它应该输出 A 来代表 10，B 代表 11，…，Z 代表 35。（提示：对于大于等于 10 的数字，使用强制转换操作符 `static_cast<char>()`。）
4. 从纽约到巴黎的国际长途电话是这样计费的：接入费 1.99 美元，前三分钟 2.00 美元，额外的每一分钟 0.45 美元。
编写一个程序，提示用户输入电话接入的分钟数，输出应该缴纳的费用。将输出格式化到小数点后两位。
5. 在一个直角三角形中，斜边长度的平方等于其他两边长度平方的和。编写一个程序，提示用户输入一个三角形三边的长度，然后输出一条信息显示该三角形是否是直角三角形。
6. 二次方程 $ax^2+bx+c=0$ 的根由下面的公式给出：

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

在该公式中，项 b^2-4ac 称为判别式。如果 $b^2-4ac=0$ ，那么方程有一个单（相等的）根；如果 $b^2-4ac>0$ ，那么方程有两个实根；如果 $b^2-4ac<0$ ，那么方程有两个复数根。编写一个程序，提示用户输入 a 的值（ x^2 的系数）， b （ x 的系数）的值以及 c 的值，然后输出方程根的类型。而且，如果 $b^2 \geq 4ac$ ，程序应该输出二次方程的根。（提示：从头文件 `cmath` 中使用函数 `pow()` 来计算平方根。第 3 章显示了怎样使用 `pow()` 函数。）

7. 编写一个程序，提示用户输入笛卡尔坐标系中某一点的 x, y 坐标。然后程序输出一条信息，显示该点是否位于坐标原点、 x （或 y 轴），或者出现在某一个象限。例如：
(0,0)是坐标原点
(4,0)在 x 轴上
(0,-3)在 y 轴上
(-2,3)在第二象限。
8. 编写一个程序来模拟一个计算器。程序应该输入两个整数和运算符。然后输出数字、运算符和结果。（对于除法，如果分母是零，那么输出一条适当的信息。）示例输出如下：
3+4=7
13*5=65
9. 修改练习 8 来处理浮点数。（将输出格式化到小数点后两位。）
10. 银行要在每个月末更新顾客的账户。银行提供了两种类型的账户：储蓄账户和活期账户。每一位顾客必须拥有一个最低限度的余额。如果顾客的余额小于最小余额，那么就会对储蓄

账户收取 10.00 美元的服务费, 对活期账户收取 25.00 美元的服务费。如果月底的余额不小于最低余额, 那么账户的利息如下:

- a. 储蓄账户的利息率为 4%。
- b. 余额大于最低余额 5 000 美元的活期账户利息率为 3%, 否则利息率是 5%。

编写一个程序, 读入一位顾客的账户 (int 类型), 账户类型 (char 类型; s 代表储蓄账户, c 代表活期账户), 账户应该持有的最低余额, 当前余额。然后程序应该输出账户、账户类型、当前余额和适当的信息。使用下面的信息来运行程序并对其进行测试。

```
46728 S 1000 2700
87324 C 1500 7689
79873 S 1000 800
89832 C 2000 3000
98322 C 1000 750
```

- 11. 编写一个程序来执行例 1.2 (第 1 章) 给出的算法, 它计算了一名售货员每月的工资。
- 12. 编写一个程序来为一个蜂窝电话公司计算和打印账单。

该公司提供两种类型的服务: 常规服务和增值服务。其收费比率依赖于服务类型。比率由下面的公式计算:

常规服务: 10.00 美元基本费用。前 50 分钟免费, 超过 50 分钟, 则每分钟收取 0.20 美元。

增值服务: 25.00 美元基本费用, 并且:

a. 6:00A.M. 到 6:00P.M., 前 75 分钟是免费的, 超过 75 分钟每分钟 0.10 美元。

b. 6:00P.M. 到 6:00A.M., 前 100 分钟是免费的, 超过 100 分钟每分钟 0.05 美元。

程序应该提示用户输入账户、服务代码 (char 类型) 和所使用服务的分钟数。服务代码 r 或 R 表示常规服务; 服务代码 p 或 P 表示增值服务。其他任何字符都是错误的。程序应该输出账号、服务类型、电话服务使用的分钟数和用户应缴费的数量。

对于增值服务, 顾客可能全天候地使用服务。所以, 为了计算账单, 必须要求用户输入白天使用服务的分钟数和夜晚使用服务的分钟数。

第5章 控制结构 II (重复)

本章要点:

- 了解重复控制结构
- 学习怎样创建并使用计数控制重复结构、标记控制重复结构、状态标志符控制重复结构和 EOF 控制重复结构
- 学习 break 和 continue 语句
- 学习怎样建立和使用嵌套控制结构

在第4章中,已经学习了如何在程序中嵌套构建选择结构。在本章中,将学习如何在程序中编写重复结构。

5.1 为什么需要重复结构

假设想将5个数字相加求平均值。从目前学过的知识来看,可能会使用下面的方法来处理(假设所有的变量被正确地声明)。

```
cin>>num1>>num2>>num3>>num4>>num5; //read five numbers
sum = num1+num2+num3+num4+num5;      //add the numbers
average = sum/5;                       //find the average
```

假设想对100、1000或更多数字进行相加并计算其平均值,将不得不声明相应数量的变量,并且还要在 cin 语句中列出来,而且还可能要在输出语句中再一次写出相应的变量,这占用了极大的空间和时间。而且,如果想用不同的值或者不同数目的值再一次运行程序,就必须重写该程序。

假设将下面的数字相加:

5 3 7 9 4

考虑下面的语句,在语句中 sum 和 num 是 int 类型的变量。

```
1. sum = 0;
2. cin>>num;
3. sum = sum + num;
```

第一条语句初始化 sum 为 0。语句 2 将 5 存储到 num 中,语句 3 将 sum 的值与 num 相加并且赋给 sum。在语句 3 之后, sum 的值是 5。

重复执行语句 2 和语句 3。在语句 2 之后(程序代码读入下一个数字),

num=3

在语句 3 之后,

sum=sum+num=5+3=8

执行到这里, sum 包含了前两个数字的和。再一次重复执行语句 2 和语句 3(第 3 次)。在语句 2 之后(程序代码读入下一个数字),

num=7

在语句3之后,

```
sum=sum+num=8+7=15
```

现在 sum 包含了前 3 个数字的和。如果再一次重复执行语句 2 和语句 3, sum 将会包含所有 4 个数字的和。

如果相加 10 个数字, 可以重复执行语句 2 和语句 3 10 次。如果相加 100 个数字, 可以重复执行语句 2 和语句 3 100 次。无论哪一种情况, 都不需要声明任何额外的变量。可以使用该 C++ 代码将任何集合的数字相加, 而前面的代码则需要修改代码。

在很多时候都需要重复执行一个语句集合。例如, 对于班级中的每一名学生来说, 确定课程分数的公式是相同的。C++ 有三种重复结构并允许多次重复执行语句, 直到满足特定的条件为止。本章介绍所有的三种循环 (重复) 结构。

5.2 while 循环 (重复) 结构

在前面的部分中, 可以看到有时重复执行一个语句集合是很有必要的。重复执行一个语句集合的一种方法是在程序中多次输出集合中的语句。例如, 如果想重复执行某语句集合 100 次, 那么需要在程序中输入 100 次该集合中的语句。然而, 这种重复执行一个集合语句的方法是可能的, 但却是不切实际的。有一种更好的办法来重复执行一个集合中的语句。前面指出, C++ 有三种重复或循环结构允许重复执行语句, 直到特定的条件得到满足。

while 语句的常用形式为:

```
while (expression)
    statement
```

在 C++ 中, while 是一个保留字。当然, statement 既可以是单语句, 也可以是复合语句。expression 就像是一个决策者并且通常是逻辑表达式。statement 称为循环体。注意 expression 两边的括号是语法的一部分。图 5.1 显示了 while 循环的执行流程。

expression 提供一个入口条件。如果它初始化为 true, statement 就会执行。循环条件 expression 然后被重新赋值。如果它的值仍为 true, 那么 statement 再一次执行。statement (循环体) 继续执行直到 expression 不再是 true。无休止重复执行的循环称为死循环。为了避免死循环, 应该保证循环体包含退出条件, 即 while 语句中表达式的值可以变为 false。

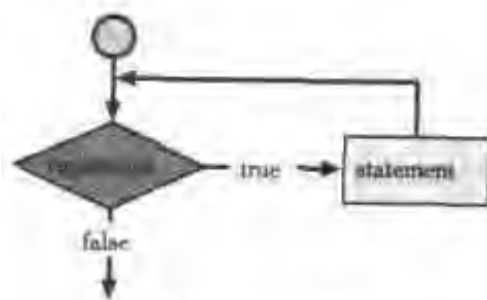


图5.1 while循环

例 5.1

考虑下面的 C++ 语句:

```
i = 0;                //Line 1
while(i <= 20)         //Line 2
{
    cout<<i<<" ";    //Line 3
    i = i + 5;        //Line 4
}
```

输出为:

```
0 5 10 15 20
```

在第一行中, 变量 i 被设置为 0。while 语句 (第 2 行) 中的表达式 $i \leq 20$ 被计算值。由于表达

式 $i \leq 20$ 的值为 true, 所以 while 循环体继续执行。while 循环体包含了第 3 行和第 4 行的语句。第 3 行语句输出 i 的值, 即 0。第 4 行语句将 i 的值加 5。在执行完第 3 行和第 4 行的语句后, while 循环 (第 2 行) 中的表达式再一次被计算值。由于 i 是 5, 所以表达式 $i \leq 20$ 的值为 true, while 循环体又一次执行。计算表达式的值和执行 while 循环体的过程一直继续, 直到 $i \leq 20$ 的值为 false。

表达式中的变量 i (第 2 行) 称为循环控制变量。

从前面的例子中, 可以看出应该注意以下几点:

- a. 在循环体内 i 值变成 25, 但是不被输出, 因为入口条件是 false。
- b. 如果忽略循环体中的语句:

```
i=i+5;
```

那么将会导致死循环并输出无数行的零。

- c. 在执行循环前必须初始化控制循环变量 i。如果语句 (第一行):

```
i=0;
```

被忽略, 那么循环可能根本不执行。(前面讲过在 C++ 中, 变量不会自动初始化。)

- d. 在上面的程序段中, 如果循环体中的两条语句交换位置, 那么很可能会出现意想不到的结果。例如, 考虑下面的语句:

```
i = 0;
while (i <= 20)
{
    i = i + 5;
    cout<<i<<" ";
}
```

这里输出是:

```
5 10 15 20 25
```

这是一种典型的语义错误, 因为几乎不可能在 $i \leq 20$ 为真的条件下, 产生一个 $i > 20$ 的结果。

例 5.2

考虑下面的 C++ 程序段:

```
i = 20;           //Line 1
while (i < 20)    //Line 2
{
    cout<<i<<" "; //Line 3
    i = i + 5;    //Line 4
}
```

该例与例 5.1 之间的差别是很容易被忽视的。在第一行中, i 被设置为 20。由于 i 是 20, 所以 while 循环体中表达式 $i < 20$ 的值是 false。由于初始的循环入口条件 $i < 20$ 是 false, 所以 while 循环体从不会执行。这样, 就不会有值输出, i 的值一直保持为 20。

当一个程序需要多次测试某些特定的数据时, 或者如果程序要对大量的数据进行操作时, 使用键盘输入数据并不是一种理想的办法。在这种情况下, 可以将数据存储到例如磁盘之类的外部设备上, 并且令程序从设备上读入数据。为了简单起见, 假设数据存储到磁盘上 (软盘或硬盘) 并且没有错误。在这种条件下有 4 种情况, 下面的 4 个部分将分别对其进行描述。

5.2.1 情况 1: 计数控制的 while 循环

假设已准确地知道有多少项数据需要被读入。在这种情况下, while 循环采取计数控制的 while 循环形式。假设文件有 N 个数据项, 可以建立一个计数器 (在 while 语句前初始化为 0) 来跟踪有

多少项已经被读入了。在执行 while 语句体之前, counter 与 N 进行比较。如果 counter < N, while 语句体执行。循环体重复执行,直到 counter 的值 $\geq N$ 。这样,在 while 语句体内部,counter 的值在读入一个新数之后就会加 1。在这种情况下,while 循环会有下面的形式:

```
counter = 0;
while(counter < N)
{
    .
    .
    .
    counter++;
    .
    .
    .
}
```

N 的值(文件中项目的数量)可以通过几种方法来确定。程序可以提示用户指定文件中项目的数字并使用 cin 语句读入该值。另一方面,可以指定文件中的第一个值作为文件中输入项的数量,这样就不需要记住输入值的数字(项目)。如果除了程序员,还有其他人输入数据的话,这将是很有用的。考虑下面的例子。

例 5.3

假设输入为:

12 8 9 2 3 90 38 56 8 23 89 7 2 8 3 8

第一个数字 12 指定了数据输入项数字。假设想将这些数字加起来并且求平均值。注意只有 12 个值被读入,其他值都被忽略掉了。完整的程序如下:

```
//Program: AVG1
#include <iostream>
using namespace std;

int main()
{
    int limit;        //store the number of items
                     //in the list
    int number;       //variable to store the number
    int sum;          //variable to store the sum
    int counter;      //loop control variable

    cout<<"Line 1: Enter data for processing"
         <<endl;
    cin>>limit;
    sum = 0;
    counter = 0;
    while(counter < limit)
    {
        cin>>number;
        sum = sum + number;
        counter++;
    }

    cout<<"Line 9: The sum of "<<limit
         <<" numbers = "<<sum<<endl;
}
```

```

        if(counter != 0)                                //Line 10
            cout<<"Line 11: The average = "
              <<sum / counter<<endl;                    //Line 11
        else                                             //Line 12
            cout<<"Line 13: No input."<<endl;            //Line 13
        return 0;
    }

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Line 1: Enter data for processing
12 8 9 2 3 90 38 56 8 23 89 7 2 8 3 8
Line 9: The sum of 12 numbers = 335
Line 11: The average = 27

```

该程序原理如下。第一行语句提示用户输入数据以进行处理。第2行语句从输入数据中读入第一个值并且将它保存到变量 `limit` 中。`limit` 的值表示输入项的数量。第3行和第4行语句将变量 `sum` 和 `counter` 初始化为0。第5行的 `while` 语句检查 `counter` 的值来确定已经读入多少项。如果 `counter` 小于 `limit`，`while` 循环会处理下一个输入。第6行语句读入下一个数字并且将它存储到变量 `number` 中。第7行语句通过将 `number` 的值与前一个值相加来更新 `sum` 的值。第8行语句将 `counter` 的值加1。第9行语句输出数字的和。第10~13行语句输出平均值。

注意 `sum` 在程序的第3行被初始化为零。在第7行，程序将输入值进行累加。第一个输入的数字与零（因为 `sum` 被初始化为零）相加，得到第一个数字正确的和。为了求得平均值，将 `sum` 除以 `counter`。如果 `counter` 是0，那么除零将会终止该程序并会得到一条错误信息。所以，在 `sum` 除以 `counter` 之前，必须检查 `counter` 是否为零。

5.2.2 情况2：标记控制的 `while` 循环

可能不是总能知道需要读入多少项数据，但可以将最后一项输入值作为结束标记。在这种情况下，在 `while` 语句前读入第一项。如果该项不等于标记，`while` 语句体执行。只要程序没有读到标记，`while` 循环就会继续执行。这种 `while` 循环称为标记控制的 `while` 循环。在这种情况下，`while` 循环会有下面的形式：

```

cin>>variable;
while(variable != sentinel)
{
    .
    .
    .
    cin>> variable;
    .
    .
    .
}

```

例 5.4

假设读入一些正整数并且对它们求平均值，但是并不知道数据项的预置数目。假设数字-999表示数据的结尾。可以用下面的程序来处理数据：

```

//Program: AVG2
#include <iostream>
using namespace std;

const int SENTINEL = -999;

```



```

int main()
{
    int number;           //variable to store the number
    int sum = 0;          //variable to store the sum
    int count = 0;        //variable to store the total
    cout<<"Line 1: Enter numbers ending with "
        <<SENTINEL<<endl;           //Line 1
    cin>>number;                 //Line 2
    while(number != SENTINEL)     //Line 3
    {
        sum = sum + number;       //Line 4
        count++;                 //Line 5
        cin>>number;             //Line 6
    }

    cout<<"Line 7: The sum of "<<count
        <<" numbers is "<<sum<<endl; //Line 7

    if(count != 0)               //Line 8
        cout<<"Line 9: The average is "
            <<sum / count<<endl;    //Line 9
    else                          //Line 10
        cout<<"Line 11: No input."<<endl; //Line 11

    return 0;
}

```

运行实例：在本运行实例中，用户输入以阴影表示。

```

Line 1: Enter numbers ending with -999
34 23 9 45 78 0 77 8 3 5 -999
Line 7: The sum of 10 numbers is 282
Line 9: The average is 28

```

程序工作原理如下。第一行语句提示用户输入以-999结尾的数字。第2行语句读入第一个数字并且将它保存到number中。第3行的while语句检查number是否不等于标记。如果number不等于标记，while循环体就会执行。第4行语句将sum的值与number的值相加来更新sum的值。第5行语句将count的值加1，第6行语句读入并存储下一个数字到number中。第4~6行语句重复执行，直到程序读到标记。第7行语句输出数字的和，第8~10行语句输出数字的平均值。

为了进一步理解该程序是如何工作的，使用下面的数据来测试该程序。

```

89 23 64 78 23 9 45 78 34
0 34 87 23 8 3 5 -999

```

下面考虑另一个标记控制的while循环的例子。在该例子中，用户通过提示输入需要处理的值，并可以输入结束标记来终止程序。

例 5.5 电话数字

下面的程序读入字母A到Z并且打印相应的电话数字。该程序使用了标记控制的while循环。为了终止该程序，用户需要输入结束标记。这也是嵌套控制结构的一个例子，其中if-else, switch和while循环是嵌套的。

```

//*****
// Program: Telephone Digits
// This is an example of a sentinel-controlled loop. This
// program converts uppercase letters to their
// corresponding telephone digits.

```



```

/*****

#include <iostream>
using namespace std;

int main()
{
    char letter;                                     //Line 1

    cout<<"This program converts uppercase "
        <<"letters to their corresponding "
        <<"telephone digits."<<endl;                 //Line 2
    cout<<"To stop the program enter Q or Z."<<endl; //Line 3
    cout<<"Enter a letter--> ";                     //Line 4
    cin>>letter;                                     //Line 5

    cout<<endl;                                       //Line 6

    while(letter != 'Q' && letter != 'Z' )           //Line 7
    {
        cout<<"The letter you entered is ---> "
            <<letter<<endl;                           //Line 8
        cout<<"The corresponding telephone "
            <<"digit is --> ";                         //Line 9
        if(letter >= 'A' && letter <= 'Z')           //Line 10
            switch(letter)                            //Line 11
            {
                case 'A': case 'B': case 'C': cout<<"2\n"; //Line 12
                                                    break; //Line 13
                case 'D': case 'E': case 'F': cout<<"3\n"; //Line 14
                                                    break; //Line 15
                case 'G': case 'H': case 'I': cout<<"4\n"; //Line 16
                                                    break; //Line 17
                case 'J': case 'K': case 'L': cout<<"5\n"; //Line 18
                                                    break; //Line 19
                case 'M': case 'N': case 'O': cout<<"6\n"; //Line 20
                                                    break; //Line 21
                case 'P': case 'R': case 'S': cout<<"7\n"; //Line 22
                                                    break; //Line 23
                case 'T': case 'U': case 'V': cout<<"8\n"; //Line 24
                                                    break; //Line 25
                case 'W': case 'X': case 'Y': cout<<"9\n"; //Line 26
            }
        else                                           //Line 27
            cout<<"You entered a bad letter."<<endl; //Line 28

        cout<<"\nEnter another uppercase letter to be"
            <<" \nconverted to the corresponding "
            <<"telephone digits."<<endl<<endl;         //Line 29
        cout<<"To stop the program enter Q or Z."
            <<endl<<endl;                               //Line 30
        cout<<"Enter a letter--> ";                   //Line 31
        cin>>letter;                                   //Line 32
        cout<<endl;                                    //Line 33
    } //end while

    return 0;
}

```

运行实例：在本运行实例中，用户输入用阴影表示。

This program converts uppercase letters to their

```

corresponding telephone digits.
To stop the program enter Q or Z.
Enter a letter--> A
The letter you entered is ---> A
The corresponding telephone digit is --> 2

Enter another uppercase letter to be
converted to the corresponding telephone digits.

To stop the program enter Q or Z.

Enter a letter--> D
The letter you entered is ---> D
The corresponding telephone digit is --> 3

Enter another uppercase letter to be
converted to the corresponding telephone digits.

To stop the program enter Q or Z.

Enter a letter--> Q

```

该程序工作原理如下。第2行和第3行语句向用户显示一些功能信息。第4行语句让用户通过提示输入一个字母，第5行语句读入并且存储一个字符到变量letter中。第7行的while循环检查字母是否为Q和Z。如果用户输入的字母既不是Q，也不是Z，那么while循环体执行。第8行语句输出用户输入的字母。第10行的if语句检查用户输入是否是大写字母。if语句的代码块中是switch语句（第11行）。如果用户输入的字母是大写字母，if语句中（第10行）的表达式值为true，switch语句就会执行。如果用户输入的字母不是大写字母，那么else语句（第27行）就会执行。第12~26行语句确定相应的电话数字。

一旦当前的字母被处理，第29和第30行的语句就会再一次通知用户下一步做什么。第31行语句提示用户输入一个字母；第32行语句读入并存储该字母到变量letter中。（注意第29行语句类似于第2行的语句，第30~33行的语句与第3~6行的语句是相同的。）在第33行语句（while循环的末尾）执行后，控制回到while循环开始处的逻辑表达式，相同的过程再一次开始。当用户输入Q或Z，程序就会终止。

注意：在例5.5的程序中，可以使用一个switch语句来编写第10~28行语句。（参看本章末尾的编程练习4）

5.2.3 情况3:状态标志符控制的while循环

状态标志符控制的while循环使用一个布尔变量来控制循环。假设found是一个布尔变量。状态标志符控制的while循环有下面的形式：

```

found = false;
while(!found)
{
    .
    .
    .
    if(expression)
        found = true;
    .
    .
    .
}

```

变量 found 称为一个状态标志符变量, 用来控制 while 循环的执行。

5.2.4 情况 4: EOF 控制的 while 循环

如果数据文件经常被修改(例如, 如果数据经常添加和删除), 那么最好不要使用标记值来读入数据。某人可能不小心删除了标记值或者使用标记添加数据, 特别是如果程序员和数据录入的人不是同一人。而且, 程序员有时并不知道标记是什么。在这种情况下, 可以使用 EOF(文件结束)控制的 while 循环。

直到现在, 我们一直使用一个输入流变量, 例如 cin 和析取运算符 >> 来读入数据并将它们存储到变量中。然而, 输入流变量在读入数据之后也可以返回一个值:

1. 如果程序到达输入数据的末尾, 那么输入流变量返回逻辑值 false。
2. 如果程序读入任何错误数据(例如向一个 int 变量读入 char), 那么输入流会进入 fail 状态。一旦输入流进入 fail 状态, 任何使用该流的进一步 I/O 操作都会认为是空操作。这种情况下, 计算机不能终止程序或给出任何错误信息, 它只是继续执行程序并忽略每一个使用该流的输入。此时, 输入流变量返回 false 值。
3. 除了情况 1 和 2, 输入流变量返回 true 值。

可以使用输入流变量的返回值来判断程序是否到达了输入流的末尾。由于输入流变量返回逻辑值 true 或 false, 所以在 while 循环中, 可以作为一个逻辑表达式。

下面是 EOF 控制的 while 循环的一个例子。

```
cin>>variable;
while(cin)
{
    .
    .
    .
    cin>>variable;
    .
    .
    .
}
```

5.2.5 eof() 函数

为了判断程序是否到达了文件的末尾, 除了检查一个输入流变量, 例如 cin 的值, C++ 还提供了一个函数。可以使用该函数和一个输入流变量来判断文件末尾状态。该函数被称为 eof()。类似于第 3 章中讨论的 get(), ignore() 和 peek() 函数, eof() 函数也是 istream 数据类型的成员。

使用 eof() 函数的语法是:

```
istreamVar.eof()
```

其中 istreamVar 是一个输入流变量, 例如 cin。

假设有下面的声明:

```
ifstream infile;
```

假设使用变量 infile 打开了一个文件。考虑表达式:

```
infile.eof()
```

这是一个逻辑(布尔)表达式。如果程序读到了输入文件 infile 的末尾, 那么该表达式的值为

true; 否则表达式的值为 false。

如果输入是文本, 那么判断文件末尾状态 (即使用函数 eof()) 是最好的办法。如果输入包含数值型数据, 那么前面判断文件末尾状态的方法是更好的方法。

假设有下面的声明:

```
ifstream infile;  
char ch;  
  
infile.open("inputDat.dat");
```

只要程序没有到达文件的末尾, 下面的 while 循环就会继续执行。

```
infile.get(ch);  
while(!infile.eof())  
{  
    cout<<ch;  
    infile.get(ch);  
}
```

只要程序没有到达文件的末尾, 表达式:

```
infile.eof()
```

就是 false, 所以 while 语句中的表达式:

```
!infile.eof()
```

值为 true。当程序读到输入文件的末尾时, 表达式:

```
infile.eof()
```

就变为 true, 所以 while 语句中的表达式:

```
!infile.eof()
```

就会变为 false, 循环就会终止。

注意: 在 DOS 环境下, 文件末尾标志是使用 ctrl+z (按住 ctrl 键并且按住 z) 来输入的。在 Unix 环境中, 文件末尾标志是使用 ctrl+d (按住 ctrl 键并且按住 d) 来输入的。

5.3 编程实例: 检查账户余额

假设银行正寻找程序员编写一个程序来计算每月顾客的活期账户余额。文件中的数据用下面的形式来存储:

```
467343 23750.40  
W 250.00  
D 1200  
W 75.00  
I 120.74  
.  
.  
.
```

第一行数据显示了账号, 然后是每月初的余额。后面的每一行包含两项: 交易代码和交易数量。交易代码 W 或 w 表示取款, D 或 d 表示存款, I 或 i 表示银行所付的利息。每月期间, 如果余额小于 1000.00 美元, 就要收取 25.00 美元的服务费。程序打印下面的信息: 账号、每月初的余额、每

月末的余额、银行所付的利息、总存款的数量、存款的次数、提款的总额、提款的次数和服务费(如果有的话)。

输入 包含上述格式的数据文件。

输出 输出采用下面的形式:

```
Account Number: 467343
Beginning Balance: $23750.40
Ending Balance: $24611.49

Interest Paid: $366.24

Amount Deposited: $2230.50
Number of Deposits: 3

Amount Withdrawn: $1735.65
Number of Withdrawals: 6
```

问题分析和算法设计

输入文件中的第一项是账号和月初余额。所以,程序首先读入账号和月初余额。然后,文件中的每一项采用下面的形式:

交易代码 交易数量

为了确定每月末的余额,需要处理包含交易代码和交易数量的每一个条目。以月初余额作为初始值,然后在处理完每一个条目后更新余额。如果交易代码是D, d, I或i,那么交易数量就被加到账户余额上。如果交易代码是W或w,那么交易数量从余额中减掉。由于程序也需要输出取款和存款的数量,所以需要记录并保存相应的取款额和存款额。这些讨论可以转化为下面的算法:

1. 声明变量。
2. 初始化变量。
3. 得到账号和初始余额。
4. 得到交易代码和交易数量。
5. 分析交易代码和更新相应的变量。
6. 重复步骤4和5,直到数据结束为止。
7. 打印结果。

变量 程序输出账号、每月开始的余额和月末的余额、银行所付的利息、存款数量、存款次数、提款数量、提款次数和服务费(如果有的话)。需要变量来存储这些信息。到目前为止所需要的变量如下:

```
acctNumber          //variable to store the account number
beginningBalance    //variable to store the beginning balance
accountBalance       //variable to store the account balance at
                      //the end of the month
amountDeposited      //variable to store total amount deposited
numberOfDeposits     //variable to store the number of deposits
amountWithdrawn      //variable to store total amount withdrawn
numberOfWithdrawals  //variable to store number of withdrawals
interestPaid         //variable to store interest amount paid
```

由于程序从一个文件中读入数据并输出到一个文件中,所以程序需要输入/输出数据流变量。在第一行之后,每一行的数据是交易代码和交易数量。程序需要一个变量来存储该信息。

当余额低于最小余额时,那么就要收取该月的服务费。在每次提款后,需要检查账户余额。如

果在一次提款后余额低于最小余额,就需要收取服务费。一个月中可能有好几次提款,一旦账户余额低于最小余额,那么后来的存款也可能使余额高于最小余额,其他的提款可能又一次将它减少到最小余额以下。然而这种情况下,只收取一次服务费。

为了实现这种想法,程序使用了一个布尔变量 `isServiceCharged` 并初始化为 `false`,只要账户余额低于最小余额,它就被设置为 `true`。在采取服务收费之前,程序检查变量 `isServiceCharged`。如果账户余额小于最小余额并且 `isServiceCharged` 是 `false`,就要收取服务费。程序需要下面的变量:

```
int      acctNumber;
double   beginningBalance;
double   accountBalance;

double   amountDeposited;
int      numberOfDeposits;

double   amountWithdrawn;
int      numberOfWithdrawals;

double   interestPaid;

char      transactionCode;
double   transactionAmount;

bool     isServiceCharged;

ifstream infile; //input file stream variable
ofstream outfile; //output file stream variable
```

常量 由于最小账户余额和服务费数量是固定的,所以程序使用了两个常量来存储它们:

```
const double minimumBalance = 1000.00;
const double serviceCharge = 25.00;
```

分析和算法设计(续前)

由于该程序比前面的程序更复杂,所以在开始编写主要算法之前,需要对上面的7个步骤进行详细地讨论。

1. **声明变量。**声明前面讨论的变量。
2. **初始化变量。**在每次存款后,存款的总数量被更新,存款的次数加1。在第一次存款前,存款总数为0,提款次数为0。所以变量 `amountDeposited` 和 `numberOfDeposits` 必须初始化为0。同样,变量 `amountWithdrawn`, `numberOfWithdrawals` 和 `interestPaid` 必须初始化为0。而且正如前面所讨论的,变量 `isServiceCharged` 初始化为 `false`。可以在声明它们的时候同时初始化这些变量。

在第一次存款、提款或付利息前,账户余额应该等于开始余额。所以,在从文件读入变量 `beginningBalance` 所代表的开始余额前,需要初始化变量 `accountBalance` 为变量 `beginningBalance` 的值。

由于数据从一个文件中读入,需要打开输入文件。如果输入文件不存在,输出一条适当的信息并且终止程序。由于输出存储到文件中,需要打开输出文件。假设输入数据在软盘A的 `Ch5_money.txt` 文件中,同时输出保存在软盘A的 `Ch5_money.out` 文件中。下面的代码用来打开文件:

```
infile.open("a:Ch5_money.txt"); //open the input file
if(!infile)
{
```



```

    cout<<"Cannot open input file"<<endl;
    cout<<"Program terminates!!!"<<endl;
    return 1;
}

```

```

outfile.open("a:Ch5_money.out"); //open the output file

```

3. 得到账号和初始余额。通过下面的输入语句来完成:

```

infile>>acctNumber>>beginningBalance;

```

4. 得到交易代码和交易数量。通过下面的输入语句来完成:

```

infile>>transactionCode>>transactionAmount;

```

5. 分析交易代码和更新相应的变量。如果 transactionCode 是 'D' 或 'd', 将 accountBalance 与 transactionAmount 相加来更新 accountBalance; 将 accountDeposited 与 transactionAmount 相加来更新 accountDeposited; 变量 numberOfDeposits 加 1。如果 transactionCode 是 'I' 或 'i', 将 accountBalance 与 transactionAmount 相加来更新 accountBalance; 将 interestPaid 与 transactionAmount 相加来更新 interestPaid。如果 transactionCode 是 'W' 或 'w', 将 accountBalance 与 transactionAmount 相减来更新 accountBalance; 将 amountWithdrawn 与 transactionAmount 相加来更新 amountWithdrawn; 变量 numberOfWithdrawals 加 1, 如果账户余额小于最小余额并且没有收取服务费, 那么从账户余额中减掉服务费并且标记服务费已经收取了。下面的 switch 语句完成该任务。

```

switch(transactionCode)
{
case 'D':
case 'd': accountBalance = accountBalance
          + transactionAmount;
          amountDeposited = amountDeposited
          + transactionAmount;
          numberOfDeposits++;
          break;
case 'I':
case 'i': accountBalance = accountBalance
          + transactionAmount;
          interestPaid = interestPaid
          + transactionAmount;
          break;
case 'W':
case 'w': accountBalance = accountBalance
          - transactionAmount;
          amountWithdrawn = amountWithdrawn
          + transactionAmount;
          numberOfWithdrawals++;
          if((accountBalance < minimumBalance)
             && (!isServiceCharged))
          {
              accountBalance = accountBalance
                              - serviceCharge;
              isServiceCharged = true;
          }
          break;
default: cout<<"Invalid transaction code"<<endl;
} //end switch

```


6. 重复步骤 4 和 5, 直到数据结束。由于输入文件中条目的数量是未知的, 所以程序需要 EOF 控制 while 循环。
7. 输出结果。通过使用输出数据来完成。

通过上述讨论, 可以得出下面的主要算法。

主要算法

1. 声明和初始化变量。
2. 打开输入文件。
3. 如果输入文件不存在, 退出程序。
4. 打开输出文件。
5. 为了用小数点和尾零这种固定方式输出浮点数, 设置操纵符 fixed 和 showpoint。为了输出浮点数到小数点后面两位, 设置精度到小数点后两位。
6. 读入 accountNumber 和 beginningBalance。
7. 设置 accountBalance 为 beginningBalance。
8. 读入 transactionCode 和 transactionAmount。
9. While (不是文件输入末尾)
 - a. 如果 transactionCode 是 'D', 那么:
 - i. 将 accountBalance 与 transactionAmount 相加。
 - ii. 变量 numberOfDeposits 加 1。
 - b. 如果 transactionCode 是 'T', 那么:
 - i. 将 accountBalance 与 transactionAmount 相加。
 - ii. 将 interestPaid 与 transactionAmount 相加。
 - c. 如果 transactionCode 是 'W', 那么:
 - i. 将 accountBalance 与 transactionAmount 相减。
 - ii. 变量 numberOfWithdrawals 加 1。
 - iii. 如果 (accountBalance < minimumBalance && !isServiceCharged), 那么:
 1. 从 accountBalance 中减去 serviceCharge。
 2. 设置 isServiceCharged 为 true。
 - d. 如果 transactionCode 是除了 'D', 'd', 'T', 't', 'W', 'w' 之外的其他字符, 输出错误信息。
10. 输出结果。

由于数据从一个输入文件中读入, 所以必须包含头文件 fstream。由于要使用操纵符 setprecision(), 必须包含头文件 iomanip。如果输入文件不存在, 屏幕应该显示适当的信息, 所以还应该包含头文件 iostream。

完整程序清单

```

/*****
// Program -- Checking Account Balance
// This program calculates a customer's checking account
// balance at the end of the month.
*****/
#include <iostream>
#include <fstream>
#include <iomanip>

```

```

using namespace std;

const double minimumBalance = 1000.00;
const double serviceCharge = 25.00;

int main()
{
    //Declare and initialize variables           //Step 1
    int acctNumber;
    double beginningBalance;
    double accountBalance;

    double amountDeposited = 0.0;
    int numberOfDeposits = 0;

    double amountWithdrawn = 0.0;
    int numberOfWithdrawals = 0;

    double interestPaid = 0.0;

    char transactionCode;
    double transactionAmount;

    bool isServiceCharged = false;

    ifstream infile;
    ofstream outfile;

    infile.open("a:Ch5_money.txt");              //Step 2
    if(!infile)                                  //Step 3
    {
        cout<<"Cannot open input file"<<endl;
        cout<<"Program terminates!!!"<<endl;
        return 1;
    }

    outfile.open("a:Ch5_money.out");              //Step 4
    outfile<<fixed<<showpoint;                    //Step 5
    outfile<<setprecision(2);                      //Step 5

    cout<<"Processing data"<<endl;
    infile>>acctNumber>>beginningBalance;         //Step 6

    accountBalance = beginningBalance;             //Step 7

    infile>>transactionCode>>transactionAmount;   //Step 8

    while(infile)                                  //Step 9
    {
        switch(transactionCode)
        {
            case 'D':                               //Step 9a
            case 'd': accountBalance = accountBalance
                        + transactionAmount;
                        amountDeposited = amountDeposited
                        + transactionAmount;
                        numberOfDeposits++;
                        break;

            case 'I':                               //Step 9b
            case 'i': accountBalance = accountBalance
                        + transactionAmount;
                        interestPaid = interestPaid

```

```

        + transactionAmount;
        break;
    case 'W': //Step 9c
    case 'w': accountBalance = accountBalance
               - transactionAmount;
               amountWithdrawn = amountWithdrawn
               + transactionAmount;
               numberOfWithdrawals++;
               if((accountBalance < minimumBalance)
                  && (!isServiceCharged))
               {
                   accountBalance = accountBalance
                                   - serviceCharge;
                   isServiceCharged = true;
               }
               break;
    default: cout<<"Invalid transaction code"<<endl;
} //end switch

infile>> transactionCode>>transactionAmount;
} //end while

//Output Results //Step 10
outfile<<"Account Number: "<<acctNumber<<endl;
outfile<<"Beginning Balance: $"<<beginningBalance<<endl;
outfile<<"Ending Balance: $"<<accountBalance
<<endl<<endl;
outfile<<"Interest Paid: $"<<interestPaid<<endl<<endl;
outfile<<"Amount Deposited: $"<<amountDeposited<<endl;
outfile<<"Number of Deposits: "<<numberOfDeposits
<<endl<<endl;
outfile<<"Amount Withdrawn: $"<<amountWithdrawn<<endl;
outfile<<"Number of Withdrawals: "<<numberOfWithdrawals
<<endl<<endl;

if(isServiceCharged)
    outfile<<"Service Charge: $"<<serviceCharge<<endl;

return 0;
}

```

运行实例:

输出文件: (A:Ch5_money.out)

Account Number: 467343
 Beginning Balance: \$23750.40
 Ending Balance: \$24611.49

Interest Paid: \$366.24

Amount Deposited: \$2230.50
 Number of Deposits: 3

Amount Withdrawn: \$1735.65
 Number of Withdrawals: 6

输入文件: (A:Ch5_money.txt)

467343 23750.40
 W 250.00
 D 1200.00

```

W 75.00
I 120.74
W 375.00
D 580.00
I 245.50
W 400.00
W 600.00
D 450.50
W 35.65

```

注意：为了使用 C++ 风格的头文件编写该程序，应该将下面的语句：

```

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

```

替换为：

```

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>

```

可能还需要将语句：

```

outp<<fixed<<showpoint;

```

替换为：

```

outp.setf(ios::fixed,ios::floatfield);
outp.setf(ios::showpoint);

```

5.4 编程实例：斐波那契数列

在目前为止，已经讲过了一个循环的例子。while 循环主要用来读入数据，尽管它还有其他用途。在 C++ 中，当特定语句需要重复执行直到满足特定条件时，就需要用到 while 循环。下面使用一个 while 循环来求解斐波那契数列。

考虑下面序列中的数字：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

给定前两个数的序列后（比如说 a_1 和 a_2 ），那么该序列的第 n 个数字 a_n ($n \geq 3$) 由下面的公式给定：

$$a_n = a_{n-1} + a_{n-2}$$

这样：

$$\begin{aligned}
 a_3 &= a_2 + a_1 \\
 &= 1 + 1 \\
 &= 2 \\
 a_4 &= a_3 + a_2 \\
 &= 1 + 2 \\
 &= 3
 \end{aligned}$$

这样的序列称为斐波那契数列。在前面的序列中， $a_2=1, a_1=1$ 。然而，给定了前两个数字，使用该过程就可以求出序列中的第 n 个数字 a_n , $n \geq 3$ 。由这种方法求出的第 n 个数字称为斐波那契数。假设 $a_2=6, a_1=3$ ，那么：

$$a_3 = a_2 + a_1 = 6 + 3 = 9; a_4 = a_3 + a_2 = 9 + 6 = 15。$$

接着,我们编写一个程序,根据给定的前两个数字来求第 n 个斐波那契数。

输入 前两个斐波那契数和求解的斐波那契数。

输出 第 n 个斐波那契数。

问题分析和算法设计

为了求 a_{10} ,即序列中的第10个斐波那契数,首先必须求得 a_9 和 a_8 ,而它又需要求得 a_7 和 a_6 等等。所以为了求解 a_{10} ,首先必须求 $a_3, a_4, a_5, \dots, a_9$ 。

该讨论翻译成下面的算法:

1. 得到前两个斐波那契数。
2. 得到求解的斐波那契数下标。也就是序列中斐波那契数的位置 n 。
3. 将斐波那契数列的前两个元素相加计算下一个斐波那契数。
4. 重复步骤2,直到求得第 n 个斐波那契数。
5. 输出第 n 个斐波那契数。

变量 为了求得当前的斐波那契数必须知道前两个斐波那契数,所以需要下面的变量: `previous1` 和 `previous2`, 来保存斐波那契数列的前两个数字; 变量 `current` 来保存当前的斐波那契数。步骤2中的算法要重复的次数取决于斐波那契数的位置。例如,如果想计算第10个斐波那契数,必须将步骤3执行8次。(用户给出了斐波那契数列的前两个数字。)所以,需要一个变量来存储步骤3应该执行的次数。还需要一个变量来跟踪步骤3中循环控制变量已经执行的次数。所以需要5个变量来进行数据操作:

```
int previous1;    //variable to store the first
                  //Fibonacci number
int previous2,    //variable to store the second
                  //Fibonacci number
int current;      //variable to store the current
                  //Fibonacci number
int counter;      //loop control variable
int nthFibonacci; //variable to store the desired
                  //Fibonacci number
```

为了计算第3个斐波那契数,将`previous1`和`previous2`的值相加并且将结果存储在`current`中。为了计算第4个斐波那契数,要将第2个斐波那契数(也就是`previous2`)的值与第3个斐波那契数(也就是`current`)的值相加。这样,当计算出第4个斐波那契数时,不再需要第1个斐波那契数。不必声明额外的变量,否则这会需要很多的变量。在计算完一个斐波那契数后并求下一个斐波那契数时,`current`成为`previous2`, `previous2`成为`previous1`。所以,可以再一次使用`current`变量来存储下一个斐波那契数。该过程重复进行,直到求解出相应的斐波那契数。开始时,序列中的前两个元素是`previous1`和`previous2`,它们是由用户输入的。从前面的讨论中可看出,这一过程需要5个变量。

主要算法

1. 提示用户输入前两个数字,也就是`previous1`和`previous2`。
2. 读入(输入)前两个数字并赋值给`previous1`和`previous2`。
3. 输出前两个斐波那契数。(回显输入值。)
4. 提示用户输入求解的斐波那契数的位置。
5. 将求解斐波那契数的位置读入到变量`nthFibonacci`中。
6. 由于已经知道了序列中的前两个斐波那契数,所以开始计算第三个斐波那契数。将`counter`

初始化为 3 来跟踪计算的斐波那契数。

7. 计算下一个斐波那契数，如下所示：

```
current=previous2+previous1;
```

8. 将 previous2 的值赋给 previous1。

9. 将 current 的值赋给 previous2。

10. 将 counter 加 1。

11. 重复步骤 7~10，直到计算出相应的斐波那契数。下面的 while 循环执行了步骤 7~10 并且求出了斐波那契数。

```
while(counter <= nthFibonacci)
{
    current = previous2 + previous1;
    previous1 = previous2;
    previous2 = current;
    counter++;
}
```

12. 输出斐波那契数，也就是 current。

完整程序清单

```
//Program: nth Fibonacci number

#include <iostream>
using namespace std;

int main()
{
    //Declare variables
    int previous1;
    int previous2;
    int current;
    int counter;
    int nthFibonacci;

    cout<<"Enter the first two Fibonacci "
        <<"numbers -> "; //Step 1
    cin>>previous1>>previous2; //Step 2
    cout<<"\nThe first two Fibonacci numbers "
        <<"are "<<previous1<<" and "<<previous2; //Step 3
    cout<<"\nEnter the desired Fibonacci "
        <<"number to be determined ->"; //Step 4
    cin>>nthFibonacci; //Step 5
    counter = 3; //Step 6

    //Steps 7-10
    while(counter <= nthFibonacci)
    {
        current = previous2 + previous1; //Step 7
        previous1 = previous2; //Step 8
        previous2 = current; //Step 9
        counter++; //Step 10
    }

    cout<<"\n\nThe "<<nthFibonacci
        <<"th Fibonacci number is: "<<current
        <<endl; //Step 12
```



```

    return 0;
} //end main

```

运行实例： 在本运行实例中，用户输入用阴影表示。(程序执行3次。)

运行实例 1：

```

Enter the first two Fibonacci numbers -> 12 16
The first two Fibonacci numbers are 12 and 16
Enter the desired Fibonacci number to be determined -> 10
The 10th Fibonacci number is 796.

```

运行实例 2：

```

Enter the first two Fibonacci numbers -> 1 1
The first two Fibonacci numbers are 1 and 1
Enter the desired Fibonacci number to be determined -> 15
The 15th Fibonacci number is 610.

```

运行实例 3：

```

Enter the first two Fibonacci numbers -> 20 25
The first two Fibonacci numbers are 20 and 25
Enter the desired Fibonacci number to be determined -> 10
The 10th Fibonacci number is 1270.

```

5.5 for 循环 (重复) 结构

前面讨论的 while 循环对于大多数形式的重复来说是足够的。这里讨论的 for 循环结构是一种特殊形式的 while 循环。它的主要目的是为了简化计数控制循环的编写。由于这个原因，for 循环一般称为计数循环。

for 循环的通常语法是：

```

for(initial statement; loop condition; update statement)
    statement

```

括号中的 initial statement, loop condition 和 update statement (称为循环控制语句) 控制 for 语句体 (statement)。图 5.2 显示了 for 循环的执行流程。

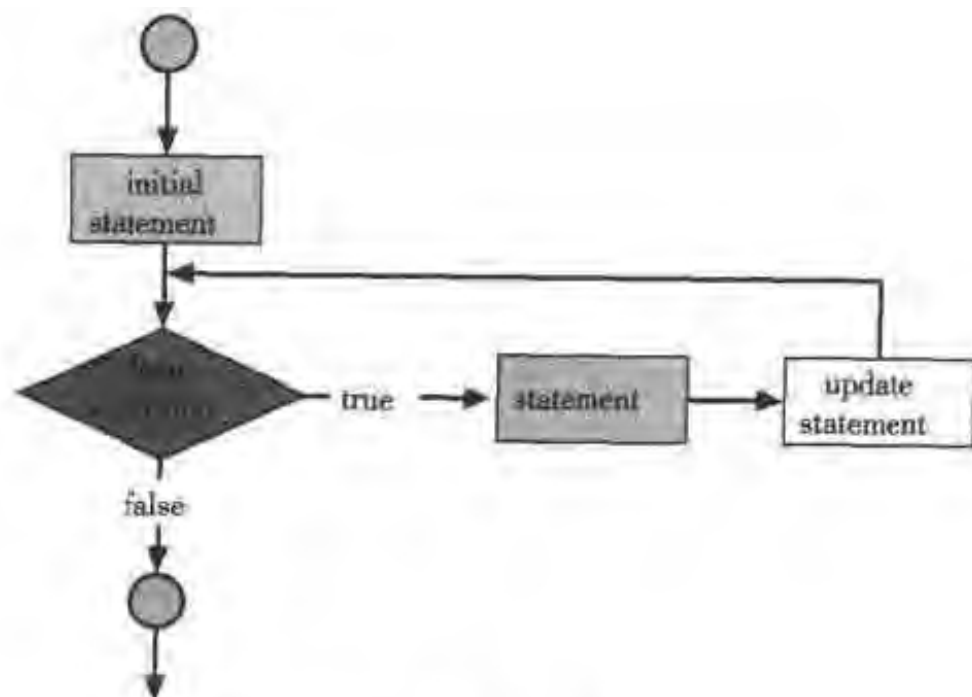


图 5.2 for 循环

for 循环执行如下：

1. initial statement 执行。
2. 计算 loop condition 的值。如果 loop condition 的值为 true，那么：
 - i. 执行 for 循环语句。
 - ii. 执行 update statement（括号中的第三个表达式）。
3. 重复步骤 2，直到 loop condition 的值为 false。

initial statement 通常初始化一个变量。

在 C++ 中，for 是一个保留字。

注意：顾名思义，for 循环中的 initial statement 是第一条执行的语句并且只执行一次。

例 5.6

下面的 for 循环打印 10 个正整数：

```
for(i = 1; i <= 10; i++)  
    cout<<i<<" ";
```

初始化语句“i=1;”将 int 变量 i 初始化为 1。接着，循环条件 i<=10 被计算。由于 i<=10 是 true，所以打印语句执行并输出 1。然后更新语句“i++”执行并将 i 设置为 2。然后再一次计算循环条件的值，它仍然是 true，等等。当 i 变为 11 时，loop condition 变为 false，for 循环就会终止，for 循环下面的语句就会被执行。

for 循环可以包含简单语句，也可以包含复合语句。

下面的例子进一步演示了 for 循环是如何执行的。

例 5.7

1. 下面的 for 循环输出 5 次文本和星号：

```
for(i = 1; i <= 5; i++)  
{  
    cout<<"Output a line of stars."<<endl;  
    cout<<"*"<<endl;  
}
```

2. 考虑下面的 for 循环：

```
for(i = 1; i <= 5; i++)  
    cout<<"Output a line of stars."<<endl;  
    cout<<"*"<<endl;
```

该循环输出 5 次文本行，但只输出一个星号。注意 for 循环只控制第一条 cout 语句，因为两条 cout 语句并没有组成一条复合语句。所以，第一条 cout 语句执行 5 次。在 for 循环执行后，第二条 cout 语句只执行一次。

3. 下面的 for 循环执行 5 条空语句：

```
for(i = 1; i <= 5; i++); //Line 1  
    cout<<"*"<<endl;    //Line 2
```

for 语句结尾的分号（第一行 cout 语句前）结束了 for 循环。for 循环的动作执行空语句。

前面的例子说明了在让 for 循环执行相应的语句时，一定要仔细处理。

下面是对 for 循环的一些说明:

- 如果循环条件开始是 false, 那么循环体不会执行。
- 当执行更新表达式时, 改变了循环控制变量的值, 它最终将循环条件的值设置为 false。如果循环条件总是 true, 那么 for 循环会进入死循环。
- C++ 允许使用 double 类型 (或任何实数类型) 的循环控制变量。因为不同的计算机会给出这些循环控制变量不同的结果, 所以应该避免使用这样的变量。
- for 语句 (在循环体之前) 末尾的分号是一个语义错误。在这种情况下, for 循环的执行空语句。
- 在 for 语句中, 如果 loop condition 被忽略, 那么默认条件为 true。
- 在 for 语句中, 可以忽略所有三条语句——initial statement, loop condition 和 update statement。

下面是一个合法的 for 循环:

```
for(;;)
    cout<<"Hello"<<endl;
```

例 5.8

1. 如果 for 循环控制表达式被正确地设置, 那么可以使用 for 循环进行向后计数。例如, 考虑下面的 for 循环:

```
for(i = 10; i >= 1; i--)
    cout<<" "<<i;
```

输出是:

```
10 9 8 7 6 5 5 4 3 2 1
```

for 循环中, 变量被初始化为 10。在每一次循环执行后, i 值减 1。只要 $i \geq 1$, 循环就会继续执行。

2. 可以使用固定的步长自加或自减循环控制变量。在下面的 for 循环中, 变量 i 初始化为 1, 在 for 循环的更新表达式中, i 被加 2。该 for 循环输出 10 个正奇数。

```
for(i = 1; i <= 20; i = i + 2)
    cout<<" "<<i;
```

例 5.9

考虑下面的例子, 其中 i 是 int 变量。

1. for(i = 10; i <= 9; i++)

```
    cout<<i<<" ";
```

for 循环中, 初始化语句将 i 设置为 10。由于初始化的循环条件 ($i \leq 9$) 是 false, 所以不会执行循环体中的语句。

2. for(i = 9; i >= 10; i--)

```
    cout<<i<<" ";
```

for 循环中, 初始化语句将 i 设置为 9。由于初始化的循环条件 ($i \geq 10$) 是 false, 所以不会执行相应的 cout 语句。

3. for(i = 10; i <= 10; i++)

```
    cout<<i<<" ";
```

在该 for 循环中, cout 语句执行一次。

4. for(i = 1; i <= 10; i++);

```
    cout<<i<<" ";
```

for 循环对 cout 语句没有影响。for 语句结尾的分号导致循环的动作是空语句。cout 语句单独

执行一次。

```
5. for(i = 1; ; i++)
    cout<<i<<" ";
```

在for循环中, 由于loop condition在for语句中被忽略, 所以loop condition总是true, 从而导致一个无限循环。

例 5.10

在这个例子中, for循环读入5个数字并且计算其和与平均值。考虑下面的程序代码, 其中i, newNum, sum 和 average 是int变量。

```
sum = 0;
for(i = 1; i <= 5; i++)
{
    cin>>newNum;
    sum = sum + newNum;
}

average = sum / 5;
cout<<"The sum is "<<sum<<endl;
cout<<"The average is "<<average<<endl;
```

for循环中, 在读入newNum后, 该值与sum相加。变量sum在for循环开始前初始化为0。这样, 在程序读入第一个数字并与sum的值相加后, 变量sum保存了第一个数字正确的sum值。

在下面的C++程序中, 建议仔细浏览每个步骤。

例 5.11

下面的C++程序计算前n个正整数的和。

```
//Program: Sum first n positive integers
//This program finds the sum of the first n positive integers.
#include <iostream>
using namespace std;

int main()
{
    int counter;    //loop control variable
    int sum;        //variable to store the sum of the numbers
    int N;          //variable to store the number of
                  //first positive integers to be added

    cout<<"Line 1: Enter the number of positive "
        <<"integers to be added:"<<endl;           //Line 1
    cin>>N;                                           //Line 2
    sum = 0;                                          //Line 3

    for(counter = 1; counter <= N; counter++)      //Line 4
        sum = sum + counter;                       //Line 5

    cout<<"Line 6: The sum of the first "<<N
        <<" positive integers is "<<sum<<endl;    //Line 6

    return 0;
}
```

运行实例: 在本运行实例中, 用户输入用阴影表示。

Line 1: Enter the number of positive integers to be added:

Line 6: The sum of the first 100 positive integers is 5050

第1行语句提示用户输入一个要被累加的正整数。第2行语句将用户输入的数字保存在N中,第3行语句将sum初始化为0。第4行的for循环执行N次。在for循环中,counter初始化为1,在循环每次重复之后会加1。所以counter范围从1到N。通过循环,counter的当前值与sum相加。所以,for循环执行后,sum包含了前N个值的和,在该例子中是100个正整数的累加和。

前面讲过,将一条控制结构语句放在另一个控制结构语句中称为嵌套。下面的编程实例演示了嵌套的一个简单例子,它也很好演示了计数循环。

5.6 编程实例: 数字分类

该程序读入一个给定的整数集合,然后分别输出奇数、偶数以及零的数量。

输入 20个整数: 正数、负数和零。

输出 零的数目、奇数的数目和偶数的数目。

问题分析和算法设计

在读入一个数字之后,需要判断它是奇数还是偶数。假设值存储在number中。将number除以2并且检查余数。如果余数是零,那么number是偶数,将偶数的数目加1,然后检查number是否为零。如果是零,那么将零的数目加1。如果余数不是零,那么将奇数的数目加1。

程序使用一条switch语句来判断number是奇数还是偶数。假设number是奇数,如果number是正数,那么除以2后余数是1;如果number是负数,那么除以2后余数是-1。如果number是偶数,那么无论正数还是负数,除以2得到的余数都为零。可以使用求余运算符%来计算余数。例如:

$6\%2=0$, $-4\%2=0$, $-7\%2=-1$, $15\%2=1$ 。

为输入的每一个数字重复前面分析数字的过程。

该讨论转化成下面的算法:

1. 对于整数集合中的每一个数字:
 - a. 获取数字。
 - b. 分析数字。
 - c. 将相应的数目加1。
2. 输出结果。

变量 为了计算零、奇数和偶数的数量,需要三个int类型的变量: zeros, evens 和 odds 来跟踪数目。还需要一个变量 number 来读入和保存需要分析的数字,另一个变量 counter 作为循环控制变量。所以,程序需要下面的变量:

```
int counter;           //loop control variable
int number;            //variable to store the number read
int zeros;             //variable to store the zero count
int evens;             //variable to store the even count
int odds;              //variable to store the odd count
```

很明显,必须将变量 zeros, evens 和 odds 初始化为零。当声明时可以直接初始化这些变量。

主要算法

1. 初始化变量。
2. 提示用户输入20个数字。

3. 对于输入集合中的数字:

- a. 读入数字。
- b. 输出数字 (回显输入值)。
- c. 如果数字是偶数, 那么:
 - i. 将偶数的数目加 1。
 - ii. 如果数字是零, 那么将零的数目加 1。

否则:

将奇数的数目加 1。

4. 输出结果。

在编写 C++ 程序前, 让我们详细地来描述步骤 1~4, 这样对于编写 C++ 指令会变得容易得多。

1. 初始化变量。可以在声明时初始化变量 zeros, evens 和 odds。
2. 使用一条输出语句提示用户输入 20 个数字。
3. 对于步骤 3, 可以使用一个 for 循环来处理和分析这 20 个数字。使用伪代码, 该步骤如下所示:

```
for(counter = 1; counter <= 20; counter++)
{
    read the number;
    output number;
    switch(number % 2)    //check the remainder
    {
        case 0: increment even count;
                if(number == 0)
                    increment zero count;
                break;
        case 1: case -1: increment odd count;
    } //end switch
} //end for
```

4. 输出结果。输出变量 zeros, evens 和 odds 的值。

完整程序清单

```
/**
 * *****
 * // Program: Counts zeros, odds, and evens
 * // This program counts the number of odd and even numbers.
 * // The program also counts the number of zeros.
 * *****
 */
#include <iostream>
#include <iomanip>

using namespace std;

const int N = 20;

int main ()
{
    //Declare variables
    int counter;    //loop control variable
    int number;     //variable to store the new number

    int zeros = 0;
    int odds = 0;
}
```

```

    int evens = 0; //Step1
    cout<<"Please enter "<<N<<" integers, "
        <<"positive, negative, or zeros."
        <<endl; //Step 2
    cout<<"The numbers you entered are --> "<<endl;
    for(counter = 1; counter <= N; counter++) //Step 3
    {
        cin>>number; //Step 3a
        cout<<setw(5)<< number; //Step 3b

        //Step 3c
        switch(number % 2)
        {
            case 0: evens++;
                    if(number == 0)
                        zeros++;
                    break;
            case 1:
            case -1: odds++;
        } //end switch
    } //end for
    cout<<endl;

    //Step 4
    cout<<"There are "<<evens<<" evens, "
        <<"which also includes "<<zeros<<" zeros"<<endl;
    cout<<"Total number of odds are: "<<odds<<endl;
    return 0;
}

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Please enter 20 integers, positive, negative, or zeros.
The numbers you entered are -->
0 0 -2 -3 -5 6 7 8 0 3 0 -23 -8 0 2 9 0 12 67 54
0 0 -2 -3 -5 6 7 8 0 3 0
-23 -8 0 2 3 0 12 67 54
There are 13 evens, which also includes 6 zeros
Total number of odds are: 7

```

5.7 do...while 循环 (重复) 结构

该部分描述了第三种类型的循环或重复结构，即do...while循环。do...while语句的形式如下所示：

```

do
    statement
while(expression);

```

当然，statement既可以是简单语句，也可以是复合语句。如果它是复合语句，那么需要将语句放在一个花括号内。图 5.3 显示了 do...while 循环的执行流程。

在 C++ 中，do 是一个保留字。

statement 首先执行，然后计算 expression 的值。如果 expression 的值为 true，那么 statement 重复执行。只要 do...while 语句中的 expression 是 true，那么就执行 statement。为了避免死循环，必须保证循环体包含一条最终使 expression 变为 false 的语句，从而使循环能够正确地结束。

例 5.12

```

i = 0;
do
{
    cout<<i<<" ";
    i = i + 5;
}
while(i <= 20);

```

该代码的输出是:

0 5 10 15 20

在输出值 20 后, 语句:

```
i=i+5;
```

将 i 的值变为 25, 所以 $i \leq 20$ 变为 false 并终止了循环的执行。

因为 while 和 for 循环都有入口条件, 所以这种循环可能不会执行循环体内相应的语句。另一方面, do...while 循环有一个出口条件, 所以至少执行一次。

例 5.13

考虑下面的两个循环:

(a)

```

i = 11;
while(i <= 10)
{
    cout<<i<<" ";
    i = i + 5;
}

```

(b)

```

i = 11;
do
{
    cout<<i<<" ";
    i = i + 5;
}
while(i <= 10);

```

在(a)中, 没有执行 while 循环体中的语句。在(b)中, do...while 循环输出了数字 11。

相应动作发生后再检查条件——在这种情况下, do...while 循环是很有用的。例如, 当读入数据时, 可先读入数据, 然后再与出口条件进行比较, 这就像医院中使用的监视设备。下面的程序演示了这种思想。

例 5.14

```

/*****
//  Program: Warning Message
//  This program reads a value and prints a warning
//  message, or rings a bell, when a value falls outside
//  a given range.
*****/

#include <iostream>
using namespace std;

const int low = 60;
const int high = 100;

```

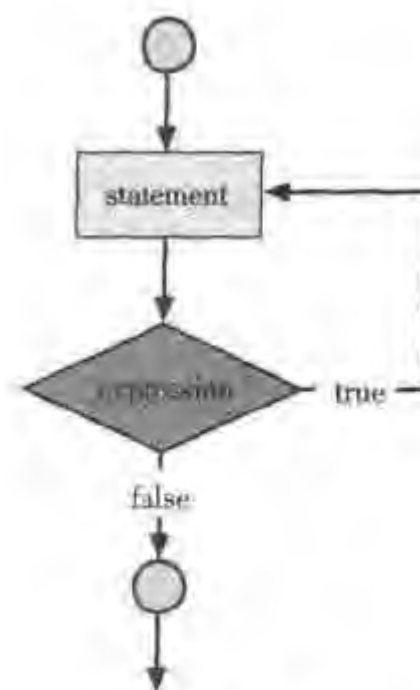


图5.3 do...while 循环

```

int main ()
{
    int pressure;
    do
        cin>>pressure;
    while((pressure > low) && (pressure < high));
    cout<<"Help"<<endl;
    return 0;
}

```

所有的三种循环在C++中都很有用处。通常可以用一种循环替换另一种循环,但这通常比较笨拙。例如,可以在程序 AVG2 中重写 while 循环(参看例 5.4)为 do...while 循环。如下所示:

```

if(number != -999)
do
{
    .
    .
    .
}
while(number != -999);

```

该结构通过下面的方式:

```

if(expression)
do
    action
while(expression);

```

替代了其相应的简洁方式:

```

while(expression)
    action

```

5.8 break 和 continue 语句

break 和 continue 语句改变了控制流程。当在 switch 结构中执行时, break 语句可以使控制流程从 switch 结构中立刻退出。同样,可以在 while, for 和 do...while 中使用 break 语句。当在一个循环结构中执行 break 语句时,它会立刻从循环体中跳出。一般地, break 语句应用于下面两种情况:

1. 从循环中退出。
2. 退出 switch 结构剩余部分。

在 break 语句执行后,程序继续执行相应结构后的第一条语句。

在一个循环中, break 的用法可以替代标志变量的使用。下面的 C++ 代码段演示了这种思想。(假设所有的变量已正确地声明。)

```

sum = 0;
cin>>num;
isNegative = false;
while(cin && !isNegative)
{
    if(num < 0)    //if the number is negative, terminate the loop
    {
        cout<<"Negative number found in the data"<<endl;

```



```

        isNegative = true;
    }
    else
    {
        sum = sum + num;
        cin >> num;
    }
}

```

假设使用 while 循环计算一个正数集合的和。如果数据集合包含了负数，那么循环就会终止并且显示相应的错误信息。while 循环使用标记变量 isNegative 执行相应的计算过程。变量 isNegative 在 while 循环前初始化为 false。在将 num 与 sum 相加之前，需要检查 num 是否是一个负数。如果 num 是一个负数，那么错误信息显示在屏幕上并且 isNegative 设置为 true。在下次循环中，当 while 语句中的表达式计算值时，结果为 false，因为 !isNegative 是 false。（注意因为 isNegative 是 true，所以 !isNegative 是 false）。

下面的 while 循环是没有使用变量 isNegative 来编写的。

```

sum = 0;
cin >> num;
while(cin)
{
    if(num < 0)    //if the number is negative, terminate the loop
    {
        cout << "Negative number found in the data" << endl;
        break;
    }

    sum = sum + num;
    cin >> num;
}

```

在这种 while 循环的形式中，当发现一个负数时，那么 if 语句中表达式的值为 true。在输出一条相应的信息之后，break 语句终止了该循环。（在执行循环中的 break 语句之后，循环中的剩余语句被忽略了。）

continue 语句用在 while，for 和 do...while 结构中。当 continue 语句在循环中执行时，它跳过循环中所有剩余的语句并且处理下一次循环。在 while 和 do...while 结构中，表达式（也就是循环控制测试）在 continue 语句之后立即被计算值。在 for 结构中，update statement 在 continue 语句后执行，然后执行 loop condition（也就是循环控制测试）。

如果前面的程序段读入一个负数，那么 while 循环就会终止。如果想丢弃负数并且读入下一个数字而不是终止程序，那么可以用 continue 语句替换 break 语句，如下面的例子所示：

```

sum = 0;
cin >> num;
while(cin)
{
    if(num < 0)
    {
        cout << "Negative number found in the data" << endl;
        cin >> num;
        continue;
    }

    sum = sum + num;
    cin >> num;
}

```

}

注意: 前面曾指出, 三种循环都有其各自的应用场合并且可以相互替换。然而, 在 while 和 do...while 语句中 continue 语句的执行不同于 for 结构中的执行。当 continue 语句在 while 或 do...while 循环中执行时, update statement 可能不执行。而在 for 结构中, update statement 总是执行的。

5.9 嵌套控制结构

该部分简单地复习了在本章和第4章所讨论的控制结构。在前面看到, 通过将一个控制结构放在另一个控制结构中, 可以取得戏剧性的和富有成效的结果。控制结构的嵌套呈现了新的活力, 同时也有一点点的复杂性。考虑下面的程序:

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

int main ()
{
    int studentId, testScore, count = 0;
    cin>>studentId;
    while(studentId != -1)
    {
        count++;
        cin>>testScore;
        cout<<"Student Id = "<<studentId<<"", test score = "
            <<testScore<<"", and grade = ";
        if(testScore >= 90)
            cout<<"A."<<endl;
        else
            if(testScore >= 80)
                cout<<"B."<<endl;
            else
                if(testScore >= 70)
                    cout<<"C."<<endl;
                else
                    if(testScore >= 60)
                        cout<<"D."<<endl;
                    else
                        cout<<"F."<<endl;
                cin>>studentId;
            } //end while
        cout<<endl<<"Students in class = "<< count<<endl;
        return 0;
    }
```

如果在 while 循环中将相应代码改写为 if 语句 (没有 else 或写为 switch 结构), 那么该程序将会如何工作呢?

考虑另一个嵌套的例子。假设有输出下面的图案:

```
*
**
***
****
*****
```

很明显, 程序将打印5行的星号。在第一行打印一个星号, 第二行两个星号, 等等。由于要输出5行, 所以需要从下面的for语句开始:

```
for(i=1;i<=5;i++)
```

在第一次循环中*i*的值是1, 在第二次循环中是2, 等等。可以使用*i*的值作为另一个嵌套for循环的限制条件, 从而控制一行中星号的数量。如下所示:

```
for(i = 1; i <= 5 ; i++)
{
    for(j = 1; j <= i; j++)
        cout<<"*";
    cout<<endl;
}
```

一步步浏览该程序, 可以看出for循环从*i*=1开始。当*i*=1时, 内部的for循环输出一个星号并且将光标移到下一行。当*i*=2时, 内部的for循环输出两个星号并且将光标移到下一行, 等等。该过程一直继续, 直到*i*变成6, 循环停止。

如果用下面的语句替换第一个for语句, 将会产生什么图案呢?

```
for(i=5;i>=1;i--)
```

为了了解更多嵌套for循环的内容, 可以参看本章的练习30。

5.10 小结

1. C++有三种循环(重复)结构: while, for 和 do...while。
2. while 语句的语法是:

```
while(expression)
    statement
```
3. 在C++中, while 是一个保留字。
4. 在while语句中, expression 两边的括号是很重要的, 它们标记表达式的开始和结尾。
5. statement 称为循环体。
6. while 循环体必须包含最终将 expression 设置为 false 的语句。
7. 计数控制的 while 循环使用计数器来控制循环。
8. 在计数器控制的while循环中, 必须在循环前初始化计数器, 而且循环体必须包含能改变计数器变量值的语句。
9. 标记是一个特殊的值, 它表示输入数据的末尾。标记必须类似于(也可以不同于)所有的数据项。
10. 标记控制的 while 循环使用一个标记来控制 while 循环。while 循环继续执行直到读入标记。
11. 在EOF控制的while循环中, 当程序检测到文件结尾标志时退出循环。
12. 在DOS环境中, 文件结尾标志是使用Ctrl+z来输入的。在Unix环境中, 文件结尾标志是使用Ctrl+d来输入的。
13. for 循环简化了计数控制的 while 循环。
14. 在C++中, for 是一个保留字。
15. for 循环的语法是:

```
for(initialize statement; loop condition; update statement)
    statement
```

statement 称为 for 循环体。

16. 在 for 循环 (for 循环体前面) 的末尾放入一个分号是语义错误。在这种情况下, for 循环执行空语句。
17. do...while 语句的语法是:

```
do
    statement
while(expression);
```

 statement 称为 do...while 循环体。
18. while 循环和 for 循环可能根本不执行, 但是 do...while 循环至少执行一次。
19. 在循环体内执行 break 语句会立刻终止循环。
20. 在循环体内执行 continue 语句会忽略循环中的剩余语句, 并且处理下一次循环。
21. 当在 while 循环或 do...while 循环中执行 continue 语句时, 循环体中的表达式 update statement 可能不执行。
22. 在 for 循环中的 continue 语句执行后, update statement 是下一条要执行的语句。

5.11 练习

1. 判断下面语句的对错。
 - a. 在计数器控制的 while 循环中, 不必初始化循环控制变量。
 - b. while 循环体可能根本不会执行。
 - c. 在一个 while 死循环中, while 表达式初始化为 false, 但是在循环一次之后, 它将一直是 true。
 - d. while 循环中:

```
J=0;
while(J<=10)
    J++;
```

 如果 J>10, 那么循环就会终止。
 - e. 标记控制的 while 循环是一种事件控制的 while 循环, 它的终止依赖于一个定值。
 - f. 循环是能让特定语句重复执行的一种控制结构。
 - g. 从未知长度的文件中读入数据时, EOF 控制的循环是一种好的选择。
 - h. 当 while 循环终止时, 控制首先返回到 while 语句前的语句, 然后控制立刻转到 while 循环后面的语句。
2. 下面 C++ 代码的输出是什么?

```
count = 1;
y = 100;
while(count < 100)
{
    y = y - 1;
    count++;
}
cout<<" y = "<<y<<" and count = "<<count<<endl;
```
3. 下面 C++ 代码的输出是什么?

```
num = 5;
while(num > 5)
    num = num + 2;
cout<<num<<endl;
```
4. 下面 C++ 代码的输出是什么?

```
num = 1;
while(num < 10)
{
    cout<<num<<" ";
    num = num + 2;
}
```

5. 下面的 while 循环何时终止?

```
ch = 'D';
while('A' <= ch && ch <= 'Z')
    ch = static_cast<char>(static_cast<int>(ch) + 1);
```

6. 假设输入是 38 45 71 4 -1。下面代码的输出是什么? 假设所有的变量已正确地声明。

```
cin>>sum;
cin>>num;
for(j = 1; j <= 3; j++)
{
    cin>>num;
    sum = sum + num;
}
cout<<"Sum = "<<sum<<endl;
```

7. 假设输入是 38 45 71 4 -1。下面代码的输出是什么? 假设所有的变量已正确地声明。

```
cin>>sum;
cin>>num;
while(num != -1)
{
    sum = sum + num;
    cin>>num;
}
cout<<"Sum = "<<sum<<endl;
```

8. 假设输入是 38 45 71 4 -1。下面代码的输出是什么? 假设所有的变量已正确地声明。

```
cin>>num;
sum = num;
while(num != -1)
{
    cin>>num;
    sum = sum + num;
}
cout<<"Sum = "<<sum<<endl;
```

9. 假设输入是 38 45 71 4 -1。下面代码的输出是什么? 假设所有的变量已正确地声明。

```
sum = 0;
cin>>num;
while(num != -1)
{
    sum = sum + num;
    cin>>num;
}
cout<<"Sum = "<<sum<<endl;
```

10. 改正下面的代码, 从而可以计算 10 个数字的和。

```
sum = 0;
while(count < 10)
    cin>>num;
    sum = sum + num;
    count++;
```

11. 下面程序的输出是什么?

```

#include <iostream>
using namespace std;
int main()
{
    int x, y, z;

    x = 4;    y = 5;
    z = y + 6;

    while(((z-x) % 4) != 0)
    {
        cout<<z<<" ";
        z = z + 7;
    }

    return 0;
}

```

12. 假设输入是:

58 23 46 75 98 150 12 176 145 -999

下面程序的输出是什么?

```

#include <iostream>
using namespace std;
int main()
{
    int num;
    cin>>num;
    while(num != -999)
    {
        cout<<num%25<<" ";
        cin>>num;
    }
    return 0;
}

```

13. 给定

```

for(i=12; i<=25; i++)
    cout<<i;

```

- 第7个输出的整数是_____。
- 语句产生了_____行输出。
- 如果i++改为i--,是否会产生编译错误?

14. 假设下面的代码被正确地插入到程序中, 指出代码的输出和格式。

```

num = 0;
for(i = 1; i <= 4; i++)
{
    num = num + 10 * (i - 1);
    cout<<num<<" ";
}

```

15. 假设下面的代码被正确地插入到程序中, 指出代码的输出和格式。

```

j = 2;
for(i = 0; i <= 5; i++)
{
    cout<<j<<" ";
    j = 2 * j + 3;
}
cout<<j<<" "<<endl;

```


16. 假设下面的代码被正确地插入到程序中:

```
s = 0;
for(i = 0; i < 5; i++)
{
    s = 2 * s + i;
    cout<<s<<" ";
}
```

a. s 最后的值是什么?

(i) 11 (ii) 4 (iii) 26 (iv) 以上全不是

b. 如果分号插入到 for 循环控制表达式右边的括号后, s 最后的值是什么?

(i) 0 (ii) 1 (iii) 2 (iv) 5 (v) 以上全不是

c. 如果用 0 替换 for 循环控制表达式中的 5, s 最后的值是什么?

(i) 0 (ii) 1 (iii) 2 (iv) 以上全不是

17. 指出下面每一条语句的输出结果 (如果有的话)。

a. for(i = 1; i <= 1; i++)

```
    cout<<"*";
```

b. for(i = 2; i >= 1; i++)

```
    cout<<"*";
```

c. for(i = 1; i <= 1; i--)

```
    cout<<"*";
```

d. for(i = 12; i >= 9; i--)

```
    cout<<"*";
```

e. for(i = 0; i <= 5; i++)

```
    cout<<"*";
```

f. for(i = 1; i <= 5; i++)

```
{
```

```
    cout<<"*";
```

```
    i = i+1;
```

```
}
```

18. 编写一个 for 语句, 将所有 1 到 100 的数字乘 3 后累加。

19. 下面程序的正确输出是什么?

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    int counter;
```

```
    for(counter = 7; counter <= 16; counter++)
```

```
        switch(counter % 10)
```

```
        {
```

```
            case 0: cout<<" ";
```

```
                break;
```

```
            case 1: cout<<"CF TEN ";
```

```
                break;
```

```
            case 2: case 8: cout<<"IS ";
```

```
                break;
```

```
            case 3: cout<<"NOT ";
```

```
                break;
```

```
            case 4: case 9: cout<<"DONE ";
```

```
                break;
```

```
            case 5: cout<<"WELL";
```

```
                break;
```

```
            case 6: cout<<".";
```

```
                break;
```

```

        case 7: cout<<"WHAT ";
                break;
        default: cout<<"Bad number. ";
    }
    cout<<endl;
    return 0;
}

```

20. 假设输入值是 5 3 8。下面代码的输出是什么？假设所有的变量已被正确地声明。

```

cin>>a>>b>>c;
for(j = 1; j < a; j++)
{
    d = b + c;
    b = c;
    c = d;
    cout<<c<<" ";
}
cout<<endl;

```

21. 下面 C++ 程序段的输出是什么？假设所有的变量已被正确地声明。

```

for(j = 0; j < 8; j++)
{
    cout<<j * 25<<" -";
    if(j != 7)
        cout<<(j + 1) * 25 - 1<<endl;
    else
        cout<<(j + 1) * 25<<endl;
}

```

22. 请指出下列代码中的错误。

```

#include <iostream>
using namespace std;
const int N = 2,137;

main ()
{
    int a, b, c, d;

    a := 3;
    b = 5;
    c = c + d;
    N = a + n;
    for(i = 3; i <= N; i++)
    {
        cout<<setw(5)<<i;
        i = i + 1;
    }
    return 0;
}

```

23. 下面哪一项只应用于while循环？哪一项只应用于do...while循环？哪一项可分别应用于这两种循环？

- 可视为一个条件循环。
- 循环体至少执行一次。
- 控制循环的逻辑表达式在进入循环前计算值。
- 循环体可能根本不执行。

24. 下面的每一项循环将执行多少次？每一项的输出是什么？


```

a.  x = 5;  y = 50;
    do
        x = x + 10;
    while(x < y);
    cout<<x<<" "<<y;
b.  x = 5;  y = 80;
    do
        x = x * 2;
    while(x < y);
    cout<<x<<" "<<y;
c.  x = 5;  y = 20;
    do
        x = x + 2;
    while(x >= y);
    cout<<x<<" "<<y;
d.  x = 5;  y = 35;
    while(x < y)
        x = x + 10;
    cout<<x<<" "<<y;
e.  x = 5;  y = 30;
    while(x <= y)
        x = x * 2;
    cout<<x<<" "<<y;
f.  x = 5;  y = 30;
    while(x > y)
        x = x + 2;
    cout<<x<<" "<<y;

```

25. 下面的程序读入一些数字并求和。数据类似于下面的数字（-1 作为结束标记）：

12 5 30 48 -1

下列代码段无法正常运行，请指出错误。

```

#include <iostream>
using namespace std;
int main()
{
    int total=0,
        count=0,
        number;
    do
    {
        cin>>number;
        total = total + number;
        count++;
    }
    while(number != -1);

    cout<<"The number of data read is "<< count<<endl;
    cout<<"The sum of the numbers entered is "<<total<<endl;
    return 0;
}

```

26. 使用与练习 25 相同的数据，下面的两个循环无法正常运行，请改正错误。

```

a.  cin>>number;
    while(number != -1)
        total = total + number;
    cin>>number;
    cout<<endl;

```

```

        cout<<total;
b. cin>>number;
   while(number != -1)
   {
       cin>>number;
       total = total + number;
   }
   cout<<endl;
   cout<<total;

```

27. 假设有下面的程序段:

```

for(number = 1; number <= 10; number++)
    cout<<setw(3)<<number;

```

编写具有相同功能的 while 和 do...while 循环。

28. 假设有下面的程序段:

```

j = 2;
for(i = 1; i <= 5; i++){
    cout<<setw(4)<<j;
    j = j + 5;
}
cout<<endl;

```

编写具有相同功能的 while 和 do...while 循环。

29. 下面程序的输出是什么?

```

#include <iostream>
using namespace std;
int main()
{
    int x, y, z;
    x = 4;    y = 5;
    z = y + 6;
    do
    {
        cout<<z<<" ";
        z = z + 7;
    }
    while(((z-x) % 4) != 0);
    return 0;
}

```

30. 为了了解嵌套 for 循环的运行机制, 浏览下面的程序段并给出每一种情况的正确输出。

```

a. int i, j;
   for(i = 1; i <= 5; i++)
   {
       for(j = 1; j <= 5; j++)
           cout<<setw(3)<<i*j;
       cout<<endl;
   }
b. int i, j;
   for(i = 1; i <= 5; i++)
   {
       for(j = 1; j <= 5; j++)
           cout<<setw(3)<<i;
       cout<<endl;
   }

```

```

c. int i, j;
   for(i = 1; i <= 5; i++)
   {
       for(j = (i + 1); j <= 5; j++)
           cout<<setw(5)<<j;
       cout<<endl;
   }
d. int i, j;
   for(i = 1; i <= 5; i++)
   {
       for(j = 1; j <= i; j++)
           cout<<setw(3)<<j;
       cout<<endl;
   }
e. const int m = 10;
   const int n = 10;
   int i, j;
   for(i = 1; i <= m; i++)
   {
       for(j = 1; j <= n; j++)
           cout<<setw(3)<<m*(i-1)+j;
       cout<<endl;
   }
f. int i, j;
   for(i = 1; i <= 9; i++)
   {
       for(j = 1; j <= (9 - i); j++)
           cout<<" ";
       for(j = 1; j <= i; j++)
           cout<<setw(1)<<j;
       for(j = (i - 1); j >= 1; j--)
           cout<<setw(1)<<j;
       cout<<endl;
   }

```

5.12 编程练习

1. 如果一个整数每一位数字的和能被9整除,那么该整数就能被9整除。编写一个程序,提示用户输入一个整数。然后该程序输出整数值和一条信息,显示该整数是否能被9整除。程序应该首先将每一位数字相加,然后检查数字的和是否能被9整除。
2. 编写一个程序,提示用户输入一个整数,然后输出该整数的每一位数字以及它们的和。例如,应该输出3456的每一位数字为3 4 5 6;输出8030的每一位数字为8 0 3 0;输出2345526的每一位数字为2 3 4 5 5 2 6;输出4000的每一位数字为4 0 0 0;输出数字-2345的每一位数字为2 3 4 5。
3. 编写一个程序,提示用户输入一个整数,然后输出反向排列的数字。例如,如果输入12345,那么输出应该是54321。输入5000则输出为0005,输入980则输出为089。
4. 重新编写例5.5。用switch结构替换第10行到第28行的语句,计算对应于大写字母的数字。
5. 例5.5只输出对应于大写字母的数字。重新编写该程序,让它能处理大写字母和小写字母并且输出相应的数字值。如果输入不是字母,那么程序必须输出相应的错误信息。

6. 编写一个程序读入一组整数, 然后分别输出奇数的和与偶数的和。
7. 编写一个程序, 提示用户输入一个正整数。然后它应该输出一条信息显示该数字是否是素数。(提示: 偶数 2 是素数, 如果奇数不能被小于等于其平方根的奇数整除, 那么该奇数是一个素数。)
8. 使用 while 循环编写一个程序执行下面的步骤:
 - a. 提示用户输入两个整数: firstNum 和 secondNum (firstNum 必须小于 secondNum)。
 - b. 输出 firstNum 与 secondNum 之间所有的奇数。
 - c. 输出 firstNum 与 secondNum 之间偶数的和。
 - d. 输出 1 到 10 之间的数字及其平方。
 - e. 输出在 firstNum 和 secondNum 之间奇数平方的和。
 - f. 输出所有的大写字母。
9. 使用 for 循环重写练习 8。
10. 使用 do...while 循环重写练习 8。
11. 大学招生办公室需要了解学生某些课程的成绩。现在, 输入文件中包含了学生课程的成绩, 文件中每一项包含字母代码以及 GPA 成绩。字母 f 代表女生, 字母 m 代表男生。文件中, 每一项占一行, 但总项的数量是未知的。编写程序, 计算并输出男生和女生的平均 GPA 成绩, 结果保留两位小数。
12. 假设以复利计算余额。P0 是最初的本金, INT 是年利率。如果 P1, P2 和 P3 分别是第一年, 第二年和第三年末的账面余额。那么:

$$P1 = P0 + P0 * INT = P0 * (1 + INT)$$

$$P2 = P1 + P1 * INT = P1 * (1 + INT) = P0 * (1 + INT) * (1 + INT) \\ = P0 * (1 + INT)^2$$

$$P3 = P2 + P2 * INT = P2 * (1 + INT) = P0 * (1 + INT) * (1 + INT) * (1 + INT) \\ = P0 * (1 + INT)^3$$

当储金存储在一个 IRA 账户中, 它通常不被征税, 直到在 59 岁之后提款。假设家人在 16 岁生日的时候为你申请了一个利率为 10% 的账户, 并且他忘记了这件事。(所以没有钱存入或提出)。在 60 岁生日的时候某人通知了你这件事。这些钱每年以 10% 的速度递增。编写程序读入最初的钱数, 并计算 60 岁生日时该账户中的本利和。

假设决定将钱再存一年。从 61 岁生日开始, 决定提出每年的利息。换言之, 提出利息, 剩余的钱保持不动。那么在你的余生, 每月将会有多少收入呢?

设计程序接受任何整数的输入。用最初的投资: 1700 美元, 3600 美元和 8500 美元进行测试。

(提示: 使用循环计算 60 岁生日的钱数。不要使用预定义的 pow() 函数。)

程序应该完成所有需要的计算, 并且输出所有相关的数据:

初始投资是 ____。如果 ____ 美元是允许以利率 10% 来计算的话, 那么在 ____ 年之后总额是 ____ 美元。

如果 ____ 美元是允许以利率 10% 来计算的话, 那么在 ____ (年+1) 之后总额是 ____ 美元。
该年期间所赚的利息是 ____ 美元。如果利息每年被提出, 那么此后每月收入是 ____ 美元。

第6章 用户自定义函数 I

本章要点:

- 学习标准(预定义)函数并掌握如何在程序中使用它们
- 学习编写用户自定义函数
- 了解函数返回值,包括实参和形参
- 掌握如何在程序中构造并使用带返回值的用户自定义函数

在第2章中,读者了解到一个C++程序是函数的集合,如main()函数。从第1章到第5章仅仅使用了main()函数,程序指令全部出现在一个函数中。这种技术只是适合比较短的程序。对于较大的程序,将全部的程序指令置于一个函数内是不实际的(尽管有可能),读者很快就会发现确实如此。读者必须学会将问题分解成可管理的多个部分。本章中首先讨论预定义函数,然后阐述用户自定义的函数。

让我们想像一下汽车厂。当制造一辆汽车时,并不是从基本的原材料开始制造,而是将预先制造好的部件集成到一起。一部分部件由公司自己制造,其余的部件由不同的公司制造。

函数就像建筑物的砖块,它们将复杂的程序分离为可管理的部分,而且还有其他的优点:

1. 在处理一个函数时,读者可以集中精力在程序的这一结构中构造、调试并完善该函数。
2. 不同的人可以同时完成不同的函数。
3. 如果一个函数在程序的多个位置使用或者在不同程序中使用,读者可以只编写一次而使用多次。

函数经常被称做模块。它们就像程序,可以将许多函数组合在一起从而组成一个大型程序。接下来讨论用户自定义函数时,就会看到情况确实如此。这种功能对于预定义函数来说不是很明显,因为这些函数的程序代码对用户来说是不可见的。然而,因为预定义函数已经为用户写好了,所以我们会先学习预定义函数,这样就可以在需要的时候使用它们。为了在程序中使用预定义函数,读者只需要了解如何使用这些函数。

6.1 标准(预定义)函数

在数学中,函数可以看做是参数与惟一值之间的规则或对应关系。因此,如果 $f(x) = 2x + 5$,那么 $f(1) = 7$, $f(2) = 9$, $f(3) = 11$ 。这里1、2和3是 f 的参数,而7、9和11是函数 f 相对应的值。

在C++中,函数的概念非常类似于上面的定义。有预定义函数(有时也称做标准函数)和用户自定义函数两种。一些预定义的数学函数有 $\text{abs}(x)$ 、 $\text{sqrt}(x)$ 和 $\text{pow}(x, y)$ 。

函数 $\text{pow}(x, y)$ 计算 x 的 y 次幂: $\text{pow}(x, y) = x^y$ 。例如, $\text{pow}(2, 3) = 8.0$, $\text{pow}(2.5, 3) = 15.625$ 。因为 $\text{pow}(x, y)$ 的值是double类型的,所以称函数 $\text{pow}()$ 是double类型的或者函数 $\text{pow}()$ 返回double类型的值。此外, x 和 y 称做函数 $\text{pow}()$ 的参数。函数 $\text{pow}()$ 有两个参数。

平方根函数 $\text{sqrt}(x)$ 在 $x \geq 0.0$ 时,计算 x 的非负平方根。例如, $\text{sqrt}(2.25)$ 等于1.5,函数 $\text{sqrt}()$ 是double类型的而且只有一个参数。

取整函数 `floor()`，计算不超过 x 的最大整数。例如，`floor(48.79)` 是 48.0，函数 `floor()` 是 `double` 类型的而且只有一个参数。

在 C++ 中，预定义函数被组织成独立的库。例如，头文件 `iostream` 包含 I/O 函数，而头文件 `cmath` 则包含数学函数。表 6.1 列出了一些预定义函数及其标准头文件。在标准头文件中，可以看到相应函数的声明，如参数类型、函数类型等规范。函数类型是函数返回值的数据类型。（对于其他预定义函数的列表，可参见附录 F。对于标准 C++ 头文件的名称，可参见附录 E 或者例子 6.1 后面的注意事项。）

表 6.1 预定义函数

函数	标准头文件	作用	参数类型	返回值类型
<code>abs(x)</code>	<code><cstdlib></code>	返回参数的绝对值： <code>abs(-7)=7</code>	<code>int</code>	<code>int</code>
<code>ceil(x)</code>	<code><cmath></code>	返回不小于 x 的最小整数值： <code>ceil(56.34)=57.0</code>	<code>double</code>	<code>double</code>
<code>cos(x)</code>	<code><cmath></code>	返回 x 的余弦值： <code>cos(0.0)=1.0</code>	<code>double</code> (弧度)	<code>double</code>
<code>exp(x)</code>	<code><cmath></code>	返回 e 的 x 次幂，这里 $e=2.718$ ： <code>exp(1.0)=2.718 28</code>	<code>double</code>	<code>double</code>
<code>fabs(x)</code>	<code><cmath></code>	返回参数的绝对值： <code>fabs(-5.67)=5.67</code>	<code>double</code>	<code>double</code>
<code>floor(x)</code>	<code><cmath></code>	返回不大于 x 的最大整数值： <code>floor(45.67)=45.00</code>	<code>double</code>	<code>double</code>
<code>pow(x, y)</code>	<code><cmath></code>	返回 x 的 y 次幂；如果 x 是负数， y 必须是整数： <code>pow(0.16, 0.5) = 0.4</code>	<code>double</code>	<code>double</code>
<code>tolower(x)</code>	<code><cctype></code>	如果 x 是大写，返回 x 的小写值。否则，返回 x	<code>int</code>	<code>int</code>
<code>toupper(x)</code>	<code><cctype></code>	如果 x 是小写，返回 x 的大写值。否则，返回 x	<code>int</code>	<code>int</code>

在程序中使用预定义函数，必须通过 `include` 语句包含相应函数声明的头文件。例如，为使用 `pow()` 函数，程序必须包括：

```
#include <cmath>
```

注意：在标准 C++ 中，包含数学函数声明的头文件是 `math.h`

例 6.1

该例子展示如何使用一些预定义函数。

```
//How to use predefined functions
#include <iostream>
#include <cmath>
#include <cctype>
#include <cstdlib>
using namespace std;
int main()
{
    int    x;
    double u,v;

    cout<<"Line 1: Uppercase a is "
         <<static_cast<char>(toupper('a'))
         <<endl;                                //Line 1
    u = 4.2;                                    //Line 2
    v = 3.0;                                    //Line 3
    cout<<"Line 4: "<<u<<" to the power of "
         <<v<<" = "<<pow(u,v)<<endl;           //Line 4
    cout<<"Line 5: 5 to the power of 4 = "
         <<pow(5,4)<<endl;                       //Line 5
    u = u + pow(3,3);                           //Line 6
}
```



```

    cout<<"Line 7: u = "<<u<<endl;           //Line 7
    x = -15;                                   //Line 8
    cout<<"Line 9: Absolute value of "<<x
        <<" = "<<abs(x)<<endl;               //Line 9
    return 0;
}

```

程序输出为:

```

Line 1: Uppercase a is A
Line 4: 4.2 to the power of 3 = 74.088
Line 5: 5 to the power of 4 = 625
Line 7: u = 31.2
Line 9: Absolute value of -15 = 15

```

该程序工作如下。第一行语句输出相应于 a 的大写字母 A。注意 toupper() 函数返回 int 值, 因此, 表达式 toupper('a') 的值是 65, 它是 'A' 的 ASCII 值。为了打印 A 而不是 65, 需要应用强制转换运算符。在第 4 行语句中, 函数 pow() 用来输出 u 的 v 次幂。在 C++ 术语中, 应该说函数 pow() 通过参数 u 和 v 被调用。在这种情况下, u 和 v 的值传递给函数 pow(), 其他语句具有相似的含义。

注意: 为了使用标准 C++ 头文件编写该程序, 需要替换语句:

```

#include <iostream>
#include <cmath>
#include <cctype>
#include <cstdlib>
using namespace std;
为:
#include <iostream.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>

```

6.2 用户定义函数

正如例子 6.1 所阐明的, 在程序中大量使用函数能够增强程序的可读性, 因为这样减少了函数 main() 的复杂性。同样, 读者完成调试了一个函数后, 就可以在程序 (或者不同的程序) 中多次使用它, 而不需要重复地编写相同的代码。例如, 在例子 6.1 中, 函数 pow() 被多次调用。

因为 C++ 没有提供程序员可能用到的所有函数, 而且设计者不可能了解用户的特殊需要, 所以用户必须学会编写自己的函数。

在 C++ 中用户自定义函数分为两类:

- 具有数据类型的函数, 也称做具有返回值的函数
- 不具有数据类型的函数, 也称做 void 函数

本章剩下的部分将讨论有返回值的函数。针对有返回值函数的很多概念也适用于 void 函数。我们将在第 7 章描述 void 函数。

6.3 具有返回值的函数

上一节介绍了一些预定义的 C++ 函数, 例如 pow()、abs()、islower() 和 toupper()。这些都是有返回值函数的例子。为了在程序中使用这些函数, 必须了解包含这些函数说明的头文件名字。还需

要在程序中使用 `include` 语句包括这些头文件并了解下列声明项目:

1. 函数的名字
2. 如果有参数, 参数的数量
3. 每一个参数的数据类型
4. 函数计算值 (即返回值) 的数据类型, 也称做函数的类型

因为函数的返回值是惟一的, 所以通常以下面三种方式中的一种使用该返回值:

- 为将来的计算而保存该返回值
- 在某些计算中使用该值
- 输出该值

这说明返回值函数既可以用于赋值语句中, 还可以用在像 `cout` 这样的输出语句中。也就是说, 返回值函数可以用于 (被调用) 表达式中。

在学习用户自定义返回值函数的语法之前, 让我们回顾一下与函数相关的事项, 前面已经描述了4个特性, 还有一个与函数 (包括返回值和 `void` 两类) 相关的特性:

5. 为实现任务所需要的代码

上述4个特性称做函数头部 (也叫做函数头); 第5个特性称做函数体。这5个特性结合在一起称做函数定义。例如, 对于函数 `abs()`, 函数头如下所示:

```
int abs(int number)
```

类似地, 函数 `abs()` 具有如下定义:

```
int abs(int number)
{
    if (number < 0)
        number = -number;
    return number;
}
```

在函数 `abs()` 头部声明的变量叫做函数 `abs()` 的形参。因此, 函数 `abs()` 的形参是 `number`。

例子 6.1 中的程序包含了几条使用函数 `pow()` 的语句。在 C++ 的语义中, 也称做调用函数 `pow()`。在本章的后面, 我们讨论在函数调用时将会发生什么情况。

假定函数 `pow()` 的头部是 `double pow (double base, double exponent)`。从函数 `pow()` 的头部可以看出, `pow()` 的形参是 `base` 和 `exponent`。设想以下的条件:

```
double u = 2.5;
double v = 3.0;
double x, y, w;

x = pow(u, v);           //Line 1
y = pow(2.0, 3.2);       //Line 2
w = pow(u, 7);           //Line 3
```

在第一行中, 函数 `pow()` 通过参数 `u` 和 `v` 被调用。在这种情况下, `u` 和 `v` 的值传递给函数 `pow()`。实际上, `u` 的数值复制到 `base` 中, `v` 的数值也复制到 `exponent` 中。在第一行函数 `pow()` 调用中的变量 `u` 和 `v` 称为该函数的实参。在第二行中, 函数 `pow()` 传递参数 2.0 和 3.2。在该调用中, 2.0 复制到 `base` 中, 3.2 复制到 `exponent` 中。此外, 在函数 `pow()` 的调用中, 实参分别是 2.0 和 3.2。同样地, 在第三

行中, 函数 `pow()` 的实参是 `u` 和 `7`, `u` 的数值复制到 `base` 中, `7` 复制到 `exponent` 中。

现在可以正式给出两种定义:

形参: 在函数头部声明的变量。

实参: 在函数调用中列出的变量或表达式。

对于预定义的函数, 只需要关心前4个特性。软件公司不会给出实际的源代码, 也就是函数体。否则在软件上的花费将很高。

语法: 具有返回值的函数

具有返回值的函数其语法为:

```
functionType functionName(formal parameter list)
{
    statements
}
```

在语法模板中, 函数类型是指函数返回值的类型。该类型也叫做具有返回值函数的数据类型。此外, 括号中的语句形成了函数体。

语法: 形参列表

形参列表的语法是:

```
dataType identifier, dataType identifier, ...
```

函数调用

调用一个具有返回值函数的语法为:

```
functionName(actual parameter list)
```

语法: 实参列表

实参列表的语法为:

```
expression or variable, expression or variable, ...
```

所以, 可以通过函数名以其实参列表 (如果有的话) 调用函数。

函数的实参列表可以是空的。但是, 如果形参列表是空的, 括号仍然是需要的。如果形参列表是空的, 具有返回值函数的函数头如下所示:

```
functionType functionName()
```

或:

```
functionType functionName(void)
```

如果形参列表是空的, 函数调用中的实参列表也是空的。在这种情况下 (指一个空的形参列表), 函数调用仍然需要括号。于是, 对空形参列表具有返回值的函数, 其调用方式是:

```
functionName()
```

函数调用中, 实参的个数和数据类型必须与形参保持一致。就是说, 实参和形参有一一对应的关系。(第7章将要讨论使用默认参数的函数。)

综上所述, 具有返回值的函数是在一个表达式中被调用的。这个表达式既可以是一个赋值语句的一部分也可以是输出语句的一部分。程序中的函数调用导致相应函数的函数体执行。

6.3.1 return 语句

当具有返回值的函数计算值时，函数通过 return 语句将该值返回。换句话说，它通过 return 语句将数值传递到函数外。

语法：return 语句

return 语句有如下语法：

```
return expr;
```

expr 是一个变量、常量或表达式。expr 被计算，所得值被返回。expr 计算出的数值，其数据类型必须符合其函数类型。

在 C++ 中，return 是一个保留字。

当 return 语句在函数中执行时，该函数立刻终止并将控制权交回给控制者。所以，当 return 语句在函数 main() 中执行时，该程序结束。

为了验证上述思想，让我们编写一个函数来计算两个数中较大的一个。由于函数比较两个数，就必须保证该函数有两个参数，并且这两个参数都是数值类型。假定数据的数据类型是浮点数（十进制），比如说 double。该函数命名为 larger。需要完成该函数的惟一工作就是实现函数体。因而，遵从函数的语法，可以像下面这样编写该函数：

```
double larger(double x, double y)
{
    double max;
    if(x >= y)
        max = x;
    else
        max = y;
    return max;
}
```

也可以如下所示：

```
double larger(double x, double y)
{
    if(x >= y)
        return x;
    else
        return y;
}
```

第一个函数 larger() 的构成需要使用一个额外的变量 max（称为内部声明，这里 max 是函数的内部变量）。第二个 larger() 函数不需要。

注意：1. 在函数定义中，x 和 y 是形参。

2. return 语句可以出现在函数的任意位置。一旦 return 语句执行，所有的后续语句被跳过。因此，计算结束时马上返回该值是一种好思路。

在编写完函数 larger() 后，下面的 C++ 代码演示了如何在函数 main() 中使用函数 larger()。

```
int main()
{
    double one, two, maxNum;
    cout<<"Larger of 5 and 6 is " //Line 1
```

```

        <<larger(5,6)<<endl; //Line 2
    cout<<"Enter two numbers: "; //Line 3
    cin>>one>>two; //Line 4
    cout<<endl; //Line 5
    cout<<"Larger of "<<one<<" and "<<two
        <<" is "<<larger(one,two)<<endl; //Line 6
    cout<<"Larger of "<<one<<" and 29 is "
        <<larger(one,29)<<endl; //Line 7
    maxNum = larger(38.45, 56.78); //Line 8
    cout<<"maxNum = "<<maxNum<<endl; //Line 9

    return 0;
}

```

- 注意：1. 在第2行中的表达式 `larger(5, 6)` 是一个函数调用，5和6是实参。
 2. 在第6行中的表达式 `larger(one, two)` 是一个函数调用，这里，one和two是两个实参。
 3. 在第7行中的表达式 `larger(one, 29)` 是一个函数调用，这里，one和29是两个实参。
 4. 在第8行中的表达式 `larger(38.45, 56.78)` 是一个函数调用。在该调用中，实参是38.45和56.78。语句中，函数 `larger()` 返回值赋给变量 `maxNum`。

注意：在函数调用中，只需要指定实参，不需要说明它的数据类型。

一旦编写完函数，就可以在程序中的任意位置使用它。函数 `larger()` 比较两个数并返回两个数中的较大值。现在，让我们编写另一个使用 `larger()` 函数的函数来比较三个数中的最大者。该函数名字为 `compareThree`。

```

double compareThree(double x, double y, double z)
{
    return larger(x, larger(y, z));
}

```

在函数 `compareThree()` 的定义中，首先计算 `y` 和 `z` 中的最大值，然后将它和 `x` 进行比较。最后，`return` 语句返回最大的数。此外，在函数的头部，`x`、`y` 和 `z` 都是形参。

6.3.2 函数原型

现在，关于如何在程序中编写和使用函数已经有了初步的了解，下一个问题就是用户自定义函数在程序中出现的次序。例如，将函数 `larger()` 放在函数 `main()` 的前面还是后面？应该将 `larger()` 放在 `compareThree()` 的前面还是后面？根据规则，必须在使用函数之前进行声明，函数 `main()` 会使用标识符 `larger`，逻辑上必须将 `larger()` 放在 `main()` 的前面。

实际上，C++ 程序员习惯上将函数 `main()` 放在所有其他用户自定义函数的前面。这种组织方式会产生一个编译错误，因为函数的编译顺序是以它们出现在程序中的顺序决定的。例如，如果函数 `main()` 放在函数 `larger()` 的前面，在编译函数 `main()` 时就会报告标识符 `larger` 没有定义。为了解决没有声明标识符的问题，可以将函数原型放在函数定义（包括 `main()` 定义）前面。

函数原型：只有函数头部而没有函数体。

语法：函数原型

具有返回值函数的函数原型是：

```
functionType functionName(parameter list);
```

注意函数原型以分号结尾。

对于函数 `larger()`，函数原型是：

```
double larger(double x, double y);
```

注意: 在编写函数原型时, 不需要必须指定参数列表中的变量名, 但必须指定每一个参数的数据类型。

可以重新编写函数 `larger()` 的函数原型, 如下所示:

```
double larger(double, double);
```

最终程序

现在读者已经对编写完整的程序、编译程序和运行程序十分了解了。下面的程序使用了函数 `larger()`、`compareThree()` 和 `main()` 来计算两个数中的最大者以及三个数中的最大者。

```
//Program: Largest of three numbers
#include <iostream>
using namespace std;

double larger(double x, double y);
double compareThree(double x, double y, double z);

int main()
{
    double one, two;                                     //Line 1
    cout<<"Line 2: Larger of 5 and 10 is "
        <<larger(5,10)<<endl;                             //Line 2
    cout<<"Line 3: Enter two numbers: ";                 //Line 3
    cin>>one>>two;                                         //Line 4
    cout<<endl;                                           //Line 5

    cout<<"Line 6: Larger of "<<one<<" and "
        <<two<<" is "<<larger(one,two)<<endl;             //Line 6
    cout<<"Line 7: Largest of 23, 34, and 12 is "
        <<compareThree(23,34,12)<<endl;                   //Line 7
    return 0;
}

double larger(double x, double y)
{
    if(x >= y)
        return x;
    else
        return y;
}

double compareThree (double x, double y, double z)
{
    return larger(x, larger(y, z));
}
```

运行实例: 在本运行实例中, 用户输入用阴影表示。

```
Line 2: Larger of 5 and 10 is 10
Line 3: Enter two numbers: 25.73
Line 6: Larger of 25 and 73 is 73
Line 7: Largest of 23, 34, and 12 is 34
```

注意: 在该程序中, 函数 `larger()` 和 `compareThree()` 的函数原型出现在函数定义之前。因此, 函数 `larger()` 和 `compareThree()` 的函数定义可以以任意的次序出现。

注意: 前面讲过, 在返回值函数中的 `return` 语句返回值。考虑下面的 `return` 语句:

return x, y; // 只有 y 的值会返回

这是一条合法的 return 语句。读者可能会认为该 return 语句返回 x 和 y 的值。记住，一条 return 语句只能返回一个值。如果 return 语句包含多于一个表达式，只有最后一个表达式的值会返回。因此上面的 return 语句中，y 的值被返回。下面的程序会进一步举例说明这一概念。

```
//A value returned by a return statement
//This program illustrates that a value-returning function
//returns only one value, even if the return statement
//contains more than one expression.

#include <iostream>
using namespace std;
int funcRet1();
int funcRet2();
int funcRet3();
int funcRet4(int z);

int main()
{
    int num = 4;
    cout<<"Line 1: Value returned by funcRet1: "
        <<funcRet1()<<endl;           //Line 1
    cout<<"Line 2: Value returned by funcRet2: "
        <<funcRet2()<<endl;           //Line 2
    cout<<"Line 3: Value returned by funcRet3: "
        <<funcRet3()<<endl;           //Line 3
    cout<<"Line 4: Value returned by funcRet4: "
        <<funcRet4(num)<<endl;         //Line 4
    return 0;
}

int funcRet1()
{
    return 23, 45; //Only 45 is returned
}

int funcRet2()
{
    int x = 5;
    int y = 6;

    return x, y; //Only the value of y is returned
}

int funcRet3()
{
    int x = 5;
    int y = 6;

    return 37, y, 2 * x; //Only the value of 2 * x is returned
}

int funcRet4(int z)
{
    int a = 2;
    int b = 3;

    return 2 * a + b, z + b; //Only the value of z + b is returned
}
```

输出如下所示:

```
Line 1: Value returned by funcRet1: 45
Line 2: Value returned by funcRet2: 6
Line 3: Value returned by funcRet3: 10
Line 4: Value returned by funcRet4: 7
```

下面是一个函数返回布尔值的例子。

例 6.2 回文数

在该例中, 函数 `isNumPalindrome()` 设计为在一个整数是回文数时返回 `true`, 否则返回 `false`。当且仅当一个整数向前读和向后读所得到的数字都是一样时, 该整数是一个回文数。例如, 整数 5, 44, 434, 1881 和 789656987 全都是回文数。

假设 `num` 是一个整数, 如果 `num < 10`, 那么它是一个回文数, 所以, 函数应该返回 `true`。假设 `num ≥ 10`, 为了确定 `num` 是否是一个回文数, 首先应该比较 `num` 第一个数字和最后一个数字, 如果 `num` 的第一个和最后一个数字不相同, 那么它不是一个回文数, 函数应该返回 `false`。如果 `num` 的第一个数字和最后一个数字是相同的, 就删除第一个数字和最后一个数字, 并且对在删除 `num` 的第一个数字和最后一个数字之后得到的新数字重复该过程。只要数字大于等于 10, 就重复这一过程。

例如, 假设输入是 18281, 因为 18281 第一个数字和最后一个数字是相同的, 所以为了得到数字 828, 应该删除第一个和最后一个数字。对数字 828 重复比较第一个和最后一个数字, 第一个数字和最后一个数字又是相同的, 在删除 828 的第一个和最后一个数字之后得到的数字是 2, 它小于 10, 所以 18281 是一个回文数。

为了删除 `num` 的第一个和最后一个数字, 首先需要求得 `num` 可以整除 10 后的最高次幂, 称它为 `pwr`。18281 能够整除 10 后的最高次幂是 4, 现在 $18281 \% 10^{pwr} = 8281$, 这样第一个数字就被删除了。同样地, 因为 $8281 / 10 = 828$, 最后一个数字也被删除了。所以, 为了删除第一个数字, 可以使用以 10^{pwr} 为除数的取模操作, 为了删除最后一个数字可以除以 10。对于下次重复操作, 需要 `pwr` 自减 2。下面的算法实现了上述讨论。

1. 如果 `num < 10`, 它是一个回文数所以函数应该返回 `true`。
2. 假定 `num` 是一个整数并且 `num ≥ 10`, 检查 `num` 是否是回文数。
 - a. 找到 `num` 整除 10 的最高次幂, 命名为 `pwr`。例如, 434 可以整除 10 的最高次幂是 2; 789656987 可以整除 10 的最高次幂是 8。
 - b. 当 `num` 大于等于 10, 比较 `num` 的第一位与最后一位。
 - b.1. 如果 `num` 的第一位数字和最后一位数字不相等, `num` 就不是一个回文数, 返回 `false`。
 - b.2. 如果 `num` 的第一位数字和最后一位数字相等。
 - b.2.1 删除 `num` 的第一位数字和最后一位数字。
 - b.2.2 `pwr` 自减 2。
 - c. 返回 `true`。

下面的函数实现了该算法:

```
bool isNumPalindrome(int num)
{
    int pwr = 0;

    if(num < 10)                                //Step 1
        return true;
    else                                        //Step 2
    {
```



```

while (num / static_cast<int>(pow(10,pwr)) >= 10) //Step 2.a
    pwr++;
while (num >= 10) //Step 2.b
{
    if((num / static_cast<int>(pow(10,pwr))) != (num % 10))
        return false; //Step 2.b.1
    else //Step 2.b.2
    {
        num = num % static_cast<int>(pow(10,pwr)); //Step 2.b.2.1
        num = num / 10; //Step 2.b.2.1
        pwr = pwr - 2; //Step 2.b.2.2
    }
} //end while
return true;
} //end else
}

```

注意：在函数 `isNumPalindrome()` 的定义中，头文件 `cmath` 中的函数 `pow()` 用来找到可以被 `number` 整除 10 的最高次幂。所以，要保证在程序中包含头文件 `cmath`。

6.3.3 执行流程

前面讲过，一个 C++ 程序是一个函数的集合。回忆一下，函数可以以任意的顺序出现。需要记住的事情是必须在使用之前进行函数声明。编译器顺序地从开始到结尾编译程序。这样，如果函数 `main()` 出现在其他用户自定义函数之前，它就会首先被编译。然而，如果 `main()` 出现在程序的末尾或者中间，那么所有定义出现在函数 `main()` 之前的函数，都会因为它们所处的位置而在函数 `main()` 之前编译。

函数原型出现在任何函数定义之前，所以编译器首先解析它们。然后编译器才可以正确地解释函数调用。然而，在程序执行时，函数 `main()` 中的第一条语句总是首先执行的，而不管在程序中函数 `main()` 位于何处。其他函数只有在被调用的时候执行。

函数调用语句将控制转移到函数体的第一条语句。一般而言，在被调用函数的最后一条语句执行之后，控制立即转回给函数调用之后的地点。返回值函数返回一个值，这样，对于一个返回值函数，在执行函数之后控制返回到调用者时，函数返回的值取代函数调用语句。执行流程会在紧跟函数调用之后的地点继续下去。

6.4 编程实例：最大数

在该编程实例中，函数 `larger()` 用来从一组数中确定最大的数。出于举例说明的目的，程序从 10 个数中选取最大数。可以很容易地增强该程序以适应任意数量的数值。

输入 10 个数字的集合

输出 10 个数中的最大数

问题分析和算法设计

假定输入数据是：

15 20 7 8 28 21 43 12 35 3

读入数据集合中的第一个数字，因为这是此时读入的惟一数字，所以假设该数字即是最大数并称为 `max`。读入第二个数字并且称为 `num`。现在比较 `max` 和 `num` 并将较大的数存入 `max`。现在 `max`

保存前两个数中的较大者。读入第三个数，将它与 max 比较并将较大者存入 max 中。此时 max 包含前三个数中的最大者。读入下一个数字，与 max 做比较并将较大数存入 max。对数据集合中剩余的每一个数重复这一过程。最终，max 会包含数据集合中的最大数。上述讨论翻译为如下算法：

1. 读入第一个数。因为这是读入的惟一数，它是目前的最大数，将它保存在变量 max 中。
2. 对于列表中每一个剩余的数字：
 - a. 读入下一个数字，保存在变量 num 中。
 - b. 比较 num 和 max，如果 $\text{max} < \text{num}$ ，num 就是新的最大数字，所以通过将 num 的值复制到 max 中，从而更新 max 值。如果 $\text{max} \geq \text{num}$ ，丢弃 num。
3. 因为现在 max 保存最大数字，将其值输出。

为了求得两个数字中的较大数，程序将使用函数 larger()。

完整程序清单

```
//Program: Largest
#include <iostream>
using namespace std;

double larger(double x, double y);

int main()
{
    double num; //variable to hold the current number
    double max; //variable to hold the larger number
    int count; //loop control variable

    cout<<"Enter 10 numbers."<<endl;
    cin>>num; //Step 1
    max = num; //Step 1

    for(count = 1; count < 10; count++) //Step 2
    {
        cin>>num; //Step 2a
        max = larger(max, num); //Step 2b
    }

    cout<<"The largest number is "<<max<<endl; //Step 3

    return 0;
} //end main

double larger(double x, double y)
{
    if(x >= y)
        return x;
    else
        return y;
}
```

运行实例：在本运行实例中，用户输入用阴影表示。

```
Enter 10 numbers.
10 56 73 42 22 67 88 26 62 11
The largest number is 88
```

6.5 编程实例：有线电视公司账单问题

在第4章中描述了一个为有线电视公司计算账单的程序。程序中，所有的编程指令包含在函数

main()中。这里,将以用户自定义函数重新编写该程序并举例说明如何组织程序。问题分析部分给出了如何将一个复杂问题分解为小的子问题。它也给出了在解决特殊子问题时,应该只将注意力集中在问题的一部分上。

程序的输入和输出和之前一样。

问题分析和算法设计

因为有两种类型的消费者:家庭用户和商务用户,程序包含两个单独的函数:一个用来计算家庭消费者的账单,另一个用来计算商务消费者的账单。函数计算账单总额并将其返回给函数main(),函数main()打印应付款额。可以调用计算家庭消费者账单的函数residential()和计算商务消费者账单的函数business()。计算账单的公式同之前一样。

正如在第4章中所述,对于账单手续费、服务连接等费用,其数据值是特定的。因此,它们作为常量来声明。

函数residential()用来计算家庭消费者账单,首先需要了解用户预订附加频道的数量。基于附加频道的数量,才能够计算出账单总额,在计算账单总额之后,函数通过return语句返回值。下面的4个步骤描述了该函数:

- a. 提示用户输入附加频道的数量
- b. 读入附加频道的数量
- c. 计算账单
- d. 返回应付款额

函数包含提示用户输入附加频道数量的语句(步骤a),并读入附加频道的数量(步骤b)。计算账单总额所需要的其他项,例如服务连接费和账单手续费,被定义成常量(在函数main()的定义前面)。因此,为了计算账单总额,函数不需要从函数main()中获得任何值。因此,该函数没有参数。

局部变量(函数residential()) 根据上述讨论,函数residential()需要变量来存储附加频道的数量和账单总额。该函数只需要两个局部变量来计算账单总额:

```
int    noOfPChannels; //number of premium channels
double bAmount;      //billing amount
```

函数residential()的定义如下所示:

```
double residential()
{
    int    noOfPChannels;
    double bAmount;

    cout<<"Enter the number of premium "
         <<"channels used : ";           //Step a
    cin>>noOfPChannels;                  //Step b

    bAmount= rBillProcessingFee +        //Step c
             rBasicServiceCost +
             noOfPChannels * rCostOfAPremiumChannel;

    return bAmount;                      //Step d
}
```

函数business()用于计算商务消费者账单,用户需要知道服务连接数量和消费者所预订的附加频道数量。然后,基于这些数据计算账单总额。接下来使用return语句返回账单总额。下面6

个步骤描述了该函数:

- a. 提示用户输入服务连接数量。
- b. 读取服务连接的数量。
- c. 提示用户输入附加频道的数量。
- d. 读取附加频道的数量。
- e. 计算账单总额。
- f. 返回应付款数量。

该函数包含提示用户输入服务连接数量和附加频道数量的语句(步骤a和c),并将其读入相应变量(步骤b和d)。其他计算账单总额所需要的项,例如服务连接费用和账单手续费,作为常量来定义(在函数main()的定义之前)。当计算账单总额时,该函数不需要从函数main()获得任何值。因此,它不需要参数。

局部变量(函数business()) 根据上述讨论,函数business()需要变量来保存服务连接和附加频道的数量。实际上,该函数只需要三个局部变量来计算账单总额:

```
int noOfBasicServiceConnections;  
int noOfPChannels; //number of premium channels  
double bAmount;    //billing amount
```

此时,函数business()的定义如下所示:

```
double business()  
{  
    int noOfBasicServiceConnections;  
    int noOfPChannels;  
    double bAmount;  
  
    cout<<"Enter the number of basic "  
        <<"service connections: ";           //Step a  
    cin>>noOfBasicServiceConnections;       //Step b  
  
    cout<<"Enter the number of premium "  
        <<"channels used :";                 //Step c  
    cin>>noOfPChannels;                     //Step d  
  
    if(noOfBasicServiceConnections <= 10)    //Step e  
        bAmount = bBillProcessingFee + bBasicServiceCost +  
                    noOfPChannels * bCostOfAPremiumChannel;  
    else  
        bAmount = bBillProcessingFee + bBasicServiceCost +  
                    (noOfBasicServiceConnections -10) *  
                    bBasicConnectionCost +  
                    noOfPChannels * bCostOfAPremiumChannel;  
  
    return bAmount;                          //Step f  
}
```

主要算法(函数main())

1. 以固定的带有小数点和尾零的十进制格式输出浮点数,设置操纵符fixed和showpoint。
2. 以两个小数位输出浮点数,设置精度为两个小数位。
3. 提示用户输入账号。
4. 读取账号。
5. 提示用户输入用户类型。

6. 读取用户类型。
7. a. 如果用户类型是 R 或者 r:
 - i. 调用函数 residential() 计算账单。
 - ii. 输出账单。
- b. 如果用户类型是 B 或者 b:
 - i. 调用函数 business() 计算账单。
 - ii. 输出账单。
- c. R, r, B 和 b 之外的用户类型视为无效用户。

完整程序清单

```
//Cable company billing program
#include <iostream>
#include <iomanip>
using namespace std;

//named constants; residential customers
const double rBillProcessingFee = 4.50;
const double rBasicServiceCost = 20.50;
const double rCostOfAPremiumChannel = 7.50;

//named constants; business customers
const double bBillProcessingFee = 15.00;
const double bBasicServiceCost = 75.00;
const double bBasicConnectionCost = 5.00;
const double bCostOfAPremiumChannel = 50.00;

double residential(); //Function prototype
double business();    //Function prototype

int main()
{
    //declare variables
    int    accountNumber;
    char   customerType;
    double amountDue;

    cout<<fixed<<showpoint;           //Step 1
    cout<<setprecision(2);             //Step 2

    cout<<"This program computes a cable bill."
        <<endl;
    cout<<"Enter account number: ";    //Step 3
    cin>>accountNumber;                //Step 4
    cout<<endl;

    cout<<"Enter customer type: R, r "
        <<"(Residential), B, b (Business): "; //Step 5
    cin>>customerType;                 //Step 6

    switch(customerType)               //Step 7
    {
        case 'r':                     //Step 7a
        case 'R': amountDue = residential(); //Step 7a.i
            cout<<"Account number = "
                <<accountNumber<<endl; //Step 7a.ii
            cout<<"Amount due = $"
                <<amountDue<<endl;    //Step 7a.ii
            break;
        case 'b':                     //Step 7b
```

```

        case 'B': amountDue = business();           //Step 7b.i
                  cout<<"Account number = "
                    <<accountNumber<<endl;         //Step 7b.ii
                  cout<<"Amount due = $"
                    <<amountDue<<endl;             //Step 7b.ii
                  break;
        default: cout<<"Invalid customer type."
                  <<endl;                           //Step 7c
    }
    return 0;
}

double residential()
{
    int    noOfPChannels; //number of premium channels
    double bAmount;       //billing Amount

    cout<<"Enter the number of premium "
        <<"channels used: ";
    cin>>noOfPChannels;
    bAmount = rBillProcessingFee +
              rBasicServiceCost +
              noOfPChannels * rCostOfAPremiumChannel;

    return bAmount;
}

double business()
{
    int    noOfBasicServiceConnections;
    int    noOfPChannels; //number of premium channels
    double bAmount;       //billing Amount

    cout<<"Enter the number of basic "
        <<"service connections: ";
    cin>>noOfBasicServiceConnections;

    cout<<"Enter the number of premium "
        <<"channels used: ";
    cin>>noOfPChannels;

    if(noOfBasicServiceConnections <= 10)
        bAmount = bBillProcessingFee + bBasicServiceCost +
                  noOfPChannels * bCostOfAPremiumChannel;
    else
        bAmount = bBillProcessingFee + bBasicServiceCost +
                  (noOfBasicServiceConnections -10) *
                  bBasicConnectionCost +
                  noOfPChannels * bCostOfAPremiumChannel;

    return bAmount;
}

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

This program computes a cable bill.
Enter account number: 21341

Enter customer type: R (Residential), B (Business): B
Enter the number of basic service connections: 25
Enter the number of premium channels used: 9
Account number = 21341
Amount due = $615.00

```

注意：为了使用标准 C++ 头文件编写该程序，将如下语句：

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

替换为:

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

同样,也需要将:

```
cout<<fixed<<showpoint;
```

替换为:

```
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::showpoint);
```

6.6 小结

1. 函数就像小型程序并被称做模块。
2. 函数能帮助程序员将程序分解成可管理的任务。
3. C++ 系统提供标准 (预定义) 函数。
4. 为了使用标准函数, 必须:
 - i. 包含函数声明的头文件名。
 - ii. 在程序中包含头文件。
 - iii. 知道函数的名字和类型、参数的数量和类型。
5. 有两种用户自定义函数: 具有返回值的函数和 void 函数。
6. 在函数头部的变量定义称做形参。
7. 函数调用时使用的表达式、变量或者常量值称做实参。
8. 函数调用时, 实参的数量和类型必须和已给定顺序的形参数量和类型相匹配。
9. 函数调用时, 应使用具有实参列表的函数名字。
10. 具有返回值的函数一定会返回一个值。所以, 返回值函数通常用在表达式中或者输出语句中。
11. 通常用户自定义函数的语法是:

```
functionType functionName(formal parameter list)
{
    statements
}
```
12. `functionType functionName (formal parameter list)` 称做函数头部 (或者函数头)。在 {和} 之间的语句称做函数体。
13. 函数头和函数体统称为函数定义。
14. 如果函数没有参数, 在函数头和函数调用处仍然需要空的圆括号。
15. 程序员可以通过空的圆括号或者括号中带有 void 来指定空的形参列表。
16. 具有返回值的函数通过 return 语句返回一个值。
17. 函数可以有多于一条的 return 语句。然而, 不论 return 语句在函数的任何位置执行, 剩下的语句都将被跳过并且退出函数。
18. 一条 return 语句只能返回一个值。
19. 函数原型是一个没有函数体的函数头, 函数原型以分号结尾。

20. 函数原型声明了函数类型、函数使用参数的数量和类型。
21. 在函数原型中,形参列表中的变量名是可选的。
22. 函数原型有助于编译器正确地解析每一个函数调用。
23. 在程序中,函数原型位于每一个函数定义之前。
24. 在使用函数原型时,用户自定义函数可以以任意顺序出现在程序中。
25. 程序执行时,总是从函数 `main()` 的第一条语句开始。
26. 用户自定义函数只是在它们被调用时执行。
27. 对函数的调用将控制从调用者转移到被调用的函数。
28. 在函数调用语句中,只需指定实参,不需要指定其数据类型和函数类型。
29. 在函数退出时,控制转移回调用者。

6.7 练习

1. 判断下面语句的对错。
 - a. 为了在程序中使用标准函数,程序员只需要知道函数的名字和如何使用该函数。
 - b. 一个具有返回值的函数只能返回一个值。
 - c. 每次函数被调用时,参数允许使用不同的值。
 - d. `return` 语句在用户自定义函数中执行时,函数立即退出。
 - e. 一个具有返回值的函数只能返回整型值。
2. 下面 C++ 程序的输出是什么? (注意,函数 `sqrt()` 返回参数的平方根。例如, `sqrt(16.0)=4.0`。函数 `sqrt()` 的声明位于头文件 `cmath` 中。)

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

int main()
{
    int counter;

    for(counter = 1; counter <= 100; counter++)
        if(pow(floor(sqrt(counter)),2) == counter)
            cout<<counter<<" ";

    cout<<endl;
    return 0;
}
```
3. 下面哪一项函数头是有效的? 如果无效,请解释原因。
 - a. `one(int a, int b)`
 - b. `int thisone(char x)`
 - c. `char another(int a,b)`
 - d. `double yetanother`
4. 考虑下列语句:

```
double num1, num2, num3;
int int1, int2, int3;
int value;

num1 = 5.0; num2 = 6.0; num3 = 3.0;
int1 = 4; int2 = 7; int3 = 8;
```

和函数原型:


```
double cube(double a, double b, double c);
```

下列语句中的哪一项是有效的？如果它们是无效的，请解释原因。

- a. `value = cube (num1, 15.0, num3);`
- b. `cout<<cube(num1, num3, num2);`
- c. `cout<<cube(6.0, 8.0, 10.5);`
- d. `cout<<num1<<num3;`
- e. `cout<<cube(num1, num3);`
- f. `value = cube(num1, int2, num3);`
- g. `value = cube(7, 8, 9);`

5. 考虑下列函数：

```
int secret(int x)
{
    int i, j;
    i = 2 * x;
    if (i > 10)
        j = x / 2;
    else
        j = x / 3;
    return j-1;
}

int another(int a, int b)
{
    int i, j;
    j = 0;
    for(i = a; i <= b; i++)
        j = j + i;
    return j;
}
```

下列每一个程序段的输出是什么？

- a. `x = 10;`
`cout<<secret(x)<<endl;`
- b. `x = 5; y = 8;`
`cout<<another(x,y)<<endl;`
- c. `x = 10; k = secret(x);`
`cout<<x<<" "<<k<<" "<<another(x,k)<<endl;`
- d. `x = 5; y = 8;`
`cout<<another(y,x)<<endl;`

6. 考虑下列函数原型：

```
int test(int, char, double, int);
double two(double, double);
char three(int, int, char, double);
```

回答下列问题：

- a. 函数 `test()` 有多少个参数？函数 `test()` 的类型是什么？
- b. 函数 `two()` 有多少个参数？函数 `two()` 的类型是什么？
- c. 函数 `three()` 有多少个参数？函数 `three()` 的类型是什么？
- d. 调用函数 `test()` 需要多少个实参？每一个参数的类型是什么？在对函数 `test()` 的调用中以什么顺序使用这些参数？
- e. 编写一条 C++ 语句，该语句打印具有实参 5, 5, 7.3 和 `z` 的函数 `test()` 所返回的值。
- f. 编写一条 C++ 语句，该语句打印具有实参 17.5 和 18.3 的函数 `two()` 所返回的值。
- g. 编写一条可以打印函数 `three()` 返回字符的 C++ 语句（使用自己的实参）。

7. 考虑下列函数:

```
int mystery(int x, double y, char ch)
{
    int u;
    if('A' <= ch && ch <= 'R')
        return(2 * x + static_cast<int>(y));
    else
        return(static_cast<int>(2*y)-x);
}
```

下列 C++ 语句的输出是什么?

- a. `cout<<mystery(5,4.3,'B')<<endl;`
- b. `cout<<mystery(4,9.7,'v')<<endl;`
- c. `cout<<2*mystery(6,3.9,'D')<<endl;`

8. 考虑下列函数:

```
int secret(int one)
{
    int I;
    int prod = 1;

    for(I = 1; I <= 3; I++)
        prod = prod * one;
    return prod;
}
```

a. 下列 C++ 语句的输出是什么?

- i. `cout<<secret(5)<<endl;`
- ii. `cout<<2 * secret(6)<<endl;`

b. 分析 `secret()` 函数的功能。

9. 给出下列程序的输出。

```
#include <iostream>
using namespace std;
int mystry(int);
int main()
{
    int n;

    for(n = 1; n <= 5; n++)
        cout<<mystry(n)<<endl;
    return 0;
}

int mystry(int k)
{
    int x, y;

    y = k;
    for(x = 1; x <= (k - 1); x++)
        y = y * (k - x);
    return y;
}
```

10. 给出下列程序的输出。

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

bool strange(int);

int main()
{
    int num = 0;
```



```

    while(num <= 29)
    {
        if(strange(num))
            cout<<"True"<<endl;
        else
            cout<<"False"<<endl;
        num = num + 4;
    }
    return 0;
}
bool strange(int n)
{
    if(n % 2 == 0 && n % 3 == 0)
        return true;
    else
        return false;
}

```

6.8 编程练习

1. 使用例子 6.2 中给出的函数 `isNumPalindrome()` 编写程序 (回文数)。用下列数字测试程序: 10, 34, 22, 333, 678, 67876, 44444 和 123454321。
2. 编写一个具有返回值的函数 `isVowel()`, 如果给定的字符是一个元音, 函数返回 `true`, 否则返回 `false`。
3. 编写一个程序, 它提示用户输入一串字符并输出元音的数量。(使用在编程练习 2 中的函数 `isVowel()`。)
4. 考虑下列程序:

```

#include <iostream>
using namespace std;

int one(int x, int y);
double two(int x, double a);

int main()
{
    int num;
    double dec;
    .
    .
    .
    return 0;
}

int one(int x, int y)
{
    .
    .
    .
}

double two(int x, double a);
{
    int first;
    double z;
    .
    .
    .
}

```

- a. 编写函数 `one()` 的定义, 在 x 大于 y 时返回 x 和 y 的和, 否则返回 x 减去 2 倍的 y 。
- b. 编写函数 `two()` 的定义:
 - i. 读入一个数并保存在 z 中。
 - ii. 通过将 z 的值加上 a 的值更新 z 的值。
 - iii. 将带有参数 6 和 8 的函数 `one()` 返回的值赋给变量 `first`。
 - iv. 通过将 `first` 的值加上 x 更新 `first` 的值。
 - v. 如果 z 值大于 `first` 值的 2 倍, 返回 z ; 否则, 返回 2 倍的 `first` 减去 z 。
- c. 编写一个 C++ 程序测试 a 和 b 中的函数 (如果需要的话, 在函数 `main()` 中声明额外的变量)。
5. 编写函数 `reverseDigit()`, 将整数作为一个参数并返回数字反转后的结果。例如, `reverseDigit(12345)` 的值是 54321。
6. 下列公式给出了笛卡尔平面上两点 (x_1, y_1) 和 (x_2, y_2) 之间的距离:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

给定圆心和圆上一点, 可以使用该公式得出圆的半径。编写一个程序提示用户输入圆心和圆上一点。然后程序输出圆的半径、直径、圆周和面积。程序必须至少有下列函数:

- a. `distance()`: 该函数以平面上两个点的坐标作为参数, 并返回它们之间的距离。
 - b. `radius()`: 该函数以圆心和圆上一点的坐标作为参数, 调用函数 `distance()` 返回圆的半径。
 - c. `circumference()`: 该函数以圆的半径作为参数并返回圆的周长。(如果 r 是半径, 圆周是 $2\pi r$ 。)
 - d. `area()`: 该函数以圆的半径作为参数并返回圆的面积。(如果 r 是半径, 面积是 πr^2 。)
- 假定 $\pi=3.1416$ 。
7. 如果 P 是一年中第一天的人口数量, B 是出生率, D 是死亡率, 年底的人口普查统计值通过下列公式给出:

$$P + \frac{B * P}{100} - \frac{D * P}{100}$$

人口增长率通过下列公式给定:

$$B - D$$

编写一个程序提示用户输入初始人口、出生率、死亡率和年数 n 。程序计算并打印 n 年后的人口统计数量。程序必须由下列函数组成:

- a. `growthRate()`: 该函数用出生率和死亡率作为参数并返回人口增长率。
 - b. `estimatedPopulation()`: 该函数用当前人口、人口增长率和年数 n 作为参数, 返回 n 年后的估算人口。
- 程序不应该接收负的出生率、负的死亡率和小于 2 的人口数量。
8. 重写第 4 章编程练习 12 中的程序 (蜂窝电话公司), 可以使用下面的函数来计算账单总额。(在该编程练习中, 不输出服务时间。)
 - a. `regularBill()`: 该函数为常规服务计算并返回账单总额。
 - b. `premiumBill()`: 该函数为增值服务计算并返回账单总额。

第7章 用户自定义函数 II

本章要点:

- 学习怎样在程序中构造并使用 void 函数
- 了解值参和引用参数的区别
- 研究引用参数和返回值函数
- 学习标识符的作用域
- 研究局部和全局变量的区别
- 研究静态变量
- 学习函数重载
- 研究具有默认参数的函数

在第6章中,我们学习了如何使用有返回值的函数。在本章中,将会探讨一般意义下的用户自定义函数,特别是那些没有数据类型的、称为 void 函数的 C++ 函数。

7.1 void 函数

void 函数和有返回值的函数具有相同的结构,它们都包含头部分和语句部分。可以把用户自定义函数置于 main() 函数前,也可以放在 main() 函数后。然而,程序总是从 main() 函数的第一条语句开始执行的。如果把用户自定义的 void 函数放在 main() 函数后,那么应该将函数原型放在 main() 函数前。因为 void 函数没有数据类型,所以头部分中的 functionType 和 void 函数体中的 return 语句是没有意义的。然而,在 void 函数中,可以使用没有任何值的 return 语句,它一般用于提前退出函数。像返回值函数一样, void 函数可以有、也可以没有形式参数。

因为 void 函数没有数据类型,所以它们在一个表达式中是不可用的(即调用)。对 void 函数的调用是一条独立的语句。这样,为了调用一个 void 函数,可以在独立的语句中使用函数名和实参(如果有)。

7.1.1 不带参数的 void 函数

该部分讨论没有参数的 void 函数。

函数定义

无参 void 函数的通常形式(语法)如下:

```
void functionName(void)
{
    statements
}
```

括号中的 void 是可选的。在 C++ 中, void 是一个保留字。

注意：像变量名一样，函数名字应该起到描述作用，所以当命名一个函数时最好使用一个有意义的名字。

函数调用

函数调用的语法如下所示：

```
functionName();
```

因为这些函数没有参数，所以没有值传递给它们。（除非使用全局变量，将在本章后面给出其定义。）函数能够很好地用于显示程序的信息或打印语句。

例 7.1

假定想打印下面的标题来显示每年春季销售额。

```
*****
*****
***** Annual *****
*****
*****
***** Spring Sale *****
*****
*****
```

标题以两行星号开始。在打印包含文本 Annual 的行之后，需要打印另外两行的星号。在打印包含了文本 Spring Sale 的行之后，需要再打印另外两行的星号。可以编写一个打印两行星号的函数，并且在需要的时候进行调用。完整的程序如下所示：

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

void printStars(void);

int main()
{
    printStars(); //Line 1
    cout<<"***** Annual *****"<<endl; //Line 2
    printStars(); //Line 3
    cout<<"***** Spring Sale *****"<<endl; //Line 4
    printStars(); //Line 5
    return 0;
}

void printStars(void)
{
    cout<<"*****"<<endl;
    cout<<"*****"<<endl;
}
```

输出：

```
*****
*****
***** Annual *****
*****
*****
***** Spring Sale *****
*****
*****
```

在第1行中, 函数 `printStars()` 被调用并且输出头两行星号。第2行语句输出包含文本 `Annual` 的星号行。在第3行中, 函数 `printStars()` 又一次被调用并且输出下两行星号。然后第4行的语句输出包含文本 `Spring Sale` 的星号行。在第5行语句中, 函数 `printStars()` 再一次被调用并且输出最后两行星号。

注意: 在 `main()` 函数中的语句 `printStars()` 是函数调用。

在前面的程序中, 可以用下面的函数来替换 `printStars()` 函数:

```
void printStars(void)
{
    int stars, lines;
    for(lines = 1; lines <= 2; lines++)      //Line 1
    {
        for(stars = 1; stars <= 30; stars++)  //Line 2
            cout<<"* ";                      //Line 3
        cout<<endl;                          //Line 4
    }
}
```

函数定义中, 外部的 `for` 循环执行两次。对于每一个外部 `for` 循环 (第1行), 内部的 `for` 循环 (第2行) 会执行30次, 每次在一行打印30个星号。第3行的语句打印一个星号。第4行的 `cout` 语句将光标定位在标准输出设备下一行的开始。因为外部 `for` 循环执行两次, 所以函数输出两行, 每行包含30个星号。

使用 `for` 循环输出两行、每行有30个星号的 `printStars()` 函数定义, 比前面给出的 `printStars()` 定义更容易修改。例如, 如果需要输出5行星号而不是两行星号, 那么只需要将数字2 (第1行的第一个 `for` 循环) 替换为数字5。同样, 如果每行需要输出40个星号而不是30个星号, 那么只需要将数字30 (第2行的第二个 `for` 循环) 替换为40。而且, 为了与调用函数建立通信并且在每次 `printStars()` 函数调用时能做不同的事情, 使用 `for` 循环的 `printStars()` 函数定义更容易修改。

前面的程序中, 函数 `printStars()` 总是打印两行星号, 其中每一行有30个星号。现在假定想打印下面的图形 (三角形的星号):

```

      *
     * *
    * * *
   * * * *
  * * * * *
```

可以编写一个类似于 `printStars()` 的函数。然而, 如果需要将这种图形扩展为20行, 则 `printStars()` 函数将会有20行。每次调用函数 `printStars()` 时, 它打印相同数量的行, 函数 `printStars()` 的灵活性较差。然而, 如果以一种方式告诉函数 `printStars()` 去打印多少行, 就可以大大地提高其灵活性。在调用函数和被调用函数之间必须存在一种通信连接。

7.1.2 带参数的 void 函数

前面的部分讨论了没有参数的 `void` 函数, 并且指出了这类函数的局限性: 信息无法从无参的 `void` 函数中传入或传出。调用函数和被调用函数之间的通信是通过使用参数来建立的。该部分讨论有参数的 `void` 函数。

函数定义

有参数的 `void` 函数定义有下面的语法:

```
void functionName(formal parameter list)
{
    statements
}
```

形式参数列表

形式参数列表有下面的语法:

```
dataType& variable, dataType& variable, ...
```

在形式参数列表中必须指定数据类型和变量名字。数据类型后的符号&有特殊意义,它只用于特定的形式参数,这将在本章后面进行讨论。

函数调用

函数调用有下面的形式:

```
functionName(actual parameter list);
```

实参列表

实参列表有下面的语法:

```
expression or variable, expression or variable, ...
```

对于返回值函数,在函数调用过程中,实参的数量与数据类型必须与形参相匹配,实参和形参有一一对应的关系。函数调用会导致被调用函数体的执行(再本章末尾将会讨论有默认参数的函数)。

例 7.2

```
void funexp(int a, double b, char c, int& x)
{
    .
    .
    .
}
```

函数funexp()包含4个参数。

参数在调用函数(例如main())和被调用函数之间提供了一种通信。每次调用时,它们使函数能对不同的数据进行操作。通常有两种形式参数:值(value)参数和引用(reference)参数。

值参数:接收相应实参内容副本的形式参数。

引用参数:接收相应实参位置(内存地址)的形式参数。

在形参列表的dataType后面加上&时,dataType后面的变量就变成了一个引用参数。

例 7.3

```
void expfun(int one, int & two, char three, double & four);
```

函数expfun()有4个参数:(1)one,int类型的值参数;(2)two,int类型的引用参数;(3)three,char类型的值参数;(4)four,double类型的引用参数。

在学习有参数的void函数之前,先来考察一下值参数和引用参数。从值参数的定义中可以看出,如果形参是值参数,那么相应实参值会复制到形参中。即值参数有自己数据的副本。所以,在程序执行期间,值参数对存储在自己内存空间中的数据进行操作。在复制数据之后,值参数与实际参数就没有联系了。

另一方面,如果形参是一个引用参数,它会接收相应实参的地址。即引用参数存储相应实参的地址。程序对数据进行操作期间,存储在引用参数中的地址将它指向相应实参内存空间。换言之,

程序执行期间, 引用参数对存储在相应实参内存空间中的数据进行操作。引用参数对数据进行的任何修改都会改变相应实参的值。例 7.5 和例 7.6 明确地显示了值参数和引用参数在程序执行期间是如何工作的。注意第 6 章中的例子只使用值参数。

现在编写 C++ 程序来输出前面给出的图案 (三角形星号)。

例 7.4

下面的 C++ 程序输出三角形的图案:

```
//Program: Print a triangle of stars
#include <iostream>
using namespace std;

void printStars(int blanks, int starsInLine);

int main()
{
    int numberOfLines;
    int counter;
    int numberOfBlanks;

    cout<<"Enter the number of star lines (1 to 20)"
        <<" to be printed-> "; //Line 1
    cin>>numberOfLines; //Line 2
    while(numberOfLines <=0 || numberOfLines > 20) //Line 3
    {
        cout<<"Number of star lines should be "
            <<"between 1 and 20"<<endl; //Line 4
        cout<<"Enter the number of star lines "
            <<"(1 to 20) to be printed-> "; //Line 5
        cin>>numberOfLines; //Line 6
    }

    cout<<endl<<endl; //Line 7
    numberOfBlanks = 30; //Line 8

    for(counter = 1; counter <= numberOfLines; counter++) //Line 9
    {
        printStars(numberOfBlanks, counter); //Line 10
        numberOfBlanks--; //Line 11
    }

    return 0; //Line 12
}

void printStars(int blanks, int starsInLine)
{
    int count;

    for(count = 1; count <= blanks; count++) //Line 13
        cout<<" "; //Line 14

    for(count = 1; count <= starsInLine; count++) //Line 15
        cout<<" *"; //Line 16

    cout<<endl;
}
```

运行实例: 在本运行实例中, 用户输入用阴影表示。

Enter the number of star lines(1 to 20)to be printed-> ■■■

```

      *
     **
    ***
   ****
  *****
 *****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****

```

在程序中, `main()` 函数中的语句 “`printStars(numberOfBlanks, counter);`” 是函数调用。变量 `numberOfBlanks` 和 `counter` 是实参。

函数 `printStars()` 运行原理如下。函数 `printStars()` 有两个参数。无论 `printStars()` 函数什么时候执行, 它在每行星号前都会输出有特定数量的空格。空格的数量和星号的数量作为参数传递给函数 `printStars()`。第一个参数 `blanks` 表明打印多少空格, 第二个参数 `starsInLine` 表明在一行要打印多少个星号。例如, 如果参数 `blank` 的值是 30, 那么函数 `printStars()` 中第一个 `for` 循环 (第 13 行) 执行 30 次, 打印 30 个空格。而且因为需要在星号间打印空格, 所以函数 `printStars()` 第二个 `for` 循环的每一次执行 (第 15 行) 都会打印 ‘*’, 即在空格之后跟着一个星号。

在 `main()` 函数中, 用户首先被要求指定要打印多少行 (第 1 行)。(在程序中, 用户被限制在 20 行之内, 这是因为三角形在屏幕上正好能放 20 行)。第 3~6 行的 `while` 循环保证程序输出只能在 1 到 20 行之间。

在 `main()` 函数中的 `for` 循环 (第 9 行) 调用函数 `printStars()` (第 10 行)。该 `for` 循环的每一次执行使用 `numberOfBlanks` 和 `counter` 指定一行中星号前面的空格数量。`printStars()` 函数的每一次调用都会比上次调用少一个空格而多一个星号。例如, `main()` 函数中 `for` 循环第一次执行指定 30 个空格和 1 个星号 (`numberOfBlanks` 和 `counter` 作为参数传递给 `printStars()` 函数)。然后 `for` 循环:

- 空格的数量每次减 1。
`numberOfBlanks--;`
- 在 `for` 循环的末尾, 星号的数量每次加 1。这由更新语句 “`counter++;`” (第 9 行) 来执行。在 `for` 语句中, 它使变量 `counter` 的值加 1。

换言之, 函数 `printStars()` 的第二次调用接收 29 个空格和 2 个星号作为参数。

7.1.3 引用参数

在复制数据之后, 值参数与实参再也没有联系, 所以值参数不能把任何结果传回调用函数。当函数执行时, 形参的任何改变都不会影响实参。实参不知道形参将会发生什么。这样, 在函数外部, 值参数不能传递信息, 值参数只在实参和形参之间提供了一种连接。所以, 使用值参数的函数具有局限性。另一方面, 由于引用参数接收地址 (即实参的内存位置), 所以引用参数能从函数中传递一个或更多的值, 并且能改变实参的值。

引用参数在下面三种情况下是有用的:

- 从函数中返回不止一个值时

- 当实参的值需要改变时
- 当传递地址能节省内存空间和大量相关数据的复制时间时

当介绍数组和类时，会讨论第三种情况。

前面讲过，当在函数形参列表中的 dataType 后面附加 & 时，dataType 后面的变量就成为引用参数。

注意：（常量引用参数）通过使用关键字 const 可以声明一个常量引用（形式）参数。第 13 章讨论了常量引用参数。直到那时，所使用的引用参数将会是本章所定义的非常量。从引用参数的定义中可以看出，常量值或表达式不能传送给一个非常量的引用参数。所以如果一个形式参数是一个非常量引用参数，那么在程序调用期间它相应的实参必须是一个变量。

例 7.5

考虑下面的程序。给定一个课程分数（值在 0 和 100 之间），它计算学生的评分等级。程序包含三个函数：

main(), getScore() 和 printGrade()。这三个函数描述如下：

1. main()
 - a. 得到课程分数
 - b. 打印评分等级
2. getScore()
 - a. 提示用户输入课程分数
 - b. 得到输入分数
 - c. 打印课程分数
3. printGrade()
 - a. 计算评分等级
 - b. 打印评分等级

完整程序如下：

```
//Program: Compute grade.
//This program reads a course score and prints the
//associated course grade.

#include <iostream>
using namespace std;
void getScore(int& score);
void printGrade(int score);

int main ()
{
    int courseScore;

    cout<<"Line 1: Based on the course score, this program "
        <<"computes the course grade."<<endl;           //Line 1

    getScore(courseScore);                               //Line 2
    printGrade(courseScore);                             //Line 3
    return 0;
}

void getScore(int& score)
{
    cout<<"Line 4: Enter course score -> ";           //Line 4
```

```
cin>>score; //Line 5
cout<<endl<<"Line 6: Course score is "
    <<score<<endl; //Line 6
}

void printGrade(int score)
{
    cout<<"Line 7: Your grade for the course is "; //Line 7
    if(score >= 90) //Line 8
        cout<<"A"<<endl;
    else if(score >= 80)
        cout<<"B"<<endl;
    else if(score >= 70)
        cout<<"C"<<endl;
    else if(score >= 60)
        cout<<"D"<<endl;
    else
        cout<<"F"<<endl;
}
```

运行实例：在本运行实例中，用户输入用阴影表示。

Line 1: Based on the course score, this program computes the course grade.

Line 4: Enter course score→ 85

Line 6: Course score is 85

Line 7: Your grade for the course is B

该程序执行原理如下。程序在第1行开始执行，会打印第1行的输出（参看运行实例）。第2行语句调用包含实参 `courseScore`（在 `main()` 函数中声明的变量）的 `getScore()` 函数。因为函数 `getScore()` 的形参 `score` 是一个引用参数，所以地址（即变量 `courseScore` 的内存位置）传递给 `score`。这样，`score` 和 `courseScore` 都指向相同的内存位置，即 `courseScore`。`score` 的任何改变都会改变 `courseScore` 的值。

控制然后转向函数 `getScore()` 中的第4行语句，打印第2行输出（参看运行实例）。该语句提示用户输入课程分数。第5行语句读入值并向 `score` 存储用户输入的值（85），实际上是 `courseScore`（因为 `score` 是一个引用变量）。因此，变量 `score` 和 `courseScore` 的值是85。第6行语句输出 `score` 的值，如运行实例第3行所显示。在第6行执行之后，控制返回函数 `main()`。

接着执行第3行语句。它是对具有实参 `courseScore` 的 `printGrade()` 函数的调用。因为函数 `printGrade()` 的形参 `score` 是一个值参数，所以参数接收相应实参 `courseScore` 的值。这样，`score` 的值是85。在将 `courseScore` 的值复制到 `score` 后，`score` 和 `courseScore` 之间就没有任何联系了。现在程序执行第7行语句，它输出第4行。第8行的 `if...else` 语句输出了相应分数的级别。因为第7行的输出语句没有包含换行符，所以 `if...else` 语句的输出是第4行输出的一部分（参看运行实例）。在 `if...else` 语句执行之后，控制又重新回到 `main()` 函数。因为在 `main()` 函数中要执行的下一条语句是 `main()` 函数的最后一条语句，所以程序终止。

程序中，`main()` 函数第一次调用 `getScore()` 函数从用户得到课程分数。`main()` 函数然后调用函数 `printGrade()` 计算和打印基于课程分数的级别。课程分数然后被 `getScore()` 重新得到，该课程分数在后面被函数 `printGrade()` 使用。因为 `getScore()` 函数重新得到的值在后面被程序使用，所以函数 `getScore()` 必须把该值传送到函数外。这样，保存该值的形参必须是一个引用参数。

下面是值参数和引用参数特性的总结。

7.2 值参数、引用参数和内存分配

函数调用时,函数体中声明的变量(局部变量)和形参的内存分配作用于函数数据区。前面讲过,对于值参数,实参的值复制到相应的形参内存单元。对于引用参数,实参的地址被传递到形参,即形参的内容是地址。在数据操作期间,形参的内容指示计算机来操作其内存单元中的数据。这样,对于引用参数,实参和形参都指向同一个内存位置。因此,程序执行期间,形参的改变将会改变实参的值。

注意: 数据流变量(例如 `ifstream` 和 `ofstream`)应该作为引用参数传递给函数。在打开输入/输出文件后,或读/写数据之后,数据流的状态可以传递到函数外。

因为参数传递对于任何一种编程语言都是很基本的,所以例 7.6 到例 7.8 进一步演示了这一概念。

例 7.6

下面的程序显示了值参数和引用参数是如何工作的。

```
//Example 7-6: Reference and value parameters

#include <iostream>
using namespace std;

void funOne(int a, int& b, char v);
void funTwo(int& x, int y, char& w);

int main()
{
    int num1, num2;
    char ch;

    num1 = 10; //Line 1
    num2 = 15; //Line 2
    ch = 'A'; //Line 3

    cout<<"Line 4: Inside main: num1 = "<<num1
         <<" , num2 = "<<num2<<" , and ch = "<<ch<<endl; //Line 4

    funOne(num1, num2, ch); //Line 5
    cout<<"Line 6: After funOne: num1 = "<<num1
         <<" , num2 = "<<num2<<" , and ch = "<<ch<<endl; //Line 6

    funTwo(num2, 25, ch); //Line 7

    cout<<"Line 8: After funTwo: num1 = "<<num1
         <<" , num2 = "<<num2<<" , and ch = "<<ch<<endl; //Line 8

    return 0;
}

void funOne(int a, int& b, char v)
{
    int one;

    one = a; //Line 9
    a++; //Line 10
```

```

    b = b * 2;                                     //Line 11
    v = 'B';                                       //Line 12

    cout<<"Line 13: Inside funOne: a = "<<a<<" , b = "<<b
        <<" , v = "<<v<<" , and one = "<<one<<endl;    //Line 13
}

void funTwo(int& x, int y, char& w)
{
    x++;                                           //Line 14
    y = y * 2;                                     //Line 15
    w = 'G';                                       //Line 16
    cout<<"Line 17: Inside funTwo: x = "<<x
        <<" , y = "<<y<<" , and w = "<<w<<endl;    //Line 17
}

```

输出:

```

Line 4: Inside main: num1 = 10, num2 = 15, and ch = A
Line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10
Line 6: After funOne: num1 = 10, num2 = 30, and ch = A
Line 17: Inside funTwo: x = 31, y = 50, and w = G
Line 8: After funTwo: num1 = 10, num2 = 31, and ch = G

```

在第1行语句执行前,内存只分配了main()函数中相应变量的内存空间,这些内存并没有被初始化。在第3行语句执行后,变量值如图7.1所示。

第4行语句产生下面的输出:

```

Line 4: Inside main: num1 = 10, num2 = 15,
and ch = A

```

第5行语句是对函数funOne()的调用。现在函数funOne()有三个参数和一个局部变量。同时,参数和局部变量的内存也被分配。因为形参b是引用参数,它接收相应实参的地址(内存位置),即num2。其他两个形参是值参数,所以它们复制相应实参的值。在第9行语句执行前,变量如图7.2所示。

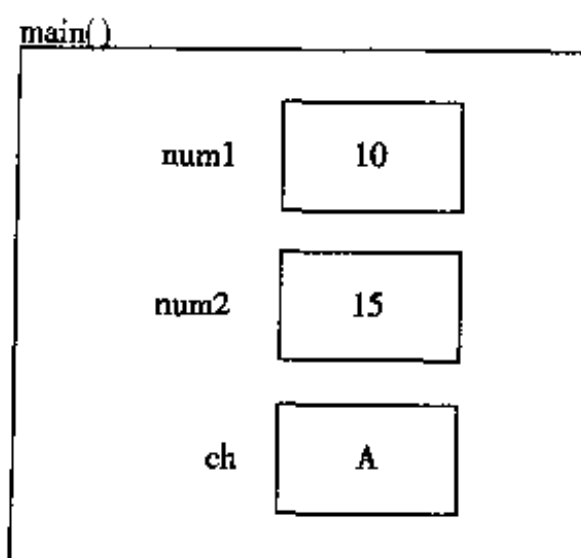


图7.1 第3行语句执行后的变量值

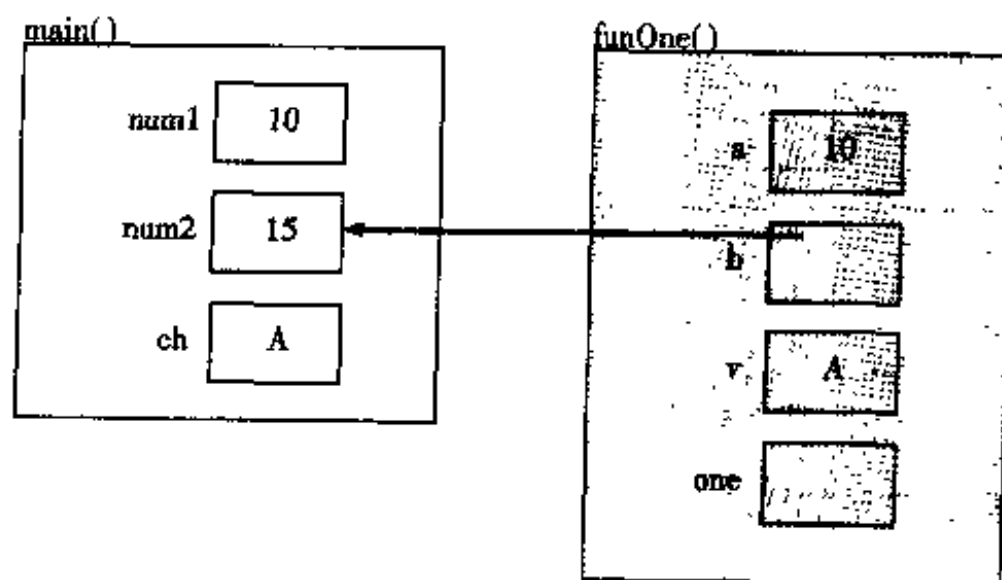


图7.2 第9行语句执行前的变量值

第9行语句执行后,变量如图7.3所示。

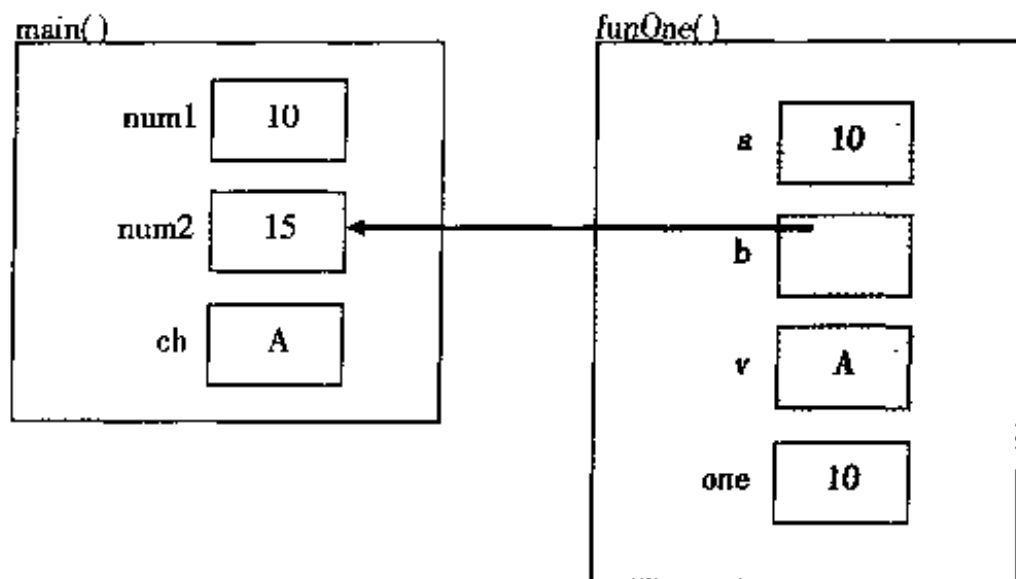


图 7.3 第 9 行语句执行后的变量值

第 10 行语句执行后，变量如图 7.4 所示。

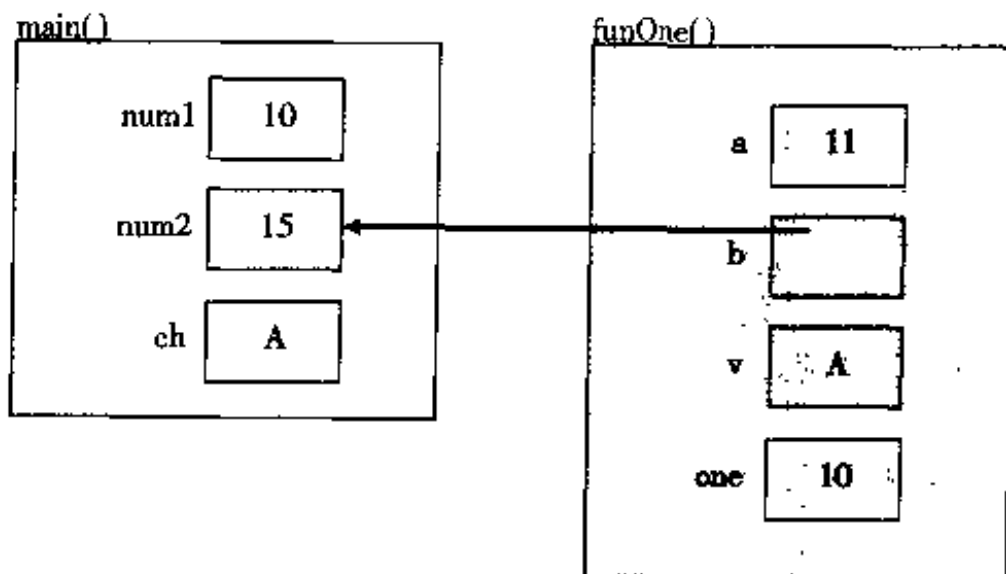


图 7.4 第 10 行语句执行后的变量值

第 11 行语句执行后，变量如图 7.5 所示。（注意变量 `b` 改变了 `num2` 的值。）

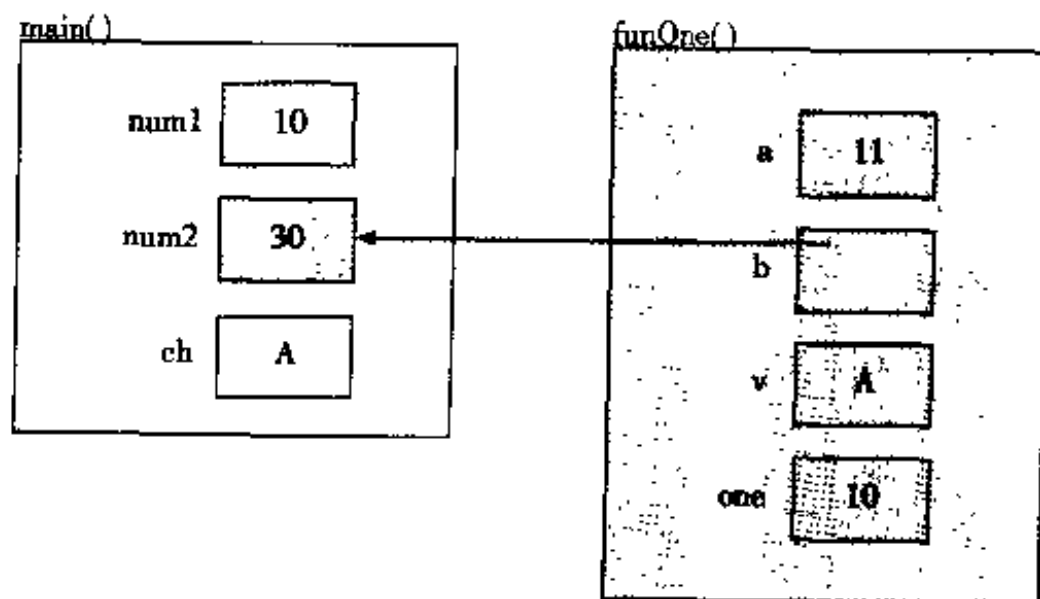


图 7.5 第 11 行语句执行后的变量值

第 12 行语句执行后，变量如图 7.6 所示。

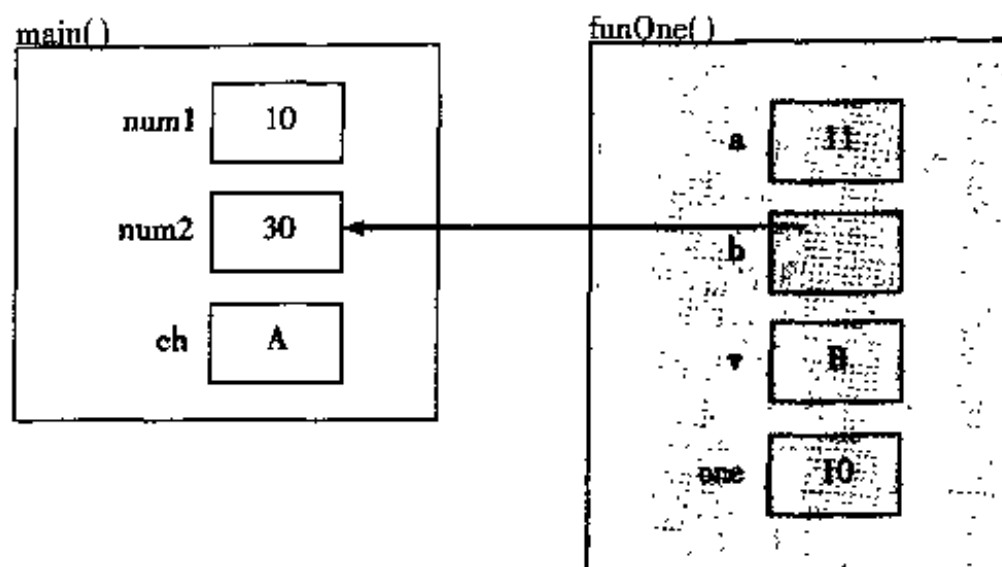


图 7.6 第 12 行语句执行后的变量值

第 13 行语句产生下面的输出:

Line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10

第 13 行语句执行后, 控制重新回到第 6 行, 而且分配给函数 funOne() 的变量其内存被释放。图 7.7 显示了 main() 函数的变量值。

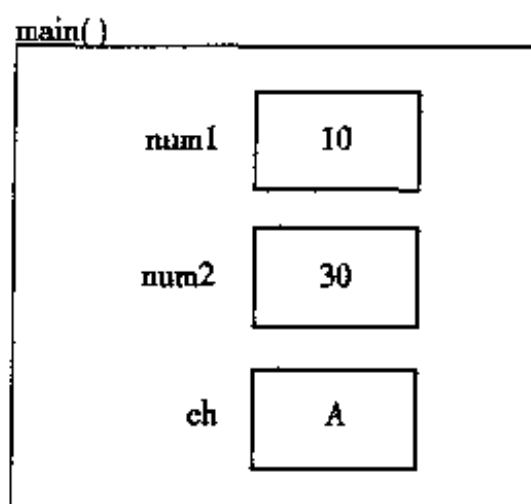


图 7.7 当控制回到第 6 行时变量的值

第 6 行语句产生下面的输出:

Line 6: After funOne: num1 = 10, num2 = 30, and ch = A

第 7 行语句是对函数 funTwo() 的调用。现在 funTwo() 有三个参数: x, y 和 w。而且 x 和 w 是引用参数, y 是值参数。这样, x 接收相应实参的地址, 即 num2, w 接收相应实参的地址, 即 ch。变量 y 将值 25 复制到内存单元。如图 7.8 所示。

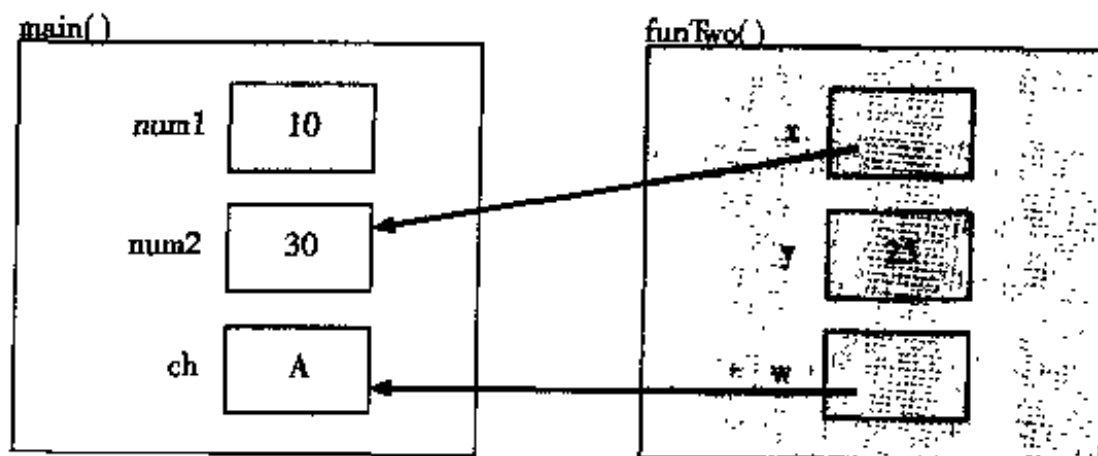


图 7.8 第 14 行语句执行前的变量

第 14 行语句执行后, 变量如图 7.9 所示。(注意, 变量 x 改变了 num2 的值。)

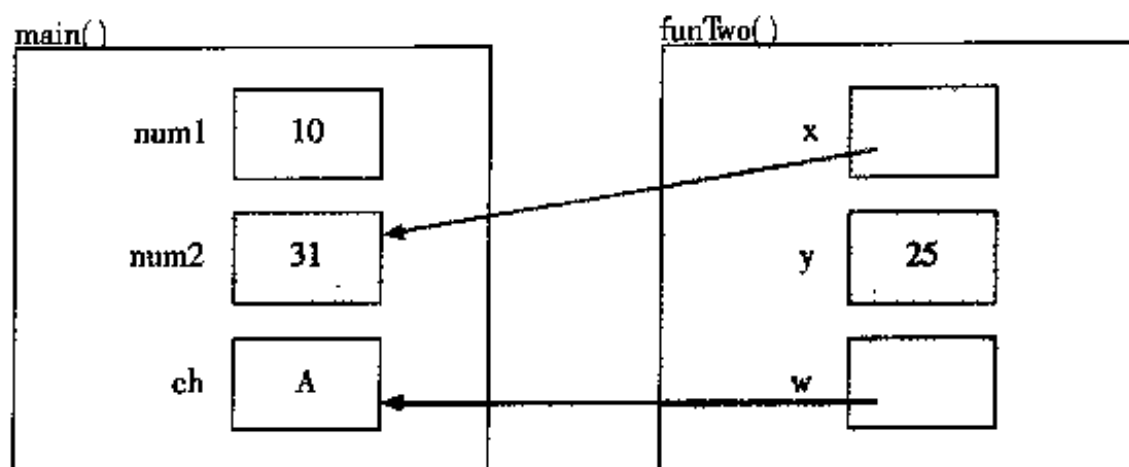


图 7.9 第 14 行语句执行后变量的值

在第 15 行语句执行后，变量如图 7.10 所示。

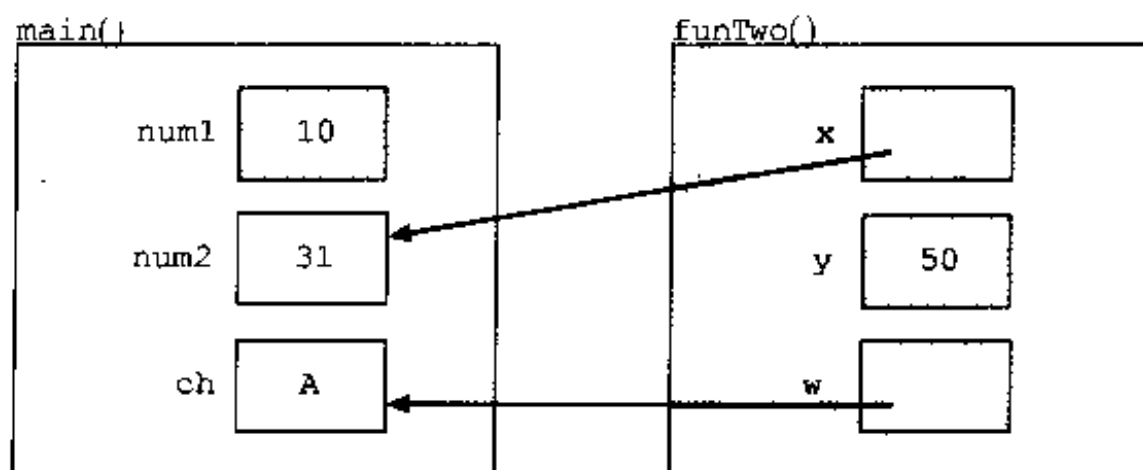


图 7.10 第 15 行语句执行后的变量值

在第 16 行语句执行后，变量如图 7.11 所示。（注意，变量 `w` 改变了 `ch` 的值。）

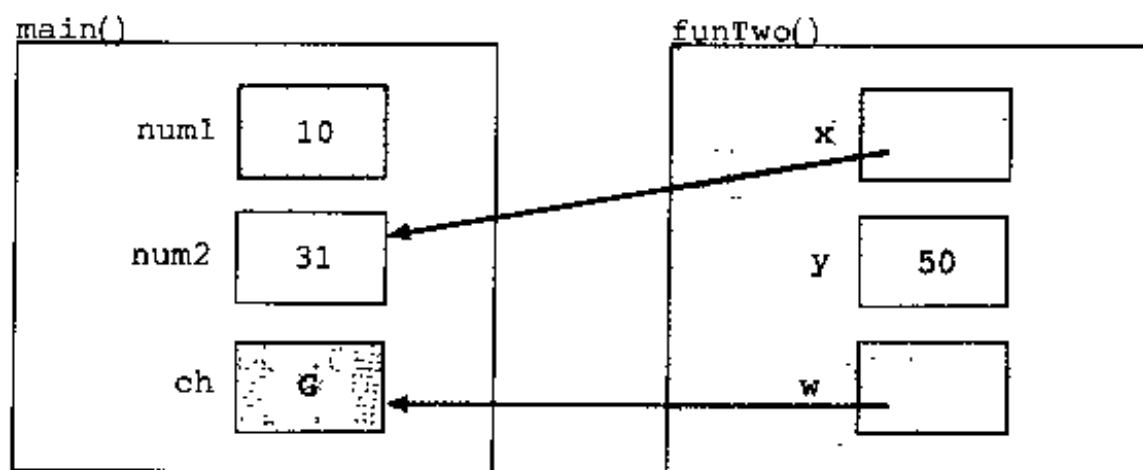


图 7.11 第 16 行语句执行后的变量值

第 17 行语句产生下面的输出：

Line 17: Inside funTwo: `x = 31`, `y = 50`, and `w = G`

第 17 行语句执行后，控制重新回到第 8 行。分配给函数 `funTwo()` 的变量其内存被释放。图 7.12 显示了 `main()` 函数中的变量值。

第 8 行语句产生下面的输出：

Line 8: After funTwo: `num1 = 10`, `num2 = 31`, and `ch = G`

第 8 行语句执行后，程序终止。

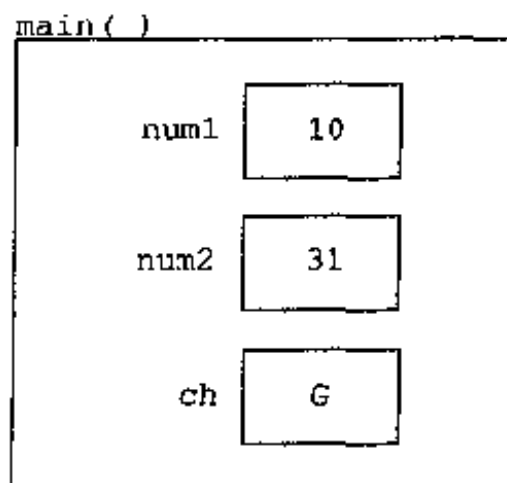


图 7.12 当控制回到第6行语句时变量的值

例 7.7

例子给出了引用参数是如何控制实参的。

```

//Example 7-7: Reference and value parameters
//Program: Makes You Think.

