国外计算机科学教材系列

C++ 编程
——数据结构与程序设计方法

C++ Programming
Program Design Including Data Structures

[美] D. S. Malik 著
娄海华 蔡旭辉 常鸿 等译
娄海华 审校

THOMSON
COURSE TECHNOLOGY

電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn

经典教材

C++编程

——数据结构与程序设计方法

C++ Programming

Program Design Including Data Structures

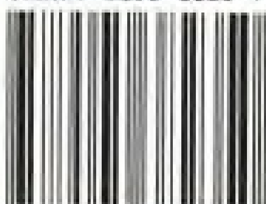
本书是一本无需任何程序设计基础、内容涵盖两个学期（CS1和CS2）的权威教程。全书的编写与设计充分考虑到学生的学习方法，借助于详尽的解释和例子对C++进行全面描述。书中还包含了标准模板库、二叉搜索树、图论、查找和排序等算法，这些内容可安排在第二学期进行。

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++ 6.0，Borland C++ Builder 5 开发工具，或是 Metrowerks CodeWarrior 来编译这些程序。
- 每章后面的程序范例都是完整的程序代码。这些程序范例中包括输入、输出、问题分析和算法设计等具体步骤，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。



THOMSON COURSE TECHNOLOGY

责任编辑：李秦华
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 7-5053-8235-7/TP · 4798　定价：69.00元

国外计算机科学教材系列

C++ 编程

——数据结构与程序设计方法

C++ Programming

Program Design Including Data Structures

[美] D. S. Malik 著

晏海华 蔡旭辉 常 鸿 等译

晏海华 审校

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内容简介

本书是由具有多年教学实践经验的美国Creighton大学D. S. Malik教授根据其课堂讲稿编写而成的C++实用教程。全书系统地介绍了ANSI/ISO标准C++，结构化程序设计和面向对象程序设计以及用C++实现的数据结构。

全书共分21章。前11章是基础部分，主要介绍C++程序设计语言、程序结构和结构化程序设计；第12章至第15章是面向对象程序设计部分，主要介绍了建立在C++程序设计语言基础上的面向对象程序设计方法；第16章至第20章是数据结构部分，主要介绍了数据结构的基本概念、算法，以及如何用C++程序设计语言来实现；第21章介绍了C++标准模板库。

本书概念清楚，内容充实，取材适当，通俗易懂，既可以直接用做大学计算机及相关专业程序设计语言教程及数据结构的参考书籍，又可以供自学者使用。

981-243-397-X

Simplified Chinese edition Copyright © 2003 by Thomson Learning and Publishing House of Electronics Industry.
C++ Programming: Program Design Including Data Structures 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：01-2002-3800

图书在版编目（CIP）数据

C++编程——数据结构与程序设计方法 /（美）马力克（Malik, D. S.）著；晏海华等译. -北京：电子工业出版社，2003.6
（国外计算机科学教材系列）
书名原文：C++ Programming: Program Design Including Data Structures
ISBN 7-5053-8235-7

Ⅰ.C... Ⅱ.①马... ②晏... 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045319号

责任编辑：李秦华
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：60.5 字数：1840千字
版 次：2003年6月第1版 2003年6月第1次印刷
定 价：69.00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重要时期，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和自由组合使用。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

由于面向对象技术的普及，C++已取代传统的过程性语言成为当今主流程序设计语言。

目前，国内许多高校将C++语言作为计算机专业开设的第一门程序设计语言课程（CS1），许多自学者亦选择C++语言作为自学的第一门程序设计语言。而当前，尽管市面上有关C++语言的书籍很多，但适合作为教材，特别是适合没有任何程序设计基础的学生的教材几乎没有。译者本人在高校从事计算机专业C++语言的教学工作多年，对此深有体会。

本书作者D. S. Malik教授在Creighton大学已讲授了50多学期的计算机专业程序设计课程，本书就是作者从课程讲稿中改进、发展而来。作者在书中不只是列出C++语言的语法和给出相应例子，而是进一步揭示出隐藏在程序设计语言中的各种概念、思想之中的“为什么”，以激发读者学习程序设计语言的兴趣。因此，本书是一本非常优秀的教材，特别适合没有任何程序设计基础的学生使用。此外，学习计算机语言的目的就是用它来解决实际问题，本书还有两个有别于一般计算机语言书籍及语言参考书的显著特点。一是书中每章都带有程序范例，在程序范例中不仅仅提供完整的程序代码清单，而且还包括程序的输入/输出、问题分析和主要算法的设计等具体程序设计步骤，使得读者能够通过具体实例学会如何用面向对象方法来分析、设计并解决问题（完整的程序设计过程），以达到培养学生程序设计能力的目的。其二，在本书中还详细描述了如何用C++实现常用的数据结构，如表、链表、栈、队列、二叉树和图，以及多种查找和排序算法，使得计算机语言和数据结构的学习有机地结合起来，以便能够使用所学程序设计语言来解决实际问题。

本书主要由晏海华、蔡旭辉、常鸿、孙希坤、樊平、尚卫东、陈庆吉和朱经伟等翻译，并由晏海华和蔡旭辉等审校。

由于时间关系及水平所限，翻译不当或错误可能在所难免，恳请读者指正和见谅。

前　言

欢迎使用本书。本书专为两个学期（CS1和CS2）的C++课程而设计，相信它一定能给广大教师和学生以耳目一新的感觉。本课程应作为计算机专业的基础课程。主要目的是激发所有学习程序设计者的学习兴趣，而没有考虑其目前已经达到何种水平。因为对学生的程序设计兴趣的培养和激发是他们学好本课程的关键因素。笔者已经成功地讲授了50多个学期的计算机专业程序设计课程，本书正是从这些课程的讲稿中不断改进、发展而来的最终成果。

《C++编程——数据结构与程序设计方法》一书，刚开始只是为笔者所在学校当时使用的一些程序设计教程所写的简单例题、习题以及大篇幅的代码举例的补充材料。很快，这本补充材料内容多得可以单独成为一本教科书。事实上，本书中所采用的编排方式和讲述方法，会使读者感到概念清晰，易于理解。本书的选材都经过反复推敲，易于学生接受。本书中选用的大多数例题，也都经过课堂上与同学们相互探讨。

无论从事何种职业，将知识在实际中运用才是最重要的。正如厨艺学习者要按照菜谱练习烹饪，小提琴初学者要练习音阶一样，程序设计的初学者一定要亲自动手编写代码来解决问题。本书并不是一本C++参考手册，我们不只是列出C++语法并给出例子，而是进一步揭示出隐藏在各种概念、思想之中的“为什么”。在每一章中，这种关键性的“为什么”的问题都将得以解答。这种方法成为学好C++语言的有效手段。只有让学生们明白“为什么”，才能激发起他们学习的兴趣。

一般来说，C++初学者需要先掌握一门其他的程序设计语言。但是，本书假定读者没有任何程序设计基础。然而，例如代数学等高等数学知识还是必需要具备的。

本书内容

C++语言是从C语言演化发展而来的。现在，它不仅只是作为软件行业广泛使用的一种程序开发语言，为数众多的大专院校已将C++作为第一门程序设计语言课程列在教学计划中。C++融合了结构化程序设计和面向对象程序设计两种方法，本书将分别阐述。

本书应在计算机专业分两个学期讲授，CS1和CS2。第一个学期讲授前12章或13章，其余部分应在第二学期讲授。

1998年7月，正式通过了ANSI/ISO标准C++。本书在主要讲述ANSI/ISO标准C++的同时，也将介绍怎样使用标准C++编写程序。虽然标准C++和ANSI/ISO标准C++大部分语法是相同的，但在本书第8章中还是对ANSI/ISO标准C++支持而标准C++不支持的一些特性做了讨论。

第1章简要回顾了计算机和程序设计语言的发展史。读者在迅速阅读本章之后可以了解一些计算机软、硬件方面的基础知识。本章也同时介绍结构化程序设计和而向对象程序设计的基本概念。

在完成第2章之后，读者将了解C++的一些基本知识，并可以着手编写一些涉及到计算的复杂程序。输入/输出是所有程序设计语言的基础。输入/输出在本章中简要介绍，并在第3章中详细讨论。

第4章和第5章将介绍用来改变程序执行顺序的控制结构。第6章和第7章将讲述用户自定义函数。建议没有任何程序设计基础的读者要在第6章和第7章中多下些功夫。第7章中使用大量举例帮助读者理解参数传递和标识符作用域的概念。

第8章将讨论用户自定义的简单数据类型（枚举类型）、ANSI/ISO标准C++的名字空间机制以及string类型。早期版本的C语言不支持枚举类型。枚举类型的用途十分有限，其主要作用是增加程序的可读性。本书的编排也充分考虑到这一点，读者可以在第一次阅读本书时略过枚举类型部分，并不会感到不连贯。枚举类型部分可以放在后面仔细阅读。

第9章将详细说明数组，第10章将介绍并讨论递归。

第11章将介绍结构。本书中介绍的结构与C语言中的结构极为相似。实际上，本章也可以略过，因为结构并不是后续章节所必须知道的预备知识。

第12章开始讲述面向对象程序开发技术（OOP），并介绍类。本章前一部分将介绍类的定义和使用，后一部分介绍抽象数据类型（ADT）。本章也将指出为什么类成为C++中实现抽象数据类型的很自然的方法。第13章继续介绍面向对象程序设计（OOD）和面向对象程序开发的基础概念，并讨论继承和组成。本章解释了为什么类成为C++中实现抽象数据类型的一种很自然机制，并介绍C++是怎样支持面向对象程序开发的。本章同时也讨论了在解决具体问题时应该怎样去定义类。

第14章将详细讨论指针。在介绍指针的基础知识和怎样在程序中使用指针之后，本章着重讨论了含有指针类型数据成员类的特性，以及怎样避免由指针引起的错误。本章进一步指出了怎样利用动态数组存储和处理列表。本章同时还将讨论怎样通过虚函数来实现多态机制。

第15章继续讲述面向对象程序设计和面向对象程序开发，并将重点讨论C++的多态机制。本章具体地讨论了C++中的两种类型的多态机制——重载和模板。

第16章和第17章专门论述数据结构。第16章将详细讨论链表，第17章将讨论栈和队列。这两章所提供的程序代码对于该类型数据结构都是通用的。这两章充分使用了面向对象程序设计的基本概念。

第18章讨论了多种查找和排序算法。在指出这些算法原理的同时，本章还提供了关于这些算法执行效率的相关分析。算法分析使程序设计者能够在解决具体问题时知道应选用何种算法。本章同时介绍了多种排序算法，教师可以根据实际情况选择讲授其中的几种算法。

第19章介绍了二叉树。本章讨论和讲述了二叉树的基本性质和各种遍历算法。本章还介绍了一种特殊的二叉树——二叉查找树，介绍并举例说明了二叉查找树的查找、插入和删除算法。本章同时提供了二叉树的非递归遍历算法。为了增加遍历算法的灵活性，本书介绍了怎样构造和传递函数指针参数。

第20章讨论了图论中的一些算法。本章在介绍了图论中的一些基本术语后，讨论了图在计算机存储器中的表示方法；同时也讨论了最短路径和最小生成树等图的遍历算法。

C++自带了功能强大的类库——标准模板库（STL），其中的数据结构和算法可以有效地在应用程序中广泛使用。第21章详尽地讲述了标准模板库。在介绍了标准模板库三种基本的组成部分后，本章说明了在程序中该如何使用顺序容器；同时也讨论了栈和队列等具体容器。本章后半部分讲述如何在程序中使用各种标准模板库算法。本章的内容相当多，如果时间有限，教师可以只讲授顺序容器、栈类、队列类和其他一些算法。

附录A给出了C++中的保留字。附录B给出了C++中运算符的优先级和结合律。附录C给出了ASCII码字符集和EBCDIC码。附录D给出了C++中可以重载的运算符集。附录E给出了ANSI/ISO标准C++和标准C++头文件的习惯命名方法。附录F讨论了一些广泛使用的程序库和相应的标准C++的头文件名。通过运行附录G中的程序，可以知道所使用系统中的各种内置数据类型所

占用的存储空间大小。附录H列出了深入学习C++语言所用到的参考资料。附录I给出了本书中部分习题的参考答案。

本书特点

本书中每一章都有如下特点，读者既可以在本书的指导下学习，也可以根据本人的实际情况选择性地学习：

- 每章的“本章要点”简述了在本章中要详细讨论的C++程序设计概念。
- “注意”强调了各章节中的概念要点。
- 375张插图详尽地解释了难于理解的概念。
- 统一编号的例题可以帮助读者理解关键概念及相关程序代码。例题中的程序代码行编号以便于参照，同时给出程序运行举例和代码行执行的结果和注释。
- 每章章尾的程序范例都是完整的程序代码。这些程序范例中包括输入、输出、问题分析和算法设计，以及完整的程序代码清单。
- 小结总结了每章的重要概念。
- 练习用来巩固学习成果，并可以用来检查学生是否真正地理解了该章的主要内容。
- 编程练习培养学生亲自动手编写C++程序的能力。

自始至终，本书在适当的时机引入相应的概念，便于学生学习。本书的写作风格不拘泥于形式，通俗易懂，很适于在课堂上讲授。在介绍重要概念之前，本书将解释为什么某些元素是必需的。然后，使用举例和小程序来讲述概念。

本书中每一章都有两种类型的程序。第一种是带有编号的范例（例如：例4.1），它们都是一些小程序，用来帮助理解重点概念。这种小程序的每一行代码都有编号，并给出了程序运行范例和逐行解释。

本书中提供了大量的程序范例，这些程序范例是本书的精髓。

本书中的程序都经过精心设计，便于读者理解。每一个程序范例都有问题分析和算法设计。然后，每一步算法都给出相应的C++程序代码。为了帮助学生提高解决问题的能力，这些程序代码详细地告诉读者怎样用C++程序设计语言实现程序设计的基本概念和基本思想。强烈建议读者仔细阅读程序范例，以便提高学习效率。

每一章后面的小结部分可以帮助读者巩固学习成果。在阅读每一章后，读者可以迅速地浏览本章的重点知识，然后使用后面提供的练习来测试学习效果。许多学生都将在考试前阅读小结中的要点，作为一种快速复习的方法。

本书提供的所有程序源代码都通过编译，保证符合ANSI/ISO标准C++和标准C++的标准。可以使用Microsoft Visual C++ 6.0，Borland C++ Builder 5开发工具或是Metrowerks CodeWarrior编译这些程序。

教学工具

教师可以在课堂讲授本书的同时使用下列辅助教学工具。这些辅助教学工具附在光盘中，教师可以根据书后所附的“教学支持说明”，向有关出版商索取。

电子教师手册 该教师手册包括以下内容：

- 附加的授课资料，帮助教师做好课前准备，包括对授课内容的建议。
- 每章后面练习的参考答案，包括编程练习。

ExamView 本书提供的ExamView是一套功能强大的考试软件包。教师可以使用它进行以下三种方式的考试：试卷式，（局域网）计算机上考试和Internet上考试。ExamView包括大量关于本书中涉及到的内容的试题，可帮助学生获得详细的学习指南。计算机上和Internet上的考试工具可以使学生们在自己的计算机上完成考试，并且可以节省教师的工作时间，因为这两种考试是计算机评分的。

PowerPoint幻灯片 本书的每一章都有相应的Microsoft PowerPoint幻灯片。这些幻灯片可以作为辅助教学工具在课堂上放映使用，也可以放在网络文件服务器上供学生复习时使用，还可以打印出来分发给学生。教师在授课时可以根据情况增加自己的幻灯片页面。

远程学习 Course Technology公司以在WebCT上提供在线教育服务而闻名，而黑板教学可以使学生们以最详尽、最大限度交互的方式进行学习。当教师为其部分课程提供网上教学方式时，不仅可以增加很多内容，如自测考题、相关连接和词汇表，更重要的是培养了学生了解21世纪获取重要信息资源的方法。希望各位教师能给其讲授的绝大多数课程同时提供在线教学和离线教学两种方式。 如果需要进一步了解怎样为讲授的课程提供远程教学方式，请与所在地的Course Technology公司销售代表联系。

程序源代码 程序源代码同时以ANSI/ISO标准C++和标准C++两种格式给出。它们既可以在网站www.course.com上获得，又可以在教学工具光盘（参见本书后面所附的“教学支持说明”）中获得。运行某些程序代码的输入文件也随源代码提供给读者。但是，运行程序时，这些输入文件需要存储在软盘并插入到计算机的软驱中。

解答文件 解答文件同时以ANSI/ISO标准C++和标准C++两种格式给出。既可以在网站www.course.com上获得，又可以在教学工具光盘中获得。运行某些程序练习代码的输入文件也随源代码提供给读者。但是，运行程序时，这些输入文件需要存储在软盘并插入到计算机的软驱中。

致谢

许多人为本书的成功出版做出了重要贡献。在这里，笔者必须要向他们表示衷心感谢。首先，我要向William Newman博士表示感谢，他负责审校第二稿手稿，找出并改正错误，并且提出了大量的改进意见。必须要向我的学生们表示感谢，他们在我准备出版该书时主动地告诉哪些部分需要重新措辞以便于理解。要向长期以来不断支持本书出版项目的S. C. Cheng教授、John N. Mordeson教授和Vasant Raval教授表示感谢。Randall L. Crist博士，他的办公室在我办公室的隔壁，在审校工作中随时帮我检查书稿，在这里向他表示感谢。必须要向Creighton大学技术转让办公室主任Lee I. Fenicle表示感谢，他积极地参与、支持此项工作，并在我感到灰心时不断地鼓励我。同时，也要向参加本书初稿审阅工作的那些不知名的工作人员表示衷心感谢，他们为本书的改进工作提出了许多宝贵意见。

在这里，向以下参加审阅的工作人员表示诚挚谢意。他们一丝不苟地审阅了本书当前版本的每一章节，并为本书的改进提出了宝贵意见。他们是：Wisconsin大学（位于River Falls）的Ahmad Abuhejleh，North Carolina大学（位于Charlotte）的C. Michael Allen，Wisconsin大学（位于Milwaukee）

的Paul Ambrose，Capitol学院的Julie Anderson，Kentucky大学的Carol Hannahs和Austin Community学院的Judy Scholl。此外，必须要向如下参加审阅本书的工作人员表示感谢，他们是：St. Edward's大学的James McGuffee和Austin Community学院的Thomas Murtagh。这些参加本书审阅的工作人员会认识到他们的建议没有被忽视。事实上，正是这些建议为本书的改进做出了重要贡献。感谢设计编辑Susan Gilbert，她认真编辑本书，并及时定版。如果没有高级编辑Jennifer Muroff的精心设计，本书也不会呈现在读者面前。在这里，要向Jennifer Muroff和制作编辑Aimee Poirier，以及Course Technology公司负责对代码进行认真测试的质量控制部门的工作人员表示诚挚的谢意。

本书献给我的父母！在他们祝福下本书得以顺利出版，我感谢他们。

最后，要感谢一直支持我工作的妻子Sadhana和女儿Shelly。特别是Shelly，在我编写本书遇到极大困难并感到万分沮丧的时候不断地鼓励我，并帮助我审阅修改的内容。Shelly总是能在我承担这种棘手的工作时为我带来欢乐。

欢迎读者对本书提出建议。建议可以发送至电子邮箱：malik@creighton.edu。

目　录

第 1 章　计算机和程序设计语言概述 1
1.1　简介 1
1.2　计算机发展史简述 1
1.3　计算机系统的组成 2
1.3.1　硬件 2
1.3.2　软件 3
1.4　计算机语言 4
1.5　程序设计语言的发展 4
1.6　高级语言程序的处理过程 5
1.7　按问题分析 – 编码 – 执行循环的程序设计 6
1.8　面向对象程序设计 9
1.9　ANSI/ISO 标准 C++ 10
1.10 小结 11
1.11 练习 12

第 2 章　C++ 基础 13
2.1　C++ 程序基础 14
2.2　数据类型 16
2.2.1　简单数据类型 16
2.2.2　浮点数据类型 18
2.2.3　字符串数据类型 19
2.3　算术运算符及其优先级 20
2.3.1　运算符优先级 22
2.4　表达式 22
2.4.1　混合表达式 23
2.4.2　类型转换（强制转换）........ 24
2.5　输入 25
2.5.1　给常量和变量分配内存 25
2.5.2　将数据存储到变量中 27
2.6　增量运算符和减量运算符 33
2.7　输出 34
2.8　预处理指令 39
2.8.1　cin 和 cout 与名字空间 40
2.8.2　在程序中使用 string 数据类型 41
2.9　程序的风格和形式 41

2.9.1 main 函数 42
2.9.2 语法 43
2.9.3 空格符的使用 43
2.9.4 分号、大括号和逗号的使用 43
2.9.5 语义规则 44
2.9.6 形式和风格 44
2.9.7 说明文档 45
2.10 其他赋值语句 46
2.11 程序范例：长度转换 46
2.12 程序范例：零钱换整 50
2.13 小结 52
2.14 练习 54
2.15 编程练习 59
第3章 输入/输出 61
3.1 I/O 流和标准 I/O 设备 61
3.1.1 cin 和析取操作符>> 62
3.2 在程序中使用预定义函数 65
3.2.1 cin 和 get 函数 66
3.2.2 cin 和 ignore 函数 67
3.2.3 putback 函数和 peek 函数 68
3.2.4 I/O 流变量和 I/O 函数间的点符号(.) 70
3.3 输入失败 70
3.3.1 clear 函数 72
3.4 输出和格式化输出 73
3.4.1 setprecision 控制符 73
3.4.2 fixed 控制符 74
3.4.3 showpoint 控制符 74
3.4.4 setw 控制符 76
3.5 其他的格式输出 78
3.5.1 fill 和 setfill 78
3.5.2 left 和 right 控制符 79
3.5.3 flush 函数 81
3.5.4 输入/输出 string 类型数据 82
3.6 文件输入/输出 83
3.7 程序范例：电影票销售和慈善捐赠 85
3.8 程序范例：学生成绩 89
3.9 小结 92
3.10 练习 93
3.11 编程练习 94

第4章　控制结构Ⅰ（选择）……97
4.1　控制结构……97
4.2　关系运算符……98
4.2.1　关系运算符和简单数据类型……99
4.2.2　关系运算符和string类型……100
4.3　逻辑运算符和逻辑表达式……100
4.3.1　运算符优先级……101
4.3.2　优化（短路）计算……104
4.3.3　int数据类型和逻辑（布尔）表达式……105
4.3.4　bool数据类型和逻辑（布尔）表达式……105
4.4　选择：if和if…else……106
4.4.1　单路选择……106
4.4.2　双路选择……108
4.4.3　复合（块）语句……110
4.4.4　多重选择：嵌套if……111
4.4.5　if…else语句和系列if语句的比较……113
4.4.6　使用伪代码进行程序开发、测试和调试程序……114
4.4.7　输入失败和if语句……116
4.4.8　恒等运算符（==）和赋值运算符（=）之间容易产生的混淆……119
4.4.9　条件运算符（?:）……120
4.5　switch结构……120
4.6　使用assert函数终止程序……125
4.7　程序范例：有线电视公司的计费程序……127
4.8　小结……131
4.9　练习……132
4.10　编程练习……135
第5章　控制结构Ⅱ（循环）……138
5.1　为什么需要循环……138
5.2　while循环（重复）结构……139
5.2.1　情形1：计数器控制的while循环……141
5.2.2　情形2：结束标记控制的while循环……142
5.2.3　情形3：标志控制的while循环……146
5.2.4　情形4：EOF控制的while循环……146
5.2.5　eof函数……147
5.3　程序范例：活期账户余额……148
5.4　程序范例：Fibonacci数列……155
5.5　for循环（重复）结构……158
5.6　程序范例：数字分类……161
5.7　do…while循环（重复）结构……164
5.8　break和continue语句……166

5.9 嵌套控制结构 168
5.10 小结 169
5.11 练习 170
5.12 编程练习 178
第6章 用户自定义函数Ⅰ 180
6.1 标准（预定义）函数 180
6.2 用户自定义函数 182
6.3 带有返回值的函数 183
6.3.1 return语句 185
6.3.2 函数原型 186
6.3.3 执行过程 190
6.4 程序范例：最大值 191
6.5 程序范例：有线电视公司的计费程序 192
6.6 小结 196
6.7 练习 197
6.8 编程练习 201
第7章 用户自定义函数Ⅱ 203
7.1 无返回值函数 203
7.1.1 不带参数的无返回值函数 203
7.1.2 带有参数的无返回值函数 205
7.1.3 引用参数 208
7.2 值参数、引用参数和内存地址 211
7.3 引用参数和带有返回值的函数 221
7.4 标识符的作用域 221
7.5 全局变量的副作用 224
7.6 静态变量和自动变量 224
7.7 函数重载简介 226
7.8 带有默认参数的函数 226
7.9 程序范例：数字分类 228
7.10 程序范例：数据比较 232
7.11 小结 240
7.12 练习 241
7.13 编程练习 245
第8章 用户自定义简单数据类型、名字空间和string类型 248
8.1 枚举类型 248
8.1.1 定义变量 249
8.1.2 赋值 249
8.1.3 枚举数据类型上的运算 250
8.1.4 枚举数据类型和循环结构 250

8.1.5 枚举数据类型的输入/输出 251
8.1.6 函数和枚举类型 252
8.1.7 在定义枚举数据类型同时定义变量 253
8.1.8 匿名数据类型 253
8.1.9 typedef 语句 254
8.2 程序范例：剪刀、石头、布的游戏 254
8.3 名字空间 262
8.4 string 数据类型 266
8.4.1 其他 string 类型数据上的操作 269
8.5 程序范例：Pig Latin 字符串 273
8.6 小结 278
8.7 练习 279
8.8 编程练习 282
第 9 章 数组和字符串 284
9.1 数组 285
9.1.1 访问数组元素 286
9.1.2 处理一维数组 287
9.1.3 数组下标越界 290
9.1.4 在定义时初始化数组 290
9.1.5 数组处理中的一些限制 291
9.1.6 数组作为函数参数 292
9.1.7 整数类型和数组下标 295
9.2 C-string（字符数组） 296
9.2.1 字符串比较 298
9.2.2 输入输出字符串 298
9.2.3 字符串输入 299
9.2.4 字符串输出 300
9.2.5 在执行时指定输入输出文件 300
9.3 平行数组 301
9.4 二维数组和多维数组 301
9.4.1 访问数组元素 302
9.4.2 在定义时初始化二维数组 303
9.4.3 二维数组和枚举数据类型 304
9.4.4 处理二维数组 305
9.4.5 将二维数组作为参数传递给函数 309
9.4.6 字符串数组 310
9.4.7 定义二维数组的另一种方法 311
9.5 多维数组 312
9.6 程序范例：代码校验 313
9.7 程序范例：文本处理 318

9.8 小结 322
9.9 练习 323
9.10 编程练习 327

第10章 递归 332
10.1 递归定义 332
10.2 使用递归解决问题 334
10.2.1 Hanoi塔：分析 341
10.3 递归与迭代的比较 342
10.4 程序范例：将二进制数转化成十进制数 342
10.5 程序范例：将十进制数转化成二进制数 345
10.6 小结 347
10.7 练习 348
10.8 编程练习 349

第11章 结构 351
11.1 结构 351
11.1.1 访问结构成员 352
11.1.2 赋值 354
11.1.3 比较（关系运算） 354
11.1.4 输入/输出 355
11.1.5 结构变量和函数 355
11.1.6 数组和结构的比较 356
11.1.7 含数组的结构 356
11.1.8 结构数组 357
11.1.9 嵌套结构 358
11.2 程序范例：销售数据分析 361
11.3 小结 373
11.4 练习 374
11.5 编程练习 375

第12章 类和数据抽象 378
12.1 类 378
12.1.1 类的内建运算符 382
12.1.2 类的作用域 382
12.1.3 函数和类 383
12.1.4 成员函数的实现 383
12.1.5 类的公有成员和私有成员的顺序 390
12.1.6 构造函数 392
12.1.7 调用构造函数 393
12.1.8 对象数组和构造函数 397
12.1.9 析构函数 397