#include <iostream>
using namespace std;

void addFirst(int& first, int& second);
void doubleFirst(int one, int two);
void squareFirst(int& ref, int val);

int main ()
{
    int num = 5;
    cout<<"Line 1: Inside main:  num = "<<num<<endl;           //Line 1
    addFirst(num,num);                                           //Line 2
    cout<<"Line 3: Inside main after addFirst:"
        <<"  num = " <<num<<endl;                               //Line 3
    doubleFirst(num,num);                                         //Line 4
    cout<<"Line 5: Inside main after "
        <<"doubleFirst:  num = "<<num<<endl;                     //Line 5
    squareFirst(num,num);                                         //Line 6
    cout<<"Line 7: Inside main after "
        <<"squareFirst:  num = "<<num<<endl;                     //Line 7
    return 0;
}

void addFirst(int& first, int& second)
{
    cout<<"Line 8: Inside addFirst:  first = "<<first
        <<" , second = "<<second<<endl;                         //Line 8
    first = first + 2;                                           //Line 9
    cout<<"Line 10: Inside addFirst:  first = "<<first
        <<" , second = "<<second<<endl;                         //Line 10
    second = second * 2;                                         //Line 11
    cout<<"Line 12: Inside addFirst:  first = "<<first
        <<" , second = "<<second<<endl;                         //Line 12
}
  
```



```

void doubleFirst(int one, int two)
{
    cout<<"Line 13: Inside doubleFirst:  one = "<<one
        <<"", two = "<<two<<endl;           //Line 13
    one = one * 2;                           //Line 14
    cout<<"Line 15: Inside doubleFirst:  one = "<<one
        <<"", two = "<<two<<endl;           //Line 15
    two = two + 2;                           //Line 16
    cout<<"Line 17: Inside doubleFirst:  one = "<<one
        <<"", two = "<<two<<endl;           //Line 17
}

void squareFirst(int& ref, int val)
{
    cout<<"Line 18: Inside squareFirst: ref = "
        <<ref <<"", val = "<< val<<endl;   //Line 18
    ref = ref * ref;                         //Line 19
    cout<<"Line 20: Inside squareFirst: ref = "
        <<ref <<"", val = "<< val<<endl;   //Line 20
    val = val + 2;                           //Line 21
    cout<<"Line 22: Inside squareFirst: ref = "
        <<ref <<"", val = "<< val<<endl;   //Line 22
}

```

输出:

```

Line 1: Inside main:  num = 5
Line 8: Inside addFirst:  first = 5, second = 5
Line 10: Inside addFirst:  first = 7, second = 7
Line 12: Inside addFirst:  first = 14, second = 14
Line 3: Inside main after addFirst:  num = 14
Line 13: Inside doubleFirst:  one = 14, two = 14
Line 15: Inside doubleFirst:  one = 28, two = 14
Line 17: Inside doubleFirst:  one = 28, two = 16
Line 5: Inside main after doubleFirst:  num = 14
Line 18: Inside squareFirst: ref = 14, val = 14
Line 20: Inside squareFirst: ref = 196, val = 14
Line 22: Inside squareFirst: ref = 196, val = 16
Line 7: Inside main after squareFirst:  num = 196

```

函数 `addFirst()` 的两个参数都是引用参数, 函数 `doubleFirst()` 的两个参数都是值参。 `main()` 函数中, 语句 “`addFirst(num,num);`” 将 `num` 分别传递给了其形参 `first` 和 `second`。变量 `first` 和 `second` 指向同一个内存位置, 即 `num`。图 7.13 演示了这种情况。

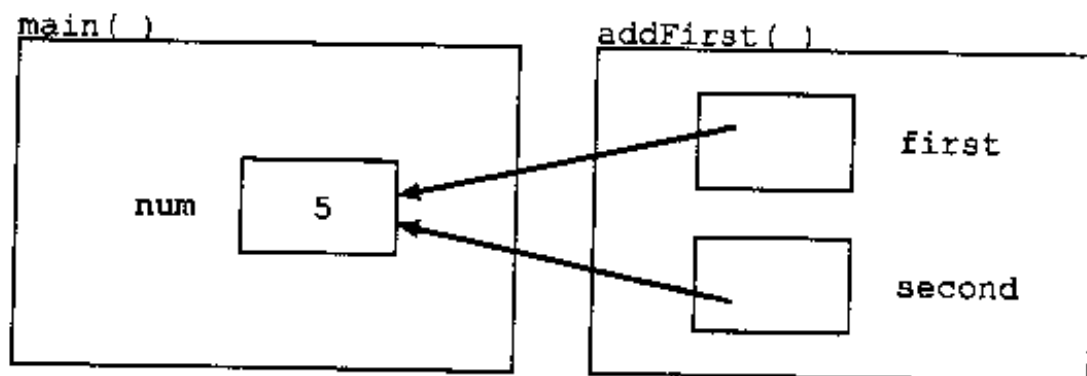


图 7.13 函数 `addFirst()` 的参数

first对其值的任何修改会改变second和num的值。同样,second对其值的任何修改也会因此改变first和num的值,因为所有三个变量都指向同一个内存位置。(注意num初始化值为5。)

函数doubleFirst()的形参是值参数。所以main()第4行)函数中的语句“doubleFirst(num,num);”将num的值复制到one和two中,因为这两个形参相应的实参是相同的。图7.14显示了这种情况。

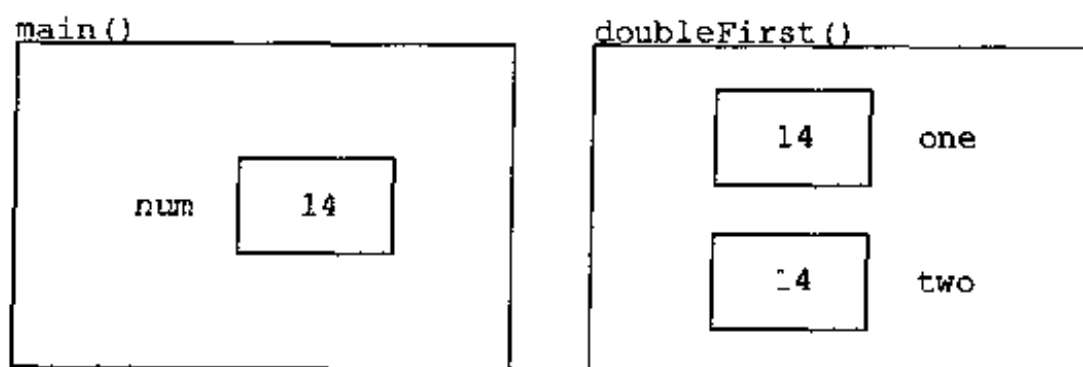


图7.14 doubleFirst()函数的参数

因为one和two都是值参,所以one对其值的任何修改都不会影响two和num的值。同样two对其值的任何修改也都不会影响one和num的值。(注意,num的值在函数doubleFirst()执行前为14。)

函数squareFirst()中的形参ref是引用参数,形参val是一个值参数。变量ref接收相应实参的地址,即num,变量val复制相应实参的值,也就是num。这样,num和ref都指向同一个内存位置,即num。图7.15显示了这种情况。

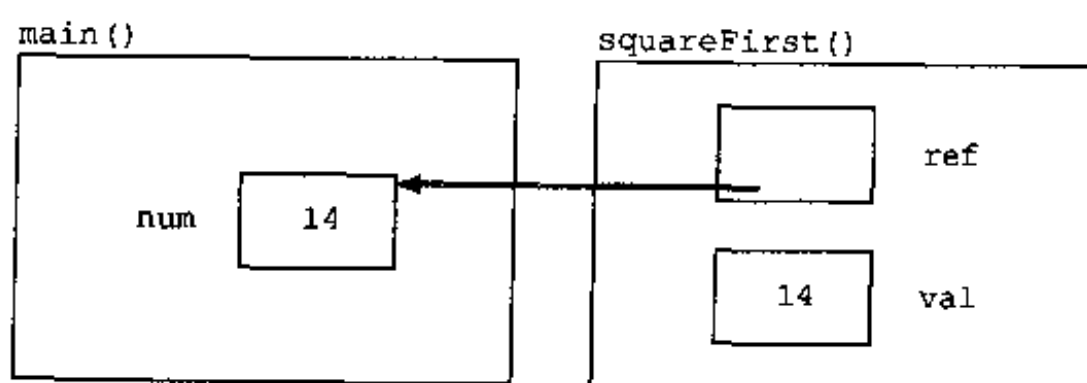


图7.15 函数squareFirst()的参数

ref的任何改变都会立即改变num的值,val的改变不会影响num。(注意,num的值在函数squareFirst()执行前是14。)

例7.8

该例显示了在任何一个函数(或块)外,声明的变量是如何工作的。也显示了当用户自定义函数的引用参数向另一个用户自定义函数的引用参数传递值时,引用参数如何影响实参。

```
//Example 7-8: Reference and value parameters
#include <iostream>
using namespace std;

int t;

void funOne(int& a, int& x);
void funTwo(int& u, int v);

int main()
{
    int num1, num2;

    num1 = 10;
    num2 = 20;
    t = 15;
}
```

//Line 1
//Line 2
//Line 3

```

    cout<<"Line 4: In main: num1 = "<<num1<<" , num2 = "
        <<num2<<" , and t = "<<t <<endl; //Line 4

    funOne(num1, t); //Line 5
    cout<<"Line 6: In main after funOne: "<<"num1 = "
        <<num1<<" , num2 = "<<num2<<" , and t = "<<t
        <<endl; //Line 6

    funTwo(num2, num1); //Line 7
    cout<<"Line 8: In main after funTwo: "<<"num1 = "
        <<num1<<" , num2 = "<<num2<<" , and t = "
        <<t<<endl; //Line 8
    return 0; //Line 9
}

void funOne(int& a, int& x)
{
    int z;
    z = a + x; //Line 10

    cout<<"Line 11: In funOne: a = "<<a<<" , x = "<<x
        <<" , z = "<<z<<" , and t = "<<t<<endl; //Line 11

    x = x + 5; //Line 12
    cout<<"Line 13: In funOne: a = "<<a<<" , x = "<<x
        <<" , z = "<<z<<" , and t = "<<t<<endl; //Line 13

    a = a + 12; //Line 14
    cout<<"Line 15: In funOne: a = "<<a<<" , x = "<<x
        <<" , z = "<<z<<" , and t = "<<t<<endl; //Line 15

    t = t + 13; //Line 16
    cout<<"Line 17: In funOne: a = "<<a<<" , x = "<<x
        <<" , z = "<<z<<" , and t = "<<t<<endl; //Line 17
}

void funTwo(int& u, int v)
{
    int aTwo;

    aTwo = u; //Line 18
    cout<<"Line 19: In funTwo: u = "<<u<<" , v = "<<v
        <<" , aTwo = "<<aTwo<<" , and t = "<<t<<endl; //Line 19

    funOne(aTwo, u); //Line 20
    cout<<"Line 21: In funTwo after funOne:"<<" u = "
        <<u<<" , v = "<<v<<" , aTwo = "<<aTwo
        <<" , and t = "<<t<<endl; //Line 21

    u = u + 13; //Line 22
    cout<<"Line 23: In funTwo: u = "<<u<<" , v = "<<v
        <<" , aTwo = "<<aTwo<<" , and t = "<<t<<endl; //Line 23

    t = 2 * t; //Line 24
    cout<<"Line 25: In funTwo: u = "<<u<<" , v = "<<v
        <<" , aTwo = "<<aTwo<<" , and t = "<<t<<endl; //Line 25
}

```

输出:

```

Line 4: In main: num1 = 10, num2 = 20, and t = 15
Line 11: In funOne: a = 10, x = 15, z = 25, and t = 15

```

```

Line 13: In funOne: a = 10, x = 20, z = 25, and t = 20
Line 15: In funOne: a = 22, x = 20, z = 25, and t = 20
Line 17: In funOne: a = 22, x = 33, z = 25, and t = 33
Line 6: In main after funOne: num1 = 22, num2 = 20, and t = 33
Line 19: In funTwo: u = 20, v = 22, aTwo = 20, and t = 33
Line 11: In funOne: a = 20, x = 20, z = 40, and t = 33
Line 13: In funOne: a = 20, x = 25, z = 40, and t = 33
Line 15: In funOne: a = 32, x = 25, z = 40, and t = 33
Line 17: In funOne: a = 32, x = 25, z = 40, and t = 46
Line 21: In funTwo after funOne: u = 25, v = 22, aTwo = 32, and t = 46
Line 23: In funTwo: u = 38, v = 22, aTwo = 32, and t = 46
Line 25: In funTwo: u = 38, v = 22, aTwo = 32, and t = 92
Line 8: In main after funTwo: num1 = 22, num2 = 38, and t = 92

```

该程序有一个变量`t`，它是在任何一个函数定义前加以声明的。因为任何一个函数都没有声明变量`t`，所以变量`t`在程序任何地方都可以被访问。而且，程序包含两个`void`函数：`funOne()`，它有两个形参，这两个形参都是引用参数；`funTwo()`有一个引用参数和一个值参数。

在第5行，`main()`函数调用函数`funOne()`，传递给`funOne()`的实参是`num1`和`t`。这样，`funOne()`的形参`a`接收`num1`的地址，`x`接收`t`的地址。`a`值的任何变化会改变`num1`，`x`值的任何变化会改变`t`。因为`t`可以在程序的任何地方被访问，所以在第16行，函数`funOne()`使用`t`本身（参看第17行的输出）并改变了`t`的值。这样，可以通过使用引用参数或`t`本身操作`t`的值。

在第7行，`mian()`函数调用函数`funTwo()`并且将`num2`和`num1`作为实参传递。现在`num2`传递给`u`，它是一个引用参数，即`u`接收`num2`的地址。这样，`u`和`num2`指向同一个内存位置，即`num2`。因为`num1`传递给`v`并且`v`是一个值参量，所以`v`复制`num1`的内容。

在第20行，函数`funTwo()`调用函数`funOne()`。传递给`funOne()`的实参是`aTwo`（`funTwo()`的局部变量）和`u`（`funTwo()`的形参）。因为`funOne()`的参数`a`和`x`都是引用参量，所以`a`和`x`都接收相应实参的地址。现在`funTwo()`的参数`u`传递给`x`。因为`u`本身是一个引用参数并且指向`num2`，所以`u`和`x`都指向`num2`。对`x`所做的任何修改都会改变`num2`的值。这样，`u`、`x`和`num2`都指向同一个内存位置，即`num2`。

7.3 引用参数和返回值函数

第6章在讨论返回值函数时，只学习了怎样使用值参数。在返回值函数中也可以使用引用参数，尽管这种方法不推荐使用。通过定义，一个返回值函数返回一个单值，该值是通过`return`语句返回的。如果一个函数需要返回更多的值，应该将函数改为`void`函数并且使用相应的引用参数来返回这些值。

7.4 变量的范围

变量是在函数头部分、块内或块外进行声明的。这自然产生了一个问题：允许在程序任何地方访问任何一个变量吗？答案是否定的。在访问一个变量时，有几条特定的规则需要遵守。变量的范围取决于变量在哪里可以被访问（可用）。首先，定义下面两种术语，这两种术语在文献中被广泛使用：

局部变量：在函数（块）内声明的变量。

局部变量在函数（块）外是不能被访问的。

全局变量：在函数定义外声明的变量。

而且, C++ 不允许函数嵌套。即不能包含另一个函数体中的函数定义。

通常, 当访问变量时应用下面的规则:

1. 全局变量可以被函数或块访问, 如果满足:
 - a. 变量在函数(块)定义前被声明。
 - b. 函数名字不同于全局变量。
 - c. 函数所有参数的名字不同于全局变量的名字。
 - d. 所有局部变量有不同于全局变量的名字。
2. (嵌套块)在块中声明的一个变量可以被访问, 如果满足:
 - a. 只有在块内, 从被声明的地方一直到块结束。
 - b. 嵌套块不具有与块外(该块包含其他的嵌套块)相同的变量。
3. 函数的范围与在任何一个块外声明的变量范围是类似的。即函数的范围与全局变量的范围是相同的。

在考虑解释这些范围规则的例子之前, 首先注意在 for 语句中声明的变量范围。C++ 允许程序员在 for 语句的初始化语句中声明一个变量。例如, 下面的 for 语句:

```
for(int count = 1; count < 10; count++)
    cout<<count<<endl;
```

声明了变量 count 并将其初始化为 1。count 变量的范围只限制在 for 循环的内部。

注意: 在 for 语句中变量声明的规则不适用于标准 C++。在标准 C++ 中, 初始化语句中的变量声明范围从它声明的那一点开始一直到包含 for 语句块的结束。(为了确保这一点, 可以检查编译器的文档。)

下面的 C++ 程序有助于了解变量范围的规则:

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

const double rate = 10.50;
int z;
double t;

void one(int x, char y);
void two(int a, int b, char x);
void three(int one, double y, int z);

int main()
{
    int num, first;
    double x, y, z;
    char name, last;
    .
    .
    .
    return 0;
}

void one(int x, char y)
{
    .
    .
    .
}
```

```

int w;

void two(int a, int b, char x)
{
    int count;
    .
    .
    .
}

void three(int one, double y, int z)
{
    char ch;
    int a;
    .
    .
    .
    //Block four
    {
        int x;
        char a;
        .
        .
        .
    } //end Block four
    .
    .
    .
}

```

表 7.1 总结了标识符的作用范围。

表 7.1 标识符的作用范围

标识符	可见性 函数 one()	可见性 函数 two()	可见性 函数 three()	可见性 four 语句块	可见性 函数 main()
rate (main()函数前)	Y	Y	Y	Y	Y
z (main()函数前)	Y	Y	N	N	N
t (main()函数前)	Y	Y	Y	Y	Y
main()函数	Y	Y	Y	Y	Y
main()函数中的局部变量	N	N	N	N	Y
one (函数名)	Y	Y	N	N	Y
x (函数 one()中的形参)	Y	N	N	N	N
y (函数 one()中的形参)	Y	N	N	N	N
w (函数 two()之前的变量声明)	N	Y	Y	Y	N
two (函数名)	Y	Y	Y	Y	Y
a (函数 two()中的形参)	N	Y	N	N	N
b (函数 two()中的形参)	N	Y	N	N	N
x (函数 two()中的形参)	N	Y	N	N	N
函数 two()中的局部变量	N	Y	N	N	N
three (函数名)	Y	Y	Y	Y	Y
one (函数 three()中的形参)	N	N	Y	Y	N
y (函数 three()中的形参)	N	N	Y	Y	N
z (函数 three()中的形参)	N	N	Y	Y	N
ch (函数 three()中的局部变量)	N	N	Y	Y	N
a (函数 three()中的局部变量)	N	N	Y	N	N
x (four 语句块中的局部变量)	N	N	N	Y	N
a (four 语句块中的局部变量)	N	N	N	Y	N

注意, 函数 `three()` 不能调用函数 `one()`, 因为函数 `three()` 中已经有一个名为 `one` 的参数。同样, 标记为 `four` 的语句块不能使用 `int` 变量 `a`, 它在函数 `three()` 中已被声明, 因为块 `four` 中有一个名字为 `a` 的标识符。

1. 在第2章中讲过, C++ 不能自动初始化变量。然而, 一些编译器将全局变量初始化为默认值。例如, 如果一个全局变量是 `int`、`char` 或 `double` 类型, 它被初始化为零。
2. 在 C++ 中, `::` 称为范围解析运算符。通过使用范围解析运算符, 在函数 (块) 定义前声明的全局变量能通过函数 (或块) 存取, 即使函数 (或块) 有一个与变量相同的名字。例如, 在前面的例子中, 通过使用范围解析运算符, 函数 `main()` 可以把全局变量 `z` 看做 `::z`。同样, 假设全局变量 `t` 在函数 (比如 `funExample()`) 定义前被声明。那么 `funExample()` 可以使用范围解析运算符来访问变量 `t`, 即使 `funExample()` 有一个标识符 `t`。使用范围解析运算符, `funExample()` 将变量 `t` 看做 `::t`。而且, 在前面的例子中, 通过使用范围解析运算符, 函数 `three()` 可以调用函数 `one()`。
3. C++ 提供了一种访问函数定义后的全局变量的方法。在这种情况下, 函数不必包含任何一个与全局变量名字相同的标识符。在前面的例子中, 全局变量 `w` 在函数 `one()` 定义后被声明。现在函数 `one()` 不包含任何名字为 `w` 的标识符。所以, 如果在函数 `one()` 内声明 `w` 是一个 `extern` 变量, 那么 `w` 就可以被函数 `one()` 访问。为了声明 `w` 是函数 `one` 内的一个外部变量, 函数 `one()` 必须包含下面的语句:

```
extern int w;
```

在 C++ 中, `extern` 是一个保留字。上面语句中, `extern` 表示 `w` 是一个在其他地方声明的全局变量。这样, 当函数 `one()` 被调用时, 不会为 `w` 分配内存, 这就像 `one()` 内声明的变量一样。在 C++ 中, 外部声明还有其他一种用途, 但是本书不对此进行讨论。

7.5 全局变量的副作用

C++ 程序可能包含全局变量。然而, 使用全局变量具有副作用。使用全局变量的任何函数都不是独立的, 特别是不能用于多个程序中。而且, 如果多于一个函数使用相同的全局变量, 并且如果出现一些错误, 那么发现是什么错误并断定错误在什么地方是很困难的。在程序的一个区域由全局变量引起的错误可能被误认为是其他地方所产生的错误。强烈建议不要使用全局变量, 而是使用适当的参数。

7.6 静态和自动变量

迄今为止讨论的变量遵循下面的规则:

1. 只要程序执行, 全局变量就一直保存在内存中。
2. 块内的在块入口处被分配内存, 在块出口被释放。例如, 当一个函数被调用时, 函数形参和局部变量被分配内存, 在函数结束时内存就会被释放。

在块入口处分配内存、在块出口释放内存的变量称为自动变量。程序执行时一直保留内存分配的变量称为静态变量。默认情况下, 全局变量是静态变量, 块内的变量是自动变量。可以通过使用保留字 `static` 在块内声明一个静态变量。声明静态变量的语法是:

```
static dataType identifier;
```

语句:

```
static int x;
```

声明 x 是一个 `int` 型静态变量。

在块内声明的静态变量是块内的局部变量, 它们的范围与该块中的任何其他标识符的范围是一样的。

静态变量经常在声明时被初始化, 语句:

```
static int x=0;
```

声明 x 是一个 `int` 型的静态变量并且初始化其值为 0。

例 7.9

下面的程序给出了静态变量的行为:

```
//Program: Static and automatic variables
#include <iostream>
using namespace std;

void test();

int main()
{
    int count;
    for(count = 1; count <= 5; count++)
        test();

    return 0;
}

void test()
{
    static int x = 0;
    int y = 10;

    x = x + 2;
    y = y + 1;
    cout<<"Inside test x = "<<x<<" and y = "
        <<y <<endl;
}
```

输出:

```
Inside test x = 2 and y = 11
Inside test x = 4 and y = 11
Inside test x = 6 and y = 11
Inside test x = 8 and y = 11
Inside test x = 10 and y = 11
```

在函数 `test()` 中, x 是一个初始化为 0 的静态变量, y 是初始化为 10 的自动变量。`main()` 函数调用了 5 次 `test()` 函数。每次函数 `test()` 被调用时, 变量 y 被分配内存, 当函数调用结束时内存被释放。这样, 每次函数 `test()` 被调用时, 它打印的 y 值是相同的。然而, 因为 x 是静态变量, 程序执行期间它的内存一直存在。变量 x 被初始化为 0, 以后对函数 `test()` 的调用都使用 x 当前的值。

因为在函数调用之间静态变量的内存一直保留, 所以静态变量允许在函数调用间使用变量值。尽管可以使用全局变量, 但静态变量的局部范围会阻止其他函数对相应值进行操作。

在考察程序例子之前, 关于函数的另一个概念也值得一提: 函数重载。

7.7 函数重载：介绍

注意：该部分直到第14章才用到，所以在第14章之前，可以跳过该部分。

在C++中，几个函数可能有同一个名字。在C++术语中，这一概念称为函数重载。所以，重载一个函数是指创建具有相同名字的几个函数。然而，如果函数有相同的名字，每一个函数必须有不同的参数。参数类型决定了哪一个函数被执行。

假定需要编写一个判断两个数值大小的函数。数值可能是整数、浮点数、字符或字符串。可以编写下面的几个函数：

```
int largerInt(int x, int y);
char largerChar(char first, char second);
double largerDouble(double u, double v);
string largerString(string first, string second);
```

函数largerInt()确定两个整数中的较大数，函数largerChar()确定两个字符中较大者，等等。所有这些函数执行同样的操作。不必给这些函数命名不同的名字，每一个函数都可以使用相同的名字，比如说larger()，也就是重载了函数larger()。这样，可以将前面的函数原型编写如下：

```
int larger(int x, int y);
char larger(char first, char second);
double larger(double u, double v);
string larger(string first, string second);
```

例如，如果调用是larger(5,3)，那么第一个函数就会执行。如果调用是larger('A','9')，那么第二个函数就会执行，等等。当不同的数据有相同的动作时，可以使用函数重载。当然，为了使函数重载能工作，必须给出相应函数的定义。

7.8 有默认参数的函数

注意：该部分直到第13章才会用到。

该部分讨论有默认参数的函数。前面讲过，函数调用时，实参和形参的数目应该相等。C++对有默认参数的函数放松了该条件。当函数名字第一次出现时，例如在原型中，可以指定默认参数的值。通常，对于有默认参数的函数应该遵守下面的规则：

- 如果没有指定默认参数的值，那么参数会使用默认值。
- 所有默认参数必须是函数最右边的参数。
- 假设函数有不只一个默认参数。在函数调用中，如果赋给默认参数的值没有被指定，那么必须省略它右边的所有参数。
- 默认值可以是常量、全局变量或函数调用。
- 除了默认参数的默认值，调用函数有权指定其他值。
- 不能将一个常量值作为默认值赋给引用参数。

考虑下面的函数原型：

```
void funcExp(int x, int y, double t, char z = 'A', int u = 67,
             char v = 'G', double w = 78.34);
```

函数funcExp()有7个参数。参数z, u, v和w是默认参数。如果在对函数funcExp()的调用中

没有指定 z , u , v 和 w 的值, 那么就会使用它们的默认值。

假设有下面的语句:

```
int a,b;
char ch;
double d;
```

下面的函数调用是合法的:

1. funcExp(a, b, d);
2. funcExp(a, 15, 34.6, 'B', 87, ch);
3. funcExp(b, a, 14.56, 'D');

在语句1中, z , u , v 和 w 的默认值被使用。在语句2中, z 的默认值替换为 'B', u 的默认值替换为 87, v 的默认值被 ch 的值替换, w 的默认值被使用。在语句3中, z 的默认值被 'D' 替换, u , v 和 w 的默认值被使用。

下面的语句是不合法的:

1. funcExp(a, 15, 34.6, 46.7);
2. funcExp(b, 25, 48.76, 'D', 4567, 78.34);

在语句1中, 因为 z 的值被忽略, 所以所有其他的默认值必须被忽略。在语句2中, 因为 v 的值被忽略, 所以 w 的值也必须被忽略。

下面是默认参数的非法函数原型。

1. void funcOne(int x, double z = 23.45, char ch, int u = 45);
2. int funcTwo(int length = 1, int width, int height = 1);
3. void funcThree(int x, int& y = 16, double z = 34);

在语句1中, 因为第二个参数 z 是一个默认参数, 所以所有其他 z 后面的参数必须是默认参数。在语句2中, 因为第一个参数是默认参数, 所以所有的参数必须是默认值。在语句3中, 常量值不能赋值给 y , 因为 y 是一个引用参数。

例 7.10 进一步地演示了有默认参数的函数。

例 7.10

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

int volume(int l = 1, int w = 1, int h = 1);
void funcOne(int& x, double y = 12.34, char z = 'B');
int main()
{
    int a = 23;
    double b = 48.78;
    char ch = 'M';

    cout<<fixed<<showpoint;
    cout<<setprecision(2);

    cout<<"Line 1: a = "<<a<<", b = "<<b
        <<", ch = "<<ch<<endl;    //Line 1
    cout<<"Line 2: Volume = "<<volume()<<endl;
```

//Line 2

```

        cout<<"Line 3: Volume = "<<volume(5,4)<<endl;           //Line 3
        cout<<"Line 4: Volume = "<<volume(34)<<endl;           //Line 4
        cout<<"Line 5: Volume = "<<volume(6,4,5)<<endl;       //Line 5
        funcOne(a);                                           //Line 6
        funcOne(a, 42.68);                                   //Line 7
        funcOne(a, 34.65, 'Q');                               //Line 8
        cout<<"Line 9: a = "<<a<<"", b = "<<b
            <<"", ch = "<<ch<<endl;                           //Line 9
        return 0;
    }

    int volume(int l, int w, int h)
    {
        return l * w * h;                                     //Line 10
    }

    void funcOne(int& x, double y, char z)
    {
        x = 2 * x;                                           //Line 11
        cout<<"Line 12: x = "<<x<<"", y = " "<<y
            <<"", z = "<<z<<endl; //Line 12
    }

```

输出:

```

Line 1: a = 23, b = 48.78, ch = M
Line 2: Volume = 1
Line 3: Volume = 20
Line 4: Volume = 34
Line 5: Volume = 120
Line 12: x = 46, y = 12.34, z = B
Line 12: x = 92, y = 42.68, z = B
Line 12: x = 184, y = 34.65, z = Q
Line 9: a = 184, b = 48.78, ch = M

```

注意: 在本书中, main()函数的定义放在用户定义函数的前面。所以必须在函数原型中指定参数的默认值, 而且只能在函数原型中(不是在函数定义中)指定默认值。

7.9 编程实例: 对数字分类

在这个例子中, 使用函数来重写第5章给出的一个程序, 从给定的整数中判断数字是奇数还是偶数。

主要的算法是相同的:

1. 初始化变量, 将 zeros, odds 和 evens 赋值为 1。
2. 读入一个数字。
3. 如果数据是偶数, 偶数数量加 1; 如果是 0, 0 的数量加 1; 如果是奇数, 则奇数的数量加 1。
4. 对于每一个数字, 重复第 2 步和第 3 步。

程序的主要部分是初始化变量、读入和分类数字, 然后输出结果。为了简化 main() 函数和进一步演示参数传递, 程序包含了:

- 初始化变量 (例如 zeros, odds 和 evens) 的函数 initialize()。
- 获取数字的函数 getNumber()。

- `classifyNumber()` 函数判断数字是奇数还是偶数 (或者是零)。该函数会增加相应数字类型的数量。
- 函数 `printResults()` 打印结果。

下面来描述一下每一个函数。

initialize() `initialize()` 函数将初始化相应变量。需要初始化的变量是 `zeros`, `odds` 和 `evens`。如前所述, 它们的初始值都是零。很明显, 函数有三个参数。因为初始化这些变量的形参值必须传递到函数外, 所以这些形参必须是引用参数。

```
void initialize(int& zeroCount, int& oddCount, int& evenCount)
{
    zeroCount = 0;
    oddCount = 0;
    evenCount = 0;
}
```

getNumber() 函数 `getNumber()` 读入一个数字并且将该数字传递给 `main()` 函数。因为只需要传递一个数字, 所以该函数只有一个参数。该函数的形参必须是一个引用参数, 因为被读入的数据要传递到函数外。该函数是:

```
void getNumber(int& num)
{
    cin >> num;
}
```

也可以将函数 `getNumber()` 编写为一个具有返回值的函数。参见编程例子后面的注意事项。

classifyNumber() 函数 `classifyNumber()` 判断数字是奇数还是偶数, 如果是偶数, 还检查它是否为零。它也更新几个变量 (`zeros`, `odds` 和 `evens`) 的值。该函数需要知道被分析的数字。所以被分析的数字必须作为一个参数来传递。而且还因为该函数也将对应的数量加 1, 所以保存数量的变量 (即 `zeros`, `odds` 和 `evens`) 也必须作为一个参数传递给该函数。这样, 函数有 4 个参数。

因为数字只被分析, 所以只需要传递它的值。这样, 相应变量的形参是一个值参数。在分析数字后, 该函数将一些变量如 `zeros`, `odds` 和 `evens` 的值加 1。所以, 对应于这些变量的形参必须是引用参数。分析数字和将对应数量加 1 的算法与前面的算法是一样的。该函数的定义是:

```
void classifyNumber(int num, int& zeroCount, int& oddCount,
                  int& evenCount)
{
    switch(num % 2)
    {
        case 0: evenCount++;           //update even count
                 if(num == 0)          //number is also zero
                     zeroCount++;      //update zero count
                 break;
        case 1:
        case -1: oddCount++;            //update odd count
    } //end switch
}
```

printResults() `printResults()` 函数打印最后的结果。为了打印结果 (即零、奇数和偶数的数量), 该函数必须访问在 `main()` 函数中声明的变量 `zeros`, `odds` 和 `evens` 的值。所以, 函数有三个值。因为该函数只打印变量的值, 所以形参是值参数。该函数的定义是:

```

void printResults(int zeroCount, int oddCount, int evenCount)
{
    cout<<"There are "<<evenCount<<" evens, "
        <<"which also includes "<<zeroCount<<" zeros"<<endl;
    cout<<"Total number of odds are: "<<oddCount<<endl;
}

```

现在给出主要算法并显示 main() 函数如何调用这些函数。

主要算法

1. 调用 initialize() 来初始化变量的值。
2. 提示用户输入 20 个数字。
3. 对于每一个数字:
 - a. 调用 getNumber() 函数读入一个数字。
 - b. 输出数字。
 - c. 调用 classifyNumber() 来分类数字并将对应的数量加 1。
4. 调用 printResults() 打印最后的结果。

完整程序清单

```

//Program: Classify Numbers
//This program counts the number of zeros, odd, and even numbers

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

const int N = 20;

//function prototypes
void initialize(int& zeroCount, int& oddCount, int& evenCount);
void getNumber(int& num);
void classifyNumber(int num, int& zeroCount, int& oddCount,
                  int& evenCount);
void printResults(int zeroCount, int oddCount, int evenCount);

int main ()
{
    //variable declaration
    int counter; //loop control variable
    int number;  //stores a number
    int zeros;   //stores the number of zeros
    int odds;    //stores the number of odd integers
    int evens;   //stores the number of even integers

    initialize(zeros, odds, evens); //Step 1

    cout<<"Please enter "<<N<<" integers."
        <<endl; //Step 2
    cout<<"The numbers you entered are--> "<<endl;

    for (counter = 1; counter <= N; counter++) //Step 3
    {
        getNumber(number); //Step 3a

        cout<<setw(3)<<number; //Step 3b
        classifyNumber(number, zeros, odds, evens); //Step 3c
    } // end for loop
}

```

```

        cout<<endl;
        printResults(zeros, odds, evens);
        return 0;
    }

    void initialize(int& zeroCount, int& oddCount, int& evenCount)
    {
        zeroCount = 0;
        oddCount = 0;
        evenCount = 0;
    }

    void getNumber(int& num)
    {
        cin>>num;
    }

    void classifyNumber(int num, int& zeroCount, int& oddCount,
                       int& evenCount)
    {
        switch(num % 2)
        {
            case 0: evenCount++;
                    if(num == 0)
                        zeroCount++;
                    break;
            case 1:
            case -1: oddCount++;
        } //end switch
    }

    void printResults(int zeroCount, int oddCount, int evenCount)
    {
        cout<<"There are "<<evenCount<<" evens, "
             <<"which also includes "<<zeroCount<<" zeros"<<endl;
        cout<<"Total number of odds are: "<<oddCount<<endl;
    }

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Please enter 20 integers.
The numbers you entered are→
0 0 12 23 45 7 -2 -8 -3 -9 4 0 1 0 -7 23 -24 0 0 12
0 0 12 23 45 7 -2 -8 -3 -9 4 0 1 0 -7 23 -24 0 0 12
There are 12 evens, which also includes 6 zeros
Total number of odds are: 8

```

注意：在该程序中，因为假定数据从标准输入设备（键盘）输入，并且getNumber()只读入一个值，所以也可以将函数getNumber()编写为一个有返回值的函数。如果编写为一个返回值的函数，那么getNumber()函数的定义是：

```

int getNumber()
{
    int num;
    cin>>num;
    return num;
}

```

在这种情况下，main()函数中的语句（函数调用）：

```
getNumber(number);
```


应该被替换为:

```
number=getNumber( );
```

当然, 这需要修改函数原型。

7.10 编程实例: 数据比较

该编程实例演示:

- 如何在同一个程序中从多于一个文件中读入数据。
- 如何向一个文件发送输出。
- 如何产生条形图。
- 在函数和参数传递的作用下, 如何使用不同数据集的同一个程序段。
- 如何使用结构化设计来解决问题, 以及如何执行参数传递。

该程序分成两个部分。首先学习如何从多个文件中读入数据。接下来学习如何产生一个条形图。

当地大学的两组学生在夏季学期参加了一些选定的课程。课程首次开课并且由不同的老师讲课。在学期末, 对于相同的课程, 两组要进行同样的测试, 而且分数记录在不同的文件中。文件中的数据有下面的形式:

```
courseNo  score1, score2, ..., scoreN -999
courseNo  score1, score2, ..., scoreM -999
.
.
.
```

编写一个程序, 计算每一组每门课程的平均分。输出是下面的形式:

Course#	Group#	Course Average
A	1	80.50
	2	82.75
B	1	78.00
	2	75.35

输入 因为两组数据记录在独立的文件中, 所以输入数据出现在两个独立的文件中。

输出 如上所示。

问题分析和算法设计

读入输入数据是很直接的。假设第一组数据保存在“a: group1.txt”, 第二组数据保存在“a: group2.txt”。在处理完一组数据后, 就可以处理第二组相同课程的数据, 然后继续这一过程, 直到处理完所有的数据。课程的数据处理是类似的并且包含两个步骤:

1. a. 求出课程的总分数
b. 计算学习该课程的学生数量
c. 用总分数除以学生的数量来求得该课程的平均分
2. 输出结果

我们只比较相应课程的平均分, 并且文件中的数据是按照课程 ID 进行排列的。为了保证只有相同课程的平均分被比较, 需要为每一组比较课程 ID。如果相应的课程 ID 是不相同的, 则输出一条错误信息并且终止程序。

该讨论建议编写一个函数 `calculateAverage()` 来计算课程的平均分。同时也应该编写另一个函数 `printResult()` 以给定的格式输出结果。通过传递适当的参数, 可以使用相同的函数 `calculateAverage()` 和 `printResult()` 为这两个组处理每一门课程的数据。(在程序的第二部分中, 修改了函数 `printResult()`。)

前面的讨论可以转换为下面的算法:

1. 初始化变量。
2. 得到第一组和第二组课程 ID。
3. 如果课程 ID 不同, 那么打印错误信息并退出程序。
4. 计算第一组和第二组的课程平均分。
5. 用给定的形式打印结果。
6. 对于每一门课程, 重复步骤 2~6。
7. 打印最终结果。

变量 (main() 函数) 前面的讨论显示, 如果想对 `main()` 函数中的变量进行数据操作的话, 则需要下面的变量:

```
char courseId1;           //course ID for group 1
char courseId2;           //course ID for group 2

int numberOfCourses;      //to find the average for each group

double avg1;              //average for a course in group 1
double avg2;              //average for a course in group 2

double avgGroup1;         //average group 1
double avgGroup2;         //average group 2
ifstream group1;          //input stream variable for group 1
ifstream group2;          //input stream variable for group 2
ofstream outfile;         //output stream variable
```

接下来讨论函数 `calculateAverage()` 和 `printResult()`。然后将它们与 `main()` 函数结合在一起。

calculateAverage() 该函数为每一门课程计算平均分。因为输入数据存储在文件中, 并且文件在 `main()` 函数中被打开, 所以必须将与输入文件联系的 `ifstream` 变量传递到该函数中。而且, 在计算完课程平均分之后, 函数必须将课程平均分传递给 `main()` 函数。所以, 该函数有两个参数, 而且都必须是引用参数。

为了求得课程平均分, 首先必须计算该课程的总分数和参加该课程考试的学生人数, 然后将总分数与学生人数相除。这样, 需要一个变量来存储分数的总和, 一个变量来存储学生的人数, 一个变量来读入和存储一个分数。当然, 必须将总分数与学生人数变量初始化为零。

局部变量 (函数 calculateAverage()) 通过前面对数据操作的讨论, 确定了函数 `calculateAverage()` 需要三个变量。

```
double totalScore;        //to store the sum of scores
int numberOfStudent;      //to store the number of students
int score;                //to read and store a course score
```

上面的讨论翻译成下面 `calculateAverage()` 函数的算法:

- a. 声明变量。
- b. 将 `totalScore` 初始化为 0.0。
- c. 将 `numberOfStudent` 初始化为 0。
- d. 获取课程分数。

- e. 将 totalScore 与步骤 d 读入的课程分数相加来更新 totalScore。
- f. 将 numberOfStudent 加 1。
- g. 重复步骤 d、e，直到课程分数等于 -999。
- h. $\text{courseAvg} = \text{totalScore} / \text{numberOfStudent}$ 。

while 循环用来重复步骤 d、e 和 f。

现在准备编写函数 calculateAverage() 的定义。

```
void calculateAverage(ifstream& inp, double& courseAvg)
{
    double totalScore = 0.0;           //Steps a and b
    int numberOfStudent = 0;           //Steps a and c
    int score;                         //Step a

    inp>>score;                        //Step d
    while(score != -999)
    {
        totalScore = totalScore + score; //Step e
        numberOfStudent++;               //Step f
        inp>>score;                     //step d
    } //end while
    courseAvg = totalScore / numberOfStudent; //Step h.
} //end calculate Average
```

printResult() 函数 printResult() 打印组的课程 ID、组数目和平均分。输出存储在一个文件中。所以必须将 4 个参数传递给该函数：与输出文件联系的 ofstream 变量、组数目、课程 ID 和该组的平均分。ofstream 变量必须是引用参数。因为该函数只使用其他变量的值，所以剩余的 3 个变量应该是值参数。而且，从输出中可以清楚地看到，只在组 1 之前打印课程 ID。使用伪代码，该算法是：

```
if(group number == 1)
    print course ID
else
    print a blank

print group number and course average
```

函数 printResult() 定义如下所示：

```
void printResult(ofstream& outp, char courseID, int groupNo,
                double avg)
{
    if(groupNo == 1)
        outp<<" "<<courseID<<" ";
    else
        outp<<" ";
    outp<<setw(8)<<groupNo<<setw(15)<<avg<<endl;
} //end printResult
```

既然已经设计和定义了函数 calculateAverage() 和 printResult()，可以为函数 main() 描述算法。然而，在概括算法前，需要注意下面的事情：两个输入文件按照课程 ID 进行排列是很可能的，但是一个文件可能比另一个文件有更少的课程。可能直到在处理完两个文件之后才会发现该错误，并且发现一个文件还有未处理的数据。

在打印最后答案，即组 1 和组 2 的平均分之前要保证检查该错误。

主要算法：main() 函数

1. 声明变量（局部声明）。

2. 打开输入文件。
3. 如果不能打开一个文件, 则打印一条信息并且终止程序。
4. 打开输出文件。
5. 为了用小数点和尾零的固定小数格式来输出浮点数, 设置操纵符 `fixed` 和 `showpoint`。而且为了将浮点数输出到小数点后两位, 要设置精度到小数点后两位。
6. 为组 1 初始化课程平均分为 0.0。
7. 为组 2 初始化课程平均分为 0.0。
8. 初始化课程数量为 0。
9. 打印标题。
10. 得到组 1 的课程 ID: `courseId1`。
11. 得到组 2 的课程 ID: `courseId2`。
12. 对于组 1 和组 2 的每一门课程:
 - a. 如果 (`courseId1!=courseId2`), 那么:

```
{  
    cout<<"Data error: Course IDs do not match.\n";  
    return 1;  
}
```
 - b. 否则:

```
{  
    i. 计算组 1 的课程平均分 (调用函数 calculateAverage() 并且传递适当的参数)。  
    ii. 计算组 2 的课程平均分 (调用函数 calculateAverage() 并且传递适当的参数)。  
    iii. 打印组 1 的结果 (调用函数 printResults() 并且传递适当的参数)。  
    iv. 打印组 2 的结果 (调用函数 printResults() 并且传递适当的参数)。  
    v. 为组 1 更新平均分。  
    vi. 为组 2 更新平均分。  
    vii. 将课程的数目加 1。  
}
```
 - c. 获取组 1 的课程 ID:`courseId1`。
 - d. 获取组 2 的课程 ID:`courseId2`。
13. a. 如果 group 2 到达文件结尾:
输出 "Ran out of data for group 2 before group 1"。
b. 否则:
如果 group 1 到达文件结尾:
输出 "Ran out of data for group 1 before group 2"。
c. 否则:
分别输出第一组和第二组的课程平均分。
14. 关闭输入和输出文件。

完整程序清单

```
//Program: Comparison of class averages  
  
#include <iostream>  
#include <iomanip>  
#include <fstream>  
using namespace std;
```

```

//function prototypes
void calculateAverage(istream& inp, double& courseAvg);
void printResult(ofstream& outp, char courseId,
                int groupNo, double avg);

int main ()
{
    //Step 1
    char courseId1;    //course ID for group 1
    char courseId2;    //course ID for group 2
    int numberOfCourses;
    double avg1;       //average for a course in group 1
    double avg2;       //average for a course in group 2
    double avgGroup1;  //average group 1
    double avgGroup2;  //average group 2
    istream group1;    //input stream variable for group 1
    istream group2;    //input stream variable for group 2
    ofstream outfile;  //output stream variable

    group1.open("a:group1.txt");    //Step 2
    group2.open("a:group2.txt");    //Step 2
    if(!group1 || !group2)         //Step 3
    {
        cout<<"Unable to open files."<<endl;
        return 1;
    }

    outfile.open("a:student.out");  //Step 4
    outfile<<fixed<<showpoint;     //Step 5
    outfile<<setprecision(2);      //Step 5
    avgGroup1 = 0.0;               //Step 6
    avgGroup2 = 0.0;               //Step 7
    numberOfCourses = 0;           //Step 8

    //print heading: Step 9
    outfile<<"Course No   Group No   Course Average"<<endl;
    group1>>courseId1;              //Step 10
    group2>>courseId2;              //Step 11
    while(group1 && group2)         //Step 12
    {
        if(courseId1 != courseId2) //Step 12a
        {
            cout<<"Data error: Course IDs do not match."<<endl;
            cout<<"Program terminates."<<endl;
            return 1;
        }
        else //Step 12b
        {
            calculateAverage(group1, avg1); //Step 12b.i
            calculateAverage(group2, avg2); //Step 12b.ii
            printResult(outfile, courseId1, 1, avg1); //Step 12b.iii
            printResult(outfile, courseId2, 2, avg2); //Step 12b.iv
            avgGroup1 = avgGroup1 + avg1; //Step 12b.v
            avgGroup2 = avgGroup2 + avg2; //Step 12b.vi
            outfile<<endl;
            numberOfCourses++; //Step 12b.vii
        }
    }
}

```

```

        group1>>courseId1;                //Step 12c
        group2>>courseId2;                //Step 12d
    } //end while

    if(group1 && !group2)                    //Step 13a
        cout<<"Ran out of data for group 2 before group 1."<<endl;
    else                                    //Step 13b
        if (!group1 && group2)
            cout<<"Ran out of data for group 1 before group 2."<<endl;
        else                                //Step 13c
        {
            outfile<<"Avg for group 1: "
                <<avgGroup1 / numberOfCourses<<endl;
            outfile<<"Avg for group 2: "
                <<avgGroup2 / numberOfCourses<<endl;
        }
    group1.close();                        //Step 14
    group2.close();                        //Step 14
    outfile.close();                       //Step 14

    return 0;
}

void calculateAverage(ifstream& inp, double& courseAvg)
{
    double totalScore = 0.0;
    int numberOfStudent = 0;
    int score;
    inp>>score;
    while(score != -999)
    {
        totalScore = totalScore + score;
        numberOfStudent++;
        inp>>score;
    } //end while
    courseAvg = totalScore / numberOfStudent;
} //end calculate Average

void printResult(ofstream& outp, char courseId,
                int groupNo, double avg)
{
    if(groupNo == 1)
        outp<<" "<<courseId<<" ";
    else
        outp<<" ";
    outp<<setw(8)<<groupNo<<setw(15)<<avg<<endl;
}

```

程序输出实例:

Course No	Group No	Course Average
A	1	62.40
	2	61.13
B	1	67.91
	2	53.30
C	1	54.59
	2	57.38
D	1	65.71
	2	64.29

```

      E          1          71.80
                2          90.00
Avg for group 1: 64.48
Avg for group 2: 65.22

```

第一组输入数据:

```

A 80 100 50 10 32 90 89 100 23 50 -999
B 80 90 80 94 90 34 23 63 23 80 90 -999
C 10 30 20 10 90 50 89 23 90 68 90 10 60 90 73 35 90 -999
D 34 80 45 89 90 23 90 12 34 90 84 100 90 59 -999
E 100 83 93 20 63 -999

```

第二组输入数据:

```

A 20 75 40 25 80 89 100 60 -999
B 80 50 70 19 10 18 80 90 90 26 -999
C 100 30 20 40 90 50 18 90 90 45 90 80 70 30 35 40 -999
D 80 85 45 92 10 90 24 90 23 65 72 90 34 100 -999
E 95 100 88 98 69 -999

```

条形图

在商业世界中,很多公司主管人员通常喜欢用一些可视化的形式,例如条形图来了解结果。很多当前可用的软件包能用几种形式来分析数据,然后用一些可视化的形式,例如条形图和圆形分格统计表来显示结果。程序的第二部分的目的就是用下面的条形图来显示前面求得的结果:

```

Course      Course Average
ID   0   10   20   30   40   50   60   70   80   90   100
    |...|...|...|...|...|...|...|...|...|...|
A   *****
    #####
B   *****
    #####
Group 1 -- ****
Group 2 -- #####

```

条形图中的每一个符号代表两个点。如果课程平均分小于2,则不会打印任何结果。因为输出采用条形图的形式,所以需要修改函数 `printResult()`。

`printBar()` 函数 `printBar()` 打印课程 ID 并且用条形图代表课程的平均分。输出存在文件中。所以必须向该函数传递4个函数:与输出文件相联系的 `ofstream` 变量、组号(打印*或#)、课程 ID 和课程的平均分。为了打印条形图,函数使用了一个循环为每两个点打印一个符号。例如,如果平均分是78.45,那么必须打印39个符号来代表该平均值。为了计算要打印的符号数量,可以使用下面的整数除法。

```
numberOfSymbols = static_cast<int>(average)/2;
```

例如, `static_cast<int>(78.45)/2 = 78/2 = 39`

通过讨论之后,可以得到函数 `printResult()` 为:

```

void printResult(ofstream& outp, char courseId,
                int groupNo, double avg)
{
    int noOfSymbols;
    int count;

    if(groupNo == 1)

```

```

        outp<<setw(3)<<courseId<<"    ";
    else
        outp<<"    ";

    noOfSymbols = static_cast<int>(avg)/2;
    if(groupNo == 1)
        for(count = 1; count <= noOfSymbols; count++)
            outp<<"*";
    else
        for(count = 1; count <= noOfSymbols; count++)
            outp<<"#";
    outp<<endl;
} //end printResults

```

可以利用函数 `printHeading()` 来打印头两行的输出。该函数的定义为:

```

void printHeading(ofstream& outp)
{
    outp<<"Course          Course Average"<<endl;
    outp<<" ID      0    10    20    30    40    50    60    70"
        <<"    80    90   100"<<endl;
    outp<<"          |...|...|...|...|...|...|...|"
        <<"...|...|...|"<<endl;
} //end printHeading

```

如果用该函数替换前面的程序,(包括函数 `printHeading()` 及其输出语句 `Group 1--****` 和 `Group 2--####`) 重新运行该程序后,前面的数据会显示如下:

程序输出实例:

```

Course          Course Average
ID      0    10    20    30    40    50    60    70    80    90   100
|...|...|...|...|...|...|...|...|...|...|
A *****
#####
B *****
#####
C *****
#####
D *****
#####
E *****
#####

Group 1 -- ****
Group 2 -- ####
Avg for group 1: 64.48
Avg for group 2: 65.22

```

注意: 比较两个输出, 哪一个更好一些?

7.11 小结

1. 没有数据类型的函数称为 `void` 函数。
2. 没有任何值的返回语句可以用在 `void` 函数中。如果返回语句用在 `void` 函数中, 那么它一般是用来提前退出一个函数。
3. `void` 函数的头部以保留字 `void` 开始。

4. 在 C++ 中, void 是一个保留字。
5. void 函数可以有参数, 也可以没有参数。
6. void 函数的调用是一条独立的语句。
7. 为了调用 void 函数, 需要在一条独立的语句中将函数名与实参一起使用。
8. 有两种类型的实参: 值参数和引用参数。
9. 值参数接收它相应实参的副本。
10. 引用参数接收相应实参的地址 (内存位置)。
11. 值参数的相应实参是一个表达式、一个变量或一个常量值。
12. 常量值不能传递给引用参数。
13. 引用参数相应的实参必须是一个变量。
14. 当在形参数数据类型的后面包含 & 时, 形参成为一个引用参数。
15. 流变量应该通过引用参数传递给函数。
16. 如果形参需要改变实参的值, 在函数头必须将该形参声明为引用参数。
17. 标识符的范围是指程序中, 标识符的可访问区域。
18. 在函数 (或块) 中声明的变量称为局部变量。
19. 在每一个函数定义 (或块) 外的变量称为全局变量。
20. 函数名字的范围与任何块外声明的标识符范围相同。
21. 参看本章的范围规则 (标识符的范围部分)。
22. C++ 不允许函数嵌套定义。
23. 自动变量在函数 (或块) 入口处分配内存, 在函数 (或块) 出口处释放内存。
24. 静态变量在整个程序执行期间都保留其内存分配。
25. 在默认情况下, 全局变量是静态变量。
26. 在 C++ 中, 函数可以重载。
27. 如果函数被重载, 所有同名的函数具有不同的参数列表。
28. C++ 允许函数有默认参数。
29. 如果没有指定默认参数的值, 该参数就会使用默认值。
30. 所有默认参数必须是函数最右边的参数。
31. 假设函数有多于一个的默认参数。在函数调用中, 如果没有指定一个默认参数的值, 那么必须忽略它右边的所有参数。
32. 默认值可以是常量、全局变量或函数调用。
33. 函数调用可以为任何参数指定除了默认值的任何值。
34. 不能为引用参数指定常量作为默认值。

7.12 练习

1. 判断下面语句的对错。
 - a. 能改变引用参数值的函数也能改变实参的值。
 - b. 变量名不能传递给值参数。
 - c. 如果 C++ 函数不使用参数, 那么包含参数的括号仍然是需要的。
 - d. 在 C++ 中, 相应实参和形参的名字必须是相同的。
 - e. 只要引用参数的值改变, 实参值也会改变。

- f. 在 C++ 中, 函数定义可以嵌套, 也就是说函数的定义可以包含在另一个函数体内。
 - g. 在程序中使用全局变量比使用局部变量是一种更好的风格, 因为这样可以避免额外的变量。
 - h. 在程序中, 全局常量和全局变量一样危险。
 - i. 静态变量在函数调用期间其内存一直保持分配。
2. 对于给出的程序代码, 指出下面所列的部分。
- a. 函数原型、函数头、函数体和函数定义。
 - b. 函数调用语句、形参和实参。
 - c. 值参数和引用参数。
 - d. 局部变量和全局变量。

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

int one;
void hello(int&, double, char);

int main()
{
    int x;
    double y;
    char z;
    .
    .
    .
    hello(x,y,z);
    .
    .
    hello(x, y-3.5, 'S');
    .
    .
    .
}

void hello(int& first, double second, char ch)
{
    int num;
    double y;
    int u ;
    .
    .
    .
}
```

3. a. 解释形参和实参的区别。
b. 解释值参数和引用参数的区别。
c. 解释局部变量和全局变量的区别。

4. 下面程序的输出是什么?

```
#include <iostream>
#include <iomanip>
using namespace std;
void test(int first, int& second);
int main ()
{
```



```

        int num;

        num = 5;
        test(24, num);
        cout<<num<<endl;
        test(num, num);
        cout<<num<<endl;
        test(num*num, num);
        cout<<num<<endl;
        test(num+num, num);
        cout<<num<<endl;

        return 0;
    }

    void test(int first, int& second)
    {
        int third;

        third = first + second * second + 2;
        first = second - first;
        second = 2 * second;
        cout<<first<<" "<<second<<" "
             <<third<<endl;
    }

```

5. 假设有下面的输入值:

7 3 6 4

2 6 3 5

给出下面程序的输出:

```

#include <iostream>
using namespace std;
void goofy(int& , int& , int , int& );
int main()
{
    int first, second, third, fourth;

    first = 3; second = 4; third = 20; fourth = 78;
    cout<<first<<" "<<second<<" "<<third<<" "<<fourth<<endl;
    goofy(first, second, third, fourth);
    cout<<first<<" "<<second<<" "<<third<<" "<<fourth<<endl;
    fourth = first * second + third - fourth;
    goofy(fourth, third, first, second);
    cout<<first<<" "<<second<<" "<<third<<" "<<fourth<<endl;
    return 0;
}

void goofy(int& a, int& b, int c, int& d)
{
    cin>>a>>b>>c>>d;
    c = a * b + d - c;
    c = 2 * c;
}

```

6. 下面程序的输出是什么?

```

#include <iostream>
using namespace std;
int x;
void mickey(int&, int);
void minnie(int, int&);

```

```

int main()
{
    int first;
    int second = 5;

    x = 6;
    mickey(first, second);
    cout<<first<<" "<<second<<" "<<x<<endl;
    minnie(first, second);
    cout<<first<<" "<<second<<" "<<x<<endl;
    return 0;
}

void mickey(int& a, int b)
{
    int first;

    first = b + 12;
    a = 2 * b;
    b = first + 4;
}

void minnie(int u, int& v)
{
    int second;

    second = x;
    v = second + 4;
    x = u + v;
}

```

7. 在下面的程序中, 请填写相应语句的执行顺序 (执行的逻辑顺序)。

```

#include <iostream>
using namespace std;

void func(int val1, int val2);
int main()
{
    int num1, num2;
    _____ cout<<"Please enter two integers.\n";
    _____ cin>>num1>>num2;
    _____ func (num1, num2);
    _____ cout<<" The two integers are "<<num1<<" , "<<num2<<endl;
    _____ return 0;
}

void func (int val1, int val2)
{
    int val3, val4;
    _____ val3 = val1 + val2;
    _____ val4 = val1 * val2;
    _____ cout<<"The sum and product are "<<val3<<" and "<<val4;
}

```

8. 下面代码段的输出是什么? (注意: alpha 和 beta 是 int 变量。)

```

alpha = 5;
beta = 10;
if(beta >= 10)
{
    int alpha = 10;
}

```

```

    beta = beta + alpha;
    cout<<alpha<<' '<<beta<<endl;
}
cout<<alpha<<' '<<beta<<endl;

```

9. 如果用下面的语句替换例 7.8 的第 7 行, 请指出程序的输出。

```
funTwo(t, num1);
```

指出每一行语句执行后变量的值。

10. 考虑下面的程序。程序的正确输出是什么? 指出每一行语句执行后变量的值。

```

#include <iostream>
using namespace std;
void funOne(int& a);
int main()
{
    int num1, num2;
    num1 = 10; //Line 1
    num2 = 20; //Line 2
    cout<<"Line 3: In main: num1 = "<<num1
        <<"", num2 = "<<num2<<endl; //Line 3
    funOne(num1); //Line 4
    cout<<"Line 5: In main after funOne: num1 = "
        <<num1<<"", num2 = "<<num2<<endl; //Line 5
    return 0; //Line 6
}

void funOne(int& a)
{
    int x = 12;
    int z;
    z = a + x; //Line 7
    cout<<"Line 8: In funOne: a = "<<a<<"", x = "<<x
        <<"", and z = "<<z<<endl; //Line 8
    x = x + 5; //Line 9
    cout<<"Line 10: In funOne: a = "<<a<<"", x = "<<x
        <<"", and z = "<<z<<endl; //Line 10
    a = a + 8; //Line 11
    cout<<"Line 12: In funOne: a = "<<a<<"", x = "<<x
        <<"", and z = "<<z<<endl; //Line 12
}

```

11. 考虑下面的函数原型:

```
void testDefaultParam(int a, int b = 7, char z = '*');
```

下面的哪个函数调用是正确的?

- (i) testDefaultParam(5);
- (ii) testDefaultParam(5, 8);
- (iii) testDefaultParam(6, '#');
- (iv) testDefaultParam(0, 0, '*');

12. 考虑下面的函数定义:

```

void defaultParam(int u, int v = 5, double z = 3.2)
{
    int a;
    u = u + static_cast<int>(2 * v + z);
}

```

```

    a = u + v * z;
    cout<<"a = "<<a<<endl;
}

```

下面的哪个函数调用是正确的?

- (i) defaultParam(6);
- (ii) defaultParam(3,4);
- (iii) defaultParam(3, 0, 2.8);

7.13 编程练习

1. 考虑下面 main() 函数的定义:

```

int main()
{
    int x, y;
    char z;
    double rate, hours;
    double amount;
    .
    .
    .
}

```

- a. 编写函数 initialize() 的定义来初始化变量 x 和 y 的值为 0, 变量 z 为空格字符。
 - b. 编写函数 getHoursRate() 的定义提示用户输入工作小时和每小时的薪金, 并初始化 main() 函数中的变量 hours 和 rate。
 - c. 编写返回值函数 payCheck() 的定义, 以工作时间和每小时的薪金计算并返回雇员的薪水。工作时间和每小时的薪金分别存储在 main() 函数的变量 hours 和 rate 中。计算薪水的方法如下: 对于前 40 个小时, 每小时薪金是给定值; 对于超过 40 小时的部分, 每小时薪金是给定值的 1.5 倍。
 - d. 编写函数 printCheck() 的定义来打印工作小时、每小时薪水和总薪金。
 - e. 编写函数 funcOne() 的定义, 提示用户输入一个数字。然后函数将 x 值乘以 2 并加上 y 值, 再减去用户输入的值。
 - f. 编写函数 nextChar() 的定义来设置存储在 z 中的下一个值。
 - g. 编写函数 main() 的定义来测试每一个函数。
2. 例 7.5 中的函数 printGrade() 被设计为一个 void 函数, 用来计算和输出课程等级。课程分数作为参数传递给函数 printGrade()。将函数 printGrade() 重新编写为一个返回值函数, 用来计算和返回课程等级。(课程等级必须在 main() 函数中输出。) 而且, 将函数的名字改为 calculateGrade()。
3. 修改 7.9 小节的编程实例。程序从标准输入设备(键盘)输入数据, 并且将结果输出到标准输出设备上(屏幕)。程序只能处理 20 个数字。重新编写该程序并加入下面的要求。
- a. 数据从一个未指定长度的文件中输入到程序中, 即程序预先并不知道文件中有多少数字。
 - b. 将程序的输出保存到一个文件中。
 - c. 修改函数 getNumber(), 让它可以从输入文件(在 main() 函数中打开)中读入一个数字, 将数字输出到一个输出文件(在 main() 函数中打开)中, 并且将读入的数字发送给 main() 函数。每行只打印 10 个数字。
 - d. 让程序求得数字的和与平均值。
 - e. 修改函数 printResult(), 可以将最后的结果输出到输出文件中(在 main() 函数中打开)。除

了输出正确的计数外，程序还应该能输出数字的和与平均值。

4. 重新编写第 5 章的编程练习 11。其中 `main()` 函数仅仅是一个函数调用的集合。程序应该使用下面的函数：

- a. 函数 `openFiles()`：该函数打开输入和输出文件，并且用固定的小数点和尾零的方式将浮点数的输出设置到小数点后两位。
- b. 函数 `initialize()`：该函数初始化变量，如 `countFemale`，`countMale`，`sumFemaleGPA` 和 `sumMaleGPA`。
- c. 函数 `sumGrades()`：该函数计算男生和女生的 GPA。
- d. 函数 `averageGrade()`：该函数计算男生和女生的平均 GPA。
- e. 函数 `printResults()`：该函数输出相关结果。
- f. 可能没有全局变量。使用正确的参数向函数传入或传出信息。

5. 编写一个程序输出某天是一年中的第几天，给定的数据格式为 `month-day-year`。例如，如果输入是 1-1-02，该天是一年中的第一天，如果输入是 12-25-02，那么该天是一年中的第 359 天。程序应该检查闰年。如果某一年能被 4 整除但是不能被 100 整除，那么该年是闰年。例如，1992 年和 2008 年能被 4 整除，但不能被 100 整除。如果某年能被 100 整除同时还能被 400 整除，那么该年也是闰年。例如，1600 年和 2000 年能被 400 整除，然而，1800 年不是闰年，因为 1800 不能被 400 整除。

6. 编写一个程序，读入一名学生的名字和测试分数。然后程序为每一名学生计算测试平均分并指定正确的等级。等级范围如下：90 ~ 100: A; 80 ~ 89: B; 70 ~ 79: C; 60 ~ 69: D; 0 ~ 59: F。程序必须使用下面的函数：

- a. void 函数 `calculateAverage()`，用来为每一名学生计算平均分。使用循环读入并计算 5 个测试分数的和。（函数不输出平均测试分数。该任务必须在 `main()` 函数中完成。）
- b. 返回值函数 `calculateGrade()`，用来计算和返回每一名学生的等级。（函数不输出等级，该任务必须在 `main()` 函数中完成。）

用下面的数据测试程序。从文件中读入数据并将输出发送到一个文件中。不要使用任何全局变量。使用正确的参数向函数传入或传出值。

```
A 75 83 77 91 76
B 80 90 95 93 48
C 78 81 11 90 73
D 92 83 30 69 87
E 23 45 96 38 59
F 60 85 45 39 67
G 27 31 52 74 83
H 93 94 89 77 97
I 79 85 28 93 82
J 85 72 49 75 63
```

输出实例：

输出应该采用下面的格式：

Student	Test1	Test2	Test3	Test4	Test5	Average	Grade
A	75	83	77	91	76		
B	80	90	95	93	48		
C	78	81	11	90	73		
D	92	83	30	69	87		
E	23	45	96	38	59		
F	60	85	45	39	67		

G	27	31	52	74	83
H	93	94	89	77	97
I	79	85	28	93	82
J	85	72	49	75	63

Class Average =

7. 编写一个程序用来处理文本文件。程序应该读取一个文本文件并且将数据输出到另一个文件。程序还应该输出单词数、行数和段落数。

必须编写和使用下面的函数：

- `initialize()`: 该函数初始化所有 `main()` 函数中的变量。
- `processBlank()`: 该函数读写空格。只要键入一个非空格字符 (除了空白字符), 一行中单词的数量就会加 1。`updateCount()` 函数将一行中单词的数量置零。在处理完空格后函数退出。
- `copyText()`: 该函数读写非空格字符。只要键入空格, 它就退出执行。
- `updateCount()`: 该函数发生在每一行的末尾。它更新总的单词数量、将行的数量加 1 并将一行的单词数量置零。如果在一行没有单词, 它将段的数量加 1。一个空格行 (两段之间) 用来区分段落, 它不能被记入行数中。
- `printTotal()`: 该函数输出单词数、行数和段落数。

程序应该从一个文件中读入数据并且将输出发送到另一个文件中。不要使用任何全局变量。使用正确的参数向函数传入或传出值。使用下面的 `main()` 函数来测试程序：

```
int main()
{
    variables declaration
    open files

    read a character
    while (not end of file)
    {
        while(not end of line)
        {
            processBlank(parameters);
            copyText(parameters);
        }
        updateCount(parameters);
        read a character;
        .
        .
    }
    printTotal(parameters);
    close files;
    return 0;
}
```


第8章 用户定义的简单数据类型、命名空间和 string 类型

本章要点:

- 学习如何创建并操纵自定义的简单数据类型——枚举类型
- 了解 typedef 语句
- 学习 namespace 机制
- 研究 string 数据类型, 并且学习如何使用各种 string 函数来操纵字符串

C++ 的简单数据类型分为三类: 整型、浮点型和枚举型。在前面的章节中, 主要使用了整型和浮点数据类型。在本章中, 将学习枚举类型。此外, 语句 “using namespace std;” (在第2章中讨论过) 在每一个使用 ANSI/ISO 标准 C++ 头文件的程序中都被使用。本章的第2部分研究该语句的目的。实际上, 将会了解到什么是命名空间机制。还将学习 string 类型和很多有用的函数, 可以使用它们来有效地对字符串进行操纵。

8.1 枚举类型

如第2章中所述, 数据类型是一个集合的值, 同时, 还定义了对该集合中数据的操作方式。例如 int 数据类型包含从 -2 147 483 648 到 2 147 483 647 的整数以及对这些数值的操作, 也就是算术操作 (+、-、*、/)。因为程序的主要目标是对数据进行操作, 所以数据类型的概念对于任何一种编程语言来说都是很重要的。通过提供数据类型, 可以指定什么值是合法的, 并且告诉用户对哪些类型可以进行哪种类型的操作。因此, 系统提供了一种内建的错误检查机制。

到现在为止所使用的数据类型主要是 int、bool、char 和 double。尽管这些数据类型对于解决任何一种问题来说基本上都是足够用的, 但是当这些数据类型不能充分解决一种特定问题时, 就可能会发生数据类型不够用的情况。C++ 为用户提供了一种机制来创建自定义的数据类型, 这大大提高了编程语言的灵活性。

在该部分中, 将会学习怎样创建称为枚举类型的简单数据类型。在后面的章节中, 将会学习更高级的技术来创建复合数据类型。

为了定义枚举类型, 需要下列项:

- 数据类型的名字。
- 数据类型值的集合。
- 对该值操作的集合。

C++ 允许定义一种新的简单数据类型, 其中可以指定它的名字和值, 但是不能指定操作。阻止用户创建自己的操作避免了潜在的系统失败。

为该数据类型指定的值必须是合法的标识符。

枚举类型的语法是:

```
enum typeName{ value1,value2,...};
```

其中 value1,value2...是称为枚举值的标识符。在 C++ 中, enum 是一个保留字。

通过列出括号中所有的值,还可以指定值之间的顺序。即 value1<value2<value3<...。这样,枚举类型是一个有序的集合。此外,赋给这些枚举值的默认值是从0开始的,即赋给 value1 的默认值是0,赋给 value2 的默认值是1,等等。(当定义枚举类型变量时,除了默认值,也可以为枚举值指定不同的值。)

例 8.1

语句:

```
enum colors{ brown, blue, red, green, yellow};
```

定义了一种称为 colors 的数据类型,属于该数据类型的值是 brown、blue、red、green 和 yellow。

例 8.2

语句:

```
enum standing{ freshman, sophomore, junior, senior};
```

定义 standing 是一种枚举类型。属于 standing 的值是 freshman、sophomore、junior 和 senior。

例 8.3

考虑下面的语句:

```
enum grades{ 'A', 'B', 'C', 'D', 'F'}; //Illegal enumeration type  
enum places{ 1st, 2nd, 3rd, 4th};      //Illegal enumeration type
```

这些是不合法的枚举类型,因为没有有一个值有合法的标识符。然而,下面是合法的枚举类型:

```
enum grades{ A, B, C, D, F};  
enum places{ first, second, third, fourth};
```

如果一个值已经在某一枚举类型中使用,它不能再被同一块中的任何其他枚举类型使用。该规则也应用于在任何块外声明的枚举类型。例 8.4 演示了这一概念。

例 8.4

考虑下面的语句:

```
enum mathStudent{ John, Bill, Cindy, Lisa, Ron};  
enum compStudent{ Susan, Cathy, John, William}; //Illegal
```

假设这些语句在同一程序的同一块中。第二个枚举类型 compStudent 是不允许的,因为值 John 在枚举类型 mathStudent 中已被使用了。

8.1.1 声明变量

一旦定义了数据类型,就可以声明该类型的变量。声明 enum 类型变量的语法与前面相同:

```
dataType identifier, identifier,...;
```

语句:

```
enum sports{ basketball, football, hockey, baseball, soccer,  
            volleyball};
```

定义了一个名为 sports 的枚举类型。语句:


```
sports popularSport, mySport;
```

声明 popularSport 和 mySport 是一个 sport 类型的变量。

8.1.2 赋值

一旦声明了变量，就可以在变量中存储值。假设有前面的声明，那么语句：

```
popularSport=football;
```

将 football 存入 popularSport 中，语句：

```
mySport=popularSport;
```

将 popularSport 的值存储到 mySport 中。

8.1.3 对枚举类型的操作

对枚举类型不允许有算术操作。所以下面的语句是不合法的：

```
mySport = popularSport + 2;           //Illegal
popularSport = football + soccer;      //Illegal
popularSport = popularSport * 2;       //Illegal
```

而且，也不允许对枚举类型进行自加和自减操作。所以下面的语句是不合法的：

```
popularSport++; //Illegal
popularSport--; //Illegal
```

假设想将 popularSport 的值加 1。可以使用 cast 运算符，如下所示：

```
popularSport = static_cast<sports>(popularSport + 1);
```

当使用类型名时，编译器假设用户知道自己在做什么。这样，上面的语句就会被编译并且在执行期间，它提前将 popularSport 的值变为列表中的下一个值。考虑下面的语句：

```
popularSport = football;
popularSport = static_cast<sports>(popularSport + 1);
```

在第二条语句后，popularSport 的值是 hockey。同样，语句：

```
popularSport = football;
popularSport = static_cast<sports>(popularSport - 1);
```

会将 basketball 存储在 popularSport 中。

关系运算符

因为枚举类型是一些值的顺序集合，所以关系运算符可以用在枚举类型中。假设有枚举类型 sports 以及前面定义的变量 popularSport 和 mySport。那么：

```
football <= soccer is true
hockey > basketball is true
baseball < football is false
```

假设：

```
popularSport = soccer;
mySport = volleyball;
```

那么：

```
popularSport < mySport is true
```

8.1.4 枚举类型和循环

前面讲过枚举类型是一种整数类型，还讲过使用 `cast` 运算符（也就是类型名）可以对枚举类型的值进行自加、自减和比较，所以可以在循环中使用这些枚举类型。假设 `mySport` 是一个前面声明的变量。考虑下面的 `for` 循环：

```
for(mySport = basketball; mySport <= soccer;
    mySport = static_cast<sports>(mySport+1))
...
```

该 `for` 循环执行 5 次。

在循环中使用枚举类型提高了程序的可读性。

8.1.5 枚举类型的输入/输出

因为输入和输出只对内建的数据类型，例如 `int`, `char`, `double`，等等，进行了定义，枚举类型既不能进行输入也不能进行输出（直接地）。然而，可以间接地输入和输出枚举类型。例 8.5 演示了这一概念。

例 8.5

假设有下面的语句：

```
enum courses{ algebra, basic, pascal, cpp, philosophy, analysis,
              chemistry, history};
courses registered;
```

第一条语句定义了枚举类型 `courses`；第二条语句声明了 `courses` 类型的变量 `registered`。可以在 `char` 数据类型的帮助下读入（即输入）该枚举类型。注意有一些枚举类型的值可以通过读入第一个字符来区分，还有一些枚举类型的值可以通过读入前两个字符来区分。例如，可以通过第一个字符来区分 `algebra` 和 `basic`，可以通过前两个字符的办法来区分 `algebra` 和 `analysis`。为了读入这些值（比如从键盘上），需要先读入两个字符，然后使用一个选择结构将值赋给变量 `registered`。这样，需要声明 `char` 类型的两个变量。

```
char ch1, ch2;
cin >> ch1 >> ch2; //read two characters
```

下面的 `switch` 语句将正确的值赋给变量 `registered`：

```
switch(ch1)
{
case 'a': if(ch2 == 'l')
          registered = algebra;
          else
          registered = analysis;
          break;
case 'b': registered = basic;
          break;
case 'c': if(ch2 == 'h')
          registered = chemistry;
          else
          registered = cpp;
          break;
```

```

    case 'h': registered = history;
               break;
    case 'p': if(ch2 == 'a')
               registered = pascal;
            else
               registered = philosophy;
               break;
    default: cout<<"Illegal input."<<endl;
}

```

同样，可以间接地输出枚举类型：

```

switch(registered)
{
    case algebra: cout<<"algebra";
                  break;
    case analysis: cout<<"analysis";
                  break;
    case basic: cout<<"basic";
                break;
    case chemistry: cout<<"chemistry";
                   break;
    case cpp: cout<<"cpp";
              break;
    case history: cout<<"history";
                 break;
    case pascal: cout<<"pascal";
                break;
    case philosophy: cout<<"philosophy";
}

```

注意：如果试图直接输出一个枚举类型的值，那么将会输出赋给该枚举类型的值。例如假设 `registered=algebra`；下面的语句将会输出零，因为赋给 `algebra` 的默认值是零：

```
cout<<registered<<endl;
```

同样，下面的语句将会输出 4：

```
cout<<philosophy<<endl;
```

8.1.6 函数和枚举类型

与任何其他简单数据类型一样，可以通过值或者引用将枚举类型作为参数传递给函数。而且，与任何其他数据类型一样，函数可以返回枚举类型的值。通过使用这种方法，可以使用函数来输入和输出枚举类型。

下面的函数从键盘输入数据并且返回枚举类型的值。假设枚举类型 `courses` 如前面所定义。

```

courses readCourses()
{
    courses registered;
    char ch1, ch2;

    cout<<"Enter the first two letters of the course: "<<endl;
    cin>>ch1>>ch2;
    switch(ch1)
    {
        case 'a': if(ch2 == 'l')
                   registered = algebra;

```

```

        else
            registered = analysis;
        break;
    case 'b': registered = basic;
        break;
    case 'c': if(ch2 == 'h')
        registered = chemistry;
        else
            registered = cpp;
        break;
    case 'h': registered = history;
        break;
    case 'p': if(ch2 == 'a')
        registered = pascal;
        else
            registered = philosophy;
        break;
    default: cout<<"Illegal input."<<endl;
} //end switch

return registered;
}

```

下面的函数输出一个枚举类型值:

```

void printEnum(courses registered)
{
    switch(registered)
    {
        case algebra: cout<<"algebra";
            break;
        case analysis: cout<<"analysis";
            break;
        case basic: cout<<"basic";
            break;
        case chemistry: cout<<"chemistry";
            break;
        case cpp: cout<<"cpp";
            break;
        case history: cout<<"history";
            break;
        case pascal: cout<<"pascal";
            break;
        case philosophy: cout<<"philosophy";
    } //end switch
} //end printEnum

```

8.1.7 定义枚举类型时声明变量

在前面的部分, 首先定义了一个枚举类型, 然后声明了该类型的变量。C++ 允许将这两个步骤结合起来。即当定义一个枚举类型时可以声明该类型的变量。例如, 语句:

```
enum grades{ A, B, C, D, F} courseGrade;
```

定义了枚举类型 `grades` 并且声明了一个该类型的变量 `courseGrade`。

同样, 语句:

```
enum coins{penny, nickel, dime, halfDollar, dollar} change, usCoins;
```

定义了枚举类型 `coins` 并且声明了该类型的两个变量 `change` 和 `usCoins`。

8.1.8 匿名数据类型

在变量声明中直接指定值而没有指定名字的数据类型称为匿名类型。下面的语句创建了一个匿名类型：

```
enum {basketball, football, baseball, hockey} mySport;
```

该语句指定了值并且声明了一个变量 `mySport`，但是没有给定数据类型的名字。然而创建一个匿名类型也有缺点。首先，因为类型没有名字，所以不能将匿名类型作为一个参数传递给函数，并且函数不能返回一个匿名类型的值。其次，在匿名类型中使用的值可以在另一个匿名类型中使用，但是相应类型中的值会不同地被对待。考虑下面的语句：

```
enum {English, French, Spanish, German, Russian} languages;  
enum {English, French, Spanish, German, Russian} foreignLanguages;
```

尽管变量 `languages` 和 `foreignLanguages` 有相同的值，但是编译器会将它们作为不同类型的变量。所以下面的语句是不合法的：

```
languages = foreignLanguages; //Illegal
```

虽然这些方法是可用的，但是应该慎重的使用它们。为了避免混淆，应该首先定义一个枚举类型然后声明这些变量。

下面描述 C++ 中的 `typedef` 语句。

8.1.9 typedef 语句

在 C++ 中，可以使用 `typedef` 语句为前面定义的数据类型创建同义字或别名。`typedef` 语句的语法通常如下所示：

```
typedef existingTypeName newTypeName;
```

在 C++ 中，`typedef` 是一个保留字。注意 `typedef` 语句不能创建任何新的数据类型，它只能为一种现存的数据类型创建一个别名。

例 8.6

语句：

```
typedef int integer;
```

为 `int` 数据类型创建了别名 `integer`。同样，语句：

```
typedef double real;
```

为 `double` 数据类型创建了一个别名 `real`。语句：

```
typedef double decimal;
```

为数据类型 `double` 创建了一个别名 `decimal`。

使用 `typedef` 语句可以创建例 8.7 所显示的布尔数据类型。

例 8.7

在第 4 章中讲过，C++ 中的逻辑（布尔）表达式的值是 1 或 0，它们实际上是 `int` 值。作为一个

逻辑值, 1 代表 true, 0 代表 false。考虑下面的语句:

```
typedef int Boolean;           //Line 1
const Boolean True = 1;        //Line 2
const Boolean False = 0;       //Line 3

Boolean flag;                  //Line 4
```

第1行语句为 int 类型创建了一个别名 Boolean。第2行和第3行声明了两个常量 True 和 False 并且将它们分别初始化为 1 和 0。第4行语句声明 flag 是一个 Boolean 类型的变量。所以下面的语句是合法的:

```
flag=True;
```

8.2 编程实例: 石头、剪子和布的游戏

每一个人都很熟悉石头、剪子和布的游戏。该游戏有两名游戏者, 每一名游戏者选择三个对象中的一个: 石头、剪子或布。如果游戏者 1 选择石头, 第2名游戏者选择布, 那么第2名游戏者将获胜, 因为布能包住石头。游戏规则如下:

- 如果两名游戏者选择相同的对象, 那么是平局。
- 如果一名游戏者选择石头, 另一名选择剪刀, 那么选择石头的游戏者将会赢得游戏, 因为石头能破坏剪刀。
- 如果一名游戏者选择石头, 另一名选择布, 那么选择布的游戏者将会赢得游戏, 因为布可以包住石头。
- 如果一名游戏者选择剪刀, 另一名选择布, 那么选择剪刀的游戏者将会赢得游戏, 因为剪刀可以剪布。

编写一个交互式的程序允许两个人来参与这一游戏。

输入 程序有两种类型的输入:

- 用户对游戏的回答
- 游戏者的选择

输出 游戏者的选择和每一次游戏的获胜者。游戏结束后, 应该输出游戏的次数和每一名游戏者获胜的次数。

问题分析和算法设计

游戏者通过键盘输入他们的选择。每一名游戏者输入 R 或 r 代表石头, P 或 p 代表布, S 或 s 代表剪刀。当第一名游戏者输入选择时, 另一名等待。当两名游戏者输入选择后并且选择有效的话, 程序输出游戏者的选择并且宣布游戏的胜者。游戏继续直到一名游戏者决定退出游戏。游戏结束后, 程序输出游戏次数和每一名游戏者获胜的次数。该讨论翻译成下面的算法:

1. 提供一个简单说明, 解释如何操作这一游戏。
2. 询问用户是否想开始这一游戏。
3. 开始游戏。
4. 如果游戏有效, 那么输出游戏和获胜者。
5. 更新总的游戏次数和获胜次数。

6. 当用户同意继续玩这个游戏时, 重复步骤 2~5。
7. 输出游戏的次数和每一名游戏者获胜的次数。

我们使用枚举类型来描述对象。

```
enum objectType{ Rock, Paper, Scissors};
```

变量 (main() 函数) 很明显, 在 main() 函数中需要下面的变量:

```
int    gameCount;    //to count the number of games played
int    winCount1;    //to count the number of games won by player 1
int    winCount2;    //to count the number of games won by player 2
int    gamewinner;   //to store the winner of a game
char   response;     //to get the user's response to play the game
char   selection1;   //player1's selection
char   selection2;   //player2's selection
objectType play1;    //player1's selection
objectType play2;    //player2's selection
```

程序包含下面的函数, 这将在后面的部分进行详细地讨论。

- **displayRules()** 函数显示一些关于游戏和游戏规则的简要信息。
- **validSelection()** 函数检查游戏者的选择是否有效。有效的选择只有 R, r, P, p, S, s。
- **retrievePlay()** 因为每局类型不能被直接读入, 所以函数转换输入的选择 (R, r, P, p, S, s) 并且返回适当的对象类型。
- **gameResult()** 函数输出游戏者的选择和游戏的获胜者。
- **convertEnum()** 函数被 gameResult() 函数调用来输出枚举类型值。
- **winningObject()** 函数确定并返回获胜对象。
- **displayResults()** 游戏结束后, 函数显示最后的结果。

函数 displayRules() 该函数没有参数。它只包含输出语句用来解释游戏和游戏规则。函数的定义是:

```
void displayRules()
{
    cout<<" Welcome to the game of Rock, Paper, and Scissors."<<endl;
    cout<<" This is a game for two players. For each game, each"<<endl;
    cout<<" player selects one of the objects, Rock, Paper, or"
        <<" Scissors."<<endl;
    cout<<" The rules for winning the game are: "<<endl;
    cout<<"1. If both players select the same object, it is a"
        <<" tie."<<endl;
    cout<<"2. Rock breaks Scissors: So the player who selects Rock"
        <<" wins."<<endl;
    cout<<"3. Paper covers Rock: So the player who selects Paper"
        <<" wins."<<endl;
    cout<<"4. Scissors cut Paper: So the player who selects Scissors"
        <<" wins."<<endl<<endl;
    cout<<"Enter R or r to select Rock, P or p to select Paper, and "
        <<"S or s to select Scissors."<<endl;
} //end displayRules
```

函数 validSelection() 该函数检查游戏者的选择是否有效。

```
if selection is 'R' or 'r' or 'S' or 's' or 'P' or 'p', then
```

```

    it is a valid selection;
    otherwise the selection is invalid.

```

下面使用 switch 语句来检查正确的选择。函数的定义是：

```

bool validSelection(char selection)
{
    switch(selection)
    {
        case 'R': case 'r':
        case 'S': case 's':
        case 'P': case 'p': return true;
        default: return false;
    }
} //end validSelection

```

函数 retrievePlay() 因为枚举类型不能直接读入，所以该函数转换输入选择 (R、r、P、p、S 或 s) 并且返回适当的对象类型。这样该函数只有一个 char 类型的参数。它是一个返回值函数并且返回一个 objectType 类型的值。使用伪代码，该函数的算法是：

```

if selection is 'R' or 'r'
    return Rock;
if selection is 'P' or 'p'
    return Paper;
if selection is 'S' or 's'
    return Scissors;

```

retrievePlay() 函数的定义是：

```

objectType retrievePlay(char selection)
{
    objectType object;
    switch(selection)
    {
        case 'R': case 'r': object = Rock;
                           break;
        case 'P': case 'p': object = Paper;
                           break;
        case 'S': case 's': object = Scissors;
    }
    return object;
} //end retrievePlay

```

函数 gameResult() 该函数判断游戏是平局或者判断哪名游戏者是获胜者。它输出游戏者的选择和游戏的获胜者。很明显，该函数有三个参数：游戏者1的选择，游戏者2的选择和返回获胜者的信息。用伪代码表示，该函数是：

```

a. if player1 and player2 have the same selection, then
    this is a tie game
b. else
    {
        1. Determine the winning object. (Call function winningObject())
        2. Output each player's choice.
        3. Determine the winning player.
        4. Return the winning player via a reference parameter to the
           function main so that the function main can update the
           winning player's win count.
    }

```


该函数的定义是:

```
void gameResult(objectType play1, objectType play2, int& winner)
{
    objectType winnerObject;
    if(play1 == play2)                                //Step a
    {
        winner = 0;
        cout<<"Both players selected ";
        convertEnum(play1);
        cout<<". This game is a tie."<<endl;
    }
    else                                              //Step b
    {
        winnerObject = winningObject(play1, play2);    //Step b1

        //Output each player's choice; Step b2
        cout<<"Player 1 selected ";
        convertEnum(play1);
        cout<<" and player 2 selected ";
        convertEnum(play2);
        cout<<". ";

        //Decide the winner; Step b3
        if(play1 == winnerObject)
            winner = 1;                                //Step b4
        else
            if(play2 == winnerObject)
                winner = 2;                            //Step b4

        //Output winner
        cout<<"Player "<<winner<<" wins this game."<<endl;
    }
} //end gameResult
```

函数 convertEnum() 因为枚举类型不能直接输出, 所以编写 convertEnum() 函数来输出 objectType 类型的对象。该函数有一个 objectType 类型的参数, 它输出与 objectType 相对应的字符串。用伪代码表示, 该函数是:

```
if object is Rock
    output "Rock"
if object is Paper
    output "Paper"
if object is Scissors
    output "Scissors"
```

函数 convertEnum() 的定义是:

```
void convertEnum(objectType object)
{
    switch(object)
    {
        case Rock: cout<<"Rock";
                    break;
        case Paper: cout<<"Paper";
                    break;
        case Scissors: cout<<"Scissors";
                    break;
    }
} //end convertEnum
```

函数 `winningObject()` 为了决定游戏的获胜者, 需要判断游戏者的选择, 然后查看游戏的规则。例如, 如果一名游戏者选择 Rock, 另一名选择 Paper, 那么选择 Paper 的游戏者将会获胜。换言之, 获胜对象是 Paper。根据给定的两个对象, 函数 `winningObject()` 确定并返回获胜对象。很清楚, 该函数有两个 `objectType` 类型的参数, 函数返回的值也是 `objectType` 类型的。该函数的定义如下:

```
objectType winningObject(objectType play1, objectType play2)
{
    if((play1 == Rock && play2 == Scissors)
        || (play2 == Rock && play1 == Scissors))
        return Rock;
    else
        if((play1 == Rock && play2 == Paper)
            || (play2 == Rock && play1 == Paper))
            return Paper;
        else
            return Scissors;
}
```

函数 `displayResults()` 游戏结束后, 函数输出最后的结果, 也就是游戏的次数和每一名游戏者获胜的次数。游戏的总次数存储在变量 `gameCount` 中, 游戏者 1 获胜的次数存储在变量 `winCount1` 中, 游戏者 2 获胜的次数存储在变量 `winCount2` 中。该函数有三个参数对应这些变量。

函数的定义是:

```
void displayResults(int gCount, int wCount1, int wCount2)
{
    cout<<"The total number of plays: "<<gCount<<endl;
    cout<<"The number of plays won by player 1: "<<wCount1<<endl;
    cout<<"The number of plays won by player 2: "<<wCount2<<endl;
} //end displayResults
```

现在开始编写 `main()` 函数的算法。

主要算法

1. 声明变量
2. 初始化变量
3. 显示游戏规则
4. 提示用户开始游戏
5. 得到用户的回答
6. while (yes)
 - {
 - a. 提示用户 1 进行选择
 - b. 游戏者 1 开始游戏
 - c. 提示用户 2 进行选择
 - d. 游戏者 2 开始游戏
 - e. 如果选择有效
 - {
 - i. 将总游戏次数加 1
 - ii. 宣布游戏的胜者

iii. 将获胜者的获胜次数加 1

}

f. 提示用户是否继续开始

g. 得到用户的回答

}

7. 输出游戏结果

完整程序清单

```
#include <iostream>

using namespace std;

enum objectType{Rock, Paper, Scissors};

//function prototypes
void displayRules();
objectType retrievePlay(char selection);
bool validSelection(char selection);
void convertEnum(objectType object);
objectType winningObject(objectType play1, objectType play2);
void gameResult(objectType play1, objectType play2, int& winner);
void displayResults(int gCount, int wCount1, int wCount2);

int main()
{
    //Step 1
    int    gameCount; //to count the number of games played
    int    winCount1; //to count the number of games won by player 1
    int    winCount2; //to count the number of games won by player 2
    int    gamewinner;
    char    response; //to get the user's response to play the game
    char    selection1;
    char    selection2;
    objectType    play1; //player1's selection
    objectType    play2; //player2's selection

    //Initialize variables; Step 2
    gameCount = 0;
    winCount1 = 0;
    winCount2 = 0;

    displayRules(); //Step 3

    cout<<"Enter Y/y to play the game: "; //Step 4
    cin>>response; //Step 5
    cout<<endl;

    while(response == 'Y' || response == 'y') //Step 6
    {
        cout<<"Player 1 enter your choice: "; //Step 6a
        cin>>selection1; //Step 6b
        cout<<endl;

        cout<<"Player 2 enter your choice: "; //Step 6c
        cin>>selection2; //Step 6d
        cout<<endl;

        //Step 6e
        if(validSelection(selection1) && validSelection(selection2))
        {
```

```

        play1 = retrievePlay(selection1);
        play2 = retrievePlay(selection2);
        gameCount++; //Step 6e.i
        gameResult(play1, play2, gamewinner); //Step 6e.ii
        if(gamewinner == 1) //Step 6e.iii
            winCount1++;
        else
            if(gamewinner == 2)
                winCount2++;
    } //end if
    cout<<"Enter Y/y to play the game: "; //Step 6f
    cin>>response; //Step 6g
    cout<<endl;
} //end while

displayResults(gameCount, winCount1, winCount2); //Step 7

return 0;
} //end main

void displayRules()
{
    cout<<" Welcome to the game of Rock, Paper, and Scissors."<<endl;
    cout<<" This is a game for two players. For each game, each"<<endl;
    cout<<" player selects one of the objects, Rock, Paper, or"
        <<" Scissors."<<endl;
    cout<<" The rules for winning the game are: "<<endl;
    cout<<"1. If both players select the same object, it is a"
        <<" tie."<<endl;
    cout<<"2. Rock breaks Scissors: So the player who selects Rock"
        <<" wins."<<endl;
    cout<<"3. Paper covers Rock: So the player who selects Paper"
        <<" wins."<<endl;
    cout<<"4. Scissors cut Paper: So the player who selects Scissors"
        <<" wins."<<endl<<endl;
    cout<<"Enter R or r to select Rock, P or p to select Paper, and "
        <<"S or s to select Scissors."<<endl;
}

bool validSelection(char selection)
{
    switch(selection)
    {
        case 'R': case 'r':
        case 'S': case 's':
        case 'P': case 'p': return true;
        default: return false;
    }
}

objectType retrievePlay(char selection)
{
    objectType object;
    switch(selection)
    {
        case 'R': case 'r': object = Rock;
                           break;
        case 'P': case 'p': object = Paper;
                           break;
        case 'S': case 's': object = Scissors;
    }
}

```

```
    }
    return object;
}

void convertEnum(objectType object)
{
    switch(object)
    {
        case Rock: cout<<"Rock";
                    break;
        case Paper: cout<<"Paper";
                    break;
        case Scissors: cout<<"Scissors";
    }
}

objectType winningObject(objectType play1, objectType play2)
{
    if((play1 == Rock && play2 == Scissors)
        || (play2 == Rock && play1 == Scissors))
        return Rock;
    else
        if((play1 == Rock && play2 == Paper)
            || (play2 == Rock && play1 == Paper))
            return Paper;
        else
            return Scissors;
}

void gameResult(objectType play1, objectType play2, int& winner)
{
    objectType winnerObject;
    if(play1 == play2)
    {
        winner = 0;
        cout<<"Both players selected ";
        convertEnum(play1);
        cout<<" . This game is a tie."<<endl;
    }
    else
    {
        winnerObject = winningObject(play1, play2);

        //Output each player's choice
        cout<<"Player 1 selected ";
        convertEnum(play1);
        cout<<" and player 2 selected ";
        convertEnum(play2);
        cout<<" . ";

        //Decide the winner
        if(play1 == winnerObject)
            winner = 1;
        else
            if(play2 == winnerObject)
                winner = 2;

        //Output the winner
        cout<<"Player "<<winner<<" wins this game."<<endl;
    }
}
```

```

}

void displayResults(int gCount, int wCount1, int wCount2)
{
    cout<<"The total number of plays: "<<gCount<<endl;
    cout<<"The number of plays won by player 1: "<<wCount1<<endl;
    cout<<"The number of plays won by player 2: "<<wCount2<<endl;
}

```

8.3 命名空间

在1998年6月, ANSI/ISO 标准 C++ 正式通过。大多数现代编译器与 ANSI/ISO 标准 C++ 是兼容的。(为了完全确认这一点, 请检查编译器文档。)两种标准实际上是相同的。ANSI/ISO 标准 C++ 语言有一些在 C++ 中不可用的特征, 本章的剩余部分将会对此进行讲述。在后面的章节中, 除非特别指定, 否则 C++ 语法可以应用于这两种标准。首先讨论 ANSI/ISO 标准 C++ 的命名空间机制。

当一个头文件, 例如 `iostream` 包含在一个程序中时, 头文件中的全局标识符也变成程序中的全局标识符。所以, 如果程序中的一个全局标识符与头文件中的全局标识符有同样的名字, 那么编译器会产生语法错误(例如 "identifier redefined")。如果程序使用第三方的库, 也会产生同样的问题。为了克服这一问题, 第三方厂家用一个特殊的符号来开始其全局标识符。在第2章中了解到, 因为编译器厂家使用一个下划线来开始他们的全局标识符, 为了避免连接错误, 在程序中不应该以下划线开始标识符名字。

ANSI/ISO 标准 C++ 试图使用命名空间机制来解决全局标识符的重叠问题。

语句 `namespace` 通常的语法是:

```

namespace namespace_name
{
    members
}

```

其中 `members` 通常是一个常量、变量声明、函数或另一个 `namespace`。注意 `namespace_name` 是一个 C++ 标识符。

在 C++ 中, `namespace` 是一个保留字。

例 8.8

语句:

```

namespace globalType
{
    const int n = 10;
    const double rate = 7.50;
    int count = 0;
    void printResult();
}

```

定义 `globalType` 是一个有 4 个成员的 `namespace`: 常量 `n` 和 `rate`、变量 `count` 和函数 `printResult()`。

`namespace` 成员的范围局限于 `namespace` 中。通常有两种方法访问 `namespace` 中的 `namespace` 成员, 如下面的阐述。

访问 `namespace` 成员的通常语法是:

```

namespace_name::identifier

```


例如, 为了访问 namespace `globalType` 的成员 `rate`, 需要下面的语句:

```
globalType::rate
```

为了访问成员 `printResult()`, 需要下面的语句:

```
globalType::printResult( );
```

前面讲过, 在 C++ 中 `::` 称为范围解析运算符。这样, 为了访问 namespace 中的一个成员, 要先使用 `namespace_name`, 然后是范围解析运算符, 最后是成员名字。也就是说, 要在成员名字前放置 `namespace_name` 的名字和范围解析运算符。

为了简化访问 namespace 成员, ANSI/ISO 标准 C++ 提供了 `using` 语句。使用语句 `using` 的语法如下:

(a) 为了简化访问所有的 namespace 变量:

```
using namespace namespace_name;
```

(b) 为了简化访问特定的 namespace 成员:

```
using namespace_name::identifier;
```

例如, `using` 语句:

```
using namespace globalType;
```

简化了对 namespace 中的 `globalType` 所有成员的访问。语句:

```
using globalType::rate;
```

简化了对 namespace `globalType` 成员 `rate` 的访问。

在 C++ 中, `using` 是一个保留字。

典型的用法是将 `using` 语句放在 namespace 声明后。例如, 对 namespace `globalType` 来说, 通常编写下面的代码:

```
namespace globalType
{
    const int n = 10;
    const double rate = 7.50;
    int count = 0;
    void printResult();
}

using namespace globalType;
```

在 `using` 语句后, 为了访问 namespace 成员, 不必将 `namespace_name` 和范围运算符放在 namespace 成员前。然而, 如果程序中 namespace 成员和全局运算符有相同的名字, 为了访问程序中的 namespace 成员, `namespace_name` 和范围解析运算符必须置于 namespace 成员前。同样, 如果 namespace 成员和一个块中的标识符有相同的名字, 为了访问块中的 namespace 成员, `namespace_name` 和范围解析运算符必须置于 namespace 成员前。

例 8.9

考虑下面的 C++ 代码:

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

```

.
.
.
int main()
{
    .
    .
    .
}
.
.
.

```

在这个例子中，可以参考头文件 `iostream` 中的全局标识符，例如 `cin`、`cout` 和 `endl`，不必在标识符名字前使用前缀 `std::`。明显的限制是：使用全局运算符（头文件 `iostream` 的）的块（或函数）不允许包含与全局标识符有相同名字的任何标识符。

例 8.10

考虑下面的 C++ 代码：

```

#include <cmath>

int main()
{
    double x = 15.3;
    double y;

    y = std::pow(x, 2);
    .
    .
    .
}

```

该例子访问头文件 `cmath` 中的函数 `pow()`。

例 8.11

考虑下面的 C++ 代码：

```

#include <iostream>
.
.
.
int main()
{
    using namespace std;
    .
    .
    .
}
.
.
.

```

在这个例子中，`main()` 函数引用头文件 `iostream` 中的全局标识符，而不必在标识符名字前使用前缀 `std::`。`using` 语句出现在 `main()` 函数中。所以，其他函数（如果有的话）应该在头文件 `iostream` 的全局标识符名字前使用前缀 `std::`，除非函数有一条类似的 `using` 语句。

例 8.12

考虑下面的 C++ 代码：

```
#include <iostream>
using namespace std;                                //Line 1

int t;                                                //Line 2
double u;                                            //Line 3

namespace exp
{
    int x;                                           //Line 4
    char t;                                         //Line 5
    double u;                                       //Line 6
    void printResult();                             //Line 7
}

using namespace exp;

int main()
{
    int one;                                         //Line 8
    double t;                                       //Line 9
    double three;                                   //Line 10

    .
    .
    .
}

void exp::printResult() //Definition of the function printResult()
{
    .
    .
    .
}
```

该 C++ 程序中：

1. 为了引用 main() 中的变量 t，可以使用范围解析运算符（即将 t 作为 ::t），因为函数 main() 存在名字为 t 的变量（第 5 行声明）。
2. 为了引用 main() 中 namespace exp 的成员 t，应该将 t 与前缀 exp::（即将 t 看做 exp::t）一起使用，这是因为 main() 中有一个名为 t（第 2 行声明）的全局变量和一个名为 t 的变量。
3. 为了引用 main() 中 namespace exp 的成员 u，将 u 与前缀 exp::（即将 t 看做 exp::u）一起使用，这是因为 main() 中有一个名为 u（第 3 行声明）的全局变量。
4. 可以采用 x 或 exp::x 的方式引用 main() 中 namespace exp 的成员 x（第 4 行声明），这是因为没有名字为 x 的全局标识符，并且函数 main() 没有包含任何名字为 x 的标识符。
5. main() 中 namespace exp 的函数，例如 printResult()，其定义通常编写在程序中的 namespace 之外。为了编写函数 printResult() 的定义，函数头中函数的名字可以是 printResult() 或 exp::printResult()（因为没有其他名为 printResult() 的全局变量）。

注意：系统提供的头文件中的标识符，例如 `iostream`、`cmath` 和 `iomanip` 是在 namespace `std` 中定义的。由于这一原因，为了简化从这些头文件中对标识符的访问，我们在编写的程序中一直使用下面的语句：

```
using namespace std;
```

8.4 string 类型

在第2章中介绍了数据类型 string。前面讲过,在 ANSI/ISO 标准前,标准 C++ 库没有提供 string 数据类型。编译器厂商经常提供自己的程序员定义的 string 类型和语法,而且一家厂商运算符的语义与另一家厂商运算符的语义通常是不相同的。

string 数据类型是一种程序员自定义的类型,它不是 C++ 语言的一部分,但是 C++ 标准库支持它。在使用 string 数据类型前,程序必须包含头文件 string, 如下所示:

```
#include<string>
```

前面讲过,在 C++ 中,字符串是零或多个字符的序列,字符串包含在双引号中。

语句:

```
string name="William Jacob";
```

声明 name 是一个 string 变量并且将 name 初始化为 "William Jacob"。在 name 中的第一个字符 W 的位置是 0,第 2 个字符 i 的位置是 1,等等。即 string 变量中字符的位置从零开始,而不是 1。

变量 name 几乎能存储任何大小的字符串。

在第3章中讨论了对 string 类型的 I/O 操作;在第4章中解释了对 string 类型的关系操作。推荐读者重新复习一下第3章和第4章,以及对 string 类型的 I/O 和关系操作。

string 数据类型还定义了其他运算符,例如二元运算符 + (允许字符串的连接操作)和下标运算符 []。现在看一下这些运算符是如何对 string 数据类型进行操作的。

假设有下面的声明:

```
string str1,str2,str3;
```

语句:

```
str1="Hello There";
```

将字符串 "Hello There" 存储到 str1 中。语句:

```
str2=str1;
```

将 str1 的值复制给 str2。

如果 str1="sunny", 那么语句:

```
str2=str1+" Day";
```

将字符串 "sunny Day" 存储到 str2 中。

假设 str1="Hello", str2="There", 那么语句:

```
str3=str1+" "+str2;
```

将 "Hello There" 存储到 str3 中。该语句等价于下面的语句:

```
str3=str1+' '+str2;
```

而且,语句:

```
str1=str1+" Mickey";
```

通过向原来的值添加 "Mickey" 从而更新了 str1 的值。所以 str1 的新值是 "Hello Mickey"。

注意：对于与 string 数据类型一起使用的 + 运算符来说，其中一个操作数必须是一个字符串变量。例如，下面的语句将不会有效：

```
str1="Hello"+"there!";      //Illegal
str2="sunny Day"+"!";       //Illegal
```

如果 str1="Hello there"，那么语句：

```
str1[6]='T';
```

用字符 T 替换了原来的字符 t。前面讲过，在 string 变量中第一个字符的位置是 0。这样，因为 t 是 str1 中的第 7 个字符，所以它的位置是 6。

在 C++ 中，[] 称为数组下标运算符。

正如前面所演示的，将下标运算符与字符的位置一起使用，可以访问字符串中的单个字符。

例 8.13

下面的程序显示了上述语句的功能。

```
//Example string operations
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name = "William Jacob";           //Line 1
    string str1, str2, str3, str4;           //Line 2

    cout<<"Line 3: Name = "<<name<<endl;     //Line 3

    str1 = "Hello There";                    //Line 4
    cout<<"Line 5: str1 = "<<str1<<endl;      //Line 5

    str2 = str1;                             //Line 6
    cout<<"Line 7: str2 = "<<str2<<endl;      //Line 7

    str1 = "Sunny";                          //Line 8
    str2 = str1 + " Day";                     //Line 9
    cout<<"Line 10: str2 = "<<str2<<endl;     //Line 10

    str1 = "Hello";                          //Line 11
    str2 = "There";                          //Line 12
    str3 = str1 + " " + str2;                 //Line 13
    cout<<"Line 14: str3 = "<<str3<<endl;     //Line 14

    str3 = str1 + ' ' + str2;                 //Line 15
    cout<<"Line 16: str3 = "<<str3<<endl;     //Line 16

    str1 = str1 + " Mickey";                  //Line 17
    cout<<"Line 18: str1 = "<<str1<<endl;     //Line 18

    str1 = "Hello there";                    //Line 19
    cout<<"Line 20: str1[6] = "<<str1[6]<<endl; //Line 20

    str1[6] = 'T';                          //Line 21
    cout<<"Line 22: str1 = "<<str1<<endl;     //Line 22
    //string input operations
    cout<<"Line 23: Enter a string with "
        <<"no blanks: ";                    //Line 23
```

```

        cin>>str1;                                //Line 24
        char ch;                                  //Line 25
        cin.get(ch);                               //read the newline character;   Line 26
        cout<<endl;                                //Line 27

        cout<<"Line 28: The string you entered = "<<str1
            <<endl;                                //Line 28
        cout<<"Line 29: Enter a sentence: ";        //Line 29
        getline(cin,str2);                          //Line 30
        cout<<endl;                                //Line 31
        cout<<"Line 32: The sentence is: "<<str2<<endl; //Line 32

        return 0;
    }

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Line 3: Name = William Jacob
Line 5: str1 = Hello There
Line 7: str2 = Hello There
Line 10: str2 = Sunny Day
Line 14: str3 = Hello There
Line 16: str3 = Hello There
Line 18: str1 = Hello Mickey
Line 20: str1[6] = t
Line 22: str1 = Hello There
Line 23: Enter a string with no blanks: Programming
Line 28: The string you entered = Programming
Line 29: Enter a sentence: Testing string operations
Line 32: The sentence is: Testing string operations

```

在为第 29 行输入数据后，必须按两次 Enter 键。前面讲过 `getline()` 函数也读入换行符。

8.4.1 其他 string 类型的数据操作

数据类型 `string` 包含一些其他字符串操作函数。其中，`length()`、`size()`、`find()`、`substr()` 和 `swap()` 将在下面进行描述。

`string` 数据类型包含一种数据类型 `string::size_type` 和一个与它相联系的常量 `string::npos`。

`string::size_type` 一个无符号整数（数据）类型

`string::npos` （数据）类型 `string::size_type` 的最大值，例如很多计算机的最大数字是 4294967295

`length()` 函数

`length()` 函数返回当前字符串中的字符数量，该返回值是一个无符号整数。调用 `length()` 函数的语法是：

```
strVar.length();
```

其中 `strVar` 是一个 `string` 类型的变量。`length()` 函数没有参数。

注意 `length()` 函数的语法。`strVar` 与 `length` 之间的点号是非常重要的，它将字符串变量的名字与单词 `length` 分开。而且，因为 `length()` 是一个没有参数的函数，所以仍然需要括号。又因为 `length()` 是一个返回值函数，所以函数调用必须出现在一个表达式中。

考虑下面的语句：

string firstName;	
string name;	
string str;	
firstName = "Elizabeth";	
name = firstName + " Jones";	
str = "It is sunny.";	
语句	输出结果
cout<<firstName.length()<<endl;	9
cout<<name.length()<<endl;	15
cout<<str.length()<<endl;	12

函数length()返回一个无符号整数,所以返回值可以存储在整型变量中。因为数据类型string在与它联系的数据类型string::size_type,所以存储length()函数返回值的变量通常是这种类型。所以不必猜测该返回值是无符号int还是无符号long。

假设有前面的声明和语句:

string::size_type len;	
语句	输出结果
len = firstName.length();	9
len = name.length();	15
len = str.length();	12

例 8.14

下面的程序演示了length()函数的使用:

```
//Example length function
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name, firstName; //Line 1
    string str; //Line 2
    string::size_type len; //Line 3

    firstName = "Elizabeth"; //Line 4
    name = firstName + " Jones"; //Line 5
    str = "It is sunny and warm."; //Line 6

    cout<<"Line 7: Length of \""<<firstName //Line 7
         <<"\" = "<<firstName.length()<<endl;

    cout<<"Line 8: Length of \""<<name //Line 8
         <<"\" = "<<name.length()<<endl;

    cout<<"Line 9: Length of \""<<str //Line 9
         <<"\" = "<<str.length()<<endl;

    len = firstName.length(); //Line 10
    cout<<"Line 11: len = "<<len<<endl; //Line 11

    len = name.length(); //Line 12
    cout<<"Line 13: len = "<<len<<endl; //Line 13

    len = str.length(); //Line 14
    cout<<"Line 15: len = "<<len<<endl; //Line 15
}
```

```
    return 0;
}
```

输出:

```
Line 7: Length of "Elizabeth" = 9
Line 8: Length of "Elizabeth Jones" = 15
Line 9: Length of "It is sunny and warm." = 21
Line 11: len = 9
Line 13: len = 15
Line 15: len = 21
```

size() 函数

有些人愿意使用单词 size 而不愿意使用单词 length。这样,为了适用这两种术语, string 类型提供了另一个名为 size() 的函数,它返回的值与 length() 返回的值是相同的。调用 size() 函数的语法是:

```
strVar.size();
```

其中 strVar 是 string 类型的变量。与函数 length() 一样,函数 size() 没有参数。

find() 函数

find() 函数搜索字符串并寻找与特定子字符串相匹配时第一次出现的位置,并且返回一个无符号整数值 (string::size_type 类型) 来给出搜索的结果。调用函数 find() 的语法是:

```
strVar.find(strExp);
```

其中 strVar 是一个字符串变量, strExp 是一个字符串表达式。字符串表达式 strExp 也可以是一个字符。如果搜索成功,那么函数 find() 返回在匹配字符串的开始位置。对于成功的搜索来说,匹配必须是准确的。如果搜索不成功,那么函数返回特殊值 string::npos ("not a position within the string")。(该值不代表“不是一个有效的位置”,因为字符串操作不会让任何字符串变为这一长度。)因为函数 find() 返回一个无符号整数,所以返回值能够存储在一个整数变量中(通常是 string::size_type 类型)。

假设 str1 和 str2 是一个 string 类型。下面的语句对于 find() 函数来说是有效的调用:

```
str1.find(str2)
str1.find("the")
str1.find('a')
str1.find(str2 + "xyz")
str1.find(str2 + 'b')
```

考虑下面的语句:

```
string sentence;
string str;
string::size_type position;

sentence = "It is cloudy and warm.";
str = "cloudy";
```

语句

cout<<sentence.find("is")<<endl;	3
cout<<sentence.find("and")<<endl;	13
cout<<sentence.find('s')<<endl;	4
cout<<sentence.find('i')<<endl;	3
cout<<sentence.find(str)<<endl;	6
cout<<sentence.find("the")<<endl;	输出 string::npos
position = sentence.find("warm");	将 17 赋与 position

注意搜索是区分大小写的。所以字符串 `sentence` 中 `i` 的位置 (小写字母) 是 3。

例 8.15

下面的程序演示了如何使用字符串函数 `find()`。

```
//Example find function
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string    sentence, str;                                //Line 1
    string::size_type position;                             //Line 2
    sentence = "It is cloudy and warm.";                    //Line 3
    str = "cloudy";                                          //Line 4
    cout<<"Line 5: Position of \"is\" in \""<<sentence
        <<"\" = "<<sentence.find("is")<<endl;              //Line 5
    cout<<"Line 6: Position of \"and\" in \""<<sentence
        <<"\" = "<<sentence.find("and")<<endl;              //Line 6
    cout<<"Line 7: Position of 's' in \""
        <<sentence<<"\" = "<<sentence.find('s')<<endl;      //Line 7
    cout<<"Line 8: Position of 'i' in \""
        <<sentence<<"\" = "<<sentence.find('i')<<endl      //Line 8
    cout<<"Line 9: Position of \""<<str<<"\" in \""
        <<sentence<<"\" = "<<sentence.find(str)<<endl;      //Line 9
    cout<<"Line 10: Position of \"the\" in \""<<sentence
        <<"\" = "<<sentence.find("the")<<endl;             //Line 10
    position = sentence.find("warm");                       //Line 11
    cout<<"Line 12: \"<<"Position = "<<position<<endl;     //Line 12
    return 0;
}
```

输出:

```
Line 5: Position of "is" in "It is cloudy and warm." = 3
Line 6: Position of "and" in "It is cloudy and warm." = 13
Line 7: Position of 's' in "It is cloudy and warm." = 4
Line 8: Position of 'i' in "It is cloudy and warm." = 3
Line 9: Position of "cloudy" in "It is cloudy and warm." = 6
Line 10: Position of "the" in "It is cloudy and warm." = 4294967295
Line 12: Position = 17
```

substr() 函数

`substr()` 函数返回一个字符串的特定子串。调用函数 `substr()` 的语法是:

```
strVar.substr(expr1, expr2);
```

其中 `expr1` 和 `expr2` 是值为无符号整数的表达式。表达式 `expr1` 指定了字符串中的一个位置 (子串的开始位置); 表达式 `expr2` 指定了返回的子串长度。

考虑下面的语句:

<pre> string sentence; string str; sentence = "It is cloudy and warm."; 语句 cout<<sentence.substr(0,5)<<endl; cout<<sentence.substr(6,6)<<endl; cout<<sentence.substr(6,16)<<endl; cout<<sentence.substr(3,6)<<endl; str = sentence.substr(0,8); str = sentence.substr(2,10); </pre>	<pre> 输出结果 Outputs: It is Outputs: cloudy Outputs: cloudy and warm. Outputs: is clo str = "It is cl" str = " is cloudy" </pre>

例 8.16

下面的程序演示了如何使用 string 函数 substr()。

```

//Example substr function
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string sentence;                                     //Line 1
    string str;                                           //Line 2

    sentence = "It is cloudy and warm.";                 //Line 3

    cout<<"Line 4: substr(0,5) in \""<<sentence
        <<"\" = \""<<sentence.substr(0,5)<<"\"<<endl;    //Line 4

    cout<<"Line 5: substr(6,6) in \""<<sentence
        <<"\" = \""<<sentence.substr(6,6)<<"\"<<endl;    //Line 5

    cout<<"Line 6: substr(6,16) in \""<<sentence
        <<"\" = \""<<endl<<"\"
        <<sentence.substr(6,16)<<"\"<<endl;            //Line 6

    cout<<"Line 7: substr(3,6) in \""<<sentence
        <<"\" = \""<<sentence.substr(3,6)<<"\"<<endl;    //Line 7

    str = sentence.substr(0,8);                           //Line 8
    cout<<"Line 9: \"<<"str = \""<<str<<"\"<<endl;        //Line 9

    str = sentence.substr(2,10);                          //Line 10
    cout<<"Line 11: \"<<"str = \""<<str<<"\"<<endl;      //Line 11

    return 0;
}

```

输出:

```

Line 4: substr(0,5) in "It is cloudy and warm." = "It is"
Line 5: substr(6,6) in "It is cloudy and warm." = "cloudy"
Line 6: substr(6,16) in "It is cloudy and warm." =
      "cloudy and warm."
Line 7: substr(3,6) in "It is cloudy and warm." = "is clo"
Line 9: str = "It is cl"
Line 11: str = " is cloudy"

```

swap() 函数

swap() 函数用来相互交换两个字符串变量的内容。使用 swap() 函数的语法是:


```
strVar1.swap(strVar2);
```

其中 strVar1 和 strVar2 是 string 变量。在语句执行后, strVar1 和 strVar2 的内容被交换。假设有下面的语句:

```
string str1 = "Warm";  
string str2 = "Cold";
```

在下面的语句执行后, str1 的值是 "Cold", str2 的值是 "Warm"。

```
str1.swap(str2);
```

注意: 其他 string 函数, 例如 empty()、clear()、erase()、insert() 和 replace() 在附录 F (头文件 string) 中提供。

8.5 编程实例: Pig Latin 字符串

在该编程实例中, 提示用户输入一个字符串, 然后以 Pig Latin 的形式输出该字符串。其规则如下:

1. 如果字符串以元音开始, 则在字符串的末尾添加字符串 "-way"。例如, 字符串 "eye" 的 Pig Latin 形式是 "eye-way"。
2. 如果字符串不是以元音开始, 那么首先在字符串的末尾添加 "-", 然后旋转字符串一个字符, 即将字符串的第一个字符移到字符串的末尾, 直到字符串的第一个字符是一个元音字母。然后在结尾添加字符串 "ay"。例如, 字符串 "There" 的 Pig Latin 形式是 "ere-Thay"。
3. 有些字符串, 例如 "by" 不包含元音。对于这种情况, 字母 y 可以被当做元音。所以, 对于该程序来说, 元音字母是 a、e、i、o、u、y、A、E、I、O、U 和 Y。这样, "by" 的 Pig Latin 形式是 "y-bay"。
4. 有些字符串不包含元音字母。例如, 字符串 "1234" 的 Pig Latin 形式是 "1234-way"。也就是说, 没有元音字母的字符串的 Pig Latin 形式是在字符串的结尾加上 "-way"。

输入 程序的输入是一个字符串。

输出 程序的输出是 Pig Latin 形式的字符串。

问题分析和算法设计

假设 str 代表一个字符串。为了将 str 转换为 Pig Latin 形式, 应该检查 str 的第一个字符 str[0]。如果 str[0] 是一个元音字母, 那么在 str 的末尾添加 "-way", 也就是 $str = str + "-way"$ 。

假设 str 的第一个字符 str[0] 不是一个元音字母。首先在字符串的末尾添加 "-"。然后从 str 中删除第一个字符并把它放在 str 的末尾。现在 str 的第二个字符变成了 str 的第一个字符。如果 str 的第一个字符不是一个元音字母, 那么检查 str 第一个字母并将它移到 str 末尾, 这一过程将会重复执行, 直到 str 的第一个字母是一个元音字母或者 str 的所有字符都被处理过 (在这种情况下, str 不包含任何元音字母)。

在该程序中, 编写一个 isVowel() 函数, 来判断一个字符是否是一个元音字母; 一个 rotate() 函数, 将 str 的第一个字符移到 str 的末尾; 一个 pigLatinString() 函数, 求出 str 的 Pig Latin 形式。前面的讨论翻译成下面的算法:

1. 获得 str。
2. 使用函数 pigLatinString() 找到 str 的 Pig Latin 形式。

3. 输出 str 的 Pig Latin 形式。

函数 `isVowel()` 该函数将一个字符作为参数, 如果该字符是一个元音字母则返回 `true`, 否则返回 `false`。函数 `isVowel()` 的定义是:

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

函数 `rotate()` 该函数将一个字符串作为参数, 删除字符串的第一个字符并把它放在字符串的末尾。这可以通过提取位置从 1 (它是第二个字符) 到字符串末尾的子串来完成, 然后在字符串结尾添加字符串的第一个字符。新的字符串作为该函数的返回值。函数 `rotate()` 的定义是:

```
string rotate(string pStr)
{
    int len = pStr.length();
    string rStr;
    rStr = pStr.substr(1, len - 1) + pStr[0];
    return rStr;
}
```

函数 `pigLatinString()` 该函数将字符串 `pStr` 作为参数并且返回 `pStr` 的 Pig Latin 形式。假设 `pStr` 代表被转换为 Pig Latin 形式的字符串。有三种可能的情况: `pStr[0]` 是元音字母; `pStr` 包含元音字母, 但是 `pStr` 的第一个字符不是元音字母; `pStr` 不包含元音字母。假设 `pStr[0]` 不是一个元音字母, 将 `pStr` 的第一个字符移到 `pStr` 的末尾。该过程一直重复, 直到 `pStr` 的第一个字符变成一个元音字母, 或者所有 `pStr` 的字符都被检查过 (这种情况下 `pStr` 不包含任何元音字母。)。该讨论转化为下面的算法:

1. 如果 `pStr[0]` 是元音字母, 那么在 `pStr` 的末尾添加 "-way"。
2. 假设 `pStr[0]` 不是元音字母。
3. 将 `pStr` 的第一个字符移到 `pStr` 的末尾。 `pStr` 的第二个字符变成 `pStr` 的第一个字符。现在 `pStr` 可能包含, 也可能不包含元音字母。可以使用一个布尔变量 `foundVowel` 作为标志, 如果 `pStr` 包含一个元音字母, 那么它就为 `true`, 否则为 `false`。
 - a. 假设 `len` 代表了 `pStr` 的长度。
 - b. 将 `foundVowel` 初始化为 `false`。
 - c. 如果 `pStr[0]` 不是一个元音字母, 那么调用函数 `rotate()` 将 `pStr[0]` 移到 `pStr` 的末尾。
 - d. 重复步骤 b, 直到 `pStr` 的第一个字符成为元音字母, 或者所有 `pStr` 的字符都被检查过。
4. 将 `pStr` 转换为 Pig Latin 形式。
5. 返回 `pStr`。

`pigLatinString()` 函数的定义是:

```

string pigLatinString(string pStr)
{
    int len;
    bool foundVowel;
    int counter;
    if(isVowel(pStr[0])) //Step 1
        pStr = pStr + "-way";
    else //Step 2
    {
        pStr = pStr + '-';
        pStr = rotate(pStr); //Step 3
        len = pStr.length(); //Step 3.a
        foundVowel = false; //Step 3.b
        for(counter = 1; counter < len - 1; counter++) //Step 3.d
            if(isVowel(pStr[counter]))
            {
                foundVowel = true;
                break;
            }
        else //Step 3.c
            pStr = rotate(pStr);
        if(!foundVowel) //Step 4
            pStr = pStr.substr(1, len) + "-way";
        else
            pStr = pStr + "ay";
    }
    return pStr; //Step 5
}

```

主要算法

1. 获得字符串。
2. 调用函数 pigLatinString() 得到字符串的 Pig Latin 形式。
3. 输出字符串的 Pig Latin 形式。

完整程序清单

```

#include <iostream>
#include <string>

using namespace std;

bool isVowel(char ch);
string rotate(string pStr);
string pigLatinString(string pStr);

int main()
{
    string str;

    cout<<"Enter a string: ";
    cin>>str;
    cout<<endl;

    cout<<"The pig Latin form of "<<str<<" is: "
        <<pigLatinString(str)<<endl;

    return 0;
}

```

```
}

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

string rotate(string pStr)
{
    int len = pStr.length();
    string rStr;
    rStr = pStr.substr(1, len - 1) + pStr[0];
    return rStr;
}

string pigLatinString(string pStr)
{
    int len;
    bool foundVowel;
    int counter;
    if(isVowel(pStr[0])) //Step 1
        pStr = pStr + "-way";
    else //Step 2
    {
        pStr = pStr + '-';
        pStr = rotate(pStr); //Step 3
        len = pStr.length(); //Step 3.a
        foundVowel = false; //Step 3.b
        for(counter = 1; counter < len - 1; counter++) //Step 3.d
        {
            if(isVowel(pStr[0]))
            {
                foundVowel = true;
                break;
            }
            else //Step 3.c
                pStr = rotate(pStr);
        }
        if(!foundVowel) //Step 4
            pStr = pStr.substr(1, len) + "-way";
        else
            pStr = pStr + "ay";
    }
    return pStr; //Step 5
}
```

运行实例：在本运行实例中，用户输入用阴影表示。

运行实例 1:

Enter a string: eye

The pig Latin form of eye is: eye-way

运行实例 2:

Enter a string: There

The pig Latin form of There is: ere-Thay

运行实例 3:

Enter a string: why

The pig Latin form of why is: y-whay

运行实例 4:

Enter a string: 123456

The pig Latin form of 123456 is: 123456-way

8.6 小结

1. 枚举类型是一个顺序值的集合。
2. C++ 的保留字 `enum` 用来创建枚举类型。
3. `enum` 的语法是:
`enum typeName{ value1,value2,...};`
其中 `value1`、`value2` 是标识符并且 `value1 < value2 < ...`
4. 不允许对枚举类型进行算术操作。
5. 关系运算符可以与 `enum` 值一起使用。
6. 枚举类型值不能直接输入或输出。
7. 可以通过值或者引用将枚举类型作为参数传递给函数。
8. 函数可以返回枚举类型的值。
9. 变量值被指定, 但没有名字的类型称为匿名类型。
10. C++ 的保留字 `typedef` 用来定义数据类型的同义字或别名。
11. 匿名类型不能作为参数传递给函数。
12. `namespace` 机制是 ANSI/ISO 标准 C++ 的特征。
13. `namespace` 成员通常是常量、变量、函数或另一个 `namespace`。
14. `namespace` 成员对于 `namespace` 来说是局部的。
15. 访问 `namespace` 外的 `namespace` 成员的一种方法是在 `namespace` 成员名字前加上 `namespace` 名字和范围解析运算符。
16. 在 C++ 中, `namespace` 是一个保留字。
17. 为了使用 `namespace` 机制, 程序必须包含 ANSI/ISO 标准 C++ 头文件, 也就是没有扩展名为 `h` 的头文件。
18. `using` 语句简化了对 `namespace` 成员的访问。
19. 在 C++ 中, `using` 是一个保留字。
20. 关键字 `namespace` 不许出现在 `using` 语句中。
21. 不使用 `using` 语句访问 `namespace` 成员时, `namespace` 名字和范围解析运算符必须在 `namespace` 成员之前。
22. 为了使用一个在没有 `namespace` 名字的标准头文件中声明的标识符, 在包含了必要的头文件

之后,下面的语句必须出现在程序中:

```
using namespace std;
```

23. 字符串是零或多个字符的序列。
24. C++ 中的字符串用双引号括起来。
25. 为了使用 string 类型,程序必须包含头文件 string。程序中使用的其他头文件必须是 ANSI/ISO 标准 C++ 风格的头文件。
26. 赋值运算符能与 string 类型一起使用。
27. 运算符 + 能用来连接两个 string 类型的值。
28. 关系运算符可应用于 string 类型。
29. 在一个字符串中,第一个字符的位置是 0,第二个字符的位置是 1,等等。
30. 字符串的长度是字符串中字符的个数。
31. 在 C++ 中,[] 称为下标运算符。
32. 为了访问字符串中一个单独的字符,可以将下标运算符与字符位置一起使用。
33. 函数 length() 返回当前字符串中字符的个数。调用 length() 函数的语法是:

```
strVar.length();
```

其中 strVar 是 string 类型的变量。
34. 函数 size() 返回当前字符串中字符的个数。调用 size() 函数的语法是:

```
strVar.size();
```

其中 strVar 是 string 类型的变量。函数 size() 与函数 length() 的作用是一样的。
35. 函数 find() 搜索字符串并寻找与特定字符串相匹配时第一次出现的位置,并且返回一个无符号整数值 (string::size_type 类型)。调用函数 find() 的语法是:

```
strVar.find(strExp)
```

其中 strVar 是字符串变量, strExp 是一个字符串表达式。
36. 函数 find() 的参数 (即 strExp) 也可以是一个字符。
37. 如果搜索成功,那么函数 find() 返回在 strVar 中匹配字符串的开始位置。如果搜索不成功,那么函数返回 npos 值。
38. substr() 函数返回字符串的特定子串。调用函数 substr() 的语法是:

```
strVar.substr(expr1, expr2);
```

其中 expr1 和 expr2 是值为无符号整数的表达式。表达式 expr1 指定了字符串中的一个位置 (子串的开始位置); 表达式 expr2 指定了要返回的子串长度。
39. swap() 函数用来交换两个字符串变量的内容。使用 swap() 函数的语法是:

```
strVar1.swap(strVar2);
```

其中 strVar1 和 strVar2 是 string 变量。该语句将 strVar1 和 strVar2 的内容交换。

8.7 练习

1. 判断下面语句的对错。

a. 下面的语句是正确的 C++ 枚举类型:

```
enum romanNumerals{ I, V, X, L, C, D, M };
```

b. 给定声明:

```
enum cars{ Ford, GM, Toyota, Honda };  
cars domesticCars = Ford;
```

则语句:


```
domesticCars = domesticCars + 1;
```

将 domesticCars 的值设置为 GM。

- c. 函数可以返回一个枚举类型的值。
- d. 可以从标准输入设备直接输入枚举类型的值。
- e. 对枚举类型惟一允许的算术操作是自加和自减。
- f. 枚举类型域中的值称为枚举数。
- g. 在相同的 C++ 程序块中, 下面的语句是合法的:

```
enum mathStudent{ Bill, John, Lisa, Ron, Cindy, Shelly };
enum historyStudent{ Amanda, Bob, Jack, Tom, Susan };
```

- h. 下面的语句创建了一个匿名类型:

```
enum { A, B, C, D, F } studentGrade;
```

- i. 可以将扩展名为 h 的头文件与 namespace 机制一起使用。
- j. 假设 str="ABCD", 在语句 "str[1]='a';" 之后, str 的值变为 "aBCD"。
- k. 假设 str="abcd", 在语句 "str=str+"ABCD";" 之后, str 的值变为 "ABCD"。

2. 编写 C++ 语句完成下面的要求:

- a. 定义枚举类型 bookType, 它的值为 Math、CSC、English、History、Physics 和 Philosophy。
- b. 声明 bookType 类型的变量 book。
- c. 将 Math 赋值给变量 book。
- d. 将 book 的值赋值为列表中的下一个值。
- e. 输出变量 book 的值。

3. 给定:

```
enum currencyType{ Dollar, Pound, Frank, Lira, Mark };
currencyType currency;
```

下面哪项语句是有效的?

- a. currency = Dollar;
- b. cin>>currency;
- c. currency = currencyType(currency + 1);
- d. for(currency = Dollar; currency <= Mark; currency++)
 cout<<"* ";

4. 给定:

```
enum cropType{ wheat, corn, rye, barley, oats };
cropType crop;
```

选出正确的答案。

- a. static_cast<int>(wheat) is 0 (i) true (ii) false
 - b. static_cast<cropType>(static_cast<int>(wheat) - 1) is wheat
 (i) true (ii) false
 - c. rye > wheat (i) true (ii) false
 - d. for(crop = wheat; crop <= oats; ++crop)
 cout<<"* ";
- outputs: ***** (i) true (ii) false

5. 下面的程序有哪些错误?

```
#include <iostream> //Line 1
int main() //Line 2
{
    cout<<"Hello World! "<<endl; //Line 3
    return 0; //Line 4
}
```

6. 下面的程序有哪些错误?

```
#include <iostream.h> //Line 1
using namespace std; //Line 2
int main() //Line 3
{
    int x = 0; //Line 4
    cout<<"x = "<<x<<endl; //Line 5
    return 0; //Line 6
}
```

7. 下面的程序有哪些错误?

```
#include <iostream> //Line 1
namespace aaa //Line 2
{
    const int x = 0; //Line 3
    double y; //Line 4
}
using namespace std; //Line 5
int main() //Line 6
{
    y = 34.50; //Line 7
    cout<<"x = "<<x<<"", y = "<<y<<endl; //Line 8
    return 0; //Line 9
}
```

8. 下面的程序有哪些错误?

```
#include <iostream> //Line 1
#include <cmath> //Line 2
using std; //Line 3
int main() //Line 4
{
    return 0; //Line 5
}
```

9. 下面程序的输出是什么?

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string str1 = "Amusement Park";
    string str2 = "Going to";
    string str3 = "the";
    string str;

    cout<<str2 + ' ' + str3 + ' ' + str1<<endl;
    cout<<str1.length()<<endl;
    cout<<str1.find('P')<<endl;
    cout<<str1.substr(1,5)<<endl;

    str = "ABCDEFGHGIJK";
    cout<<str<<endl;
    cout<<str.length()<<endl;
}
```



```

    str[0] = 'a';
    str[2] = 'd';

    cout<<str<<endl;
    return 0;
}

```

8.8 编程练习

1. a. 定义一个枚举类型 `triangleType`, 它的值为 `scalene`、`isosceles`、`equilateral` 和 `noTriangle`。
 b. 编写函数并将三个成员作为参数, 每一个代表三角形的边长。函数应该返回三角形的形状。(注意: 在一个三角形中, 两边之和应该大于第三边。)
 c. 编写程序, 提示用户输入三角形的边长, 输出三角形的形状。
2. 重写第4章练习12, 其中所有的常量用 `namespace` 来定义。
3. 重新编写 Pig Latin 程序, 让它能处理一个未指定长度的文本。如果单词以标点符号结尾, 那么用 Pig Latin 形式将标点符号放在字符串的末尾。例如, Hello 的 Pig Latin 形式是 ello-Hay。假设文本包含了下面的标点符号: , (逗号) . (句点) ? (问号) ; (分号) : (冒号)。
4. 编写程序来计算联邦所得税。税款计算方法如下: 对于未婚者, 免税标准是 4000 美元; 对于已婚者, 免税标准是 7000 美元。同时, 还需要交付薪水总额的 6% 作为养老金。税款如下:
 如果税前薪水为:
 - 在 0 美元和 15 000 美元之间, 税率是 15%
 - 在 15 001 美元和 40 000 美元之间, 税款是 2250 美元外加超过 15 000 美元税前工资的 25%
 - 超过 40 000 美元, 税款是 8460 美元外加超过 40 000 美元税前工资的 35%
 提示用户输入下面的信息:
 - 婚姻状况
 - 如果婚姻状况是“已婚”, 那么询问小于 14 岁的子女数量
 - 总收入 (如果婚姻状况是“已婚”, 而且配偶都有工资, 输入总薪水)
 - 养老金所占总收入的百分比
 程序至少包含下面的函数:
 - a. 函数 `getData()`: 函数要求用户输入相关的数据
 - b. 函数 `taxAmount()`: 函数计算和返回应缴税款
 为了计算税前工资, 应减去免税标准的金额、养老金和个人免税额, 即每人 1500 美元。
5. 如果整数集合 a, b, c 满足 $a^2 + b^2 = c^2$, 则该集合称为毕达格拉斯三元组。例如, 整数 3, 4, 5 组成了一个毕达格拉斯三元组。为了找到毕达格拉斯三元组, 使用下面的方程式。假设 m 和 n 是整数。如果 $a = m^2 - n^2$, $b = 2mn$, $c = m^2 + n^2$, 那么 a, b, c 是毕达格拉斯三元组。编写一个程序, 提示用户输入 m 和 n 的值, 输出对应于 m 和 n 的毕达格拉斯三元组。
6. (小数计算) 编写一个程序让用户执行小数的算术操作。小数的形式是 a/b , 其中 a 和 b 是整数并且 $b \neq 0$ 。程序必须是菜单驱动的, 允许用户选择运算符 (+、-、*、/) 和输入小数的分子和分母。而且, 程序至少必须包含下面的函数:
 - a. 函数 `menu()`: 函数告诉用户程序的目的, 解释如何输入数据并允许用户选择操作。
 - b. 函数 `addFractions()`: 函数将 4 个整数作为参数, 代表两个小数的分子和分母, 将小数相加, 并且返回计算结果的分子和分母。

- c. 函数 `subtractFractions()` : 函数将 4 个整数作为参数, 代表两个小数的分子和分母, 将小数相减并且返回计算结果的分子和分母。
- d. 函数 `multiplyFractions()` : 函数将 4 个整数作为参数, 代表两个小数的分子和分母, 将小数相乘并且返回计算结果的分子和分母。
- e. 函数 `divideFractions()` : 函数将 4 个整数作为参数, 代表两个小数的分子和分母, 将小数相除并且返回计算结果的分子和分母。

示例输出如下所示:

$3/4 + 2/5 = 23/20$

$2/3 * 3/5 = 6/15$

答案不需要最简形式。

第9章 数组和字符串

本章要点:

- 学习数组
- 研究怎样声明并对数组中的数据进行操作
- 了解“越界”的含义
- 熟悉数组处理的限制
- 了解如何将数组作为参数传递给函数
- 学习C字符串
- 掌握处理C字符串函数的用法
- 学习如何从一个C字符串中输入或输出数据
- 学习关联数组

在前面的章节中,学习了简单数据类型。在第2章中,读者已经了解到C++数据类型分为三类,其中一类是结构化数据类型。第9章和后面的几章将集中讨论结构化数据类型。

前面讲过,如果某个类型的变量每次只能存储一个值,那么该数据类型称为简单数据类型。而结构化数据类型的每一个数据项是其他数据项的集合。简单数据类型是结构化数据类型的基础。我们要讨论的第一个结构化数据类型是数组。在第12章和第13章中,将讨论其他结构化类型。

在正式定义数组前,先考虑下面的问题。我们先编写一个C++程序来读入5个数字,计算它们的和并且用倒序打印数字。

在第5章中学习了怎样读入数据、打印数据、计算和。这里的不同点在于用倒序来打印数字,所以直到打印完第5个数字后才能打印第4个数字,等等。

这意味着在开始用倒序打印之前需要存储所有的数字。目前为止,可以用下面的程序来完成这一任务。

```
//Program to read five numbers, find their sum, and print the
//numbers in reverse order.

#include <iostream>
using namespace std;

int main()
{
    int item0, item1, item2, item3, item4;
    int sum;

    cout<<"Enter five integers: ";
    cin>>item0>>item1>>item2>>item3>>item4;
    cout<<endl;

    sum = item0 + item1 + item2 + item3 + item4;

    cout<<"The sum of the numbers = "<<sum<<endl;
    cout<<"The numbers in reverse order are: ";
    cout<<item4<<" "<<item3<<" "<<item2<<" "
```

```
<<item1<<" "<<item0<<endl;
return 0;
}
```

该程序可以很好地运行。然而,如果需要读入100(或更多)个数字并用倒序打印它们,将不得不声明100个变量,并且还要编写很多的cin和cout语句。这样,对于大量的数据来说,这种类型的程序是不可能实现的。

注意前面程序中的下面几点:

1. 必须声明5个变量,因为数字要被倒序打印。
2. 所有的变量都是int类型,也就是相同的数据类型。
3. 声明这些变量的方法表明,除了最后一个字符外(它是一个数字),要存储这些数字的变量具有相同的名字。

说明1告诉用户必须声明5个变量。说明3表明,如果能以某种方法将最后一个字符(也就是数字)输入到一个计数器变量,并且使用for循环从0到4对读入进行计数,使用另一个for循环从4到0进行打印计数,那么这一过程将会变得很容易。最后,因为所有的变量具有相同的数据类型和确定的数量,所以,可以用一条更简单的语句来完成上述过程。

在C++中,一种称为数组的数据结构可以实现这一目标。

9.1 数组

数组是一个固定数目的元素集合,在数组中,所有的元素都具有相同的数据类型。一维数组中,元素按列表的形式进行排列。本章的剩余部分只讨论一维数组。(第10章讨论二维和 multidimensional 数组) 声明一维数组的通常格式为:

```
dataType arrayName[ intExp ] ;
```

其中intExp是一个值为正整数的表达式。而且intExp指定了数组中元素的个数。

例 9.1

语句:

```
int num [ 5 ] ;
```

声明了一个有5个元素的数组num,每一个元素都是int类型。元素分别是num[0]、num[1]、num[2]、num[3]和num[4]。图9.1演示了数组num。

9.1.1 访问数组元素

访问数组元素通常的语法是:

```
arrayName[ indexExp ] ;
```

其中称为下标的indexExp是任何一个值为非负整数的表达式。下标值指定了数组中元素的位置。

C++中,[]是一个运算符,称为数组下标运算符。而且,在C++中,数组的下标从0开始。

考虑下面的语句:

```
int list[ 10 ] ;
```

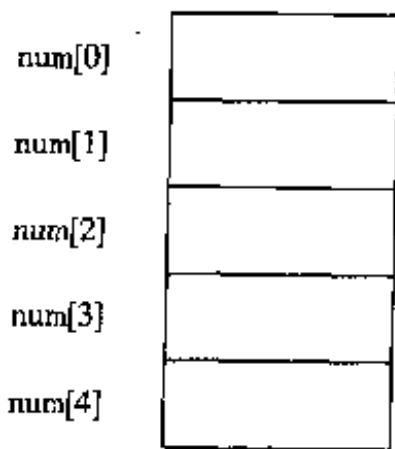


图9.1 数组num

该语句声明了具有10个元素的数组 `list`。元素是 `list[0]`、`list[1]`、...`list[9]`。换言之，语句声明了10个变量。（参看图9.2）

赋值语句：

```
list[5] = 34;
```

将34存储在 `list[5]` 中，即数组 `list` 的第6个元素。（参看图9.3。）

<code>list[0]</code>	
<code>list[1]</code>	
<code>list[2]</code>	
<code>list[3]</code>	
<code>list[4]</code>	
<code>list[5]</code>	
<code>list[6]</code>	
<code>list[7]</code>	
<code>list[8]</code>	
<code>list[9]</code>	

图9.2 数组 `list`

<code>list[0]</code>	
<code>list[1]</code>	
<code>list[2]</code>	
<code>list[3]</code>	
<code>list[4]</code>	
<code>list[5]</code>	34
<code>list[6]</code>	
<code>list[7]</code>	
<code>list[8]</code>	
<code>list[9]</code>	

图9.3 在语句“`list[5]=34;`”执行后的数组 `list`

假设 `i` 是一个 `int` 变量。那么赋值语句：

```
list[3] = 63;
```

等效于赋值语句：

```
i = 3;
list[i] = 63;
```

如果 `i` 是4,那么赋值语句：

```
list[2*i-3] = 58;
```

将58存储在 `list[5]` 中，因为 $2*i-3$ 等于5。下标表达式首先被计算，然后给出数组中元素的位置。

接着，考虑下面的语句：

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

第一条语句将10存储到 `list[3]` 中，第二条语句将35存储到 `list[6]` 中，第三条语句将 `list[3]` 与 `list[6]` 中的内容求和后，并将结果存储到 `list[5]` 中。（参看图9.4。）

<code>list[0]</code>	
<code>list[1]</code>	
<code>list[2]</code>	
<code>list[3]</code>	10
<code>list[4]</code>	
<code>list[5]</code>	45
<code>list[6]</code>	35
<code>list[7]</code>	
<code>list[8]</code>	
<code>list[9]</code>	

图9.4 在语句“`list[3]=10;`”、“`list[6]=35;`”和“`list[5]=list[3]+list[6];`”执行后的数组 `list`

例 9.2

也可以声明如下的数组：

```
const int arraySize = 10;
int list[ arraySize];
```

可以先声明一个常量，然后使用常量的值声明一个数组并且指定它的大小。

注意：当声明数组时，其大小必须是已知的。例如，不能声明下面的数组：

```
int arraySize;                                //Line 1

cout<<"Enter the size of the array: ";        //Line 2
cin>>arraySize;                                //Line 3
cout<<endl;                                    //Line 4

int list[ arraySize];                          //Line 5; not allowed
```

当程序执行时，第2行语句要求用户输入数组的大小。第3行语句将数组的大小输入到arraySize中。当编译器编译第1行时，变量arraySize的值是未知的。所以编译期间，变量arraySize的值是未知的。这样，当编译器编译第5行时，数组的大小是未知的，而且编译器不知道应该为数组分配多大的内存。在第15章中，将会学习程序执行期间如何指定数组的大小，然后使用指针分配相应大小的数组。程序执行期间使用指针创建的数组称为动态数组。目前，当声明一个数组时，其大小必须是已知的。

9.1.2 处理一维数组

对一维数组进行的基本操作包括：初始化数据、输入数据、输出存储在数组中的数据、计算数组中最大和/或最小的元素。而且，如果数据是数值类型的，那么可以计算数组的和与平均值。这种操作需要对数组进行遍历。数组元素的遍历可以通过一个循环加以实现。

例如，假设有下面的语句：

```
int list[100];    //list is an array of the size 100
int i;
```

下面的for循环从第一个元素开始遍历了数组list的每一个元素。

```
for(i = 0; i < 100; i++)    //Line 1
    process list[i]         //Line 2
```

如果处理list时需要向list中输入数据，那么第2行语句可以采用输入语句的格式，比如cin语句。例如，下面的语句从键盘读入了100个数字并且将数字存储在list中。

```
for(i = 0; i < 100; i++)
    cin>>list[i];
```

同样，如果处理list需要输出数据，那么第2行语句可以采用输出语句的格式。例9.3进一步演示了如何处理一维数组。

例 9.3

该例子显示了如何使用循环来处理数组。下面的声明在整个程序中都被使用：

```
double sale[10];
int index;
double largestSale, sum, average;
```


第一条语句声明了有 10 个元素的数组 `sale`，每一个元素都是 `double` 类型。

- a. 初始化一个数组：下面的循环将数组 `sale` 中的每一个元素初始化为 0.0。

```
for(index = 0; index < 10; index++)
    sale[index] = 0.0;
```

- b. 向数组中读入数据：下面的循环向数组 `sale` 中输入数据。为了简单起见，假定数据从键盘输入。

```
for(index = 0; index < 10; index++)
    cin >> sale[index];
```

- c. 打印一个数组：下面的循环输出了数组 `sale`。为了简单起见，假设数据输出到屏幕上。

```
for(index = 0; index < 10; index++)
    cout << sale[index] << " ";
```

- d. 计算数组的和与平均值：因为数组 `sale` 代表特定的销售数据，所以需要计算总销售量与平均销售量。下面的 C++ 代码计算数组 `sale` 的总销售量与平均销售量：

```
sum = 0;
for(index = 0; index < 10; index++)
    sum = sum + sale[index];
average = sum / 10;
```

- e. 求解数组中的最大元素：现在讨论求解数组中最大元素（也就是最大值的元素）的算法。然而，用户通常更关心的是数据中最大元素的位置。

当然，如果知道位置（也就是数组中最大元素的下标），可以很容易地求出数组中最大元素的值。

假设 `maxIndex` 包含数组 `sale` 中最大元素的下标。假设列表中的第一个元素是最大的元素，所以 `maxIndex` 初始化为 0。然后将列表中的每一个元素与 `maxIndex` 指向的元素进行比较，只要在数组中发现一个比 `maxIndex` 所指元素大的元素，那么就更新 `maxIndex`，让它指向新的较大的元素。算法如下：

```
maxIndex = 0;
for(index = 1; index < 10; index++)
    if(sale[maxIndex] < sale[index])
        maxIndex = index;
```

```
largestSale = sale[maxIndex];
```

让我们通过一个例子来演示该算法是如何工作的。假设数组 `sale` 如图 9.5 所示。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
sale	12.50	8.35	19.60	25.00	14.00	39.43	35.90	98.23	66.65	35.64

图 9.5 数组 `sale`

代码确定了数组 `sale` 中最大的元素。在 `for` 循环开始前，`maxIndex` 初始化为 0 并且 `for` 循环初始化 `index` 为 1。输出显示了 `for` 循环每一次循环期间 `maxIndex`、`index` 和相应的数组元素：

index	maxIndex	sale[maxIndex]	sale[index]	sale[maxIndex] < sale[index]
1	0	12.50	8.35	12.50 < 8.35 is false
2	0	12.50	19.60	12.50 < 19.60 is true;

				maxIndex = 2
3	2	19.60	25.00	19.60 < 25.00 is true;
				maxIndex = 3
4	3	25.00	14.00	25.00 < 14.00 is false
5	3	25.00	39.43	25.00 < 39.43 is true;
				maxIndex = 5
6	5	39.43	35.90	39.43 < 35.90 is false
7	5	39.43	98.23	39.43 < 98.23 is true;
				maxIndex = 7
8	7	98.23	66.25	98.23 < 66.25 is false
9	7	98.23	35.64	98.23 < 35.64 is false

在for循环执行后，maxIndex=7并给出了数组sale中最大的元素。这样，largestSale=sale[maxIndex]=98.23。

注意：可以编写一个算法计算数组中最小的元素，该算法与求解一个数组中最大元素的算法是类似的。（参看本章编程练习2。）

既然知道如何声明和处理数组，我们可以重新编写本章开始讨论的程序。前面讲过，该程序读入5个数字，然后计算和并打印倒序的数字。

例9.4

```
//Program to read five numbers, find their sum, and
//print the numbers in reverse order.

#include <iostream>
using namespace std;

int main()
{
    int item[5]; //declare an array item of five components
    int sum;
    int counter;

    cout<<"Enter five numbers."<<endl;

    sum = 0;

    for(counter = 0; counter < 5; counter++)
    {
        cin>>item[counter];
        sum = sum + item[counter];
    }

    cout<<"The sum of the numbers is: "<<sum<<endl;
    cout<<"The numbers in reverse order are: ";

    //print the numbers in reverse order
    for(counter = 4; counter >= 0; counter--)
        cout<<item[counter]<<" ";

    cout<<endl;

    return 0;
}
```

运行实例：在本运行实例中，用户输入用阴影表示。

```
Enter five numbers.
12 76 34 52 89
The sum of the numbers is: 263
The numbers in reverse order are: 89 52 34 76 12
```


9.1.3 数组下标越界

考虑下面的声明：

```
double num[10];  
int i;
```

如果 $i=0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ ，那么元素 $\text{num}[i]$ 是有效的，即 i 是一个有效的下标。

如果 $\text{index} \geq 0$ 并且 $\text{index} \leq \text{arraySize}-1$ ，那么可以说下标在范围内。如果 $\text{index} < 0$ 或 $\text{index} > \text{arraySize}-1$ ，那么下标是越界的。

但是在 C++ 中，没有预防下标越界的功能。这样，C++ 不会检查下标值是否在范围内，也就是在 0 和 $\text{arraySize}-1$ 之间。如果下标超出了范围，那么程序会试图访问由下标指定的元素，然而无论下标指向什么位置，该内存位置都会被访问。这种情况会导致意外地改变或访问内存中的信息。因此，如果执行其间下标超出范围，那么可能会发生一些很奇怪的事情。保证下标在范围内是程序员的一个主要责任。

下面的循环将导致下标越界。

```
for(i=0; i<=10; i++)  
    list[i] = 0;
```

其中假设 list 是一个有 10 个元素的数组。当 i 变为 10 时，循环测试到 $i \leq 10$ 的值为 true，循环体就执行，这样的结果是将 0 保存到 $\text{list}[10]$ 中。逻辑上讲， $\text{list}[10]$ 是不存在的。

9.1.4 声明期间的数组初始化

与其他简单变量一样，数组在声明时也可以被初始化。例如，下面的 C++ 语句声明了一个有 5 个元素的数组 list 并且初始化了这些元素。

```
double sales[5] = {12.25, 32.50, 16.90, 23, 45.68};
```

这些值放在花括号之间并且用逗号隔开。其中：

```
sales[0] = 12.25, sales[1] = 32.50, sales[2] = 16.90, sales[3] = 23.00,  
sales[4] = 45.68.
```

在声明时初始化数组可以不必指定数组的大小。数组的大小由花括号中初始化值的数目来决定。然而，必须在数组名字后面包含方括号。所以，前面的语句等价于：

```
double sales[] = {12.25, 32.50, 16.90, 23, 45.68};
```

尽管在声明期间进行初始化时不必指定数组的大小，但这样做却是一种很好的习惯。

声明期间数组的部分初始化

当声明并初始化一个数组时，不必初始化数组的所有元素。这一过程称为声明期间数组的部分初始化。然而，如果声明期间部分初始化一个数组时，必须注意一些问题。下面的例子解释了当声明并部分初始化一个数组时发生的情况。

语句：

```
int list[10] = {0};
```

声明 list 是一个包含 10 个元素的数组并且将所有元素都初始化为 0。语句：

```
int list[10] = {8, 5, 12};
```

声明 `list` 是一个包含 10 个元素的数组，并且将 `list[0]` 初始化为 8、`list[1]` 初始化为 5、`list[2]` 初始化为 12，其他元素都初始化为 0。这样，如果所有的值都没有在初始化语句中指定，那么数组元素的值都为 0。注意，在声明语句时，可以忽略数组的大小。例如，语句：

```
int list[] = { 5, 6, 3 };
```

声明 `list` 是一个包含 3 个元素的数组，并且将 `list[0]` 初始化为 5、`list[1]` 初始化为 6、`list[2]` 初始化为 3。与此相比，语句：

```
int list[25] = { 4, 7 };
```

声明 `list` 是一个包含 25 个元素的数组。前两个元素分别初始化为 4 和 7，其他元素初始化为 0。

9.1.5 对数组处理的一些限制

假设 `x` 和 `y` 是有相同类型和大小（比如说 25）的两个数组，数组 `x` 已经初始化，现在想将数组 `x` 的值赋值给 `y`。下面的语句是不合法的：

```
y=x; //illegal
```

将一个数组复制给另一个数组时，必须进行对应元素的复制，也就是说，一次只能复制一个元素。可以使用下面的循环来实现：

```
for(j=0; j<25; j++)  
    y[j] = x[j];
```

同样，比较数组（向数组读入数据，打印数组的内容）时也必须将对应的元素进行比较。所以下面的语句是不合法的：

```
cin>>x; //illegal  
cout<<y; //illegal  
if(x <= y) //illegal  
.  
.  
.
```

所以，C++ 不允许对一个数组进行聚集操作。对一个数组的聚集操作是将整个数组作为一个单元的操作。

9.1.6 数组作为函数的参数

在学习了怎样对数组操作后，很自然地产生了一个问题：数组怎样作为一个参数传递给函数呢？

只作为引用参数：在 C++ 中，数组只作为引用参数进行传递。

因为数组只作为引用参数被传递，所以当声明数组作为形参时不必使用使用符号 `&`。

当声明一维数组作为形参时，数组的大小通常是被忽略的。当声明一维数组作为形参时，如果指定了它的大小，那么它将被编译器忽略。

下面的函数将一个 `int` 数组作为参数：

```
void initialize(int list[])  
{  
    int count;  
    for(count = 0; count < 5; count++)  
        list[count] = 0;  
}
```

因为函数中的for循环执行5次,所以函数将大小为5的数组初始化为0。如果数组的大小更改了,需要重新编写另一个函数。然而,如果在函数头中添加另外一个形参(比如说size),那么在调用期间可以使用size的值来控制for循环重复的次数(即使用size代替5),并且将实际数组的名字与大小一起传递给函数,这样可以对任何大小的数组使用相同的函数。重新编写该函数如下:

```
void initialize(int list[], int size)
{
    int count;
    for(count = 0; count < size; count++)
        list[count] = 0;
}
```

函数initialize()的第一个参数是一个任意大小的int数组。当调用函数initialize()时,实际数组的大小作为另一个参数传递给函数initialize()。

作为形参的常量数组

当形参是一个引用参数时,那么无论什么时候改变形参,实参也会相应地改变。然而,尽管数组总是作为引用参数被传递,仍然可以阻止函数改变实参。可以通过使用保留字const在声明形参时做到这一点。考虑下面的函数:

```
void example(int x[], const int y[], int sizeX, int sizeY)
{
    .
    .
    .
}
```

其中函数example()能够修改数组x,但是不能修改数组y。试图改变数组y将会导致一个编译错误。如果不想让函数修改数组,那么将形参数组声明为常量是一种很好的编程习惯。

例 9.5

该例子显示了如何编写函数进行数组处理以及将数组声明为一个形参。

```
//Function to initialize an array to 0
void initializeArray(int x[], int sizeX)
{
    int counter;
    for(counter = 0; counter < sizeX; counter++)
        x[counter] = 0;
}

//Function to read data and store it in an array
void fillArray(int x[], int sizeX)
{
    int counter;
    for(counter = 0; counter < sizeX; counter++)
        cin >> x[counter];
}

//Function to print the array
void printArray(const int x[], int sizeX)
{
    int counter;
    for(counter = 0; counter < sizeX; counter++)
        cout << x[counter] << " ";
}
```

```
//Function to find and return the sum of an array
int sumArray(const int x[], int sizeX)
{
    int counter;
    int sum = 0;
    for(counter = 0; counter < sizeX; counter++)
        sum = sum + x[counter];

    return sum;
}

//Function to find and return the index of the
//largest element of an array
int indexLargestElement(const int x[], int sizeX)
{
    int counter;
    int maxIndex = 0; //Assume first element is the largest
    for(counter = 1; counter < sizeX; counter++)
        if(x[maxIndex] < x[counter])
            maxIndex = counter;

    return maxIndex;
}

//Function to copy one array into another array
void copyArray(const int x[], int y[], int length)
{
    int counter;
    for(counter = 0; counter < length; counter++)
        y[counter] = x[counter];
}
```

注意, 为了使函数 `copyArray()` 能正确运行, 数组 `y` 必须和数组 `x` 大小相等。

数组的地址

数组的地址是第一个数组元素的地址 (也就是内存位置)。例如, 如果 `list` 是一维数组, 那么 `list` 的地址是元素 `list[0]` 的地址。

当数组作为参数传递时, 实际数组的地址被传递给形参。

如果允许数组通过值传递, 那么计算机必须为形参的元素分配内存, 并且将实际数组的内容复制到相应的形参中。如果数组很大, 那么这样进行处理不仅浪费内存, 在复制数据的时候还会浪费大量的计算机时间。

函数不能返回一个数组类型的值

C++ 不允许函数返回一个数组。前面描述的函数 `sumArray()` 和 `largestElement()` 返回 `int` 类型的值。

例 9.6

下面的程序演示了在函数调用中数组如何作为实参被传递。

```
//Arrays as parameters to functions
#include <iostream>
using namespace std;
const int arraySize = 10;
```

```

void initializeArray(int x[], int sizeX);
void fillArray(int x[], int sizeX);
void printArray(const int x[], int sizeX);
int sumArray(const int x[], int sizeX);
int indexLargestElement(const int x[], int sizeX);
void copyArray(const int x[], int y[], int length);

int main()
{
    int listA[arraySize] = {0}; //Line 1
    int listB[arraySize]; //Line 2

    cout<<"Line 1: listA elements: "; //Line 3
    printArray(listA, arraySize); //Line 4
    cout<<endl; //Line 5

    initializeArray(listB, arraySize); //Line 6

    cout<<"Line 7: ListB elements: "; //Line 7
    printArray(listB, arraySize); //Line 8
    cout<<endl<<endl; //Line 9

    cout<<"Line 10: Enter "<<arraySize
        <<" integers: "; //Line 10
    fillArray(listA, arraySize); //Line 11
    cout<<endl; //Line 12
    cout<<"Line 13: After filling listA the elements are:"
        <<endl; //Line 13
    printArray(listA, arraySize); //Line 14
    cout<<endl<<endl; //Line 15

    cout<<"Line 16: Sum of the elements of listA is: "
        <<sumArray(listA, arraySize)<<endl<<endl; //Line 16
    cout<<"Line 17: Location of the largest element in listA is: "
        <<indexLargestElement(listA, arraySize) + 1
        <<endl; //Line 17
    cout<<"Line 18: Largest element in listA is: "
        <<listA[indexLargestElement(listA, arraySize)]
        <<endl<<endl; //Line 18

    copyArray(listA, listB, arraySize); //Line 19
    cout<<"Line 20: After copying the elements of "
        <<"listA into listB"<<endl
        <<" listB elements are: "; //Line 20
    printArray(listB, arraySize); //Line 21
    cout<<endl; //Line 22

    return 0;
}

//Place the definition of the functions initializeArray, fillArray,
//and so on here. Example 9-5 gives the definitions of these
//functions.