12.2 数据抽象、类和抽象数据类型 397
12.3 结构和类的比较 399
12.4 信息隐藏 400
12.5 可执行代码 402
12.6 程序范例：糖果自动销售机 405
12.7 小结 416
12.8 练习 417
12.9 编程练习 419
第13章 继承和组成 421
13.1 继承 421
13.1.1 基类成员函数的重定义 423
13.1.2 基类和派生类的构造函数 426
13.1.3 派生类的头文件 432
13.1.4 头文件的多重包含 432
13.1.5 C++ 流类 433
13.1.6 类的受保护成员 434
13.1.7 公有继承、受保护继承和私有继承 434
13.2 组成 437
13.3 面向对象程序设计（OOD）和面向对象编程（OOP） 441
13.3.1 标识类、对象和操作 442
13.4 程序范例：成绩单 443
13.5 小结 460
13.6 练习 460
13.7 编程练习 466
第14章 指针、类、表及虚函数 469
14.1 指针数据类型与指针变量 469
14.1.1 声明指针变量 469
14.2 取地址运算符 470
14.3 递引用运算符 470
14.4 类、结构及指针变量 474
14.5 初始化指针变量 476
14.6 动态变量 477
14.7 指针变量的运算 478
14.8 动态数组 479
14.9 浅拷贝、深拷贝及指针 481
14.10 类与指针：一些特性 483
14.10.1 析构函数 483
14.10.2 赋值运算符 484
14.10.3 拷贝构造函数 485

14.11 基于数组的表 491
14.11.1 拷贝构造函数 496
14.11.2 搜索 497
14.11.3 插入 498
14.11.4 删除 498
14.12 继承、指针和虚函数 500
14.12.1 类与虚析构函数 505
14.13 取地址运算符和类 505
14.14 小结 507
14.15 练习 509
14.16 编程练习 514
第 15 章 重载与模板 515
15.1 为什么需要重载运算符 515
15.2 运算符重载 516
15.2.1 运算符函数的语法 516
15.2.2 重载运算符：一些限制 516
15.2.3 this 指针 517
15.2.4 类的友元函数 521
15.2.5 运算符函数作为成员函数和非成员函数 523
15.2.6 重载双目运算符 524
15.2.7 重载流插入和流析取运算符 530
15.2.8 重载赋值运算符 533
15.2.9 重载单目运算符 538
15.2.10 重载增量运算符和减量运算符 538
15.2.11 运算符重载：成员与非成员的比较 543
15.2.12 类和指针数据成员（再叙）......543
15.2.13 运算符重载：结束语 543
15.3 程序范例：clockType 544
15.4 程序范例：复数 549
15.5 重载数组下标运算符 552
15.6 程序范例：newString 554
15.7 函数重载 559
15.8 模板 559
15.8.1 函数模板 559
15.8.2 类模板 561
15.8.3 基于数组的表（再叙）......562
15.9 小结 568
15.10 练习 570
15.11 编程练习 573

第 16 章　链表 578
16.1　链表 578
16.1.1　链表：一些属性 579
16.1.2　插入及删除节点 581
16.1.3　创建链表 584
16.2　作为抽象数据类型的链表 587
16.3　有序链表 598
16.4　双向链表 607
16.5　程序范例：影碟店 614
16.6　小结 629
16.7　练习 630
16.8　编程练习 633

第 17 章　栈和队列 636
17.1　栈 636
17.1.1　栈操作 639
17.2　使用数组实现栈 639
17.2.1　栈头文件 647
17.3　程序范例：最高 GPA 651
17.4　使用链表实现栈 654
17.5　栈应用：后缀表达式计算器 665
17.6　消除递归：逆向打印链表的非递归算法 672
17.7　队列 677
17.7.1　队列操作 677
17.7.2　使用数组实现队列 678
17.7.3　用链表实现队列 684
17.7.4　从类 linkedListType 派生队列 687
17.8　队列应用：模拟 689
17.8.1　客户 690
17.8.2　服务者（器） 693
17.8.3　服务者（器）表 696
17.8.4　等待客户队列 699
17.8.5　主程序 701
17.9　小结 706
17.10　练习 707
17.11　编程练习 710

第 18 章　查找和排序算法 712
18.1　查找算法 712
18.1.1　顺序查找 714
18.1.2　有序表 715

18.1.3 折半查找 716
18.1.4 有序表的插入 719
18.2 渐近表示法 721
18.2.1 基于比较的查找算法的下界 722
18.3 排序算法 722
18.3.1 选择排序：基于数组表 723
18.3.2 分析：选择排序 727
18.4 插入排序：基于数组表 727
18.5 插入排序：基于链表 731
18.5.1 分析：插入排序 734
18.5.2 基于比较的排序算法的下界 735
18.6 快速排序：基于数组表 735
18.6.1 分析：快速排序 741
18.7 归并排序：基于链表 741
18.7.1 拆分 743
18.7.2 归并 744
18.7.3 分析：归并排序 747
18.8 程序范例：选举结果统计 748
18.9 小结 765
18.10 练习 766
18.11 编程练习 767
第19章 二叉树 769
19.1 二叉树简介 769
19.1.1 拷贝树 774
19.1.2 二叉树遍历 774
19.1.3 实现二叉树 777
19.2 二叉搜索树 783
19.2.1 二叉搜索树：分析 793
19.3 非递归二叉树遍历算法 794
19.3.1 非递归的中序遍历 795
19.3.2 非递归的前序遍历 796
19.3.3 非递归的后序遍历 797
19.4 二叉树遍历与函数参数 799
19.5 程序范例：影碟店 801
19.6 小结 810
19.7 练习 811
19.8 编程练习 813
第20章 图 814
20.1 简介 814
20.1.1 图的定义和符号 814

20.2 图的表示 817
20.2.1 邻接矩阵 817
20.2.2 邻接表 817
20.3 图操作 818
20.4 图遍历 822
20.4.1 深度优先算法 823
20.4.2 广度优先遍历 824
20.5 最短路径算法 826
20.5.1 最短路径 827
20.6 最小生成树 831
20.7 小结 838
20.8 练习 838
20.9 编程练习 840
第21章 标准模板库 842
21.1 STL的组成部分 842
21.1.1 容器类型 842
21.1.2 顺序容器 843
21.1.3 顺序容器：向量 843
21.1.4 所有容器公共的成员函数 847
21.1.5 顺序容器公共的成员函数 847
21.1.6 copy算法 848
21.1.7 顺序容器：双端队列 851
21.1.8 顺序容器：表 853
21.2 迭代器 857
21.2.1 迭代器的类型 857
21.2.2 流迭代器 860
21.3 关联容器 861
21.3.1 关联容器：集合和多重集合 861
21.3.2 声明集合和多重集合关联容器 861
21.3.3 集合/多重集合中元素的插入和删除 862
21.4 容器适配器 864
21.4.1 栈 865
21.4.2 队列 866
21.5 容器、相关的头文件以及迭代器 867
21.6 算法 867
21.6.1 STL算法分类 868
21.6.2 函数对象 869
21.6.3 插入迭代器 873
21.6.4 STL算法 875
21.6.5 函数fill和fill_n 875

21.6.6 函数 generate 和 generate_n 876
21.6.7 函数 find，find_if，find_end 和 find_first_of 877
21.6.8 函数 remove，remove_if，remove_copy 和 remove_copy_if 881
21.6.9 函数 replace，replace_if，replace_copy 和 replace_copy_if 884
21.6.10 函数 swap，iter_swap 和 swap_ranges 887
21.6.11 函数 search，search_n，sort 和 binary_search 889
21.6.12 函数 adjacent_find，merge 和 inplace_merge 892
21.6.13 函数 reverse，reverse_copy，rotate 和 rotate_copy 895
21.6.14 函数 count，count_if，max_element，min_element 和 random_shuffle 897
21.6.15 函数 for_each 和 transform 900
21.6.16 函数 includes，set_intersection，set_union，set_difference 和 set_symmetric_difference 902
21.6.17 函数 accumulate，adjacent_difference，inner_product 和 partial_sum 909
21.7 小结 913
21.8 练习 915
21.9 编程练习 917
附录 A 保留字 918
附录 B 运算符优先级 919
附录 C 字符集 920
附录 D 运算符重载 922
附录 E ANSI/ISO 标准 C++ 和标准 C++ 中头文件命名规则 923
附录 F 头文件 924
附录 G 系统中数据类型长度 930
附录 H 参考文献 931
附录 I 部分参考答案 932

第1章　计算机和程序设计语言概述

本章要点：

- 了解各种不同类型的计算机
- 了解计算机系统中的软件部分和硬件部分
- 了解计算机语言
- 了解计算机程序设计语言的发展过程
- 了解高级程序设计语言
- 理解什么是编译器以及作用
- 理解计算机怎样处理和运行使用高级程序设计语言编写的程序
- 理解什么是算法以及解决实际问题的方法
- 理解结构化程序设计和面向对象程序设计的基本方法
- 理解标准C++和ANSI/ISO标准C++

1.1　简介

现在我们生活在一个以光速传递信息的时代。通过计算机，信息技术的发展日新月异，并且极大地改变了我们的生活方式和沟通方式。像Internet这种几年前还不为人们所知的新名词，现在也已经家喻户晓了。通过计算机，可以随时与想交流的人互相通信。申请工作时，再也不必通过邮寄信件的方式投送简历了。在很多情况下，可以在Internet上申请工作。还可以实时地了解到各支股票的表现，并在几秒钟内完成交易。小学生们可以经常在Internet上冲浪，使用计算机完成他们的课程任务。学生们再也不用通过敲击打字机或是用手写去完成他们的论文了。取而代之的是使用功能强大的文字处理软件来完成论文。许多人使用计算机管理他们的财务以保持收支平衡。

上述都是计算机极大地改变我们日常生活的例子。而这些都是通过不同的软件——计算机程序来实现的。例如：文字处理软件就是一种程序，通过它可以完成论文，设计精美的个人简历，甚至可以用它编写一本书。本书就是在功能强大的文字处理软件的帮助下完成的。没有软件，计算机只是一堆废铜烂铁。然而，计算机软件是用程序设计语言编写的。C++正是一门非常适合用来编写完成特定功能软件的程序设计语言。

直到20世纪90年代初，教师在讲授程序设计语言课程之前，还需要花几个星期的时间讲述如何使用计算机。现在的情况大不相同了，高中毕业的学生就已经可以熟练地使用计算机了。本书的主要目的是讲述怎样使用一种叫做C++的程序设计语言来编写程序，而不是介绍该如何使用计算机。然而，在开始程序设计以前，还是很有必要介绍计算机的一些基本术语和组成部分。本章将简要介绍计算机的主要组成部分，计算机语言发展史和通过编写计算机程序来解决问题的一些基本思想。

1.2　计算机发展史简述

在20世纪50年代，计算机只是供极少数人使用的庞然大物。包括外科手术、记账、文字处理和计算等所有工作都可以在没有计算机的参与下完成。在20世纪60年代，只有大公司才能买得起的价值数百万美元的计算机问世了。这些计算机不仅个头庞大，而且只有计算机专家才能使用。到20世纪70年代

代中期，计算机体积变小而且价格也降低了很多。到20世纪90年代中期，计算机的价格降到了普通百姓都可以接受的水平。20世纪90年代末，小型计算机的价格更加便宜，速度也有大幅提高。虽然计算机可以分为大型、中型和小型等几种类别，但其基本组成是相同的。

1.3 计算机系统的组成

计算机是一种具有执行命令功能的电子设备。计算机可以实现的基本功能包括：输入（得到数据）、输出（显示结果）、存储、执行算术和逻辑运算。

在当今的个人计算机市场上销售的计算机通常标明如下信息：1.4 GHz奔腾4处理器、128 MB内存、40 GB硬盘、19英寸[①] SVGA显示器，预装操作系统、游戏、百科全书、文字处理和财务管理等应用程序。这些计算机的组成部分可以分成两大类：硬件和软件。诸如1.4 GHz奔腾4处理器、128 MB内存、40 GB硬盘和19英寸SVGA显示器等属于硬件，而操作系统、游戏、百科全书和应用程序属于软件。首先，先让我们了解什么是硬件。

1.3.1 硬件

硬件主要包括以下几个部分：中央处理器（CPU，Central Processing Unit）、内存（MM，Main Memory）也称随机存储器（RAM，Random Access Memory）、输入/输出设备（Input/Output Device）、外部存储器（Secondary Storage）。键盘、鼠标和外部存储器都是常见的输入设备，而显示器屏幕、打印机和外部存储器都是常见的输出设备。下面将详细介绍这些组成部分。

中央处理器

中央处理器是计算机的控制中心，也是个人计算机中价值最高的硬件。CPU速度越快，计算机的性能就越好。CPU由以下几个部分组成：控制单元（CU，Control Unit）、程序计数器（PC，Program Counter）、指令寄存器（IR，Instruction Register）、算术逻辑单元（ALU，Arithmetic Logic Unit）和累加器（ACC，Accumulator），图1.1说明了CPU的组成。

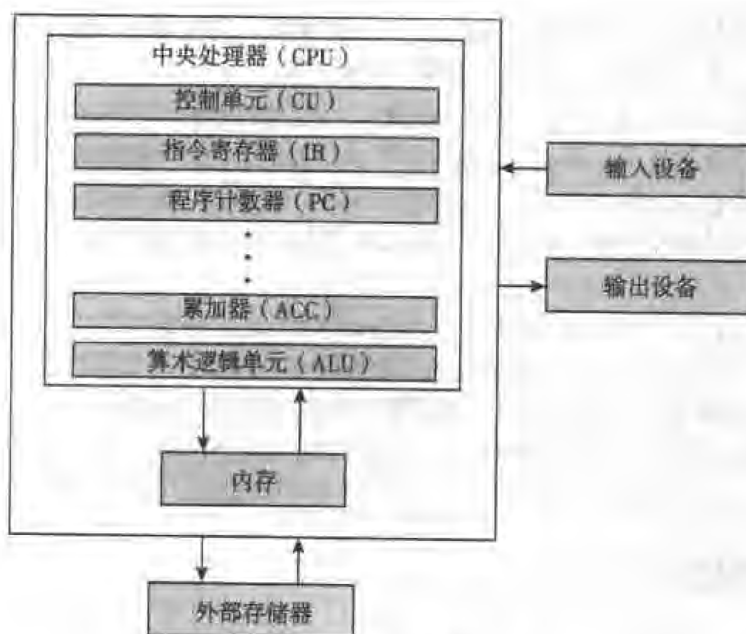


图1.1 计算机的硬件组成

① 1英寸=2.54 cm——编者注。

- 与CPU相似，控制单元是CPU的控制中心。它有三个主要的作用：获取并解释指令、控制信息（数据和指令）在内存中的存取，以及控制CPU内部各组成部分的工作。
- 程序计数器指向CPU下一步要处理的指令。
- 指令寄存器存储CPU当前正在处理的指令。
- 算术逻辑单元执行所有的算术和逻辑运算。
- 累加器存储经过算术逻辑单元计算后的数据。

内存

内存直接与CPU相连。程序只有进入内存，才能被运行。同样，只有将程序所用到的所有数据装入内存，这些数据才能被处理。当计算机关闭后，内存中的所有数据也将永久地丢失了。

内存是由按顺序编号的一系列存储单元组成的。在内存中，每一个存储单元都有一个惟一的地址。通过地址，可以方便地在内存单元中存取信息。图1.2是含有100个存储单元的内存示意图。

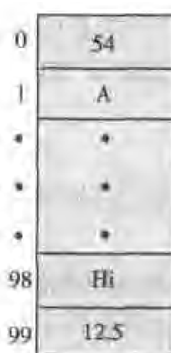


图1.2 有100个存储单元的内存

在当今的计算机中，内存通常有多达数百万到数十亿个内存单元。在图1.2中所示的内存中存有数据，这些数据既可以是程序指令也可以是程序所要使用的数据。

外部存储器

因为程序的指令和数据都要在程序运行之前装入内存，并且一旦计算机关闭，内存中所有的东西都将丢失，所以需要有一个可以不丢失数据的设备来存储这些指令和数据。这种可以永久地存储数据（除非设备本身不可用或是这些数据已被改写）的设备称为外部存储器。为了可以将内存中的数据传送给外部存储器，必须使它们彼此直接相连。典型的外部存储器包括：硬盘、软盘、Zip盘、光盘（CD-ROM）和磁带。

输入/输出设备

计算机必须要具备读入程序指令及数据并及时显示计算结果的能力。这种负责读入程序指令和数据的设备称为输入设备。键盘、鼠标和外部存储器都是输入设备。而负责显示计算结果的设备称为输出设备。显示器和外部存储器都是输出设备。

1.3.2 软件

软件是可以完成特定任务的程序。例如：可以使用文字处理程序写信、写论文乃至写一本书。程序分为两类：系统程序和应用程序。

系统程序是用来控制计算机的。计算机在启动时首先装载的系统程序称为操作系统。操作系统的作用是监控计算机中所有的活动和提供服务，没有操作系统的计算机什么也不能干。操作系统提供的服务包括：内存管理、输入/输出管理和外部存储器管理等。通过操作系统对外部存储器的管理，可以方便地对外部存储器中的数据实现存取。有一点必须指出，操作系统本身也是用程序设计语言编写的。

应用程序是用来执行特定任务的。文字处理程序、电子表格程序和游戏程序都属于应用程序。所有的应用程序都是用程序设计语言来编写的。应用程序要在操作系统程序之上才能运行。

1.4 计算机语言

在键盘上键入字母A并看到它显示在屏幕上时，有没有想过实际在计算机存储器里存储的究竟是什么？计算机所采用的是何种语言？计算机将在键盘上键入的字母存储成何种形式？

有一点要提醒读者的是，计算机只是一种电子设备，计算机内能处理的也只是一些电信号。电信号分为两种：模拟信号和数字信号。模拟信号以连续波的形式存在，用来表示诸如声音一类的数据。例如录音带就是用模拟信号来记录数据的。数字信号则采用0和1序列来记录信息。0代表低电压，而1代表高电压。用数字信号记录数据比用模拟信号更加可靠，并且可以精确地在不同的设备上复制。不知是否留心，拷贝后录音带的质量不如原带质量好。

既然在计算机中处理的是数字信号，那么计算机的语言，称为机器语言（Machine Language），自然也就是0和1序列。数字0或1被称为二进制数字（Binary Digit），或者称为位（Bit）。这种0和1序列有时就被称为二进制代码（Binary Code）。

位（Bit）：二进制数0或1。

连续的8个位称为1个字节（Byte）。键盘上的所有字母、数字和特殊符号（如“{”或“*”）都用一序列位惟一地表示。

在个人计算机上最常见的编码方案是7位的美国标准信息交换代码（ASCII，American Standard Code for Information Interchange）。ASCII码字符集由编号从0到127的128个字符组成。因此，在ASCII字符集中处于第一位置的字符的编号是0，第二个是1，以此类推。在这种方案中，字母A的编码是1000001。事实上，字符A是ASCII码字符集中第66个字符，但是它的编号是65，因为第一个字符的编号是0。而且，1000001是十进制数65的二进制数表示。数字字符3的ASCII编码是0110011。附录C中给出了完整的可打印的ASCII字符集。

在计算机中，所有的字符都是由一个连续的8个位（即字节）表示的。因为ASCII码是7位编码，所以必须要在其最左边加上一个0以满足计算机字符编码的要求。因此，在计算机内部，字母A表示为01000001，数字字符3表示为00110011。

当然，还有其他的编码方案，例如被IBM所采用的EBCDIC码和近几年制订的统一编码（Unicode）。EBCDIC码由256个字符组成，而Unicode则由65 536个字符组成。Unicode字符的存储需要两个字节。

1.5 程序设计语言的发展

最基本的计算机语言，机器语言，使用二进制位来表示程序指令。即便是绝大多数计算机完成的都是相近的功能，不同的计算机的设计者也可能会采用不同的二进制代码集来表示程序指令。因此，不同的计算机使用的机器语言并不一定相同。惟一相同的一点是，现代计算机都是以二进制代码的形式来存储和处理数据的。

在早期的计算机中，程序都是用机器语言编写的。为了弄清怎样使用机器语言来编写程序指令，假设我们要用到下面的周工资计算公式：

```
wages（工资）= rates（每小时工资）× hours（工作小时）
```

进一步假定，二进制代码100100表示装载，100110代表计算，100010代表存储。如果使用机器语言来计算周工资，需要如下的指令序列：

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

为了使用机器语言计算周工资，程序员必须要记住各种操作的机器指令编码。同时，为了读取数据，程序员还要记住所有数据在内存中的地址。这种需要记住大量具体编码来编写程序的方法不仅难于实现，而且极易出错。

汇编语言（Assembly Language）的出现简化了程序员的工作。在汇编语言中，采用便于记忆的方法（Mnemonic）定义程序指令，表 1.1 给出了汇编语言程序指令和相应的机器指令。

表 1.1　汇编语言指令和相应的机器指令范例

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

使用汇编语言的计算周工资程序如下：

```
LOAD   rate
MULT   hours
STOR   wages
```

很明显，使用汇编语言编写程序要比使用机器语言容易得多。然而，计算机并不能直接执行由汇编语言编写的程序。汇编语言指令必须要首先转换成机器语言指令。一个叫做汇编语言编译器（Assembler）的程序完成这种转换。

汇编语言编译器　是一种将汇编语言指令转换成为相应的机器语言指令的程序。

从机器语言到汇编语言，使程序的编写变得容易一些。但是，程序员还是必须要考虑具体的机器指令。接下来出现的高级语言（High-level Language）使编写程序更加容易，而且更接近英语、法语、德语和西班牙语等自然语言。Basic，FORTRAN，COBOL，Pascal，C，C++ 和 Java 都属于高级语言。在本书中，将学习 C++ 这种高级语言。

在 C++ 中，可以使用如下代码来计算周工资：

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

用C++编写的程序指令更加容易记忆，而且便于有算法基础的C++初学者理解。像汇编语言遇到的问题一样，计算机也无法直接执行用高级语言编写的程序代码。用C++编写的程序代码必须要转换成机器代码之后才能运行。一种称为编译器的程序可以完成这种转换。

编译器（Compiler）　是一种将高级语言编写的程序转变成为相应的机器语言指令的程序。

1.6　高级语言程序的处理过程

在弄清了什么是机器语言和高级语言的基本概念，以及计算机只能识别机器语言之后，还要知道在计算机上运行用高级语言编写的程序的必要步骤。在计算机上运行由高级语言编写的程序需要如图1.3所示的 5 个步骤：

1. 在文本编辑器上创建符合高级语言规则和语法的程序，这种程序称为源程序（Source Program）。

   源程序　是一种由高级语言编写的程序。

2. 只有符合高级语言规则的源程序，即源程序在语法上是正确的，才能转换成相应的机器语言。编译器负责保证源程序的语法正确性并将其转换成机器语言。即编译器检查源程序的语法，如果没有错误，就将其转换成相应的机器语言。这种转换后的机器语言程序称为目标程序。

   目标程序 是高级语言程序相应的机器语言版本。

3. 通常使用软件开发工具包（SDK，Software Development Kit）来开发相应的高级语言程序。软件开发工具包中包含有许多帮助程序员开发的程序。例如：软件开发工具包中含有能够显示程序运行结果的代码以及许多数学函数的代码，它们使得程序员的工作更加容易。因此，只要所要的代码已经存在，就可以直接去使用它们，而不是重新编写。一旦编写的源程序通过编译，就必须将其与软件开发工具包中被使用的程序合并，以便生成可以在计算机上运行的最终程序。这些已经编写好并可以直接使用的程序（以目标程序形式存在）存放在库（Library）中。连接程序（Linker）负责连接通过编译的目标程序和库中的程序。

   连接程序 是一种将目标程序和所用到的软件开发工具包中的程序结合在一起，并生成可执行代码的程序。

4. 下一步，就是要将可执行的程序装入内存以便运行。装载程序（Loader）完成这项工作。

   装载程序 是一种将可执行程序装入内存的程序。

5. 最后一步是运行程序。

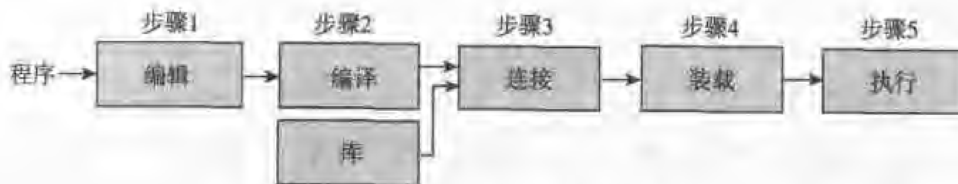


图 1.3 高级语言程序的执行过程

作为程序员，只需要关心第一步。因此，必须要学会、理解并且精通程序设计语言的语法规则来编写源程序。前面已经提到，程序是使用软件开发工具包（SDK）来开发的。著名的C++软件开发工具包有：Microsoft公司的Visual C++，Borland公司的C++ Builder和Metrowerks公司的CodeWarrior。这些软件开发工具包都包含编写源程序所需的编辑器，检查语法错误的编译器，连接目标程序和库程序的连接程序以及运行可执行程序的程序。而且，这些软件开发工具包都有很好的用户友好性。当编译程序时，编译器不仅可以指出语法错误，而且通常还可以告诉你应该怎样改正错误。此外，只要使用一个简单的命令，就可以将目标程序和库程序连接在一起。这个命令在Visual C++中是Build或Rebuild，在C++ Builder中是Build或Make，而在CodeWarrior中则是Make（如果需要关于这些命令的详细说明，请参阅相应软件开发工具包文档）。如果程序未经编译，这些命令将首先编译该程序，然后再连接生成可执行代码。

注意： 一般来说，高级语言的开发需要上述5个步骤。然而，C++程序设计语言的开发过程多出一个步骤。多出来的这一步骤在第一步编写源程序和第二步编译源程序之间。当然，这一步骤也是由软件开发工具包自动完成的。在第2章介绍完C++的一些基础知识以后，将知道C++程序的执行过程。前面已经讨论过，作为程序员，只需要关心第一步，即熟练掌握C++程序设计语言的规则，编写源程序。

1.7 按问题分析－编码－执行循环的程序设计

编写程序的过程就是解决问题的过程。不同的人采用不同的方法解决问题。一些好的解决问题方法清晰易懂，不仅可以解决问题，而且还能展示问题是如何解决的。当问题有所改变时，这些方法只需稍加改动就可以解决新问题。

如果想成为优秀的问题解决者，进而成为优秀的程序员，必须采用良好的解决问题方法。常用的问题解决方法包括以下几步：分析问题、找出问题需求、设计解决方案（即算法，Algorithm）和解决问题。

算法　是在限定时间内可以完成的解决问题的具体步骤。

一个问题的解决过程在程序设计里可分为以下3个步骤：

1. 分析问题，找出问题需求，并设计算法。
2. 在某种程序设计语言上，如C++中实现算法，并验证其有效性。
3. 通过改写程序以适应问题的变化。

图1.4总结了程序设计过程的3个步骤。

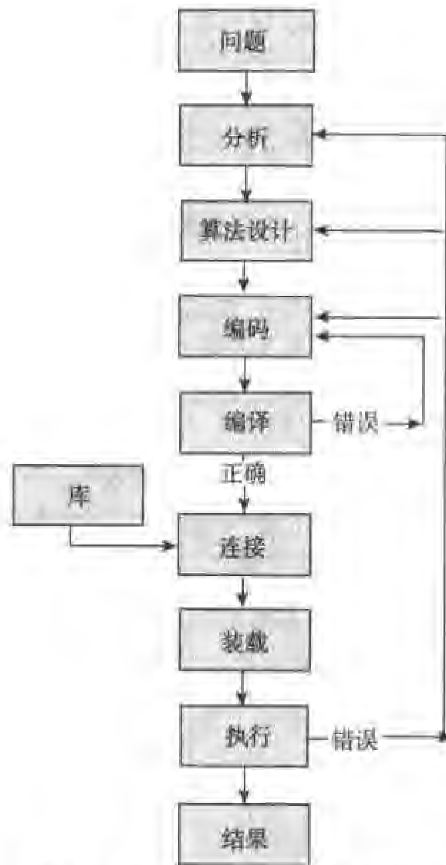


图1.4　分析问题－编码－执行循环

开发一个程序解决问题，首先必须从分析问题开始。然后设计算法，用高级语言编写程序指令（编码），最后在计算机中运行程序。分析问题是第一步而且也是最重要的一步。在分析问题时，需要做以下几件事情：

1. 彻底弄清问题。
2. 找出问题需求。问题需求包括程序是否需要用户交互，是否需要处理数据，是否需要输出数据，输出什么样的数据。如果涉及到数据处理，程序员必须要知道需要处理何种数据以及数据的格式。因此，需要参考样例数据。如果产生输出数据，则必须弄清怎样生成数据以及其格式。
3. 如果问题复杂，可将问题分为若干子问题，然后重复步骤1和步骤2。也就是说，复杂的问题需要分析其每个子问题并找出子问题的需求。

这种将复杂问题分为若干子问题的程序设计方法称为结构化设计（Structured Design）。结构化设计方法的显著特点是：自顶向下设计（Top-down Design）、逐步细化（Stepwise Refinement）和模块化程序设计（Modular Programming）。在结构化程序设计中，复杂问题被分为若干子问题。每个子问题再经分析，然后确定解决子问题的方案。最后，将所有子问题的解决方案合并，确定整个问题的解决方案。结构化设计的实施过程称为结构化程序设计（Structured Programming）。

在经过仔细的问题分析以后，下一步就是设计算法解决问题。如果问题被分为若干子问题，这些子问题也需要分别设计算法。算法一旦确定，首先就是要检验其正确性。算法正确性某些时候可以通过代入试验数据来检验，但有时必须通过数学分析来证明。

当算法设计和正确性检验完成后，下一步就是在高级语言上实现算法。可以使用文本编辑器来编写源程序。源程序代码必须要符合程序设计语言的规则。可以通过编译器来检验源代码的语法正确性。如果编译器提示源代码有错，则必须要找到错误并改正错误，并再次运行编译器。在所有的错误都被改正以后，编译器将源程序转换成目标程序。然后，由连接程序完成目标程序和所用到的软件开发工具包中的程序的连接工作。接下来，装载程序将连接好的程序装入内存以便运行。

最后是执行程序。编译器只保证程序符合语言的语法规则，并不能保证它可以在计算机中正确运行。运行中的程序可能由于逻辑错误而异常终止，比如说遇到了除数为0的情况。即使是程序正常结束，程序也可能产生错误结果。在这种情况下，可以通过重新检查源代码、算法甚至重新分析问题来纠正错误。

正如前面所述，在编写程序指令之前需要做许多的准备工作。通常可以使用纸和笔完成这些准备工作。采用这种程序设计方法有许多优点。经过仔细分析和设计的程序有助于及早发现错误。而且经过细致分析，精心设计的程序代码有很强的可读性并且容易修改。即使是经验十分丰富的程序员也需要在分析问题和设计算法上花费大量时间。

通过学习本书，读者不仅可以掌握C++程序设计语言的规则，而且可以学到分析问题的方法。本书的每一章中都有许多标记为“程序范例”的实际应用问题范例。这些程序范例中讲述了分析和解决问题的方法，并且帮助读者理解所在章节中涉及到的重要概念。仔细阅读程序范例中的内容，将使读者受益匪浅。

例 1.1 在本例中，将设计一个计算矩形周长和面积的算法。

要想计算矩形周长和面积，必须要知道矩形的长和宽。矩形周长和面积的计算公式如下：

```
perimeter = 2 × (length + width)  [周长=2 ×（长+宽）]
area = length × width             [面积=长×宽]
```

因此，计算矩形周长和面积的算法如下所示：

1. 输入矩形的长。
2. 输入矩形的宽。
3. 用下列等式计算矩形周长：

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

4. 用下列等式计算矩形面积：

   ```
   area = length × width
   ```

例 1.2 在本例中，将设计一个算法来计算当地百货公司销售人员的月工资。

每一个销售人员都有基本工资，并根据以下准则领取奖金。如果销售人员在该百货公司工作5年或5年以下，每年有\$10的奖金；如果工作时间超过5年，每年有\$20的奖金。销售人员还可以获得销售奖金，准则是：如果销售额在\$5 000到\$10 000之间，可以获得3%的佣金；如果销售额超过\$10 000，可以获得6%的佣金。

要想计算销售人员的月工资总额，必须要知道销售人员的月基本工资，在该公司工作的年限和该月的销售额。假设用 payCheck 表示月工资总额，baseSalary 表示基本工资，noOfServiceYears 表示该销售人员在公司的工作年数，bonus 表示奖金，totalSale 表示该销售人员当月的销售额，additionalBonus 表示销售奖金。

奖金的计算公式是：

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

接下来，计算销售奖金的公式是：

```
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)
```

经过上面分析，现在可以写出计算销售人员月工资总额的算法。

1. 输入 baseSalary。
2. 输入 noOfServiceYears。
3. 根据下面公式计算奖金：

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

4. 输入 totalSale。
5. 根据下面公式计算销售奖金：

```
if(totalSale is less than 5000)
   additionalBonus=0
otherwise
   if(totalSale is greater than or equal to5000 and?
         totalSale is less than10000)
         additionalBonus= totalSale × (0.03)
otherwise
      additionalBonus= totalSale × (0.06)
```

6. 根据如下公式计算 payCheck：

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

1.8 面向对象程序设计

上一节介绍了结构化程序设计方法。本节将简要介绍应用十分广泛的面向对象程序设计（OOD）方法。

在面向对象程序设计中，解决问题过程的第一步是定义对象（Object）和确定各对象之间的相互联系。定义对象是解决问题的基础。例如：假设需要编写当地影碟店里出租影碟的自动处理程序，那么这个问题的两个基本对象就是影碟和顾客。

在定义了对象以后，下一步就是要确定与每个对象有关的数据以及在这些数据上可能进行的操作。例如，影碟对象包括数据：电影名字、主要演员、制片人、制片公司，以及影碟的拷贝数量等。在影碟对象上可能进行的操作有：查看影碟名字、当某一影碟被借走后将其拷贝数量减 1、当顾客归还某一影碟时将其拷贝数量加 1。

上例说明了对象是由数据和在数据上进行的操作组成。对象将数据和在数据上进行的操作封装成一个整体。在面向对象程序设计中，程序最终由许多相互联系的对象组成。支持进行面向对象程序设计的程序设计语言称为面向对象程序设计语言（OOP，Object-Oriented Programming）。在后面章节中，将了解到面向对象程序设计的种种优点。

因为对象由数据和在数据上进行的操作两部分组成，所以在设计和使用对象之前，需要先了解数据在计算机存储器中是怎样表示的，怎样操纵数据和怎样实现操作。在第 2 章中，将了解到 C++ 的基本数据类型，并且知道怎样在存储器中表示数据和操纵数据。第 3 章将讨论 C++ 程序是怎样读入数据以及怎样将 C++ 程序的运行结果输出。

为了实现对数据的操作，需要设计算法并使用程序设计语言来实现算法。因为在复杂程序中对一种数据元素，可能会进行很多种操作。为了区分这些操作并且可以方便、有效的方式使用这些操作，可以使用函数（Function）来实现算法。在第 2 章和第 3 章简要地介绍函数之后，第 6 章和第 7 章将详细描述函数的使用方法。一些算法需要程序做出判断，这个过程叫做选择（Selection）。一些算法需要反复执行某些语句，这个过程叫做循环（Repetition）。还有些算法既需要选择又需要循环。在本书第 4 章、第 5 章中将介绍这种选择循环机制，称为控制结构（Control Structure）。此外，在第 9 章中将介绍怎样使用一种叫做数组（Array）的数据结构来处理相同类型的数据，如销售统计表中的数据项。

最后，为了能够使用对象，还必须要将对象中的数据和在数据上进行的操作封装成一个整体。在 C++ 中，将这种对象中的数据和在数据上进行的操作封装成的整体称为类（Class）。第 12 章将介绍类的实现机制、怎样使用类以及怎样创建类。

正如上面所述，在可以使用面向对象程序设计方法编写程序以前还有许多需要了解的知识。为了便于读者学习，本书特意将控制结构（第 4 章和第 5 章）和用户定义函数（第 6 章和第 7 章）分别分为两章来介绍。

对于某些问题，使用结构化方法进行程序设计十分有效。而另一些问题则需要使用面向对象程序设计方法来设计。例如：如果该问题需要使用数学函数对大量数据进行处理，则可以采用结构化方法并且遵循其设计步骤来设计。C++语言库自带了大量的十分好用的数学函数，供程序员方便地进行数值计算。另一方面，如果需要编写程序符合人们的操作习惯，则应该采用面向对象程序设计方法来设计。C++ 是特别为面向对象程序设计而开发的程序设计语言。而且，在 C++ 中可以将面向对象设计方法和结构化设计方法结合使用。

无论是结构化设计方法还是面向对象设计方法都需要熟练掌握程序设计语言的基本组成部分。在接下来的几章中，将了解到这两种设计方法都需要掌握的 C++ 程序设计语言基本组成部分。在介绍了基本概念之后，第 2 章至第 11 章将描述怎样使用结构化设计方法编写程序解决问题，从第 12 章以后主要叙述怎样使用面向对象设计方法解决问题。

1.9　ANSI/ISO 标准 C++

C++ 程序设计语言由 C 语言发展而来，并于 20 世纪 80 年代初期由 Bjarne Stroustrup 在贝尔实验室设计出来。从 20 世纪 80 年代初到 20 世纪 90 年代初的时间里，出现了好几种版本的 C++ 编译器。虽然几乎所有的编译器都支持 C++ 的基本部分，但是 C++ 语言，本书中称为标准 C++，在不同的编译器上略有不同。因此，不能保证某一 C++ 程序在所有 C++ 编译器上都能通过编译。

为了解决这个问题，20 世纪 90 年代初期，成立了由美国国家标准协会（ANSI，American National Standard Institution）和国际标准化组织（ISO，International Standard Organization）的联合小组，共同制

订C++的标准语法。在1998年年中，ANSI/ISO C++语言标准被正式批准。现在，绝大多数的编译器都支持这项新标准。

本书将主要描述ANSI/ ISO支持的C++语法，即ANSI/ISO标准C++。然而，读者可能会使用早期编写的C++程序（或仍使用老的C++编译器），以后在每一章中都将告诉读者需要在ANSI/ISO标准C++的程序中做哪些改动。虽然标准C++和ANSI/ISO标准C++大部分语法是相同的，但是还是有一些只有ANSI/ISO标准C++支持的特性，本书将指出这些特性并详细讨论。

1.10　小结

1. 计算机是一种具有算术和逻辑运算功能的电子设备。
2. 计算机系统由两部分组成：硬件和软件。
3. 中央处理器（CPU）和内存属于硬件部分。
4. 控制单元（CU）是中央处理器诸多部分中的一个，负责控制程序的执行。
5. 算术逻辑单元（ALU）是中央处理器中负责算术和逻辑运算的部分。
6. 所有的程序必须装入内存后才能运行。
7. 当计算机关闭以后，内存中所有的数据都将丢失。
8. 外部存储器可以永久地保存数据。硬盘、软盘、Zip盘和光盘都属于外部存储器。
9. 数据通过输入设备输入到计算机中，常见的输入设备包括键盘和鼠标。
10. 计算机将输出数据输出到显示屏等输出设备上。
11. 软件是在计算机上运行的程序。
12. 操作系统负责监控计算机中的所有活动并且提供服务。
13. 计算机中最基本的语言是被称为由0和1序列组成的机器语言。计算机可以直接识别自己的机器语言。
14. 一个位是一个二进制数0或1。
15. 一个字节是连续的8个位。
16. 汇编语言使用容易记忆的汇编指令。
17. 汇编语言编译器（Assembler）是用来将汇编语言程序翻译成机器语言的程序。
18. 编译器（Compiler）是将高级语言程序翻译为机器语言的程序，翻译成的机器语言称为目标代码。
19. 连接程序的作用是将程序的目标代码和软件开发工具包（SDK）中提供的程序连接在一起，并生成可执行代码。
20. 使用高级语言编写程序，需要5个步骤：编辑、编译、连接、装载和运行。
21. 装载程序将可执行代码装入计算机。
22. 算法是在有限的时间内可以完成的解决问题的具体步骤。
23. 解决问题的过程可以分为三个步骤：分析问题和设计算法，用高级语言实现算法，以及维护程序。
24. 使用结构化方法编写的程序有以下优点：易于理解、易于测试、易于调试和易于修改。
25. 在结构化设计方法中，复杂问题被分为若干子问题。然后分别解决每个子问题，所有子问题的解决方案就构成了整个问题的解决方案。
26. 在面向对象设计方法中，整个程序由许多相互关联的对象组成。
27. 对象由数据和在数据上的操作组成。
28. 1998年年中，通过了ANSI/ISO标准C++。

1.11 练习

1. 判断下列语句的正误：
   a. 汇编语言中采用的是便于记忆的程序指令。
   b. 编译器（Compiler）将汇编语言程序翻译成机器语言程序。
   c. 算术逻辑单元只执行算术运算，如果发生错误，输出逻辑错误信息。
   d. 装载程序将程序从内存装入中央处理器（CPU）以便执行。
   e. 使用高级语言开发程序需要经过6个步骤。
   f. 中央处理器（CPU）在控制单元（CU）的控制下进行运算。
   g. RAM是随时可用的存储器（Readily Available Memory）。
   h. 使用高级程序设计语言编写的程序称为源程序。
   i. 操作系统是计算机开机后第一个被装入内存的程序。
   j. 解决问题过程的第一步是分析问题。
2. 说出中央处理器的几个组成部分。
3. 控制单元的作用是什么？
4. 说出两种输入设备。
5. 说出两种输出设备。
6. 为什么需要外部存储器？
7. 为什么要将高级程序语言编写的程序翻译成机器语言？
8. 为什么要采用高级程序设计语言而不是机器语言来编写程序？
9. 在用高级程序设计语言编写程序之前，先进行问题分析和算法设计的好处是什么？
10. 设计一种算法用来计算4门课程成绩的加权平均值。这4门课程的成绩和相应的权数用以下格式给出：

    第1门课程的成绩　　第1门课程的权数

    ...

    示例数据如下所示：

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

11. 一个销售人员每周一离开家去工作并于周五返回。他每天驾驶由公司提供的汽车工作。这个销售人员每天都记录下其加油数量（刚好够当天使用）。假设给出里程表的开始读数（即周一离开家之前的读数）和最终读数（周五返回家后的读数），设计一个算法计算每公里的耗油量。示例数据如下所示：

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

第2章 C++基础

本章要点：

- 了解C++程序的基本组成部分，包括：函数、特殊符号和标识符
- 了解简单数据类型和string数据类型
- 理解怎样使用算术运算符
- 理解怎样计算算术表达式
- 理解什么是赋值语句及其作用
- 理解怎样通过输入语句将数据输入到计算机内存中
- 熟练掌握增量运算符和减量运算符
- 理解怎样使用输出语句输出计算结果
- 理解怎样使用预处理程序及其必要性
- 了解怎样设计结构良好的程序，包括使用注释来提高程序可读性
- 了解怎样编写C++程序

在本章中，将了解到C++语言的基础知识。既然是要学习C++程序设计语言，那么自然会遇到两个问题。第一个问题是什么是计算机程序，第二个问题是什么是程序设计。计算机程序（Computer Program）或者称为程序是能够完成特定任务的语句序列。程序设计（Programming）是设计和编写程序的过程。它们是程序设计中的两个重要概念，但并不是程序设计过程的全部（有必要学完一整本书以便了解程序设计的全部内涵）。也许通过类比的方法可以更加容易地领会程序设计的实质，现在就让我们用每个人都有所了解的烹饪来打个比方。食谱就相当于程序，对烹饪略知一二的人都会认同：

1. 按照食谱上的做法比自己发明一种做法做菜更容易。
2. 食谱有好坏之分。
3. 一些食谱简便易学，而另一些则难以掌握。
4. 一些食谱写得详细可靠，不同的人按照这种食谱做出菜来都差不多，而按照另一些食谱做出菜来则大相径庭。
5. 在按照食谱做菜之前，必须要知道各种厨具的用法。
6. 只有具有丰富的烹饪经验，才可能自己发明一种食谱。

上述6点也适用于程序设计。让我们进一步使用烹饪的例子进行类比。假设你正在培训厨师，会怎么做？会让受训者先品尝一些美味，并希望他们会从中产生一些烹饪的灵感吗？会让受训者按照许多食谱不断练习，直到他们按照某个食谱做出来的菜味道十分好为止？或者先教给受训者各种厨具的用法和各种原料及调料的基本特性，并介绍它们怎样搭配味道才会更好？在怎样传授烹调技术问题上会有如此多的分歧，那么讲授程序设计也是如此。

学习程序设计跟学习厨艺或是学习演奏乐器有相似之处，都需要直接与工具打交道。单单学习食谱，绝不会成为厨师（即使是水平一般的厨师）。同样，只是学习乐器演奏方面的书籍，也绝不会成为演奏家。学习程序设计也是如此。程序设计学习者首先要具备扎实的程序语言知识，此外还必须在计算机上亲自编写程序来验证结果的正确性。

2.1 C++程序基础

C++程序由一个或多个子程序（Subprogram）[或者称为函数（Function）]组成。简单来讲，子程序或函数是许多语句的集合。当被激活、运行后，子程序或函数可以完成特定的功能。计算机系统中有一些已经写好供程序员使用的函数，这些函数被称为预定义（Predefined）或是标准（Standard）函数。但是，为了实现更多的功能，程序员必须学会编写自己的函数。

每一个C++程序都有一个名叫main的函数。就是说，如果程序只由一个函数组成，那么这个函数一定是main。在第6章以前，除了使用一些预定义函数外，将主要学习怎样编写main函数。在完成本章后，读者将知道怎样编写简单的main函数。

下面是一个C++程序范例，在本例中，无须知道程序的细节。

例2.1 这是一个C++程序范例。

```
#include <iostream>

using namespace std;

int main()
{
  cout<<"Welcome to C++ Programming"<<endl;
  return 0;
}
```

如果运行这段程序，在计算机屏幕上将会出现下面一行：

```
Welcome to C++ programming
```

如果从未见过程序设计语言编写的程序，例2.1中的程序看起来就像是外语。如果要看懂外语句子的意思，那么就要学习字母表、词汇和语法。学习程序设计语言也是如此。为了编写程序，必须要学会程序设计语言中的特殊符号、保留字和语法规则。语法规则（Syntax Rule）将告诉你哪些语句（指令）合法，即可以被程序设计语言所接受，而哪些又是非法的。还要学习语义规则（Semantic Rule）。语义规则决定语句的含义。弄清了程序语言规则、符号和保留字，才可以编写程序解决问题。语法规则决定哪些语句是合法的。

程序设计语言（Programming Language） 是规则、符号和保留字的集合。

可以使用多种方法来描述程序设计语言的语法规则。用来描述语法规则的语言称为元语言（Metalanguage）。本书使用一种称为语法模板（Syntax Template）的元语言来描述语法规则。语法模板是形象化的，便于直观理解。例如，函数main的语法模板如下所示：

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

注意：语法模板中带有阴影的语句是可选的。

在本节的余下部分里，读者将了解到C++程序设计语言中的部分特殊符号。其余特殊符号将在以后章节出现时，再加以介绍。同样，语法规则和语义规则也将穿插在全书中讲解。

任何程序设计语言中最小的单元都被称为记号（Token）。在C++中，记号分为：特殊符号、保留字和标识符。

特殊符号

下面列出了一些特殊符号：

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

第一行是加、减、乘、除等数学符号。第二行是在英语语法中规定的标点符号。在C++中，逗号用来分隔列表(List)中的数据项；分号用来标识一条C++语句的结束。注意，虽然没有列在上面，空格符(Blank)也是一种特殊符号。可以使用键盘上的空格键产生一个空格符（只按一次）。第三行的记号都由两个字符组成，但是只作为一个符号。两个字符之间不能有任何其他字符，包括空格符。

保留字

第二种记号是保留字。下面列出了一些保留字：

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

保留字（Reserved Word），又称为关键字（Keyword）。组成保留字的字母必须是小写的。正如特殊符号一样，每个保留字都被视为一个符号。而且任何程序设计语言的保留字都不能重新定义。因此，不能用任何别的符号来代替保留字。附录A中列出了全部的保留字。

标识符

第三种记号是标识符。标识符是在程序中出现的变量、常量或者函数等的名字。一些标识符是预定义的，而另一些则是用户定义的。所有的标识符必须符合C++的标识符规则。

标识符　在C++中，所有标识符由字母、数字和下划线（_）组成，而且必须以字母或下划线开头。

标识符只可以由字母、数字和下划线组成，其他的字符都不可以用在标识符中。

注意： C++语言是区分大小写（Case Sensitive）的，即同一字母的大写字母和小写字母被认为是不相同的。因此，NUMBER和number是不同的标识符。同样，X和x也是不同的标识符。

在C++中，标识符的长度是任意的。但实际上，如果标识符的长度超过了系统的最大有效字符数（Maximum Significant Number of Characters），并且在最大有效字符数之内的部分是相同的，则有些编译器将它们视为同一标识符（即使这些标识符在最大有效字符数之外的部分是不同的。可以参阅系统说明书中关于标识符长度限制部分）。例如，如果系统只能区分标识符的前7个字符，那么下面两个不同的标识符将被视为同一标识符：

```
program1
program2
```

两个经常使用的预定义标识符是：输出数据时用到的cout和输入数据时用到的cin。与保留字不同，预定义标识符可以被重新定义，虽然这种做法是不明智的。

例2.2　下面是C++中的合法标识符：

```
first
conversion
payRate
counter1
```

表 2.1 中列出了一些非法的标识符，并说明为什么是非法的。

表 2.1 非法的标识符范例

非法的标识符	说明
employee Salary	在 employee 和 Salary 之间不能有空格
Hello!	标识符中不能有叹号
one+two	标识符中不能有加（+）号
2nd	标识符不能以数字开头

注意： 编译器厂商提供的编译器通常使用一些以下划线开头的标识符。为了防止连接程序在连接目标程序和软件开发工具包（SDK）所提供的系统资源时产生错误，尽量不要在程序中定义以下划线开头的标识符。

2.2 数据类型

C++程序的主要功能就是进行数据处理。不同的程序处理不同的数据。用来计算员工工资单的程序要对数字进行加、减、乘、除运算，并使用一些数据来表示工作小时和工资率。同样，将一个班级的学生名字按字母表排序的程序，需要对名字数据进行处理。就是说，用来进行算术计算的程序不能用来处理名字排序，同样也不能对于名字进行乘除等运算。为了区分这些基本差异，C++ 将数据分为不同的类型，并且规定在某种类型的数据上只能进行特定运算。虽然这种数据类型的划分会使初学者感到困惑，但是 C++ 语言通过一些内建的检查机制来防止数据类型上的误用。

数据类型 是数据集及定义在数据集上的一些运算。

C++ 的数据类型可以分为 3 大类（如图 2.1 所示）：

1. 简单数据类型（Simple data type）
2. 结构类型（Structured data type）
3. 指针类型（Pointer）

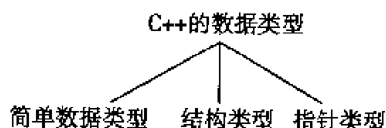


图 2.1 C++ 数据类型

在接下来的几章中将主要介绍简单数据类型。

2.2.1 简单数据类型

简单数据类型是 C++ 语言的基本类型，将在第 9 章中介绍的结构类型（也是由简单数据类型组成的）。简单数据类型又可分为 3 类：

1. 整型（Integral）：整数类型
2. 浮点型（Floating-point）：小数类型
3. 枚举型（Enumeration type）：用户定义的类型

图 2.2 中列出了 3 种简单数据类型。

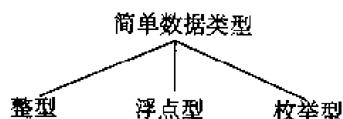


图 2.2 简单数据类型

注意：C++ 中允许程序员使用枚举类型创建自己的简单数据类型，枚举类型将在第 8 章中介绍。

整型进一步可以分为如图 2.3 所示的 9 种类型。

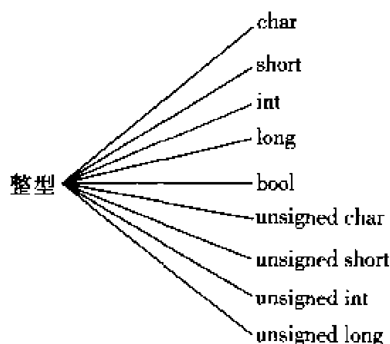


图 2.3　整型数据类型

为什么在同一数据类型中还要分成许多种类呢？因为每个种类所表示数据的有效取值范围各不相同。例如：char 类型表示的整数的有效取值范围为 -128~127；int 类型表示的整数的有效取值范围为 -2 147 483 648 ~ 2 147 483 647；short 类型表示的整数的有效取值范围为 -32 768~32 767。

在程序中要根据实际对数据有效取值范围的要求来选择不同数据类型。在早期编写程序时，计算机和内存资源都很宝贵，只有很少量的内存供程序运行和数据运算使用。因此，程序员必须谨慎使用有限的内存资源。编写程序本来就已经是一件很复杂的事情，所以程序员也就无暇对怎样优化使用有限的内存做更多的考虑了。因此，将同一数据类型根据其可能的取值范围分为不同的子类型，就成了有效地使用内存的简便方法。程序员首先考虑需要处理的数据的有效取值范围，然后再决定选用何种数据类型。

近来，一些程序设计语言将简单数据类型只分为 5 类：integer（整数）类型、real（实数）类型、char（字符）类型、bool（布尔）类型和 enumeration（枚举）类型。而在本书中，int 类型、bool 类型和 char 类型都属于简单类型的整型。

表 2.2 中列出了这 3 种整型数据类型的有效取值范围和占用的内存空间大小。

表 2.2　3 种简单数据类型的有效取值范围和占用的内存空间

数据类型	有效取值范围	内存空间（以字节为单位）
int	-2 147 483 648~2 147 483 647	4
bool	true 和 false	1
char	-128~127	1

注意：本表中的数据只供参考。不同的编译器所规定的有效取值范围可能不同，详情参阅编译器的说明文档。想了解具体系统中的整型数据所占内存空间的大小，请在该系统上运行附录G中的程序——Memory Size on a System。如果想进一步了解其取值范围，请运行附录 F 中的另一程序——头文件 climits。

int 数据类型

本节将介绍 int 数据类型。实际上，本节中所做的讨论也适用于其他整数类型。

C++ 中的整数和数学中的整数一样，指的是如下所示一类数字：

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

注意所示数据中的两个规则：

1. 正整数前可以不加 “+” 号。
2. 数字的各位间不可以用 “,” 做分隔符。因为，在 C++ 中，“,” 用于在列表（List）中分隔数据项。因此，“36,782” 将被解释为两个整数：36 和 782。

bool 数据类型

bool 型数据只有两个取值：true 和 false。true 和 false 也被称为逻辑（布尔）值。这种类型的数据主要用于逻辑（布尔）表达式中。只产生 true 和 false 两种结果的表达式称为逻辑（布尔）表达式。逻辑表达式将在第 4 章中重点描述。在 C++ 中，bool，true 和 false 都是保留字。

注意：bool 型数据是 C++ 中最新增加的数据类型。为确保可以使用这种类型的数据，请参阅所在系统的说明文档。

char 数据类型

char 型数据是最小的整型数据。除了用于表示小整数（-128~127），char 型数据主要用来表示字母、数字和特殊符号等字符。因此，char 型数据可以表示计算机键盘上的所有字符。在使用 char 型数据时，要将每一个字符用单引号括住。下面是一些 char 型数据范例：

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

注意，空格符也是一个字符，用一个被单引号括住的空格来表示，即' '。

char 型数据只允许单引号中有一个字符。因此，'abc' 不是 char 型数据。而且，即使是被视为单一符号的，如 '!=' 一类的特殊字符，也不是 char 型数据。键盘上的所有字符，即可打印字符，都是 char 型数据。