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Line 1: listA elements: 0 0 0 0 0 0 0 0 0 0
Line 7: ListB elements: 0 0 0 0 0 0 0 0 0 0
Line 10: Enter 10 integers: 11 17 25 63 56 43 98 29 5 12
Line 13: After filling listA the elements are:

```

```

33 77 25 63 56 48 98 39 5 12
Line 16: Sum of the elements of listA is: 456
Line 17: Location of the largest element in listA is: 7
Line 18: Largest element in listA is: 98
Line 20: After copying the elements of listA into listB
        listB elements are: 33 77 25 63 56 48 98 39 5 12

```

程序的输出是很容易理解的。第1行语句声明了包含10个元素的数组listA, 并且将listA的每一个元素初始化为0。第2行语句声明了一个包含10个元素的数组listB。第4行语句调用函数printArray()并且输出存储在listA中的值。第11行语句调用函数fillArray()向数组listA输入数据。第16行语句调用函数sumArray()并且输出listA中所有元素的和。同样, 第18行语句输出listA中元素的最大值。

9.1.7 整数数据类型和数组下标

C++ 允许使用任何一种整数类型来作为数组的下标。这种灵活性极大地提高了程序的可读性。考虑下面的语句:

```

enum paintType{ Green, Red, Blue, Brown, White, Orange, Yellow };
double paintSale[ 7 ];
paintType paint;

```

下面的循环将数组 paintSale 初始化为 0:

```

for(paint = Green; paint <= Yellow;
    paint = static_cast<paintType>(paint + 1))
    paintSale[paint] = 0.0;

```

下面的语句更新了 Red 涂料的销售量:

```

paintSale[ Red ] = paintSale[ Red ] + 75.69;

```

正如所看到的, 上面的代码比使用整数作为下标的代码更容易理解。由于这一原因, 只要需要, 就可以使用枚举类型或其他整数数据类型作为下标。

其他声明数组的方法

假设一个班级有20名学生, 用户需要知道学生的分数。因为学生的数量可能发生改变, 所以当声明数组时不必指定数组的大小, 可以声明如下的数组:

```

const int noOfStudents = 20;
int testScore[ noOfStudents ];

```

声明数组的其他形式是:

```

const int size = 50;           //Line 1
typedef double list[ size];    //Line 2
list a;                        //Line 3
list myList;                   //Line 4

```

第2行语句定义了一种数据类型list, 它是一个有50个double类型元素的数组。第3行和第4行语句声明了两个变量a和mylist, 都是包含50个double类型元素的数组。当然, 这些语句等价于下面的语句:

```

double a[ 50 ];

```



```
double mylist[ 50];
```

9.2 C 字符串（字符数组）

到现在为止，我们并没有讨论字符数组。字符数组很有用并且不同于处理其他类型的数组。C++ 提供了很多可以与字符数组一起使用的（预定义的）函数。

字符数组：其元素为 char 类型的数组。

前面讲过，最广泛使用的字符集是 ASCII 码和 EBCDIC 码。ASCII 字符集中的一个字符是空字符，它是不可打印的。同时，空字符用 '\0' 来表示。

语句：

```
ch = '\0';
```

将空字符保存到 ch 中，其中 ch 是一个 char 变量。

后面将会看到，空字符在处理字符数组中扮演了一个非常重要的角色。因为空字符的对照序列是 0，所以空字符小于 char 数据集中任何一个字符。

对于字符数组来说，最常用的术语是 C 字符串。然而，两者之间有一个微小的差别。字符串是 0 或多个字符的序列，字符串是用双引号来括起来的。在 C++ 中，C 字符串是以 '\0' 作为终止的，也就是说在一个 C 字符串中，最后一个字符总是空字符。一个字符数组可能不包含空字符，但是在 C 字符串中最后一个字符总是空字符。正如所看到的，空字符除了出现在最后一个位置外，不会出现在其他任何地方。而且，C 字符串存储在（一维）字符数组中。

在这一部分中，除非特别说明，否则所说的“字符串”都是指 C 字符串。下面是这种字符串的一些例子：

```
"John L. Johnson"  
"Hello there."
```

从字符串的定义中可以明显的看出，'A' 与 "A" 之间是有差别的。前者是字符 A，后者是字符串 A。因为字符串是以 null 结尾的，所以 "A" 代表了两个字符：'A' 和 '\0'。同样，"Hello" 代表 6 个字符：'H'、'e'、'l'、'l'、'o' 和 '\0'。为了存储 'A'，只需要一个 char 类型的内存单元；而为了存储 "A"，需要两个 char 类型的内存单元，一个用来存储 'A'，另一个用来存储 '\0'。同样，为了在计算机内存中存储字符串 "Hello"，需要 6 个 char 类型的内存单元。

考虑下面的语句：

```
char name[ 16];
```

该语句声明了一个包含 16 个 char 类型元素的数组 name。因为字符串以 null 结尾并且 name 有 16 个元素，所以能存储在 name 中字符串的最大长度是 15。如果在 name 中存储一个长度为 10 的字符串，那么 name 的前 11 个元素被使用，后 5 个字符则没有被使用。

语句：

```
char name[ 16] = { 'J', 'o', 'h', 'n', '\0' };
```

声明了一个包含 16 个 char 类型元素的数组 name，并且将字符串 "John" 保存在该数组中。char 数组变量声明期间，C++ 允许字符串符用在初始化语句中。所以上面的语句等价于：

```
char name[ 16] = "John";           //Line A
```

前面讲过，如果数组在声明期间初始化时，数组的大小可以被忽略。语句：

```
char name[] = "John";           //Line B
```

声明了一个长度足够大的字符串变量 name (在该情况下是 5) 并且将 "John" 存储在数组中。上面的两条语句是有差别的: 两条语句都将 "John" 存储在 name 中, 但是行 A 中数组 name 的大小是 16, 而行 B 语句中 name 的大小是 5。

适用于其他数组的大部分规则也适用于字符数组。考虑下面的语句:

```
char studentName[ 26];
```

假设有将 "Lisa L. Johnson" 存储在 studentName 中。因为集合操作 (例如赋值和比较) 不允许应用于数组, 所以下面的语句是不合法的:

```
studentName = "Lisa L. Johnson"; //Illegal
```

C++ 提供了一组用于字符串操作的函数, 头文件 cstring 声明了这些函数。我们经常使用的三个函数是: strcpy() (字符串复制)、strcmp() (字符串比较)、strlen() (字符串长度, 计算字符串的长度)。表 9.1 总结了这些函数。

表 9.1 strcpy()、strcmp() 和 strlen() 函数

函数	功能
strcpy(s1,s2)	将字符串 s2 复制到 s1 中, s1 的长度不小于 s2 的长度
strcmp(s1,s2)	如果字符串 s1 小于字符串 s2, 返回值小于 0; 如果字符串 s1 与字符串 s2 相等, 返回值为 0; 如果字符串 s1 大于字符串 s2, 返回值大于 0
strlen(s)	返回字符串的长度且不包含空字符

为了使用这些函数, 程序必须通过 include 语句包含头文件 cstring。也就是说下面的语句必须包含在程序中:

```
#include <cstring>
```

9.2.1 字符串比较

在 C++ 中, 字符串是通过使用系统的对照序列逐个字符进行比较的。假设使用的是 ASCII 字符集。

1. 字符串 "Air" 小于字符串 "Boat", 因为 "Air" 的第一个字符小于 "Boat" 的第一个字符。
2. 字符串 "Air" 小于字符串 "An", 因为两个字符串的第一个字符是相同的, 但是 "Air" 的第二个字符 'i' 小于 "An" 的第二个字符 'n'。
3. 字符串 "Bill" 小于字符串 "Billy", 因为 "Bill" 和 "Billy" 的前 4 个字符串是相同的, 但是 "Bill" 的第 5 个字符 (即 '\0', 空字符) 小于 "Billy" 的第 5 个字符, 即 'y'。(前面讲过, C 字符串以 null 结尾。)
4. 字符串 "Hello" 小于字符串 "hello", 因为 "Hello" 的第一个字符 'H' 小于 "hello" 的第一个字符 'h'。

正如所看到的, 函数 strcmp() 将其第一个字符串参数与第二个字符串参数进行逐个字符地比较。

例 9.7

假设有下面的语句:

```
char studentName[ 21];
char myname[ 16];
char yourname[ 16];
```

下面的语句演示了字符串函数是如何工作的。

语句	功能
<code>strcpy(myname, "John Robinson");</code>	<code>myname = "John Robinson"</code>
<code>strlen("John Robinson");</code>	返回字符串 "John Robinson" 的长度: 13
<code>int len;</code>	
<code>len = strlen("Sunny Day");</code>	len 赋值为 9
<code>strcpy(yourname, "Lisa Miller");</code>	<code>yourname = "Lisa Miller"</code>
<code>strcpy(studentName, yourname);</code>	<code>studentName = "Lisa Miller"</code>
<code>strcmp("Bill", "Lisa");</code>	返回值小于 0
<code>strcpy(yourname, "Kathy Brown");</code>	<code>yourname = "Kathy Brown"</code>
<code>strcpy(myname, "Mark G. Clark");</code>	<code>myname = "Mark G. Clark"</code>
<code>strcmp(myname, yourname);</code>	返回值大于 0

注意: 在本章中, 定义 C 字符串是由零或多个字符组成的序列。C 字符串用双引号括起来并以 null 结尾, 所以 C 字符串 "Hello" 包含 6 个字符, 虽然只有 5 个字符在双引号内。所以, 为了将 C 字符串 "Hello" 保存在计算机内存中, 必须使用一个大小为 6 的字符数组。C 字符串的长度是双引号内实际字符串的数量, 例如, 字符串 "Hello" 的长度是 5。这样, 逻辑上 C 字符串是零或多个字符组成的序列, 但是物理上 (也就是存储 C 字符串的计算机内存), C 字符串至少有一个字符。因为字符串的长度是双引号内字符的实际数目, 所以定义 C 字符串是零或多个字符的序列。然而, 必须记住当比较 C 字符串, 特别是像 "Bill" 和 "Billy" 这样的字符串时, 保存在计算机内存中的 C 字符串最后的空字符扮演了一个很重要的角色。

9.2.2 读 / 写字符串

前面提到过, 大多数应用于数组的规则也能应用于字符串。某些操作, 例如赋值和比较不允许用于数组。甚至数组的输入 / 输出也是逐个元素完成的。然而, C++ 允许字符串的输入和输出 (也就是字符数组)。

为了进行讨论, 需要使用下面的声明:

```
char name[ 31];
```

9.2.3 字符串输入

因为聚集操作允许用于字符串输入, 所以语句:

```
cin>>name;
```

将输入字符串存储到 `name` 中。输入字符串的长度必须小于 30。如果输入字符串的长度是 4, 那么计算机将存储数 4 个字符和空字符 '\0'。如果输入字符串的长度大于 30, 那么因为没有对数组下标进行检查, 所以计算机继续将字符串存储到 `name` 后续的内存单元中。这一过程可能会造成严重的问题, 因为后续内存单元中的数据将会被破坏。

前面讲过, 析取运算符 `>>` 会忽略所有空白字符, 并且一旦发现第一个空白字符或无效数据就会停止读入数据到当前变量中。所以, 包含空格的字符串不能使用析取运算符 `>>` 来读入。例如, 如果姓氏和名字用空格隔开, 那么信息就不能读入到 `name` 中。

如何向一个字符数组中输入带有空白字符的字符串呢? 可以使用 `get()` 函数。前面讲过函数 `get()` 用来读入字符数据。前面所使用的 `get()` 函数 (第 3 章) 只能读入一个单字符。然而, 函数 `get()` 也能用来读入字符串。为了读入字符串, 可以使用有两个参数形式的 `get()` 函数。第一个参数是字符

串变量, 第二个参数指定了向一个字符串变量中读入多少个字符。

为了读入字符串, `get()`函数与输入流变量, 例如 `cin` 一起使用时, 通常的形式为:

```
cin.get(str,m+1);
```

该语句存储了 `m` 个字符, 所有的字符存储到 `str` 中, 直到遇到换行符 `'\n'`。换行符不会存储在 `str` 中。如果输入字符串的字符小于 `m` 个字符, 那么读入就会停在换行符上。

考虑下面的语句:

```
char str[31];  
cin.get(str,31);
```

如果输入是:

```
William T.Johnson
```

那么 "William T.Johnson" 将保存到 `str` 中。如果输入是:

```
Hello there.My name is Mickey Mouse.
```

那么字符串 "Hello there.My name is Mickey" 将保存到 `str` 中。

假设有下面的语句:

```
char str1[26];  
char str2[26];  
char discard;
```

和两行的输入:

```
Summer is warm.  
Winter will be cold.
```

如果想将第1个字符串保存到 `str1` 中, 第2个字符串保存到 `str2` 中。 `str1` 和 `str2` 都可以保存在长度为25的字符串中。因为第1行字符的数量是15, 所以读入停止在 `'\n'` 处。必须在第1行的末尾读入并忽略换行符, 才能将第2行保存在 `str2` 中。下面的语句将第1行保存在 `str1` 中, 第2行保存在 `str2` 中:

```
cin.get(str1,26);  
cin.get(discard);  
cin.get(str2,26);
```

9.2.4 字符串输出

字符串的输出是另一种允许对数组进行的操作。可以将插入运算符 `<<` 与输出数据流变量, 例如 `cout` 一起使用来输出字符串。例如语句:

```
cout<<name;
```

将 `name` 的内容输出到屏幕上。插入运算符 `<<` 输出 `name` 的内容, 直到发现空字符。这样, 如果 `name` 的长度是4, 那么上面的语句只输出4个字符。如果 `name` 不包含空字符, 那么将会看到奇怪的输出, 因为插入运算符继续从后续的内存中输出数据直到找到 `'\0'`。

9.2.5 运行时指定输入/输出文件

在第3章中学习了怎样从一个文件中读入数据。在随后的几章中, 输入文件的名字包含在 `open()` 语句中。所以, 程序总是可以从同一个输入文件接收数据。但是现实应用中, 数据实际可能在多处

进行收集并且保存在独立的文件中。而且,为了比较目的,用户可能想独立地对每一个文件进行处理,然后在独立的文件中保存输出。为了有效地完成这一任务,用户宁愿在执行时而不是程序代码中指定输入和/或输出文件的名字。C++ 允许用户在执行时指定输入/输出文件的名字。

考虑下面的语句:

```
ifstream infile;
ofstream outfile;

char fileName[51];           //assume file name is at most
                              //50 characters long
```

下面的语句提示并且允许用户在执行时指定输入/输出文件:

```
cout<<"Enter the input file name: ";
cin>>fileName;

infile.open(fileName);      //open the input file
.
.
.
cout<<"Enter the output file name: ";
cin>>fileName;

outfile.open(fileName);     //open the output file
```

字符串类型和输入/输出文件

在第8章中讨论了 `string` 数据类型,而 `string` 类型的值不是以 `null` 结尾的。`string` 类型的变量也可以用来读入和存储输入/输出文件的名字。然而,函数 `open()` 的参数必须是一个以 `null` 结尾的字符串,也就是 C 字符串。所以,如果使用一个 `string` 类型的变量读入输入/输出文件的名字,然后使用该变量来打开一个文件,那么该变量的值首先需要转换为一个 C 字符串(也就是以 `null` 结尾的字符串)。头文件 `string` 包含了函数 `c_str()`,它将 `string` 类型的值转换成以 `null` 结尾的字符串数组(也就是 C 字符串)。使用函数 `c_str()` 的语法是:

```
strVar.c_str()
```

其中 `strVar` 是一个 `string` 类型的变量。

下面的语句演示了怎样使用一个 `string` 类型的变量在程序执行期间读入输入/输出文件的名字以及打开这些文件。

```
ifstream infile;
string  fileName;

cout<<"Enter the input file name: ";
cin>>fileName;

infile.open(fileName.c_str()); //open the input file
```

当然,也必须在程序中包含头文件 `string`。对于输出文件,也需要有类似的转换。

9.3 关联数组

如果两个(或更多的)数组相应的元素具有相关的信息,那么这些数组称为关联数组。

假设需要知道学生课程的成绩以及 ID 数字。如果一个班级有 50 名学生,其 ID 包含 5 位数字。因为有 50 名学生,所以需要 50 个变量分别存储学生的成绩和 ID。可以声明两个数组: `int` 类型的 `studentId` 和 `char` 类型的 `courseGrade`,每一个数组有 50 个元素。而且, `studentId[0]` 和 `courseGrade[0]`

将会保存第一个学生的ID和课程成绩。studentId[1]和courseGrade[1]将会保存第二个学生的ID和课程成绩，等等。

语句：

```
int studentId[ 50];  
char courseGrade[ 50];
```

声明了这两个数组。

9.4 编程实例：代码检测

当信息加密后通过传输信道被传送时，它通常是作为比特序列——0和1加以传送的。由于传输信道的噪声，传送的信息可能遭到破坏。也就是说，在目的地接收到的信息与传送的信息是不相同的。一些比特可能发生了改变。有几种技术来检查所传送信息的有效性。一种技术是将同一信息发送两次。在目的地，两个信息副本逐个比特地进行比较。如果相应的比特是相同的，那么接收的信息就是正确的。

下面，编写一个程序来检查在目的地收到的信息是否是正确的。为了简单起见，假设对应于消息的密码是数字（0到9）序列，并且信息的最大长度是250个数字。而且，信息中的第一个数字是信息的长度。例如，如果密码是：

7 9 2 7 8 3 5 6

那么信息包含7位数字并传送两次。

上述信息按如下方式加以传送：

7 9 2 7 8 3 5 6 7 9 2 7 8 3 5 6

输入 包含密码及其副本的文件。

输出 如果代码接收无误，输出应包括密码、副本以及一条发送成功的消息。如下所示：

Code	Digit	Code	Digit	Copy
	9			9
	2			2
	7			7
	8			8
	3			3
	5			5
	6			6

Message transmitted OK.

问题分析和算法设计

因为必须对相应数字的密码及其副本进行比较，所以首先读入密码并且将它保存到数组中。然后读入副本的第一个数字，并且与密码的第一个数字相比较，等等。如果数字不相同，那么在相应数字后打印一条信息来表示这种情况，然后处理下一个数字。因为信息的最大长度是250，所以使用一个有250个元素的数组。密码以及副本的第一个数字表示代码的长度。该讨论翻译成下面的算法：

1. 打开输入和输出文件。
2. 如果输入文件不存在，那么退出程序。
3. 读入密码的长度。
4. 如果密码的长度大于250，那么终止程序，因为该程序中代码的最大长度是250。
5. 将密码读入并存储到数组中。

6. 读入副本的长度。

7. 如果密码的长度和副本的长度相同, 则比较代码, 否则打印错误信息。

为了简化函数 `main()`, 可以编写一个函数 `readCode()` 读入密码, 函数 `compareCode()` 来比较代码。下面讨论一下这两个函数。

readCode() 该函数首先读入密码的长度。如果密码的长度大于 250, 那么将布尔变量 `lengthCodeOk`, 即一个引用参数设置成 `false` 并且函数终止。`lengthCodeOk` 的值传递给调用函数并显示密码是否有效地读入。如果代码的长度小于 250, 那么 `readCode()` 函数读入并保存密码到一个数组中。因为输入信息存储在一个文件中并且文件在 `main()` 函数中打开, 所以对应于输入文件的输入流变量必须作为一个参数传递给该函数。而且, 在读入密码的长度和密码本身之后, `readCode()` 函数必须将这些值传递给 `main()` 函数。所以, 该函数有 4 个参数: 输入文件流变量、保存密码的数组、代码长度和布尔类型参数 `lengthCodeOk`。函数 `readCode()` 的定义如下:

```
void readCode(istream& infile, int list[], int& length,
              bool& lenCodeOk)
{
    int count;
    lenCodeOk = true;
    infile>>length; //get the length of the secret code
    if(length > maxCodeSize)
    {
        lenCodeOk = false;
        return;
    }
    for(count = 0; count < length; count++) //get the secret code
        infile>>list[count];
}
```

compareCode() 该函数将密码与其副本进行比较。所以, 它必须访问包含密码和密码长度的数组。密码的副本及其长度存储在输入文件中。这样, 对应于输入文件的输入流变量必须作为一个参数传递给该函数。而且, `compareCode()` 函数将密码与其副本进行比较并打印适当的信息。因为输出将保存在一个文件中, 所以对应于输出文件的输出流变量也必须作为一个参数传递给函数。所以该函数有 4 个参数: 输入文件流变量、输出文件流变量、包含密码的数组和密码的长度。该讨论可以翻译成下面的算法:

- a. 声明变量。
- b. 设置布尔变量 `codeOK` 为 `true`。
- c. 读入密码副本的长度。
- d. 如果密码的长度与其副本的长度不同, 那么输出错误信息并且终止该函数。
- e. 对于输入文件中的每一个数字:
 - e.1 读入密码副本的下一个数字。
 - e.2 从密码及其副本中输出相应的数字。
 - e.3 如果相应的数字不同, 那么输出一条错误信息并且将布尔变量 `codeOK` 设置为 `false`。
- f. 如果布尔变量 `codeOK` 是 `true`, 那么:

输出一条信息显示密码被正确地传输。

否则:

输出一条错误信息。

通过对算法的讨论,可以得到函数 `compareCode()` 的定义是:

```
void compareCode(ifstream& infile, ofstream& outfile,
                 int list[], int length)
{
    int length2;
    int digit;
    bool codeOk;
    int count;

    codeOk = true;
    infile>>length2;
    if(length != length2) //compare the lengths of the codes; Step d
    {
        cout<<"The original code and its copy are not of"
            <<" the same length."<<endl;
        return;
    }
    outfile<<"Code Digit    Code Digit Copy"<<endl;
    for(count = 0; count < length; count++)
    {
        infile>>digit;
        outfile<<setw(7)<<list[ count]<<setw(20)<<digit;
        if(digit != list[ count])
        {
            outfile<<"          code digit not the same"<<endl;
            codeOk = false;
        }
        else
            outfile<<endl;
    }

    if(codeOk)
        outfile<<"Message transmitted OK."<<endl;
    else
        outfile<<"Error in transmission. Retransmit!!"<<endl;
}
```

下面是函数 `main()` 的算法。

主要算法

1. 声明变量。
2. 打开文件。
3. 调用函数 `readCode()` 读入密码。
4. 如果 (密码的长度 ≤ 250)
 - 调用函数 `compareCode()` 比较代码。
 - 否则
 - 输出一条错误信息。

完整程序清单

```

//Program: Check Code
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

const int maxCodeSize = 250;

void readCode(ifstream& infile, int list[],
              int& length, bool& lenCodeOk);
void compareCode(ifstream& infile, ofstream& outfile,
                 int list[], int length);

int main()
{
    //Step 1
    int codeArray[maxCodeSize]; //array to store the secret code
    int codeLength;             //variable to store the length
                                //of the secret code
    bool lengthCodeOk;          //variable to indicate if the length of the
                                //secret code is less than or equal to 250
    ifstream incode;             //input file stream variable
    ofstream outcode;            //output file stream variable
    char inputfile[25];          //variable to store the name of the
                                //input file
    char outputfile[25];         //variable to store the name of the
                                //output file

    cout<<"Enter the input file name: ";
    cin>>inputfile;
    cout<<endl;

    //Step 2
    incode.open(inputfile);
    if(!incode)
    {
        cout<<"Cannot open the input file."<<endl;
        return 1;
    }

    cout<<"Enter the output file name: ";
    cin>>outputfile;
    cout<<endl;

    outcode.open(outputfile);

    readCode(incode, codeArray, codeLength, lengthCodeOk); //Step 3
    if(lengthCodeOk) //Step 4
        compareCode(incode, outcode, codeArray, codeLength);
    else
        cout<<"Length of the secret code must be <= "
              <<maxCodeSize<<<<endl; //Step 5

    incode.close();
    outcode.close();

    return 0;
}

void readCode(ifstream& infile, int list[], int& length,
              bool& lenCodeOk)
{

```

```

        int count;
        lenCodeOk = true;
        infile>>length; //get the length of the secret code
        if(length > maxCodeSize)
        {
            lenCodeOk = false;
            return;
        }
        for(count = 0; count < length; count++) //get the secret code
            infile>>list[ count];
    }

    void compareCode(ifstream& infile, ofstream& outfile,
                    int list[], int length)
    {
        int length2;
        int digit;
        bool codeOk;
        int count;

        codeOk = true;
        infile>>length2;
        if(length != length2)
        {
            cout<<"The original code and its copy are not of"
                <<" the same length."<<endl;
            return;
        }

        outfile<<"Code Digit    Code Digit Copy"<<endl;
        for(count = 0; count < length; count++)
        {
            infile>>digit;
            outfile<<setw(7)<<list[ count]<<setw(20)<<digit;
            if(digit != list[ count])
            {
                outfile<<"           coce digit not the same"<<endl;
                codeOk = false;
            }
            else
                outfile<<endl;
        }

        if(codeOk)
            outfile<<"Message transmitted OK."<<endl;
        else
            outfile<<"Error in transmission. Retransmit!!"<<endl;
    }
}

```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Enter the input file name: a:Ch9_SecretCodeData.txt
Enter the output file name: a:Ch9_SecretCodeOut.txt
输入文件中的数据: ( a:Ch9_SecretCodeData.txt )

```

```

7 9 2 7 8 3 5 6 7 9 2 7 8 3 5 6

```


输出文件: (a:Ch9_SecretCodeOut.txt)

Code	Digit	Code	Digit	Copy
	9			9
	2			2
	7			7
	8			8
	3			3
	5			5
	6			6

Message transmitted OK.

9.5 编程实例：文本处理

(行和字母计数) 编写程序读入一个给定的文本, 然后输出该文本, 并且打印文本中行的数量和每一个字母出现的次数 (不区分大小写)。

因为有 26 个字母, 所以使用一个有 26 个元素的数组来执行字母计数。还需要一个变量来存储行数。

文本存储在文件中, 假设为 a:textin.txt。输出将会保存在一个文件中, 假设为 a:textout.out。

输入 包含需要处理的文本文件。

输出 包含文本、行数和文本中字母出现的次数。

问题分析和算法设计

为了得到想要的输出, 必须输出原来的文本。也就是说, 如果文本包含任何空白字符, 那么它们也必须输出。而且, 程序必须计算文本中的行数。所以, 必须知道行在哪里结束, 也就是说必须处理换行符, 所以不能使用析取运算符来处理输入文件。因为还要处理字母数目, 所以需要使用 `get()` 函数来读入文本。

首先描述为了开发该程序所需要的变量, 这将会简化下面的讨论。

变量 (main() 函数) 程序需要存储字母数和行数, 所以需要一个变量来存储行数, 26 个变量来执行字母计数。使用一个有 26 个元素的数组来执行字母计数。还需要一个变量读入和存储每一个字符, 因为输入文件是逐个字符读入的。因为数据从输入文件读入并且输出被保存到一个文件中, 所以需要输入流变量来打开输入文件, 输出流变量打开一个输出文件。这些语句表明函数 `main()` (至少) 需要下面的变量:

```
int      lineCount;           //variable to store the line count
int      letterCount[ 26];    //array to store the letter count
char     ch;                  //variable to read a character
ifstream infile;              //input file stream variable
ofstream outfile;             //output file stream variable
```

声明中, `letterCount[0]` 存储 'A' 的数量, `letterCount[1]` 存储 'B' 的数量, 等等。很明显, 变量 `lineCount` 和数组 `letterCount` 必须初始化为 0。

该程序的主要算法是:

1. 声明变量。
2. 打开输入和输出文件。
3. 初始化变量。
4. 当输入文件中有大量数据时:
 - 4.1 对于一行中的每一个字符:

4.1.1 读和写字符。

4.1.2 将对应的字母数量加1。

4.2 将行的数目加1。

5. 输出行数和字母数。

6. 关闭文件。

为了简化函数 `main()`，可以将它分成4个函数：

- 函数 `initialize()`
- 函数 `copyText()`
- 函数 `characterCount()`
- 函数 `writeTotal()`

下面的部分详细地描述了每一个函数。然后在这些函数的帮助下，描述 `main()` 函数的算法。

`initialize()` 函数初始化变量 `lineCount` 和数组 `letterCount` 为0。所以它有两个参数：一个对应于变量 `lineCount`，另一个对应于数组 `letterCount`。很明显，对应于 `lineCount` 的参数必须是一个引用参数。该函数的定义是：

```
void initialize(int& lc, int list[])
{
    int j;
    lc = 0;
    for(j = 0; j < 26; j++)
        list[j] = 0;
}
```

`copyText()` 函数读入一行并且输出该行。只要发现非空白字符，它就会调用函数 `characterCount()` 来更新字母数目。很明显，该函数有4个参数：输入文件流变量、输出文件流变量、`char` 变量和更新字母个数的数组。

注意，函数 `copyText()` 不能执行字母计数，但是仍然可以将数组 `letterCount` 传递给它。采用这一步骤是因为该函数调用函数 `characterCount()`，它需要数组 `letterCount` 更新适当的字母数目。所以，必须将数组 `letterCount` 传递给 `copyText()` 函数，这样可以将数组传递给函数 `characterCount()`。

```
void copyText(ifstream& intext, ofstream& outtext, char& ch,
              int list[])
{
    while(ch != '\n')                //Process the entire line
    {
        outtext<<ch;                  //Output the character
        characterCount(ch, list);     //Call function character count
        intext.get(ch);               //Read the next character
    }
    outtext<<ch;                      //Output the newline character
}
```

`characterCount()` 函数将字母数目加1。为了将相应的字母数目加1，函数必须获取相应字母。所以，函数 `characterCount()` 有两个参数：`char` 变量和更新字母数目的数组。利用伪代码，该函数表示为：

- a. 将字母转化为大写字母。
- b. 找到对应于该字母的数组下标。

- c. 如果下标是有效的, 那么将相应的数目加1。在这一步必须保证字符是一个字母。由于只计算字母的数目, 所以其他字符, 例如逗号、连接号和逗点都被忽略了。

按照这一算法, 函数的定义是:

```
void characterCount(char ch, int list[])
{
    int index;

    ch = toupper(ch);                //Step a
    index = static_cast<int>(ch) - 65; //Step b
    if(0 <= index && index < 26)      //Step c
        list[index]++;
}
```

writeTotal() 函数输出行数和字母数。它有三个参数: 输出文件流、行数和输出字母数目的数组。该函数的定义是:

```
void writeTotal(ofstream& outtext, int lc, int list[])
{
    int index;

    outtext<<"The number of lines = "<<lc<<endl;
    for(index = 0; index < 26; index++)
        outtext<<static_cast<char>(index+65)<<" count = "
            <<list[index]<<endl;
}
```

现在描述 **main()** 函数的算法。

主要算法

1. 声明变量。
2. 打开输入输出文件。
3. 如果输入文件不存在, 那么退出程序。
4. 打开输出文件。
5. 初始化变量, 例如 **lineCount** 和数组 **letterCount**。
6. 读入第一个字符。
7. **while** (不是输入文件的末尾)
 - 7.1 调用函数 **copyText()**。
 - 7.2 将行数加1 (将变量 **lineCount** 加1)。
 - 7.3 读入下一个字符。
8. 输出行数和字符数。调用函数 **writeTotal()**。
9. 关闭文件。

完整程序清单

```
//Program: Line and letter count
#include <iostream>
#include <fstream>
#include <cctype>

using namespace std;

void initialize(int& lc, int list[]);
void copyText(ifstream& intext, ofstream& outtext, char& ch,
    int list[]);
```

```

void characterCount(char ch, int list[]);
void writeTotal(ofstream& outtext, int lc, int list[]);

int main()
{
    //Step 1; Declare variables
    int lineCount;
    int letterCount[26];
    char ch;
    ifstream infile;
    ofstream outfile;

    infile.open("a:textin.txt"); //Step 2
    if(!infile) //Step 3
    {
        cout<<"Cannot open input file."<<endl;
        return 1;
    }

    outfile.open("a:textout.out"); //Step 4
    initialize(lineCount, letterCount); //Step 5
    infile.get(ch); //Step 6
    while(infile) //Step 7
    {
        copyText(infile, outfile, ch, letterCount); //step 7.1
        lineCount++; //Step 7.2
        infile.get(ch); //Step 7.3
    }

    writeTotal(outfile, lineCount, letterCount); //Step 8
    infile.close(); //Step 9
    outfile.close(); //Step 9
    return 0;
}

void initialize(int& lc, int list[])
{
    int j;
    lc = 0;
    for(j = 0; j < 26; j++)
        list[j] = 0;
}

void copyText(ifstream& intext, ofstream& outtext, char& ch,
              int list[])
{
    while(ch != '\n') //Process the entire line
    {
        outtext<<ch; //Output the character
        characterCount(ch, list); //Call function character count
        intext.get(ch); //Read the next character
    }
    outtext<<ch; //Output the newline character
}

void characterCount(char ch, int list[])
{
    int index;
    ch = toupper(ch); //Step a

```

```
        index = static_cast<int> (ch) - 65; //Step b
        if(0 <= index && index < 26) //Step c
            list[index]++;
    }

    void writeTotal(ofstream& outtext, int lc, int list[])
    {
        int index;

        outtext<<endl<<endl;
        outtext<<"The number of lines = "<<lc<<endl;
        for(index = 0; index < 26; index++)
            outtext<<static_cast<char>(index+65)<<" count = "
                <<list[index]<<endl;
    }
```

9.6 小结

1. 如果某类型的变量一次只能存储一个值，那么该类型就是简单类型变量。
2. 在结构化数据类型中，每一个数据项是其他数据项的集合。
3. 数组是具有固定数量元素的结构化数据类型。每一个元素的数据类型都是相同的，而且可以使用在数组中的对应位置来访问这些元素。
4. 一维数组的元素是以列表的形式来排列的。
5. C++ 对数组越界不进行检查。
6. 在 C++ 中，数组下标是从 0 开始的。
7. 数组下标可以是任何值为非负整数的表达式。下标值必须小于数组的大小。
8. 除了字符数组（C 字符串）的输入/输出，数组不包含聚集操作。
9. 数组可以在声明期间初始化。如果初始部分小于数组的大小，那么剩余的元素将初始化为 0。
10. 数组地址是数组第一个元素的地址。例如，如果 list 是一维数组，那么 list 的地址是 list[0] 的地址。
11. 当声明一维数组作为形参时，通常忽略数组大小。如果指定一维数组的大小，那么在形参声明中，编译器将会忽略它。
12. 在函数调用语句中，将一个数组作为参数传递时，只能使用它的名字。
13. 作为函数调用的参数，数组只能作为引用参数来传递。
14. 因为只能作为引用参数被传递，所以当声明数组为形参时，不需要在数据类型后使用 & 符号。
15. 函数不能返回一个数组。
16. 尽管数组作为引用参数被传递，但是当声明数组作为形参时，仍然可以在数据类型前使用保留字 const 阻止函数修改数组。
17. 单独的数组元素可以作为参数传递给函数。
18. 在 C++ 中，字符串是用双引号括起来的一系列字符。
19. 在 C++ 中，C 字符串是以 null 结尾的。
20. 在 C++ 中，null 字符用 '\0' 来表示。
21. 在 ASCII 字符集中，null 字符的对照序列是 0。
22. C 字符串存储在字符数组中。

23. 字符数组在声明期间可以使用字符进行初始化。
24. C字符串的输入和输出是C++惟一允许的聚集操作
25. 头文件 `cstring` 包含了用于C字符串操作函数的详细声明。
26. 常用的C字符串操作函数包括: `strcpy()`、`strcmp()`和`strlen()`。
27. C字符串是逐个字符进行比较的。
28. 因为字符串保存在数组中, 所以字符串中的单个字符可以使用数组元素下标进行访问。
29. 关联数组用来保存相关的信息。

9.7 练习

1. 判断下面语句的对错。

- a. `double` 类型是简单数据类型。
- b. 一维数组是结构化数据类型的例子。
- c. 数组既可以作为值也可以作为引用参数传递给函数。
- d. 函数能返回数组。
- e. 数组的大小是在编译期决定的。
- f. 对 `int` 数组惟一允许的聚集操作是自加和自减操作。
- g. 下列语句更新了数组 `list` 中第5个元素的内容。

```
int list[10];  
list[5] = list[3] + list[2];
```

- h. 如果数组下标越界, 那么程序将会终止并且产生错误。
- i. 在C++中, 允许对字符串进行聚集操作。

j.

```
char name[16] = "John K. Miller";
```

声明 `name` 是一个有15个字符的数组, 因为字符串 "John K. Miller" 只有14个字符。

k.

```
char str = "Sunny Day";
```

声明 `str` 是一个未指定长度的字符串。

2. 给定声明:

```
char string15[16];
```

判断下面的语句是否有效。如果语句是无效的, 解释原因。

- a. `strcpy(string15, "Hello there");`
- b. `strlen(string15);`
- c. `string15 = "Jacksonville";`
- d. `cin >> string15;`
- e. `cout << string15;`
- f. `if(string15 >= "Nice day")`
 `cout << string15;`
- g. `string15[6] = 't';`

3. 给定声明:

```
char str1[15];
```

```
char str2[15] = "Good day";
```

判断下面的语句是否有效。如果语句是无效的, 解释原因。

- a. `str1 = str2;`
- b. `if(str1 == str2)`


```

        cout<<" Both strings are of the same length."<<endl;
c. if(strlen(str1) >= strlen(str2))
    str1 = str2;
d. if(strcmp(str1,str2) < 0)
    cout<<"str1 is less than str2."<<endl;

```

4. 给定声明:

```
char name[ 8] = "Shelly";
```

如果下面的语句不能输出 Shelly, 请解释原因。

- `cout<<name;`
- `for(int j = 0; j < 6; j++)`
`cout<<name[ j];`
- `int j = 0;`
`while(name[ j] != '\0')`
`cout<<name[ j++];`
- `int j = 0;`
`while(j < 8)`
`cout<<name[ j++];`

5. 给定声明:

```
char str1[ 21];
char str2[ 21];
```

- 编写一条 C++ 语句将 "Sunny Day" 存储到 str1 中。
 - 编写一条 C++ 语句将 str1 的长度保存到 int 变量 length 中。
 - 编写一条 C++ 语句将 name 的值复制到 str2 中。
 - 编写一条 C++ 代码, 如果 str1 小于或等于 str2 则输出 str1, 否则输出 str2。
6. 编写 C++ 语句完成下面的任务:
- 声明包含 15 个 int 类型元素的数组 alpha。
 - 输出数组 alpha 的第 10 个元素的值。
 - 将数组 alpha 第 5 个元素的值设置为 35。
 - 将数组 alpha 第 9 个元素的值设置为数组 alpha 的第 6 个和第 13 个元素的和。
 - 将数组 alpha 第 4 个元素的值设置为第 8 个元素值的 3 倍减去 57。
 - 输出数组 alpha, 每行打印 5 个元素。

7. 考虑下面的函数头:

```

void funcOne(int alpha[], int size);
int funcSum(int x, int y);
void funcTwo(const int alpha[], int beta[]);

```

和声明:

```

int list[ 50];
int Alist[ 60];
int num;

```

编写 C++ 语句完成下面的要求:

- 分别用实参 list 和 50 调用函数 funcOne()。
 - 函数 funcSum() 的参数为 50 和数组 list 的第 4 个元素, 输出其返回值。
 - 函数 funcSum() 的参数分别为数组 list 的第 13、10 个元素, 输出其返回值。
 - 分别用实参 list 和 Alist 调用函数 funcOne()。
8. 假设数组 list 包含 5 个 int 类型的元素。在下面的 C++ 代码执行后, 存储在 list 中的值是什么?
- ```

for(I = 0; I < 5; I++)
{
 list[I] = 2 * I + 5;
}

```

```

 if(I % 2 == 0)
 list[I] = list[I] - 3;
}

```

9. 假设数组list包含6个int类型的元素。在下面的C++代码执行后,存储在list中的值是什么?

```

list[0] = 5;
for(I = 1; I < 6; I++)
{
 list[I] = I * I + 5;
 if(I > 2)
 list[I] = 2 * list[I] - list[I-1];
}

```

10. 假设下面的声明:

```

char name[21];
char yourName[21];
char studentName[31];

```

判断下面的语句是否有效。如果语句是无效的,解释原因。

- a. cin>>name;
- b. cout<<studentName;
- c. yourName[ 0] = '\0';
- d. yourName = studentName;
- e. if(yourName == name)
 

```
 studentName = name;
```
- f. int x = strcmp(yourName,studentName);
- g. strcpy(studentName,Name);
- h. for(int j = 0; j < 21; j++)
 

```
 cout<<name[j];
```

11. 下面程序的输出是什么?

```

#include <iostream>
using namespace std;
int main()
{
 int count;
 int alpha[5];

 alpha[0] = 5;
 for(count = 1; count < 5; count++)
 {
 alpha[count] = 5 * count + 10;
 alpha[count -1] = alpha[count] -4;
 }
 cout<<"list elements: ";
 for(count = 0; count < 5; count++)
 cout<<alpha[count]<<" ";
 cout<<endl;
 return 0;
}

```

12. 下面程序的输出是什么?

```

#include <iostream>
using namespace std;

int main()
{
 int j;
 int one[5];
 int two[10];
}

```



```

 for(j = 0; j < 5; j++)
 one[j] = 5 * j + 3;
 cout<<"One contains: ";
 for(j = 0; j < 5; j++)
 cout<<one[j]<<" ";
 cout<<endl;
 for(j = 0; j < 5; j++)
 {
 two[j] = 2 * one[j] - 1;
 two[j + 5] = one[4 - j] + two[j];
 }
 cout<<"Two contains: ";
 for(j = 0; j < 10; j++)
 cout<<two[j]<<" ";
 cout<<endl;
 return 0;
}

```

## 9.8 编程练习

1. 编写一个 C++ 程序，声明包含 50 个 double 类型元素的数组 alpha。将前 25 个元素的值初始化为下标的平方，后 25 个元素的值初始化为下标的立方。输出数组，每一行打印 10 个元素。
2. 编写处理函数 `smallestIndex()`，将 int 数组及其大小作为参数，返回数组中最小元素的下标。编写一个程序来测试该函数。
3. 编写一个程序，处理一个包含学生测试分数的文件，其中测试分数在 0 ~ 200 之间。然后计算出分数在下列范围内的学生数量：0 ~ 24, 25 ~ 49, 50 ~ 74, 75 ~ 99, 100 ~ 124, 125 ~ 149, 150 ~ 174 和 175 ~ 200。输出分数范围和学生数量。（使用下面的数据来运行程序：76, 89, 150, 135, 200, 76, 12, 100, 150, 28, 178, 189, 167, 200, 175, 150, 87, 99, 129, 149, 176, 200, 87, 35, 157, 189。）
4. 在体操和游泳比赛中，每一个选手的分数按如下方式计算：去掉最高分和最低分，然后将剩余的分数加起来。编写一个程序允许用户输入 8 个裁判的分数并且输出选手的最终得分。将输出格式化到小数点后两位。（裁判的打分在 1 和 10 之间，1 是最低分，10 是最高分。）例如，如果分数是 9.2, 9.3, 9.0, 9.9, 9.5, 9.5, 9.6 和 9.8，那么该选手的得分是 56.90 分。
5. 编写一个程序提示用户输入一个字符串，然后用大写字母输出该字符串。（使用字符数组来存储字符串。）
6. 学生的 ID 和测试答案保存在一个文件中。文件中的第一个条目用下面的形式包含了测试的答案：  
 TFFTFFTTTTFFTFTFFT  
 文件中的其他每一个条目是学生的 ID、空格、学生的答案。例如，条目：  
 ABC54301 TFFTFFTT TFFTFFTTFT  
 表示学生的 ID 是 ABC54301，第 1 题的答案是正确的，第 2 题的答案是错误的，等等。学生没有回答第 9 题。测试有 20 个问题，班级的学生不少于 150 人。每一个正确的答案为两分，每一个错误的答案扣掉一分，没有回答得 0 分。编写一个程序来处理测试数据。输出应该是：学生的 ID、答案、测试分数和测试等级。假设有下面的等级范围：90% ~ 100%，A；80% ~ 89.99%，B；70% ~ 79.99%，C；60% ~ 69.99%，D；0% ~ 59.99%，F。
7. 编写一个程序，允许用户输入在一个当地选举中 5 位候选人的姓氏和每位候选人得到的票数。

然后程序应该输出每一位候选人的姓氏、候选人的得票数和候选人占总票数的百分比。程序还应该输出选举的获胜者。样例输出是：

| Candidate | Votes Received | % of Total Votes |
|-----------|----------------|------------------|
| Johnson   | 5000           | 25.91            |
| Miller    | 4000           | 20.72            |
| Duffy     | 6000           | 31.09            |
| Robinson  | 2500           | 12.95            |
| Ashtony   | 1800           | 9.33             |
| Total     | 19300          |                  |

The Winner of the Election is Duffy.

8. 编写一个程序允许用户输入学生的名字，后面是测试分数，然后输出（假设班级中最多人数是50）：
- (i) 班级平均分。
  - (ii) 所有分数低于班级平均分的学生名字并显示一条相应的信息。
  - (iii) 最高测试分数和所有获得最高测试分数的学生名字。

## 第10章 数组的应用和扩展

本章要点：

- 学习如何执行顺序搜索算法
- 学习如何使用选择排序算法对数组进行排序
- 学习如何执行二分搜索算法
- 学习如何对二维数组中的数据进行操作
- 了解多维数组

在第9章中介绍了数组是一种结构化的数据类型。如果数据类型都是相同的，那么数组是存储和处理这些数据的一种简便的方法。可以使用循环有效地进行输入/输出、初始化和其他操作。而且，可以使用一条单语句将整个数组的值作为一个参数来传递。下面的部分继续讨论一维数组，并且向读者演示如何更有效地使用它们处理顺序表。然后了解二维数组和多维数组。

### 10.1 顺序表处理

顺序表是相同类型值的集合。因为所有的值是相同类型的，所以数组能够方便地存储顺序表，特别是一维数组。顺序表的大小是顺序表中元素的数目。因为顺序表的大小需要增加和减少，所以用来存储顺序表的数组应该按照顺序表中最大元素数目进行声明。

对顺序表执行的基本操作包括：

1. 搜索顺序表并寻找给定的项。
2. 对顺序表进行排序。
3. 向顺序表中插入一项。
4. 从顺序表中删除一项。

下面的部分介绍搜索和排序算法。

#### 10.1.1 搜索

搜索顺序表中给定的项是顺序表最常见的操作之一。为了搜索顺序表，需要三条信息：

1. 顺序表，也就是包含顺序表的数组。
2. 顺序表的长度。
3. 要搜索的项。
4. 如果找到该项，然后报告 success 以及该项所在的位置。
5. 如果没找到该项，那么报告 failure。

为了执行4和5，可以编写下面的返回值函数：如果在顺序表中找到了搜索项，那么函数返回搜索项在顺序表中的位置；否则返回-1，表示搜索不成功。从该输出中可以明显地看出要编写的返回值函数有三个参数：

1. 包含顺序表的数组 `list`。
2. 顺序表的长度 `listLength`。(注意 `listLength ≤ arraySize`)
3. 要搜索的项 `searchItem`。

这里描述的算法称为顺序搜索或线性搜索。它从数组第一个元素顺序搜索数组，将 `searchItem` 与数组（也就是 `list`）中的元素进行比较并且重复进行搜索，直到找到搜索项或顺序表中没有与 `searchItem` 进行匹配的数据。

考虑如图 10.1 所示的包含 7 个元素的顺序表。

|      | [ 0 ] | [ 1 ] | [ 2 ] | [ 3 ] | [ 4 ] | [ 5 ] | [ 6 ] | [ 7 ] |
|------|-------|-------|-------|-------|-------|-------|-------|-------|
| list | 35    | 12    | 27    | 18    | 45    | 16    | 38    | ...   |

图 10.1 包含 7 个元素的顺序表

假设需要确定 27 是否在顺序表中。顺序搜索的原理如下：首先，将 27 与 `list[0]` 进行比较，也就是将 27 与 35 进行比较。因为 `list[0] ≠ 27`，所以将 27 再与 `list[1]` 进行比较（也就是与顺序表中的第二项，12 进行比较）。因为 `list[1] ≠ 27`，再将 27 与顺序表中的下一个元素进行比较，也就是 27 与 `list[2]` 进行比较。因为 `list[2] = 27`，所以搜索停止。这是一个成功的搜索。

现在来搜索 10。和前面一样，搜索从顺序表中的第一个元素，也就是 `list[0]` 开始。这次搜索项（也就是 10）与顺序表中的每一项进行比较。结果，在顺序表中没有发现与搜索项相同的数据。这是一个不成功的搜索。

只要在顺序表中找到与搜索项相同的元素，就必须停止搜索并且报告“success”。（在这种情况下，通常需要返回匹配项所在顺序表中的位置。）否则，在搜索项与顺序表中的每一个元素进行比较之后，必须停止搜索并且报告“failure”。

假设包含顺序表元素的数组名是 `list`。前面的讨论转换为下面的顺序搜索算法：

```
found is set to false;
for(loc = 0; loc < length; loc++)
 if (list[loc] is equal to searchItem)
 {
 found is set to true
 exit loop
 }
if(found)
 return loc;
else
 return -1;
```

下面的函数对顺序表执行顺序搜索。为了演示相应算法，假设顺序表元素是 `int` 类型。

```
int seqSearch(const int list[], int listLength, int searchItem)
{
 int loc;
 bool found = false;
 for(loc = 0; loc < listLength; loc++)
 if(list[loc] == searchItem)
 {
 found = true;
 break;
 }
 if(found)
```

```

 return loc;
 else
 return -1;
}

```

如果函数 `seqSearch()` 返回一个大于或等于 0 的值, 那么就是一个成功的搜索, 否则是一个不成功的搜索。

从该算法中可以看出, 通过将 `searchItem` 与 `list` 中的第一个元素进行比较开始搜索。如果 `searchItem` 等于 `list` 中的第一个元素那么就退出循环, 否则 `loc` 就会加 1 来指向 `list` 中的下一个元素。然后将 `searchItem` 与 `list` 中的下一个元素进行比较, 等等。

假设有一个包含 1000 个元素的顺序表。(参看图 10.2) 如果搜索项是顺序表中的第二项, 顺序搜索将进行两次比较确定搜索项是否在顺序表中。同样, 如果搜索项是顺序表中的第 900 项, 那么顺序查找将进行 900 次比较确定搜索项是否在顺序表中。如果搜索项不在顺序表中, 顺序搜索将进行 1000 次比较。

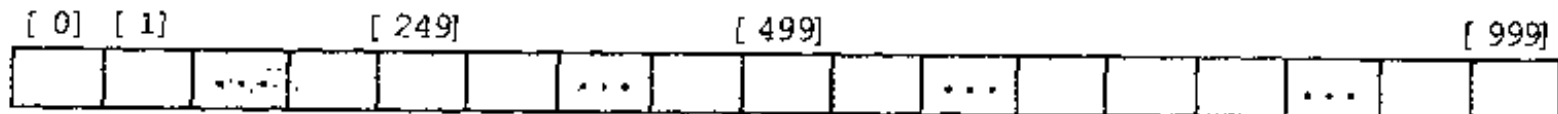


图 10.2 包含 1000 个元素的顺序表

因此, 如果 `searchItem` 在 `list` 的结尾, 那么将会进行多次比较来搜索 `searchItem`。而且, 如果 `searchItem` 不在 `list` 中, 那么 `searchItem` 要与 `list` 中的每一个元素进行比较。所以顺序查找对于大的顺序表来说效率并不高。而且可以证明, 顺序查找平均比较次数是顺序表大小的一半。所以, 对于大小为 1000 的顺序表来说, 顺序搜索平均要进行 500 次比较。

前面的查找算法假设顺序表是未排序的。如果顺序表被排序, 那么可以在一定程度上改进查找算法。下面来讨论怎样排序顺序表。

### 10.1.2 排序顺序表: 选择排序

文献中有很多可用的算法, 这里要讨论的排序算法称为选择排序。

在选择排序中, 通过选择顺序表中的一个元素并且将它移动到特定位置来重新排序顺序表。算法找到顺序表中未排序部分最小元素的位置, 然后将它移动到顺序表中未排序部分的头部。第一次需要查找整个顺序表中最小项的位置, 第二次从顺序表中的第二个元素开始查找最小项的位置, 等等。

作为一个例子, 假设有如图 10.3 所示的顺序表。

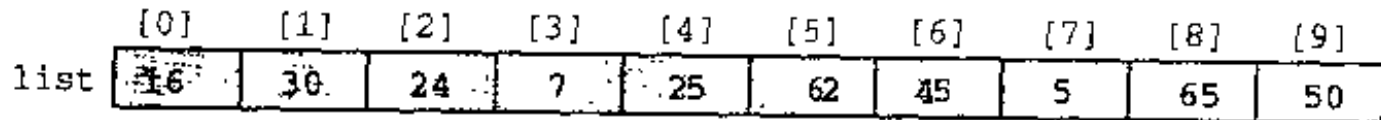


图 10.3 包含有 10 个元素的顺序表

开始时, 顺序表是未被排序的, 所以需要找到顺序表中的最小项, 也就是在位置 7, 如图 10.4 所示。

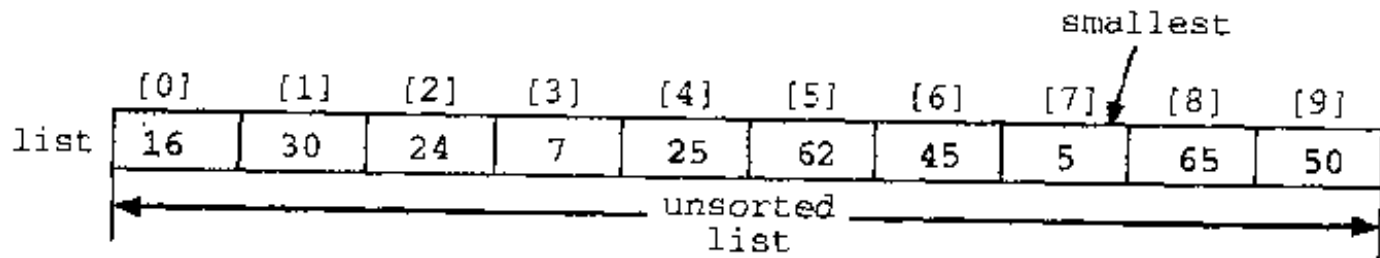


图 10.4 未排序顺序表的最小元素

因为是最小项, 所以它必须移动到位置0。所以将16 (即  $\text{list}[0]$ ) 与5 (即  $\text{list}[7]$ ) 进行交换, 如图10.5所示。

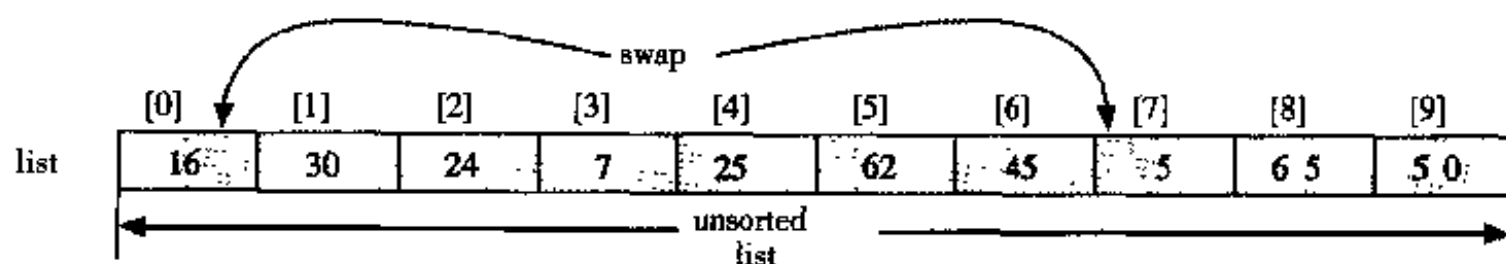


图10.5 交换元素  $\text{list}[0]$  和  $\text{list}[7]$

在交换这两个元素之后, 顺序表如图10.6所示。

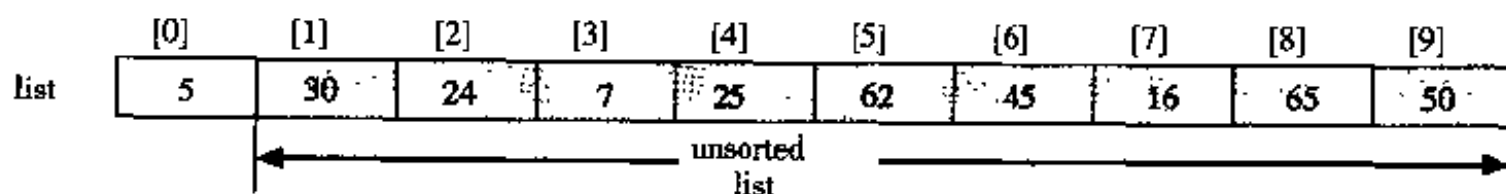


图10.6 交换元素  $\text{list}[0]$  和  $\text{list}[7]$  后的顺序表

现在未排序顺序表是  $\text{list}[1] \dots \text{list}[9]$ 。继续搜索顺序表未排序部分中最小的元素。最小元素在位置3, 如图10.7所示。

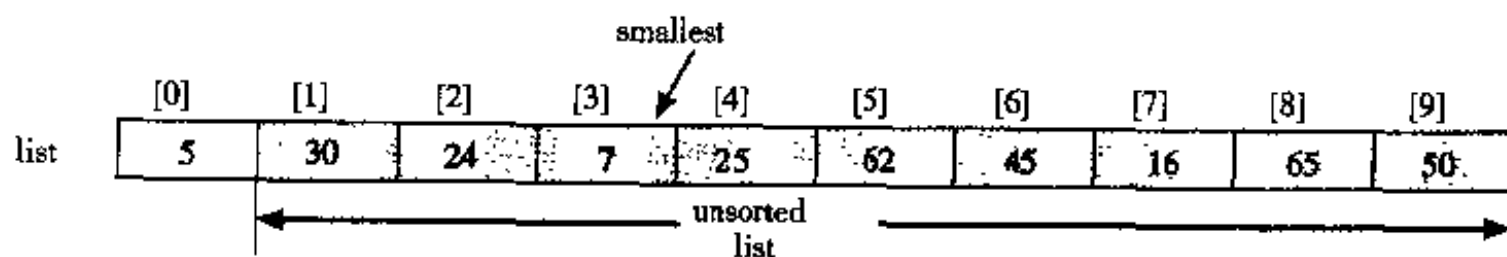


图10.7 顺序表未排序部分中最小的元素

因为顺序表未排序部分中最小的元素在位置3, 所以它必须移动到位置1。也就是将7 ( $\text{list}[3]$ ) 和30 ( $\text{list}[1]$ ) 进行交换, 如图10.8所示。

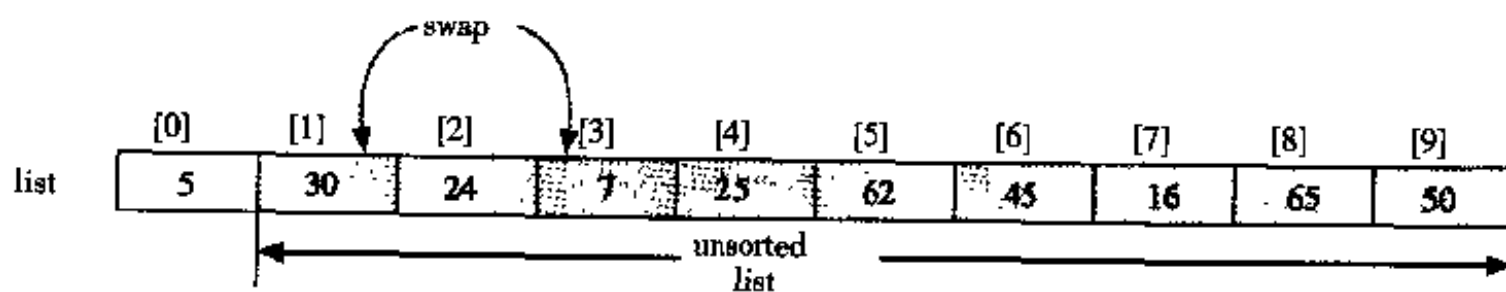


图10.8 交换  $\text{list}[1]$  和  $\text{list}[3]$

在交换  $\text{list}[1]$  和  $\text{list}[3]$  后, 顺序表如图10.9所示。

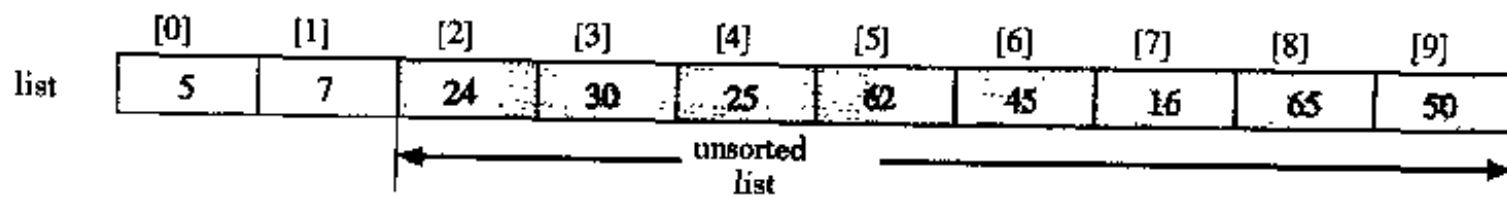


图10.9 交换  $\text{list}[1]$  和  $\text{list}[3]$  后的顺序表

现在未排序的顺序表是  $\text{list}[2] \dots \text{list}[9]$ 。重复寻找顺序表中未排序部分的最小元素 (位置), 并且将它移到顺序表未排序部分的开始。这样选择排序包含下面两个步骤:

在顺序表的未排序部分中:

a. 查找顺序表中未排序部分的最小元素位置。



b. 将最小元素移到顺序表未排序部分的开始。

最初, 整个顺序表 (即 `list[0]...list[length-1]`) 是没有排序的顺序表。在执行完一次步骤 a 和 b 之后, 未排序的顺序表是 `list[1]...list[length-1]`。再一次执行步骤 a 和 b 之后, 未排序的顺序表是 `list[2]...list[length-1]`, 等等。通过这种方法, 可以继续跟踪顺序表未排序的部分, 并且在 for 循环的帮助下重复步骤 a 和 b:

```
for(index = 0; index < length - 1; index++)
{
 a. Find the location, smallestIndex, of the smallest element in
 list[index]...list[length].
 b. Swap the smallest element with list[index]. That is, swap
 list[smallestIndex] with list[index].
}
```

通过第一次循环, 找到了 `list[0]...list[length - 1]` 中最小的元素, 并且将最小的元素与 `list[0]` 交换。通过第二次循环, 找到了 `list[1]...list[length - 1]` 中最小的元素, 并且将最小的元素与 `list[1]` 交换, 等等。

步骤 a 类似于查找顺序表中最大项下标的算法, 这在第 9 章讨论过。(参看第 9 章编程练习 2。) 这里需要查找顺序表中最小项的下标。步骤 a 的通常形式是:

```
smallestIndex = index; //assume first element is the smallest
for(minIndex = index + 1; minIndex < length; minIndex++)
 if(list[minIndex] < list[smallestIndex])
 smallestIndex = minIndex; //current element in the list is
 //smaller than the smallest so
 //far, so update smallestIndex
```

步骤 b 交换 `list[smallestIndex]` 与 `list[index]` 的内容。下面的语句完成这一任务:

```
temp = list[smallestIndex];
list[smallestIndex] = list[index];
list[index] = temp;
```

在编写选择排序函数之前, 让我们应用这一算法排序如图 10.10 所示的顺序表。

|      |       |       |       |       |       |       |
|------|-------|-------|-------|-------|-------|-------|
|      | [ 0 ] | [ 1 ] | [ 2 ] | [ 3 ] | [ 4 ] | [ 5 ] |
| list | 14    | 6     | 23    | 2     | 9     | 20    |

图 10.10 未排序的顺序表

将包含顺序表元素的数组称为 `list`。`list` 的长度为 6, 即 `length=6`。

循环 1: 排序 `list[0]...list[5]`。

步骤 a: 求 `list[0]...list[5]` 中最小元素的下标。(参看图 10.11。)

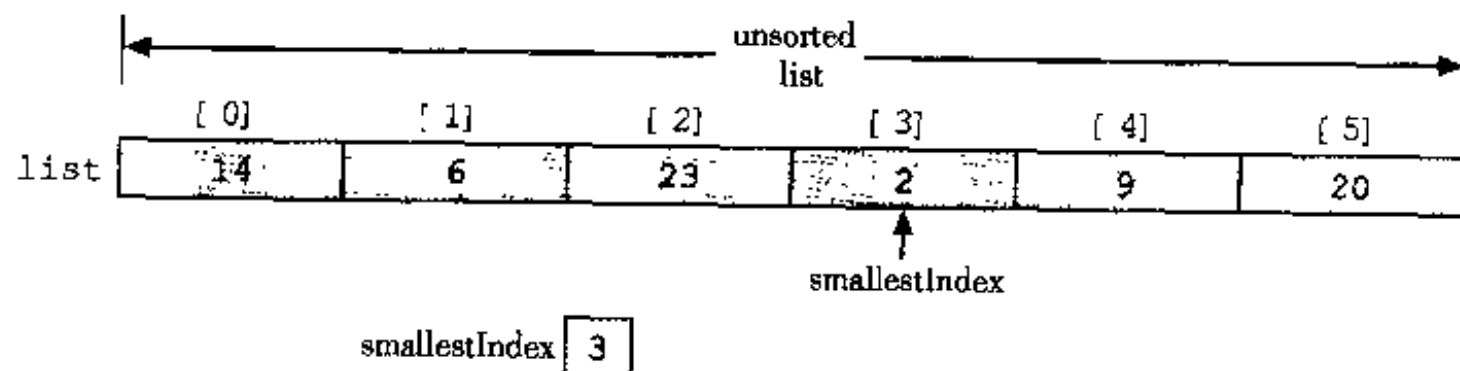


图 10.11 未排序顺序表中最小元素的位置是 3

步骤 b: 交换 `list[smallestIndex]` 与 `list[index]`, 也就是交换 `list[3]` 与 `list[0]`。图 10.12 显示了排序后的数组。

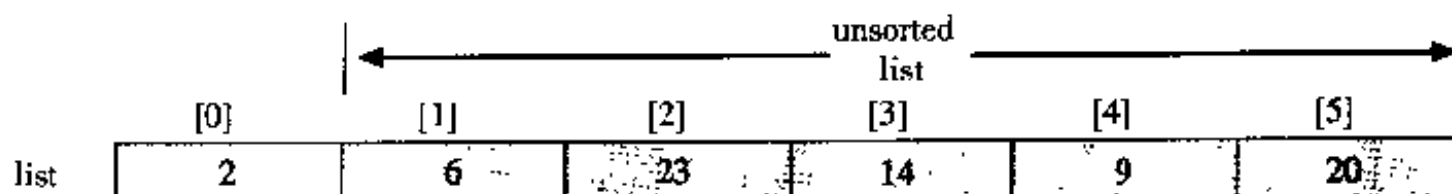


图 10.12 交换 list[3]与 list[0]之后的顺序表

循环 2: 排序 list[1]...list[5]。这里 index=1。

步骤 a: 求 list[1]...list[5]中最小元素的下标。(参看图 10.13。)

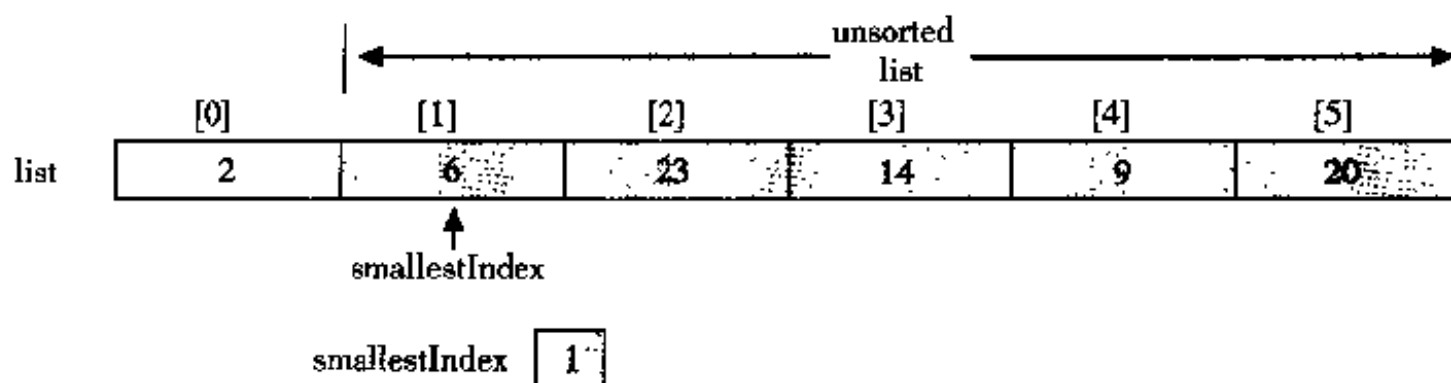


图 10.13 未排序顺序表中最小元素的位置是 1

步骤 b: 交换 list[smallestIndex]与 list[index], 也就是交换 list[1]与 list[1]。图 10.14 显示了排序后的数组。

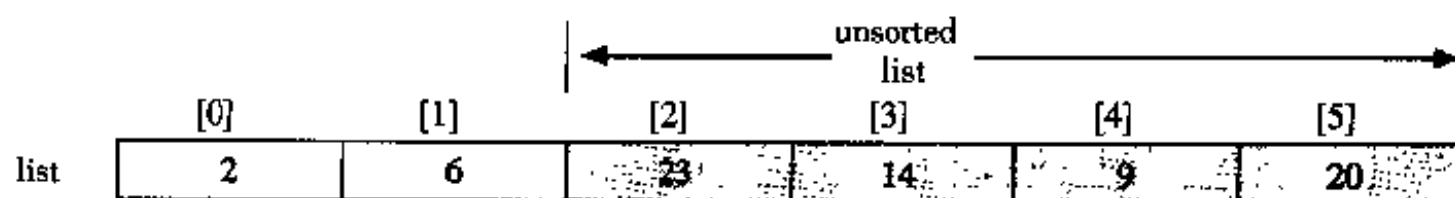


图 10.14 交换 list[1]与 list[1]之后的顺序表

循环 3: 排序 list[2]...list[5]。

步骤 a: 求 list[2]...list[5]中最小元素的下标。(参看图 10.15。)

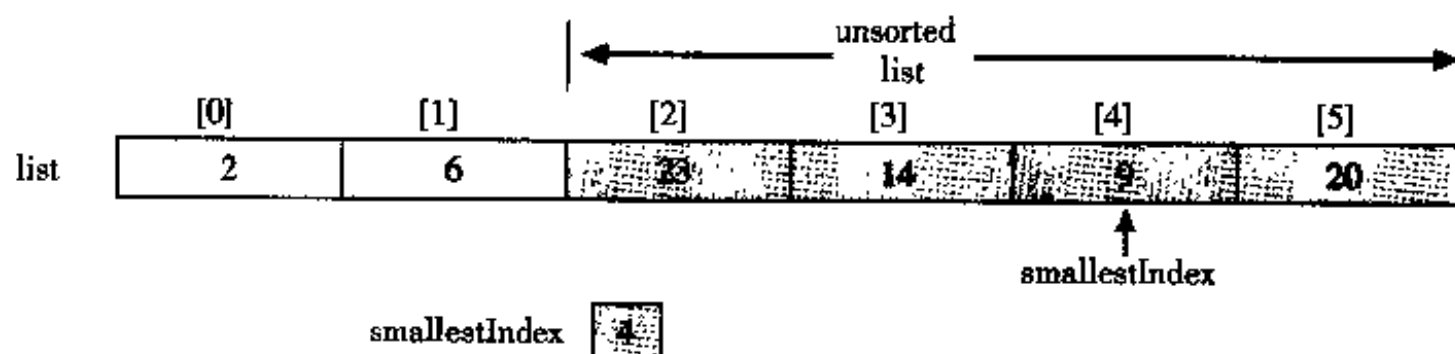


图 10.15 未排序顺序表中最小元素的位置是 4

步骤 b: 交换 list[smallestIndex]与 list[index], 也就是交换 list[4]与 list[2]。图 10.16 显示了排序后的数组。

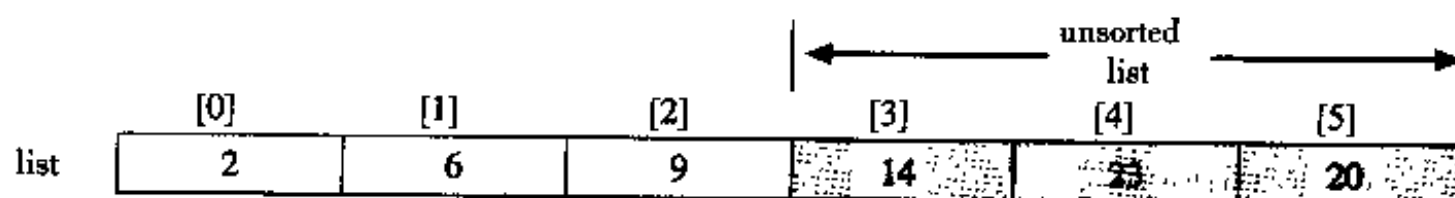


图 10.16 交换 list[4]与 list[2]之后的顺序表

循环 4: 排序 list[3]...list[5]。

步骤 a: 求 list[3]...list[5]中最小元素的下标。(参看图 10.17。)



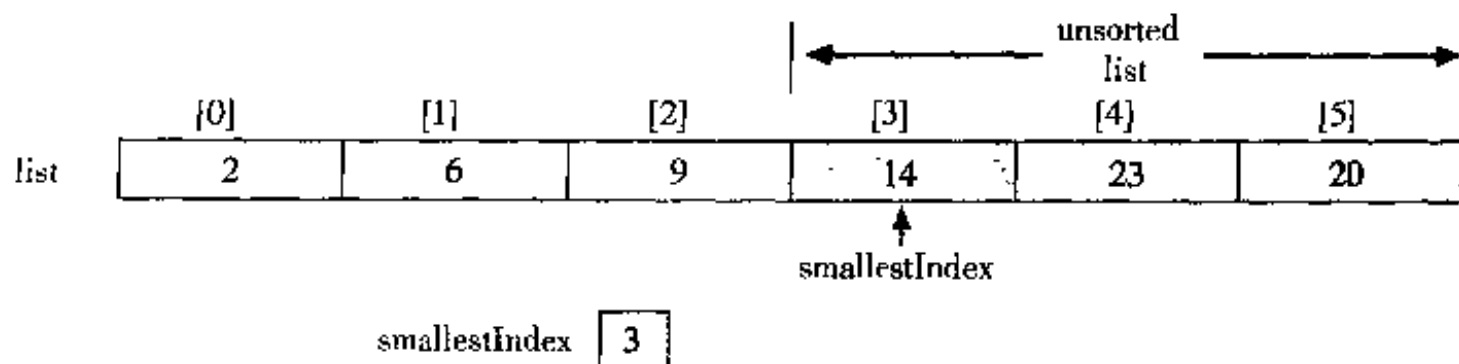
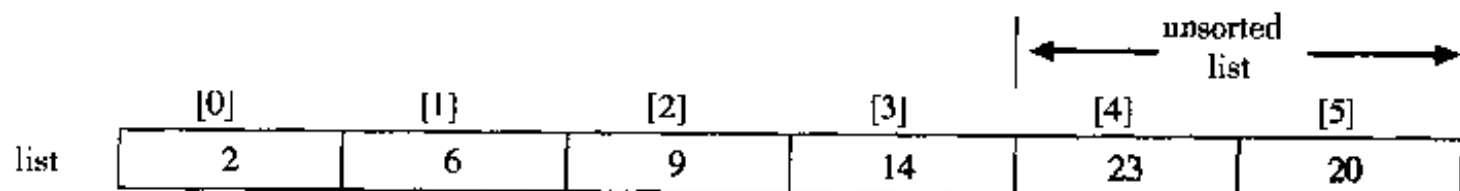


图 10.17 未排序顺序表中最小元素的位置是 3

步骤 b: 交换 `list[smallestIndex]` 与 `list[index]`, 也就是交换 `list[3]` 与 `list[3]`。图 10.18 显示了排序后的数组。

图 10.18 交换 `list[3]` 与 `list[3]` 之后的顺序表

循环 5: 排序 `list[4]...list[5]`。

步骤 a: 求 `list[4]...list[5]` 中最小元素的下标。(参看图 10.19。)

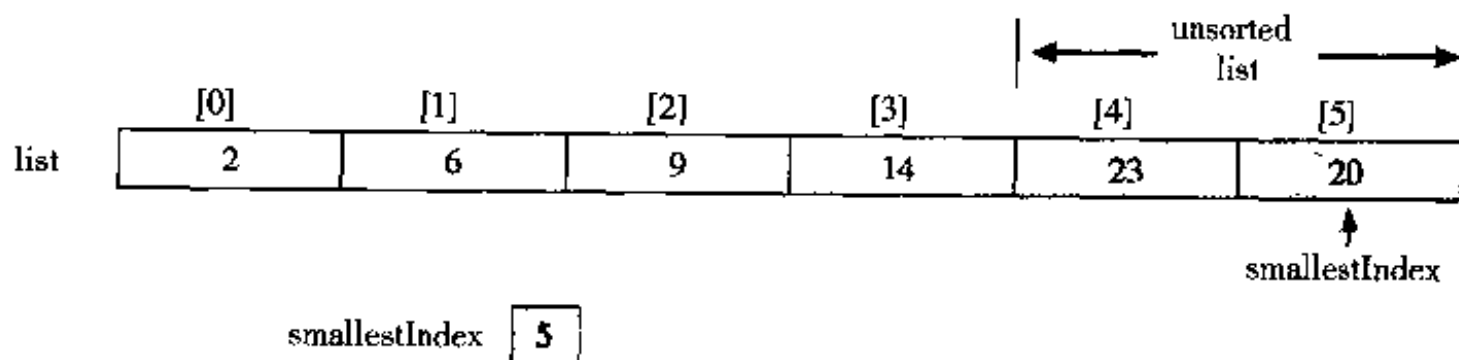
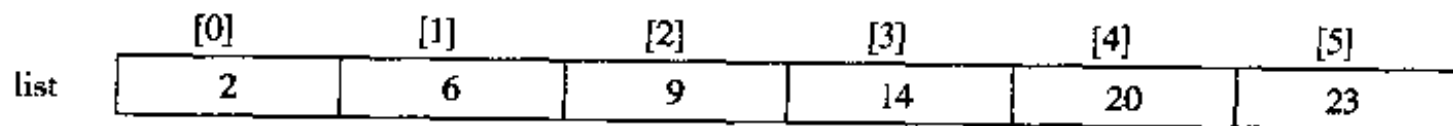


图 10.19 未排序顺序表中最小元素的位置是 5

步骤 b: 交换 `list[smallestIndex]` 与 `list[index]`, 也就是交换 `list[5]` 与 `list[4]`。图 10.20 显示了排序后的数组。

图 10.20 交换 `list[4]` 与 `list[5]` 之后的顺序表

在重复执行步骤 a 和 b 5 次之后, 顺序表排序完毕。注意顺序表的长度是 6, 但是程序只需要重复 5 次步骤 a 和 b, 也就是 `length-1` 次。

函数 `selectionSort()` 的定义是:

```
void selectionSort(int list[], int length)
{
 int index;
 int smallestIndex;
 int minIndex;
 int temp;

 for(index = 0; index < length - 1; index++)
 {
 //Step a
 smallestIndex = index;
```

```

 for(minIndex = index + 1; minIndex < length; minIndex++)
 if(list[minIndex] < list[smallestIndex])
 smallestIndex = minIndex;

 //Step b
 temp = list[smallestIndex];
 list[smallestIndex] = list[index];
 list[index] = temp;
 }
}

```

### 10.1.3 顺序查找排序表

现在,利用排序顺序表重新编写顺序查找。考虑如图10.21所示的包含7个元素的排序顺序表。

|      | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|
| list | 4   | 12  | 27  | 38  | 45  | 53  | 58  | ... |

图 10.21 包含7个元素的排序的顺序表

假设想确定38是否在顺序表中,从顺序表的第一个元素开始搜索,搜索自然会停在list[3]。现在假设想确定40是否在顺序表中,从顺序表的描述中可以看出,搜索不应该超过list[4],因为list[4]>=40。

从这两种情况中看出,当顺序表中的某一项大于或等于searchItem或者list中没有与searchItem相匹配的元素时,就可以停止搜索。对一个排序顺序表进行顺序搜索的通常形式为:

```

found is set to false;
for(loc = 0; loc < listLength; loc++)
 if (list[loc] is greater than or equal to searchItem)
 {
 found is set to true
 exit loop
 }

```

注意,如果list中的某个元素大于或等于搜索项时,那么循环将会停止(也就是found被设置为true)。实际上,只有当该项等于顺序表中的某个元素时,found才应该为true。所以必须在循环外正确设置found的值。下面的语句完成该项任务:

```

if(found)
 if(list[loc] == searchItem)
 return loc;
 else
 return -1;
else
 return -1;

```

现在完成顺序查找函数的定义并执行对一个排序表的查找。

```

int seqOrderdSearch(const int list[], int listLength, int searchItem)
{
 int loc; //Line 1
 bool found = false; //Line 2
 for(loc = 0; loc < listLength; loc++) //Line 3
 if(list[loc] >= searchItem) //Line 4
 {
 found = true; //Line 5
 }
}

```

```

 break; //Line 6
 }
 if(found) //Line 7
 if(list[loc] == searchItem) //Line 8
 return loc; //Line 9
 else //Line 10
 return -1; //Line 11
 else //Line 12
 return -1; //Line 13
}

```

下面描绘一下图 10.22 所示的算法。

|      | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|
| list | 4   | 18  | 29  | 35  | 44  | 59  | 65  | 98  |

图 10.22 有 8 个元素的排序顺序表

顺序表的长度是 8，所以 length=8。

假设要搜索的项是 35，所以 searchItem=35。在 for 循环执行前，loc=0 并且 found=false。然后描述 for 循环的执行，显示每一次循环执行期间 loc、found 以及特定元素的值：

| Iteration | loc | found | list[loc] | list[loc] >= searchItem        |
|-----------|-----|-------|-----------|--------------------------------|
| 1         | 0   | false | 4         | 4 >= 35 is false; loc = 1      |
| 2         | 1   | false | 18        | 18 >= 35 is false; loc = 2     |
| 3         | 2   | false | 29        | 29 >= 35 is false; loc = 3     |
| 4         | 3   | false | 35        | 35 >= 35 is true; found = true |

经过 4 次循环后，found 变为 true，break 语句退出循环。在 for 循环后，下面的语句执行：

```

if(found) //Line 7
 if(list[loc] == searchItem) //Line 8
 return loc; //Line 9
 else //Line 10
 return -1; //Line 11
else //Line 12
 return -1; //Line 13

```

因为 found 是 true，所以第 8 行的 if 语句执行。因为 list[loc]=35 并且 searchItem=35，所以表达式：

list[loc]==searchItem

的值为 true，而且第 9 行的 return 语句返回 loc 的值（3）。这是一个成功的查找。

现在查找 40，也就是 searchItem=40。在 for 循环执行前，loc=0 而且 found=false。如前所述，下面描述 for 循环的执行，显示每一次循环执行期间 loc、found 以及特定元素的值。

| Iteration | loc | found | list[loc] | list[loc] >= searchItem        |
|-----------|-----|-------|-----------|--------------------------------|
| 1         | 0   | false | 4         | 4 >= 40 is false; loc = 1      |
| 2         | 1   | false | 18        | 18 >= 40 is false; loc = 2     |
| 3         | 2   | false | 29        | 29 >= 40 is false; loc = 3     |
| 4         | 3   | false | 35        | 35 >= 40 is false; found = 4   |
| 5         | 4   | false | 44        | 44 >= 40 is true; found = true |

经过 5 次循环后，found 变为 true 并且 break 语句退出循环。在 for 循环后，下面的语句（第 7 行和第 13 行之间）执行：

```

if(found) //Line 7
 if(list[loc] == searchItem) //Line 8

```

```

 return loc; //Line 9
 else //Line 10
 return -1; //Line 11
else //Line 12
 return -1; //Line 13

```

因为found是true, 所以第8行的if语句执行。因为list[loc]=44并且searchItem=40, 所以表达式:

```
list[loc] == searchItem
```

的值为false, 而且跳过第9行的语句, 第11行的return语句执行, 它返回-1。这是一个不成功的查找。

可以证明为了确定查找项是否在顺序表中, 其平均查找次数为排序表大小的一半。也就是说, 对于包含1000个元素的顺序表(在成功的情况下), 顺序查找平均进行500次比较。

#### 10.1.4 二分法检索

顺序查找对于排序顺序表来说能执行得较好, 但是对于大型顺序表来说, 它仍然不是最有效的。一般情况下, 它要搜索一半的顺序表。然而, 如果顺序表被排序, 可以使用另一种称为二分法检索的检索算法。二分法检索比顺序检索要快得多。应用二分法检索的必要条件是顺序表必须经过排序。

二分法检索类似于字典搜索。首先检索项与顺序表中的中间元素进行比较, 如果搜索项小于顺序表的中间元素, 那么将搜索限制在顺序表的前半部分, 否则搜索顺序表的后半部分。

考虑下面的排序顺序表, 如图10.23所示。

|      |     |     |     |     |     |     |     |     |     |     |      |      |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
|      | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |
| list | 4   | 8   | 19  | 25  | 34  | 39  | 45  | 48  | 66  | 75  | 89   | 95   |

图 10.23 长度为12的顺序表

假设想确定75是否在顺序表中。开始时, 搜索范围是整个顺序表(参看图10.24)。

|      |             |     |     |     |     |     |     |     |     |     |      |      |
|------|-------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
|      | search list |     |     |     |     |     |     |     |     |     |      |      |
|      | [0]         | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |
| list | 4           | 8   | 19  | 25  | 34  | 39  | 45  | 48  | 66  | 75  | 89   | 95   |
|      |             |     |     |     |     | ↑   |     |     |     |     |      |      |
|      |             |     |     |     |     | mid |     |     |     |     |      |      |

图 10.24 搜索顺序表 list[0]...list[11]

首先将75与顺序表的中间元素list[5](即39)进行比较。因为 $75 \neq \text{list}[5]$ 并且 $75 > \text{list}[5]$ , 所以将搜索限制在顺序表list[6]...list[11]中, 如图10.25所示。

|      |     |     |     |     |     |     |             |     |     |     |      |      |
|------|-----|-----|-----|-----|-----|-----|-------------|-----|-----|-----|------|------|
|      |     |     |     |     |     |     | search list |     |     |     |      |      |
|      | [0] | [1] | [2] | [3] | [4] | [5] | [6]         | [7] | [8] | [9] | [10] | [11] |
| list | 4   | 8   | 19  | 25  | 34  | 39  | 45          | 48  | 66  | 75  | 89   | 95   |
|      |     |     |     |     |     | ↑   |             |     |     |     |      |      |
|      |     |     |     |     |     | mid |             |     |     |     |      |      |

图 10.25 搜索顺序表 list[6]...list[11]

对list[6]...list[11]重复该过程, 这时顺序表的长度为6。

因为需要不断地确定顺序表的中间元素, 所以二分法检索通常应用于数组顺序表。为了确定顺序表的中间元素, 将搜索顺序表的开始下标first与结尾下标last相加, 并且除以2来计算中间下标,

也就是  $mid = (first + last) / 2$ 。开始时  $first = 0$ , (因为 C++ 的数组下标从 0 开始并且长度代表顺序表中元素的数量)  $last = listLength - 1$ 。

下面的 C++ 函数执行二分法检索算法。如果在顺序表中找到搜索项, 就返回其位置。如果搜索项不在顺序表中, 就返回 -1。

```
int binarySearch(const int list[], int listLength, int searchItem)
{
 int first = 0;
 int last = listLength - 1;
 int mid;

 bool found = false;
 while(first <= last && !found)
 {
 mid = (first + last) / 2;

 if(list[mid] == searchItem)
 found = true;
 else
 if(list[mid] > searchItem)
 last = mid - 1;
 else
 first = mid + 1;
 }

 if(found)
 return mid;
 else
 return -1;
} //end binarySearch
```

在二分法检索算法中, 每通过一次循环就会进行两个关键字比较。惟一的例外就是成功的情况, 这种情况下, 通过上一次循环, 只进行了一项关键字比较。

下面使用如图 10.26 所示的顺序表来浏览一下二分法检索算法。

|      | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
| list | 4   | 8   | 19  | 25  | 34  | 39  | 45  | 48  | 66  | 75  | 89   | 95   |

图 10.26 用于二分法检索的排序顺序表

该顺序表的大小是 12, 也就是  $listLength = 12$ 。假设要搜索的项是 89, 即  $searchItem = 89$ 。在 while 循环执行前,  $first = 0, last = 11, found = false$ 。下面跟踪 while 循环的执行, 显示每一次重复执行期间  $first$ 、 $last$ 、 $mid$ 、 $found$  的值以及比较次数:

| Iteration | first | last | mid | list[mid] | Number of Key Comparisons |
|-----------|-------|------|-----|-----------|---------------------------|
| 1         | 0     | 11   | 5   | 39        | 2                         |
| 2         | 6     | 11   | 8   | 66        | 2                         |
| 3         | 9     | 11   | 10  | 89        | 1 (found is true)         |

该项在位置 10 被找到并且比较的总次数是 5。

下面, 在顺序表中搜索 34, 即  $searchItem = 34$ 。在 while 循环执行前,  $first = 0, last = 11, found = false$ 。与前面一样, 跟踪 while 循环的执行, 显示每一次循环执行期间  $first$ 、 $last$ 、 $mid$ 、 $found$  的值以及比较次数:

| Iteration | first | last | mid | list[mid] | Number of Key Comparisons |
|-----------|-------|------|-----|-----------|---------------------------|
| 1         | 0     | 11   | 5   | 39        | 2                         |

|   |   |   |   |    |                   |
|---|---|---|---|----|-------------------|
| 2 | 0 | 4 | 2 | 19 | 2                 |
| 3 | 3 | 4 | 3 | 25 | 2                 |
| 4 | 4 | 4 | 4 | 34 | 1 (found is true) |

该项在位置 4 被找到并且比较的总次数是 7。

现在搜索 22, 即 `searchItem=22`。在 `while` 循环执行前, `first=0, last=11, found=false`。跟踪 `while` 循环的执行, 显示每一次循环执行期间 `first`、`last`、`mid`、`found` 的值以及比较次数:

| Iteration | first | last | mid                                   | list[mid] | Number of Key Comparisons |
|-----------|-------|------|---------------------------------------|-----------|---------------------------|
| 1         | 0     | 11   | 5                                     | 39        | 2                         |
| 2         | 0     | 4    | 2                                     | 19        | 2                         |
| 3         | 3     | 4    | 3                                     | 25        | 2                         |
| 4         | 3     | 2    | The loop stops (because first > last) |           |                           |

这是一个不成功的搜索。比较的总次数是 6。

从这些对二分法检索跟踪中可以看出, 每经过一次循环, 就减少了子表大小的一半。也就是说循环下一次搜索子表的数量是上一次子表数量的一半。

### 二分法检索的性能

假设 `L` 是一个有 1000 个元素的排序顺序表并且想确定 `x` 是否在 `L` 中。因为 `L` 是排序的, 所以可以应用二分法检索来搜索 `x`。假设 `L` 如图 10.27 所示。

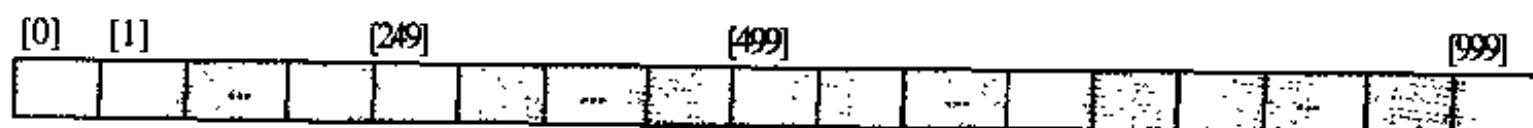


图 10.27 顺序表 L

`while` 循环的第一次循环在 `L[0]...L[999]` (它有 1000 个项) 中搜索 `x`。`while` 循环将 `x` 与 `L[499]` 进行比较。(参看图 10.28。)

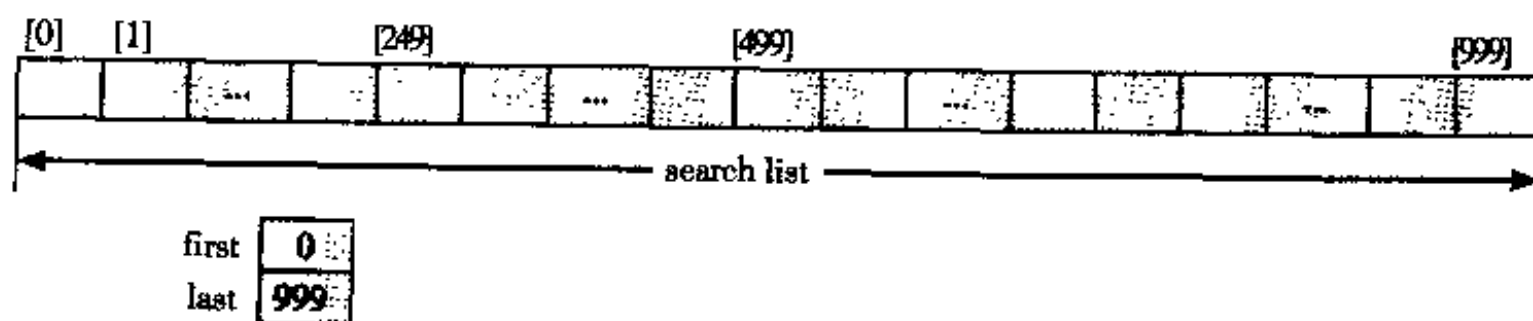


图 10.28 搜索顺序表

假设  $x \neq L[499]$ 。如果  $x < L[499]$ , 那么 `while` 循环下一次将会在 `L[0]...L[498]` 中查找 `x`, 否则 `while` 循环将会在 `L[500]...L[999]` 中查找 `x`。假设  $x < L[499]$ , `while` 循环在 `L[0]...L[498]` 中寻找 `x`, 它是一个有 499 项的顺序表, 如图 10.29 所示。

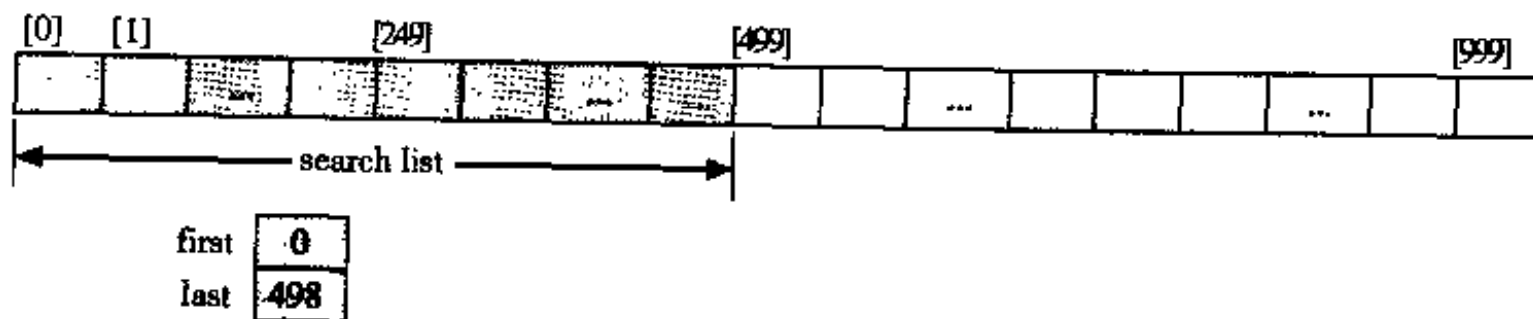


图 10.29 在第一次循环后搜索顺序表

`while` 循环将 `x` 与 `L[249]` 进行比较。假设  $x > L[249]$ , 那么 `while` 循环下一次将会在 `L[250]...L[498]` 中查找 `x`, 它是一个有 249 项的顺序表, 如图 10.30 所示。



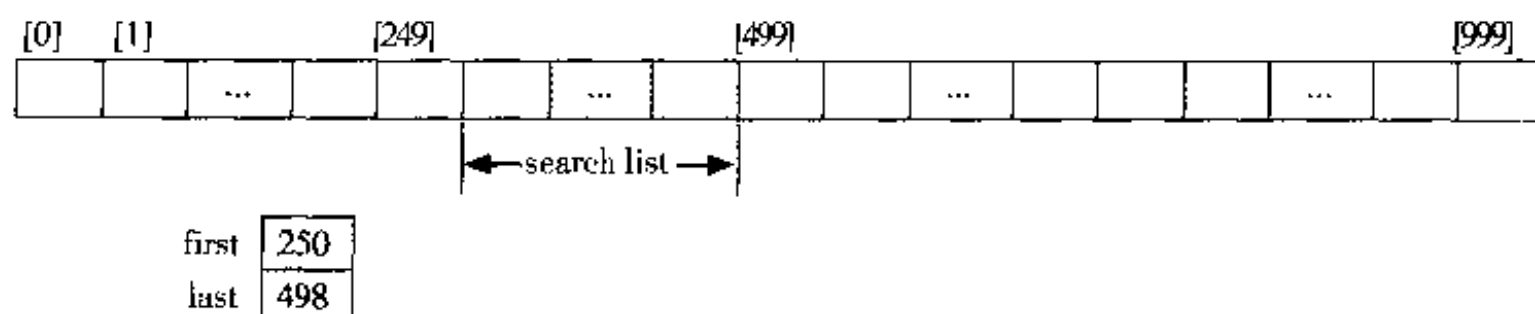


图 10.30 在第二次循环后搜索顺序表

while 循环的每一次执行都会将搜索顺序表的大小减少一半。因为  $1000 \approx 1024 = 2^{10}$ , 所以 while 循环最多执行 11 次来确定  $x$  是否在  $L$  中。因为 while 循环每一次执行进行两次 (项) 比较 (也就是  $x$  与  $L$  中的元素进行两次比较), 二分法检索最多进行 22 次比较来确定  $x$  是否在  $L$  中。另一方面, 顺序检索平均要进行 500 次比较来确定  $x$  是否在  $L$  中。

假设  $L$  的大小为 1 000 000。由于  $1\,000\,000 \approx 1\,048\,576 = 2^{20}$ , 所以在二分法检索中, while 循环最多执行 21 次来确定一个元素是否在  $L$  中。while 循环的每一次执行将会进行两次 (项) 比较。所以, 为了确定一个元素是否在  $L$  中, 二分法检索最多要进行 42 次比较。另一方面, 顺序检索平均要进行 500 000 次 (项) 比较来确定元素是否在  $L$  中。

通常, 如果  $L$  是一个大小为  $n$  的排序顺序表, 那么为了确定元素是否在  $L$  中, 二分法检索将最多进行  $2 \times \log_2 n + 2$  次 (项) 比较。

## 10.2 二维和多维数组

本章的剩余部分讨论二维数组和对多维数组操作的方法。

在前面的部分中学习了怎样有效地使用一维数组来进行顺序表处理。如果数据以顺序表的形式提供, 可以使用一维数组。然而, 有时数据以表格的形式提供。例如, 假设想知道本地代理商有多少特定颜色的小汽车在库存中。代理商销售 5 种不同颜色、6 种类型的小汽车。图 10.31 显示了样本数据。

| inStock    | [ Red ] | [ Brown ] | [ Black ] | [ White ] | [ Gray ] |
|------------|---------|-----------|-----------|-----------|----------|
| [ GM ]     | 10      | 7         | 12        | 10        | 4        |
| [ Ford ]   | 18      | 11        | 15        | 17        | 10       |
| [ Toyota ] | 12      | 10        | 9         | 5         | 12       |
| [ BMW ]    | 16      | 6         | 13        | 8         | 3        |
| [ Nissan ] | 10      | 7         | 12        | 6         | 4        |
| [ Volvo ]  | 9       | 4         | 7         | 12        | 11       |

图 10.31 表 inStock

可以用表格的形式查看这些数据。该表格有 30 项, 每项是一个整数。因为所有的表格项是相同类型的, 所以可以声明一个有 30 个 int 类型的一维数组。一维数组的前 5 个元素可以存储第一行数据, 下面 5 个元素存储表格的第二行数据, 等等。换言之, 可以在一个一维数组中模拟表格中给出的数据。

如果这样做,那么在一维数组中对数据进行操作的算法将会有点复杂,因为必须注意一行在哪里结束,另一行在哪里开始。还必须正确地计算特定元素的下标。C++使用二维数组简化了对表格中数据进行操纵的算法。该部分讨论怎样声明一个二维数组以及对二维数组中数据进行操作的方法。

**二维数组:**用行和列(二维)进行排列的、具有固定数目元素的集合,在该集合中,所有元素都是相同类型的。

声明一个二维数组的语法是:

```
dataType arrayName[intExp1][intExp2];
```

其中intExp1和intExp2是结果为正整数值的表达式。两个表达式intExp1和intExp2分别指定了数组行和列的数目。

语句:

```
double sales[10][5];
```

声明了一个10行5列的二维数组sales,其中每一个元素都是double类型的。与一维数组的情况相似,行被编号为0...9,列被编号为0...4。(参看图10.32。)

| sales | [0] | [1] | [2] | [3] | [4] |
|-------|-----|-----|-----|-----|-----|
| [0]   |     |     |     |     |     |
| [1]   |     |     |     |     |     |
| [2]   |     |     |     |     |     |
| [3]   |     |     |     |     |     |
| [4]   |     |     |     |     |     |
| [5]   |     |     |     |     |     |
| [6]   |     |     |     |     |     |
| [7]   |     |     |     |     |     |
| [8]   |     |     |     |     |     |
| [9]   |     |     |     |     |     |

图 10.32 二维数组 sales

### 10.2.1 访问数组元素

为了访问二维数组的元素,需要一对下标:分别用于行位置和列位置。

访问二维数组中元素的语法是:

```
arrayName[indexExp1][indexExp2];
```

其中indexExp1和indexExp2是非负整数值的表达式。indexExp1指定行位置,indexExp2指定列位置。

语句:

```
sales[5][3] =25.75;
```

将25.75存储到数组sales中行数为5、列数为3的元素中(即第6行、第4列)。(参看图10.33。)

假设:

```
int i=5;
int j=3;
```

那么前面的语句:

```
sales[5][3] =25.75;
```



等价于：

```
sales[i][j] = 25.75;
```

所以下标也可以是变量。

|       |     |     |       |     |     |
|-------|-----|-----|-------|-----|-----|
| sales | [0] | [1] | [2]   | [3] | [4] |
| [0]   |     |     |       |     |     |
| [1]   |     |     |       |     |     |
| [2]   |     |     |       |     |     |
| [3]   |     |     |       |     |     |
| [4]   |     |     |       |     |     |
| [5]   |     |     | 25.75 |     |     |
| [6]   |     |     |       |     |     |
| [7]   |     |     |       |     |     |
| [8]   |     |     |       |     |     |
| [9]   |     |     |       |     |     |

图 10.33 sales[5][3]

### 10.2.2 声明期间二维数组的初始化

与一维数组一样，当声明二维数组时，也可以进行初始化。下面的例子演示了这一概念。考虑下面的语句：

```
int board[4][3] = { { 2, 3, 1 },
 { 15, 25, 13 },
 { 20, 4, 7 },
 { 11, 18, 14 } };
```

该语句声明 board 是一个 4 行 3 列的二维数组。第 1 行元素是 2, 3 和 1；第 2 行元素是 15, 25 和 13；第 3 行元素是 20, 4 和 7；第 4 行元素是 11, 18 和 14。图 10.34 显示了数组 board。

|       |     |     |     |
|-------|-----|-----|-----|
| board | [0] | [1] | [2] |
| [0]   | 2   | 3   | 1   |
| [1]   | 15  | 25  | 13  |
| [2]   | 20  | 4   | 7   |
| [3]   | 11  | 18  | 14  |

图 10.34 二维数组 board

当声明二维数组时，为了对其进行初始化：

1. 每一行的元素用花括号包含起来并且用逗号分开。
2. 所有的行用花括号括起来。
3. 对于数字数组，如果没有指定一行中所有的元素，那么未被指定的元素初始化为 0。在这种情况下，可以指定一个值从而初始化一行。

### 10.2.3 二维数组和枚举类型

可以对数组下标使用枚举类型。考虑下面的语句：

```
const int rows = 6;
const int columns = 5;
```

```
enum carType{ GM, Ford, Toyota, BMW, Nissan, Volvo};
enum colorType{ Red, Brown, Black, White, Gray};

int inStock[rows][columns];
```

语句定义了 carType 和 colorType 枚举类型, 然后声明 inStock 是一个6行5列的二维数组。假设 inStock 中的每一行对应一种汽车类型, 每一列对应一种颜色类型。也就是说, 第1行对应于汽车类型 GM, 第2行对应于汽车类型 Ford, 等等。同样, 第1列对应于颜色类型 Red, 第2列对应于颜色类型 Brown, 等等。假设 inStock 中的每一项代表特定类型的汽车数量和颜色。(参看图 10.35。)

| inStock  | [Red] | [Brown] | [Black] | [White] | [Gray] |
|----------|-------|---------|---------|---------|--------|
| [GM]     |       |         |         |         |        |
| [Ford]   |       |         |         |         |        |
| [Toyota] |       |         |         |         |        |
| [BMW]    |       |         |         |         |        |
| [Nissan] |       |         |         |         |        |
| [Volvo]  |       |         |         |         |        |

图 10.35 二维数组 inStock

语句:

```
inStock[1][3] =15;
```

等同于下面的语句 (参看图 10.36):

```
inStock[Ford][White] =15;
```

| inStock  | [Red] | [Brown] | [Black] | [White] | [Gray] |
|----------|-------|---------|---------|---------|--------|
| [GM]     |       |         |         |         |        |
| [Ford]   |       |         |         | 15      |        |
| [Toyota] |       |         |         |         |        |
| [BMW]    |       |         |         |         |        |
| [Nissan] |       |         |         |         |        |
| [Volvo]  |       |         |         |         |        |

图 10.36 inStock[Ford][White]

第2行语句简单地传送了信息, 也就是将 White Ford 的数量设置为 15。该例子演示了枚举类型可以使程序具有可读性并易于管理。

### 10.2.4 处理二维数组

有三种方法可以处理二维数组：

1. 处理整个数组。
2. 处理数组特定的行，称为行处理。
3. 处理数组特定的列，称为列处理。

初始化和输出数组是处理整个二维数组的例子。求一行（列）中最大的元素或一行（列）的和是行（列）处理的例子。使用下面的声明来进行讨论：

```
const int rows = 7; //this can be set to any number
const int columns = 6; //this can be set to any number
int matrix[rows][columns];
int row;
int col;
int sum;
int largest;
int temp;
```

图 10.37 显示了数组 matrix。

| matrix | [0] | [1] | [2] | [3] | [4] | [5] |
|--------|-----|-----|-----|-----|-----|-----|
| [0]    |     |     |     |     |     |     |
| [1]    |     |     |     |     |     |     |
| [2]    |     |     |     |     |     |     |
| [3]    |     |     |     |     |     |     |
| [4]    |     |     |     |     |     |     |
| [5]    |     |     |     |     |     |     |
| [6]    |     |     |     |     |     |     |

图 10.37 二维数组 matrix

因为二维数组的元素具有相同类型，所以任何行或列的元素都属于相同类型。这意味着二维数组的每一行和每一列是一个一维数组。所以，当处理二维数组特定的行或列时，可以采用与处理一维数组类似的算法。

假设需要处理 matrix 下标为 5 的行，（matrix 的第 6 行）。该行的元素是：

```
matrix[5][0],matrix[5][1],matrix[5][2],matrix[5][3],matrix[5][4],matrix[5][5]
```

在这些元素中第 1 个下标（行位置）固定为 5。第 2 个下标（列位置）范围从 0 到 5。所以，可以使用下面的 for 循环来处理行数 5：

```
for (col=0;col<columns;col++)
 process matrix[5][col]
```

该 for 循环等价于下面的 for 循环：

```
row=5
for(col=0;col<columns;col++)
 process matrix[row][col]
```

同样, 假想处理 matrix 下标为 2 的列, 也就是 matrix 的第 3 列。该列的元素是:

```
matrix[0][2], matrix[1][2], matrix[2][2], matrix[3][2], matrix[4][2],
matrix[5][2], matrix[6][2]
```

这里第 2 个下标 (列位置) 固定为 2。第 1 个下标 (行位置) 范围从 0 到 6。这种情况下, 可以使用下面的 for 循环来处理 matrix 下标为 2 的列:

```
for (row=0;row<rows;row++)
 process matrix[row][2]
```

该 for 循环等价于下面的 for 循环:

```
col=2;
for (row=0;row<rows;row++)
 process matrix[row][col]
```

下面讨论特定的处理算法。

### 初始化

假想将下标为 4 的行初始化为 0, 也就是第 5 行初始化为 0。正如前面所解释的, 下面的 for 循环将下标为 4 的行初始化为 0:

```
row=4;
for (col=0;col<columns;col++)
 matrix[row][col] =0;
```

如果想将数组 matrix 初始化为 0, 那么还可以将第 1 个下标 (行的位置) 置于一个循环中。通过使用下面的嵌套 for 循环, 可以将 matrix 的每一个元素初始化为 0:

```
for (row=0;row<rows;row++)
 for (col=0;col<columns;col++)
 matrix[row][col] =0;
```

### 输出

通过使用一个嵌套 for 循环, 可以输出 matrix 的元素。下面的嵌套 for 循环打印 matrix 的元素, 每行打印一行的元素:

```
for (row=0;row<rows;row++)
{
 for (col=0;col<columns;col++)
 cout<<setw(5)<<matrix[row][col] <<" ";
 cout<<endl;
}
```

### 输入

下面的 for 循环将数据输入到下标为 4 的行中, 也就是 matrix 的第 5 行:

```
row=4;
for (col=0;col<columns;col++)
 cin>>matrix[row][col];
```

与前面一样, 通过在循环中设置行数字, 可以将数据输入给 matrix 的每一个元素。下面的 for 循环将数据输入到 matrix 的每一个元素中:

```

for(row=0;row<rows;row++)
 for(col=0;col<columns;col++)
 cin>>matrix[row][col] ;

```

### 行求和

下面的 for 循环计算 matrix 中第 5 行的总和:

```

sum=0;
row=4;
for(col=0;col<columns;col++)
 sum=sum+matrix[row][col] ;

```

通过在循环中设置行数字,可以分别计算每一行的和。下面是计算各行累加和的 C++ 代码:

```

//Sum of each individual row
for(row = 0; row < rows; row++)
{
 sum = 0;
 for(col = 0; col < columns; col++)
 sum = sum + matrix[row][col] ;

 cout<<"Sum of row "<<row+1<<" = "<<sum<<endl;
}

```

### 列求和

与行求和的情况一样,下面的嵌套 for 循环计算每一个单独列的和:

```

//Sum of each individual column
for(col = 0; col < columns; col++)
{
 sum = 0;
 for(row = 0; row < rows; row++)
 sum = sum + matrix[row][col] ;

 cout<<"Sum of column "<<col+1<<" = "<<sum<<endl;
}

```

### 每一行和每一列中的最大元素

对二维数组其他可能的操作是计算每一行和每一列中的最大元素以及计算两个对角线的和。下面给出执行这些操作的 C++ 代码。

下面的 for 循环确定下标为 4 的行中的最大元素:

```

row = 4;
largest = matrix[row][0] ; //assume that the first element of the
 //row is the largest
for(col = 1; col < columns; col++)
 if(largest < matrix[row][col])
 largest = matrix[row][col] ;

```

下面的 C++ 代码确定了每一行和每一列中的最大元素:

```

//Largest element in each row
for(row = 0; row < rows; row++)
{
 largest = matrix[row][0] ; //assume that the first element
 //of the row is the largest
 for(col = 1; col < columns; col++)
 if(largest < matrix[row][col])
 largest = matrix[row][col] ;
}

```

```

 cout<<"Largest element of row "<<row+1<<" = "<<largest<<endl;
}
//Largest element in each column
for(col = 0; col < columns; col++)
{
 largest = matrix[0][col]; //assume that the first element of
 //the column is the largest
 for(rcw = 1; row < rows; row++)
 if(largest < matrix[row][col])
 largest = matrix[row][col];

 cout<<"Largest element of col "<<col+1<<" = "<<largest<<endl;
}

```

### 倒转对角线

假设 *matrix* 是一个方阵二维数组，也就是说行数和列数是相同的。这样 *matrix* 有一个主对角线和一个对等的对角线。假设有下面的定义：

```

const int rows=4;
const int columns=4;

```

*matrix* 主对角线的元素是 *matrix*[0][0]、*matrix*[1][1]、*matrix*[2][2]和 *matrix*[3][3]。对等对角线的元素是 *matrix*[0][3]、*matrix*[1][2]、*matrix*[2][1]和 *matrix*[3][0]。

假设数组 *matrix* 如图 10.38 所示。

| <i>matrix</i> | [0] | [1] | [2] | [3] |
|---------------|-----|-----|-----|-----|
| [0]           | 1   | 8   | 10  | 11  |
| [1]           | 34  | 2   | 12  | 45  |
| [2]           | 0   | 13  | 3   | 20  |
| [3]           | 14  | 35  | 56  | 4   |

图 10.38 二维数组 *matrix*

在倒转两个对角线之后，数组 *matrix* 如图 10.39 所示。

| <i>matrix</i> | [0] | [1] | [2] | [3] |
|---------------|-----|-----|-----|-----|
| [0]           | 4   | 8   | 10  | 14  |
| [1]           | 34  | 3   | 13  | 45  |
| [2]           | 0   | 12  | 2   | 20  |
| [3]           | 11  | 35  | 56  | 1   |

图 10.39 在倒转两个对角线后的数组 *matrix*



很明显,为了倒转主对角线,需要下列步骤:

1. 交换 `matrix[0][0]` 与 `matrix[3][3]`。
2. 交换 `matrix[1][1]` 与 `matrix[2][2]`。

为了倒转对等对角线,需要下列步骤:

1. 交换 `matrix[0][3]` 与 `matrix[3][0]`。
2. 交换 `matrix[1][2]` 与 `matrix[2][1]`。

下面的 for 循环将对角线倒转:

```
//Reverse the main diagonal
for(row = 0; row < rows / 2; row++)
{
 temp = matrix[row][row];
 matrix[row][row] = matrix[rows-1-row][rows-1-row];
 matrix[rows-1-row][rows-1-row] = temp;
}
//Reverse the opposite diagonal
for(row = 0; row < rows / 2; row++)
{
 temp = matrix[row][rows-1-row];
 matrix[row][rows-1-row] = matrix[rows-1-row][row];
 matrix[rows-1-row][row] = temp;
}
```

注意: 该 C++ 代码倒转了一个方阵二维数组的对角线, 二维数组与一维数组的处理方式类似。

### 10.2.5 将二维数组作为参数传递给函数

二维数组能作为一个参数传递给函数并且能通过引用参数加以传递。地址(实际参数第1个元素的地址)被传递给形参。如果 `matrix` 是二维数组的名字, 那么 `matrix[0][0]` 是 `matrix` 的第1个元素。

当在计算机的内存中排列二维数组时, C++ 使用行优先的形式。即第1行首先排列, 然后是第2行, 第3行, 等等。

对于一维数组的情况, 当声明它作为一个形参时, 通常忽略数组的大小。因为 C++ 用行顺序的形式来存储二维数组, 所以为了直接计算一个元素的地址, 编译器必须知道一行在哪里结束, 下一行从哪里开始。这样, 当声明二维数组作为形参时, 可以忽略第一维的大小, 但是不能忽略第二维的大小, 也就是说必须指定列的数目。

考虑下面的函数定义:

```
void example(int table[][5], int rowsize)
{
 .
 .
 .
}
```

该函数将一个未指定数目的行, 但是有5列的二维数组作为一个参数。在函数调用期间, 实际参数列的数目必须与形参的列数相匹配。

### 10.2.6 字符串数组

假设需要执行一个操作,例如以字母顺序排列姓名列表。因为每一个名字是一个字符串,所以存储姓名的简单方法是使用一个数组。C++中的字符串可以使用数据类型 `string` 或字符数组(C字符串)进行操作。

而且,在一些编译器中,数据类型 `string` 在标准C++中不是可用的。该部分演示了两种方法来对字符串列表进行操作。

#### 字符串数组和 `string` 类型

使用数据类型 `string` 来处理字符串顺序表是很直观的。假设顺序表最多包含100个名字。可以声明包含100个元素 `string` 类型的数组,如下所示:

```
string list[100];
```

可以对 `string` 类型的值进行基本的操作,例如赋值、比较和输入/输出。所以, `list` 中的数据可以像第9章和本章的第一部分讨论的一维数组一样进行处理。

#### 字符串数组和C字符串(字符数组)

假设顺序表中最大字符串(例如名字)长度是15个字符,并且顺序表有100个字符串。可以声明如下的100行、16列的二维数组(参看图10.40):

```
char list[100][16];
```

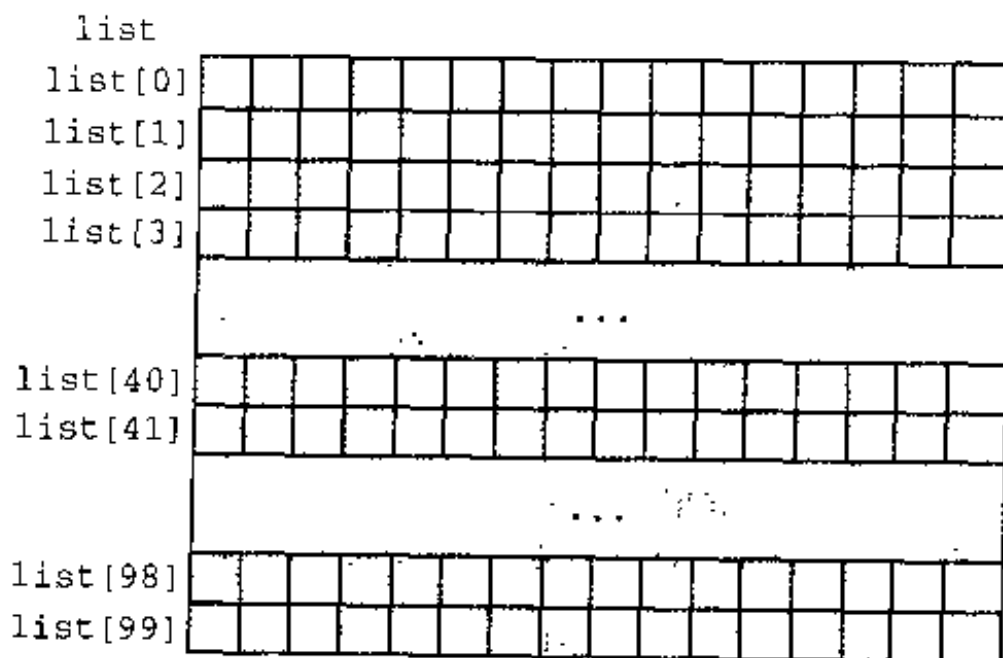


图 10.40 字符串数组 `list`

现在对于每一个变量 `j`,  $0 \leq j \leq 99$ , `list[j]` 的长度最多是15。下面的语句将 "Snow White" 存储到 `list[1]` 中(参看图10.41):

```
strcpy(list[1], "Snow White");
```

假设需要读入并存储数据到 `list` 中并且每行只有一项。下面的 `for` 循环完成该任务:

```
for(j=0; j<100; j++)
 cin.get(list[j], 16);
```

下面的 `for` 循环输出每行字符串:

```
for(j=0; j<100; j++)
 cout<<list[j]<<endl;
```

也可以使用其他字符串函数(例如 `strcmp()` 和 `strlen()`)和 `for` 循环来处理 `list`。



**注意：**string 数据类型包含一些其他操作，例如赋值、连接和相应的关系操作。如果使用标准 C++ 头文件并且数据类型 string 在编译器中是可用的，推荐使用数据类型 string 来处理的字符串顺序表。

|          |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
|----------|---|---|---|---|--|---|---|---|---|---|----|--|--|--|--|--|--|--|--|
| list     |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
| list[0]  |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
| list[1]  | S | n | o | w |  | W | h | i | t | e | \0 |  |  |  |  |  |  |  |  |
| list[2]  |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
| list[3]  |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
|          |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
|          |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
|          |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
| list[40] |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
| list[41] |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
|          |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
|          |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
|          |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
| list[98] |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |
| list[99] |   |   |   |   |  |   |   |   |   |   |    |  |  |  |  |  |  |  |  |

图 10.41 数组 list, 显示 list[1]

### 10.2.7 声明二维数组的另一种方法

**注意：**该部分可以跳过，不会造成不连续。

如果知道程序使用表格的大小，那么可以使用 typedef 首先定义一个二维数组类型，然后声明该类型的变量。例如，考虑下面的语句：

```
const int numberOfRows=20;
const int numberOfColumns=10;

typedef int tableType[numberOfRows][numberOfColumns] ;
```

语句定义了一个二维数组数据类型 tableType，现在可以声明该类型的变量。所以：

```
tableType matrix;
```

声明了一个 20 行、10 列的二维数组 matrix。

当声明形参时，也可以使用该数据类型，如下面代码所示：

```
void initialize(tableType table)
{
 int row;
 int col;
 for(row = 0; row < numberOfRows; row++)
 for(col = 0; col < numberOfColumns; col++)
 table[row][col] = 0;
}
```

该函数将 tableType 类型的变量作为参数，即一个二维数组，并且将它初始化为 0。

通过定义一种数据类型，在声明二维数组为一个变量或形参时，或者函数调用期间将数组作为参数传递时，就不需要再检查列的准确数目，

## 10.3 多维数组

在第 9 章中，数组是固定数目的、具有相同类型元素（内容）的集合。一维数组其元素以顺序

表的形式进行编排；在二维数组中，元素以表格的形式进行排列。也可以定义三维数组或更大的数组。在C++中，对数组的维数没有限制。下面是数组的常用定义：

**数组：**用 $n$  ( $n \geq 1$ ) 维来排列的、具有相同类型元素的集合称为 $n$ 维数组。

声明 $n$ 维数组常用的语法是：

```
dataType arrayName[intExp1][intExp2] ...[intExpn] ;
```

其中 $\text{intExp1}, \text{intExp2}, \dots, \text{intExpn}$  是结果为正整数值的常量表达式。

访问 $n$ 维数组元素常用的语法是：

```
arrayName[indexExp1][indexExp2] ...[indexExpn]
```

其中 $\text{indexExp1}, \text{indexExp2}, \dots, \text{indexExpn}$  是结果为非负整数值的表达式。

例如语句：

```
double carDealers[10][5][7] ;
```

声明`carDealers`是一个三维数组。第一维的大小是10，第二维的大小是5，第三维的大小是7。第一维范围从0到9，第二维范围从0到4，第三维范围从0到6。数组`carDealers`的地址是第1个数组元素的地址，即`carDealers[0][0][0]`的地址。数组`carDealers`中的元素总数是 $10 \times 5 \times 7 = 350$ 。

语句：

```
carDealers[5][3][2] = 15564.75
```

将元素`carDealers[5][3][2]`的值设置为15564.75。

可以使用循环来处理多维数组。例如，嵌套for循环：

```
for(i = 0; i < 10; i++)
 for(j = 0; j < 5; j++)
 for(k = 0; k < 7; k++)
 carDealers[i][j][k] = 0.0;
```

将整个数组初始化为0.0。

当在函数中声明多维数组作为形参时，可以忽略第一维的大小，但是不能忽略其他维。作为参数，多维数组只能作为引用参数进行传递而且函数不能返回数组。同时，没有检查机制来确定数组下标是否在边界内。

## 10.4 编程实例：选举结果

当地大学将要举行学生会主席的选举。由于保密的原因，选举委员会主席决定用计算机进行记票。编写一个程序，以便分析数据并产生获胜者。

该大学有4个部，每个部有几个系。由于选举的原因，这4个部门分别记为`Region1`、`Region2`、`Region3`和`Region4`。部门中的每一个系管理其选举过程并且向选举委员会直接汇报每一位候选人得到的票数。选举结果由下面的形式来报告：

```
candidate_name region# number_of_votes_for_this_candidate
```

选举委员会想用下面的表格形式来输出结果：

```
-----Election Results-----
Candidate
Name Region1 Region2 Region3 Region4 Total
```

```

Balto 0 0 0 272 272
Doc 25 71 156 97 349
.
.
.
Winner: ???, Votes Received: ???
Total votes polled: ???

```

输出中候选人的名字必须按字母大小进行排序。

对于该程序, 假设6位候选人参加学生会主席的竞选。当然可以改进该程序并包含任何数目的候选人。

数据由两个文件提供。文件candData.txt包含了竞选学生会主席的候选人名单。该文件中候选人的名字没有以特定的顺序排列。在第二个文件 voteData.txt 中, 每一行包含了如下形式的选举结果:

```
candidateName regionNumber numberOfVotesForThisCandidate
```

voteData.txt 文件中的每一行包含了候选人的名字、region 号码和候选人在该区得到的选票数。例如, 包含选举数据的输入文件应如下所示:

```

Donald 1 23
Pluto 2 56
Doc 1 25
Pluto 4 23
.
.
.

```

第1行显示了 Donald 在第一区收到了 23 张选票。

**输入** 两个文件, 一个文件包含候选人的名字, 另一个文件包含前面描述的选举数据。

**输出** 选举结果和获胜者。

### 问题分析和算法设计

从输出中可以明显看出, 程序必须按区组织选举数据。该程序还必须计算每位候选人的总得票数和选举总票数。而且, 候选人的名字必须按字母顺序来排序。

因为候选人名字(字符串)的数据类型和票数(整数)的类型是不同的, 所以需要两个独立的数组, 分别用来保存候选人的名字和选举数据。保存候选人名字的数组是一维数组, 而且该数组中的每一个元素是一个字符串。这里, 不使用一个单独的二维数组来保存选举数据, 而是使用一个二维数组来保存输出(部门选举结果)和一个一维数组来保存每一位候选人得到的总票数。这三个数组是关联的数组(参看图 10.42)。

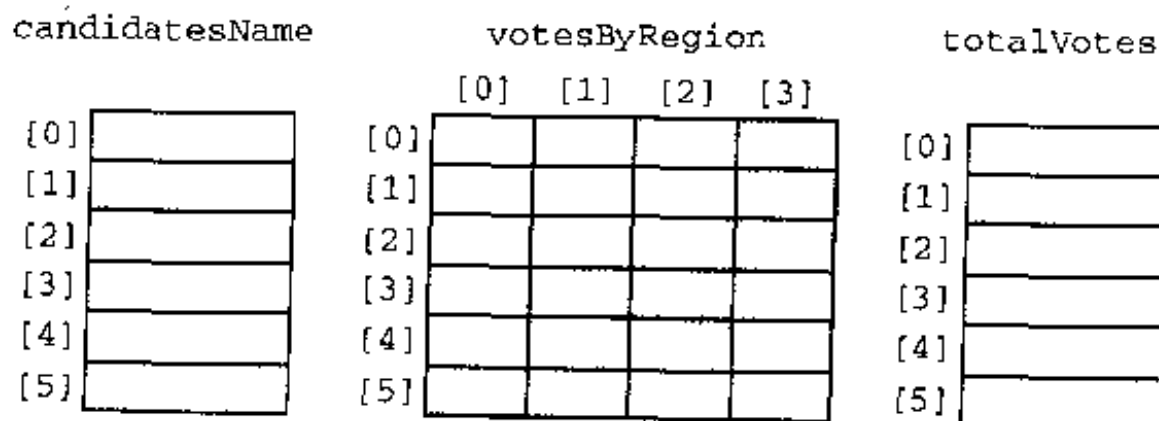


图 10.42 关联数组: candidatesNames、votesByRegion 和 totalVotes

这三个数组中的第1行数据对应于名字保存在 `candidatesName` 数组中第1行的候选人,等等。在 `votesByRegion` 数组中,第1列对应于第一部门,第2列对应于第二部门,等等。前面讲过,在C++中,数组下标从0开始。所以,如果程序中该数组的名字是 `votesByRegion`,那么 `votesByRegion[ ][0]` 指向第1列,也就是第一部门,等等。

为了简单起见,在该讨论的剩余部分假设:在编写的程序中,候选人名字数组为 `candidatesName`,部门选举结果数组为 `votesByRegion`,总选举结果数组为 `totalVotes`。

首先,需要从输入文件 `candData.txt` 中将候选人的名字读入到数组为 `candidatesName` 中。在候选人的名字存储到数组 `candidatesName` 中后,必须对该数组进行排序。

接着,必须处理投票数据。文件 `voteData.txt` 中每一项都包含一个 `candidateName`, `regionNumber` 和 `votesReceivedByTheCandidate`。为了处理每一项,要找到数组 `votesByRegion` 中相应的项,并且将 `votesReceivedByTheCandidate` 添加到该项中来进行更新。所以数组 `votesByRegion` 必须初始化为0。(在本部分的后面将会详细地描述处理投票数据。)

在处理投票数据后,下一步是计算每一位候选人总得票数。该任务可以通过将每一个部门的投票相加来完成。所以必须将数组 `totalVotes` 初始化为0。最后输出前面所显示的结果。

该讨论转化为下面的算法:

1. 将候选人的名字读入到数组 `candidatesName` 中。
2. 排序数组 `candidatesName`。
3. 初始化数组 `votesByRegion` 和 `totalVotes`。
4. 处理投票数据。
5. 计算每位候选人的总得票数。
6. 输出结果。

因为输入数据由两个独立的文件提供,所以在程序中必须打开这两个文件。可以在函数 `main()` 中打开这两个文件。

为了执行算法中的6个步骤,该程序包含下面描述的7个函数。

**函数 `getCandidatesName()`** 函数从输入文件 `candData.txt` 中读入数据并且填充到数组 `candidatesName` 中。输入文件在函数 `main()` 中打开。可以看到函数有三个参数:对应于输入文件的参数,对应于数组 `candidatesName` 的参数,传递数组 `candidatesName` 行数的参数。该函数如下所示:

```
void getCandidatesName(ifstream& inp, string cNames[], int noOfRows)
{
 int i;
 for(i = 0; i < noOfRows; i++)
 inp >> cNames[i]; //read candidate_name
}
```

在调用该函数后,保存数据的数组如图10.43所示。

**函数 `sortCandidatesName()`** 函数使用选择排序算法对数组 `candidatesName` 进行排序。该函数有两个参数:对应于数组 `candidatesName` 的参数和传递数组 `candidatesName` 行数的参数。该函数如下所示:

```
void sortCandidatesName(string cNames[], int noOfRows)
{
 int i, j;
 int min;
```

```

 //selection sort
 for(i = 0; i < noOfRows - 1; i++)
 {
 min = i;
 for(j = i + 1; j < noOfRows; j++)
 if(cNames[j] < cNames[min])
 min = j;
 cNames[i].swap(cNames[min]);
 }
 }
}

```

在调用该函数后，数组如图 10.44 所示。

| candidatesName |        | votesByRegion |     |     |     | totalVotes |  |
|----------------|--------|---------------|-----|-----|-----|------------|--|
|                |        | [0]           | [1] | [2] | [3] |            |  |
| [0]            | Goofy  | [0]           |     |     |     | [0]        |  |
| [1]            | Mickey | [1]           |     |     |     | [1]        |  |
| [2]            | Donald | [2]           |     |     |     | [2]        |  |
| [3]            | Pluto  | [3]           |     |     |     | [3]        |  |
| [4]            | Doc    | [4]           |     |     |     | [4]        |  |
| [5]            | Balto  | [5]           |     |     |     | [5]        |  |

图 10.43 在读入候选人名单后的数组 candidatesName、votesByRegion 和 totalVotes

| candidatesName |        | votesByRegion |     |     |     | totalVotes |  |
|----------------|--------|---------------|-----|-----|-----|------------|--|
|                |        | [0]           | [1] | [2] | [3] |            |  |
| [0]            | Balto  | [0]           |     |     |     | [0]        |  |
| [1]            | Doc    | [1]           |     |     |     | [1]        |  |
| [2]            | Donald | [2]           |     |     |     | [2]        |  |
| [3]            | Goofy  | [3]           |     |     |     | [3]        |  |
| [4]            | Mickey | [4]           |     |     |     | [4]        |  |
| [5]            | Pluto  | [5]           |     |     |     | [5]        |  |

图 10.44 在对名字进行排序后的数组 candidatesName、votesByRegion 和 totalVotes

**函数 initialize()** 函数 initialize() 将数组 votesByRegion 和 totalVotes 初始化为零。该函数必须有三个参数：对应于数组 votesByRegion 的参数，对应于数组 totalVotes 的参数和传递数组 votesByRegion 行数的参数。注意两个数组有相同数目的行。该函数的定义是：

```

void initialize(int vbRegion[][noOfRegions], int tVotes[],
 int noOfRows)
{
 int i, j;
 for(i = 0; i < noOfRows; i++)
 for(j = 0; j < noOfRegions; j++)
 vbRegion[i][j] = 0;
 for(i = 0; i < noOfRows; i++)
 tVotes[i] = 0;
}

```

在调用该函数后，数组 votesByRegion 和 totalVotes 如图 10.45 所示。

| candidatesName |        | votesByRegion   |   |   |   |   | totalVotes |   |
|----------------|--------|-----------------|---|---|---|---|------------|---|
|                |        | [0] [1] [2] [3] |   |   |   |   |            |   |
| [0]            | Balto  | [0]             | 0 | 0 | 0 | 0 | [0]        | 0 |
| [1]            | Doc    | [1]             | 0 | 0 | 0 | 0 | [1]        | 0 |
| [2]            | Donald | [2]             | 0 | 0 | 0 | 0 | [2]        | 0 |
| [3]            | Goofy  | [3]             | 0 | 0 | 0 | 0 | [3]        | 0 |
| [4]            | Mickey | [4]             | 0 | 0 | 0 | 0 | [4]        | 0 |
| [5]            | Pluto  | [5]             | 0 | 0 | 0 | 0 | [5]        | 0 |

图 10.45 初始化后的数组 candidatesName、votesByRegion 和 totalVotes

**处理投票数据** 处理投票数据是很简单的。文件 voteData.txt 中的每一项有下面的形式：

candidateName regionNumber numberOfVotesForThisCandidate

处理投票数据的通常算法如下所示。对于 voteData.txt 中的每一项，需要如下步骤：

- 获得 candidateName, regionNumber 和 numberOfVotesForTheCandidate。
- 找到对应于该候选人的数组 candidatesName 的行号。
- 从数组 votesByRegion 中找到对应于 regionNumber 的相应列。
- 通过添加 numberOfVotesForTheCandidate 来更新数组 votesByRegion 中相应的项。

步骤 b 需要搜索数组 candidatesName 来找到特定候选人所在的位置（即行号）。因为数组 candidatesName 是经过排序的，所以可以使用二分检索算法来找到对应于特定候选人的行号。因此，程序还包含函数 binSearch() 对数组 candidatesName 执行二分检索算法。

假设三个数组如图 10.46 所示。

| candidatesName |        | votesByRegion   |     |    |    |    | totalVotes |   |
|----------------|--------|-----------------|-----|----|----|----|------------|---|
|                |        | [0] [1] [2] [3] |     |    |    |    |            |   |
| [0]            | Balto  | [0]             | 0   | 0  | 50 | 0  | [0]        | 0 |
| [1]            | Doc    | [1]             | 10  | 0  | 56 | 0  | [1]        | 0 |
| [2]            | Donald | [2]             | 76  | 13 | 0  | 0  | [2]        | 0 |
| [3]            | Goofy  | [3]             | 0   | 45 | 0  | 0  | [3]        | 0 |
| [4]            | Mickey | [4]             | 80  | 0  | 78 | 0  | [4]        | 0 |
| [5]            | Pluto  | [5]             | 100 | 0  | 0  | 20 | [5]        | 0 |

图 10.46 数组 candidatesName、votesByRegion 和 totalVotes

假设从输入文件中读入的一项是：

Donald 2 35

必须在对应于该候选人的数组中找到所在行的位置。为了找到该行，需要搜索数组 candidatesName 找到对应于该名字的行。Donald 对应于数组 candidatesName 中的第 2 行。（参看图 10.47。）

为了处理该项，需要访问数组 votesByRegion 的第 2 行。因为 Donald 从部门 2 得到了 35 张选票，访问第 3 行、第 2 列（votesByRegion[2][1]）并且将 35 与前一个值相加来更新该项。下面的语句完成该任务：

votesByRegion[2][1]=votesByRegion[2][1]+35;

在处理完该项后，三个数组如图 10.48 所示。



| candidatesName |        | votesByRegion |     |     |     | totalVotes |     |   |
|----------------|--------|---------------|-----|-----|-----|------------|-----|---|
|                |        | [0]           | [1] | [2] | [3] |            |     |   |
| [0]            | Balto  | [0]           | 0   | 0   | 50  | 0          | [0] | 0 |
| [1]            | Doc    | [1]           | 10  | 0   | 56  | 0          | [1] | 0 |
| [2]            | Donald | [2]           | 76  | 13  | 0   | 0          | [2] | 0 |
| [3]            | Goofy  | [3]           | 0   | 45  | 0   | 0          | [3] | 0 |
| [4]            | Mickey | [4]           | 80  | 0   | 78  | 0          | [4] | 0 |
| [5]            | Pluto  | [5]           | 100 | 0   | 0   | 20         | [5] | 0 |

图 10.47 Donald 的位置和 region=2

Donald                      region = 2

candidatesName                      votesByRegion                      totalVotes

|     |        | [0] | [1] | [2] | [3] |    |     |   |
|-----|--------|-----|-----|-----|-----|----|-----|---|
| [0] | Balto  | [0] | 0   | 0   | 50  | 0  | [0] | 0 |
| [1] | Doc    | [1] | 10  | 0   | 56  | 0  | [1] | 0 |
| [2] | Donald | [2] | 76  | 48  | 0   | 0  | [2] | 0 |
| [3] | Goofy  | [3] | 0   | 45  | 0   | 0  | [3] | 0 |
| [4] | Mickey | [4] | 80  | 0   | 78  | 0  | [4] | 0 |
| [5] | Pluto  | [5] | 100 | 0   | 0   | 20 | [5] | 0 |

图 10.48 处理完 Donald 2 35 后的数组 candidatesName、votesByRegion 和 totalVotes

接下来描述函数 binSearch() 和处理得票数据的函数 processVotes()。

**函数 binSearch()** 函数对数组 candidatesName 执行二分检索算法。它类似于本章前面描述的函数 binarySearch()。它的定义是：

```
int binSearch(string cNames[], int noOfRows, string name)
{
 int first, last, mid;
 bool found;

 first = 0;
 last = noOfRows - 1;
 found = false;

 while(!found && first <= last)
 {
 mid = (first + last) / 2;
 if(cNames[mid] == name)
 found = true;
 else
 if(cNames[mid] > name)
 last = mid - 1;
 else
 first = mid + 1;
 }

 if(found)
 return mid;
 else
 return -1;
}
```

函数 `processVotes()` 该函数处理得票数据。很明显,它必须能访问数组 `candidatesName` 和 `votesByRegion` 以及输入文件 `voteData.txt`, 还需要告知该函数每一个数组中的行数。这样,函数 `processVotes()` 有4个参数:访问输入文件 `voteData.txt` 的参数,对应于数组 `candidatesName` 的参数,对应于数组 `votesByRegion` 的参数和传递每一个数组行数的参数。该函数的定义是:

```
void processVotes(ifstream& inp, string cNames[],
 int vbRegion[][noOfRegions], int noOfRows)
{
 string candName;
 int region;
 int noOfVotes;
 int loc;

 inp>>candName>>region>>noOfVotes; //Step a
 while(inp)
 {
 loc = binSearch(cNames,noOfRows,candName); //Step b
 if(loc != -1)
 vbRegion[loc][region - 1] = vbRegion[loc][region - 1]
 + noOfVotes; //Steps c and d
 inp>>candName>>region>>noOfVotes; //Step a
 }
}
```

计算总得票数 (函数 `addRegionsVote()`) 在处理完得票数据后,下一个步骤是用来计算每位候选人总得票数。假设在处理完得票数据后,数组如图 10.49 所示。

| candidatesName |        | votesByRegion |     |     |     | totalVotes |   |
|----------------|--------|---------------|-----|-----|-----|------------|---|
|                |        | [0]           | [1] | [2] | [3] |            |   |
| [0]            | Balto  | [0]           | 0   | 0   | 272 | [0]        | 0 |
| [1]            | Doc    | [1]           | 25  | 71  | 156 | [1]        | 0 |
| [2]            | Donald | [2]           | 110 | 158 | 0   | [2]        | 0 |
| [3]            | Goofy  | [3]           | 0   | 34  | 0   | [3]        | 0 |
| [4]            | Mickey | [4]           | 112 | 141 | 156 | [4]        | 0 |
| [5]            | Pluto  | [5]           | 285 | 56  | 0   | [5]        | 0 |

图 10.49 处理完得票数据后的数组 `candidatesName`、`votesByRegion` 和 `totalVotes`

在计算完每位候选人总得票数后,三个数组如图 10.50 所示。

| candidatesName |        | votesByRegion |     |     |     | totalVotes |     |
|----------------|--------|---------------|-----|-----|-----|------------|-----|
|                |        | [0]           | [1] | [2] | [3] |            |     |
| [0]            | Balto  | [0]           | 0   | 0   | 272 | [0]        | 272 |
| [1]            | Doc    | [1]           | 25  | 71  | 156 | [1]        | 349 |
| [2]            | Donald | [2]           | 110 | 158 | 0   | [2]        | 268 |
| [3]            | Goofy  | [3]           | 0   | 34  | 0   | [3]        | 34  |
| [4]            | Mickey | [4]           | 112 | 141 | 156 | [4]        | 498 |
| [5]            | Pluto  | [5]           | 285 | 56  | 0   | [5]        | 387 |

图 10.50 计算完每位候选人总得票数后的数组 `candidatesName`、`votesByRegion` 和 `totalVotes`

为了计算每位候选人总得票数,将数组 `votesByRegion` 中每一行的内容相加,并且将总和存储到数组 `totalVotes` 对应的行中。该任务通过函数 `addRegionsVote()` 来完成。

函数 `addRegionsVote()` 计算每位候选人总得票数。该函数必须访问数组 `votesByRegion` 和 `totalVotes`。



而且,还必须告知该函数每一个数组中的行数。函数有三个参数:对应于数组 `votesByRegion` 的参数,对应于数组 `totalVotes` 的参数和传递每一个数组行数的参数。该函数的定义是:

```
void addRegionsVote(int vbRegion[][noOfRegions], int tVotes[],
 int noOfRows)
{
 int i,j;
 for(i = 0; i < noOfRows; i++)
 for(j = 0; j < noOfRegions; j++)
 tVotes[i] = tVotes[i] + vbRegion[i][j];
}
```

下面将描述为了得到期望结果所需要的其他函数。

**函数 `printHeading()`** 函数输出输入数据的前 4 行,所以它包含特定的输出语句。该函数的定义是:

```
void printHeading()
{
 cout<<" -----Election Results-----"
 <<endl<<endl;
 cout<<"Candidate Votes"<<endl;
 cout<<"Name Region1 Region2 Region3 "
 <<"Region4 Total"<<endl;
 cout<<"-----"
 <<"-----"<<endl;
}
```

**函数 `printResults()`** 该函数显示输出内容的剩余行。很明显,它必须访问三个数组中的每一个数组,还必须告知该函数每一个数组中的行数。(注意每一个数组有相同的行数。)这样,该函数有 4 个参数。假设参数 `cNames` 对应于 `candidatesName`, 参数 `vbRegion` 对应 `votesByRegion` 以及对应于 `totalVotes` 的参数 `Votes`。

假设变量 `sumVotes` 保存选举中的总投票数,变量 `largestVotes` 保存候选人得到的最多票数,变量 `winLoc` 保存数组 `candidatesName` 中获胜候选人的下标。该函数的算法是:

i. 初始化 `sumVotes`、`largestVotes` 和 `winLoc` 为 0。

ii. 对于每一个数组中的每一行:

```
a. if(largestVotes < tVotes[i])
{
 largestVotes = tVotes[i];
 winLoc = i;
}
```

b. `sumVotes=sumVotes+tVotes[i];`

c. 输出每一数组中的相应行。

iii. 输出最终结果。

该函数的定义是:

```
void printResults(string cNames[], int vbRegion[][noOfRegions],
 int tVotes[], int noOfRows)
{
 int i, j;
 int largestVotes = 0; //Step i
 int winLoc = 0; //Step i
 int sumVotes = 0; //Step i
```

```

 for(i = 0; i < noOfRows; i++) //Step ii
 {
 if(largestVotes < tVotes[i]) //Step ii.a
 {
 largestVotes = tVotes[i];
 winLoc = i;
 }

 sumVotes = sumVotes + tVotes[i]; //Step ii.b

 //Step ii.c
 cout<<left; //output name left justified
 cout<<setw(9)<<cNames[i]<<" ";
 cout<<right;
 for(j = 0; j < noOfRegions; j++)
 cout<<setw(8)<<vbRegion[i][j]<<" ";
 cout<<setw(6)<<tVotes[i]<<endl;
 }

 //Step iii
 cout<<endl<<endl<<"Winner: "<<cNames[winLoc]
 <<" , Votes Received: "<<tVotes[winLoc]<<endl<<endl;
 cout<<"Total votes polled: "<<sumVotes<<endl;
 }
}

```

现在给出主要算法。

假设 main() 函数中的变量是:

```

string candidatesName[noOfCandidates]; //array to store
 //candidate's name
int votesByRegion[noOfCandidates][noOfRegions]; //array to hold
 //voting data by region
int totalVotes[noOfCandidates]; //array to hold total votes
 //received by each candidate
ifstream infile; //input file variable

```

候选人的名字在文件 candData.txt 中, 投票数据在文件 voteData.txt 中。

**主要算法: 函数 main()**

1. 声明变量。
2. 打开输入文件 candData.txt。
3. 如果输入文件不存在, 退出程序。
4. 从文件 candData.txt 中读入数据到数组 candidatesName 中。
5. 排序数组 candidatesName。
6. 关闭文件 candData.txt。
7. 打开输入文件 voteData.txt。
8. 如果输入文件不存在, 退出程序。
9. 初始化数组 votesByRegion 和 totalVotes。
10. 处理投票数据并将结果保存到数组 votesByRegion 中。
11. 计算每位候选人总得票数并将结果保存在数组 totalVotes 中。
12. 打印标题。
13. 打印结果。

**完整程序清单**

```
#include <iostream>
```

```

#include <fstream>
#include <string>
#include <iomanip>

using namespace std;

const int noOfCandidates = 6;
const int noOfRegions = 4;

void printHeading();
void initialize(int vbRegion[][noOfRegions], int tVotes[],
 int noOfRows);
void getCandidatesName(ifstream& inp, string cNames[],
 int noOfRows);
void sortCandidatesName(string cNames[], int noOfRows);
int binSearch(string cNames[], int noOfRows, string name);
void processVotes(ifstream& inp, string cNames[],
 int vbRegion[][noOfRegions], int noOfRows);
void addRegionsVote(int vbRegion[][noOfRegions], int tVotes[],
 int noOfRows);

void printResults(string cNames[], int vbRegion[][noOfRegions],
 int tVotes[], int noOfRows);

int main()
{
 //Declare variables; Step 1
 string candidatesName[noOfCandidates];
 int votesByRegion[noOfCandidates][noOfRegions];
 int totalVotes[noOfCandidates];
 ifstream infile;

 infile.open("a:candData.txt"); //Step 2
 if(!infile) //Step 3
 {
 cout<<"Input file (candData.txt) does not exist."<<endl;
 return 1;
 }

 getCandidatesName(infile, candidatesName,
 noOfCandidates); //Step 4
 sortCandidatesName(candidatesName, noOfCandidates); //Step 5
 infile.close(); //Step 6
 infile.open("a:voteData.txt"); //Step 7
 if(!infile) //Step 8
 {
 cout<<"Input file (voteData.txt) does not exist."<<endl;
 return 1;
 }

 initialize(votesByRegion, totalVotes, noOfCandidates); //Step 9
 processVotes(infile, candidatesName, votesByRegion,
 noOfCandidates); //Step 10
 addRegionsVote(votesByRegion, totalVotes,
 noOfCandidates); //Step 11
 printHeading(); //Step 12
 printResults(candidatesName, votesByRegion,
 totalVotes, noOfCandidates); //Step 13
 return 0;
}

```

```
void initialize(int vbRegion[][noOfRegions], int tVotes[],
 int noOfRows)
{
 int i, j;
 for(i = 0; i < noOfRows; i++)
 for(j = 0; j < noOfRegions; j++)
 vbRegion[i][j] = 0;
 for(i = 0; i < noOfRows; i++)
 tVotes[i] = 0;
}

void getCandidatesName(ifstream& inp, string cNames[], int noOfRows)
{
 int i;
 for(i = 0; i < noOfRows; i++)
 inp >> cNames[i];
}

void sortCandidatesName(string cNames[], int noOfRows)
{
 int i, j;
 int min;

 //selection sort
 for(i = 0; i < noOfRows - 1; i++)
 {
 min = i;
 for(j = i + 1; j < noOfRows; j++)
 if(cNames[j] < cNames[min])
 min = j;
 cNames[i].swap(cNames[min]);
 }
}

int binSearch(string cNames[], int noOfRows, string name)
{
 int first, last, mid;
 bool found;
 first = 0;
 last = noOfRows - 1;
 found = false;
 while(!found && first <= last)
 {
 mid = (first + last) / 2;
 if(cNames[mid] == name)
 found = true;
 else
 if(cNames[mid] > name)
 last = mid - 1;
 else
 first = mid + 1;
 }
 if(found)
 return mid;
 else
```

```

 return -1;
 }

 void processVotes(ifstream& inp, string cNames[],
 int vbRegion[][noOfRegions], int noOfRows)
 {
 string candName;
 int region;
 int noOfVotes;
 int loc;

 inp>>candName>>region>>noOfVotes;
 while(inp)
 {
 loc = binSearch(cNames, noOfRows, candName);
 if(loc != -1)
 vbRegion[loc][region - 1] = vbRegion[loc][region - 1]
 + noOfVotes;
 inp>>candName>>region>>noOfVotes;
 }
 }

 void addRegionsVote(int vbRegion[][noOfRegions], int tVotes[],
 int noOfRows)
 {
 int i, j;
 for(i = 0; i < noOfRows; i++)
 for(j = 0; j < noOfRegions; j++)
 tVotes[i] = tVotes[i] + vbRegion[i][j];
 }

 void printHeading()
 {
 cout<<" -----Election Results-----"
 <<endl<<endl;
 cout<<"Candidate Votes"<<endl;
 cout<<"Name Region1 Region2 Region3 "
 <<"Region4 Total"<<endl;
 cout<<"-----"
 <<"-----"
 <<endl;
 }

 void printResults(string cNames[], int vbRegion[][noOfRegions],
 int tVotes[], int noOfRows)
 {
 int i, j;
 int largestVotes = 0;
 int winLoc = 0;
 int sumVotes = 0;

 for(i = 0; i < noOfRows; i++)
 {
 if(largestVotes < tVotes[i])
 {
 largestVotes = tVotes[i];
 winLoc = i;
 }

 sumVotes = sumVotes + tVotes[i];
 }
 }

```

```

 cout<<left;
 cout<<setw(9)<<cNames[i]<<" ";
 cout<<right;
 for(j = 0; j < noOfRegions; j++)
 cout<<setw(8)<<vbRegion[i][j]<<" ";
 cout<<setw(6)<<tVotes[i]<<endl;
 }

 cout<<endl<<endl<<"Winner: "<<cNames[winLoc]
 <<" , Votes Received: "<<tVotes[winLoc]<<endl<<endl;
 cout<<"Total votes polled: "<<sumVotes<<endl;
}

```

运行实例:

-----Election Results-----

| Candidate<br>Name | Region1 | Region2 | Votes<br>Region3 | Region4 | Total |
|-------------------|---------|---------|------------------|---------|-------|
| Balto             | 0       | 0       | 0                | 272     | 272   |
| Doc               | 25      | 71      | 156              | 97      | 349   |
| Donald            | 110     | 158     | 0                | 0       | 268   |
| Goofy             | 0       | 34      | 0                | 0       | 34    |
| Mickey            | 112     | 141     | 156              | 89      | 498   |
| Pluto             | 285     | 56      | 0                | 46      | 387   |

Winner: Mickey, Votes Received: 498

Total votes polled: 1808

输入文件:

| candData.txt | voteData.txt |
|--------------|--------------|
| Goofy        | Goofy 2 34   |
| Mickey       | Mickey 1 56  |
| Donald       | Donald 2 56  |
| Pluto        | Pluto 1 78   |
| Doc          | Doc 4 29     |
| Balto        | Balto 4 78   |
|              | Mickey 2 63  |
|              | Donald 1 23  |
|              | Pluto 2 56   |
|              | Doc 1 25     |
|              | Pluto 4 23   |
|              | Doc 4 12     |
|              | Balto 4 82   |
|              | Mickey 3 67  |
|              | Donald 2 67  |
|              | Doc 3 67     |
|              | Balto 4 23   |
|              | Mickey 1 56  |
|              | Donald 2 35  |
|              | Pluto 1 27   |
|              | Doc 2 34     |
|              | Pluto 4 23   |
|              | Mickey 4 89  |
|              | Pluto 1 23   |
|              | Doc 3 89     |
|              | Mickey 3 89  |



```
Pluto 1 67
Doc 2 37
Balto 4 89
Mickey 2 78
Donald 1 87
Pluto 1 90
Doc 4 56
```

## 10.5 小结

1. 顺序表是相同类型元素的集合。
2. 顺序表的长度是顺序表中元素的数量。
3. 一维数组是保存和处理顺序表的一种简便方式。
4. 顺序查找从顺序表的第一个元素来搜索给定项，并将搜索项与顺序表中的其他元素进行比较，直到找到搜索项或顺序表中没有与搜索项相匹配的数据。
5. 平均来讲，顺序查找通常要查找顺序表的一半。
6. 对很短的顺序表来说，顺序查找是一种很好的方法。
7. 二分查找比顺序查找要快得多。
8. 二分查找需要顺序表元素是按顺序排列的，也就是说排序的。
9. 对于一个长度为 1024 的顺序表，为了确定一项是否在顺序表中，二分查找算法不会超过 22 次比较。
10. 选择排序通过找到顺序表中最小（最大）元素并将它移动到顺序表的开头（结尾）来排序顺序表。
11. 在二维数组中，元素以表格的形式进行排列。
12. 为了访问二维数组中的元素，需要一对下标：一个用于行位置，一个用于列位置。
13. 二维数组中，行从 0 到 `rowsize-1` 编号，列从 0 到 `columnsize-1` 编号。
14. 如果 `matrix` 是二维数组，那么 `matrix` 的地址是数组元素 `matrix[0][0]` 的地址。
15. 在行处理中，二维数组一次处理一行。
16. 在列处理中，二维数组一次处理一列。
17. 当声明二维数组作为一个形参时，可以忽略第一维的大小，但是不能忽略第二维的大小。
18. 当二维数组作为实际参数传递时，实参和形参的列数必须匹配。
19. 在计算机内存中，C++ 以行优先的形式存储二维数组。

## 10.6 练习

1. 判断下面语句的对错。
  - a. 顺序查找顺序表需要顺序表是升序的。
  - b. 二分查找顺序表需要顺序表是排序的。
  - c. 二分查找对于排序顺序表较快，而对于未排序顺序表较慢。
  - d. 二分查找对于大的顺序表较快，而顺序查找对于小的顺序表较快。
  - e. 作为参数，二维数组既可以作为值也可以作为引用参数传递。
2. 考虑下面的顺序表：

63 45 32 98 46 57 28 100

使用顺序查找，为了确定下面的项是否在顺序表中，需要进行多少次比较？（前面讲过“比

较”表示关键字比较,而不是下标比较。)

- a. 90
- b. 57
- c. 63
- d. 120

3. 考虑下面的顺序表:

5 12 17 35 46 65 78 85 93 110 115

对排序的序列使用顺序查找,为了确定下面的项是否在顺序表中,需要进行多少次比较?

- a. 35
- b. 60
- c. 78
- d. 120

4. 考虑下面的顺序表:

2 10 17 45 49 55 68 85 92 98 110

使用二分检索,为了确定下面的项是否在顺序表中,需要进行多少次比较?在每次循环后,显示 first、last 和 middle 的值以及比较次数。

- a. 15
- b. 49
- c. 98
- d. 99

5. 考虑下面的声明:

```
const int carTypes=5;
const int colorTypes=6;
```

```
double sales[carTypes][colorTypes];
```

- a. 数组 sales 有多少个元素?
  - b. 数组 sales 的行数是多少?
  - c. 数组 sales 的列数是多少?
  - d. 为了按照 carTypes 来求 sales 的和,需要什么类型的处理?
  - e. 为了按照 colorTypes 来求 sales 的和,需要什么类型的处理?
6. 编写 C++ 语句完成下面的操作:
- a. 声明 10 行、20 列的 int 类型数组 alpha。
  - b. 初始化数组 alpha 为 0。
  - c. 将 1 保存在第 1 行, 2 保存到其他行。
  - d. 将 5 保存到第 1 列, 并且保证每一个其他列的值是前一列值的两倍。
  - e. 每行打印 alpha 的一行。
  - f. 每行打印 alpha 的一列。

7. 考虑下面的声明:

```
int beta[3][3];
```

在下面的每一条语句执行后, beta 中存储的数据是什么?