现在，除了ASCII码字符集外还有其他的字符集。最常见的字符集是：ASCII码和EBCDIC码。ASCII 码字符集中有 128 个字符，而 IBM 的 EBCDIC 码字符集中则有 256 个字符。附录 C 中同时给出了这两种字符集。

128 个（0~127）char 型整数，分别代表 ASCII 码字符集中的 128 个不同字符的值。例如：char 型整数值 65 表示字母 'A'，char 型整数值 43 则表示字符 '+'。因此，字符集中的每个字符都有一个预先定义好的次序。在字符集中，字符的排列顺序用来比较不同字符的“大小”。例如：字母 'B' 对应的整数值是 66，所以字母 'A' 小于字母 'B'。同样，字符 '+' 小于字母 'A'，因为字符 '+' 对应的整数值是 43，小于字母 'A' 对应的整数值是 65。

注意：ASCII 码字符集中第 14 个字符是换行符，表示为 '\n'（注意，换行符在 ASCII 码字符集中的值是 13，因为第 1 个字符的值是 0）。虽然换行符由两个字符组成，但还是被认为是一个字符。同样在 C++ 中，水平制表符表示为 '\t'，空字符（null）表示为 '\0'（反斜线后面紧跟数字 0）。还要指出的是，ASCII 码字符集中的前 32 个字符是不可打印字符（附录 C 中有这些字符的详细介绍）。

2.2.2 浮点数据类型

为了处理小数，C++提供了浮点数据类型，本节将介绍浮点数据类型。为了便于讨论，先来复习一下数学课程中的一些基本概念。

科学记数法（Scientific Notation）是我们熟知的用于表示数字的方法，特别是表示小数的方法。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++ 采用了一种称为浮点数表示法（Floating-point Notation）的科学记数法。表 2.3 范例说明了 C++ 是怎样将实数在某种机器上用浮点数表示法表示出来的。在 C++ 中，字母 E 表示指数。

表 2.3 用 C++ 浮点数表示法表示的实数范例

实数	C++ 浮点数表示法
75.924	7.592400E1
0.18	1.800000E-1
0.000 045 3	4.530000E-5

（续表）

实数	C++浮点数表示法
-1.482	-1.482 000 E0
7 800.0	7.800 000 E3

C++提供了3种数据类型来存储实数：float，double和long double。正如在整数类型中介绍的一样，这3种浮点数具有不同的有效取值范围。图2.4显示了这3种数据类型。

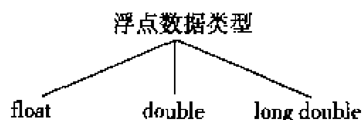


图2.4 浮点数据类型

注意：*在许多最新的编译器上，double和long double是相同的。因此，这种编译器中只有float和double两种表示实数的数据类型。*

float：C++中float型数据用于表示-3.4E+38~3.4E+38范围内任意实数。float型数据在内存中占用4个字节。

double：C++中double型数据用于表示-1.7E+308~1.7E+308范围内的任意实数。double型数据在内存中占用8个字节。

在不同系统中，float和double型数据的最大值和最小值可能是不同的。确定某一系统中的float和double型数据的最大值和最小值有两种方法：参阅编译器的说明文档，或是运行附录F中的程序（头文件cfloat）。

除了有效取值范围的差别以外，float和double型数据还有一点不同，即最大有效数字位数（即小数点后有效的小数位数）不同。float型数据的有效数字位数是6位或7位，而double型数据的有效数字位数是15位。最大有效数字位数也称做精度（Precision）。有时，将float型数据称为单精度型数据，将double型数据称为双精度型数据。如果编写的程序涉及到小数，在一般情况下使用float类型就足够了。但是，如果需要6位或7位以上精度的小数，则要选择double类型。

注意：*如果在程序中使用float类型的数据，某些编译器会提出类似于这样的警告信息："从double类型转换成float类型时将截去部分小数"。可以使用double类型数据，来避免出现这种警告信息。为了说明方便和避免出现上述警告信息，本书中将主要使用double型浮点数。*

最后，将讨论字符串（string）数据类型。

2.2.3 字符串数据类型

字符串（string）数据类型是程序员定义的数据类型。与前面讲到的简单数据类型不同，字符串数据类型并不是可以直接在程序中使用的。为了使用这种类型的数据，必须要涉及到程序库中所提供的部分功能。程序库所提供的这部分功能将在以后章节中介绍。字符串数据类型是ANSI/ISO标准C++所支持的数据类型。

注意：*在ANSI/ISO标准C++问世之前，标准C++库不提供字符串数据类型。编译器厂商通常提供自己定义的字符串数据类型。因而，在字符串上的各种运算的语法和语义解释也就因厂商而异。*

字符串是由0个或多个字符组成的字符序列。字符串在C++中用双引号（"）引起来。不含任何字符的字符串称为空串（Null或Empty）。下面是一些字符串的范例，注意""是空字符串。

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

字符串中的每个字符都有相对位置。第一个字符的相对位置是0，第二个字符的相对位置是1，以此类推。字符串的长度是字符串中字符的个数。

例 2.3

字符串	字符在字符串中的位置	字符串的长度
"William Jacob"	'W' 的位置是 0	13
	第一个 'i' 的位置是 1	
	' '（空格字符）的位置是 7	
	'J' 的位置是 8	
	'b' 的位置是 12	
"Mickey"	'M' 的位置是 0	6
	'i' 的位置是 1	
	'c' 的位置是 2	
	'k' 的位置是 3	
	'e' 的位置是 4	
	'y' 的位置是 5	

在计算字符串长度的时候，必须把空格字符的个数也计算在内。例如，下面字符串的长度是22。

```
"It is a beautiful day."
```

注意：标准C++不支持字符串数据类型。

2.3 算术运算符及其优先级

计算机的重要用途之一就是进行数值计算。C++可以在整型和浮点型数据上进行标准的算术运算。5种算术运算符是：

1. + 加法运算符
2. - 减法运算符
3. * 乘法运算符
4. / 除法运算符
5. % 求余运算符（求模运算符）

+，-，*，/运算符可以用在整型数和浮点型数中，进行相应的算术运算。%运算符只能用于整型数的求余运算中。在整型数中使用"/"运算符进行除法运算时，只能得到普通除法中的商。就是说，小数部分将被截去，而且不进行四舍五入进位。

读者应该对下列算术表达式的计算相当熟悉：

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

在这些表达式中，x和y是未知数。算术表达式（Arithmetic Expression）通常由算术运算符和数字组成。在表达式中出现的数字称为操作数（Operand）。与某个运算符一起运算的数值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。并且，当进行运算符"+"运算时，表达式3 + 4的运算结果是7。注意，在表达式3 + 4中，运算符"+"有两个操作数。具有两个操作数的运算符称为双目运算符（Binary Operator）。

只有一个操作数的运算符称为单目运算符（Unary Operator）。在表达式：

```
-5
```

中，运算符“-”只有一个操作数5。所以，“-”是单目运算符。如前所述，正数前不必加“+”。因此，在表达式：

```
+27
```

中，“+”也是单目运算符。

单目运算符　只有一个操作数的运算符

双目运算符　有两个操作数的运算符

在下面表达式中：

```
3 + 4
23 - 45
```

“+”和“-”都是双目运算符。因此，“+”和“-”既是单目运算符又是双目运算符。但是，诸如*，/，%等算术运算符只是双目运算符。

在例2.4中，说明了算术运算符，特别是“/”和“%”，是怎样和整型数在一起运算的。正如在例2.4中所见到的，在整型数中使用“/”进行普通除法运算，计算结果将只是商（截去余数部分后的整数部分）。

例2.4

算术表达式	运算结果	说明
2+5	7	
13+89	102	
34-20	14	
45-90	-45	
2*7	14	
5/2	2	在除式5/2中，商是2余数是1。因此，整数除法5/2的结果是2
14/7	2	
34%5	4	在除式34/5中，商是6余数是4。因此，34%5的结果是4
-34%5	-4	在除式-34/5中，商是-6余数是-4。因此，-34%5的结果是-4
34%-5	4	在除式34/-5中，商是-6余数是4。因此，34%-5的结果是4
-34%-5	-4	在除式-34/-5中，商是6余数是-4。因此，-34%-5的结果是-4
4%6	4	在除式4/6中，商是0余数是4。因此，4%6的结果是4

注意，在除式34/5和-34/-5中，商相同，都是6，但是余数不相同。在除式34/5中，余数是4；而在除式-34/-5中，余数是-4。

下面例2.5说明了算术运算符是怎样和浮点数一起进行运算的。

例2.5

表达式	计算结果
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 运算符优先级

当算术表达式中有两个及两个以上算术运算符时，C++使用算术运算符优先级规则来计算表达式的值。根据优先级规则，算术运算符：

```
*, /, %
```

的优先级要高于：

```
+, -
```

注意算术运算符“*”，“/”和“%”具有相同的优先级，同样算术运算符“+”和“-”也具有相同的优先级。

当算术运算符具有相同的优先级时，C++采用从左到右的顺序计算表达式的值。为了避免混淆，可以使用括号将优先要计算的部分括起来。例如，根据优先级规则，表达式：

```
3*7-6+2*5/4+6
```

的计算过程如下：

```
(3*7)-6+((2*5)/4)+6
=21-6+(10/4)+6        (*优先)
=21-6+2+6             (/优先)
=15+2+6               (-优先)
=17+6                 (+优先)
=23                   (+优先)
```

注意在上例中，使用括号指明了算术运算符的优先顺序。也可以使用括号来改变算术运算符的计算顺序（见例 2.6 所示）。

例 2.6

在表达式：

```
3+4*5
```

中，“*”运算在“+”运算之前计算。因此，该表达式的值是 23。而在表达式：

```
(3+4)*5
```

中，“+”运算在“*”运算之前计算，该表达式的计算结果是 35。

因为算术运算符的计算顺序是从左到右，所以除非遇到括号，算术运算符的结合律（Associativity）是从左到右的。

注意：（字符算术运算）既然 char 也是整型数据，C++ 允许对 char 型数据进行算术运算。但是，要慎重使用这种计算。字符 '8' 和整数 8 是不相同的。整数 8 的值是 8，而字符 '8' 的值是 56，是 ASCII 码中字符 '8' 的值。

当进行算术表达式运算时，8 + 7 = 15，而 '8' + '7' = 111，'8' + 7 = 56 + 7 = 63。要注意的是，因为 '8' * '7' = 56 * 55 = 3 080，而 ASCII 码字符集中只有 128 个值，所以 '8' * '7' 的计算结果在 ASCII 码字符集中是没有定义的。

这些例子说明在进行字符算术表达式运算时，很可能会发生错误。如果必须使用字符算术表达式，一定要十分谨慎。

2.4 表达式

到目前为止，我们只讨论了算术运算符。所以在本节中，将详细说明算术表达式。

如果表达式中所有的操作数（即数字）都是整数，这种表达式称为整型表达式。如果表达式中所有的操作数都是浮点数，这种表达式称为浮点型表达式，或是小数表达式。整型表达式只会产生整数结果，浮点型表达式也只产生浮点数结果。为了便于理解，让我们先看几个例子。

例2.7　考虑如下C++整型表达式：

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

在这些表达式中，x，y和z都是整型变量，即只能存储整数。变量将在以后章节中介绍。

例2.8　考虑如下C++浮点型表达式：

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

本例中x和y都是浮点型变量，即只能存储浮点数。变量将在以后章节中介绍。

计算单纯的整型或是浮点型表达式很简单。正如以前介绍的，如果运算符具有相同的优先级别，整个表达式从左到右进行计算。可以使用括号来避免混淆。

2.4.1　混合表达式

如果一个表达式中操作数的类型不同，这种表达式称为混合表达式（Mixed Expression）。混合表达式中既有整型数又有浮点型数。下面列出了一些混合表达式：

```
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“+”有一个整型操作数和一个浮点型操作数。在第二个表达式中，运算符“/”的两个操作数都是整数。“+”的第一个操作数是6/4的计算结果，第二个操作数是浮点型数。第三个例子是更复杂的整型数和浮点型数的混合表达式。一个显而易见的问题是：C++怎样计算混合类型表达式？

C++使用两个规则来计算混合表达式：

1. 当处理混合表达式中的运算符时：
   a. 如果两个操作数都是相同的类型（即都是整型数或都是浮点型数），则运算符根据操作数的类型进行计算。如果是整型操作数，则产生整型的计算结果；如果是浮点型操作数，则产生浮点型结果。
   b. 如果运算符有两种不同类型的操作数（即一个整型数一个浮点型数），则在计算过程中，先将整型数转换成浮点数（小数部分为0），然后再进行运算。结果也为浮点型数。
2. 整个表达式根据优先级规则进行计算。乘法、除法和求余运算符在加法和减法运算符前计算。具有相同优先级的运算符从左到右进行计算。可以使用括号来改变优先级。

这些规则保证，当进行混合表达式运算时，可以按优先级一次只计算一个运算符。如果运算符的两个操作数具有相同的数据类型，应依据上述规则1(a)进行计算。也就是，只有整型操作数的运算符的运算结果同样是整型数；只有浮点型操作数的运算符的运算结果同样是浮点型数。如果运算符的两个操作数一个是整型而另一个是浮点型，则在计算这个运算符之前，先将整型数转换为浮点数。下面的例子说明了怎样计算混合表达式。

例 2.9

混合表达式	计算	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

这些例子说明，只有在算术运算符中有整型数和浮点型数两个操作数的情况下，整型数才需要转换成浮点型数，其他情况不进行转换。

现在，考虑下面的例子。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（强制转换）

在前一节中已经介绍过，在计算混合表达式时，需要先将整型数转换成浮点数（末尾添加.0为小数部分）。我们把将某种类型的数据自动转换成另一种类型，称为隐性数据类型转换（Implicit Type Coercion）。前面的例子已经说明，如果没有这种自动数据类型的转换，运算结果将会难以预料。

除了隐性数据类型转换，C++还提供了强制数据类型转换，即显性数据类型转换。强制转换运算符（Cast operator），也称类型转换（Type conversion）或称类型强制转换（Type casting）。使用强制转换运算符进行数据类型显性转换的形式为：

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

首先计算表达式 expression 的值。然后，将其转换成 dataTypeName 所指定的类型。在 C++ 中，static_cast 是保留字。

将浮点型数据转换成整型数据只是简单地将浮点型数据的小数部分截掉，也就是说，不进行四舍五入进位。下面例子说明了强制转换运算符是怎样做类型转换的。一定要弄清楚最后两个表达式是怎样计算的。

例 2.10

表达式	计算过程
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 (因为 static_cast<double>(15)=15.0) =15.0/2.0=7.5
static_cast<double>(15/2)	= static_cast<double>(7)(因为 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
```

考虑另一些例子。在这些例子中，x = 15，y=23，z=3.75。在 C++ 中，表达式的值如下所示：

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，强制转换运算符还可以采用以下形式：dataType(expression)。这种形式称为类 C 强制转换（C-like casting）。然而，static_cast 形式比类 C 形式更加可靠。

可以使用强制转换运算符显性地将char型数据转换成int型数据，或者将int型数据转换成char型数据。将 char 型数据转换成 int 型数据会用到字符编码值。例如，在 ASCII 码字符集中，static_cast <int> ('A') = 65，static_cast <int> ('8') = 56。相反，static_cast <char> (65) = 'A'，static_cast <char> (56) = '8'。

本章前面已经介绍了C++表达式的构成和计算。如果想在一个表达式中用到另一个表达式的结果，则必须先将另一个表达式的计算结果存储起来。还有许多情况需要存储表达式的计算结果。比如，一些表达式很复杂，需要耗费大量的计算机时间来计算。这时，只需计算一次表达式的值，并将其结果存储起来以便以后使用，不仅可以节省计算机时间，缩短程序执行时间，而且可以避免再次键入时可能发生的错误。在 C++ 中，表达式的值如果不被存储，就会丢失。

因此，表达式的值如果不经存储，就不能在以后的计算中使用。在下一节中，将介绍如何存储和使用表达式的值。

2.5　输入

前面已经讲过，C++ 程序的主要作用是进行数值计算和数据处理。而且，所有的数据必须输入到内存中才能进行处理。在本节中，主要介绍如何将数据输入到计算机内存中。将数据输入到计算机内存中需要两个步骤：

1. 在计算机中分配内存单元。
2. 在程序中使用语句将数据输入到分配好的内存单元中。

2.5.1　给常量和变量分配内存

在为内存中的数据分配存储单元时，不仅要知道数据在内存单元中的位置，而且要知道在内存单元中存储数据的类型。知道数据在内存单元中的位置是极为重要的，因为某一存储单元中的数据，可能会在程序中的不同地方多次用到。前面已经提到，知道已存储数据的类型也是极为重要的。因为只有知道数据类型，才能对其进行正确的计算。而且，还要知道存储的数据在整个程序执行过程中是否要发生改变。

一些数据很特殊。例如，所有兼职员工的工资率通常都是相同的；将英寸[1]转换为厘米的转换公式也是固定的，因为1英寸等于2.54 cm。这种数据在内存中存储时需要加以保护，防止在程序执行过程中

① 1英寸=2.54 cm——编者注。

意外地被改动。在C++中，可以使用命名常量来标识分配的内存单元，这种内存单元中的数据在整个程序执行过程中是固定不变的。

命名常量(Named Constant) 所存储的内容在整个程序执行过程中固定不变的内存空间。

还有一些数据的值，在程序执行过程中需要改变。例如，在每次考试后，每个学生的平均成绩和课程数都会发生改变。同样，在每次工资上调后，员工的工资率也会发生改变。这些数据必须存储在内容可以在程序执行过程中被改变的存储单元中。在C++中，存储内容在程序执行过程中可以发生改变的内存单元，称为变量。更加正式的定义如下所示。

变量(Variable) 存储内容可以在程序执行过程中发生改变的内存空间。

C++使用定义语句（Declaration Statement）来分配存储空间。定义命名常量的语法如下所示：

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

其中const是C++的保留字。

例2.11 考虑下列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分配一个double类型数据的存储空间，并将其命名为conversion，并存储数值2.54。在含有该语句的程序中，只要用到了将英寸转换为厘米的转换公式，就可以使用存储在conversion中的这个数值。其他语句的含意类似。

使用命名常量与直接使用数据本身相比，有一个主要的优点。当该数值需要修改时，不必在程序中所有使用到该数值的地方都做改动。而只需要修改程序中该命名常量的定义，重新编译，然后就可以使用新值进行计算了。另外，使用命名常量还可以避免在程序中多处用到同一数据时发生键入错误。如果拼错了命名常量名，编译器会报错误信息。而如果数值写错了，编译器将不会发出任何警告信息。

定义一个或多个变量的语法形式如下所示：

```
dataType identifier1, identifier2, . . . ;
```

例2.12 考虑下列语句：

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

第1条语句告诉编译器分配一个double类型数据的存储空间，并将其命名为amountDue。第2、3条语句与第1条相似。第4条语句则告诉编译器分配两个int型数据存储空间，第1个存储空间的名字是x，第2个存储空间的名字是y。第5条语句告诉编译器分配一个string类型的存储空间，并将其命名为name。

从现在开始，本书使用“变量”一词来表示变量的存储空间。

注意： 在C++中，所有的变量在使用前必须先定义。如果在程序中使用了一个未经定义的变量，编译器将报错误信息，提示该变量未经定义。

注意： 在介绍和讨论完数据类型、变量和常量的基本概念以后，可以给出简单数据类型的正式定义。如果将一个变量或命名常量称为“简单”数据类型，就是说在某一时刻，在该变量或命名常量中只能存储一个数值。例如：如果x是一个int类型变量，则在某一时刻，在x中只能存储一个整型数值。

2.5.2　将数据存储到变量中

在介绍完怎样定义变量之后，下一个问题是：怎样将数据存储到已定义好的变量中？在C++中，可以使用两种方法将数据存储到变量中：

1. 使用 C++ 的赋值语句
2. 使用输入（读入）语句

赋值语句

赋值语句的形式如下所示：

```
variable = expression;
```

在赋值语句中，表达式值的数据类型必须和变量的数据类型相符。赋值语句的执行过程是这样的：先计算等号右边表达式的值，再将计算好的值存储到等号左边的变量（内存空间）中。

变量的初始化（Initialized）是指：第一次将数据值存储到变量中。

在 C++ 中，运算符 "=" 称为赋值运算符（Assignment Operator）。

例 2.13　假设有下列的变量定义：

```
int I, J;
double sale;
char first;
string str;
```

现在，考虑下列变量赋值语句：

```
I=4;
J=4*5-11;
sale=0.02*1000;
first='D';
str="It is a sunny day.";
```

对于每个赋值语句，计算机首先计算等号右边表达式的值，然后再将计算结果存储到等号左边的标识符所对应的内存空间中。第 1 条语句将 4 存储到变量 I 中；第 2 条语句将 9 存储到变量 J 中；第 3 条语句将 20.00 存储到变量 sale 中；第 4 条语句将字符 'D' 存储到变量 first 中；第 5 条语句将字符串 "It is a sunny day." 存储到变量 str 中。

一个 C++ 语句，例如：

```
I = I + 2;
```

的意思是：先取变量 I 中的值，将 I 中的值加 2，再将计算结果存储到变量 I 中。即，必须先计算等号右边算术表达式的值，再将计算结果存储到等号左边变量所表示的内存空间中。因此，语句：

```
I = 6;
I = I + 2;
```

和语句：

```
I = 8;
```

的结果是一样的，都是将 8 赋给变量 I。需要注意的是，如果变量 I 未经过初始化，则语句 I = I + 2 是无意义的。

一旦新值赋给 I，I 中存储的原有数值将丢失。

假设，I，J 和 K 都是 int 型变量。下列语句：

```
I=12;
I= I+6;
J= I;
K= J/2;
K= K/3;
```

的执行结果是 K 等于 3。在一系列语句中跟踪变量值的变化，称为“走查”（Walkthrough）。走查是学习程序设计语言和编写代码的有力工具。在以后章节中，将进一步介绍如何对程序中的语句进行走查。

注意： 假设 x，y，z 都是 int 型变量。在 C++ 中，下列语句是合法的：

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

在这条语句中，首先将 z 的值赋给 y，然后再将 y 的值赋给 x。因为，赋值运算符“=”的计算顺序是从右到左，所以称赋值运算符的结合律是从右到左。

在学完怎样定义变量和怎样将数据存储到变量中以后，下面将介绍怎样存储表达式的值。如果某一表达式的值已被存储，就可以在以后出现该表达式时，直接使用该值，而不是使用表达式本身。这样，本章前面提出的问题也得到了解决。为了存储表达式的值并在以后使用，需要做以下事情：

1. 定义与表达式值类型相同的变量。例如，如果表达式的值是 int 类型，则要定义 int 类型的变量。
2. 使用赋值语句将表达式的值赋给定义好的变量。本操作将表达式的值存储到变量中。
3. 在以后所有用到该表达式值的地方，直接使用该变量来代替表达式本身。

下面例题将说明这一概念：

例 2.14 假设变量的定义如下：

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

再假设需要计算表达式 $-b + (b^2 - 4ac)$ 和 $-b - (b^2 - 4ac)$，并将这两个表达式的计算结果存储到变量 x 和 y 中。因为子表达式 b^2-4ac 在两个表达式中同时出现，所以可以先计算子表达式的值并将其存储到变量 d 中。然后，在下面语句中直接使用变量 d 中子表达式的值：

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

前面已经提到，如果在表达式中使用变量，那么必须先将这些变量初始化，表达式才会得出有意义的结果。我们还学会了如何给定义后的变量赋值。C++ 还允许同时完成变量的定义和初始化。在了解怎样使用输入（读入）语句前，先来介绍这个重要问题。

定义和初始化变量

在定义变量时，C++ 并不自动在其存储空间中存入任何有意义的值。也就是说，C++ 并不自动初始化变量。例如，C++ 在定义 int 和 double 类型变量时，并不像有些程序设计语言那样，将其同时初始化为 0。但是，这并不意味着在定义之后，变量中什么内容也没有。定义变量，也就是将相应大小的内存空间分配给该变量。

在第 1 章中曾经说明，内存是由一些连续的有序存储单元组成的。每一个存储单元都可以存储数据。并且，内存单元中的数据只是位的序列，而位也只是电子信号而已。因此当计算机启动后，一些位是 0，而另一些位则是 1。究竟哪些位应该是 0，而哪些位应该是 1 完全取决于系统的定义。但是，如果在计算机的内存单元中指定存储某个数据，那么这些位的状态就由存入的数据来确定。

在数据处理期间，计算机仅将数据存储在指定的某个存储单元中来完成计算。如果只定义了变量而没有保存其值，那么计算机内存中的值仍是上一次的值，而人们并不知道哪些值是有效的。

如果只定义了变量而没有给变量赋值，那么变量所指内存单元中的数据就是垃圾数据。但是，计算机并不能判断内存中的哪些数据是有效的，并使用这些有效数据进行下一步的计算。使用未经初始化的变量将会导致错误的计算结果。为了弥补这一缺陷，C++允许在定义变量的同时进行初始化。下面是一些定义变量后进行初始化的例子：

```
int first, second;
char ch;
double x, y;

first=13;
second=10;
ch='';
x=12.6;
y=123.456;
```

也可以使用下列C++语句同时完成变量的定义和初始化：

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

第一条C++语句定义了两个int类型的变量：first和second，并给first赋值为13，给second赋值为10。

注意： 实际上，并不是所有的变量都要在定义时赋初值。而是根据程序的需要或是程序员的选择来决定哪些变量需要在定义时赋值。但是，需要注意的是所有的变量必须在赋值以后才可以使用。

输入（读入）语句

在前面一节中，已经介绍了如何使用赋值语句给变量赋值。在本节中，将说明怎样使用C++的输入（读入）语句将数据从标准输入设备读入到变量中。

注意： 在大多数情况下，标准输入设备是键盘。

当使用键盘向计算机输入数据时，称使用者在进行交互操作。

使用标准设备向程序中的变量输入数据，是通过cin和运算符“>>”来实现的。cin和运算符“>>”的使用语法是：

```
cin >> variable1 >> variable2 . . . ;
```

这条语句称为输入（读入）语句。也可以称之为cin语句。在C++中，“>>”称为流析取操作符（Stream Extraction Operator）。

输入语句的工作过程如下，假设miles是double类型的变量，语句：

```
cin >> miles;
```

告诉计算机从标准输入设备中读入一个double类型的数据，并将其存储到名字为miles的内存空间中。

在cin语句中可以使用多个变量名来一次读入多个数据。假设feet和inch都是int类型变量。下面语句：

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

将一次从键盘上读入两个整型数据，并把它们分别存储到名字为feet和inch的内存空间中。

注意： 在程序执行过程中，如果需要在一行中输入多于一个数据时，需要用空格符或制表符来分隔不同的数据项。另外，还可以通过每个数据项单独占一行的方法来一次输入多个数据。

如前所述，可以使用两种方法给变量赋值：使用赋值语句或者使用输入语句。考虑如下的变量定义：

```
int feet;
```

既可以使用如下的赋值语句将数值35赋给变量feet：

```
feet = 35;
```

又可以使用如下的输入语句在程序执行过程中将数值35赋给变量feet：

```
cin >> feet;
```

如果使用赋值语句来初始化变量feet，那么在每次程序执行该语句时，变量feet中的值都固定不变（都为初始值）。当然，也可以通过修改源程序代码，重新编译，重新运行的方法改变变量feet中的值。而使用输入语句，可以在每次程序运行中将值输入到变量feet中。因此，输入语句要比赋值语句使用得频繁一些。

有时，必须使用赋值语句来初始化变量。特别是对于那些只用于内部计算，而不是用于输入和存储的数据来说，更是如此。

注意：C++并不能在定义变量时自动完成变量的初始化。变量可以在定义时初始化，而没有在定义时初始化的变量必须在后续的代码中通过赋值语句或是输入语句来完成初始化。

注意：假设需要使用输入语句将一个字符输入到char类型变量中。在程序执行过程中，只需要输入字符本身，而不必输入单引号。例如，假设ch是一个char类型的变量。考虑如下的输入语句：

```
cin >> ch;
```