- a. 

```
for(i = 0; i < 3; i++)
 for(j = 0; j < 3; j++)
 beta[i][j] = 0;
```
- b. 

```
for(i = 0; i < 3; i++)
```



```

 for(j = 0; j < 3; j++)
 beta[i][j] = i + j;
c. for(i = 0; i < 3; i++)
 for(j = 0; j < 3; j++)
 beta[i][j] = i * j;
d. for(i = 0; i < 3; i++)
 for(j = 0; j < 3; j++)
 beta[i][j] = 2 * (i + j) % 4;

```

## 10.7 编程练习

1. 编写一个函数 `remove()`，它有 3 个参数：整数类型的数组、数组的长度和一个整数（比如说 `removeItem`）。该函数应该找到并删除数组中第一次出现的 `removeItem`。如果该值不存在或数组为空，输出适当的信息。（注意在删除元素后，数组大小应该减 1。）假设数组未排序。
2. 编写一个函数 `removeAt()`，它有 3 个参数：整数类型的数组、数组的长度和一个整数（比如说 `index`）。函数应该能删除 `index` 所指示的数组元素。如果 `index` 超出范围或数组为空，则输出适当的信息。（注意在删除元素后，数组大小应该减 1。）假设数组未排序。
3. 编写一个函数 `removeAll()`，它有 3 个参数：整数类型的数组、数组的长度和一个整数（比如说 `removeItem`）。该函数应该找到并删除数组中所有出现的 `removeItem`。如果该值不存在或数组为空，输出适当的信息。（注意在删除元素后，数组大小应该减少相应值。）假设数组未排序。
4. 对一个排序的数组重做练习 1, 2, 3。
5. 编写一个函数 `insertAt()`，它有 4 个参数：整数类型的数组、数组的长度、一个整数（比如说 `insertItem`）和另一个整数（比如说 `index`）。该函数应该向数组中由 `index` 指定的位置中插入 `insertItem`。如果数组为空，输出适当的信息。（注意 `index` 必须在 0 和 `maxSize` 之间，也就是  $0 \leq \text{index} < \text{maxSize}$ 。）假设数组未排序。

6. 考虑下面的函数 `main()`：

```

int main()
{
 int inStock[10][4];
 int alpha[20];
 int beta[20];
 int gamma[4] = {11, 13, 15, 17};
 int delta[10] = {3, 5, 2, 6, 10, 9, 7, 11, 1, 8};
 .
 .
 .
}

```

- a. 编写将一维数组初始化为 0 的函数 `setZero()`。
- b. 编写函数 `inputArray()` 的定义，提示用户输入 20 个数字并将这些数字保存到 `alpha` 中。
- c. 编写函数 `doubleArray()` 的定义，将 `beta` 的元素初始化为 `alpha` 对应元素的两倍。要保证函数不能修改 `alpha` 的值。
- d. 编写函数 `copyGamma()` 的定义，将 `inStock` 的前 5 行设置为 `gamma`，`inStock` 其他的行设置为 `inStock` 前一行的 3 倍。要保证函数不能修改 `gamma` 的值。
- e. 编写函数 `copyAlphaBeta()` 的定义，将 `alpha` 复制到 `inStock` 的前 5 行，将 `beta` 复制到 `inStock` 的后 5 行。要保证函数不能修改 `alpha` 和 `beta` 的值。
- f. 编写函数 `printArray()` 的定义并打印一个 `int` 类型的一维数组。每行打印 15 个元素。

- g. 编写函数 `setInStock()` 的定义, 提示用户为 `inStock` 的第1列输入元素。该函数应该将其他列的元素设置为前一列对应元素的两倍减去 `delta` 中对应元素所得到的值。
- h. 编写 C++ 语句调用 a 到 g 部分的每一个函数。
- i. 编写 C++ 语句来测试 `main()` 函数和 a 到 g 讨论的函数。
7. 编写一个程序, 使用一个二维数组来存储一年中每月的最高和最低温度。程序应该输出该年的平均最高温度、平均最低温度以及该年的最高温度和最低温度。程序必须包含下面的函数:
- a. 函数 `getData()`: 该函数将数据读入并存储到二维数组中。
  - b. 函数 `averageHigh()`: 该函数计算并返回该年的平均高温。
  - c. 函数 `averageLow()`: 该函数计算并返回该年的平均低温。
  - d. 函数 `indexHighTemp()`: 该函数返回数组中最高温度的下标。
  - e. 函数 `indexLowTemp()`: 该函数返回数组中最低温度的下标。
8. 编写一个程序读入一个正整数序列, 并且输出一个特定数字在序列中出现的次数。假设数据最多有 100 个数字, -999 标志输入数据结束。数字必须以升序输出。例如, 对于数据: 15 40 28 62 95 15 28 13 62 65 48 95 65 62 65 95 95  
输出是:

| Number | Count |
|--------|-------|
| 13     | 1     |
| 15     | 2     |
| 28     | 2     |
| 40     | 1     |
| 48     | 1     |
| 62     | 3     |
| 65     | 3     |
| 95     | 4     |

9. (飞机座位分配问题) 编写一个程序用来分配飞机的座位。该飞机有 13 排, 每排有 6 个座位。第 1 排和第 2 排是头等舱, 其他排座位是经济舱。而且, 第 1 排到第 7 排为非吸烟区。程序必须提示用户输入下面的信息:
- a. 机票类型 (头等舱或经济舱)
  - b. 对于经济舱, 是否为吸烟区
  - c. 需要预订的座位

以下面的形式输出座位计划:

|        | A | B | C | D | E | F |
|--------|---|---|---|---|---|---|
| Row 1  | * | * | X | * | X | X |
| Row 2  | * | X | * | X | * | X |
| Row 3  | * | * | X | X | * | X |
| Row 4  | X | * | X | * | X | X |
| Row 5  | * | X | * | X | * | * |
| Row 6  | * | X | * | * | * | X |
| Row 7  | X | * | * | * | X | X |
| Row 8  | * | X | * | X | X | * |
| Row 9  | X | * | X | X | * | X |
| Row 10 | * | X | * | X | X | X |
| Row 11 | * | * | X | * | X | * |
| Row 12 | * | * | X | X | * | X |
| Row 13 | * | * | * | * | X | * |

其中 \* 表示座位可以出售, X 表示座位已售出。将该程序编写为一个菜单驱动的程序, 允许用户进行适当的选择并显示用户的选择。

## 10. 魔方问题:

- a. 编写函数 `createArithmeticSeq()`, 提示用户输入两个数字 `first` 和 `diff`。然后函数创建一个按算术顺序排列的、有 16 个元素的数组并输出该算术序列。例如, 如果 `first=21, diff=5`, 那么算术序列是 21 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96。
- b. 编写一个函数 `matricize()`, 它将一个有 16 个元素的一维数组和一个 4 行 4 列的二维数组作为参数。(其他的值, 例如数组的大小, 也必须作为参数来传递。)该函数将一维数组的元素放入到二维数组中。例如, 如果 `A` 是在 `a` 中创建的一维数组, `B` 是二维数组, 那么将 `A` 的元素放入到 `B` 中后, 数组 `B` 是:  
21 26 31 36  
41 46 51 56  
61 66 71 76  
81 86 91 96
- c. 编写一个函数 `reverseDiagonal()`, 倒转二维数组的两个对角线。例如, 如果二维数组如 `b` 所示, 那么在倒转两个对角线后, 二维数组是:  
96 26 31 81  
41 71 66 56  
61 51 46 76  
36 86 91 21
- d. 编写一个函数 `magicCheck()`, 它将一个有 16 个元素的一维数组、一个 4 行 4 列的二维数组以及数组的大小作为参数。通过将一维数组所有的元素相加后被 4 除, 该函数得到了 `magicNumber`。该函数然后将每一行、每一列以及两个对角线的元素相加, 然后将每一次和与 `magicNumber` 相比较。如果每一行、每一列以及两个对角线的元素和等于 `magicNumber`, 那么函数输出 “It is a magic square”, 否则输出 “It is not a magic number”。不必打印每一行、每一列以及两个对角线的元素和。
- e. 编写一个函数 `printMatrix()`, 输出二维数组的元素, 每行显示数组的一行。输出应该尽可能接近正方形的形式。
- f. `a` 到 `e` 编写的函数对于应用一个任何大小的数组来说通常是足够用的。(也就是说, 该工作应该使用循环来完成, 而不是逐次地将相应元素相加。)如果它调用两个单独地函数: 一个用来将行和列相加, 一个用来将对角线相加, 那么函数 `magicCheck()` 会工作的更好。(在这种情况下, 程序将 `magicNumber` 与每一个函数的和 (行、列和对角线) 进行比较。)
- g. 使用下面的 `main()` 函数来测试从 `a` 到 `e` 编写的程序。

```
const int rows = 4;
const int columns = 4;
const int listSize = 16;
...
int main()
{
 int list[listSize];
 int matrix[rows][columns];

 createArithmeticSeq(list, listSize);
 matricize(list, matrix, rows);
 printMatrix(matrix, rows);
 reverseDiagonal(matrix, rows);
 printMatrix(matrix, rows);
 magicCheck(list, matrix, listSize, rows);

 return 0;
}
```

# 第 11 章 递 归

本章要点:

- 递归的定义
- 递归定义的基本形式和一般形式
- 递归算法
- 递归函数
- 利用递归函数实现递归算法

在前面章节中,设计问题解决方案时使用的最为常见的技术是迭代。然而,对某些问题使用迭代技术是十分复杂的。本章将介绍另一种解决问题的技术——递归,并且提供几个例子来说明递归是如何工作的。

## 11.1 递归的定义

把问题简化成其最简形式从而加以解决的过程称为递归。在解决某些特定类型的问题时,递归算法十分是强大的,如果采用其他方法将会极其复杂。下面,考虑一个大部分人都很熟悉的问题:在数学中,整数的阶乘定义如下:

$$0! = 1 \quad (11.1)$$

$$n! = n \times (n - 1)! \quad \text{if } n > 0 \quad (11.2)$$

其中,  $0!$  定义为 1, 如果  $n$  是一个大于 0 的整数, 首先找到  $(n - 1)!$  的值, 然后用它乘以  $n$ 。要计算  $(n - 1)!$ , 将再次使用定义。如果  $(n - 1) > 0$ , 则使用等式 11.2; 否则, 使用等式 11.1。因此, 对于大于 0 的整数  $n$ ,  $n!$  的值是通过先计算  $(n - 1)!$  (即  $n!$  被简化计算), 然后再乘以  $n$  得到的。

应用这一定义计算  $3!$ 。这里  $n = 3$ , 因为  $n > 0$ , 所以使用等式 11.2:

$$3! = 3 \times 2!$$

接下来计算  $2!$ 。这里  $n = 2$ , 因为  $n > 0$ , 所以使用等式 11.2:

$$2! = 2 \times 1!$$

现在计算  $1!$ 。还是使用等式 11.2, 因为  $n = 1 > 0$ 。因此:

$$1! = 1 \times 0!$$

最后, 使用等式 11.1 计算到  $0!$ , 它的值是 1。将  $0!$  代入  $1!$  得到  $1! = 1$ ,  $2! = 2 \times 1! = 2 \times 1 = 2$ , 从而  $3! = 3 \times 2! = 3 \times 2 = 6$ 。

等式 11.1 中的形式是直接的——等式的右侧不包含阶乘符号。而等式 11.2 中的形式是以简化后的形式给出的。等式 11.1 和等式 11.2 所给出的阶乘定义叫做递归定义。等式 11.1 称为基本形式 (可以直接求值的形式); 等式 11.2 称为一般形式。

递归定义: 将问题定义为其自身的简化形式。

从前面阶乘的定义中, 可以清楚地看到:

1. 每一个递归定义必须含有一种(或多种)基本形式。
2. 一般形式必须能够最终转化到基本形式。
3. 由基本形式来终止递归。

递归的概念在计算机科学中应用在很相似的一类领域中。这里先来讨论递归算法和递归函数。通过将问题简化到其自身的简单形式来求解的算法称为递归算法。递归算法必须含有一种或多种基本形式, 并且一般形式必须最终转化为基本形式。

调用其本身的函数称为递归函数。即, 递归函数内部在计算本次调用结果之前包含一条导致调用该函数本身的语句。递归算法是由递归函数来实现的。

接下来, 我们编写一个递归函数来实现递归算法。

```
int fact(int num)
{
 if(num == 0)
 return 1;
 else
 return num * fact(num - 1);
}
```

图 11.1 跟踪了以下语句的执行情况:

```
cout<<fact(4)<<endl;
```

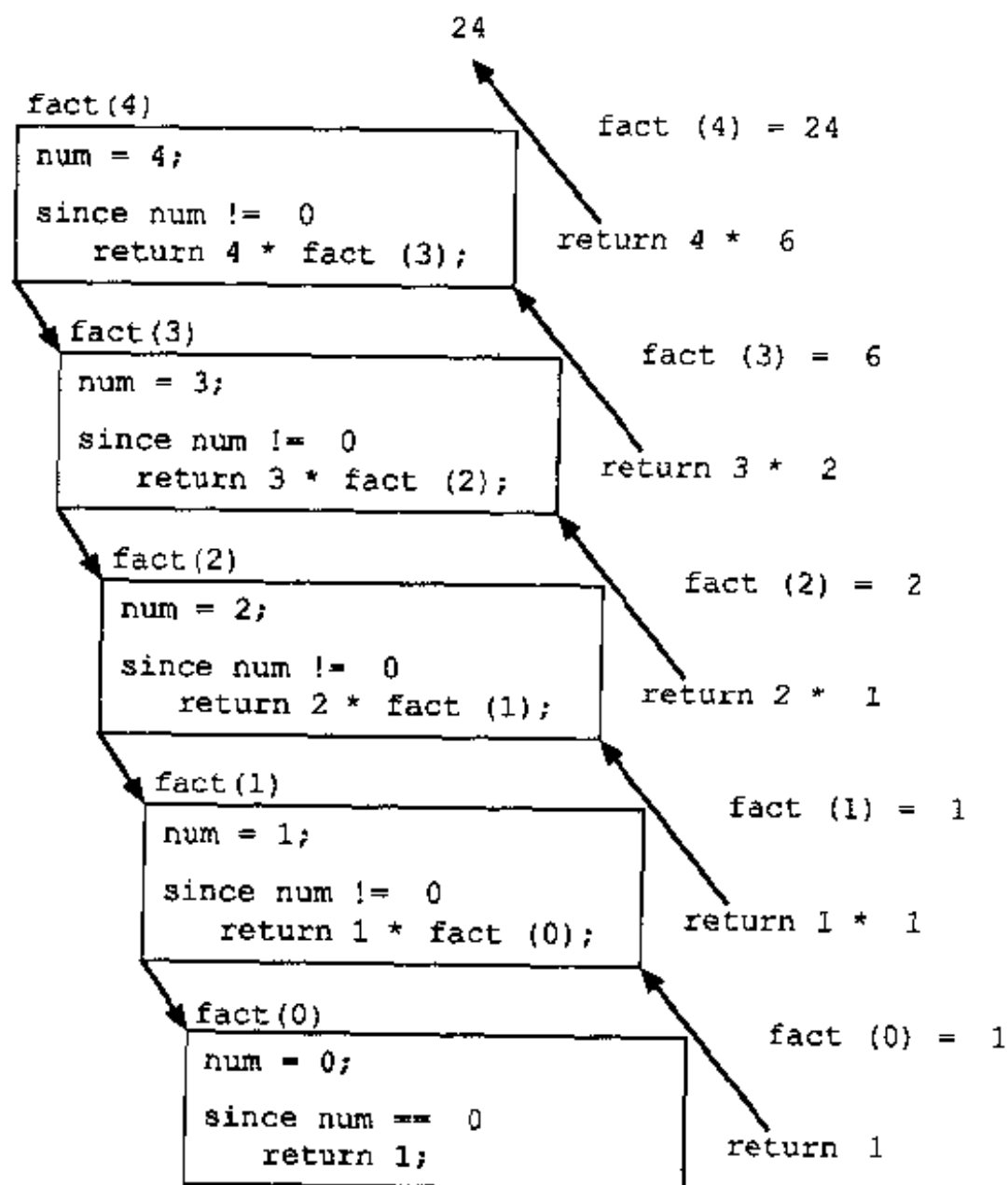


图 11.1 `fact(4)` 的执行情况

cout 语句运行的输出结果为:

24

在图 11.1 中, 下箭头表示对函数 fact() 的成功调用, 上箭头表示传递给调用者 (调用函数) 的返回值。

- 逻辑上, 可以将递归函数看做无限多个自身的副本
- 每次对递归函数的调用 (即每一次递归调用) 都有其自有的代码、参数集和局部变量。
- 在完成了某次递归调用后, 程序控制回到发出调用的环境——前一次调用。当控制权回到前一次调用时, 当前的 (递归) 调用必须执行完毕。前一次调用的逻辑在递归调用之后立即继续执行。

函数体内最后一条语句称为尾部递归函数。函数 fact() 就是尾部递归函数的一个实例。

## 11.2 利用递归解决问题

例 11.1 到 11.3 说明了递归算法是如何在 C++ 中利用递归函数实现的。

### 例 11.1 数组最大元素

在第 9 章中, 使用了循环来寻找一个数组中最大的元素。在本例中, 使用递归算法来实现。考虑图 11.2 中给出的数组:

|      |       |       |       |       |       |       |       |
|------|-------|-------|-------|-------|-------|-------|-------|
|      | [ 0 ] | [ 1 ] | [ 2 ] | [ 3 ] | [ 4 ] | [ 5 ] | [ 6 ] |
| list | 5     | 8     | 2     | 10    | 9     | 4     |       |

图 11.2 有 6 个元素的 list 数组

图 11.2 中所给数组中的最大元素是 10。

假设 list 是该数组的名字。list[a]...list[b] 代表从 list[a]、list[a+1], ..., list[b] 的数组元素。例如, list[0]...list[5] 代表数组元素 list[0]、list[1]、list[2]、list[3]、list[4] 和 list[5]。同样, list[1]...list[5] 代表数组元素 list[1]、list[2]、list[3]、list[4] 和 list[5]。为便于写出查找数组 list 中最大元素的递归算法, 需要用递归的方式来思考问题。

如果 list 的长度是 1, 那么 list 只有一个元素, 它同样也是最大元素。假设 list 的长度大于 1, 要找出 list[a]...list[b] 中的最大元素, 需要首先找到 list[a+1]...list[b] 中的最大元素, 然后将其与 list[a] 进行比较。即, list[a]...list[b] 中最大值可以用下面的表达式给出:

maximum(list[a], largest(list[a+1]...list[b]))

可以利用这一表达式找出图 11.2 中所示数组的最大值。该数组有 6 个元素, 用 list[0]...list[5] 表示。现在 list 中最大元素为:

maximum(list[0], largest(list[1]...list[5]))

即 list 中最大元素是 list[0] 和 list[1]...list[5] 中最大元素的最大者。要找到 list[1]...list[5] 中的最大元素, 再次使用同一表达式 (因为数组的长度大于 1)。则 list[1]...list[5] 中的最大元素为:

maximum(list[1], largest(list[2]...list[5]))

依次类推。可以发现每次使用前面的表达式在子数组中查找最大元素时, 下一次调用的子数组的长度递减 1。最终, 子数组的长度变为 1, 它只包含一个元素, 该元素同时也是该数组的最大元



素。从这一点开始，可以在递归调用中逐一返回。将本次讨论转换为伪代码所表示的递归算法：

```

if the size of the list is 1
 the only element in the list is the largest element
else
 to find the largest element in list[a]...list[b]
 a. find the largest element in list[a+1]...list[b] and call it
 max
 b. compare the elements list[a] and max
 if(list[a] >= max)
 the largest element in list[a]...list[b] is list[a]
 otherwise
 the largest element in list[a]...list[b] is max

```

这一算法可以转换为如下的 C++ 函数：

```

int largest(const int list[], int lowerIndex, int upperIndex)
{
 int max;
 if(lowerIndex == upperIndex) //size of the sublist is 1
 return list[lowerIndex];
 else
 {
 max = largest(list, lowerIndex + 1, upperIndex);
 if(list[lowerIndex] >= max)
 return list[lowerIndex];
 else
 return max;
 }
}

```

考虑图 11.3 给出的 list。

|      |       |       |       |       |       |       |       |
|------|-------|-------|-------|-------|-------|-------|-------|
|      | [ 0 ] | [ 1 ] | [ 2 ] | [ 3 ] | [ 4 ] | [ 5 ] | [ 6 ] |
| list | 5     | 10    | 12    | 8     |       |       |       |

图 11.3 有 4 个元素的 list 数组

让我们跟踪以下语句的执行情况：

```
cout<<largest(list, 0, 3);
```

这里 list 有 4 个元素，upperIndex = 3。图 11.4 跟踪了 largest(list, 0, 3) 的执行情况。表达式 largest(list, 0, 3) 的返回值是 12，即 list 中的最大元素。

下面的 C++ 程序使用函数 largest() 来求出 list 中的最大元素：

```

//Largest Element in an Array
#include <iostream>
using namespace std;
int largest(const int list[], int lowerIndex, int upperIndex);
int main()
{
 int intArray[10] = {23, 43, 35, 38, 67, 12, 76, 10, 34, 8};
 cout<<"The largest element in intArray: "
 <<largest(intArray, 0, 9);
}

```

```

 cout<<endl;
 return 0;
 }

 int largest(const int list[], int lowerIndex, int upperIndex)
 {
 int max;
 if(lowerIndex == upperIndex) //size of the sublist is 1
 return list[lowerIndex];
 else
 {
 max = largest(list, lowerIndex + 1, upperIndex);
 if(list[lowerIndex] >= max)
 return list[lowerIndex];
 else
 return max;
 }
 }
}

```

## 运行实例:

The largest element in intArray: 76

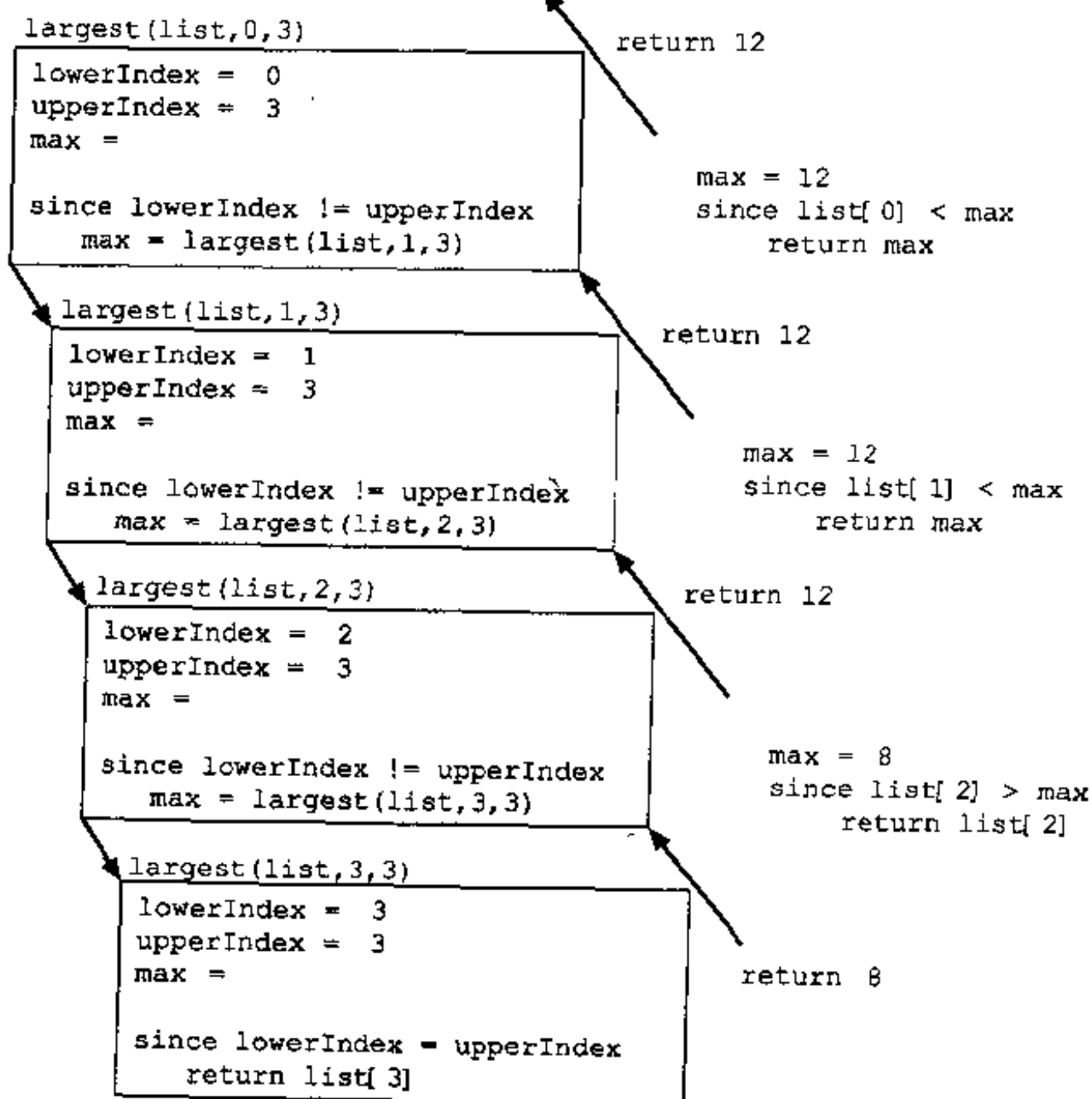


图 11.4 largest (list, 0, 3) 的执行过程



### 例 11.2 斐波那契 (Fibonacci) 数

在第 5 章中, 我们设计了一个程序来计算斐波那契数。在本例中, 编写一个递归函数 `rFibNum()` 来完成同样的工作。函数 `rFibNum()` 有三个参数: 斐波那契数列的前两个数字以及数字  $n$ , 返回第  $n$  个斐波那契数的值。函数 `rFibNum()` 返回斐波那契数列的第  $n$  个斐波那契数。

回忆一下, 第 3 个斐波那契数是前两个斐波那契数的和。数列的第 4 个斐波那契数是第 2 和第 3 个数的和。因此, 要计算第 4 个斐波那契数, 需要将第 2 个数与第 3 个数 (它是第 1 个数与第 2 个数的和) 相加。下面的递归算法计算第  $n$  个斐波那契数, 这里  $a$  表示第 1 个斐波那契数,  $b$  为第 2 个斐波那契数,  $n$  代表第  $n$  个斐波那契数:

$$\text{rFibNum}(a, b, n) = \begin{cases} a & \text{if } n=1 \\ b & \text{if } n=2 \\ \text{rFibNum}(a, b, n-1) + \text{rFibNum}(a, b, n-2) & \text{if } n>2. \end{cases} \quad (11.3)$$

假设需要计算:

`recFibNumber(2, 5, 4)`

这里  $a=2, b=5, n=4$ 。即求解斐波那契数列的第 4 个数, 序列中的第 1 个数是 2, 第 2 个数是 5。因为  $4>2$ ,

$$1. \text{rFibNum}(2, 5, 4) = \text{rFibNum}(2, 5, 3) + \text{rFibNum}(2, 5, 2)$$

接下来, 求解 `rFibNum(2, 5, 3)` 和 `rFibNum(2, 5, 2)`。首先求 `rFibNum(2, 5, 3)`。这里,  $a=2, b=5, n=2$ 。

$$1.a \text{ rFibNum}(2, 5, 3) = \text{rFibNum}(2, 5, 2) + \text{rFibNum}(2, 5, 1)$$

因此从公式 11.3 给出的定义可得:

$$1.a.1 \text{ rFibNum}(2, 5, 2) = 5$$

要确定 `rFibNum(2, 5, 1)` 的值, 注意, 这时  $a=2, b=5, n=1$ 。因此根据公式 11.3 的定义:

$$1.a.2 \text{ rFibNum}(2, 5, 1) = 2$$

将 `rFibNum(2, 5, 2)` 和 `rFibNum(2, 5, 1)` 的值代入 (1.a) 得到

$$\text{rFibNum}(2, 5, 3) = 5 + 2 = 7$$

现在, 求解 `rFibNum(2, 5, 2)`。正如 (1.a.1) 所示, `rFibNum(2, 5, 2) = 5`。可以将 `rFibNum(2, 5, 3)` 和 `rFibNum(2, 5, 2)` 的值代入 (1) 中得到:

$$\text{rFibNum}(2, 5, 4) = 7 + 5 = 12$$

下面的递归函数实现了该算法:

```
int rFibNum(int a, int b, int n)
{
 if(n == 1)
 return a;
 else if(n == 2)
 return b;
 else
 return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

现在，跟踪一下下面语句的执行：

```
cout<<rFibNum(2, 3, 5)<<endl;
```

在该表达式中，第1个数为2，第2个数为3，需要求解斐波那契数列的第5个数。图11.5跟踪了表达式 `rFibNum(2, 3, 5)` 的执行过程。返回值是13，即以2作为第1个数，以3作为第2个数的斐波那契数列中第5个数的值。

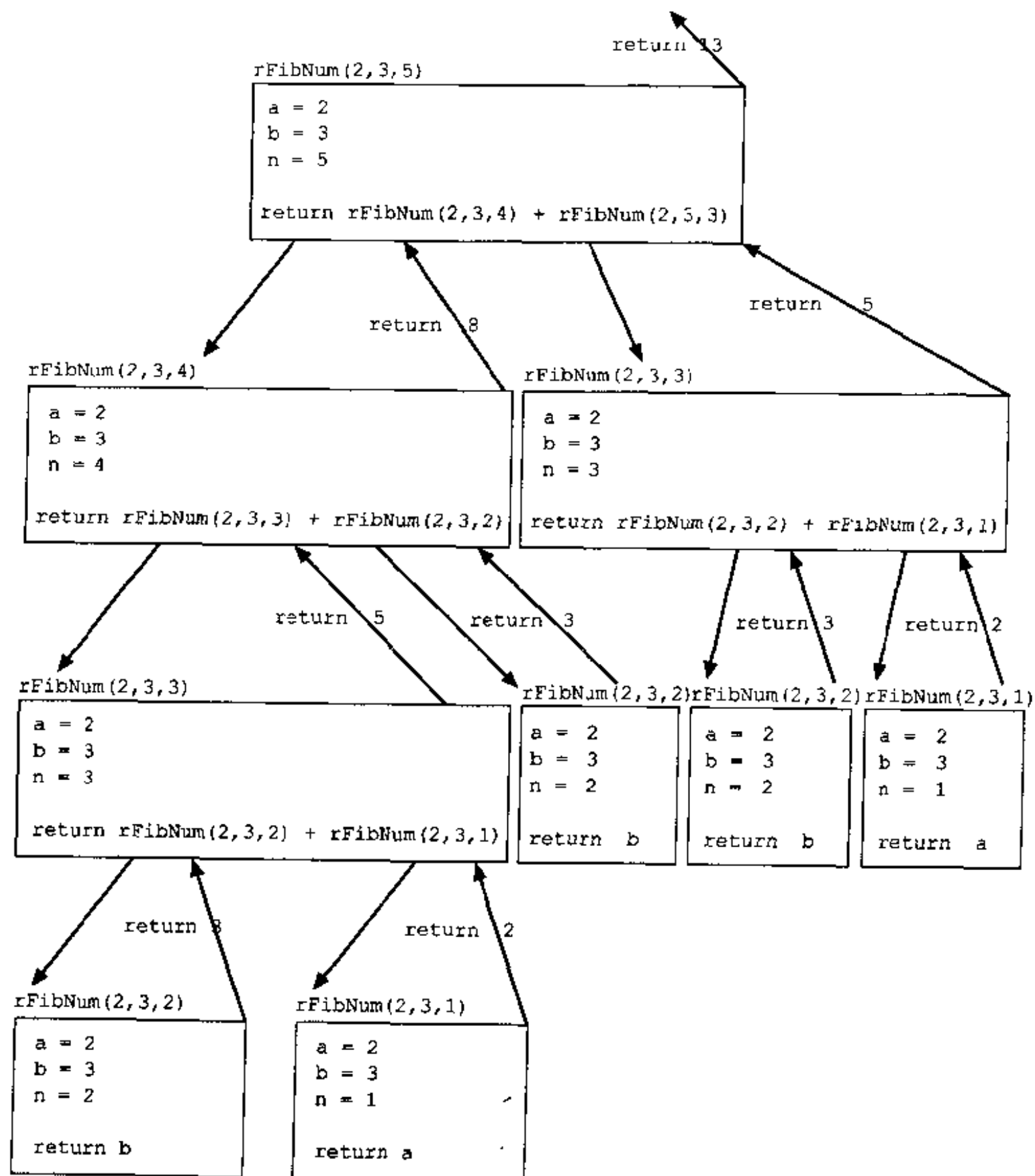


图11.5 `recFibNumber(2, 3, 5)` 的执行情况

下面的 C++ 程序使用了函数 `rFibNum()`：

```
//Chapter 11: Fibonacci Number
#include <iostream>
using namespace std;
```

```

int rFibNum(int a, int b, int n);
int main()
{
 int firstFibNum;
 int secondFibNum;
 int nth;

 cout<<"Enter first Fibonacci number: ";
 cin>>firstFibNum;
 cout<<endl;

 cout<<"Enter second Fibonacci number: ";
 cin>>secondFibNum;
 cout<<endl;

 cout<<"Enter desired Fibonacci number: ";
 cin>>nth;
 cout<<endl;

 cout<<"Fibonacci number at position "<<nth<<" is: "
 << rFibNum(firstFibNum, secondFibNum, nth)<<endl;

 return 0;
}

int rFibNum(int a, int b, int n)
{
 if(n == 1)
 return a;
 else if(n == 2)
 return b;
 else
 return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}

```

**运行实例：**在本运行实例中，用户输入用阴影表示。

#### 运行实例 1

```

Enter first Fibonacci number: 2
Enter second Fibonacci number: 3
Enter desired Fibonacci number: 6
Fibonacci number at position 6 is: 31

```

#### 运行实例 2

```

Enter first Fibonacci number: 3
Enter second Fibonacci number: 4
Enter desired Fibonacci number: 6
Fibonacci number at position 6 is: 29

```

#### 运行实例 3

```

Enter first Fibonacci number: 12
Enter second Fibonacci number: 13
Enter desired Fibonacci number: 15
Fibonacci number at position 15 is: 9582

```

### 例 11.3 汉诺塔问题

在19世纪,欧洲流行一种称为汉诺塔的游戏。相传 Brahama 寺庙的僧侣们在宇宙创立之初被赋予三根钻石柱子,其中一根柱子上串有64个金碟子,每一个金碟子的尺寸都比下面的碟子略小,僧侣们的任务是将第1根柱子上的64个金碟子移动到第3根柱子上,移动碟子的规则如下:

1. 一次只可以移动一个碟子。
2. 拿掉的碟子必须放在其中一根柱子上。
3. 大碟子不可以放在比它小的碟子之上。

僧侣们还被告知,一旦他们将所有的金碟子从第1根柱子移动到第3根柱子上,宇宙将会终结。

这里的目标是编写一个程序,输出在从第1根柱子传送到第3根柱子的过程中所有的移动序列。

图 11.6 显示了三个碟子的汉诺塔问题。

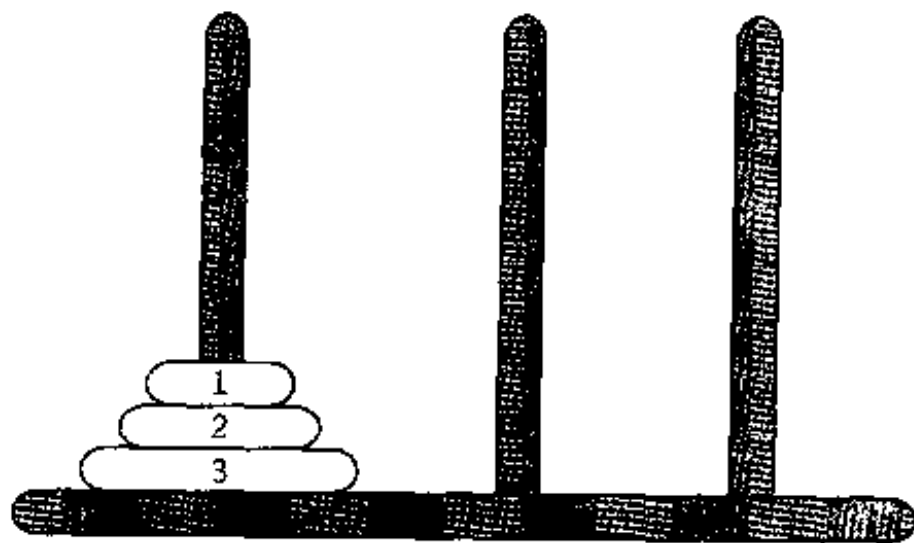


图 11.6 三个碟子的汉诺塔问题

和前面一样,我们以递归的方式进行思考。首先考虑1号柱只包含一个碟子的情形,这时,碟子可以直接从1号柱移动到3号柱。然后考虑1号柱包含两个碟子的情形,首先将第1个碟子从1号柱移动到2号柱,然后将第2个碟子从1号柱移动到3号柱。最后,将第1个碟子从2号柱移动到3号柱。下面考虑1号柱包含3个碟子的情况进而推广为64个碟子的情况(事实上,可以是任意数量的碟子)。

假设1号柱上有三个碟子,要将3号碟子移动到3号柱,其上的两个碟子必须被移到2号柱。然后将3号碟子从1号柱移到3号柱。要将上面两个碟子从2号柱移到3号柱,必须使用与前面相同的策略,只是这时用1号柱作为过渡。图 11.7 显示了三个碟子的汉诺塔问题的解法。

现在将这一问题归纳为64个碟子的情况。最开始,1号柱包含全部的64个碟子。直到其上的63个碟子从1号柱移到了2号柱,64号碟子才能从1号柱移到3号柱。所以首先将上面63个碟子从1号柱移动到2号柱,然后将64号碟子从1号柱移到3号柱。要将63号碟子从2号柱移到3号柱,首先将上面62个碟子从2号柱移到1号柱,然后将63号碟子从2号柱移到3号柱。使用类似的过程来移动62号碟子。将以上的讨论转换成下列以伪代码形式给出的递归算法。假设1号柱上面有 $n$ 个碟子,这里 $n \geq 1$ 。

1. 将上面的 $n-1$ 个碟子从1号柱移动到2号柱,用3号柱作为过渡。
2. 将 $n$ 号碟子从1号柱移到3号柱。
3. 将上面的 $n-1$ 个碟子从2号柱移动到3号柱,用1号柱作为过渡。

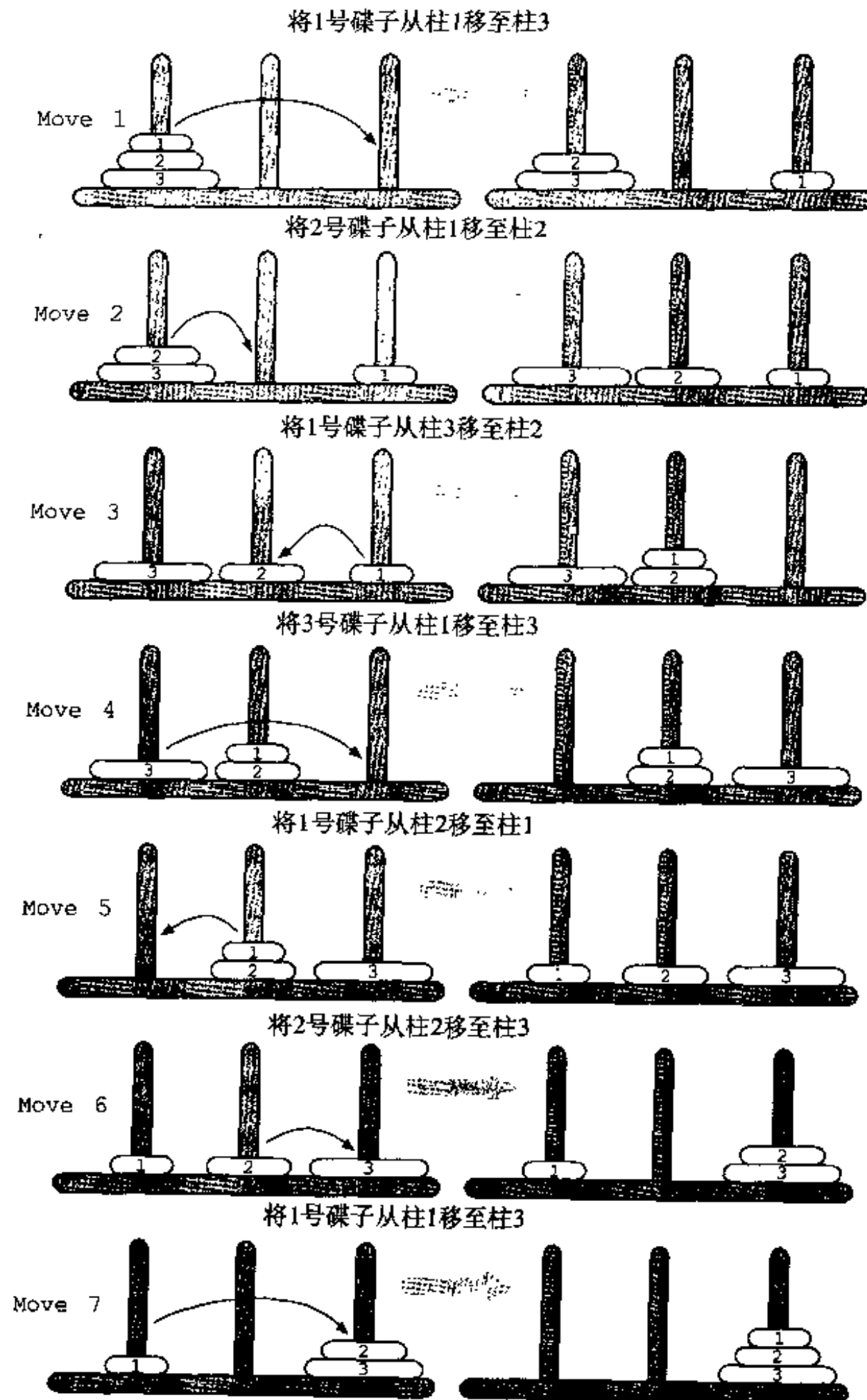


图 11.7 三个碟子的汉诺塔问题的解法

将该算法转换为 C++ 函数：

```
void moveDisks(int count, int needle1, int needle3, int needle2)
{
 if(count > 0)
 {
 moveDisks(count-1, needle1, needle2, needle3);
 cout<<"Move disk "<<count<<" from "<<needle1
 <<" to "<<needle3<<". "<<endl;
 moveDisks(count-1, needle2, needle3, needle1);
 }
}
```

### 11.2.1 汉诺塔：时间分析

如果1号柱有3个碟子,那么将所有这3个碟子从1号柱移到3号柱所需要的移动步数为 $2^3 - 1 = 7$ 。类似地,如果1号柱有64个碟子,那么将所有这64个碟子从1号柱子移到3号柱子所需要的移动步数为 $2^{64} - 1$ 。由于:

$$\begin{aligned} 2^{10} &\approx 1024 \approx 1000 = 10^3, \\ 2^{64} &= 2^4 \times 2^{60} \approx 2^4 \times 10^8 = 1.6 \times 10^{13} \end{aligned}$$

一年中所含有的秒数约等于 $3.2 \times 10^7$ 。假设僧侣每秒移动一个盘子,而且不需要休息,那么:

$$\begin{aligned} 1.6 \times 10^{13} &= 5 \times 3.2 \times 10^{11} = 5 \times (3.2 \times 10^7) \times 10^{11} \\ &= (3.2 \times 10^7) \times (5 \times 10^{11}) \end{aligned}$$

将所有64个碟子从1号柱移动到3号柱所需时间大致为 $5 \times 10^{11}$ 年。据估计,宇宙据今大约有150亿( $= 1.5 \times 10^{10}$ )年。而 $5 \times 10^{11} = 50 \times 10^{10} \approx 33 \times (1.5 \times 10^{10})$ 。这样推算,宇宙的寿命将会是其现在的33倍。

假设一台计算机每秒可以进行10亿( $= 10^9$ )次移动。那么这台计算机一年可以进行的移动数是:

$$(3.2 \times 10^7) \times 10^9 = 3.2 \times 10^{16}$$

那么产生 $2^{64}$ 次移动所需时间是:

$$2^{64} \approx 1.6 \times 10^{13} = 1.6 \times 10^{16} \times 10^{-3} = (3.2 \times 10^{16}) \times 500$$

因此,对于一台每秒产生10亿次移动的计算机,大约需要500年的时间完成 $2^{64}$ 次移动。

## 11.3 递归还是迭代

在第5章中,曾设计过一个程序用来计算斐波那契数。该程序使用了一个循环来进行计算。换句话说,第5章所设计的程序使用了迭代控制结构来重复执行一系列的语句。迭代控制结构使用循环结构,例如while、for或do...while来重复执行一系列的语句。在例11.2中,设计了一个递归函数来计算斐波那契数。从中可以看出,在递归中是通过函数反复调用其自身来实现重复执行一系列语句的。进一步而言,在递归中使用选择控制结构来控制重复调用。

在第10章中,使用了迭代控制结构(for循环)来查找数组中的最大元素。在本章中,我们使用了递归来查找。同样可以使用迭代控制逻辑来设计出计算非负整数阶乘的算法。之所以使用了递归,是为了说明递归是如何工作的。

因此可以看到,在处理一类特殊问题时,通常有两种途径——迭代和递归。用户也许会询问迭代和递归究竟哪一个更好?对这一问题不能简单地回答,从问题的本质上讲,确定最佳解法的一个关键因素是效率。

在例7.6(第7章)中,当跟踪问题的执行轨迹时,可以看出每当一个函数被调用时,系统都会为其参数列表和内部(自动)变量分配内存空间。当函数结束时,这部分内存空间会释放掉。

本章中,当跟踪递归函数的执行时,每一次(递归)调用都有其自己的一系列参数和内部(自动)变量分配的内存空间,然后在函数退出调用时再将其释放。每次执行(递归)函数都会伴随着内存空间和计算时间上的开销。因此,递归函数的执行速度要慢于其迭代实现方式。在较慢的计算机上,特别是那些内存有限的机器上,递归函数执行的速度会比较慢。

然而,今天的计算机拥有廉价的内存和快速的计算速度。因此,递归函数执行的低效并不会察



觉到。考虑到当今计算机强大的计算能力,迭代和递归两种算法的取舍取决于问题的实质。当然,对于某些问题,比如任务控制系统来说,效率绝对是至关重要的。因此,效率就成为方案选取的决定性因素。

作为一般规则,如果认为迭代解法比递归解法更为明显和易于理解,那么就可以使用迭代,同时也会更高效。另一方面,客观存在着一些问题(例如汉诺塔问题),其递归解法更易于理解或更易于构造(事实上,为汉诺塔问题构造迭代解法是非常困难的)。考虑到递归的威力,如果一个问题有着内在的递归规律,那么就应该考虑使用递归解法。

## 11.4 编程实例:将一个数字从二进制表示转换为十进制表示

计算机语言(称为机器语言)是一系列的0和1组成的。当在键盘上按下A键时,01000001被存入计算机。而且,在ASCII字符集中A的编码是65。事实上,A的二进制表示是01000001,而其十进制表示是65。

通常所使用的计数系统称为十进制系统,计算机所使用的计数系统称为二进制系统。在本例和下一编程实例中,将会讨论如何将数字从十进制表示转换为二进制表示,或者相反。

**二进制到十进制** 要将数字从二进制表示转换为十进制,需要首先找到二进制中每一位的权值。二进制数每一位的权值是从右向左分配的。最右边的权值为0,与其相邻的左一位权值为1,再向左一位的权值为2,依次类推。对于二进制数1001101,它的每一位权值如下所示:

|        |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|
| weight | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|        | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

可以使用每一位权值来计算相应的十进制数。对于每一位,将其乘以2的权值次幂然后将所有的数加起来。对于上面的二进制数,相应的十进制数为:

$$\begin{aligned}
 &1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
 &= 64 + 0 + 0 + 8 + 4 + 0 + 1 \\
 &= 77
 \end{aligned}$$

对于一个将二进制数转化为十进制数的程序,需要注意两点:(1)必须已知二进制数中每一位的权值,(2)权值是按照从右向左的顺序分配的。因为现在不知道要转换的二进制数有多少位,所以必须从右向左进行每一位的处理。

在处理完一位后,可以将权值加1,赋给其左边相邻的一位。同样,每一位必须从二进制数中取出并乘以2的权值次幂。要将位取出,可以使用求余操作。请看下而以伪代码形式给出的递归算法:

```

if(binaryNumber > 0)
{
 bit = binaryNumber % 10; //extract the rightmost bit
 decimal = decimal + bit * power(2, weight);
 binaryNumber = binaryNumber / 10; //remove the rightmost bit
 weight++;
 convert the binaryNumber into decimal
}

```

算法假定为 decimal 和 weight 分配的内存存在使用此算法前已经被赋值为0。该算法转换为 C++ 递归函数为:

```

void binToDec(int binaryNumber, int& decimal, int& weight)

```

```

{
 int bit;
 if(binaryNumber > 0)
 {
 bit = binaryNumber % 10;
 decimal = decimal + bit * static_cast<int>(pow(2, weight));
 binaryNumber = binaryNumber / 10;
 weight++;
 binToDec(binaryNumber, decimal, weight);
 }
}

```

在此函数中, decimal 和 weight 为引用参数。与其对应的实参相应地初始化为 0。在展开最右边一位后, 函数更新 decimal 的值和下一位的 weight 值。假设 decimalNumber 和 bitWeight 是 int 型变量, 考虑下面的语句:

```

decimalNumber = 0;
bitWeight = 0;
binToDec(1101, decimalNumber, bitWeight);

```

图 11.8 跟踪了上面语句的执行情况, 它显示了每一次函数调用后变量 decimalNumber 和 bitWeight 的值的变化情况。

```

binToDec(1101, decimalNumber, bitWeight);

```

图 11.8 中, 每个下箭头代表着一次成功的函数调用。因为函数 binToDec() 的最后一行语句是一个函数调用, 在这行语句执行过后, 没有任何情况发生。在语句:

```

binToDec(1101, decimalNumber, bitWeight);

```

执行之后, 变量 decimalNumber 的值是 13。

下面的 C++ 程序测试了函数 binToDec():

```

//Chapter 11: Program - Binary to Decimal
#include <iostream>
#include <cmath>
using namespace std;
void binToDec(int binaryNumber, int& decimal, int& weight);
int main()
{
 int decimalNum;
 int bitWeight;
 int binaryNum;

 decimalNum = 0;
 bitWeight = 0;
 cout<<"Enter number in binary: ";
 cin>>binaryNum;
 cout<<endl;
 binToDec(binaryNum, decimalNum, bitWeight);
 cout<<"Binary "<<binaryNum<<" = "<<decimalNum
 <<" decimal"<<endl;
 return 0;
}

void binToDec(int binaryNumber, int& decimal, int& weight)

```



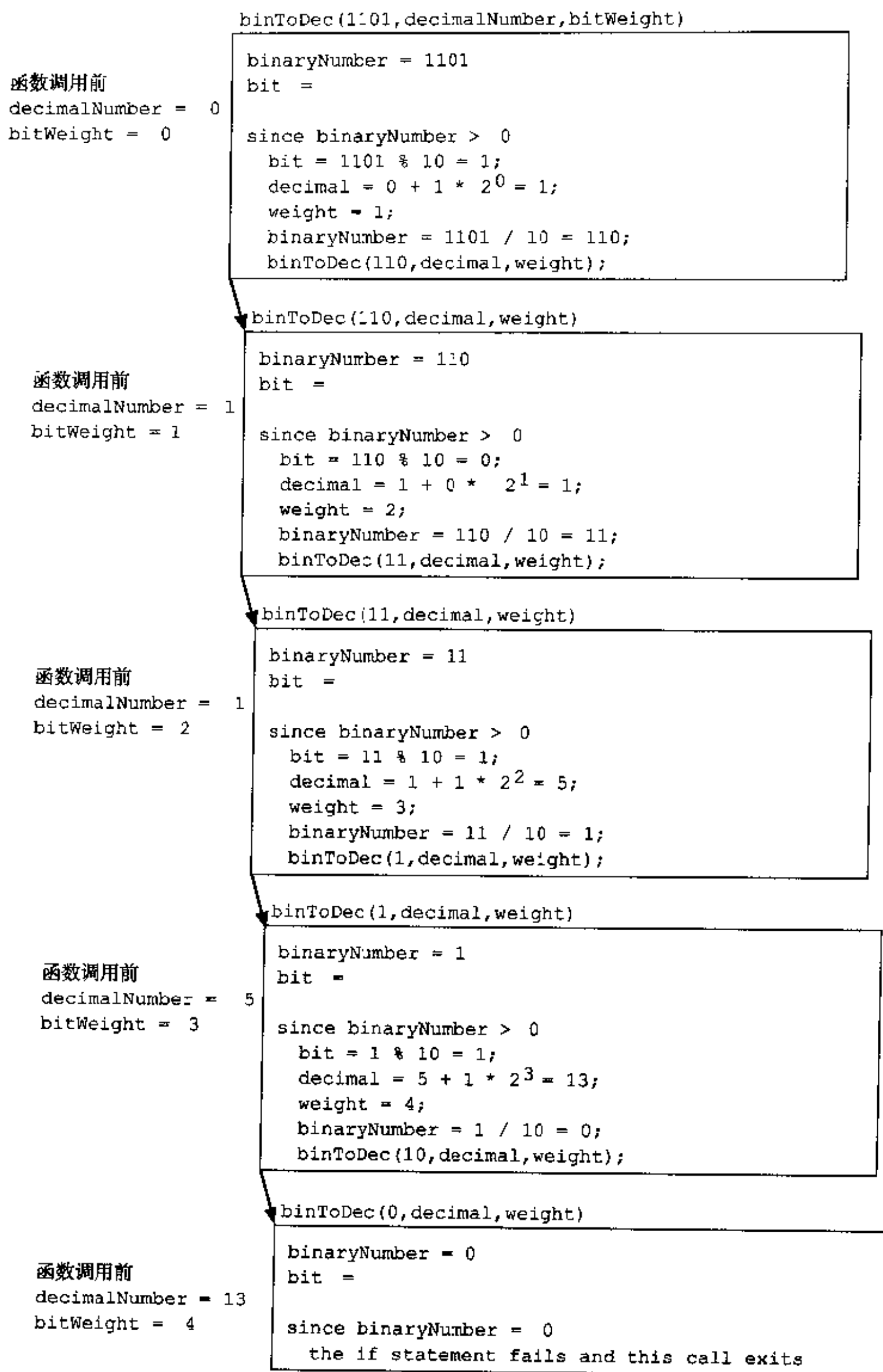


图 11.8 binToDec(1101, decimalNumber, bitWeight) 的执行情况

```

{
 int bit;
 if(binaryNumber > 0)
 {
 bit = binaryNumber % 10;
 decimal = decimal + bit * static_cast<int>(pow(2, weight));
 binaryNumber = binaryNumber / 10;
 weight++;
 binToDec(binaryNumber, decimal, weight);
 }
}

```

**运行实例：**在本运行实例中，用户的输入用阴影表示。

Enter number in binary: 11010110

Binary 11010110 = 214 decimal

## 11.5 编程实例：将一个数字从十进制转换为二进制

前面的编程实例讨论和设计了如何将数字从二进制表示转换为十进制格式。本例将讨论并使用递归将一个非负整数从十进制形式转换成二进制形式。

设  $x$  是一个整数。将  $x$  被 2 除所得的余数称为  $x$  的最右一位。因此，33 的最右一位是 1，因为  $33\%2$  等于 1。28 的最右一位是 0，因为  $28\%2$  等于 0。

首先用一个例子来说明算法如何将一个整数从十进制形式转换为二进制形式。

假设需要计算 35 的二进制表示，首先，将 35 被 2 除，商是 17，余数（即 35 的最右一位）是 1。接下来，将 17 被 2 除，商是 8，余数（即 17 的最右一位）是 1。然后，将 8 被 2 除，商是 4，余数（即 8 的最右一位）是 0。继续这一过程，直到商为 0。

35 的最右一位直到打印了 17 的最右一位才能够打印出来；17 的最右一位直到打印了 8 的最右一位才能够打印出来，依次类推。因此，35 的二进制表示是 17（即 35 被 2 除的商）的二进制表示以及 35 的最右一位。

这样，要将一个整数  $num$  从十进制表示转换成二进制表示，需要首先将  $num$  除以 2 的商转换成二进制数，然后将  $num$  的最右一位添加到  $num$  除以 2 的二进制表示的后面即可。

将以上讨论转换为如下递归算法，其中  $binary(num)$  代表  $num$  的二进制表示：

1. 如果  $num = 0$ ,  $binary(num) = num$ .
2. 如果  $num > 0$ ,  $binary(num) = binary(num/2)$  并连接  $num \% 2$ 。

下面的递归函数实现了该算法：

```

void decToBin(int num, int base)
{
 if(num > 0)
 {
 decToBin(num/base, base);
 cout<<num % base;
 }
}

```

图 11.9 跟踪了下面语句的执行：

```
decToBin(13, 2);
```

在这里, num 是 13, base 是 2。

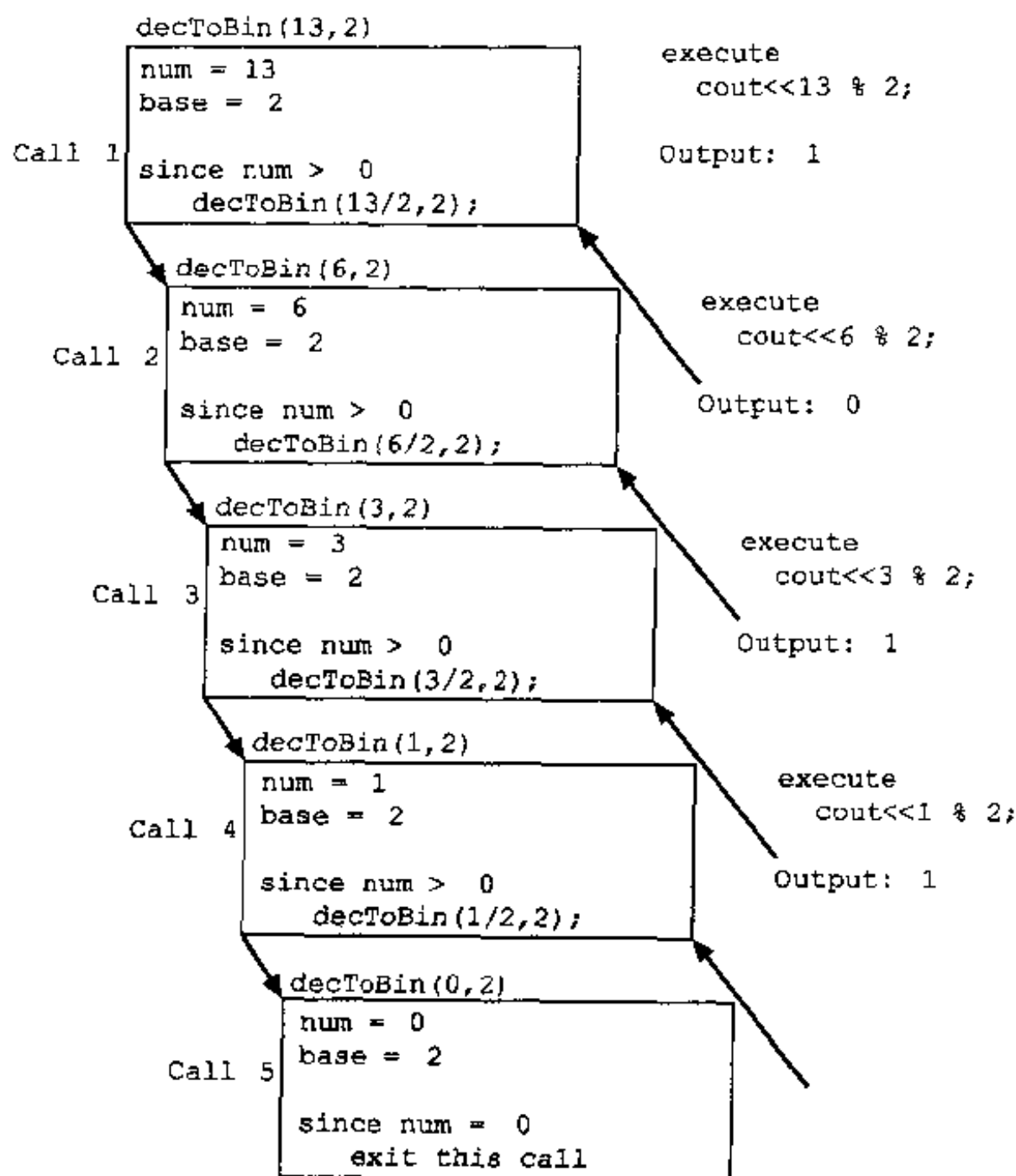


图 11.9 decToBin(13, 2) 的执行情况

由于 call 5 中的 if 语句失败, 函数调用并不打印任何信息。第 1 次输出是 call 4 产生的, 打印出 1; 第 2 次输出是 call 3 产生的, 打印出 1; 第 3 次输出是 call 2 产生的, 打印出 0; 第 4 次输出是 call 1 产生的, 打印出 1。因此, 语句:

```
decToBin(13, 2);
```

的输出是:

```
1101
```

下面的 C++ 程序测试了函数 decToBin():

```
//Chapter 11: Program - Decimal to Binary
#include <iostream>
using namespace std;
void decToBin(int num, int base);
int main()
{
 int decimalNum;
 int base;
```

```
 base = 2;

 cout<<"Enter number in decimal: ";
 cin>>decimalNum;
 cout<<endl;
 cout<<"Decimal "<<decimalNum<<" = ";
 decToBin(decimalNum, base);
 cout<<" binary"<<endl;

 return 0;
 }

 void decToBin(int num, int base)
 {
 if(num > 0)
 {
 decToBin(num/base, base);
 cout<<num % base;
 }
 }
}
```

运行实例：在本运行实例中，用户的输入用阴影表示。

Enter number in decimal: 57

Decimal 57 = 111001 binary

## 11.6 小结

1. 通过将问题转化为其自身简化形式加以解决的过程称为递归。
2. 递归定义将问题定义为其自身的简化形式。
3. 每个递归定义含有一种或多种基本形式。
4. 递归算法通过将问题转化为简化方式加以解决。
5. 每一个递归算法含有一种或多种基本形式。
6. 基本形式的答案可以直接得到。
7. 如果一个函数调用其自身，则称之为递归函数。
8. 递归算法是通过递归函数实现的。
9. 每一个递归函数必须含有一种或多种基本形式。
10. 通常，递归将问题分解为其自身的简化形式。
11. 一般形式必须能最终归结为基本形式。
12. 基本形式停止递归。
13. 逻辑上讲，一个递归函数可以有无限多个自身的副本。对递归函数的每一次调用都有其各自独立的函数体副本。
14. 对递归函数的每一次调用都有其各自独立的参数列表和临时变量副本。
15. 每次递归调用必须在控制权返回到前一次调用时执行完毕。前次调用在本次调用语句之后立即继续执行。
16. 如果最后一条执行语句是递归调用，称为尾部递归函数。

## 11.7 练习

1. 判断下列表述正误。

- a. 每一个递归定义必须含有一种或多种基本形式。
- b. 每一个递归函数必须含有一种或多种基本形式。
- c. 一般形式可以停止递归。
- d. 在一般形式中, 问题的解可以直接获得。
- e. 递归函数通常返回一个值。

2. 考虑下面递归函数:

```
void funcRec(int u, char v)
{
 if(u == 0)
 cout<<v;
 else if(u == 1)
 cout<<static_cast<char>(static_cast<int>(v) + 1);
 else
 funcRec(u - 1, v);
}
```

回答下列问题:

- a. 指出基本形式。
- b. 指出一般形式。
- c. 下面语句的输出是什么?

```
funcRec(5, 'A');
```

3. 对于下列函数:

```
int test(int x, int y)
{
 if(x == y)
 return x;
 else if(x > y)
 return (x + y);
 else
 return test(x + 1, y - 1);
}
```

下面语句的输出是什么?

- a. `cout<<test(5, 10)<<endl;`
- b. `cout<<test(3, 9)<<endl;`

4. 对于下列函数:

```
int Func(int x)
{
 if(x == 0)
 return 2;
 else if(x == 1)
 return 3;
 else
 return (Func(x - 1) + Func(x - 2));
}
```

下面语句的输出是什么?

- a. `cout<<Func(0)<<endl;`
- b. `cout<<Func(1)<<endl;`
- c. `cout<<Func(2)<<endl;`
- d. `cout<<Func(5)<<endl;`

## 11.8 编程练习

1. 编写一个递归函数 `vowels()`，返回一个字符串中的元音并给出测试程序。
2. 编写一个递归函数，返回一个 `int` 型数组中所有元素值的和并给出测试程序。
3. 当字符串前后读取结果均相同时，该字符串称做回文。例如，“`madam`”和“`madam I'm adam`”都是回文。请编写一个使用递归函数来检查一个字符串是否是回文的程序。要求程序中必须包含一个带有返回值的递归函数，当字符串是回文时返回 `true`；否则返回 `false`。不要使用任何全局变量。
4. 编写一个程序，使用递归函数来反序打印给定字符串。要求必须包含一个反序打印字符串的递归函数；不要使用任何全局变量。
5. 编写一个递归函数 `reverseDigits()`，它以一个整数作为参数，同时返回该数字的逆序并给出测试程序。
6. 编写一个递归函数 `power()`，以两个整数  $x$ （非 0）和  $y$  为参数并返回  $x$  的  $y$  次幂。可以使用以下递归定义来计算  $x$  的  $y$  次幂。

if  $y \geq 0$ ,

$$power(x, y) = \begin{cases} 1 & \text{if } y = 0 \\ x & \text{if } y = 1 \\ x * power(x, y-1) & \text{if } y > 1 \end{cases}$$

if  $y < 0$ ,

$$power(x, y) = \frac{1}{power(x, -y)}$$

7. (最大公约数) 给定两个整数  $x$  和  $y$ ，下面的递归定义给出  $x$  和  $y$  的最大公约数：

$$gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, x \% y) & \text{if } y \neq 0 \end{cases}$$

注意：在此定义中，`%` 是指 `mod` 操作。

编写一个递归函数 `gcd()`，以两个整数  $x$  和  $y$  为参数，返回它们的最大公约数并给出测试程序。

8. (递归连续查找) 第 10 章中给出的顺序查找算法是非递归的，编写一个递归版本的顺序查找函数。
9. (递归二分查找) 第 10 章中给出的二分查找算法是非递归的，编写一个递归版本的二分查找函数。
10. 在编程实例中，曾将一个整数的十进制表示转换为二进制表示，读者已经学会了如何将十进制数转化为相应的二进制数。计算机科学中还涉及到另外两种计数系统：八进制数和十六进制数。可以构造一个由八进制数转换成十六进制数的算法。

八进制计数系统中的数值变化范围是 0, 1, 2, 3, 4, 5, 6 和 7。十六进制计数系统中的数值变化范围是 0, 1, 2, 3, 4, 5, 6, 7, 9, A, B, C, D, E 和 F。这里，A 对应的十进制数是 10，B 对应的十进制数是 11，依次类推。

将十进制数转换成等值的八进制数（十六进制数）与前面讨论过的转换二进制数相同。这里可以将十进制数被 8（对八进制数）除和被 16（对十六进制数）除。假设  $a_b$  表示以  $b$  为基



数的数  $a$ 。例如,  $75_{10}$  表示以 10 为基数的 75 (十进制),  $83_{16}$  代表以 16 为基数的 83 (十六进制数)。那么:

$$753_{10} = 1361_8$$

$$753_{10} = 2F1_{16}$$

将十进制数转换为二、八和十六进制的方法可以扩展为以任何数为基数。假设要将一个十进制数转化为等值的以  $b$  为基数的数, 这里  $b$  介于 2 到 36 之间, 就可以按照将十进制转换为二进制的方法进行转换, 不同的是要将数  $n$  被  $b$  除。

以 20 为基数的数, 数值的变化范围是 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I 和 J。

编写一个程序, 使用递归函数将一个十进制数转换为以  $b$  为基数的数, 其中  $b$  介于 2 到 36 之间。程序应让用户输入十进制数和需要的基数。

使用下列数据测试程序:

9098 基数为 20

692 基数为 2

753 基数为 16

## 第 12 章 记录（结构体）

### 本章要点：

- 什么是记录（结构）
- 基于结构的各種操作
- 使用结构来操作数据的方法
- 结构和函数之间的关系
- 如何在结构中使用数组
- 如何使用一个以结构为元素的数组

在第 9 章和第 10 章中，读者学习了如何使用数组将一系列相同类型的数据组织起来，以及如何处理数组中存储的数据。我们还学习了如何进行顺序表操作，比如查找和排序。

注意：跳过本章将不会造成任何不连续。

在本章中，将学习如何将互相关联的、不同类型的数据组织起来。C++ 提供了另一种称为结构体（在某些语言中称为“记录”）的结构化数据类型，从而将互相关联的不同类型数据组织在一起。数组是同类型数据组成的数据结构；结构体则是不同类型数据组成的数据结构。在本章中对结构体的论述与 C 语言中对结构体的论述相似。因此，本章将要介绍的结构可以称为“类 C 的结构体”。第 13 章将会介绍和讨论另外一种结构化的数据类型——类。

### 12.1 记录（结构体）

假设要编写一个处理学生数据的程序。每一名学生的记录至少包括该学生的姓名、ID、GPA、所学课程以及课程分数。其中各个部分都与某一名学生相联系，然而各个部分分别属于不同的数据类型，我们不可能使用一个数组将所有与该学生相关的部分组织起来。C++ 提供了结构化的数据类型——结构体将不同类型的数据组织起来。将相互关联而又不属于同一类型的数据组织起来带来了很多好处。例如，使用一个变量作参数就可以将所有相关部分传递给函数。

结构体：由数量固定的组成部分构成的集合，其中的组成部分以按名称的方式加以访问，各组成部分可以是彼此不同的数据类型。

结构体的各个组成部分称为结构体的成员。在 C++ 中结构体的一般语法如下所示：

```
struct structName
{
 dataType1 identifier1;
 dataType2 identifier2;
 .
 .
 .
 dataTypeN identifierN;
};
```

在 C++ 中，struct 是一个保留字。结构体的成员即使被括在两个花括号之间（即，它们形成了



一个块)也不被认为是完整的声明。因此,右花括号后的分号对于结束结构体的声明是十分必要的。而这个分号同样是语法的一部分。

语句:

```
struct employeeType
{
 string firstName;
 string lastName;
 string address1;
 string address2;
 double salary;
 string deptID;
};
```

定义了一个由6个成员组成的结构体`employeeType`。成员 `firstName`、`lastName`、`address1`、`address2` 和 `deptID` 是 `string` 型的,而成员 `salary` 是 `double` 型的。

像其他任何类型定义一样,结构体是一个定义并非声明。即它仅仅定义了一种数据类型,没有任何内存因此被分配。

一旦数据类型被定义了,就可以用它声明相应类型的变量了。可以首先定义一个结构体 `studentType`,然后声明这种类型的变量。

```
struct studentType
{
 string firstName;
 string lastName;
 char courseGrade;
 int testScore;
 int programmingScore;
 double GPA;
};

//variable declaration
studentType newStudent;
studentType student;
```

上面语句声明了两个 `studentType` 型的结构体变量, `newStudent` 和 `student`。系统会分配足够大的内存来存放 `firstName`、`lastName`、`courseGrade`、`testScore`、`programmingScore` 和 `GPA` (见图 12.1)。

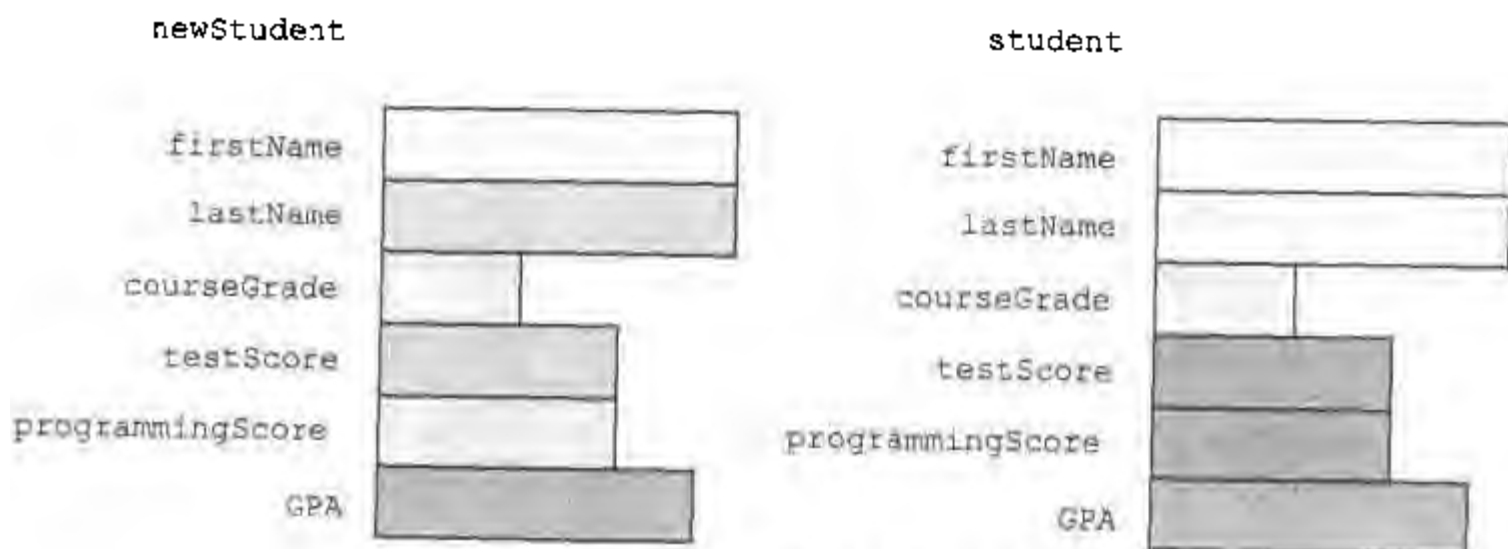


图 12.1 结构体 `newStudent` 和 `student`

### 12.1.1 访问结构体成员

在数组中,可以使用数组名并结合元素的相对位置(下标)来访问其元素。数组名和下标之间

使用方括号。要访问一个结构体的成员，可以使用结构体变量的名字和成员的名字，它们之间使用一个圆点隔开。访问结构体成员的语法为：

```
structVariableName.memberName
```

“结构体名字.成员名字”与其他任何变量一样。例如，`newStudent.courseGrade` 是一个 `char` 型的变量，`newStudent.firstName` 是一个字符串型的变量，等等。因此，可以像平时处理普通变量一样去处理结构体的成员。例如，可以将它们用于赋值语句或者输入、输出语句中。

在 C++ 中，圆点是一个运算符，称为成员访问运算符。

假设要将 `newStudent` 的成员 `GPA` 初始化为 0.0。下面的语句可以完成这一任务：

```
newStudent.GPA = 0.0;
```

类似地，语句：

```
newStudent.firstName = "John";
newStudent.lastName = "Brown";
```

将 “John” 存储在 `newStudent` 的成员 `firstName` 中，将 “Brown” 存储在 `lastName` 中。

在前面三条赋值语句执行过后，`newStudent` 如图 12.2 所示。

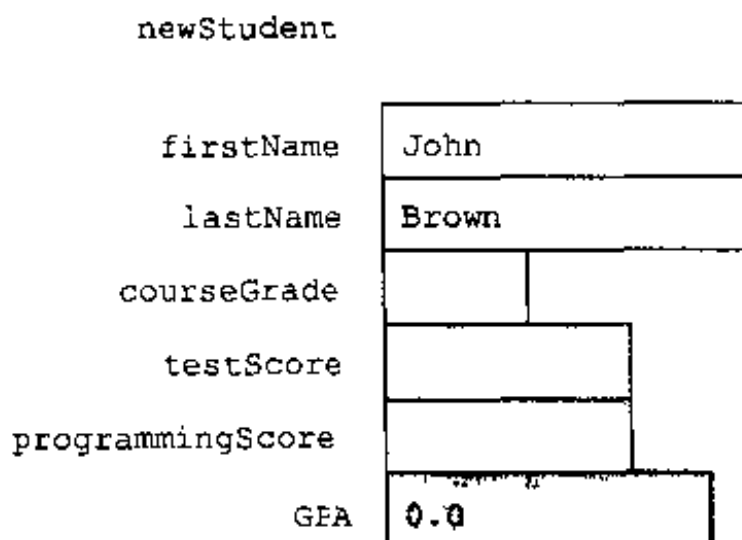


图 12.2 结构体 newStudent

语句：

```
cin>>newStudent.firstName;
```

从标准输入设备中读入字符串并将其存储在 `newStudent.firstName` 中。

语句：

```
cin>>newStudent.testScore>>newStudent.programmingScore;
```

从键盘读入两个整数并将它们分别存储在 `newStudent.testScore` 和 `newStudent.programmingScore` 中。

假设 `score` 是一个 `int` 型的变量，语句：

```
score = newStudent.testScore + newStudent.programmingScore;
```

将 `newStudent.testScore` 与 `newStudent.programmingScore` 的内容相加并存储在 `score` 中。

下面的语句将分数存储在 `newStudent.courseGrade` 中：

```
if(score >= 90)
 newStudent.courseGrade = 'A';
else if(score >= 80)
 newStudent.courseGrade = 'B';
```

```

else if(score >= 70)
 newStudent.courseGrade = 'C';
else if(score >= 60)
 newStudent.courseGrade = 'D';
else
 newStudent.courseGrade = 'F';

```

### 12.1.2 赋值

通过赋值语句，可以将一个结构体变量的值赋给另一个相同类型的结构体变量。假设 newStudent 如图 12.3 所示。

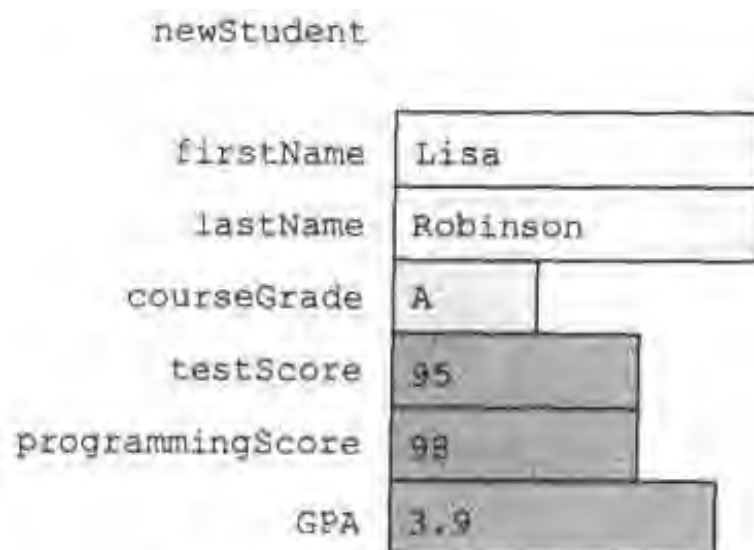


图 12.3 结构体 newStudent

语句：

```
student = newStudent;
```

将 newStudent 的内容复制给 student。在这条赋值语句执行过后，student 的值如图 12.4 所示。

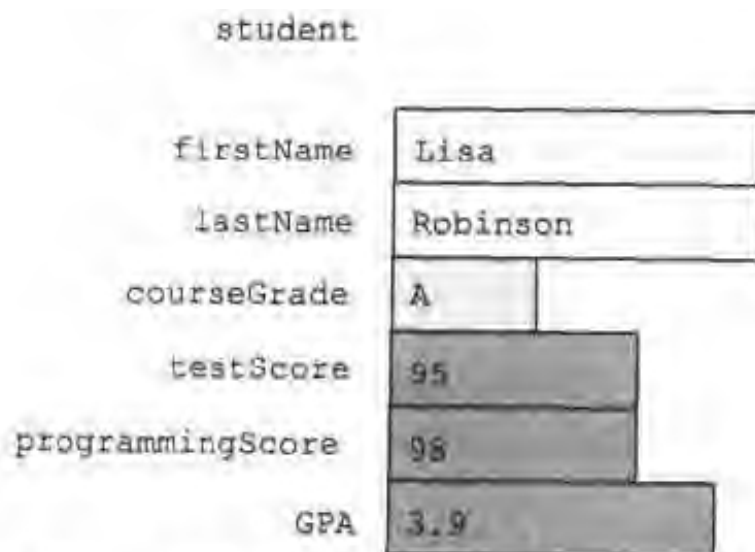


图 12.4 “student = newStudent;” 之后的 student

事实上，赋值语句：

```
student = newStudent;
```

与下面的语句等价：

```

student.firstName = newStudent.firstName;
student.lastName = newStudent.lastName;
student.courseGrade = newStudent.courseGrade;
student.testScore = newStudent.testScore;
student.programmingScore = newStudent.programmingScore;
student.GPA = newStudent.GPA;

```

### 12.1.3 比较(关系运算)

比较结构体变量只能在成员的范畴内进行。和数组一样,没有针对结构体一级的聚集关系运算。例如,之前声明的 `newStudent` 和 `student`。此外,假设想知道 `student` 和 `newStudent` 是否代表同一名学生,如果 `newStudent` 和 `student` 具有相同的 `firstName` 和 `lastName`,则它们代表的是同一名学生,必须对它们进行以成员为单位的比较:

```
if(student.firstName == newStudent.firstName &&
 student.lastName == newStudent.lastName)
```

```
·
·
·
```

尽管可以使用赋值语句将一个结构体的内容赋给另一个相同类型的结构体,却并不能在结构体变量之间使用关系运算符。因此,下面的语句将是非法的:

```
if(student == newStudent) //illegal
```

```
·
·
·
```

### 12.1.4 输入/输出

没有针对结构体变量的聚集输入/输出操作。一次只能够读入结构体变量中的一个成员。同样,一次只能够写入一个结构体变量中的成员。

读者已经看到了如何将数据读入结构体变量,现在让我们看看如何将一个结构体变量输出。语句:

```
cout<<newStudent.firstName<<" "<<newStudent.lastName<<" "
 <<newStudent.courseGrade<<" "<<newStudent.testScore<<" "
 <<newStudent.programmingScore<<" "<<newStudent.GPA<<endl;
```

将输出结构体变量 `newStudent` 的内容。

### 12.1.5 结构体变量和函数

回忆一下,数组只能以引用的方式传递给函数,而且函数不能够返回数组类型的变量。然而,

- 一个结构体变量既可以以传值方式,也可以以引用的方式作为参数进行传递。
- 函数可以返回结构体类型的返回值。

下面的函数读取并存储一名学生的 `firstName`、`lastName`、`testScore`、`programmingScore` 和 `GPA`。同时它会确定该学生的成绩等级并存储在其成员 `courseGrade` 中。

```
void readIn(studentType& student)
{
 int score;

 cin>>student.firstName>>student.lastName;
 cin>>student.testScore>>student.programmingScore;
 cin>>student.GPA;

 score = newStudent.testScore + newStudent.programmingScore;

 if(score >= 90)
 student.courseGrade = 'A';
```

```

 else if(score >= 80)
 student.courseGrade = 'B';
 else if(score >= 70)
 student.courseGrade = 'C';
 else if(score >= 60)
 student.courseGrade = 'D';
 else
 student.courseGrade = 'F';
}

```

语句:

```
readIn(newStudent);
```

调用函数 readIn(), 该函数将适当的信息存储到变量 newStudent 中。

类似地, 可以编写函数用来打印结构体变量的内容。例如, 下面的函数会在屏幕上输出一个 studentType 类型结构体变量的内容:

```

void printStudent(studentType student)
{
 cout<<student.firstName<<" "<<student.lastName<<" "
 <<student.courseGrade<<" "<<student.testScore<<" "
 <<student.programmingScore<<" "<<student.GPA<<endl;
}

```

### 12.1.6 数组与结构体

通过前面的讨论可以看出, 数组和结构体之间既有相似之处又存在不同之处。如表 12.1 所示:

表 12.1 数组和结构体

|   | 操作    | 数组       | 结构体   |
|---|-------|----------|-------|
| 1 | 算术    | 无        | 无     |
| 2 | 赋值    | 无        | 有     |
| 3 | 输入/输出 | 无(字符串除外) | 无     |
| 4 | 比较    | 无        | 无     |
| 5 | 参数传递  | 只有引用     | 传值和引用 |
| 6 | 函数返回值 | 不能       | 能     |

### 12.1.7 结构体内的数组

在第 10 章中讨论了顺序表的相关算法。一个顺序表有两个相关的要素: 值(即元素)和长度。在第 10 章讨论过的查找和排序函数中, 需要将其作为参数分别进行传递。因为值和长度都与顺序表相关联, 可以定义一个包含这两项的结构体, 然后只需向函数传递一个参数即可。首先, 定义下面的结构体:

```

const arraySize = 1000;
struct listType
{
 int listElem[arraySize]; //array containing the list
 int listLength; //length of the list
}
int seqSearch(const listType& list, int searchItem)
{
 int loc;
 bool found = false;
}

```

```

 for(loc = 0; loc < list.listLength; loc++)
 if(list.listElem[loc] == searchItem)
 {
 found = true;
 break;
 }
 if(found)
 return loc;
 else
 return -1;
}

```

在该函数中, 因为 listLength 是 list 的一个成员, 可以使用 list.listLength 来访问它。同样地, 可以通过 list.listElem[loc] 访问 list 的元素。

注意, 函数 seqSearch() 的第一个参数 list 声明为一个常量引用型参数, 这意味着 list 接收实参的地址, 但 list 并不能修改实参。有关常量引用将会在第 13 章中详细讨论。

类似地, 也可以重写排序、二分查找和其他顺序表处理函数。

### 12.1.8 数组中的结构体

假设一个公司有 50 名全职雇员, 需要打印出他们每月的薪水并跟踪累计消费状况。首先, 定义一个雇员的记录:

```

struct employeeType
{
 string firstName;
 string lastName;
 int personID;
 string deptID;
 double yearlySalary;
 double monthlySalary;
 double yearToDatePaid;
 double monthlyBonus;
};

```

每一名雇员有下列成员: firstName、lastName、personID、deptID、yearlySalary、monthlySalary、yearToDatePaid 和 monthlyBonus。

因为有 50 名雇员并且每一名雇员的数据类型是相同的, 所以可以使用一个有 50 个元素的数组来处理雇员的数据。

```
employeeType employees[50];
```

这条语句声明了一个有 50 个 employeeType 型元素组成的数组 employees (见图 12.5)。employees 的每元素都是一个结构变量。如图 12.5 显示了 employees[2] 的情况。

假设还进行了下面的声明:

```
int counter;
```

每一名雇员的(初始数据) firstName、lastName、personID、deptID 和 yearlySalary 由一个文件提供。为了讨论方便, 假设每一名雇员的数据存储在一个文件 employee.dat 中。下面的 C++ 代码将数据载入雇员的数组中。假设 yearToDatePaid 的初始值为 0, monthlyBonus 取决于雇员每个月的表现。



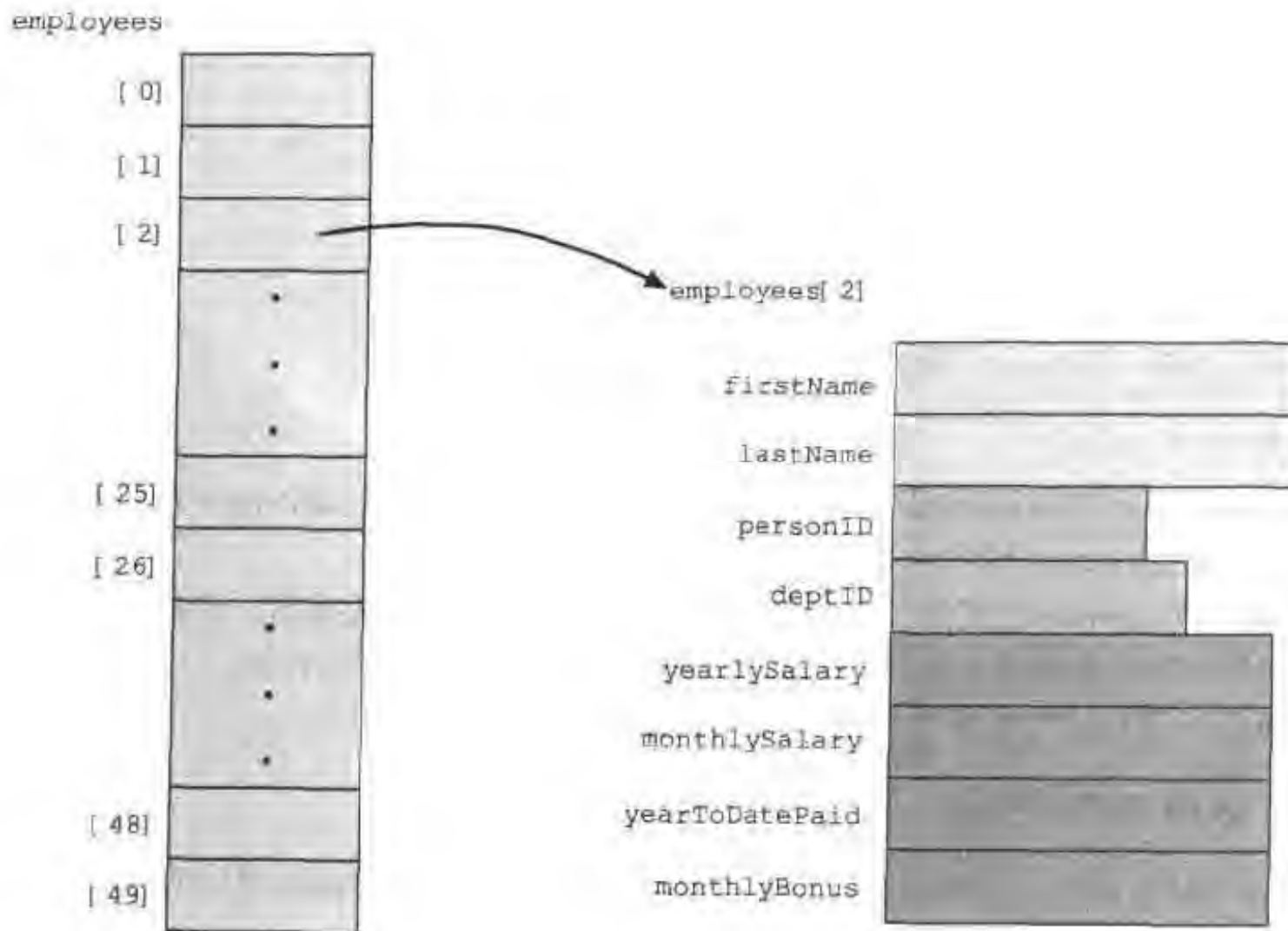


图 12.5 结构体数组 employees

```

ifstream infile; //input stream variable
//assume that employee.dat file has been opened
for(counter = 0; counter < 50; counter++)
{
 infile>>employees[counter] .firstName
 >>employees[counter] .lastName
 >>employees[counter] .personID
 >>employees[counter] .deptID
 >>employees[counter] .yearlySalary;
 employees[counter] .monthlySalary =
 employees[counter] .yearlySalary/12;
 employees[counter] .yearToDatePaid = 0.0;
 employees[counter] .monthlyBonus = 0.0;
}

```

对于一个给定的月份，monthlyBonus已经存储在每一名雇员的记录中，程序需要计算每月的薪水并更新 yearToDatePaid 的值。下面的循环计算并打印出雇员每月的薪水：

```

double payCheck; //variable to calculate the paycheck
for(counter = 0; counter < 50; counter++)
{
 cout<<employees[counter] .firstName<<" "
 <<employees[counter] .lastName<<" ";

 payCheck = employees[counter] .monthlySalary +
 employees[counter] .monthlyBonus;

 employees[counter] .yearToDatePaid =
 employees[counter] .yearToDatePaid +
 payCheck;

 cout<<setprecision(2)<<payCheck<<endl;
}

```

### 12.1.9 含有结构的结构

读者已经看到了结构和数组是如何结合在一起来组织信息的, 以及以数组为成员的结构体, 还看到了以结构体为元素的数组。在本节中, 将学习在结构内使用另外的结构体来组织数据的情况。

请看下面的雇员记录:

```
struct employeeType
{
 string firstname;
 string middlename;
 string lastname;
 string emplID;
 string address1;
 string address2;
 string city;
 string state;
 string zip;
 string hiremonth;
 string hireday;
 string hireyear;
 string quitmonth;
 string quitday;
 string quityear;
 string phone;
 string cellphone;
 string fax;
 string pager;
 string email;
 string deptID;
 double salary;
};
```

正如所看到的, 大量的信息被压缩到一个结构体内, 该结构体有 22 个成员。其中一些成员会比其他成员访问更为频繁; 而一些成员之间的关联要比其他成员更为紧密; 一些成员将会有着相似的潜在结构。例如, 雇佣日期和离职日期都是 `date` 型的数据。可以按照下面形式来重新组织这一结构:

```
struct nameType
{
 string first;
 string middle;
 string last;
};

struct addressType
{
 string address1;
 string address2;
 string city;
 string state;
 string zip;
};

struct dateType
{
 string month;
```



```

 string day;
 string year;
 };

 struct contactType
 {
 string phone;
 string cellphone;
 string fax;
 string pager;
 string email;
 };

```

这里，将雇员的名字、地址和联系方式分成了子项并且还定义了 `dateType` 结构。于是，可以重新构造雇员记录：

```

 struct employeeType
 {
 nameType name;
 string emplID;
 addressType address;
 dateType hiredate;
 dateType quitdate;
 contactType contact;
 string deptID;
 double salary;
 };

```

雇员结构体中的信息比前一种定义更容易管理。例如，假设要定义一位客户的记录，每一位客户都有 `first name`、`last name` 和 `middle name` 以及地址和某种联系方式。因此，可以通过将结构体 `nameType`、`addressType`、`contactType` 以及客户的特定成员信息组织在一起来方便地建立客户的记录。

下面，声明 `employeeType` 型变量并讨论如何访问其成员。

```

 //variable declaration
 employeeType newEmployee;
 employeeType employees[100]; //declare 100 employees' records

```

语句：

```
newEmployee.salary = 45678.00;
```

将 `newEmployee` 的 `salary` 设置为 45678.00。语句：

```

newEmployee.name.first = "Mary";
newEmployee.name.middle = "Beth";
newEmployee.name.last = "Simmons";

```

将 `newEmployee` 的 `firstname`、`middle name` 和 `last name` 分别设置为 “Mary”、“Beth” 和 “Simmons”。请注意，`newEmployee` 有一个 `name` 成员，可以通过 `newEmployee.name` 来访问它，另外请注意 `newEmployee.name` 是一个有三个成员的结构体。这里，将成员访问的标准应用到了结构体 `newEmployee.name` 的成员 `first`，所以，`newEmployee.name.first` 是存储名字的成员。

语句：

```
cin>>newEmployee.name.first;
```

读入并存储了一个字符串到 `newEmployee.name.first` 中。语句：

```
newEmployee.salary = newEmployee.salary * 1.05;
```

更新了 newEmployee 中的 salary 成员。

for 循环:

```
for(j = 0; j < 100; j++)
 cin>>employees[j].name.first>>employees[j].name.middle
 >>employees[j].name.last;
```

读入并存储了100名雇员的姓名, 因为 employees 是一个数组, 可以使用下标来访问其元素。例如, employees[50] 代表数组的第51个元素(回忆一下, 数组的下标是从0开始的)。因为 employees[50] 是一个结构, 所以可以使用成员访问方法来选择其中的某一成员。

## 12.2 编程实例: 销售数据分析

假设某一公司有6名销售人员, 在每个月末, 每一名销售人员的总销售额和该销售人员的 ID 以及月份被记录进一个文件。在每一年的年终, 公司经理要看到如下形式的报表:

```
----- Annual Sales Report -----
ID QT1 QT2 QT3 QT4 Total
12345 1892.00 0.00 494.00 322.00 2708.00
32214 343.00 892.00 9023.00 0.00 10258.00
23422 1395.00 1901.00 0.00 0.00 3296.00
57373 893.00 892.00 8834.00 0.00 10619.00
35864 2882.00 1221.00 0.00 1223.00 5326.00
54654 893.00 0.00 392.00 3420.00 4705.00
Total 8298.00 4906.00 18743.00 4965.00
```

Max Sale by SalesPerson: ID = 57373, Amount = \$10619.00

Max Sale by Quarter: Quarter = 3, Amount = \$18743.00

报表中, QT1 代表第一季度(月份1到3), QT2 代表第二季度(月份4到6), QT3 代表第三季度(月份7到9), QT4 代表第四季度(月份10到12)。

销售人员的 ID 存储在一个文件中, 销售数据存储在另一个文件中。销售数据是以下列形式进行存储的:

```
salesPersonID month saleAmount
.
.
.
```

此外, 销售数据有特定的存放顺序, 它并没有以 ID 来排序。

编写一个程序来输出以上形式的报表。

输入 包含每一位销售人员 ID 的文件和包含销售数据的文件。

输出 包含上述年销售报表的文件。

### 问题分析和算法设计

基于问题的需求, 显然, 每一名销售人员的主要信息是他们的 ID、季度销售额和年销售总额。因为信息属于不同的类型, 所以可以在结构体的帮助下进行组织, 定义如下:

```
struct salesPersonRec
{
 string ID; //salesperson's ID
```

```

double saleByQuarter[4]; //array to store the total
 //sales for each quarter
double totalSale; //salesperson's yearly sales amount
};

```

因为有 6 位销售人员，可以使用一个由 6 个元素组成的数组，其中每一个元素的数据类型是 `salesPersonRec`，定义如下：

```
salesPersonRec salesPersonList[noOfSalesPersons];
```

其中 `noOfSalesPersons` 的值是 6。

因为程序需要计算公司每一季度的总销售额，所需要一个包含 4 个元素的数组来存放数据。注意，这些数据将用来存放每一季度的最大销售额。因此，程序需要下面的数组：

```
double totalSaleByQuarter[4];
```

在整个讨论过程中都会使用它。数组 `salesPersonList` 如图 12.6 所示。

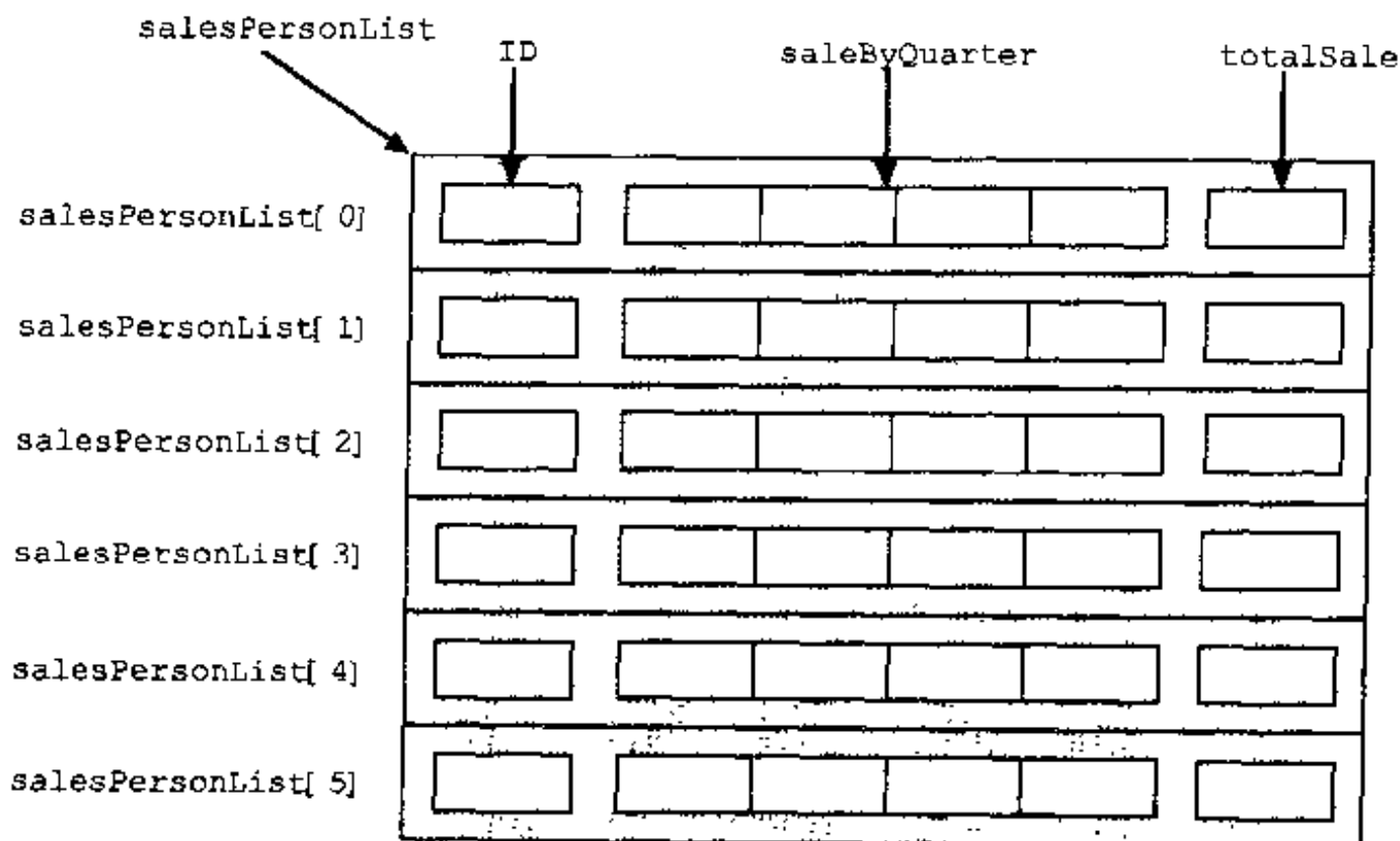


图 12.6 数组 `salesPersonList`

程序的第一步要将销售人员的 ID 读入到数组 `salesPersonList` 中，并且将每一位销售人员的季度销售额和销售总额初始化为 0。这一步骤之后，`salesPersonList` 如图 12.7 所示。

下一步要处理销售数据，这会更为简单。对文件中的每一个包含销售数据的项：

1. 读取销售人员的 ID、月份和当月销售额。
2. 查找数组 `salesPersonList` 以定位相应的信息。
3. 计算与月份相对应的季度。
4. 通过与将当月销售额相加来更新季度销售额。一旦销售数据文件处理完毕：
  - a. 计算销售人员的总销售额。
  - b. 计算季度总销售额。
  - c. 打印报表。

将讨论转换成算法：

1. 初始化数组 `salesPersonList`。

2. 处理销售数据。
3. 计算个人总销售额。
4. 计算季度总销售额。
5. 打印报表。
6. 计算并打印个人最大销售额。
7. 计算并打印最大季度销售额。

|                    | ID    | saleByQuarter |     |     |     | totalSale |
|--------------------|-------|---------------|-----|-----|-----|-----------|
| salesPersonList[0] | 12345 | 0.0           | 0.0 | 0.0 | 0.0 | 0.0       |
| salesPersonList[1] | 32214 | 0.0           | 0.0 | 0.0 | 0.0 | 0.0       |
| salesPersonList[2] | 23422 | 0.0           | 0.0 | 0.0 | 0.0 | 0.0       |
| salesPersonList[3] | 57373 | 0.0           | 0.0 | 0.0 | 0.0 | 0.0       |
| salesPersonList[4] | 35864 | 0.0           | 0.0 | 0.0 | 0.0 | 0.0       |
| salesPersonList[5] | 54654 | 0.0           | 0.0 | 0.0 | 0.0 | 0.0       |

图 12.7 初始化后的数组 salesPersonList

为降低主程序的复杂性,需要对以上7个步骤分别编写相应的函数。

函数 `initialize()`: 函数从输入文件中读入销售人员 ID, 并将其存储在 `salesPersonList` 数组中。同时初始化季度销售额和销售总额为 0。该函数的定义如下:

```
void initialize(ifstream& indata, salesPersonRec list[],
 int listSize)
{
 int count;
 quarterType quarter;
 for(count = 0; count < listSize; count++)
 {
 indata >> list[count].ID; //get salesperson's ID
 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter + 1))
 list[count].saleByQuarter[quarter] = 0.0;
 list[count].totalSale = 0.0;
 }
}
```

函数 `getData()`: 函数从输入文件中读入销售数据并将其存储在数组 `salesPersonList` 的相应部分。该函数的算法如下:

1. 读入销售人员的 ID、月份和当月销售额。
2. 查找定位数组 `salesPersonList` (因为没有存储销售人员的 ID, 将使用顺序查找)。
3. 根据月份确定季度。

4. 将当月销售额加进季度销售额中对其进行更新。

假设读入的数据项为：

57373 2 350

该销售人员的 ID 是 57373，月份是 2，销售额是 350。假设数组 salesPersonList 如图 12.8 所示。

|                    | ID    | saleByQuarter |        |        |        | totalSale |
|--------------------|-------|---------------|--------|--------|--------|-----------|
| salesPersonList[0] | 12345 | 150.80        | 0.0    | 0.0    | 654.92 | 0.0       |
| salesPersonList[1] | 32214 | 0.0           | 439.90 | 0.0    | 0.0    | 0.0       |
| salesPersonList[2] | 23422 | 0.0           | 0.0    | 0.0    | 564.76 | 0.0       |
| salesPersonList[3] | 57373 | 354.80        | 0.0    | 0.0    | 0.0    | 0.0       |
| salesPersonList[4] | 35864 | 0.0           | 0.0    | 763.90 | 0.0    | 0.0       |
| salesPersonList[5] | 54654 | 783.45        | 0.0    | 0.0    | 563.80 | 0.0       |

图 12.8 数组 salesPersonList

ID 57373 对应数组 salesPersonList[3]，而月份 2 对应季度 1。因此，我们将 350 与 354.80 相加得到新的季度销售额 704.80。在处理完这一项后数组如图 12.9 所示。

|                    | ID    | saleByQuarter |        |        |        | totalSale |
|--------------------|-------|---------------|--------|--------|--------|-----------|
| salesPersonList[0] | 12345 | 150.80        | 0.0    | 0.0    | 654.92 | 0.0       |
| salesPersonList[1] | 32214 | 0.0           | 439.90 | 0.0    | 0.0    | 0.0       |
| salesPersonList[2] | 23422 | 0.0           | 0.0    | 0.0    | 564.76 | 0.0       |
| salesPersonList[3] | 57373 | 704.80        | 0.0    | 0.0    | 0.0    | 0.0       |
| salesPersonList[4] | 35864 | 0.0           | 0.0    | 763.90 | 0.0    | 0.0       |
| salesPersonList[5] | 54654 | 783.45        | 0.0    | 0.0    | 563.80 | 0.0       |

图 12.9 处理完项 57373 2350 后的数组 salesPersonList

函数 `getData()` 的定义如下:

```
void getData(ifstream& infile, salesPersonRec list[],
 int listSize)
{
 int count;
 quarterType quarter;
 string sID;
 int month;
 double amount;

 infile>>sID; //Step 1
 while(infile)
 {
 infile>>month>>amount; //Step 1
 for(count=0; count< listSize; count++) //Step 2
 {
 if(sID == list[count].ID)
 {
 if(1 <= month && month <= 3) //Step 3
 quarter = QT1;
 else if(4 <= month && month <= 6)
 quarter = QT2;
 else if(7 <= month && month <= 9)
 quarter = QT3;
 else
 quarter = QT4;

 list[count].saleByQuarter[quarter] += amount; //Step 4
 break; //exit for loop
 } //end if
 } //end for

 infile>>sID; //Step 1
 } //end while
} //end getData
```

**函数 `saleByQuarter()`:** 函数找出每一季度的销售总额。要找到每一季度的销售总额, 需要将每一位销售人员的季度销售额相加。显然, 该函数必须访问数组 `salesPersonList` 和数组 `totalSaleByQuarter`, 同时, 该函数需要知道每一个数组的元素个数。因此, 函数设计成有三个参数, 定义如下:

```
void saleByQuarter(salesPersonRec list[], int listSize,
 double totalByQuarter[])
{
 quarterType quarter;
 int count;

 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter+1))
 totalByQuarter[quarter] = 0.0;

 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter+1))
 for(count = 0; count < listSize; count++)
 totalByQuarter[quarter] +=
 list[count].saleByQuarter[quarter];
}
```



函数 `totalSaleByPerson()`: 函数计算每一位销售人员的年销售额。为此, 需要将每一位销售人员 4 个季度的销售额相加, 显然, 该函数必须访问数组 `salesPersonList`, 它同样需要知道数组的记录数。因此, 函数设计为有两个参数, 定义如下:

```
void totalSaleByPerson(salesPersonRec list[], int listSize)
{
 int count;
 quarterType quarter;
 for(count = 0; count < listSize; count++) //for each salesperson
 for(quarter = QT1; quarter <= QT4; //for each quarter
 quarter = static_cast<quarterType>(quarter + 1))
 list[count].totalSale +=
 list[count].saleByQuarter[quarter];
}
```

函数 `printReport()`: 函数以特定格式打印年销售报表。算法的伪代码如下:

- a. 打印标题, 即输出的前三行。
- b. 打印每位销售人员的数据。
- c. 打印表的最后一行。

注意, 下面两个函数将会产生输出的最后两行。

显然, 函数 `printReport()` 必须访问数组 `salesPersonList` 和数组 `totalSaleByQuarter`。同样, 因为输出将存入文件, 因此, 该函数设计为有 4 个参数: 对应数组 `salesPersonList`、对应数组 `totalSaleByQuarter`、数组的大小、`ofstream` 变量。定义如下:

```
void printReport(ofstream& outfile, salesPersonRec list[],
 int listSize, double saleByQuarter[])
{
 int count;
 quarterType quarter;
 //Step a
 outfile<<"----- Annual Sales Report -----"<<endl;
 outfile<<endl;
 outfile<<" ID QT1 QT2 QT3 "
 <<"QT4 Total"<<endl;
 outfile<<"
 <<"-----"
 for(count = 0; count < listSize; count++) //Step b
 {
 outfile<<list[count].ID<<" ";
 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter + 1))
 outfile<<setw(10)<<list[count].saleByQuarter[quarter];
 outfile<<setw(10)<<list[count].totalSale<<endl;
 }

 outfile<<"Total "; //Step c
 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter + 1))
 outfile<<setw(10)<<saleByQuarter[quarter];

 outfile<<endl<<endl<<endl;
}
```

函数 `maxSaleByPerson()`: 函数打印最大销售额的销售人员的名字。要找到该销售人员, 需要查找每一位销售人员的销售总额并找出最大者。因为每位销售人员的销售总额保存在数组 `salesPersonList` 中, 函数必须访问该数组。同样, 因为输出存储在一个文件中, 该函数必须访问与输出文件相关的 `ofstream` 变量。因此, 函数设计为有三个参数: 数组 `salesPersonList`、数组的大小以及输出文件变量。

查找最大销售额的算法与查找数组中最大元素的算法(第10章讨论过)相似。函数定义如下:

```
void maxSaleByPerson(ofstream& outData, salesPersonRec list[],
 int listSize)
{
 int maxIndex = 0;
 int count;

 for(count = 1; count < listSize; count++)
 if(list[maxIndex].totalSale < list[count].totalSale)
 maxIndex = count;

 outData<<"Max Sale by SalesPerson:ID = "<<list[maxIndex].ID
 <<"", Amount = $"<<list[maxIndex].totalSale<<endl;
}
```

函数 `maxSaleByQuarter()`: 函数打印最大销售额的所在季度。因此, 需要查找每一个季度的销售总额并找到最大者。由于每季度的销售总额存储在数组 `totalSaleByQuarter` 中, 函数必须要访问它。同样, 输出要存储在一个文件中, 函数必须访问与输出文件相关的变量 `ofstream`。因此函数有两个参数: 数组 `totalSaleByQuarter` 和对应输出文件变量。

查找最大销售额的算法与查找数组中最大元素的算法(第10章讨论过)相似。函数定义如下:

```
void maxSaleByQuarter(ofstream& outData, double saleByQuarter[])
{
 quarterType quarter;
 quarterType maxIndex = QT1;

 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter+1))
 if(saleByQuarter[maxIndex] < saleByQuarter[quarter])
 maxIndex = quarter;

 outData<<"Max Sale by Quarter:Quarter = "
 << static_cast<int>(maxIndex) + 1
 <<"", Amount = $"<<saleByQuarter[maxIndex]<<endl;
}
```

要使程序具有更好的灵活性, 需要在执行过程中提示用户指定输入和输出文件。现在, 让我们编写函数 `main()` 的算法。

#### 主要算法

1. 声明变量。
2. 提示用户输入包含销售人员 ID 的文件名。
3. 读入输入文件名。
4. 打开输入文件。
5. 如果输入文件不存在, 程序退出。
6. 初始化数组 `salesPersonList`。调用函数 `initialize()`。
7. 关闭输入文件。



8. 提示用户输入包含销售数据的文件名。
9. 读入输入文件名。
10. 打开文件。
11. 如果输入文件不存在, 程序退出。
12. 提示用户输入输出文件名。
13. 读入输出文件名。
14. 打开输出文件。
15. 要输出格式为显示小数点尾部补 0 的浮点数, 设置控制符 `fixed` 和 `showpoint`。同样要输出两个小数位的浮点数, 将精度设置为两位。
16. 处理销售数据。调用函数 `getData()`。
17. 计算季度销售总额。调用函数 `saleByQuarter()`。
18. 计算每一位销售人员的销售总额。调用函数 `totalSaleByPerson()`。
19. 打印表格式报表。调用函数 `printReport()`。
20. 找出并打印最大年销售额的销售人员。调用函数 `maxSaleByPerson()`。
21. 找出并打印最大销售额季度。调用函数 `maxSaleByQuarter()`。
22. 关闭文件。

#### 完整程序清单

```
//Program:Sales data analysis
#include<iostream>
#include<fstream>
#include<iomanip>
#include<string>

using namespace std;

Const int noOfSalesPerson = 6;

enum quarterType{ QT1,QT2,QT3,QT4 };

struct salesPersonRec
{
 string ID; //salesperson's ID
 double saleByQuarter[4];
 double totalSale;
};

void initialize(ifstream& indata, salesPersonRec list[], int listSize);
void getData(ifstream& infile, salesPersonRec list[], int listSize);
void saleByQuarter(salesPersonRec list[], int listSize,
 double totalByQuarter[]);
void totalSaleByPerson(salesPersonRec list[], int listSize);
void maxSaleByPerson(ofstream& outData, salesPersonRec list[],
 int listSize);
void maxSaleByQuarter(ofstream& outData, double saleByQuarter[]);
void printReport(ofstream& outfile, salesPersonRec list[],
 int listSize, double saleByQuarter[]);

int main()
{
 //Step 1
 ifstream infile; //input file stream variable
 ofstream outfile; //output file stream variable
 char inputfile[25]; //variable to hold the input file name
```

```

char outputfile[25]; //variable to hold the output file name
double totalSaleByQuarter[4]; //array to hold the
 //sales by quarter

salesPersonRec salesPersonList[noOfSalesPerson]; //array to
 //hold the salesperson's data
cout<<"Enter SalesPerson ID file name : "; //Step2
cin>>inputfile; //Step3
cout<<endl;

infile.open(inputfile); //Step4
if(!infile) //Step5
{
 cout<<"Cannot open input file."<<endl;
 return 1;
}

initialize(infile, salesPersonList, noOfSalesPerson); //Step 6
infile.close(); //reclaim input file stream variable; Step7
cout<<"Enter sales data file name: "; //Step 8
cin>>inputfile; //Step 9
cout<<endl;

infile.open(inputfile); //Step 10
if(!infile) //Step 11
{
 cout<<"Cannot open input file."<<endl;
 return 1;
}

count<<"Enter output file name: "; //Step 12
cin>>outputfile; //Step 13
cout<<endl;

outfile.Open(outputfile); //Step 14
outfile<<fixed<<showpoint<<setprecision(2); //Step 15

getData(infile, salesPersonList, noOfSalesPerson); //Step 16
saleByQuarter(salesPersonList, noOfSalesPerson,
 totalSaleByQuarter); //Step 17
totalSaleByperson(salesPersonList, noOfSalesPerson); //Step 18
printReport(outfile, salesPersonList, noOfSalesPerson,
 totalSaleByQuarter); //Step 19
maxSaleByperson(outfile, salesPersonList,
 noOfSalesPerson); //Step 20
maxSaleByQuarter(outfile, totalSaleByQuarter); //Step 21
infile.close(); //Step 22
outfile.close(); //Step 22
return 0;
}

void initialize(ifstream& indata, salesPersonRec list[], int listSize)
{
 int count;
 quarterType quarter;
 for(count = 0; count < listSize; count++)
 {

```

```

 indata>>list[count].ID; //get salesperson's ID
 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter + 1))
 list[count].saleByQuarter[quarter] = 0.0;
 list[count].totalSale = 0.0;
 }
}

void getData(ifstream& infile, salesPersonRec list[], int listSize)
{
 int count;
 quarterType quarter;
 string sID;
 int month;
 double amount;

 infile>>sID; //get salesperson's ID
 while(infile)
 {
 infile>>month>>amount; //get the sales month and sales amount
 for(count = 0; count < listSize; count++)
 {
 if(sID == list[count].ID)
 {
 if(1 <= month && month <= 3)
 quarter = QT1;
 else if(4 <= month && month <= 6)
 quarter = QT2;
 else if(7 <= month && month <= 9)
 quarter = QT3;
 else
 quarter=QT4
 list[count].saleByQuarter[quarter] += amount;
 break;
 } //end if
 } //end for
 infile>>sID;
 } //end while
} //end getData

void saleByQuarter(salesPersonRec list[], int listSize,
 double totalByQuarter[])
{
 quarterType quarter;
 int count;

 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter+1))
 totalByQuarter[quarter] = 0.0;

 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter+1))
 for(count = 0; count < listSize; count++)
 totalByQuarter[quarter] +=
 list[count].saleByQuarter[quarter];
}

void totalSaleByPerson(salesPersonRec list[], int listSize)
{

```

```

 int count;
 quarterType quarter;

 for(count = 0; count < listSize; count++)
 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter + 1))
 list[count].totalSale +=
 list[count].saleByQuarter[quarter];
}

void maxSaleByPerson(ofstream& outData, salesPersonRec list[],
 int listSize)
{
 int maxIndex = 0;
 int count;
 for(count = 1; count < listSize; count++)
 if(list[maxIndex].totalSale < list[count].totalSale)
 maxIndex = count;

 outData<<"Max Sale by SalesPerson: ID = "<<list[maxIndex].ID
 <<" , Amount = $"<<list[maxIndex].totalSale<<endl;
}

void maxSaleByQuarter(ofstream& outData, double saleByQuarter[])
{
 quarterType quarter;
 quarterType maxIndex = QT1;
 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter+1))
 if(saleByQuarter[maxIndex] < saleByQuarter[quarter])
 maxIndex = quarter;

 outData<<"Max Sale by Quarter: quarter = "
 <<static_cast<int>(maxIndex) + 1
 <<" , Amount = $"<<saleByQuarter[maxIndex]<<endl;
}

void printReport(ofstream& outfile, salesPersonRec list[],
 int listSize, double saleByQuarter[])
{
 int count;
 quarterType quarter;

 outfile<<"----- Annual Sales Report -----"<<endl;
 outfile<<endl;
 outfile<<" ID QT1 QT2 QT3 "
 <<"QT4 Total"<<endl;
 outfile<<" _____"
 <<" _____"<<endl;

 for(count = 0; count < listSize; count++)
 {
 outfile<<list[count].ID<<" ";
 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter + 1))
 outfile<<setw(10)<<list[count].saleByQuarter[quarter];
 outfile<<setw(10)<<list[count].totalSale<<endl;
 }

 outfile<<"Total ";
}

```

```

 for(quarter = QT1; quarter <= QT4;
 quarter = static_cast<quarterType>(quarter + 1))
 outfile<<setw(10)<<saleByQuarter[quarter];

 outfile<<endl<<endl<<endl;
}

```

运行实例:

输入文件: 销售人员 ID

```

12345
32214
23422
57373
35864
54654

```

输入文件: 销售人员数据

```

12345 1 893
32214 1 343
23422 3 903
57373 2 893
35864 5 329
54654 9 392
12345 2 999
32214 4 892
23422 4 895
23422 2 492
57373 6 892
35864 10 1223
54654 11 3420
12345 12 322
35864 5 892
54654 3 893
12345 8 494
32214 8 9023
23422 6 223
23422 4 783
57373 8 8834
35864 3 2882

```

输出文件:

| ----- Annual Sales Report ----- |         |         |          |         |          |
|---------------------------------|---------|---------|----------|---------|----------|
| ID                              | QT1     | QT2     | QT3      | QT4     | Total    |
| 12345                           | 1892.00 | 0.00    | 494.00   | 322.00  | 2708.00  |
| 32214                           | 343.00  | 892.00  | 9023.00  | 0.00    | 10258.00 |
| 23422                           | 1395.00 | 1901.00 | 0.00     | 0.00    | 3296.00  |
| 57373                           | 893.00  | 892.00  | 8834.00  | 0.00    | 10619.00 |
| 35864                           | 2882.00 | 1221.00 | 0.00     | 1223.00 | 5326.00  |
| 54654                           | 893.00  | 0.00    | 392.00   | 3420.00 | 4705.00  |
| Total                           | 8298.00 | 4906.00 | 18743.00 | 4965.00 |          |

Max Sale by SalesPerson: ID = 57373, Amount = \$10619.00  
 Max Sale by Quarter: Quarter = 3, Amount = \$18743.00

## 12.3 小结

1. 结构体是一种混合类型数据的集合。

2. 结构体中各部分可以是不同类型。

3. 定义一个结构体的语法为:

```
struct structName
{
 dataType identifier;
 dataType identifier;
 .
 .
};
```

4. 在 C++ 中, struct 是一个保留字。

5. 在 C++ 中, 结构体是一种定义, 并不进行内存分配。只有声明变量时才会分配内存。

6. 结构体的组成部分称为结构体的成员。

7. 结构体的成员通过名字来访问。

8. 结构体成员通过圆点运算符 (.) 来访问。如果 employeeType 是一个结构体, employee 是 employeeType 类型的变量, name 是 employee 的一个成员, 那么, employee.name 是一个变量并且可以像其他变量一样进行操作。

9. 在 C++ 中, 圆点运算符 (.) 叫做成员访问运算符。

10. 结构体唯一的内建操作是赋值和成员访问操作。

11. 结构体间不允许进行算术和关系运算。

12. 作为函数的参数, 结构体既可以用传值方式也可以用引用的方式进行传递。

13. 函数可以返回一个结构体类型的返回值。

14. 一个结构体可以作为另一个结构体的成员。

## 12.4 练习

1. 判断对错:

a. 结构体是同类型的数据结构集。

b. 函数不能返回结构体类型的返回值。

c. 结构体的成员可以是其他结构体。

d. 惟一允许在结构体上的运算是赋值和成员选择运算。

e. 数组可以作为结构体的成员。

f. 在 C++ 中, 在结构体上允许进行一些聚集运算。

g. 因为结构体包含有限数量的成员, 在结构体上使用关系运算符是允许的。

2. 考虑下列语句:

```
struct nameType
{
 string first;
 string last;
};
struct dateType
{
 int month;
```



```

 int day;
 int year;
};
struct personalInfoType
{
 nameType name;
 int pID;
 dataType dob;
};
personalInfoType person;
personalInfoType classList[100];
nameType student;

```

判断下列语句是否合法，如果非法，说明原因。

- a. person.name.first = "william";
- b. cout<<person.name;
- c. classList[1] = person;
- d. classList[20].pID = 000011100;
- e. person = classList[20];
- f. student = person.name;
- g. cin>>student;
- h. for(int j=0; j<100; j++)
  - classList[j].pID = 00000000;
- i. classList.dob.date = 1;
- j. student = name;

3. 考虑以下语句 (nameType 如练习 2 中所定义):

```

struct employeeType
{
 nameType name;
 int performanceRating;
 int pID;
 string dept;
 double salary;
};
employeeType employees[100];
employeeType newEmployee;

```

判断下列语句是否合法，如果非法，说明原因。

- a. newEmployee.name = "John Smith";
- b. cout<<newEmployee.name;
- c. employees[35] = newEmployee;
- d. if (employees[45].pID == 555334444)
  - employees[45].performanceRating = 1;
- e. employees.salary = 0;

4. 对于练习 2 和 3 中的声明，编写 C++ 语句完成下列功能：

- a. 将下列信息存储在 newEmployee 中：

```

name: Mickey Doe
pID: 111111111
jobRating: 2
dept: ACCT
salary: 34567.78

```

- b. 在数组 employees 中，初始化每一个 performanceRating 成员为 0。
- c. 将数组 employees 的第 20 个元素复制给 newEmployee。
- d. 将 5735.87 与数组 employees 中第 15 个元素的 salary 成员相加。

## 12.5 编程练习

1. 编写程序读入学生的名字和考试分数, 输出每一位学生的名字、考试分数和相应的等级。同时要求查找并打印最高分数以及相应学生的名字。

学生数据存放在 `studentType` 类型的结构体变量中, 该类型含有三个成员: `string` 型的 `studentName`, `int` 型的 `testScore` (`testScore` 介于 0 和 100 之间) 和 `char` 型的 `grade`。假设班级有 20 位学生, 可以使用一个含有 20 个 `studentType` 类型元素的数组。

程序必须至少包含下列函数:

- a. 将学生数据读入数组的函数。
- b. 给每一位学生评分数的函数。
- c. 找到最高分的函数。
- d. 打印最高分值和相应学生名字的函数。

程序必须以下列形式输出每一位学生的名字: `last name` 后接一个逗号, 一个空格, `first name`。同时名字必须左对齐。此外, `main()` 函数除了声明变量, 打开输入、输出文件之外仅仅作为函数调用的集合。

2. 定义一个结构体 `menuItem` 并含有两个成员: `string` 类型的 `menuItem` 成员和 `double` 类型的 `menuPrice` 成员。
3. 编写一个程序帮助一家当地餐馆建立其早餐记账系统。程序应有以下功能:
  - a. 显示餐馆提供的各种早餐项。
  - b. 允许顾客从菜单中选择多种早餐项。
  - c. 计算并打印账单。

假设餐馆提供下列项目 (每一项的价格显示在其右边):

|               |        |
|---------------|--------|
| Plain Egg     | \$1.45 |
| Bacon and Egg | \$2.45 |
| Muffin        | \$0.99 |
| French Toast  | \$1.99 |
| Fruit Basket  | \$2.49 |
| Cereal        | \$0.69 |
| Coffee        | \$0.50 |
| Tea           | \$0.75 |

使用在编程练习 2 中定义的 `menuItem` 类型的数组 `menuList`, 程序中必须包含下列函数:

- 函数 `getData()`: 该函数将数据读入数组 `menuList`。
- 函数 `showMenu()`: 该函数显示餐馆提供的各种项目并提示顾客如何选取项目。
- 函数 `printCheck()`: 该函数计算并打印账单 (注意, 账单中包含 5% 的税)。

输出的一个例子如下:

```
Welcome to Johnny's Restaurant
Bacon and Egg $2.45
Muffin $0.99
Coffee $0.50
Tax $0.20
Amount Due $4.14
```

程序输出的格式需要保留小数点后两位, 同时菜单行的名字必须左对齐, 并假设顾客对每一项只选择一份。

4. 重写编程练习 3 以便顾客可以对每一菜单项选择多份。程序输出的例子如下:

```
Welcome to Johnny's Restaurant
```



|   |               |        |
|---|---------------|--------|
| 1 | Bacon and Egg | \$2.45 |
| 2 | Muffins       | \$1.98 |
| 1 | Coffee        | \$0.50 |
|   | Tax           | \$0.25 |
|   | Amount Due    | \$5.18 |

5. 编写一个程序，其主函数 `main()` 只负责声明变量和函数调用。该程序读取一段英文文本，输出每一个字母以及它们的出现次数，如下面的函数 `printResult()` 所示（不能够有全局变量，所有信息必须在函数间传递，并使用一个结构体来存储这些信息）。程序必须至少包含下列函数：

- 函数 `openFile()`：打开输入和输出文件。将文件流作为参数（通过引用）。如果文件不存在，程序打印适当信息并退出。程序必须提示用户指定输入和输出文件名。
- 函数 `count()`：统计函数 `openFile()` 打开的英文文本文件中大写字母 A~Z 和小写字母 a~z 的出现次数，并将这些信息存储在结构体数组中，因此，数组及相应文件流变量需要作为参数进行传递。
- 函数 `printResult()`：打印大写、小写字母出现的次数，以及大写字母在所有出现的 A~Z 中所占的百分比和小写字母在所有出现的 a~z 中所占的百分比，百分比的形式必须是：“25%”。这些信息必须来自一个结构体数组参数。

## 第 13 章 类和数据抽象

### 本章要点:

- 什么是类
- 类的私有、保护和公有成员
- 类是如何实现的
- 构造函数和析构函数
- 抽象数据类型 (ADT)
- 如何运用类来实现 ADT
- 信息隐藏
- 在 C++ 中如何实现信息隐藏

在第 12 章中, 我们学习了如何使用结构体将不同类型的数据组织在一起。第 12 章中给出的结构体定义与 C 语言中结构体的定义相似。然而, C++ 中的结构体成员既可以是数据, 也可以是函数。C++ 提供了另外一种结构化的数据类型称为类, 它是专门设计用来组织数据和函数的。本章首先介绍类并解释如何使用它们, 然后讨论结构体和类的相似与不同之处。

注意: 第 12 章并不是本章的预备知识。事实上, 结构体和类有着相似的性质, 正如接下来将在“结构体与类”一节中讨论的那样。

### 13.1 类

在第 1 章中介绍了称为面向对象设计 (OOD) 的方法学。在 OOD 中, 第一步是找出称为对象的组成部分, 对象将数据和作用在该数据上的操作结合在一个单元内。在 C++ 中, 允许将数据和作用在该数据上的操作结合于一个单元内的机制称为类。既然读者已经知道如何在计算机内存中存储和操纵数据并掌握了如何构建自己的函数, 我们就为学习如何建立对象做好了准备。本章和后续章节中将会使用 OOD 来设计并实现程序, 本章首先解释如何定义一个类以及如何在程序中使用它。

类的组成部分称为类的成员。定义类的一般语法为:

```
class classIdentifier
{
 classMembersList
};
```

其中 classMembersList 由变量声明和 (或) 函数组成。即, 类的成员既可以是变量 (用来储存数据) 又可以是函数。

- 如果类的成员是变量, 可以像声明其他变量一样声明它。
- 如果类的成员是函数, 一般的做法是使用函数原型来定义它。
- 如果类的成员是函数, 它可以访问类中的任何成员——数据和函数。即, 当定义类的成员函数时, 定义的函数可以直接访问该类中的任何成员而无需将其声明为参数, 惟一的限制条件是在使用一个标识符之前必须先声明它。

在 C++ 中, `class` 是一个保留字, 它只定义了一种数据类型, 并没有内存被分配。请注意右花括号之后的分号 (;), 它也是语法的一部分, 因此, 漏写该分号会导致一个语法错误。

类的成员归为三类: 私有 (`private`)、公有 (`public`) 和保护 (`protected`)。本章主要讨论前两种, 即私有和公有。

下面是关于类的公有和私有的性质:

- 默认情况下, 一个类中所有的成员都是私有的。
- 如果一个类中的某个成员是私有的, 则无法在该类的外部访问它(例 13.1 解释了这一概念)。
- 公有的成员可以被该类以外的部分访问(例 13.1 解释了这一概念)。
- 使用标志 `public` 和冒号将成员声明为公有的。

在 C++ 中, `private`、`protected` 和 `public` 是保留字, 它们称为成员访问限定符。

假设想要在程序中定义一个类来实现一天中的时间。由于一天中的时间是由时钟来记时的, 我们称这个类为 `clockType`。另外用三个 `int` 型变量来表示时间: 分别代表小时、分钟和秒, 同时还需要对时间进行如下操作:

1. 设置时间。
2. 得到当前时间。
3. 打印时间。
4. 将时间增加一秒。
5. 将时间增加一分钟。
6. 将时间增加一小时。
7. 比较两个时间是否相等。

从上面讨论中, 显然类 `clockType` 有 10 个成员: 3 个数据成员和 7 个函数成员 (如图 13.1)。

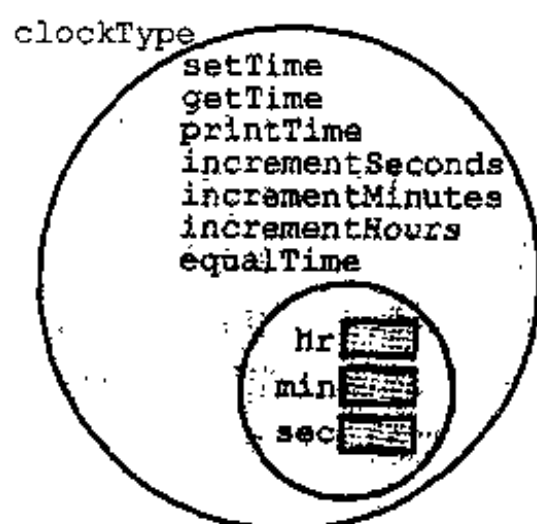


图 13.1 类 `clockType` 及其成员

类 `clockType` 的一些成员将会是私有的; 其他的将是公有的。成员本身的性质将会决定哪些是公有的, 哪些是私有的。一般的规则是, 需要从类的外部访问的成员应该定义成公有的, 任何不应该从外部来访问的成员应该定义成私有的。例如, 用户应该可以设置和打印时间, 因此, 相应的设置时间和打印时间的函数应该声明为公有的。

类似地, 增加时间和比较两个时间是否相等的函数应为公有的。另一方面, 那些直接操作数据成员 `hr`、`min` 和 `sec` 的函数要声明为私有的。另外, 如果用户可以直接访问数据成员, 那么诸如 `setTime()` 之类的函数就不需要了。本章的第二部分 (从“信息隐藏”一节开始) 将会讲述为什么一些成员是公有的, 而另一些则是私有的。

下面语句定义了类 clockType:

```
class clockType
{
public:
 void setTime(int, int, int);
 void getTime(int&, int&, int&);
 void printTime() const;
 void incrementSeconds();
 void incrementMinutes();
 void incrementHours();
 bool equalTime(const clockType& otherClock) const;
private:
 int hr;
 int min;
 int sec;
};
```

在该定义中:

- 类 clockType 有 7 个成员函数: setTime()、getTime()、printTime()、incrementSeconds()、incrementMinutes()、incrementHours() 和 equalTime(); 有 3 个数据成员: hr、min 和 sec。
- 3 个数据成员 hr、min 和 sec 是私有的, 不能够从类的外部进行访问 (例 13.1 说明了这一概念)。
- 7 个成员函数 setTime()、getTime()、printTime()、incrementSeconds()、incrementMinutes()、incrementHours() 和 equalTime() 可以直接访问数据成员 (hr、min 和 sec)。另一方面, 并没有将数据成员作为上述函数的参数。
- 在函数 equalTime() 中, 参数 otherClock 是以常量进行引用的。即, 在调用函数 equalTime() 时, 参数 otherClock 接收到实参的地址, 但 otherClock 并不能够更改实参的值。如果将 otherClock 定义为一个传值参数, 则 otherClock 会存储一个实参值的副本, 这将会导致性能下降 (参照稍后的“引用参数和类对象 (变量)”一节)。
- 在成员函数 printTime() 和 equalTime() 中的 const 表明, 这些函数不能够更改 clockType 型变量的数据成员值。

**注意:** 私有和公有成员可以以任何顺序出现, 可以在最开始声明私有成员然后再声明公有成员。稍后的“公有和私有成员的顺序”一节中将讨论这一问题。

**注意:** 在类 clockType 的定义中, 所有数据成员都是私有的, 而且所有的成员函数都是公有的。然而, 成员函数也可以是私有的。例如, 如果一个成员函数只为实现其他函数而服务, 就可以将其定义为私有的。类似地, 数据成员也可以是公有的。

函数 setTime() 设定三个数据成员——hr、min 和 sec——作为通过参数传递进来的给定值。函数 printTime() 打印时间, 即 hr、min 和 sec 的值。函数 incrementSeconds() 将时间增加 1 秒, 函数 incrementHours() 将时间增加 1 小时, 而函数 equalTime() 比较两个时间值的相等性。

注意, 函数 equalTime() 只有一个参数, 尽管需要将两个时间来进行比较。在“成员函数的实现”一节中, 这一点将会得到解释。

#### 变量 (对象) 的声明

一旦定义了一个类, 就可以用它作为数据类型来声明变量。在 C++ 术语中, 类变量称为类对

象或者类实例。为使读者尽快适应这一术语,从现在开始,将以类对象或简单地以对象来代表一个类变量。声明类对象的语法和声明其他类型变量类似,下面的语句声明了类 `clockType` 的两个对象:

```
clockType myClock;
clockType yourClock;
```

每一个对象都有 10 个成员: 7 个成员函数和 3 个数据成员。每一个对象都有为 `hr`、`min` 和 `sec` 单独分配的内存空间。

实际上,系统只为每一个类对象的数据成员分配内存。C++ 编译器只为一个类的函数成员生成一个副本,每个类对象执行相同的函数成员副本。因此,在描绘类图时,通常将显示类中所有的成员。然而,在描绘类的对象图时,将只显示其数据成员。图 13.2 显示了对象 `myClock` 和对象 `yourClock` 中相应的数据成员值。

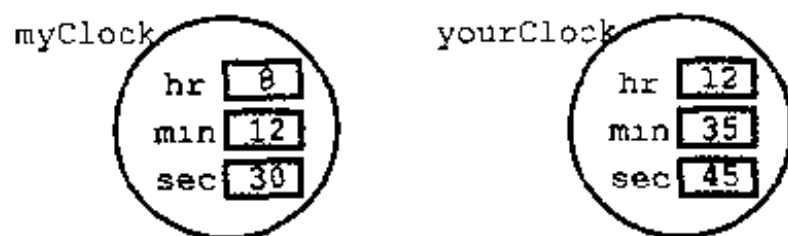


图 13.2 对象 `myClock` 和 `yourClock`

### 访问类成员

一旦声明了对象,就可以访问它的公有成员。访问一个类成员的一般语法为:

```
classObjectName.memberName
```

回忆一下,在 C++ 中,圆点 (.) 叫做成员访问运算符。

例 13.1 说明了如何访问一个类的成员。参看下列语句:

#### 例 13.1

```
myClock.setTime(5,2,30);
myClock.printTime();
yourClock.setTime(x,y,z); //Assume x, y, and z are
 //variables of the type int
if(myClock.equalTime(yourClock))
{
 .
 .
 .
}
```

这些语句是合法的,即它们的语法是正确的。

在第一条句中,成员函数 `setTime()` 被执行。值 5、2 和 30 作为参数传递给函数 `setTime()`, 函数使用这些值将 `myClock` 的三个数据成员 `hr`、`min` 和 `sec` 分别设置为 5、2 和 30。类似地,第二条语句执行成员函数 `printTime()` 并且输出 `myClock` 三个数据成员的内容。第三条语句中,变量 `x`、`y` 和 `z` 用来设置 `yourClock` 的三个数据成员。

在第四条语句中,成员函数 `equalTime()` 被执行,并相应地比较 `myClock` 和 `yourClock` 的三个数据成员,因为在这条语句中,`equalTime()` 是对象 `myClock` 的一个成员,所以它还需要一个对象(在本例中,是 `yourClock`)来进行比较。这解释了为什么函数 `equalTime()` 只有一个参数。

一个类对象只可以访问公有型成员。因此,下面的语句是非法的,因为 `hr` 和 `min` 是类 `clockType` 的私有成员,并且不能够被对象 `myClock` 和 `yourClock` 所访问。

```
myClock.hr = 10; //illegal
myClock.min = yourClock.min; //illegal
```

### 13.1.1 类的内建运算符

大多数 C++ 内建运算符都不适合于类。不能够使用算术运算符进行类变量的算术运算（除非它们被重载，参见第 16 章）。例如，不能使用运算符 + 来将两个类变量（如 clockType 型变量）相加。同样，不能够使用关系运算符比较两个类变量是否相等（除非它们被重载，参见第 16 章）。

对类对象可用的两个内建运算符是成员访问（.）和赋值（=）运算符。读者已经看到了如何使用对象的名字加上一个圆点，再加上成员名字来访问类的某一成员。

接下来将用一个例子来展示赋值语句是如何工作的。

#### 赋值运算和类

假设 myClock 和 yourClock 是前面定义过的 clockType 类型变量，myClock 和 yourClock 的值如图 11.3 所示。



图 11.3 对象 myClock 和 yourClock

语句：

```
myClock = yourClock; //Line A
```

将 yourClock 的值复制给 myClock。即：

1. yourClock.hr 的值复制给 myClock.hr。
2. yourClock.min 的值复制给 myClock.min。
3. yourClock.sec 的值复制给 myClock.sec。

换句话说，yourClock 的三个成员值相应地复制给了 myClock 的三个数据成员。因此，一条赋值语句执行了以成员为单位的复制。在 Line A 执行之后，myClock 和 yourClock 的值如图 13.4 所示。

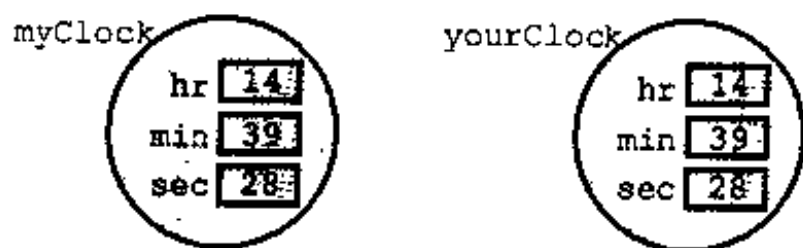


图 13.4 赋值语句 “myClock = yourClock;” 执行之后的对象 myClock 和 yourClock

### 13.1.2 类的作用域

类对象即可以是自动的（即，每当控制到达其声明时生成，并在控制退出其所在语法块时销毁），也可以是静态的（即，当控制到达其声明处时创建一次，并当程序结束时销毁）。同样，可以声明一个类对象的数组。类对象与其他变量有着相同的作用域。类成员的作用域与结构体成员的作用域相同。即，对于类而言，类成员是类的本地变量。可以使用类对象的名字和访问运算符（.）从类的外部访问类成员。



### 13.1.3 函数和类

下面的规则描述了函数和类之间的关系：

- 类对象可以作为函数的参数，也可以作为函数的返回值。
- 类作为函数参数，既可以用传值方式也可以用引用的方式。
- 如果类对象以传值方式进行参数传递，则实际对象的各数据成员值相应地复制到形参的数据成员。

#### 引用参数和类对象（变量）

回忆一下，当变量以传值方式进行参数传递时，形参复制了实参值。即，系统为形参分配内存并将其复制为实参的值。类对象也可以以传值的方式进行参数传递。

假设一个类有几个数据成员，需要大量的内存存储数据，而我们需要用传值的方式来传递参数，则形参接收对应变量的副本，即编译器必须为形参分配内存以便复制实参数据成员的值。这种操作需要额外的大量存储空间和相当可观的计算时间用来将实参的值复制给形参。

另一方面，如果变量以引用的方式传递，形参只是接收到实参的地址，因此，引用方式是一种更为有效的参数传递方法。如果参数以引用方式传递，那么当形参发生变化时，实参的值也会发生变化。然而有时候，并不希望函数改变数据成员的值，在 C++ 中，可以通过在形参声明中使用关键字 `const` 的方法阻止函数对参数的值进行修改。作为例子，考虑下列函数定义：

```
void testTime(const clockType& otherClock)
{
 clockType dClock;
 ...
}
```

函数 `testTime()` 包含一个引用参数 `otherClock`。参数 `otherClock` 用关键字 `const` 进行声明。因此，在对函数 `testTime()` 的调用过程中，形参 `otherClock` 接收实参的地址，但是 `otherClock` 不能够改变实参的内容。例如，在执行下面语句后，`myClock` 的值将不会被改变：

```
testTime(myClock);
```

通常，如果希望以传值的方式传递类对象，最好将其用前面描述过的 `const` 关键字声明为引用型传递。

回忆一下，如果形参是传值参数，则在函数体定义内可以改变形参的值，就是说，可以通过赋值语句改变形参的值（当然，并不会影响实参）。然而，如果形参声明为 `const` 的引用参数，就不能在函数体的定义中通过赋值语句改变它的值。因此，在函数 `testTime()` 的定义中，不能改变 `otherClock` 的值。例如，下列语句如果出现在 `testTime()` 的定义中将是非法的：

```
otherClock.setTime(5, 34, 56); //illegal
otherClock = dClock; //illegal
```

### 13.1.4 函数成员的实现

当定义类 `clockType` 时，对于函数成员只包括函数原型。为使它们能够正常工作，必须编写相应的算法。一种实现方式是直接在类定义中给出函数的定义而不是函数原型。但是，类的定义会因此面变得十分冗长而难于理解。提供函数原型而不是函数实现的另外原因是与信息隐藏有关，即希望隐藏数据操作的细节。我们会在本章稍后的部分进行讨论（参照“信息隐藏”一节）。

下面，让我们来编写类 `clockType` 的成员函数。即将要编写 `setTime()`、`getTime()`、`printTime()`、

incrementSeconds()、equalTime()等函数的定义。因为标识符 setTime、printTime 等是类的局部变量,所以不能够在类之外对其(直接)进行引用。为了引用这些标识符,可以使用范围解析运算符(::)。在函数定义的头部,函数名是类名加上范围解析运算符以及函数名组成的。例如,函数 setTime() 的定义如下:

```
void clockType::setTime(int hours, int minutes, int seconds)
{
 if(0 <= hours && hours < 24)
 hr = hours;
 else
 hr = 0;

 if(0 <= minutes && minutes < 60)
 min = minutes;
 else
 min = 0;

 if(0 <= seconds && seconds < 60)
 sec = seconds;
 else
 sec = 0;
}
```

注意,函数 setTime() 的定义中对小时、分钟和秒的有效性进行了检查。如果所给值超出了范围,则 hr、min 和 sec 初始化为 0。现在解释一下当被类 clockType 的对象访问时,成员函数 setTime() 是如何工作的。

成员函数 setTime() 是一个 void 函数并且有三个参数。因此,

- 对该函数的调用是独立的一条语句。
- 必须在调用时使用三个参数。

另外,由于 setTime() 是类 clockType 的一个成员,由定义可以看出,它可以直接访问数据成员 hr、min 和 sec。

假设 myClock 是 clockType 类型的对象(正如前面声明的),如图 13.5 所示,myClock 对象 myClock 有三个数据成员。

考虑下面语句:

```
myClock.setTime(3, 48, 52);
```

对象 myClock 访问成员 setTime()。在语句“myClock.setTime(3, 48, 52);”中, setTime() 被对象 myClock 访问。因此,函数 setTime() 内部所引用的三个变量——hr、min 和 sec——是 myClock 的三个数据成员。因此,通过前面语句以参数传递的值 3、48 和 52 被函数 setTime() 分别赋给了 myClock 的三个数据成员(参见函数 setTime() 的定义)。在执行过上述语句后,对象 myClock 如图 13.6 所示。

下面给出类 clockType 其他函数成员的定义,它们都很简单且易于理解。

```
void clockType::getTime(int& hours, int& minutes, int& seconds)
{
 hours = hr;
 minutes = min;
 seconds = sec;
}
```



图13.5 对象 myClock

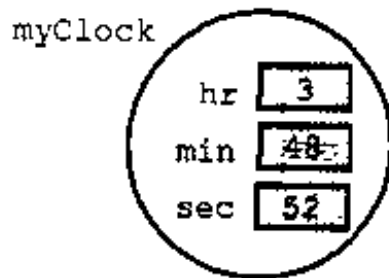


图13.6 在语句“myClock.setTime(3, 48, 52);”执行过后的对象 myClock



```
}

void clockType::printTime() const
{
 if(hr < 10)
 cout<<"0";
 cout<<hr<<":";
 if(min < 10)
 cout<<"0";
 cout<<min<<":";

 if(sec < 10)
 cout<<"0";
 cout<<sec;
}

void clockType::incrementHours()
{
 hr++;
 if(hr > 23)
 hr = 0;
}

void clockType::incrementMinutes()
{
 min++;
 if(min > 59)
 {
 min = 0;
 incrementHours(); //increment hours
 }
}

void clockType::incrementSeconds()
{
 sec++;
 if(sec > 59)
 {
 sec = 0;
 incrementMinutes(); //increment minutes
 }
}
```

从函数 `incrementMinutes()` 和 `incrementSeconds()` 的定义中可以看出, 一个成员函数可以调用其他成员函数。

函数 `equalTime()` 的定义如下:

```
bool clockType::equalTime(const clockType& otherClock) const
{
 return(hr == otherClock.hr
 && min == otherClock.min
 && sec == otherClock.sec);
}
```

假设 `myClock` 和 `yourClock` 都是如前面声明的 `clockType` 类型对象, 而且它们的内容如图 13.7 所示。



图 13.7 对象 myClock 和 yourClock

考虑下面语句：

```
if (myClock.equalTime(yourClock))
{
 .
 .
 .
}
```

在表达式：

```
myClock.equalTime(yourClock)
```

中，对象 myClock 访问成员函数 equalTime()，yourClock 的值传递给形参 otherClock，如图 13.8 所示。

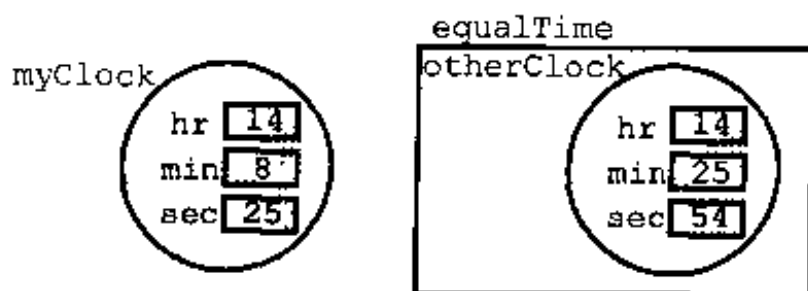


图 13.8 对象 myClock 和参数 otherClock

otherClock 的数据成员 hr、min 和 sec 的值分别是 14、25 和 54。换句话说，当 equalTime() 函数体执行后，otherClock.hr 的值是 14，otherClock.min 的值是 25，otherClock.sec 的值是 54。函数 euqalTime() 是 myClock 的成员函数，当函数 equalTime() 执行时，其内部的变量 hr、min 和 sec 是对象 myClock 的数据成员。因此，myClock 的成员 hr 与 otherClock.hr 相比较；myClock 的成员 min 与 otherClock.min 相比较；myClock 的成员 sec 与 otherClock.sec 相比较。

因此，从函数 equalTime() 的定义可以看出为什么它只有一个参数。

既然知道了如何定义一个类、如何访问一个类的成员以及如何实现成员函数，可以使用类来编写一个简单的程序。下面将编写一个简单的程序来使用类 clockType。

```
//Program that uses the class clockType
int main()
{
 clockType myClock;
 clockType yourClock;

 int hours;
 int minutes;
 int seconds;

 myClock.setTime(5,4,30); //Line 1
 cout<<"Line 2: myClock: "; //Line 2
 myClock.printTime(); //Line 3
 cout<<endl; //Line 4

 cout<<"Line 5: yourClock: "; //Line 5
 yourClock.printTime(); //Line 6
}
```

```

 cout<<endl; //Line 7
 yourClock.setTime(5,45,16); //Line 8
 cout<<"Line 9: After setting - yourClock: "; //Line 9
 yourClock.printTime(); //Line 10
 cout<<endl; //Line 11
 if(myClock.equalTime(yourClock)) //Line 12
 cout<<"Line 13: Both times are equal."
 <<endl; //Line 13
 else //Line 14
 cout<<"Line 15: The two times are not equal"
 <<endl; //Line 15

 cout<<"Line 16: Enter hours, minutes, and "
 <<"seconds: "; //Line 16
 cin>>hours>>minutes>>seconds; //Line 17
 cout<<endl; //Line 18
 myClock.setTime(hours,minutes,seconds); //Line 19
 cout<<"Line 20: New myClock: "; //Line 20
 myClock.printTime(); //Line 21
 cout<<endl; //Line 22
 myClock.incrementSeconds(); //Line 23
 cout<<"Line 24: After incrementing the clock by "
 <<"one second, myClock: "; //Line 24
 myClock.printTime(); //Line 25
 cout<<endl; //Line 26

 return 0;
 } //end main

```

函数要求用户输入三个数值：小时、分钟和秒。假设输入是 5 23 59，则程序执行的全程跟踪如下：

#### 输出：

```

Line 2: myClock: 05:04:30
Line 5: yourClock: (The value of yourClock is undefined here).
Line 9: After setting - yourClock: 05:45:16
Line 15: The two times are not equal
Line 16: Enter hours, minutes, and seconds: 5 23 59

Line 20: New myClock: 05:23:59
Line 24: After incrementing the time by one second, myClock: 05:24:00

```

下一个例子将类的定义、成员函数的定义和函数 main() 相结合以便创建一个完整的程序。

#### 例 13.2

```

//The complete program listing of the program that defines
//and uses class clockType

#include <iostream>
using namespace std;

class clockType
{
public:
 void setTime(int, int, int);
 void getTime(int&, int&, int&);
 void printTime() const;

```

```

 void incrementSeconds();
 void incrementMinutes();
 void incrementHours();
 bool equalTime(const clockType&) const;
private:
 int hr;
 int min;
 int sec;
};

int main()
{
 clockType myClock;
 clockType yourClock;

 int hours;
 int minutes;
 int seconds;

 myClock.setTime(5,4,30); //Line 1
 cout<<"Line 2: myClock: "; //Line 2
 myClock.printTime(); //Line 3
 cout<<endl; //Line 4

 cout<<"Line 5: yourClock: "; //Line 5
 yourClock.printTime(); //Line 6
 cout<<endl; //Line 7

 yourClock.setTime(5,45,16); //Line 8

 cout<<"Line 9: After setting - yourClock: "; //Line 9
 yourClock.printTime(); //Line 10
 cout<<endl; //Line 11

 if(myClock.equalTime(yourClock)) //Line 12
 cout<<"Line 13: Both times are equal."
 <<endl; //Line 13
 else //Line 14
 cout<<"Line 15: The two times are not equal"
 <<endl; //Line 15

 cout<<"Line 16: Enter hours, minutes, and "
 <<"seconds: "; //Line 16
 cin>>hours>>minutes>>seconds; //Line 17
 cout<<endl; //Line 18

 myClock.setTime(hours,minutes,seconds); //Line 19

 cout<<"Line 20: New myClock: "; //Line 20
 myClock.printTime(); //Line 21
 cout<<endl; //Line 22

 myClock.incrementSeconds(); //Line 23
 cout<<"Line 24: After incrementing the clock by "
 <<"one second, myClock: "; //Line 24
 myClock.printTime(); //Line 25
 cout<<endl; //Line 26

 return 0;
} //end main

void clockType::setTime(int hours, int minutes, int seconds)
{
 if(0 <= hours && hours < 24)

```

```
 hr = hours;
 else
 hr = 0;

 if(0 <= minutes && minutes < 60)
 min = minutes;
 else
 min = 0;

 if(0 <= seconds && seconds < 60)
 sec = seconds;
 else
 sec = 0;
}

void clockType::getTime(int& hours, int& minutes, int& seconds)
{
 hours = hr;
 minutes = min;
 seconds = sec;
}

void clockType::incrementHours()
{
 hr++;
 if(hr > 23)
 hr = 0;
}

void clockType::incrementMinutes()
{
 min++;
 if(min > 59)
 {
 min = 0;
 incrementHours();
 }
}

void clockType::incrementSeconds()
{
 sec++;
 if(sec > 59)
 {
 sec = 0;
 incrementMinutes();
 }
}

void clockType::printTime() const
{
 if(hr < 10)
 cout<<"0";
 cout<<hr<<":";

 if(min < 10)
 cout<<"0";
 cout<<min<<":";

 if(sec < 10)
 cout<<"0";
 cout<<sec;
```

```

 }

 bool clockType::equalTime(const clockType& otherClock) const
 {
 return (hr == otherClock.hr
 && min == otherClock.min
 && sec == otherClock.sec);
 }

```

**实例运行：**在本运行实例中，用户的输入用阴影表示。

```

Line 2: myClock: 05:04:30
Line 5: yourClock: 0-858993460:0-858993460:0-858993460
Line 9: After setting - yourClock: 05:45:16
Line 15: The two times are not equal
Line 16: Enter hours, minutes, and seconds: 5 23 59
Line 20: New myClock: 05:23:59
Line 24: After incrementing the time by one second, myClock: 05:24:00

```

如输出的第二行所示 (Line 5: ), yourClock 的值是依赖于机器的，在不同的计算机上可能会得到不同的值。

### 13.1.5 类中公有和私有的顺序

C++ 没有规定声明公有和私有的固定顺序，可以以任何顺序进行定义。要注意的惟一一件事情是：默认情况下，所有的成员都是私有的，可以使用 public 关键字将一个变量声明为公有的。如果是在公有成员的声明语句后声明私有成员（正如在 clockType 中所做的那样），则必须使用 private 关键字进行私有成员的定义。

可以用例 13.3 至例 13.5 中的任何一种方式来定义类 clockType。

#### 例 13.3

```

class clockType
{
public:
 void setTime(int, int, int);
 void getTime(int&, int&, int&);
 void printTime() const;
 void incrementSeconds();
 void incrementMinutes();
 void incrementHours();
 bool equalTime(const clockType&) const;

private:
 int hr;
 int min;
 int sec;
};

```

#### 例 13.4

```

class clockType
{
private:
 int hr;
 int min;
 int sec;
};

```



```

public:
 void setTime(int, int, int);
 void getTime(int&, int&, int&);
 void printTime() const;
 void incrementSeconds();
 void incrementMinutes();
 void incrementHours();
 bool equalTime(const clockType&) const;
};

```

### 例 13.5

```

class clockType
{
 int hr;
 int min;
 int sec;

public:
 void setTime(int, int, int);
 void getTime(int&, int&, int&);
 void printTime() const;
 void incrementSeconds();
 void incrementMinutes();
 void incrementHours();
 bool equalTime(const clockType&) const;
};

```

在例 13.5 中, 因为标识符 `hr`、`min` 和 `sec` 不在任何关键字后, 所以它们是私有的。

一般的做法是先列出类的公有成员, 然后是私有成员, 这样可以将注意力集中到公有成员上。

### 13.1.6 构造函数

在例 13.2 中的程序中, 当不调用函数 `setTime()` 而打印 `yourClock` 的值时, 输出结果十分古怪 (即垃圾值, 参见运行实例的 Line 5)。这归咎于这样的事实: C++ 并不自动初始化变量。因为私有成员不能够从类以外访问 (例子中是数据成员), 如果忘记调用函数 `setTime()` 来初始化这些变量, 程序将会产生错误的结果。

要保证类中的数据成员被初始化, 必须使用构造函数。存在着两种构造函数: 带参数的构造函数和不带参数的构造函数。不带参数的构造函数称为默认构造函数。

构造函数具有如下性质:

- 构造函数的名字与类的名字相同。
- 构造函数尽管是一个函数, 但没有任何类型, 即它既不属于返回值函数也不属于 `void` 函数。
- 类可以有多个构造函数。然而, 一个类的所有构造函数的名字都相同。
- 如果类有多个构造函数, 则它们的参数是各不相同的。
- 当类对象创建时, 构造函数会自动地执行; 由于它们没有类型, 不能够像其他函数那样进行调用。
- 当类对象声明时调用哪一个构造函数取决于传递给它的参数类型。

可以将类 `clockType` 的定义加以扩展, 使其包含两个构造函数。

```

class clockType
{
public:
 void setTime(int, int, int);

```

```

 void getTime(int&, int&, int&);
 void printTime() const;
 void incrementSeconds();
 void incrementMinutes();
 void incrementHours();
 bool equalTime(const clockType&) const;
 clockType(int, int, int); //constructor with parameters
 clockType(); //default constructor

private:
 int hr;
 int min;
 int sec;
};

```

类 `clockType` 的定义包含两个构造函数：一个有三个参数，另外一个没有参数。现在我们来编写这两个构造函数的定义。

```

clockType::clockType(int hours, int minutes, int seconds)
{
 if(0 <= hours && hours < 24)
 hr = hours;
 else
 hr = 0;

 if(0 <= minutes && minutes < 60)
 min = minutes;
 else
 min = 0;
 if(0 <= seconds && seconds < 60)
 sec = seconds;
 else
 sec = 0;
}

clockType::clockType() //default constructor
{
 hr = 0;
 min = 0;
 sec = 0;
}

```

从这些构造函数的定义可以看出，默认构造函数将三个数据成员 `hr`、`min` 和 `sec` 设置为 0，而带参数的构造函数将数据成员设置为形参值。此外，还可以定义调用函数 `setTime()` 的构造函数，如下所示：

```

clockType::clockType(int hours, int minutes, int seconds)
{
 setTime(hours, minutes, seconds);
}

```

### 13.1.7 调用构造函数

回忆一下，当类对象声明时，构造函数会自动执行。因为一个类可能会有包括默认构造函数在内的不止一种构造函数，下面讨论如何调用特定的构造函数。

#### 调用默认构造函数

假设一个类包含有默认构造函数，调用默认构造函数的语法如下：

```

className classObjectName;

```



例如, 语句:

```
clockType yourClock;
```

将 yourClock 声明为 clockType 类型的对象。这种情况下, 默认构造函数会执行, 并且 yourClock 的数据成员初始化为 0。

**注意:** 如果声明了一个对象并且希望默认构造函数被执行, 声明语句中, 对象名后面是不需要空括号的。实际上, 如果偶然包含了空括号, 编译器将会产生一个语法错误信息。例如, 下面的语句是非法的:

```
clockType yourClock(); //illegal
```

### 调用带参数的构造函数

假设一个类中包含有带参数的构造函数, 下面的语法将会调用带参数的构造函数:

```
className classObjectName(argument1, argument2, ...);
```

其中, argument1, argument2, 等等既可以是变量, 也可以是表达式。

**注意:**

- 参数的数量和类型 (和顺序) 应该与所定义的一种构造函数相匹配。
- 如果参数的类型 (和顺序) 不与任何构造函数相匹配, 则 C++ 将使用类型转换并寻找最佳匹配。例如, 一个整型数可能会转换成小数部分为 0 的浮点数。任何不确定因素都会导致编译错误。

考虑语句:

```
clockType myClock(5, 12, 40);
```

这条语句声明了一个 clockType 类型的对象 myClock。这里传递了三个 int 型的值, 并与 clockType 类的一个带参数的构造函数相匹配。因此, 类 clockType 带参数的构造函数被执行并且对象 myClock 的三个数据成员分别赋值为 5、12 和 40。

例 13.6 显示了构造函数是如何工作的。

### 例 13.6

考虑下面的类定义:

```
class testClass
{
public:
 void print(); //Line a
 testClass(); //Line b
 testClass(int, int); //Line c
 testClass(int, double, int); //Line d
 testClass(double, char); //Line e

private:
 int x;
 int y;
 double z;
 char ch;
};
```

该类有 5 个成员函数, 其中包括 4 个构造函数以及 1 个数据成员。假设 testClass 的成员函数

定义如下:

```
void testClass::print()
{
 cout<<"x = "<<x<<"", y = "<<y<<"", z = "<<z
 <<"", ch = "<<ch<<endl;
}

testClass::testClass() //default constructor
{
 x = 0;
 y = 0;
 z = 0;
 ch = '*';
}

testClass::testClass(int a, int b)
{
 x = a;
 y = b;
 z = 0;
 ch = '*';
}

testClass::testClass(int a, double c, int b)
{
 x = a;
 y = b;
 z = c;
 ch = '*';
}

testClass::testClass(double c, char d)
{
 x = 0;
 y = 0;
 z = c;
 ch = d;
}
```

考虑下列声明:

```
testClass one; //Line 1
testClass two(5,6); //Line 2
testClass three(5,4.5,7); //Line 3
testClass four(4,9,12); //Line 4
testClass five(3.4,'D'); //Line 5
```

对于对象 one (在 Line 1 中声明), 由于没有传递任何参数, 默认构造函数 (Line b) 被执行。因此, one 的数据成员 x、y、z 和 ch 分别初始化为 0、0、0 和 '\*'。对于对象 two (Line 2 中声明), 将执行 Line c 中的构造函数。因为传递给对象 two 的参数是 int 型的 5 和 6, 所以, 对象 two 的数据成员 x、y、z 和 ch 被分别初始化为 5、6、0 和 '\*' (参见 Line c 的构造函数定义)。

传递给对象 three (Line 3 中声明) 的参数是 5、4.5 和 7。这些参数与 Line d 的构造函数相匹配, 因此, 当对象 three 创建时, Line d 的构造函数执行了。从 Line d 的构造函数定义可知, three 的数据成员 x、y、z 和 ch 分别初始化为 5、7、4.5 和 '\*'。

传递给对象 four (Line 4 中声明) 的参数是 4、9 和 12。类 testClass 不含有任何带三个 int 型参数的构造函数。然而, 有一个以 int、double 和 int 为参数的构造函数 (参见 Line d)。因为 int 值

可以转换为小数部分为 0 的 double 型值, 所以当对象 four 创建时, Line d 的构造函数执行, 所以 four 的数据成员 x、y、z 和 ch 相应地赋值为 4、12、6.0 和 '\*' (参见 Line d 构造函数的定义)。

传递给对象 five 的参数是 3.4 和 'D', 这些参数与 Line e 定义的构造函数相匹配, 所以当初始化对象 five 的数据成员时, Line e 中的构造函数执行。由 Line e 中的定义看出, five 的数据成员 x、y、z 和 ch 相应地赋值为 0、0、3.4 和 'D' (参见 Line e 构造函数的定义)。

**注意:** 如果传递给对象的值不匹配任何构造函数, 而且没有任何类型转换的可能, 则会产生一个编译错误。

下列程序测试了前面的类对象。

```
int main()
{
 testClass one; //Line 1
 testClass two(5, 6); //Line 2
 testClass three(5, 4.5, 7); //Line 3
 testClass four(4, 9, 12); //Line 4
 testClass five(3.4, 'D'); //Line 5
 one.print(); //Line 6; output one
 two.print(); //Line 7; output two
 three.print(); //Line 8; output three
 four.print(); //Line 9; output four
 five.print(); //Line 10; output five

 return 0;
}
```

**输出:**

```
x = 0, y = 0, z = 0, ch = *
x = 5, y = 6, z = 0, ch = *
x = 5, y = 7, z = 4.5, ch = *
x = 4, y = 12, z = 9, ch = *
x = 0, y = 0, z = 3.4, ch = D
```

正如前面所解释的, Line 1 到 Line 5 的语句声明和初始化了对象 one、two、three、four 和 five。Line 6 到 Line 10 的语句输出了这些对象的值, 即这些对象的数据成员。

### 构造函数和默认参数

构造函数同样有默认参数。在这种情况下, 声明形参的方法和一般函数中定义默认参数的方法一样(在第 7 章中讨论了带默认参数的函数)。使用定义默认参数的规则, 在类 clockType 的定义中, 可以使用下列语句代替构造函数(请回忆一下函数原型, 形参的名字是可选的)。

```
clockType clockType(int = 0, int = 0, int = 0);
```

在实现文件中, 该构造函数的定义与带参数的构造函数定义是一致的。

使用这种转换, 无参构造函数或带有所有默认参数的构造函数称为默认构造函数。

### 例 13.7

使用默认构造函数, 可以将类 testClass (前面定义过的) 定义如下:

```
class testClass
{
public:
 void print();

 testClass(int = 0, double = 0.0, int = 0, char = '*');
```

```
private:
 int x;
 int y;
 double z;
 char ch;
};
```

构造函数的定义如下:

```
testClass::testClass(int a, double c, int b, char d)
{
 x = a;
 y = b;
 z = c;
 ch = d;
}
```

可以用下列方式声明对象 one、two、three、four 和 five:

```
testClass one;
testClass two(5,0.0,6);
testClass three(5,4.5,7);
testClass four(4,9,12);
testClass five(0,3.4,0,'D');
```

在对象 two 的声明中, 使用了 z 和 ch 的默认值替换了 x 和 y 的默认值。因为 z 在 y 之前声明, 要使用 z 的默认值而替换 y 的默认值, 必须像声明中那样指定 z 的默认值。类似地, 在对象 five 的声明中, 使用 x 和 y 的默认值替换了 z 和 ch 的默认值, 因为 x 在 z 和 ch 之前, y 在 ch 之前, 所以必须像声明中那样指定 x 和 y 的默认值。

使用默认值简化了构造函数的声明和定义。因此, 只要可能 (在不产生混淆的前提下), 都应有效地利用带参数的构造函数。

### 13.1.8 类对象 (变量) 数组和构造函数

如果类含有构造函数并且声明了一个类对象数组, 那么该类必须具有默认构造函数。默认构造函数用于初始化 (数组中的) 每一个类对象。假设声明了包含 100 个对象的数组, 那么给每一个类对象指定不同的构造函数是不现实的。基于前面对类 clockType 的定义, 语句:

```
clockType clocks[100];
```

声明了一个有 100 个 clockType 型对象的数组 clocks (参见图 13.9)。

每一个数组元素的数据初始化为 0。

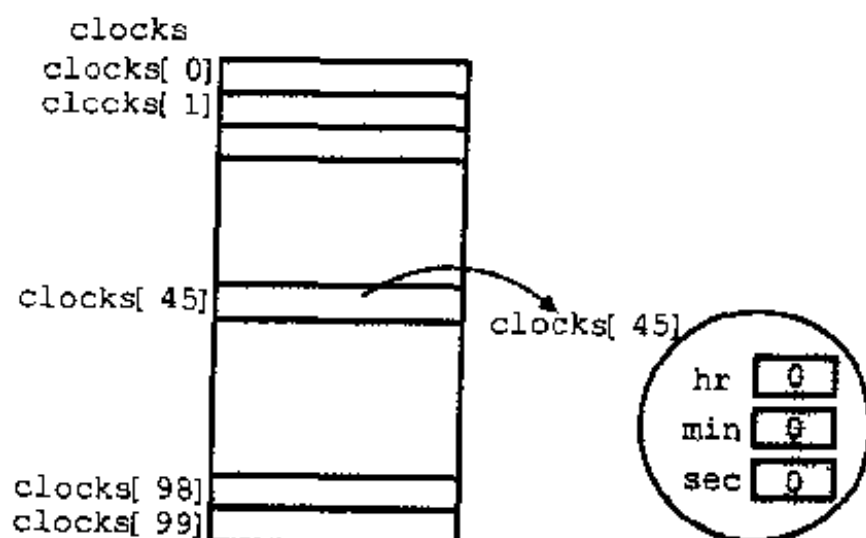


图 13.9 对象数组 clocks

### 13.1.9 析构函数

像构造函数一样,析构函数也是函数。而且它并没有任何类型:既不是返回值函数也不是 void 函数。然而一个类只能有一个析构函数,而且析构函数不能够带有参数。析构函数的名字是字符“~”加上类的名字。例如,类 clockType 的析构函数名字是:

```
~clockType();
```

当类对象销毁时,析构函数自动执行,本章后面将会讨论析构函数。

## 13.2 数据抽象、类和抽象数据类型

读者是否想过汽车的引擎是如何工作的?大多数人想知道的只是如何启动和驾驶汽车,而并不会费心地去关心引擎是如何工作的。通过将汽车引擎的设计细节与汽车的使用分离开,制造商得以帮助驾驶者将注意力集中在如何驾驶汽车上。在日常生活中还有其他类似的例子,大多数情况下,只需关注于如何使用某种东西,而不是它们是如何工作的。

将设计细节(即汽车引擎是如何工作的)同使用分离开的方法叫做抽象。换句话说,抽象集中在引擎能够做什么,而不是它如何工作。因此,抽象是将逻辑属性同实现相分离。驾驶汽车是一个逻辑属性;引擎的构造包含了实现细节。驾驶员只需对引擎做什么有一个抽象的观点,而无须关心引擎的具体实现。

抽象也可以应用于数据,前面几节定义了一种数据类型 clockType。数据类型 clockType 有三个数据成员和一些基本操作:

1. 设置时间。
2. 返回时间。
3. 打印时间。
4. 对时间增加 1 秒。
5. 对时间增加 1 分钟。
6. 对时间增加 1 小时。
7. 比较两个时间是否相等。

clockType 的实际实现被封装了。数据抽象将数据的逻辑属性与其具体实现加以分离。clockType 的定义和它的基本操作是逻辑属性;clockType 在计算机中的存储方式以及进行这些操作的算法是 clockType 的实现细节。

**抽象数据类型 (ADT):** 只描述逻辑属性而不包含实现细节的数据类型。

像任何其他数据类型一样,ADT 具有三个元素:ADT 的名字,称为类型名;属于 ADT 的数据集,称为域以及数据操作集。通过下列转换,可以定义 clockType 的 ADT 为如下形式:

```
dataTypeName
 clockType
domain
 Each clockType value is a time of day in the form of hours,
 minutes, and seconds.
operations
 Set the time.
 Return the time.
 Print the time.
 Increment the time by one second.
```

```

Increment the time by one minute.
Increment the time by one hour.
Compare the two times to see whether they are equal.

```

### 例 13.8

顺序表被定义为一组相同类型的值。因为顺序表中的所有值都是相同的类型，所以使用数组是一种方便的方法。

```

dataTypeName
 listType
domain
 Every element of the type listType is a set of, say 1000 numbers.
operations
 Check to see if the list is empty.
 Check to see if the list is full.
 Search the list for a given item.
 Delete an item from the list.
 Insert an item in the list.
 Sort the list.
 Destroy the list.
 Print the list.

```

显然，接下来的问题是如何在一个程序中实现ADT。要实现ADT，必须表示数据并编写进行操作的算法。

前面的几节使用类将数据和函数组织起来，而且定义了一个只包含操作集的类，用来实现操作的函数是分开来编写的。因此，可以看到类是实现ADT的一种方便方法。事实上，在C++中，类就是为处理ADT而专门设计的。

下面的类listType定义了顺序表的ADT。为确定起见，假设顺序表有一系列int型的元素（参见图13.10）。

```

class listType
{
public:
 bool isEmptyList();
 bool isFullList();
 void search(int searchItem, int& found);
 void insert(int newElement);
 void remove(int removeElement);
 void destroyList();
 void printList();
 listType(); //constructor

private:
 int list[1000];
 int length;
};

```

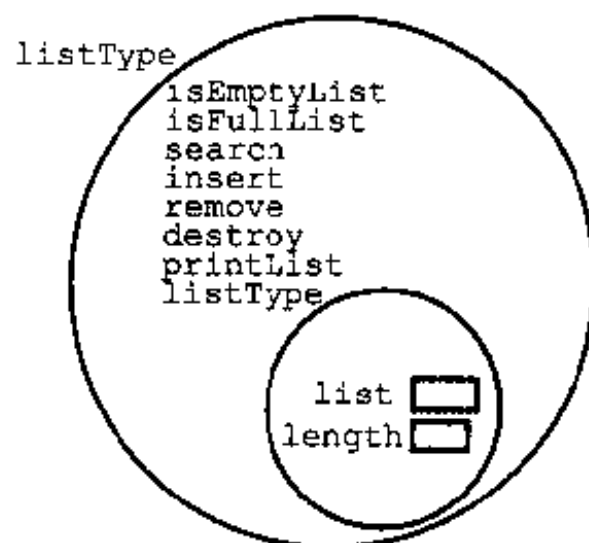


图13.10 类listType

## 13.3 结构体与类

在第12章中，将结构体定义为一个由各组成部分构成的成员集合，其中组成部分可以是不同的类型。结构体中成员的定义只包含数据。然而，C++结构体与C++类十分相似。在一个类中，“结构”的成员还可以是函数，其中包括构造函数和析构函数。结构体与类惟一不同之处在于：默认情况下，结构中成员是公有的，而类中成员则是私有的。也可以在结构中使用关键字private将成员定义为私有的。



在 C 语言中,结构的定义与第 12 章中给出的 C++ 中的定义相似。因为 C++ 是由 C 语言发展而来,标准的 C 结构体被 C++ 很好地承接了下来。然而,在 C++ 中结构的定义扩展为包含成员函数、构造函数和析构函数。将来,由于类专门针对 ADT 而设计,类的定义可能会演化为与类 C 结构大不相同的形式。

C++ 的类和结构体有着相同的性能,然而,大多数程序员坚持对结构的使用严格限定在类 C 结构的范畴内,而不允许它们包含成员函数。换句话说,如果一个类中的所用数据成员都是公有的,并且该类没有成员函数,通常用结构进行数据组织。

## 13.4 信息隐藏

前面几节在程序中定义了类 `clockType` 来实现时间问题,然后编写了一个程序使用类 `clockType`。实际上,我们可以将类 `clockType` 与实现操作的函数定义以及函数 `main()` 结合起来以便完成程序。即将类 `clockType` 的操作描述和实现细节直接结合到程序中。

在程序中同时包含对象的操作描述和实现细节是好的做法吗?绝对不是。第一,对象的私有数据成员对对象不再是私有的了。用户可以通过数据操作函数实现对数据进行直接访问。因此,用户可以随意更改操作。如果在一个项目中有几个程序使用相同的对象,并且它们可以直接修改对象的内部数据,就不能保证所有程序员都用相同的方式使用相同的对象。所以,必须隐藏实现的细节。用户应该只知道对象的功能,而无须了解其实现。隐藏实现细节使用户不必再理会那些额外的代码。同样,通过隐藏实现细节,可以保证在整个项目中所有对对象的使用都是以相同的方式进行的。这样,一旦对象经过编写、调试和适当的测试,就会变得很少出错。

本节讨论如何隐藏一个对象的实现细节,为了便于说明,将使用类 `clockType`。

要在程序中实现 `clockType`,用户必须声明 `clockType` 类型的对象,并且知道哪些操作是允许的,其功能分别是什么。所以用户必须可以访问操作的描述细节。因为用户并不关心实现细节,必须将这些细节分离到另外的文件中,称为实现文件。同样由于操作描述可能会很长,应避免用户在程序中将其直接包含。然而,用户必须能够查看操作描述的细节以便可以正确地调用函数。因此必须将操作描述细节置于单独的文件中,称包含操作描述细节的文件为头文件(或接口文件)。

实现文件包括函数的定义以便实现针对对象的操作。该文件除了包含诸如预处理指令等部分之外,还应该包含 C++ 语句。由于一个 C++ 程序只能含有一个 `main()` 函数,实现文件并不包含 `main()` 函数,只有用户程序才包含 `main()` 函数。因为实现文件不包含 `main()` 函数,不能够为该文件生成可执行代码,实际上,可以从实现文件产生称为目标代码的文件,然后用户可以将实现文件产生的目标代码与使用类的程序产生的目标代码进行连接,以产生最终的可执行代码。

最后,头文件有一个扩展名 `h`,而实现文件的扩展名为 `cpp`。假设类 `clockType` 的描述细节存在于名为 `clock` 的文件中,那么,该文件的全名应该是 `clock.h`。如果类 `clockType` 的实现细节所在的文件名为 `clockImp`,则它的全名必须是 `clockImp.cpp`。

文件 `clockImp.cpp` 只包含函数的定义,而没有类的定义。因此,要解决一个未定义标识符的问题(比如某一函数名或者变量名),需要在 `include` 语句的帮助下将头文件 `clock.h` 包含进文件 `clockImp.cpp` 中。任何使用类 `clockType` 的程序都需要下列 `include` 语句,以及定义类 `clockType` 相关操作的实现文件。

```
#include "clock.h"
```

注意,头文件 `clock.h` 被双引号而不是尖括号包含。头文件称为用户定义头文件,用户定义头文件使用双引号,而系统提供的头文件(例如 `iostream`)则使用尖括号。

下面是类 clockType 的描述文件和实现文件（注释中的 Post 是指 Post 条件，即该函数执行结束后的情形）：

```
//clock.h, the header file for the class clockType
class clockType
{
public:
 void setTime(int hours, int minutes, int seconds);
 //Function to set the time
 //Post: The time is set according to the
 //parameters: hr = hours; min = minutes;
 // sec = seconds;

 void getTime(int& hours, int& minutes, int& seconds)
 //Function to return the time
 //Post: hours = hr; minutes = min;
 // seconds = sec;

 void printTime() const;
 //Function to print the time
 //Time is printed in the form hh:mm:ss

 void incrementSeconds();
 //Function to increment the time by one second
 //Post: The time is incremented by one second
 //If the before-increment time is 23:59:59, the time
 //is reset to 00:00:00

 void incrementMinutes();
 //Function to increment the time by one minute
 //Post: The time is incremented by one minute
 //If the before-increment time is 23:59:53, the time
 //is reset to 00:00:53

 void incrementHours();
 //Function to increment the time by one hour
 //Post: The time is incremented by one hour
 //If the before-increment time is 23:45:53, the time
 //is reset to 00:45:53

 bool equalTime(const clockType& otherClock) const;
 //Function to compare the two times
 //Function returns true if this time is equal to
 //otherClock, otherwise returns false

 clockType(int hours, int minutes, int seconds);
 //Constructor with parameters
 //Post: The time is set according to
 // the parameters
 // hr = hours; min = minutes; sec = seconds

 clockType();
 //Default constructor with parameters
 //Post: time is set to 00:00:00
 // hr = 0; min = 0; sec = 0

private:
 int hr; //store hours
 int min; //store minutes
 int sec; //store seconds
};
```



```
//clockImp.cpp, the implementation file
#include <iostream>
#include "clock.h"

using namespace std;
.
.
.
//The definitions of the member functions of the class clockType go here
.
.
.
```

接下来,描述一下包含使用类 clockType 的程序用户文件。

```
//TestClock.cpp. The user program that uses the class clockType
#include <iostream>
#include "clock.h"
using namespace std;
.
.
.
//Place the definitions of the function main and the other
//user-defined functions here
.
.
.
```

## 13.5 可执行代码

前面几节讨论了如何隐藏类的实现细节,要在程序中使用对象,在程序执行期间必须能够访问到对象的实现细节(对象操作的实现)。本节讨论一个客户程序是如何对对象的实现细节进行访问的。为便于说明,将使用类 clockType 进行讲解。

如前面解释过的,要使用类 clockType,程序必须通过 include 语句包含头文件 clock.h。例如,下面的程序段包含头文件 clock.h:

```
//Program test.cpp
#include "clock.h"
.
.
.
int main()
{
.
.
.
}
```

程序 test.cpp 只能够包含头文件而不能包含实现文件。要从文件 test.cpp 创建出可执行程序,必须经过下列步骤:

1. 单独编译文件 clockImp.cpp 并建立目标文件 clockImp.obj。目标代码文件包含机器语言代码,但是并不是可执行文件形式。假设在计算机系统中,命令 cc 调用 C++ 编译器或者连接器(或者两者皆有)。命令行:

```
cc -c clockImp.cpp
```

会创建目标代码文件 clockImp.obj。

2. 要从源文件 test.cpp 生成可执行代码, 需要编译源文件 test.cpp, 创建目标文件 test.obj, 然后将 test.obj 和 clockImp.obj 进行连接来创建可执行文件 test.exe。如下命令将会创建可执行文件 test.exe:

```
cc test.cpp clockImp.obj
```

注意:

1. 要为源文件创建目标代码文件, 可以使用命令行选项 -c。例如, 要为名为 exercise.cpp 创建目标代码文件, 使用下列命令行:

```
cc -c exercise.cpp
```

2. 要将多个目标文件与一个源文件连接在一起, 可以在命令行中列出所有的目标文件。例如, 要连接 A.obj, B.obj 和源文件 test.cpp, 使用命令行:

```
cc test.cpp A.obj B.obj
```

3. 如果源文件被修改, 则必须重新编译。
4. 如果在一个源文件中所做的修改影响到其他文件, 则其他文件必须重新编译并重新连接。
5. 用户必须能够访问头文件和目标文件。访问头文件使用户了解该对象的功能以及如何使用它; 对目标文件的访问使用户可以将自己的程序与目标文件进行连接, 从而生成可执行代码。用户并不需要访问包含实现细节的源文件。

如第1章中所述, Visual C++、C++ Builder 和 Code Warrior 将编辑器、编译器和连接器集成在一起。使用一个命令, 程序就可以进行编译并与其他必要的文件连接。这些集成环境同样以项目的形式管理着多个程序文件。一个项目可能包含多个文件, 这些文件称为项目文件。通常, 这些集成环境会有一个叫做 build、rebuild 或者 make 的命令。当在项目中使用 build、rebuild 或者 make 命令时, 集成环境会自动地编译、连接所有相关的文件并创建出可执行代码。当项目中一个或者多个文件发生变动时, 可以使用这些命令重新进行编译和连接。

例 13.9 进一步说明了类是如何设计和实现的。例 13.9 中设计的类 personType 是十分有用的, 读者将会在后续章节中继续使用它。

### 例 13.9

人的属性包括姓氏和名字。其典型操作是设置和打印名字。下面的语句定义了一个包含这些属性的类 personType (参见图 13.11)。

```
class personType
{
public:
 void print() const;
 //Function to output the first name and last name
 //in the form firstName lastName

 void setName(string first, string last);
 //Function to set firstName and lastName according to
 //the parameters
 //Post: firstName = first; lastName = last;

 void getName(string& first, string& last);
 //Function to return firstName and lastName via the parameters
 //Post: first = firstName; last = lastName;

 personType(string first, string last);
```

```

 //constructor with parameters
 //Set firstName and lastName according to the parameters
 //Post: firstName = first; lastName = last;

personType();
 //default constructor
 //Initialize firstName and lastName to empty string
 //Post: firstName = ""; lastName = "";

private:
 string firstName; //store the first name
 string lastName; //store the last name
};

```

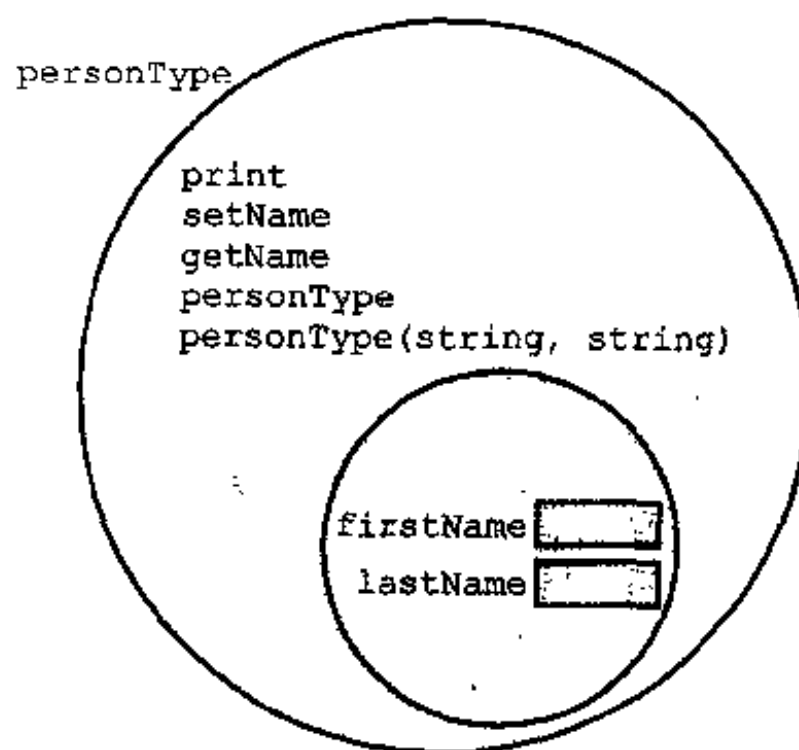


图 13.11

类 personType 的成员函数定义为:

```

void personType::print() const
{
 cout<<firstName<<" "<<lastName;
}

void personType::setName(string first, string last)
{
 firstName = first;
 lastName = last;
}

void personType::getName(string& first, string& last)
{
 first = firstName;
 last = lastName;
}

 //constructor with parameters
personType::personType(string first, string last)
{
 firstName = first;
 lastName = last;
}

personType::personType() //default constructor
{

```

```

 firstName = "";
 lastName = "";
 }

```

可以用一个带默认参数的构造函数来替换类 `personType` 现有的两个构造函数，这一工作留给读者作为练习。

## 13.6 编程实例：糖果机

购买糖果最为常见的地方就是糖果机。体育馆新近购买了一台糖果机，但是这台糖果机不能很好地工作。这台机器销售糖果、薯条、口香糖和小甜饼。假设需要为这台糖果机编写一个程序以使它可以投入运行。

程序应该有下列功能：

1. 向顾客显示糖果机所售的各种商品。
2. 让顾客进行选择。
3. 向顾客显示所选商品的价格。
4. 收款。
5. 发送顾客所选择的商品。

**输入** 商品选择和商品的价格。

**输出** 所选商品。

### 问题分析和算法设计

一台糖果机有两个主要部分：一个内置现钞收银机和几个分送器用来取出并发送商品。

**收银机** 首先讨论收银机的属性。收银机中存有一些现钞，可以找零。它从顾客那里得到现金，如果顾客所给的数额比所需的多，它可以将零钱找还给顾客。为简便起见，假设任何时刻顾客所付现金都正好是购买商品所需要的数目。收银机还应该能够在任何时刻告知糖果机的主人机器内总共有多少现金。下面的类定义了收银机的属性（参见图 13.12）。

```

class cashRegister
{
public:
 int currentBalance();
 //Function to show the current amount in the cash
 //register
 //Post: The value of the data member cashOnHand is returned

 void acceptAmount(int amountIn);
 //This function receives the amount deposited by
 //the customer and updates the amount in the register
 //Post: cashOnHand = cashOnHand + amountIn

 cashRegister(int cashIn = 500);
 //Constructor to set the cash in the register to a
 //specific amount
 //Post: cashOnHand = cashIn;
 //If no value is specified when the object is
 //declared, the default value assigned
 //to cashOnHand is 500

private:

```

```

 int cashOnHand; //variable to store the cash
 //in the register
};

```

接下来, 给出实现类 `cashRegister` 的函数定义。这些函数的定义十分简单并易于理解。

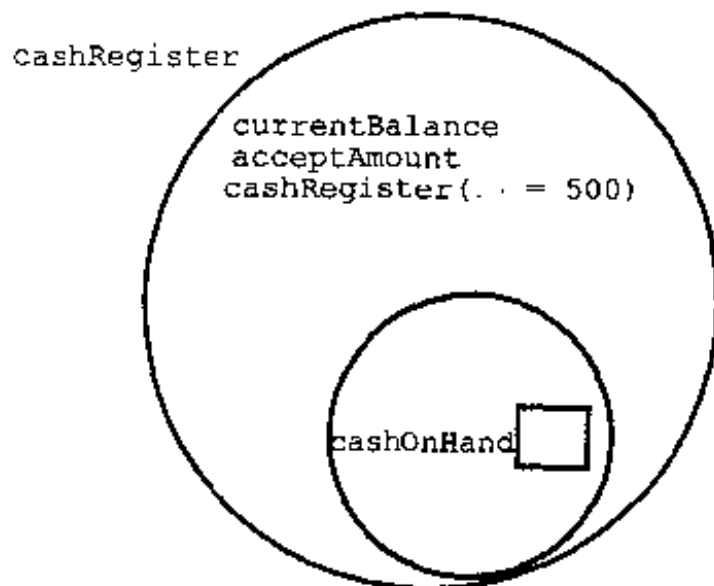


图13.12 类 `cashRegister`

函数 `currentBalance()` 显示当前收银机内的现金数量。它返回私有数据成员 `cashOnHand` 的值。所以它定义为:

```

int cashRegister::currentBalance()
{
 return cashOnHand;
}

```

函数 `acceptAmount()` 接收顾客输入的现金。它将顾客输入的金额加到当前总金额中并将其更新。其定义如下:

```

void cashRegister::acceptAmount(int amountIn)
{
 cashOnHand += amountIn;
}

```

在类 `cashRegister` 的定义中, 构造函数声明为带有一个默认参数的形式。因此, 当用户不带任何参数声明对象时, 构造函数会使用默认值初始化数据成员 `cashOnHand`。回忆一下, 由于在类的定义中指定了构造函数参数的默认值, 所以就不需要再显式地指定默认值了。构造函数的定义如下:

```

cashRegister::cashRegister(int cashIn)
{
 if (cashIn >= 0)
 cashOnHand = cashIn;
 else
 cashOnHand = 500;
}

```

注意, 构造函数的定义对参数 `cashIn` 的有效性进行了检查, 如果 `cashIn` 的值小于 0, 则数据成员 `cashOnHand` 就被赋值为 500。

**分送器** 分送器发送所选的商品 (如果有的话)。在分送器中应该能够显示商品的数目以及售价。下面的类 `dispenserType` 定义了分送器的属性 (参见图 13.13)。

```

class dispenserType

```

```

{
public:
 int count();
 //Function to return the number of items in the machine
 //Post: The value of the data member numberOfProducts
 is returned
 int productCost();
 //Function to return the cost of the item
 //The value of the data member cost is returned

 void makeSale();
 //Function to reduce the number of items by 1
 //Post: numberOfProducts = numberOfProducts -1;

 dispenserType(int setNoOfProducts = 50, int setCost = 50);
 //Constructor to set the cost and number of items
 //in the dispenser specified by the user
 //Post: numberOfProducts = setNoOfProducts;
 // cost = setCost;
 //If no value is specified for either of the two
 //parameters, the default values are assigned to the
 //data members numberOfProducts and cost.

private:
 int numberOfProducts; //variable to store the number of
 //items in the dispenser
 int cost; //variable to store the cost of an item
};

```

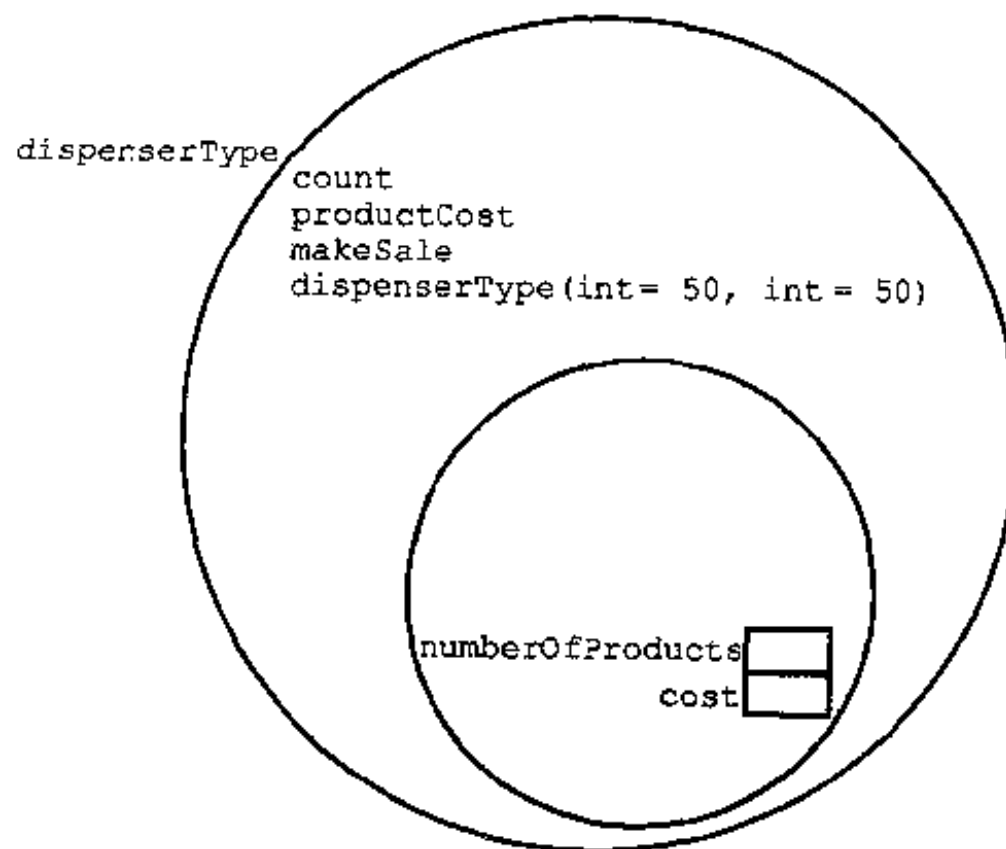


图 13.13 类 dispenserType

因为糖果机出售 4 种商品，所以应该定义 4 种 dispenserType 类型的对象。例如，语句：

```
dispenserType chips(100, 65);
```

将 chips 声明为一个 dispenserType 类的对象，并且将商品数量设置为 100，将商品（薯条）的售价设定为 65 美分（参见图 13.14）。

接下来，讨论函数定义以实现类 dispenserType 中的操作。

函数 count() 返回货架上某种商品的数目。由于当前商品的数目存储在私有的数据成员



numberOfProducts 中,函数将返回私有数据成员 numberOfProducts 的值。函数的定义如下:

```
int dispenserType::count()
{
 return numberOfProducts;
}
```

函数 productCost() 返回商品的价格,因为它存储在私有的数据成员 cost 中,函数将返回私有数据成员 cost 的值。函数的定义如下:

```
int dispenserType::productCost()
{
 return cost;
}
```

当一件商品被出售,分送器上该商品的数目应该减少 1。因此,函数 makeSale() 将分送器内商品的数目减少 1,即函数将私有数据成员 numberOfProducts 的值减少 1。函数的定义如下:

```
void dispenserType::makeSale()
{
 numberOfProducts--;
}
```

构造函数的定义检查参数值的有效性,如果值小于 0,默认值就会赋值给数据成员。构造函数的定义如下:

```
//constructor
dispenserType::dispenserType(int setNoOfProducts, int setCost)
{
 if(setNoOfProducts >= 0)
 numberOfProducts = setNoOfProducts;
 else
 numberOfProducts = 50;

 if(setCost >= 0)
 cost = setCost;
 else
 cost = 50;
}
```

### 主程序

当程序执行时,它必须完成下列功能:

1. 显示糖果机所销售的各种商品。
2. 显示如何选择某一商品。
3. 显示如何结束程序。

另外,在顾客进行某项选择后(退出程序除外),这些指示信息必须加以显示,以便当顾客需要购买多件商品时,不至于不知所措。一旦用户做出了适当的选择,糖果机就应该包含相应的动作。如果用户选择购买某种商品并且该商品存有现货,糖果机就应该显示该商品的价格并提示用户交款。如果用户放置的现金正好是所选商品的售价,糖果机应该售出该商品并显示适当的信息。

将讨论转换成下面的算法:

1. 向顾客显示选择。

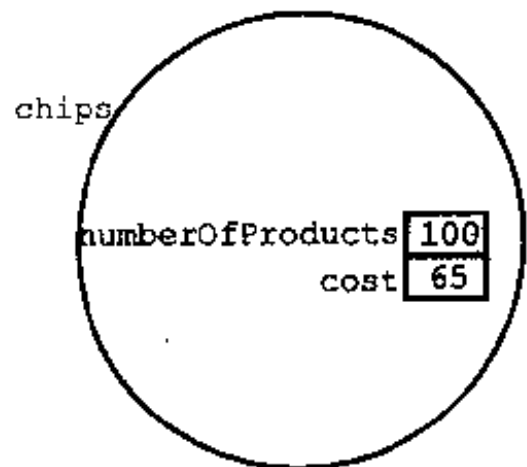


图13.14 对象 chips

2. 得到顾客的选择。
3. 如果选择是有效的并且对应的分送器非空, 售出该商品。

将该程序转化三个函数: `showSelection()`、`sellProduct()`和 `main()`。

`showSelection()`: 函数显示必要的信息以帮助顾客选择和购买一件商品。基本上, 它包含下列输出显示 (假设糖果机销售 4 种商品)。

- a. `*** Welcome to Shelly's Candy Shop ***`
- b. To select an item, enter
- c. 1 for Candy
- d. 2 for Chips
- e. 3 for Gum
- f. 4 for Cookies
- g. 9 to exit

该函数的定义如下:

```
void showSelection()
{
 cout<<"*** Welcome to Shelly's Candy Shop ***"<<endl;
 cout<<"To select an item, enter "<<endl;
 cout<<"1 for Candy"<<endl;
 cout<<"2 for Chips"<<endl;
 cout<<"3 for Gum"<<endl;
 cout<<"4 for Cookies"<<endl;
 cout<<"9 to exit"<<endl;
} //end showSelection
```

`sellProduct()`: 函数尝试售出顾客选择的商品, 因此它必须能够访问所选商品对应的分送器。函数要做的第一件事情是检查分送器中的商品是否售空。如果是, 函数提示顾客该商品已经售完。如果非空, 则提示用户投放购买该商品所需数量的现金。

如果用户没有放置足够的现金, `sellProduct()`告诉用户还需要多少现金。如果顾客还是没有提供足够的现金, 函数将已投入的现金退回。如果投入的现金足够, 则函数接收现金并且售出商品。售出商品的含义是将该商品所对应的分送器的计数减1, 并且通过将该商品的价格加到收银机中总销售金额中并更新它 (由于在本程序中并不返还顾客放置的多余的钱, 所以只需直接将用户投入的现金数相加即可)。

从讨论中可以看出: 函数 `sellProduct()`必须能够访问与商品对应的分送器 (将对应的商品数目减1并且显示商品的售价) 以及收银机 (更新销售额)。因此, 该函数有两个参数: 分别对应分送器和收银机。因此, 两个参数必须是引用传递。

该函数中算法的伪代码表示如下:

- a. if 分送器非空:
  - i. 显示并提示用户输入现金。
  - ii. 得到顾客投入的金额。
  - iii. if 金额少于商品售价:
    1. 显示并提示用户放入额外的金额。
    2. 计算顾客投入的总金额。
  - iv. if 顾客放入的金额足够:



1. 更新收银机的销售额。
  2. 售出商品——即将相应分送器中计数减 1。
  3. 显示适当信息。
- v. if 顾客放入的金额少于所选商品的售价, 返还现金。
- b. if 分送器已空, 通知用户商品已售完。

sellProduct() 的定义如下:

```
void sellProduct(dispenserType& product, cashRegister& pCounter)
{
 int amount; //variable to hold the amount entered
 int amount2; //variable to hold the extra amount needed

 if(product.count() > 0) //Step a
 {
 cout<<"Please deposit "<<product.productCost()
 <<" cents"<<endl; //Step a.i
 cin>>amount; //Step a.ii
 if(amount < product.productCost()) //Step a.iii
 {
 cout<<"Please deposit another "
 <<product.productCost() -amount //Step a.iii.1
 <<" cents"<<endl;
 cin>>amount2; //Step a.iii.2
 amount = amount + amount2; //Step a.iii.3
 }
 if(amount >= product.productCost()) //Step a.iv
 {
 pCounter.acceptAmount(amount); //Step a.iv.1
 product.makeSale(); //Step a.iv.2
 cout<<"Collect your item at the bottom"
 <<" and enjoy."<<endl; //Step a.iv.3
 }
 else
 cout<<"The amount is not enough. "
 <<"Collect what you deposited."<<endl; //Step a.v
 cout<<"*-*"
 <<endl<<endl;
 }
 else
 cout<<"Sorry this item is sold out."<<endl; //Step b
} //end sellProduct
```

### main() 函数

main() 函数描述如下:

1. 创建收银机——即声明 cashRegister 型变量。
2. 创建 4 个分送器——即声明 4 个 dispenserType 型对象并初始化。例如, 语句:

```
dispenserType candy(100, 50);
```

创建了分送器对象 candy 来处理糖果。商品数量设置为 100, 商品的售价为 50 美分。

3. 声明其他必要的变量。
4. 显示选项, 调用函数 showSelection()。

5. 得到顾客选择。
6. while 程序不退出 ( 选择 9 导致程序退出 ):
  - a. 售出商品, 调用函数 sellProduct()。
  - b. 显示选项, 调用函数 showSelection()。
  - c. 得到顾客选择。

函数 main() 的定义如下所示:

```
int main()
{
 cashRegister counter; //Step 1
 dispenserType candy(100,50); //Step 2
 dispenserType chips(100,65); //Step 2
 dispenserType gum(75,45); //Step 2
 dispenserType cookies(100,85); //Step 2

 int choice; //Step 3

 showSelection(); //Step 4
 cin>>choice; //Step 5

 while(choice != 9) //Step 6
 {
 switch(choice) //Step 6a
 {
 case 1: sellProduct(candy, counter);
 break;
 case 2: sellProduct(chips, counter);
 break;
 case 3: sellProduct(gum, counter);
 break;
 case 4: sellProduct(cookies, counter);
 break;
 default: cout<<"Bad Selection"<<endl;
 } //end switch

 showSelection(); //Step 6b
 cin>>choice; //Step 6c
 } //end while

 return 0;
} //end main
```

#### 完整程序清单

```
//Candy Machine Header File

class cashRegister
{
public:
 int currentBalance();
 //Function to return the current amount in the cash
 //register
 //The value of the data member cashOnHand is returned

 void acceptAmount(int amountIn);
 //This function receives the amount deposited by
 //the customer and updates the amount in the register
 //Post: cashOnHand = cashOnHand + amountIn
```

```

 cashRegister(int cashIn = 500);
 //Constructor to set the cash in the register to a
 //specific amount
 //Post: cashOnHand = cashIn;
 //If no value is specified when the object is
 //declared, the default value assigned
 //to cashOnHand is 500

private:
 int cashOnHand; //variable to store the cash
 //in the register
};

class dispenserType
{
public:
 int count();
 //Function to return the number of items in the machine
 //The value of the data member numberOfProducts is returned

 int productCost();
 //Function to return the cost of the item
 //The value of the data member cost is returned

 void makeSale();
 //Function to reduce the number of items by 1
 //Post: numberOfProducts = numberOfProducts -1;

 dispenserType(int setNoOfProducts = 50, int setCost = 50);
 //Constructor to set the cost and number of items
 //in the dispenser specified by the user
 //Post: numberOfProducts = setNoOfProducts;
 // cost = setCost;
 //If no value is specified for either of the two
 //parameters, the default value is assigned to the
 //data members numberOfProducts and cost
private:
 int numberOfProducts; //variable to store the number of
 //items in the dispenser
 int cost; //variable to store the cost of an item
};

//Implementation file candyMachineImp.cpp
//This file contains the definitions of the functions
//to implement the operations of the classes
//cashRegister and dispenserType

#include <iostream>
#include "candyMachine.h"
using namespace std;
int cashRegister::currentBalance()
{
 return cashOnHand;
}

void cashRegister::acceptAmount(int amountIn)
{
 cashOnHand += amountIn;
}

cashRegister::cashRegister(int cashIn)

```

```
{
 if(cashIn >= 0)
 cashOnHand = cashIn;
 else
 cashOnHand = 500;
}

int dispenserType::count()
{
 return numberOfProducts;
}

int dispenserType::productCost()
{
 return cost;
}

void dispenserType::makeSale()
{
 numberOfProducts--;
}

dispenserType::dispenserType(int setNoOfProducts, int setCost)
{
 if(setNoOfProducts >= 0)
 numberOfProducts = setNoOfProducts;
 else
 numberOfProducts = 50;
 if(setCost >= 0)
 cost = setCost;
 else
 cost = 50;
}

//Main program:
#include <iostream>
#include "candyMachine.h"

using namespace std;

void showSelection();
void sellProduct(dispenserType& product,
 cashRegister& pCounter);

int main()
{
 cashRegister counter;
 dispenserType candy(100,50);
 dispenserType chips(100,65);
 dispenserType gum(75,45);
 dispenserType cookies(100,85);

 int choice; //variable to hold the selection
 showSelection();
 cin>>choice;

 while(choice != 9)
 {
 switch(choice)
 {
```

```

 case 1: sellProduct(candy, counter);
 break;
 case 2: sellProduct(chips, counter);
 break;
 case 3: sellProduct(gum, counter);
 break;
 case 4: sellProduct(cookies, counter);
 break;
 default: cout<<"Bad Selection"<<endl;
 } //end switch
 showSelection();
 cin>>choice;
} //end while

 return 0;
} //end main

void showSelection()
{
 cout<<"*** Welcome to Shelly's Candy Shop ***"<<endl;
 cout<<"To select an item, enter "<<endl;
 cout<<"1 for Candy"<<endl;
 cout<<"2 for Chips"<<endl;
 cout<<"3 for Gum"<<endl;
 cout<<"4 for Cookies"<<endl;
 cout<<"9 to exit"<<endl;
} //end showSelection

void sellProduct(dispenserType& product,
 cashRegister& pCounter)
{
 int amount; //variable to hold the amount entered
 int amount2; //variable to hold the extra amount needed

 if(product.count() > 0) //if dispenser is not empty
 {
 cout<<"Please deposit "<<product.productCost()
 <<" cents"<<endl;
 cin>>amount;

 if(amount < product.productCost())
 {
 cout<<"Please deposit another "
 <<product.productCost() -amount
 <<" cents"<<endl;
 cin>>amount2;
 amount = amount + amount2;
 }

 if(amount >= product.productCost())
 {
 pCounter.acceptAmount(amount);
 product.makeSale();
 cout<<"Collect your item at the bottom and enjoy."<<endl;
 }
 else
 cout<<"The amount is not enough. "
 <<"Collect what you deposited."<<endl;
 }

 cout<<"*-*"
 <<endl<<endl;
}

```

i

**运行实例：**在本运行实例中，用户的输入用阴影表示。

\*\*\* Welcome to Shelly's Candy Shop \*\*\*

## 13.7 小结

1. 类是固定数量的组成部分构成的集合。
2. 类的组成部分称为类成员。
3. 类的成员通过名字来访问。
4. 在 C++ 中, `class` 是保留字。
5. 类的成员可以归结为下面三种之一: 私有、保护和公有。
6. 类的私有成员不能从类的外部直接进行访问。
7. 类的公有成员可以从类的外部直接进行访问。
8. 默认情况下, 所有类的成员都是私有的。
9. 公有成员使用符号 `public` 声明。
10. 私有成员使用符号 `private` 声明。
11. 一个类的成员可以是变量 (即数据) 也可以是函数。



12. 如果类的成员是函数，通常使用函数原型来声明它。
13. 如果类的成员是变量，它的声明方式与任何其他类型变量一样。
14. 在类的定义中，不能在变量声明时进行初始化。
15. 在 C++ 中，类是一个定义，并不进行内存分配；内存分配发生在声明类变量时。
16. 在 C++ 中，类变量称为类的对象或者简称对象。
17. 任何使用类的程序（软件）称为类的客户程序。
18. 类的成员，可以通过使用类变量名字、圆点运算符（.）、成员名的方法进行访问。
19. 类中唯一的内建操作是赋值和成员访问。
20. 类作为函数参数时，既可以用传值方式，也可以用引用方式。
21. 类可以作为函数的返回值。
22. 构造函数保证了当对象声明时数据成员被初始化。
23. 构造函数的名字与类的名字相同。
24. 一个类可以有多于一个的构造函数。
25. 不带参数的构造函数称为默认构造函数。
26. 构造函数在类对象创建时自动执行。
27. 析构函数在类对象销毁时自动执行。
28. 一个类只能有一个析构函数，而且析构函数没有参数。
29. 析构函数的名字是 ~ 加上类的名字（中间没有空格）。
30. 构造函数和析构函数没有任何类型，即它们既不属于返回值函数也不属于 void 函数。它们不能够像其他函数那样被调用。
31. 只描述逻辑属性而不包含实现细节的数据类型称为抽象数据类型（ADT）。
32. 类是 C++ 中专门设计用来处理 ADT 的数据类型。
33. 要实现 ADT，必须对数据进行描述并且编写相关的算法以实现其操作。

## 13.8 练习

1. 判断对错：
  - a. 类的数据成员必须是相同类型的。
  - b. 类的函数成员必须是公有的。
  - c. 一个类可以有不止一个构造函数。
  - d. 一个类可以有不止一个析构函数。
  - e. 构造函数和析构函数都可以有参数。
2. 找出下列类定义中的语法错误。

```
a. class AA
{
 public:
 void print();
 int sum();
 AA();
 int AA(int , int);
 private :
 int x ;
 int y ;
};
```

```

b. class BB
{
 int one ;
 int two;
public:
 bool equal();
 print();
 BB(int, int);
}
c. class CC
{
public:
 void set(int, int);
 void print();
 CC();
 CC(int, int);
 bool CC(int, int);
private:
 int u;
 int v;
};

```

## 3. 考虑下列定义:

```

class xClass
{
public:
 void func();
 void print() const;
 xClass ();
 xClass (int, double);
private:
 int u;
 double w;
};

```

xClass x;

- a. 对象 x 有多少个成员?
- b. 对象 x 有多少个私有成员?
- c. 对象 x 有多少个构造函数?
- d. 编写成员函数 func() 的定义, 其中 u 设置为 10, w 设置为 15.3。
- e. 编写成员函数 print() 的定义, 以便打印 u 和 w 的值。
- f. 编写类 xClass 默认构造函数的定义, 其中私有数据成员初始化为 0。
- g. 编写 C++ 语句, 打印对象 x 数据成员的值。
- h. 编写 C++ 语句, 声明一个 xClass 类型的对象, 并且将其数据成员分别设置为 20 和 35.0。

## 4. 考虑下列类定义:

```

class CC
{
public:
 CC(); //Line 1
 CC(int); //Line 2
 CC(int, int) //Line 3
 CC(double, int) //Line 4
};

```



```
private:
 int u;
 double v;
};
```

a. 分别给出下列声明中执行的行号:

- (i) CC one;
- (ii) CC two(5, 6);
- (iii) CC three(3.5, 8);

b. 编写 Line 1 构造函数的定义, 其中私有数据成员初始化为 0。

c. 编写 Line 2 构造函数的定义, 其中私有数据成员 u 初始化为参数的值, 私有数据成员 w 初始化为 0。

d. 编写 Line 3 和 Line 4 构造函数的定义, 其中私有数据成员初始化为相应参数的值。

5. 考虑下列类的定义:

```
class testClass
{
public:
 int sum();
 //return the sum of the private data members
 void print() const;
 //print the values of the private data members
 testClass();
 //default constructor
 //initialize the private data members to 0
 testClass(int a, int b);
 //constructors with parameters
 //initialize the private data members to the values
 //specified by the parameters
 //Post condition: x = a; y = b;
private:
 int x;
 int y;
};
```

a. 按照类 testClass 定义中所描述的, 编写成员函数的定义。

b. 编写一个测试程序来测试类 testClass 的各种操作。

## 13.9 编程练习

1. 编写程序将以罗马数字形式输入的数字转化为十进制数。程序中需要包含一个叫做 romanType 的类。类 romanType 的对象应该完成下列功能:

- a. 存储罗马数字。
- b. 将数字转换并存储为十进制数。
- c. 根据用户的要求, 将数字以罗马数字或者十进制数的形式打印出来。

罗马数字对应的十进制数为:

|   |      |
|---|------|
| M | 1000 |
| D | 500  |
| C | 100  |
| L | 50   |

|   |    |
|---|----|
| X | 10 |
| V | 5  |
| I | 1  |

d. 使用下列罗马数字测试程序: MCXIV, CCCLIX, MDCLXVI。

2. 设计并实现类 `dayType` 从而完成计算日期的功能。类 `dayType` 存储日期的标识, 例如 Sun 代表星期日。程序能够进行下列操作:

a. 设置日期。

b. 打印日期。

c. 返回日期。

d. 返回当前日期的后一天。

e. 返回当前日期的前一天。

f. 计算并返回当前日期累加指定天数后的结果。例如, 如果当前是星期一, 4天后是星期五。

类似地, 如果当前是星期二, 13天后其结果是星期一。

g. 添加适当的构造函数。

3. 写出函数的定义以实现编程练习 2 类 `dayType` 中的操作。

4. 例 13.9 定义了类 `personType` 来储存某人的名字, 其中只包含了用来打印名字和设置名字的成员函数。将类 `personType` 重新定义以便增加下列功能:

a. 只设置 last name。

b. 只设置 first name。

c. 存储和设置 middle name。

d. 检查给定的 last name 是否与本人的 last name 相同。

e. 检查给定的 first name 是否与本人的 first name 相同。

编写该类的函数定义以实现各种操作。

5. a. 一本书的属性包括: 标题、作者、出版商、ISBN、价格和库存量。设计一个类 `bookType` 将一本书定义为一个 ADT。

i. 类 `bookType` 的每一个对象可以存储下列关于书的信息: 标题、最多 4 个作者、出版商、ISBN、价格和库存量。请添加一个数据成员来表示作者数量。

ii. 包含 `bookType` 类对象所需各种操作的成员函数。例如, 对于标题, 通常的操作是显示标题、设置标题和检查给定标题是否与原书相同。类似地, 对于库存量通常的操作是显示库存量、设置库存量、更新库存量、返回库存量等。请添加对出版商、ISBN、价格和作者的相应操作, 并添加适当的构造函数和析构函数 (如果需要)。

b. 编写类 `bookType` 成员函数的定义。

c. 编写程序, 使用类 `bookType` 并且测试类 `bookType` 中的各种操作。声明一个包括 100 个 `bookType` 型元素的数组并进行一些相应的操作, 例如, 标题查找、ISBN 查找以及更新某本书的库存量。

6. 设计类 `memberType`。

a. 类的对象可以存储某人的名字、会员 ID、已购书数量和消费总额。

b. 包含成员函数完成 `memberType` 中的各种操作。例如, 更改、设置和显示某人名字。更新、修改和显示购书数量和消费总额。

c. 如果需要, 添加合适的构造函数和析构函数。

d. 编写类 `memberType` 的成员函数。

7. 使用编程练习 5 和练习 6 中设计的类编写一个模拟书店的程序。书店有两种顾客：会员和一般消费者。每一位会员每年必须缴纳 10 美元并得到每本书 5% 的折扣。
- 对于每位会员，书店记录其购书数量以及消费总额，会员每购满 11 本书，书店都要计算前 10 本书的平均价格，然后将该价格作为对第 11 本书所打的折扣并将购书总额清 0。
- 编写可以处理 1000 本书和 500 位会员的程序。程序要通过菜单提供给用户各种选择，使用户可以方便地使用程序。

## 第14章 继承和组合

本章要点:

- 学习继承
- 学习派生类和基类
- 研究如何重定义基类的成员函数
- 分析基类和派生类的构造函数
- 学习如何构造派生类的头文件
- 研究三种类型的继承: 公有、保护和私有
- 学习组合
- 熟悉三种基本的面向对象设计原则

在第13章中介绍了类、抽象数据类型(ADT)和用C++实现ADT的方法。通过使用类可以把数据和操作结合到单一单元中。因此,对象成为了一个自包含的实体。操作可以直接访问数据,但对象的内部状态不能直接地加以操作。

除了实现ADT以外,类还有其他特征。例如,可以由现存的类创建出新的类。这一重要特征有利于代码重用。在C++中,可以用多种方法把两个或多个类联系起来。两种常用的、有意义的方法是继承(“is-a”关系)和组合(“has-a”关系)。

### 14.1 继承

假设要设计一个partTimeEmployee类,用来实现和处理兼职雇员的信息。与兼职雇员相关的主要特征是名字、薪水和工作的小时数。在例13.9(第13章)中,曾设计了一个类来实现姓名。每一名兼职雇员都是一个人,所以,与其从零开始设计一个partTimeEmployee类,不如通过增加新的成员(变量和/或函数)扩展personType类(例13.9)的定义。

当然,我们不想直接修改personType类,增加和/或删除它的类成员。实际上,需要在保持personType类不变的同时创建partTimeEmployee类并增加所需的成员。例如,因为personType类已经具有存储姓名的成员变量,在partTimeEmployee类中将不再包含这种成员。实际上,这些成员变量将从personType类中继承而来。(将在例14.1中设计这样一个类。)

在第13章中,曾广泛地研究和设计了clockType类,从而处理一天中的时间问题。clockType类有三个成员变量存储小时、分钟和秒。某些应用程序,除了小时、分钟和秒之外,也许还需要存储时区。在这种情况下,需要创建一个新的类extClockType以扩展clockType类的定义,从而容纳新的信息。也就是说,通过增加一个成员变量timeZone和必要的处理时间的成员函数(见本章末的编程练习1),来获得extClockType类。在C++中,允许完成这种任务的机制是继承原则。继承是“is-a”关系,例如,“每一名雇员是一个人”。

继承能够在已有类的基础上创建新的类。在已有类基础上创建的类称为派生类,已有类则成为基类。派生类继承了基类的属性。因此,与其完全从零开始创建新的类,不如利用继承的优点,减少软件的复杂度。

每个派生类也许会成为未来某个派生类的基类。继承可以是单重继承也可以是多重继承。在单重继承中，派生类从一个基类派生而来；在多重继承中，派生类派生于多个基类。本章主要讨论单重继承。

继承可以看做是树状结构或分层结构，其中基类下面是它的派生类。如图14.1中的树状图所示。

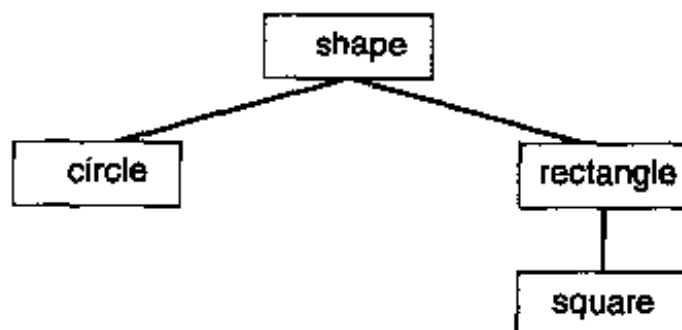


图 14.1 继承层次体系

在该图示中，shape 是基类。circle 类和 rectangle 类派生于 shape 类，square 类派生于 rectangle 类。每一个 circle 类和 rectangle 类都是一个 shape 类。

假设定义了一个 shape 类。为了指定 circle 类派生于 shape 类，通常使用如下的语法：

```
class circle: public shape
{
 .
 .
 .
};
```

public 称为成员访问描述符。它说明 shape 类中所有的公有成员都被 circle 类继承为公有成员。换句话说，shape 类中的公有成员成为了 circle 类中的公有成员。

另一方面，考虑下面的 circle 类定义：

```
class circle: private shape
{
 .
 .
 .
};
```

在该定义中，shape 的公有成员成为了 circle 类的私有成员。因此，任何 circle 类型的对象都不能直接访问这些成员。前面的 circle 定义等同于：

```
class circle: shape
{
 .
 .
 .
};
```

就是说，如果不使用 public 或 private 中的任何一种访问描述符，基类中的公有成员就继承为私有成员。

派生类的一般语法是：

```
class className: memberAccessSpecifier baseClassName
{
 member list
};
```

其中访问描述符包括 public、protected 或 private。没有指定访问描述符时，默认为私有继承。（在本章稍后讨论保护继承。）

下列有关基类和派生类的概念应该牢记于心。

1. 基类的私有成员对于基类来说是私有的。因此，派生类的成员不能直接访问它们。换句话说，当定义派生类的成员函数时，不能直接访问基类的私有成员。
2. 基类的公有成员可以被派生类继承为公有或私有成员。就是说，基类的公有成员可以成为派生类的公有或私有成员。
3. 派生类可以包括额外的成员变量和/或成员函数。
4. 派生类可以重定义基类的公有成员函数。就是说，派生类可以拥有名字、参数个数和参数类型与基类的成员函数都相同的成员函数。不过，这种重定义仅对派生类的对象有效，不影响基类的对象。
5. 基类的所有成员变量都是派生类的成员变量。同样，基类的成员函数（除非重定义）也都是派生类的成员函数。（当在派生类中访问基类的成员时，记住第1条规则。）

下一节描述与继承相关的两个重要问题。第一个问题是在派生类中重定义基类的成员函数。当讨论这一问题的时候，也将关注如何在派生类中访问基类的私有成员（变量）。第二个关键的继承问题和构造函数相关。派生类的构造函数不能直接访问基类的私有成员变量。因此，需要保证当执行派生类的构造函数时，从基类继承而来的私有成员变量被初始化。

### 14.1.1 重定义基类的成员函数

假设 derivedClass 类派生于 baseClass 类，derivedClass 和 baseClass 都有某些成员变量。那么可以得知，derivedClass 类同时包含了 baseClass 的成员变量。假设 baseClass 包含一个打印 baseClass 成员变量值的 print() 函数。现在 derivedClass 包含了从 baseClass 继承而来的成员变量。假设要包括一个打印 derivedClass 成员变量的函数，可以任意地命名该函数。然而，在 derivedClass 类中，可以把该函数命名为 print()（与 baseClass 所用的函数名相同）。这称为重定义基类的成员函数。下面将通过例子阐述如何重定义基类的成员函数。

**注意：**为了在派生类中重定义基类的公有成员函数，派生类中相应的函数必须有相同的名字、参数个数和类型。换句话说，在派生类中重定义的函数必须有同样的名字和相同的参数集。如果基类和派生类中相应的函数有相同的名字但参数集不同，那么这是派生类中的函数重载，这种情况也是允许的。

考虑下面类的定义（参见图 14.2）：

```
class baseClass
{
public:
 void print() const;

private:
 int u;
 int v;
 char ch;
};
```

baseClass 类有 4 个成员。假设 baseClass 的成员函数 print() 如下定义：

```
void baseClass::print() const
```

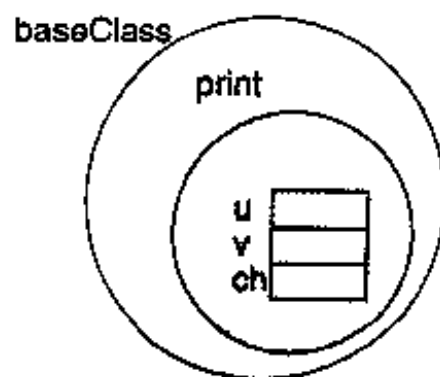


图14.2 baseClass类



```

{
 cout<<"Base Class: u = "<<u<<" , v = "<<v
 <<" , ch = "<<ch<<endl;
}

```

现在考虑下面类的定义 (参见图 14.3):

```

class derivedClass: public baseClass
{
public:
 void print() const;

private:
 int first;
 double second;
};

```

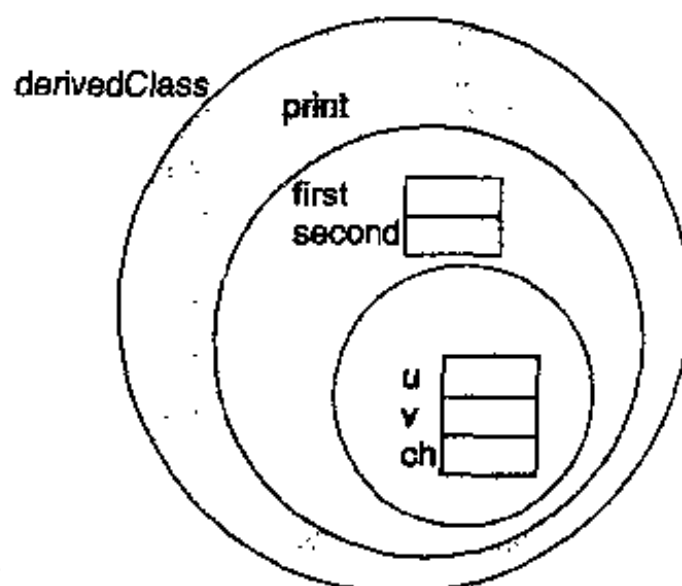


图14.3 derivedClass类

从 derivedClass 类的定义可知, 显然 derivedClass 类派生于 baseClass 类, 而且是一种公有继承。因此, baseClass 所有的公有成员继承为 derivedClass 的公有成员。derivedClass 类还重写了 print() 函数。

下面, 让我们写出 derivedClass 类中成员函数 print() 的定义。

derivedClass 类有 5 个成员变量: u、v、ch、first 和 second。derivedClass 的 print() 函数打印这些成员变量的值。为了写出 derivedClass 类中成员函数 print() 的定义, 牢记如下原则:

- u、v 和 ch 成员变量是 baseClass 类的私有成员, 因此不能在 derivedClass 中直接访问。所以, 对于 derivedClass 成员函数 print() 的定义, 不能直接引用 u、v 和 ch。
- baseClass 类的 u、v 和 ch 成员变量在 derivedClass 中可以通过 baseClass 的公有成员进行访问。

所以, 当编写 derivedClass 类成员函数 print() 的定义时, 首先调用 baseClass 的成员函数 print() 打印 u、v 和 ch 的值。在打印 u、v 和 ch 的值之后, 输出 first 和 second 的值。

为了在 derivedClass 的 print() 函数定义中调用 baseClass 的 print() 函数, 必须使用下面语句:

```
baseClass::print();
```

语句保证了调用的是 baseClass 而不是 derivedClass 的成员函数 print()。

derivedClass 类中成员函数 print() 的定义为:

```

void derivedClass::print() const
{
 baseClass::print();
 cout<<"Derived Class: first = "<<first
 <<" , second = "<<second<<endl;
 cout<<"
 <<"_____<<endl;
}

```

现在考虑下面的语句:

```

baseClass baseObject;
derivedClass derivedObject;

```

第一条语句声明 baseObject 是一个 baseClass 类的对象。因此, baseObject 对象有三个成员变量: u、v 和 ch。第二条语句声明 derivedObject 是一个 derivedClass 类的对象。因此, derivedObject 对象有 5 个成员变量: u、v、ch、first 和 second。(参见图 14.4。)

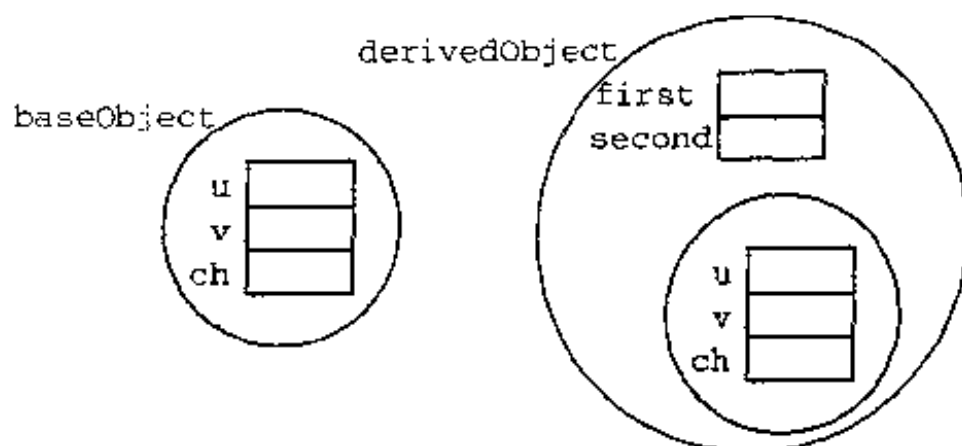


图 14.4 baseObject 对象和 derivedObject 对象

假设 baseObject 的 u、v 和 ch 成员变量的值分别是 5、6 和 '\*'。derivedObject 的 u、v、ch、first 和 second 成员变量的值分别是 12、76、&、32 和 16.38。(参见图 14.5。)

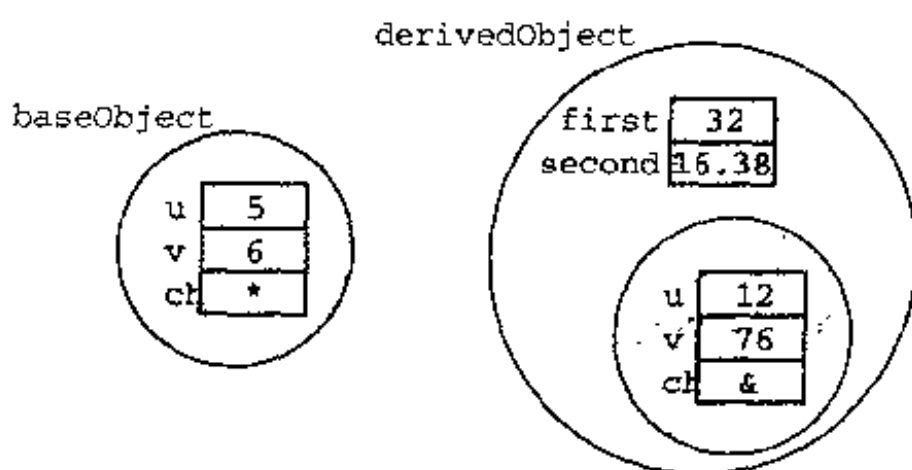


图 14.5 baseObject 对象和 derivedObject 对象的值

考虑下列语句:

```
baseObject.print();
derivedObject.print();
```

在第一条语句中, baseClass 的 print() 函数被执行; 在第二条语句中, 与 derivedClass 类相关的 print() 函数被执行。如果派生类重定义了基类的成员函数, 重定义仅对派生类的对象起作用。因此, 第一条语句的输出为:

```
Base Class: u = 5, v = 6, ch = *
```

第二条语句的输出为:

```
Base Class: u = 12, v = 76, ch = &
Derived Class: first = 32, second = 16.38
```

### 14.1.2 派生类和基类的构造函数

派生类可以有自己的私有成员变量, 当然也可以有自己的构造函数。构造函数通常的作用是初始化成员变量。当声明一个派生类对象时, 该对象继承了基类的成员, 但派生类对象不能直接访问基类的私有成员变量, 派生类的成员函数同样如此。也就是说, 派生类的成员函数不能直接访问基类的私有成员。

因此, 派生类的构造函数只能(直接)初始化派生类的私有成员变量。当声明一个派生类对象时, 它必须自动地执行基类的一个构造函数。因为构造函数不能像其他函数一样调用, 派生类构造函数的执行必须引发基类中某个构造函数的执行。事实也是如此, 而且, 基类构造函数的调用是在派生类构造函数定义的头部指定的。

下面通过一个例子来阐明上述观点。首先定义一个基类, 然后定义一个派生类, 这两个类都有



自己的构造函数。

考虑下面的类定义 (参见图 14.6):

```
class baseClass
{
public:
 void print(); //baseClass Line 1
 baseClass(); //baseClass Line 2
 baseClass(int x, int y); //baseClass Line 3
 baseClass(int x, int y, char w); //baseClass Line 4
private:
 int u;
 int v;
 char ch;
};
```

baseClass 类有 3 个构造函数、1 个成员函数 print() 和 3 个成员变量。假如 baseClass 的成员函数和构造函数定义如下:

```
void baseClass::print()
{
 cout<<"Base Class: u = "<<u<<"", v = "
<<v
 <<"", ch = "<<ch<<endl;
}

//default constructor; baseClass Line 2
baseClass::baseClass()
{
 u = 0;
 v = 0;
 ch = '*';
}

//constructor; baseClass Line 3
baseClass::baseClass(int x, int y)
{
 u = x;
 v = y;
 ch = '*';
}

//constructor; baseClass Line 4
baseClass::baseClass(int x, int y, char w)
{
 u = x;
 v = y;
 ch = w;
}
```

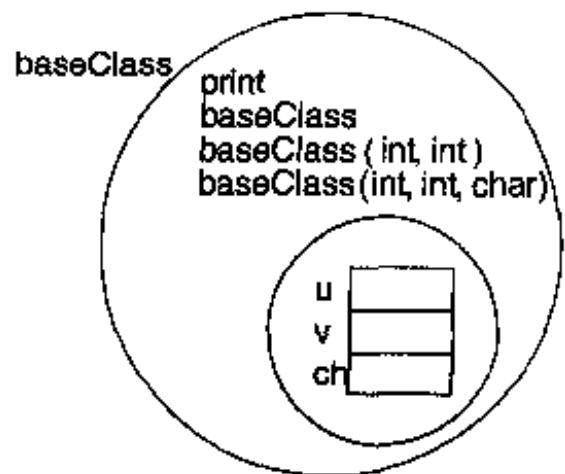


图14.6 带构造函数的baseClass类

现在考虑如下的类定义 (参见图 14.7):

```
class derivedClass: public baseClass
{
public:
 void print(); //derivedClass Line 1
 derivedClass(); //derivedClass Line 2
 derivedClass(int x, int y,
```

```

 int one, double two); //derivedClass Line 3
derivedClass(int x, int y, char w,
 int one, double two); //derivedClass Line 4

private:
 int first;
 double second;
};

```

derivedClass 类派生于 baseClass 类，是公有继承。derivedClass 类有 5 个成员变量：u、v、ch、first 和 second。成员变量 u、v 和 ch 继承自 baseClass 类。

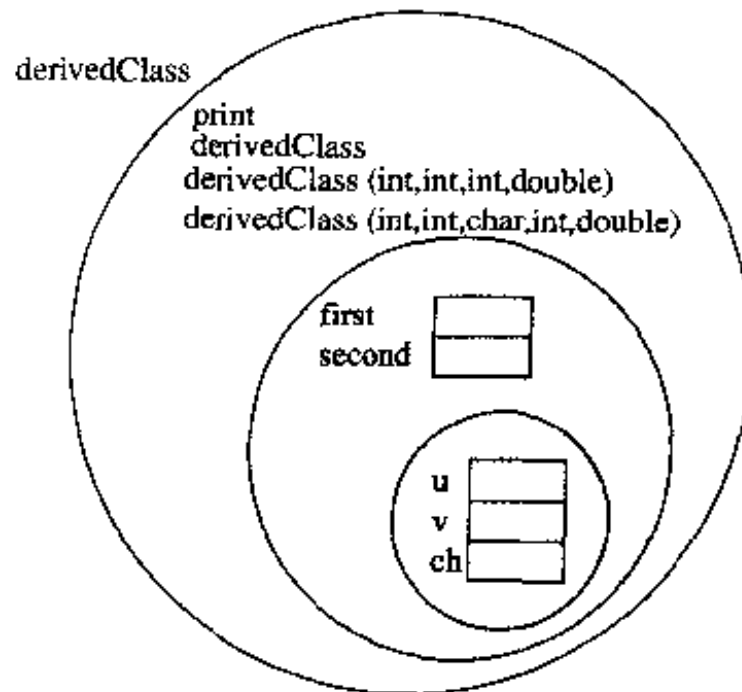


图14.7 带构造函数的derivedClass类

让我们先写出 derivedClass 类中成员函数 print() 的定义。

```

void derivedClass::print()
{
 baseClass::print();
 cout<<"Derived Class: first = "<<first
 <<" , second = "<<second<<endl;
 cout<<"_____ "
 <<"_____ "<<endl;
}

```

现在写出 derivedClass 构造函数的定义。注意，对基类构造函数的调用是在派生类构造函数的头部指定的。

首先，写出 derivedClass 默认构造函数的定义。如果一个类包含默认构造函数且在对象声明时不指定值，默认构造函数将执行和初始化对象。因为 baseClass 类包含默认构造函数，所以当编写 derivedClass 默认构造函数的定义时，不指定基类的构造函数。

```

derivedClass::derivedClass() //default constructor
{
 first = 0;
 second = 0;
}

```

下面，讨论带参数的构造函数的定义。为了引发（带参数的）基类构造函数的执行，必须在派生类构造函数的头部指定基类构造函数的名字和参数。

考虑如下 derivedClass 类构造函数的定义：



```

 return 0;
 }

```

### 输出

```

Line 1: baseObject1: Base Class: u = 0, v = 0, ch = *
Line 2: baseObject2: Base Class: u = 5, v = 6, ch = *
Line 3: baseObject3: Base Class: u = 4, v = 8, ch = K
Line 4: -*
Line 5: Derived Class Objects.
Line 6: derivedObject1:
Base Class: u = 0, v = 0, ch = *
Derived Class: first = 0, second = 0

Line 7: derivedObject2:
Base Class: u = 12, v = 13, ch = *
Derived Class: first = 45, second = 8.5

Line 8: derivedObject3:
Base Class: u = 21, v = 23, ch = Y
Derived Class: first = 76, second = 64.58

```

强烈建议读者实践一下上面的程序。

**注意:** 假设一个基类 `baseClass` 包含私有成员变量和构造函数。`derivedClass` 类派生于 `baseClass` 并且没有成员变量。那么, `derivedClass` 的成员变量都是继承自 `baseClass`。构造函数不能像其他函数一样调用, 而且 `baseClass` 的成员变量不能由 `derivedClass` 的成员函数直接访问。为了保证 `derivedClass` 类型对象成员变量的初始化, 尽管 `derivedClass` 没有成员变量, 它必须有适当的构造函数。`derivedClass` 的构造函数 (带参数) 仅仅是调用 `baseClass` 的构造函数 (带参数)。所以, 定义 `derivedClass` 的构造函数 (带参数) 时, 构造函数定义的头部包含对适当的 `derivedClass` 构造函数 (带参数) 的调用, 而且构造函数的函数体是空的。

### 例 14.1

假设要定义一个类来记录雇员的各方面信息, 雇员包括全职雇员和兼职雇员。兼职雇员按其工时和时薪计算薪水。假设要定义一个类来记录兼职雇员的信息, 例如姓名、薪水和工时。那么就可以打印雇员的姓名及其薪水。因为每位雇员都是一个人, 而例 13.9 (第 13 章) 定义了 `personType` 类存储姓和名以及对姓名的必要操作, 可以基于 `personType` 类定义一个 `partTimeEmployee` 类。(参见图 14.8) 同样也可以重定义 `print()` 函数以打印适当的信息。

```

class partTimeEmployee: public personType
{
public:
 void print();
 //Function to output the first name, last name, and
 //the wages in the form:
 //firstName lastName wages are $$$$.$$
 double calculatePay();
 //Function to calculate and return the wages
 void setNameRateHours(string first, string last,
 double rate, double hours);
 //Function to set the first name, last name, payRate,
 //and hoursWorked according to the parameters.
 //The parameters first and last are passed to the
 //base class. payRate = rate; hoursWorked = hours;

```

```

partTimeEmployee(string first, string last,
 double rate, double hours);
//Constructor with parameters
//Set the first name, last name, payRate, and
//hoursWorked according to the parameters.
//Parameters first and last are passed to the
//base class. payRate = rate; hoursWorked = hours;

partTimeEmployee();
//default constructor
//Set the first name, last name, payRate, and
//hoursWorked to the default values.
//The first name and last name are initialized to an empty
//string by the default constructor of the base class.
//payRate = 0; hoursWorked = 0;

private:
 double payRate; //store the pay rate
 double hoursWorked; //store the hours worked
};

```

partTimeEmployee类包含两个构造函数：默认构造函数和带参数的构造函数。这些构造函数都可以用一个带默认参数的构造函数来代替，具体细节留做读者练习。

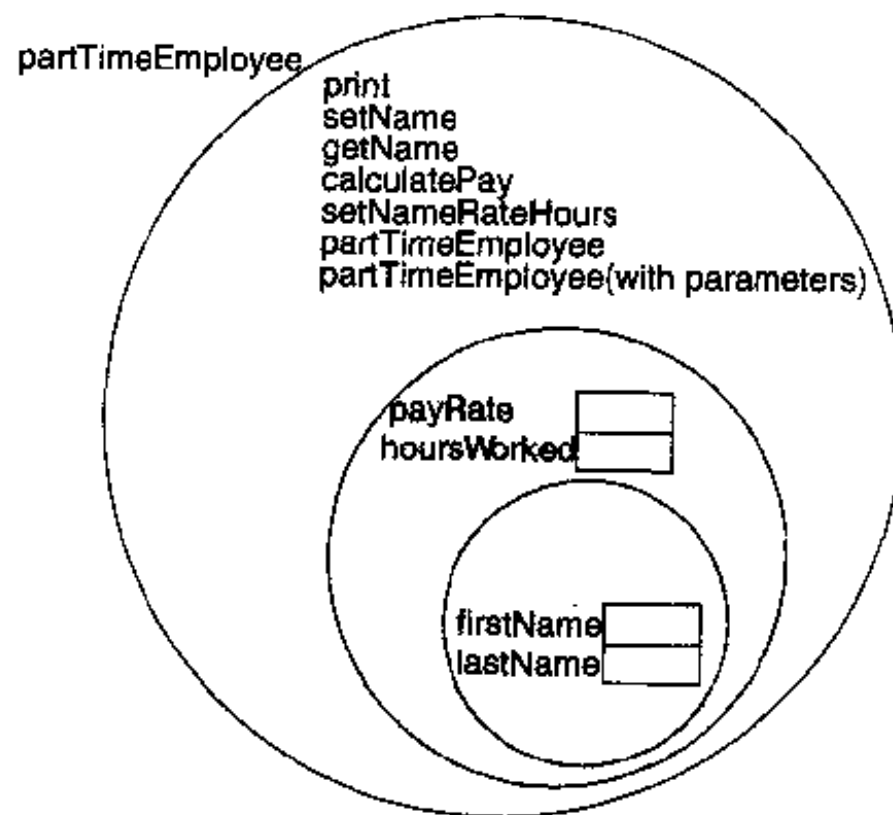


图 14.8 partTimeEmployee 类

partTimeEmployee 类成员函数的定义如下：

```

void partTimeEmployee::print()
{
 personType::print(); //print the name of the employee
 cout<<" wages are : "<<calculatePay()<<endl;
}

double partTimeEmployee::calculatePay()
{
 return (payRate * hoursWorked);
}

void partTimeEmployee::setNameRateHours(string first,
 string last, double rate, double hours)
{

```

```

 personType::setName(first, last);
 payRate = rate;
 hoursWorked = hours;
 }

 //constructor with parameters
 partTimeEmployee::partTimeEmployee(string first, string last,
 double rate, double hours)
 : personType(first, last)
 {
 payRate = rate;
 hoursWorked = hours;
 }

```

在带参数的构造函数定义中，函数头部包含了对基类带参数构造函数的调用。

```

partTimeEmployee::partTimeEmployee() //default constructor
{
 payRate = 0;
 hoursWorked = 0;
}

```

在默认构造函数的定义中，因为函数头部不包含对基类构造函数的调用，基类的默认构造函数将被执行。

### 14.1.3 派生类的头文件

上一节解释了如何由已定义类派生出新的类。为了定义新的类，必须创建新的头文件。基类已经被声明，头文件中包含了它们的声明。那么，为了从已定义类中创建新的类，新类的头文件必须包含相应的命令，告诉计算机何处寻找基类的定义。

假设 personType 类的声明置于头文件 person.h 中。为了创建 partTimeEmployee 类的定义，头文件，假如名为 ptEmployee.h，必须在 partTimeEmployee 类的定义前包含预处理指令：

```
#include "person.h"
```

具体而言，头文件 ptEmployee.h 如下：

```

//Header file ptEmployee.h
#include "person.h"

class partTimeEmployee: public personType
{
public:
 void print() const;
 double calculatePay();
 void setNameRateHours(string, string, double, double);
 partTimeEmployee(string, string, double, double);
 partTimeEmployee(); //default constructor

private:
 double payRate;
 double hoursWorked;
};

```

成员函数的定义可以放在单独的文件中。如果在用户程序中包括系统头文件，如 iostream，则用尖括号把头文件括起来；如果在程序中包括用户定义的头文件，则用双引号把头文件括起来。

### 14.1.4 头文件的多重包括

上一节讨论了如何创建派生类的头文件。要在程序中包括头文件，需要使用预处理命令。在程



序被编译之前，首先由预处理器处理程序。考虑如下头文件：

```
//Header file test.h
const int One = 1;
const int Two = 2;
```

假设为了使用标志符 One 和 Two，头文件 testA.h 包括了文件 test.h。具体而言，假设头文件 testA.h 如下所示：

```
//Header file testA.h
#include "test.h"
.
.
.
```

现在考虑下面的程序代码：

```
//Program headerTest.cpp
#include "test.h"
#include "testA.h"
.
.
.
```

当程序 headerTest.cpp 编译时，它首先经过预处理器处理。预处理器首先包括了头文件 test.h，然后包括了头文件 testA.h。当头文件 testA.h 被包括时，因为包含预处理器指令 #include "test.h"，头文件 test.h 在程序中被包括了两次。第二次包括头文件 test.h 导致编译时错误，如标志符 One 已经声明。这一问题的产生是因为头文件 test.h 的第一次包括时已经定义了变量 One 和 Two。为了避免在程序中对文件的多重包括，需要在头文件中使用特定的预处理器命令。下面，首先使用这些预处理器命令重写头文件 test.h，然后解释这些命令的含义。

```
//Header file test.h
#ifndef H_test
#define H_test
const int One = 1;
const int Two = 2;
#endif
```

- a. #ifndef H\_test 含义为“如果没有定义 H\_test”
- b. #define H\_test 含义为“定义 H\_test”
- c. #endif 含义为“end if”

这里，H\_test 是预处理标志符。

这些命令的效果为：如果标志符 H\_test 没有定义，必须定义标志符 H\_test，让 #define 和 #endif 之间的语句通过编译器进行编译。如果头文件 test.h 在程序中被第二次包括，#ifndef 语句就不成立，#endif 前的语句都将被忽略掉。实际上，所有的头文件都以相似的预处理命令写成。

### 14.1.5 C++ 流类

在第3章中详细描述了如何使用标准 I/O 设备和文件 I/O 进行输入/输出 (I/O)。即如何使用 cin 对象、析取运算符 >>、get() 及 ignore() 等函数从标准输入设备中读取数据；以及使用 cout 对象和插入运算符 << 发送输出到标准输出设备。使用 cin 和 cout 时，程序要包含 iostream 头文件，它包括了 istream 类和 ostream 类的定义。此外，对于文件 I/O，程序要包含 ifstream 头文件，使用 ifstream 类型

的对象进行文件输入,使用 `ofstream` 类型的对象进行文件输出。本节简要描述流类的相关性及其在 C++ 中的实现。

在 C++ 中,流类是使用继承机制实现的,如图 14.9 所示。

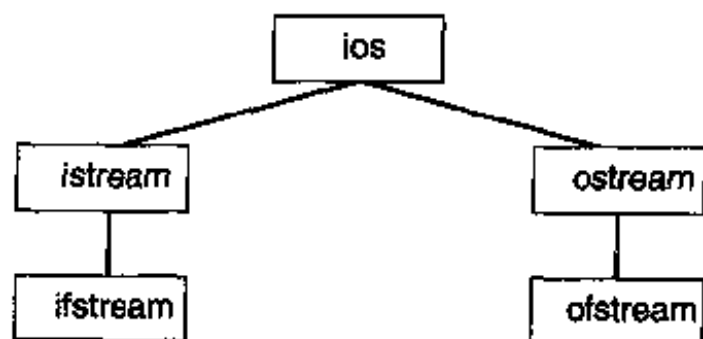


图 14.9 C++ 流类层次体系

图 14.9 展现了在前面章节中所遇到的流类。从图 14.9 可以看出, `ios` 类是所有流类的基类。 `istream` 类和 `ostream` 类直接派生于 `ios` 类。 `ifstream` 类派生于 `istream` 类, `ofstream` 类派生于 `ostream` 类。此外,通过使用多重继承机制, `iostream` 类(不要和头文件 `iostream` 混淆,它们是不同的概念)派生于 `istream` 类及 `ostream` 类,而 `fstream` 类派生于 `iostream` 类( `iostream` 类和 `fstream` 类不是本书讨论内容)。

`ios` 类包含格式化标志以及访问和/或修改这些标志设置的成员函数。为了确定 I/O 状态, `ios` 类包含了一个整型状态字。该整型状态字不断更新并报告流的状态。

`istream` 类和 `ostream` 类负责提供用于内存和设备间数据传输的操作。 `istream` 类定义了析取运算符 `>>`、 `get()` 和 `ignore()` 等函数。 `ostream` 类定义了用于 `cout` 对象的插入运算符 `<<`。

`ifstream` 类派生于 `istream` 类并提供文件输入操作。与之类似, `ofstream` 类派生于 `ostream` 类并提供文件输出操作。 `ifstream` 类型的对象用于文件输入; `ofstream` 类型的对象用于文件输出。头文件 `fstream` 包含 `ifstream` 类和 `ofstream` 类的定义。

### 14.1.6 类的保护成员

类的私有成员对于类来说是私有的,不能在类之外直接访问。只有该类的成员函数可以访问私有成员。如前所述,派生类不能访问基类的私有成员。然而,有时派生类需要访问基类的私有成员。如果让一个私有的成员变为公有,那么该成员就可以被随意地访问。类的成员分为三种:公有、私有和保护。所以,基类给予其派生类对私有成员访问权时还需要防止类外对该成员的直接访问,可以使用成员访问描述符 `protected` 声明该成员。因此,类中保护成员的访问性介于公有和私有之间。派生类可以直接访问基类的保护成员。

总而言之,如果基类的成员需要被派生类访问,该成员需要使用成员访问描述符 `protected` 加以声明。

### 14.1.7 继承为公有、保护或私有

假设类 B 派生于类 A。那么 B 不能直接访问 A 的私有成员。也就是说, A 的私有成员在 B 中是隐藏的。A 的公有及保护成员又如何呢? 本节给出访问基类成员时适用的一些规则。

考虑下面的语句:

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



在该语句中, 成员访问描述符为 public、protected 或 private。

1. 如果成员访问描述符为 public:

- a. A 的公有成员是 B 的公有成员, 可以在 B 类中直接访问。
- b. A 的保护成员是 B 的保护成员, 可以被 B 的成员函数 (和友元函数) 直接访问。
- c. A 的私有成员在 B 中是隐藏的。B 的成员函数 (和友元函数) 可以通过 A 的公有或保护成员访问其私有成员。

2. 如果成员访问描述符为 protected:

- a. A 的公有成员是 B 的保护成员, 可以被 B 的成员函数 (和友元函数) 访问。
- b. A 的保护成员是 B 的保护成员, 可以被 B 的成员函数 (和友元函数) 访问。
- c. A 的私有成员在 B 中是隐藏的。B 的成员函数 (和友元函数) 可以通过 A 的公有或保护成员访问其私有成员。

3. 如果成员访问描述符为 private:

- a. A 的公有成员是 B 的私有成员, 可以被 B 的成员函数 (和友元函数) 访问。
- b. A 的保护成员是 B 的私有成员, 可以被 B 的成员函数 (和友元函数) 访问。
- c. A 的私有成员在 B 中是隐藏的。B 的成员函数 (和友元函数) 可以通过 A 的公有或保护成员访问其私有成员。

注意: 在第 16 章中将描述友元函数。

例 14.2 阐明了派生类的成员函数如何直接访问基类的保护成员。

例 14.2 在派生类中访问保护成员。

考虑下面 bClass 类的定义:

```
class bClass
{
public:
 void setData(double);
 void setData(char, double);
 void print() const;
 bClass(char = '*', double = 0.0);
protected:
 char bCh;
private:
 double bX;
};
```

bClass 类的定义包含了一个 char 类型的保护成员变量 bCh 和一个 double 类型的私有成员变量 bX。它还包含一个重载的成员函数 setData(), 该函数的一个版本用来设置两个成员变量, 另一版本仅用来设置其中的私有成员变量。此外, bClass 类有一个带默认参数的构造函数。假设成员函数和构造函数的定义如下:

```
void bClass::setData(double u)
{
 bX = u;
}

void bClass::setData(char ch, double u)
{
 bCh = ch;
```

```

 bX = u;
 }

 void bClass::print() const
 {
 cout<<"Base class: bCh = "<<bCh<<"", bX = "<<bX<<endl;
 }

 bClass::bClass(char ch, double u)
 {
 bCh = ch;
 bX = u;
 }

```

下面, 使用公有继承从 bClass 类派生出 dClass 类, 如下所示:

```

class dClass: public bClass
{
public:
 void setData(char, double, int);
 void print() const;

private:
 int dA;
};

```

dClass 类包含一个 int 型的私有成员变量 dA, 还包含一个带三个参数的成员函数 setData() 以及函数 print()。

现在先写出 setData() 函数的定义。bCh 是 bClass 类的保护成员变量, 因此它可以在 setData() 函数中直接访问。然而, 因为 bX 是 bClass 类的私有成员变量, setData() 函数不能直接访问。所以, dClass 类的 setData() 函数必须通过 bClass 类的 setData() 函数设置 bX。dClass 类的 setData() 函数可以如下所示:

```

void dClass::setData(char ch, double v, int a)
{
 bClass::setData(v);

 bCh = ch; //initialize bCh using the assignment
 //statement
 dA = a;
}

```

注意到 setData() 函数的定义调用了带一个参数的函数 bClass::setData() 来设置 bX 成员变量, 然后直接设置 bCh 的值。下面, 让我们写出 (dClass 类的) print() 函数的定义。

bClass 类仅有一个函数 print() (它没有像 setData() 函数一样被重载), 而我们不想打印 bCh 的值两次。所以, 首先调用 (bClass 类的) print() 函数, 然后仅输出 dA 的值。print() 函数的定义为:

```

void dClass::print() const
{
 bClass::print();

 cout<<"Derived class dA = "<<dA<<endl;
}

```

下面的程序阐明 bClass 和 dClass 类的对象如何工作。假定 bClass 类的定义在头文件 protectMembClass.h 中, dClass 类的定义在头文件 protectMembInDerivedCl.h 中。

```

//Accessing protected members of a base class
//in the derived class

```

```

#include <iostream>
#include "protectMembClass.h"
#include "protectMembInDerivedCl.h"

using namespace std;
int main()
{
 bClass bObject; //Line 1
 dClass dObject; //Line 2
 bObject.print(); //Line 3
 cout<<endl; //Line 4
 cout<<"*** Derived class object ***"<<endl; //Line 5
 dObject.setData('&', 2.5, 7); //Line 6
 dObject.print(); //Line 7
 return 0;
}

```

### 输出

```

Base class: bCh = *, bX = 0
*** Derived class object ***
Base class: bCh = &, bX = 2.5
Derived class dA = 7

```

编写 dClass 类的成员函数定义时, 保护成员变量 bCh 可以直接访问。然而, dClass 类的对象不能直接访问 bCh。就是说, 下面的语句是非法的 (语法错误):

```
dObject.bCh = '&'; //illegal
```

## 14.2 组合

组合是把两个类联系起来的另一种方法。在组合中, 类的一个或多个成员是另一个类的对象。组合为 “has-a” 关系, 例如, “每个人都有生日”。

第13章中的例13.9定义了一个 personType 类。personType 类存储一个人的姓和名。假如想要记录一个人的额外信息, 如个人 ID (社会保险号等) 和生日。因为每个人都有其 ID 和生日, 可以定义一个新的类, 称为 personallInfo, 该类中一个成员为 personType 类型的对象。可以为 personallInfo 声明额外的成员来存储个人 ID 和生日。

首先定义另一个类 dateType, 仅仅用来存储一个人的生日。然后, 由 personType 类和 dateType 类构造出 personallInfo 类。通过这一方法, 可以展示如何使用两个类定义一个新的类。

为了定义 dateType 类, 需要有三个成员变量存储月、日和年。作用于日期上的一些操作为设置日期和打印日期。下面的语句定义了 dateType 类 (参见图 14.10):

```

class dateType
{
public:
 void setDate(int month, int day, int year);
}

```

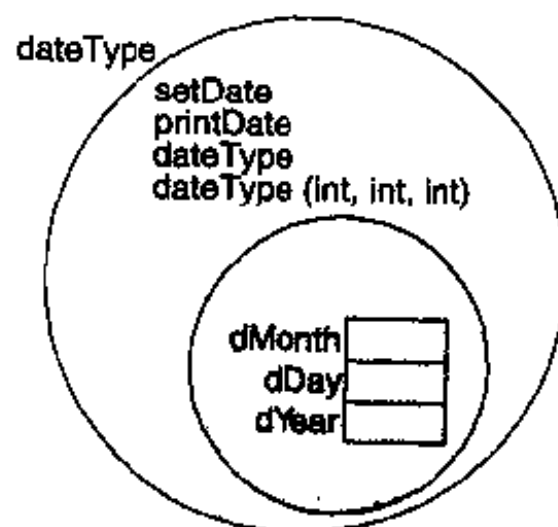


图14.10 dateType类

```

 //Function to set the date
 //Data members dMonth, dDay, and dYear are set
 //according to the parameters
 //Post: dMonth = month; dDay = day;
 // dYear = year;

void getDate(int& month, int& day, int& year);
 //Function to return the date
 //Post: month = dMonth; day = dDay;
 // year = dYear;

void printDate() const;
 //Function to output the date in the form mm-dd-yyyy

dateType(int month, int day, int year);
 //constructor to set the date
 //Data members dMonth, dDay, and dYear are set
 //according to the parameters
 //Post: dMonth = month; dDay = day;
 // dYear = year;

dateType();
 //default constructor
 //Data members dMonth, dDay, and dYear are set to
 //the default values
 //Post: dMonth = 1; dDay = 1; dYear = 1900;

private:
 int dMonth; //variable to store the month
 int dDay; //variable to store the day
 int dYear; //variable to store the year
};

```

dateType类的定义由两个构造函数构成：默认构造函数和带参数的构造函数。这两个构造函数都可以由一个带默认参数的构造函数代替。在本章的编程练习2中，将要求读者完成这一任务。

dateType类的成员函数定义如下：

```

void dateType::setDate(int month, int day, int year)
{
 dMonth = month;
 dDay = day;
 dYear = year;
}

```

在把日期存储到成员变量之前，setDate()函数的定义并不检查该日期是否有效。就是说，它不确认month是否在1和12之间，year是否大于0以及day是否有效（例如，对于1月而言，day应该在1和31之间）。在本章的编程练习2中，将要求读者重写setDate()函数的定义，以确保日期在存储到成员变量之前是有效的。其他成员函数的定义如下：

```

void dateType::getDate(int& month, int& day, int& year)
{
 month = dMonth;
 day = dDay;
 year = dYear;
}

void dateType::printDate() const
{
 cout<<dMonth<<"-"<<dDay<<"-"<<dYear;
}

```

```

 //constructor with parameter
dateType:: dateType(int month, int day, int year)
{
 dMonth = month;
 dDay = day;
 dYear = year;
}

```

和setDate()函数一样,在编程练习2中,将要求读者重写构造函数的定义,在把month、day和year存储到成员变量之前应确保它们的有效性。

```

dateType:: dateType() //default parameter
{
 dMonth = 1;
 dDay = 1;
 dYear = 1900;
}

```

下面给出personalInfo类的定义(参见图14.11):

```

class personalInfo
{
public:
 void setpersonalInfo(string first, string last, int month,
 int day, int year, int ID);
 //Function to set the personal information
 //Data members are set according to the parameters
 //Post: firstName = first; lastName = last;
 // dMonth = month; dDay = day; dYear = year;
 //personID = ID;

 void printpersonalInfo () const;
 //Function to print personal information

 personalInfo(string first, string last, int month,
 int day, int year, int ID);
 //constructor with parameters
 //Data members are set according to the parameters
 //Post: firstName = first; lastName = last;
 //dMonth = month; dDay = day; dYear = year;
 //personID = ID;
 personalInfo();
 //default constructor
 //Data members are set according to the default values
private:
 personType name;
 dateType bDay;
 int personID;
};

```

在给出personalInfo类成员函数的定义之前,先讨论bDay和name对象的构造函数是如何被调用的。

当类对象进入其作用域时,类的构造函数是自动执行的。假设有如下的语句:

```
personalInfo student;
```

当student对象进入它的作用域,其成员bDay对象和name对象也进入了它们的作用域;这将导致它们的构造函数之一被执行。因此,当给出类构造函数的定义时,需要知道如何传递参数给成员

对象（就是 bDay 和 name）的构造函数。构造函数没有类型，不能像其他函数一样调用。成员对象（如 bDay）构造函数的参数是在类构造函数定义的头部指定的。此外，类的成员对象是按照它们声明的次序（不是它们在构造函数的成员初始化列表中的排列次序）构造（初始化）的，而且是在包含它们的类对象被构造之前。所以，在该例子中，name 对象首先被初始化，然后是 bDay，最后是 student。

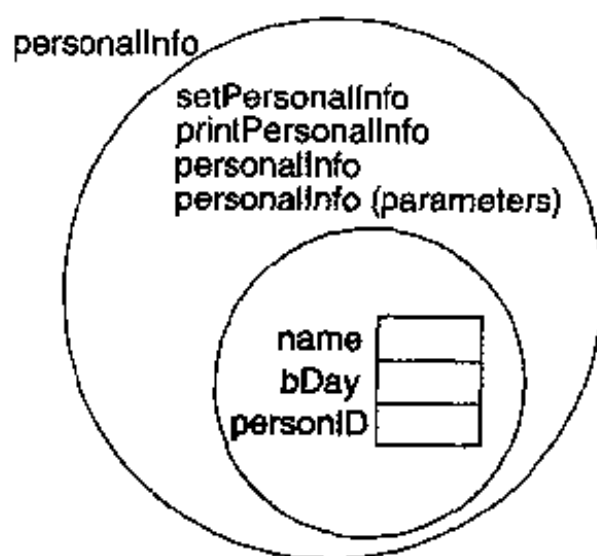


图14.11 personalInfo类

下而的语句阐述了如何传递参数给成员对象的构造函数：

```
personalInfo::personalInfo(string first, string last, int month,
 int day, int year, int ID)
 :name(first,last), bDay(month,day,year)
{
 .
 .
 .
}
```

personalInfo 类成员函数的定义如下：

```
void personalInfo::setpersonalInfo(string first, string last,
 int month, int day, int year, int ID)
{
 name.setName(first,last);
 bDay.setDate(month,day,year);
 personID = ID;
}

void personalInfo::printpersonalInfo () const
{
 name.print();
 cout<<"'s date of birth is ";
 bDay.printDate();
 cout<<endl;
 cout<<"and personal ID is "<<personID;
}

personalInfo::personalInfo(string first, string last, int month,
 int day, int year, int ID)
 :name(first,last), bDay(month,day,year)
{
 personID = ID;
}

personalInfo::personalInfo() //default constructor
```



```
{
 personID = 0;
}
```

因为在默认构造函数的头部没有参数传递给成员对象，bDay 和 name 对象的默认构造函数被执行。

**注意：**在继承的情况下，使用类名调用基类的构造函数。在组合的情况下，使用成员对象名调用它自己的构造函数。

### 14.3 面向对象设计 (OOD) 和面向对象编程 (OOP)

本书的前 11 章使用自上而下的方法进行编程，也称为结构化编程。问题分割为模块，每个模块解决问题的特定部分——先确定数据需求，然后写出操作这些数据的函数。函数和数据是分离的，函数以一种被动的方式作用于数据。因此，结构化编程有特定的局限性。在结构化编程中，函数依赖于数据，而且函数是专门设计成用来解决某个特定问题的。在一个程序中重用为另一个程序而写的函数是非常困难的，甚至是不可能的。因为诸如此类的原因，结构化编程对于大型软件开发来说是低效的。

第 13 章开始介绍了类的基本概念，然后学习了类是如何定义和使用的。随后讨论了数据需求和对数据的逻辑操作。通过类，可以把数据及作用于数据之上的操作结合于一个单一单元中，数据和操作被封装在一个单一单元中。同样通过类，能够把数据和操作数据的算法分离。不过，实现数据操作的函数具有对数据的直接访问权。本章解释了如何通过继承（还有组合）由已有的类创建新的类。此外，对象还有隐藏信息细节的能力。这些都是面向对象设计 (OOD) 的一些特征。

OOD 的三个基本原则是：

- **封装** 把数据及作用于数据之上的操作结合于单一单元中的能力。
- **继承** 由已有对象创建新对象的能力。
- **多态** 使用相同的表达式来代表不同操作的能力。

在 OOD 中，对象是基本的实体；在结构化编程中，函数是基本的实体。在 OOD 中，我们调试对象；在结构化编程中，我们调试函数。在 OOD 中，程序是相互作用的对象集合；在结构化编程中，程序是相互作用的函数集合。此外，OOD 有利于代码重用。如果一个对象没有错误，它可以在许多程序中被重用，因为它是自包含的实体。面向对象编程 (OOP) 实现了 OOD。

为了创建对象，需要知道如何表示数据和操作数据的函数。前 10 章对于任何类型的编程，无论是结构化还是面向对象，都是必需的。

C++ 通过使用类来支持 OOP。读者已经在本章和第 13 章中体验了 OOP 的前两个特征：封装和继承。在第 16 章中将讨论 OOD 的第三个特征：多态。多态的函数或运算符有多种不同的形式。

在 C++ 中，函数名和运算符可以被重载。当函数调用或运算符求值依赖于所使用的参数进行时，重载就发生了。例如，如果操作数都是整数，除法运算符给出整型结果。否则，除法运算符给出小数结果。假如类有构造函数，如果当对象声明时没有参数传递给它，默认构造函数被执行。否则，带参数的构造函数之一被执行。不过，所有的构造函数都有相同的名字。

C++ 还提供基于参数的多态性。在基于参数的多态性中，（数据）类型在没有明确的情况下被实例化。模板（在第 16 章讨论）提供基于参数的多态性。此外，C++ 提供虚函数作为在继承层次体系中实现多态的手段，该方法允许在运行时选择适当的成员函数。（在第 15 章中讨论虚函数。）

有多种 OOP 语言存在, 包括 Ada、Modula-2、Object Pascal、Turbo Pascal、Eiffel、C++、Java 和 Smalltalk。最早的 OOP 语言为 Simula, 于 1967 年开发。OOP 的术语主要来自于 Smalltalk——施乐研究中心于 20 世纪 70 年代开发的 OOP 语言。OOP 语言使用许多“别致的”词汇, 如方法、消息传递, 等等。

OOP 是编程过程的自然和直观的方法。当观察一个对象时, 人们会立即想起它能做什么。例如, 当想起汽车时, 还想起了作用于汽车之上的操作, 如启动汽车和驾驶汽车。当程序员想起一个链表时, 他们还想起了作用于链表之上的操作, 如查找、排序和插入。OOP 允许 ADT 的创建和使用。在 C++ 中, 通过使用类来实现 ADT。

当声明类变量时, 对象被创建。对象通过函数调用彼此交互。每个对象都有内部状态和外部状态。私有成员构成了内部状态; 公有成员构成了外部状态。只有对象可以操作它自己的内部状态。

### 14.3.1 确定类、对象和操作

在本书的前 12 章中, 在问题分析阶段我们分析了问题, 确定了数据以及概述了算法。为了减小 main() 函数的复杂性, 曾编写了操作数据的函数。在第 13 章中, 使用 OOD 技术首先确定了构成整个问题的对象。对象独立于主程序进行设计和实现。OOD 最困难的部分是确定类和对象。本节将描述一个通用和简单的确定类和对象的技术。

可以首先描述问题, 确定所有的名词和动词。由名词表选择类, 由动词表选择操作。

例如, 假设需要编写一个计算和打印圆柱体表面积和体积的程序。可以把问题陈述如下: 写一个程序, 用来输入圆柱体的尺寸以及计算和打印表面积和体积。

在该语句中, 名词表: 程序、尺寸、圆柱体、表面积和体积。可以很容易把圆柱体设想为一个类, 假设名为 cylinderType, 由该类可以创建许多不同尺寸的圆柱体对象。名词尺寸、表面积和体积是一个圆柱体的属性而不是类。

在确定了类之后, 下一步是确定三个信息:

- 可以对该类的对象执行的操作
- 可以作用于该类型对象之上的操作
- 该类型对象必须维护的信息

由问题描述中确定的动词表, 可以选择一个可能的操作名单, 这些操作是类对象可以执行或已经执行于自身的。例如, 由圆柱体问题描述的动词表——写、输入、计算和打印——对圆柱体对象而言可能的操作是输入、计算和打印。

对于 cylinderType 类, 尺寸代表了数据。底面的中心、底面的半径及圆柱体的高是尺寸特征。可以通过构造函数或其他函数给对象输入数据。动词“计算”用于确定体积和表面积。由此可以推断操作为: cylinderVolume() 和 cylinderSurfaceArea()。与之类似, 动词“打印”用于在输出设备上显示体积和表面积。在本章的编程练习 5 中, 将要求读者设计一个类以实现圆柱体的特征。

通过来自问题描述的名词和动词确定类不是惟一可行的技术。相关著作描述了多种其他 OOD 技术。然而, 该技术对于本书的编程练习而言足够了。

## 14.4 编程实例: 分数报告

该编程实例进一步阐明了继承和组合的概念。

一旦学生的分数被记录下来, 注册办公室就要尽快准备分数报告。但是, 也许一些注册的学生还没有缴纳他们的学费。



1. 如果一名学生已经缴纳了学费，分数报告上包括各科分数及平均成绩。
2. 如果一名学生没有缴纳学费，将不打印分数。对于这些学生，分数报告上应包含一条信息，说明分数由于没有缴纳学费而被保留。分数报告还应显示学费清单。

注册办公室和业务办公室需要编写一个程序来分析学生的数据和打印适当的分数报告。数据以下面的格式存储在一个文件里：

```
15000 345
studentName studentID isTuitionPaid numberOfCourses
courseName courseNumber creditHours grade
courseName courseNumber creditHours grade
.
.
.
studentName studentID isTuitionPaid numberOfCourses
courseName courseNumber creditHours grade
courseName courseNumber creditHours grade
.
.
.
```

第1行说明注册学生的数量和每个学分的学费。学生的数据随后给出。  
示例输入文件如下：

```
3 345
Lisa Miller 890238 Y 4
Mathematics MTH345 4 A
Physics PHY357 3 B
ComputerSci CSC478 3 B
History HIS356 3 A
.
.
.
```

第1行说明有3名学生注册，学费为每学分345美元。下面给出学生 Lisa Miller 的课程数据：Lisa Miller 的ID是890238、已缴纳学费、选修4门课。她所选修数学课的课程号是MTH345，该课程有4个学分，期中考试分数为A，等等。

每名学生的输出为如下格式：

```
Student Name: Lisa Miller
Student ID: 890238
Number of courses enrolled: 4

Course No Course Name Credits Grade
CSC478 ComputerSci 3 B
HIS356 History 3 A
MTH345 Mathematics 4 A
PHY357 Physics 3 B

Total number of credits: 13
Mid-Semester GPA: 3.54
```

从该输出可知，课程必须按照课程号进行排序。为了计算GPA，假设分数A等于4分，B等于3分，C等于2分，D等于1分，F等于0分。

**输入** 包含前述格式数据的文件。假定输入文件的名字为“a:stData.txt”。

**输出** 包含前述格式数据的文件。

### 问题分析及算法设计

首先确定程序的主要组成部分。大学有学生,而每名学生需要选修课程。因此,两个主要的组成部分为学生和课程。

**课程** 课程的主要特征有课程名、课程号和学分。尽管学生的分数并不是真正的课程属性,但是为了简化程序,该组成部分包括了学生分数。

需要在课程类型的对象上执行的一些基本操作如下:

1. 设置课程信息。
2. 打印课程信息。
3. 显示学分。
4. 显示课程号。
5. 显示分数。

下面的类把课程定义为一个 ADT (参见图 14.12):

```
class courseType
{
public:
 void setCourseInfo(string cName, string cNo,
 char grade, int credits);
 //Function to set the course information
 //The course information is set according to the
 //incoming parameters.
 //Post: courseName = cName; courseNo = cNo;
 // courseGrade = grade; courseCredits = credits;

 void print(bool isGrade);
 //Function to print the course information
 //This function prints the course information on the
 //screen. Furthermore, if the bool parameter isGrade is
 //true, the grade is shown; otherwise, three stars
 //are printed.

 void print(ofstream& outp, bool isGrade);
 //Function to print the course information
 //This function sends the course information to a file.
 //Furthermore, if the bool parameter isGrade is true,
 //the grade is shown; otherwise, three stars are printed.

 int getCredits();
 //Function to return the credit hours
 //The value of the private data member courseCredits
 //is returned.

 void getCourseNumber(string& cNo);
 //Function to return the course number
 //Post: cNo = courseNo;

 char getGrade();
 //Function to return the grade for the course
 //The value of the private data member courseGrade
 //is returned.

 courseType(string cName = "", string cNo = "",
 char grade = '*', int credits = 0);
 //constructor
}
```

```

//The object is initialized according to the
//parameters.
//Post: courseName = cName; courseNo = cNo;
// courseGrade = grade; courseCredits = credits;

private:
 string courseName; //variable to store the course name
 string courseNo; //variable to store the course number
 char courseGrade; //variable to store the grade
 int courseCredits; //variable to store the course credits
};

```

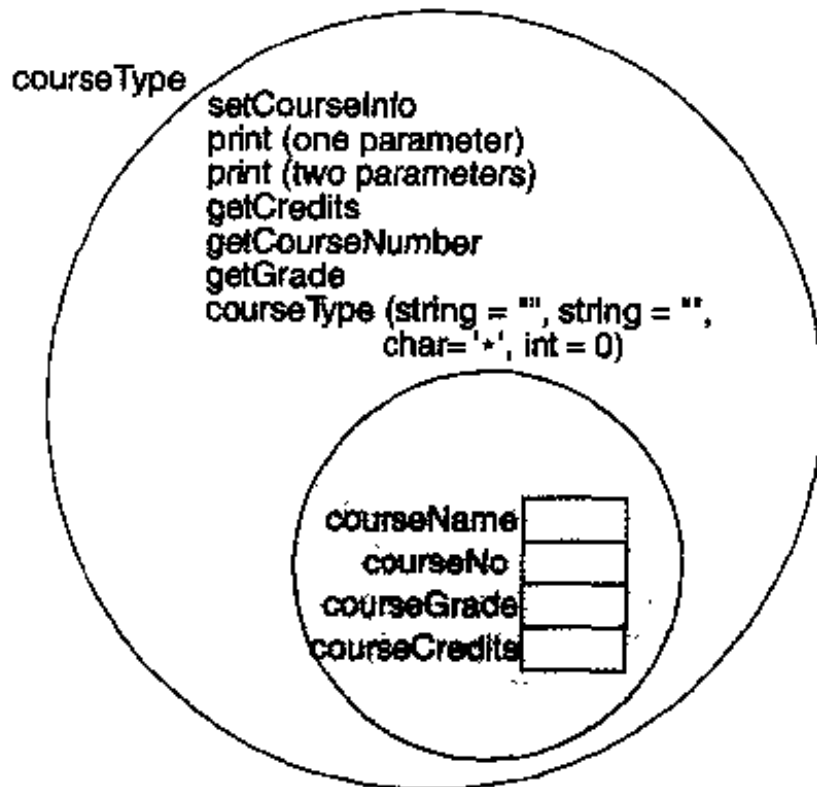


图 14.12 courseType 类

下面，将讨论实现 courseType 类操作的函数定义。这些定义非常直观，易于理解。setCourseInfo() 函数按照参数的值设置私有成员变量的值。其定义为：

```

void courseType::setCourseInfo(string cName, string cNo,
 char grade, int credits)
{
 courseName = cName;
 courseNo = cNo;
 courseGrade = grade;
 courseCredits = credits;
}

```

带一个参数的 print() 函数在屏幕上打印课程信息。如果 bool 型参数 isGrade 为 true，分数就打印在屏幕上；否则，在分数处显示三个星号。此外，以左对齐而不是右对齐（默认）的格式打印课程名和课程号。因此，需要设置左操纵符。在打印分数和学分之前操作符将被清除。下列步骤描述了该函数：

1. 设置左操纵符。
2. 打印课程号。
3. 打印课程名。
4. 清除左操纵符。
5. 打印学分。
6. if isGrade 为 true:  
    输出分数。

else

输出三个星号。

print()函数的定义为:

```
void courseType::print(bool isGrade)
{
 cout<<left; //Step 1
 cout<<setw(8)<<courseNo<<" "; //Step 2
 cout<<setw(15)<<courseName; //Step 3
 cout.unsetf(ios::left); //Step 4
 cout<<setw(3)<<courseCredits<<" "; //Step 5
 if(isGrade) //Step 6
 cout<<setw(4)<<courseGrade<<endl;
 else
 cout<<setw(4)<<"***"<<endl;
}
```

带两个参数的print()函数把课程信息发送到文件。除了把输出发送到以参数指定的文件以外,该函数的定义和前述print()函数的定义是一样的。该函数的定义为:

```
void courseType::print(ofstream& outp, bool isGrade)
{
 outp<<left; //Step 1
 outp<<setw(8)<<courseNo<<" "; //Step 2
 outp<<setw(15)<<courseName; //Step 3
 outp.unsetf(ios::left); //Step 4
 outp<<setw(3)<<courseCredits<<" "; //Step 5
 if(isGrade) //Step 6
 outp<<setw(4)<<courseGrade<<endl;
 else
 outp<<setw(4)<<"***"<<endl;
}
```

构造函数以默认值声明。如果courseType类型的对象声明时没有指定值,构造函数使用默认值初始化对象。使用默认值时,对象的成员变量初始化如下:courseNo为"",courseName为"",courseGrade为'\*, creditHours为0。否则,使用对象声明时的指定值来初始化对象。构造函数的定义为:

```
courseType::courseType(string cName, string cNo,
 char grade, int credits)
{
 courseName = cName;
 courseNo = cNo;
 courseGrade = grade;
 courseCredits = credits;
}
```

其他函数的定义是简单明了的。

```
int courseType::getCredits()
{
 return courseCredits;
}

char courseType::getGrade()
{
 return courseGrade;
}
```

```

 }

 void courseType::getCourseNumber(string& cNo)
 {
 cNo = courseNo;
 }

```

下面讨论程序的另一个组成部分——学生。

**学生** 学生的主要特征为学生姓名、学生 ID、注册的课程数量、注册的课程名以及每门课的分

数。因为每名学生都必须缴纳学费，所以还应有一个成员变量来指明该学生是否已缴纳学费。每名学生都是一个人，每名学生都需要选修课程。前面已经设计了一个 personType 类来处理人的姓和名，还设计了一个类来处理课程信息。因此可知，可以通过 personType 类让 studentType 类记录学生的信息，而该类的一个成员为 courseType 类型。可以根据需要增加更多的成员。

作用于 studentType 类型对象之上的基本操作如下：

1. 设置学生信息。
2. 打印学生信息。
3. 计算所学习课程的学分。
4. 计算 GPA。
5. 计算学费清单。
6. 因为分数报告将以升序打印课程，所以需要将课程按照课程号排序。

下面的类把 studentType 定义为 ADT。假定学生每个学期选修的课程不超过 6 门（参见图 14.13）：

```

class studentType: public personType
{
public:
 void setInfo(string fname, string lname, int ID,
 int nOfCourses, bool isTPaid,
 courseType courses[]);
 //Function to set a student's information
 //The private data members are set according
 //to the parameters.

 void print(double tuitionRate);
 //Function to print a student's grade report

 void print(ofstream& out, double tuitionRate);
 //Function to print a student's grade report
 //The output is stored in a file specified by the
 //parameter out.

 studentType();
 //default constructor
 //The private data members are initialized.

 int getHoursEnrolled();
 //Function to return the credit hours in which a student
 //is enrolled

 double getGpa();
 //Function to return the grade-point average

 double billingAmount(double tuitionRate);
 //Function to return the tuition fees

```

```

private:
 void sortCourses();
 //Function to sort the courses
 //This function sorts the array coursesEnrolled.

 int sId; //variable to store the student ID
 int numberOfCourses; //variable to store the number
 //of courses
 bool isTuitionPaid; //variable to indicate if the tuition
 //is paid
 courseType coursesEnrolled[6]; //array to store the courses
};

```

注意到对数组 `coursesEnrolled` 进行排序的成员函数 `sortCourses()` 为 `studentType` 类的私有成员。这是由于该函数仅用于内部数据操作，类的使用者不需要访问该成员。

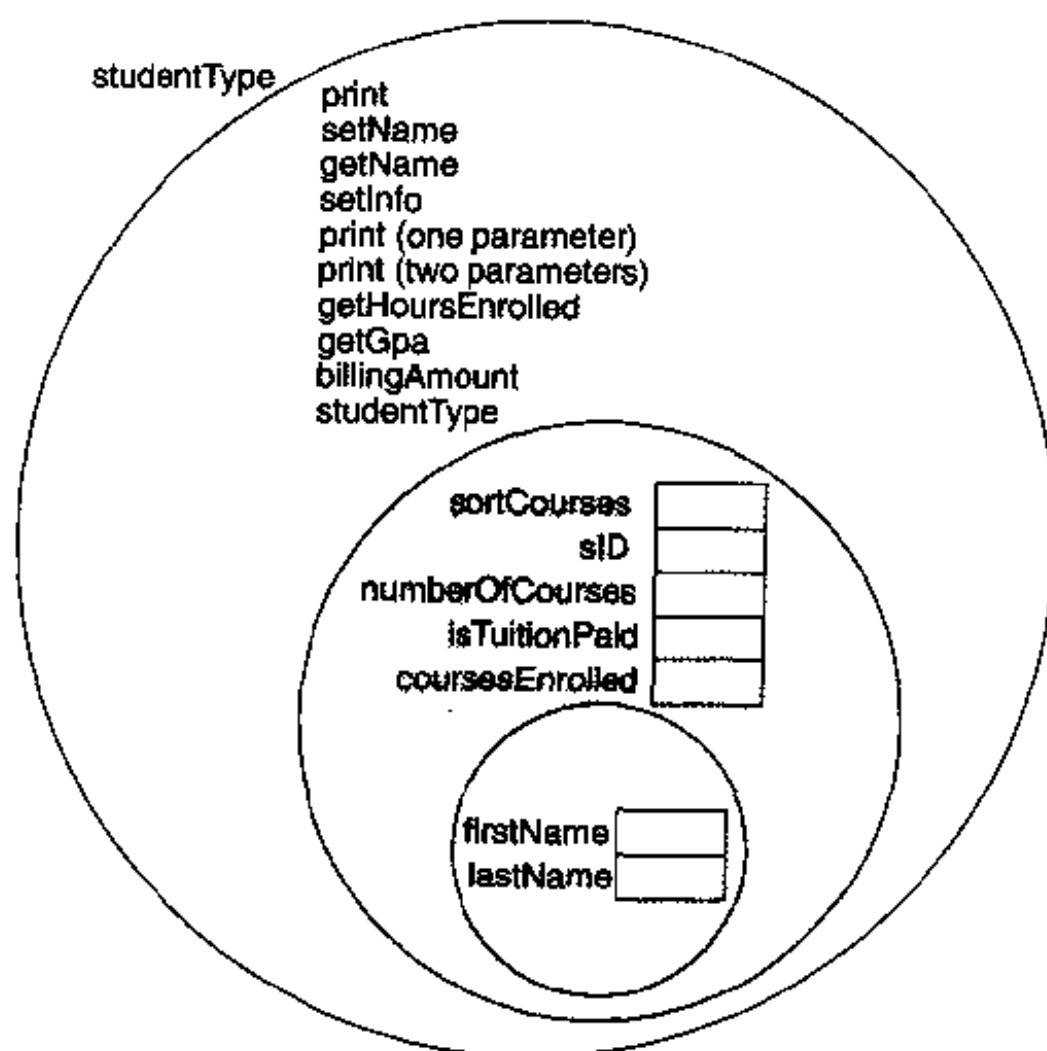


图 14.13 类 `studentType`

下面讨论实现 `studentType` 类操作的函数定义。

`setInfo()` 函数首先按照传入的参数初始化私有成员变量，然后调用 `sortCourses()` 函数对 `coursesEnrolled` 数组按课程号排序。`studentType` 类派生于 `personType` 类，存储姓和名的变量为该类的私有成员变量。因此，可以调用 `personType` 类的成员函数 `setName()` 并传递适当的变量设置姓和名。`setInfo()` 函数的定义如下：

```

void studentType::setInfo(string fName, string lName, int ID,
 int nOfCourses, bool isTPaid,
 courseType courses[])
{
 int i;

 personType::setName(fName, lName); //set the name
 sId = ID; //set the student ID
 isTuitionPaid = isTPaid; //set isTuitionPaid
}

```



```

 numberOfCourses = nOfCourses ; //set the number of courses
 for(i = 0; i < numberOfCourses; i++) //set the array
 coursesEnrolled[i] = courses[i]; //coursesEnrolled
 sortCourses(); //sort the array coursesEnrolled
 }

```

默认构造函数初始化私有成员变量为默认值。注意，因为私有成员变量 `coursesEnrolled` 为 `courseType` 类型而且是一个数组，`courseType` 类的默认构造函数自动执行，整个数组将被初始化。

```

studentType::studentType()
{
 numberOfCourses = 0;
 sId = 0;
 isTuitionPaid = false;
}

```

带一个参数的 `print()` 函数在屏幕上打印分数报告。如果学生已缴纳学费，将显示分数和总平均分。否则，将打印三个星号代替分数并且不显示 GPA。此外，需要显示一条信息，说明分数由于没有缴纳学费而被保留，同时还应显示应付学费。该函数有以下步骤：

1. 输出学生姓名。
2. 输出学生 ID。
3. 输出学生选修的课程数量。
4. 输出标题: CourseNo CourseName Credits Grade
5. 打印每门课的信息。
6. 打印总学分。
7. 设置必需的标志, 用于以带小数点和尾零的固定十进制格式输出 GPA 和学费清单。此外, 设置精度为两个小数位。
8. if `isTuitionPaid` 为 true:
  - 输出 GPA。
- else
  - 输出学费清单及关于保留分数的信息。

```

void studentType::print(double tuitionRate)
{
 int i;

 cout<<"Student Name: "; //Step 1
 personType::print(); //Step 1
 cout<<endl;

 cout<<"Student ID: "<<sId<<endl; //Step 2

 cout<<"Number of courses enrolled: "
 <<numberOfCourses<<endl; //Step 3
 cout<<endl;

 cout<<left; //set output left-justified
 cout<<"Course No"<<setw(15)<<" Course Name"
 <<setw(8)<<"Credits"
 <<setw(6)<<"Grade"<<endl; //Step 4

 cout.unsetf(ios::left);

 for(i = 0; i < numberOfCourses; i++) //Step 5

```

```

 coursesEnrolled[i].print(isTuitionPaid);
 cout<<endl;

 cout<<"Total number of credit hours: "
 <<getHoursEnrolled()<<endl; //Step 6

 cout<<fixed<<showpoint<<setprecision(2); //Step 7

 if(isTuitionPaid) //Step 8
 cout<<"Mid-Semester GPA: "<<getGpa()<<endl;
 else
 {
 cout<<"*** Grades are being held for not paying "
 <<"the tuition. ***"<<endl;
 cout<<"Amount Due: $"<<billingAmount(tuitionRate)
 <<endl;
 }

 cout<<"-*"
 <<"-*"
 <<endl<<endl;
}

```

带两个参数的print()函数发送输出到屏幕。该函数的定义大部分与前面带一个参数的print()函数相同。因为personType类没有发送输出到文件的函数，所以首先使用personType类的成员函数getName来获取姓和名。然后，打印姓和名。该函数目前的定义和前述print()函数的定义是一样的。函数的定义为：

```

void studentType::print(ofstream& outp, double tuitionRate)
{
 int i;
 string first;
 string last;

 personType::getName(first, last);

 outp<<"Student Name: "<<first<<" "<<last<<endl;
 outp<<"Student ID: "<<sId<<endl;
 outp<<"Number of courses enrolled: "
 <<numberOfCourses<<endl;
 outp<<endl;

 outp<<left;
 outp<<"Course No"<<setw(15)<<" Course Name"
 <<setw(8)<<"Credits"
 <<setw(6)<<"Grade"<<endl;

 outp.unsetf(ios::left);

 for(i = 0; i < numberOfCourses; i++)
 coursesEnrolled[i].print(outp, isTuitionPaid);
 outp<<endl;

 outp<<"Total number of credit hours: "
 <<getHoursEnrolled()<<endl;

 outp<<fixed<<showpoint<<setprecision(2);

 if(isTuitionPaid)
 outp<<"Mid-Semester GPA: "<<getGpa()<<endl;
 else
 {

```





```

 return sum / getHoursEnrolled();
 }

```

sortCourses()函数把coursesEnrolled数组按照课程号进行排序。为了对数组排序,可以使用选择排序算法。因为将比较课程号,而课程号是字符串且为courseType类的私有成员变量,所以需要首先获取课程号并存储在局部变量中。

```

void studentType::sortCourses()
{
 int i, j;
 int minIndex;
 courseType temp; //variable to swap data
 string course1;
 string course2;
 for(i = 0; i < numberOfCourses - 1; i++)
 {
 minIndex = i;
 for(j = i + 1; j < numberOfCourses; j++)
 {
 //get course numbers
 coursesEnrolled[minIndex].getCourseNumber(course1);
 coursesEnrolled[j].getCourseNumber(course2);

 if(course1 < course2)
 minIndex = j;
 } //end for

 temp = coursesEnrolled[minIndex];
 coursesEnrolled[minIndex] = coursesEnrolled[i];
 coursesEnrolled[i] = temp;
 } //end for
} //end sortCourses

```

### 主程序

现在已经完成了 courseType 和 studentType 类的设计,下面将使用这些类完成该程序。

程序将限制最多处理10名学生。注意,该程序可以很容易地进行增强以处理任何数量的学生。

因为上述类的print()函数已经实现了必需的计算以打印最终分数报告,主程序要做的工作很少。实际上,主程序所要做的全部工作就是声明用于存储学生数据的对象,把数据读入这些对象,然后打印分数报告。因为输入来自于一个文件而输出将发送到一个文件中,需要声明流变量以访问输入和输出文件。该程序的主要算法为:

1. 声明变量。
2. 打开输入文件。
3. 如果输入文件不存在,程序退出。
4. 打开输出文件。
5. 获取已注册学生的数量和学费。
6. 读入学生数据。
7. 打印分数报告。

### 变量

程序最多处理10名学生。因此,必须声明一个包括10个成员的studentType类型的数组存储学生的数据。还需要存储已注册学生的数量和学费。因为数据将从文件中读入,而输出将发送到文件

中,所以需要两个流变量来访问输入和输出文件。因此,需要以下的变量:

```
studentType studentList[maxNumberOfStudents]; //array to store
 //the students' data

int noOfStudents; //variable to store the number of students
double tuitionRate; //variable to store the tuition rate

ifstream infile; //input stream variable
ofstream outfile; //output stream variable
```

**getStudentData()函数** 该函数有三个参数:一个参数用于访问输入文件,一个参数用于访问 studentList 数组,还有一个参数用于传递已注册学生的数量。该函数伪代码形式定义如下:

对于大学中的每名学生:

1. 获取姓、名、学生 ID 和 isPaid 值。

2. if isPaid 为 'Y':

    将 isTuitionPaid 设置为 true。

else

    将 isTuitionPaid 设为 false。

3. 获取该学生所学习课程的数量。

4. 对于每门课程:

    获取课程名、课程号、学分和分数。

    读入课程信息到一个 courseType 类型的对象中。

5. 读入数据到一个 studentType 类型的对象中。

需要声明几个局部变量以读取及存储数据。getStudentData()函数的定义为:

```
void getStudentData(ifstream& infile,
 studentType studentList[],
 int numberOfStudents)
{
 //Local variable
 string fName; //variable to store the first name
 string lName; //variable to store the last name
 int ID; //variable to store the student ID
 int noOfCourses; //variable to store the number of courses
 char isPaid; //variable to store Y/N; that is,
 //is tuition paid?
 bool isTuitionPaid; //variable to store true/false

 string cName; //variable to store the course name
 string cNo; //variable to store the course number
 int credits; //variable to store the course credit hours
 char grade; //variable to store the course grade

 int count; //loop control variable
 int i; //loop control variable

 courseType courses[6]; //array of objects to store the course
 //information

 for(count = 0; count < numberOfStudents; count++)
 {
 infile<<fName<<lName<<ID<<isPaid; //Step 1
 if(isPaid == 'Y') //Step 2
```

```

 isTuitionPaid = true;
 else
 isTuitionPaid = false;

 infile<<noOfCourses; //Step 3
 for(i = 0; i < noOfCourses; i++) //Step 4
 {
 infile<<cName<<cNo<<credits<<grade; //Step 4.a
 courses[i].setCourseInfo(cName, cNo,
 credits, grade); //Step 4.b
 }

 studentList[count].setInfo(fName, lName, ID,
 noOfCourses, isTuitionPaid,
 courses); //Step 5
} //end for
}

```

**printGradeReports()**函数 该函数打印分数报告。对于每名学生，该函数调用 `studentType` 类的 `print()` 函数来打印分数报告。`printGradeReports()` 函数的定义为：

```

void printGradeReports(ofstream& outfile,
 studentType studentList[],
 int numberOfStudents,
 double tuitionRate)
{
 int count;

 for(count = 0; count < numberOfStudents; count++)
 studentList[count].print(outfile, tuitionRate);
}

```

### 完整程序清单

```

//Header file courseType.h
#ifndef H_courseType
#define H_courseType

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

//The definition of the class courseType goes here.
.
.
.
#endif

//Implementation file courseTypeImp.cpp
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
#include "courseType.h"

using namespace std;
//The definitions of the member functions of the class
//courseType go here.
.
.
.

```

```
//Header file personType.h
#ifndef personType_H
#define personType_H

//The definition of the class personType goes here.
.
.
.

#endif

//Implementation File personTypeImp.cpp
#include <iostream>
#include <string>
#include "person.h"
using namespace std;

//The definitions of the member functions of the class
//personType go here.
.
.
.

//Header file studentType.h
#ifndef H_studentType
#define H_studentType

#include <fstream>
#include <string>
#include "person.h"
#include "courseType.h"

using namespace std;

//The definition of the class studentType goes here.
.
.
.

#endif

//Implementation file studentTypeImp.cpp
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "person.h"
#include "courseType.h"
#include "studentType.h"

using namespace std;

//The definitions of the member functions of the class
//studentType go here.
.
.
.

//Main program
```

```
#include <iostream>
#include <fstream>
#include <string>
#include "studentType.h"

using namespace std;

const int maxNumberOfStudents = 10;

void getStudentData(ifstream& infile,
 studentType studentList[],
 int numberOfStudents);

void printGradeReports(ofstream& outfile,
 studentType studentList[],
 int numberOfStudents,
 double tuitionRate);

int main()
{
 studentType studentList[maxNumberOfStudents];
 int noOfStudents;
 double tuitionRate;
 ifstream infile;
 ofstream outfile;
 infile.open("a:stData.txt");
 if(!infile)
 {
 cout<<"Input file does not exist. "
 <<"Program terminates."<<endl;
 return 1;
 }
 outfile.open("a:sDataOut.txt");
 infile<<noOfStudents; //get the number of students
 infile<<tuitionRate; //get the tuition rate
 getStudentData(infile, studentList, noOfStudents);
 printGradeReports(outfile, studentList,
 noOfStudents, tuitionRate);

 return 0;
}

void getStudentData(ifstream& infile,
 studentType studentList[],
 int numberOfStudents)
{
 //Local variable
 string fName; //variable to store the first name
 string lName; //variable to store the last name
 int ID; //variable to store the student ID
 int noOfCourses; //variable to store the number of courses
 char isPaid; //variable to store Y/N; that is,
 //is tuition paid?
 bool isTuitionPaid; //variable to store true/false

 string cName; //variable to store the course name
 string cNo; //variable to store the course number
 int credits; //variable to store the course credit hours
```



```

 char grade; //variable to store the course grade
 int count; //loop control variable
 int i; //loop control variable

 courseType courses[6]; //array of objects to store the course
 //information

 for(count = 0; count < numberOfStudents; count++)
 {
 infile<<fName<<lName<<ID<<isPaid; //Step 1
 if(isPaid == 'Y') //Step 2
 isTuitionPaid = true;
 else
 isTuitionPaid = false;

 infile<<noOfCourses; //Step 3
 for(i = 0; i < noOfCourses; i++) //Step 4
 {
 infile<<cName<<cNo<<credits<<grade; //Step 4.a
 courses[i].setCourseInfo(cName, cNo,
 grade, credits); //Step 4.b
 }

 studentList[count].setInfo(fName, lName, ID,
 noOfCourses, isTuitionPaid,
 courses); //Step 5
 } //end for
}

void printGradeReports(ofstream& outfile,
 studentType studentList[],
 int numberOfStudents,
 double tuitionRate)
{
 int count;
 for(count = 0; count < numberOfStudents; count++)
 studentList[count].print(outfile,tuitionRate);
}

```

### 输出示例

Student Name: Lisa Miller  
 Student ID: 890238  
 Number of courses enrolled: 4

| Course No | Course Name | Credits | Grade |
|-----------|-------------|---------|-------|
| CSC478    | ComputerSci | 3       | B     |
| HIS356    | History     | 3       | A     |
| MTH345    | Mathematics | 4       | A     |
| PHY357    | Physics     | 3       | B     |

Total number of credit hours: 13  
 Mid-Semester GPA: 3.54

-----

Student Name: Bill Wilton  
 Student ID: 798324  
 Number of courses enrolled: 5

| Course No | Course Name | Credits | Grade |
|-----------|-------------|---------|-------|
|-----------|-------------|---------|-------|

|        |             |   |     |
|--------|-------------|---|-----|
| BIO234 | Biology     | 4 | *** |
| CHM256 | Chemistry   | 4 | *** |
| ENG378 | English     | 3 | *** |
| MTH346 | Mathematics | 3 | *** |
| PHL534 | Philosophy  | 3 | *** |

Total number of credit hours: 17

\*\*\* Grades are being held for not paying the tuition. \*\*\*

Amount Due: \$5865.00

-----

Student Name: Dandy Goat

Student ID: 746333

Number of courses enrolled: 6

| Course No | Course Name | Credits | Grade |
|-----------|-------------|---------|-------|
| BUS128    | Business    | 3       | C     |
| CHM348    | Chemistry   | 4       | B     |
| CSC201    | ComputerSci | 3       | B     |
| ENG328    | English     | 3       | B     |
| HIS101    | History     | 3       | A     |
| MTH137    | Mathematics | 3       | A     |

Total number of credit hours: 19

Mid-Semester GPA: 3.16

-----

### 输入文件

```

3 345
Lisa Miller 890238 Y 4
Mathematics MTH345 4 A
Physics PHY357 3 B
ComputerSci CSC478 3 B
History HIS356 3 A

Bill Wilton 798324 N 5
English ENG378 3 B
Philosophy PHL534 3 A
Chemistry CHM256 4 C
Biology BIO234 4 A
Mathematics MTH346 3 C

Dandy Goat 746333 Y 6
History HIS101 3 A
English ENG328 3 B
Mathematics MTH137 3 A
Chemistry CHM348 4 B
ComputerSci CSC201 3 B
Business BUS128 3 C

```

## 14.5 小结

1. 继承和组合是把两个或多个类联系起来的有效方法。
2. 继承是“is-a”关系。
3. 组合是“has-a”关系。
4. 在单重继承中，派生类仅派生于一个已经存在的类，称为基类。
5. 在多重继承中，派生类派生于一个以上的基类。



6. 基类的私有成员对于基类是私有的, 派生类不能直接访问它们。
7. 基类的公有成员可以被派生类继承为公有或私有。
8. 派生类可以重定义基类的成员函数, 但重定义仅适用于派生类的对象。
9. 对基类构造函数的调用是在派生类构造函数定义的头部指定的。
10. 当初始化派生类的对象时, 基类的构造函数首先被执行。
11. 复习本章给出的继承规则。
12. 在组合情况下, 一个类的成员是另一个类的对象。
13. 在组合情况下, 对成员对象构造函数的调用是在类构造函数定义的头部指定的。
14. OOD 的三个基本原则为封装、继承和多态。
15. 确定类、对象和操作的一种简单办法是用语言描述问题, 然后确定所有的名词和动词。从名词列表中选择类(对象), 从动词列表中选择操作。

## 14.6 练习

1. 判断下列语句的对错。
  - a. 派生类的构造函数在函数定义的头部指定对基类构造函数的调用。
  - b. 派生类的构造函数使用类名指定对基类构造函数的调用。
  - c. 假设  $x$  和  $y$  为类,  $x$  的一个成员变量是  $y$  类型的对象, 两个类都存在构造函数。可以在类  $x$  的构造函数中调用类  $y$  的构造函数。
  - d. 派生类必须有一个构造函数。
2. 画一个类层次体系图, 在该体系中, 多个类派生于单一基类。
3. 假设 `employeeType` 类派生于 `personType` 类(参见例13.9, 第13章)。给出可以扩展 `employeeType` 类成员变量和成员函数的例子。
4. 解释类的私有成员和保护成员之间的差别。
5. 考虑下面的类定义:

```
class aClass
{
public:
 void print() const;
 void set(int, int);
 aClass();
 aClass(int, int);

private:
 int u;
 int v;
};
```

下面的类定义有什么错误?

```
a.
class bClass public aClass
{
public:
 void print()
 void set(int, int, int);
private:
 int z;
}
```

```

b.
class cClass: public aClass
{
public:
 void print();
 int sum();
 cClass();
 cClass(int)
}

```

6. 考虑下面的语句:

```

class yClass
{
public:
 void one();
 void two(int, int);
 yClass();
private:
 int a;
 int b;
};

class xClass: public yClass
{
public:
 void one();
 xClass();
private:
 int z;
};

yClass y;
xClass x;

```

a. yClass 的私有成员是 xClass 的公有成员。对还是错?

b. 把下列语句标为有效或无效。如果一个语句是无效的, 解释原因。

```

(i)
 void yClass::one()
 {
 cout<<a+b<<endl;
 }
(ii)
 y.a = 15;
 x.b = 30;
(iii)
 void xClass::one()
 {
 a = 10;
 b = 15;
 z = 30;
 cout<<a+b+z<<endl;
 }
(iv)
 cout<<y.a<<" "<<y.b<<" "<<x.z<<endl;

```

7. 利用练习 6 的声明:

a. 写出 yClass 的默认构造函数, 使得 yClass 的私有成员变量初始化为 0。

b. 写出 xClass 的默认构造函数, 使得 xClass 的私有成员变量初始化为 0。

- c. 写出 yClass 的成员函数 two() 的定义, 使得私有成员变量 a 初始化为 two() 第一个参数的值, 私有成员变量 b 初始化为第二个参数的值。

8. 下列代码有何错误?

```
class classA
{
protected:
 void setX(int a); //Line 1
 //Post: x = a; //Line 2
private: //Line 3
 int x; //Line 4
};
.
.
.
int main()
{
 classA aObject; //Line 5
 aObject.setX(4); //Line 6
 return 0; //Line 7
}
```

9. 考虑下列代码:

```
class one
{
public:
 void print() const;
 //Output the values of x and y
protected:
 void setData(int u, int v);
 //Post: x = u; y = v;
private:
 int x;
 int y;
};
class two: public one
{
public:
 void setData(int a, int b, int c);
 //Post: x = a; y = b; z = c;
 void print() const;
 //Output the values of x, y, and z
private:
 int z;
};
```

- a. 写出 two 类 setData() 函数的定义。

- b. 写出 two 类 print() 函数的定义。

10. 下列 C++ 程序的输出是什么?

```
#include <iostream>
#include <string>

using namespace std;

class baseClass
{
```

```

public:
 void print() const;

 baseClass(string s = " ", int a = 0);
 //Post: str = s; x = a
protected:
 int x;
private:
 string str;
};

class derivedClass: public baseClass
{
public:
 void print() const;

 derivedClass(string s = "", int a = 0, int b = 0);
 //Post: str = s; x = a; y = b
private:
 int y;
};

int main()
{
 baseClass baseObject("This is base class", 2);
 derivedClass derivedObject("DDDDDD", 3, 7);

 baseObject.print();
 derivedObject.print();

 return 0;
}

void baseClass::print() const
{
 cout<<x<<" "<<str<<endl;
}

baseClass::baseClass(string s, int a)
{
 str = s;
 x = a;
}

void derivedClass::print() const
{
 cout<<"Derived class: "<<y<<endl;
 baseClass::print();
}

derivedClass::derivedClass(string s, int a, int b)
 :baseClass("Hello Base", a + b)
{
 y = b;
}

```

11. 下列程序的输出是什么?

```

#include <iostream>

using namespace std;

```

```
class baseClass
{
public:
 void print() const;

 int getX();

 baseClass(int a = 0);

protected:
 int x;
};

class derivedClass: public baseClass
{
public:
 void print() const;

 int getResult();

 derivedClass(int a = 0, int b = 0);

private:
 int y;
};

int main()
{
 baseClass baseObject(7);
 derivedClass derivedObject(3,8);

 baseObject.print();
 derivedObject.print();

 cout<<"**** "<<baseObject.getX()<<endl;
 cout<<"#### "<<derivedObject.getResult()<<endl;

 return 0;
}

void baseClass::print() const
{
 cout<<"In base: x = "<<x<<endl;
}

baseClass::baseClass(int a)
{
 x = a;
}

int baseClass::getX()
{
 void x;
}

void derivedClass::print() const
{
 cout<<"In derived: x = "<<x<<" , y = "<<y<<" ; x + y = "
 <<x + y<<endl;
}

int derivedClass::getResult()
{

```

```

 return x + y;
}

derivedClass::derivedClass(int a, int b)
 :baseClass(a)
{
 y = b;
}

```

## 14.7 编程练习

1. 在第13章中,曾设计了clockType类解决时间问题。某些程序,除了小时、分钟和秒之外,可能还需要存储时区。通过增加一个存储时区的成员变量,从clockType类派生出extClockType类。增加必要的成员函数和构造函数以使类可以使用。此外,写出成员函数和构造函数的定义。最后,编写测试程序测试该类。
2. 在本章中设计了dateType类以在程序中实现日期,但成员函数setDate()和带参数的构造函数在把日期存储到成员变量之前并不检查该日期是否有效。重写setDate()函数和构造函数的定义,使得月、日期和年的值在存储日期到成员变量之前进行验证。此外,用一个带默认参数的构造函数代替默认构造函数和带参数的构造函数。增加一个成员函数isLeapYear(),检查某年是否为闰年。最后,编写测试程序测试该类。
3. 平面上的一点由x坐标和y坐标表示。设计一个类pointType,可以存储和处理平面上的一点。通过pointType类,可以执行对一点的操作,例如显示点坐标、设置点的坐标、打印点的坐标、返回x坐标和返回y坐标。此外,编写测试程序测试对点的各种操作。
4. 每个圆有一个圆心和半径。给出半径,可以确定圆的面积和周长。给出圆心,可以确定它在平面上的位置。设计一个类circleType,可以存储圆的半径和圆心。因为圆心是平面上的一点,而编程练习3中已设计了类来保存点的属性,读者必须从pointType类中派生出circleType类。必须能够执行对圆的一般操作,如设置半径、打印半径、计算和打印面积及周长以及执行对圆心的一般操作。
5. 每个圆柱体包含一个底面和高,底面是一个圆。设计一个类cylinderType,可以保存圆柱的属性和执行对圆柱的操作。从编程练习4中的circleType类派生出该类。可以作用于圆柱的一些操作如下:计算和打印体积、计算和打印表面积、设置高、设置底面的半径、设置底面的圆心。
6. 设计一个在线地址簿,记录姓名、地址、电话号码、家庭成员、朋友和同事的生日。程序应该能够处理最多500个条目。
  - a. 定义一个类addressType,可以存储街道、城市、州和邮政编码。使用适当的函数来打印和存储地址。此外,使用构造函数自动地初始化成员变量。
  - b. 使用personType类(第13章例13.9所定义)、dateType类(本章编程练习2中所设计)和addressType类定义一个extPersonType类。给该类增加一个成员变量,把人员分类为家庭成员、朋友或同事。此外,增加一个成员变量以存储电话号码。增加(或重载)函数以打印和存储适当的信息。使用构造函数自动地初始化成员变量。
  - c. 使用前面定义的类型addressBookType。addressBookType类型的对象应该能够处理最多500个条目。  
程序应该能够执行下列操作:



- (i) 从磁盘读入数据到地址簿中。
  - (ii) 按照姓氏对地址簿进行排序。
  - (iii) 按照姓氏查找。
  - (iv) 打印指定人员的地址、电话号码和生日（如果有的话）。
  - (v) 打印生日在两个指定日期之间的人员姓名。
  - (vi) 打印两个姓氏之间的所有人的姓名。
  - (vii) 按照用户的需求，打印所有家庭成员、朋友或同事的姓名。
7. 在编程练习2中设计和实现了dateType类来记录日期，但该类的操作很有限。重定义dateType类，使得可以对日期执行除了已定义操作以外的下列操作：
- a. 设置月。
  - b. 设置日期。
  - c. 设置年。
  - d. 返回月。
  - e. 返回日期。
  - f. 返回年。
  - g. 测试某年是否为闰年。
  - h. 返回某月中的天数。例如，如果日期为3-12-2002，返回的天数为31，因为3月份有31天。
  - i. 返回一年中已经过的天数。例如，如果日期为3-18-2002，一年中已经过的天数为77天。  
注意返回的天数包括当前日期。
  - j. 返回一年中剩余的天数。例如，如果日期为3-18-2002，一年中剩余的天数为288天。
  - k. 计算在日期中增加相应天数后的新日期。例如，如果日期为3-18-2002，要增加的天数为25，新的日期为4-12-2002。
8. 写出用于实现编程练习7中dateType类所有操作函数的定义。
9. 编程练习7中定义的dateType类以数字形式打印日期。一些应用程序也许会要求以另一种格式打印日期，如March 24, 2002。派生出extDateType类，使得日期可以用其中任何一种格式打印。
- 将extDateType类增加一个成员变量，使得月份可以用字符串格式存储。增加一个成员函数，以月（字符串）、年形式输出日期。例如，March 2002。
10. 使用extDateType类（编程练习9）和dayType类（第13章编程练习2）设计calendarType类，使得在给定月和年后可以打印该月的日历。为了打印每月的日历，必须知道该月的第一天和天数。因此，必须以dayType类型存储月份的第一天。很明显，可以设置日期中的日期为1以及设置月和年为用户指定值，月和年可以存储到extDateType类型的对象中。因此，calendarType类有两个成员变量：dayType类型的对象和一个extDateType类型的对象。
- 设计calendarType类，使得程序可以打印1500年1月1日起任何月份的日历。注意，1500年1月1日是星期一。为了计算某月的第一天，可以从1500年1月1日起增加适当的天数。对于calendarType类，应包括下列操作：
- a. 计算并打印日历中月份的第1天。可以调用函数firstDayOfMonth()。
  - b. 设置月。
  - c. 设置年。
  - d. 返回月。
  - e. 返回年。

- f. 打印指定月份的日历。
  - g. 增加适当的构造函数来初始化成员变量。
11. a. 编写 `calendarType` 类成员函数的定义, 实现 `calendarType` 类的操作。
- b. 编写测试程序, 打印指定月或年的日历。例如, 2002 年 9 月的日历为:

| September 2002 |     |     |     |     |     |     |
|----------------|-----|-----|-----|-----|-----|-----|
| Sun            | Mon | Tue | Wed | Thu | Fri | Sat |
| 1              | 2   | 3   | 4   | 5   | 6   | 7   |
| 8              | 9   | 10  | 11  | 12  | 13  | 14  |
| 15             | 16  | 17  | 18  | 19  | 20  | 21  |
| 22             | 23  | 24  | 25  | 26  | 27  | 28  |
| 29             | 30  |     |     |     |     |     |



## 第15章 指针、类和虚函数

### 本章要点:

- 学习指针数据类型和指针变量
- 研究如何声明和操作指针变量
- 学习取址运算符和取值运算符
- 研究动态变量
- 研究如何使用 new 和 delete 运算符操作动态变量
- 学习指针算术
- 研究动态数组
- 认识数据的浅 (shallow) 复制和深 (deep) 复制
- 研究带指针成员变量的类
- 学习虚函数
- 分析取址运算符和类的关系

C++ 的数据类型分为三类: 简单数据类型、结构化数据类型和指针。目前为止, 读者只学习了前两类数据类型。本章讨论第三类数据类型: 指针数据类型。我们将首先学习如何声明指针变量 (简称指针) 和操作指针指向的数据。稍后, 将在学习动态数组和链表时应用这些概念。链表将在第17章中讨论。

### 15.1 指针数据类型和指针变量

指针数据类型的值是计算机内存地址。如同许多其他语言一样, 在C++中没有名字和指针变量类型相关联。因为指针数据类型的值是地址 (存储单元), 指针变量是内容为存储单元地址的变量。

**指针变量:** 内容为地址 (内存地址) 的变量。

#### 15.1.1 声明指针变量

因为没有名字和指针数据类型相关联, 指针变量的声明和其他变量有所不同。当声明一个指针变量时, 同时也指定了所指向的存储单元中值的数据类型。在C++中, 指针变量声明是通过在数据类型和变量名之间使用星号来完成的。声明一个指针变量的通用语法是:

```
dataType *identifier;
```

作为例子, 考虑下列语句:

```
int *p;
char *ch;
```

在这些语句中, p 和 ch 都是指针变量。p 的内容 (适当地赋值后) 指向 int 类型的存储单元, 而 ch 的内容指向 char 类型的存储单元。通常 p 称为 int 类型的指针变量, ch 称为 char 类型的指针变量。

在讨论指针如何工作之前，让我们先观察下列结论。语句：

```
int *p;
```

等同于语句：

```
int* p;
```

并等同于语句：

```
int * p;
```

因此，\* 字符可以出现在数据类型名字和变量名之间的任何位置。

现在考虑下面的语句：

```
int* p, q;
```

在该语句中，只有 p 是指针变量，q 不是。这里 q 是一个 int 型变量。为了避免混淆，应该把 \* 字符和变量名相连。那么前述语句可以写成：

```
int *p, q;
```

当然，语句：

```
int *p, *q;
```

将把 p 和 q 都声明为指向 int 类型的指针变量。

现在已经知道如何声明指针，下面将讨论如何使指针指向一个存储空间及如何操作存储在这些存储单元中的数据。

因为指针的值是内存地址，指针可以存储指定类型的存储空间地址。例如，如果 p 是指向 int 类型的指针，p 可以存储任何 int 类型的内存空间地址。C++ 提供了两种运算符：取址运算符 (&) 和取值运算符 (\*)，用以协同指针操作。下两节将描述这些运算符。

## 15.2 取址运算符 (&)

在 C++ 中，& 符号称为取址运算符。它是一个一元运算符，返回其操作数的地址。例如，给出语句：

```
int x;
int *p;

p = &x;
```

将 x 的地址赋给 p。就是说，x 和 p 的值引用相同的存储单元。

## 15.3 取值运算符 (\*)

到目前为止的每一章都使用了 \* 字符，作为二元乘法运算符。C++ 同样使用 \* 作为一元运算符。当用做一元运算符时，\* 通常称为取值运算符或间接运算符，引用其操作数（指针）所指的對象。例如，给出语句：

```
int x = 25;
int *p;
p = &x; //store the address of x in p
```

语句:

```
cout<<*p<<endl;
```

打印存储  $p$  所指存储空间中的值, 该值为  $x$ 。此外, 语句:

```
*p = 55;
```

将 55 存储到  $p$  所指的存储单元, 即  $x$  中。

考虑下列语句:

```
int *p;
int num;
```

在这些语句中,  $p$  是指向  $\text{int}$  类型的指针变量而  $\text{num}$  是  $\text{int}$  类型的变量 (参见图 15.1)。

假定存储单元 1200 分配给  $p$  而存储单元 1800 分配给  $\text{num}$ 。语句:

```
num = 78;
```

将 78 存储到  $\text{num}$  中, 即存储在单元 1800 中 (参见图 15.2)。

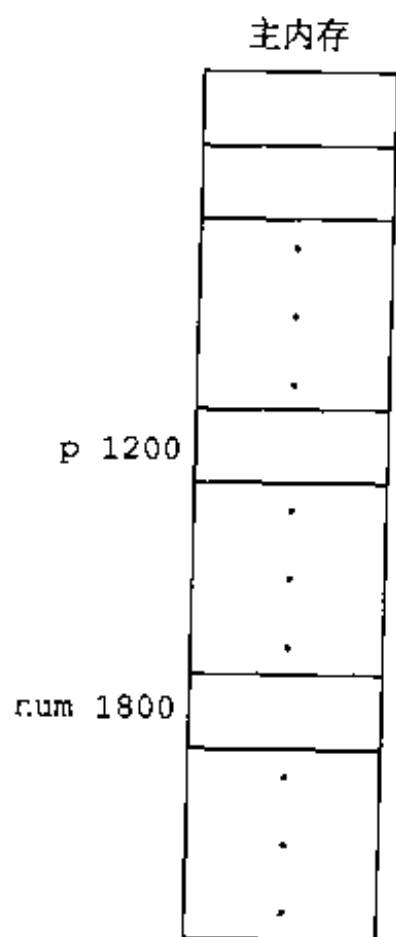


图 15.1 主内存、 $p$  和  $\text{num}$

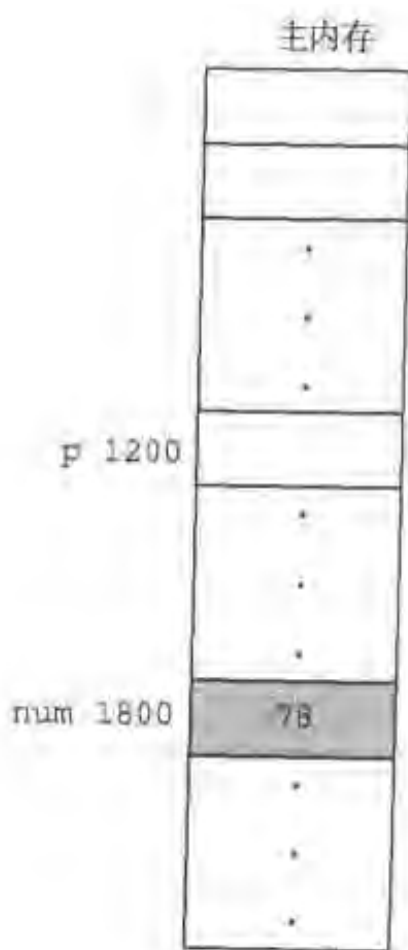


图 15.2 在执行 “ $\text{num} = 78;$ ” 后的  $\text{num}$  值

语句:

```
p = #
```

将地址 1800 存储到  $p$  中。在该语句执行后,  $*p$  和  $\text{num}$  都引用存储单元 1800 中的内容, 即  $\text{num}$  的值 (参见图 15.3)。

赋值语句:

```
*p = 24;
```

将改变存储单元 1800 中的内容, 因此还改变了  $\text{num}$  的内容。(参见图 15.4。)

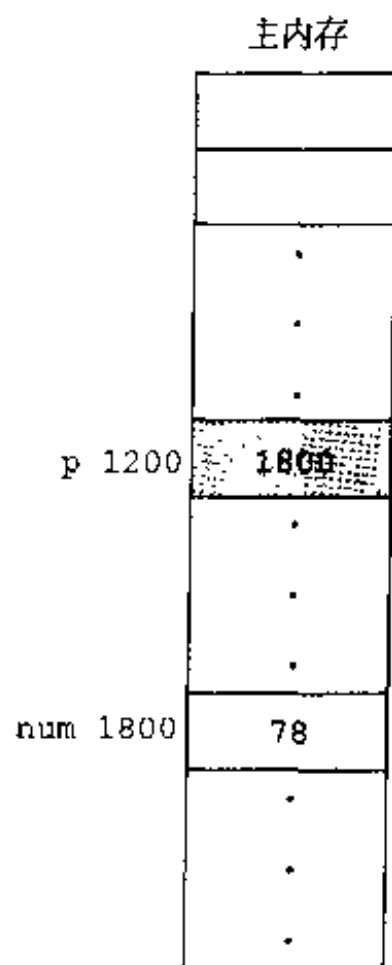


图 15.3 在执行 “p = &amp;num;” 后的 p 值

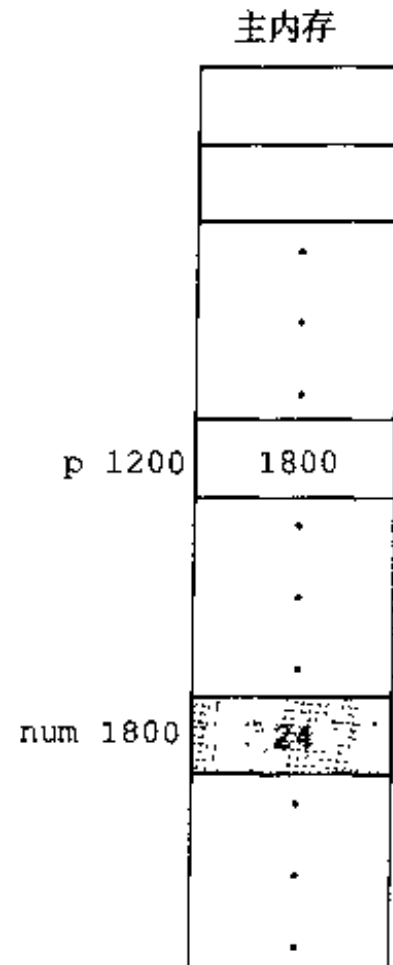


图 15.4 在执行 “\*p = 24;” 后的 \*p 和 num 的值

让我们总结一下前述讨论。

1. &p、p 和 \*p 都有不同的含义。
2. &p 表示 p 的地址 1200（参见图 15.4）。
3. p 表示变量 p 的内容（图 15.4 中的 1800）。
4. \*p 表示 p 所指向（存储单元 1200 的内容所指向）的存储单元（图 15.4 中的 1800）的内容（图 15.4 中的 24）。

### 例 15.1

考虑下列语句：

```
int *p;
int x;
```

假设有图 15.5 所示的 p 和 x 的存储单元。

&p、p、\*p、&x 和 x 的值如下：

|    | 值        |
|----|----------|
| &p | 1400     |
| p  | ??(未知)   |
| *p | 不存在(未定义) |
| &x | 1750     |
| x  | ??(未知)   |

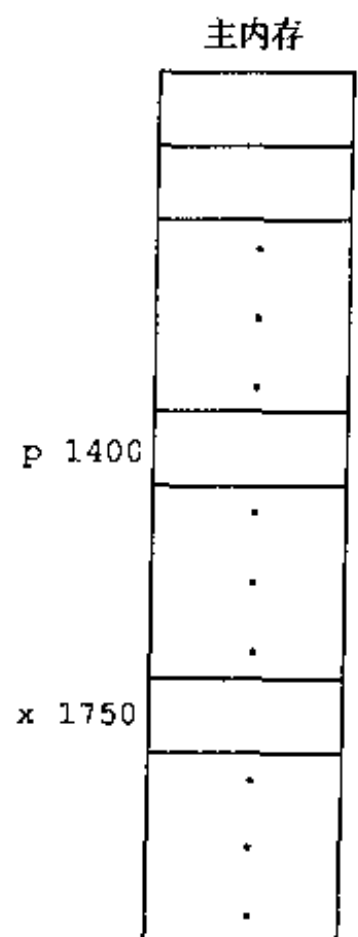


图15.5 主内存、p和x

假设下列语句以给定的顺序执行：

```
x = 50;
p = &x;
*p = 38;
```

&p、p、\*p、&x 和 x 的值在每条语句执行之后显示。在语句：

x = 50;

执行后,  $\&p$ 、 $p$ 、 $*p$ 、 $\&x$  和  $x$  的值如下:

|       | 值        |
|-------|----------|
| $\&p$ | 1400     |
| $p$   | ??(未知)   |
| $*p$  | 不存在(未定义) |
| $\&x$ | 1750     |
| $x$   | 50       |

在语句:

```
p = &x;
```

执行后,  $\&p$ 、 $p$ 、 $*p$ 、 $\&x$  和  $x$  的值如下:

|       | 值    |
|-------|------|
| $\&p$ | 1400 |
| $p$   | 1750 |
| $*p$  | 50   |
| $\&x$ | 1750 |
| $x$   | 50   |

注意在语句“ $p = \&x;$ ”执行后,  $p$  包含  $x$  的地址, 因此  $*p$  和  $x$  引用相同的存储空间, 就是  $x$ 。所以,  $*p$  的值为 50。

在语句:

```
*p = 38;
```

执行后,  $\&p$ 、 $p$ 、 $*p$ 、 $\&x$  和  $x$  的值如下。(因为  $*p$  和  $x$  引用相同的存储空间,  $x$  的值同样改变为 38。)

|       | 值    |
|-------|------|
| $\&p$ | 1400 |
| $p$   | 1750 |
| $*p$  | 38   |
| $\&x$ | 1750 |
| $x$   | 38   |

由例 15.1 可知:

1. 如下的声明:

```
int *p;
```

仅为  $p$  分配内存, 并不包括  $*p$ 。稍后, 将学习如何为  $*p$  分配内存。

2. 假定如下语句:

```
int *p;
```

```
int x;
```

那么:

a.  $p$  是一个指针变量。

b.  $p$  的内容仅指向  $\text{int}$  类型的存储单元。

c. 存储单元  $x$  存在且类型为  $\text{int}$ , 所以, 赋值语句:

```
p = &x;
```

是合法的。在该赋值语句执行之后,  $*p$  是有效和有意义的。

**例 15.2**

下列程序阐明了指针变量是如何工作的：

```
//Chapter 15: Example 15-2
#include <iostream>
using namespace std;
int main()
{
 int *p;
 int x = 37;
 cout<<"Line 1: x = "<<x<<endl; //Line 1
 p = &x; //Line 2
 cout<<"Line 3: *p = "<<*p
 <<" , x = "<<x<<endl; //Line 3
 *p = 58; //Line 4
 cout<<"Line 5: *p = "<<*p
 <<" , x = "<<x<<endl; //Line 5
 cout<<"Line 6: Address of p = "<<&p<<endl; //Line 6
 cout<<"Line 7: Value of p = "<<p<<endl; //Line 7
 cout<<"Line 8: Value of the memory location "
 <<"pointed to by *p = "<<*p<<endl; //Line 8
 cout<<"Line 9: Address of x = "<<&x<<endl; //Line 9
 cout<<"Line 10: Value of x = "<<x<<endl; //Line 10
 return 0;
}
```

**运行实例**

```
Line 1: x = 37
Line 3: *p = 37, x = 37
Line 5: *p = 58, x = 58
Line 6: Address of p = 006BFDF4
Line 7: Value of p = 006BFDF0
Line 8: Value of the memory location pointed to by *p = 58
Line 9: Address of x = 006BFDF0
Line 10: Value of x = 58
```

上述程序运行如下。第1行语句输出x的值，第2行语句将x的地址存储到p中。第3行语句输出\*p和x的值。因为p包含x的地址，\*p和x的值是相同的，如第3行的输出所示。第4行语句改变\*p的值为58，第5行语句输出\*p和x的值，它们还是相同的。第6行和第10行之间的语句输出p的地址、p的值、\*p的值、x的地址和x的值。注意p的值和x的地址是相同的，因为通过第2行语句的执行，x的地址存储到p中。（注意，分别由第6、7和9行输出显示的p的地址、p的值和x的地址是与机器相关的。在用户的机器上运行该程序，可能会得到不同的值。）

**15.4 类、结构和指针变量**

在上节中，学习了如何声明和操作简单数据类型的指针，如int和char。还可以把指针声明为指向其他数据类型的指针，如类。下面将学习如何声明和操作指向类和结构的指针。（类和结构有



类似的特性。类和结构的惟一区别是，类的所有成员默认情况下是私有的，而结构的所有成员默认情况下是公有的。所以，下面的讨论对两者都适用。)

考虑下面的结构声明：

```
struct studentType
{
 char name[26] ;
 double gpa;
 int sID;
 char grade;
};

studentType student;
studentType* studentPtr;
```

在上述声明中，student 是一个 studentType 类型的对象，studentPtr 是指向 studentType 类型的指针变量。下面的语句将 student 的地址存储到 studentPtr 中：

```
studentPtr = &student;
```

下面的语句将 3.9 存储到 student 对象的成员 gpa 中：

```
(*studentPtr).gpa = 3.9;
```

表达式(\*studentPtr).gpa 是指针取值和类成员选择的混合形式。在 C++ 中，点运算符的优先级比取值运算符的优先级高。因此，圆括号是必需的。为了简化通过指针访问类成员的操作，C++ 提供了另一种运算符 ->。运算符 -> 包括两个连续的符号：连字号和“大于”号。

使用运算符 -> 访问类（结构）成员的语法为：

```
pointerVariableName->classMemberName
```

因此，语句：

```
(*studentPtr).gpa = 3.9;
```

等同于语句：

```
studentPtr->gpa = 3.9;
```

使用运算符 -> 并通过指针访问类（结构）成员消除了圆括号和取值运算符的使用。因为打字错误是不可避免的，而圆括号的遗漏可能会导致程序的异常终止或错误结果，所以当通过指针访问类（结构）成员时，本书使用符号 ->。

例 15.3 阐明了指针是如何作用于类成员函数的。

### 例 15.3

考虑下面的类：

```
class classExample
{
public:
 void setX(int a);
 //Function to set the value of the data member x
 //Post condition: x = a;
 void print() const;
 //Function to output the value of x
private:
 int x;
};
```

成员函数的定义如下：

```
void classExample::setX(int a)
{
 x = a;
}

void classExample::print() const
{
 cout<<"x = "<<x<<endl;
}
```

考虑下面的 main() 函数：

```
int main()
{
 classExample *cExpPtr; //Line 1
 classExample cExpObject; //Line 2

 cExpPtr = &cExpObject; //Line 3

 cExpPtr->setX(5); //Line 4
 cExpPtr->print(); //Line 5

 return 0;
}
```

输出：

x = 5

在 main() 函数中，第 1 行语句声明 cExpPtr 为指向 classExample 类型的指针，而第 2 行语句声明 cExpObject 为 classExample 类型的对象。第 3 行语句将 cExpObject 的地址存储到 cExpPtr 中。（参见图 15.6。）

在第 4 行语句中，指针 cExpPtr 访问成员函数 setX() 以设置成员变量 x 的值（参见图 15.7）。

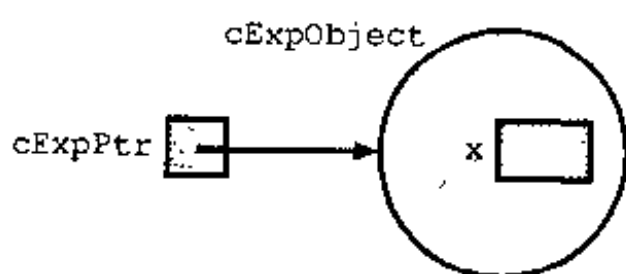


图 15.6 “cExpPtr = &cExpObject;” 执行后的 cExpObject 和 cExpPtr

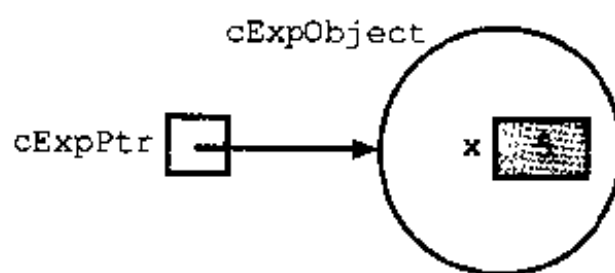


图 15.7 “cExpPtr->setX(5);” 执行后的 cExpObject 和 cExpPtr

在第 5 行语句中，指针 cExpPtr 访问成员函数 print() 以打印 x 的值，如上所示。

## 15.5 初始化指针变量

因为 C++ 不会自动地初始化变量，如果不想指针指向任意位置的话，必须初始化指针变量。指针变量使用常值 0 初始化，称为空指针。因此，语句 “p = 0;” 将空指针存储到 p 中；就是说，p 不指向任何位置。一些程序员使用常量 NULL 来初始化指针变量。下面两条语句是等价的：

```
p = NULL;
p = 0;
```

注意：数字 0 是惟一能直接赋给指针变量的数值。



## 15.6 动态变量

在上节中,学习了如何声明指针变量,如何将变量地址存储到指针变量中,以及如何使用指针操作数据。然而,我们仅学习了如何使用指针操作已使用其他变量创建的存储空间中的数据。换句话说,指针只是操作已有存储单元中的数据。既然可以通过变量创建并访问相应内存空间,那么指针的优势是什么呢?在本节中,将了解指针背后的真正作用。具体而言,将学习如何在程序执行中使用指针分配和释放内存。

在程序执行中创建的变量称为动态变量。C++在指针的帮助下创建动态变量。C++提供了两个运算符new和delete,分别创建和销毁动态变量。当程序需要一个新的变量时,使用new运算符;当程序不再需要一个动态变量时,使用delete运算符。

在C++中,new和delete是保留字。

new运算符有两种形式:一种用来分配单个变量,另一种用来分配变量数组。使用new运算符的语法为:

```
new dataType; //to allocate a single variable
new dataType[intExp]; //to allocate an array of variables
```

其中,intExp是值等于正整数的表达式。

new运算符分配指定类型的内存(变量),返回指向它的指针——就是该内存的地址。此外,分配的内存是未经初始化的。

考虑下面的声明:

```
int *p;
char *q;
int x;
```

语句:

```
p = &x;
```

存储x的地址到p中。然而,没有分配新的内存。另一方面,考虑下面的语句:

```
p = new int;
```

该语句于程序执行当中在内存的某处创建一个变量,存储所分配内存的地址到p中。分配的内存通过指针取值访问,也就是\*p。类似地,语句:

```
q = new char[16];
```

创建一个char类型、包括16个元素的数组并将数组地址存储到q中。

因为动态变量是未命名的,它不能被直接访问。它是通过new返回的指针间接访问的。下列语句阐述了这一概念:

```
int *p; //p is a pointer of the type int
char *name; //name is a pointer of the type char
string *str; //str is a pointer of the type string

p = new int; //allocates memory of the type int
 //and stores the address of the
 //allocated memory in p
*p = 28; //stores 28 in the allocated memory

name = new char[5]; //allocates memory for an array of
 //five components of the type char and
```

```

//stores the base address of the array
//in name
strcpy(name, "John"); //stores John in name
str = new string; //allocates memory of the type string
//and stores the address of the
//allocated memory in str
*str = "Sunny Day"; //stores the string "Sunny Day" in
//the memory pointed to by str

```

当不再需要动态变量时,可以被销毁。也就是说,内存可以被释放。C++运算符 delete 用来销毁动态变量。使用运算符 delete 的语法有两种形式:

```

delete pointervariable; //to destroy a single dynamic variable
delete [] pointervariable; //to destroy a dynamically created array

```

因此,语句:

```

delete p;
delete [] name;

```

释放了指针 p 和 name 所引用的内存。

## 15.7 指针变量的运算

允许作用于指针变量的运算是赋值、关系运算以及一些有限的算术运算。指针变量的值可以赋给相同类型的另一个指针变量。两个相同类型的指针变量可以比较是否相等,等等。可以对指针变量加或减一个整数值。指针变量间也可以进行减法操作。

例如,假设有以下语句:

```
int *p, *q;
```

语句:

```
p = q;
```

复制 q 的值到 p 中。在该语句执行后, p 和 q 都指向相同的存储单元。对 \*p 的任何改变都相应地改变 \*q 的值,反之亦然。

如果 p 和 q 具有相同值,即指向相同的内存单元,表达式:

```
p == q
```

值为 true。类似地,表达式:

```
p != q
```

值为 true,如果 p 和 q 指向不同的内存单元。

指针算术运算不同于数字的算术运算。首先,让我们使用下列语句解释指针变量的增加和减少:

```

int *p;
double *q;
char *chPtr;
studentType *stdPtr; //studentType data type is as defined before

```

int 类型变量分配的内存大小为 4 个字节, double 类型变量为 8 个字节,而 char 类型变量为 1 个字节。给 studentType 类型变量分配的内存大小为 39 个字节。

语句:

```
p++; 或 p = p + 1;
```

将 p 的值增加 4 个字节, 因为 p 是指向 int 类型的指针。类似地, 语句:

```
q++;
chPtr++;
```

分别将 q 的值增加 8 个字节, 将 chPtr 的值增加 1 个字节。语句:

```
stdPtr++;
```

将 stdPtr 的值增加 39 个字节。

递增运算符将指针变量的值增加所指变量的尺寸值。类似地, 递减运算符将指针变量的值减少所指变量的尺寸值。

此外, 语句:

```
p = p + 2;
```

将 p 的值增加 8 个字节。

因此, 当把一个指针变量增加一个整数时, 指针变量的值增加所指内存尺寸的整数倍。类似地, 当从指针变量减去一个整数值时, 指针变量的值减少所指内存尺寸的整数倍数。

**注意:** 指针运算可能会很危险。使用指针运算, 程序可能会意外地访问其他变量的存储单元以及在无警告的情况下改变它们的内容, 从而使程序员难以查明问题所在。如果一个指针变量试图访问其他变量的存储空间或非合法的存储空间, 某些系统可能会终结程序, 并输出适当的错误信息。进行指针算术时一定要特别小心。

## 15.8 动态数组

在第 9 章中, 学习了如何声明和处理数组。第 9 章中讨论的数组称为静态数组, 因为它们的大小是在编译时确定的。静态数组的一个限制是每次执行程序时, 数组的大小是固定的, 因此使用相同的数组处理相同类型的不同数据集合是不可能的。解决这一限制的一种办法是声明一个足够大的数组以处理各种数据集合。然而, 如果数组很大而数据集合很小, 声明将会导致内存浪费。另一方面, 如果在程序执行当中可以提示用户输入数组的大小, 然后创建适当大小的数组, 将是极其有帮助的。尤其是当不能估计数组尺寸大小时, 该方法将是特别有用的。在本节中, 将学习如何在程序执行当中创建和处理数组。

在程序的执行当中创建的数组称为动态数组。为了创建动态数组, 可以使用 new 运算符的第二种形式。

语句:

```
int *p;
```

声明 p 为指向 int 类型的指针变量。语句:

```
p = new int[10];
```

分配 10 个连续的 int 型存储单元, 然后把存储单元的首地址存储到 p 中。换句话说, new 运算符创建了包括 10 个成员的 int 类型数组, 它返回数组的地址, 赋值运算符把数组的地址存储到 p 中。因此, 语句:

```
*p = 25;
```

将25存储到第1个存储单元中,语句:

```
p++; //p points to the next array component
*p = 35;
```

将35存储到第2个存储单元。因此,通过使用递增和递减运算符,可以访问数组的成员。当然,在执行一些递增运算后,很可能失去第1个数组成员的地址。C++允许使用数组符号访问这些存储单元。例如,语句:

```
p[0] = 25;
p[1] = 35;
```

分别把25和35存储到第1和第2个数组成员中。就是说,p[0]引用第1个数组成员,p[1]引用第2个数组成员,等等。通常,p[i]引用第(i+1)个数组成员。在上述语句执行后,p仍然指向第1个数组成员。此外,下面的for循环把每一个数组成员初始化为0:

```
for(j = 0; j < 10; j++)
 p[j] = 0;
```

这里j是一个整型变量。

当使用数组符号处理p所指向的数组时,p固定在第1个存储单元。此外,p是程序执行时动态创建的数组,称为动态数组。

注意:语句:

```
int list[10];
```

声明list为包括10个成员的数组。现在list本身是一个变量,而存储在list中的值为数组的地址,即第1个数组成员的地址。因为它包含一个指针类型的值,list是一个指针变量。然而,递增和递减运算不能应用于list,因为list总是指向第1个数组成员。任何对list的递增和递减运算将导致编译时错误。而且,如果p是一个int类型的指针变量,那么语句:

```
p = list;
```

复制list的值到p中。对p的递增和递减运算是允许的。

数组名是一个常量指针。

#### 例 15.4

下面的程序阐明了如何获取用户的响应并得到数组的大小,以及在程序执行时创建动态数组。考虑下面的语句:

```
int *intList; //Line 1
int arraySize; //Line 2

cout<<"Enter array size: "; //Line 3
cin>>arraySize; //Line 4
cout<<endl; //Line 5

intList = new int[arraySize]; //Line 6
```

第1行语句声明intList为指向int类型的指针,第2行语句声明arraySize为int类型的变量。第3行语句提示用户输入数组的大小,第4行语句输入数组的大小到变量arraySize中。第6行语句创建一个大小为arraySize的数组,数组的地址存储在intList中。intList可以像任何其他数组一样对待。例如,可以使用数组符号处理intList的元素及传递intList作为函数参数。

#### 15.8.1 函数和指针

指针变量可以用值传递或引用传递的方式作为函数的参数。为了在函数头部声明一个指针为值

参数, 可以使用与声明变量相同的机制。为了使一个形参成为一个引用类型的参数, 当在函数头部声明形参时使用 &。所以, 为了声明一个形参为引用参数, 必须使用 &。在数据类型名和标志符名之间, 必须使用 \* 以使标志符为指针类型, 使用 & 以使标志符为引用类型。为了声明一个指针为引用参数, & 和 \* 应该以什么样的次序出现在数据类型名和标志符之间呢? 在 C++ 中, 在函数头部声明一个指针为引用参数时, \* 必须出现在 & 之前, 即数据类型名和标志符之间。下面的例子阐述了这一概念:

```
void example(int* &p, double *q)
{
 .
 .
 .
}
```

在该例子中, p 和 q 都是指针。参数 p 是一个引用参数, 参数 q 是一个值参数。

### 指针和函数返回值

在 C++ 中, 函数可以返回指针类型的值。例如, 函数:

```
int* testExp(...)
{
 .
 .
 .
}
```

的返回值是一个指向 int 类型的指针。

## 15.9 指针的浅 (shallow) 复制与深 (deep) 复制

在前面, 讨论了指针算术, 还解释了指针可能会无意地访问另一个 (无关的) 指针的数据。该事件也许会导致不为人知的错误结果。这里, 我们讨论指针的另一种特性。为了方便讨论, 使用图示显示指针及其相关内存。考虑下面的语句:

```
int *p;
p = new int;
```

第 1 条语句声明 p 为指向 int 类型的指针变量。第 2 条语句分配 int 类型的内存, 所分配内存的地址存储到 p 中。地址如图 15.8 所示。

方框表示分配的内存 (在该情况下, 为 int 类型), p 和箭头表示 p 指向所分配的内存。现在考虑下面的语句:

```
*p = 87;
```

该语句将 87 存储到 p 所指的内存中, 如图 15.9 所示。

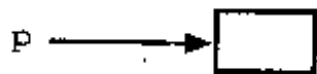


图 15.8 指针 p 和它指向的内存

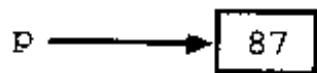


图 15.9 所指内存中包含值 87 的指针 p

考虑下列语句:

```
int *first;
int *second;
first = new int[10];
```

前两条语句声明 `first` 和 `second` 为指向 `int` 类型的指针变量。第3条语句创建一个包括10个成员的数组，数组的地址存储到 `first` 中（参见图 15.10）。

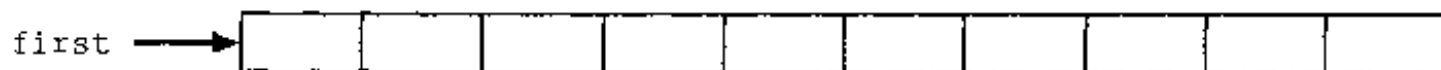


图 15.10 指针 `first` 和它所指向的数组

假设一些相关数据存储在 `first` 所指的数组中。具体而言，假设数组如图 15.11 所示。

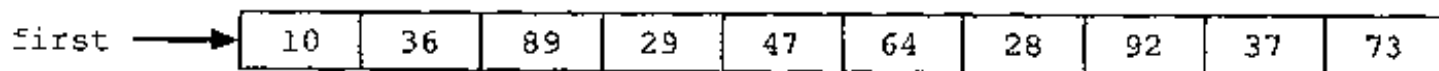


图 15.11 指针 `first` 和它的数组

考虑下面的语句：

```
second = first; //Line A
```

该语句复制 `first` 的值到 `second` 中。在该语句执行后，`first` 和 `second` 均指向相同的数组，如图 15.12 所示。

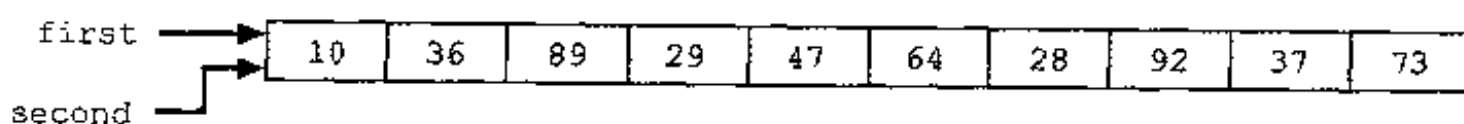


图 15.12 在 “`second = first;`” 执行后的 `first` 和 `second`

执行下面的语句：

```
delete [] second;
```

在该语句执行后，`second` 所指向的数组被删除了。该操作导致图 15.13 所示的结果。

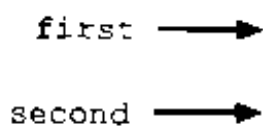


图 15.13 在 “`delete [] second;`” 执行后的 `first` 和 `second`

因为 `first` 和 `second` 指向相同的数组，在语句：

```
delete [] second;
```

执行后，`first` 变成无效。因此，如果程序在稍后试图访问 `first` 所指的内存的话，程序不是访问了错误的内存，就是异常终结。这种情况就是浅复制的一个例子。在浅复制中，两个或多个相同类型的指针指向相同的内存，就是说，它们指向相同的数据。

另一方面，假设有下列语句 “`second = first;`” (Line A)：

```
second = new int[10];
for(int j = 0; j < 10; j++)
 second[j] = first[j];
```

第1条语句创建了一个包括10个成员的 `int` 类型数组，数组的地址存储到 `second` 中。第2条语句复制 `first` 所指向的数组到 `second` 所指向的数组中（参见图 15.14）。

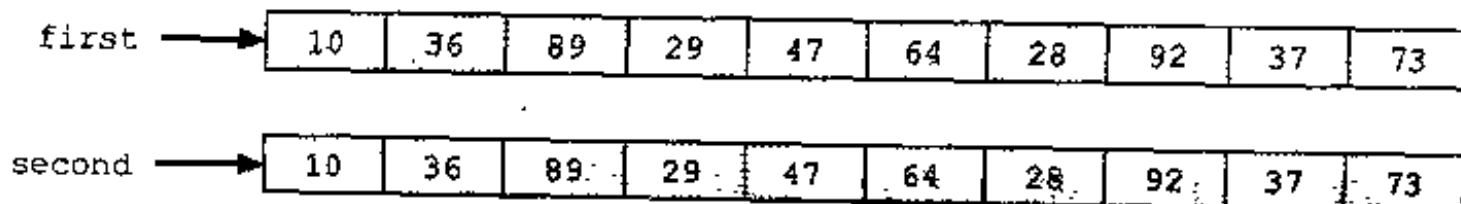


图 15.14 指向各自数据的 `first` 和 `second`



first 和 second 现在都指向它们自己的数据。如果 second 删除了它的内存，将不会对 first 产生影响。这种情况就是深复制的一个例子。在深复制中，两个或多个指针有它们自己的数据。

从上述讨论可知，必须知道何时使用浅复制和何时使用深复制。

## 15.10 类和指针：一些特性

如果一个指针变量指向一个类，在上节中讨论了如何使用箭头符号通过指针访问类的成员。因为类可以有指针成员变量，本节讨论此种类的一些特性。为了分别讨论，将使用下面的类：