如果想通过该语句输入k，那么在程序执行过程中，只需要输入字符k（而不是'k'）。同样，如果想使用输入语句将字符串输入到一个string类型变量中，那么在程序执行过程中，也只需要输入不加引号的字符串。

例2.15 本例进一步说明了怎样使用赋值语句和输入语句给变量赋值。考虑如下的变量定义：

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

假设下列语句按给定的顺序执行：

```
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;
```

再假设输入的数据是：

```
8  16.3  Goofy  D
```

输入行中共有4个数值，8，16.3，Goofy和D，并且值与值之间用空格符分隔。

观察各变量值在每一条语句执行后的变化情况。为了清晰地反映出特定语句将如何改变变量值，我们采用图形来说明每条语句给变量值带来的变化（在图2.5到图2.17中，问号“？”表示该变量的值未知）。

在语句1执行前，所有变量的值均未知，如图2.5所示。

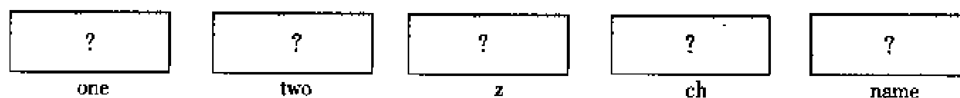


图2.5 语句1执行前所有变量的值

语句1将4存储到名字为one的变量中。在语句1执行后，所有变量的值如图2.6所示。

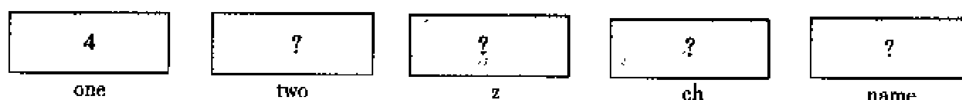


图2.6 语句1执行后所有变量的值

语句2先计算表达式2 * one + 6的值（计算结果是14），并将表达式的计算结果存储到变量two中。在语句2执行后，所有变量的值如图2.7所示。

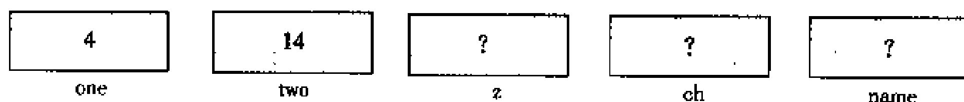


图2.7 语句2执行后所有变量的值

语句3先计算表达式(one + 1) / 2.0（计算结果是2.5），并将表达式的计算结果存储到变量z中。在语句3执行后，所有变量的值如图2.8所示。

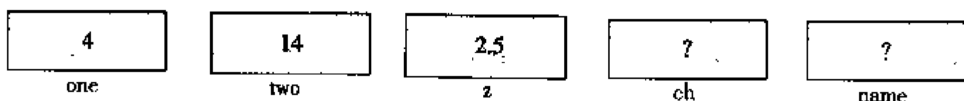


图2.8 语句3执行后所有变量的值

语句4将字符A存储到变量ch中。在语句4执行后，所有变量的值如图2.9所示。

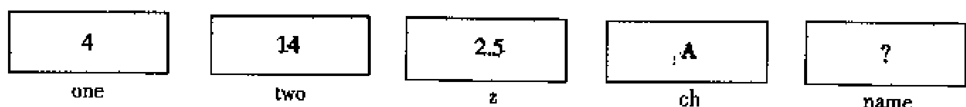


图2.9 语句4执行后所有变量的值

语句5从键盘上读入一个数字（该数字是8），并将数字8存储到变量two中。这条语句把变量two中原有的数据替换成新值。在语句5执行后，所有变量的值如图2.10所示。

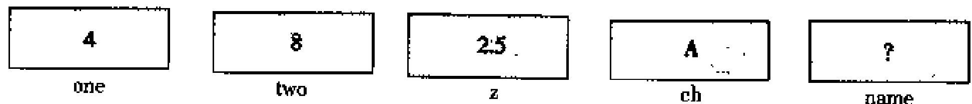


图2.10 语句5执行后所有变量的值

语句6从键盘上读入一个数（该数是16.3），并将该数存储到变量z中。这条语句把变量z中原有的数据替换成新值。在语句6执行后，所有变量的值如图2.11所示。

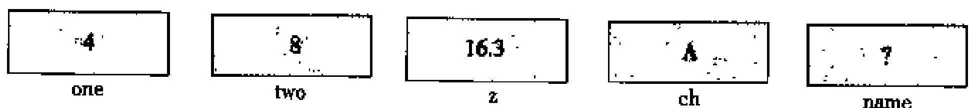


图2.11 语句6执行后所有变量的值

注意变量name仍然没有初始化。语句7先计算表达式2 * two + static_cast <int> (z)(计算结果是32)，并将表达式的计算结果存储到变量 one 中。这条语句把变量 one 中原有的数据替换成新值。在语句 7 执行后，所有变量的值如图 2.12 所示。

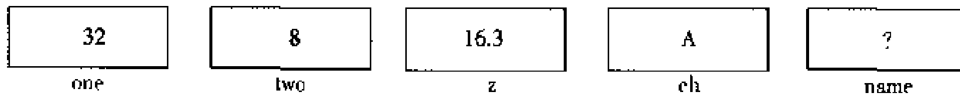


图 2.12　语句 7 执行后所有变量的值

语句 8 从键盘上读入一个字符串（该字符串是 Goofy），并将该字符串存储到变量 name 中。在语句 8 执行后，所有变量的值如图 2.13 所示。

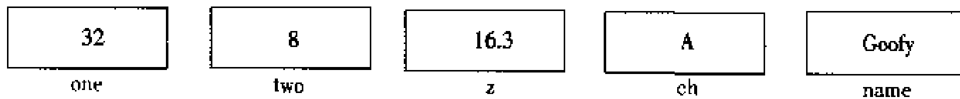


图 2.13　语句 8 执行后所有变量的值

语句 9 先计算表达式 two + 1（计算结果是 9），并将表达式的计算结果存储到变量 two 中。这条语句将变量 two 中原有的数值加 1。在语句 9 执行后，所有变量的值如图 2.14 所示。

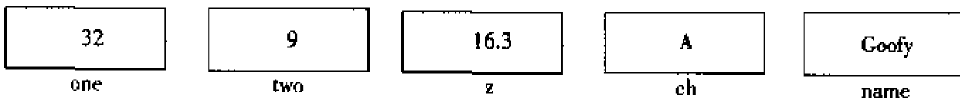


图 2.14　语句 9 执行后所有变量的值

语句 10 从键盘上读入下一个字符（该字符是 D），并将该字符存储到变量 ch 中。这条语句把变量 ch 中原有的数据替换成新值。在语句 10 执行后，所有变量的值如图 2.15 所示。

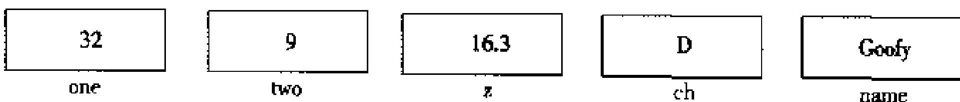


图 2.15　语句 10 执行后所有变量的值

语句 11 先计算表达式 one + static_cast <int>(ch)，它等于 32 + static_cast <int>('D')，又等于 32 +68 = 100。表达式 static_cast <int>('D')的计算结果是 ASCII 码字符集中字符 D 的编码值，即 68。然后，语句 11 将表达式的计算结果存储到变量 one 中。这条语句把变量 one 中原有的数据替换成新值。在语句 11 执行后，所有变量的值如图 2.16 所示。

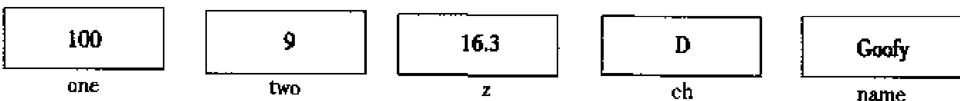


图 2.16　语句 11 执行后所有变量的值

最后，语句 12 先计算表达式 one - z（等于 100 - 16.3 = 100.0 - 16.3 = 83.7），并将表达式的计算结果存储到变量 z 中。在最后一条语句执行后，所有变量的值如图 2.17 所示。

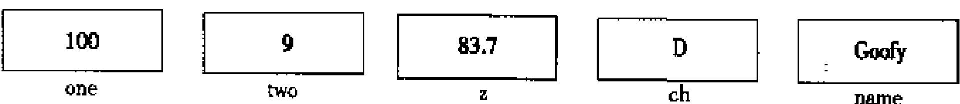


图 2.17　语句 12 执行后所有变量的值

注意：当程序中发生错误并且得到非正常结果的时候，应该对程序中所有改变变量值的语句进行代码走查。例2.15说明了怎样进行程序代码走查。代码走查是很有效地排除程序错误的方法。

注意：如果一个表达式的计算结果是浮点型数，并且不经过强制转换直接赋给int类型变量，则小数部分将被截掉。当这种情况发生时，编译器可能会发出报警信息来警告这种隐性的数据类型转换。

2.6　增量运算符和减量运算符

在介绍了怎样定义变量以及怎样向变量中输入数据之后，在本节中，将介绍另外两个运算符：增量运算符和减量运算符。这两个运算符的使用频率很高，是C++程序员的得力助手。

假设count是int类型变量。语句：

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

将变量count中的值加1。在执行这条赋值语句时，计算机首先计算等号右边表达式count + 1的值。然后，再将计算结果赋给等号左边的变量count。

正如将在后面章节中所见到的，上面的语句被大量地使用在记录某件事情发生的次数中。为了提高这种语句的执行效率，C++提供了两个特殊的运算符：增量运算符“++”，将变量的值加1；减量运算符“--”，将变量的值减1。增量运算符和减量运算符各有两个形式：前置表达式和后置表达式。

增量运算符的使用语法是：

前置增量表达式：　++variable
后置增量表达式：　variable++

减量运算符的使用语法是：

前置减量表达式：　--variable
后置减量表达式：　variable--

下面，让我们看一些举例。语句：

```
++count;
```

或者

```
count++;
```

将变量count中的值加1。类似地，语句：

```
--count;
```

或者

```
count--;
```

将变量count中的值减1。

增量运算符和减量运算符都是C++内建的运算符。所以，可以通过这两个运算符，不使用赋值语句，迅速地实现变量增量或减量。

从上面举例中可以看到，无论是前置增量表达式还是后置增量表达式，都可以将变量的值加1；同样，无论是前置减量表达式还是后置减量表达式，都可以将变量的值减1。那么，前置表达式和后置表达式之间到底有什么区别呢？如果将这两个运算符使用在表达式中，那么这种区别就会很明显地显示出来。

假设x是int类型变量。如果在表达式中使用++x，那么先将变量x的值加1，然后再在表达式中使用x中的新值。相反地，如果在表达式中使用x++，那么表达式中将使用x的当前值，然后再将变量x的值加1。下面举例清楚地说明了前置增量表达式和后置增量表达式之间的区别。

假设x和y是int类型变量。考虑下面的语句：

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

第一条语句将5赋给变量x。在计算含有前置增量表达式的第二条语句时，首先将变量x加1变成6。然后，再将这个新值6赋给变量y。在第二条语句执行后，变量x和y中的值都是6。

现在，再考虑下面的语句：

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

与前面举例中一样，第一条语句将5赋给变量x。在第二条语句中，使用了变量x的后置增量表达式。在执行第二条语句时，先使用变量x中的当前值作为表达式x++的值。然后，将变量x加1变成6。最后，将表达式x++的值5存储到变量y中。在第二条语句执行后，变量x中的值是6，而变量y中的值是5。

下面举例进一步说明了前置增量表达式和后置增量表达式的使用方法。

例2.16 假设x和y是int类型变量，并且有下面的语句：

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

第一条语句将5赋给变量a。在执行第二条语句时，首先计算表达式2 + (++a)的值。因为使用的是变量a的前置表达式，所以先将变量a加1变成6，然后2加6等于8。最后，将8赋给变量b。因此，在第二条语句执行后，a的值是6，b的值是8。

注意：本书将在单独的语句中，大量地使用增量运算符和减量运算符。也就是说，不将变量的增量表达式和减量表达式作为其他表达式的一部分来使用。

2.7 输出

如果不能将计算结果显示到输出设备上，那么程序的作用将变得微乎其微。虽然前面已经介绍过怎样将数据输入到计算机内存和怎样处理数据，但是仍然没有说明该如何显示程序的结果。

注意：标准输出设备通常指显示器。

在C++中，程序向标准输出设备中输出数据，是通过cout和运算符“<<”来实现的。使用cout和运算符“<<”的语法是：

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

这条语句称为输出语句（Output Statement）。有时候也称之为cout语句。在C++中，运算符“<<”称为流插入运算符（Stream Insertion Operator）。

使用cout语句输出数据时，应遵循下面两条规则：

1. 先计算表达式的值，再将其输出到输出设备的当前光标位置上。
2. 控制符（Manipulator）用来定义输出数据的格式。最简单的控制符是endl，这个控制符的作用是使光标移动到下一行的开始处。

下一例子说明了 cout 语句是怎样执行的。

注意： 当使用输出语句输出 char 类型变量时，输出的只是字符本身，而没有单引号（除非单引号本身是要输出的字符）。例如，假设变量 ch 是 char 类型变量，并且 ch = 'A'，那么语句：

```
cout << ch;
```

或者

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

的输出结果都是 A。

同样，使用输出语句输出 string 类型变量时，输出的也将是字符串本身而没有引号（除非引号是要输出字符串的一部分）。

例 2.17　考虑下列语句。输出结果在每条语句的右边。

语句	输出结果
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.

注意语句 9 的输出结果。如前所述，在 C++ 中，换行符是 '\n'。它的作用是使光标移到下一行的开始处。因此，如果\n 出现在输出语句的字符串里时，将使光标移到输出设备的下一行开始处。这也就解释了为什么 Hello 和 there 分别在两行中输出。

注意： 在 C++ 中，反斜杠字符 "\" 被称为转义字符（escape character），\n 称为换行符（newline escape sequence）。

前面已经提到，C++中所有的变量必须要经过初始化才能使用。否则，变量中的值将没有任何意义。而 C++ 并不能自动地初始化变量。C++ 输出语句：

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

只有在变量 a 被赋值后才有意义。例如，C++ 语句：

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

输出结果是 45。

例 2.18　考虑下列语句。假定所有语句都是按书中所给出的顺序执行。该段 C++ 代码的输出结果列在后面。

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

a=65;                                   //Line1
b=78;                                   //Line2

cout<<29/4<<endl;                       //Line3
cout<<3.0/2<<endl;                      //Line4
```

```
cout<<"Hello there.\n";                     //Line5
cout<<7<<endl;                               //Line6
cout<<3+5<<endl;                             //Line7
cout<<"3+5";                                 //Line8
cout<<endl;                                  //Line9
cout<<2+3* 6<<endl;                          //Line10
cout<<"a"<<endl;                             //Line11
cout<<a<<endl;                               //Line12
cout<<b<<endl;                               //Line13
cout<<c<<'\n';                               //Line14
cout<<d;                                     //Line15
cout<<endl;                                  //Line16
```

在下面的输出结果中，标题“输出结果所对应的行号”和行号并不是输出结果的组成部分。行号的作用是可以方便地找出输出结果所对应的语句行。

```
                                             输出结果所对应的行号
7                                            Line3
1.5                                          Line4
Hello there.                                 Line5
7                                            Line6
8                                            Line7
3+5                                          Line8
20                                           Line10
a                                            Line11
65                                           Line12
78                                           Line13
6749684                                      Line14
4203005                                      Line15
```

本例中大部分结果是不言而喻的。注意第7行、第8行、第9行和第10行语句的输出结果。第7行语句输出表达式3+5的计算结果8，并将光标移到下一行的开头处。第8行语句输出字符串3+5。注意第8行语句的输出部分中只有字符串3+5，因此在输出3+5后，光标仍停留在5后面的位置上，而不是移到下一行的开头处。

第9行语句的输出部分中只有控制符endl。它的作用是使光标移到下一行的开头处。因此，当第10行中的语句执行后，结果输出到下一行的开头处。需要注意的是，“输出结果所对应的行号”列中，并没有列出第9行语句的输出结果。这是因为第9行语句输出的是不可打印的字符。它只是将光标移到下一行的开头处。第10条语句输出表达式2 + 3 * 6的计算结果20。控制符endl将光标移到下一行的开头处。

需要特别注意的是第14行和第15行语句。变量c和d并没有经过初始化。因此第14行语句输出的变量c的值6749684和第15行语句输出的变量d的值4203005，都是没有意义的。如果在你的系统中运行这两条语句，可能会得到不同的c和d的值。

注意：输出和使用表达式中变量的值并不会破坏和修改变量的值。

现在让我们认真研究换行符'\n'。考虑如下C++语句：

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

如果按顺序执行上面语句，则输出结果为：

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

现在考虑下面的C++语句：

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

这两条语句的输出结果是：

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

当\n出现在字符串中时，它的作用是将光标移动到下一行的开头处。要特别注意\n可以出现在字符串中的任何位置上。例如，下面语句：

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

的输出结果是：

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

同时注意输出语句：

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

与输出语句：

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

与下面的输出语句作用相同：

```
cout << endl;
```

都是将光标移到下一行的开头处。

因此，输出语句：

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

与下列输出语句的作用是一样的：

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

例2.19 考虑下面的C++语句：

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

或者：

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

或者：

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

这三段代码的输出结果都是：

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

例2.20　C++语句：

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

和语句：

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

的输出结果都是：

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

例2.21　假设把如下的句子在屏幕上的一行中输出：

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

显然，需要使用cout语句输出这些句子。然而在语句行里，如果使用一条cout输出这个句子，则很可能需要占用两行。当然，可以使用多条cout语句，例如：

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

注意，在第一行语句的后面使用了分号，第二行语句的开头又使用了标识符cout，说明它们是不同的两条语句。还要注意，第一行语句后面没有出现控制符endl。这两条语句的作用是在同一行中输出这个句子。同样，也可以使用下面的语句达到同样的目的：

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

注意，在这条语句中，第一行的后面没有出现分号，第二行的开头也没有使用标识符cout。因为第一行语句后面没有分号，所以cout语句在同一行中继续输出第二个句子。还要注意的是，这里每一个句子都用双引号引住。虽然，同一字符串被分成了两个，但它们都是同一cout语句的组成部分。

如果cout语句中的字符串很长，而又想将其在同一行上输出，那么可以使用上述两种方法将其分割。但是，下面的语句是不正确的：

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

也就是说，回车（Enter）键不能用在字符串中。在程序中，不能使用键盘上的回车键将字符串分割在不同的行上。

前面提到的换行符\n，其作用是将光标移到下一行的开头处。C++中还有一些其他转义字符（Escape Sequence），使用户可以方便地控制输出。表2.4列出了一些常用的转义字符。

表2.4　常用的转义字符

	转义字符	说明
\n	换行符	将光标移到下一行的开头处
\t	水平制表符	将光标移到下一个制表符位置上
\b	退格符	将光标向左移动一个字符位置
\r	回车符	将光标移到本行（而不是下一行）的开头处

（续表）

	转义字符	说明
\\	反斜线	输出反斜线
\'	单引号	输出单引号
\"	双引号	输出双引号

下列例子说明了转义字符的作用。

例2.22 语句：

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

的输出结果是：

```
The newline escape sequence is \n
```

语句：

```
cout << "The tab character is represented as \'\\t\'" << endl;
```

的输出结果是：

```
The tab character is represented as '\t'
```

语句：

```
cout << "The string \"Sunny\" contains five characters\n";
```

的输出结果是：

```
The string "Sunny" contains five characters
```

注意： 只有在程序中包含了特定的头文件，cin和cout语句才能正常使用。下一节，将介绍什么是头文件，怎样使用头文件，以及为什么要使用头文件。在第3章中，将详细说明cin和cout究竟是怎么一回事。

2.8 预处理指令

C++只明确定义了很少几个运算符，如算术运算符和赋值运算符。在C++程序中使用到的大部分函数和符号，都是由库文件提供的。每一个库文件都有一个名字，并通过头文件来引用。例如，在ANSI/ISO标准C++中，用来执行输入/输出（I/O）操作的函数都包含在头文件iostream中；用来执行数学运算功能的函数，如乘方函数、绝对值函数和正弦函数，都包含在头文件cmath中。如果要在程序中执行I/O操作或是进行数学运算，就必须要告诉计算机该如何找到这些必要的文件。可以使用预处理指令和头文件名来告诉计算机如何找到这些库文件。预处理指令由一个叫做预处理程序（Preprocessor）的程序来处理。

预处理指令是一些由预处理程序提供的命令，这些命令可以在编译以前改变源程序中的内容。所有的预处理命令都以#号开头，并且在末尾处都没有分号，因为预处理命令并不是C++语句。可以使用预处理指令include将头文件包含到C++程序中。

在C++程序中包含头文件（SDK提供）的通用语法是：

```
#include <headerFileName>
```

例如，下面的语句可以将头文件iostream包含在C++程序中：

```
#include <iostream>
```

注意： 用来包含头文件的预处理指令应该放在程序的开头处。因为只有这样，才可以在整个程序范围内使用定义在这些头文件中的标识符（前面讲过，C++中的标识符必须在定义之后才可以使用）。

一些特定的头文件是由ANSI/ISO标准C++提供的。附录F中列出了一些常用的头文件。当然，每个程序员也可以创建自己的头文件。创建和使用自己的头文件的方法将在第12章中介绍。

在标准C++中，头文件都带有扩展名“.h”（而在ANSI/ISO标准C++中，头文件没有扩展名）。例如，在标准C++中，用来执行I/O操作的函数都包含在头文件iostream.h中。许多很有用的数学函数都在名为math.h的头文件中。因此，在标准C++中程序中包含头文件iostream.h，需要使用如下语句：

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

注意： 附录E中介绍了ANSI/ISO标准C++头文件名和标准C++头文件名的转换方法。

预处理命令由预处理程序在源文件被编译以前处理。因此，执行C++程序总共需要6个步骤：编辑、预处理、编译、连接、装载和执行。图2.18中说明了执行一个C++程序需要的6个步骤。

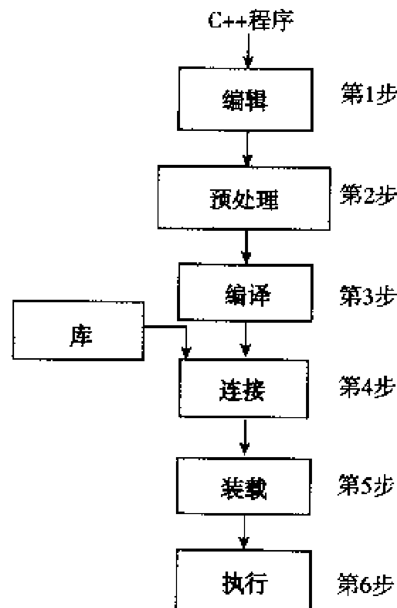


图2.18 C++程序的执行过程

从图2.18中可以看出，C++系统有3个基本的组成部分：程序开发环境、C++程序设计语言和C++库。所有这些都是C++系统的必要组成部分。程序开发环境由图2.18中的6个步骤组成。C++程序设计语言将分散在本书各个章节中论述，而C++库也将在必要时予以介绍。

2.8.1 cin和cout与名字空间

前面已经讲过，cin和cout都是预定义的标识符。在ANSI/ISO标准C++中，这些标识符定义在头文件iostream中，并且属于某个名字空间（namespace）。该名字空间的名字为std（名字空间的正式定义和详细讨论将在第8章中介绍。现在，只需要知道怎样使用cin，cout以及iostream中的其他标识符）。

可以采用不同的方法使用名字空间std中的标识符。一种方法是在整个程序中使用std::cin和std::cout来代替标识符cin和cout。另一种方法是在程序中包括下面的语句：

```
using namespace std;
```

这条语句应该紧跟在语句#include <iostream>之后。这样，就可以不加前缀“std::”而直接使用标识符cin和cout了。为了简化标识符cin和cout的使用，本书中采用了第二种形式。也就是在程序中直接使用标识符cin和cout，但要在程序中包含下列语句：

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

在C++中，namespace和using都是保留字。

注意：名字空间机制是ANSI/ISO标准C++的新特性。随着学习C++程序设计知识的深入，将了解到另外一些头文件。例如，头文件cmath中包含了许多功能强大的数学函数；头文件iomanip中包含了许多函数和控制符。通过这些函数和控制符，可以方便地制订数据输出格式。然而，正如头文件iostream中的标识符一样，所有ANSI/ISO标准C++库文件中的标识符都必须包含在特定的名字空间中。

这些头文件都包含在名字空间std中。因此，在讨论到ANSI/ISO标准C++头文件中的这些标识符时，本书将采用没有前缀“std::”的格式。另外在本书所示的程序中也将使用“using namespace std;”来简化使用标识符。注意，如果程序中涉及到多个头文件，也只需要使用一个“using namespace std;”语句。这时，using语句应该放在所有的头文件之后。

在标准C++中，cin和cout在头文件iostream.h中定义。因此，如果在标准C++程序中使用cin和cout，则必须在程序中包含如下语句：

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

2.8.2 在程序中使用string数据类型

前面已经提过，string数据类型是程序员定义的数据类型，并不能直接在程序中使用。要想使用string数据类型，就必须用到在头文件string中的定义。因此，为了使用string数据类型，必须在程序中包含如下的预处理指令：

```
#include <string>
```

一些编译器不允许通过使用标准C++格式头文件（带有扩展名.h的头文件）来使用string数据类型。也就是说，如果要使用string数据类型，那么程序中所有头文件都必须是ANSI/ISO标准C++格式，即没有.h扩展名。在使用cin，cout和string数据类型的程序中，必须包含下列预处理指令和语句（参阅编译器的说明文档）：

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

注意：在本章的程序范例中所有的源代码，都有ANSI/ISO标准C++和标准C++两种格式。本书中除了第8章以外所有章节中的程序代码，都使用ANSI/ISO标准C++格式头文件。在接下来的几章中，如果需要，也将给出与ANSI/ISO标准C++格式头文件相对应的标准C++格式头文件。在与本书相对应的Internet站点上，也同时以两种格式给出程序源代码。

2.9 程序的风格和形式

通过前面章节对C++程序基本概念的学习，现在可以编制一些实用的小程序了。但是在着手编写程序代码之前，应该先了解什么是结构良好的程序。结构良好的C++程序不仅易于理解，而且便于修改。试图去理解或修改语法正确但结构混乱的程序代码是一件极为头痛的事情。

所有的C++程序都由两部分组成：预处理指令和程序。预处理指令的作用是：告诉预处理程序在源程序被编译以前应该对代码该做怎样的改动。程序则由语句组成，并能够完成一定的功能。预处理指令和程序语句一起组成了C++的源代码。源代码必须存储在扩展名为.cpp的源文件中。

当程序通过编译后，编译器将生成目标代码。目标代码存储在文件扩展名为.obj的目标文件中。当目标代码与某些系统资源连接后，将生成可执行代码。可执行代码存储在文件扩展名为.exe的可执行文件中。目标文件、可执行文件与源文件的名称都一样，区别只在于扩展名不同。例如，如果用来存储源代码的源程序名叫firstProg.cpp，那么目标代码将存储在firstProg.obj中，而可执行代码则存储在firstProg.exe中。

注意： 上面给出的文件扩展名.cpp，.obj和.exe是依系统而定义的。为保证正确使用，请参阅所使用系统SDK的说明文档。

所有的C++程序必须遵循一定的语言规则。C++程序必须包括一个main函数；程序源代码还必须符合语法规则。程序设计语言的语法规则与人类语言的语法规则一样，规定什么是正确的，什么是错误的；哪些是合法的，哪些是非法的。另外一些规则的作用是保证语言在使用上的准确性，也就是说保证语言的语义。接下来的一节将讲述怎样使用各种C++程序设计元素来构建实用程序。在本节中将介绍：main函数、语法规则、空格键的使用、分号的使用、花括号的使用、逗号的使用、语义、形式和风格、说明文档（包括注释和标识符命名）以及提示行。

2.9.1 main函数

前面已经说明，所有的C++程序都必须有一个main函数。最基本的main函数由函数头和函数体两部分组成。函数头的形式如下所示：

```
typeOfFunction main(argument list)
```

例如，语句：

```
int main(void)
```

表明函数main将返回一个int类型的值，并且没有参数。虽然参数列表中的void不是必需的，但是参数列表的括号却是必需的。因此，该语句与下面语句等价：

```
int main()
```

在C++中，void是保留字。

函数体必须用一对大括号括起来，通常包含两种类型的语句：

- 定义语句
- 可执行语句

定义语句的作用是定义诸如变量等标识符。在C++中，变量和标识符可以定义在程序的任何位置，但必须在使用以前定义。

例2.23 下面是一些变量定义语句：

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

可执行语句的作用是执行计算、数据处理、输出和输入，等等。我们已经接触到的一些可执行语句，如赋值语句、输入语句和输出语句。

例2.24　下面是一些可执行语句：

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

本书中的main函数的语法如下所示：

```
int main()
{
  statement1
      .
      .
      .
  statementn
  return0;
}
```

在main函数的语法中，每一条语句（statement1，…，statementn）不是定义语句就是可执行语句。C++程序的函数体中必须包含return语句，而且必须作为最后一条语句出现。如果return语句被错放在main函数体的其他部分，可能会出现意想不到的后果。语句“return 0;”的作用将在第6章中予以介绍。在C++中，return是保留字。

2.9.2　语法

程序设计语言的语法规则规定什么是合法的语句，什么是非法的语句。编译器负责检查语法错误。例如，考虑下面的C++语句：

```
int x;        //Line1
int y         //Line2:syntax error
double z;     //Line3

y= w+ x;      //Line4:syntax error
```

在编译这些代码时，第2行将出现编译错误，因为在变量y的定义语句后面没有使用分号；第4行也会出现编译错误，因为使用了未经定义的标识符w。

第1章中已经说明，源程序最开始是通过文本编辑器输入到计算机中去的。在键入源程序代码时，不可避免地会出现这样或那样的错误。因此，在程序编译过程中，很有可能会报出语法错误。而且，如果程序中出现一处错误，通常会导致接下来几行也会出现错误。经常会出现只漏写了一个字符，编译器却报出七、八条错误的情况。然而，一旦改正了第一处语法错误，并重新编译程序，由这个语法错误引起的其他错误也就一同解决了。当对C++程序设计语言越来越熟练的时候，你将知道怎样迅速的找到并改正这些语法错误。编译器不仅可以发现语法错误，而且还可以提示甚至告诉程序员错误出现在哪里以及如何改正这些错误。

2.9.3　空格符的使用

在C++中，一个或多个空格符可以用来分隔输入数据中的不同数据项。空格符还可以用来分隔保留字、标识符和其他一些符号。但是，保留字和标识符中绝对不可以出现空格符。

2.9.4　分号、大括号和逗号的使用

所有的C++语句必须用分号结尾以标志语句结束。分号也被称为语句结束符（Statement Terminator）。

需要注意的是，大括号并不是C++语句本身的一部分。大括号可以被视为界定符号，因为它将函数体括住，并将其与程序的其他部分分隔开来。以后章节中还将介绍大括号的其他作用。

逗号的作用是分隔列表中的数据项。在定义同一类型的多个变量时，还可以使用逗号来分隔不同的变量名。

2.9.5 语义规则

由程序设计语言中的一系列规则所确定的含义称为语义（Semantics）。例如，算术运算符的优先顺序规则就是语义规则。

如果程序中有语法错误，编译器一定会报出警告信息。但是，如果程序中含有语义错误，那么将会怎样呢？很有可能出现虽然已经消除了程序中的所有语法错误，但是程序仍然不能运行的情况。即使是程序可以运行，也可能出现并非预期的结果。例如，下面两个表达式在语法上都是正确的，但却有不同的含义：

```
2 + 3 * 5
```

和

```
(2 + 3) * 5
```

如果这两个表达式在程序中被误用，那么将出现非预期的结果。虽然表达式中的数字是相同的，但是含义却有很大差别。本书将详细介绍语义规则。

2.9.6 形式和风格

你或许觉得C++中的规则过于繁琐。但事实上，也正是这些规则给予C++程序设计极大的灵活性。例如，考虑下面两种定义变量的方法：

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

和

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

这两种变量定义的方法，在计算机看来是等效的。但是，前一种方法更加清晰易懂。当然无论是这两种方法中的哪种方法，如果漏写了一个分号或是逗号，都会导致出现编译错误信息。

应该怎样恰当地使用空格符呢？什么时候会对程序产生影响，什么时候又不会产生影响呢？考虑下面两条语句：

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

和

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

这两个变量定义都是正确的。这里第二条语句中的多余空格符不会对程序产生任何影响。再考虑下面的语句：

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

这条语句在语法上是错误的。由于在保留字int和标识符a之间没有使用空格做分隔符，编译器将认为这是一个新变量inta。

语法规则和语义规则的严谨性，可以使程序员自由地选用自己喜欢而且易于理解的风格进行程序设计。

2.9.7　说明文档

一个好的程序员编写出来的程序不仅可以自己看懂，而且还可以让别人看懂。因此，必须要恰当地给程序做注释。注释清晰的程序即使在很长时间之后也很容易理解和修改。可以使用注释来帮助理解程序。程序中的注释应该说明程序的用途、标明作者以及解释特定语句的含义。

注释

C++ 中有两种类型的注释：单行注释和多行注释。单行注释以双斜线 // 开头，可以出现在程序行中的任何位置。编译器将忽略掉 // 后面的所有内容。多行注释以 /* 开头，并以 */ 结尾。编译器将忽略掉 /* 和 */ 中间的所有内容。

可以在程序的开头处插入一些关于程序和程序编写者的注释信息。也可以在程序的关键步骤前面插入一些关于这些步骤的解释信息。

标识符命名

在给标识符命名时，应该选用有意义的名字。例如，考虑下面两段代码：

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

x= y * a;
```

和

```
const double conversion=2.54;
double centimeters;
double inches;

centimeters= inches * conversion;
```

正如你所见到的，选用有意义的标识符名可以简化注释。

选用单词组合成有意义的标识符，如 inchperfoot，可以减轻程序注释的工作量。还可以通过大写每一个单词的首字母或是在单词间加下划线来使这些标识符容易理解。例如，可以选用 inchPerFoot 或是 inch_per_foot 来增加标识符的清晰性。

提示行

一个好的程序在同计算机交互时可以通过清晰的提示信息指示用户下一步该怎样做。没有什么事情比坐在计算机前苦思冥想该键入何种数据更使人感到沮丧了。提示行是用来提示用户该做什么的可执行语句。例如，考虑下面的 C++ 语句，其中 num 是 int 类型变量：

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

当执行上述代码时，第一个 cout 语句将在计算机屏幕上输出如下信息：

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

在看到这一行提示信息后，用户知道要输入一个数字并按下回车键。如果程序中只包含第二条语句，用户将不知道这时该输入数字，因此计算机将处于无休止的等待中。前面的 cout 语句是说明提示行作用的很好例子。

只要在程序中需要用户输入数据，就应该使用提示行。而且，提示行应该尽可能详尽地提示用户什么是有效输入数据。例如，本例中前一条语句不仅告诉用户应该输入数字，而且告诉用户输入的数字应该在1和10之间。

2.10 其他赋值语句

到目前为止，你在本书中见到的都是简单赋值语句。在另一些情况下，可以使用一种名为复合赋值语句的语句来代替多个简单赋值语句，使程序变的更加简洁。对于算术运算符，C++定义了下面一种赋值运算符，这里op是任何一种算术运算符：

```
op =
```

可以使用复合赋值运算符来改写使用简单赋值运算符的代码：

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

可以改写为：

```
variable op = expression;
```

因此，可以通过使用复合赋值运算符将算术运算符和赋值运算符结合在一起使用，达到精简程序的目的。

例2.25 本例中给出了多个复合赋值运算符以及等价的简单赋值运算符。

简单赋值语句	复合赋值语句
`I= I + 5;`	`I += 5;`
`counter = counter + 1;`	`counter += 1;`
`sum = sum + number;`	`sum += number;`
`amount = amount* (interest + 1);`	`amount *= interest + 1;`
`x = x/( y + 5);`	`x /= y + 5;`

注意：任何复合赋值语句都可以转化成简单赋值语句。但是，并不是所有的简单赋值语句都可以（轻易地）转化成复合赋值语句。例如，考虑下面的赋值语句：

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

如果要将其改写成为复合赋值语句，那么赋值号右面的x必须是所有项的公共因子，而本例中却并不是这样。因此，不能直接将其转化成复合赋值语句。实际上，相应的复合赋值语句应该是：

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

这要比简单赋值语句复杂得多。

注意：在本书的程序代码中，通常只使用复合赋值运算符+=。语句“a = a + b;”对应的复合赋值运算符表达式语句是“a += b;”。

2.11 程序范例：长度转换

编写一个程序，该程序读入以英尺和英寸[①]为单位的长度，然后将其转换成以厘米为单位的长度，并将转换结果输出。假定给定的英尺和英寸长度都是整数。

输入 以英尺和英寸为单位的长度。

输出 以厘米为单位的长度。

① 1英尺＝0.3048 m；1英寸＝2.54 cm——编者注。

问题分析与算法设计

输入的长度是以英尺和英寸为单位，必须将其转换成相应的以厘米为单位的长度。1英寸等于2.54厘米。首先，程序应该将以英尺和英寸为单位的长度全部换算成以英寸为单位的长度。然后，使用转换公式，1英寸=2.54厘米，将其换算成以厘米为单位的长度。将以英尺和英寸为单位的长度转换成以英寸为单位的长度的方法是：将英寸数乘以12，因为1英尺等于12英寸，再加上所给的英寸数。

例如，假定输入的长度是5英尺7英寸，则应该按如下方法，将其转换成相应的英寸：

总英寸数 = (12 * 英尺数) + 英寸数
　　　= 12 * 5 + 7
　　　= 67

然后，再使用转换公式，1英寸=2.54厘米，将其转换成相应的厘米数。

厘米数 = 总英寸数 * 2.54
　　= 67 * 2.54
　　= 170.18

在上述问题分析的基础上，设计的算法如下：

1. 读入以英尺和英寸为单位的长度
2. 将其转换成总英寸数
3. 将总英寸数换算成相应的厘米数
4. 输出以厘米为单位的长度

变量

本程序需要读入的数据有两个：英尺数和英寸数。因此，需要定义两个变量：一个存储英尺数，另一个存储英寸数。因为程序还要将其换算成以英寸为单位的长度，所以还要定义一个存储总英寸数的变量。同样，还需要定义一个存储相应厘米数的变量。所以，共要定义如下的变量：

```
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
```

命名常量

为了计算出相应的厘米数，需要将总英寸数乘以2.54。可以定义一个命名常量来避免直接使用数值2.54本身。同样为了计算出总英寸数，需要将英尺数乘以12并加上英寸数。可以定义另一个命名常量来避免直接使用数值12本身。在程序中使用命名常量可以简化以后的修改工作。

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

主要算法

1. 提示用户输入数据（如果没有提示行，用户将在计算机屏幕前感到无所适从）。
2. 读入数据。
3. 将输入数据回显在计算机荧光屏上（没有这一步，在程序执行时，用户将不知道输入的数据是什么）。
4. 转换成以英寸为单位的长度。
5. 输出以英寸为单位的长度。
6. 将以英寸为单位的长度换算成以厘米为单位的长度。
7. 输出以厘米为单位的长度。

程序

在问题分析和算法设计完成后，下一步就是将其在C++代码上实现。因为，这是我们亲自动手编制的第一道程序，先简要回顾一下必要的步骤。

程序的开头处应该有标明程序用途和功能的注释。因为本程序涉及到两个输入数据(英尺数和英寸数)和一个输出数据(相应的厘米数)，所以需要使用系统资源来处理输入/输出。也就是说，程序中要使用输入语句将数据读入计算机，并使用输出语句输出换算结果。因为要使用键盘输入数据并在屏幕上显示结果，所以本程序要使用到头文件 iostream。因此，本程序在标注上述注释内容之后，应该使用预处理指令包含头文件。

本程序需要使用两种数据类型来进行数据处理：命名常量和变量。命名常量通常定义在 main 函数之前，以便可以在程序中任何地方都使用它们。

本程序中只有一个函数，main 函数，所有的程序指令都包含在它的函数体中。程序还需要使用变量来存储数据，这些变量也要定义在函数体中。在 main 函数的函数体中定义变量的原因，将在第 7 章中介绍。main 函数的函数体中还要包括实现算法的 C++ 语句。因此，main 函数的形式如下所示：

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

应该按照以下步骤，编写完整的单位转换程序：

1. 首先标注注释
2. 包含所需的头文件
3. 定义所需的常量
4. 编写 main 函数

完整的 ANSI/ISO 标准 C++ 程序代码清单：

```
//********************************************************
//   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: Step1- Step7
  cout<<"Enter two integers, one for feet,"
      <<"one for inches:";                          //Step1
  cin>>feet>>inches;                                //Step2
  cout<<endl;
  cout<<"The numbers you entered are"<<feet
      <<" for feet"<<"and"<<inches
      <<" for inches."<<endl;                       //Step3

  totalInches= inchesPerFoot* feet+ inches;         //Step4
  cout<<endl;
  cout<<"The total number of inches="
      <<totalInches<<endl;                          //Step5
  centimeter= conversion* totalInches;              //Step6
  cout<<"The number of centimeters="
      <<centimeter<<endl;                           //Step7
  return 0;
}
```

程序运行结果 在本示例中，用户输入的数据加有阴影。

```
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
```

完整的标准 C++ 程序代码清单

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

#include <iostream.h>

    //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:";                          //Step1
  cin>>feet>>inches;                                //Step2
  cout<<endl;
  cout<<"The numbers you entered are"<<feet
      <<" for feet"<<"and"<<inches
      <<" for inches."<<endl;                       //Step3

  totalInches = inchesPerFoot* feet+ inches;        //Step4
  cout<<endl;
```

```
    cout<<"The total number of inches="
        <<totalInches<<endl;                    //Step5
    centimeter= conversion* totalInches;        //Step6
    cout<<"The number of centimeters="
        <<centimeter<<endl;                     //Step7
    return 0;
}
```

2.12 程序范例：零钱换整

编写一个程序，该程序读入以美分为单位的美元货币数，将其换算成相应的以50美分（half-dollar）、25美分（quarter）、10美分（dime）、5美分（nickel）和1美分（cent或者penny）为单位的美元货币数。例如，483美分等于9个50美分，加上1个25美分，加上1个5美分，加上3个1美分。

输入 以美分为单位的货币数

输出 相同数量的以50美分、25美分、10美分、5美分和1美分为单位的货币数

问题分析与算法设计

假定给定的货币数是646美分。为了计算出50美分的数量，需要用50除646，商是12，余数是46。商12即是50美分的数量，而余数46还需要进一步换整。

接下来，用25除46，商是1，也就是25美分的数量。余数是21，还需要进一步换整。将这个过程继续到10美分、5美分和1美分。可以使用求余运算符%，来计算整数除法的余数。

将646美分换整需要以下计算：

1. 货币数 = 646
2. 50美分的数量 = 646 / 50 = 12
3. 余下的货币数 = 646 % 50 = 46
4. 25美分的数量 = 46 / 25 = 1
5. 余下的货币数 = 46 % 25 = 21
6. 10美分的数量 =21 /10 = 2
7. 余下的货币数 =21 % 10 = 1
8. 5美分的数量 =1 / 5 = 0
9. 1美分的数量 = 余下的货币数 = 1 % 5 = 1

通过上述讨论可以得出如下算法：

1. 读入美分货币数
2. 计算50美分的数量
3. 计算余下的货币数
4. 计算25美分的数量
5. 计算余下的货币数
6. 计算10美分的数量
7. 计算余下的货币数
8. 计算5美分的数量
9. 计算余下的货币数
10. 余下的货币数也就是1美分的数量

变量

通过上述的讨论和算法分析，很明显程序中需要存储空间来存储50美分、25美分等单位的货币数。然而50美分、25美分等货币数在后续的计算中并不需要再使用，所以只需要在程序中输出这些数值，而不需要存储它们。事实上，程序中只需要一个变量change来存储每次剩余的美分数量。

```
int change;
```

命名常量

为了将美分货币数换整，程序中需要使用：50美分的值50，25美分的值25，10美分的值10，5美分的值5。因为这些数值既具有特定意义，又在程序中多次使用到，所以应该将它们定义为命名常量。使用命名常量可以简化后面程序的修改工作。

```
const int Halfdollar = 50;
const int Quarter  = 25;
const int Dime = 10;
const int Nickel = 5;
```

主要算法

1. 提示用户输入数据
2. 读入数据
3. 在计算机屏幕上回显输入的美分数量
4. 计算并输出50美分的数量
5. 计算余下的美分数
6. 计算并输出25美分的数量
7. 计算余下的美分数
8. 计算并输出10美分的数量
9. 计算余下的美分数
10. 计算并输出5美分的数量
11. 计算余下的美分数
12. 输出余下的美分数

完整的程序代码清单

```
//*******************************************************
// 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;
}
```

程序运行结果 在本示例中，用户输入的数据加有阴影。

```
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  1
The number of nickels to be returned are 1
The number of pennies to be returned are 3
```

注意：如果要编写零钱换整的标准C++程序，只需要将本例中的下列语句：

```
#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. 每个C++程序中都必须含有一个main函数。
3. 在C++中，标识符是常量、变量和函数等的名字。
4. C++标识符由字母、数字和下划线组成，首字符必须是字母或下划线。

5. 保留字不能作为标识符在程序中使用。
6. C++ 中的所有保留字都由小写字母组成（参阅附录 A）。
7. 最常用的字符集有ASCII码字符集，它由128个字符组成；EBCDIC码字符集由256个字符组成。
8. 字符编码值是字符在字符集中预先定义好的顺序。
9. C++ 中的算术运算符有：加法运算符（+）、减法运算符（-）、乘法运算符（*）、除法运算符（/）和求余运算符（%）。
10. 求余运算符 %，只能用来计算整型操作数。
11. 按照优先级规则和结合律规则来计算算术表达式。
12. 在整型算术表达式（整型表达式）中，所有的操作数都是整数；在浮点型算术表达式中，所有的操作数都是小数。
13. 混合类型表达式由整数和小数两种操作数组成。
14. 在计算表达式中的运算符时，只要该运算符具有混合类型操作数，那么要先将整型操作数通过添加值为 0 的小数部分，转换成浮点型操作数。
15. 可以使用强制类型转换符将数据值显性地转换成另一种类型。
16. 常量的值在程序运行过程中不能改变。
17. 命名常量使用关键字 const 来定义。
18. 命名常量在定义时初始化。
19. 所有的变量都必须经过定义后才能使用。
20. C++ 并不能自动初始化变量。
21. 所有变量都有名字、值、数据类型和一定大小的存储空间。
22. 当变量赋予新值后，原来的数据将被破坏掉。
23. 只有赋值语句或是输入语句才可以改变变量中的值。
24. 输出或使用变量中的值并不能改变或破坏变量中的值。
25. 在 C++ 中，“>>”被称为析取运算符。
26. 将数据从标准输入设备中输入到计算机是通过 cin 和析取运算符“>>”来实现的。
27. 将多个数据输入到程序中时，多个数据项之间通常用空格符、换行符或是水平制表符分隔。
28. 在 C++ 中，“<<”被称为插入运算符。
29. 将数据从计算机输出到标准输出设备中是通过 cout 和插入运算符“<<”完成的。
30. 如果要在 ANSI/ISO 标准 C++ 中使用 cin 和 cout，程序中必须包含头文件 iostream，并且要包括语句 using namespace std; 或是带有前缀的 std::cin 和 std::cout。
31. 控制符 endl 的作用是将光标移动到下一行的开头处。
32. 字符“\”被称为转义字符。
33. 字符序列 \n 被称为换行转义字符。
34. 所有的预处理指令都必须以 # 开头。
35. 预处理指令由预处理程序在程序编译前处理。
36. 预处理指令 #include <iostream>告诉预处理程序在程序中包含头文件 iostream。
37. 所有的 C++ 语句必须用分号结束。在 C++ 中，分号被称为语句结束符。
38. C++ 系统由三部分组成：环境、语言和标准库。
39. 标准库并不是 C++ 语言本身的一部分。标准库中含有执行诸如数学运算等操作的函数。
40. C++ 源程序文件通常使用.cpp 作为扩展名。
41. 单行注释以一对斜线 // 开头，并可以出现在代码行中的任何位置上。
42. 多行注释要用 /* 和 */ 括起来。

43. 编译器将忽略掉注释。
44. 提示行是用来提示用户怎样做的可执行语句。
45. 复合算术赋值语句的形式是 op =，其中 op 是任意的算术运算符。
46. 对于算术运算符，复合赋值语句的形式是“variable op = expression;”。其等价的简单赋值运算语句的形式是“variable = variable op (expression);”，其中 op 是任意的算术运算符。

2.14 练习

1. 判断下面说法的正误。
   a. 标识符可以是数字和字母的任意组合。
   b. 在 C++ 中，保留字和预定义标识符之间没有区别。
   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; 并且 b = 3; 则在语句 a = b; 执行之后，b 中的值仍是 3。
   f. 在语句“cin >> y;”中，变量 y 只可以是 int 类型或是 double 类型。
   g. 在 cout 语句中，换行字符可以出现在字符串中。
   h. 下面是合法的 C++ 程序：

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

   i. 在混合表达式中，所有的操作数都要转化成浮点数。
   j. 假设 x = 5。在语句 y = x++; 执行后，y 等于 5，而 x 等于 6。
   k. 假设 a = 5。在语句 ++a; 执行后，a 的值仍为 5，因为该表达式的值并没有存储到别的变量中。
2. 下面哪些标识符是合法的？
   a. `RS6S6`
   b. `MIX-UP`
   c. `STOP!`
   d. `exam1`
   e. `September1Lecture`
   f. `2May`
   g. `Mike's`
   h. `First Exam`
   i. `J`
   j. `Three`
3. 下面哪些是 C++ 中的保留字？
   a. `int`
   b. `long`
   c. `Char`
   d. `CHAR`
   e. `Float`
   f. `Double`

4. 选择正确的答案。

a. 15/2 的值是：

(i) 7　(ii) 7.5　(iii) 7 1/2　(iv) 0.75　(v) 以上都不是

b. 18/3 的值是：

(i) 6　(ii) 0.167　(iii) 6.0　(iv) 以上都不是

c. 22%7 的值是：

(i) 3　(ii) 1　(iii) 3.142　(iv) 22/7

d. 5%7 的值是：

(i) 0　(ii) 2　(iii) 5　(iv) 未定义

e. 17.0/4 的值是：

(i) 4　(ii) 4.25　(iii) 4 1/4　(iv) 未定义

f. 5-3.0+2 的值是：

(i) 0　(ii) 0.0　(iii) 4　(iv) 4.0

g. 7-5*2+1 的值是：

(i) -2　(ii) 5　(iii) 6　(iv) 以上都不是

h. 15.0/3.0+2.0 的值是：

(i) 3　(ii) 3.0　(iii) 5　(iv) 以上都不是

5. 假定 x = 5，y = 6，z = 4 和 w = 3.5。如果可以，计算下列各表达式的值；如果不可以，说明理由。

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;
```

下面各赋值语句中哪些是合法的？如果不合法，请说明原因。假定下面出现的其他变量都已定义。

a. `n = m = 5;`

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

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

d. `m + n = l;`

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

f. `l + 1 = n;`

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

h. `m = n % l;`

i. `n = x % 5;`

j. `x = x + 5;`

k. `n = 3 + 4.6`

7. 使用代码走查的方法追踪变量 e 中的值。假定所有的变量都经过定义。

```
a = 3;
b = 4;
c = (a % b)*6;
d = c / b;
e = (a+b+c+d)/4;
```

8. 下面哪些变量的定义是正确的？如果变量定义不正确请说明原因，并给出正确的定义。

```
n = 12;                //Line1
char letter = ;        //Line2
int one = 5, two;      //Line3
double x, y, z;        //Line4
```

9. 下面的 C++ 赋值语句中哪些是合法的？假定 I，x 和 percent 都是 double 型变量。

a. `I = I + 5;`
b. `x + 2 = x;`
c. `x = 2.5 * x;`
d. `percent = 10%.`

10. 按照下面的要求写出 C++ 语句。

a. 定义 int 型变量 x 和 y。
b. 将 int 型变量 x 初始化为 10，将 char 型变量 ch 初始化为 'B'。
c. 将 int 型变量 x 中的值加 5。
d. 将数值 25.3 赋给 double 型变量 z。
e. 将 int 型变量 y 中的值赋给 int 型变量 z。
f. 互换 int 型变量 x 和 y 的值（如果需要，可以定义另外的变量）。
g. 输出 int 型变量 x 和表达式 2 * x + 5 – y 中的值，其中 x 和 y 都是 double 型变量。
h. 定义 char 型变量 grade，并赋值为 'A'。
i. 定义一些 int 型变量，存储 4 个整数值。
j. 将 double 型变量 z 中的值转变成最近的整数，并将其存储到 int 型变量 x 中。

11. 按照下面的要求写出 C++ 表达式。

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

12. 假定 x，y，z 和 w 都是 int 型变量。在下面所有代码执行过后，它们中的值各是多少？

```
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 都是 int 型变量，w 和 t 都是 double 型变量。在下面所有代码执行过后，它们中的值各是多少？

```
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都是int型变量，并且x = 2，y = 5，z = 6。下面各条语句的输出结果是什么？
    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. 下面各条语句的输出结果是什么？假定a和b都是int型变量，c是double型变量，并且a = 13，b = 5，c = 17.5。

	输出结果
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. 怎样输出回车（Carriage Return）？
17. 下面的C++语句哪些是正确的？
    a. `cout<<"Hello There!"<<endl;`
    b.
    ```
    cout<<"Hello";
          <<" There!"<<endl;
    ```
    c.
    ```
    cout<<"Hello"
          <<" There!"<<endl;
    ```
    d. `cout<<'Hello There!'<<endl;`

18. 下面两个程序中含有语法错误，请纠正它们。在处理每一行语句时，假定前面的语句都没有错误。

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，写出下面语句的等价复合赋值语句。

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. 将下面复合赋值语句改写成等价的简单赋值语句。

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都是int型变量，并且a = 5，b = 6。在每一条语句执行过后，每个变量中的值分别是多少？如果某个变量在语句中没有定义，请标记为UND（未定义）。

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

22. 假定a，b和sum都是int型变量，c是double型变量。在下面每一条语句执行过后，每个变量中的值是多少？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. 下面程序的输出结果是什么？假定输入的数据是：

```
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. 下面程序需要输入的数据类型是什么？并指出应该按何种顺序输入。

```
#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. 编写可以产生下面输出的程序。

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

2. 编写可以产生下面输出的程序。

```
CCCCCCCCC        ++               ++
CC               ++               ++
CC         ++++++++++++++   +++++++++++++++
CC         ++++++++++++++   +++++++++++++++
CC               ++               ++
CCCCCCCCC        ++               ++
```

3. 编写一个程序，该程序提示用户输入一个小数，然后将小数转换成最近的整数，并输出。
4. 编写一个程序，该程序提示用户输入矩形的长度和宽度，输出矩形的面积和周长。
5. 编写一个程序，该程序提示用户输入5个考试分数，输出平均考试分数。
6. 编写可以产生下面输出的程序。

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

7. 编写可以产生下面输出的程序。

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

8. 编写一个程序，该程序提示用户输入5个小数，然后将这5个小数加起来，并转换成最近的整数输出。
9. 编写一个程序完成下面的要求。
   a. 提示用户输入5个小数。
   b. 输出这5个小数。
   c. 将每一个小数分别转换成最近的整数。
   d. 将这5个整数加起来。
   e. 输出这5个整数的和、平均数。
10. 编写一个程序，该程序提示用户输入以厘米为单位的长度，将其换算成最近的以英寸为单位的整数长度，最后输出按照码（yard）、英尺（foot）、英寸（inch）顺序输出该长度[①]。例如，123英寸应该换算成：

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

而不是：

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

也不是：