```
class pointerDataClass
{
public:
 ...
private:
 int x;
 int lenP;
 int *p;
};
```

同样考虑下面的语句（参见图 15.15）：

```
pointerDataClass objectOne;
pointerDataClass objectTwo;
```

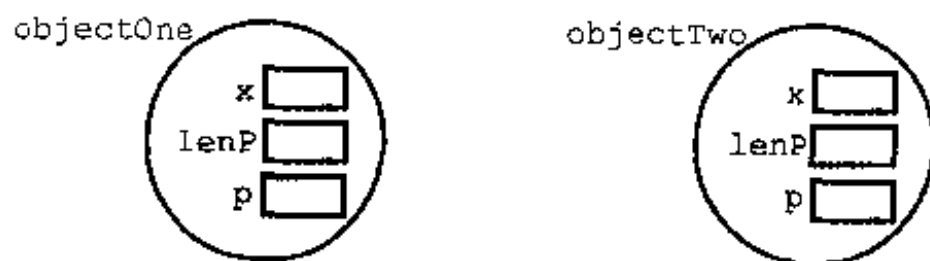


图 15.15 objectOne 和 objectTwo 对象

### 15.10.1 析构函数

objectOne 对象有一个指针成员变量 p。假设在程序执行时 p 创建了一个动态数组。当 objectOne 超出其作用域时，所有成员变量都被销毁。然而，p 创建了一个动态数组，而动态内存必须使用 delete 运算符释放。因此，如果指针 p 不使用 delete 运算符释放动态数组，动态数组的存储空间将保持已分配的标志，尽管不能访问它。如何保证当 p 被销毁时，p 创建的动态存储区域同样被销毁呢？假设 objectOne 如图 15.16 所示。

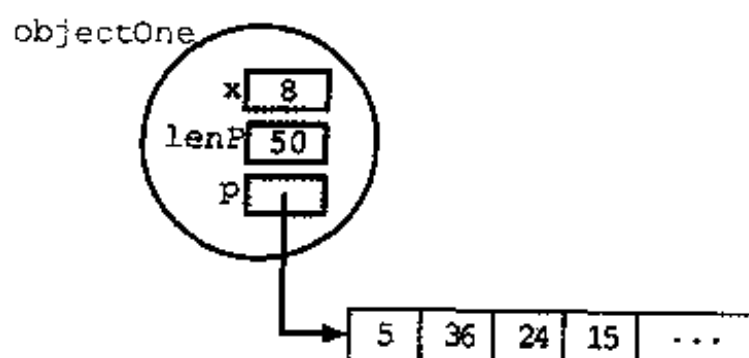


图 15.16 objectOne 对象和它的数据

如果一个类有析构函数，只要类的对象超出作用域，析构函数将自动地执行（见第 13 章）。因此，可以把必要的代码放在析构函数中，以保证当 objectOne 超出作用域时，p 创建的内存被释放掉。例如，pointerDataClass 类的析构函数定义为：

```

pointerDataClass::~~pointerDataClass()
{
 delete [] p;
}

```

当然，在类的定义中必须包括析构函数作为类的成员。可以通过包括析构函数扩展 `pointerDataClass` 类的定义。此外，本节的剩余部分假定析构函数的定义如前所述，即析构函数释放 `p` 指向的存储空间。

```

class pointerDataClass
{
public:
 ~pointerDataClass();
 ...
private:
 int x;
 int lenP;
 int *p;
};

```

**注意：**为了让析构函数正确地起作用，指针 `p` 必须包含有效的值。如果 `p` 没有被正确地初始化（`p` 的值是无意义的）而析构函数被执行，程序将会异常终止或者析构函数释放了无关的存储空间。所以，使用指针时必须加倍小心。

### 15.10.2 赋值运算符

本节讨论带指针成员变量的类其内建赋值运算符的局限性。假设 `objectOne` 和 `objectTwo` 如图 15.17 所示。

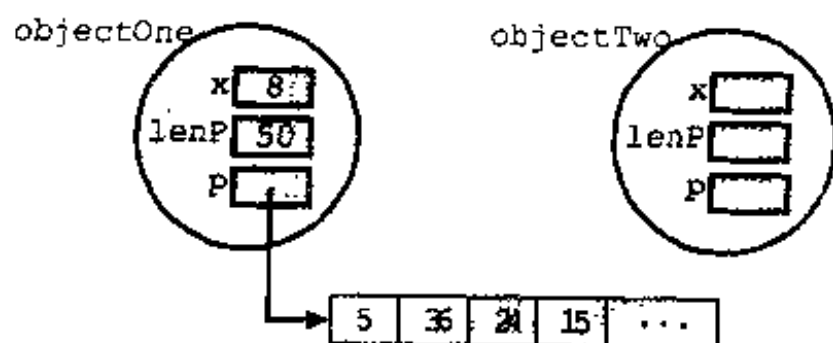


图 15.17 `objectOne` 和 `objectTwo` 对象

赋值运算符是类的内建运算符之一。例如，语句：

```
objectTwo = objectOne;
```

将 `objectOne` 的成员变量复制到 `objectTwo` 中。就是说，`objectOne.x` 的值被复制到 `objectTwo.x`，`objectOne.p` 的值被复制到 `objectTwo.p`。因为 `p` 是指针，这种以成员为单位的逐一数据复制将导致数据的浅复制。`objectTwo.p` 和 `objectOne.p` 都指向相同的存储空间，如图 15.18 所示。

现在如果 `objectTwo.p` 释放它所指向的存储空间，`objectOne.p` 将成为无效指针。这种情况可能会经常发生，如果 `pointerDataClass` 类有一个析构函数，当 `pointerDataClass` 类型的对象超出它的作用域时，释放 `p` 所指向的存储空间。因此必须有一定方法避免这一隐患。为了避免带指针成员变量的类导致数据浅复制，C++ 允许程序员扩展赋值运算符的定义。这一过程称为赋值运算符的重载。在第 16 章中解释了如何通过运算符重载完成这一工作。一旦赋值运算符被正确地重载，`objectOne` 和 `objectTwo` 对象都会正确地包含自己的数据，如图 15.19 所示。



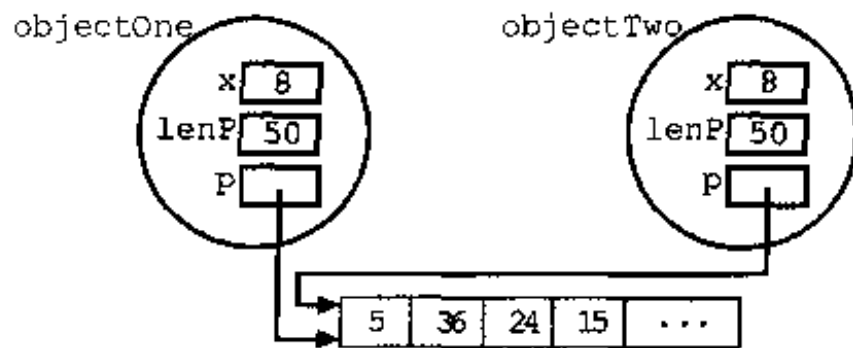


图 15.18 在“objectTwo = objectOne;”执行后的 objectOne 对象和 objectTwo 对象

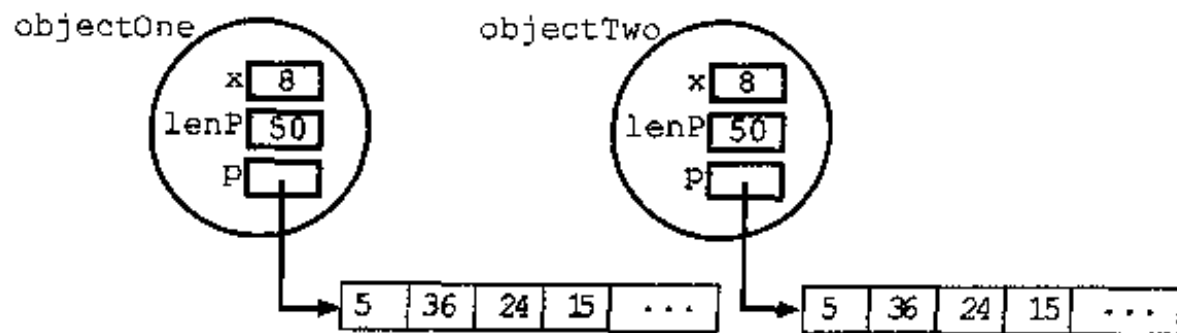


图 15.19 objectOne 和 objectTwo 对象

### 15.10.3 复制构造函数

当声明一个类对象时，可以使用一个相同类型的、已有对象的值对它进行初始化。例如，考虑下面的语句：

```
pointerDataClass objectThree(objectOne);
```

objectThree 对象被声明，同时使用 objectOne 的值进行初始化。即 objectOne 成员变量的值复制到 objectThree 相应的成员变量中。该初始化过程称为默认按成员初始化。默认按成员初始化源自于构造函数，称为复制构造函数（由编译器提供）。就像赋值运算符的情况一样，因为 pointerDataClass 类包含指针成员变量，该默认初始化过程会导致如图 15.20 所示的数据浅复制。（假定 objectOne 如前所述。）

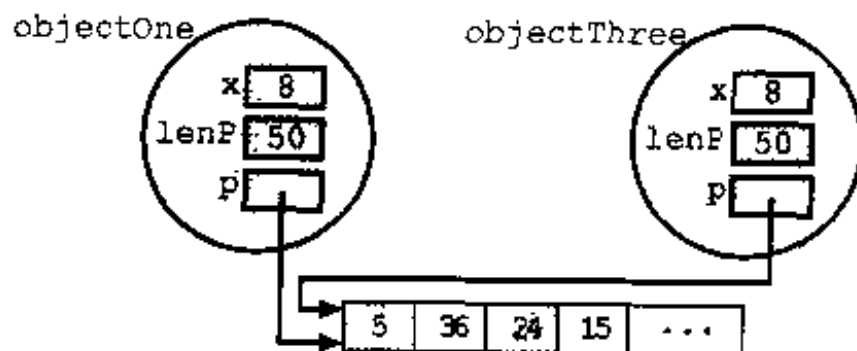


图 15.20 objectOne 对象和 objectThree 对象

在描述如何克服这一缺陷之前，让我们再描述一种会导致数据浅复制的情形。对这两种问题的解决方案是一样的。

作为函数的参数时，类对象可以以引用或值的方式传递。记住，pointerDataClass 类包含了释放 p 所指向存储空间的析构函数。假设 objectOne 如图 15.21 所示。

考虑下面的函数原型：

```
void destroyList(pointerDataClass paramObject);
```

函数 destroyList() 有一个值参数 paramObject。现在考虑下面的语句：

```
destroyList(objectOne);
```

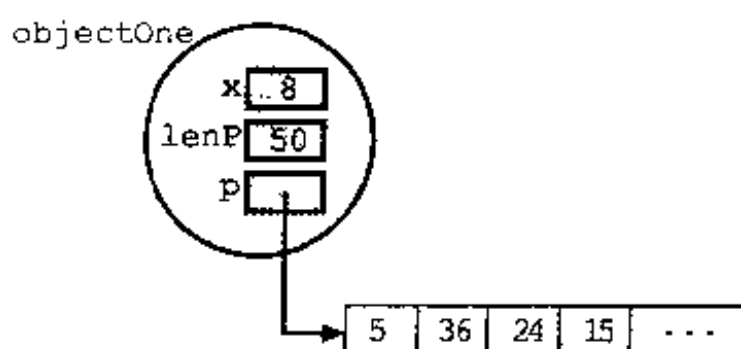


图 15.21 objectOne 对象

在该语句中，objectOne 作为 destroyList() 函数的参数。因为 paramObject 为值参数，复制构造函数复制 objectOne 的成员变量到 paramObject 对应的成员变量中。和前面的情况一样，paramObject.p 和 objectOne.p 将指向相同的存储空间，如图 15.22 所示。

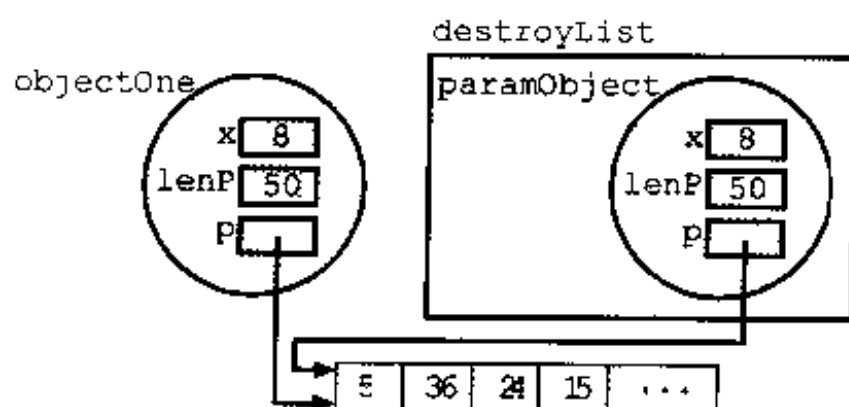


图 15.22 objectOne 和 paramObject 对象中指向相同数组的指针成员变量

因为 objectOne 以值的方式传递，paramObject 的成员变量应该有它们自己的数据副本。特别地，paramObject.p 应该有它自己的存储空间。如何保证确实如此呢？

如果一个类有指针成员变量：

- 在对象声明时，一个对象的初始化使用另一个对象的值，如果允许按成员数据复制的方式，将导致数据的浅复制。
- 如果一个对象作为参数按值传递，而且允许按成员数据复制的方式，将导致数据的浅复制。

在这两种情况中，为了强制一个对象拥有它自己的数据副本，必须重载编译器提供的复制构造函数的定义。就是说，必须提供自己的复制构造函数的定义。这通常是通过在类的定义中放置一条包括复制构造函数的语句，然后写出复制构造函数的定义完成的。那么，无论何时需要执行复制构造函数，系统将执行提供的复制构造函数的定义，而不是编译器所提供的。所以，对于 pointerDataClass 类来说，可以通过在 pointerDataClass 类中包括复制构造函数而克服数据浅复制的问题。例 15.5 阐明了这一点。

复制构造函数在两种情况下自动地执行（如前所述）：

- 当一个对象被声明且使用另一个对象的值初始化时
- 当一个对象作为值参数传递时

因此，一旦 pointerDataClass 的复制构造函数正确地定义，objectOne.p 和 objectThree.p 都将有它们自己的数据副本。类似地，objectOne.p 和 paramObject.p 将有它们自己的数据副本，如图 15.23 所示。

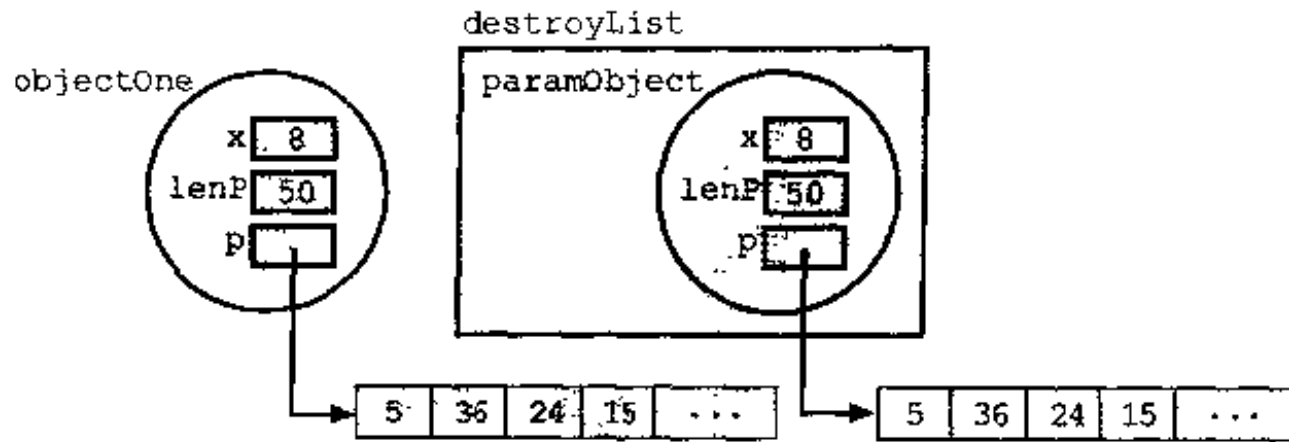


图 15.23 拥有自己的数据的 objectOne 和 paramObject 对象的指针成员变量

当函数 destroyList() 退出时, 形参 paramObject 超出了作用域, paramObject 对象的析构函数释放 paramObject.p 所指向的存储空间。然而, 释放过程对 objectOne 没有效果。

在一个类的定义中包含复制构造函数的通常语法是:

```
className(const className& otherObject);
```

例 15.5 阐述了如何在一个类中编写构造函数以及它的工作方式。

### 例 15.5

考虑下面的类:

```
class pointerDataClass
{
public:
 void print() const;
 //Function to output the value of x and
 //the value of the array p
 void setData();
 //Function to input data into x and
 //into the array p
 void destroyP();
 //Function to deallocates the memory space
 //occupied by the array p

 pointerDataClass(int sizeP = 10);
 //constructor
 //Creates an array of the size specified by the
 //parameter sizeP; the default array size is 10

 ~pointerDataClass();
 //destructor
 //deallocates the memory space occupied by the array p

 pointerDataClass (const pointerDataClass& otherObject);
 //copy constructor

private:
 int x;
 int lenP;
 int *p; //pointer to an int array
};
```

假设 pointerDataClass 类成员的定义如下:

```
void pointerDataClass::print() const
{
 cout<<"x = "<<x<<endl;
```

```
 cout<<"p = ";
 for(int i = 0; i < lenP; i++)
 cout<<p[i]<<" ";
 cout<<endl;
 }

 void pointerDataClass::setData()
 {
 cout<<"Enter an integer for x: ";
 cin>>x;
 cout<<endl;

 cout<<"Enter "<<lenP<<" numbers: ";
 for(int i = 0; i < lenP; i++)
 cin>>p[i];

 cout<<endl;
 }

 void pointerDataClass::destroyP()
 {
 lenP = 0;
 delete [] p;
 p = NULL;
 }

 pointerDataClass::pointerDataClass(int sizeP)
 {
 x = 0;
 if(sizeP <= 0)
 {
 cout<<"Array size must be positive"<<endl;
 cout<<"Creating an array of size 10"<<endl;

 lenP = 10;
 }
 else
 lenP = sizeP;

 p = new int[lenP];
 }

 pointerDataClass::~pointerDataClass()
 {
 delete [] p;
 }

 //copy constructor
 pointerDataClass::pointerDataClass
 (const pointerDataClass& otherObject)
 {
 x = otherObject.x;
 lenP = otherObject.lenP;
 p = new int[lenP];

 for(int i = 0; i < lenP; i++)
 p[i] = otherObject.p[i];
 }
```

考虑下面的函数 main()。假定 pointerDataClass 类的定义在头文件 ptrDataClass.h 中。

```
#include <iostream>
#include "ptrDataClass.h"

using namespace std;
void testCopyConst(pointerDataClass temp);

int main()
{
 pointerDataClass one(5); //Line 1
 one.setData(); //Line 2
 cout<<"Line 3: ###Object one's data###"<<endl; //Line 3
 one.print(); //Line 4
 cout<<"Line 5: _____"
 <<"_____ "<<endl; //Line 5
 pointerDataClass two(one); //Line 6
 cout<<"Line 7: ^^Object two's data^^"<<endl; //Line 7
 two.print(); //Line 8
 cout<<"Line 9: _____"
 <<"_____ "<<endl; //Line 9
 two.destroyP(); //Line 10
 cout<<"Line 11: ~~~ Object one's data after "
 <<"destroying object two.p ~~~"<<endl; //Line 11
 one.print(); //Line 12
 cout<<"Line 13: _____"
 <<"_____ "<<endl; //Line 13
 cout<<"Line 14: Calling the function testCopyConst"
 <<endl; //Line 14
 testCopyConst(one); //Line 15
 cout<<"Line 16: _____"
 <<"_____ "<<endl; //Line 16
 cout<<"Line 17: After a call to the function "
 <<"testCopyConst, object one is:"<<endl; //Line 17
 one.print(); //Line 18
 return 0; //Line 19
}

void testCopyConst(pointerDataClass temp)
{
 cout<<"Line 20: *** Inside function "
 <<"testCopyConst ***"<<endl; //Line 20
 cout<<"Line 21: Object temp data:"<<endl; //Line 21
 temp.print(); //Line 22
 temp.setData(); //Line 23
 cout<<"Line 24: After changing the object "
 <<"temp, its data is: "<<endl; //Line 24
 temp.print(); //Line 25
 cout<<"Line 26: *** Exiting function "
 <<"testCopyConst ***"<<endl; //Line 26
}
```

运行实例：在本运行实例中，用户输入用阴影表示。

```

Enter an integer for x: 28
Enter 5 numbers: 2 4 6 8 10
Line 3: ###Object one's data###
x = 28
p = 2 4 6 8 10
Line 5:
Line 7: ^^^Object two's data^^^
x = 28
p = 2 4 6 8 10
Line 9:
Line 11: ~~~ Object one's data after destroying object two.p ~~~
x = 28
p = 2 4 6 8 10
Line 13:
Line 14: Calling the function testCopyConst
Line 20: *** Inside function testCopyConst ***
Line 21: Object temp data:
x = 28
p = 2 4 6 8 10
Enter an integer for x: 65
Enter 5 numbers: 1 3 5 7 9
Line 24: After changing the object temp, its data is:
x = 65
p = 1 3 5 7 9
Line 26: *** Exiting function testCopyConst ***
Line 16:
Line 17: After a call to the function testCopyConst, object one is:
x = 28
p = 2 4 6 8 10

```

下面描述 `main()` 函数是如何工作的。第 1 行语句创建一个 `pointerDataClass` 类型的对象 `one`。该语句还创建一个大小为 5 的数组并把数组的地址存储到 `one.p` 中。第 2 行到第 5 行语句输入数据到 `one` 中，同时输出 `one` 的数据。第 6 行语句创建对象 `two`，同时使用 `one` 的值初始化 `two`。复制构造函数复制 `one.x` 到 `two.x` 中，创建一个大小为 5 的数组，把数组的地址存储到 `two.p` 中，然后复制数组 `one.p` 到 `two.p` 中。就是说，`one.p` 和 `two.p` 都有它们自己的数据副本。第 8 行到第 13 行的语句阐明了这一点。例如，第 8 行语句输出 `two` 中的数据，第 11 行语句释放 `two.p` 所指向的存储空间，第 12 行语句输出 `one` 的值。类似，第 15 行到第 19 行语句阐明了当对象 `one` 作为值参数传递给函数 `testCopyConst()` 时，复制构造函数提供它自己的数据副本给形参 `test`。在函数 `testCopyConst()` 的定义内，`temp` 的值被改变，这对 `one` 并没有影响。此外，当函数 `testCopyConst()` 终止时，`pointerDataClass` 类的析构函数释放 `temp.p` 所占据的存储空间，这对 `one.p` 并没有影响。

对于带指针成员变量的类，通常要做三件事情：

1. 在类中包含析构函数。
2. 重载类的赋值运算符。
3. 包含复制构造函数。

第 16 章将讨论赋值运算符的重载。在此之前，当讨论带指针成员变量的类时，将只实现上述三项中的析构函数和复制构造函数。



## 15.11 继承、指针和虚函数

注意：可以跳过本节而不影响学习的连续性。

作为参数时，一个类对象可以用值方式或引用方式传递。前面的章节中还说明了实参和形参必须匹配。然而，在使用类的情况下，C++ 允许用户传递一个派生类的对象给一个基类类型的形参。首先，讨论当形参为引用参数或指针的情况。考虑下面的类：

```
class baseClass
{
public:
 void print();
 baseClass(int u = 0);

private:
 int x;
};

class derivedClass: public baseClass
{
public:
 void print();
 derivedClass(int u = 0, int v = 0);

private:
 int a;
};
```

baseClass 类有三个成员。derivedClass 类派生自 baseClass 类，它同样有三个自己的成员。这两个类都有一个成员函数 print()。假设这两个类的成员函数定义如下：

```
void baseClass::print()
{
 cout<<"In baseClass x = "<<x<<endl;
}

baseClass::baseClass(int u)
{
 x = u;
}

void derivedClass::print()
{
 cout<<"In derivedClass ***: ";
 baseClass::print();
 cout<<"In derivedClass a = "<<a<<endl;
}

derivedClass::derivedClass(int u, int v)
 : baseClass(u)
{
 a = v;
}
```

考虑下面的函数：

```
void callPrint(baseClass& p)
{
 p.print();
}
```

函数 `callPrint()` 有一个 `baseClass` 类型的形参 `p`。可以通过使用一个 `baseClass` 类型或 `derivedClass` 类型的对象作为参数调用函数 `callPrint()`。此外，函数 `callPrint()` 的函数体调用成员函数 `print()`。考虑下面的 `main()` 函数：

```
int main()
{
 baseClass one(5); //Line 1
 derivedClass two(3, 15); //Line 2

 one.print(); //Line 3
 two.print(); //Line 4

 cout<<"*** Calling the function callPrint ***"<<endl;
 //Line 5
 callPrint(one); //Line 6
 callPrint(two); //Line 7

 return 0;
}
```

输出：

```
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
*** Calling the function callPrint ***
In baseClass x = 5
In baseClass x = 3
```

考虑一下第6行和第7行语句。第6行语句调用函数 `callPrint()`，传递对象 `one` 作为参数，该语句产生第5行的输出。第7行语句同样调用函数 `callPrint()`，不过传递对象 `two` 为参数，该语句产生第6行的输出。第6行和第7行语句产生的输出只显示 `x` 的值，尽管每次都传递不同类型的对象作为参数。（因为在第7行中，对象 `two` 作为参数传递给函数 `callPrint()`，第7行语句产生的输出应该和输出结果的第2行和第3行相同。）因为这两条语句（第6行和第7行）都引起 `baseClass` 类的成员函数 `print()` 的执行，成员函数 `print()` 在 `callPrint()` 函数体中的绑定发生在编译时。因为 `callPrint()` 函数的形参 `p` 是 `baseClass` 类型，编译器把 `baseClass` 类的函数 `print()` 与语句 “`p.print();`” 相结合。具体而言，在编译期绑定时，调用特定函数所必需的代码是由编译器生成的。（编译期绑定也称为静态绑定。）

对于第7行语句，实参为 `derivedClass` 类型。因此，当 `two` 的函数体执行时，逻辑上 `two` 对象的 `print()` 函数应该执行，但实际上不是这样。那么，在程序执行当中，C++ 是如何纠正这一问题并产生正确的函数调用呢？C++ 通过提供虚函数的机制纠正这一问题。虚函数的绑定发生在程序执行期，而不是在编译期。这种形式的绑定称为运行期绑定。在运行期绑定中，编译器不生成调用一个特定函数的代码，而是生成足够的信息以使运行期系统能够产生特定的代码以执行相应的函数调用。运行期绑定也称为动态绑定。

在 C++ 中，虚函数是使用保留字 `virtual` 声明的。下面使用这一特征重定义前面的类：

```
class baseClass
{
public:
 virtual void print(); //virtual function
 baseClass(int u = 0);

private:
 int x;
};
```



```

class derivedClass: public baseClass
{
public:
 void print();
 derivedClass(int u = 0, int v = 0);
private:
 int a;
};

```

注意，只需要在基类中声明一个虚函数。

成员函数 print() 的定义和前面相同。执行修改后的程序，输出如下。

输出：

```

In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
*** Calling the function callPrint ***
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15

```

输出表明，对于第 7 行语句，derivedClass 的 print() 函数被执行（见输出的最后两行）。

前面的讨论同样适用于类的指针作为形参而派生类的指针作为实参传递的情况。为了阐明这一特征，假设有上述的类。（假定 baseClass 类的定义在头文件 baseClass.h 中，derivedClass 类的定义在头文件 derivedClass.h 中。）考虑下面的程序：

```

//Chapter 15: Virtual Functions
#include <iostream>
#include "derivedClass.h"
using namespace std;
void callPrint(baseClass *p);
int main()
{
 baseClass *q; //Line 1
 derivedClass *r; //Line 2

 q = new baseClass(5); //Line 3
 r = new derivedClass(3,15); //Line 4

 q->print(); //Line 5
 r->print(); //Line 6

 cout<<"*** Calling the function callPrint ***"
 <<endl; //Line 7
 callPrint(q); //Line 8
 callPrint(r); //Line 9

 return 0;
}

void callPrint(baseClass *p)
{
 p->print();
}

```

输出：

```

In baseClass x = 5

```

```

In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
*** Calling the function callPrint ***
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15

```

前面的例子表明, 如果一个形参 *p* 是引用参数或指针, 且 *p* 使用基类的虚函数, 可以有效地传递一个派生类对象给 *p* 作为实参。然而, 如果 *p* 为值参数, 这种传递一个派生类对象给 *p* 作为实参的机制不起作用, 即使 *p* 使用虚函数。如果一个形参为值参数, 实参的值被复制到形参中。所以, 如果形参为类, 实参的成员变量被复制到形参的对应成员变量中。

假设有上面的类 *baseClass* 和 *derivedClass*, 考虑下面的函数定义:

```

void callPrint(baseClass p) //p is a value parameter
{
 p.print();
}

```

假设有下面的声明:

```

derivedClass two;

```

对象 *two* 有两个成员变量 *x* 和 *a*, 成员变量 *x* 从基类继承而来。考虑下面的函数调用:

```

callPrint(two);

```

在该语句中, 因为形参 *p* 为值参数, *two* 的成员变量被复制到 *p* 的成员变量中。然而, 因为 *p* 是一个 *baseClass* 类型的对象, 它只有一个成员变量。结果, 只有 *two* 的成员变量 *x* 会复制到 *p* 的成员变量 *x* 中。此外, 函数体中的语句:

```

p.print();

```

将导致 *baseClass* 类成员函数 *print()* 的执行。

下列程序的输出进一步阐明了这一概念。假定 *baseClass* 类的声明在头文件 *baseClass.h* 中, *derivedClass* 类的声明在头文件 *derivedClass.h* 中。

```

//Chapter 15: Virtual functions and value parameters
#include <iostream>
#include "derivedClass.h"
using namespace std;
void callPrint(baseClass p);
int main()
{
 baseClass one(5); //Line 1
 derivedClass two(3, 15); //Line 2
 one.print(); //Line 3
 two.print(); //Line 4
 cout<<"*** Calling the function callPrint ***"
 <<endl; //Line 5
 callPrint(one); //Line 6
 callPrint(two); //Line 7
 return 0;
}

```

```
void callPrint(baseClass p) //p is a value parameter
{
 p.print();
}
```

输出:

```
In baseClass x = 5
In derivedClass ***: In baseClass x = 3
In derivedClass a = 15
*** Calling the function callPrint ***
In baseClass x = 5
In baseClass x = 3
```

仔细观察第6行和第7行语句的输出(输出的最后两行)。在第7行中,因为形参 $p$ 为值参数,类 $two$ 的成员变量复制到 $p$ 的成员变量中。然而,因为 $p$ 为 $baseClass$ 类型的对象,它仅有一个成员变量。所以,只有 $two$ 的成员变量 $x$ 复制到 $p$ 的成员变量 $x$ 中。此外,函数 $callPrint()$ 中的语句“ $p.print();$ ”执行 $baseClass$ 类的 $print()$ 函数,而不是 $derivedClass$ 类中的 $print()$ 函数。所以,输出的最后一行只显示 $x$ 的值( $two$ 的成员变量)。

注意:基类类型的对象不能传递给派生类类型的形参。

### 15.11.1 类和虚析构函数

带指针成员变量的类必须有一个析构函数。当类对象超出其作用域时,析构函数自动地执行。因此,如果类对象创建了动态对象,可以设计析构函数来释放这些对象的存储空间。如果一个派生类对象传递给一个基类类型的形参,基类的析构函数将被执行,无论派生类对象是以值还是引用的方式传递。然而,逻辑上,当派生类对象超出作用域时,应该执行派生类的析构函数。

为了纠正这一问题,基类的析构函数必须是虚函数。基类的虚析构函数自动地使派生类的析构函数成为虚函数。当一个派生类对象传递给一个基类类型的形参,那么当该对象超出作用域时,派生类的析构函数被执行。当执行了派生类的析构函数后,基类的析构函数执行。所以,当派生类对象销毁后,派生类对象的基类部分(就是从基类继承而来的成员)也被销毁。

如果基类包含虚函数,应该使基类的析构函数也成为虚函数。

## 15.12 取址运算符和类

本章使用了取址运算符 $\&$ 来存储一个变量的地址到指针变量中。取址运算符还被用来创建另一个对象的别名。考虑下面的语句:

```
int x;
int &y = x;
```

第1条语句声明 $x$ 为整型变量,第2条语句声明 $y$ 为 $x$ 的别名。就是说, $x$ 和 $y$ 都引用相同的存储单元。因此, $y$ 就像一个常量指针变量。语句:

```
y = 25;
```

设置 $y$ 的值(也就是 $x$ 的值)为25。类似地,语句:

```
x = 2 * x + 30;
```

更新x的值（也就是y的值）。

取址运算符还可以被用来返回一个类的私有成员变量的地址。然而，如果不小心的话，该操作可能会导致严重的程序错误。下面的例子有助于阐明这一观点。

考虑下面的类定义：

```
//header file testadd.h
#ifndef H_testAdd
#define H_testAdd

class testAddress
{
public:
 void setX(int);
 void printX() const;
 int& addressOfX(); //this function returns the
 //address of the private data member

private:
 int x;
};

#endif
```

实现成员函数的函数定义如下：

```
//Implementation file testAdd.cpp
#include <iostream>
#include "testAdd.h"
using namespace std;

void testAddress::setX(int inX)
{
 x = inX;
}

void testAddress::printX() const
{
 cout<<x;
}

int& testAddress::addressOfX()
{
 return x;
}
```

因为函数 addressOfX() 的返回类型为整型存储单元的地址，语句：

```
return x;
```

返回x的地址。

下面，编写一个使用 testAddress 类的简单程序，然后阐述会出现什么问题。稍后，将显示如何解决该问题。

```
//Test program
#include <iostream>
#include "testAdd.h"
using namespace std;

int main()
{
```

```

 testAddress a;
 int &y = a.addressOfX();

 a.setX(50);
 cout<<"x in class testAddress = ";
 a.printX();
 cout<<endl;

 y = 25;
 cout<<"After y = 25, x in class testAddress = ";
 a.printX();
 cout<<endl;

 return 0;
}

```

输出:

```

x in class testAddress = 50
After y = 25, x in class testAddress = 25

```

在上面的程序中, 当语句:

```
int &y = a.addressOfX();
```

执行后, y 成为对象 a 的私有成员变量 x 的别名。因此, 语句:

```
y = 25;
```

改变了 x 的值。

在第 13 章中说明了私有成员变量在类外是不可访问的。然而, 通过返回它们的地址, 程序员可以操作它们。解决这一问题的一种办法是不提供私有成员变量的地址给类的使用者。然而, 在某些时候返回一个私有成员变量的地址是必需的, 将在下一章中看到这种情况。如何防止程序直接操作私有成员变量呢? 为了解决这一问题, 可以在函数的返回类型前使用保留字 const。这么做仍然可以返回私有成员变量的地址, 但是同时防止了程序员直接操作私有成员变量。下面使用这一特征重写 testAddress 类。

```

#ifndef H_testAdd
#define H_testAdd

class testAddress
{
public:
 void setX(int);
 void printX() const;
 const int& addressOfX(); //this function returns the
 //address of the private data
 //member

private:
 int x;
};
#endif

```

函数 addressOfX() 在实现文件中的定义为:

```

const int& testAddress::addressOfX()
{
 return x;
}

```

同样的程序现在将产生一个编译期错误。

### 15.13 小结

1. 指针变量包含其他变量的地址作为它们的值。
2. 在 C++ 中, 没有名字和指针变量类型相关联。
3. 指针变量是通过在数据类型和变量之间使用 \* 来声明的。例如, 语句:  

```
int *p;
char *ch;
```

声明 p 和 ch 为指针变量。p 的值指向 int 类型的存储空间, 而 ch 的值指向 char 类型的存储空间。通常, p 称为指向 int 类型的指针变量, ch 称为指向 char 类型的指针变量。
4. 在 C++ 中, & 称为取址运算符。
5. 取址运算符返回其操作数的地址。例如, 如果 p 是一个指向 int 类型的指针变量, num 是一个 int 型变量, 语句:  

```
p = #
```

设置 p 的值为 num 的地址。
6. 当用做一元运算符时, \* 称为取值运算符。
7. 指针变量指明的存储单元通过使用取值运算符 \* 来访问。例如, 如果 p 为指向 int 类型的指针变量, 语句:  

```
*p = 25;
```

设置 p 值所指向存储单元的值为 25。
8. 可以使用成员访问运算符 -> 来访问指针所指向对象的成员。
9. 指针变量使用 0 (整数零)、NULL 或相同类型变量的地址来初始化。
10. 可以直接赋给指针变量的惟一数值是 0。
11. 允许作用于指针变量的惟一算术运算为递增 (++)、递减 (--), 增加一个整数值、减少一个整数值和减去另一个指针变量的值。
12. 指针算术不同于普通算术。当指针增加一个整数值时, 指针变量所增加的值为指针所指对象大小的整数倍。类似地, 当指针减去一个整数值时, 指针变量所减少的值为指针所指对象大小的整数倍。
13. 指针变量可以使用关系运算符进行比较。(比较相同类型的指针是有意义的。)
14. 指针变量的值可以赋给相同类型的另一个指针变量。
15. 程序执行当中创建的变量称为动态变量。
16. 运算符 new 用来创建动态变量。
17. 运算符 delete 用来释放动态变量所占据的内存。
18. 在 C++ 中, new 和 delete 都是保留字。
19. 运算符 new 有两种形式: 一种创建单个的动态变量, 另一种创建动态变量数组。
20. 如果 p 为指向 int 类型的指针变量, 语句:  

```
p = new int;
```

在内存的某处分配 int 类型的存储空间, 然后存储所分配存储空间的地址到 p 中。
21. 运算符 delete 有两种形式: 一种释放单个动态变量占据的内存, 另一种释放动态变量数组占据的内存。
22. 如果 p 为指向 int 类型的指针, 语句:



```
delete p;
```

释放 p 所指向的内存。

23. 数组名是一个常量指针。它总是指向相同的存储单元，即第一个数组成员的地址。

24. 为了创建一个动态数组，可以使用创建动态变量数组的 new 运算符形式。例如，如果 p 为指向 int 类型的指针，语句：

```
p = new int[10];
```

创建一个包括 10 个成员的 int 类型数组。数组的地址存储在 p 中。p 称为动态数组。

25. 数组下标可以用来访问动态数组的成员。例如，假设 p 是一个包括 10 个成员动态数组。那么 p[0] 引用第一个数组成员，p[1] 引用第二个数组成员，等等。具体而言，p[i] 引用第 [i+1] 个数组成员。

26. 在程序执行当中创建的数组称为动态数组。

27. 如果 p 为动态数组，那么语句：

```
delete [] p;
```

释放 p 占据的内存——就是 p 的内容。

28. 在浅复制中，两个或多个相同类型的指针指向相同的存储空间。就是说，它们指向相同的数据。

29. 在深复制中，两个或多个相同类型的指针有它们自己的数据副本。

30. 如果类有析构函数，当类对象超出作用域时，析构函数自动地执行。

31. 如果类有指针成员变量，内建的赋值运算符将导致数据的浅复制。

32. 当一个对象被声明且使用其他对象的值进行初始化，或当一个对象以值方式传递给一个参数时，复制构造函数被执行。

33. C++ 允许用户传递一个派生类对象给一个基类类型的形参。

34. 虚函数的绑定发生在执行期，而不是在编译期，这称为动态或运行期绑定。

35. 在 C++ 中，虚函数使用保留字 virtual 声明。

36. 取址运算符可以用来返回类的私有成员变量的地址。

## 15.14 练习

1. 判断下列语句的正确性。

a. 在 C++ 中，pointer 为保留字。

b. 在 C++ 中，指针变量使用保留字 pointer 来声明。

c. 语句“delete p;”释放指针变量 p。

d. 语句“delete p;”释放 p 所指向的动态变量。

e. 假设声明：

```
int list[10];
```

```
int *p;
```

语句：

```
p = list;
```

在 C++ 中是有效的。

f. 假设声明：

```
int *p;
```

语句：

```
p = new int[50];
```

动态地分配一个包括 50 个成员的 `int` 类型数组, `p` 包含该数组的地址。

g. 取址运算符返回其操作数的地址和值。

h. 如果 `p` 为指针变量, 那么语句 “`p = p * 2;`” 在 C++ 中是有效的。

2. 假设声明:

```
int x;
int *p;
int *q;
```

判断下列语句的有效性。如果语句无效, 请解释原因。

- a. `p = q;`
- b. `*p = 56;`
- c. `p = x;`
- d. `*p = *q;`
- e. `q = &x;`
- f. `*p = q;`

3. 下列 C++ 代码的输出是什么?

```
int x;
int y;
int *p = &x;
int *q = &y;
*p = 35;
*q = 98;
*p = *q;
cout<<x<<" "<<y<<endl;
cout<<*p<<" "<<*q<<endl;
```

4. 下列 C++ 代码的输出是什么?

```
int x;
int y;
int *p = &x;
int *q = &y;
x = 35; y = 46;
p = q;
*p = 78;
cout<<x<<" "<<y<<endl;
cout<<*p<<" "<<*q<<endl;
```

5. 假设声明:

```
int num = 6;
int *p = #
```

下列语句中的哪一条增加了 `num` 的值?

- a. `p++;`
- b. `(*p)++;`
- c. `num++;`
- d. `(*num)++;`

6. 下列代码的输出是什么?

```
int *p;
int *q;
p = new int;
q = p;
*p = 46;
*q = 39;
cout<<*p<<" "<<*q<<endl;
```



7. 下列代码的输出是什么?

```
int *p;
int *q;
p = new int;
*p = 43;
q = p;
*q = 52;
p = new int;
*p = 78;
q = new int;
*q = *p;
cout<<*p<<" "<<*q<<endl;
```

8. 下列代码有何错误?

```
int *p; //Line 1
int *q; //Line 2

p = new int; //Line 3
*p = 43; //Line 4

q = p; //Line 5
*q = 52; //Line 6

delete q; //Line 7

cout<<*p<<" "<<*q<<endl; //Line 8
```

9. 下列代码的输出是什么?

```
int x ;
int *p ;
int *q ;
p = new int[10] ;
q = p ;
*p = 4 ;
for(int j = 0; j < 10; j++)
{
 x = *p ;
 p++;
 *p = x + j ;
}

for(int k = 0; k < 10; k++)
{
 cout<<*q<<" ";
 q++;
}

cout<<endl;
```

10. 下列代码的输出是什么?

```
int *secret;
int j;

secret = new int[10] ;
secret[0] = 10 ;
for(j = 1; j < 10; j++)
 secret[j] = secret[j -1] + 5 ;
for(j = 0; j < 10; j++)
 cout<<secret[j]<<" ";
cout<<endl;
```

11. 解释浅复制和深复制的区别。

12. 下列代码有何错误?

```
int *p; //Line 1
int *q; //Line 2

p = new int [5]; //Line 3
*p = 2; //Line 4

for(int i = 1; i < 5; i++) //Line 5
 p[i] = p[i-1] + i; //Line 6

q = p; //Line 6

delete [] p; //Line 7
for(int j = 0; j < 5; j++) //Line 8
 cout<<q[j]<<" "; //Line 9

cout<<endl; //Line 10
```

13. 下列代码的输出是什么?

```
int *p;
int *q;
int i;

p = new int [5];
p[0] = 5;

for(i = 1; i < 5; i++)
 p[i] = p[i-1] + 2 * i;

cout<<"Array p: ";
for(i = 0; i < 5; i++)
 cout<<p[i]<<" ";
cout<<endl;

q = new int[5];

for(i = 0; i < 5; i++)
 q[i] = p[4 - i];

cout<<"Array q: ";
for(i = 0; i < 5; i++)
 cout<<q[i]<<" ";

cout<<endl;
```

14. 复制构造函数的目的是什么?

15. 指出复制构造函数执行的两种情形。

16. 指出带指针成员变量的类应该考虑的三项原则。

17. 假设有下列的类, classA 和 classB:

```
class classA
{
public:
 virtual void print() const;
 void doubleNum();
 classA(int a = 0);
private:
 int x;
};

void classA::print() const
```

```
{
 cout<<"ClassA x: "<<x<<endl;
}
void classA::doubleNum()
{
 x = 2 * x;
}

classA::classA(int a)
{
 x = a;
}

class classB: public classA
{
public:
 void print() const;
 void doubleNum();
 classB(int a = 0, int b = 0);
private:
 int y;
};

void classB::print() const
{
 classA::print();
 cout<<"ClassB y: "<<y<<endl;
}

void classB::doubleNum()
{
 classA::doubleNum();
 y = 2 * y;
}

classB::classB(int a, int b)
 : classA(a)
{
 y = b;
}
```

下面的main()函数的输出是什么?

```
int main()
{
 classA *ptrA;
 classA objectA(2);
 classB objectB(3,5);

 ptrA = &objectA;
 ptrA->doubleNum();
 ptrA->print();
 cout<<endl;

 ptrA = &objectB;
 ptrA->doubleNum();
 ptrA->print();
 cout<<endl;
}
```

- ```
        return 0;
    }

```
18. 如果练习 17 中 `classA` 的定义由下面的定义代替, `main()` 函数的输出是什么?
- ```
class classA
{
public:
 virtual void print() const;
 virtual void doubleNum();
 classA(int a = 0);
private:
 int x;
};

```
19. 编译期绑定和运行期绑定的区别是什么?

### 15.15 编程练习

1. 使用动态数组重写第 9 章的编程练习 5。
2. 使用动态数组重写第 9 章的编程练习 6。
3. 使用动态数组重写第 9 章的编程练习 7。必须要求用户输入候选者的数量, 然后创建适当的数组来保存数据。

## 第 16 章 重载和模板

### 本章要点:

- 学习重载
- 认识对运算符重载的限制
- 研究 this 指针
- 学习友元函数
- 研究类的成员和非成员
- 研究如何重载各种不同的运算符
- 学习模板
- 研究如何构造函数模板和类模板

在第 14 章中,学习了 C++ 的类是如何把数据以及作用于数据的操作结合到一个单一实体中的。把数据和作用于数据的操作结合起来的能力称为封装。它是面向对象设计 (OOD) 的第一个原则。在第 13 章中定义了抽象数据类型 (ADT), 描述了 C++ 的类是如何实现 ADT 的。第 14 章讨论了如何通过继承的机制从已有类派生出新的类。继承是 OOD 的第二个原则, 有利于代码重用。

本章的第一个主题是运算符重载, 第二个主题是模板。模板使程序员能够为相关联的函数和类写出通用的代码。在本章中, 将通过函数模板的使用简化函数重载。

### 16.1 为什么需要运算符重载

第 13 章定义和实现了 clockType 类, 还显示了如何在程序中使用 clockType 类来表示一天的时间。现在回顾一下 clockType 类的一些特征。

考虑下面的语句:

```
clockType myClock(8,23,34);
clockType yourClock(4,5,30);
```

第 1 条语句声明 myClock 为 clockType 类型的对象, 初始化 myClock 的成员变量 hr、min 和 sec 的值分别为 8、23 和 34。第 2 条语句声明 yourClock 为 clockType 类型的对象, 初始化 yourClock 的成员变量 hr、min 和 sec 的值分别为 4、5 和 30。

现在考虑下面的语句:

```
myClock.printTime();
myClock.incrementSeconds();
if(myClock.equalTime(yourClock))
. . .
```

第 1 条语句以 hr: min: sec 的格式打印 myClock 的值。第 2 条语句将 myClock 的值增加一秒。第 3 条语句检查 myClock 的值是否和 yourClock 的值相同。

然而, 如果可以使用插入运算符<<来输出 myClock 的值, 使用递增运算符++把 myClock 的值增加一秒, 使用关系运算符进行比较, 将可以大大地增强 C++ 的灵活性。具体而言, 可以使用下面的语句而不是前面的语句:

```
cout<<myClock;
myClock++;
if(myClock == yourClock)
.
.
.
```

作用于类的惟一内建运算符为赋值运算符和成员选择运算符。所以, 其他运算符不能直接应用于类对象。然而, C++ 允许程序员扩展大部分运算符的定义, 使得运算符——例如关系运算符、算术运算符用于数据输出的插入运算符以及用于数据输入的析取运算符——可以应用于类。在 C++ 术语中, 这种行为称为运算符重载。除了运算符重载之外, 本章还将讨论函数重载, 该内容在第 7 章中有过简单的介绍。

## 16.2 运算符重载

回忆一下算术运算符/是如何运作的。如果/的两个操作数都是整数, 结果为整数; 否则, 结果为浮点数。类似地, 流插入运算符<<和流析取运算符>>也被重载了。运算符>>既用做流析取运算符也用做右移运算符。运算符<<既用做流插入运算符也用做左移运算符。这些都是运算符重载的例子。

运算符重载的其他例子是+和-。+和-的结果对于整数算术、浮点数算术和指针算术来说是不同的。

C++ 允许用户重载大部分运算符, 使得运算符可以在一个特定的应用程序中有效发挥起作用。C++ 不允许用户创建新的运算符。大部分的已有运算符可以被重载来操作类对象。

为了重载一个运算符, 必须编写相应的函数(就是函数头和函数体)。重载运算符的函数名为保留字 operator 后跟所重载的运算符。例如, 重载运算符>= 的函数名为:

```
operator>=
```

**运算符函数:** 重载运算符的函数。

### 16.2.1 运算符函数的语法

运算的结果为值, 所以, 运算符函数为返回值的函数。

运算符函数的头部语法为:

```
returnType operator operatorSymbol(formal parameter list)
```

在 C++ 中, operator 为保留字。

作用于类的惟一内建运算符为赋值(=)和成员选择。为了对类对象使用其他运算符, 它们必须被显式地重载。运算符重载为用户定义的数据类型提供了和内建数据类型一样的简明表达式。

为了对类重载一个运算符:

1. 在类的定义中包括声明重载运算符函数(就是运算符函数)的语句。
2. 编写出运算符函数的定义。



当在类的定义中包括一个运算符函数时，必须遵循某些特定的规则。这些规则在本章稍后的“作为成员函数和非成员函数的运算符函数”一节中描述。

### 16.2.2 重载运算符：一些限制

当重载运算符时，牢记下列规则：

1. 不能改变运算符的优先级。
2. 不能改变结合性。（例如，算术运算符的结合性为从左到右，这一点不能被改变。）
3. 默认参数不能和重载的运算符一起使用。
4. 不能改变运算符的操作数数量。
5. 不能创建新的运算符，只有已有运算符可以被重载。不能被重载的运算符包括：

`. * :: ? : sizeof`

6. 运算符作用于内建类型的含义保持不变。
7. 运算符可以被重载用于用户定义类型的对象以及用户定义类型对象与内建类型对象的组合。

### 16.2.3 this 指针

类的成员函数可以（直接地）访问该类一个特定对象的成员变量。有时需要成员函数引用一个对象的整体，而不是该对象的个别成员变量。如何在成员函数的定义中，特别是当对象没有作为一个参数传递时，引用对象的整体（作为一个单一的个体）呢？类的每一个对象维持一个（隐藏的）指向它自己的指针，该指针的名字为 `this`。在 C++ 中，`this` 为保留字。`this` 指针可供用户使用。当一个对象调用某成员函数时，成员函数引用该对象的 `this` 指针。例如，假设 `test` 是一个类，它有一个成员函数称为 `one()`。假设 `one()` 的定义如下：

```
test test::one()
{
 .
 .
 .
 return *this;
}
```

如果 `x` 和 `y` 为 `test` 类型的对象，那么语句：

```
y = x.one();
```

复制对象 `x` 的值到对象 `y` 中。就是说，`x` 的成员变量复制到 `y` 对应的成员变量中。当 `x` 对象调用函数 `one()` 时，成员函数 `one()` 定义中的 `this` 指针引用 `x` 对象，因此 `this` 表示 `x` 的地址而 `*this` 表示 `x` 的值。

下面的例子阐明了 `this` 指针是如何工作的。

#### 例 16.1

考虑下面的类：

```
class thisPointerClass
{
public:
 void set(int a, int b, int c);
 void print() const;
```

```

 thisPointerClass updateXYZ();
 //Post: x = 2 * x; y = y + 2;
 // z = z * z;

 thisPointerClass(int a = 0, int b = 0, int c = 0);

private:
 int x;
 int y;
 int z;
};

```

假设 thisPointerClass 类成员函数的定义如下:

```

void thisPointerClass::set(int a, int b, int c)
{
 x = a;
 y = b;
 z = z;
}

void thisPointerClass::print() const
{
 cout<<"x = "<<x
 <<" , y = "<<y
 <<" , z = "<<z<<endl;
}

thisPointerClass thisPointerClass::updateXYZ()
{
 x = 2 * x;
 y = y + 2;
 z = z * z;

 return *this;
}

```

函数 updateXYZ() 的定义更新 x、y 和 z 的值。使用 this 指针返回整个对象的值, 即成员变量 x、y 和 z 的值被返回。

```

thisPointerClass::thisPointerClass(int a, int b, int c)
{
 x = a;
 y = b;
 z = c;
}

```

考虑下面的 main() 函数:

```

int main()
{
 thisPointerClass object1(3, 5, 7); //Line 1
 thisPointerClass object2; //Line 2

 cout<<"Object 1: "; //Line 3
 object1.print(); //Line 4

 object2 = object1.updateXYZ(); //Line 5

 cout<<"After updating object1: "; //Line 6
 object1.print(); //Line 7
 cout<<"Object 2: "; //Line 8
}

```



```

 object2.print(); //Line 9
 return 0;
}

```

输出:

```

Object 1: x = 3, y = 5, z = 7
After updating object1: x = 6, y = 7, z = 49
Object 2: x = 6, y = 7, z = 49

```

输出的大部分是自解释的。第5行语句计算表达式 `object1.updateXYZ()` 的值, 该表达式更新 `object1` 对象成员变量的值。`object1` 的值随后由 `this` 指针返回, 如 `updateXYZ()` 函数的定义所示。然后赋值运算符复制 `object1` 的值到 `object2` 中。

### 例 16.2

在例13.9(第13章)中, 设计了一个类在程序中实现一个人的姓名。这里, 可以扩展 `personType` 类的定义单独地设置一个人的姓和名, 然后返回整个对象。`personType` 类的扩展定义如下所示:

```

class personType
{
public:
 void print() const;
 //Function to output the first name and last name
 //Post: The name is printed in the form firstName lastName

 void setName(string first, string last);
 //Function to set firstName and lastName according to
 //the parameters
 //Post: firstName = first; lastName = last;

 personType& setLastName(string last);
 //Function to set the last name
 //Post: lastName = last;
 //After setting the last name, a reference
 //to the object, that is, the address of the
 //object, is returned

 personType& setFirstName(string first);
 //Function to set the first name
 //Post: firstName = first;
 //After setting the first name, a reference
 //to the object, that is, the address of the
 //object, is returned

 void getName(string& first, string& last);
 //Function to return firstName and lastName via
 //the parameters
 //Post: first = firstName; last = lastName;

 personType(string first = "", string last = "");
 //Constructor with parameters
 //Set firstName and lastName according to the parameters
 //Post: firstName = first; lastName = last;

private:
 string firstName; //store the first name
 string lastName; //store the last name
};

```

注意在该personType类的定义中,使用了带默认参数的构造函数取代了默认构造函数和带参数的构造函数。

函数print()、setTime()、getName()和构造函数的定义和以前的一样(见例13.9)。函数setFirstName()和setLastName()的定义如下:

```
personType& personType::setLastName(string last)
{
 lastName = last;
 return *this;
}
personType& personType::setFirstName(string first)
{
 firstName = first;
 return *this;
}
```

考虑下面的main()函数:

```
int main()
{
 personType student1("Angela", "Clodfelter"); //Line 1
 personType student2; //Line 2
 personType student3; //Line 3
 cout<<"Line 4 -- Student 1: "; //Line 4
 student1.print(); //Line 5
 cout<<endl; //Line 6
 student2.setFirstName("Shelly").setLastName("Malik"); //Line 7
 cout<<"Line 8 -- Student 2: "; //Line 8
 student2.print(); //Line 9
 cout<<endl; //Line 10
 student3.setFirstName("Chelsea"); //Line 11
 cout<<"Line 12 -- Student 3: "; //Line 12
 student3.print(); //Line 13
 cout<<endl; //Line 14
 student3.setLastName("Tomek"); //Line 15
 cout<<"Line 16 -- Student 3: "; //Line 16
 student3.print(); //Line 17
 cout<<endl; //Line 18
 return 0;
}
```

输出:

```
Line 4 -- Student 1: Angela Clodfelter
Line 8 -- Student 2: Shelly Malik
Line 12 -- Student 3: Chelsea
Line 16 -- Student 3: Chelsea Tomek
```

第1、2和3行语句分别声明和初始化对象student1、student2和student3。对象student2和student3初始化为空字符串。第5行语句输出student1的值(见输出的第4行,该行包括第4、5和6行的输

出)。第7行语句的作用如下。在语句:

```
student2.setFirstName("Shelly").setLastName("Malik");
```

中, 首先表达式:

```
student2.setFirstName("Shelly")
```

被执行, 因为点运算符的结合性是从左到右。该表达式设置名为“Shelly”的对象, 然后返回对对象 student2 的引用。

因此, 下一个执行的表达式为:

```
student2.setLastName("Malik")
```

该表达式设置 student2 的姓为“Malik”。第9行语句输出 student2 的值。第11行语句设置 student3 对象的名为“Chelsea”, 第13行语句输出 student3 的值。注意第12行的输出, 该输出仅显示名, 没有姓, 因为还没有设置 student3 的姓。student3 的姓还是空的, 该值在第3行语句声明 student3 时设置。下面, 第15行语句设置 student3 的姓, 第16行语句输出 student3 的值。

#### 16.2.4 类的友元函数

在一个类作用域之外定义的函数称为该类的友元函数。友元函数是类的非成员函数, 但可以访问该类的私有成员变量。为了使一个函数成为某类的友元函数, 在(类定义中的)函数原型之前使用保留字 friend。(保留字 friend 仅出现在类定义中的函数原型中, 而不在友元函数的定义中。)

考虑下面的语句:

```
class classIllusFriend
{
 friend void two(...);
 .
 .
 .
};
```

在 classIllusFriend 类的定义中, two() 被声明为 classIllusFriend 的友元函数。就是说, 它是 classIllusFriend 类的非成员函数。当编写函数 two() 的定义时, 任何 classIllusFriend 类型的对象——two() 的局部变量或形参——都可以在函数 two() 的定义之内访问它自己的私有成员。(例 16.3 阐述了该概念。)此外, 因为友元函数不是类的成员, 它的声明可以被放置在类的私有、保护或公有部分。

##### 友元函数的定义

当编写友元函数的定义时, 友元函数头部的函数名之前没有类名和范围解释运算符。而且, 保留字 friend 不出现在友元函数定义的头部。因此, 函数 two() 的定义为:

```
void two(...){
 .
 .
 .
}
```

当然, 需要将友元函数的定义放在实现文件中。

下面一节阐述成员函数和非成员函数(友元函数)的区别, 同时将为一个特定类重载某些运算符。

下面的例子说明友元函数如何访问类的私有成员。

### 例 16.3

考虑下面的类：

```
class classIllusFriend
{
 friend void two(classIllusFriend cLFObject);

public:
 void print();
 void setx(int a);

private:
 int x;
};
```

在 classIllusFriend 类的定义中，two() 声明为友元函数。假设 classIllusFriend 类成员函数的定义如下：

```
void classIllusFriend::print()
{
 cout<<"In class classIllusFriend: x = "<<x<<endl;
}

void classIllusFriend::setx(int a)
{
 x = a;
}
```

现在考虑下面的 two() 函数定义：

```
void two(classIllusFriend cLFObject) //Line 1
{
 classIllusFriend localTwoObject; //Line 2
 localTwoObject.x = 45; //Line 3
 localTwoObject.print(); //Line 4
 cout<<endl; //Line 5
 cout<<"Line 6: In Friend Function two accessing "
 <<"private data member x "
 <<localTwoObject.x<<endl; //Line 6

 cLFObject.x = 88; //Line 7
 cLFObject.print(); //Line 8
 cout<<endl; //Line 9
 cout<<"Line 10: In Friend Function two accessing "
 <<"private data member x "
 <<cLFObject.x<<endl; //Line 10
}
```

函数 two() 包含一个形参 cLFObject 和一个局部变量 localTwoObject，均为 classIllusFriend 类型。在第 3 行语句中，localTwoObject 对象访问它的私有成员变量 x，设置它的值为 45。如果 two() 没有声明为 classIllusFriend 类的友元函数，该语句将导致语法错误，因为对象不能直接访问它的私有成员。类似地，在第 7 行语句中，形参 cLFObject 访问它的私有成员变量 x，设置它的值为 88。同样，如果 two 没有被声明为 classIllusFriend 类的友元函数，该语句将导致语法错误。第 6 行语句通过直接访问 localTwoObject 的私有成员变量 x 来输出它的值。类似地，第 10 行语句通过直接访问并输出

cLFObjct 的 x 值。函数 two() 还通过使用 print() 函数来打印 x 的值 (见第 4 行和第 8 行语句)。

现在考虑下面的 main() 函数定义:

```
int main()
{
 classIllusFriend aObject; //Line 11
 aObject.setx(32); //Line 12
 cout<<"Line 13: aObject.x: "; //Line 13
 aObject.print(); //Line 14
 cout<<endl; //Line 15
 cout<<"*~*~*~*~*~* Testing Friend Function "
 <<"two *~*~*~*~*~*"<<endl<<endl; //Line 16
 two(aObject); //Line 17
 return 0;
}
```

输出:

```
Line 13: aObject.x: In class classIllusFriend: x = 32
~~*~*~*~* Testing Friend Function two *~*~*~*~*~*
In class classIllusFriend: x = 45
Line 6: In Friend Function two accessing private data member x 45
In class classIllusFriend: x = 88
Line 10: In Friend Function two accessing private data member x 88
```

输出的大部分是自解释的。第 17 行语句调用函数 two() (classIllusFriend 类的友元函数), 传递 aObject 对象为实参。注意函数 two() 产生的最后 6 行输出, 包括两个空行。

### 16.2.5 作为成员函数和非成员函数的运算符函数

当在类的定义中包括运算符函数时, 必须遵循某些特定的规则。本节讨论这些规则。

大部分的运算符函数可以是成员函数或非成员函数 (友元函数)。为了使一个运算符函数成为类的成员或非成员函数, 请牢记下面的规则:

1. 为类重载()、[]、->或=中任何一个运算符函数必须声明为该类的成员函数。
2. 假设重载类 OpOverClass 的运算符 op:
  - a. 如果 op 最左面的操作数是一个不同类型的对象 (非 OpOverClass 类型), 为 OpOverClass 类重载运算符 op 的函数必须是非成员函数, 即 OpOverClass 类的友元函数。
  - b. 如果为 OpOverClass 类重载运算符 op 的运算符函数为 OpOverClass 类的成员函数, 当应用 op 到 OpOverClass 类型的对象时, op 最左面的操作数必须是 OpOverClass 类型。

当在类的定义中包括一个运算符函数时, 必须遵循这些规则。

在本章的稍后可以看到, 为类重载插入运算符<<和析取运算符>>的函数必须是非成员函数, 即该类的友元函数。

除了前面提到的某些特定运算符之外的运算符可以被重载为成员函数或非成员函数。下面的讨论说明了这两类函数的区别。

为了方便讨论, 将使用下面的类来阐明运算符重载:

```
class OpOverClass
{
 .
 .
 .
private:
 int a;
 int b;
};
```

OpOverClass类有两个int类型的私有成员变量a和b。将在重载运算符时对OpOverClass类增加运算符函数。

此外,假设有下列语句:

```
OpOverClass x;
OpOverClass y;
OpOverClass z;
```

x、y和z为OpOverClass类型的对象。

### 16.2.6 重载二元运算符

C++包括二元和一元运算符,同时还有三元运算符。现在讨论如何重载各种二元和一元运算符。

假设重载类OpOverClass的二元运算符+。该运算符可以被重载为OpOverClass类的成员函数或友元函数。下面将描述这两种重载该运算符的方法。

#### 重载+为成员函数

假设+重载为OpOverClass类的成员函数。为OpOverClass类重载+的函数名为:

```
operator+
```

因为x和y都是OpOverClass类型的对象,可以执行运算:

```
x + y
```

编译器把该表达式翻译为下面的表达式:

```
x.operator+(y)
```

在该表达式中,可以清楚地看到函数operator+只有一个参数,就是y。

因为operator+是OpOverClass类的成员,而x是OpOverClass类型的对象,在前面的语句中,operator+可以直接访问x对象的私有成员变量。因此,运算符+的第一个参数为调用函数operator+的对象,第二个参数传递为该函数的参数。

假设operator+把对象的成员变量相加。那么,表达式:

```
x + y
```

的结果为OpOverClass类型的对象。所以,运算符函数operator+的返回类型为OpOverClass类型。

下面,描述为OpOverClass类重载运算符+的函数原型和函数定义。

在OpOverClass类定义中的operator+函数原型为:

```
OpOverClass operator+(const OpOverClass&) const;
```

现在讨论如何编写OpOverClass类中operator+函数的定义。



函数 `operator+` 把对象的对应成员变量相加。例如，在表达式：

`x + y`

中，`x` 的成员变量 `a` 和 `y` 的成员变量 `a` 相加，`x` 的成员变量 `b` 和 `y` 的成员变量 `b` 相加。

因为 `operator+` 为 `OpOverClass` 类的成员，当编写 `operator+` 函数的定义时，类名和作用域运算符 `::` 必须出现在函数头部。`OpOverClass` 类中的 `operator+` 函数的定义为：

```
OpOverClass OpOverClass::operator+
 (const OpOverClass& otherObject) const
{
 OpOverClass temp;
 temp.a = a + otherObject.a;
 temp.b = b + otherObject.b;
 return temp;
}
```

在该函数定义中，把对象的成员变量和 `otherObject` 对象相应的成员变量相加。结果存储在局部变量 `temp` 中。最后，`temp` 对象的值被返回。注意 `operator+` 函数的返回类型为 `OpOverClass`。

类似地，可以重载其他二元算术运算符。

#### 重载二元（算术）运算符为成员函数的一般语法

下面描述重载二元运算符为类成员函数的一般函数形式。

函数原型（包括在类的定义中）：

```
returnType operator op(const className&) const;
```

这里 `op` 表示所要重载的二元运算符，`returnType` 为函数返回值的类型，`className` 为进行运算符重载的类名。

函数定义：

```
returnType className::operator op
 (const className& otherObject) const
{
 //algorithm to perform the operation
 return (value);
}
```

下面的例子阐述了如何重载和使用二元运算符。

#### 例 16.4

现在为 `OpOverClass` 类重载 `+` 和 `*`，这些运算符被重载为成员函数。

```
class OpOverClass
{
public:
 void print() const;

 //Overload the arithmetic operators
 OpOverClass operator+(const OpOverClass&) const;
 OpOverClass operator*(const OpOverClass&) const;
 OpOverClass(int i = 0, int j = 0);

private:
 int a;
 int b;
};
```

函数 print()、operator+、operator\* 和构造函数的定义为:

```
void OpOverClass::print() const
{
 cout<<"("<<a<<", "<<b<<")";
}

OpOverClass::OpOverClass(int i, int j)
{
 a = i;
 b = j;
}

OpOverClass OpOverClass::operator+
 (const OpOverClass& right) const
{
 OpOverClass temp;
 temp.a = a + right.a;
 temp.b = b + right.b;
 return temp;
}

OpOverClass OpOverClass::operator*
 (const OpOverClass& right) const
{
 OpOverClass temp;
 temp.a = a * right.a;
 temp.b = b * right.b;
 return temp;
}
```

考虑下面的 main() 函数:

```
int main()
{
 OpOverClass u(23, 45); //Line 1
 OpOverClass v(12, 10); //Line 2
 OpOverClass w1; //Line 3
 OpOverClass w2; //Line 4

 cout<<"Line 5: u = "; //Line 5
 u.print(); //Line 6; output u
 cout<<endl; //Line 7

 cout<<"Line 8: v = "; //Line 8
 v.print(); //Line 9; output v
 cout<<endl; //Line 10

 w1 = u + v; //Line 11; add u and v

 cout<<"Line 12: w1 = "; //Line 12
 w1.print(); //Line 13; output w1
 cout<<endl; //Line 14

 w2 = u * v; //Line 15; multiply u and v

 cout<<"Line 16: w2 = "; //Line 16
 w2.print(); //Line 17; output w2
 cout<<endl; //Line 18

 return 0;
}
```



输出:

```
Line 5: u = (23, 45)
Line 8: v = (12, 10)
Line 12: w1 = (35, 55)
Line 16: w2 = (275, 450)
```

main()函数是自解释的,读者可以仔细检查一下。例如,第6行语句输出u的值。看一下注释为Line 5的输出语句,它包含了第5、6和7行语句的输出。类似地,第9行语句输出v的值。第11行语句把u和v相加,把结果赋给w1。类似地,第15行语句把u和v相乘,将结果赋给w2。

### 重载关系运算符为成员函数

现在阐述如何重载关系运算符为类的成员函数。具体而言,为OpOverClass类重载等号运算符==。

因为关系运算的结果为true或false,函数operator==的返回类型为bool类型。

在OpOverClass类定义中包括的函数原型的语法为:

```
bool operator==(const OpOverClass&) const;
```

两个(相同类型的)对象通常是一样的,如果它们对应的成员变量相同的话。所以,OpOverClass类中operator==函数的定义为:

```
bool OpOverClass::operator==
 (const OpOverClass& right) const
{
 return(a == right.a && b == right.b);
}
```

### 重载二元关系运算符为成员函数的一般语法

下面描述重载二元运算为类成员函数的一般函数形式。

函数原型(包括在类的定义中):

```
bool operator op(const className&) const;
```

这里op为所要重载的关系运算符,className为对运算符op进行重载的类名。

函数定义:

```
bool className::operator op(const className& right) const
{
 //Compare and return the value
}
```

下面的例子阐述了重载关系运算符的使用。

### 例 16.5

在这个例子中,需要为OpOverClass类重载运算符==和!=为成员函数。

OpOverClass类和函数的定义为:

```
class OpOverClass
{
public:
 void print() const;
 //Overload the relational operators
 bool operator==(const OpOverClass&) const;
 bool operator!=(const OpOverClass&) const;
```

```

 OpOverClass(int i = 0, int j = 0);
private:
 int a;
 int b;
};

//The definitions of the function print and the constructor
//are the same as in Example 16-4
bool OpOverClass::operator==(const OpOverClass& right) const
{
 return(a == right.a && b == right.b);
}

bool OpOverClass::operator!=(const OpOverClass& right) const
{
 return(a != right.a || b != right.b);
}

```

下面的程序测试运算符 == 和 !=:

```

int main()
{
 OpOverClass u(23, 45); //Line 1
 OpOverClass v(12, 10); //Line 2
 OpOverClass w(23, 45); //Line 3

 if(u == v) //Line 4
 cout<<"Line 5: u and v are equal"<<endl; //Line 5
 else //Line 6
 cout<<"Line 7: u and v are not equal"<<endl; //Line 7

 if(u == w) //Line 8
 cout<<"Line 9: u and w are equal"<<endl; //Line 9
 else //Line 10
 cout<<"Line 11: u and w are not equal"<<endl; //Line 11
 return 0;
}

```

输出:

```

Line 7: u and v are not equal
Line 9: u and w are equal

```

### 非成员函数的二元运算符

假设 + 重载为 OpOverClass 类的非成员函数, 同时有下面的运算:

$x + y$

在这种情况下, 表达式被编译为:

`operator+(x, y)`

这里函数 `operator+` 有两个参数。在该表达式中, 函数 `operator+` 既不是 `x` 对象的成员, 也不是 `y` 对象的成员。所要相加的对象是作为参数传递给函数 `operator+` 的。

为了在类的定义中包括作为非成员函数的运算符函数 `operator+`, 保留字 `friend` 必须出现在函数头部。此外, 函数 `operator+` 必须有两个参数。

因此, 为了在 `OpOverClass` 类的定义中包括作为非成员函数的 `Operator+`, 它在 `OpOverClass` 定义中的原型为:

```
friend OpOverClass operator+(const OpOverClass&,
 const OpOverClass&);
```

operator+ 函数的定义如下:

```
OpOverClass operator+(const OpOverClass& firstObject,
 const OpOverClass& secondObject)
{
 OpOverClass temp;
 temp.a = firstObject.a + secondObject.a;
 temp.b = firstObject.b + secondObject.b;
 return temp;
}
```

在前面的定义中, firstObject 和 secondObject 对应的成员变量被相加, 结果存储到 temp 中。类的私有成员对类来说是局部的, 因此不能在类之外访问。如果我们遵循该规则, 那么因为 operator+ 不是 OpOverClass 类的成员, 在 operator+ 函数的定义中, 诸如 firstObject.a 的表达式肯定是非法的, 因为 a 是 firstObject 的私有成员。然而, 因为 operator+ 声明为 OpOverClass 类的友元函数, OpOverClass 类型的对象可以在 operator+ 的定义中访问它的私有成员。此外, 注意到在函数头部, 类名 OpOverClass 和作用域运算符没有包括在 operator+ 的函数名之前, 因为函数 operator+ 不是该类的成员。

### 重载二元 (算术) 运算符为非成员函数的一般语法

下面描述重载二元运算符为类的非成员函数的一般函数形式。

#### a. 函数原型 (包括在类的定义中):

```
friend returnType operator op(const className&,
 const className&);
```

这里 op 表示所要重载的二元运算符, returnType 为函数返回值的类型, className 为进行运算符重载的类名。

#### b. 函数定义:

```
returnType operator op(const className& firstObject,
 const className& secondObject)
{
 //algorithm to perform the operation
 return (value);
}
```

### 重载关系运算符为非成员函数

现在阐述如何重载关系运算符为类的非成员函数。具体而言, 为 OpOverClass 类重载等号运算符 ==。

显然, 函数 operator== 的返回值类型为 bool。

#### 函数原型 (包括在 OpOverClass 类的定义中):

```
friend bool operator==(const OpOverClass&, const OpOverClass&);
```

OpOverClass 类中 operator== 的函数定义如下所示:

```
bool operator==(const OpOverClass& firstObject,
 const OpOverClass& secondObject)
{
```

```

 return(firstObject.a == secondObject.a &&
 firstObject.b == secondObject.b);
 }

```

### 重载二元关系运算符为非成员函数的一般语法

下面描述重载二元关系运算符为类的非成员函数的一般函数形式。

函数原型（包括在类的定义中）：

```
friend bool operator op(const className&, const className&);
```

这里 op 为所要重载的关系运算符，className 为进行运算符重载的类名。

函数定义：

```

bool operator op (const className& firstObject,
 const className& secondObject)
{
 //Compare and return the value
}

```

## 16.2.7 重载流插入 (<<) 和析取 (>>) 运算符

因为如下的原因，为某类重载插入运算符<<或析取运算符>>的运算符函数必须是该类的非成员函数。

考虑下面的表达式：

```
cin>>x
```

在该表达式中，>>最左面的操作数（cin）为istream类型的对象，而不是OpOverClass类型的对象。因为>>最左面的操作数不是OpOverClass类型的对象，为OpOverClass类重载析取运算符的运算符函数必须是OpOverClass类的非成员函数。

类似地，为OpOverClass类重载流插入运算符的运算符函数必须是OpOverClass类的非成员函数。

### 重载流插入运算符 (<<)

下面描述为类重载流插入运算符<<的一般语法：

函数原型（包括在类的定义中）：

```
friend ostream& operator<<(ostream&, const className&);
```

函数定义：

```

ostream& operator<<(ostream& osObject, const className& cObject)
{
 //local declaration if any
 //Output members of cObject
 //osObject<<. . .

 //return stream object
 return osObject;
}

```

在前面的函数定义中：

- 参数均为引用参数。
- 第 1 个参数 osObject 为 ostream 对象的引用。

- 第2个参数通常为对某类的常量引用，因为（回顾一下第13章）传递一个对象为参数的最有效方式是引用。在这种情况下，形参不需要复制实参的成员变量。在前面的定义中，保留字 `const` 出现在类名之前，因为只需要打印该对象的成员变量。就是说，函数不应该修改该对象的成员变量。
- 函数返回类型为 `ostream` 对象的引用。

因为如下原因，重载运算符 `<<` 函数的返回类型必须是 `ostream` 对象的引用。

假设 `OpOverClass` 类重载了运算符 `<<`。语句：

```
cout<<x;
```

等同于语句：

```
operator<<(cout,x);
```

这是一条合法的语句，因为两个实参都是对象而不是对象的值。第1个参数 `cout` 为 `ostream` 类型；第2个参数 `x` 为 `OpOverClass` 类型。

现在考虑下面的语句：

```
cout<<x<<y;
```

该语句等同于语句：

```
operator<<(operator<<(cout,x),y); //Line A
```

因为运算符 `<<` 的结合性为从左到右。

执行前面的语句必须首先执行表达式：

```
cout<<x
```

也就是表达式：

```
operator<<(cout,x)
```

在执行该表达式后，函数 `operator<<` 返回值都将成为运算符 `<<` 左面的自变量（函数 `operator<<` 的第1个参数），这样就可以输出 `y` 对象的值（参见 Line A）。因为运算符 `<<` 左面的自变量必须是一个 `ostream` 类型的对象，表达式：

```
cout<<x
```

必须返回 `cout` 对象（不是它的值）才能输出 `y` 的值。

所以，函数 `operator<<` 的返回值类型必须是对 `ostream` 类型对象的引用。

### 重载流析取运算符 (>>)

下面描述为类重载流析取运算符 `>>` 的一般语法。

函数原型（包括在类的定义中）：

```
friend istream& operator>>(istream&, className&);
```

函数定义：

```
istream& operator>>(istream& isObject, className& cObject)
{
 //local declaration if any
 //Read data into cObject
 //isObject>>...
```

```

 //return stream object
 return isObject;
 }

```

在前面的函数定义中:

- 两个参数均为引用参数。
- 第一个参数 isObject 为 istream 对象的引用。
- 第二个参数通常为类对象的引用。读取的数据将存储在该对象中。
- 函数返回类型为 istream 对象的引用。

类似地, 函数 operator>> 的返回值类型必须是 istream 对象的引用。那么就可以成功地执行下面类型的语句:

```
cin>>x>>y;
```

下一个例子阐述了 OpOverClass 类对流插入和析取运算符的重载。

### 例 16.6

OpOverClass 类的定义和运算符函数的定义为:

```

class OpOverClass
{
 //overload the stream insertion and extraction operators
 friend ostream& operator<<(ostream&, const OpOverClass&);
 friend istream& operator>>(istream&, OpOverClass&);

public:
 //overload the arithmetic operators
 OpOverClass operator+(const OpOverClass&) const;
 OpOverClass operator*(const OpOverClass&) const;
 OpOverClass(int i = 0, int j = 0);

private:
 int a;
 int b;
};

//The definitions of the functions operator+, operator*,
//and the constructor are the same as in Example 16-4
ostream& operator<<(ostream& osObject, const OpOverClass& right)
{
 osObject<<"("<<right.a<<"", "<<right.b<<"")";
 return osObject;
}

istream& operator>>(istream& isObject, OpOverClass& right)
{
 isObject>>right.a>>right.b;
 return isObject;
}

```

考虑下面的 main() 函数:

```

int main()
{
 OpOverClass u(23, 45);

```

//Line 1



```

 OpOverClass v; //Line 2
 cout<<"Line 3: u : "<<u<<endl; //Line 3
 cout<<"Line 4: Enter two integers: "; //Line 4
 cin>>v; //Line 5
 cout<<endl; //Line 6
 cout<<"Line 7: v : "<<v<<endl; //Line 7
 cout<<"Line 8: u + v : "<<u + v<<endl; //Line 8
 cout<<"Line 9: u * v : "<<u * v<<endl; //Line 9
 return 0;
}

```

**运行实例：**在本运行实例中，用户输入用阴影表示。

```

Line 3: u = (23, 45)
Line 4: Enter two integers: 5:6

Line 7: v = (5, 6)
Line 8: u + v = (28, 51)
Line 9: u * v = (115, 270)

```

第1和第2行语句声明和初始化 *u*、*v* 为 *OpOverClass* 类型的对象。第3行语句使用 *cout* 和插入运算符输出 *u* 的值。第5行语句使用 *cin* 和析取运算符输入数据到 *v* 中。第7行语句使用 *cout* 和插入运算符输出 *v* 的值。第8行 *cout* 语句把 *u* 和 *v* 相加，然后输出结果。类似地，第9行 *cout* 语句把 *u* 和 *v* 相乘，然后输出结果。输出结果说明流插入和析取运算符都已成功地重载。

### 16.2.8 重载赋值运算符 (=)

对类允许的运算之一为赋值运算。赋值运算符实现对类成员变量的按成员复制。例如，语句：

```
x = y;
```

等同于语句：

```

x.a = y.a;
x.b = y.b;

```

在第15章中，内建的赋值运算符对于没有指针成员变量的类工作得很好。所以，为了避免指针成员变量的数据浅复制，必须显式地重载赋值运算符。

回忆一下，为了重载类的赋值运算符 *=*，运算符函数 *operator=* 必须为该类的成员函数。

**为类重载赋值运算符 *=* 的一般语法**

下面描述为类重载赋值运算符 *=* 的一般语法。

**函数原型**（包括在类的定义中）：

```
const className& operator=(const className&);
```

**函数定义：**

```

const className& className::operator=(const className& rightObject)
{
 //local declaration, if any

 if(this != &rightObject) //avoid self-assignment
 {
 //algorithm to copy rightObject into this object
 }

 //return the object assigned
}

```

```

 return *this;
 }

```

在函数 `operator=` 的定义中:

- 只有一个形参。
- 形参通常为类的常量引用。
- 函数返回类型为类的常量引用。

现在解释为什么函数 `operator=` 的返回类型必须是类的引用。

假设为 `OpOverClass` 类重载赋值运算符 `=`。语句:

```
x = y;
```

等同于语句:

```
x.operator=(y);
```

就是说, 对象 `y` 成为函数 `operator=` 的实参。

现在考虑语句:

```
x = y = z;
```

因为运算符 `=` 的结合性为从右到左, 该语句等同于语句:

```
x.operator=(y.operator=(z)); //Line B
```

显然, 必须首先执行表达式:

```
y.operator=(z)
```

就是表达式:

```
y = z
```

表达式:

```
y.operator=(z)
```

的返回值将成为函数 `operator=` 的参数, 这样就可以赋值给对象 `x` (参见 Line B)。因为函数 `operator=` 的形参为引用参数, 表达式:

```
y.operator=(z)
```

必须返回对象的引用, 而不是值。就是说, 它必须返回 `y` 对象的引用, 而不是 `y` 的值。因此, 为类重载赋值运算符 `=` 的函数返回值类型必须为类的引用。

现在考虑语句:

```
x = x;
```

这里, 试图复制 `x` 的值到 `x`, 即该语句是自赋值。

程序必须防止这种语句的出现, 因为它们浪费计算机时间。

`operator=` 的函数体确实防止了这种赋值形式。

考虑运算符函数 `operator=` 函数体中的 `if` 语句:

```

 if(this != &rightObject) //avoid self-assignment
 {
 //algorithm to copy rightObject into this object
 }

```



现在语句:

```
x = x;
```

编译为语句:

```
x.operator=(x);
```

因为函数 `operator=` 被 `x` 对象调用, `operator=` 函数体中的 `this` 指针引用 `x` 对象。此外, 因为 `x` 还是函数 `operator=` 的参数, 形参 `rightObject` 同样引用了 `x` 对象。所以, 在表达式:

```
this != &rightObject
```

中, `this` 表示 `x` 的地址, `&rightObject` 同样表示 `x` 的地址。这样, 表达式的值将为 `false`。所以, `if` 语句体将被跳过。

下面的例子阐述了如何重载赋值运算符。

### 例 16.7

考虑下面的类:

```
class cAssignmentOprOverload
{
public:
 const cAssignmentOprOverload&
 operator=(const cAssignmentOprOverload& otherList);
 //Overload the assignment operator

 void print() const;
 //Function to print the list

 void insertEnd(int item);
 //Function to insert an item at the end of the list
 //Post: if the list is not full, length++;
 // list[length] = item
 // if the list is full
 // output an appropriate message

 void destroyList();
 //Function to destroy the list
 //Post: length = 0; maxSize = 0;
 // list = NULL

 cAssignmentOprOverload(int size = 0);
 //constructor
 //Post: length = 0; maxSize = size;
 // list is an array of size maxSize

private:
 int maxSize;
 int length;
 int *list;
};
```

`cAssignmentOprOverload` 类成员函数的定义为:

```
void cAssignmentOprOverload::print() const
{
 if(length == 0)
 cout<<"List is empty"<<endl;
 else
```

```

 {
 for(int i = 0; i < length; i++)
 cout<<list[i]<<" ";
 cout<<endl;
 }
 }

 void cAssignmentOprOverload::insertEnd(int item)
 {
 if(length == maxSize)
 cout<<"List is full"<<endl;
 else
 list[length++] = item;
 }

 void cAssignmentOprOverload::destroyList()
 {
 delete [] list;
 list = NULL;
 length = 0;
 maxSize = 0;
 }

 cAssignmentOprOverload::cAssignmentOprOverload(int size)
 {
 maxSize = size;
 length = 0;
 if(maxSize == 0)
 list = NULL;
 else
 list = new int[maxSize];
 }

 const cAssignmentOprOverload& cAssignmentOprOverload::operator=
 (const cAssignmentOprOverload& otherList)
 {
 if(this != &otherList) //avoid self-assignment; Line 1
 {
 if(list != NULL) //Line 2
 destroyList(); //Line 3
 maxSize = otherList.maxSize; //Line 4
 length = otherList.length; //Line 5
 if(maxSize != 0) //Line 6
 {
 list = new int[maxSize]; //Line 7
 if(length != 0) //Line 8
 for(int i = 0; i < length; i++) //Line 9
 list[i] = otherList.list[i]; //Line 10
 }
 else //Line 11
 list = NULL; //Line 12
 }

 return *this; //Line 13
 }
}

```

重载赋值运算符的函数工作如下。第1行语句检查一个对象是否在复制自己。第2行语句检查list是否为非空。如果它是非空，通过释放list所占据的内存，list将被销毁。第4行和第5行语句

分别复制 otherList 的成员变量 maxSize 和 length 的值到 list 的 maxSize 和 length 中。如果 otherList 不为 NULL 也不为空, 第6行至第10行语句创建数组 list 并复制 otherList 到 list。如果 otherList 为 NULL, list 初始化为 NULL。第13行语句返回 list 的地址, 因为函数 operator= 的返回值类型为引用类型。

下面的函数测试 cAssignmentOprOverload 类:

```
int main()
{
 cAssignmentOprOverload intList1(10); //Line 14
 cAssignmentOprOverload intList2; //Line 15
 cAssignmentOprOverload intList3; //Line 16

 int i; //Line 17
 int number; //Line 18

 cout<<"Line 19: Enter 5 integers: "; //Line 19
 for(i = 0; i < 5; i++) //Line 20
 {
 cin>>number; //Line 21
 intList1.insertEnd(number); //Line 22
 }

 cout<<endl<<"Line 23: intList1: "; //Line 23
 intList1.print(); //Line 24

 intList3 = intList2 = intList1; //Line 25

 cout<<"Line 26: intList2: "; //Line 26
 intList2.print(); //Line 27

 intList2.destroyList(); //Line 28

 cout<<endl; //Line 29
 cout<<"Line 30: intList2: "; //Line 30
 intList2.print(); //Line 31

 cout<<"Line 32: After destroying intList2, "
 <<"intList1: "; //Line 32
 intList1.print(); //Line 33
 cout<<"Line 34: After destroying intList2, "
 <<"intList3: "; //Line 34
 intList3.print(); //Line 35
 cout<<endl; //Line 36

 return 0;
}
```

**运行实例:** 在本运行实例中, 用户的输出用阴影表示。

```
Line 19: Enter 5 integers: 8 5 3 7 2
Line 23: intList1: 8 5 3 7 2
Line 26: intList2: 8 5 3 7 2

Line 30: intList2: List is empty
Line 32: After destroying intList2, intList1: 8 5 3 7 2
Line 34: After destroying intList2, intList3: 8 5 3 7 2
```

第14行语句创建大小为10的intList1; 第15行和第16行语句创建(默认)大小为50的intList2和intList3。第20行至22行语句输入数据到intList1, 第24行语句输出intList1。第25行语句首先复制intList1到intList2, 然后复制intList2到intList3。第27行语句输出intList2(参见运行实例中的

第 26 行, 该行包括第 26 行和 27 行的输出)。第 28 行语句销毁 `intList2`。第 31 行语句输出 `intList2`, 它是空的 (参见运行实例中的第 30 行, 该行包括第 30 行和第 31 行的输出)。在销毁 `intList2` 之后, 程序输出 `intList1` 和 `intList3` 的内容 (参见运行实例中的第 32 行和第 34 行)。从运行实例中可清楚得知, `intList2` 的析构函数不影响 `intList1` 和 `intList3`, 因为 `intList1` 和 `intList3` 有它们自己的数据。

### 16.2.9 重载一元运算符

重载一元运算符的过程和重载二元运算符类似。惟一的不同是, 在二元运算符情况下, 运算符有两个操作数; 在一元运算符情况下, 运算符只有一个操作数。所以, 要为类重载一元运算符:

1. 如果运算符函数为类的成员, 则没有参数。
2. 如果运算符函数为非成员函数——就是类的友元函数——则有一个参数。

下面, 描述如何重载递增和递减运算符。

### 16.2.10 重载递增 (++) 和递减 (--) 运算符

递增运算符有两种形式: 前递增 (`++u`) 和后递增 (`u++`), 这里 `u` 是变量, 例如 `int` 类型的变量。在前递增的情况下, 变量 `u` 的值在使用之前增加 1; 在后递增的情况下, `u` 的值在增加 1 之前在表达式中使用。

#### 重载前递增运算符

重载前递增运算符是很直观的。在函数定义中, 首先增加对象的值, 然后为 `OpOverClass` 类重载前递增运算符。此外, 假设运算符函数 `operator++` 为 `OpOverClass` 类的成员, 那么运算符函数 `operator++` 没有参数。因为运算符函数 `operator++` 没有参数, 可以使用 `this` 指针返回对象增加的值。(出于阐述的目的, 假定递增运算符将 `OpOverClass` 类的成员变量 `a` 和 `b` 增加 1。)

```
OpOverClass OpOverClass::operator++()
{
 //increment the object
 ++a;
 ++b;
 return *this; //return the incremented value of the object
}
```

因为 `x` 是 `OpOverClass` 类型的对象, 语句:

```
++x;
```

将 `x` 的值增加 1, `x` 的 `this` 指针返回增加后的 `x` 值。此外, 因为 `++x` 不出现在表达式中, `*this` 返回的值丢失了。现在 `y` 也是 `OpOverClass` 类型的对象, 那么语句:

```
y = ++x;
```

将 `x` 的值增加 1, `x` 的 `this` 指针返回增加后的 `x` 值, 存储在 `y` 中。

#### 重载前递增运算符 ++ 为成员函数的一般语法

下面描述重载前递增运算符 `++` 为成员函数的一般语法。

函数原型 (包括在类的定义中):

```
className operator++();
```

函数定义:

```

className className::operator++()
{
 //increment the value of the object by 1
 return *this;
}

```

重载前递增运算符的运算符函数也可以是 OpOverClass 类的非成员函数，将在下面进行讨论。

因为运算符函数 operator++ 为 OpOverClass 类的非成员函数，它有一个参数并且为 OpOverClass 类型的对象。（出于阐述的目的，假定递增运算符将 OpOverClass 类的成员变量 a 和 b 增加 1。）

```

OpOverClass operator++(OpOverClass& incObj)
{
 //increment the object
 (incObj.a)++;
 (incObj.b)++;
 return incObj; //return the incremented value of the object
}

```

### 重载前递增运算符 ++ 为非成员函数的一般语法

下面描述重载前递增运算符 ++ 为非成员函数的一般语法。

函数原型（包括在类的定义中）：

```
friend className operator++(className&);
```

函数定义：

```

className operator++(className& incObj)
{
 //increment incObj by 1
 return incObj;
}

```

### 重载后递增运算符

现在讨论如何重载后递增运算符。和前递增运算符的情况一样，首先描述重载该运算符为类的成员函数。

在前递增和后递增的情况下，运算符函数的名字是一样的——operator++。为了区别前递增和后递增运算符重载，可以在运算符函数的头部使用一个（int 类型）虚参数。那么，OpOverClass 类的后递增运算符的函数原型为：

```
OpOverClass operator++(int);
```

语句：

```
x++;
```

被编译器编译为语句：

```
x.operator++(0);
```

然后执行带参数的 operator++ 函数。参数 0 仅用来区别前递增和后递增运算符函数。

后递增运算符首先使用表达式中对象的值，然后把对象的值增加 1。那么实现该函数所需的步骤为：

1. 保存对象的值。例如，保存在 temp 中。
2. 增加对象的值。

3. 返回保存在 temp 中的值。

OpOverClass 类中后递增运算符的函数定义为:

```
OpOverClass OpOverClass::operator++(int u)
{
 OpOverClass temp = *this; //use this pointer to copy
 //the value of the object
 //increment the object
 a++;
 b++;
 return temp; //return the old value of the object
}
```

**重载后递增运算符 ++ 为成员函数的一般语法**

下面描述重载后递增运算符 ++ 为成员函数的一般语法。

函数原型 (包括在类的定义中):

```
className operator++(int);
```

函数定义:

```
className className::operator++(int u)
{
 className temp = *this; //use this pointer to copy
 //the value of the object
 //increment the object
 return temp; //return the old value of the object
}
```

后递增运算符也可以重载为类的非成员函数。在这种情况下, 运算符函数 operator++ 有两个参数。为 OpOverClass 类重载后递增运算符为非成员函数的函数定义为:

```
OpOverClass operator++(OpOverClass& incObj, int u)
{
 OpOverClass temp = incObj; //copy incObj into temp
 //increment incObj
 (incObj.a)++;
 (incObj.b)++;
 return temp; //return the old value of the object
}
```

**重载后递增运算符 ++ 为非成员函数的一般语法**

下面描述重载后递增运算符 ++ 为非成员函数的一般语法。

函数原型 (包括在类的定义中):

```
friend className operator++(className&, int);
```

函数定义:

```
className operator++(className& incObj, int u)
{
 className temp = incObj; //copy incObj into temp
 //increment incObj
}
```



```

 return temp; //return the old value of the object
}

```

递减运算符可以以类似的方式重载，具体细节留做读者练习。

现在编写 `OpOverClass` 类的定义并说明运算符函数如何出现在类的定义中。因为某些运算符既可以被重载为成员函数也可以被重载为非成员函数，程序将给出 `OpOverClass` 类两个等价的定义。在第一个定义中，递增、递减、算术和关系运算符被重载为成员函数。在第二个定义中，递增、递减、算术和关系运算符被重载为非成员函数。

`OpOverClass` 类的定义如下：

```

//Definition of the class OpOverClass
//The increment, decrement, arithmetic, and relational
//operator functions are members of the class.

class OpOverClass
{
 //overload the stream insertion and extraction operators
 friend ostream& operator<<(ostream&, const OpOverClass&);
 friend istream& operator>>(istream&, OpOverClass&);

public:
 //overload the arithmetic operators
 OpOverClass operator+(const OpOverClass&) const;
 OpOverClass operator-(const OpOverClass&) const;
 OpOverClass operator*(const OpOverClass&) const;
 OpOverClass operator/(const OpOverClass&) const;

 //overload the increment and decrement operators
 OpOverClass operator++(); //pre-increment
 OpOverClass operator++(int); //post-increment
 OpOverClass operator--(); //pre-decrement
 OpOverClass operator--(int); //post-decrement

 //overload the relational operators
 bool operator==(const OpOverClass&) const;
 bool operator!=(const OpOverClass&) const;
 bool operator<=(const OpOverClass&) const;
 bool operator<(const OpOverClass&) const;
 bool operator>=(const OpOverClass&) const;
 bool operator>(const OpOverClass&) const;

 //constructors
 OpOverClass();
 OpOverClass(int i, int j);

 //include other functions as needed
private:
 int a;
 int b;
};

```

下面是 `OpOverClass` 类的另一个定义，在该定义中，算术、递增、递减和关系运算符重载为非成员函数。

```

//Definition of the class OpOverClass
//The increment, decrement, arithmetic, and relational
//operator functions are nonmembers of the class.

class OpOverClass

```

```

{
 //overload the stream insertion and extraction operators
 friend ostream& operator<<(ostream&, const OpOverClass&);
 friend istream& operator>>(istream&, OpOverClass&);

 //overload the arithmetic operators
 friend OpOverClass operator+(const OpOverClass&,
 const OpOverClass&);
 friend OpOverClass operator-(const OpOverClass&,
 const OpOverClass&);
 friend OpOverClass operator*(const OpOverClass&,
 const OpOverClass&);
 friend OpOverClass operator/(const OpOverClass&,
 const OpOverClass&);

 //overload the increment and decrement operators
 friend OpOverClass operator++(OpOverClass&);
 //pre-increment
 friend OpOverClass operator++(OpOverClass&, int);
 //post-increment
 friend OpOverClass operator--(OpOverClass&);
 //pre-decrement
 friend OpOverClass operator--(OpOverClass&, int);
 //post-decrement

 //overload the relational operators
 friend bool operator==(const OpOverClass&,
 const OpOverClass&);
 friend bool operator!=(const OpOverClass&,
 const OpOverClass&);
 friend bool operator<=(const OpOverClass&,
 const OpOverClass&);
 friend bool operator<(const OpOverClass&,
 const OpOverClass&);
 friend bool operator>=(const OpOverClass&,
 const OpOverClass&);
 friend bool operator>(const OpOverClass&,
 const OpOverClass&);

public:
 //constructors
 OpOverClass();
 OpOverClass(int i, int j);

 //include other functions as needed

private:
 int a;
 int b;
};

```

为 OpOverClass 类重载运算符的函数定义留做读者练习。(参见本章的编程练习 1 和 2。)

### 16.2.11 运算符重载：成员与非成员

前面部分阐述了如何重载运算符。某些特定的运算符必须重载为类的成员函数，而某些运算符必须重载为非成员（友元）函数。既可以重载为成员函数也可以重载为非成员函数的运算符又如何呢？例如，二元算术运算符+可以被重载为成员函数和非成员函数。如果重载+为成员函数，那么运算符+可以直接访问其中一个对象的成员变量，只需要传递一个对象为参数。另一方面，如果重载+为非成员函数，必须把两个对象都作为参数传递。因此，重载+为非成员可能会要求额外的内



存和计算机时间来进行局部的数据复制。所以,出于效率的目的,只要可能,应该重载运算符为成员函数。

### 16.2.12 类和指针成员变量(复习)

在第15章中描述了包含指针成员变量的类特性。既然已经讨论了如何重载各种不同的运算符,现在回顾一下带指针成员变量的类特性,以及出于完整性的原因,应该如何在程序中避免它们。

对类的惟一内建运算符为赋值和成员选择运算符。赋值运算符提供了按成员的数据复制,对象的成员变量被复制到另一个相同类型对象相应的成员变量中。这种按成员的复制对带指针成员变量的类不能很好地工作。由于带指针成员变量的类而产生的其他问题和释放动态内存相关,如当对象超出作用域时和以值方式传递类对象为参数时。为了解决这些问题,包含指针成员变量的类必须:

1. 显式地重载赋值运算符
2. 包含复制构造函数
3. 包含析构函数

### 16.2.13 运算符重载:结束语

下面,将研究三个阐述运算符重载的例子。在深入钻研这些例子之前,必须牢记如下规则:假设为类 `OpOverClass` 重载运算符 `op`。无论何时对 `OpOverClass` 类型的对象使用运算符 `op`,将会执行为 `OpOverClass` 类重载运算符 `op` 的函数。

## 16.3 编程练习: clockType

在第14章中定义了 `clockType` 类在程序中实现一天的时间。我们使用函数实现了打印时间、增加时间、比较两个时间是否相等这些操作。本例子重定义了 `clockType` 类。它还重载了流插入和析取运算符,从而更方便地进行输入和输出;重载了关系运算符进行时间的比较;重载了递增运算符把时间增加一秒。使用 `clockType` 类的程序要求用户以 `hr: min: sec` 的格式输入时间。

修改后的 `clockType` 类定义如下:

```
//Header file newClock.h

#ifndef H_newClock
#define H_newClock

#include <iostream>
using namespace std;

class clockType
{
 friend ostream& operator<< (ostream&, const clockType&);
 friend istream& operator>> (istream&, clockType&);
public:
 void setTime(int hours, int minutes, int seconds);
 //Function to set the private data members
 //hr, min, and sec
 //Post: hr = hours; min = minutes; sec = seconds
 void getTime(int& hours, int& minutes, int& seconds);
 //Function to return the time
 //Post: hours = hr; minutes = min; seconds = sec;
```

```

clockType operator++();
 //Overload the pre-increment operator
 //Post: Time is incremented by one second

bool operator==(const clockType& otherClock) const;
 //Overload the equality operator
 //Function returns true if the time is equal
 //to otherTime, otherwise it returns the value false

bool operator<=(const clockType& otherClock) const;
 //Overload the less than or equal to operator
 //Function returns true if the time is less
 //than or equal to otherTime, otherwise it returns the
 //value false

clockType(int hours = 0, int minutes = 0, int seconds = 0);
 //Constructor to initialize the object with the values
 //specified by the user. If no values are specified,
 //the default values are assumed.
 //Post: hr = hours; min = minutes; sec = seconds

private:
 int hr; //variable to store the hours
 int min; //variable to store the minutes
 int sec; //variable to store the seconds
};

#endif

```

现在让我们编写实现 clockType 类的操作函数定义。注意 clockType 类只重载了前递增运算符。不过，为了连续性，类也应该重载后递增运算符。这一步留做读者练习（参见本章的编程练习3。）

首先，编写 operator++ 函数的定义。将时间增加一秒的算法如下：

- a. 将秒数增加1。
- b. 如果秒 > 59:
  - b.1. 设置秒为0。
  - b.2. 将分钟增加1。
  - b.3. 如果分钟 > 59:
    - b.3.1. 设置分钟为0。
    - b.3.2. 将小时增加1。
    - b.3.3. 如果小时 > 23:
      - b.3.3.1 设置小时为0。

c. 返回增加后的对象值。

operator++ 函数的定义为：

```

 //overload the pre-increment operator
clockType clockType::operator++()
{
 sec++; //Step a
 if(sec > 59) //Step b
 {
 sec = 0; //Step b.1
 min++; //Step b.2
 if(min > 59) //Step b.3

```

```

 {
 min = 0; //Step b.3.1
 hr++; //Step b.3.2
 if(hr > 23) //Step b.3.3
 hr = 0; //Step b.3.3.1
 }
 }

 return *this; //Step c
}

```

`operator==` 函数的定义很简单。两个时间如果有相同的小时、分钟和秒则认为是相同的。所以，`operator==` 函数的定义为：

```

//overload the equality operator
bool clockType::operator==(const clockType& otherClock) const
{
 return(hr == otherClock.hr && min == otherClock.min
 && sec == otherClock.sec);
}

```

`operator<=` 函数的定义在下面给出。第一时间小于或等于第二时间，当且仅当：

1. 第一时间的小时小于第二时间的小时
2. 第一时间的小时等于第二时间的小时，但第一时间的分钟小于第二时间的分钟
3. 第一时间的小时和分钟和第二时间的相同，但第一时间的秒小于或等于第二时间的秒

`operator<=` 函数的定义为：

```

//overload the less than or equal to operator
bool clockType::operator<=(const clockType& otherClock) const
{
 return((hr < otherClock.hr) ||
 (hr == otherClock.hr && min < otherClock.min) ||
 (hr == otherClock.hr && min == otherClock.min &&
 sec <= otherClock.sec));
}

```

构造函数和 `setTime()` 函数的定义和第13章给出的一样。出于完整性的考虑，它们也在此列出。

```

//constructor with parameters
clockType::clockType(int hours, int minutes, int seconds)
{
 if(0 <= hours && hours < 24)
 hr = hours;
 else
 hr = 0;

 if(0 <= minutes && minutes < 60)
 min = minutes;
 else
 min = 0;

 if(0 <= seconds && seconds < 60)
 sec = seconds;
 else
 sec = 0;
}

```

```

void clockType::setTime(int hours, int minutes, int seconds)
{
 if(0 <= hours && hours < 24)
 hr = hours;
 else
 hr = 0;

 if(0 <= minutes && minutes < 60)
 min = minutes;
 else
 min = 0;

 if(0 <= seconds && seconds < 60)
 sec = seconds;
 else
 sec = 0;
}

void clockType::getTime(int& hours, int& minutes, int& seconds)
{
 hours = hr;
 minutes = min;
 seconds = sec;
}

```

现在讨论 `operator<<` 函数的定义。时间必须以下面的形式输出：

hh: mm: ss

以该格式输出时间的算法和第13章中给出的 `clockType` 的 `printTime()` 函数是一样的。这里，在上述格式打印时间之后，必须返回 `ostream` 对象。所以，`operator<<` 函数的定义为：

```

//overload the stream insertion operator
ostream& operator<< (ostream& osObject, const clockType& timeOut)
{
 if(timeOut.hr < 10)
 osObject<<'0';
 osObject<<timeOut.hr<<':';

 if(timeOut.min < 10)
 osObject<<'0';
 osObject<<timeOut.min<<':';

 if(timeOut.sec < 10)
 osObject<<'0';
 osObject<<timeOut.sec;

 return osObject; //return the ostream object
}

```

现在讨论 `operator>>` 函数的定义。程序的输入格式为：

hh: mm: ss

就是说，输入格式为小时后跟冒号，接着是分钟后跟冒号，然后是秒。输入时间的算法为：

- 获取输入数值，把它存储在成员变量 `hr` 中。
- 获取下一个输入，是一个冒号，丢弃它。
- 获取下一个输入，是一个数值，把它存储在成员变量 `min` 中。

- d. 获取下一个输入, 是一个冒号, 丢弃它。
- e. 获取下一个输入, 是一个数值, 把它存储在成员变量 sec 中。
- f. 返回 istream 对象。

显然, 需要一个 char 类型的局部变量来读取冒号。

operator>>函数的定义为:

```
//overload the stream extraction operator
istream& operator>> (istream& isObject, clockType& timeIn)
{
 char ch;

 isObject>>timeIn.hr; //Step a
 isObject.get(ch); //Step b; read and discard :
 isObject>>timeIn.min; //Step c
 isObject.get(ch); //Step d; read and discard :
 isObject>>timeIn.sec; //Step e

 return isObject; //Step f
}
```

下面的测试程序使用了 clockType 类:

```
//Program that uses the class clockType
#include <iostream>
#include "newClock.h"
using namespace std;
int main()
{
 clockType myClock(5,6,23); //Line 1
 clockType yourClock; //Line 2

 cout<<"Line 3: myClock = "<<myClock<<endl; //Line 3
 cout<<"Line 4: yourClock = "<<yourClock<<endl; //Line 4

 cout<<"Line 5: Enter the time in the form "
 <<"hr:min:sec "; //Line 5
 cin>>myClock; //Line 6
 cout<<"Line 7: The new time of myClock = "
 <<myClock<<endl; //Line 7
 ++myClock; //Line 8

 cout<<"Line 9: After incrementing the time, myClock = "
 <<myClock<<endl; //Line 9

 yourClock.setTime(13,35,38); //Line 10
 cout<<"Line 11: After setting the time, yourClock = "
 <<yourClock<<endl; //Line 11
 if(myClock == yourClock) //Line 12
 cout<<"Line 13: The times of myClock and "
 <<"yourClock are equal"<<endl; //Line 13
 else //Line 14
 cout<<"Line 15: The times of myClock and "
 <<"yourClock are not equal"<<endl; //Line 15
 if(myClock <= yourClock) //Line 16
 cout<<"Line 17: The time of myClock is less "
 <<"than or equal to "<<endl;
```

```

 <<"the time of yourClock"<<endl; //Line 17
 else //Line 18
 cout<<"Line 19: The time of myClock is "
 <<"greater than the time of yourClock"
 <<endl; //Line 19

 return 0;
}

```

**运行实例：**在本运行实例中，用户输入用阴影表示。

```

Line 3: myClock = 05:06:23
Line 4: yourClock = 00:00:00
Line 5: Enter the time in the form hr:min:sec 4:50:59
Line 7: The new time of myClock = 04:50:59
Line 9: After incrementing the time, myClock = 04:51:00
Line 11: After setting the time, yourClock = 13:35:38
Line 15: The times of myClock and yourClock are not equal
Line 17: The time of myClock is less than or equal to
the time of yourClock

```

## 16.4 编程练习：复数

对于  $a+ib$ ，其中  $i^2=-1$ ， $a$  和  $b$  为实数，这种格式称为复数。 $a$ 、 $b$  分别是复数  $a+ib$  的实部和虚部。复数还可以表示为有序对  $(a, b)$ 。复数的加法和乘法由下面的规则定义：

$$(a+ib)+(c+id)=(a+c)+i(b+d)$$

$$(a+id)*(c+id)=(ac-bd)+i(ad+bc)$$

如果使用有序对符号，这些规则可以写成：

$$(a, b)+(c, d)=((a+c), (b+d))$$

$$(a, b)*(c, d)=((ac-bd), (ad+bc))$$

C++ 没有运行操作复数的内建数据类型。在本例子中，将构造一个可以用来处理复数的数据类型 `complexNumber`，并重载流插入和析取运算符得到更方便的输入和输出形式。还将重载运算符 `+` 和 `*` 执行复数的加法和乘法操作。如果  $x$  和  $y$  为复数，可以计算诸如  $x+y$  和  $x*y$  之类表达式的值。

```

//Specification file complexType.h
#ifndef H_complexNumber
#define H_complexNumber

class complexType
{
 //overload the stream insertion and extraction operators
 friend ostream& operator<< (ostream&, const complexType&);
 friend istream& operator>> (istream&, complexType&);

public:
 void setComplex(const double& real, const double& imag);
 //Set the complex number according to the parameters
 //Post: realPart = real; imaginaryPart = imag

 complexType(double real = 0, double imag = 0);
 //constructor
 //Initialize the complex number according to the parameters
 //Post: realPart = real; imaginaryPart = imag

```



```

 complexType operator+(const complexType& otherComplex) const;
 //overload +
 complexType operator*(const complexType& otherComplex) const;
 //overload *
 bool operator==(const complexType& otherComplex) const;
 //overload ==

private:
 double realPart; //variable to store the real part
 double imaginaryPart; //variable to store
 //the imaginary part
};
#endif

```

现在将写出实现 `complexType` 类中各种不同操作的函数定义。

大部分函数的定义是简单明了的。我们将只讨论重载流插入运算符<<和流析取运算符>>的函数定义。

为了以下面的格式输出复数：

(a,b)

这里 a 为实部，b 为虚部，算法为：

- a. 输出左括号 “(”。
- b. 输出实部。
- c. 输出逗号 “,”。
- d. 输出虚部。
- e. 输出右括号 “)”。

所以，`operator<<`函数的定义为：

```

ostream& operator<<(ostream& osObject, const complexType& complex)
{
 osObject<<"("; //Step a
 osObject<<complex.realPart; //Step b
 osObject<<","; //Step c
 osObject<<complex.imaginaryPart; //Step d
 osObject<<")"; //Step e
 return osObject; //Return the ostream object
}

```

下面，讨论重载流析取运算符>>的函数定义。

输入格式为：

(3,5)

在该输入中，复数的实部为 3，虚部为 5。

读取复数的算法为：

- a. 读取并丢弃左括号。
- b. 读取并存储实部。
- c. 读取并丢弃逗号。
- d. 读取并存储虚部。
- e. 读取并丢弃右括号。

遵循这些步骤, `operator>>` 函数的定义为:

```
istream& operator>> (istream& isObject, complexType& complex)
{
 char ch;

 isObject>>ch; //Step a
 isObject>>complex.realPart; //Step b
 isObject>>ch; //Step c
 isObject>>complex.imaginaryPart; //Step d
 isObject>>ch; //Step e

 return isObject; //Return the istream object
}
```

其他函数的定义如下:

```
bool complexType::operator==(const complexType& otherComplex) const
{
 return(realPart == otherComplex.realPart &&
 imaginaryPart == otherComplex.imaginaryPart);
}

//constructor
complexType::complexType(double real, double imag)
{
 realPart = real;
 imaginaryPart = imag;
}

void complexType::setComplex(const double& real, const double& imag)
{
 realPart = real;
 imaginaryPart = imag;
}

//overload the operator +
complexType complexType::operator+
 (const complexType& otherComplex) const
{
 complexType temp;

 temp.realPart = realPart + otherComplex.realPart;
 temp.imaginaryPart = imaginaryPart
 + otherComplex.imaginaryPart;

 return temp;
}

//overload the operator *
complexType complexType::operator*
 (const complexType& otherComplex) const
{
 complexType temp;

 temp.realPart = (realPart * otherComplex.realPart) -
 (imaginaryPart * otherComplex.imaginaryPart);
 temp.imaginaryPart = (realPart * otherComplex.imaginaryPart) +
 (imaginaryPart * otherComplex.realPart);

 return temp;
}
```



下面的程序阐述了 complexType 类的使用:

```
//Program that uses the class complexType

#include <iostream>
#include "complexType.h"

using namespace std;

int main()
{
 complexType num1(23,34); //Line 1
 complexType num2; //Line 2
 complexType num3; //Line 3

 cout<<"Line 4: Num1 = "<<num1<<endl; //Line 4
 cout<<"Line 5: Num2 = "<<num2<<endl; //Line 5

 cout<<"Line 6: Enter the complex number "
 <<"in the form (a,b) "; //Line 6
 cin>>num2; //Line 7
 cout<<endl; //Line 8

 cout<<"Line 9: New value of num2 = "
 <<num2<<endl; //Line 9

 num3 = num1 + num2; //Line 10

 cout<<"Line 11: Num3 = "<<num3<<endl; //Line 11
 cout<<"Line 12: "<<num1<<" + "<<num2
 <<" = "<<num1 + num2<<endl; //Line 12
 cout<<"Line 13: "<<num1<<" * "<<num2
 <<" = "<<num1 * num2<<endl; //Line 13

 return 0;
}
```

运行实例: 在本运行实例中, 用户输入用阴影表示。

```
Line 4: Num1 = (23, 34)
Line 5: Num2 = (0, 0)
Line 6: Enter the complex number in the form (a,b) (3,4)
Line 9: New value of num2 = (3, 4)
Line 11: Num3 = (26, 38)
Line 12: (23, 34) + (3, 4) = (26, 38)
Line 13: (23, 34) * (3, 4) = (-67, 194)
```

可以扩展该数据类型执行复数的减法和除法。

下面, 我们将定义一个类, 称为 newString, 然后重载赋值和关系运算符。就是说, 当说明一个 newString 类型的对象时, 将能够使用赋值运算符复制一个字符串; 能够使用关系运算符比较两个字符串。

不过, 在讨论 newString 类之前, 分析一下对运算符[]的重载。回想一下, 我们已经使用过运算符[]来访问数组的成员。为了访问 newString 类型字符串中的单一字符, 必须为 newString 类重载运算符[]。

## 16.5 重载数组索引 (下标) 运算符 ([])

回想一下, 为类重载运算符[]的函数必须是该类的成员函数。此外, 因为数组可以声明为常量

或非常量, 需要重载运算符[]来处理这两种情况。

为非常量数组声明运算符函数 operator[] 为类成员函数的一般语法为:

```
Type& operator[] (int index);
```

为常量数组声明运算符函数 operator[] 为类成员函数的一般语法为:

```
const Type& operator[] (int index) const;
```

这里 Type 为数组元素的数据类型。

假设 classTest 为包含数组成员变量的类。重载运算符函数 operator[] 的 classTest 定义为:

```
class classTest
{
public:
 Type& operator[] (int index);
 //overload the operator for nonconstant arrays
 const Type& operator[] (int index) const;
 //overload the operator for constant arrays
 .
 .
 .
private:
 Type *list; //pointer to the array
 int arraySize;
};
```

这里 Type 为数组元素的数据类型。

为 classTest 类重载运算符[]的函数定义为:

```
//overload the operator [] for nonconstant arrays
Type& classTest::operator[] (int index)
{
 assert(0 <= index && index < arraySize);
 return(list[index]); //return a pointer of the
 //array component
}

//overload the operator [] for constant arrays
const Type& classTest::operator[] (int index) const
{
 assert(0 <= index && index < arraySize);
 return(list[index]); //return a pointer of the
 //array component
}
```

考虑下面的语句:

```
classTest list1;
classTest list2;
const classTest list3;
```

对于语句:

```
list1[2] = list2[3];
```

用于非常量数组的运算符函数 operator[] 的函数体被执行。对于语句:

```
list1[2] = list3[5];
```

首先用于常量数组的运算符函数 `operator[]` 的函数体被执行, 因为 `list3` 为常量数组。接着, 用于非常量数组的运算符函数 `operator[]` 的函数体被执行, 以完成赋值语句的执行。

## 16.6 编程练习: newString

在第 9 章中讨论了 C 字符串。我们知道:

1. C 字符串由一个或多个字符组成
2. C 字符串包括在双引号标志中
3. C 字符串以空字符结束
4. C 字符串存储在字符数组中

在本例子中, 字符串指的是 C 字符串。

允许对字符串进行的惟一操作为输入和输出。为了使用其他操作, 程序员需要包括头文件 `cstring`, 它包含用于字符串操作的许多函数声明。

最初, C++ 不提供任何内建的处理字符串的数据类型。不过, 更新的 C++ 版本提供字符串类来处理字符串和对字符串的操作。

在本例子中, 我们的目标是定义用于字符串操作的类, 同时进一步阐述运算符重载。具体而言, 重载赋值运算符、关系运算符以及为了方便输入和输出重载流插入和析取运算符。我们称该类为 `newString`。首先给出 `newString` 类的定义:

```
//Header file myString.h
#ifndef H_myString
#define H_myString
#include <iostream>
using namespace std;

class newString
{
 //overload the stream insertion and extraction operators
 friend ostream& operator<<(ostream&, const newString&);
 friend istream& operator>>(istream&, newString&);

public:
 const newString& operator=(const newString&);
 //overload the assignment operator
 newString(const char *);
 //constructor; conversion from the char string
 newString();
 //default constructor to initialize the string to null
 newString(const newString&);
 //copy constructor
 ~newString();
 //destructor
 char &operator[] (int);
 const char &operator[] (int) const;
 //overload the relational operators
 bool operator==(const newString&) const;
 bool operator!=(const newString&) const;
 bool operator<=(const newString&) const;
 bool operator<(const newString&) const;
 bool operator>=(const newString&) const;
 bool operator>(const newString&) const;
```

```

private:
 char *strPtr; //pointer to the char array
 //that holds the string
 int strLength; //data member to store the length
 //of the string
};
#endif

```

newString 类有两个私有成员变量：分别用于存储字符串及其长度。

下面，给出实现 newString 类操作的函数定义。实现文件包括头文件 cassert，因为使用了 assert() 函数。第4章或附录F中的头文件 cassert 包含了对 assert() 函数的解释。

```

//Implementation file myString.cpp
#include <iostream>
#include <iomanip>
#include <cstring>
#include <cassert>
#include "myString.h"

using namespace std;

//constructor: conversion from the char string to newString
newString::newString(const char *str)
{
 strLength = strlen(str);
 strPtr = new char[strLength+1]; //allocate memory to store
 //the char string

 assert(strPtr != NULL);
 strcpy(strPtr, str); //copy string into strPtr
}

//default constructor to store the null string
newString::newString()
{
 strLength = 1;
 strPtr = new char[1];
 assert(strPtr != NULL);
 strcpy(strPtr, "");
}

newString::newString(const newString& rightStr) //copy constructor
{
 strLength = rightStr.strLength;
 strPtr = new char[strLength + 1];
 assert(strPtr != NULL);
 strcpy(strPtr, rightStr.strPtr);
}

newString::~newString() //destructor
{
 delete [] strPtr;
}

//overload the assignment operator
const newString& newString::operator=(const newString& rightStr)
{
 if(this != &rightStr) //avoid self-copy
 {
 delete [] strPtr;
 strLength = rightStr.strLength;
 }
}

```

```

 strPtr = new char[strLength + 1];
 assert(strPtr != NULL);
 strcpy(strPtr, rightStr.strPtr);
 }
 return *this;
}

char& newString::operator[] (int index)
{
 assert(0 <= index && index < strLength);
 return strPtr[index];
}

const char& newString::operator[] (int index) const
{
 assert(0 <= index && index < strLength);
 return strPtr[index];
}

//overload the relational operators
bool newString::operator==(const newString& rightStr) const
{
 return(strcmp(strPtr, rightStr.strPtr) == 0);
}

bool newString::operator<(const newString& rightStr) const
{
 return(strcmp(strPtr, rightStr.strPtr) < 0);
}

bool newString::operator<=(const newString& rightStr) const
{
 return(strcmp(strPtr, rightStr.strPtr) <= 0);
}

bool newString::operator>(const newString& rightStr) const
{
 return(strcmp(strPtr, rightStr.strPtr) > 0);
}

bool newString::operator>=(const newString& rightStr) const
{
 return(strcmp(strPtr, rightStr.strPtr) >= 0);
}

bool newString::operator!=(const newString& rightStr) const
{
 return(strcmp(strPtr, rightStr.strPtr) != 0);
}

//overload the stream insertion operator <<
ostream& operator<<(ostream& osObject, const newString& str)
{
 osObject<<str.strPtr;
 return osObject;
}

//overload the stream extraction operator >>
istream& operator>>(istream& isObject, newString& str)
{
 char temp[81];
 isObject>>setw(81)>>temp;

```

```

 str = temp;
 return isObject;
 }

```

考虑 operator>> 函数定义中的语句:

```
is>>setw(81)>>temp;
```

因为 temp 声明为大小为 81 的数组, 可以存储到 temp 中的最大字符串长度为 80。该语句 (输入语句) 中的操纵器 setw() 保证不超过 80 个字符被读到 temp 中。

这些函数的大部分是非常简单明了的。下面解释重载转换构造函数、赋值运算符和复制构造函数。

转换构造函数是单参数的函数, 它把参数转换为类的对象。在当前的情况下, 转换构造函数把一个字符串转换为 newString 类型的对象。

注意, 仅为 newString 类型的对象显式地重载了赋值运算符。不过, 重载的赋值运算符同样起作用, 如果想存储一个字符串到 newString 对象中的话。考虑声明:

```
newString str;
```

和语句:

```
str = "Hello there";
```

编译器把该语句翻译为:

```
str.operator=("Hello there");
```

1. 首先, 编译器自动地调用转换构造函数创建一个 newString 类型的对象临时存储字符串 "Hello there"。
2. 其次, 编译器调用重载的赋值运算符将临时的 newString 对象赋给 str 对象。

因此, 显式地重载赋值运算符将字符串存储到 newString 类型的对象中是不必要的。

下面, 编写一个 C++ 程序测试对 newString 类的一些操作。

```

//Test Program
#include <iostream>
#include "myString.h"
using namespace std;
int main()
{
 newString str1 = "Sunny"; //initialize str1 using
 //the assignment operator
 const newString str2("Warm"); //initialize str2 using
 //the conversion constructor
 newString str3; //initialize str3 to null
 newString str4; //initialize str4 to null

 cout<<"Line 1: "<<str1<<" "<<str2<<" ***"
 <<str3<<"###."<<endl; //Line 1

 if(str1 <= str2) //compare str1 and str2; Line 2
 cout<<"Line 3: "<<str1<<" is less than "<<str2
 <<endl; //Line 3
 else //Line 4
 cout<<"Line 5: "<<str2<<" is less than "
 <<str1<<endl; //Line 5
}

```



```

cout<<"Line 6: Enter a string with a length "
 <<"of at least 7 --> "; //Line 6
cin>>str1; //input str1; Line 7
cout<<endl<<"Line 8: The new value of str1 = "
 <<str1<<endl; //Line 8

str4 = str3 = "Birth Day"; //Line 9
cout<<"Line 10: str3 = "<<str3<<" , str4 = "
 <<str4<<endl; //Line 10

str3 = str1; //Line 11
cout<<"Line 12: The new value of str3 = "<<str3<<endl; //Line 12

str1 = "Bright Sky"; //Line 13
str3[1] = str1[5]; //Line 14
cout<<"Line 15: After replacing the second character "
 <<"of str3 = "<<str3<<endl; //Line 15

str3[2] = str2[0]; //Line 16
cout<<"Line 17: After replacing the third character "
 <<"of str3 = "<<str3<<endl; //Line 17

str3[5] = 'g'; //Line 18
cout<<"Line 19: After replacing the sixth character "
 <<"of str3 = "<<str3<<endl; //Line 19

return 0;

```

**运行实例：**在本运行实例中，用户输入用阴影表示。

```

Line 1: Sunny Warm ***###.
Line 3: Sunny is less than Warm
Line 6: Enter a string with a length of at least 7 --> 123456789
Line 8: The new value of str1 = 123456789
Line 10: str3 = Birth Day, str4 = Birth Day
Line 12: New value of str3 = 123456789
Line 15: After replacing the second character of str3 = 1t3456789
Line 17: After replacing the third character of str3 = 1tW456789
Line 19: After replacing the sixth character of str3 = 1tW45g789

```

前面的程序工作如下。第1行语句输出 str1、str2 和 str3 的值。注意 str3 的值打印在 \*\*\* 和 ### 之间。因为 str3 为空，\*\*\* 和 ### 之间没有任何输出，如 Line 1 所示。第2~5行语句比较 str1 和 str2，然后输出结果。第7行语句输入长度至少为7的字符串到 str1 中，第8行语句输出 str1 的新值。注意语句（参见第9行）：

```
str4 = str3 = "Birth Day";
```

因为赋值运算符的结合性为从右到左，首先语句“str3 = "Birth Day";”执行，然后语句“str4 = str3;”执行。第10行语句输出 str3 和 str4 的值。第14、16 和 18 行语句使用数组下标运算符[]单独地操作 str3 中的字符。

## 16.7 函数重载

前面一节讨论了运算符重载。类似于内建类型操作，对于用户自定义数据类型，运算符重载提供了一组简洁的操作符号。和运算符一起使用的参数类型决定了所进行的动作。类似于运算符重载，C++ 允许程序员重载函数名。在第7章中曾介绍了函数重载，为了在下面的讨论中更方便地参考，

让我们先复习一下该概念。

类可以有多个构造函数，但类所有的构造函数都有相同的名字，该名字和类名相同。这种情况是函数重载的一个例子。重载函数指的是有多个相同名字的函数，但有不同的参数。参数的类型决定了将执行哪一个函数。

为了使函数重载起作用，必须给出每个函数的定义。下一节将讲述如何重载带统一代码段的函数，并把为不同函数生成代码的工作留给编译器。

## 16.8 模板

模板是C++一个强有力的特性。它们允许程序员为一组相关的函数或类编写统一的代码段，分别称为函数模板和类模板。使用模板的语法为：

```
template <class Type>
 declaration;
```

这里Type为数据类型，declaration为函数声明或类声明。在C++中，template为保留字。头部中的class是任何用户定义类型或内建类型。Type是模板的形参。

和函数的参数为变量类似，模板的参数为类型（数据类型）。

### 16.8.1 函数模板

在第7章中曾介绍了函数重载，函数larger()被重载并计算两个整数、字符、浮点数或字符串中的较大者。为了实现函数larger()，需要为数据类型int、char、double和string编写4个函数定义。然而，每个函数的函数体是类似的。C++通过提供函数模板简化了重载函数的过程。

函数模板的一般语法为：

```
template <class Type>
 function definition;
```

这里Type是模板的形参，它用来指定函数的参数类型、返回类型和声明函数内的变量。下面的语句：

```
template <class Type>
Type larger(Type x, Type y)
{
 if(x >= y)
 return x;
 else
 return y;
}
```

定义了一个函数模板larger()，它返回两项中的较大者。在函数头部，形参x和y的类型为Type，它由函数调用时的实参类型指定。语句：

```
cout<<larger(5,6)<<endl;
```

为对函数模板larger()的调用。因为5和6为int类型，数据类型int取代了Type，编译器将生成相应的代码。

和通常的情况一样，如果忽略了函数模板定义中的函数体，函数模板就是原型。

下面的例子阐述了函数模板的使用。



**例 16.8**

在该例子中, 使用函数模板 `larger()` 来确定两项中的较大者。

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

template <class Type>
Type larger(Type x, Type y);

int main()
{
 cout<<"Line 1: Larger of 5 and 6 = "
 <<larger(5,6)<<endl; //Line 1

 cout<<"Line 2: Larger of A and B = "
 <<larger('A','B')<<endl; //Line 2

 cout<<"Line 3: Larger of 5.6 and 3.2 = "
 <<larger(5.6,3.2)<<endl; //Line 3

 newString str1 = "Hello"; //Line 4
 newString str2 = "Happy"; //Line 5

 cout<<"Line 6: Larger of "<<str1<<" and "
 <<str2<<" = "<<larger(str1,str2)
 <<endl; //Line 6

 return 0;
}

template<class Type>
Type larger(Type x, Type y)
{
 if(x >= y)
 return x;
 else
 return y;
}
```

**输出:**

```
Line 1: Larger of 5 and 6 = 6
Line 2: Larger of A and B = B
Line 3: Larger of 5.6 and 3.2 = 5.6
Line 6: Larger of Hello and Happy = Hello
```

**16.8.2 类模板**

与函数模板类似, 类模板用来为一组相关的类编写统一的代码段。例如, 在第 13 章中定义了一个抽象数据类型 `list`, 表元素类型为 `int`。如果表元素由 `int` 变为 `char`、`double` 或 `string`, 则需要为每一个元素类型分别编写不同的类。作用于表的操作和实现这些操作的算法大部分是相同的。使用类模板, 可以创建一个通用的类 `listType`, 编译器可以为特定的实现生成相应的源代码。

使用类模板的语法为:

```
template<class Type>
class declaration
```

类模板称为参数化的类型, 因为, 需要基于参数类型而产生特定的类。

下面的语句把 `listType` 定义为类模板:

```

template<class elemType>
class listType
{
public:
 bool isEmpty();
 //Function returns true if the list is empty;
 //otherwise, it returns false.
 bool isFull();
 //Function returns true if the list is full,
 //otherwise, it returns false.
 void search(const elemType& searchItem, bool& found);
 //Search the list for searchItem
 //Post: found is set to true if the
 //searchItem is found in the list; otherwise, found
 //is set to false.
 void insert(const elemType& newElement);
 //Insert newElement in the list
 //Prior to insertion, the list must not be full
 //Post: the list is an old list plus the
 //newElement
 void remove(const elemType& removeElement);
 //If removeElement is found in the list, it is deleted
 //If the list is empty, output the message "Cannot delete
 //from the empty list"
 //Post: the list is an old list minus
 //removeElement if removeElement is found in the list
 void destroyList();
 //Post: length = 0
 void printList();
 //Output the elements of the list
 listType();
 //default constructor
 //Set the length of the list to 0
 //Post: length = 0
protected:
 elemType list[100]; //array to hold the list elements
 int length; //variable to store the number of
 //elements in the list
};

```

类模板 listType 的定义是一种通用的定义, 只包括对表的基本操作。为了从该表派生一个特定的表、增加或重写操作, 可以声明包含表元素的数组和表长度为 protected 类型。

下面, 将描述一个特定的表。假设需要创建一个处理整型数据的表。语句:

```
listType<int> intList; //Line 1
```

声明 intList 为包括 100 个成员的表, 每个成员均为 int 类型。类似地, 语句:

```
listType<newString> stringList; //Line 2
```

声明 stringList 为包括 100 个成员的表, 每个成员均为 newString 类型。

在第 1 行和第 2 行语句中, listType<int> 和 listType<newString> 称为类模板 listType<elemType> 的模板实例化或实例化, 这里 elemType 为模板头部中的类参数。可以用内建类型或用户定义类型创建一个实例化模板。

类模板的函数成员被认为是函数模板。因此, 当给出一个类模板函数成员的定义时, 必须遵循函数模板的定义。例如, listType 类中成员 insert() 的定义为:

```

template<class elemType>
void listType<elemType>::insert(elemType newElement)
{
 .
 .
 .
}

```

在成员函数定义的头部，类的名字由参数 elemType 指定。

第 1 行语句声明 intList 为包括 100 个成员的列表。当编译器为 intList 生成代码时，它在 listType 类的定义中用 int 取代 elemType。listType 类成员函数定义中的模板参数（例如，insert() 定义中的 elemType）同样被 int 取代。

### 类模板的头文件和实现文件

到现在为止，我们把类的定义（在头文件中）和成员函数的定义（在实现文件中）放在不同的文件中。目标代码由实现文件生成，然后和用户代码连接在一起。然而，这种把类定义和成员函数定义分离的机制不适用于类模板。传递参数给函数是在运行时起作用，而传递参数给类模板是在编译时起作用。因为传递给类的实参是在用户代码中指定的，而编译器不能在没有传递给模板实参的情况下初始化函数模板，因此，不能独立于用户代码编译实现文件。

这一问题有几种可能的解决办法。可以把类定义和函数模板的定义直接放在客户程序中，或者可以把类定义和函数模板的定义一起放在同一个头文件中。另一种选择是把类定义和函数定义放在不同的文件中（和通常一样），但是需要在头文件的末尾包括一个对实现文件的编译指示符。在任何一种情况下，函数定义和客户代码都是在一起编译的。为了便于阐述，将把类定义和函数定义放在同一个头文件中。

下面的例子示范了类模板的使用。

### 例 16.9

在本例子中，将编写一个使用 listType 类的程序。所包括的一些操作如下：检查表是否为空、检查表是否已满、对表排序以及打印表。因为可以动态地分配数组，用户可以指定数组的大小。默认的数组大小为 50。程序将操作一个整数表和一个字符串表。因为先前定义的 newString 类允许使用关系运算符进行比较和使用赋值运算符来进行赋值，下面将使用 newString 类来声明字符串。

listType 类的定义为：

```

//Header file listType.h

#ifndef H_listType
#define H_listType

#include <iostream>
#include <cassert>
using namespace std;

template <class elemType>
class listType
{
public:
 bool isEmpty();
 //Returns true if the list is empty, false otherwise
 bool isFull();
 //Returns true if the list is full, false otherwise
 int getLength();
 //Returns the length of the list, which is the number of

```

```

 //elements currently in the list
int getMaxSize();
 //Returns the maximum number of elements that can
 //be stored in the list
void sort();
 //Sorts the list
 //Post: the list elements are in ascending order
void print() const;
 //Outputs the elements of the list
void insertAt(const elemType& item, int position);
 //Post: list[position] = item; length++;
 //If the position is out of range, the program is aborted
listType(int listSize = 50);
 //constructor
 //Creates an array of the size specified by the
 //parameter listSize; the default array size is 50
 //Post: list contains the base
 // address of the array, length = 0 and
 // maxsize = listSize

~listType();
 //destructor
 //Delete all elements of the list
 //Post: the array list is deleted
private:
 int maxSize; //maximum number that can be
 //stored in the list
 int length; //number of elements in the list
 elemType *list; //pointer to the array that holds the
 //list elements
};

template<class elemType>
bool listType<elemType>::isEmpty()
{
 return (length == 0)
}

template<class elemType>
bool listType<elemType>::isFull()
{
 return (length == maxSize);
}

template<class elemType>
int listType<elemType>::getLength()
{
 return length;
}

template<class elemType>
int listType<elemType>::getMaxSize()
{
 return maxSize;
}

 //constructor; the default array size is 50
template<class elemType>
listType<elemType>::listType(int listSize)
{

```



```
int number; //Line 4
cout<<"List 5: Processing the integer list"
 <<endl; //Line 5
cout<<"List 6: Enter 5 integers: "; //Line 6
for(counter = 0; counter < 5; counter++) //Line 7
{
 cin>>number; //Line 8
 intList.insertAt(number, counter); //Line 9
}

cout<<endl; //Line 10
cout<<"List 11: The list you entered is: "; //Line 11
intList.print(); //Line 12
cout<<endl; //Line 13
cout<<"Line 14: After sorting, the list is: "; //Line 14
intList.sort(); //Line 15
intList.print(); //Line 16
cout<<endl; //Line 17

newString str; //Line 18

cout<<"Line 19: Processing the string list"
 <<endl; //Line 19
cout<<"Line 20: Enter 5 strings: "; //Line 20
for(counter = 0; counter < 5; counter++) //Line 21
{
 cin>>str; //Line 22
 stringList.insertAt(str, counter); //Line 23
}

cout<<endl; //Line 24
cout<<"Line 25: The list you entered is: "
 <<endl; //Line 25
stringList.print(); //Line 26
cout<<endl; //Line 27
cout<<"Line 28: After sorting, the list is: "
 <<endl; //Line 28
stringList.sort(); //Line 29
stringList.print(); //Line 30
cout<<endl; //Line 31
int intListSize; //Line 32

cout<<"Line 33: Enter the size of the integer "
 <<"list: "; //Line 33
cin>>intListSize; //Line 34

listType<int> intList2(intListSize); //Line 35

cout<<"Line 36: Processing the integer list"
 <<endl; //Line 36
cout<<"Line 37: Enter "<<intListSize
 <<" integers: "; //Line 37

for(counter = 0; counter < intListSize; counter++) //Line 38
{
 cin>>number; //Line 39
 intList2.insertAt(number, counter); //Line 40
}

cout<<endl; //Line 41
```



```

 cout<<"Line 42: The list you entered is: "<<endl; //Line 42
 intList2.print(); //Line 43
 cout<<endl; //Line 44
 cout<<"Line 45: After sorting, the list is: "
 <<endl; //Line 45
 intList2.sort(); //Line 46
 intList2.print(); //Line 47
 cout<<endl; //Line 48
 cout<<"Line 49: Length of the list = "
 <<intList2.getLength()<<endl; //Line 49
 cout<<"Line 50: Maximum size of the list = "
 <<intList2.getMaxSize()<<endl; //Line 50

 return 0;
 }

```

**运行实例：**在本运行实例中，用户输入用阴影表示。

```

List 5: Processing the integer list
List 6: Enter 5 integers: 19 15 66 24 34

List 11: The list you entered is: 19 15 66 24 34

Line 14: After sorting, the list is: 15 19 24 34 66

Line 19: Processing the string list
Line 20: Enter 5 strings: summer cold winter warm sunny

Line 25: The list you entered is:
summer cold winter warm sunny
Line 28: After sorting, the list is:
cold summer sunny warm winter

Line 33: Enter the size of the integer list: 10
Line 36: Processing the integer list
Line 37: Enter 10 integers: 23 65 34 8 11 5 3 16 45 2

Line 42: The list you entered is:
23 65 34 8 11 5 3 16 45 2

Line 45: After sorting, the list is:
2 3 5 8 11 16 23 34 45 65

Line 49: Length of the list = 10
Line 50: Maximum size of the list = 10

```

## 16.9 小结

1. 如果运算符对于不同数据类型具有不同的意义，则称为运算符重载。
2. 在 C++ 中，>> 用做流析取运算符和右移运算符。类似地，<< 用做流插入运算符和左移运算符。两者均为运算符重载的例子。
3. 重载运算符的函数称为运算符函数。
4. 运算符函数头部的语法为：  
returnType operator operatorSymbol(parameters)
5. 在 C++ 中，operator 为保留字。
6. 运算符函数是返回值的函数。
7. 除了赋值运算符和成员选择运算符之外，如果要对类重载一个运算符，该运算符必须被重载。默认情况下，赋值运算符按成员进行复制。

8. 对于带指针成员变量的类, 赋值运算符必须被显式地重载。
9. 运算符重载为用户定义的数据类型提供了和内建类型一样的简明符号。
10. 当重载运算符时, 不能改变它的优先级, 也不能改变它的结合性。默认参数不能和重载的运算符一起使用, 不能改变运算符的参数个数。运算符对于内建数据类型的含义保持不变。
11. 不能创建新的运算符。只能重载现有的运算符。
12. 大部分的 C++ 运算符可以被重载。
13. 不能被重载的运算符包括 `., *, ::, ?:` 和 `sizeof`。
14. `this` 指针引用一个整体的对象。
15. 重载运算符 `()`、`[]`、`->` 或 `=` 的运算符函数必须是类的成员函数。
16. 友元函数为类的非成员函数。
17. 友元函数的头部前面为保留字 `friend`。
18. 在 C++ 中, `friend` 为保留字。
19. 如果一个运算符函数为类的成员函数, 运算符最左面的操作数必须是该类的对象(或对该类对象的引用)。
20. 作为类的成员函数, 二元运算符函数只有一个参数; 作为非成员函数则有两个参数。
21. 为类重载流插入运算符 `<<` 和流析取运算符 `>>` 的运算符函数必须是该类的友元函数。
22. 重载前递增 `++` 运算符时, 运算符函数为该类的成员函数, 它必须不含参数。类似地, 重载前递减 `--` 运算符时, 运算符函数为该类的成员函数, 它必须不含参数。
23. 重载后递增 `++` 运算符时, 运算符函数为该类的成员函数, 它必须有一个参数, 类型为 `int`。用户不需指定该参数的值。函数头部的虚参数帮助编译器生成正确的代码。后递减运算符也有类似的规范。
24. 复制构造函数以一个对象的值初始化相同类型的另一个对象。必须传递一个引用参数给所复制的对象。
25. 当一个形参为值参数时, 复制构造函数将传递实参的值给形参。
26. 转换构造函数是一个单参数的函数。
27. 转换构造函数转换其参数为该类的对象。编译器隐式地调用这样的函数。
28. 包含指针成员变量的类必须重载赋值运算符并包含复制构造函数和析构函数。
29. 在 C++ 中, 函数名可以被重载。
30. 重载函数的每个实例有不同参数集。
31. 在 C++ 中, `template` 为保留字。
32. 使用模板, 可以为一组相关的函数编写统一的代码段——称为函数模板。
33. 使用模板, 可以为一组相关的类编写统一的代码段——称为类模板。
34. 模板的语法为:

```
template <class Type>
declaration;
```

这里 `Type` 为用户定义的标志符并作为参数进行类型(数据类型)传递, `declaration` 为函数或类的声明。头部中的 `class` 是任何用户定义的数据类型或内建数据类型。
35. 类模板称为参数化的类型。
36. 在类模板中, 参数 `Type` 指明如何定制一个通用类模板为特定的模板类。
37. 参数 `Type` 在每个类头部和成员函数的定义中使用。
38. 假设 `cType` 为类模板, `func()` 为 `cType` 的成员函数。 `func()` 函数定义的头部为:



```
template <class Type>
funcType cType<Type>::func(parameters)
```

这里 funcType 为函数的类型，如 void。

39. 假设 cType 为类模板，可以用 int 作为参数。语句：

```
cType<int> x;
```

声明 x 为 cType 类型的对象，传递给 cType 类的类型为 int。

## 16.10 练习

1. 判断下列语句的正确性。

- 在 C++ 中，所有的运算符都可以为用户定义数据类型而重载。
- 在 C++ 中，运算符不能为内建数据类型而重定义。
- 重载运算符的函数称为运算符函数。
- C++ 允许用户创建自己的运算符。
- 不能改变运算符的优先级，但可以改变运算符的结合性。
- 重载函数的每个实例都有相同的参数数量。
- 无须为只有 int 型成员变量的类重载关系运算符。
- 类模板的成员函数是一个函数模板。
- 当编写友元函数的定义时，关键字 friend 必须出现在函数的头部。
- 模板提供了软件重用的能力。
- 重载前递增运算符 (++) 和后递增运算符 (++ ) 的运算符函数时，其函数头部是相同的，因为两个运算符有相同的符号。

2. 什么是友元函数？

3. 假设用户定义类 mystery 并重载运算符 <<。为什么必须重载 << 为友元函数？

4. 假设二元运算符 + 重载为 strange 类的成员函数。函数 operator+ 包含多少个参数？

5. 类应该什么时候重载赋值运算符和定义复制构造函数？

6. 考虑下面的声明：

```
class strange
{
.
.
.
};
```

- 在 strange 类中编写一条重载运算符 >> 的声明语句。
- 在 strange 类中编写一条重载运算符 = 的声明语句。
- 在 strange 类中编写一条重载二元运算符 + 为成员函数的声明语句。
- 在 strange 类中编写一条重载运算符 == 为成员函数的声明语句。
- 在 strange 类中编写一条重载后递增运算符 ++ 为成员函数的声明语句。

7. 采用练习 6 的声明。

- 在 strange 类中编写一条重载二元运算符 + 为友元函数的声明语句。
- 在 strange 类中编写一条重载运算符 == 为友元函数的声明语句。
- 在 strange 类中编写一条重载后递增运算符 ++ 为友元函数的声明语句。

8. 在下列代码中查找错误：

```

class mystery //Line 1
{
 ...
 bool operator <= (mystery); //Line 2
 ...
};

bool mystery::<=(mystery rightObj) //Line 3
{
 ...
}

```

9. 在下列代码中查找错误:

```

class mystery //Line 1
{
 ...
 bool operator <= (mystery, mystery); //Line 2
 ...
};

```

10. 在下列代码中查找错误:

```

class mystery //Line 1
{
 ...
 friend operator+ (mystery); //Line 2
 //overload binary +
 ...
};

```

11. 重载前递增运算符为类的成员函数时需要多少个参数?
12. 重载前递增运算符为类的友元函数时需要多少个参数?
13. 重载后递增运算符为类的成员函数时需要多少个参数?
14. 重载后递增运算符为类的友元函数时需要多少个参数?
15. 对于复数  $a+ib$ , 其共轭值为  $a-ib$ ,  $a+ib$  的绝对值为  $\sqrt{a^2+b^2}$ 。扩展编程实例中 complexType 类的定义: 重载运算符  $\sim$  和  $!$  为成员函数, 使得  $\sim$  返回复数的共轭值而  $!$  返回绝对值。写出这些运算符函数的定义。
16. 重做练习 15, 将  $\sim$  和  $!$  重载为非成员函数。

17. 在下列代码中查找错误:

```

template <class type> //Line 1
class strange //Line 2
{
 ...
};

strange<int> s1; //Line 3
strange<type> s2; //Line 4

```

18. 考虑下面的声明:

```

template <class type>
class strange
{
 ...
private:
 Type a;
 Type b;
};

```

- a. 编写一条语句, 声明 sObj 为 strange 类型的对象, 私有成员变量 a 和 b 为 int 类型。
- b. 在 strange 类中编写一条重载运算符 == 为成员函数的声明语句。
- c. 如果两个 strange 类型对象的相应成员变量相等的话, 则认为这两个对象是相等的。写出重载为 strange 类成员函数的 operator== 函数定义。

19. 考虑下面函数模板的定义:

```
template <class Type>
Type surprise(Type x, Type y)
{
 return x + y ;
}
```

下列语句的输出是什么?

- a. `cout<<surprise(5,7)<<endl;`
- b. `string str1 = "Sunny";`  
`string str2 = " Day";`  
`cout<<surprise(str1, str2)<<endl;`

20. 考虑下面函数模板的定义:

```
Template <class Type>
Type funcExp(Type list[], int size)
{
 int j;
 Type x = list[0];
 Type y = list[size -1];
 for(j = 1; j < (size -1)/2; j++)
 {
 if(x < list[j])
 x = list[j];
 if(y > list[size -1 -j])
 y = list[size -1 -j];
 }
 return x + y;
}
```

假设有下列的声明:

```
int list[10] = { 5, 3, 2, 10, 4, 19, 45, 13, 61, 11};
string strList[] = { "One", "Hello", "Four", "Three", "How", "Six"};
```

下列语句的输出是什么?

- a. `cout<<funExp(list,10);`
- b. `cout<<funExp(strList,6)<<endl;`

21. 写出交换两个变量内容的函数模板定义。

22. a. 为 newString 类重载运算符 + 以进行字符串连接。例如, 如果 s1 为 "Hello", s2 为 "there", 语句:

```
s3 = s1 + s2;
```

应该把 "Hello there" 赋值给 s3, 这里 s1、s2 和 s3 为 newString 类型的对象。

b. 为 newString 类重载运算符 += 以进行下面的字符串连接: 假设 s1 为 "Hello", s2 为 "there"。那么语句:

```
s1 += s2;
```

应该把 "Hello there" 赋值给 s1, 这里 s1 和 s2 为 newString 类型的对象。

## 16.11 编程练习

1. a. 写出重载递增、递减、算术和关系运算符为 OpOverClass 类成员函数的函数定义。  
b. 编写测试程序, 测试对 OpOverClass 类的各种不同操作。
2. a. 写出重载递增、递减、算术和关系运算符为 OpOverClass 类非成员函数的函数定义。  
b. 测试程序, 测试对 OpOverClass 类的各种不同操作。
3. a. 通过重载后递增运算符函数为 clockType 成员函数而扩展 clockType 类的定义。  
b. 写出为 a 部分中所定义的 clockType 类重载后递增运算符的函数定义。
4. a. 前面 clockType 类中的递增和关系运算符重载为成员函数。重写 clockType 类的定义, 把这些运算符重载为非成员函数。此外, 重载后递增运算符为 clockType 类的非成员函数。  
b. 写出 a 部分中所设计的 clockType 类的成员函数定义。  
c. 编写测试程序, 测试 a 和 b 部分所设计类中的各种操作。
5. a. 扩展 complexType 类的定义, 使得它能进行减法和除法操作。重载减法和除法运算符为该类的成员函数。  
如果  $(a,b)$  和  $(c,d)$  为复数,  $(a,b)-(c,d)=(a-c,b-d)$ 。  
如果  $(c,d)$  非零,  
 $(a,b)/(c,d) = ((ac+bd)/(c^2+d^2), (-ad+bc)/(c^2+d^2))$ 。  
b. 写出重载 a 部分所定义的运算符 - 和 / 的函数定义。  
c. 编写测试程序, 测试对 complexType 类的各种不同操作。结果格式化为两位小数。
6. a. 重写 complexType 类的定义, 重载算术和关系运算符为非成员函数。  
b. 写出 a 部分所定义的 complexType 类成员函数定义。  
c. 编写测试程序, 测试对 a 和 b 部分中所设计的 complexType 类的各种不同操作。结果格式化为两位小数。
7. a. 扩展 newString 类的定义如下:  
i. 重载运算符 + 和 += 进行字符串连接。  
ii. 增加函数 length() 返回字符串的长度。  
b. 写出实现 a 部分中所定义操作的函数定义。  
c. 编写测试程序, 测试对 newString 对象的各种不同操作。
8. a. 重写编程练习 7 中所定义和扩展的 newString 类定义, 重载关系运算符为非成员函数。  
b. 写出 a 中所设计的 newString 类定义。  
c. 编写测试程序, 测试对 newString 类的各种不同操作。
9. 有理分式的格式为  $a/b$ , 这里 a 和 b 为整数且 b 不为 0。在本练习中, “分式” 指的是有理分式。假设  $a/b$  和  $c/d$  为分式, 对分式的算术运算由下列规则定义:

$$a/b + c/d = (ad+bc)/bd$$

$$a/b - c/d = (ad-bc)/bd$$

$$a/b \times c/d = ac/bd$$

$$(a/b)/(c/d) = ad/bc, c/d \neq 0.$$

其中, 比较规则如下所示: 如果  $ad \text{ op } bc$ , 则  $a/b \text{ op } c/d$ 。这里 op 为任何一种关系运算。例如, 如果  $ad < bc$ , 则  $a/b < c/d$ 。

设计一个类 fraction 对分式进行算术和关系运算。重载算术和关系运算符, 使得可以用适当的符号来进行运算。此外, 重载流插入和析取运算符以方便输入和输出。

编写一个 C++ 程序, 使用 fraction 类对分式进行运算。

此外, 测试下面的论述。假设  $x$ 、 $y$  和  $z$  为 fraction 类型的对象。如果输入为 2/3, 语句:

```
cin>>x;
```

应该存储 2/3 到  $x$  中。语句:

```
cout<<x+y<<endl;
```

应该以分式的格式输出  $x+y$  的值。语句:

```
z = x + y;
```

应该以分式的格式存储  $x$ 、 $y$  的和在  $z$  中。答案不必为最简形式。

10. 在 C++ 中没有对数组越界进行检查。然而, 在程序执行中, 数组下标越界可能会导致严重的问题。C++ 中数组下标同样是以 0 开始。

设计和实现 myArray 类, 解决数组下标越界的问题, 同时允许用户以任何正或负的整数值开始数组下标。每个 myArray 类型的对象是一个 int 类型的数组。在执行中, 当访问数组成员的时候, 如果下标越界, 程序必须以适当的错误信息终止。考虑下面的语句:

```
myArray<int> list(5); //Line 1
```

```
myArray<int> myList(2,13); //Line 2
```

```
myArray<int> yourList(-5,9); //Line 3
```

第 1 行语句声明 list 为包括 5 个成员的数组, 成员类型为 int, 成员为: list[0], list[1], ..., list[4];

第 2 行语句声明 myList 为包括 11 个成员的数组, 成员类型为 int, 成员为: myList[2], myList[3], ...,

myList[12]; 第 3 行语句声明 yourList 为包括 14 个成员的数组, 成员类型为 int, 成员为: yourList[-5], yourList[-4], ..., yourList[0], ..., yourList[8]。编写一个程序测试 myArray 类。

11. 编程练习 10 只处理整型数组。使用类模板重新设计 myArray 类, 使得该类可以用于任何需要数组处理数据的应用程序。

12. 设计一个类进行各种不同的矩阵操作。矩阵是一组按行和列排列的数列。所以, 矩阵的每个元素都有一个行位置和一个列位置。如果  $A$  是一个 5 行 6 列的矩阵, 我们说矩阵  $A$  大小为  $5 \times 6$ , 有时候记为  $A_{5 \times 6}$ 。很明显, 二维数组适合用于存储矩阵。两个矩阵如果大小相同就可以进行相加和相减。假设  $A=[a_{ij}]$  和  $B=[b_{ij}]$  是两个大小为  $m \times n$  的矩阵,  $a_{ij}$  表示  $A$  第  $i$  行第  $j$  列的元素, 等等。  $A$  和  $B$  的和及差由下列公式给出:

$$A+B=[a_{ij}+b_{ij}]$$

$$A-B=[a_{ij}-b_{ij}]$$

$A$  和  $B$  的乘法操作 ( $A*B$ ) 只有在  $A$  的列数和  $B$  的行数相等的情况下才是有意义的。如果  $A$  大小为  $m \times n$ ,  $B$  大小为  $n \times t$ , 那么  $A*B=[c_{ik}]$  大小为  $m \times t$ , 元素  $c_{ik}$  由公式:

$$c_{ik}=a_{i1}b_{1k}+a_{i2}b_{2k}+\dots+a_{in}b_{nk}$$

给出。设计和实现 matrixType 类, 该类可以存储任何大小的矩阵。重载运算符 +、- 和 \*, 分别执行加法、减法和乘法运算, 重载运算符 << 以输出矩阵。此外, 编写测试程序测试各种不同的矩阵运算。

13. a. 在第 13 章的编程练习 1 中, 定义了一个 romanType 类在程序中实现罗马数字。在该练习中, 还实现了一个函数 romanToDecimal(), 将罗马数字转换为对应的十进制数字。

修改 romanType 类的定义, 使得成员变量声明为 protected。使用编程练习 7 中的 newString 类操作字符串。此外, 重载流插入和析取运算符, 以得到更方便的输入和输出形式。流插入运算符以罗马格式输出罗马数字。

此外, 还应该包括一个成员函数 decimalToRoman(), 将十进制数字 (必须为正整数) 转换为对应的罗马数字格式。写出成员函数 decimalToRoman() 的定义。

为了简单起见,假定只有字母I可以出现在其他字母的前面,而且它只出现在字母V和X的之前。例如,4表示为IV,9表示为IX,39表示为XXXIX,49表示为XXXIX。此外,40表示为XXXX,190表示为CLXXXX,等等。

- b. 从 romanType 类派生出 extRomanType 类。在 extRomanType 类中,重载算术运算符 +、-、\* 和 / 使得对罗马数字可以进行算术运算。此外,重载前递增和后递增运算符及递减运算符为 extRomanType 类的成员函数。

为了对罗马数字进行加法(减法、乘法或除法)运算,需要用它们的十进制表示法进行加法(减法、乘法或除法)运算,然后把结果转换为罗马数字格式。对于减法,如果第一个数字小于第二个数字,输出信息应指出“因为第一个数字小于第二个,不能相减。”类似地,对于除法,分子必须大于分母。递增和递减运算符也有类似的规定。

- c. 写出重载 b 部分中所描述运算符的函数定义。  
d. 在下面的程序中测试 extRomanType 类。(包括适当的头文件。)

```
int main()
{
 extRomanType num1("XXXIV");
 extRomanType num2("XV");
 extRomanType num3;

 cout<<"Num1 = "<<num1<<endl;
 cout<<"Num2 = "<<num2<<endl;
 cout<<"Num1 + Num2 = "<<num1+num2<<endl;
 cout<<"Num1 * Num2 = "<<num1*num2<<endl;

 cout<<"Enter two numbers in Roman format: ";
 cin>>num1>>num2;
 cout<<endl;

 cout<<"Num1 = "<<num1<<endl;
 cout<<"Num2 = "<<num2<<endl;

 num3 = num2 * num1;
 cout<<"Num3 = "<<num3<<endl;

 cout<<"--num3: "<<--num3<<endl;
 cout<<"++num3: "<<++num3<<endl;

 return 0;
}
```

14. 例16.9设计了类模板 listType 在程序中实现表操作。为了阐述方便,例子只包含了排序操作。扩展该类模板的定义,以包括删除和查找操作。写出实现类模板 listType 成员函数的定义。此外,编写测试程序测试对表的各种不同操作。

15. (股票市场)编写程序帮助本地的股票交易公司自动化其股票系统。该公司只在股票市场投资。公司打算在每个交易日后生成并公布它的股票清单,让投资者可以了解他们的股票在当天的交易状况。假定该公司只投资10种不同的股票。所需的输出是产生两个列表,一个按股票符号排序,另一个按每股收益从高到低排序。

输入数据以下面的格式在文件中提供:

symbol openingPrice closingPrice todayHigh todayLow prevClose volume

例如,示例数据为:

MSMT 112.50 115.75 116.50 111.75 113.50 6723823

CBA 67.50 75.50 78.75 67.50 65.75 378233



第1行指明股票符号为MSMT,当天的开盘价为112.50,收盘价为115.75,当日最高价为116.50,当日最低价为111.75,昨日收盘价为113.50,所持有的股票数为6723823。

按股票符号排序的清单必须是下面的格式:

```
***** First Investor's Heaven *****
***** Financial Report *****
Stock Today Previous Percent
Symbol Open Close High Low Close Gain Volume

 ABC 123.45 130.95 132.00 125.00 120.50 8.67% 10000
 AOLK 80.00 75.00 82.00 74.00 83.00 -9.64% 5000
 CSCO 100.00 102.00 105.00 98.00 101.00 0.99% 25000
 IBD 68.00 71.00 72.00 67.00 75.00 -5.33% 15000
 MSET 120.00 140.00 145.00 140.00 115.00 21.74% 30920
Closing Assets: $9628300.00
--*
```

以两个步骤逐步展开本编程练习。在第一步中(a部分),设计和实现一个股票对象。在第二步中(b部分),设计和实现一个对象来操作股票列表。

a. (股票对象)设计和实现股票对象。该类记录股票对象的各种不同特征,称为 stockType。

股票的数据包括股票符号、股票价格和股票数。此外,需要输出开盘价、最高价、最低价、前一天的价格和当天的每股收益/损失。这些是股票的全部特征。所以,股票对象应该存储所有这些信息。

对每个股票对象执行下列操作:

- i. 设置股票信息。
- ii. 打印股票信息。
- iii. 显示不同的价格。
- iv. 计算和打印收益/损失。
- v. 显示持股数。

a.1. 股票列表的自然排序方式是按股票符号排序。重载关系运算符,通过两个股票符号来比较股票对象。

a.2. 重载插入运算符<<以方便输出。

a.3. 因为数据存储在文件中,重载流析取运算符>>以方便输入。

例如,假设 infile 为一个 ifstream 对象,使用 infile 对象打开输入文件。假设 myStock 为一个股票对象。那么,语句:

```
infile>>myStock;
```

从输入文件中读取数据,然后存储在 myStock 对象中。(注意,该语句读取数据,然后把数据存储在 myStock 的相关成员中。)

b. 既然已经设计和实现了 stockType 类从而可以在程序中实现一个股票对象,现在创建一个股票对象清单。

可以把实现股票对象清单的类称为 stockListType。

stockListType 类必须派生于在前面的练习中设计和实现的 listType 类。然而,stockListType 类是一个非常具体的类,用来创建一个股票对象清单。所以,stockListType 类不再是一个模板。

增加和/或重写 listType 类的操作,以实现对股票清单必需的操作。

下面的语句从 `listType` 类派生出 `stockListType` 类：

```
class stockListType: public listType<stockType>
{
 member list
};
```

成员变量中的清单元素、清单的长度和最大清单尺寸在 `listType` 类中声明为 `protected`。所以，这些成员可以在 `stockListType` 中直接访问。

因为该公司还要求产生按收益/损失排序的清单，需要把股票清单按该要素进行排序。然而，没必要把清单按每股收益/损失手工地排序，而是提供一个按要素进行排序的逻辑。为了达到该目的，增加一个数组成员变量，保存按每股收益/损失排序的股票清单的下标值。该数组称为 `sortIndicesGainLoss`。当打印按每股收益/损失排序的清单时，使用数组 `sortIndicesGainLoss` 来打印清单。数组 `sortIndicesGainLoss` 的元素将指出所要打印的股票清单的成员。

- c. 编写一个程序，使用这两个类来自动处理该公司的股票数据分析过程。



# 第17章 链 表

## 本章要点：

- 学习链表
- 了解链表的基本特性
- 研究链表的插入和删除操作
- 研究如何建立和操作一个链表
- 学习包含头和尾节点的链表
- 学习如何构造一个双向链表

我们已经知道数据是如何通过使用一个称为顺序表的数组进行有序地组织和处理的。我们已经对顺序表执行了几种操作，如排序、插入、删除和查找。如果数据没有排序，那么在表中查找一项可能会非常耗时，特别是对于大的顺序表。一旦数据进行了排序，可以使用二分查找改进查找算法。然而，在这种情况下，插入和删除将会非常耗时，特别是对于大的顺序表，因为这些操作需要数据移动。此外，因为数组大小在执行时必须是固定的，只有在空间允许的情况下才能增加新的表项。所以，使用数组进行数据组织是有局限性的。

本章将克服上述的一些问题。第15章说明了如何使用指针进行内存（变量）的动态分配和释放。本章将使用指针组织和处理表中的数据，称为链表。当数据存储在一个数组中时，数组成员的内存是连续的——就是说，内存块是连续分配的。

## 17.1 链表

链表是一组称为节点的成员集合。每个节点（除了最后一个节点）包含下一个节点的地址。因此，链表中的每个节点有两个组成部分：一个存储相关的信息（就是数据），另一个存储地址，称为链表中下一个节点的链接。链表中第一个节点的地址存储在一个单独的位置，称为 head（头节点）或 first（首节点）。图 17.1 为一个节点的图示。

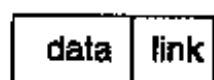


图 17.1 节点的结构

**链表：**一系列称为节点的项，其中，节点的顺序由存储在每个节点中称为链接的地址决定。图 17.2 中的表是链表的一个例子。

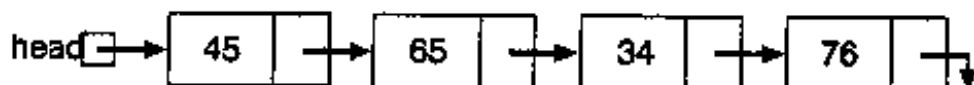


图 17.2 链表

最后一个节点的向下箭头说明该链接字段为 NULL。每个节点中的箭头表示所指的节点地址存储在该节点中。为了更好地理解这一符号，假设第一个节点位于存储单元 1200，第二个节点位于存储单元 1575。如图 17.3 所示。

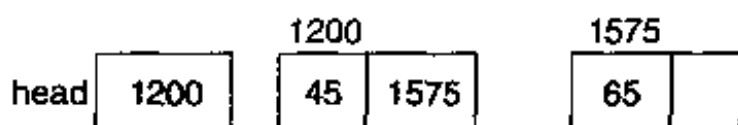


图 17.3 链表和链接的值

head 的值为 1200, 第 1 个节点的数据部分为 45, 链接部分包含 1575, 即第 2 个节点的地址。为了方便描述, 将在画链表图时使用箭头符号。

因为链表的每个节点有两个组成部分, 需要声明每个节点为类或结构。每个节点的数据类型取决于具体的应用, 即所处理的数据类型。然而, 每个节点的链接部分都是一个指针。该指针变量所指向的数据类型是节点本身的类型。对于前面的链表, 节点的定义如下。(假设数据的类型为 int。)

```

struct nodeType
{
 int info;
 nodeType *link;
};

```

变量的声明为:

```
nodeType *head;
```

### 17.1.1 链表: 一些特性

为了更好地理解链表和节点的概念, 下面描述链表的一些重要特性。考虑图 17.4 中的链表。

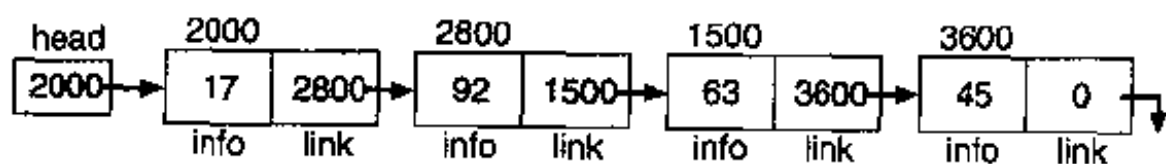


图 17.4 包括 4 个节点的链表

该链表有 4 个节点。第 1 个节点的地址存储在指针 head 中。每个节点有两个组成部分: info 存储信息, link 存储下一个节点的地址。为了简单起见, 假定 info 的类型为 int。

假设第 1 个节点位于存储单元 2000, 第 2 个节点位于存储单元 2800, 第 3 个节点位于存储单元 1500, 第 4 个节点位于存储单元 3600。所以, head 的值为 2000, 第 1 个节点链接部分的值为 2800, 第 2 个节点链接部分的值为 1500, 等等。此外, 最后一个节点链接部分的值为 0 说明该值为 NULL, 通过一个向下的箭头表示。每个节点顶部的数字为该节点的地址。下面的表显示了该链表的属性。

|                  | 值    |                                                 |
|------------------|------|-------------------------------------------------|
| head             | 2000 |                                                 |
| head->info       | 17   | 因为 head 为 2000 而存储单元 2000 处节点的 info 值为 17       |
| head->link       | 2800 |                                                 |
| head->link->info | 92   | 因为 head->link 为 2800 而存储单元 2800 处节点的 info 值为 92 |

假设 current 是一个和 head 指针相同类型的指针。那么语句:

```
current = head;
```

将 head 的值复制到 current 中。(参见图 17.5。)

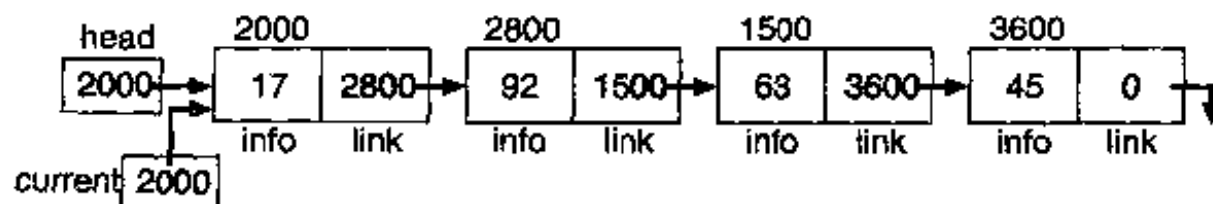


图 17.5 在“current = head;”执行后的链表

很明显，在图 17.5 中：

|                     | 值    |
|---------------------|------|
| current             | 2000 |
| current->info       | 17   |
| current->link       | 2800 |
| current->link->info | 92   |

现在考虑语句：

```
current = current->link;
```

该语句把 current->link 的值 2800 复制到 current。所以，在该语句执行后，current 指向该链表的第 2 个节点。（当使用链表时，通常使用这种语句来把指针指向链表中的下一个节点。）参见图 17.6。

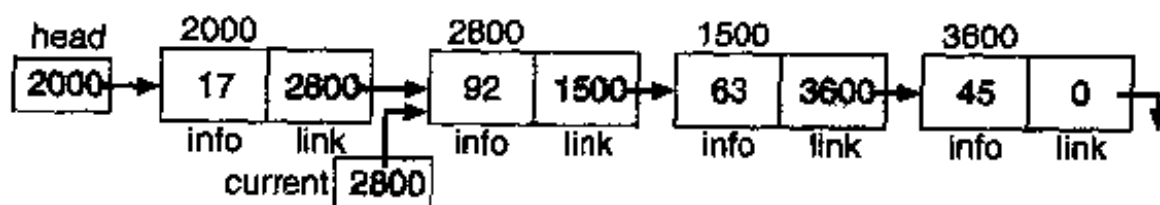


图 17.6 在“current = current-&gt;link;”执行后的链表

在图 17.6 中：

|                     | 值    |
|---------------------|------|
| current             | 2800 |
| current->info       | 92   |
| current->link       | 1500 |
| current->link->info | 63   |

最后，注意在图 17.6 中：

|                                 | 值        |
|---------------------------------|----------|
| head->link->link                | 1500     |
| head->link->link->info          | 63       |
| head->link->link->link          | 3600     |
| head->link->link->link->info    | 45       |
| current->link->link             | 3600     |
| current->link->link->info       | 45       |
| current->link->link->link       | 0 (NULL) |
| current->link->link->link->info | 不存在      |

从现在起，当使用链表时，将只用箭头符号。

### 遍历链表

链表的基本操作如下：查找链表以确定某个特定的项是否在链表中、在链表中插入一项、从链表中删除一项。这些操作需要遍历链表。就是说，给定指向链表第 1 个节点的指针后，必须从头遍

历链表。

假设 head 指针指向链表的第 1 个节点, 最后一个节点的链接部分为 NULL。不能使用 head 指针来遍历链表, 因为如果这样的话将失去链表的头节点。该问题的产生是由于链接是单向的。head 指针包含第一个节点的地址, 第一个节点包含第二个节点的地址, 第二个节点包含第三个节点的地址, 等等。如果移动 head 使其指向第二个节点, 第一个节点就失去了 (除非保存指向该节点的指针)。如果持续移动 head 指向下一个节点, 将失去该链表的所有节点。(除非在移动 head 后保存每个节点的指针, 而这是不实际的, 因为这将需要额外的计算机时间和维持链表的存储空间。)

所以, 总是需要 head 指向第一个节点。那么必须使用另一个相同类型的指针来遍历链表。假设 current 是一个和 head 类型相同的指针。下面的代码遍历链表:

```
current = head;
while(current != NULL)
{
 //Process current
 current = current->link;
}
```

例如, 假设 head 指向一个数值链表。下面的代码输出每个节点中存储的数据:

```
current = head;
while(current != NULL)
{
 cout<<current->info<<" ";
 current = current->link;
}
```

### 17.1.2 链表项的插入和删除

本节讨论如何在一个链表中插入和删除一项。考虑下面的节点定义。(为简单起见, 假定 info 的类型为 int。下一节讨论使用模板及作为抽象数据类型 (ADT) 的链表, 这将使用节点的通用定义。)

```
struct nodeType
{
 int info;
 nodeType *link;
};
```

对于下面的变量声明:

```
nodeType *head, *p, *q, *newNode;
```

**插入**

考虑如图 17.7 所示的链表:

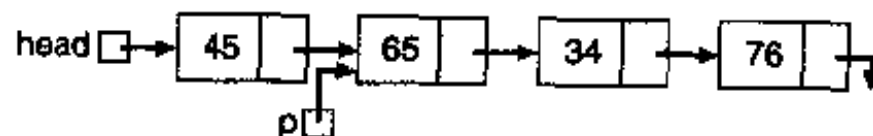


图 17.7 插入前的链表

假设 p 指向 info 值为 65 的节点, 一个 info 值为 50 的新节点将被创建和插入到 p 之后。下面的语句创建一个新的节点以及如何将 50 存储到 info 字段中:

```
newNode = new nodeType; //create newNode
newNode->info = 50; //store 50 in the new node
```

第1条语句“`newNode = new nodeType;`”在内存的某处创建一个节点，并把新节点的地址存储到 `newNode` 中。第2条语句“`newNode->info = 50;`”把 50 存储到新节点的 `info` 字段中。（参见图 17.8。）

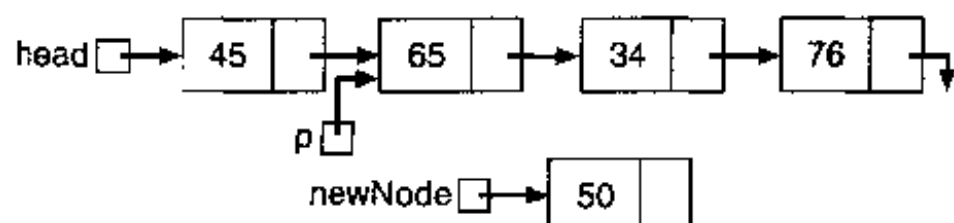


图 17.8 创建 `newNode` 并存储 50

下面的语句在链表中所要求的位置插入节点：

```
newNode->link = p->link;
p->link = newNode;
```

在第1条语句“`newNode->link = p->link;`”执行后，链表如图 17.9 所示。

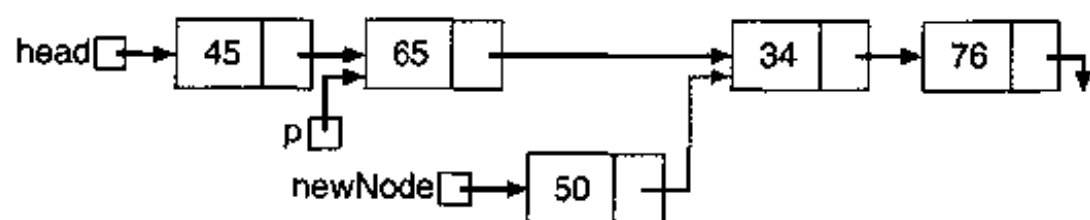


图 17.9 “`newNode->link = p->link;`”执行后的链表

在第2条语句“`p->link = newNode;`”执行后，链表如图 17.10 所示。

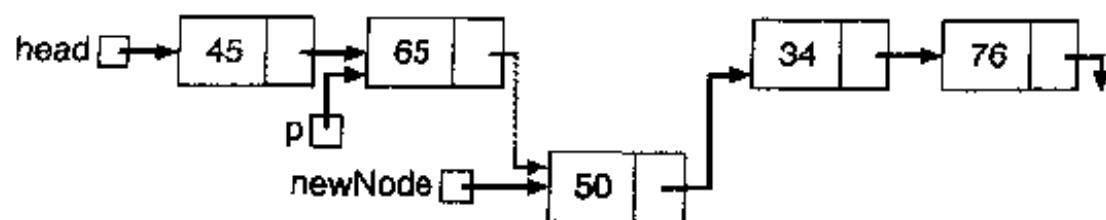


图 17.10 “`p->link = newNode;`”执行后的链表

注意插入节点的语句次序是非常重要的，因为要在链表中插入 `newNode`，只需要一个指针 `p` 来调整链表节点的链接。假设颠倒语句的次序，以下面的次序执行语句：

```
p->link = newNode;
newNode->link = p->link;
```

图 17.11 显示了这些语句执行后的链表。

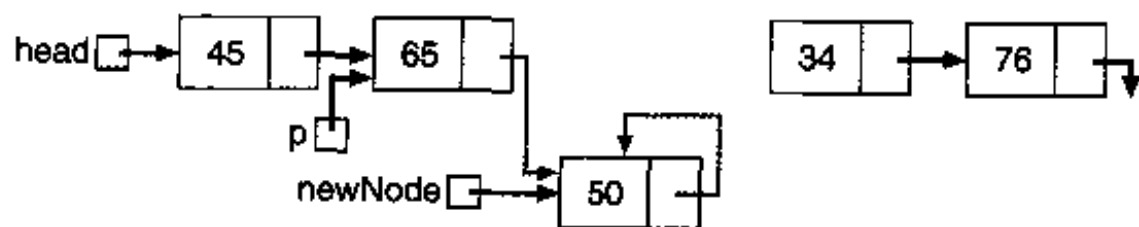


图 17.11 “`p->link = newNode;`”及“`newNode->link = p->link;`”执行后的链表

从图 17.11 中可以清楚地看出：`newNode` 指向自己，链表的剩余部分失去了。

可以使用两个指针稍微简化插入代码。假设 `q` 指向 `info` 值为 34 的节点。（参见图 17.12。）

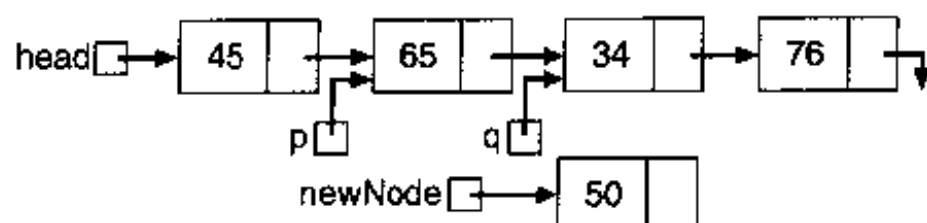


图 17.12 带指针 p 和 q 的链表

下面的语句在 p 和 q 之间插入 newNode:

```
newNode->link = q;
p->link = newNode;
```

这些语句执行的次序并不重要。为了阐述这种情况，假设以下面的次序执行语句：

```
p->link = newNode;
newNode->link = q;
```

在语句“p->link = newNode;”执行后，链表如图 17.13 所示。

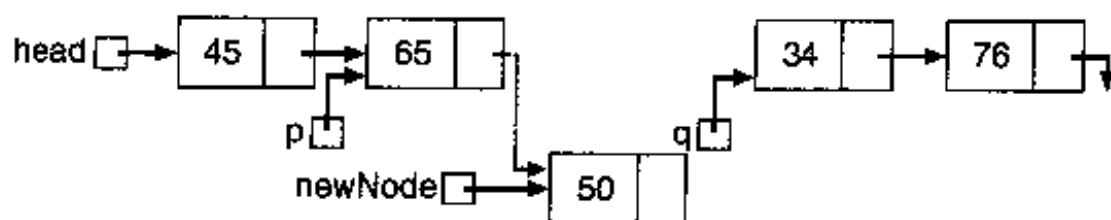


图 17.13 “p-&gt;link = newNode;”执行后的链表

因为指针 q 指向剩余的链表，剩余的链表并没有丢失。在语句“newNode->link = q;”执行后，链表如图 17.14 所示。

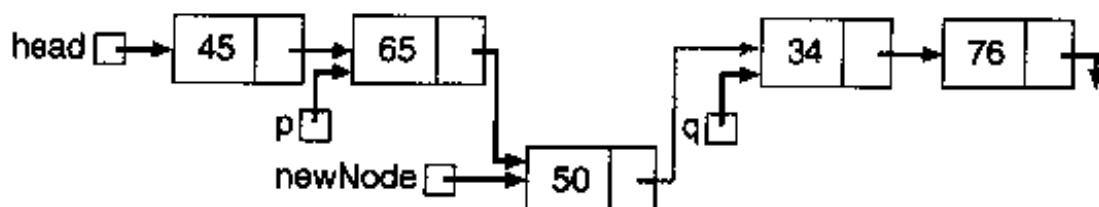


图 17.14 “newNode-&gt;link = q;”执行后的链表

## 删除

考虑图 17.15 所示的链表。

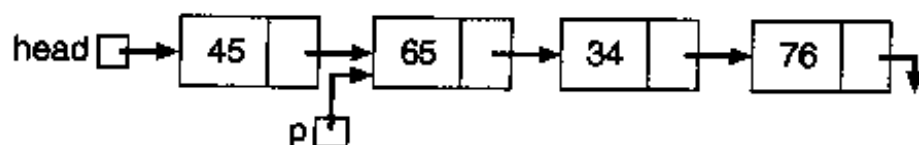


图 17.15 待删除的节点 info 值为 34

假设 info 值为 34 的节点将从链表中删除。下面的语句把节点从链表中删除：

```
p->link = p->link->link;
```

图 17.16 显示了前述语句执行后的链表。

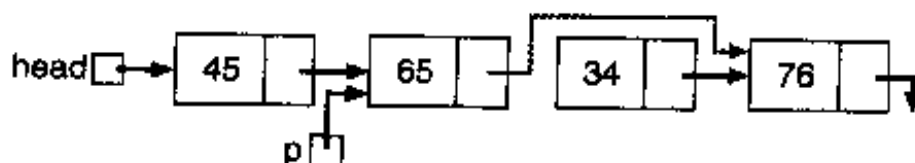


图 17.16 “p-&gt;link = p-&gt;link-&gt;link;”执行后的链表

从图 17.16 可以清楚地看出，info 值为 34 的节点从链表中删除。然而，该节点仍然占用内存，就是说，该节点被悬空了。为了释放内存，需要一个指向该节点的指针。下面的语句从链表中删除



节点, 然后释放该节点所占据的内存:

```
q = p->link;
p->link = q->link;
delete q;
```

在语句“`q = p->link;`”执行后, 链表如图 17.17 所示。

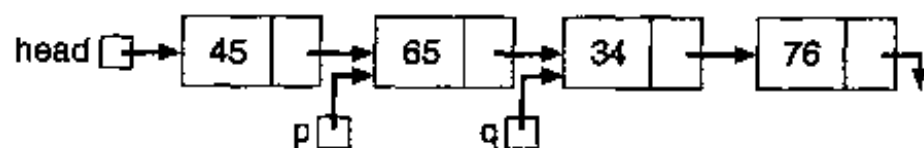


图 17.17 “`q = p->link;`”执行后的链表

在语句“`p->link = q->link;`”执行后, 链表如图 17.18 所示。

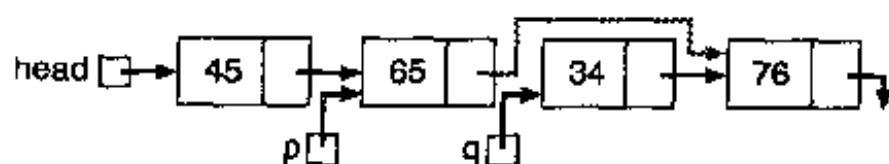


图 17.18 “`p->link = q->link;`”执行后的链表

在语句“`delete q;`”执行后, 链表如图 17.19 所示。

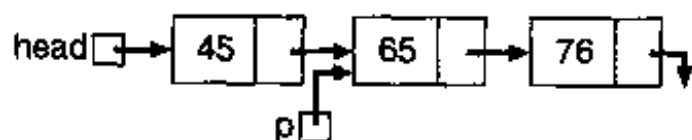


图 17.19 “`delete q;`”执行后的链表

### 17.1.3 建立一个链表

既然已经知道如何在链表中插入一个节点, 下面学习如何建立一个链表。首先, 考虑一个通用的链表。如果读取的数据没有排序的话, 链表将是没有顺序的。这样的链表有两种建立方式: 向前式和向后式。在向前式中, 新的节点总是插入到链表的末端; 在向后式中, 新的节点总是插入到链表的开始。下面将讨论这两种情况

#### 建立一个向前式的链表

假设节点为通常的 info-link 形式, info 的类型为 int。假定处理下面的数据:

2 15 8 24 34

需要三个指针来建立链表: 一个指针指向链表的第 1 个节点, 该指针不能被删除; 一个指针指向链表的最后一个节点; 还有一个指针用来创建新的节点。考虑下面的变量声明:

```
nodeType *first, *last, *newNode;
int num;
```

假设 first 指向链表的第 1 个节点。最初的链表是空的, 因此 first 和 last 均为 NULL。那么, 必须用语句:

```
first = NULL;
last = NULL;
```

来初始化 first 和 last 为 NULL。

下面, 考虑下列语句:

```
1 cin >> num; //read and store a number in num
```

```

2 newNode = new nodeType; //allocate memory of the type nodeType
 //and store the address of the
 //allocated memory in newNode
3 newNode->info = num; //copy the value of num into the
 //info field of newNode
4 newNode->link = NULL; //initialize the link field of
 //newNode to NULL
5 if (first == NULL) //if first is NULL, the list is empty;
 //make first and last point to newNode
 {
5a first = newNode;
5b last = newNode;
 }
6 else //list is not empty
 {
6a last->link = newNode; //insert newNode at the end of the list
6b last = newNode; //set last so that it points to the
 //actual last node in the list
 }

```

现在执行这些语句。最初，first 和 last 为 NULL。所以链表如图 17.20 所示。

在语句 1 执行后，num 为 2。语句 2 创建一个节点并将该节点的地址存储到 newNode 中。语句 3 将 2 存储到 newNode 的 info 字段中，语句 4 存储 NULL 到 newNode 的 link 字段中。（参见图 17.21。）

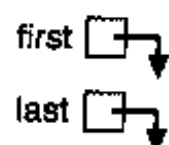


图 17.20 空链表

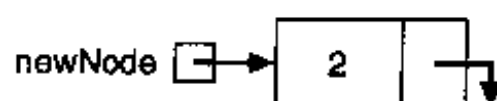


图 17.21 info 值为 2 的 newNode

因为 first 为 NULL，所以执行语句 5a 和 5b。图 17.22 显示了结果链表。

现在重复语句 1 到 6b。在语句 1 执行后，num 为 15。语句 2 创建一个节点，把该节点的地址存储到 newNode 中。语句 3 存储 15 到 newNode 的 info 字段中，语句 4 存储 NULL 到 newNode 的 link 字段中。（参见图 17.23。）

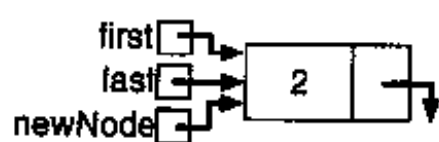


图 17.22 插入 newNode 后的链表

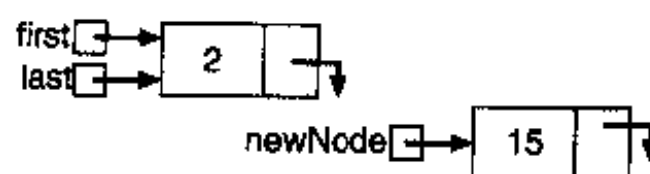


图 17.23 链表和 info 值为 15 的 newNode

因为 first 不为 NULL，所以执行语句 6a 和 6b。图 17.24 显示了结果链表。

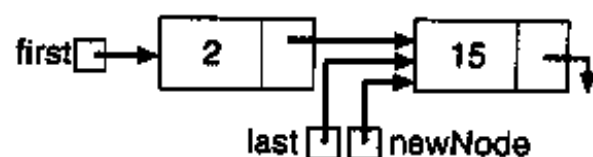


图 17.24 在末端插入 newNode 后的链表

现在重复三次语句 1 到 6b。图 17.25 显示了结果链表。

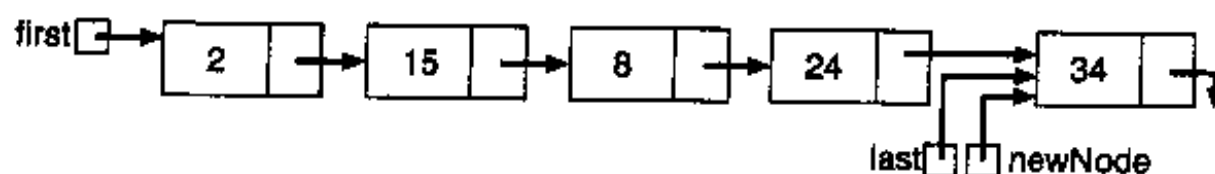


图 17.25 在插入 8、24 和 34 后的链表



可以把前面的语句放在一个循环中,然后执行循环直到某个特定的条件满足,从而建立链表。实际上可以编写一个 C++ 函数来建立一个链表。

假设读取一个以 -999 结束的整数集。下面的函数 buildListForward() (以向前的方式) 建立一个链表,然后返回指向所建立链表的指针:

```
nodeType* buildListForward()
{
 nodeType *first, *newNode, *last;
 int num;

 cout<<"Enter a list of integers ending with -999.\n";
 cin>>num;
 first = NULL;

 while (num != -999)
 {
 newNode = new nodeType;
 newNode->info = num;
 newNode->link = NULL;

 if (first == NULL)
 {
 first = newNode;
 last = newNode;
 }
 else
 {
 last->link = newNode;
 last = newNode;
 }
 cin>>num;
 } //end while

 return first;
} //end buildListForward
```

### 建立一个向后的链表

现在考虑建立一个向后链表的情况。对于前面给出的数据 2、15、8、24 和 34, 链表如图 17.26 所示。

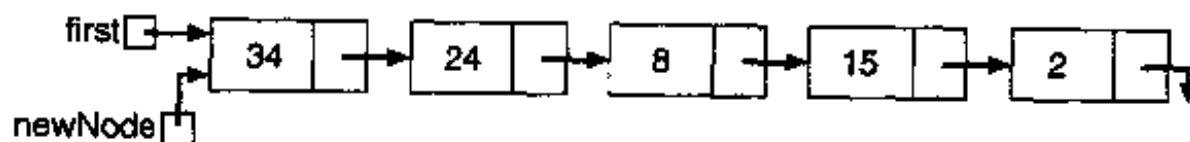


图 17.26 以向后方式建立的链表

因为新的节点总是插入在链表的开始,不需要知道链表的末端,因此指针 last 是不必要的。此外,在插入新的节点于链表的开始后,新的节点成为链表的第 1 个节点。所以,必须更新指针 first 的值,使它正确地指向链表的第 1 个节点。可以看出,只需要两个指针来建立链表:一个指针指向链表头,一个指针创建新的节点。因为最初链表为空,指针 first 必须初始化为 NULL。伪代码形式的算法为:

1. 初始化 first 为 NULL。
2. 对于数据中的每一项:
  - a. 创建新的节点 newNode。
  - b. 存储数据项到 newNode 中。

- c. 在 first 之前插入 newNode。
- d. 更新指针 first 的值。

下面的 C++ 函数以向后方式建立链表，然后返回所建链表的指针：

```
nodeType* buildListBackward()
{
 nodeType *first, *newNode;
 int num;

 cout<<"Enter a list of integers ending with -999.\n";
 cin>>num;
 first = NULL;

 while (num != -999)
 {
 newNode = new nodeType; //create a node
 newNode->info = num; //store the data in newNode
 newNode->link = first; //put newNode at the beginning of
 //the list
 first = newNode; //update the head pointer of the
 //list, that is, first
 cin>>num; //read the next number
 }

 return first;
} //end buildListBackward
```

## 17.2 作为 ADT 的链表

前一节讲述了链表的基本特性以及如何构造和操作链表。因为链表是非常重要的数据结构，所以本节讨论作为抽象数据类型（ADT）的链表，而不是特定的链表如整数链表或字符串链表。本节使用模板给出链表的一个通用定义，该定义将在下一节和本书的后面章节中使用。本章的编程实例也将使用这个通用的链表定义。

作用于链表的基本操作为：

1. 初始化链表。
2. 检查链表是否为空。
3. 检查链表是否已满。
4. 打印链表。
5. 取得链表的长度。
6. 销毁链表。
7. 在链表中查找给定项。
8. 在链表中插入一项。
9. 从链表中删除一项。

下面的链表操作同样包括在内。它用于检查第 1 个节点的信息。

10. 获取第 1 个节点中包含的信息。

首先，讨论作用于任意链表的操作——就是说，链表可以是有序的或无序的。下一节讨论有序链表。

如果一个链表是无序的,那么可以在链表的开始或末端插入新的项。此外,这样的链表最初可以用向前或向后的方式建立。函数 `buildListForward()` 要求新的项插入到末端,而函数 `buildListBackward()` 要求新的项插入到开始处。为了使这两种情况都适用,将编写两个函数: `insertFirst()` 在链表的开始插入新项, `insertLast()` 在链表的末端插入新项。此外,为了使算法更加有效率,将在链表中维持两个指针: `first` 指向链表的第一个节点; `last` 指向链表的最后一个节点。

下面的类定义了作为 ADT 的链表:

```
//Definition of the node

template<class Type>
struct nodeType
{
 Type info;
 nodeType<Type> *link;
};

template<class Type>
class linkedListType
{
public:
 const linkedListType<Type>& operator=
 (const linkedListType<Type>&);
 //Overload the assignment operator
 void initializeList();
 //Initialize the list to an empty state
 //Post: first = NULL, last = NULL
 bool isEmptyList();
 //Function returns true if the list is empty;
 //otherwise, it returns false
 bool isFullList();
 //Function returns true if the list is full;
 //otherwise, it returns false
 void print();
 //Output the data contained in each node
 //Post: None
 int length();
 //Return the number of elements in the list
 void destroyList();
 //Delete all nodes from the list
 //Post: first = NULL, last = NULL
 void retrieveFirst(Type& firstElement);
 //Return the data contained in the first node of the list
 //Post: firstElement = first element of the list
 void search(const Type& searchItem);
 //Outputs "Item is found in the list" if searchItem is in
 //the list; otherwise, outputs "Item is not in the list"
 void insertFirst(const Type& newItem);
 //newItem is inserted in the list
 //Post: first points to the new list and
 // newItem is inserted at the beginning of the list
 void insertLast(const Type& newItem);
 //newItem is inserted in the list
 //Post: first points to the new list,
 // newItem is inserted at the end of the list, and
 // last points to the last node in the list
 void deleteNode(const Type& deleteItem);
 //If found, the node containing deleteItem is deleted
 //from the list
 //Post: first points to the first node and
```

```

 // last points to the last node of the updated list
LinkedListType();
 //default constructor
 //Initializes the list to an empty state
 //Post: first = NULL, last = NULL
LinkedListType(const LinkedListType<Type>& otherList);
 //copy constructor
~LinkedListType();
 //destructor
 //Deletes all nodes from the list
 //Post: list object is destroyed

protected:
 NodeType<Type> *first; //pointer to the first node of the list
 NodeType<Type> *last; //pointer to the last node of the list
};

```

注意LinkedListType类的成员变量为protected而不是private, 因为将从该类派生出其他的类。本章稍后的“有序链表”一节和第18章中将引用到该类。

LinkedListType类的定义包括一个重载赋值运算符的成员函数。对于包括指针成员变量的类, 赋值运算符必须显式地重载(参见第15章和第16章)。因此, 该类的定义同样包括一个复制构造函数。下面讨论成员函数的实现。

成员函数isEmptyList()和isFullList()是很直观的。如果first为NULL则链表为空。因为存储数据的内存是动态分配的,(逻辑上)该链表不会为满。(该链表只有在用完存储空间的情况下才为满。)因此, 函数isFullList总是返回false。实现这些操作的函数定义如下:

```

template<class Type>
bool LinkedListType<Type>::isEmptyList()
{
 return(first == NULL);
}

template<class Type>
bool LinkedListType<Type>::isFullList()
{
 return false;
}

```

### 默认构造函数

默认构造函数LinkedListType()是很简单易懂的。它只是把链表初始化为空状态。我们知道, 当声明一个LinkedListType类型的对象而没有传递值时, 默认构造函数将自动地执行。

```

template<class Type>
LinkedListType<Type>::LinkedListType() //default constructor
{
 first = NULL;
 last = NULL;
}

```

### 销毁链表

函数destroyList()释放每个节点所占据的内存。从第一个节点起遍历链表, 使用delete运算符释放内存。这里需要一个临时指针来释放内存。一旦整个链表被销毁, 必须把first和last设为NULL。

```

template<class Type>
void LinkedListType<Type>::destroyList()
{

```

```

nodeType<Type> *temp; //pointer to deallocate the memory
 //occupied by the node
while(first != NULL) //while there are nodes in the list
{
 temp = first; //set temp to the current node
 first = first->link; //advance first to the next node
 delete temp; //deallocate the memory occupied by temp
}
last = NULL; //initialize last to NULL; first has already
 //been set to NULL by the while loop
}

```

### 初始化链表

函数initializeList()初始化链表为空状态。注意当声明链表对象时,默认构造函数或复制构造函数已经初始化了链表。该操作实际上重新初始化链表为空状态,因此它必须删除链表中的节点(如果有的话)。这一任务可以使用destroyList()操作完成,该操作还把指针first和last设为NULL。

```

template<class Type>
void linkedListType<Type>::initializeList()
{
 destroyList(); //if the list has any nodes, delete them
}

```

### 打印链表

成员函数print()打印每个节点中包含的数据。为了打印包含在每个节点中的数据,必须从第1个节点开始遍历链表。因为指针first总是指向链表的第1个节点,因此需要另一个指针来遍历链表。(如果使用first遍历链表,整个链表都将丢失。)

```

template<class Type>
void linkedListType<Type>::print()
{
 nodeType<Type> *current; //pointer to traverse the list
 current = first; //set current so that it points to
 //the first node
 while(current != NULL) //while there is more data to print
 {
 cout<<current->info<<" ";
 current = current->link;
 }
} //end print

```

### 链表的长度

为了获取链表的长度(即链表中有多少个节点),可以设置一个计数器(初始化为0),然后遍历链表,对于每一个节点需要将计数器增加1。

```

template<class Type>
int linkedListType<Type>::length()
{
 int count = 0;
 nodeType<Type> *current; //pointer to traverse the list
 current = first;
 while (current!= NULL)
 {
 count++;
 current = current->link;
 }
}

```

```
 return count;
 } //end length
```

### 获取第1个节点的数据

函数 `retrieveFirst()` 返回第1个节点包含的信息，其定义如下所示。

```
template<class Type>
void linkedListType<Type>::retrieveFirst(Type& firstElement)
{
 firstElement = first->info; //copy the info of the first node
} //end retrieveFirst
```

### 搜索链表

成员函数 `search()` 在链表中查找指定项。如果找到该项，函数输出 “Item is found in the list”；否则输出 “Item is not in the list”。因为链表不是任意访问的数据结构，必须从第1个节点起按次序搜索链表。

下面的步骤描述该函数：

1. 比较查找项和链表中的当前节点。如果当前节点的 `info` 字段和查找项的相同，停止查找；否则，使下一个节点成为当前节点。
2. 重复步骤1直到找到查找项或链表中再也没有数据可以和查找项相比较。

```
template<class Type>
void linkedListType<Type>::search(const Type& item)
{
 nodeType<Type> *current; //pointer to traverse the list
 bool found;

 if(first == NULL) //list is empty
 cout<<"Cannot search an empty list. "<<endl;
 else
 {
 current = first; //set current to point to the first
 //node in the list
 found = false; //set found to false
 while(!found && current != NULL) //search the list
 {
 if(current->info == item) //item is found
 found = true;
 else
 current = current->link; //make current point to
 //the next node
 }
 if(found)
 cout<<"Item is found in the list."<<endl;
 else
 cout<<"Item is not in the list."<<endl;
 } //end else
} //end search
```

### 插入第1个节点

函数 `insertFirst()` 在链表的开始插入新的项，即在 `first` 指向的节点之前。实现该函数需要下列步骤

1. 创建新的节点。
2. 存储新的项到新节点中。
3. 在 `first` 之前插入新节点。



```

template<class Type>
void linkedListType<Type>::insertFirst(const Type& newItem)
{
 nodeType<Type> *newNode; //pointer to create the new node

 newNode = new nodeType<Type>; //create the new node
 newNode->info = newItem; //store the new item in the node
 newNode->link = first; //insert newNode before first
 first = newNode; //make first point to the
 //actual first node

 if(last == NULL) //if the list was empty, newNode is also
 //the last node in the list
 last = newNode;
}

```

### 插入最后一个节点

成员函数 insertLast() 的定义和成员函数 insertFirst() 的定义类似。这里在 last 后插入新的节点。函数 insertLast() 如下所示:

```

template<class Type>
void linkedListType<Type>::insertLast(const Type& newItem)
{
 nodeType<Type> *newNode; //pointer to create the new node
 newNode = new nodeType<Type>; //create the new node
 newNode->info = newItem; //store the new item in the node
 newNode->link = NULL; //set the link field of newNode
 //to NULL

 if(first == NULL) //if the list is empty, newNode is
 //both the first and last node
 {
 first = newNode;
 last = newNode;
 }
 else //the list is not empty, insert newNode after last
 {
 last->link = newNode; //insert newNode after last
 last = newNode; //make last point to the actual last node
 }
} //end insertLast

```

### 删除节点

下面讨论成员函数 deleteNode() 的实现, 该函数从链表中删除 info 字段为指定 info 值的节点。需要考虑下面几种情况:

1. 链表为空。
2. 第 1 个节点的 info 字段为指定值。在这种情况下, 需要调整指针 first。
3. info 字段为指定值的节点在链表中的某处。如果所要删除的节点为最后一个节点, 必须调整指针 last。
4. 链表不包含 info 字段为指定值的节点。

如果链表为空, 可以简单地打印一条信息, 说明链表为空。如果链表不为空, 搜索链表查找 info 字段为指定值的节点。如果找到这样的节点, 删除该节点。伪代码形式的算法为:

```

if list is empty
 Output(cannot delete from an empty list);

```

```

else
{
 if the first node is the node with the given info,
 adjust the head pointer, that is, first, and deallocate
 the memory;
 else
 {
 search the list for the node with the given info;
 if such a node is found, delete it
 }
}

```

**情况 1: 链表为空。**

如果链表为空, 显示一条如伪代码所示的错误信息。

**情况 2: 链表不为空。所要删除的节点为第 1 个节点。**

这种情况有两种情形: 链表只有一个节点以及链表有多于一个的节点。考虑有一个节点的链表, 如图 17.27 所示。

假设要删除 37。在删除后, 链表为空。所以, 在删除后, first 和 last 均设置为 NULL。

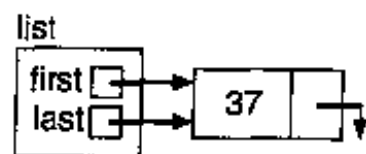


图17.27 有一个节点的链表

现在考虑有多于一个节点的链表, 如图 17.28 所示。

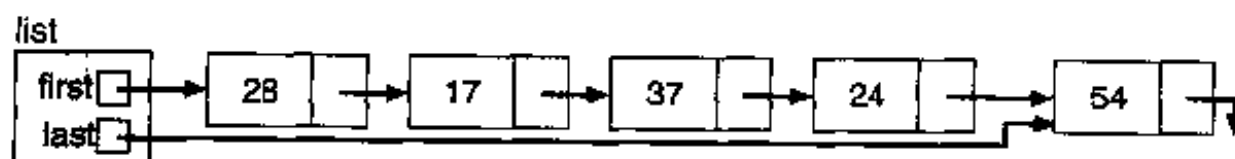


图 17.28 有多于一个节点的链表

假设将要删除的节点为 28。在删除该节点后, 第 2 个节点成为第 1 个节点。所以, 在删除该节点后, 指针 first 的值改变了, 就是说, 在删除节点后, first 包含 info 字段为 17 的节点地址。图 17.29 显示了删除 28 后的链表。

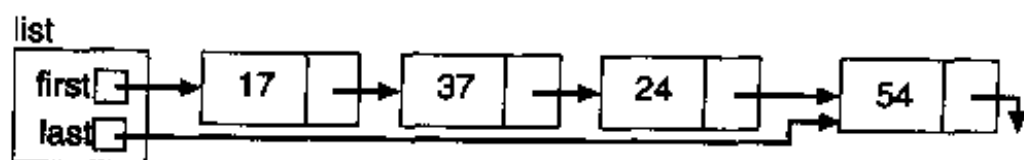


图 17.29 删除 info 值为 28 的节点后的链表

**情况 3: 所删除的节点不是第 1 个节点, 但在该链表中的某处。**

这种情况有两种情形: (a) 所要删除的节点不是最后一个节点, (b) 所要删除的节点是最后一个节点。下面分别阐述这两种情形。

**情况 3a: 所要删除的节点是最后一个节点。**

考虑图 17.30 所示的链表。

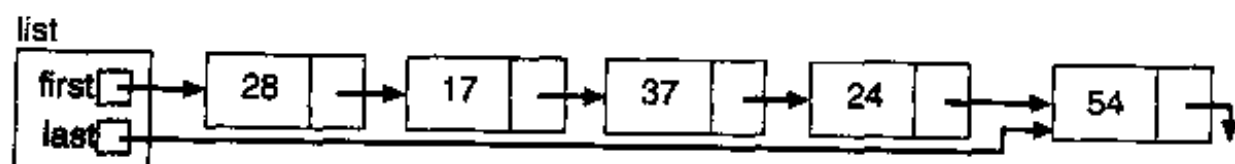


图 17.30 删除 37 之前的链表

假设所要删除的节点为 37。在删除该节点后, 链表如图 17.31 所示。注意, 删除 37 不需要改变 first 和 last 的值。前一个节点 (17) 的 link 字段的值改变了。在删除节点后, info 值为 17 的节点包含节点 24 的地址。



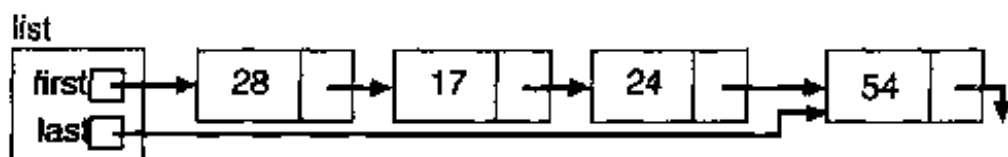


图 17.31 删除 37 后的链表

情况 3b: 所要删除的节点为最后一个节点。

考虑图 17.32 所示的链表。假设所要删除的节点为 54。

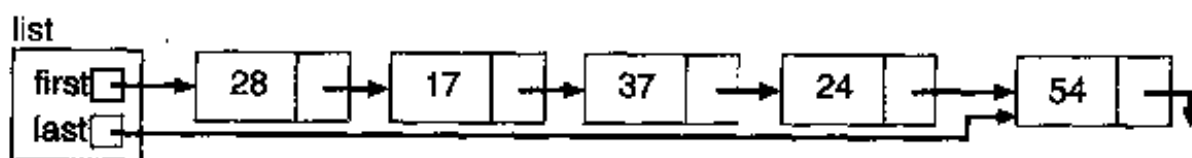


图 17.32 删除 54 之前的链表

在删除 54 后, info 值为 24 的节点成为最后一个节点。所以, 删除 54 要求改变指针 last 的值。在删除 54 后, last 包含 info 值为 24 的节点地址。图 17.33 显示了结果链表。

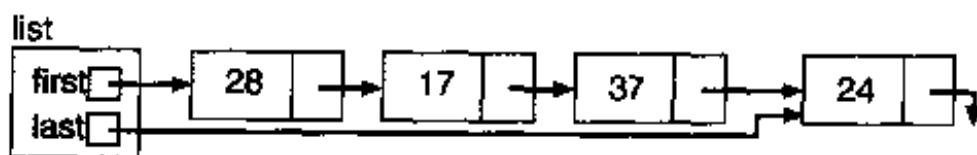


图 17.33 删除 54 后的链表

情况 4: 所要删除的节点不在链表中。在这种情况下, 链表不需要调整。可以简单地输出一条信息, 说明所要删除的项不在链表中。

从情况 2、3 和 4 可以知道, 删除节点需要遍历链表。因为链表不是任意访问的数据结构, 所以必须按次序搜索链表。可以单独地处理情况 1, 因为它不需要遍历链表。我们从第 2 个节点开始按次序搜索链表, 如果所要删除的节点在链表的中间, 则需要调整所要删除节点的前一个节点的 link 字段。因此, 需要一个指针指向前一个节点。当搜索链表查找指定 info 值时, 需要使用两个指针: 一个指针检查当前节点的 info 值, 另一个指针记录当前节点的前一个节点。如果所要删除的节点为最后一个节点, 必须调整指针 last。

函数 deleteNode 的定义为:

```
template<class Type>
void linkedListType<Type>::deleteNode(const Type& deleteItem)
{
 nodeType<Type> *current; //pointer to traverse the list
 nodeType<Type> *trailCurrent; //pointer just before current
 bool found;

 if(first == NULL) //Case 1: list is empty
 cout<<"Cannot delete from an empty list.\n";
 else
 {
 if(first->info == deleteItem) //Case 2
 {
 current = first;
 first = first->link;
 if(first == NULL) //list had only one node
 last = NULL;
 delete current;
 }
 else //search the list for the node with the given info
 {
```

```

found = false;
trailCurrent = first; //set trailCurrent to point to
 //the first node
current = first->link; //set current to point to the
 //second node

while((!found) && (current != NULL))
{
 if(current->info != deleteItem)
 {
 trailCurrent = current;
 current = current->link;
 }
 else
 found = true;
} // end while

if(found) //Case 3: if found, delete the node
{
 trailCurrent->link = current->link;

 if(last == current) //node to be deleted was
 //the last node
 last = trailCurrent; //update the value of last
 delete current; //delete the node from the list
}
else
 cout<<"Item to be deleted is not in the list."<<endl;
} //end else
} //end deleteNode

```

### 析构函数

在给出复制构造函数的定义之前,让我们先考虑析构函数。析构函数的目的是当类对象超出作用域时释放链表节点所占据的内存。因为内存是动态分配的,重新设置first和last指针并不释放链表节点所占据的内存。因此必须遍历链表,从第1个节点开始,删除链表中的每一个节点。现在还不清楚实现析构函数的函数定义是否和函数destroyList()的定义类似,这里出于完整性考虑先给出它的定义。回忆一下,当类对象超出作用域时,析构函数将自动地执行。

```

template<class Type>
linkedListType<Type>::~~linkedListType() //destructor
{
 nodeType<Type> *temp;

 while(first != NULL) //while there are nodes left in the list
 {
 temp = first; //set temp to point to the current node
 first = first->link; //advance first to the next node
 delete temp; //deallocate memory occupied by temp
 } //end while

 last = NULL; //initialize last to NULL; first is already NULL
} //end destructor

```

### 复制构造函数

因为linkedListType类包含指针成员变量,所以该类的定义包含复制构造函数。我们知道,如果形参为值参数,复制构造函数提供它自己的数据复制给形参。当声明一个对象且使用其他对象对其进行初始化时,复制构造函数同样执行。(第15章中有更详细的信息。)

复制构造函数对链表进行完全相等的数据复制。所以,需要从第一个节点起遍历所要复制的链表。对应于原始链表中的每一个节点:

1. 创建一个节点,称为 newNode。
2. 复制(原始链表中)节点的 info 到 newNode 中。
3. 在新创建链表的末端插入 newNode。

复制构造函数的定义如下:

```
template<class Type>
LinkedListType<Type>::LinkedListType
 (const LinkedListType<Type>& otherList)
{
 NodeType<Type> *newNode; //pointer to create a node
 NodeType<Type> *current; //pointer to traverse the list
 if(otherList.first == NULL) //otherList is empty
 {
 first = NULL;
 last = NULL;
 }
 else
 {
 current = otherList.first; //current points to the
 //list to be copied

 //copy the first node
 first = new NodeType<Type>; //create the node
 first->info = current->info; //copy the info
 first->link = NULL; //set the link field of
 //the node to NULL
 last = first; //make last point to the
 //first node
 current = current->link; //make current point to the
 //next node

 //copy the remaining list
 while(current != NULL)
 {
 newNode = new NodeType<Type>; //create a node
 newNode->info = current->info; //copy the info
 newNode->link = NULL; //set the link of
 //newNode to NULL
 last->link = newNode; //attach newNode after last
 last = newNode; //make last point to
 //the actual last node
 current = current->link; //make current point to
 //the next node
 }
 //end while
 }
 //end else
}
//end copy constructor
```

### 重载赋值运算符

为 LinkedListType 类重载赋值运算符的函数定义和复制构造函数的定义非常相似。出于完整性考虑,下面给出它的定义。

```
template<class Type>
const LinkedListType<Type>& LinkedListType<Type>::operator=
 (const LinkedListType<Type>& otherList)
```

```

{
 nodeType<Type> *newNode; //pointer to create a node
 nodeType<Type> *current; //pointer to traverse the list

 if(this != &otherList) //avoid self-copy
 {
 if(first != NULL) //if the list is not empty, destroy the list
 destroyList();

 if(otherList.first == NULL) //otherList is empty
 {
 first = NULL;
 last = NULL;
 }
 else
 {
 current = otherList.first; //current points to the
 //list to be copied

 //copy the first element
 first = new nodeType<Type>; //create the node
 first->info = current->info; //copy the info
 first->link = NULL; //set the link field of
 //the node to NULL
 last = first; //make last point to the first node
 current = current->link; //make current point to the next
 //node of the list being copied

 //copy the remaining list
 while(current != NULL)
 {
 newNode = new nodeType<Type>;
 newNode->info = current->info;
 newNode->link = NULL;
 last->link = newNode;
 last = newNode;
 current = current->link;
 } //end while
 } //end else
 } //end else

 return *this;
}

```

### 17.3 有序链表

前面讲述了C++中如何使用指针变量来动态分配和释放内存,以及如何建立和处理链表。本节讨论如何建立有序链表并描述对这些链表的各种操作。下面是作用于(有序)链表的通用操作:

1. 初始化链表。
2. 检查链表是否为空。
3. 检查链表是否为满。
4. 打印链表。
5. 逆序打印链表。
6. 销毁链表。
7. 在链表中查找指定项。

8. 在链表中插入一项。
9. 从链表中删除一项。
10. 获取链表的长度。
11. 复制链表。

作用于有序链表的许多操作和作用于上一节讨论的通用链表操作类似。因为链表是有序的,因此只需要修改实现查找、插入和删除操作的算法。因此,定义有序链表为 ADT 的类定义由 `linkedListType` 类派生而来。

```
template<class Type>
class orderedLinkedListType: public linkedListType<Type>
{
public:
 void search(const Type& item);
 //Outputs "Item is found in the list" if searchItem is in
 //the list; otherwise, outputs "Item is not in the list"
 void insertNode(const Type& newItem);
 //newItem is inserted in the list
 //Post: first points to the new list and
 // newItem is inserted at the proper place in the list
 void deleteNode(const Type& deleteItem);
 //If found, the node containing deleteItem is deleted
 //from the list
 //Post: first points to the first node of the
 // new list
 void printListReverse() const;
 //This function prints the list in reverse order
 //Because the original list is in ascending order, the
 //elements will be printed in descending order
private:
 void reversePrint(nodeType<Type> *current) const;
 //This function is called by the public member
 //function to print the list in reverse order
};
```

这里指针 `last` 不起作用,程序将忽略该指针。

### 搜索链表

首先讨论查找操作。实现查找操作的算法和前面讨论的通用链表查找算法是类似的。因为链表是有序的,所以可以稍加改进查找算法。和前面一样,函数将从链表的第1个节点开始查找。只要在链表中找到 `info` 值大于或等于查找项的节点就停止查找,或者算法查找整个链表。

下列步骤描述了该算法:

1. 比较查找项和链表中的当前节点。如果当前节点的 `info` 值大于或等于查找项,停止查找;否则,使下一个节点成为当前节点。
2. 重复步骤1直到在链表中找到大于或等于查找项的一项,或者链表中再也没有数据可供和查找项比较。

注意,循环并不显式地检查查找项是否等于链表中的一项。因此,在循环执行后,必须检查查找项是否等于链表中的项。

```
template<class Type>
void orderedLinkedListType<Type>::search(const Type& item)
```

```

{
 bool found;
 nodeType<Type> *current; //pointer to traverse the list

 found = false; //initialize found to false
 current = first; //start the search at the first node

 if(first == NULL)
 cout<<"Cannot search an empty list."<<endl;
 else
 {
 while(current != NULL && !found)
 if(current->info >= item)
 found = true;
 else
 current = current->link;
 if(current == NULL) //item is not in the list
 cout<<"Item is not in the list"<<endl;
 else
 if(current->info == item) //test for equality
 cout<<"Item is found in the list"<<endl;
 else
 cout<<"Item is not in the list"<<endl;
 } //end else
 } //end search
}

```

### 插入节点

为了在有序链表中插入一项，必须找到新项所应放置的位置，然后在链表中插入该项。为了在链表中找到新项的位置，和前面一样，需要搜索链表。这里使用两个指针 `current` 和 `trailCurrent` 来搜索链表。指针 `current` 指向 `info` 值正在和插入项进行比较的节点，`trailCurrent` 指向 `current` 前的节点。因为链表是有序的，查找算法和前述一样存在下列情况：

1. 链表最初为空。包含新项的节点是链表中惟一的节点，因此也是第1个节点。
2. 新项小于链表中的最小项，新项放在链表的开始处。在这种情况下，需要调整链表的头指针 `first`。
3. 在链表中的某处插入新项。
  - 3a. 新项大于链表中所有的项。在这种情况下，新项插入到链表的末端。因此，`current` 的值为 `NULL`，新项插入到 `trailCurrent` 之后。
  - 3b. 新项插入到链表中间的某处。在这种情况下，新项插入在 `trailCurrent` 和 `current` 之间。

下面的语句可以完成步骤 3a 和 3b。假定 `newNode` 指向新的节点。

```

trailCurrent->link = newNode;
newNode->link = current;

```

下面阐述这些情况。

**情况 1：链表为空。**

考虑图 17.34 所示的链表。

假设要在链表中插入 27。为了完成这一任务，需要创建一个节点并复制 27 到该节点，设置该节点的链接为 `NULL`，然后使 `first` 和 `last` 指向该节点。图 17.35 显示了结果链表。



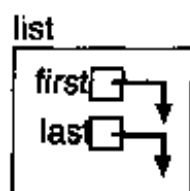


图 17.34 空链表

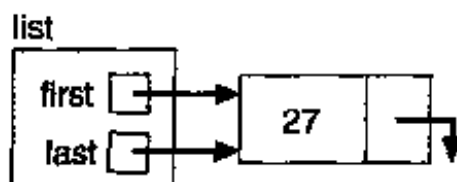


图 17.35 插入 27 后的链表

注意，在插入 27 后，first 和 last 的值都改变了。

情况 2：链表不为空，插入项小于链表中的最小项。考虑图 17.36 所示的链表。

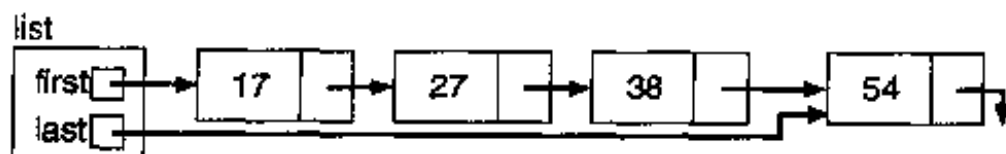


图 17.36 插入 10 之前的非空链表

假设要插入 10。在插入 10 到链表中之后，info 值为 10 的节点成为链表的第一个节点，这要求改变 first 的值。图 17.37 显示了结果链表。

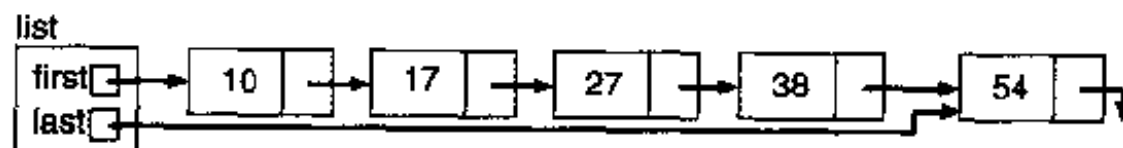


图 17.37 插入 10 后的链表

情况 3：链表不为空，插入项大于链表的第 1 项。如前所述，该情况有两种情形。

情况 3a：插入项大于链表中的最大项，所以它应放在链表的末端。考虑图 17.38 所示的链表。

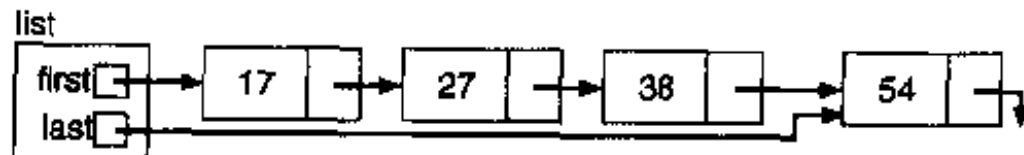


图 17.38 插入 65 之前的链表

假设要在链表中插入 65。在插入 65 之后，结果链表如图 17.39 所示。（注意插入 65 要求改变 last 的值。）

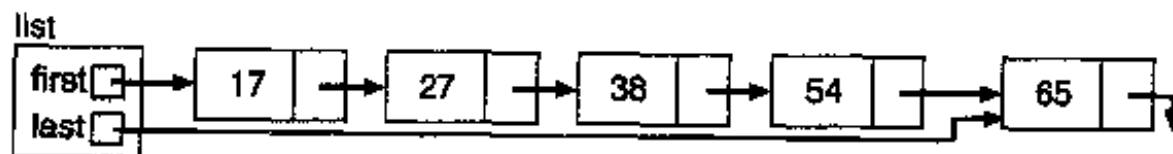


图 17.39 插入 65 后的链表

情况 3b：插入项放在链表中间的某处。考虑图 17.40 所示的链表。

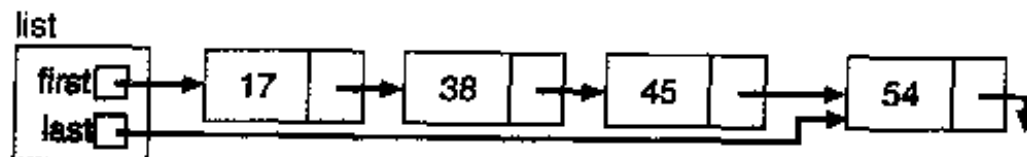


图 17.40 插入 27 之前的链表

假设要在链表中插入 27。很明显，27 应放在 17 和 38 之间，这要求改变 info 值为 17 的节点的链接。在插入 27 后，结果链表如图 17.41 所示。

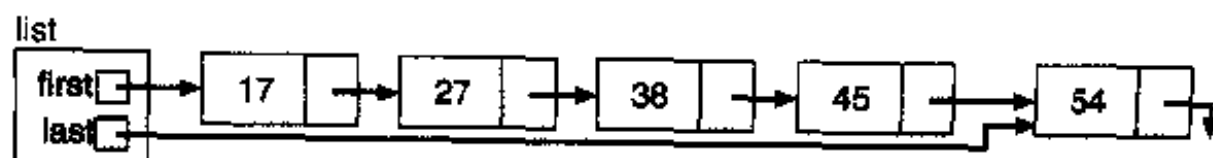


图 17.41 插入 27 后的链表

从情况3可以知道,必须首先遍历链表以找到插入新项的位置。此外,必须用两个指针 `current` 和 `trailCurrent` 来遍历链表。指针 `current` 用来遍历链表及比较链表节点的 `info` 值和插入项。指针 `trailCurrent` 指向 `current` 前的节点。例如,在情况3b中,当查找停止时, `trailCurrent` 指向节点17而 `current` 指向节点38,新项插入到 `trailCurrent` 之后。在情况3a中,当搜索链表并找到放置65的位置之后, `trailCurrent` 指向节点54而 `current` 为 `NULL`。

函数 `insertNode()` 如下所示:

```
template<class Type>
void orderedLinkedListType<Type>::insertNode(const Type& newItem)
{
 nodeType<Type> *current; //pointer to traverse the list
 nodeType<Type> *trailCurrent; //pointer just before current
 nodeType<Type> *newNode; //pointer to create a node
 bool found;

 newNode = new nodeType<Type>; //create the node
 newNode->info = newItem; //store newItem in the node
 newNode->link = NULL; //set the link field of the node
 //to NULL

 if(first == NULL) //Case 1
 first = newNode;
 else
 {
 current = first;
 found = false;

 while(current != NULL && !found) //search the list
 if(current->info >= newItem)
 found = true;
 else
 {
 trailCurrent = current;
 current = current->link;
 }

 if(current == first) //Case 2
 {
 newNode->link = first;
 first = newNode;
 }
 else //Case 3
 {
 trailCurrent->link = newNode;
 newNode->link = current;
 }
 } //end else
} //end insertNode
```

### 删除节点

为了从一个有序链表中删除一个指定项,首先需要搜索链表,查找所要删除的项是否在链表中。实现该操作的函数和通用链表的删除操作函数相同。这里因为链表是有序的,可以稍加改进算法。

和 `insertNode()` 的情况一样,函数使用两个指针 `current` 和 `trailCurrent` 来搜索链表。和 `insertNode()` 操作类似,有下列几种情况:



1. 链表最初为空, 这将产生错误。不能从空链表中删除。
2. 删除项包含在链表的第 1 个节点中。必须调整链表的头指针 first。
3. 删除项在链表中的某处。在这种情况下, current 指向包含删除项的节点, trailCurrent 指向 current 前的节点。
4. 链表不为空, 但删除项不在链表中。

函数 deleteNode() 的定义为:

```
template<class Type>
void orderedLinkedListType<Type>::deleteNode(const Type& deleteItem)
{
 nodeType<Type> *current; //pointer to traverse the list
 nodeType<Type> *trailCurrent; //pointer just before current
 bool found;

 if(first == NULL) //Case 1
 cout<<"Cannot delete from an empty list."<<endl;
 else
 {
 current = first;
 found = false;

 while(current != NULL && !found) //search the list
 {
 if(current->info == deleteItem)
 found = true;
 else
 {
 trailCurrent = current;
 current = current->link;
 }
 }

 if(current == NULL) //Case 4
 cout<<"Item to be deleted is not in the list."<<endl;
 else
 {
 if(current->info == deleteItem) //item to be deleted is
 //in the list
 {
 if(first == current) //Case 2
 {
 first = first->link;
 delete current;
 }
 else //Case 3
 {
 trailCurrent->link = current->link;
 delete current;
 }
 }
 else //Case 4
 cout<<"Item to be deleted is not in the list."<<endl;
 }
 }
} //end deleteNode
```

### 逆序打印链表 (递归)

前述有序链表的节点以升序排列。然而, 某些应用程序可能要求数据以降序打印, 这意味着必

须逆序打印链表。现在讨论函数 `reversePrint()`。给定一个指向链表的指针, 该函数以逆序打印链表的元素。

考虑图 17.42 所示的链表。

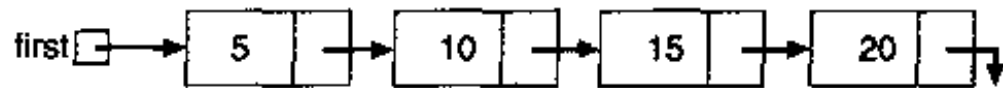


图 17.42 链表

对于图 17.42 中的链表, 输出格式应该为:

20 15 10 5

因为链接是单向的, 所以不能从最后一个节点向后遍历链表。下面讨论如何有效地使用递归来以相反的次序打印链表。

对于递归方式, 不能打印第 1 个节点的 `info` 值, 除非已经打印完剩余的链表 (就是第 1 个节点的尾部)。类似地, 不能打印第 2 个节点的 `info` 值, 除非已经打印完第 2 个节点的尾部, 等等。当每次考虑节点的尾部时, 链表的大小都减小 1。最后链表的大小减为零, 这时递归停止。下面先写出伪代码形式的算法。(假设 `current` 为指向链表的指针。)

```

if(current != NULL)
{
 reversePrint(current->link); //print the tail
 cout<<current->info<<endl; //print the node
}

```

这里没有基类, 它被隐藏了。只有在指向链表的指针不为 `NULL` 的情况下才打印链表。此外, 在 `if` 语句内, 递归调用作用于链表的尾部。因为最后链表的尾部将为空, 所以下一个调用的 `if` 语句将失败, 递归将停止。此外, 注意出现在递归调用后的语句 (如打印节点的 `info` 值)。这样, 当返回调用函数时, 必须执行剩余的语句。回想一下, 函数只有在最后一条语句执行后才退出。(这里“最后一条语句”不是指形式上的最后一条语句, 而是指逻辑上的最后一条语句。)

下面写出上述函数的 C++ 定义, 然后应用到链表上。

```

template<class Type>
void orderedLinkedListType<Type>::reversePrint
 (nodeType<Type> *current) const
{
 if(current != NULL)
 {
 reversePrint(current->link); //print the tail
 cout<<current->info<<" "; //print the node
 }
}

```

考虑语句:

```
reversePrint(first);
```

这里 `first` 是指向 `nodeType<Type>` 类型的指针。

下面跟踪该语句的执行, 它是对于图 17.42 所示链表的函数调用。因为形参为值参数, 实参的值传递给形参。参见图 17.43。

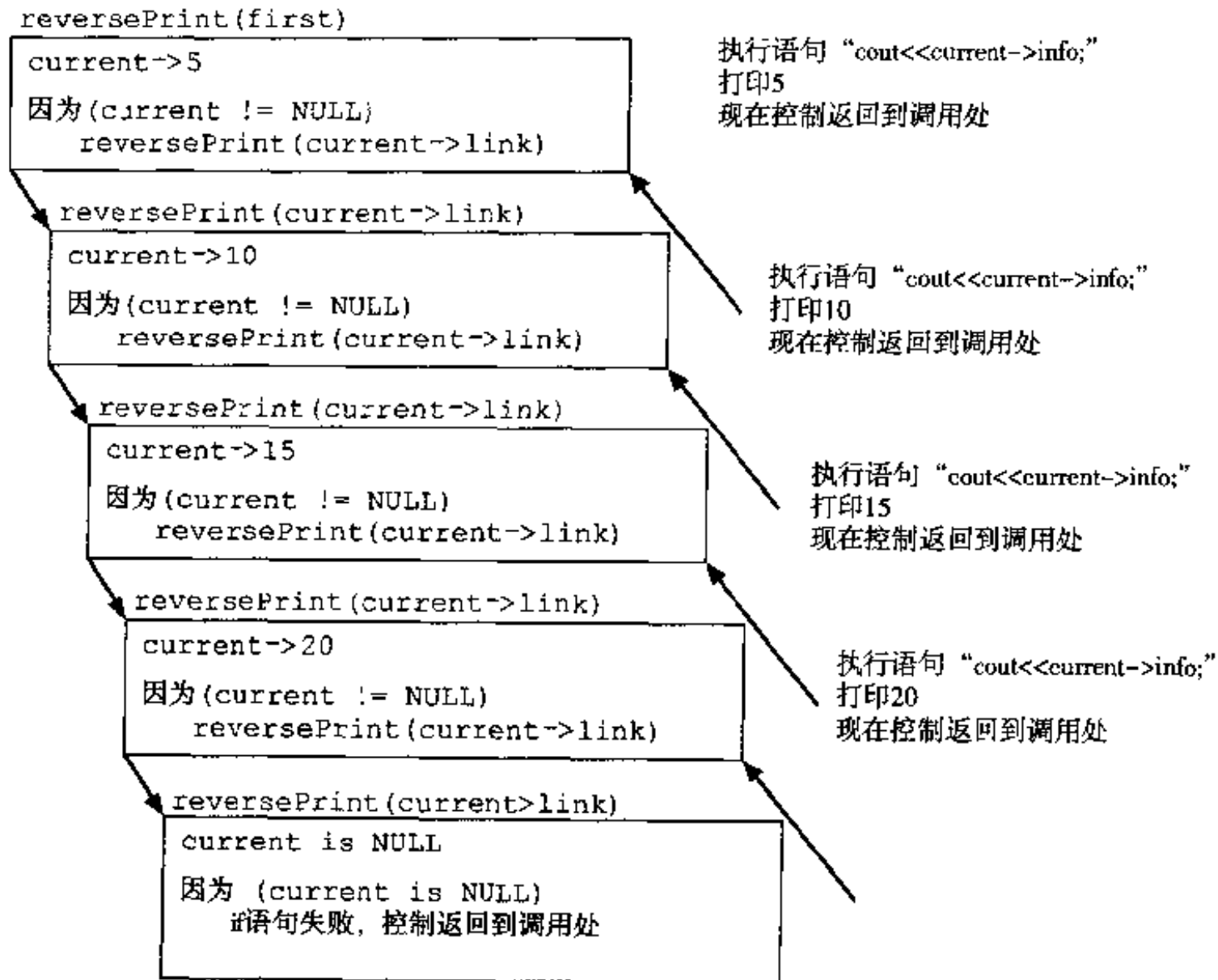


图 17.43 语句 "reversePrint(first);" 的执行情况

**printListReverse()** 既然已经编写出了函数 `reversePrint()`, 下面给出函数 `printListReverse()` 的定义:

```

template<class Type>
void orderedLinkedListType<Type>::printListReverse() const
{
 reversePrint(first);
 cout<<endl;
}

```

### 有序链表的头文件

出于完整性考虑, 下面将说明如何创建定义 `orderedListType` 类以及对这种链表操作的头文件。(假定 `linkedListType` 类的定义和实现操作的函数定义在头文件 `linkedList.h` 中。)

```

//Ordered linked list derived from general linked list
//Header File: orderedList.h

#ifndef H_orderedListType
#define H_orderedListType

#include <iostream>
#include "linkedList.h"

using namespace std;

template<class Type>
class orderedLinkedListType: public linkedListType<Type>
{
public:
 void search(Type item);

```

```

void insertNode(Type newItem);
void deleteNode(Type deleteitem);
void printListReverse() const;

private:
 void reversePrint(nodeType<Type> *current) const;
};

//Definitions of the functions search, insertNode,
//deleteNode, printListReverse, and reversePrint go here
.
.
.
#endif

```

下面的程序测试对有序链表的各种不同操作:

```

//Program to test the various operations on an ordered linked list
#include <iostream>
#include "orderedList.h"
using namespace std;
int main()
{
 orderedLinkedListType<int> list1, list2; //Line 1
 int num; //Line 2

 cout<<"Line 3: Enter integers ending with -999"
 <<endl; //Line 3
 cin>>num; //Line 4
 while(num != -999) //Line 5
 {
 list1.insertNode(num); //Line 6
 cin>>num; //Line 7
 }

 cout<<endl; //Line 8
 cout<<"Line 9: List 1: "; //Line 9
 list1.print(); //Line 10
 cout<<endl; //Line 11

 cout<<"Line 12: List 1 in the reverse order: "
 <<endl; //Line 12
 list1.printListReverse(); //Line 13
 cout<<endl; //Line 14

 list2 = list1; //test the assignment operator Line 15

 cout<<"Line 16: List 2: "; //Line 16
 list2.print(); //Line 17
 cout<<endl; //Line 18

 cout<<"Line 19: Enter the number to be "
 <<"deleted: "; //Line 19
 cin>>num; //Line 20
 cout<<endl; //Line 21

 list2.deleteNode(num); //Line 22

 cout<<"Line 23: After deleting the node, "
 <<"List 2: " <<endl; //Line 23
 list2.print(); //Line 24
}

```

```

 cout<<endl; //Line 25
 return 0;
 }

```

**运行实例：**在本运行实例中，用户输入用阴影表示。

```

Line 3: Enter integers ending with -999
23 65 34 72 12 82 36 55 29 -999

Line 9: List 1: 12 23 29 34 36 55 65 72 82
Line 12: List 1 in the reverse order:
82 72 65 55 36 34 29 23 12

Line 16: List 2: 12 23 29 34 36 55 65 72 82
Line 19: Enter the number to be deleted: 36

Line 23: After deleting the node, List 2:
12 23 29 34 55 65 72 82

```

前面的输出是自解释的，具体细节留做读者练习。

## 17.4 双向链表

双向链表的每一个节点都有一个向前指针和一个向后指针。每一个节点都包含下一个节点的地址（最后一个节点除外），每一个节点也都包含前一个节点的地址（第1个节点除外）。（参见图 17.44。）

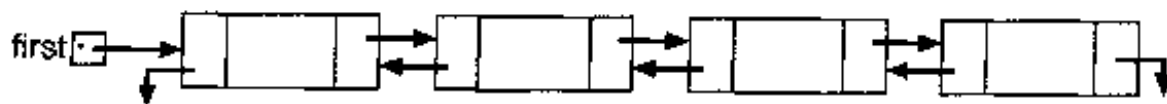


图 17.44 双向链表

双向链表可以从任何一个方向进行遍历。即可以从第1个节点开始遍历链表，如果已给出指向最后一个节点的指针，当然也可以从最后一个节点开始遍历链表。

双向链表的基本操作为：

1. 初始化链表。
2. 销毁链表。
3. 检查链表是否为空。
4. 检查链表是否为满。
5. 在链表中查找指定项。
6. 在链表中插入一项。
7. 从链表中删除一项。
8. 获取链表的长度。
9. 打印链表。

下面将描述对有序双向链表的操作。下面的类定义了一个作为 ADT 的双向链表：

```

//Definition of the node
template <class Type>
struct nodeType
{
 Type info;
 nodeType<Type> *next;

```

```

 nodeType<Type> *back;
};

template <class Type>
class doublyLinkedList
{
public:
 const doublyLinkedList<Type>& operator=
 (const doublyLinkedList<Type> &);
 //overload the assignment operator
 void initializeList();
 //Initialize the list to an empty state
 //Post: first = NULL
 bool isEmptyList();
 //This function returns true if the list is empty;
 //otherwise, it returns false
 void destroy();
 //Delete all nodes from the list
 //Post: first = NULL
 void print();
 //Output the info contained in each node
 int length();
 //This function returns the number of nodes in the list
 void search(const Type& searchItem);
 //Outputs "Item is found in the list" if searchItem
 //is in the list; otherwise, outputs "Item not in the list"
 void insertNode(const Type& insertItem);
 //newItem is inserted in the list
 //Post: first points to the new list and the
 // newItem is inserted at the proper place in the list
 void deleteNode(const Type& deleteItem);
 //If found, the node containing the deleteItem is deleted
 //from the list
 //Post: first points to the first node of the
 // new list
 doublyLinkedList();
 //default constructor
 //Initialize the list to an empty state
 //Post: first = NULL
 doublyLinkedList(const doublyLinkedList<Type>& otherList);
 //copy constructor
 ~doublyLinkedList();
 //destructor
 //Post: the list object is destroyed
private:
 nodeType<Type> *first; //pointer to the list
};

```

实现双向链表操作的函数和前面讨论的类似。这里，因为每个节点有两个指针back和next，某些操作需要调整每个节点中的这两个指针。对于插入和删除操作，因为可以从任何一个方向遍历链表，所以只使用一个指针来遍历链表并称该指针为current指针。同时可以使用current所指节点的next和back指针设置trailCurrent的值。下面将给出每个函数的定义，除了两个例外。复制构造函数和重载赋值运算符的函数留做读者练习。

### 默认构造函数

默认构造函数初始化双向链表为空状态。它设置first为NULL。



```

template<class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
 first= NULL;
}

```

### isEmptyList()

如果链表为空，该操作返回 true；否则返回 false。如果指针 first 为 NULL，链表为空。

```

template<class Type>
bool doublyLinkedList<Type>::isEmptyList()
{
 return(first == NULL);
}

```

### 销毁链表

该操作删除链表中的所有节点，使链表成为空状态。需要从第 1 个节点开始遍历链表，删除每一个节点。

```

template<class Type>
void doublyLinkedList<Type>::destroy()
{
 nodeType<Type> *temp; //pointer to delete the node
 while(first != NULL)
 {
 temp = first;
 first = first->next;
 delete temp;
 }
}

```

### 初始化链表

该操作初始化双向链表为空状态。这一任务可以使用 destroy() 操作完成。函数 initializeList() 的定义为：

```

template<class Type>
void doublyLinkedList<Type>::initializeList()
{
 destroy();
}

```

### 链表的长度

链表的长度是链表中节点的数量。该操作计算链表中节点的数量，然后返回计数值。

```

template<class Type>
int doublyLinkedList<Type>::length()
{
 int length = 0;
 nodeType<Type> *current; //pointer to traverse the list
 current = first; //set current to point to the first node
 while(current != NULL)
 {
 length++; //increment the length
 current = current->next; //advance current
 }
}

```

```

 return length;
}

```

### 打印链表

该函数输出包含在每个节点中的 info 值。需要从第 1 个节点开始遍历链表。

```

template<class Type>
void doublyLinkedList<Type>::print()
{
 nodeType<Type> *current; //pointer to traverse the list
 current = first; //set current to point to the first node
 while(current != NULL)
 {
 cout<<current->info<<" "; //output info
 current = current->next;
 } //end while
} //end print

```

### 搜索链表

如果 searchItem 在链表中，函数 search() 输出 “Item is found in the list”；否则输出 “Item not in the list”。查找算法和有序链表的查找算法完全相同。

```

template<class Type>
void doublyLinkedList<Type>::search(const Type& searchItem)
{
 bool found;
 nodeType<Type> *current; //pointer to traverse the list
 if(first == NULL)
 cout<<"Cannot search an empty list"<<endl;
 else
 {
 found = false;
 current = first;

 while(current != NULL && !found)
 if(current->info >= searchItem)
 found = true;
 else
 current = current->next;

 if(current == NULL)
 cout<<"Item not in the list"<<endl;
 else
 if(current->info == searchItem) //test for equality
 cout<<"Item is found in the list"<<endl;
 else
 cout<<"Item not in the list"<<endl;
 } //end else
} //end search

```

### 插入节点

因为是在双向链表中插入一项，插入节点需要调整某些节点的两个指针。和前面一样，需要找到应插入新项的位置、创建节点、存储新的项、调整新节点和链表中其他特定节点的链接字段。有 4 种情况：

1. 在一个空链表中插入数据项。



2. 在一个非空链表的开始插入数据项。
3. 在一个非空链表的末端插入数据项。
4. 在一个非空链表的某处插入数据项。

情况 1 和 2 都需要改变指针 first 的值, 情况 3 和 4 类似。下面将说明情况 4。

考虑图 17.45 所示的双向链表。

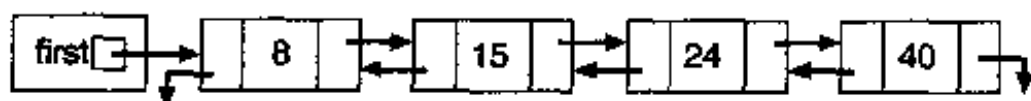


图 17.45 插入 20 之前的双向链表

假设要在链表中插入 20。在插入 20 之后, 结果链表如图 17.46 所示。

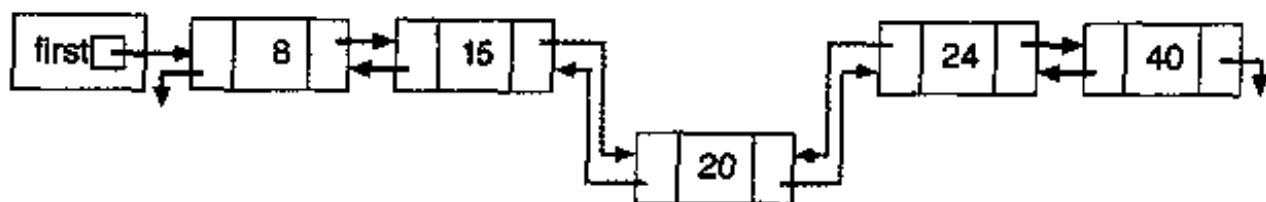


图 17.46 插入 20 后的双向链表

从图 17.46 可知, 节点 15 的 next 指针、节点 24 的 back 指针、节点 20 的 next 和 back 指针需要调整。

函数 insertNode() 的定义为:

```
template<class Type>
void doublyLinkedList<Type>::insertNode(const Type& insertItem)
{
 nodeType<Type> *current; //pointer to traverse the list
 nodeType<Type> *trailCurrent; //pointer just before current
 nodeType<Type> *newNode; //pointer to create a node
 bool found;

 newNode = new nodeType<Type>; //create the node
 newNode->info = insertItem; //store the new item in the node
 newNode->next = NULL;
 newNode->back = NULL;

 if(first == NULL) //if the list is empty, newNode is the only node
 first = newNode;
 else
 {
 found = false;
 current = first;

 while(current != NULL && !found) //search the list
 if(current->info >= insertItem)
 found = true;
 else
 {
 trailCurrent = current;
 current = current->next;
 }

 if(current == first) //insert newNode before the first node
 {
 first->back = newNode;
 newNode->next = first;
 first = newNode;
 }
 else
 {
 trailCurrent->next = newNode;
 newNode->back = trailCurrent;
 newNode->next = current;
 current->back = newNode;
 }
 }
}
```

```

 }
 else
 {
 //insert newNode between trailCurrent and current
 if(current != NULL)
 {
 trailCurrent->next = newNode;
 newNode->back = trailCurrent;
 newNode->next = current;
 current->back = newNode;
 }
 else
 {
 trailCurrent->next = newNode;
 newNode->back = trailCurrent;
 }
 } //end else
} //end else
} //end insertNode

```

### 删除节点

该操作从双向链表中删除指定项（如果存在）。和前面一样，需要首先搜索链表，查找所要删除的项是否在链表中。查找算法和前面的一样。和 insertNode() 操作类似，该操作（如果所要删除的项在链表中）要求调整某些节点中的两个指针。删除操作有下面几种情况：

1. 链表为空。
2. 所要删除的项是链表的第 1 个节点，这要求改变指针 first 的值。
3. 所要删除的项在链表中的某处。
4. 所要删除的项不在链表中。

下面示范情况 3。考虑图 17.47 所示的链表。

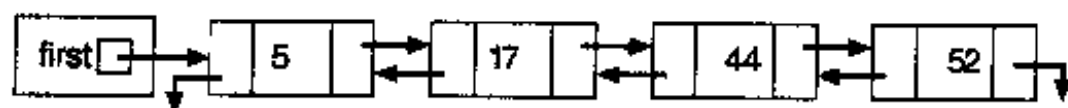


图 17.47 在删除 17 之前的双向链表

假设所要删除的项是 17。首先以两个指针搜索链表，找到 info 值为 17 的节点，然后调整受影响节点的链接字段。（参见图 17.48。）

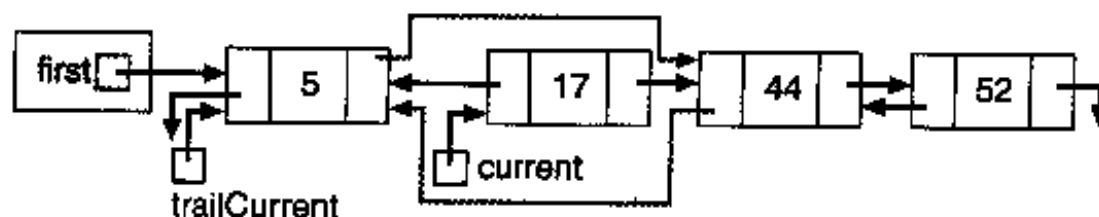


图 17.48 调整节点 17 的前后节点链接

下面删除 current 指向的节点。（参见图 17.49。）

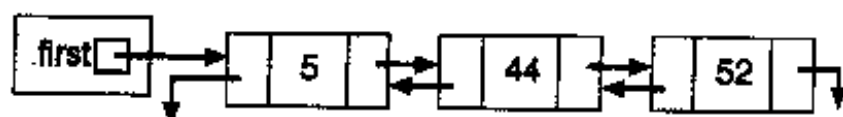


图 17.49 删除 info 值为 17 节点后的链表

函数 deleteNode() 的定义为：

```

template<class Type>
void doublyLinkedList<Type>::deleteNode(const Type& deleteItem)
{
 nodeType<Type> *current; //pointer to traverse the list
 nodeType<Type> *trailCurrent; //pointer just before current
 bool found;
 if(first == NULL)
 cout<<"Cannot delete from an empty list"<<endl;
 else
 if(first->info == deleteItem) //node to be deleted is the
 //first node
 {
 current = first;
 first = first->next;

 if(first != NULL)
 first->back = NULL;

 delete current;
 }
 else
 {
 found = false;
 current = first;

 while(current != NULL && !found) //search the list
 if(current->info == deleteItem)
 found = true;
 else
 current = current->next;

 if(current == NULL)
 cout<<"Item to be deleted is not in the list"<<endl;
 else
 if(current->info == deleteItem) //check for equality
 {
 trailCurrent = current->back;
 trailCurrent->next = current->next;

 if(current->next != NULL)
 current->next->back = trailCurrent;

 delete current;
 }
 else
 cout<<"Item to be deleted is not in the list."<<endl;
 }
 } //end deleteNode
}

```

## 17.5 编程实例：音像商店

对于家庭或个人，一个周末或假日的好去处是到音像商店租电影。社区里有一家新的音像店就要开张。然而，它没有程序记录其影碟和顾客间的出租业务。商店经理需要有人给他们的系统编写一个程序，使得音像店可以正常工作。该程序应该可以执行下列操作：

1. 出租影碟。
2. 返还影碟。
3. 创建音像商店拥有的影碟清单。



```

 newString director, newString productionCo,
 int setInStock);
 //This function sets the details of a video
 //Private data members are set according to the parameters.
 //Post: videoTitle = title; movieStar1 = star1;
 // movieStar2 = star2; movieProducer = producer;
 // movieDirector = director;
 // movieProductionCo = productionCo;
 // copiesInStock = setInStock;
int getNoOfCopiesInStock() const;
 //This function checks the number of copies in stock
 //The value of the data member copiesInStock is returned.
void checkOut();
 //This function rents a video.
 //The number of copies in stock is decremented by one.
void checkIn();
 //This function checks in a video.
 //The number of copies in stock is incremented by one.
void printTitle() const;
 //This function prints the title of a movie
void printInfo() const;
 //This function prints the details of a video
 //The title of the movie, stars, director, and so on are
 //displayed on the screen.
bool checkTitle(newString title);
 //This function checks whether the title is the same as the
 //title of the video
 //Returns the value true if the title is the same as the
 //title of the video, false otherwise.
void updateInStock(int num);
 //This function increments the number of copies in stock by
 //adding the value of the parameter num
 //Post: copiesInStock = copiesInStock + num;
void setCopiesInStock(int num);
 //This function sets the number of copies in stock
 //Post: copiesInStock = num;

newString getTitle();
 //Returns the title of the video

videoType(newString title = "", newString star1 = "",
 newString star2 = "", newString producer = "",
 newString director = "", newString productionCo = "",
 int setInStock = 0);
 //constructor
 //The private data members are set according to the
 //incoming parameters. If no values are specified, the
 //default values are assigned.
 //Post: videoTitle = title; movieStar1 = star1;
 // movieStar2 = star2; movieProducer = producer;
 // movieDirector = director;
 // movieProductionCo = productionCo;
 // copiesInStock = setInStock;
bool operator==(videoType);
bool operator!=(videoType);

private:
 newString videoTitle; //variable to store the name
 //of the movie

```

```

 newString movieStar1; //variable to store the name
 //of the star
 newString movieStar2; //variable to store the name
 //of the star
 newString movieProducer; //variable to store the name
 //of the producer
 newString movieDirector; //variable to store the name
 //of the director
 newString movieProductionCo; //variable to store the name
 //of the production company
 int copiesInStock; //variable to store the number of
 //copies in stock
};

```

为了便于输出, 需要为 videoType 类重载输出流插入运算符<<。注意 videoType 类使用第16章中设计的新String类来操作字符串。

下面将编写 videoType 类中每一个函数的定义。

```

void videoType::setVideoInfo(newString title, newString star1,
 newString star2, newString producer,
 newString director, newString productionCo,
 int setInStock)
{
 videoTitle = title;
 movieStar1 = star1;
 movieStar2 = star2;
 movieProducer = producer;
 movieDirector = director;
 movieProductionCo = productionCo;
 copiesInStock = setInStock;
}

void videoType::checkOut()
{
 if(getNoOfCopiesInStock() > 0)
 copiesInStock--;
 else
 cout<<"Currently out of Stock"<<endl;
}

void videoType::checkIn()
{
 copiesInStock++;
}

int videoType::getNoOfCopiesInStock() const
{
 return copiesInStock;
}

void videoType::printTitle() const
{
 cout<<"Video Title: "<<videoTitle<<endl;
}

void videoType::printInfo() const
{
 cout<<"Video Title: "<<videoTitle<<endl;
 cout<<"Stars: "<<movieStar1<<" and "<<movieStar2<<endl;
}

```



```

 cout<<"Producer: "<<movieProducer<<endl;
 cout<<"Director: "<<movieDirector<<endl;
 cout<<"Production Company: "<<movieProductionCo<<endl;
 cout<<"Copies in stock: "<<copiesInStock<<endl;
 }

 bool videoType::checkTitle(newString title)
 {
 return(videoTitle == title);
 }

 void videoType::updateInStock(int num)
 {
 copiesInStock += num;
 }

 void videoType::setCopiesInStock(int num)
 {
 copiesInStock = num;
 }

 newString videoType::getTitle()
 {
 return videoTitle;
 }

 videoType::videoType(newString title, newString star1,
 newString star2, newString producer,
 newString director,
 newString productionCo, int setInStock)
 {
 videoTitle = title;
 movieStar1 = star1;
 movieStar2 = star2;
 movieProducer = producer;
 movieDirector = director;
 movieProductionCo = productionCo;
 copiesInStock = setInStock;
 }

 videoType::videoType()
 {
 copiesInStock = 0;
 }

 bool videoType::operator==(videoType other)
 {
 return (videoTitle == other.videoTitle);
 }

 bool videoType::operator!=(videoType other)
 {
 return (videoTitle != other.videoTitle);
 }

 ostream& operator<<(ostream& osObject, const videoType &video)
 {
 osObject<<endl;
 osObject<<"Video Title: "<<video.videoTitle<<endl;
 osObject<<"Stars: "<<video.movieStar1<<" and "
 <<video.movieStar2<<endl;
 osObject<<"Producer: "<<video.movieProducer<<endl;
 }

```

```

osObject<<"Director: "<<video.movieDirector<<endl;
osObject<<"Production Company: "<<video.movieProductionCo
 <<endl;
osObject<<"Copies in stock: "<<video.copiesInStock<<endl;
osObject<<"_____<<endl;
return osObject;
}

```

**影碟列表** 该程序要求维护商店所有影碟的列表，而且应该能够在列表中增加新的影碟。一般而言，我们不会知道商店中有多少影碟，从商店中增加或删除一张影碟将改变商店中的影碟数。所以，需要使用一个链表来创建影碟列表。（参见图 17.52。）



图 17.52 影碟列表

在本章的开始，定义了 `linkedListType` 类来创建一个链表对象。同时还定义了相应的基本操作，如在链表中插入和删除一张影碟。然而，一些操作是专门用于影碟列表的，如出租影碟、返还影碟、设置影碟拷贝数，等等。这些操作在 `linkedListType` 类中没有。所以将从 `linkedListType` 类派生出 `videoListType` 类并增加这些操作。（参见图 17.53。）

`videoListType` 类的定义为：

```

class videoListType:public linkedListType<videoType>
{
public:
 bool videoSearch(newString title);
 //This function searches the list to see whether a
 //particular title, specified by the parameter title,
 //is in the store.
 //Returns true if the title is found, false otherwise.
 bool isVideoAvailable(newString title);
 //This function returns true if at least one copy of a
 //particular video is in the store.
 void videoCheckOut(newString title);
 //This function checks out a video, that is,
 //rents a video.
 //Post: copiesInStock is decremented by one.
 void videoCheckIn(newString title);
 //This function checks in a video returned by a customer.
 //Post: copiesInStock is incremented by one.
 bool videoCheckTitle(newString title);
 //This function returns true if a particular video is
 //in the store.
 void videoUpdateInStock(newString title, int num);
 //This function updates the number of copies of a video
 //by adding the value of the parameter num. The
 //parameter title specifies the name of the video for
 //which the number of copies is to be updated.
 //Post: copiesInStock = copiesInStock + num;
 void videoSetCopiesInStock(newString title, int num);
 //This function resets the number of copies of a video.
 //The parameter title specifies the name of the video
 //for which the number of copies is to be reset, and the
 //parameter num specifies the number of copies.
 //Post: copiesInStock = num;
 void videoPrintTitle();
}

```



```

//This function prints the titles of all the videos in
//the store.

private:
 void searchVideoList(newString title, bool& found,
 nodeType<videoType>* &current);
//This function searches the video list for a
//particular video, specified by the parameter title.
//If the video is found, the parameter found is set to
//true; it is set to false otherwise. The parameter
//current points to the node containing the video.
};

```

注意, videoListType 类通过公有继承派生于 linkedListType 类。此外, linkedListType 是一个类模板, 同时 videoType 类作为参数传递给该类。就是说, videoListType 类不是一个模板。因为现在处理的是非常特定的数据类型, videoListType 不再需要是模板。因此, 链表中每个节点的 info 类型为 videoType。通过 videoType 类的成员函数, 某些成员, 如 videoType 类型对象的 videoTitle 和 copiesInStock 现在可以被访问了。

作用于影碟列表的主要操作是返还和出租影碟。这两种操作都需要搜索列表, 查找所要返还或出租的影碟在影碟列表中的位置。其他操作包括检查某张特定影碟是否在商店中、更新某影碟的拷贝数, 等等, 同样要求搜索影碟列表。为了简化查找过程, 将编写一个函数在影碟列表中查找某特定影碟。如果找到影碟, 该函数设置参数 found 为 true, 返回指向该影碟的指针, 使得对该影碟对象的返还、出租和其他操作可以执行。注意, 函数 searchVideoList() 是 videoListType 类的私有成员变量, 因为它只用于内部操作。

首先描述查找过程。

考虑图 17.54 所示的影碟列表节点。

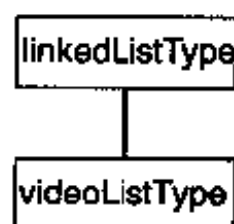


图 17.53 派生于 linkedListType 的 videoListType

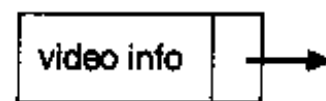


图 17.54 影碟列表节点

组成部分 info 为 videoType 类型, 包含了和影碟有关的必要信息。实际上, 节点的组成部分 info 有 7 个成员: videoTitle、movieStar1、movieStar2、movieProducer、movieDirector、movieProductionCo 和 copiesInStock。(参见 videoType 类的定义。) 所以, 影碟列表的节点有图 17.55 所示的形式。

这些成员变量都是私有的, 不能被直接访问。videoType 类的成员函数将帮助检查和设置某特定成员的值。

假设指针 current 指向影碟列表中的一个节点。(参见图 17.56。)

现在

```
current->info
```

引用节点的 info 部分。假设需要知道节点中存储的影碟名字是否和变量 title 指定的名字相同, 如果存储在节点中的影碟名字和参数 title 指定的名字相同, 表达式:

```
current->info.checkTitle(title)
```

为 true, 否则为 false。(注意, 成员函数 checkTitle() 为值返回函数。videoType 类中有它的声明。)

作为另一个例子, 假设要设置该节点的成员变量 copiesInStock 为 25。因为 copiesInStock 是一个

私有成员变量，它不能被直接访问。所以语句：

```
current->info.copiesInStock = 10; //illegal
```

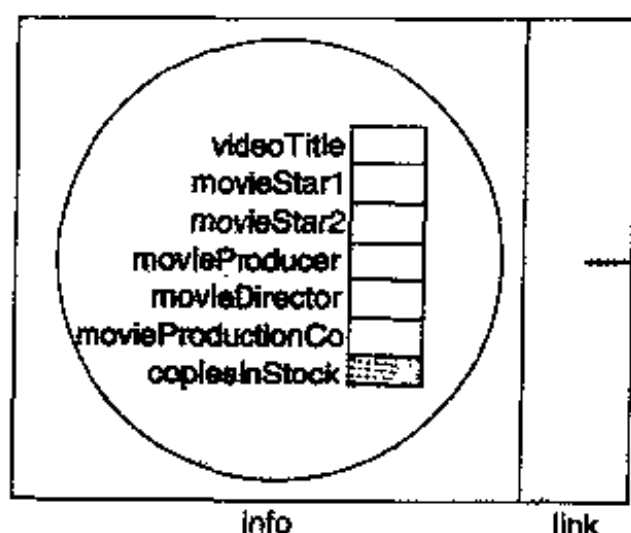


图 17.55 显示了 info 成员的影碟列表节点

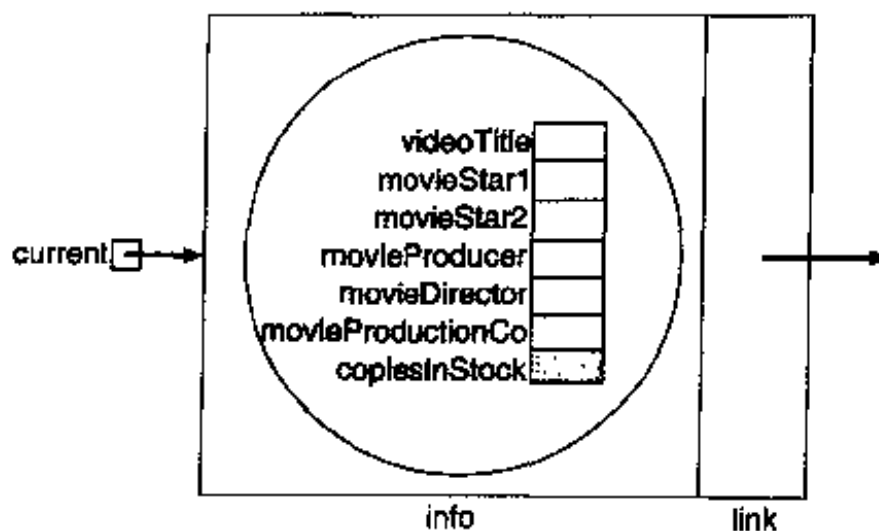


图 17.56 指针 current 和影碟列表节点

是不正确的，将产生编译期错误。必须使用成员函数 setCopiesInStock()，如下所示：

```
current->info.setCopiesInStock(10);
```

我们已经知道如何访问存储在节点中的成员变量，现在描述搜索影碟列表的算法。

```
if video list is empty
 Error
else
 while (not found)
 if the title of the current video is the same as the
 desired title
 stop search
 else
 check the next node
```

下面的函数定义执行所需的查找：

```
void videoListType::searchVideoList(newString title, bool& found,
 nodeType<videoType>* &current)
{
 found = false; //set found to false
 if(first == NULL) //list is empty
 cout<<"Cannot search an empty list. "<<endl;
 else
 {
 current = first; //set current to point to the first node
 //in the list
 found = false; //set found to false
 while(!found && current != NULL) //search the list
 if(current->info.checkTitle(title)) //item is found
 found = true;
 else
 current = current->link; //advance current to
 //the next node
 } //end else
 } //end searchVideoList
```

如果查找成功，参数 found 设置为 true，参数 current 指向包含该影碟 info 的节点。如果不成功，found 设置为 false，current 将为 NULL。

videoListType 类其他函数的定义如下:

```
bool videoListType::isVideoAvailable(newString title)
{
 bool found;
 nodeType<videoType> *location;

 searchVideoList(title, found, location);

 if(found)
 found = (location->info.getNoOfCopiesInStock() > 0);
 else
 found = false;

 return found;
}

void videoListType::videoCheckIn(newString title)
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location); //search the list
 if(found)
 location->info.checkIn();
 else
 cout<<"Video not in stock "<<endl;
}

void videoListType::videoCheckOut(newString title)
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location); //search the list
 if(found)
 location->info.checkOut();
 else
 cout<<"Video not in stock "<<endl;
}

bool videoListType::videoCheckTitle(newString title)
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location); //search the list
 return found;
}

void videoListType::videoUpdateInStock(newString title, int num)
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location); //search the list
 if(found)
 location->info.updateInStock(num);
 else
 cout<<"Video not in stock "<<endl;
}
```

```
void videoListType::videoSetCopiesInStock(newString title, int num)
{
 bool found = false;
 nodeType<videoType> *location;
 searchVideoList(title, found, location);
 if(found)
 location->info.setCopiesInStock(num);
 else
 cout<<"Video not in stock "<<endl;
}

bool videoListType::videoSearch(newString title)
{
 bool found = false;
 nodeType<videoType> *location;
 searchVideoList(title, found, location);
 return found;
}

void videoListType::videoPrintTitle()
{
 nodeType<videoType>* current;
 current = first;
 while(current != NULL)
 {
 current->info.printTitle();
 current = current->link;
 }
}
```

## 第2部分：顾客

**顾客对象** 顾客对象存储和顾客有关的信息，如姓名、账号和顾客所租的影碟列表。因此，音像商店程序有两个组成部分：

1. 包含影碟相关信息的影碟对象，已在前面描述。
2. 包含顾客相关信息的顾客对象，将在下面描述。

下面将创建对应于这两个对象的列表：

- a. 商店所有影碟的列表（如前所述）。
- b. 所有顾客的列表。

下面将描述顾客对象。

顾客的主要特征为：

1. 顾客的名字。
2. 顾客的姓氏。
3. 顾客的账号。
4. 所租影碟列表。

在例 13.19（第 13 章）中已经设计了一个 personType 类，描述了对人名的必要操作。所以，可以从 personType 类派生出 customerType 类，同时增加需要的额外成员（参见图 17.57）。不过，需要

首先重定义 `personType` 类，以利用已学习的面向对象设计的新特征，如运算符重载，然后派生出 `customerType` 类。

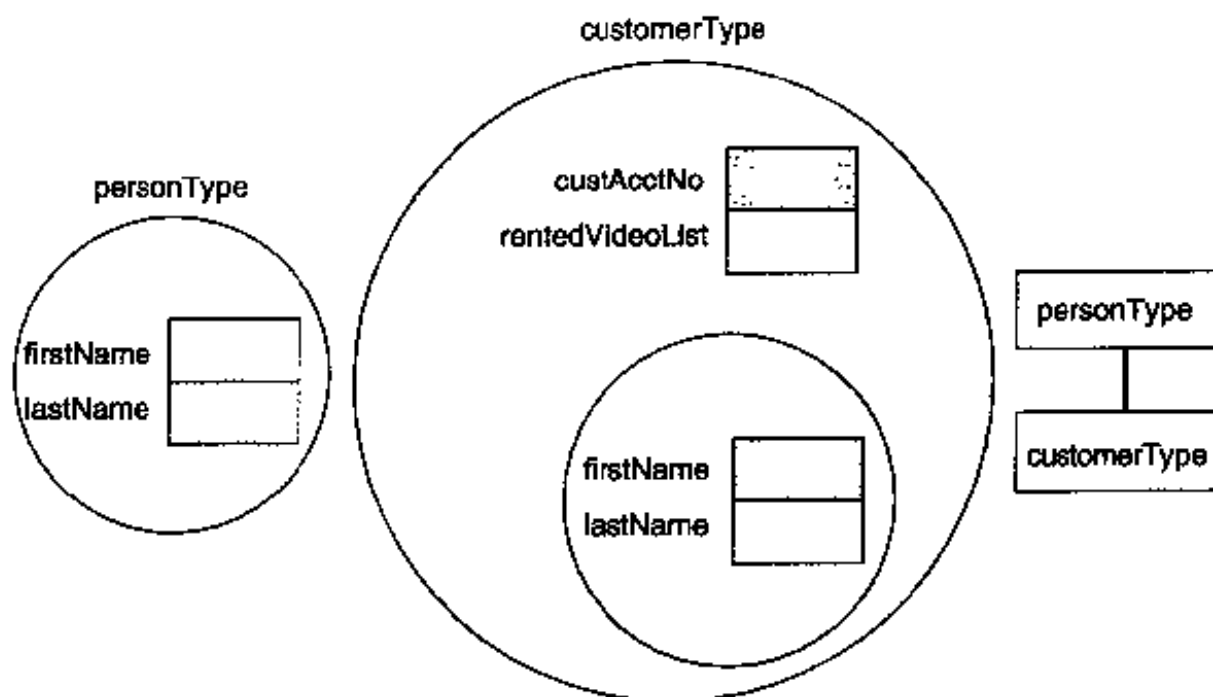


图 17.57 `personType` 类和 `customerType` 类的成员变量

对 `personType` 类对象的基本操作包括：

1. 打印姓名。
2. 设置姓名。
3. 显示名字。
4. 显示姓氏。

类似地，对 `customerType` 类对象的基本操作包括：

1. 打印姓名、账号和所租影碟列表。
2. 设置姓名和账号。
3. 出租一张影碟，即将出租影碟加入到列表中。
4. 返还一张影碟，即从列表中删除所租影碟。
5. 显示账号。

顾客部分的实现细节留做读者练习。（参见本章的编程练习 5。）

### 主程序

现在写出主程序来测试影碟对象。假定影碟必需的数据存储在一个文件中。程序将打开该文件，创建音像商店所拥有影碟的列表。输入文件中的数据格式如下：

```
video title (that is, the name of the movie)
movie star1
movie star2
movie producer
movie director
movie production co.
number of copies
.
.
.
```

下面将编写一个函数 `createVideoList()` 从输入文件中读取数据并创建影碟列表。同时还将编写

一个函数 `displayMenu()` 显示用户可以做出的不同选择, 如返还影碟或出租影碟。`main()` 函数的算法为:

1. 打开输入文件。  
如果输入文件不存在, 退出程序。
2. 创建影碟列表 (`createVideoList()`)。
3. 显示菜单 (`displayMenu()`)。
4. 执行各种不同的操作。

打开输入文件是很容易的。下面描述步骤2和步骤3, 它们通过两个单独的函数完成: `createVideoList()` 和 `displayMenu()`。

#### `createVideoList()`

该函数从输入文件中读取数据并创建影碟链表。因为数据从文件中读取而输入文件是在 `main()` 函数中打开的, 所以需要传递输入文件指针给该函数。同时还需要传递 `main()` 函数中声明的影碟链表指针给该函数。这两个参数都是引用参数。下面, 从每张影碟中读取数据, 然后把影碟插入到链表中。通常的算法为:

- a. 读取数据, 存储到一个影碟对象中。
- b. 插入影碟到链表中。
- c. 对于文件中的影碟数据重复步骤 a 和 b。

#### `displayMenu()`

该函数通知用户所操作的选项。它包含下列输出语句:

- a. Select one of the following
- b. 1: To check whether a particular video is in the store
- c. 2: To check out a video
- d. 3: To check in a video
- e. 4: To check whether a particular video is in the store
- f. 5: To print the titles of all the videos
- g. 6: To print a list of all the videos
- h. 9: To exit

步骤4的伪代码为:

```

a. get choice
b. while (choice != 9)
{
 switch(choice)
 {
 case 1: a. get the movie name
 b. search the video list
 c. if found report success
 else report "failure"
 case 2: a. get the movie name
 b. search the video list
 c. if found check out the video
 else report "failure"
 case 3: a. get the movie name
 b. search the video list
 c. if found check in video
 }
}

```



```

 else report "failure"
 case 4: a. get the movie name
 b. search the video list
 c. if found
 if number of copies > 0
 report "success"
 else
 report "currently out of stock"
 else report "failure"
 case 5: print the titles of the videos
 case 6: print all the videos in the store
 default: bad selection
} //end switch

displayMenu();
get choice;
} //end while

```

### 主程序清单

```

#include <iostream>
#include <fstream>
#include "myString.h"
#include "linkedList.h"
#include "videoType.h"
#include "videoLinkedListType.h"

using namespace std;

void createVideoList(ifstream& infile, videoListType& videoList);
void displayMenu();

int main()
{
 videoListType videoList;
 int choice;
 char ch;
 char title[50];
 ifstream infile;

 //open input file
 infile.open("a:videoDat.txt");
 if(!infile)
 {
 cout<<"Input file does not exist"<<endl;
 return 1;
 }

 //create video list
 createVideoList(infile, videoList);
 infile.close();

 //show menu
 displayMenu();
 cout<<"Enter choice: ";
 cin>>choice; //get request
 cout<<endl;

 //process requests
 while(choice != 9)
 {
 switch(choice)

```

```
{
case 1: cout<<"Enter Title: ";
 cin.get(ch);
 cin.get(title,50);
 cout<<endl;
 if(videoList.videoSearch(title))
 cout<<"Title found"<<endl;
 else
 cout<<"Video not in store"<<endl;

 break;

case 2: cout<<"Enter Title: ";
 cin.get(ch);
 cin.get(title,50);
 cout<<endl;

 if(videoList.videoSearch(title))
 {
 if(videoList.isVideoAvailable(title))
 {
 videoList.videoCheckOut(title);
 cout<<"Enjoy your movie: "<<title<<endl;
 }
 else
 cout<<"Currently "<<title
 <<" is out of stock."<<endl;
 }
 else
 cout<<"Video not in store"<<endl;
 break;

case 3: cout<<"Enter title: ";
 cin.get(ch);
 cin.get(title,50);
 cout<<endl;

 if(videoList.videoSearch(title))
 {
 videoList.videoCheckIn(title);
 cout<<"Thanks for returning "<<title<<endl;
 }
 else
 cout<<"This video is not from our store"<<endl;
 break;

case 4: cout<<"Enter title: ";
 cin.get(ch);
 cin.get(title,50);
 cout<<endl;
 if(videoList.videoSearch(title))
 {
 if(videoList.isVideoAvailable(title))
 cout<<"Currently in stock"<<endl;
 else
 cout<<"Out of stock"<<endl;
 }
 else
 cout<<"Video not in store "<<endl;
 break;
```



```

 case 5: videoList.videoPrintTitle();
 break;

 case 6: videoList.print();
 break;

 default: cout<<"Bad Selection"<<endl;
 } //end switch

 displayMenu(); //display menu

 cout<<"Enter choice: ";
 cin>>choice; //get the next request
 cout<<endl;
} //end while

return 0;
}

void createVideoList(ifstream& infile, videoListType& videoList)
{
 char Title[50];
 char Star1[50];
 char Star2[50];
 char Producer[50];
 char Director[50];
 char ProductionCo[70];
 char ch;
 int InStock;

 videoType newVideo;

 infile.get(Title, 50);
 infile.get(ch);
 while(infile)
 {
 infile.get(Star1, 50);
 infile.get(ch);
 infile.get(Star2, 50);
 infile.get(ch);
 infile.get(Producer, 50);
 infile.get(ch);
 infile.get(Director, 50);
 infile.get(ch);
 infile.get(ProductionCo, 70);
 infile.get(ch);
 infile>>InStock;
 infile.get(ch);
 newVideo.setVideoInfo(Title, Star1, Star2, Producer,
 Director, ProductionCo, InStock);
 videoList.insertFirst(newVideo);

 infile.get(Title, 50);
 infile.get(ch);
 } //end while
} //end createVideoList

void displayMenu()
{
 cout<<"Select one of the following "<<endl;
 cout<<"1: To check whether a particular video is in the store"
 <<endl;
}

```

```
cout<<"2: To check out a video"<<endl;
cout<<"3: To check in a video"<<endl;
cout<<"4: To check whether a particular video is in the store"
 <<endl;
cout<<"5: To print the titles of all the videos"<<endl;
cout<<"6: To print a list of all the videos"<<endl;
cout<<"9: To exit"<<endl;
}
```

## 17.6 小结

1. 链表是一系列称为节点的项，节点的次序由地址确定，称为链接并存储在每个节点中。
2. 指向链表的指针——指向链表第1个节点的指针——存储在单独的位置，称为 head（头节点）或 first（首节点）。
3. 链表是一种动态数据结构。
4. 链表的长度是链表中节点的数量。
5. 向链表中插入或删除项不要求数据移动，但需要调整指针。
6. 单向链表只能从一个方向遍历。
7. 对链表的查找是按次序进行的。
8. 链表的头（或首）指针总是固定的，指向链表的第1个节点。
9. 为了遍历一个链表，程序必须使用与链表头指针不同的指针，并初始化为指向链表的第1个节点。
10. 当按值传递一个类对象时，复制构造函数复制实际对象的值到形参中。
11. 带头和尾节点的链表简化了插入和删除操作。
12. 头和尾节点不是实际链表的一部分。实际链表元素位于头和尾节点之间。
13. 如果在包含头和尾节点的链表中，其惟一节点为头和尾节点，该链表为空。
14. 在双向链表中，节点有两个链接：一个链接指向下一个节点，另一个链接指向前一个节点。
15. 双向链表可以从任何一个方向遍历。
16. 在双向链表中，插入和删除项要求调整节点的两个指针。

## 17.7 练习

1. 判断下列语句为 true 或 false。
    - a. 在链表中，元素的次序由用于存储元素的节点次序决定。
    - b. 在链表中，节点的内存分配是按次序进行的。
    - c. 单向链表可以从任何一个方向遍历。
    - d. 在链表中，节点总是插入在开始处或末端，因为链表不是随意访问的数据结构。
    - e. 在包含头和尾节点的链表中插入（和删除）项比在一个普通链表中简单，因为前一种链表不包含特殊情况。
    - f. 链表的头指针不能用于遍历链表。
- 考虑图 17.58 所示的链表。假定节点为通常的 info-link 形式。使用该链表回答练习 2 ~ 练习 7。如果需要，可以声明额外的变量。（假定 List、p、s、A 和 B 为指向 nodeType 类型的指针。）

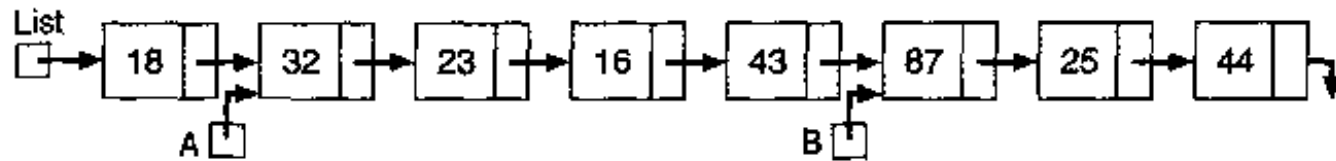


图 17.58 用于练习 2~ 练习 7 的链表

2. 下列 C++ 语句的输出是什么?

- a. `cout<<List->info;`
- b. `cout<<A->info;`
- c. `cout<<B->link->info;`
- d. `cout<<List->link->link->info`

3. 下列关系表达式的值是什么?

- a. `List->info >= 18`
- b. `List->link == A`
- c. `A->link->info == 16`
- d. `B->link == NULL`
- e. `List->info == 18`

4. 判断下列语句的有效性。如果某条语句无效, 说明原因。

- a. `A = B;`
- b. `List->link = A->link;`
- c. `List->link->info = 45;`
- d. `*List = B;`
- e. `*A = *B;`
- f. `B = A->link->info;`
- g. `A->info = B->info;`
- h. `List = B->link->link;`
- i. `B = B->link->link->link;`

5. 写出完成下列任务的 C++ 语句。

- a. 使 A 指向包含 23 的节点。
- b. 使 List 指向包含 16 的节点。
- c. 使 B 指向链表中的最后一个节点。
- d. 使 List 指向一个空链表。
- e. 设置包含 25 的节点值为 35。
- f. 创建 info 值为 10 的新节点, 插入到 A 所指的节点后。
- g. 输出 info 值为 23 的节点。此外, 释放该节点所占据的内存。

6. 下列 C++ 代码的输出是什么?

```

p = List;
while(p != NULL)
 cout<<p->info<<" ";
 p = p->link;
cout<<endl;

```

7. 如果下列 C++ 语句是有效的, 显示输出。如果是无效的, 解释其原因。

- a. `s = A;`  
`p = B;`  
`s->info = B;`  
`p = p->link;`  
`cout<<s->info<<" "<<p->info<<endl;`
- b. `p = A;`  
`p = p->link;`  
`s = p;`  
`p->link = NULL;`

```
s = s->link;
cout<<p->info<<" "<<s->info<<endl;
```

8. 显示下列 C++ 代码的输出。假定代码为通常的 info-link 形式, info 为 int 类型。(List 和 ptr 为指向 nodeType 类型的指针。)

```
a. List = new nodeType;
 List->info = 10;
 ptr = new nodeType;
 ptr->info = 13;
 ptr->link = NULL;
 List->link = ptr;
 ptr = new nodeType;
 ptr->info = 18;
 ptr->link = List->link;
 List->link = ptr;
 cout<<List->info<<" "<<ptr->info<<" ";
 ptr = ptr->link;
 cout<<ptr->info<<endl;

b. List = new nodeType;
 List->info = 20;
 ptr = new nodeType;
 ptr->info = 28;
 ptr->link = NULL;
 List->link = ptr;
 ptr = new nodeType;
 ptr->info = 30;
 ptr->link = List;
 List = ptr;
 ptr = new nodeType;
 ptr->info = 42;
 ptr->link = List->link;
 List->link = ptr;
 ptr = List;
 while(ptr != NULL)
 {
 cout<<ptr->info<<endl;
 ptr = ptr->link;
 }
```

9. 考虑下列 C++ 语句。(linkedListType 类和本章中所定义的不同。)

```
linkedListType<int> list;

list.insertFirst(15);
list.insertLast(28);
list.insertFirst(30);
list.insertFirst(2);
list.insertLast(45);
list.insertFirst(38);
list.insertLast(25);
list.deleteNode(30);
list.insertFirst(18);
list.deleteNode(28);
list.deleteNode(12);
list.print();
```

该程序段的输出是什么?

10. 假设输入为:

18 30 4 32 45 36 78 19 48 75 - 999

下列 C++ 代码的输出是什么? (linkedListType 类和本章中所定义的不同。)

```

linkedListType<int> list;
linkedListType<int> copyList;
int num;

cin>>num;
while(num != -999)
{
 if(num % 5 == 0 || num % 5 == 3)
 list.insertFirst(num);
 else
 list.insertLast(num);
 cin>>num;
}
list.print();
cout<<endl;

copyList = list;

copyList.deleteNode(78);
copyList.deleteNode(35);

cout<<"Copy List = ";
copyList.print();
cout<<endl;

```

## 17.8 编程练习

1. (在线地址簿) 第14章中的编程练习6最多可以处理500项。使用链表重做该编程练习, 从而可以处理任意数量的条目。在程序中增加下列操作:
  - a. 在地址簿中增加或删除一项。
  - b. 当程序终止时, 将地址簿中的数据写到磁盘上。
2. 通过增加下列操作扩展 `linkedListType` 类:
  - a. 查找和删除链表中 `info` 值最小的节点。(只删除第1次出现的最小节点。只遍历链表一次。)
  - b. 查找和删除链表中 `info` 值为给定值的所有节点。(只遍历链表一次。)
3. 作用于链表的其他一些常用操作是 `divideMid()` 和 `divideAt()`。
  - a. `divideMid()`: 该操作把链表分裂为两个(基本)相同大小的子链表。  
 例如, 假设给定的链表为 13 72 89 65 34。那么两个子链表为 13 72 89 和 65 34。类似地, 如果原始链表为 12 67 34 65, 两个子链表为 12 67 和 34 65。
  - i. 增加 `divideMid()` 操作到 `linkedListType` 类中, 如下所示:

```

void divideMid(linkedListType<Type> &sublist);
//This operation divides the given list into two sublists of
// (almost) equal size.
//Post: first points to the first node and last
// points to the last node of the first sublist.
// sublist.first points to the first node and
// sublist.last points to the last node of the
// second sublist.

```

考虑下列语句:

```

linkedListType<int> myList;
linkedListType<int> subList;

```

假设 `myList` 指向元素为 34 65 27 89 12 的链表。语句:

```

myList.divideMid(subList);

```

将myList分为两个子链表: myList指向元素为34 65 27的链表, 而subList指向元素为89 12的链表。

ii. 写出实现divideMid()操作的函数模板定义。

b. divideAt(): 该操作在给定info值的节点处分裂链表。

假设oldList指向有下列元素的链表:

10 18 34 6 28 92 56 48

指定节点的info值为6。那么两个子链表为:

10 18 34 和 6 28 92 56 48

i. 增加下面的操作到linkedListType类中:

```
void divideAt(linkedListType &secondList, Type item);
//Divide the list at the node with the info item into two
//sublists.
//Post: first and last point to the first and
// last nodes of the first sublist.
// secondList.first and secondList.last point to the
// first and last nodes of the second sublist.
```

考虑下列语句:

```
linkedListType<int> myList;
linkedListType<int> otherList;
```

假设myList指向包括下列元素的链表:

34 65 18 39 27 89 12。语句:

```
myList.divideAt(otherList, 18);
```

将myList分裂为两个子链表: myList指向包括元素34 65的链表, otherList指向包括元素18 39 27 89 12的子链表。

ii. 写出实现divideAt()操作的函数模板定义。

4. a. 增加下面的操作到orderedLinkedListType类中:

```
void mergeLists(orderedLinkedListType<Type> &list1,
 orderedLinkedListType<Type> &list2);

//This operation creates a new list by merging the elements
//of list1 and list2.
//Post: first points to the merged list
//list1 and list2 are empty
```

考虑下列语句:

```
orderedLinkedListType<int> newList;
orderedLinkedListType<int> list1;
orderedLinkedListType<int> list2;
```

假设list1指向包括元素2 6 7的链表, list2指向包括元素3 5 8的链表。语句:

```
newList.mergeLists(list1, list2);
```

创建一个新的链表, 包括次序为2 3 5 6 7 8的元素, newList对象指向该链表。此外, 在前面的语句执行后, list1和list2为空。

b. 写出实现合并链表操作的函数模板mergeLists()的定义。

5. (音像商店编程练习)

a. 完成音像商店编程实例中定义的customerType类的设计和实现。

b. 设计和实现customerListType类, 创建和维护一家音像商店的顾客链表。

6. (音像商店编程练习) 完成音像商店程序的设计和实现。



## 第18章 栈和队列

本章要点：

- 什么是栈
- 栈操作
- 使用数组实现栈
- 使用链表实现栈
- 栈的应用
- 使用栈代替递归
- 什么是队列
- 队列操作
- 使用数组实现队列
- 使用链表实现队列
- 队列的应用

在本章中，将讨论两种十分有用的数据结构，栈和队列。它们在计算机科学中都有着十分广泛的应用。

### 18.1 栈

假设有一个包含几个函数的程序，为确定起见，假设分别为 A、B、C 和 D。假设函数 A 调用函数 B，函数 B 调用函数 C，函数 C 调用函数 D。当函数 D 返回时，控制回到函数 C；当函数 C 返回时，控制回到函数 B；当函数 B 返回时，控制回到函数 A。在程序执行期间计算机是如何安排函数调用的？计算机又是如何安排递归函数调用的？在第 17 章中，曾设计了一个逆序打印链表的递归函数。那么如何编写一个非递归的逆序打印链表的程序？

本节将讨论称为栈的数据结构，计算机中就是使用它来实现函数调用。还可以利用栈将递归函数转换为非递归函数。栈在计算机科学中还有着大量的应用。在讨论过如何实现栈之后，将研究一些栈的应用。

栈是由一系列相同类型的元素组成的，并且添加和删除元素的操作只发生在一端，称之为栈顶。例如，在自助餐厅中，一叠托盘中上面第 2 个托盘只有在第 1 个托盘被拿走时才能够被取走。要拿到书架中位于数学和历史书下面的计算机书，则必须首先将数学和历史书拿开，之后，计算机书成为最顶层的元素（即栈顶元素）。图 18.1 显示了栈的一些例子。

栈底部的元素在栈的最下部，而栈顶元素是最后添加到栈中的元素。因为栈只从一侧（即栈顶）进行元素的添加和删除，这意味着最后添加的元素将被最先删除。正因为如此，栈也称为是一种“后进先出”（LIFO）的数据结构。

**栈：**元素只从一端进行添加和删除的数据结构；栈是一种“后进先出”（LIFO）数据结构。

在了解了什么是栈之后，下面来看看在栈之上有什么样的操作。因为新元素可以添加到栈中，故可以进行添加操作，称为压入（push），即将一个元素添加到栈中。同样地，栈顶的元素可以从

栈中移出,称为弹出(pop),即将栈顶元素从栈中删除。图18.2至图18.7说明了压入和弹出操作。

压入和弹出操作以下面的方式进行工作:设想地板上有很多盒子,需要被放置到旁边的桌子上。最开始,所有的盒子都在地板上,堆栈是空的(参见图18.2)。

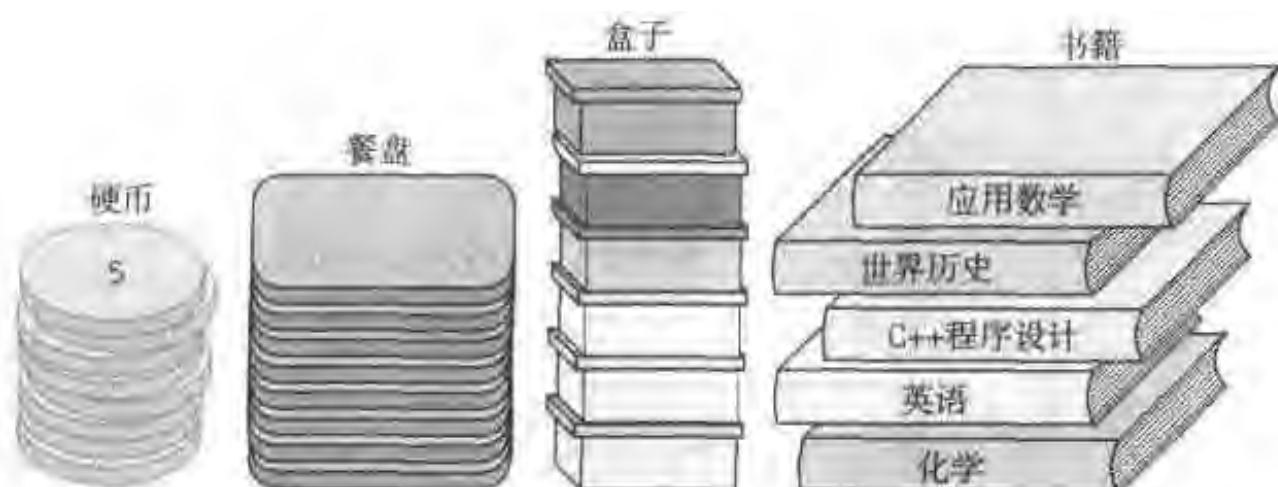


图18.1 各种栈

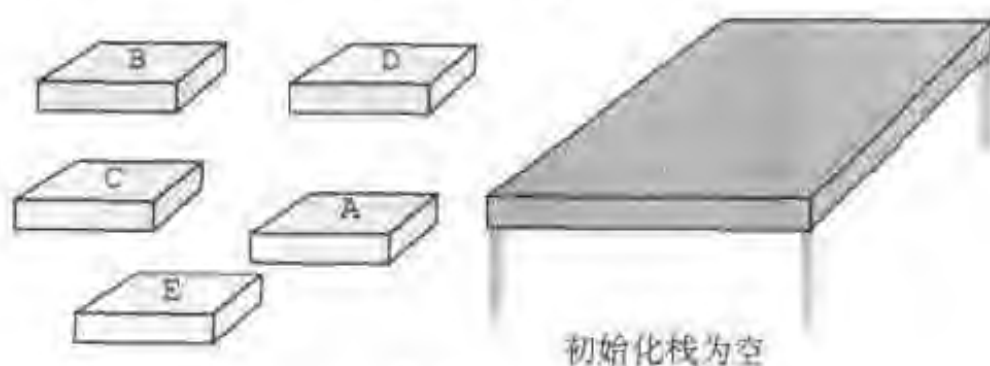


图18.2 空栈

首先将盒子A放到栈上,在压入操作后,栈的情况如图18.3所示。

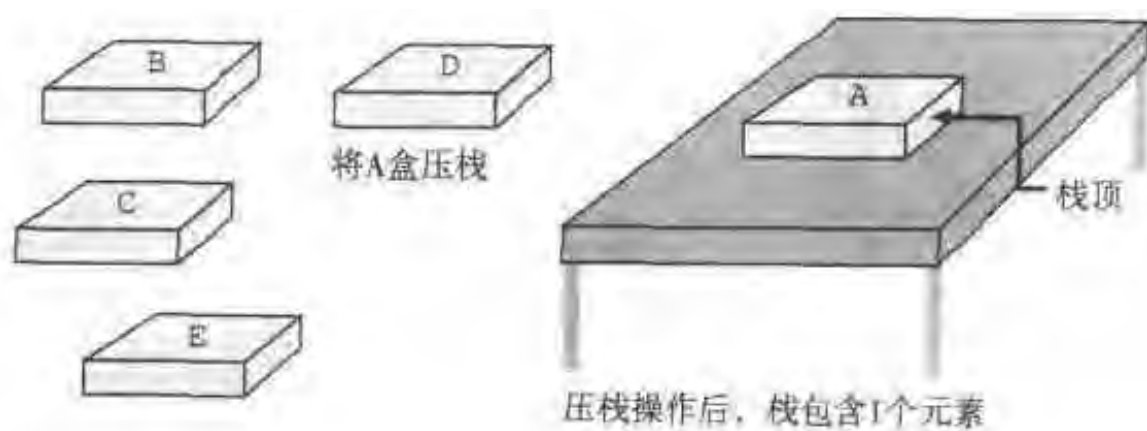


图18.3 压入盒子A后的栈

然后,将盒子B压入栈。在此操作之后,栈的情况如图18.4所示。

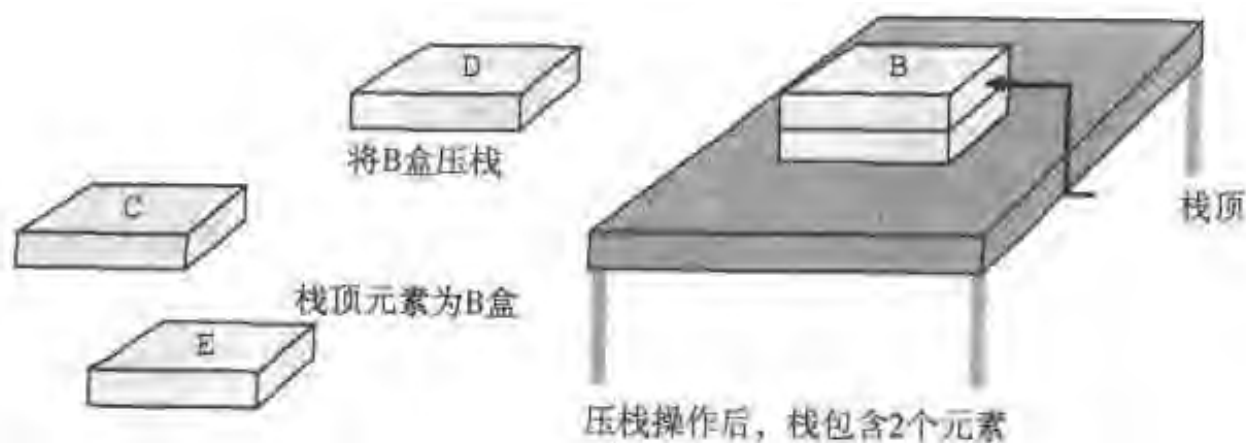


图18.4 压入盒子B后的栈

接下来,将盒子C压入栈中,如图18.5所示。



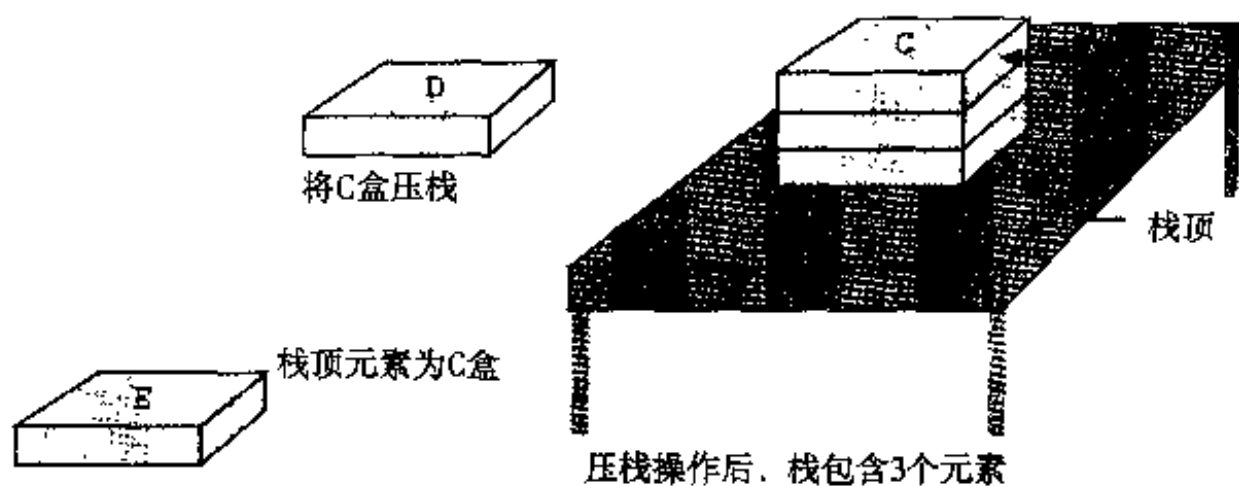


图 18.5 压入盒子 C 后的栈

再将盒子 D 压入，如图 18.6 所示。

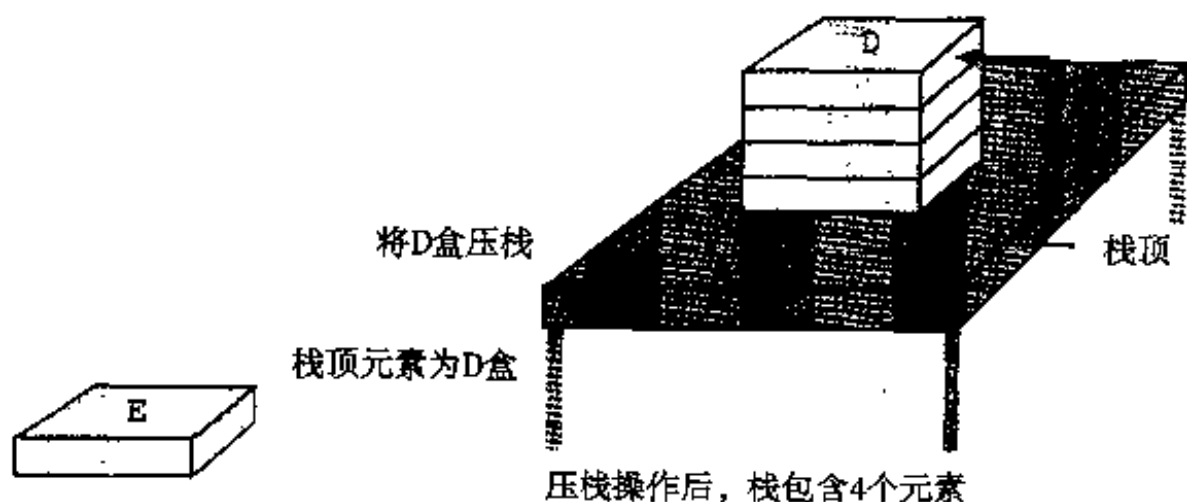


图 18.6 压入盒子 D 后的栈

下面，我们对栈进行弹出操作，如图 18.7 所示。

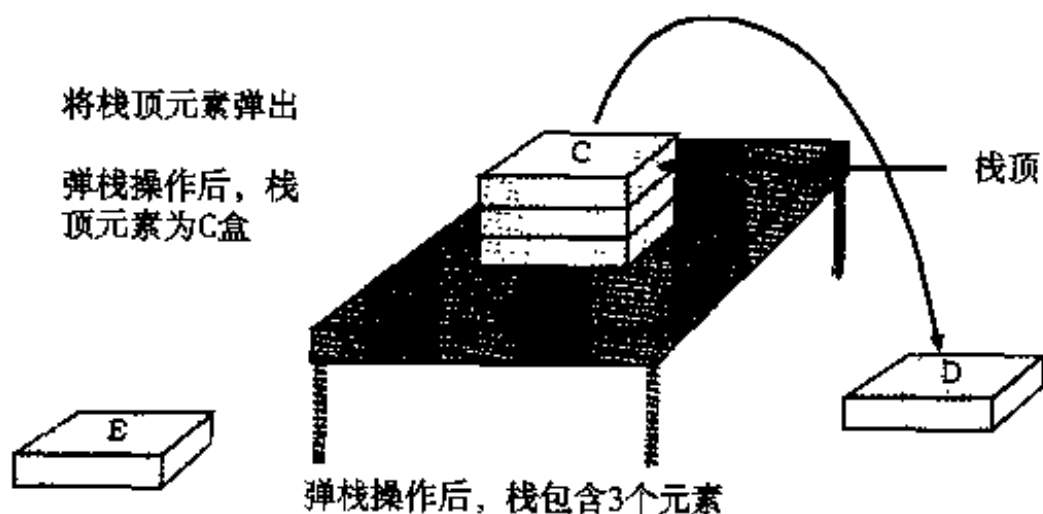


图 18.7 弹出操作后的栈

只要栈非空，就可以对它进行弹出操作，只要栈中有空间，就可以向它进行压入操作。紧跟在压入和弹出之后的是 `isFullStack()` 操作（检查栈是已满）和 `isEmptyStack()` 操作（检查栈是否为空）。当进行添加和删除操作时，栈是不断变化的，在开始使用一个栈时，栈必须是空的。因此，需要另外一个称为 `initializeStack()` 的操作用来将栈初始化为空。另一个有用的操作是 `destroyStack()`。这一操作通常将删除栈中所有的元素，使栈变为空状态。因此，要成功地实现一个栈，至少需要这 6 种操作，我们会在下一节详细描述。有时可能还需要栈上的其他操作，这取决于特定的实现细节。

### 18.1.1 栈操作

栈上的基本操作如下：

1. initializeStack(): 将栈初始化为空状态。
2. destroyStack(): 删除栈中所有的元素, 使之处于空状态。
3. isEmptyStack(): 检查栈是否为空。如果为空, 返回 true, 否则, 返回 false。
4. isFullStack(): 检查栈是否为满状态。如果为满状态, 返回 true, 否则, 返回 false。
5. push(): 向栈顶添加新的元素。它需要一个栈并输入一个新元素。在进行此操作之前, 栈必须存在并且不满。
6. pop(): 将栈顶元素从栈中删除, 并将其存入称为 poppedElement 的变量中。该操作的输入是一个栈和存放弹出元素的位置 (poppedElement)。在此操作之前, 栈必须存在并且不为空。

现在考虑如何实现抽象的栈数据结构, 像 push() 和 pop() 这样的栈操作目前还不可用, 因此必须编写用来实现栈操作的函数。

## 18.2 栈的数组实现

因为栈中所有的元素都是相同类型的, 因此可以使用数组来实现一个栈。栈的第1个元素可以放在数组中第1个元素的位置, 第2个放在数组中第2个元素的位置, 依次类推。栈顶位置就是最后添加到栈元素的下标值。

在栈的这种实现里, 栈元素存储存在数组中, 并且数组是一种可以随机访问的数据结构, 即可以直接访问数组中的任何元素。然而根据定义, 栈是一种只能从一端访问 (压入或弹出), 即“后进先出”型的数据结构。因此, 栈元素只能从栈顶而不能从底部或者中间进行访问。栈的这一特性是非常重要的, 并且必须从一开始就得到贯彻执行。

要记录栈顶在数组中的位置, 可以直接声明另外一个变量称为 top。

下面的类 stackType 将栈以抽象数据类型 (ADT) 的形式加以定义。通过使用指针可以动态分配数组, 所以可以让用户来指定数组的大小 (即栈的大小)。假设栈的默认大小是 100。因为类 stackType 有一个指针数据成员 (指向存储栈数据的数组), 必须重载赋值运算符并且包含复制构造函数和析构函数。下面给出了一个栈的一般定义, 如果有特殊需要, 可以在声明栈对象的时候传递栈元素的类型。

```
template<class Type>
class stackType
{
public:
 const stackType<Type>& operator=(const stackType<Type>&);
 //overload the assignment operator
 void initializeStack();
 //Initialize the stack to an empty state
 //Post: top = 0
 bool isEmptyStack();
 //This function returns true if the stack is empty;
 //otherwise, it returns false.
 bool isFullStack();
 //This function returns true if the stack is full;
 //otherwise, it returns false.
 void destroyStack();
 //Remove all elements from the stack
 //Post: top = 0
 void push(const Type& newItem);
 //Add newItem to the stack
 //Post: stack is changed and the newItem
```

```

 // is added to the top of stack
void pop(Type& poppedItem);
 //Remove the top element of the stack
 //Post: the stack is changed and the top element
 // is removed from the stack. The top element
 // of the stack is saved in poppedItem.
stackType(int stackSize = 100);
 //constructor
 //Create an array of size stackSize to hold the
 //stack elements. The default stack size is 100.
 //Post: the variable list contains the base
 // address of the array, top = 0 and
 // maxStackSize = stackSize
stackType(const stackType<Type>& otherStack);
 //copy constructor
~stackType();
 //destructor
 //Remove all the elements from the stack
 //Post: the array (list) holding the stack
 // elements is deleted

private:
 int maxStackSize; //variable to store the maximum stack size
 int top; //variable to point to the top of the stack
 Type *list; //pointer to the array that holds
 //the stack elements
};

```

注意：由于 C++ 中数组以下标 0 开始，所以需要区分 top 的值和 top 所表示的数组位置。如果 top 为 0，则栈为空；如果 top 非 0，则栈不为空并且栈顶元素的数组下标为 top - 1。

图 18.8 显示了这一数据结构，其中 stack 是一个 stackType 型的对象。请注意，top 可以从 0 变化为 maxStackSize。如果 top 非 0，则 top - 1 是栈顶元素的下标。这里，maxStackSize = 100。

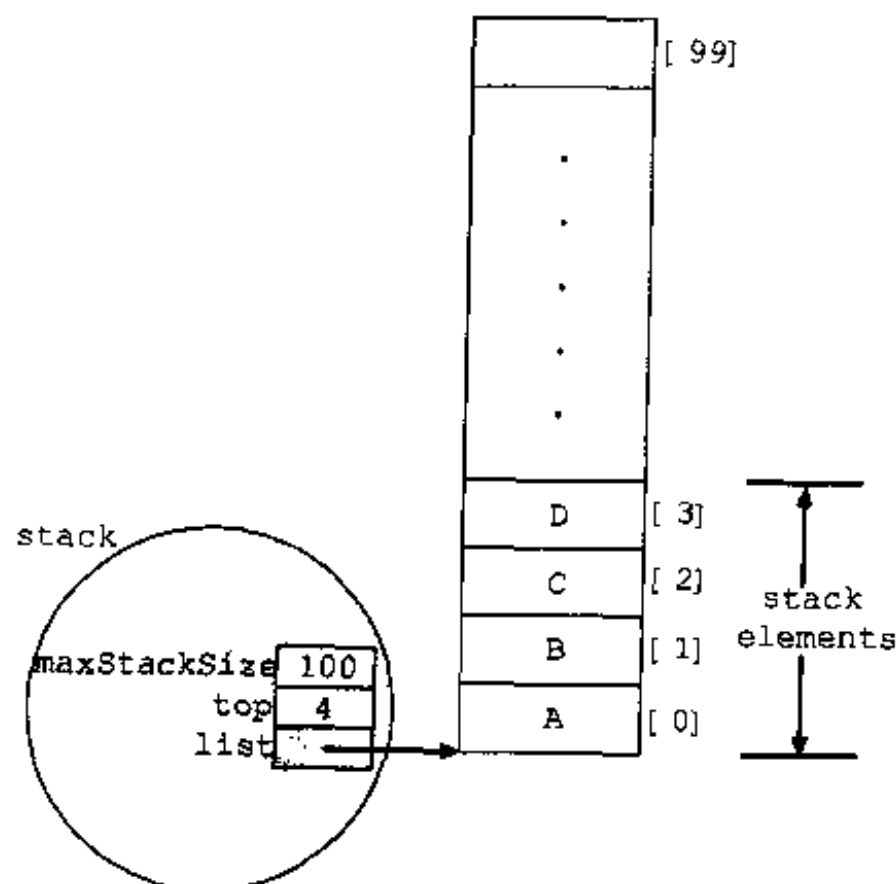


图 18.8 栈的例子

注意，指针 list 包含数组的地址（用来存储栈的元素），即数组第 1 个元素的地址。下面将定义用来实现类 stackType 操作的各种成员函数。

### 初始化栈

考虑以下 `initializeStack()` 操作, 因为 `top` 的值标记了栈是否为空, 所以可以简单地将 `top` 的值设置为 0 来初始化栈 (参见图 18.9)。

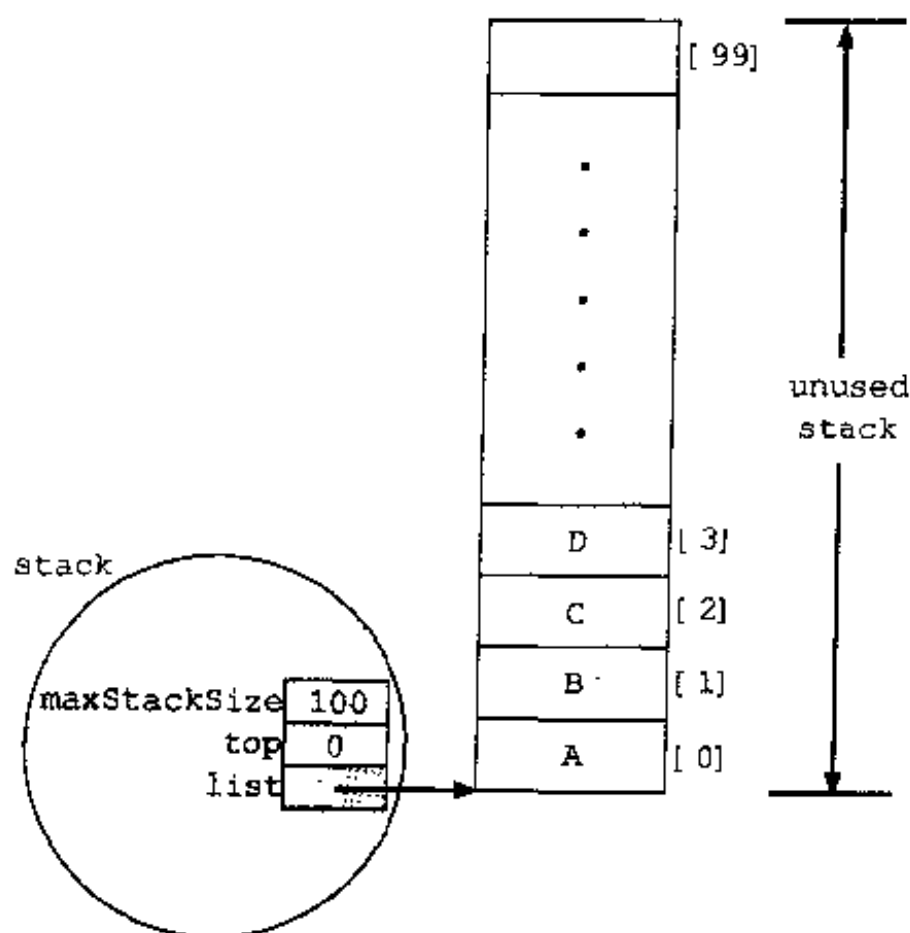


图 18.9 空栈

函数 `initializeStack()` 的定义如下所示:

```
template<class Type>
void stackType<Type>::initializeStack()
{
 top = 0;
} //end initializeStack
```

### 销毁栈

在栈的数组实现中, `destroyStack()` 操作与 `initializeStack()` 相似, 如果将 `top` 设置为 0, 则栈中的所有元素都被清除。尽管栈的元素还存放在数组中, 但 `top` 的值表明栈已经为空了。

```
template<class Type>
void stackType<Type>::destroyStack()
{
 top = 0;
} //end destroyStack
```

### 判断栈是否空

`top` 的值标志着栈是否为空, 如果 `top` 是 0, 栈为空; 否则, 栈不为空。函数 `isEmptyStack()` 的定义如下:

```
template<class Type>
bool stackType<Type>::isEmptyStack()
{
 return(top == 0);
} //end isEmptyStack
```

### 判断栈是否满

下面看一下操作 `isFullStack()`。如果 `top` 等于 `maxStackSize`，则栈为满。函数 `isFullStack()` 的定义如下：

```
template<class Type>
bool stackType<Type>::isFullStack()
{
 return(top == maxStackSize);
} //end isFullStack
```

### 构造函数和析构函数

构造函数和析构函数的实现很直观。带参数的构造函数将栈的大小设定为用户指定的值，并且将 `top` 设置为 0，然后创建一个合适的数组用来存储栈元素，构造函数使用默认值 100 来创建大小为 100 的数组。构造函数简单地给数组（同时也是栈）分配内存，并且设置 `top` 为 0。构造函数和析构函数的定义如下：

```
//constructor
template<class Type>
stackType<Type>::stackType(int stackSize)
{
 if(stackSize <= 0)
 {
 cout<<"The size of the array to hold the stack must "
 <<"be positive."<<endl;
 cout<<"Creating an array of size 100."<<endl;
 maxStackSize = 100;
 }
 else
 maxStackSize = stackSize; //set the stack size to
 //the value specified by
 //the parameter stackSize
 top = 0; //set top to 0
 list = new Type[maxStackSize]; //create the array to
 //hold the stack elements
} //end constructor

template<class Type>
stackType<Type>::~stackType() //destructor
{
 delete[] list; //deallocate the memory occupied by the array
} //end destructor
```

### 复制构造函数

当栈对象作为（传值）参数传递给函数时，复制构造函数将被调用。它将实参的数据成员对应地复制给形参的数据成员。它的定义如下：

```
template<class Type>
stackType<Type>::stackType(const stackType<Type>& otherStack)
{
 int j;

 maxStackSize = otherStack.maxStackSize;
 top = otherStack.top;
 list = new Type[maxStackSize]; //create the array
 if(top != 0) //if otherStack is not empty
```

```

 for(j = 0; j < top; j++) //copy other stack onto this stack
 list[j] = otherStack.list[j];
 } //end copy constructor

```

### 重载赋值运算符 (=)

回忆一下,对于有指针数据成员类,赋值操作符必须被显式地重载。针对类 `stackType` 的赋值运算符,重载函数定义如下:

```

template<class Type>
const stackType<Type>& stackType<Type>::operator=
 (const stackType<Type>& otherStack)
{
 int j;
 if(this != &otherStack) //avoid self-copy
 {
 if(maxStackSize != otherStack.maxStackSize)
 cout<<"Cannot copy. The two stacks are of "
 <<"different sizes"<<endl;
 else
 {
 top = otherStack.top;
 if(top != 0) //if the otherStack is not empty
 for(j = 0; j < top; j++) //copy the otherStack
 //onto this stack
 list[j] = otherStack.list[j];
 } //end else
 } //end if

 return *this;
} //end operator=

```

### 压入操作

将元素添加或者压入栈由两步组成。回忆一下, `top` 的值标志着栈中元素的数量, `top - 1` 给出了栈顶元素的数组下标。因此,压入操作步骤如下:

1. 将 `newItem` 存储在以 `top` 为下标的数组元素中。
2. `top` 加 1。

图 18.10 和图 18.11 说明了压入操作。

(a) 假设压入操作前,栈的情况如图 18.10 所示。

(b) 假设 `newItem` 为 'y'。在压入操作后,栈的情况如图 18.11 所示。

使用前面的算法,函数 `push()` 的定义如下:

```

template<class Type>
void stackType<Type>::push(const Type& newItem)
{
 list[top] = newItem; //add newItem at the top of the stack
 top++; //increment top
} //end push

```

这里给出的函数 `push()` 在存储 `newItem` 之前没有检查栈是否为满,因此,在调用函数 `push()` 之前,必须检查栈是否为满。所以,函数 `push()` 的调用方法为 (假设 `stack` 是 `stackType` 型的对象):

```

if (!stack.isFullStack())
 stack.push(newItem);

```



如果试图向一个已满的栈中添加新的元素，其导致的状态称为上溢。溢出检查有很多方法，其中一种就是前面的做法，或者可以在 `push()` 函数体内部进行溢出检查，算法如下：

```

If the stack is full
 Overflow is true
Else
 Overflow is false
 Add the new item to the stack
 Increment top

```

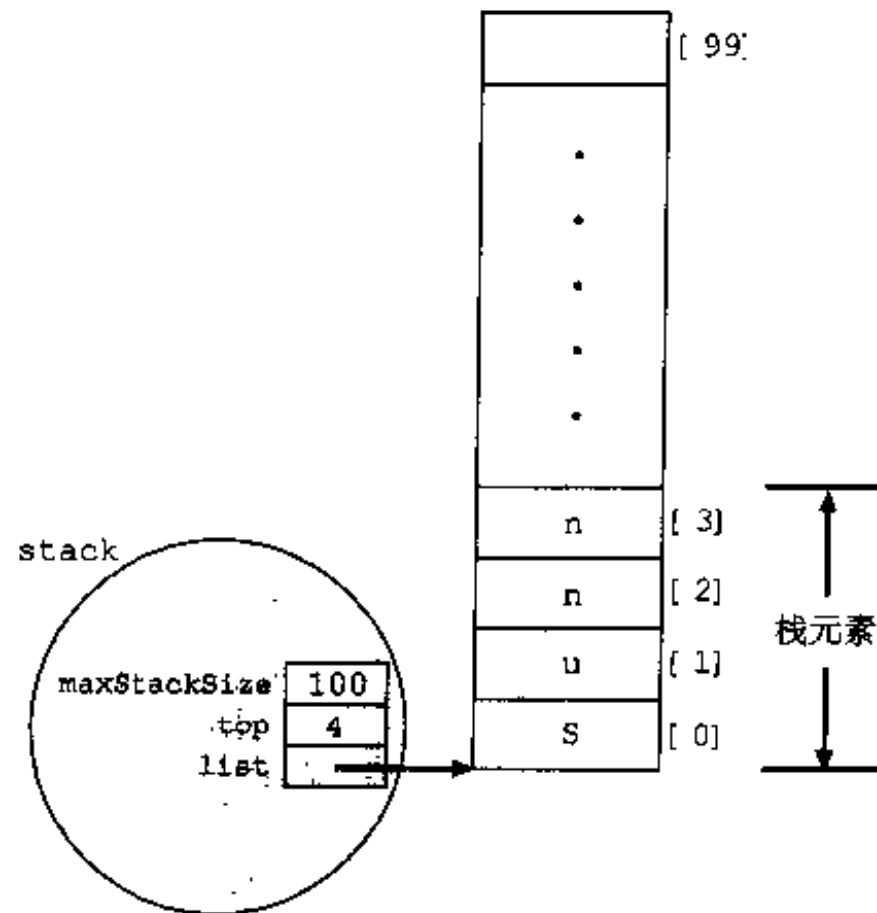


图 18.10 压入 `y` 前的栈

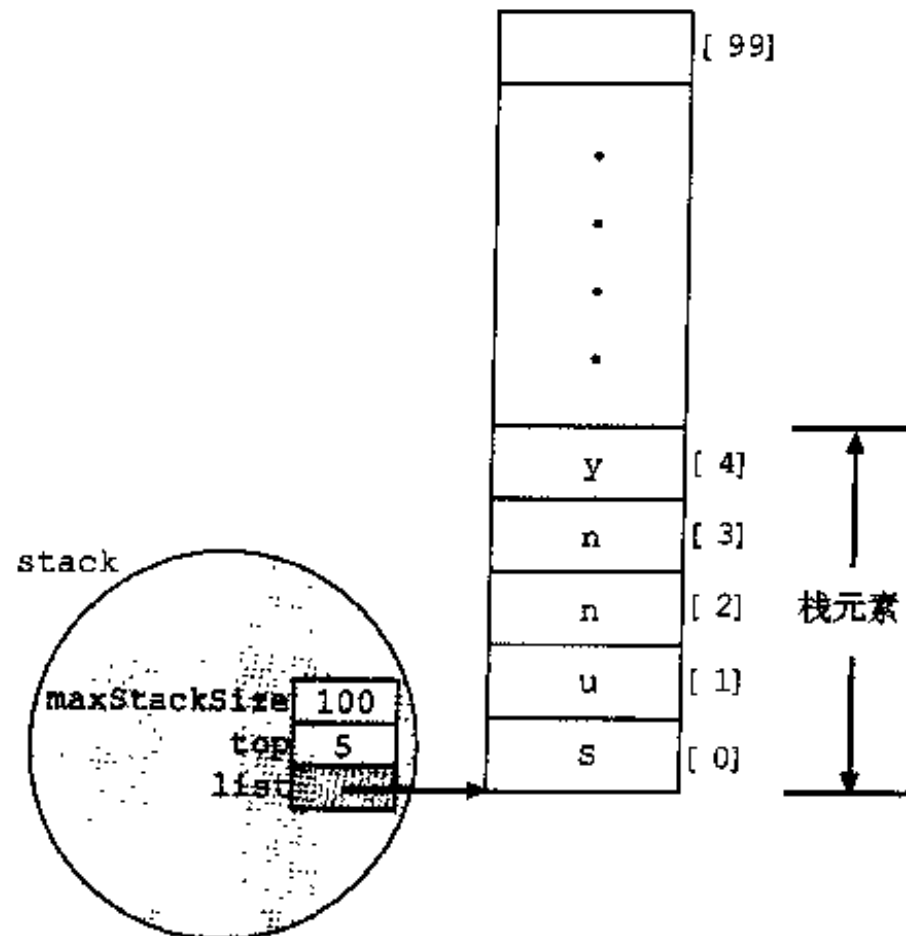


图 18.11 压入 `y` 后的栈

### 弹出操作

要从栈中删除（或称为弹出）一个元素，可以反向操作 `push()`，因此，`pop()` 操作的算法如下：

1. 递减 `top` 的值。
2. 将栈顶元素赋给 `poppedItem`。

图 18.12 和图 18.13 说明了 `pop` 操作。

(a) 假设在 `pop()` 操作之前，栈的情况如图 18.12 所示：

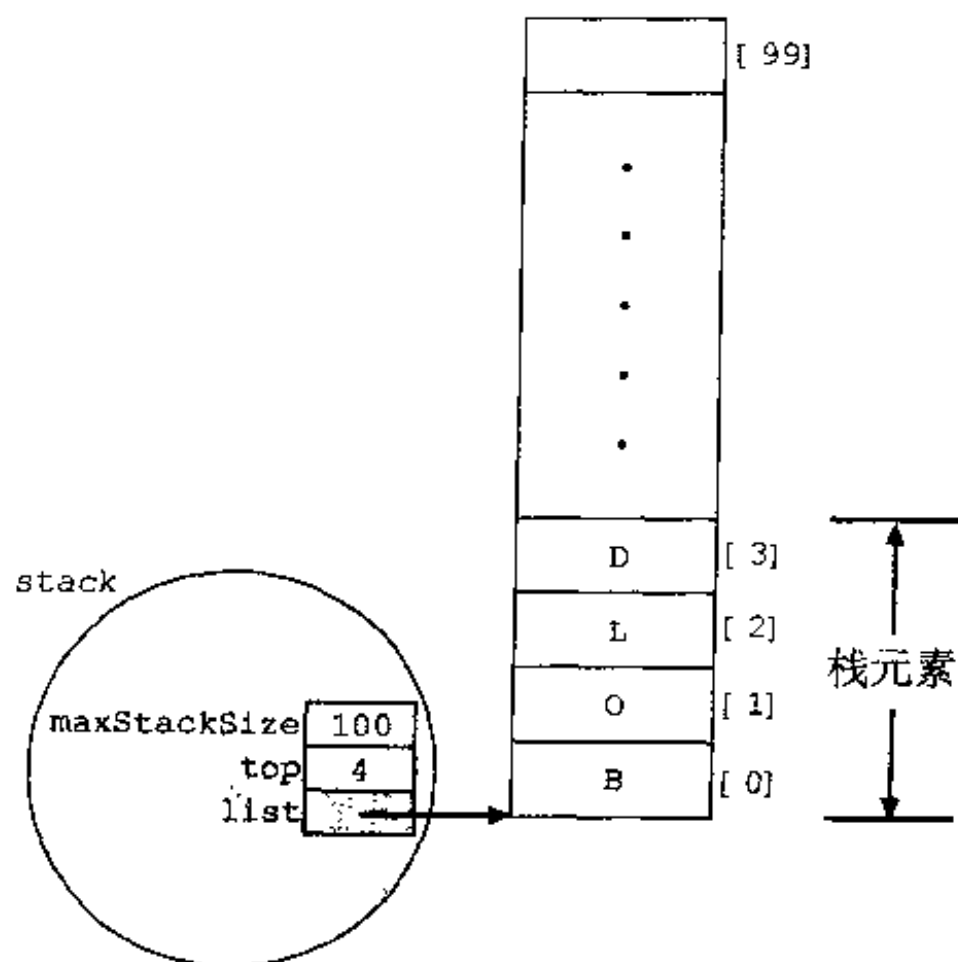


图 18.12 在弹出 D 之前的栈

(b) 在 `pop()` 操作之后，栈的情况如图 18.13 所示。

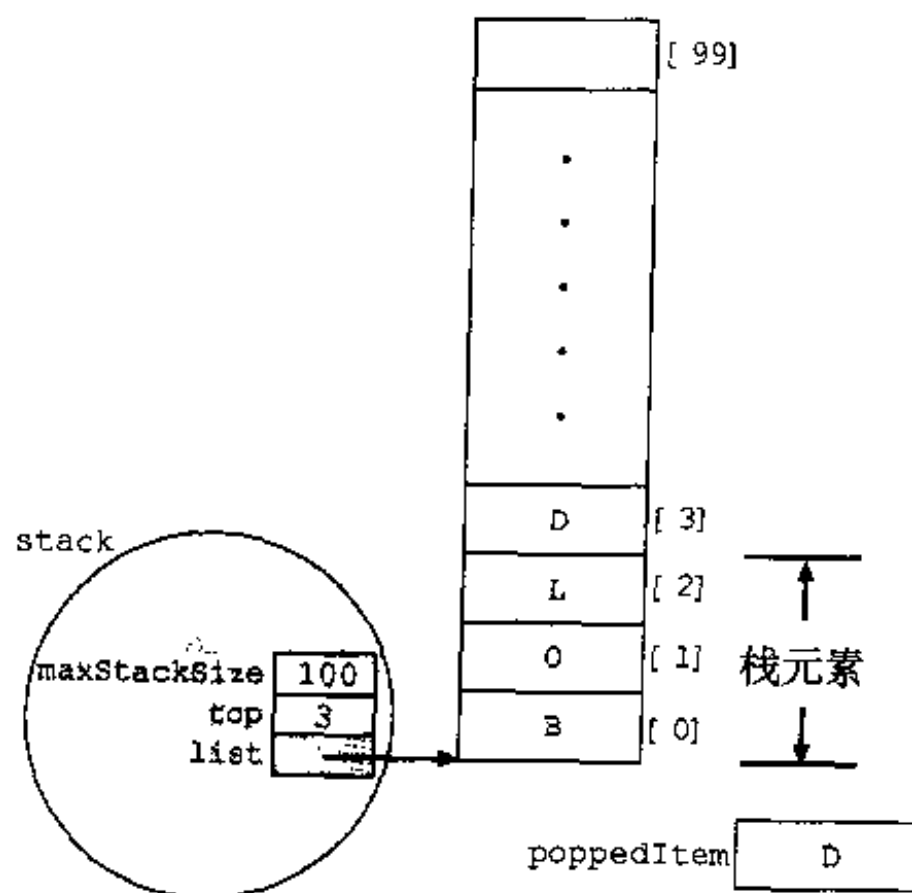


图 18.13 弹出 D 之后的栈



使用前面的算法，函数 pop() 的定义是：

```
template<class Type>
void stackType<Type>::pop(Type& poppedItem)
{
 top--; //decrement top
 poppedItem = list[top]; //copy the top element of
 //the stack onto poppedItem
} //end pop
```

必须在调用函数 pop() 之前检查栈是否为空（假设 stack 是一个 stackType 类型的对象）。

```
if(!stack.isEmptyStack())
 stack.pop(poppedItem);
```

从空栈中弹出元素导致的状态称为下溢。像 push() 的情形一样，pop() 的错误检查可以有多种方法。如果在 pop 函数体内进行下溢检查，则函数 pop() 用伪代码表示的算法如下所示：

```
If the stack is empty
 Underflow is true
Else
 Underflow is false
 Decrement top
 Assign the top element of the stack to poppedItem
```

### 18.2.1 栈的头文件

可以将类和实现函数的定义放在一起创建头文件。考虑到完整性，接下来讨论头文件（为了节省空间，只提供了类的定义，没有提供详细说明）。假设包含类 stackType 定义的头文件名是 myStack.h。我们会在使用到栈的程序中包含这个头文件。

```
//Header file: myStack.h

#ifndef H_StackType
#define H_StackType

template<class Type>
class stackType
{
public:
 const stackType<Type>& operator=(const stackType<Type>&);
 //overload the assignment operator
 void initializeStack();
 bool isEmptyStack();
 bool isFullStack();
 void destroyStack();
 void push(const Type& newItem);
 void pop(Type& poppedItem);
 stackType(int stackSize = 100);
 //constructor to specify the size of the stack
 stackType(const stackType<Type>& otherStack);
 //copy constructor
 ~stackType(); //destructor

private:
 int maxStackSize;
 int top;
 Type *list;
};
```

```
template<class Type>
void stackType<Type>::initializeStack()
{
 top = 0;
}

template<class Type>
bool stackType<Type>::isEmptyStack()
{
 return(top == 0);
}

template<class Type>
bool stackType<Type>::isFullStack()
{
 return(top == maxStackSize);
}

template<class Type>
void stackType<Type>::destroyStack()
{
 top = 0;
}

template<class Type>
void stackType<Type>::push(const Type& newItem)
{
 list[top] = newItem;
 top++;
}

template<class Type>
void stackType<Type>::pop(Type& poppedItem)
{
 top--;
 poppedItem = list[top];
}

template<class Type>
stackType<Type>::stackType(int stackSize)
{
 if(stackSize <= 0)
 {
 cout<<"The size of the array to hold the stack must "
 <<"be positive."<<endl;
 cout<<"Creating an array of size 100."<<endl;
 maxStackSize = 100;
 }
 else
 maxStackSize = stackSize;
 top = 0;
 list = new Type[maxStackSize];
} //end constructor

template<class Type>
stackType<Type>::~~stackType() //destructor
{
 top = 0;
 delete [] list;
}
```

```

template<class Type>
const stackType<Type>& stackType<Type>::operator=
 (const stackType<Type>& otherStack)
{
 int j;
 if(this != &otherStack) //avoid self-copy
 {
 if(maxStackSize != otherStack.maxStackSize)
 cout<<"Cannot copy. The two stacks are of "
 <<"different sizes."<<endl;
 else
 {
 top = otherStack.top;
 if(top != 0)
 for(j = 0; j < top; j++)
 list[j] = otherStack.list[j];
 }
 }
 return *this;
}

template<class Type> //copy constructor
stackType<Type>::stackType(const stackType<Type>& otherStack)
{
 int j;

 maxStackSize = otherStack.maxStackSize;
 top = otherStack.top;
 list = new Type[maxStackSize];

 if(top != 0)
 for(j = 0; j < top; j++)
 list[j] = otherStack.list[j];
}

#endif

```

### 例 18.1

下面给出示例程序，首先编写一个使用类 `stackType` 的程序来测试栈的操作（重点测试赋值操作和复制构造函数）。程序以及输出如下所示：

```

//Program to test the various operations of a stack
#include <iostream>
#include "myStack.h"

using namespace std;

void testCopyConstructor(stackType<int> otherStack);

int main()
{
 stackType<int> stack(50);
 stackType<int> copyStack(50);
 stackType<int> dummyStack(100);
 int x;

 stack.initializeStack();
 stack.push(23);
 stack.push(45);
}

```

```
stack.push(38);
copyStack = stack; //copy stack into copyStack
while(!copyStack.isEmptyStack()) //print copyStack
{
 copyStack.pop(x);
 cout<<"Inside copyStack "<<x<<endl;
}

copyStack = stack;
testCopyConstructor(stack); //test the copy constructor
if(!stack.isEmptyStack())
{
 cout<<"Original stack is not empty"<<endl;
 stack.pop(x);
 cout<<"Top element of the original stack: "<<x<<endl;
}

dummyStack = stack; //copy stack into dummyStack
return 0;
}

void testCopyConstructor(stackType<int> otherStack)
{
 int x;
 if(!otherStack.isEmptyStack())
 {
 cout<<"Other stack is not empty"<<endl;
 otherStack.pop(x);
 cout<<"Top element of the other stack: "<<x<<endl;
 }
}
```

输出:

```
Inside copyStack 38
Inside copyStack 45
Inside copyStack 23
Other stack is not empty
Top element of the other stack: 38
Original stack is not empty
Top element of the original stack: 38
Cannot copy. The two stacks are of different sizes.
```

### 18.3 程序实例：最大 GPA

在本例中,将编写一个 C++ 程序读入包含学生名字和 GPA 的数据文件。程序打印出最高的 GPA 和所有得到该 GPA 分值的学生名字。该程序只扫描输入文件一次。

输入 程序读入包含每一位学生名字和 GPA 的数据文件。样例数据如下:

```
3.5 Bill
3.6 John
2.7 Lisa
3.9 Kathy
3.4 Jason
3.9 David
3.4 Jack
```

输出 最高的 GPA 和所有得到这一 GPA 分数的学生名字。例如对于上面的数据, 最高的 GPA 是 3.9, 得到这一分数的学生是 Kathy 和 David。

### 程序分析和算法设计

首先读入 GPA 和学生的名字, 对于读入的第 1 个数据项, 假设是目前为止的最高 GPA。下面读入第 2 个 GPA 和学生的名字, 然后将第 2 个 GPA 与当前的最高 GPA (第 1 个 GPA) 进行比较, 会产生三种情况:

1. 新的 GPA 大于当前最高 GPA, 在这种情况下:
  - (a) 更新最高 GPA 的值。
  - (b) 销毁栈——即将栈中的学生名字全部删除。
  - (c) 在栈中保存得到当前最高 GPA 的学生名字。
2. 新的 GPA 等于当前最高 GPA。将该学生的名字压入栈内。
3. 新的 GPA 小于当前最高 GPA, 这样, 丢弃得到此分数的学生名字。

然后读入下一个 GPA 和学生名字, 重复步骤 1 至 3, 直到达到文件尾部。

从上面的讨论中可知, 程序需要下列变量:

```
double GPA; //variable to hold the current GPA
double highestGPA; //variable to hold the highest GPA
newString name; //variable to hold the name of the student
stackType<newString> stack; //object to implement the stack
```

注意, 在这里使用了在第 16 章中设计的类 newString 来表示学生的名字, 所以可以使用赋值运算符将名字存储进 newString 类变量中。

将前面的讨论转换成下列算法:

1. 声明变量。
2. 打开输入文件。
3. 如果输入文件不存在, 程序退出。
4. 设置输出的浮点数为固定的小数格式, 尾数补 0。将精度设为小数点后两位。
5. 读取 GPA 和学生名字。
6. highestGPA = GPA。
7. 初始化栈。
8. while (!EOF)
  - {
  - 8.1 if (GPA > highestGPA)
    - {
    - 8.1.1 clearstack(stack);
    - 8.1.2 push(stack, name);
    - 8.1.3 highestGPA = GPA;
    - }
  - 8.2 else  
if (GPA == highestGPA)  
push(stack, name);

8.3 读入 GPA 和学生名字。

}

9. 输出最高 GPA。

10. 输出得到最高 GPA 的学生名字。

#### 完整程序清单

```
//Program Highest GPA

#include <iostream>
#include <iomanip>
#include <fstream>
#include "myString.h"
#include "myStack.h"

using namespace std;

int main()
{
 //Step 1
 double GPA;
 double highestGPA;
 newString name;
 stackType<newString> stack(100);
 ifstream infile;

 infile.open("a:Ch18_HighestGPADData.txt"); //Step 2
 if(!infile) //Step 3
 {
 cout<<"Input file does not exist. "
 <<"Program terminates!"<<endl;
 return 1;
 }

 cout<<fixed<<showpoint; //Step 4
 cout<<setprecision(2); //Step 4
 infile>>GPA>>name; //Step 5
 highestGPA = GPA; //Step 6
 stack.initializeStack(); //Step 7
 while(infile) //Step 8
 {
 if(GPA > highestGPA) //Step 8.1
 {
 stack.destroyStack(); //Step 8.1.1
 if(!stack.isFullStack()) //Step 8.1.2
 stack.push(name);
 highestGPA = GPA; //Step 8.1.3
 }
 else
 {
 if(GPA == highestGPA) //Step 8.2
 if(!stack.isFullStack())
 stack.push(name);
 else
 {
 cout<<"Stack overflow. Program terminates."<<endl;
 return 1; //exit program
 }
 }
 }
}
```



```

 }
 infile>>GPA>>name; //Step 8.3
 }

 cout<<"Highest GPA = "<<highestGPA<<endl; //Step 9
 cout<<"Students holding the highest GPA are:"
 <<endl;

 while(!stack.isEmptyStack()) //Step 10
 {
 stack.pop(name);
 cout<<name<<endl;
 }

 cout<<endl;
 return 0;
}

```

### 运行实例

输入文件 ( a:Ch18\_HighestGPADData.txt )

```

3.4 Holt
3.2 Bolt
2.5 Colt
3.4 Tom
3.8 Ron
3.8 Mickey
3.6 Pluto
3.5 Donald
3.8 Cindy
3.7 Dome
3.9 Andy
3.8 Fox
3.9 Minne
2.7 Goofy
3.9 Doc
3.4 Danny