```
10 feet(foot), 3 inch(es)
```

更不是：

```
10.25 feet(foot)
```

① 1码＝3英尺＝0.914 4 m；1英尺＝12英寸＝0.304 8 m；1英寸＝2.54 cm——编者注。

第3章　输入/输出

本章要点：

- 理解什么是流，以及如何使用输入、输出流
- 理解怎样从标准输入设备中读入数据
- 理解怎样在程序中使用预定义函数
- 理解怎样使用 get，ignore，fill，putback 和 peek 流输入函数
- 了解怎样处理输入失败
- 理解怎样将数据输出到标准输出设备中
- 理解怎样在程序中使用控制符来格式化输出
- 了解怎样输出和输入 string 类型数据
- 理解怎样输入和输出文件

在第2章中已经简单介绍过C++ 中的输入/输出（I/O）的指令，这些指令负责将数据读入程序，并将计算结果输出到计算机屏幕上。cin和析取运算符>>负责将数据从键盘上读入到计算机中；cout和插入运算符<<负责将数据输出到屏幕上。因为I/O操作是所有程序的基础，所以本章要进一步对C++ 的I/O操作做细致的讨论。首先介绍的是向标准输出设备中输出数据和从标准输入设备中读入数据的函数。然后再介绍怎样使用控制符来格式化输出数据。读者还可以从本章中了解到C++ 中与标准输入/输出设备进行I/O操作时的限制，以及怎样使用其他设备进行I/O操作。

3.1　I/O流和标准I/O设备

程序可以执行三种基本的操作：将数据输入到程序中、处理数据和输出结果。在第2章中已经介绍过如何使用算术运算符处理数字类型的数据。在后面章节中，将会介绍如何处理非数字类型的数据。因为I/O处理程序编写起来十分复杂，所以C++ 提供了许多实用的I/O操作函数来处理这些操作。在本章中，读者将了解到各种不同的I/O操作方法，使用这些方法可以大大提高程序的灵活性。

在C++ 中，I/O数据是一些从源到目标的字节序列，被称为字节流（Stream of Bytes）。除了图像数据、数字语音数据外，这些字节通常代表的是字符。也就是说，流（Stream）是从源设备到目标设备的字符序列。流分两种类型：输入流和输出流。

输入流（Input Stream）　从输入设备到计算机的字符序列。

输出流（Output Stream）　从计算机到输出设备的字符序列。

如前所述，标准输入设备通常是键盘，而标准输出设备通常是屏幕。为了将数据从键盘读入或是将结果输出到屏幕上，程序中必须要包含头文件iostream。这个头文件中包含了两个数据类型的定义：istream（输入流）和ostream（输出流）。这个头文件中同时还包含了两个变量定义：cin（读做see-in），表示普通输入（Common Input）；cout（读做see-out），表示普通输出（Common Output）。这两个变量的定义与下面的C++ 语句相似：

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

为了可以使用cin和cout，C++程序中必须包含下面的预处理命令：

```
#include <iostream>
```

注意： 第2章中已经说明，使用语句using namespace std;可以简化cin和cout的使用。如果不使用语句using namespace std; 你必须采用std::cin和std::cout的方式来使用这两个标识符。在第8章中，将详细了解到语句“using namespace std;”的含义。

istream类型的变量被称为输入流变量（Input Stream Variables），ostream类型的变量被称为输出流变量（Output Stream Variables）。一个流变量不是输入流变量就是输出流变量。

因为在C++中已经定义了变量cin和cout，并且它们具有特定的含义，所以为了避免混淆，请不要在程序中重新定义它们。

在函数中使用变量cin可以将数据从标准输入设备中读入到计算机中。也可以只使用析取运算符>>将从标准输入设备中读入数据。下一节将详细地解释运算符>>是怎样工作的。在接下来的章节中，还可以了解到怎样使用函数get，ignore，peek和putback以特定方式输入数据。

3.1.1　cin和析取操作符>>

在第2章中，已经接触到了怎样使用cin和析取运算符>>从标准输入设备中读入数据。考虑下面的C++语句：

```
cin >> payRate;
```

当这条语句执行时，计算机将从键盘上输入的数字存储到变量payRate中。因此，如果变量payRate是double类型，用户输入数字15.50，这条语句执行后变量payRate中的值就变成15.50。

析取运算符>>是双目运算符，所以有两个操作数。左边的操作数必须是输入流类型的变量，如变量cin。因为输入语句的目的是将数据读入并且存储到内存的存储单元里，而且变量实际上指向的也正是存储单元，所以右边的操作数应该是变量。

注意： 析取运算符>>只是为将数据输入到简单数据类型的变量而设计的。因此，析取运算符>>右边的操作数应该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，以便可以将数据存储到其他类型的变量中。这种机制将在第15章中介绍。

使用cin和析取运算符的输入语句语法如下所示：

```
cin >> variable1 >> variable2...;
```

正如在该语法中见到的一样，一条cin语句可以通过使用多个运算符>>一次读入多个数据。每遇到一个>>，输入语句就在输入流中抽取一个数据项。例如，可以使用如下的一条cin语句一次读入payRate和hoursWorked两个变量。

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

这条cin语句和下面两条cin语句的作用是完全相同的。可以根据个人习惯和风格来选用某种cin语句方式：

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

析取运算符是怎样工作的呢？当扫描下一个输入的数据项时，>>略过所有的空白（Whitespace）字符。空白字符包括空格符和其他不可打印字符，如水平制表符和换行符等。因此，当输入数据的数据项以一个或多个空格符或是换行符分隔时，析取运算符>>只是从输入流中查找下一个输入数据项。例如，假设payRate和hoursWorked是两个double类型的变量。考虑下面的语句：

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

无论输入的数据是：

```
15.50 48.30
```

还是：

```
15.50	48.30
```

或

```
15.50
48.30
```

该语句都会将15.50存储到变量payRate中，将48.30存储到变量hoursWorked中。注意第一种输入方式的分隔符是空格符，第二种输入方式的分隔符是水平制表符，最后一种输入方式的分隔符是换行符。

现在假设输入的数据是2。析取运算符>>是怎样分辨出它到底是字符2，还是数字2呢？输入数据的类型是由析取运算符>>右边操作数的数据类型来决定的。如果析取运算符>>右边操作数的数据类型是char，输入的数据2将被视为字符；如果析取运算符>>右边操作数的数据类型是int或者double，输入的数据2将被视为数字。

假定下一个输入的数据是25，并且输入语句如下所示：

```
cin >> a;
```

这里a是简单数据类型的变量。如果a是char类型的变量，那么只有字符2存储到a中；如果a是int类型的变量，那么整数25将存储到a中；如果a是double类型的变量，那么要将其转化成小数25.0再存储到a中。表3.1总结了上述讨论，输入到不同的简单数据类型变量a中的数据都是合法的。

表3.1　简单数据类型变量的合法输入

a的数据类型	a的合法数据输入
char	除空格符外的一个可打印字符
int	一个整数，前面可以有（+或-）符号
double	一个小数，前面可以有（+或-）符号。如果输入的是整数，通过在整数后面加上.0的小数部分先将其转化为小数

当将数据读入char类型变量时，析取运算符>>首先略掉数据前面的所有空白符，找到并存储下一个字符，并在读入了一个字符后就立即停止。当将数据读入int或是double类型变量时，析取运算符>>首先略掉数据前面的所有空白符，如果遇到正、负号就读入它，然后读入数字，包括浮点型数的小数点，并在遇到空白符或是其他非数字字符时停止。

例3.1　假定有如下的变量定义：

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

下面的语句将说明析取运算符是怎样工作的：

	语句	输入	内存中的值
1	cin>>ch;	A	ch='A'
2	cin>>ch;	AB	ch='A','B'被保留在输入流里等待被读入
3	cin>>a;	48	a=48
4	cin>>a;	46.35	a=46，.35被保留在输入流里等待被读入
5	cin>>z;	74.35	z=74.35

6	cin>>z;	39	z=39.0
7	cin>>z>>a;	65.78 38	z=65.78, a=38
8	cin>>a>>b;	4 60	a=4, b=60
9	cin>>a>>ch>>z;	57 A 26.9	a=57, ch='A', z=26.9
10	cin>>a>>ch>>z;	57 A 26.9	a=57, ch='A', z=26.9
11	cin>>a>>ch>>z;	57 A 26.9	a=57, ch='A', z=26.9
12	cin>>a>>ch>>z;	57A26.9	a=57, ch='A', z=26.9
13	cin>>z>>ch>>a;	36.78B34	z=36.78, ch='B', a=34
14	cin>>z>>ch>>a;	36.78 B34	z=36.78, ch='B', a=34
15	cin>>a>>b>>z;	11 34	a=11, b=34，计算机等待下一个输入的数
16	cin>>a>>z;	46 32.4 68	a=46, z=32.4,68被保留在输入流里等待被读入
17	cin>>ch>>a;	256	ch='2', a=56
18	cin>>a>>ch;	256	a=256，计算机等待输入变量ch的值
19	cin>>ch1>>ch2;	A B	ch1='A', ch2='B'

在语句1中，析取运算符>>在输入流中读取字符'A'，并且存储到变量ch里。在语句2中，析取运算符>>在输入流中读取字符'A'，将其存储到变量ch里，字符'B'被保留在输入流里等待下一次读取。同样，在语句16中，数值68被保留在输入流里等待下一次读取。在语句15中，11存储到a中，34存储到b中，而这时输入流中已经没有足够的数据以存储到该语句要求的那么多变量里。在这种情况下，计算机将等待（不断等待）下一个输入数据。只有在等待的所有数据全部被输入后，计算机才能执行下一条语句。

在语句4中，析取运算符>>在输入流中读取数据46，并且存储到变量a里。注意变量a是一个整型变量并且46后面的字符是小数点（.非数字），所以只有46从输入流中被读取出来。因此，遇到小数点（.）后，该条输入语句停止，.35被保留在输入流里等待下一次读取。在语句5中，析取运算符>>在输入流中读取74.35并将其存储到变量z中。

在语句6中，z是double类型变量，而输入的39是一个整型数据。因此，39首先被转换成小数39.0，然后再被存储到变量z中。

在语句7中，析取运算符>>右边第一个变量是double类型，第二个变量是int类型。因此，语句7首先从输入流中读取数值65.78并将其存储到变量z中。然后析取运算符略掉65.78后面的所有空格，将数值38读取并存储到变量a中。语句8的执行过程与语句7相似。

注意从语句9到语句12，输入语句是相同的，只是输入的数据是不同的。在语句9中，所有数据都在同一行输入中，并以空格符作为分隔符。在语句10中，输入数据分成两行。前两个数据之间以空格符作为分隔符，而第三个数据则在下一行中输入。在语句11中，三个数据之间都是以换行符作为分隔符。在语句12中，所有的数据都在同一行中输入，并且数据项之间没有任何分隔符。注意第二个输入数据是非数字字符。这些语句的执行过程如下。

在语句9中，析取运算符>>首先在输入流中读取数字57，并且存储到变量a里。然后析取运算符略掉57后面的空格符，读取字符'A'，并将其存储到变量ch里。接下来，析取运算符略掉'A'后面的空格符，从输入流中读取数字26.9，并且存储到变量z里。

在语句10中，析取运算符>>首先在输入流中读取数字57，并且存储到变量a里。然后析取运算符略掉57后面的两个空格符，读取字符'A'，并将其存储到变量ch里。接下来，析取运算符略掉'A'后面的换行符'\n'，从输入流中读取数字26.9，并且存储到变量z里。

在语句11中，析取运算符>>首先在输入流中读取数字57，并且存储到变量a里。然后析取运算符略掉57后面的换行符'\n'，读取字符'A'，并将其存储到变量ch里。接下来，析取运算符略掉'A'后面的换行符'\n'，从输入流中读取数字26.9，并且存储到变量z里。

语句9、语句10和语句11说明，无论输入的各数据项之间的分隔符是空格符还是换行符，析取运算符总能正确地找到下一个输入的数据项。

在语句12中，析取运算符>>首先在输入流中读取数字57，并且存储到变量a里。然后析取运算符输入流中读取字符'A'，并将其存储到变量ch里。最后，读取数字26.9，并且存储到变量z里。

在语句13中，析取运算符>>右边第一个操作数是double类型变量z。析取运算符从输入流中读取36.78，并将36.78存储到变量z里。接下来，读取'B'并存储到变量ch里。最后，读取34并存储到变量a里。语句14的执行过程与语句13差不多。

在语句17中，析取运算符>>右边第一个操作数是char类型变量。所以，析取运算符从输入流中读取出第一个非空白字符'2'。字符'2'被存储到变量ch里。下一个析取运算符>>右边的操作数是int类型变量，所以下一个输入数值56被读取并存储到变量a里。

在语句18中，析取运算符>>右边第一个操作数是int类型变量。所以，析取运算符从输入流中读出第一个数据项256，并将其存储到变量a里。此时，计算机等待下一个输入项以存储到变量ch里。

在语句19中，'A'被存储到变量ch1里。然后，析取运算符略掉空格符，将'B'存储到变量ch2里。

注意： 前面已经提到过，在程序执行过程中输入如字母等字符时，输入时不需要使用单引号将字符括住。

如果输入流中的数据项个数多于程序所要求的数据项个数，计算机将会怎样处理呢？在程序终止后，输入流中所有剩余的数据项将被丢弃掉。输入数据项的类型应该和输入语句中变量的数据类型相一致。前面已经提到，输入一个double类型数据时，输入的数字不必非要有小数部分。如果输入的是没有小数部分的整数时，该整数将被自动转化成小数。但是，计算机不允许出现除此之外的任何类型不匹配的情况发生。例如，如果试图将char类型数据输入到int或是double类型的变量中，将会导致严重错误，这种错误被称为输入失败（Input Failure）。输入失败将在本章的后面部分中介绍。

当试图将非数字字符读入到int类型变量中时，会出现什么情况呢？本章后边的例3.5，会说明这个问题。

当析取运算符从输入流中扫描下一个数据项时，将略掉所有的空格符和换行符等空白字符。但是在很多时候，这些空白字符本身也需要作为输入内容的一部分读到程序中。例如，当处理以换行符分隔的多行文本时，程序必须要知道文本中换行符的位置。如果不知道换行符的位置，程序也就无法区分不同的文本行。接下来的几个章节中，将要介绍怎样使用get，ignore，putback和peek等函数将数据读入到计算机里。这些函数处理的都是istream类型数据，所以称为istream成员函数（Istream Member Function）。诸如get等函数通常也被称为流函数（Stream Member Function或者Stream Function）。

本章在介绍怎样使用get，ignore，putback和peek等输入函数和其他一些I/O函数前，先要介绍什么是函数以及函数是怎样工作的？在本书第6章和第7章中，读者将详细了解函数，并可以编写自己的函数。

3.2 在程序中使用预定义函数

第2章中已经提到，函数又称为子程序，是一些指令的集合。当程序执行时，可以完成特定的任务。第2章中接触到的main函数，是在程序执行时自动运行的。其他所有的函数必须被激活，即调用，才能运行。C++自带了大量功能强大的函数，称为预定义函数（Predefined Function）。在接下来的几节中，将了解到怎样使用流函数来实现某种具体的I/O操作。本节并不介绍怎样编写自己的函数，而只介绍怎样使用作为C++语言一部分的预定义函数。

第2章中介绍过，预定义函数被组织在称为头文件的库集中。一个头文件中可能包含许多个函数。因此，为了使用某个预定义函数，必须要知道函数名和关于该函数使用中的一些规定。

函数pow十分有用，称为乘方函数。可以在程序中使用pow函数计算型如x^y的表达式。也就是，pow(x, y) = x^y。例如，pow(2, 3) = $2^3 = 8$，pow(4, 0.5) = $4^{0.5} = \sqrt{4} = 2$。函数pow中的数据x和y被称为函数pow的参数（Argument或Parameter）。例如在pow(2, 3)中，参数是2和3。

像pow(2, 3)这种含有函数的表达式称为函数调用（Function Call）。通过对函数pow的调用，使函数pow的代码得以执行，如在例中计算2^3。头文件cmath中含有乘方函数pow的完整定义。

为了可以使用预定义函数，必须要知道包含该预定义函数头文件名，并且将此头文件包含到程序中。此外，必须要知道函数名称、参数个数、每个参数的类型，以及函数的作用。例如，如果想使用函数pow，必须要在程序中包含头文件cmath。函数pow有两个参数，并且两个参数都是数值类型。该函数的功能是计算以第一个参数为底，第二个参数为指数的表达式的值。

因为I/O是所有程序设计语言的基本组成部分，而亲自动手编写I/O程序对每个程序员来说又不是一件轻而易举的事情，所以所有程序设计语言都提供了一系列支持I/O操作的实用函数。在本章的剩余部分里，将了解到怎样在程序中使用这些函数。作为一个程序员，必须要熟练掌握这些函数，以便可以在程序中自由、灵活地使用。在此，介绍的第一个函数是get。

3.2.1 cin和get函数

前面已经说明，析取运算符在扫描下一个输入数据时，将略掉所有的前置空白符。考虑下面的变量定义：

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

和输入数据：

```
A 25
```

考虑下面的语句：

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

计算机在执行这条语句时，将字母'A'存储到变量ch1中，然后略掉空格符，将字符'2'存储到变量ch2中，最后将数字5存储到变量num中。但是，如果本来的用意是将字母'A'存储到变量ch1中，将空格符存储到变量ch2中，并将数字25存储到变量num中，那该怎么办？很明显，不能使用析取运算符>>来达到这样的目的。

前面已经提到，程序有时候需要处理输入数据中的全部内容，其中包括空格符或是换行符等空白符。例如，用来处理带换行的多行文本的程序。因为析取运算符将略掉所有的空白符，除非程序中包含可以捕获这些换行符的代码，计算机将不会知道哪里是一行的开头，哪里是一行的结尾。

通过cin变量，可以使用get函数，该函数的作用是读入字符数据。get函数可以将输入流中的所有字符，包括空白符，读入到参数列表中指定变量对应的内存存储空间中。get函数的使用方式灵活多样，让我们先来讨论下面这种方式：

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

在cin.get语句中，varChar是char类型变量。函数名后面括号中的varChar，称为该函数的参数。这条语句的作用是将下一个输入的字符存储到变量varChar中。

下面再次考虑输入数据：

```
A 25
```

为了将字母 'A' 存储到变量 ch1 中，将空格符存储到变量 ch2 中，并将数字 25 存储到变量 num 中，可以使用如下的 get 函数：

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

由于这种形式的 get 函数只有一个参数并且一次只能读入一个字符，所以必须使用两个 get 函数来达到从输入流中相继读入两个字符的目的。需要注意的是，不能使用 get 函数读入变量 num 所需要的数据，因为 num 是 int 类型的变量。这种形式的 get 函数只能读入 char 类型的数据。

上面的一系列 cin.get 语句等价于下面的语句：

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

注意：get 函数还有其他的使用形式，在第 9 章中将会见到它的另一个使用形式。在接下来的几章中，只需要了解这里介绍的用法。

3.2.2 cin 和 ignore 函数

如果程序只使用输入流中的部分数据（比方说，一行以内的所有字符），可以使用 ignore 流函数忽略掉不需要的那部分数据。使用 ignore 函数的语法是：

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

这里，intExp 是结果为整型数据的表达式，chExp 是结果为字符型数据的表达式。实际上，表达式 intExp 的值用来指定被忽略掉字符的最大个数。

假设表达式 intExp 的值是 100。该语句的意思是要忽略掉接下来的 100 个字符，或者是首次遇到表达式 chExp 中指定字符以前的所有字符。下面来看一个具体的例子：

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

当执行这条语句时，程序将忽略掉下面的 100 个字符或是只忽略掉下一个换行符前的所有字符。如果输入流中下面 120 个字符中都没有换行符，那么程序将忽略掉下面 100 个字符，并且在输入流中标记下一个被读入的是第 101 个字符。但是如果输入流中第 75 个字符是换行符，那么程序将只忽略掉下面 75 个字符，并且在输入流中标记下一个被读入的是第 76 个字符。同样，语句：

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

的作用是忽略掉下面 100 个字符或是只忽略掉第一个 'A' 出现前的所有字符。

例 3.2 *考虑下面的变量定义：*

```
int a, b;
```

和输入数据：

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

现在考虑下面的语句：

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

第一条语句，cin >> a;，将 25 存储到变量 a 中。第二条语句，cin.ignore(100, '\n');，将忽略掉第一行中所有剩余的字符。第三条语句，cin >> b;，将 12（从下一行中）存储到变量 b 中。

例 3.3 考虑下面的变量定义：

```
char ch1, ch2;
```

和输入数据：

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

现在考虑下面的语句：

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

第一条语句，cin >> ch1;，将字母 'H' 存储到变量 ch1 中。第二条语句，cin.ignore(100, '.');将忽略掉字符'.'（句号）出现前的所有字符。第三条语句，cin >> ch2;将字母 'M' 存储到变量 ch2 中（注意析取运算符>>将略掉字符前面的所有空白符，所以本例中析取运算符>>将略掉'.'（句号）后面的空格符，并将 'M' 存储到 ch2 中）。

3.2.3　putback 函数和 peek 函数

假定程序需要处理的数据中既有数字，又有字符。而且，这些数据中的数字要作为数字来处理。在前面见到的例子中，程序并不能判明输入的字符究竟是数字还是真正的字符。这样，程序需要在输入的字符中逐一判断每个字符到底是不是数字字符。如果是数字字符，还需要完全读入该数字中的全部字符，并将这些字符转换成为相应的数字。这种程序代码冗长复杂。令人高兴的是，C++ 提供了两个极为有用的函数来处理这种情况。

putback 流函数可以将由 get 函数在输入流中读取的最后一个字符退回到输入流中。peek 流函数检查输入流，并且告诉程序输入流中下一个字符是什么，但并不实际读取这个字符。通过这些函数，程序可以在判明下一个输入的字符是数字之后，将其直接作为数字读人。这样就再也不用将数字先作为字符读人，然后再将其转换成数字了。

使用 putback 函数的语法是：

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

这里 istreamVar 是一个输入流类型的变量，如 cin，而 ch 是一个 char 类型的变量。

peek 函数返回输入流中的下一个字符，但并不真正读取这个字符。也就是说，peek 函数在输入流中查找下一个输入的字符，并在找到该字符以后，将其存储到目标内存单元中，但并不把它从输入流中移走。也就是说，在使用了函数 peek 以后，即便是已经读入了该字符，输入流中下一个等待读人的字符仍然保持不变。

使用 peek 函数的语法是：

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

这里 istreamVar 是一个输入流类型的变量，如 cin，而 ch 是一个 char 类型的变量。

注意 peek 函数的使用。首先，peek 函数应该用在赋值表达式中，而不能像 get，ignore 和 putback 等函数那样单独使用。第二，peek 函数的参数列表为空。除非你对函数已经相当的熟悉并学会了编写自己的函数，否则要格外注意怎样使用这些预定义函数。

下面的例子说明该如何使用 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;
}
```

程序运行结果 在本程序运行中，输入的数据加有阴影。

```
Line1: Enter a string: abcd
Line4: After first cin.get(ch); ch= a
Line6: After second cin.get(ch); ch= b
Line9: After putback and then cin.get(ch); ch= b
Line11: After cin.peek(); ch= c
Line13: After cin.get(ch); ch= c
```

用户输入abcd，现在让我们看看在本段程序中get，putback和peek函数的执行结果。第1行语句的作用是提示用户输入一个字符串。第2行语句中的cin.get(ch);在输入流中读取一个字符，并把它存储到变量ch中。所以，在第2行语句执行完后，ch中的值是'a'。

第4行中的语句cout输出变量ch中的值。第5行中的语句cin.get(ch);在输入流中读取下一个字符'b'，并将其存储到变量ch中。此时，ch中的值是'b'。

第6行中的语句cout输出变量ch中的值。第7行中的语句cin.putback(ch);将由get函数读取的字符'b'退回到输入流中。因此，输入流中下一个被读取的字符仍然是'b'。

第8行中的语句cin.get(ch);从输入流中读取下一个字符'b'，并将其存储到变量ch中。此时，ch中的值是'b'。第9行中的语句cout输出变量ch中的值'b'。

第10行中的语句ch = cin.peek( );在输入流中查看下一个输入字符，即'c'，并将其存储到变量ch中。第11行中的语句cout输出变量ch中的值。第12行中的语句cin.get(ch)；在输入流中读取下一个字符，并将其存储到变量ch中。第13行中的语句cout输出变量ch中的值，该值仍为'c'。

注意第10行中的语句 ch = cin.peek( );并不从输入流中取走字符'c'，而只是在输入流中查看它。第11行和第13行中的输出语句说明了这种机制。

3.2.4 I/O流变量和I/O函数间的点符号(.)

在前面几节中，已经介绍了怎样通过对数据流的操纵将数据读到程序中，并介绍了get，ignore，peek和putback函数的使用方法。必须要按照上面所示的方法使用这些函数。例如，应该按照下面语句所示的方法来使用get函数：

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

如果漏掉点符号——即变量cin和函数名get之间的英文中的句号，将会导致语法错误。例如，在语句：

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

中，cin和get是用点符号分隔开的两个不同的标识符。在语句：

```
cinget(ch);
```

中，cinget变成一个单独的标识符。如果程序中出现语句cinget(ch);，编译器将试图解析该函数，并会报出错误信息。同样，在语句“cin.getch;”中，由于漏写了括号，也会导致语法错误。还要注意的是，输入函数必须要同输入流变量一同使用。如果试图单独使用任何输入函数，即没有使用输入流变量，编译器将会报出语法错误，如“未定义变量”。例如，语句get(ch);将会导致语法错误。

正如所看到的，istream类型变量有许多函数，每个函数的功能也不相同。get，ignore等函数是istream类型变量的成员（Member）函数。用来分隔流变量名与成员变量名或是成员函数名的点，称为点符号（Dot Notation）。实际上，C++中的符号点被称为成员存取运算符（Member Access Operator）。

注意： 在C++中，数据类型istream和ostream有另外一个名字——类（Class）。变量cin和cout也有另外一个名字——对象（Object）。因此，cin被称为输入流对象，而cout被称为输出流对象。实际上，流变量也称为流对象。这些概念将在第12章中介绍。

3.3 输入失败

程序执行过程中不可避免地会出现这样或那样的问题。一个语法正确的程序也可能产生不正确的结果。例如，假设下面的公式用来计算兼职员工的月工资单：

```
wage = payRate * hoursWorked;
```

如果在程序中不小心将*写成了+，则计算出的工资数额将是不正确的，即使公式在语法上是正确的。

当程序试图读入一个非法输入数据时会怎么样呢？例如，如果程序试图将一个字母读入到一个int型变量中，结果将会怎样呢？如果输入数据的类型与对应变量的类型不匹配，程序会报出错误。例如，如果程序试图将一个字母读入到一个int型或是double型变量中，将会导致输入失败（Input Failure）。考虑下面的语句：

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

如果输入的数据是：

```
W 54
```

则语句：

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

将会导致输入失败，因为程序试图将字符'W'输入到int类型变量a中。如果输入的数据是：

```
35 67.93 48 78
```

输入语句是：

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

那么将会把35存储到a中，把67.93输入到x中，把48输入到b中。

现在考虑下面的输入语句，输入数据同上：

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

该语句将35存储到a中，将67存储到b中。输入在遇到小数点（.）后停止。因为下一个变量c是int类型的变量，计算机试图将小数点（.）读入到变量c中，这会导致错误。这时输入流处于一种错误状态（Fail State）。

当输入流处于错误状态时会怎么样呢？当某个输入流处于错误状态时，接下来使用该输入流的所有I/O语句将被忽略掉。遗憾的是，程序将继续使用变量中的现有数值进行下一步计算，而不会报出输入流错误。这将导致更加严重的错误。例3.5中的程序说明了输入失败的后果。这个程序在你的计算机系统中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
    return0;
}
```

程序运行结果　在这些程序运行中，用户输入的数据加有阴影。

程序运行结果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行、第2行、第3行、第4行语句定义变量a，b，c和d，并分别将其初始化为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个整数读入到变量a，b，c和d中。在本程序运行中，输入的第2个数据项是字符'K'。cin语句试图将这个字母读入到变量b中。然而，因为变量b是int类型变量，所以输入流进入错误状态。注意，正如第9行语句的输出结果说明的

那样，变量b，c和d中的值没有发生变化（如果程序中使用的是标准C++格式头文件，第9行语句输出的变量b，c和d的值可能会与本例不同）。

程序运行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
```