```

### 输出:

```

Highest GPA = 3.90
Students holding the highest GPA are:
Doc
Minne
Andy

```

## 18.4 栈的链表实现

由于数组的大小是固定的,使用(线性)数组表示的栈只能压入固定数量的元素。如果程序中元素的数量超过了数组的大小,程序会产生错误并退出,所以必须解决这类问题。

通过使用指针变量可以动态地分配和释放内存,通过使用链表可以动态地组织数据(比如排序链表)。接下来,将利用这些概念动态地实现一个栈。

回忆一下,在栈的线性(数组)表示中,  $top$  的值表示栈中元素的数量,值  $top - 1$  指向栈顶元素。在  $top$  的帮助下,可以找到栈顶元素,检查栈是否为空,等等。

与线性表示类似,在链表表示中,  $top$  用于定位栈顶元素。然而,又有一些微小的差别,在前一种情况中,  $top$  给出了数组的下标值;而后一种情况中,  $top$  给出的是栈顶元素的(内存)地址。

下列语句以 ADT 的形式定义了链表栈:

```
//Definition of the node
template <class Type>
struct nodeType
{
 Type info;
 nodeType<Type> *link;
};
template<class Type>
class linkedStackType
{
public:
 const linkedStackType<Type>& operator=
 (const linkedStackType<Type>&);
 //overload the assignment operator
 void initializeStack();
 //Initialize the stack to an empty state
 //Post: stack elements are removed; top = NULL
 bool isEmptyStack();
 //This function returns true if the stack is empty;
 //otherwise, it returns false
 bool isFullStack();
 //This function returns true if the stack is full;
 //otherwise, it returns false
 void push(const Type& newItem);
 //Add newItem to the stack.
 //Post: the stack is changed and the newItem
 // is added to the top of stack. top points to
 // the updated stack.
 void pop(Type& poppedElement);
 //Remove the top element of the stack.
 //Post: the stack is changed and the top
 // element is removed from the stack. The top
 // element of the stack is saved in poppedElement.
 void destroyStack();
 //Remove all elements of the stack, leaving the
 //stack in an empty state.
 //Post: top = NULL
 linkedStackType();
 //default constructor
 //Post: top = NULL
 linkedStackType(const linkedStackType<Type>& otherStack);
 //copy constructor
 ~linkedStackType();
 //destructor
 //All elements of the stack are removed from the stack
private:
 nodeType<Type> *top; //pointer to the stack
};
```

下面的例子显示了空链表栈和非空链表栈。

### 例 18.2

(a) 空栈: 假设 stack 是一个 linkedStackType 类型的对象 (参见图 18.14)。

(b) 非空栈 (参见图 18.15)。

在图 18.15 中, 栈顶元素是 C, 即最后一个压进栈的元素是 C。





图 18.14 空链表栈

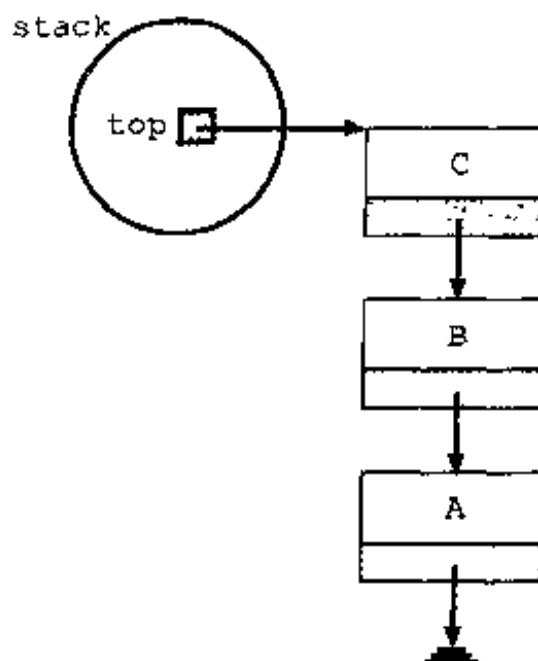


图 18.15 非空链表栈

下面讨论实现链表栈各种操作的函数定义。

### 默认构造函数

要考虑的第1种操作是默认构造函数。当一个栈对象被声明时，默认构造函数将栈初始化为空状态。因此，该函数将 top 设置为 NULL。函数的定义为：

```
template<class Type> //default constructor
linkedStackType<Type>::linkedStackType()
{
 top = NULL;
}
```

### 销毁栈

在栈的链表表示中，函数 destroyStack() 做的工作要比线性表示复杂一些。在数组表示中，通过简单地将 top 设置为 0 将栈销毁。在栈的链表表示中，存储栈元素的内存是动态分配的，因此，需要将 top 设置为 NULL，并且必须释放掉元素所占用的内存。

```
template<class Type>
void linkedStackType<Type>::destroyStack()
{
 nodeType<Type> *temp; //pointer to delete the node
 while(top != NULL) //while there are elements in the stack
 {
 temp = top; //set temp to point to the current node
 top = top->link; //advance top to the next node
 delete temp; //deallocate the memory occupied by temp
 }
} // end destroyStack
```

### 初始化栈

操作 initializeStack() 将栈再次初始化为空状态。因为栈中可能有一些元素，当使用链表来表示栈时，必须释放元素所占用的内存。这项任务可以通过调用成员函数 destroyStack() 来完成。注意，函数 destroyStack() 也将 top 设置为 NULL。函数的定义如下：

```
template<class Type>
void linkedStackType<Type>::initializeStack()
{
 destroyStack();
}
```

### 测试空栈和满栈

操作 `isEmptyStack()` 和 `isFullStack()` 要直观得多。如果 `top` 是 `NULL`，则栈为空；由于栈元素是动态分配和释放的，除非计算机的内存耗尽，否则栈永远不会满，所以，函数 `isFullStack()` 永远返回 `false`。这两个函数的定义如下：

```
template<class Type>
bool linkedStackType<Type>::isEmptyStack()
{
 return(top == NULL);
}

template<class Type>
bool linkedStackType<Type>::isFullStack()
{
 return false;
}
```

接下来讨论 `push()` 和 `pop()` 操作。从图 18.15 看出，`newElement` 将会添加到 `top` 指向的链表头部。在 `pop()` 的情况下，`top` 指向的节点将被删除。两种情况下，`top` 的值都被更新了。

### 压入操作

考虑图 18.16 所示的栈的情况。

假设要压入的新元素是 'D'。首先为新元素分配内存，然后将 'D' 存储在新节点内并将新节点插入在链表的头部，最后，更新 `top` 的值。语句：

```
newNode = new nodeType<Type>; //create the new node
newNode->info = newElement;
```

创建一个节点，将节点的地址存入变量 `newNode` 中，将 `newElement` 存入 `newNode` 的 `info` 域。由此得到图 18.17 所示的情形。

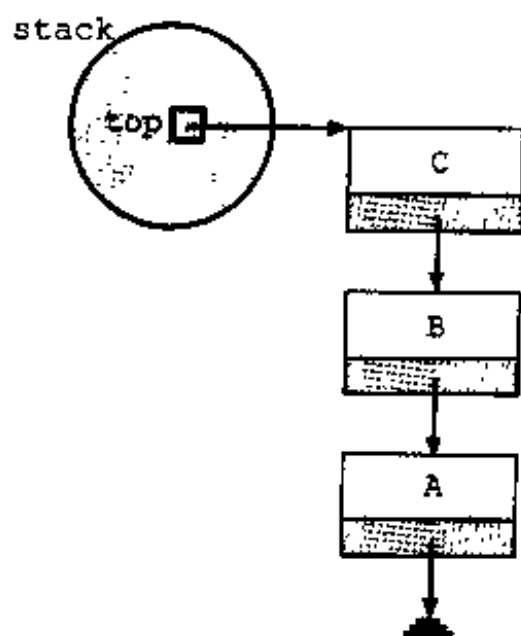


图 18.16 `push()` 操作前的栈

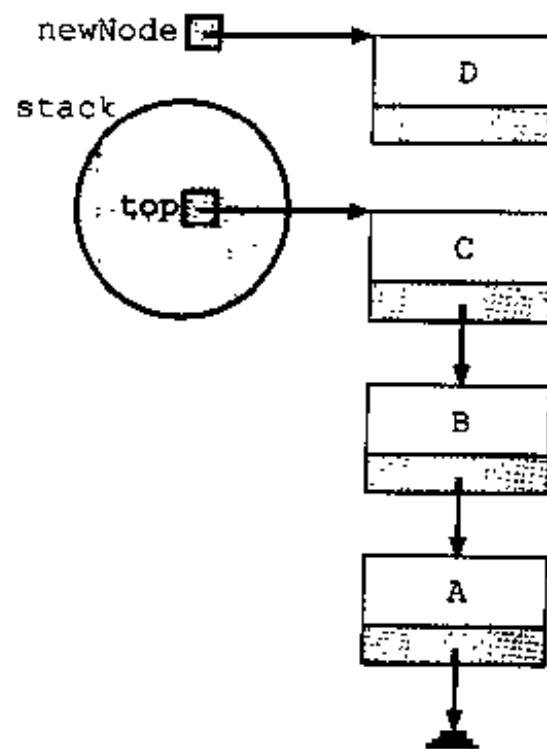


图 18.17 栈和 `newNode`

语句：

```
newNode->link = top;
```

将 `newNode` 插入栈的顶部，如图 18.18 所示。

语句:

```
top = newNode;
```

将 top 的值更新, 如图 18.19 所示。

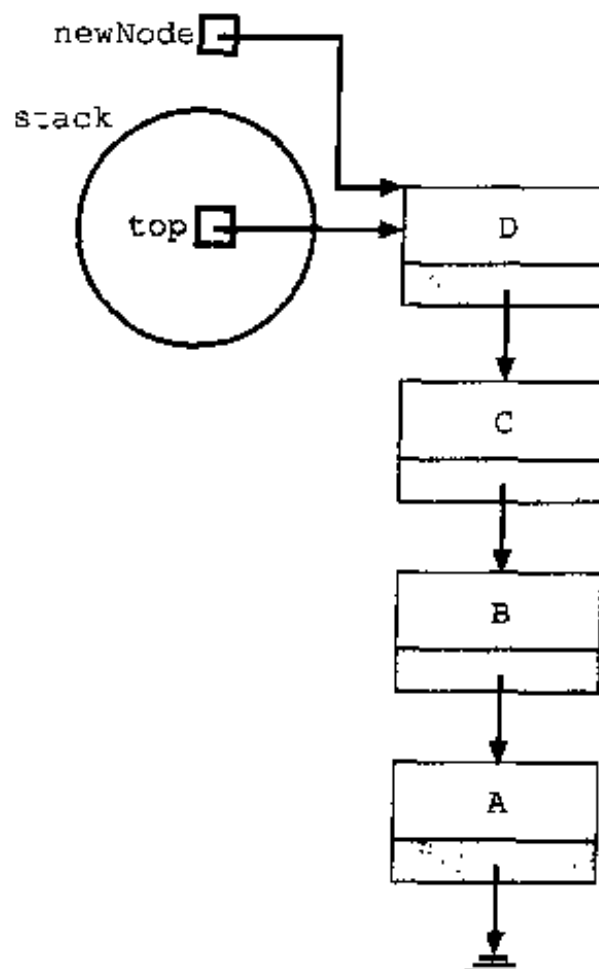
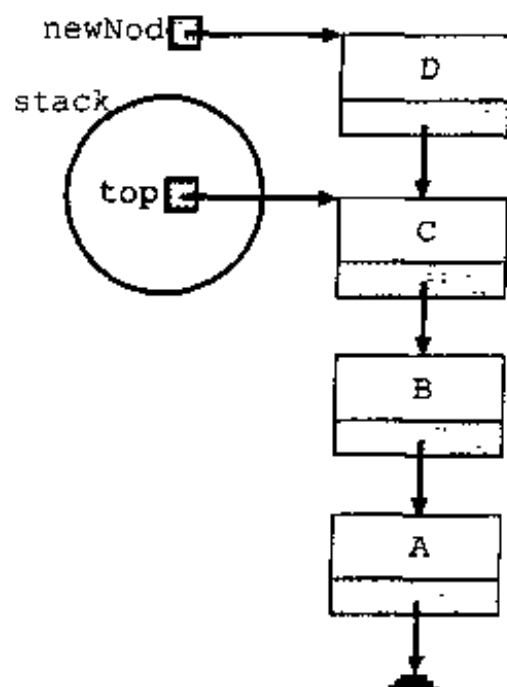


图 18.18 语句 “newNode->link = top;” 执行之后的栈    图 18.19 语句 “top = newNode;” 执行之后的栈

函数 push() 的定义如下:

```
template<class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
 nodeType<Type> *newNode; //pointer to create the new node
 newNode = new nodeType<Type>; //create the node
 newNode->info = newElement; //store newElement in the node
 newNode->link = top; //insert newNode before top
 top = newNode; //set top to point to the top node
} //end push
```

在这一实现中, 不需要在插入操作之前检查栈是否满。逻辑上讲, 栈是永远不会满的。

### 弹出操作

现在来考虑 pop() 操作, 它是 push() 的逆向操作。参见图 18.20 所示的栈。

假设栈顶元素存储在 poppedElement 中, 语句:

```
poppedElement = top->info;
```

将栈顶元素复制到 poppedElement 中。语句:

```
temp = top;
```

使 temp 指向栈顶, 而语句:

```
top = top->link;
```

使栈中的第 2 个元素成为栈顶元素, 这样就得到了如图 18.21 所示的情况。

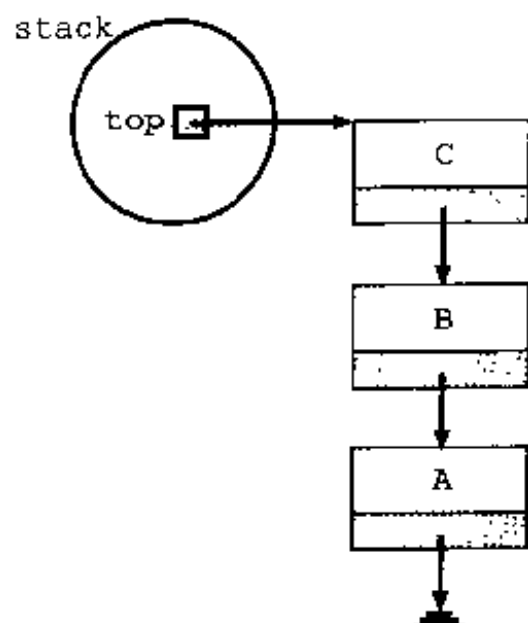


图 18.20 pop()操作之前的栈

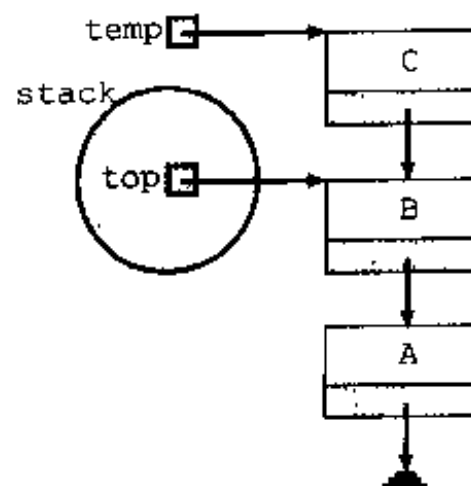


图 18.21 语句“top = top-&gt;link;”执行之后的栈

最后, 语句:

```
delete temp;
```

释放 temp 指向的内存。图 18.22 显示了栈的结果。

函数 pop() 的定义如下:

```
template<class Type>
void linkedStackType<Type>::pop (Type&
poppedElement)
{
 nodeType<Type> *temp; //pointer
 to deallocate memory

 poppedElement = top->info; //copy
 the top element into

 temp = top;
 top = top->link;
 delete temp;
} //end pop
```

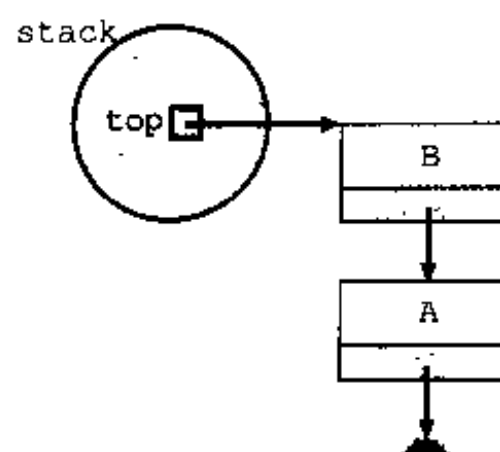


图 18.22 语句“delete temp;”执行之后的栈

```
//poppedElement
//set temp to point to the top node
//advance top to the next node
//delete the top node
```

必须在进行 pop() 操作之前, 检查栈是否为空。因此, 调用函数 pop() 之前必须调用函数 isEmptyStack()。这样, 函数 pop() 的调用如下所示 (假设 stack 是一个 linkedStackType 类型的对象):

```
if (!stack.isEmptyStack())
 stack.pop(poppedElement);
```

### 构造函数和析构函数

前面已经讨论了默认构造函数, 要完善栈操作的实现, 接下来给出复制构造函数和析构函数以及重载操作符的函数定义 (这些函数与第 17 章中链表部分的讨论类似)。

```
template<class Type> //copy constructor
linkedStackType<Type>::linkedStackType(const linkedStackType<Type>&
otherStack)
{
 nodeType<Type> *newNode, *current, *last;

 if (otherStack.top == NULL)
 top = NULL;
 else
 {
```

```

 current = otherStack.top; //set current to point to the
 //stack to be copied

 //copy the top element of the stack
 top = new nodeType<Type>; //create the node
 top->info = current->info; //copy the info
 top->link = NULL; //set the link field of the
 //node to null
 last = top; //set last to point to the node
 current = current->link; //set current to point to the
 //next node

 //copy the remaining stack
 while(current != NULL)
 {
 newNode = new nodeType<Type>;
 newNode->info = current->info;
 newNode->link = NULL;
 last->link = newNode;
 last = newNode;
 current = current->link;
 } //end while
 } //end else
} //end copy constructor

template<class Type> //destructor
linkedStackType<Type>::~~linkedStackType()
{
 nodeType<Type> *temp;
 while(top != NULL) //while there are elements in the stack
 {
 temp = top; //set temp to point to the current node
 top = top->link; //advance first to the next node
 delete temp; //deallocate the memory occupied by temp
 } //end while
} //end destructor

```

### 重载赋值运算符 (=)

针对类 linkedStackType, 重载赋值运算符的函数定义如下:

```

template<class Type> //overloading the assignment operator
const linkedStackType<Type>& linkedStackType<Type>::operator=
 (const linkedStackType<Type>& otherStack)
{
 nodeType<Type> *newNode, *current, *last;
 if(this != &otherStack) //avoid self-copy
 {
 if(top != NULL) //if the stack is not empty, destroy it
 destroyStack();

 if(otherStack.top == NULL)
 top = NULL;
 else
 {
 current = otherStack.top; //set current to point to
 //the stack to be copied
 //copy the top element of otherStack
 top = new nodeType<Type>; //create the node

```

```

 top->info = current->info; //copy the info
 top->link = NULL; //set the link field of the
 //node to null
 last = top; //make last point to the node
 current = current->link; //make current point to
 //the next node

 //copy the remaining elements of the stack
 while(current != NULL)
 {
 newNode = new nodeType<Type>;
 newNode->info = current->info;
 newNode->link = NULL;
 last->link = newNode;
 last = newNode;
 current = current->link;
 } //end while
 } //end else
} //end if
return *this;
} //end operator=

```

栈的定义和前面讨论的实现栈操作的函数是通用的。对于栈的链表表示, 必须将栈和实现栈的函数定义一起存储在一个(头)文件中。同样, 当声明一个栈对象时, 必须将栈元素的类型作为参数传递给类型 `linkedStackType`。例如, 语句:

```
linkedStackType<int> stack;
```

将 `stack` 声明为一个 `linkedStackType` 型的对象, 并且栈元素的类型是 `int` 型。类似地, 语句:

```
linkedStackType<newString> stringStack;
```

将 `stringStack` 声明为一个 `linkedStackType` 型的对象并且栈元素的数据类型是 `newString`。

在例 18.3 中, 编写了一个简短的程序用来检查链表栈上的各种操作。我们鼓励读者自己动手编写这样的程序。

### 例 18.3

假设类 `linkedStackType` 和用于实现栈操作的函数定义包含在头文件 `linkedStack.h` 中。

```

//This program tests the various operations of a linked stack
#include <iostream>
#include "linkedStack.h"
using namespace std;
void testCopy(linkedStackType<int> OStack);
int main()
{
 linkedStackType<int> stack;
 linkedStackType<int> otherStack;
 linkedStackType<int> newStack;
 int num;

 stack.push(34);
 stack.push(43);
 stack.push(27);
 newStack = stack;

 cout<<"After the assignment operator, newStack: "<<endl;

```



```

 while(!newStack.isEmptyStack())
 {
 newStack.pop(num);
 cout<<num<<endl;
 }

 otherStack = stack;
 cout<<"Testing the copy constructor"<<endl;
 testCopy(otherStack);
 cout<<"After the copy constructor, otherStack: "<<endl;
 while(!otherStack.isEmptyStack())
 {
 otherStack.pop(num);
 cout<<num<<endl;
 }

 return 0;
 }

 void testCopy(linkedStackType<int> OStack) //function to test the
 //copy constructor
 {
 int num;
 cout<<"Stack in the function testCopy:"<<endl;
 while(!OStack.isEmptyStack())
 {
 OStack.pop(num);
 cout<<num<<endl;
 }
 }
}

```

输出:

```

After the assignment operator, newStack:
27
43
34
Testing the copy constructor
Stack in the function testCopy:
27
43
34
After the copy constructor, otherStack:
27
43
34

```

#### 18.4.1 从类 linkedListType 派生出的栈

如果将栈的函数 push() 和在第 17 章讨论过的通用链表的函数 insertFirst() 相比较, 可以发现实现这些操作的算法是相同的。通过对其他函数 (例如, initializeStack() 和 initializeList(), isEmptyList() 和 isEmptyStack(), 等等) 的比较表明, 类 linkedStackType 可以从类 linkedListType 中派生出来。而 pop() 函数可以按照前面几节讨论的方式实现。

下面定义类 linkedStackType。它是从类 linkedListType 中派生出的, 同时提供了实现栈操作的

函数定义。

```

#ifndef H_derivedLinkedStack
#define H_derivedLinkedStack
#include <iostream>
#include "linkedList.h"
using namespace std;

template<class Type>
class linkedStackType: public linkedListType<Type>
{
 public:
 void initializeStack();
 bool isEmptyStack();
 bool isFullStack();
 void push(const Type& newItem);
 void pop(Type& poppedElement);
 void destroyStack();
};

template<class Type>
void linkedStackType<Type>::initializeStack()
{
 linkedListType<Type>::initializeList();
}

template<class Type>
bool linkedStackType<Type>::isEmptyStack()
{
 return linkedListType<Type>::isEmptyList();
}

template<class Type>
bool linkedStackType<Type>::isFullStack()
{
 return linkedListType<Type>::isFullList();
}

template<class Type>
void linkedStackType<Type>::destroyStack()
{
 linkedListType<Type>::destroyList();
}

template<class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
 linkedListType<Type>::insertFirst(newElement);
}

template<class Type>
void linkedStackType<Type>::pop(Type& poppedElement)
{
 nodeType<Type> *temp;
 poppedElement = first->info;
 temp = first;
 first = first->link;
 delete temp;
}
#endif

```



## 18.5 栈的应用：后缀表达式计算器

通常用来书写算术表达式的符号称为中缀符，因为这些运算符是写在操作数之间的。例如，在表达式  $a+b$  中，运算符  $+$  在操作数  $a$  和  $b$  之间。在中缀符号中，运算符是具有优先级的。即必须从左到右计算表达式的值，并且乘法和除法有着高于加法和减法的优先级。如果希望以不同的顺序计算表达式，必须包含括号。例如，在表达式  $a+b \times c$  中，首先使用操作数  $b$  和  $c$  计算乘法，然后用操作数  $a$  和  $b \times c$  的结果来计算加法。

在 20 世纪 50 年代早期，波兰数学家 Lukasiewicz 发现，如果将运算符写在操作数之前（前缀式或者波兰式。例如， $+ab$ ）或者之后（后缀式或者逆波兰式。例如， $ab+$ ），则括号可以被省略。例如，表达式：

$$a + b * c$$

在后缀表达式中为：

$$abc * +$$

下面的例子显示了中缀表达式和对等的后缀表达式。

例 18.4

| 中缀表达式                       | 对等的后缀表达式             |
|-----------------------------|----------------------|
| $a + b$                     | $ab +$               |
| $a + b * c$                 | $abc * +$            |
| $a * b + c$                 | $ab * c +$           |
| $(a + b) * c$               | $ab + c *$           |
| $(a - b) * (c + d)$         | $ab - cd + *$        |
| $(a + b) * (c - d / e) + f$ | $ab + cde / - * f +$ |

不久之后，人们发现后缀表达式在计算机科学中有着极为重要的应用。实际上，很多编译器都是首先将算术表达式转换为某种后缀表达式，然后再将后缀表达式翻译为机器代码的。可以使用下面的算法对后缀表达式进行求值。

从左到右扫描表达式，当找到一个运算符时，存储起来以便得到需要的操作数并进行运算，然后继续这一过程。

请看下面的后缀表达式：

$$6 \ 3 \ + \ 2 \ * \ =$$

下面使用栈和上述算法来进行求值。

1. 读入第 1 个符号 6，将其压入栈（参见图 18.23）。
2. 读入下一个符号 3，将其压入栈（参见图 18.24）。



图 18.23 压入 6 后的栈

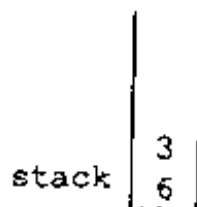


图 18.24 压入 3 后的栈

3. 读入下一个符号（运算符  $+$ ），因为一个运算符需要两个操作数才能求值，所以将栈弹出两次。进行运算并把运算结果压入栈中（参见图 18.25）。

执行运算:  $op2 + op1 = 6 + 3 = 9$ 。

将结果压入栈中 (参见图 18.26)。

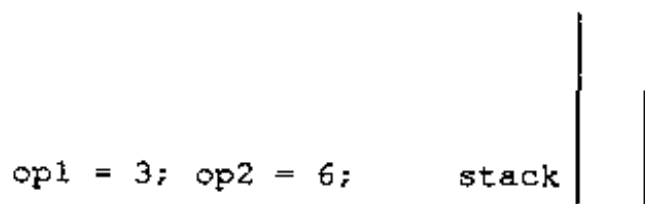


图 18.25 弹出两次后的栈

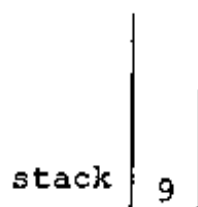


图 18.26 将  $op2 + op1$  的结果 (9) 压入后的栈

4. 读入下一个符号 2, 将其压入栈中 (参见图 18.27)。

5. 读入下一个符号 (运算符 \*), 因为一个运算符需要两个操作数才能求值, 所以将栈弹出两次。进行运算并把运算结果压入栈中 (参见图 18.28 和图 18.29)。

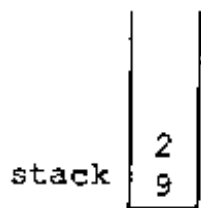


图 18.27 压入 2 后的栈

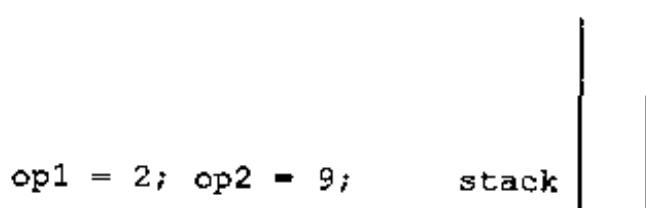


图 18.28 压入弹出两次后的栈

执行运算:  $op2 * op1 = 9 \times 2 = 18$ 。

将结果压入栈中 (参见图 18.29)。

6. 扫描下一个符号 (等号 =), 标志着表达式的结束。因此, 可以打印结果。表达式的结果在栈中, 所以将其弹出并打印, 如图 18.30 所示。



图 18.29 将  $op2 * op1$  的结果 (18) 压入后的栈

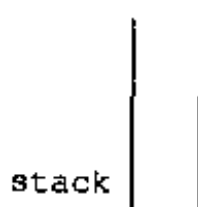


图 18.30 弹出元素后的栈

表达式  $63 + 2 * = 18$ 。

从前面的讨论中, 可以看出当读入一个符号而非数字时, 会产生下列情况:

1. 读入的符号是 +、-、/、\* 或者 = 之一。

a. 如果符号是 +、-、/ 或者 \*, 符号是一个运算符, 所以必须进行求值计算。因为一个运算符需要两个操作数, 栈必须至少弹出两个元素, 否则, 表达式有错。

b. 如果符号是 = (等号), 表达式结束并且必须打印结果。在这一步中, 栈中必须只包含一个元素, 否则表达式有错。

2. 符号不是 +、-、/、\* 或者 = 当中任何一个, 这种情况下, 表达式包含非法的运算符。

同样可以看出, 当遇到一个操作数 (数字) 时, 该数会被压进栈, 因为运算符在操作数之后。请看下面的表达式:

(i)  $76 + 3 ; 6 - =$

(ii)  $14 - 23 * =$

(iii)  $1423 + =$

表达式 (i) 含有非法运算符; 表达式 (ii) 中 + 没有足够的操作数; 表达式 (iii) 含有太多操作数。在表达式 (iii) 的情况下, 当遇到等号 (=) 时, 栈中有两个元素, 这一错误直到准备输出表达式的值

时才被发现。

要使输入便于阅读, 可以将表达式写成下面的形式:

#6 #3 + #2 \* =

符号 # 存在于表达式中每个数字的前面, 如果扫描到的符号是 #, 那么下一个输入将是一个数字 (即操作数) 或者是一个等号 (标志表达式结束)。假设表达式中只包含 +、-、/ 和 \* 运算符。

程序将整个后缀表达式和结果一起输出。如果表达式有错误, 则表达式被丢弃。在这种情况下, 程序输出整个表达式和一条适当的错误信息。由于表达式可能含有错误, 所以在处理下一个表达式之前, 必须将栈清空。

### 主算法逻辑

依照前面的讨论, 伪代码形式的主算法为:

```

read the first ch;
while more data to process
{
 clear stack;
 output ch;

 while (ch is not = '=') //process each expression
 // = marks the end of an expression
 {
 switch(ch)
 {
 case '#': read a number
 output the number;
 push the number onto the stack;
 break;
 default: Assume that ch is an operation
 evaluate the operation;
 } //end switch

 if no error was found, then
 {
 read next ch;
 output ch;
 }
 } //end while

 if the expression did not contain any error(s), then
 output the result;
 else
 discard the result;

 start processing the next expression;
}

```

**求值:** 该函数计算一个表达式的值。每次运算需要两个操作数, 操作数保存在栈中。因此, 栈至少包含两个数字。如果栈包含少于两个数字, 则表达式有错误。这种情况下, 整个表达式被丢弃并且输出一条适当的错误信息。该函数同样会检查任何其他的非法操作。函数的伪代码形式为:

```

if stack is empty
{
 error in the expression
 set expressionOk to false
}

```

```

else
{
 pop stack, op1 //get the first number from the stack
 if stack is empty
 {
 error in the expression
 set expressionOk to false
 }
 else
 {
 pop stack, op2 //get the second number

 //if the operation is legal, perform the
 //operation and push the result onto the stack
 switch(ch)
 {
 case '+': //perform the operation and push the result
 //onto the stack
 stack.push(op2+op1);
 case '-': //perform the operation and push the result
 //onto the stack
 stack.push(op2-op1);
 case '*': //perform the operation and push the result
 //onto the stack
 stack.push(op2*op1);
 case '/': //perform the operation and push the result
 //onto the stack
 stack.push(op2/op1);
 otherwise operation is illegal
 {
 output an appropriate message;
 set expressionOk to false
 }
 } //end switch
 }
}

```

丢弃：该函数在发现表达式有错误的时候被调用。它只是读入输入并将其输出，直到遇到=。

```

while(ch != '=')
{
 read ch;
 output ch;
}

```

### 完整程序清单

```

//Postfix Calculator
#include <iostream>
#include <iomanip>
#include <fstream>
#include "mystack.h"

using namespace std;

void evaluate(ofstream& out, stackType<double>& stack,
 char& ch, bool& expressionOk);
void discard(ifstream& in, ofstream& out, char& ch);

int main()
{
 double num, result;

```

```

bool expressionOk;
char ch;
stackType<double> stack(100);
ifstream infile;
ofstream outfile;

infile.open("a:Ch18_RpnData.txt");
if(!infile)
{
 cout<<"Cannot open input file. Program terminates!"<<endl;
 return 1;
}

outfile.open("a:Ch18_RpnOutput.txt");
outfile<<fixed<<showpoint;
outfile<<setprecision(2);

infile>>ch;
while(infile)
{
 stack.initializeStack();
 expressionOk = true;
 outfile<<ch;
 while(ch != '=')
 {
 switch(ch)
 {
 case '#': infile>>num;
 outfile<<num<<" ";
 if(!stack.isFullStack())
 stack.push(num);
 else
 {
 cout<<"Stack overflow. "
 <<"Program terminates!"<<endl;
 return 1;
 }
 break;
 default: evaluate(outfile, stack, ch, expressionOk);
 } //end switch

 if(expressionOk) //if no error
 {
 infile>>ch;
 outfile<<ch;
 if(ch != '#')
 outfile<<" ";
 }
 else
 discard(infile, outfile, ch);
 } //end while (ch != '=')

 if(expressionOk) //if no error, print the result
 {
 if(!stack.isEmptyStack())
 {
 stack.pop(result);

```

```

 if(stack.isEmptyStack())
 outfile<<result<<endl;
 else
 outfile<<" (Error: Too many operands)"<<endl;
 } //end if
 else
 outfile<<" (Error in the expression)"<<endl;
}
else
 outfile<<" (Error in the expression)"<<endl;
outfile<<" _____ "<<endl<<endl;
infile>>ch; //begin processing the next expression
} //end while

infile.close();
outfile.close();
return 0;
} //end main

void evaluate(ofstream& out, stackType<double>& stack,
 char& ch, bool& expressionOk)
{
 double op1, op2;
 if(stack.isEmptyStack())
 {
 out<<" (Not enough operands)";
 expressionOk = false;
 }
 else
 {
 stack.pop(op1);
 if(stack.isEmptyStack())
 {
 out<<" (Not enough operands)";
 expressionOk = false;
 }
 else
 {
 stack.pop(op2);
 switch(ch)
 {
 case '+': stack.push(op2 + op1);
 break;
 case '-': stack.push(op2 - op1);
 break;
 case '*': stack.push(op2 * op1);
 break;
 case '/': stack.push(op2 / op1);
 break;
 default: out<<" (Illegal operator)";
 expressionOk = false;
 } //end switch
 } //end else
 } //end else
} //end evaluate

```



```

void discard(istream& in, ostream& out, char& ch)
{
 while(ch != '\n')
 {
 in.get(ch);
 out<<ch;
 }
} //end discard

```

运行实例:

输入文件

```

#35 #27 + #3 * =
#26 #28 + #32 #2 ; - #5 / =
#23 #30 #15 * / =
#2 #3 #4 + =
#20 #29 #9 * ; -
#25 #23 - + =
#34 #24 #12 #7 / * + #23 - =

```

输出:

```

#35.00 #27.00 + #3.00 * = 186.00

#26.00 #28.00 + #32.00 #2.00 ; (Illegal operator) - #5 / = (Error in the
expression)

#23.00 #30.00 #15.00 * / = 0.05

#2.00 #3.00 #4.00 + = (Error: Too many operands)

#20.00 #29.00 #9.00 * ; (Illegal operator) = (Error in the expression)

#25.00 #23.00 - + (Not enough operands) = (Error in the expression)

#34.00 #24.00 #12.00 #7.00 / * + #23.00 - = 52.14

```

## 18.6 消除递归: 非递归逆序打印链表的算法

在第 17 章中, 曾使用递归逆序打印一个链表, 在本节中, 将学习如何使用栈来设计一个非递归的算法逆序打印链表。

请看如图 18.31 所示的链表。

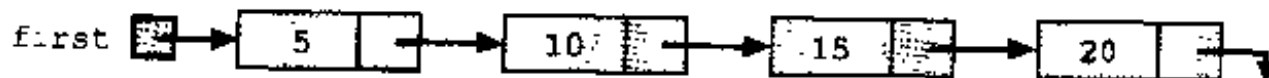


图 18.31 链表

要逆序打印链表, 首先需要得到链表的最后一个节点, 可以从第 1 个节点向后遍历。然而, 一旦找到了最后一个节点, 怎样返回到前一个节点呢? 可以再一次从第 1 个节点以适当的结束条件向后遍历, 但是这样显然浪费了可观的计算时间, 尤其是当链表较长时这种浪费会更为明显。而且如果对链表中的每一个节点都这样做, 程序会变得十分缓慢。接下来将展示如何使用栈来有效地逆序

打印链表。

在打印完某一个节点的 info 后,需要立即移到这个节点之后的节点上。例如,在打印了 20 后,需要移动到含有 15 的节点。因此,在遍历链表并得到最后一个节点的过程中,需要保存对每一个节点的指针。例如,对图 18.31 所示的链表,必须保存对应于包含 info 5、10 和 15 的每一个节点的指针。在打印了 20 之后,需要移到含有 info 15 的节点,在打印了 15 之后,需要移到含有 info 10 的节点,依次类推。这样看来,必须将每一个节点的指针保存在一个栈中(“后进先出”)。

因为通常表中节点的数量是不固定的,可以使用链表实现栈。假设 stack 是一个 linkedListType 类型的对象, current 是与 first 相同类型的指针。则下面的语句:

```
current = first; //Line 1
while (current != NULL) //Line 2
{
 stack.push(current); //Line 3
 current = current->link; //Line 4
}
```

在 Line 1 语句执行后, current 指向第 1 个节点(参见图 18.32)。

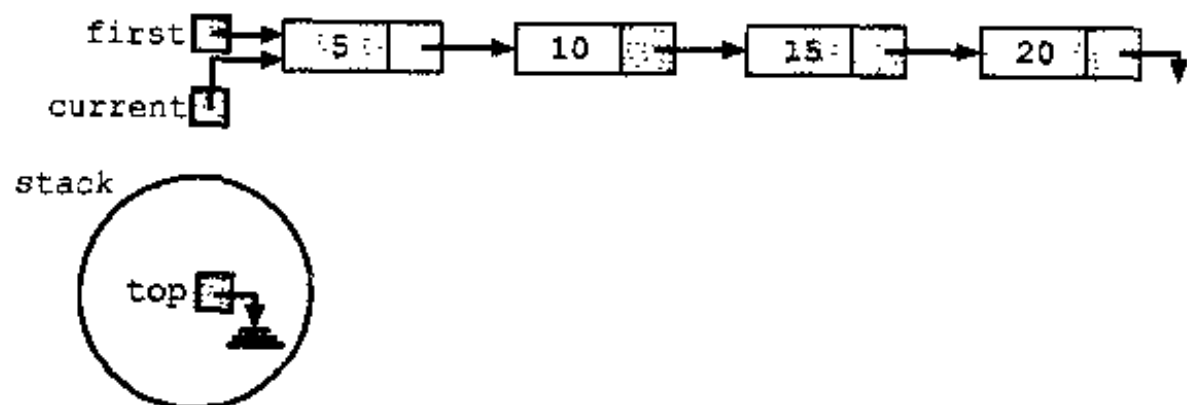


图 18.32 语句“current = first;”执行后的链表

因为 current 不是 NULL, Line 3 和 Line 4 语句得到执行(参见图 18.33)。

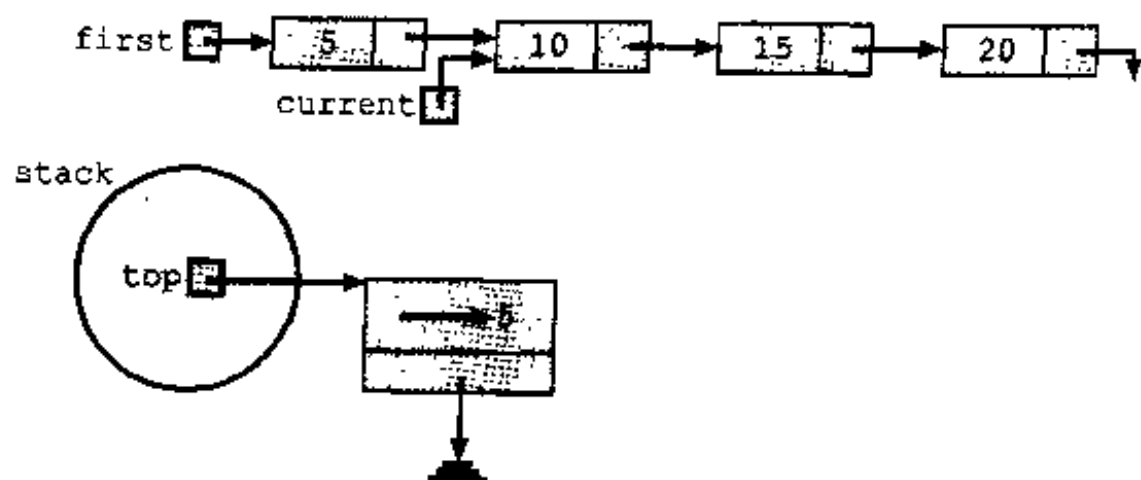


图 18.33 语句“stack.push(current);”和“current = current->link;”执行之后的链表和栈

在 Line 4 语句执行之后, Line 2 的循环条件被重新计算。因为 current 不是 NULL, 循环条件的值是 true, 所以 Line 3 和 Line 4 的语句得到执行(参见图 18.34)。

在 Line 4 语句执行之后, Line 2 的循环条件被重新计算。因为 current 不是 NULL, 循环条件的值是 true, 所以 Line 3 和 Line 4 的语句得到执行(参见图 18.35)。

在 Line 4 语句执行之后, Line 2 的循环条件被重新计算。因为 current 不是 NULL, 循环条件的值是 true, 所以 Line 3 和 Line 4 的语句得到执行(参见图 18.36)。



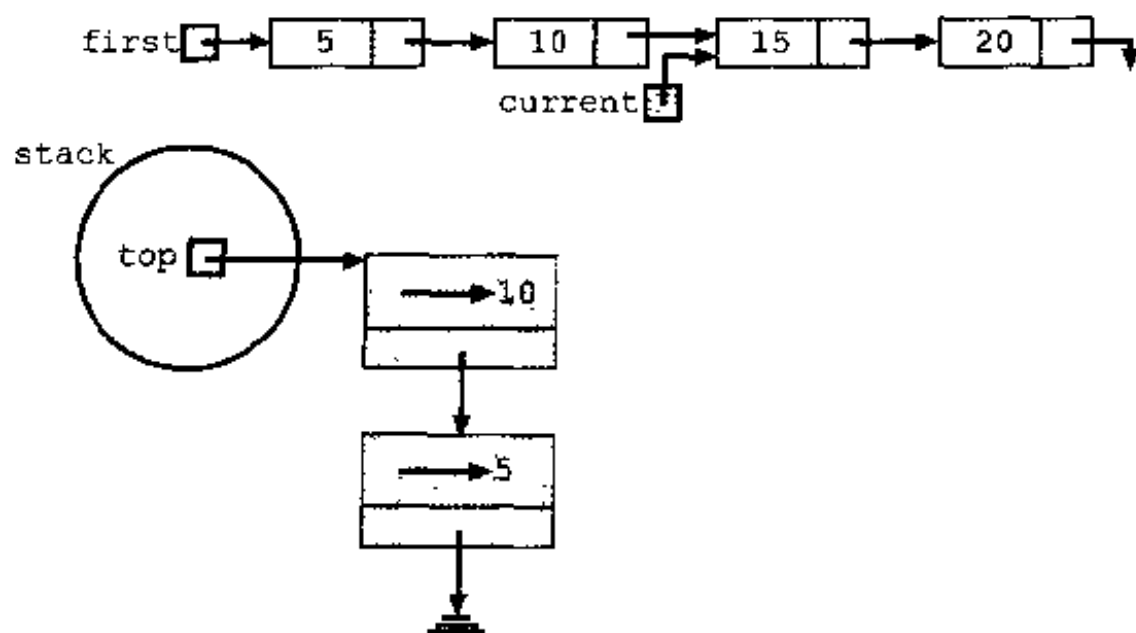


图 18.34 语句“stack.push(current);”和“current = current->link;”执行之后的链表和栈

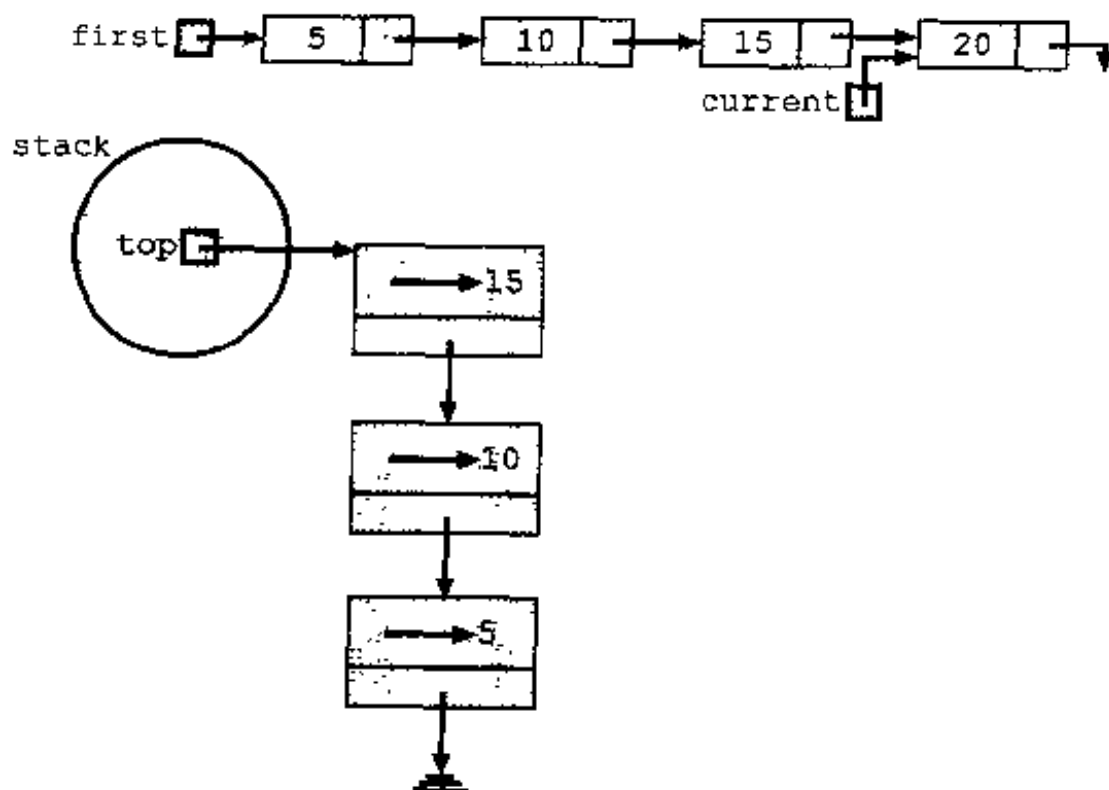


图 18.35 语句“stack.push(current);”和“current = current->link;”执行之后的链表和栈

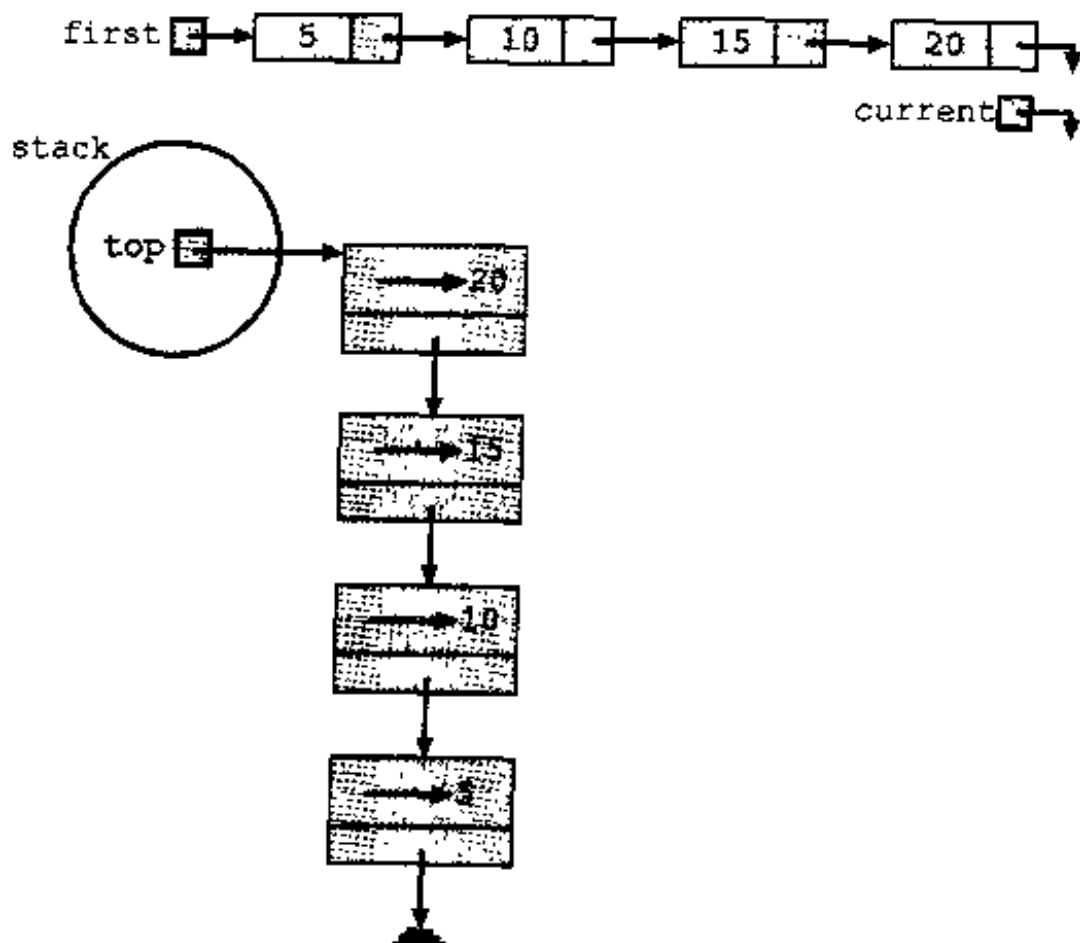


图 18.36 语句“stack.push(current);”和“current = current->link;”执行之后的链表和栈

在 Line 4 语句执行之后, Line 2 的循环条件被重新计算。因为 current 是 NULL, 循环条件的值是 false, 所以 Line 2 的 while 循环终止了。从图 18.36 中可以看出, 链表中指向每一个节点的指针都被存储在栈中。栈的栈顶元素包含有指向链表中最后一个节点的指针, 依次类推。现在执行下列语句:

```
while (!stack.isEmptyStack()) //Line 5
{
 stack.pop(current); //Line 6
 cout<<current->info<<" "; //Line 7
}
```

Line 5 的循环条件值为 true, 因为栈非空。因此 Line 6 和 Line 7 语句得到执行, 在执行完 Line 6 语句后, current 指向最后一个节点 (参见图 18.37)。

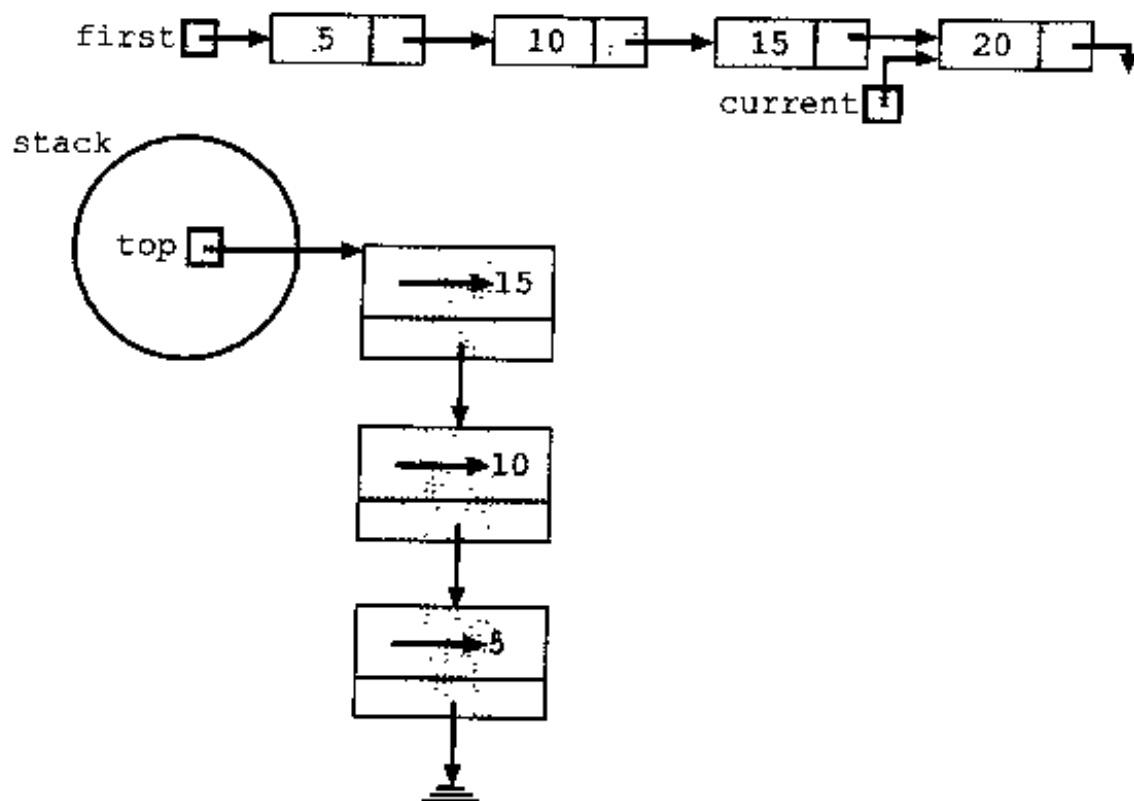


图 18.37 语句 “stack.pop(current);” 执行之后的链表和栈

Line 7 语句输出 current->info, 等于 20。接下来 Line 5 的循环条件开始求值, 它的值为 true, 所以 Line 6 和 Line 7 语句得到执行, Line 6 语句执行之后的结果如图 18.38 所示。

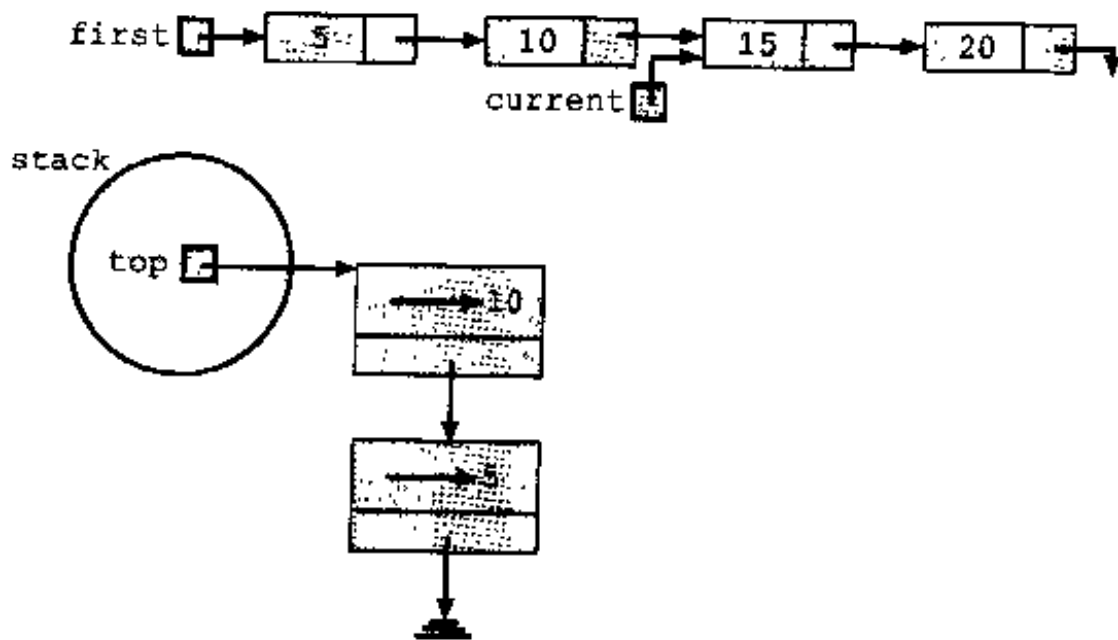


图 18.38 语句 “stack.pop(current);” 执行之后的链表和栈

Line 7 语句输出 current->info, 等于 15。接下来 Line 5 的循环条件开始求值, 它的值为 true, 所以 Line 6 和 Line 7 语句得到执行, Line 6 语句执行之后的结果如图 18.39 所示。

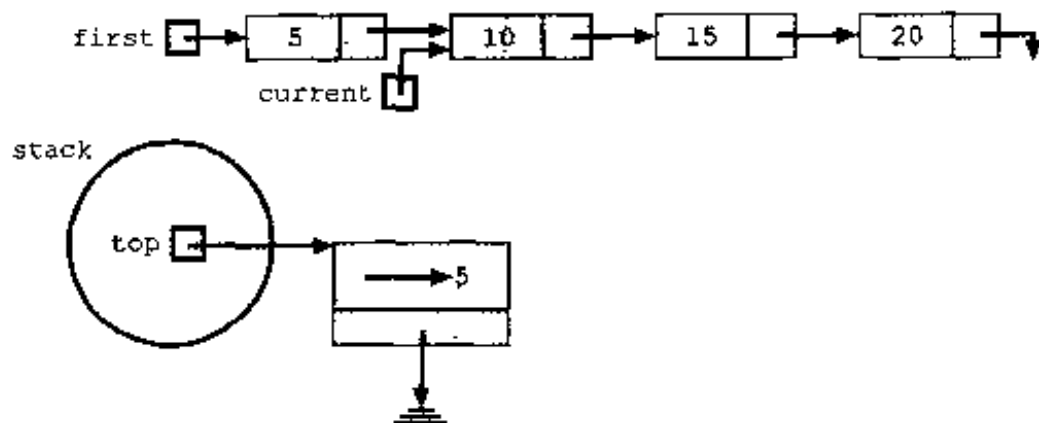


图 18.39 语句“stack.pop(current);”执行之后的链表和栈

Line 7 语句输出 `current->info`, 等于 10。接下来 Line 5 的循环条件开始求值, 它的值为 true, 所以 Line 6 和 Line 7 语句得到执行, Line 6 语句执行之后的结果如图 18.40 所示。

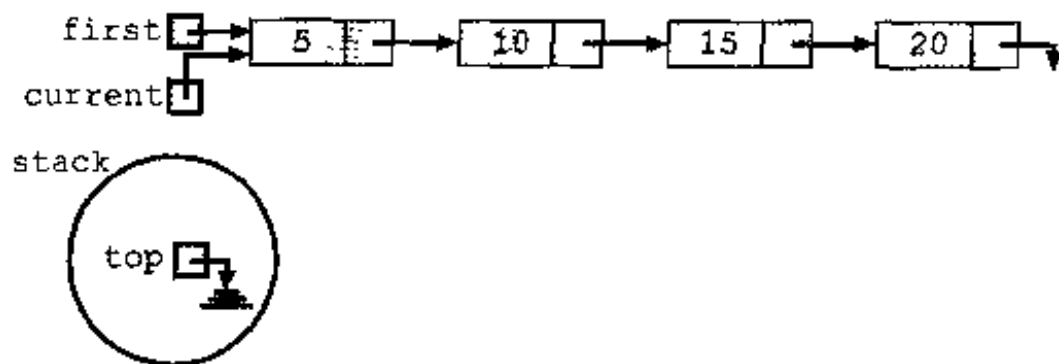


图 18.40 语句“stack.pop(current);”执行之后的链表和栈

Line 7 语句输出 `current->info`, 等于 5。接下来 Line 5 的循环条件开始求值, 它的值为 false, 所以 while 循环终止。Line 5 中的 while 循环产生如下输出:

20 15 10 5

## 18.7 队列

本节讨论另外一种重要的数据结构——队列。计算机科学中队列的概念与日常生活中队列的概念很相似。在银行或杂货店中就存在队列的情景; 在高速公路收费站, 可以看到等待交费的汽车组成的队列。同样, 由于计算机发出打印请求的速度要比打印机的打印速度快得多, 在打印机一端经常会形成由等待打印的文档所组成的队列。处理队列的一般规则是排在前面的顾客最先得到服务, 而新到的顾客则要排到队列的尾部。所以, 队列是一种“先进先出”的数据结构。

队列在计算机科学中有着大量的应用。系统中无论何时使用先进先出的模型, 都会用到队列。在本节的结尾, 将讨论队列的一种最为广泛的应用: 计算机仿真。但首先, 需要实现一个队列。接下来的几节中, 将要讨论如何设计一个类以便实现以 ADT 来描述的队列。

队列是一组相同类型元素的集合, 其中数据从一端添加 (这一端称为队尾), 而从另一端删除 (这一端称为队首)。比如, 考虑在银行中的一队顾客, 这些顾客在队列中等待银行的服务, 如存、取现金, 申请贷款, 资金转账, 等等。每一位新来的顾客, 都需要排到队尾。一旦银行的出纳员准备好为新的顾客服务时, 排在队中最前面的顾客将得到服务。

当要添加新元素时, 总会访问队列的队尾; 而每当要从队列中删除元素时, 总会访问队列的头部。像在栈中一样, 队列中部的元素是不可直接访问的, 即使队列的元素被存储在数组中也是如此。

**队列:** 一种“先进先出”的数据结构, 其中元素都是相同类型的。元素被添加到队列的最后, 称为队列的队尾, 并从队列的另一端删除, 称为队列的头部。

### 18.7.1 队列的操作

从队列的定义中可以看到两种关键的操作是添加和删除。我们称添加操作为 `addQueue()`，称删除操作为 `deQueue()`。因为既不能够从空队列中删除元素，也不能够向满队列中添加元素，所以还需要另外两个操作来成功地实现 `addQueue()` 和 `deQueue()` 操作：`isEmptyQueue()`（检查队列是否为空）和 `isFullQueue()`（检查队列是否为满）。

同样需要一个操作 `initializeQueue()` 将队列初始化为空状态，以及一个 `destroyQueue()` 操作来销毁队列，将其状态变为空。下面描述了队列的一些操作：

1. `initializeQueue()`：将队列初始化为空状态。
2. `destroyQueue()`：删除队列中所有的元素并将其设置为空状态。
3. `isEmptyQueue()`：检查队列是否为空，当队列为空时返回 `true`，否则返回 `false`。
4. `isFullQueue()`：检查队列是否为满，当队列满时返回 `true`，否则返回 `false`。
5. `addQueue()`：将一个新元素添加到队列的尾部。这一操作的输入为一个队列和一个新元素。在进行此操作之前，队列必须存在并且不为满。
6. `deQueue()`：将元素从队列的头部删除并存储在 `deqElement` 中。该函数的输入包含一个队列和一个保存被删除元素的参数。在进行这一操作之前，队列必须存在并且必须不为空。

像栈一样，队列可以存储在数组中，也可以存储在链表结构中。这两种情况下面都会考虑到。由于元素从队列的一端添加，从另一端删除，所以需要两个指针管理队列的头部和尾部，将它们称为 `front` 和 `rear`。

### 18.7.2 队列的数组实现

在给出实现用 ADT 描述的队列类定义之前，需要确定类中需要多少数据成员。当然，需要一个数组来存放队列的元素，变量 `front` 和 `rear` 跟踪队列的第 1 个和最后一个元素，变量 `maxQueueSize` 限定队列的最大数据量。因此，至少需要 4 个数据成员。

在编写用来实现队列中各种操作的算法之前，需要确定如何使用 `front` 和 `rear` 访问队列的元素；`front` 和 `rear` 如何标志队列是满还是空。假设 `front` 给出了队列中第 1 个元素的数组下标，`rear` 给出了队列中最后元素的数组下标。要向队列中添加元素，首先需要将 `rear` 指向下一个数组元素的位置，然后将新元素存放在 `rear` 所指向的位置。要从队列中删除元素，首先需要取出 `front` 所指向的数组元素，然后将 `front` 指向队列中的下一个元素。因此，`front` 的值在每一次 `deQueue()` 操作后都会改变，而 `rear` 的值在每一次 `addQueue()` 操作之后都会改变。

下面看一看当执行一次 `deQueue()` 操作后 `front` 值的变化情况，以及执行一次 `addQueue()` 操作后 `rear` 值的变化情况。假设存放队列的数组有 100 个元素。

开始时，队列为空。在执行操作：

```
addQueue(Queue, 'A');
```

之后，数组如图 18.41 所示。

在执行完另外两次 `addQueue()` 操作后，

```
addQueue(Queue, 'B');
addQueue(Queue, 'C');
```

数组如图 18.42 所示。



图 18.41 第一次执行 addQueue() 操作之后的队列

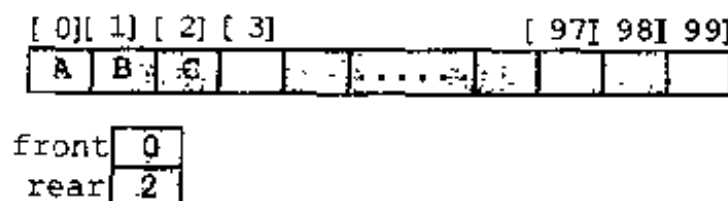


图 18.42 执行过两次 addQueue() 操作之后的队列

现在考虑 deQueue() 操作。

```
deQueue(Queue, deqElement);
```

在此操作之后，存放队列数组的情况如图 18.43 所示。

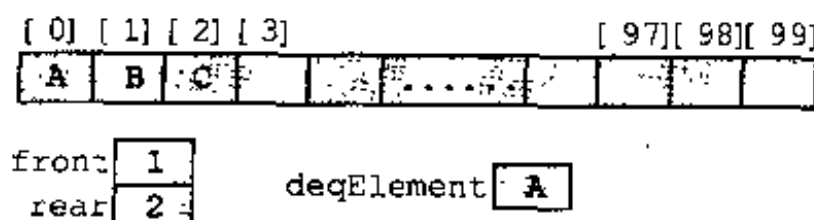


图 18.43 执行 deQueue() 操作之后的队列

这种队列的设计可以正常工作吗？假设 A 代表添加（即 addQueue()）一个元素到队列，D 代表从队列中删除（即 deQueue()）一个元素。考虑下面的操作序列：

AAADADADADADADADA...

操作最终将 rear 的数组下标指向数组中的最后一个元素，造成队列已满的假相。然而，队列中只有两到三个元素并且数组的前部是空的（如图 18.44）。

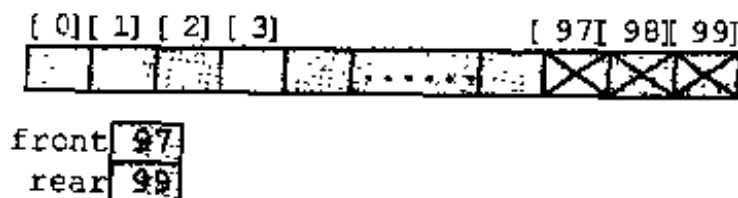


图 18.44 在进行操作序列 AAADADADADADADA... 之后的队列

这一问题的解决方案之一是当队列的 rear 指向数组的最后一个元素时，需要检查 front 的下标，如果 front 的下标显示数组的前部有空间，那么当 rear 指向了数组的尾部时，可以将队列的所有元素向前移动，使得队列的元素以数组的第 1 个元素为起始点。如果队列很小，那么这种解决方案还不错，否则，程序的执行速度会减慢。

另外一种方案是假设数组是循环的——即数组的第 1 个元素的位置紧跟在其最后一个元素之后（如图 18.45 所示）。

下面考虑队列中相应数组为循环数组的情况。

假设有如图 18.46 所示的队列。

在操作：

```
addQueue(Queue, 'Z');
```

之后，队列如图 18.47 所示。

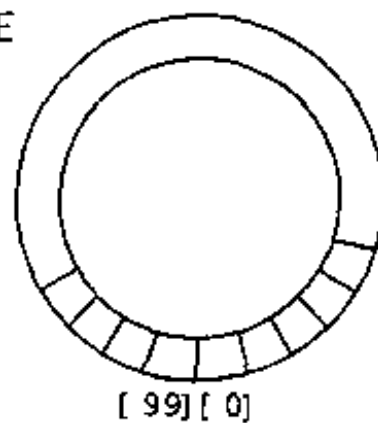


图 18.45 循环数组

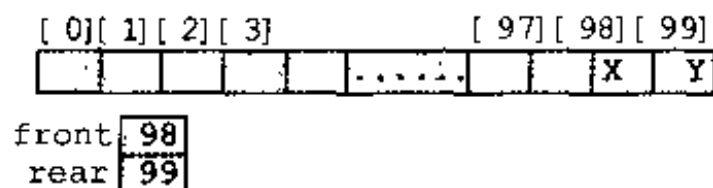


图 18.46 包含两个元素而且下标为 98 和 99 的队列



图 18.47 执行一次 addQueue() 操作之后的队列

因为包含队列的数组是循环数组，可以使用下列语句将 rear (front) 移动到数组的下一个位置：

```
rear = (rear + 1) % maxQueue;
```

如果  $\text{rear} < \text{maxQueue} - 1$ ，那么  $\text{rear} + 1 \leq \text{maxQueue} - 1$ ，所以  $(\text{rear} + 1) \% \text{maxQueue} = \text{rear} + 1$ 。如果  $\text{rear} == \text{maxQueue} - 1$  (即 rear 指向数组的最后一个元素)， $\text{rear} + 1 == \text{maxQueue}$ ，所以  $(\text{rear} + 1) \% \text{maxQueue} == 0$ 。这种情况下，rear 将会被设置为 0，即数组的起始位置。

这种队列的设计似乎工作得很好。在开始编写实现队列操作的算法之前，请考虑以下两种情况：

情况 1：假设在某种操作后，包含队列的数组如图 18.48 所示。



图 18.48 包含一个元素的队列

在操作：

```
deQueue(Queue, deqElement);
```

之后，数组的结果如图 18.49 所示。



图 18.49 deQueue() 操作之后的队列

情况 2：现在考虑如图 18.50 所示的队列。

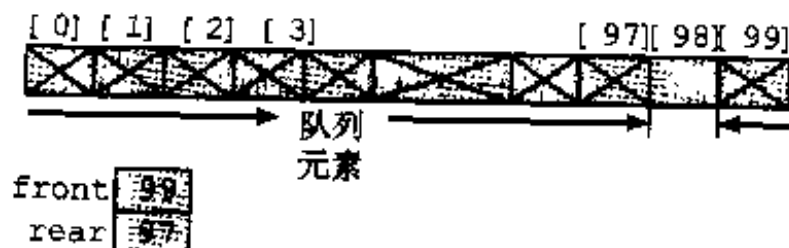


图 18.50 包含 99 个元素的队列

在执行操作：



```
addQueue(Queue, 'Z');
```

数组的结果如图 18.51 所示:

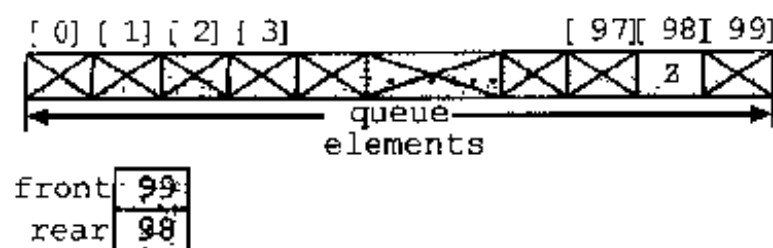


图 18.51 在 addQueue() 操作之后的满队列

图 18.49 和图 18.51 中的数组具有相同的 front 和 rear 值。然而, 图 18.49 所表示的是一个空的队列, 而图 18.51 所表示的是一个满的队列。所以后一种队列设计带来了另外一个问题, 如何区分空队列和满队列。

这一问题有多种解决方法, 其中之一是保存一个计数, 在数据成员 front 和 rear 之外, 需要另外一个变量 count 来实现队列。count 的值在队列添加新元素时递增, 在队列删除元素时递减。这样, 函数 initializeQueue() 和 destroyQueue() 将 count 设置为 0。这种方案在用户需要知道队列的元素数量时十分有用。

另一种解决方法是让 front 表示队列中第 1 个元素之前的元素数组下标, 而非 (真正的) 队列首元素本身。这种情况下, 假设 rear 还是表示队列中最后一个元素的数组下标, 如果 front == rear, 则队列为空。这种方案中, front 所指向的位置 (即队列的第 1 个真实元素之前的位置) 是不被占用的。如果下一个可用的空间是 front 所指向的特殊保留位置, 队列的状态是满的。最后, 由于 front 所指向的位置始终为空, 如果对应数组的大小是 100, 则队列的容量为 99 个元素 (如图 18.52)。

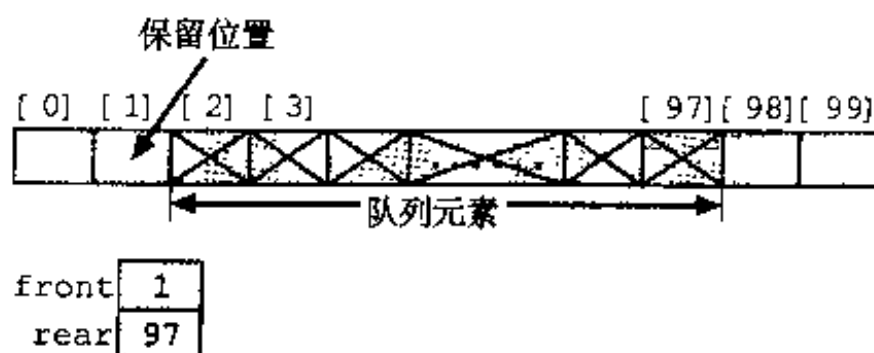


图 18.52 存放带保留位置的队列数组

让我们使用第 1 种解决方案实现队列。即使用变量 count 来标记队列是否为满。

下面的类使用 ADT 将队列进行了定义。由于数组可以动态地分配, 所以将由用户来决定实现队列数组的大小。数组默认大小为 100。

下面只给出队列的定义, 其声明文档留给读者作为练习。类中队列操作的文档与栈类中的相似。

```
template<class Type>
class queueType
{
public:
 const queueType<Type>& operator=(const queueType<Type>&);
 //overload the assignment operator
 void initializeQueue();
 void destroyQueue();
 int isEmptyQueue();
 int isFullQueue();
 void addQueue(Type queueElement);
 void deQueue(Type& deqElement);
```

```

queueType(int queueSize = 100);
queueType(const queueType<Type>& otherQueue);
 //copy constructor
~queueType();
 //destructor

private:
 int maxQueueSize;
 int count;
 int front;
 int rear;
 Type *list; //pointer to the array that holds
 //the queue elements
};

```

接下来看一下实现队列操作的函数定义。

### 初始化队列

首先考虑 `initializeQueue()` 操作。这一操作将队列初始化为空状态。因此, 需要将 `front` 初始化为 0, `rear` 初始化为 `maxQueueSize - 1`, `count` 初始化为 0。参见图 18.53。

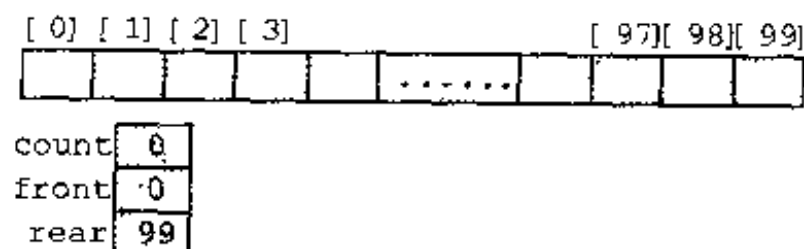


图 18.53 空队列

```

template<class Type>
void queueType<Type>::initializeQueue()
{
 front = 0;
 rear = maxQueueSize - 1;
 count = 0;
}

```

### 销毁队列

`destroyQueue()` 操作删除队列中所有元素并将队列设置为空状态。因为队列存储在数组中, 可以简单地通过设置 `front`、`rear` 和 `count` 的值来删除队列中所有的元素。所以在队列的这种实现中, `destroyQueue()` 操作与 `initializeQueue()` 操作一样。

```

template<class Type>
void queueType<Type>::destroyQueue()
{
 front = 0;
 rear = maxQueueSize - 1;
 count = 0;
}

```

### 判断队列是空还是满

如前面所讨论的, 如果 `count == 0` 则队列为空, 如果 `count == maxQueueSize` 则队列为满。函数的定义为:

```

template<class Type>
int queueType<Type>::isEmptyQueue()
{

```



```

 return(count == 0);
 }

 template<class Type>
 int queueType<Type>::isFullQueue()
 {
 return(count == maxQueueSize);
 }

```

### addQueue()

下面实现 addQueue() 操作。因为 rear 直接指向队列的最后一个元素，所以要向队列添加一个新元素时，需要首先将 rear 移向下一个数组元素的位置，然后将新元素的值存储到 rear 所指向的位置，同时将 count 的值加 1。函数的定义为：

```

template<class Type>
void queueType<Type>::addQueue(Type newElement)
{
 rear = (rear + 1) % maxQueueSize; //use the mod operator to
 //advance rear because
 //the array is circular

 count++;
 list[rear] = newElement;
}

```

由于函数 addQueue() 在向队列中添加元素时并不检查队列是否为满，所以必须在调用函数 addQueue() 之前调用函数 isFullQueue() 检查队列是否为满。假设 queue 是一个 queueType 类型的对象，则对函数 addQueue() 的调用如下所示：

```

if(!queue.isFullQueue())
 queue.addQueue(newElement);

```

### deQueue()

实现 deQueue() 需要访问 front。由于 front 指向队列第 1 个元素，为了将队列的第 1 个元素删除，需要访问 front 所指向的数组元素并将 count 减 1，然后将 front 移向队列的下一个元素。函数 deQueue() 的定义如下：

```

template<class Type>
void queueType<Type>::deQueue(Type& deqElement)
{
 deqElement = list[front];
 count--;
 front = (front + 1) % maxQueueSize; //use the mod operator to
 //advance front because
 //the array is circular
}

```

函数 deQueue() 在删除队列元素时并不检查队列是否为空。因此，必须在调用函数 deQueue() 之前，先调用函数 isEmptyQueue() 来检查队列是否为空。对函数 deQueue() 的调用语句如下：

```

if(!queue.isEmptyQueue())
 queue.deQueue(deqElement);

```

### 构造函数和析构函数

为完整实现队列的各种操作，接下来考虑构造函数和析构函数的实现。构造函数从用户那里得到 maxQueueSize 并将 maxQueueSize 设置为用户指定的值，然后创建大小为 maxQueueSize 的数组。如果用户没有指定队列的大小，构造函数使用默认值将数组的大小设置为 100。构造函数还将初始

化 front 和 rear 的值将队列的状态设置为空。构造函数的定义如下:

```
//constructor
template<class Type>
queueType<Type>::queueType(int queueSize) //constructor
{
 if(queueSize <= 0)
 {
 cout<<"The size of the array to hold the queue must "
 <<"be positive."<<endl;
 cout<<"Creating an array of size 100."<<endl;
 maxQueueSize = 100;
 }
 else
 maxQueueSize = queueSize; //set maxQueueSize to queueSize

 front = 0; //initialize front
 rear = maxQueueSize - 1; //initialize rear
 count = 0;
 list = new Type[maxQueueSize]; //create the array to
 //hold the queue elements
}
```

存放队列元素的数组被动态地创建,因此,当队列对象被销毁时,析构函数只是释放存放队列元素数组所占用的内存。实现析构函数的函数定义如下:

```
template<class Type>
queueType<Type>::~~queueType() //destructor
{
 delete [] list;
}
```

复制构造函数和重载赋值运算符的函数定义留给读者作为练习(这些函数的定义与前面讨论过的链表和栈类似)。

### 18.7.3 队列的链表实现

因为存放队列元素的数组大小是固定的,所以队列只能够存放有限数目的元素。队列的数组实现通过设置 front 和 rear 的值来对数组进行相应操作。队列的链表实现简化了队列数组实现的许多特殊情况,同时由于用于存放队列元素的内存是动态分配的,所以理论上讲,这种实现方式的队列永远不会满。本节讨论队列的链表实现。

因为元素在 rear 一端添加,在另一端(front)删除,所以需要知道队列的头和尾。因此,需要两个指针 front 和 rear 维护一个队列。下面的类定义了用 ADT 描述链表实现的队列。

下面只给出了队列的定义,其具体细节留给读者作为练习。

```
//Definition of the node
template <class Type>
struct nodeType
{
 Type info;
 nodeType<Type> *link;
};

template<class Type>
class linkedQueueType
{
```

```

public:
 const linkedQueueType<Type>& operator=
 (const linkedQueueType<Type>&);
 //overload the assignment operator
 bool isEmptyQueue();
 bool isFullQueue();
 void destroyQueue();
 void initializeQueue();
 void addQueue(const Type& newElement);
 void deQueue(Type& deqElement);
 linkedQueueType(); //default constructor
 linkedQueueType(const linkedQueueType<Type>& otherQueue);
 //copy constructor
 ~linkedQueueType(); //destructor
private:
 nodeType<Type> *front; //pointer to the front of the queue
 nodeType<Type> *rear; //pointer to the rear of the queue
};

```

实现队列各种操作的函数定义很直观。如果 front 是 NULL, 则队列为空。存放队列的内存是动态分配的, 因此队列永远不会满 (除非系统内存耗尽), 所以函数 isFullQueue() 返回 false。

操作 destroyQueue() 删除队列中所有的元素并将队列的状态设置为空。由于存放队列中元素的内存是动态分配的, 所以该操作从队列的第 1 个节点开始逐一释放节点所占用的内存。

下面给出了实现操作 isEmptyQueue()、isFullQueue() 和 destroyQueue() 的函数定义:

```

template<class Type>
bool linkedQueueType<Type>::isEmptyQueue()
{
 return(front == NULL);
}

template<class Type>
bool linkedQueueType<Type>::isFullQueue()
{
 return false;
}

template<class Type>
void linkedQueueType<Type>::destroyQueue()
{
 nodeType<Type> *temp;
 while(front != NULL) //while there are elements left
 //in the queue
 {
 temp = front; //set temp to point to the current node
 front = front->link; //advance first to the next node
 delete temp; //deallocate memory occupied by temp
 }
 rear = NULL; //set rear to NULL
}

```

操作 initializeQueue() 将队列初始化为空状态, 当队列中没有元素时其状态为空。和栈的情形一样, initializeQueue() 操作将队列重新初始化为空状态, 注意, 当队列对象被声明时, 构造函数就将队列初始化了。这一操作必须删除队列中所有的元素, 这一工作可以通过调用函数 destroyQueue() 完成。

```
template<class Type>
void linkedQueueType<Type>::initializeQueue()
{
 destroyQueue();
}
```

### addQueue()和deQueue()操作

addQueue()操作向队列尾部添加一个新的元素,要实现这一操作需要访问指针 rear。同样地,操作 deQueue()将元素从队列删除,需要用到指针 front。实现这些操作的函数定义如下所示:

```
template<class Type>
void linkedQueueType<Type>::addQueue(const Type& newElement)
{
 nodeType<Type> *newNode;

 newNode = new nodeType<Type>; //create the node
 newNode->info = newElement; //store the info
 newNode->link = NULL; //initialize the link field to NULL
 if(front == NULL) //if initially the queue is empty
 {
 front = newNode;
 rear = newNode;
 }
 else //add newNode at the end
 {
 rear->link = newNode;
 rear = rear->link;
 }
} //end addQueue

template<class Type>
void linkedQueueType<Type>::deQueue(Type& deqElement)
{
 nodeType<Type> *temp;

 deqElement = front->info; //copy the info of the first element
 temp = front; //make temp point to the first node
 front = front->link; //advance front to the next node
 delete temp; //delete the first node

 if(front == NULL) //if after deletion the queue is empty,
 rear = NULL; //set rear to NULL
} //end deQueue
```

函数 deQueue()并不检查队列是否为空,所以必须在调用它之前调用函数 isEmptyQueue()进行检查。对函数 deQueue()的调用语句为:

```
if (!queue.isEmptyQueue())
 queue.deQueue(deqElement);
```

默认构造函数的定义与函数 initializeQueue()相似。当队列对象超出作用域范围时,析构函数将队列销毁,即释放所有队列元素占用的内存空间。析构函数的定义与函数 destroyQueue()相似。用来实现复制构造函数和重载赋值运算符的函数定义与栈中相应函数类似,在这里将它作为读者练习。

### 18.7.4 从类 linkedListType 派生队列

从实现队列各种操作的函数定义中可以看出,队列的链表实现与以向前方式创建的链表实现非

常相似 (参见第 17 章)。addQueue() 的实现方式与 insertFirst() 相似, 而 initializeQueue() 则与 initializeList() 相似, isEmptyQueue() 和 isEmptyList() 类似, destroyQueue() 与 destroyList() 相似。deQueue() 操作也可以像前面那样实现。指针 front 与指针 first 相同, 指针 rear 与指针 last 相同。这种对应关系表明, 可以从类 linkedListType (参见第 17 章) 派生出用来实现队列的类。

下面从类 linkedListType 派生出类 linkedQueueType, 并且给出使用为链表定义的操作实现队列的各种操作。

```
//Queue derived from the class linkedListType
//Header file: queueLinked.h
#ifndef H_QueueType
#define H_QueueType

#include <iostream>
#include "linkedList.h"

using namespace std;

template<class Type>
class linkedQueueType: public linkedListType<Type>
{
public:
 bool isEmptyQueue();
 bool isFullQueue();
 void destroyQueue();
 void initializeQueue();
 void addQueue(const Type& newElement);
 void deqQueue(Type& deqElement);
};

template<class Type>
void linkedQueueType<Type>::initializeQueue()
{
 linkedListType<Type>::initializeList();
}

template<class Type>
void linkedQueueType<Type>::destroyQueue()
{
 linkedListType<Type>::destroyList();
}

template<class Type>
bool linkedQueueType<Type>::isEmptyQueue()
{
 return linkedListType<Type>::isEmptyList();
}

template<class Type>
bool linkedQueueType<Type>::isFullQueue()
{
 return linkedListType<Type>::isFullList();
}

template<class Type>
void linkedQueueType<Type>::addQueue(const Type& newElement)
{
 linkedListType<Type>::insertLast(newElement);
}

template<class Type>
```

```

void linkedQueueType<Type>::deqQueue(Type& deqElement)
{
 nodeType<Type> *temp;

 deqElement = first->info; //copy the info of the first element
 temp = first; //make temp point to the first node
 first = first->link; //advance front to the next node
 delete temp; //delete the first node
 if(first == NULL) //if after deletion the queue is empty,
 last = NULL; //set last to NULL
}

#endif

```

### 例 18.5

下面的程序测试了队列中的各种操作，它使用从类 `linkedListType` 派生出的链表版本。

```

//Program to test the queue operations: QueueTest.cpp
#include <iostream>
#include "linkedList.h"
#include "queueLinked.h"

using namespace std;

int main()
{
 linkedQueueType<int> queue;
 linkedQueueType<int> copyQueue;

 int num;

 cout<<"Queue Operations"<<endl;
 cout<<"Enter numbers ending with -999"<<endl;
 cin>>num;

 while(num != -999)
 {
 queue.addQueue(num); //add an element to the queue
 cin>>num;
 }

 copyQueue = queue; //copy the queue into copyQueue
 cout<<"Queue contains: ";
 while(!copyQueue.isEmptyQueue())
 {
 copyQueue.deqQueue(num); //remove an element from
 //the queue
 cout<<num<<" ";
 }
 cout<<endl;
 return 0;
}

```

**运行实例：**在本运行实例中，用户输入用阴影表示。

```

Queue Operations
Enter numbers ending with -999
23 76 64 56 28 91 21 11 82 -999
Queue contains: 23 76 64 56 28 91 21 11 82

```



## 18.8 队列的应用：仿真

使用一种系统模型来模拟另一种系统的行为称为仿真。例如，物理仿真器包括风洞，用来辅助汽车车身的设计；飞行仿真器用来培训飞行员，等等。仿真技术也用于设计研究真实系统行为的计算机模型（稍后将会讨论一些真实系统的计算机模型）。使用计算机模型来模拟十分危险或者非常昂贵的真实系统，通常要比使用真实系统安全和便宜得多，而且可以在不危及研究人员人身安全的情况下研究系统的内部情况。而且，计算机仿真在研究十分复杂的、很难建立数学模型的系统时特别有用。对于这样的系统，计算机模拟可以保证描述的精度。在数学仿真中，程序中的各个函数或过程用来模仿真实系统的行为。

假设一家电影院的经理正在听取顾客对于购票缓慢问题的投诉。目前，电影院只有一位收银员。有人准备在其附近再开一家电影院，经理担心会流失顾客。这位经理想要雇佣足够的收银员以便顾客不会为买票等待太长的时间，同时又不想因为雇佣更多的收银员从而导致潜在的时间和金钱上的浪费。经理想要知道的是，顾客在买票时需要等待多长时间，因此这位经理想请人编写一个程序来模拟电影院的情形。

在计算机模拟中，所要研究的对象通常用数据来代表。对于电影院问题，对象是顾客和收银员，收银员为顾客服务。需要确定顾客要等待的平均时间并通过使用编程语言，在函数的帮助下实现算法，模拟研究对象的动作。因此，函数用来实现对象的行为。在 C++ 类的帮助下，可以将数据以及作用于数据之上的操作结合为一个单元。因此，研究对象可以由类代表。类的数据成员描述了研究对象的各种属性，函数成员描述了数据上的行为。如果改变数据值或者改变函数的定义（即改变用来实现行为的算法），将会导致仿真结果的变化。计算机仿真的主要目的是产生能够反映真实系统性能的结果，以及对某一既定系统的表现进行预测。

在电影院问题中，当收银员为一位顾客服务时，其他顾客必须等候。因为排在最前面的顾客最先被服务，而队列是实现“先进先出”系统的有效方法，所以队列是模拟电影院问题的重要数据结构。本节将研究以队列为基本数据结构的计算机仿真，并用这些仿真模仿系统（称为队列系统）的行为。在这一系统中，队列的对象等待各种服务者的服务，换句话说，一个队列系统包含服务者和等待服务的对象队列。在日常生活中经常遇到各种队列系统，例如，杂货店和银行都是队列系统。而且当用户向一个共享的网络打印机发送一个打印请求时，打印请求就会进入一个队列。之前提交的打印请求通常会先于后续请求。因此，当文档等待打印时，打印机就是一个服务者。

接下来讨论一种可以应用于多种场合的队列系统，例如银行、杂货店、电影院、打印机或者大型机环境，其上有多个人使用相同的处理器来运行各自的程序。要描述一个队列系统，可以使用术语“服务者”代表为对象提供服务的实体。例如，银行中，银行柜员就是一个服务者，在杂货店或者电影院收银员就是服务者。将接受服务的对象称为“顾客”，并称服务者为顾客提供服务所需要的时间为“处理时间”。

由于队列系统包括服务者和等待顾客，下面将模拟包含多个服务者和需要服务的顾客等待队列。排在队列最前面的顾客等待可用的服务者服务。当某一服务者空闲下来，排在队列最前面的顾客移动到该空闲服务者并得到服务。

当第一位顾客到达而且所有的服务者都处于空闲状态时，顾客移动到第一个服务者；当下一个顾客到达时，如果某一服务者可用，则顾客立即移动到可用的服务者那里；否则，顾客在队列中等待。为了模拟一个队列系统，需要知道服务者的数量、顾客到达的时间间隔和影响系统的事件数量。

下面再来研究一下电影院系统。假设服务者的数量为 1，平均处理一位顾客的时间是 6 分钟，

而且平均每4分钟到达一位新的顾客。系统的性能依赖于系统中有多少服务者、服务一位顾客所需要的时间和顾客到达的频率。如果为一位顾客服务的时间过长而顾客到达的频率高,则需要更多的服务者。这样的系统可以使用时间驱动仿真进行模拟。在时间驱动仿真中,时钟被实现为计数器,时间推移一分钟计数器加1。仿真程序会运行一段固定的时间,如果仿真程序需要运行100分钟,计数器从1开始到100截止,这可以通过循环来实现。

在本节描述的仿真程序中,需要得到顾客的平均等待时间。要计算一位顾客的平均等待时间,需要累加每一位顾客的等待时间,然后将累计时间除以到达的顾客数量。当一位顾客到达时,将排到队列的队尾,同时等待时间计时开始。如果队列为空并且服务者空闲,则到达的顾客会立即得到服务,所以等待时间为0。另一方面,如果当顾客到达时,等待队列不为空而且服务者处于繁忙状态,则顾客必须等待下一个可用的服务者,因此,该顾客的等待时间开始计数。可以为每一位顾客使用一个计时器,当顾客到达时,其定时器设置为0,并且每经过一个时钟单元递增1。

假设平均一位顾客的处理时间为5分钟,当服务者空闲而且顾客的等待队列非空,则在等待队列最前端的顾客开始得到处理。因此必须监控顾客接受服务的时间,当顾客到达服务者,处理时间就被设置为5,然后每隔一个时间单位减1。当处理时间变为0,服务者就变为空闲。因此,时间仿真系统中的顾客和服务者两个对象需要实现为时间驱动。

接下来,在设计主要算法和实现仿真器之前,先来设计实现两个对象(顾客和服务者)的类。

### 18.8.1 顾客

每一位顾客有一个编号、到达时间、等待时间、处理时间和离开时间。如果知道到达时间,可以将到达时间、等待时间和处理时间相加而得到离开时间。将实现顾客对象的类命名为customerType。从而,类customerType有4个数据成员:customerNumber、arrivalTime、waitingTime和transactionTime。它们每一个都是int型的。customerType类型对象中的基本操作为:设置顾客的编号、到达时间和等待时间;每经过一个时间单位递增等待时间;返回等待时间;返回到达时间;返回处理时间;返回顾客编号。下面的类customerType实现了顾客的ADT(参见图18.54):

```
class customerType
{
public:
 customerType(int cN = 0, int arrvTime = 0, int wTime = 0,
 int tTime = 0);
 //constructor to initialize the data members
 //according to the parameters
 //In the object declaration if no value is specified,
 //the default is assigned
 //Post: customerNumber = cN;
 // arrivalTime.clock = arrvTime;
 // waitingTime.clock = wTime;
 // transactionTime.clock = tTime;
 void setCustomerInfo(int customerN = 0, int inTime = 0,
 int wTime = 0, int tTime = 0);
 //Data members are set according to the parameters
 //Post: customerNumber = customerN;
 // arrivalTime.clock = arrvTime;
 // waitingTime.clock = wTime;
 // transactionTime.clock = tTime;
 int getWaitingTime() const;
 //Return the value of the data member waitingTime
 void setWaitingTime(int time);
```





```

{
 customerNumber = customerN;
 arrivalTime = inTime;
 waitingTime = wTime;
 transactionTime = tTime;
}

```

函数 `getWaitingTime()` 返回当前等待的时间。函数 `getWaitingTime()` 的定义如下:

```

int customerType::getWaitingTime() const
{
 return waitingTime;
}

```

函数 `incrementWaitingTime()` 将变量 `waitingTime` 的值加 1, 其定义如下:

```

void customerType::incrementWaitingTime()
{
 waitingTime++;
}

```

函数 `setWaitingTime()`、`getArrivalTime()` 和 `getTransactionTime()` 的定义与函数 `incrementWaitingTime()` 相似:

```

void customerType::setWaitingTime(int time)
{
 waitingTime = time;
}

int customerType::getArrivalTime()
{
 return arrivalTime;
}

int customerType::getTransactionTime()
{
 return transactionTime;
}

```

函数 `getCustomerNumber()` 返回顾客的编号, 即返回数据成员 `customerNumber` 的值。

```

int customerType::getCustomerNumber()
{
 return customerNumber;
}

```

### 18.8.2 服务者

在任何时间内, 服务者的状态不是为顾客服务就是空闲。可以使用一个字符串变量设置服务者的状态。每一个服务者都有一个定时器, 同时, 由于程序或许需要知道哪一个服务者正在为哪位顾客提供服务, 因此服务者也需要存有其正在提供服务的顾客信息。因此, 与服务者相关的三个数据成员为: `status`、`transactionTime` 和 `currentCustomer`。服务者必要的基本操作包括: 检查服务者是否为空闲; 将服务者设置为空闲; 将服务者设置为忙; 设置处理时间 (即为某一顾客提供服务所花费的时间); 返回剩余处理时间 (以决定是否将服务者设定为空闲); 如果服务者忙, 则每经过一个时间单位就将处理时间减少一个单位, 依次类推。下面的类 `serverType` 实现了服务者的 ADT (参见图 18.55):

```

class serverType
{
public:
 serverType();
 //default constructor
 //Sets the values of the data members to their
 //default values
 //Post: currentCustomer is initialized by its
 // default constructor
 // status = "free"
 // transaction time is initialized to 0
 bool isFree() const;
 //Returns true if the server is free; false otherwise
 void setBusy();
 //Set the status of the server to busy
 //Post: status = "busy";
 void setFree();
 //Set the status of the server to free
 //Post: status = "free";
 void setTransactionTime(int t);
 //Set the transaction time according to the parameter t
 //Post: transactionTime = t;
 void setTransactionTime();
 //Set the transaction time according to the current
 //customer's transaction time
 //Post: transactionTime = currentCustomer.transactionTime;
 int getRemainingTransactionTime();
 //Returns the remaining transaction time value of the
 //data member transactionTime is returned
 void decreaseTransactionTime();
 //Decrease the transactionTime by 1
 //Post: transactionTime--;
 void setCurrentCustomer(customerType cCustomer);
 //Set the info of the current customer according to the
 //parameter cCustomer
 //Post: currentCustomer = cCustomer;
 int getCurrentCustomerNumber();
 //Returns the customer number of the current customer
 int getCurrentCustomerArrivalTime();
 //Returns the arrival time of the current customer
 //The value of the data member arrivalTime of
 //currentCustomer is returned
 int getCurrentCustomerWaitingTime();
 //Returns the current waiting time of the current customer
 //The value of the data member transactionTime is returned
 int getCurrentCustomerTransactionTime();
 //Returns the current customer's transaction time
 //The value of the data member clock of transactionTime of
 //the current customer is returned

private:
 customerType currentCustomer;
 string status;
 int transactionTime;
};

```

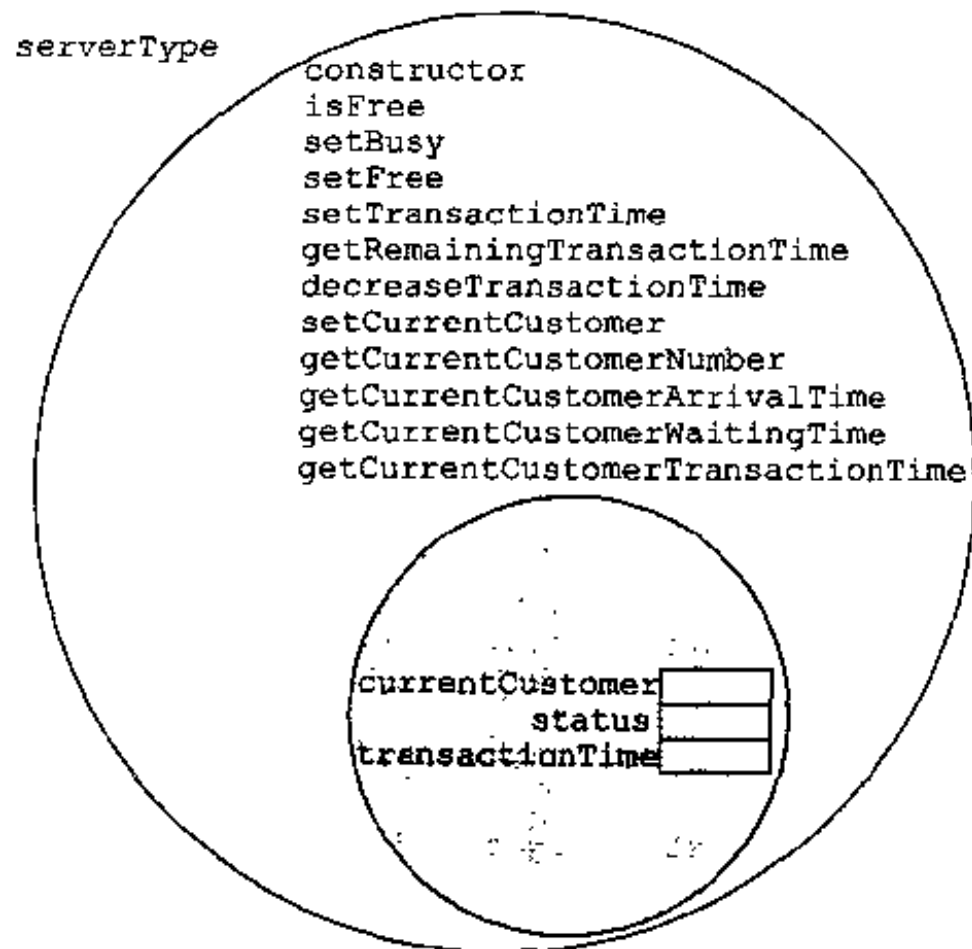


图 18.55 类 serverType

类 serverType 的成员函数定义如下所示:

```

serverType::serverType()
{
 status = "free";
 transactionTime = 0;
}

bool serverType::isFree() const
{
 return (status == "free");
}

void serverType::setBusy()
{
 status = "busy";
}

void serverType::setFree()
{
 status = "free";
}

void serverType::setTransactionTime(int t)
{
 transactionTime = t;
}

void serverType::setTransactionTime()
{
 int time;
 time = currentCustomer.getTransactionTime();
 transactionTime = time;
}

void serverType::decreaseTransactionTime()
{

```

```

 transactionTime--;
 }
 int serverType::getRemainingTransactionTime()
 {
 return transactionTime;
 }
 void serverType::setCurrentCustomer(customerType cCustomer)
 {
 currentCustomer = cCustomer;
 }
 int serverType::getCurrentCustomerNumber()
 {
 return currentCustomer.getCustomerNumber();
 }
 int serverType::getCurrentCustomerArrivalTime()
 {
 return currentCustomer.getArrivalTime();
 }
 int serverType::getCurrentCustomerWaitingTime()
 {
 return currentCustomer.getWaitingTime();
 }
 int serverType::getCurrentCustomerTransactionTime()
 {
 return currentCustomer.getTransactionTime();
 }
}

```

因为正在设计的是一种可以用于多种场合的通用仿真器,因此还需要另外设计两个类:一个类用来创建和处理服务者表,另外一个类用来创建和处理顾客的等待队列,接下来的两节中描述了这两个类。

### 18.8.3 服务者表

服务者表是一个服务者的集合。任何时间内,服务者的状态不是空闲就是忙。对于排在队列最前端的顾客,需要在服务者表中为其找到空闲的服务者。如果所有的服务者都忙,则顾客必须等待直到有一个服务者变为空闲。因此实现服务者表的类包含两个数据成员:一个存储服务者的编号,另外一个维护了服务者的链表。在程序执行的过程中,按照用户所指定的数目,使用动态数组创建服务者表。服务者表必需的基本操作有:返回一个空闲服务者的编号;当服务者处于可用状态并且顾客准备好进行处理时,将服务者设置为忙状态;当模拟结束时,可能还有一些处于忙状态的服务者,所以需要返回处于忙状态的服务者编号;每过一个时间单位,将每个处于忙状态服务者的 transactionTime 值减小一个单位;如果服务者的 transactionTime 值变为 0,将服务者设置为空闲。下面的类 serverListType 实现了服务者表的 ADT (参见图 18.56):

```

class serverListType
{
public:
 serverListType(int num = 1);
 //constructor to initialize a list of servers
 //Post: numServers = num
 // A list of servers, specified by num, is
 // created and each server is initialized to free.

```

```

~serverListType();
 //destructor
 //Post: the list of servers is destroyed.
int getFreeServerID();
 //Search the list of servers.
 //If a free server is found, return its ID;
 //otherwise, return -1.
int getNumberOfBusyServers();
 //Returns the number of busy servers
void setServerBusy(int serverID, customerType cCustomer,
 int tTime);
 //Set the server specified by serverID to busy.
 //To serve the customer specified by cCustomer, the
 //transaction time is set according to the
 //parameter tTime.
void setServerBusy(int serverID, customerType cCustomer);
 //Set the server specified by serverID to busy.
 //To serve the customer specified by cCustomer, the
 //transaction time is set according to the
 //customer's transaction time
void updateServers();
 //The transaction time of each busy server is decremented
 //by one unit. If the transaction time of a busy server
 //is reduced to zero, the server is set to free and a
 //message indicating which customer is served, together
 //with the customer's departing time, is printed on
 //the screen.
void updateServers(ofstream& outFile);
 //The transaction time of each busy server is decremented
 //by one unit. If the transaction time of a busy server
 //is reduced to zero, the server is set to free and a
 //message indicating which customer is served, together
 //with the customer's departing time, is sent to a file.
private:
 int numOfServers;
 serverType *servers;
};

```

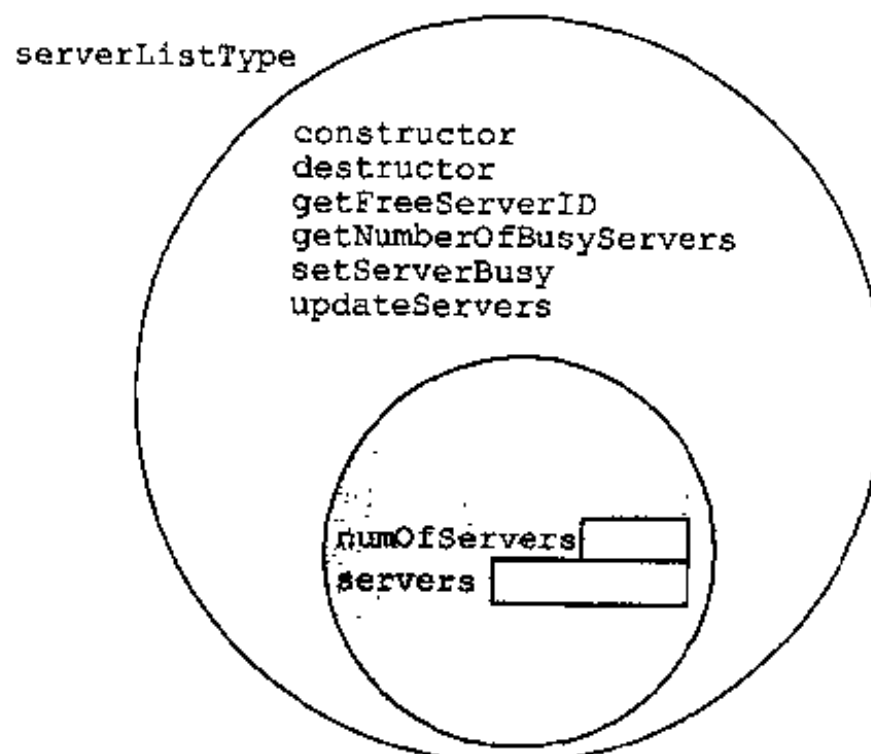


图 18.56 类 serverListType



下面是类 `serverListType` 的成员函数定义。

构造函数和析构函数的定义如下所示：

```
serverListType::serverListType(int num)
{
 numOfServers = num;
 servers = new serverType[num]; //create a list of servers
 for(int i = 0; i < num; i++)
 servers[i].setFree(); //initialize each server to free
}

serverListType::~serverListType()
{
 delete [] servers;
}
```

函数 `getFreeServerID()` 搜索服务者列表，如果找到某一空闲服务者，就返回该服务者的 ID，否则返回 -1，表示所有服务者列表的服务者都处于忙状态。函数的定义如下：

```
int serverListType::getFreeServerID()
{
 int serverID = -1;
 int i;
 for(i = 0; i < numOfServers; i++)
 if(servers[i].isFree())
 {
 serverID = i;
 break;
 }
 return serverID;
}
```

函数 `getNumberOfBusyServers()` 搜索服务者列表，确定并返回处于繁忙状态的服务者数目。其定义如下：

```
int serverListType::getNumberOfBusyServers()
{
 int busyServers = 0;
 int i;
 for(i = 0; i < numOfServers; i++)
 if(!servers[i].isFree())
 busyServers++;
 return busyServers;
}
```

函数 `setServerBusy()` 将服务者的状态设置为忙，该函数被重载。服务者的 `serverID` 通过参数传递给函数。一个函数根据参数 `tTime` 来设置服务者的处理时间，另一个函数通过存储在对象 `cCustomer` 内的处理时间来进行设置。这些函数的定义为：

```
void serverListType::setServerBusy(int serverID,
 customerType cCustomer,
 int tTime)
{
 servers[serverID].setBusy();
}
```





```

 <<" clock unit "
 <<servers[i].getCurrentCustomerArrivalTime()
 + servers[i].getCurrentCustomerWaitingTime()
 + servers[i].getCurrentCustomerTransactionTime()
 <<endl;
 servers[i].setFree();
 }
}

```

#### 18.8.4 顾客等待队列

当某位顾客到达时,将排到队列的尾部。当某一服务者变为可用时,排在队列最前面的顾客离开队列并得到处理。在每一个时间单位过后,等待队列中每一位顾客的等待时间都被加 1。本章设计的 ADT queueType 拥有除了将顾客的等待时间增加一个单位之外的所有操作。下面将从类 queueType 中派生出一个 waitingCustomerQueueType 类,并加入了实现顾客队列所需的额外操作。类 waitingCustomerQueueType 的定义如下所示:

```

class waitingCustomerQueueType: public queueType<customerType>
{
public:
 waitingCustomerQueueType(int size = 100);
 //The queue is initialized according to the
 //parameter size
 //The value of the size is passed to the constructor
 //of queueType
 ~waitingCustomerQueueType();
 //The queue is destroyed
 void updateWaitingQueue();
 //Increment the waiting time of each customer in the
 //queue by one time unit
};

```

下面给出了成员函数的定义。

构造函数和析构函数的定义如下:

```

waitingCustomerQueueType::waitingCustomerQueueType(int size)
 :queueType<customerType>(size)
{
}

waitingCustomerQueueType::~~waitingCustomerQueueType()
{
}

```

函数 updateWaitingQueue() 将等待队列中每一位顾客的等待时间增加一个单位。类 waitingCustomerQueueType 是从类 queueType 派生的。因为 queueType 的数据成员是 private 类型,函数 updateWaitingQueue() 不能够直接访问队列的元素。访问队列元素的惟一方法是使用 deQueue() 操作。在增加了等待时间后,元素可以使用 addQueue() 操作添加至队列。

addQueue() 操作向队列尾部添加元素,如果对队列的每一个元素分别进行 deQueue() 和 addQueue() 操作,则最终原队列首元素不会改变。每一个 deQueue() 操作都跟有 addQueue() 操作,如何确定队列中的所有元素是否都被处理过? 不能够在队列上使用 isEmptyQueue() 或者 isFullQueue() 操作,因为队列永远也不会空或者满

一种解决方法是创建一个临时队列,原队列中的每一个元素被删除、处理并插入到临时队列中。

原  
书  
缺  
页

原  
书  
缺  
页

原  
书  
缺  
页

原  
书  
缺  
页

```
Server No: 2 Customer number 9 departed at
 clock unit 48
Customer number 11 arrived at time unit 49
Customer number 12 arrived at time unit 51
Server No: 1 Customer number 10 departed at
 clock unit 52
Customer number 13 arrived at time unit 52
Customer number 14 arrived at time unit 53
Server No: 2 Customer number 11 departed at
 clock unit 54
Customer number 15 arrived at time unit 54
Server No: 1 Customer number 12 departed at
 clock unit 57
Server No: 2 Customer number 13 departed at
 clock unit 59
Customer number 16 arrived at time unit 59
Server No: 1 Customer number 14 departed at
 clock unit 62
Server No: 2 Customer number 15 departed at
 clock unit 64
Customer number 17 arrived at time unit 66
Server No: 1 Customer number 16 departed at
 clock unit 67
Server No: 2 Customer number 17 departed at
 clock unit 71
Customer number 18 arrived at time unit 71
Server No: 1 Customer number 18 departed at
 clock unit 76
Customer number 19 arrived at time unit 78
Server No: 1 Customer number 19 departed at
 clock unit 83
Customer number 20 arrived at time unit 90
Customer number 21 arrived at time unit 92
Server No: 1 Customer number 20 departed at
 clock unit 95
Server No: 2 Customer number 21 departed at
 clock unit 97

Simulation ran for 100 time units
Number of servers: 2
Average transaction time: 5
Average arrival time difference between customers: 4
Total wait time: 20
Number of customers completed transaction: 21
Number of customers left in servers: 0
Customers left in queue: 0
Average wait time: 0.95
***** END SIMULATION *****
```

### 运行实例 3: (省略了顾客的到达和离开时间。)

```
Customer number 1 arrived at time unit 4
Customer number 2 arrived at time unit 8
Server No: 1 Customer number 1 departed at
 clock unit 9
Customer number 3 arrived at time unit 9
Customer number 4 arrived at time unit 12
Server No: 1 Customer number 2 departed at
 clock unit 14
```



```
Server No: 1 Customer number 3 departed at
 clock unit 19
Customer number 5 arrived at time unit 21
Server No: 1 Customer number 4 departed at
 clock unit 24
Server No: 1 Customer number 5 departed at
 clock unit 29
Customer number 6 arrived at time unit 37
Customer number 7 arrived at time unit 38
Customer number 8 arrived at time unit 41
Server No: 1 Customer number 6 departed at
 clock unit 42
Customer number 9 arrived at time unit 43
Customer number 10 arrived at time unit 44
...

Simulation ran for 1000 time units
Number of servers: 1
Average transaction time: 5
Average arrival time difference between customers: 4
Total wait time: 8008
Number of customers completed transaction: 197
Number of customers left in servers: 1
Customers left in queue: 15
Average wait time: 37.60
***** END SIMULATION *****
```

#### 运行实例4: (省略了顾客的到达和离开时间。)

```
Customer number 1 arrived at time unit 4
Customer number 2 arrived at time unit 8
Server No: 1 Customer number 1 departed at
 clock unit 9
Customer number 3 arrived at time unit 9
Customer number 4 arrived at time unit 12
Server No: 2 Customer number 2 departed at
 clock unit 13
Server No: 1 Customer number 3 departed at
 clock unit 14
Server No: 3 Customer number 4 departed at
 clock unit 17
Customer number 5 arrived at time unit 21
Server No: 1 Customer number 5 departed at
 clock unit 26
Customer number 6 arrived at time unit 37
Customer number 7 arrived at time unit 38
Customer number 8 arrived at time unit 41
Server No: 1 Customer number 6 departed at
 clock unit 42
Server No: 2 Customer number 7 departed at
 clock unit 43
Customer number 9 arrived at time unit 43
Customer number 10 arrived at time unit 44
Server No: 3 Customer number 8 departed at
 clock unit 46
...

Simulation ran for 1000 time units
Number of servers: 3
```

```
Average transaction time: 5
Average arrival time difference between customers: 4
Total wait time: 13
Number of customers completed transaction: 212
Number of customers left in servers: 1
Customers left in queue: 0
Average wait time: 0.06
***** END SIMULATION *****
```

## 18.9 小结

1. 栈是一种数据结构，其中元素只在一端进行添加和删除。
2. 栈是一种“后进先出”（LIFO）的数据结构。
3. 栈上的基本操作有：将元素压入栈、从栈中弹出元素、初始化栈、销毁栈、检查栈是否为空以及检查栈是否为满。
4. 栈既可以使用数组实现，也可以使用链表实现。
5. 栈中部的数据不能够被直接访问。
6. 要使用栈顶元素，必须先将它从栈中弹出（删除）。
7. 栈是数组和链表的约束版本。
8. 后缀符号不需要使用括号来强调优先级。
9. 在后缀表达式中，运算符写在操作数的后边。
10. 后缀表达式依照下列规则进行求值：
  - a. 从左向右扫描表达式。
  - b. 如果找到运算符，向前找到所需的操作数并求值。
11. 队列是一种数据结构，其中元素从一端添加，从另一端删除。
12. 队列是一种“先进先出”（FIFO）的数据类型。
13. 队列上的基本操作有：向队列添加元素、从队列中删除元素、初始化队列、销毁队列、检查队列是否为空以及检查队列是否为满。
14. 队列既可以使用数组实现，也可以使用链表实现。
15. 队列中部的数据不能够被直接访问。
16. 要使用队首元素，必须先将它从队列中删除。
17. 队列是数组和链表的约束版本。

## 18.10 练习

1. 针对下列语句：

```
stackType<int> stack;
int x, y;
写出下段代码的输出。
stack.initializeStack();

x = 4;
y = 6;
stack.push(7);
stack.push(x);
stack.push(x + 5);
stack.pop(y);
```



```

stack.push(x + y);
stack.push(y - 2);
stack.push(3);
stack.pop(x);
cout<<"x = "<<x<<endl;
cout<<"y = "<<y<<endl;
while(!stack.isEmptyStack())
{
 stack.pop(y);
 cout<<y<<endl;
}

```

2. 针对下列语句:

```

stackType<int> stack;
int x, y;

```

假设输入是:

14 45 34 23 10 5 -999

写出下段代码的输出。

```

stack.initializeStack();

stack.push(5);
cin>>x;
while (x != -999)
{
 if(x % 2 == 0)
 {
 if(!stack.fullStack())
 stack.push(x);
 }
 else
 cout<<"x = "<<x<<endl;
 cin>>x;
}

cout<<"Stack Elements: ";
while(!stack.isEmptyStack())
{
 stack.pop(y);
 cout<<" "<<y;
}
cout<<endl;

```

3. 请计算下列后缀表达式的值。

a.  $8\ 2\ +\ 3\ *\ 16\ 4\ /\ -\ =$

b.  $12\ 25\ 5\ 1\ /\ /\ *\ 8\ 7\ +\ -\ =$

c.  $70\ 14\ 4\ 5\ 15\ 3\ /\ *\ -\ -\ /\ 6\ +\ =$

d.  $3\ 5\ 6\ *\ +\ 13\ -\ 18\ 2\ /\ +\ =$

4. 请将下列中缀表达式转换为后缀表达式。

a.  $(A + B) * (C + D) - E$

b.  $A - (B + C) * D + E / F$

c.  $((A + B) / (C - D) + E) * F - G$

d.  $A + B * (C + D) - E / F * G + H$

5. 针对下列语句:

```

stackType<int> stack;
queueType<int> queue;
int x, y;
给出下列代码的输出。
queue.initializeQueue();
x = 4;
y = 5;
queue.addQueue(x);
queue.addQueue(y);
queue.deQueue(x);
queue.addQueue(x + 5);
queue.addQueue(16);
queue.addQueue(x);
queue.addQueue(y - 3);

cout<<"Queue Elements: ";
while(!queue.isEmptyQueue())
{
 queue.deQueue(y);
 cout<<" "<<y;
}

```

6. 针对下列语句:

```

stackType<int> stack;
queueType<int> queue;
int x, y;

```

假设输入是:

15 28 14 22 64 35 19 32 7 11 13 30 -999

给出下列代码的输出。

```

stack.initializeStack();
queue.initializeQueue();
stack.push(0);
queue.addQueue(0);
cin>>x;
while(x != -999)
{
 switch(x % 4)
 {
 case 0: stack.push(x);
 break;
 case 1: if(!stack.isEmptyStack())
 {
 stack.pop(y);
 cout<<"Stack Element = "<<y<<endl;
 }
 else
 cout<<"Sorry stack is empty"<<endl;
 break;
 case 2: queue.addQueue(x);
 break;
 case 3: if(!queue.isEmptyQueue())
 {
 queue.deQueue(y);
 cout<<"Queue Element = "<<y<<endl;
 }
 else

```

```

 cout<<"Sorry queue is empty"<<endl;
 break;
 } //end switch

 cin>>x;
} //end while

cout<<"Stack Elements: ";
while(!stack.isEmptyStack())
{
 stack.pop(x);
 cout<<x<<" ";
}

cout<<endl;
cout<<"Queue Elements: ";
while(!queue.isEmptyQueue())
{
 queue.deQueue(x);
 cout<<x<<" ";
}

cout<<endl;

```

7. 编写函数模板 `reverseStack()`，分别以栈对象和队列对象为参数，参数类型相同。函数 `reverseStack()` 使用队列将栈中的元素排列顺序反转。
8. 编写函数模板 `reverseQueue()`，分别以栈对象和队列对象为参数，参数类型相同。函数 `reverseQueue()` 使用栈将队列中的元素排列顺序反转。
9. 编写函数模板 `printListReverse()`，使用栈来逆序打印链表中的元素。假设函数是第 17 章中类 `linkedListType` 的一个成员函数。
10. 向类 `queueType` 中添加一个操作（队列的数组实现），返回队列中元素的数量。写出实现这一操作的函数模板定义。

## 18.11 编程练习

1. 如果两个栈有相同的元素数量而且对应元素分别相等，则这两个栈相等。针对类 `stackType` 重载运算符 `==`，如果两个栈相等返回 `true`；否则返回 `false`。写出重载这一运算符的函数模板定义。
2. 对类 `linkedStackType` 重写编程练习 1。
3. a. 向类 `StackType` 中添加操作：
 

```
void reverseStack(stackType<Type> &otherStack);
```

 该操作将栈中的元素以相反的顺序复制给另外一个栈。  
 考虑下面的语句：
 

```
stackType<int> stack1;
stackType<int> stack2;
```

 语句：
 

```
stack1.reverseStack(stack2);
```

 将 `stack1` 中的元素逆序复制给 `stack2`。即 `stack1` 中的栈顶元素是 `stack2` 中的栈底元素，依次类推。`stack2` 中原来的内容被销毁，`stack1` 保持不变。
- b. 写出实现操作 `reverseStack()` 的函数模板定义。
4. 对类 `linkedStackType` 重写编程练习 3a 和 3b。

5. 编写一个程序输出匹配（成对出现）符号的信息，比如，小括号和花括号。例如，表达式  $\{25 + (3 - 6) * 8\}$  包含匹配的符号。
6. 编写程序利用栈降序打印一个正整数的素数因子。
7. 在第11章中的编程实例中，曾使用递归的方法将一个整数从二进制表示转换为十进制表示。编写一个程序，使用栈将一个整数从二进制表示转换为十进制表示。
8. 在第11章中的编程实例中，曾使用递归的方法将一个整数从十进制表示转换为二进制表示。编写一个程序，使用栈将一个整数从十进制表示转换为二进制表示。
9. （中缀转后缀）编写一个程序将中缀表达式转换为等价的后缀表达式。

转换的规则如下：

假设 infix 代表中缀表达式，pfx 代表后缀表达式，将 infix 转换为 pfx 的规则为：

- a. 初始化 pfx 为空表达式并初始化栈。
- b. 从 infix 中得到符号 sym。
  - b.1. 如果 sym 是操作数，将 sym 追加到 pfx 中。
  - b.2. 如果 sym 是 (，将 sym 压栈。
  - b.3. 如果 sym 是 )，弹出栈中所有符号并追加到 pfx 中，直到遇到一个左括号。弹出并丢弃这个左括号。
  - b.4. 如果 sym 是运算符：
    - b.4.1 弹出栈中所有位于最近一个左括号之上、优先级高于或者等于 sym 的运算符并追加到 pfx 中。
    - b.4.2 将 sym 压入栈中。
- c. 在处理完 infix 后，栈中可能会留有一些运算符，弹出栈中所有的符号并将其追加到 pfx 中。在这一程序中，只考虑下列算术运算符：+、-、\* 和 /。假设程序将要处理的表达式没有错误。

设计一个类存放中缀和后缀字符串并且包含下列操作：

1. getInfix()：储存中缀表达式。
  2. showInfix()：输出中缀表达式。
  3. showPostfix()：输出后缀表达式。
- 其他一些可能需要的操作包括：
4. convertToPostfix()：将中缀表达式转换为后缀表达式，并将转换结果存入 postfixString 中。
  5. precedence()：确定两个操作符的优先级。如果前一个运算符高于或者等于第2个操作符，返回 true；否则返回 false。

包含构造函数和析构函数，从而实现自动初始化和动态分配内存。

使用下列表达式测试程序：

1.  $A + B - C;$
2.  $(A + B) * C;$
3.  $(A + B) * (C - D);$
4.  $A + ((B + C) * (E - F) - G) / (H - I);$
5.  $A + B * (C + D) - E / F * G + H;$

对于每一个表达式，答案必须是下列形式：

Infix Expression:  $A + B - C;$

Postfix Expression:  $A B + C -$

10. 针对类 `queueType` 编写重载赋值运算符和复制构造函数的函数定义, 并给出测试程序。
11. 针对类 `linkedQueueType` 编写重载赋值运算符和复制构造函数的函数定义, 并给出测试程序。
12. 本章描述了一种带保留位置的数组队列, 该位置用来区分队列的空状态与满状态。编写出类的定义以及实现这一队列的成员函数定义, 并编写一个测试程序测试队列上的各种操作。
13. 编写函数 `runSimulation()` 的定义以完成计算机仿真程序的设计 (参见“队列的应用: 仿真”一节)。使用不同的数据测试程序。在本练习中, 使用一个随机数产生器确定某一给定时间单位内是否有顾客到达。

## 附录A 保留字

|                  |          |             |          |
|------------------|----------|-------------|----------|
| and              | and_eq   | asm         | auto     |
| bitand           | bitor    | bool        | break    |
| case             | catch    | char        | class    |
| compl            | const    | const_cast  | continue |
| default          | delete   | do          | double   |
| dynamic_cast     | else     | enum        | explicit |
| export           | extern   | false       | float    |
| for              | friend   | goto        | if       |
| inline           | int      | long        | mutable  |
| namespace        | new      | not         | not_eq   |
| operator         | or       | or_eq       | private  |
| protected        | public   | register    | signed   |
| reinterpret_cast | return   | short       | struct   |
| sizeof           | static   | static_cast | throw    |
| switch           | template | this        | typeid   |
| true             | try      | typedef     | using    |
| typename         | union    | unsigned    | wchar_t  |
| virtual          | void     | volatile    |          |
| while            | xor      | xor_eq      |          |



## 附录 B 运算符的优先级

下面的表显示了 C++ 中运算符的优先级（由高到低）和结合性。

| 操作符                           | 结合性  |
|-------------------------------|------|
| :: (二元范围解析)                   | 从左向右 |
| :: (一元范围解析)                   | 从右向左 |
| ()                            | 从左向右 |
| [] -> .                       | 从左向右 |
| ++ -- (作为前缀运算符)               | 从右向左 |
| typeid dynamic_cast           | 从右向左 |
| static_cast const_cast        | 从右向左 |
| reinterpret_cast              | 从右向左 |
| ++ -- (作为前缀运算符) ! +(一元) -(一元) | 从右向左 |
| ~ & (地址运算符) *(取值运算符)          | 从右向左 |
| new delete sizeof             | 从右向左 |
| ->* -- .*                     | 从左向右 |
| * / %                         | 从左向右 |
| + -                           | 从左向右 |
| << >>                         | 从左向右 |
| < <= > >=                     | 从左向右 |
| == !=                         | 从左向右 |
| &                             | 从左向右 |
| ^                             | 从左向右 |
|                               | 从左向右 |
| &&                            | 从左向右 |
|                               | 从左向右 |
| ?:                            | 从右向左 |
| = += -= *= /= %=              | 从右向左 |
| <<= >>= &=  = ^=              | 从右向左 |
| throw                         | 从右向左 |
| , (逗号运算符)                     | 从左向右 |

## 附录 C 字符集

### C.1 ASCII (美国信息交换标准码)

下表显示了 ASCII 字符集。

| ASCII |     |     |     |     |     |     |     |     |     |     |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|       | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
| 0     | nul | soh | stx | etx | eot | enq | ack | bel | bs  | ht  |
| 1     | lf  | vt  | ff  | cr  | so  | si  | del | dc1 | dc2 | dc3 |
| 2     | dc4 | nak | syn | etb | can | em  | sub | esc | fs  | gs  |
| 3     | rs  | us  | b   |     | "   | #   | \$  | %   | &   | '   |
| 4     | (   | )   | *   | +   | ,   | -   | .   | /   | 0   | 1   |
| 5     | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | :   | ;   |
| 6     | <   | =   | >   | ?   | @   | A   | B   | C   | D   | E   |
| 7     | F   | G   | H   | I   | J   | K   | L   | M   | N   | O   |
| 8     | P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y   |
| 9     | Z   | [   | \   | ]   | ^   | _   | `   | a   | b   | c   |
| 10    | d   | e   | f   | g   | h   | i   | j   | k   | l   | m   |
| 11    | n   | o   | p   | q   | r   | s   | t   | u   | v   | w   |
| 12    | x   | y   | Z   | {   |     | }   | ~   | del |     |     |

第 1 列的数字 0~12 指定了 ASCII 字符集中字符十进制表示的左边一位数字，而第 2 行的数字 0~9 指定了右边一位的数字。例如，行号为 6（第 1 列中的数字）列号为 5（第 2 行中的数字）的字母是 A。因此，位于位置 65 上的字符（即第 66 个字符）是 A。而且，在位置 32 上的字符代表空格符。

前 32 个字符，即位置是 00~31 和位置是 127 的字符是不可打印字符。下表显示了这些字符的缩写及其含义。

|     |                     |     |                          |     |                  |
|-----|---------------------|-----|--------------------------|-----|------------------|
| nul | null character      | ff  | form feed                | can | cancel           |
| soh | start of header     | cr  | carriage return          | em  | end of medium    |
| stx | start of text       | so  | shift out                | sub | substitute       |
| etx | end of text         | si  | shift in                 | esc | escape           |
| eot | end of transmission | del | data link escape         | fs  | file separator   |
| enq | enquiry             | dc1 | device control 1         | gs  | group separator  |
| ack | acknowledge         | dc2 | device control 2         | rs  | record separator |
| bel | bell                | dc3 | device control 3         | us  | unit separator   |
| bs  | back space          | dc4 | device control 4         | b   | space            |
| ht  | horizontal tab      | nak | negative acknowledge     | del | delete           |
| lf  | line feed           | syn | synchronous idle         |     |                  |
| vt  | vertical tab        | etb | end of transmitted block |     |                  |



## C.2 EBCDIC (扩充的二进制编码的十进制交换码)

下表显示了 EBCDIC 字符集中的一些字符。

| EBCDIC |   |    |   |   |   |   |   |   |   |   |
|--------|---|----|---|---|---|---|---|---|---|---|
|        | 0 | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6      |   |    |   |   | b |   |   |   |   |   |
| 7      |   |    |   |   |   | . | < | ( | + |   |
| 8      | & |    |   |   |   |   |   |   |   |   |
| 9      | ! | \$ | * | ) | ; | ~ | - | / |   |   |
| 10     |   |    |   |   |   |   | ^ | , | % | _ |
| 11     | > | ?  |   |   |   |   |   |   |   |   |
| 12     |   | `  | : | # | @ | ' | = | " |   | a |
| 13     | b | c  | d | e | f | g | h | i |   |   |
| 14     |   |    |   |   |   | j | k | l | m | n |
| 15     | o | p  | q | r |   |   |   |   |   |   |
| 16     |   | -  | s | t | u | v | w | x | y | z |
| 17     |   |    |   |   |   |   |   | \ | { | } |
| 18     | [ | ]  |   |   |   |   |   |   |   |   |
| 19     |   |    |   | A | B | C | D | E | F | G |
| 20     | H | I  |   |   |   |   |   |   |   | J |
| 21     | K | L  | M | N | O | P | Q | R |   |   |
| 22     |   |    |   |   |   |   | S | T | U | V |
| 23     | W | X  | Y | Z |   |   |   |   |   |   |
| 24     | 0 | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

第1列数字 6~24 指定了 EBCDIC 字符集中字符十进制表示的左边一位数字, 而第2行数字 0~9 指定了右边一位的数字。例如, 行号为 19 (第1列中的数字), 列号为 3 (第2行中的数字) 的字符是 A。因此, 在位置是 193 的字符 (即第 194 个字符) 是 A。而且, 在位置 64 上的字符代表空格符。该表没有显示出 EBCDIC 字符集中所有的字符。实际上, 位置是 00~63 和 250~255 的字符是不可打印的控制字符。

## 附录 D 运算符重载

下表列出了所有可以被重载的运算符。

| 可重载的运算符 |    |    |    |     |     |     |        |
|---------|----|----|----|-----|-----|-----|--------|
| +       | -  | *  | /  | %   | ^   | &   |        |
| !       | && |    | =  | ==  | <   | <=  | >      |
| >=      | != | += | -= | *=  | /=  | %=  | ^=     |
| !=      | &= | << | >> | >>= | <<= | ++  | --     |
| ->*     | ,  | -> | [] | ()  | -   | new | delete |

下表列出了不可重载的运算符。

| 不可重载的运算符 |   |    |    |        |
|----------|---|----|----|--------|
| .        | * | :: | ?: | sizeof |

## 附录 E 在 ANSI/ISO 标准 C++ 和 标准 C++ 中头文件的名称约定

本书中的程序是使用 ANSI/ISO 标准 C++ 编写的，同时（在较早章节中）指出了如何使用标准 C++ 编写同样的程序。从本书的程序中可以清楚地看出，在标准 CPP 中头文件带有扩展名 .h，而在 ANSI/ISO 标准 CPP 中则不带扩展名。而且，某些头文件的名称，比如 math.h，在 ANSI/ISO 标准 C++ 中以字母 c 开头。C++ 语言自 C 语言发展而来，因此，某些头文件，例如 math.h、stdlib.h 和 string.h，都是从 C 语言中沿袭过来的，而诸如 iostream.h，iomanip.h 和 fstream.h 等是特别为 C++ 设计的。回忆一下，当一个头文件被包含进某一个程序中，头文件中的全局标识符也就变成了程序中的全局标识符。在 ANSI/ISO 标准 C++ 中，为了利用命名空间机制的先进之处，所有的头文件都被修改过，这样标识符都被声明在某一个命名空间中。回忆一下，这一命名空间的名称是 std。在 ANSI/ISO 标准 C++ 中，专门为 C++ 设计的头文件扩展名 .h 被省略了，同时它们名称的开始加上了字母 c。下面是在标准 C++ 和 ANSI/ISO C++ 中使用最为普遍的头文件名称。

| 标准 CPP 头文件名 | ANSI/ISO CPP 头文件名 |
|-------------|-------------------|
| assert.h    | cassert           |
| ctype.h     | cctype            |
| float.h     | cfloat            |
| fstream.h   | fstream           |
| iostream.h  | iostream          |
| iomanip.h   | iomanip           |
| limits.h    | climits           |
| math.h      | cmath             |
| stdlib.h    | cstdlib           |
| string.h    | cstring           |

例如，要包含头文件 iostream，需要下面的语句：

```
#include <iostream>
```

而且，要使用如 cin，cout，endl 等标识符，在使用该标识符之前，程序必须使用语句：

```
using namespace std;
```

或者在标识符前面加 std:: 前缀。

## 附录 F 头文件

C++ 标准库包含很多预定义的函数、常量和特定的数据类型。本附录将讨论一些使用最为广泛的函数（以及常量）。要得到更多的关于函数、常量的信息和解释，请参阅系统文档。标准 C++ 中对应的头文件名显示在括号中。

### F.1 头文件 `cassert` (`assert.h`)

下面的表描述了函数 `assert()`。它的声明包含在头文件 `cassert` (`assert.h`) 中。

|                                 |                                                                                       |                                                                                                                                                                                                                                                      |
|---------------------------------|---------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>assert(expression)</code> | <code>expression</code> 是 <code>int</code> 型表达式。<br><code>expression</code> 通常是逻辑表达式。 | <ul style="list-style-type: none"><li>● 如果 <code>expression</code> 的值是非零的 (<code>true</code>)，程序继续执行。</li><li>● 如果 <code>expression</code> 的值为 0 (<code>false</code>)，程序立刻停止执行。<br/><code>expression</code>、包含该段源代码的文件名以及该行在源文件中的行号会被打印出来。</li></ul> |
|---------------------------------|---------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

注意：要禁用 `assert` 语句，可以将预处理项 `#define NDEBUG` 放在 `#include <cassert>` 之前。

### F.2 头文件 `cctype` (`ctype.h`)

下表显示了头文件 `cctype` (`ctype.h`) 中的各种函数。

| 函数名和参数                   | 参数类型                                  | 函数返回值                                                                                                                                                                                                             |
|--------------------------|---------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>isalnum(ch)</code> | <code>ch</code> 是 <code>char</code> 型 | 函数以如下方式返回 <code>int</code> 型值： <ul style="list-style-type: none"><li>● 如果 <code>ch</code> 是字母或者数字字符，即 ('A'-'Z', 'a'-'z', '0'-'9')，返回非零值 (<code>true</code>)</li><li>● 否则，返回 0 (<code>false</code>)</li></ul>      |
| <code>isctrl(ch)</code>  | <code>ch</code> 是 <code>char</code> 型 | 函数以如下方式返回 <code>int</code> 型值： <ul style="list-style-type: none"><li>● 如果 <code>ch</code> 是一个控制字符（在 ASCII 字符集中值在 0~31 之间或者是 127），返回非零值 (<code>true</code>)</li><li>● 否则，返回 0 (<code>false</code>)</li></ul>       |
| <code>isdigit(ch)</code> | <code>ch</code> 是 <code>char</code> 型 | 函数以如下方式返回 <code>int</code> 型值： <ul style="list-style-type: none"><li>● 如果 <code>ch</code> 是一个数字 ('0'-'9')，返回非零值 (<code>true</code>)</li><li>● 否则，返回 0 (<code>false</code>)</li></ul>                              |
| <code>islower(ch)</code> | <code>ch</code> 是 <code>char</code> 型 | 函数以如下方式返回 <code>int</code> 型值： <ul style="list-style-type: none"><li>● 如果 <code>ch</code> 是小写字母 ('a'-'z')，返回非零值 (<code>true</code>)</li><li>● 否则，返回 0 (<code>false</code>)</li></ul>                              |
| <code>isprint(ch)</code> | <code>ch</code> 是 <code>char</code> 型 | 函数以如下方式返回 <code>int</code> 型值： <ul style="list-style-type: none"><li>● 如果 <code>ch</code> 是一个包含空格在内的可打印字符（在 ASCII 字符集中值在 ' ' 到 '~' 之间），返回非零值 (<code>true</code>)</li><li>● 否则，返回 0 (<code>false</code>)</li></ul> |
| <code>ispunct(ch)</code> | <code>ch</code> 是 <code>char</code> 型 | 函数以如下方式返回 <code>int</code> 型值： <ul style="list-style-type: none"><li>● 如果 <code>ch</code> 是一个标点符号返回非零值 (<code>true</code>)</li><li>● 否则，返回 0 (<code>false</code>)</li></ul>                                       |
| <code>isspace(ch)</code> | <code>ch</code> 是 <code>char</code> 型 | 函数以如下方式返回 <code>int</code> 型值： <ul style="list-style-type: none"><li>● 如果 <code>ch</code> 是一个空白字符（空格、换行符、tab、回车符、制表符），返回非零值 (<code>true</code>)</li><li>● 否则，返回 0 (<code>false</code>)</li></ul>                  |

(续表)

| 函数名和参数      | 参数类型        | 函数返回值                                                                              |
|-------------|-------------|------------------------------------------------------------------------------------|
| isupper(ch) | ch 是 char 型 | 函数以如下方式返回 int 型值:<br>● 如果 ch 是一个大写字母 ('A'-'Z'), 返回非零值 (true)<br>● 否则, 返回 0 (false) |
| tolower(ch) | ch 是 char 型 | 函数以如下方式返回 char 型值:<br>● 如果 ch 是一个大写字母, 返回对应的小写字母<br>● 否则, 返回 ch                    |
| toupper(ch) | ch 是 char 型 | 函数以如下方式返回 char 型值:<br>● 如果 ch 是一个小写字母, 返回对应的大写字母<br>● 否则, 返回 ch                    |

### F.3 头文件 cfloat ( float.h )

在第 2 章中, 列举了属于浮点数类型的最大值和最小值, 这些值是系统相关的。这些最大值和最小值存储在预定义常数中。头文件 cfloat 包含很多这样的常量。下表列出了其中的一些值。

| 常量       | 描述           |
|----------|--------------|
| FLT_DIG  | 浮点数的有效数字位数   |
| FLT_MAX  | 正浮点数的最大值     |
| FLT_MIN  | 正浮点数的最小值     |
| DBL_DIG  | 双精度数的有效数字位数  |
| DBL_MAX  | 双精度数的最大值     |
| DBL_MIN  | 双精度数的最小值     |
| LDBL_DIG | 长双精度数的有效数字位数 |
| LDBL_MAX | 长双精度数的最大值    |
| LDBL_MIN | 长双精度数的最小值    |

下列的程序能够打印出系统中以上常数的值。

```
#include <iostream>
#include <cfloat>

using namespace std;

int main()
{
 cout<<"Approximate number of significant digits "
 <<"in a float value "<<FLT_DIG<<endl;
 cout<<"Maximum positive float value "<<FLT_MAX<<endl;
 cout<<"Minimum positive float value "<<FLT_MIN<<endl;
 cout<<"Approximate number of significant digits "
 <<"in a double value "<<DBL_DIG<<endl;
 cout<<"Maximum positive double value "<<DBL_MAX<<endl;
 cout<<"Minimum positive double value "<<DBL_MIN<<endl;
 cout<<"Approximate number of significant digits "
 <<"in a long double value "<<LDBL_DIG<<endl;
 cout<<"Maximum positive long double value "<<LDBL_MAX
 <<endl;
 cout<<"Minimum positive long double value "<<LDBL_MIN
 <<endl;
 return 0;
}
```

如果想在前面的程序中使用标准 C++ 头文件, 可以将语句:

```
#include <iostream>
#include <cfloat>
using namespace std;
```

替换为:

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

## F.4 头文件 climits ( limits.h )

在第 2 章中,列出了整数数据类型的最大值和最小值。同样地,这些值是系统相关的,这些值存储在预定义常数中。头文件 climits 包含了这些常数。下表列出了其中的一些值。

| 常量        | 描述                   |
|-----------|----------------------|
| CHAR_BIT  | byte 的位数             |
| CHAR_MAX  | char 型的最大值           |
| CHAR_MIN  | char 型的最小值           |
| SHRT_MAX  | short 型的最大值          |
| SHRT_MIN  | short 型的最小值          |
| INT_MAX   | int 型的最大值            |
| INT_MIN   | int 型的最小值            |
| LONG_MAX  | long 型的最大值           |
| LONG_MIN  | long 型的最小值           |
| UCHAR_MAX | unsigned char 型的最大值  |
| USHRT_MAX | unsigned short 型的最大值 |
| UINT_MAX  | unsigned int 型的最大值   |
| ULONG_MAX | unsigned long 型的最大值  |

下面的程序可以打印出这些常量在系统中的值。

```
#include <iostream>
#include <climits>

using namespace std;

int main()
{
 cout<<"Number of bits in a byte "<<CHAR_BIT<<endl;
 cout<<"Maximum char value "<<CHAR_MAX<<endl;
 cout<<"Minimum char value "<<CHAR_MIN<<endl;
 cout<<"Maximum short value "<<SHRT_MAX<<endl;
 cout<<"Minimum short value "<<SHRT_MIN<<endl;
 cout<<"Maximum int value "<<INT_MAX<<endl;
 cout<<"Minimum int value "<<INT_MIN<<endl;
 cout<<"Maximum long value "<<LONG_MAX<<endl;
 cout<<"Minimum long value "<<LONG_MIN<<endl;
 cout<<"Maximum unsigned char value "<<UCHAR_MAX<<endl;
 cout<<"Maximum unsigned short value "<<USHRT_MAX<<endl;
 cout<<"Maximum unsigned int value "<<UINT_MAX<<endl;
 cout<<"Maximum unsigned long value "<<ULONG_MAX<<endl;

 return 0;
}
```

要在上述程序中使用标准 C++ 头文件,只需将语句:



```
#include <iostream>
#include <climits>
using namespace std;
```

替换为:

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

## F.5 头文件 cmath ( math.h )

下表显示了各种数学函数。

| 函数名和参数    | 参数类型                                                               | 函数返回值                                |
|-----------|--------------------------------------------------------------------|--------------------------------------|
| acos(x)   | x 是一个浮点表达式, $-1.0 \leq x \leq 1.0$                                 | x 的反余弦值, 介于 0.0 和 $\pi$ 之间           |
| asin(x)   | x 是一个浮点表达式, $-1.0 \leq x \leq 1.0$                                 | x 的正弦值, 介于 $-\pi/2$ 和 $\pi/2$ 之间     |
| atan(x)   | x 是一个浮点表达式                                                         | x 的反正切值, 介于 $-\pi/2$ 和 $\pi/2$ 之间    |
| ceil(x)   | x 是一个浮点表达式                                                         | 不小于 x 的最小的整数值                        |
| cos(x)    | x 是一个浮点表达式, x 以弧度表示                                                | x 的余弦值, 例如, 如果 $x = 90$ , cos(x) 为 0 |
| cosh(x)   | x 是一个浮点表达式                                                         | x 的双曲余弦值                             |
| exp(x)    | x 是一个浮点表达式                                                         | e 的 x 次幂 ( $e = 2.718...$ )          |
| fabs(x)   | x 是一个浮点表达式                                                         | x 的绝对值                               |
| floor(x)  | x 是一个浮点表达式                                                         | x 的反余弦值, 不大于 x 的最大整数值                |
| log(x)    | x 是一个浮点表达式, $x \geq 0.0$                                           | x 的自然对数 (以 e 为底)                     |
| log10(x)  | x 是一个浮点表达式, $x \geq 0.0$                                           | x 的常用对数 (以 10 为底)                    |
| pow(x, y) | x 和 y 是浮点数表达式, 如果 $x = 0.0$ ,<br>y 必须为正; 如果 $x \leq 0.0$ , y 必须为整数 | x 的 y 次幂                             |
| sin(x)    | x 是一个浮点表达式, x 以弧度表示                                                | x 的正弦值, 例如, 如果 $x = 90$ , sin(x) 为 1 |
| sinh(x)   | x 是一个浮点表达式                                                         | x 的双曲正弦值                             |
| sqrt(x)   | x 是一个浮点表达式, $x \geq 0.0$                                           | x 的平方根                               |
| tan(x)    | x 是一个浮点表达式, $x \geq 0.0$                                           | x 的正切值, 例如, 如果 $x = 45$ , tan(x) 为 1 |
| tanh(x)   | x 是一个浮点表达式                                                         | x 的双曲正切值                             |

## F.6 头文件 cstdlib ( stddef.h )

该头文件包含了下列符号常量: NULL, 即系统相关的空指针 (通常为 0)

## F.7 头文件 cstring ( string.h )

下表显示了各种字符串函数。

| 函数名和参数                  | 参数类型                                                              | 函数返回值                                                                                               |
|-------------------------|-------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| strcat(destStr, srcStr) | destStr 和 srcStr 是以 null 结尾的<br>char 数组; destStr 必须足够大,<br>以便存储结果 | 返回 destStr 的地址; srcStr 包括 null 字符并被连接到 destStr 的尾部                                                  |
| strcmp(str1, str2)      | str1 和 str2 是以 null<br>结尾的 char 数组                                | 返回值为以下几种形式:<br>● 负数 int 值, 如果 $str1 < str2$<br>● 0, 如果 $str1 = str2$<br>● 正 int 值, 如果 $str1 > str2$ |
| strcpy(destStr, srcStr) | destStr 和 srcStr 是以 null 结尾的<br>char 数组                           | 返回 destStr 的地址; srcStr 被复制到 destStr                                                                 |
| strlen(str)             | str 是一个以 null 结尾的 char 数组                                         | 返回字符串的长度 (不包括 '\0')                                                                                 |

## F.8 头文件 string

该头文件（不要与头文件 `cstring` 混淆）提供了一种用户定义的数据类型 `string`。与 `string` 类型相关的是数据类型 `string::size_type` 和常量 `string::npos`，如下所示：

```
string::size_type unsigned integer 型
string::npos string::size_type 型的最大值
```

与类型 `string` 相关的还有几个函数，下表显示了其中一些函数。除非声明，否则 `str`、`str1` 和 `str2` 是 `string` 型变量（对象）。`string` 型变量（例如 `str`）第一个字符的位置是 0，第二个字符是 1，依次类推。

| 函数名和参数                                | 形参列表                                                                                                                                       | 功能和函数返回值                                                                                                                                                                              |
|---------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>str.c_str()</code>              | 无                                                                                                                                          | 一个以 null 结尾的与 <code>str</code> 中的字符对应的 C-string 的基址                                                                                                                                   |
| <code>getline(istreamVar, str)</code> | <code>istreamVar</code> 是（ <code>istream</code> 或者 <code>ifstream</code> 型的）输入流变量。 <code>str</code> 是 <code>string</code> 对象               | <code>istreamVar</code> 中的第一个换行符之前的所有字符被存入 <code>str</code> （换行符被读入，但并没有被存储）。该函数返回值通常忽略                                                                                               |
| <code>str.empty()</code>              | 无                                                                                                                                          | 如果 <code>str</code> 为空返回 <code>true</code> ，即 <code>str</code> 中包含的字符数为 0；否则，返回 <code>false</code>                                                                                    |
| <code>str.length()</code>             | 无                                                                                                                                          | <code>string::size_type</code> 型的值，给出字符串中的字符数                                                                                                                                         |
| <code>str.size()</code>               | 无                                                                                                                                          | <code>string::size_type</code> 型的值，给出字符串中的字符数                                                                                                                                         |
| <code>str.find(strExp)</code>         | <code>str</code> 是一个 <code>string</code> 对象， <code>strExp</code> 是一个 <code>string</code> 计算表达式。表达式 <code>strExp</code> 还可以是一个字符            | <code>find()</code> 函数搜索 <code>str</code> 以寻找 <code>strExp</code> 所指定的字符串或者字符第一次出现的位置。如果搜索成功，函数 <code>find()</code> 返回 <code>str</code> 中的位置。如果搜索失败，函数返回特殊值 <code>string::npos</code> |
| <code>str.substr(pos, len)</code>     | 参数为无符号整数类型。 <code>pos</code> 为子串的开始位置， <code>len</code> 为子串长度。 <code>pos</code> 值必须小于 <code>str.length()</code>                            | 以 <code>pos</code> 开始并含有 <code>str</code> 子串的临时字符串。子串的长度最大为 <code>len</code> 个字符。如果 <code>len</code> 太大，意味着“到达 <code>str</code> 的串尾”                                                  |
| <code>str1.swap(str2)</code>          | <code>string</code> 类型的参数                                                                                                                  | <code>str1</code> 和 <code>str2</code> 的内容被交换                                                                                                                                          |
| <code>str.clear()</code>              | 无                                                                                                                                          | 删除 <code>str</code> 中所有的字符                                                                                                                                                            |
| <code>str.erase()</code>              | 无                                                                                                                                          | 删除 <code>str</code> 中所有的字符                                                                                                                                                            |
| <code>str.erase(m)</code>             | <code>string::size_type</code> 类型的参数                                                                                                       | 删除 <code>str</code> 中索引从 <code>m</code> 开始之后所有字符                                                                                                                                      |
| <code>str.erase(m, n)</code>          | 两个 <code>int</code> 型的参数                                                                                                                   | 删除 <code>str</code> 中索引从 <code>m</code> 开始的 <code>n</code> 个字符。<br>如果 <code>n &gt; str</code> 的长度，则删除从第 <code>m</code> 个字符开始之后所有的字符                                                   |
| <code>str.insert(m, c)</code>         | <code>m</code> 是 <code>string::size_type</code> 类型的参数， <code>c</code> 是一个字符                                                                | 在索引 <code>m</code> 的位置插入字符 <code>c</code>                                                                                                                                             |
| <code>str.insert(m, n, c)</code>      | <code>m</code> 和 <code>n</code> 是 <code>string::size_type</code> 类型的参数； <code>c</code> 是一个字符                                               | 在 <code>str</code> 索引 <code>m</code> 的位置插入 <code>n</code> 个字符 <code>c</code>                                                                                                          |
| <code>str1.insert(m, str2)</code>     | <code>m</code> 是 <code>string::size_type</code> 类型参数；<br><code>str1</code> 和 <code>str2</code> 是 <code>string</code> 型的参数                  | 在 <code>str1</code> 索引 <code>m</code> 的位置插入 <code>str2</code> 的所有字符                                                                                                                   |
| <code>str1.replace(m, n, str2)</code> | <code>m</code> 和 <code>n</code> 是 <code>string::size_type</code> 类型参数；<br><code>str1</code> 和 <code>str2</code> 是 <code>string</code> 型的参数 | 以索引 <code>m</code> 开始，将 <code>str1</code> 的后续 <code>n</code> 个字符用 <code>str2</code> 的所有字符替换。如果 <code>n &gt; str1</code> 的长度，则 <code>str1</code> 中直到结尾的所有字符将被替换                        |



## 附录 G 系统中数据类型的内存尺寸

下列程序可以打印出系统中各种内建数据类型的内存尺寸(程序的输出显示了运行该程序的系统中各种内建数据类型的尺寸)。

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

int main()
{
 cout<<"Size of char = "<<sizeof(char)
 <<endl;
 cout<<"Size of int = "<<sizeof(int)
 <<endl;
 cout<<"Size of short = "<<sizeof(short)
 <<endl;
 cout<<"Size of unsigned int = "<<sizeof(unsigned int)
 <<endl;
 cout<<"Size of long = "<<sizeof(long)
 <<endl;
 cout<<"Size of bool = "<<sizeof(bool)
 <<endl;
 cout<<"Size of float = "<<sizeof(float)
 <<endl;
 cout<<"Size of double = "<<sizeof(double)
 <<endl;
 cout<<"Size of long double = "<<sizeof(long double)
 <<endl;
 cout<<"Size of unsigned short = "<<sizeof(unsigned short)
 <<endl;
 cout<<"Size of unsigned long = "<<sizeof(unsigned long)
 <<endl;

 return 0;
}
```

输出:

```
Size of char = 1
Size of int = 4
Size of short = 2
Size of unsigned int = 4
Size of long = 4
Size of bool = 1
Size of float = 4
Size of double = 8
Size of long double = 8
Size of unsigned short = 2
Size of unsigned long = 4
```

## 附录 H 部分练习答案

### 第 1 章

1. a.True; b. False; c. False; d. False; e. False; f. False; g. False; h.True
2. 控制单元, 指令寄存器, 程序计数器, 算术逻辑单元和累加器。
4. 键盘和鼠标。
5. 屏幕和打印机。
8. 高级语言中的指令更接近于自然语言, 例如英语, 因此, 比机器语言更容易理解和学习。

### 第 2 章

1. a. False; b. False; c. False; d.True; e.True; f. False; g.True; h.True; i. False; j.True; k. False
3. a, b
5. a. 3; b. 不可能; c. 不可能; d. 38.5; e. 1; f. 2; g. 2; h. 420.0
8. 只有变量声明 “double x, y, z;” 是正确的
9. a 和 c 是正确的
12.  $x = 5$   
 $y = 2$   
 $z = 3$   
 $w = 9$
14. a.  $x = 2, y = 5, z = 6$   
b.  $x + y = 7$   
c. Sum of 2 and 6 is 8  
d.  $z / x = 3$   
e. 2 times 2 = 4
17. a 和 c 是正确的
20. a.  $x = x + 5 - z;$   
b.  $y = y * (2 * x + 5 - z);$   
c.  $w = w + 2 * z + 4;$   
d.  $x = x - (z + y - t);$   
e.  $sum = sum + num;$

22.

|                    | a  | b | c    | sum |
|--------------------|----|---|------|-----|
| $sum = a + b + c;$ | 3  | 5 | 14.1 | 22  |
| $c /= a;$          | 3  | 5 | 4.7  | 22  |
| $b += c - a;$      | 3  | 6 | 4.7  | 22  |
| $a *= 2 * b + c;$  | 50 | 6 | 4.7  | 22  |

### 第 3 章

1. a. True; b.True; c. False; d. False; e.True; f.True

2. a.  $x = 5, y = 28, ch = '3'$

b.  $x = 28, y = 36, ch = '5'$

c.  $x = 5, y = 8, ch = '2'$

d.  $x = 5, y = 28, ch = ' '$

5. `cout<<setfill('*')<<setw(35)<<'* '<<endl;`

7. a. 不正确的数据可能引起输入流进入失败状态。

b. 当输入流进入失败状态时, 所有与之相关的后续输入都被忽略掉。不管变量为何值, 程序将继续执行。

## 第 4 章

1. a. False; b. False; c. False; d. True; e. False; f. False; g. False; h. False; i. False; j. True

2. a. i; b. i; c. ii; d. i; e. iii

5. 省略 else 后的分号。正确语句如下:

```
if(score >= 60)
 cout<<"You pass."<<endl;
else
 cout<<"You fail."<<endl;
```

7.15

9.96

10.

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

int main ()
{
 int a, b, c found;
 cout<<"Enter two integers: ";
 cin>>a>>b;

 if(a > a*b && 10 < b)
 found = 2* a > b;
 else
 {
 found = 2 * a < b;
 if(found)
 a = 3;
 c = 15;
 if(b)
 {
 b = 0;
 a = 1;
 }
 }
 return 0;
}
```

## 第 5 章

1. a. False; b. True; c. False; d. True; e. True; f. True; g. True

2.  $y = 1$  and  $count = 100$

3. 5

5. `if ch > 'Z' or ch < 'A'`  
 6. `Sum = 112`  
 8. `Sum = 157`  
 12. 8 23 21 0 23 0 12 1 20  
 14. 0 10 30 60  
 15. 2 7 17 37 77 157  
 17. a. \*  
     b. Infinite loop  
     c. Infinite loop  
     d. \*\*\*\*  
     e. \*\*\*\*  
     f. \*\*\*

20. 11 19 30 49

26.

a.  
`cin>>number;`  
`while(number != -1)`  
`{`  
`total = total + number;`  
`cin>>number;`  
`}`  
`cout<<endl;`  
`cout<<total;`

27. a.

`number = 1;`  
`while(number <= 10)`  
`{`  
`cout<<setw(3)<<number;`  
`number++;`  
`}`

30.

a.  
 1 2 3 4 5  
 2 4 6 8 10  
 3 6 9 12 15  
 4 8 12 16 20  
 5 10 15 20 25  
 b.  
 1 1 1 1 1  
 2 2 2 2 2  
 3 3 3 3 3  
 4 4 4 4 4  
 5 5 5 5 5

## 第 6 章

1. a. False; b. True; c. True; d. True; e. False  
 3. a. 错误。缺少函数返回类型  
     b. 正确。

- c. 错误。参数 b 缺少数据类型
- d. 错误。函数名后缺少括号
- 5. a. 4
- b. 26
- c. 1040
- d. 0
- 8. a. (i) 125; (ii) 432
- b. 函数计算参数 x 的立方

## 第 7 章

1. a.True; b. False; c.True; d. False; e.True; f. False; g. False; h. False; i.True

5.

```
3 4 20 78
7 3 20 4
7 5 6 2
```

8.

```
10 20
5 20
```

9.

```
Line 4: In main: num1 = 10, num2 = 20, and t = 15
Line 11: In funOne: a = 10, x = 15, z = 25, and t = 15
Line 13: In funOne: a = 10, x = 20, z = 25, and t = 20
line 15: In funOne: a = 22, x = 20, z = 25, and t = 20
Line 17: In funOne: a = 22, x = 33, z = 25, and t = 33
Line 6: In main after funOne: num1 = 22, num2 = 20, and t = 33
Line 19: In funTwo: u = 20, v = 22, aTwo = 20, and t = 33
Line 11: In funOne: a = 20, x = 20, z = 40, and t = 33
Line 13: In funOne: a = 20, x = 25, z = 40, and t = 33
line 15: In funOne: a = 32, x = 25, z = 40, and t = 33
Line 17: In funOne: a = 32, x = 25, z = 40, and t = 46
Line 21: In funTwo after funOne: u = 25, v = 22, aTwo = 32, and t = 46
Line 23: In funTwo: u = 38, v = 22, aTwo = 32, and t = 46
Line 25: In funTwo: u = 38, v = 22, aTwo = 32, and t = 92
Line 8: In main after funTwo: num1 = 22, num2 = 38, and t = 92
11. (i), (ii), and (iv) are correct.
```

## 第 8 章

- 1. a. True; b. False; c.True; d. False; e. False; f.True; g.True; h.True; i. False; j.True; k. False
- 4. a. true; b. false; c. true; d. false
- 6. 命名空间机制不能够与标准 C++ 风格头文件（即以 .h 结尾的头文件）一起使用。因此，Line 1 中删除 .h。
- 8. Line 3 中缺少关键字 namespace。

## 第 9 章

- 1. a. True; b.True; c. False; d. False; e.True; f. False; g. False; h. False; i.True; j. False; k. False
- 2. a. 正确

- b. 正确
- c. 不正确; 赋值运算符不是为 C-string 定义的。
- d. 正确
- e. 正确
- f. 不正确; 关系运算符不是为 C-string 定义的。
- g. 正确

5.

- a. strcpy(str1, "Sunny Day");
- b. length = strlen(str1);
- c. strcpy(str2, name);
- d.
 

```
if(strcmp(str1, str2) <= 0)
 cout<<str1;
else
 cout<<str2;
```

7.

- a. funcOne(list, 50);
- d. funcTwo(list, Alist);

9. a. 正确

b. 正确

c. 正确

d. 不正确; 赋值运算不是为 C-string 定义的。

e. 不正确; 赋值运算符和关系运算符不是为 C-string 定义的。

f. 正确

g. 正确

h. 正确

10. 列出的元素: 11 16 21 26 30

## 第 10 章

1. a. False; b. True; c. False; d. False; e. False

2. a. 8

b. 6

c. 1

d. 8

4. a.

| 循环 | first | last | mid  | list[mid] | 比较次数 |
|----|-------|------|------|-----------|------|
| 1  | 0     | 10   | 5    | 55        | 2    |
| 2  | 0     | 4    | 2    | 17        | 2    |
| 3  | 0     | 1    | 0    | 2         | 2    |
| 4  | 1     | 1    | 1    | 10        | 2    |
| 5  | 2     | 1    | 循环终止 |           |      |

这是一个不成功的查找。比较的总数是 8。

5. a. 30

b. 5



e. column

6. a. `int alpha[10][20];`

b.

```
for(j = 0; j < 10; j++)
 for(k = 0; k < 20; k++)
 alpha[j][k] = 0;
```

e.

```
for(j = 0; j < 10; j++)
{
 for(k = 0; k < 20; k++)
 cout<<alpha[j][k]<<" ";
 cout<<endl;
}
```

7. a. beta 初始化为 0。

d.

```
First row of beta: 0 2 0
Second row of beta: 2 0 2
Third row of beta: 0 2 0
```

## 第 11 章

1. a. True; b. True; c. False; d. False; e. False

2. a. Base case:

```
if(u == 0)
 cout<<v;
else if(u == 1)
 cout<<static_cast<char>(static_cast<int>(v) + 1);
```

b. Recursive case: `funcRec(u - 1, v);`

c. B

## 第 12 章

1. a. False; b. False; c. True; d. True; e. True; f. True; g. False

3. a. 不正确; `newEmployee` 的成员 `name` 是一个结构体。需要指定成员 `name` 来存储值 "John Smith"。例如:

```
newEmployee.name.first = "John";
newEmployee.name.first = "Smith";
```

b. 不正确; `newEmployee` 的成员 `name` 是一个结构, 结构体上没有集成的输入操作。正确的写法是:

```
cin>>newEmployee.name.first;
cin>>newEmployee.name.last;
```

c. 正确

d. 正确

e. 不正确; `employees` 是一个数组, 数组上没有聚集赋值运算。

## 第 13 章

1. a. False; b. False; c. True; d. False; e. False

4. a. (i) Line 1 中的构造函数

(ii) Line 3 中的构造函数

(iii) Line 4 中的构造函数

b.

```
CC::CC()
{
 u = 0;
 v = 0;
}
```

c.

```
CC::CC(int x)
{
 u = x;
 v = 0;
}
```

## 第 14 章

1. a.True; b.True; c.True; d. False

4. 类的私有成员是 `private` 的, 它们不能够被派生类的成员函数直接访问。基类的 `protected` 成员可以被派生类的成员函数直接访问。

5. a. 其中:

```
class bClass public aClass
```

应为:

```
class bClass: public aClass
```

b. 花括号 “)” 后缺少分号。

7.

a.

```
yClass::yClass()
{
 a = 0;
 b = 0;
}
```

b.

```
xClass::xClass()
{
 z = 0;
}
```

c.

```
void yClass::two(int u, int v)
{
 a = u;
 b = v;
}
```

11.

In base: `x = 7`

In derived: `x = 3, y = 8; x + y = 11`

\*\*\*\* 7

#### 11



## 第 15 章

1. a. False; b. False; c. False; d. True; e. True; f. True; g. False; h. False
2. a. 正确  
b. 正确  
c. 不正确; p 是一个指针变量, x 是一个 int 型变量。x 的值不能够赋给 p。  
d. 正确  
e. 正确  
f. 不正确; \*p 是一个 int 型变量, q 是一个指针变量。q 的值不能够赋给 \*p。
5. b 和 c
7. 78 78
8. Line 5 语句将 p 复制给 q。在这条语句执行后, p 和 q 都指向同一个内存地址。Line 7 的语句释放了 q 所指向的内存空间, 同时使 p 和 q 失效。因此, Line 8 所打印的值无法预料。
9. 4 4 5 7 10 14 19 25 32 40
12. Line 6 处的语句将 p 的值复制给 q。在语句执行过后, p 和 q 都指向相同的数组。Line 7 的语句释放了 p 所指向的数组内存空间, 因而也使 q 失效。因此, Line 9 处打印的值无法预料。
16. 如果类包含指针数据成员, 那么它应该包含析构函数、重载赋值运算符并显式地提供复制构造函数。
17.  
ClassA x: 4  
ClassA x: 6  
ClassB y: 5
19. 在编译期绑定中, 编译器产生调用函数的必要代码; 在运行期绑定中, 运行时系统产生必要的代码以便适当地进行函数调用。

## 第 16 章

1. a. False; b. True; c. True; d. False; e. False; f. True; g. False; h. True; i. False; j. True; k. False
5. 当类有指针数据成员时。
7.  
a. friend strange operator+(const strange&, const strange&);  
b. friend bool operator==(const strange&, const strange&);  
c. friend strange operator++(strange&, int);
9. 在 Line 2 中, 缺少 bool 前的 friend。正确的语句为:  
friend bool operator<=(mystery, mystery); //Line 2
11. None.
14. Two.
17. Line 4 有错。模板实例化可以只是内建的类型, 也可以是用户定义的类型。在尖括号之间的“type”必须用内建类型或者用户定义类型替换。
19. a. 12 b. Sunny Day
20. a. 21 b. OneHow
21.  
template <class Type>  
void swap(Type &x, Type &y)

```

{
 Type temp;
 temp = x;
 x = y;
 y = temp;
}

```

## 第 17 章

1. a. False; b. False; c. False; d. False; e. True; f. True

3. a. True; b. True; c. False; d. False; e. True

4.

a. 正确

b. 正确

c. 正确

d. 不正确; B 是一个指针, 而 \*List 是一个结构体。

e. 正确

f. 不正确; B 是一个指针, 而 A->link->info 是 int 型的。

g. 正确

h. 正确

i. 正确

6. 一个无限循环, 不断地打印 18。

7. a. 这段代码不正确。语句 “s->info = B;” 是不正确的, 因为 B 是一个指针, 而 s->info 是 int 型。

b. 这段代码不正确。在语句 “s = s->link;” 执行过后, s 为 NULL, 所以 s->info 不存在。

## 第 18 章

1.

x = 3

y = 9

7

13

4

7

H

2.

x = 45

x = 23

x = 5

Stack Elements: 10 34 14 5

4. a. AB+CD+\*E-

b. ABC+D\*-EF/+

c. AB+CD-/E+F\*G-

d. ABCD+\*+EF/G\*-H+

6.

Queue Element = 0

Queue Element = 14

Queue Element = 22

```
Sorry queue is empty
Sorry queue is empty
Stack Element = 32
Stack Elements: 64 28 0
Queue Elements: 30
```

8.

```
template <class Type>
void reverseQueue(queueType<Type> &q, stackType<Type> &s)
{
 Type elem;
 while (!q.isEmptyQueue())
 {
 q.dequeue(elem);
 s.push(elem);
 }
 while (!s.isEmptyStack())
 {
 s.pop(elem);
 q.addQueue(elem);
 }
}
```

[ G e n e r a l   I n f o r m a t i o n ]

书名= C + + 编程——从问题分析到程序设计

作者=

页数= 7 2 4

S S 号= 1 1 1 3 8 4 4 2

出版日期=

封面  
书名  
版权  
前言  
目录  
正文