在本程序运行中，第6行的cin语句将37读入到a中，将653读入到b中，然后试图将小数点读入到c中。因为c是int类型变量，小数点被视为一个字符，所以输入流进入错误状态。正如第9行语句的输出结果说明的那样，在本例中变量c和d中的值没有发生变化（如果在程序中使用的是标准C++格式头文件，第9行语句输出的变量c和d的值可能会与本例不同）。

3.3.1 clear函数

当输入流遇到错误时，系统将忽略掉所有使用该输入流的I/O语句。可以使用clear函数将输入流恢复到正常状态。

clear函数的语法如下：

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

这里istreamVar是输入流变量，如cin。

在使用clear函数将输入流恢复成正常状态后，仍然需要清除掉输入流中剩余的垃圾数据，这可以通过ignore函数来完成。例3.6说明了这种情况。

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;
}
```

程序运行结果 在本程序运行中，用户的输入数据加有阴影。

```
Line 3: Enter a number followed by a character: 78 d
```

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

Line 10: a = 65, b = 88
```

第1行、第2行语句定义变量a和d，并分别将其初始化为23和34。第3行语句提示用户输入一个数字和一个字符。第4行语句将数字读入到变量a中，并试图将字符读入到变量b中。由于变量b是int类型变量，试图将字符读入到变量b中将导致输入流进入错误状态。正如第5行语句的输出结果说明的那样，变量b中的值没有发生变化。

第6行语句使用clear函数将输入流恢复成正常状态，第7行忽略掉输入流中剩余的其他数据。第8行语句再次提示用户输入两个数字。第9行语句将两个数字读入变量a和b中。接下来，第10行语句输出变量a和b的值（如果在程序中使用的是标准C++格式头文件，第5行语句输出的变量b的值可能会与本例不同）。

3.4 输出和格式化输出

除了编写高效率的程序代码，对于程序员来说，按照要求输出程序运行结果也是相当重要的。第2章简要介绍了向标准输出设备中输出数据的执行过程。而且，还介绍了怎样使用插入运算符<<和控制符endl向标准输出设备中输出数据。

然而，有时需要的不仅只是输出计算结果这么简单而已。有时需要按指定的方式输出浮点数数据。例如，工资单上的数字需要有两位有效小数的精度；科学试验中的浮点数字可能需要6位、7位，甚至10位的有效小数精度。而且，输出的数据可能要求按照某一列对齐；或者要求在字符串和数字之间使用其他字符，而不是使用空格符作为分隔符。例如，在输出目录页时，章节标题和对应页码之间需要使用点或是破折号填充。在本节中，将了解到各种不同的输出函数和控制符。使用这些函数和控制符，可以使用户自如地控制程序输出。

cout和插入运算符<<的使用语法如下所示：

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

这里首先计算expression的值，然后将其输出，控制符是用来控制输出的。目前遇到的最简单的控制符是endl，作用是将光标移到下一行的开头处。

需要关注的其他控制符有：setprecision，fixed，showpoint和setw。下面几节将介绍这些控制符。

3.4.1 setprecision控制符

可以使用setprecision来控制输出的浮点数小数精度。默认的浮点数输出格式是科学记数法。一些软件开发工具包（SDK）可能采用默认的最大为6位有效小数的格式来输出浮点数。然而，当输出工资单时，输出的数字只要有两位小数精度就够了。可以使用setprecision控制符来指定输出数字有两位小数精度。

使用setprecision控制符的通用语法是：

```
setprecision(n);
```

其中n是有效小数的个数。

setprecision控制符可以与插入运算符<<一同使用。例如，语句：

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

将输出的所有小数都指定为有两位有效小数，直到另一个setprecision语句再次改变输出精度为止。注意，有效小数的个数，或者说是精度，是作为参数传递给setprecision语句的。

为了使用setprecision，必须在程序中包含头文件iomanip。因此，需要使用下面的包含语句：

```
#include <iomanip>
```

3.4.2 fixed控制符

可以使用其他的控制符进一步控制浮点数的输出格式。使用fixed控制符，可以使输出的浮点数按固定小数点格式（Fixed Decimal Format）输出。下面的语句将以固定小数点格式向标准输出设备中输出浮点数：

```
cout << fixed;
```

上面的语句被执行以后，所有的浮点数都将以固定小数点格式输出，直到fixed控制符被取消。可以使用输出流成员函数unsetf来取消fixed控制符。例如，下面的语句可以取消标准输出设备中的fixed控制符：

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

在fixed控制符被取消以后，浮点数将恢复默认的输出格式。scientific控制符用来指定浮点数按照科学记数法的格式输出。

3.4.3 showpoint控制符

当计算机输出以固定小数点格式输出小数时，如果该小数点后部分的值是0，那么默认输出的数字将没有小数点和小数点后面的部分。可以使用showpoint控制符来强制显示输出数字的小数点和小数点后的0。下面的语句用来指定输出的数字必须包含小数点和小数尾部的0：

```
cout << showpoint;
```

当然，可以使用下面语句指定计算机以固定小数点格式向标准输出设备输出小数，并且强制显示小数点和小数尾部的0：

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

例3.7 下面的程序说明了控制符setprecision，fixed和showpoint的使用方法。

```
//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行、第2行、第3行语句定义变量x，y和z，并将它们分别初始化为15.674，235.73和9 525.986 4。第4行语句指定计算机以固定小数点格式输出小数，并且指定强制显示小数点和小数尾部的0。第5行语句指定输出的小数有两位精度。

第6行、第7行、第8行语句指定以两位精度输出变量x，y和z的值。注意，第8行语句输出的小数是经过四舍五入的。第9行语句指定输出的小数应该有3位精度。第10行、第11行、第12行语句指定以3位精度输出变量x，y和z的值。注意，第11行输出变量y的值应该有3位小数。但是由于变量y中的值本身只有两位小数，所以0作为第三位小数输出。

第13行语句指定输出的小数应该有3位精度。第14行、第15行、第16行语句指定以4位精度输出变量x，y和z的值。注意，第14行中输出的变量x的值的第4位小数是0，第15行中输出的变量y的值的第3位、第4位小数都是0。

第17行语句首先指定输出的浮点数有3位小数，并以3位小数输出变量x的值。在输出变量x的值以后，第17行语句指定输出的浮点数有两位小数，并以两位小数输出变量y的值。最后，指定输出的浮点数有4位小数，并以4位小数输出变量z的值。

还可以使用流函数setf来指定fixed，scientific和showpoint。在这种情况下，fixed，scientific和showpoint分别以ios::fixed，ios::scientific和ios::showpoint的形式使用，它们被称为格式标志(Formatting Flag)。

标志ios::fixed和ios::scientific都是C++数据类型ios::floatfield的一部分。当指定fixed或者scientific标志时，必须保证标志ios::fixed和ios::scientific不能同时出现，而且必须将ios::floatfield作为函数setf的第二个参数使用。下面的语句说明怎样通过标志ios::fixed，以固定小数点格式向标准输出设备输出浮点数：

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

同样，下面的语句说明了怎样使用标志ios::showpoint指定向标准输出设备输出小数，并且强制显示小数点和小数尾部的0：

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

还可以使用C++提供的setiosflags控制符来设定ios::fixed，ios::scientific和ios::showpoint标志。为了使用setiosflags控制符，程序中必须包括头文件iomanip。

下面的语句说明了怎样使用setiosflags控制符来设定ios::fixed，ios::scientific和ios::showpoint标志：

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

可以在setiosflags控制符中同时使用多个标志，每个标志之间使用符号|作为分隔符。下面的语句同时在标准输出设备中设定标志ios::fixed和ios::showpoint：

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

注意： 在ANSI/ISO标准C++中，fixed，scientific和showpoint既是控制符又是格式标志。但是，在某些标准C++编译器中，它们只能作为格式标志使用。所以在标准C++的环境下，需要通过setf函数或者setiosflags控制符两种方式来使用它们。

3.4.4 setw控制符

setw控制符用来指定输出的表达式值占用的列数（域宽）。表达式的值可以是字符串或者是数字。setw(n)控制符用来指定输出表达式的值占用n列。注意，默认输出的数据是右对齐的。因此，如果指定输出数据占用8列，并且输出的数据实际只占4列，那么前面4列应该是空的。而且，如果实际数据所占的列数大于指定的占用列数，输出数据则自动扩展到所需要占用的列数，而不是被截短。例如，如果x是int类型的变量，下面的语句用来向标准输出设备中输出x的值，并且指定x的值占用5列：

```
cout << setw(5) << x << endl;
```

为了使用setw控制符，程序中必须包括头文件iomanip。因此，下面的语句是必需的：

```
#include <iomanip>
```

与setprecision控制符（控制所有的输出数据精度直到重新设置）不同，setw只控制下一个输出数据的格式。

例3.8 下面的程序说明了控制符setw的作用。

```
//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行、第2行、第3行语句定义变量x，a和y，并将它们分别初始化为19，345和76.384。第4行语句指定计算机以固定小数点格式输出小数，并且指定强制显示小数点和小数尾部的0。第5行语句输出的是一行中各列的位置编号，这是输出的第1行数据。

第6行语句输出变量x的值，并指定占用5列。因为x的值中只有两位数字，所以只占用两位就够了。因此，第2行输出数据中的前3列是空的。第7行语句在前5列中输出变量a的值，在接下来的5列中输出字符串"Hi"，在最后5列中输出变量x的值。因为字符串"Hi"中只有两个字符，而分配给它域宽却是5列，所以前3列是空的。第4行输出的是空行，因为控制符endl在第7行语句中出现了两次。

第8行语句将所有输出的浮点数设置为两位小数格式。第9行语句在前6列中输出变量a的值，在接下来的6列中输出变量y的值，在最后的6列中输出变量x的值。第10行输出语句与第9行输出语句类似。注意第9行和第10行语句输出的数字按列对得十分整齐。第7行输出的是空行，因为控制符endl在第10行语句中出现了两次。

第11行语句在前5列中输出变量a的值，然后又输出了x的值。注意第11行语句中的控制符setw只控制a的输出。因此，在a的值输出以后，x的值在当前光标位置上输出（见第8行输出）。

在第12行cout语句中，只给变量a的输出分配了两列。然而，变量a的值有3位数字，所以输出的变量a的值自动扩展为3列。变量x的值随后在第4列中输出。因为变量x的值中只有两位数字，所以输出x的值只需要占用两位。但是，给变量x的输出分配了4列，所以前两列是空的（见第9行输出）。

3.5 其他的格式输出

前面小节中已经介绍了怎样使用setprecision，fixed和showpoint控制符来控制浮点数的输出格式，还介绍了怎样使用setw控制符来指定输出数据所占的列数。虽然，使用这些控制符足可以生成美观的报表，但是在另一些时候，你会需要更灵活地控制数据输出。在本节中，将了解到一些其他的控制输出格式的方法。

3.5.1 fill和setfill

前面已经提过，在使用setw控制符的情况下，如果指定的列数超过了数据实际需要的列数，那么输出结果是右对齐的，并且在左边没有数据的列中填充空格符。输出流变量有一个名为fill的函数和一个名为setfill的控制符。它们可以用来指定在没有数据的列中填充除空格符以外的其他字符。

使用fill函数的语法是：

```
ostreamVar.fill(ch);
```

这里ostreamVar是输出流变量，ch是一个字符。例如，语句：

```
cout.fill('*');
```

用来在标准输出设备上指定填充字符为'*'。

使用setfill控制符的语法是：

```
ostreamVar << setfill(ch);
```

这里ostreamVar是输出流变量，ch是一个字符。例如，语句：

```
cout << setfill('#');
```

用来指定在标准输出设备上的填充字符为'#'。

为了使用setfill控制符，程序中必须包含头文件iomanip。

如果使用fill函数来设置填充字符，那么它必须是一条单独的语句。setfill控制符只是输出语句的一部分。下面的例子说明了函数fill和控制符setfill的使用方法。

例3.9 下面的程序说明了fill和setfill的作用。

```
//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、第2行语句定义变量x和y，并分别将其初始化为15和7634。第3行语句的输出结果（第1行输出）是一行中各列的位置编号，这些编号为后面语句的输出结果起参照作用。第4行语句输出变量x的值占用5列；变量y的值占用7列；字符串"Warm"占用8列。这条语句的输出行的填充字符是空格符。

第5行语句指定填充字符是'*'。第6行语句输出变量x的值占用5列；变量y的值占用7列；字符串"Warm"占用8列。因为变量x的值只有2位数字而分配给它的输出列宽却有5列，所以未被使用的前3列以字符'*'填充。程序为变量y分配了7列，而y的值只有4位数字，前3列以'*'填充。同样，程序为字符串"Warm"的输出分配了8列，而字符串"Warm"实际只占4位，所以前4列以字符'*'填充。见输出行第3行。

第7行语句的输出结果（输出行第4行）与第6行语句的输出结果相似。只是变量y和字符串"Warm"前面的填充字符变为#。第8行语句的输出结果（输出行第5行）中，x前面的填充字符是@，y前面的填充字符是#，字符串"Warm"前面的填充字符是^。setfill控制符设置了这些填充字符。

第9行语句将填充字符设置为空格符。第10行语句输出了变量x，y的值和字符串"Warm"，并使用空格符作为填充字符。见输出行第6行。

3.5.2 left和right控制符

前面已经提到，如果在setw控制符中指定的输出列数（域宽）大于实际输出的数据所需要占用的列数，默认时输出的数据将右对齐。但是有时输出数据需要左对齐。这时，可以使用left控制符指定为左对齐。

设置left控制符的语法是：

```
ostreamVar << left;
```

这里ostreamVar是输出流变量。例如，下面的语句指定以左对齐方式向标准输出设备中输出数据：

```
cout << left;
```

可以使用unsetf流函数取消left控制符的格式控制。取消left控制符的语法是：

```
ostreamVar.unsetf(ios::left);
```

这里ostreamVar是输出流变量。取消left会控制符使输出格式恢复成默认格式。例如，下面语句用来在标准输出设备中取消left控制符：

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

设置right控制符的语法是：

```
ostreamVar << right;
```

这里ostreamVar是输出流变量。例如，下面的语句指定以右对齐方式向标准输出设备中输出数据：

```
cout << right;
```

例3.10 下面的程序说明了控制符left和right的作用。

```
//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
```

本程序的输出结果与例3.9中程序的输出结果相似。惟一的差异是在第4行至第9行语句中指定的输出方式为左对齐。建议读者仔细阅读本程序。

也可以使用setf流函数来设置左对齐方式或者右对齐方式输出数据。在这种情况下，left和right的使用方式分别是ios::left和ios:: right，它们被称为格式标志。ios::left和ios::right格式标志都是C++数据类型ios::adjustfield的一部分。必须保证ios::left和ios:: right格式标志不能同时出现，而且必须将ios:: adjustfield作为setf函数的第二个参数使用。

使用 left 标志的语法是：

```
ostreamVar.setf(ios::left, ios::adjustfield);
```

这里 ostreamVar 是输出流变量。例如，下面语句指定以左对齐方式向标准输出设备中输出数据：

```
cout.setf(ios::left, ios::adjustfield);
```

可以使用输出 unsetf 流函数取消 ios::left 或 ios:: right 格式标志。取消 ios::left 标志的语法是：

```
ostreamVar.unsetf(ios::left);
```

这里 ostreamVar 是输出流变量，取消 ios::left 标志将会恢复默认的输出格式。

可以使用 setiosflags 流控制符来设置 ios::left 和 ios:: right 标志。例如，下面语句为标准输出设备设置 ios::left 标志：

```
cout << setiosflags(ios::left);
```

注意：下面语句使用 setiosflags 控制符为标准输出设备设置 ios::fixed，ios::showpoint 和 ios::left 标志：

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

注意：在标准 C++ 中，left 和 right 控制符只可以作为格式标志使用。因此，必须以 setf 函数或 setiosflags 控制符来设置 left 和 right。详细信息请参阅编译器的说明文档。

3.5.3 flush 函数

endl 控制符和 '\n' 换行符都可以将光标移动到输出设备中下一行的开头处。但是，endl 控制符还有另外的用途。

当程序向输出设备中输出数据时，输出的数据先被存放在计算机的缓冲区（Buffer）内。当缓冲区存满时，这些数据才真正地输出到输出设备。但是，如果输出的字符序列中出现了 endl 控制符，那么缓冲区内的所有数据将立即输出到输出设备，而无论缓冲区是否已经存满。因此，endl 控制符的作用是将光标移到输出设备中下一行的开头处，并且清空缓冲区。

很有可能出现在程序终止时，并没有输出所有的输出数据的情况。这是因为在程序终止时，缓冲区并不一定是满的，所以也就没有将缓冲区中的数据写到输出设备。

在 C++ 中，可以使用 flush 函数来清空缓冲区，即使缓冲区中的数据不是满的。与 endl 控制符不同的是，flush 函数并不把光标移到下一行的开头处。

使用 flush 函数的语法是：

```
ostreamVar.flush();
```

这里 ostreamVar 是输出流变量，例如 cout。

与 endl 一样，flush 可以作为控制符使用。在这种情况下，flush 使用在输出语句中，并不加括号。例如，下面的语句将数据从缓冲区写到标准输出设备：

```
cout << flush;
```

例 3.11　考虑下面语句，其中 num 是 int 类型变量：

```
cout<<"Enter an integer:";       //Line 1
cin >>num;                       //Line 2
cout<<endl;                      //Line 3
```

第 1 行语句输出文字："Enter an integer:"。在输出这一行文字后，光标停留在冒号后面的位置上。注意，第 1 行语句的输出首先被送到缓冲区中。如果缓冲区中的数据没有存满，那么这行提示文字

就不会被显示出来，这时用户也就不知道下一步应该做什么。可以在第1行语句后面使用endl控制符。但是如果这样做，在输出这行文字后，光标将被移到下一行的开头处，用户也就必须在下一行中输入数字。而这样做并不是最恰当的。还可以使用下面语句替换掉第1行语句：

```
cout << "Enter an integer:" << flush;        //Line 1
```

在这种情况下，文字行“Enter an integer:”，即使在缓冲区中数据没有存满时也会立即被输出到标准输出设备上。而且，在输出这行文本后，光标将停留在分号的下一个位置上。用户将在分号后面输入数字。

注意：本章介绍了许多流函数和流控制符。为了使用get，ignore，fill，clear和setf等流函数，就必须在程序中包含头文件iostream。有两种类型的控制符：带参数的控制符的和不带参数的控制符。带参数的控制符又被称为含参流控制符（Parameterized stream manipulators）。
例如，setprecision，setw，setfill和setiosflags等控制符都是含参流控制符。而endl，fixed，scientific，showpoint和left等都是无参流控制符。因为flush也可以作为不带参数的控制符使用，所以它也是无参流控制符。
为了可以使用含参流控制符，即带有参数的控制符，就必须在程序中包含头文件iomanip。如果只使用无参流控制符，则不需要包含头文件iomanip，因为无参流控制符包含在另一个头文件iostream中。

3.5.4 输入/输出string类型数据

可以使用输入流变量，例如cin和析取运算符>>将字符串读入到string类型变量中。例如，如果输入的字符串是"Shelly"，那么下面的语句将把该字符串输入到string类型的变量name中：

```
string name; //declaration
cin >> name; //input statement
```

注意，析取运算符在输入前先略掉字符串前所有的空白符，并将输出后在遇到第一个空白字符时立即停止输入操作。因此，不能使用析取运算符读入含有空格符的字符串。例如，假设变量name的定义如上，并且输入的字符串是：

```
Alice Wonderland
```

那么在下面的语句执行过后，变量name中的值将是"Alice"：

```
cin >> name;
```

可以使用getline函数来读入含有空格符的字符串。使用getline函数的语法是：

```
getline(istreamVar, strVar);
```

这里istreamVar是输入流变量，strVar是string类型变量。该函数读入以'\n'换行符为分隔符的字符串。

getline函数在遇到当前行的结束标志'\n'后，才能停止读入字符。换行符虽然也被读入，但并不存储到string类型变量中。

考虑下面的语句：

```
string myString;
```

如果输入的是如下29个字符：

```
bbbbHello there. How are you?
```

这里，b代表空格符。在语句：

```
getline(cin, myString);
```

执行过后，*myString* 中的值是：

```
myString = "    Hello there. How are you?"
```

所有的29个字符，包括前面的4个空格符，将被存储到变量myString中。

类似地，还可以使用输出流变量，如cout和插入运算符<<输出string类型变量中的值。

3.6 文件输入/输出

前面几节中主要讨论了怎样从键盘（标准输入设备）输入数据以及怎样向屏幕（标准输出设备）输出数据。但是从键盘读入数据和向屏幕输出数据有许多的局限性。当输入的数据量很少时，很适合从键盘中读入数据；当输出的数据量很少时（一屏以内），适合向屏幕输出数据。

但是，在需要输入大量数据的情况下，在每次程序运行时都从键盘输入数据将是一件很费力的事。不仅如此，从键盘输入大量数据时，还极容易产生输入错误，而这些输入错误又会导致程序的运行失败或产生错误的计算结果。必须找到一种可以避免上述问题的数据输入方法。这种数据输入方法就是在运行前准备好输入数据，并在每次运行程序时直接使用这些数据。

假定需要在会议上显示程序运行的结果。以分发打印好的程序运行结果显然比直接在屏幕上显示运行结果要好得多。例如，在召开重要的会议前，给每位与会者一份打印报告。另外，有时还需要将程序的运行结果存储起来，以便在另一个程序中作为输入使用。

本节将介绍怎样从其他输入设备，如硬盘（即外部存储器）中读入数据以及怎样将数据输出到硬盘中。C++允许程序直接从硬盘上读入数据以及向硬盘输出数据。程序可以通过文件I/O操作向外部存储器的文件读写数据。文件的正式定义如下所示：

文件　在外部存储器中一块可以存储信息的区域。

标准I/O头文件，iostream中的数据类型和变量只能用来从标准输入设备读入或者向标准输出设备输出数据。除此以外，C++还提供了用来支持文件I/O操作的头文件fstream。头文件fstream包含了两种数据类型的定义：ifstream表示输入文件流，作用与istream相似；ofstream表示输出文件流，作用与ostream相似。

C++已经定义变量cin和cout，并将它们与标准输入/输出设备分别联系起来。而且，还可以将>>，get，ignore，putback，peek等与cin联合使用，将<<，fill，setfill等与cout联合使用来完成特定的功能。虽然同样的函数和控制符也可以用于文件I/O中，但头文件fstream中并没有定义相应的变量来使用它们。必须自己定义文件流变量（File Stream Variables），其中包括输入中使用到的ifstream变量和输出中使用到的ofstream变量。然后就可以将这些变量与>>和<<或者其他的I/O函数联合使用了。注意C++不能自动初始化用户自定义变量。一旦定义了fstream变量，就必须将这些变量与输入/输出文件联系起来。

文件I/O过程可分为5个步骤：

1. 在程序中包含头文件fstream
2. 定义文件流变量
3. 将这些文件流变量与输入/输出源联系起来
4. 将文件流变量与>>，<<以及其他I/O函数联合使用
5. 关闭文件

我们现在将详细介绍这5个步骤。然后程序范例将解释怎样在程序中完成这5个步骤。

第1步需要在程序中包含头文件fstream，该任务可以通过下面的语句来实现：

```
#include <fstream>
```

第2步定义文件流变量，考虑下面语句：

```
ifstream inData;
ofstream outDate;
```

第1行语句将inData定义成输入文件流变量，第2行语句将outData定义成输出文件流变量。

第3步将定义好的文件流变量与输入/输出文件联系起来。这一步也称为打开文件。可以使用open流成员函数打开文件。打开文件的语法是：

```
fileStreamVariable.open(sourceName, fileOpeningMode);
```

这里fileStreamVariable是文件流变量，sourceName是输入/输出文件的文件名，fileOpeningMode指定文件打开的方式。fileOpeningMode加有阴影表示在语法定义中这部分是可选的。对于ifstream类型变量，默认的打开方式是输入数据；对于ofstream类型变量，默认的打开方式是输出数据。表3.2中列出了各种文件打开方式。

表3.2 文件打开方式

文件打开方式	说明
ios::in	以输入方式打开文件，这是ifstream类型变量默认的打开方式
ios::out	以输出方式打开文件，这是ofstream类型变量默认的打开方式
ios::nocreate	如果文件不存在，则打开操作失败
ios::app	如果文件存在，则从文件末尾处开始写入数据（附加）；如果文件不存在，创建一个新文件
ios::ate	以输出方式打开文件并将文件读写位置移到文件尾。输出数据可以写到文件中任何地方
ios::trunc	如果文件存在，其中的内容将被删除（截掉）
ios::noreplace	如果文件存在，则打开操作失败

假定在程序中按照上面第2步定义了文件流变量。并且假定输入数据存放在A驱动器软盘上的prog.dat文件中，输出数据存放在A驱动器软盘上的prog.out文件中。下面的语句将inData和prog.dat以及outData和prog.out分别建立起关联。也就是说，prog.dat文件用于数据输入，prog.out文件用于数据输出。

```
inData.open("A:prog.dat");      //open input file
outData.open("A:prog.out");     //open output file
```

通常第4步按照下面的方法进行读写，即将文件流变量与>>,<<或者其他的输入/输出函数联合使用。除了文件流变量必须先经过定义以外，使用文件流变量的语法与使用cin和cout的语法完全一样。例如，语句：

```
inData >> payRate;
```

将数据从prog.dat文件中读入到payRate变量中。语句：

```
outData << "The paycheck is: $" << pay << endl;
```

将The paycheck is: $565.78输出到prog.out文件中。这里假定变量pay中的数据是565.78。

一旦I/O操作完成，需要执行第5步中的关闭文件操作。关闭文件意味着文件流变量与外部存储器中文件的存储区失去连接，文件流变量也随之进入空闲状态。一旦文件流变量空闲，其他的I/O文件就可以与它们建立连接。可以使用close流函数来关闭文件。例如，关闭在上面打开的文件的语句是：

```
inData.close();
outData.close();
```

注意： 在某些系统中，并不需要关闭文件。因为在程序结束时，所有打开的文件将自动关闭。不管怎样，自己关闭文件是一个良好的习惯。另外，在使用同一个文件流变量打开另一个文件前，必须先关闭正在与之关联的文件。

在程序中使用文件I/O，通常采用下面的形式：

```
#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步需要打开文件以便进行I/O操作。打开文件意味着将程序中定义的文件流变量与外部存储器上的物理文件建立连接。输入文件在打开前必须已经存在。如果open语句中指定的文件不存在，将导致open语句失败，而且导致输入流进入错误状态。输出文件在打开前并不要求一定存在。如果在open语句中指定的文件不存在，计算机将按照其名字建立一个新的空文件。如果指定的文件已经存在，在默认情况下，该文件中所有已经存在的内容将被删除。

注意： 在本节中介绍了怎样使用open函数打开文件，以及怎样向这些文件读写数据。假设程序为了读取数据试图打开一个不存在的文件，则会导致open语句失败，并且导致输入流进入错误状态。为了防止这种情况发生，一些系统将自动为输入数据建立空的输入文件，以避免open语句失败。

可以通过在open语句中使用第二个参数ios::nocreate，来避免程序在打开不存在的输入文件时自动创建空文件。第二个参数ios::nocreate的作用是使在输入文件不存在时导致open语句失败。例如，假定变量的定义和初始值不变。下面的语句打开输入文件prog.dat；如果该文件不存在，open操作失败。

```
inData.open("A:prog.dat", ios::nocreate);
```

3.7 程序范例：电影票销售和慈善捐赠

某地的一家电影院的售票情况很好。为了帮助当地慈善事业，剧院的所有者决定将电影票总销售额的一部分捐赠给慈善事业。本例设计并编写一个程序，该程序提示用户输入电影名称、成人票价、儿童票价、销售的成人票数量、销售的儿童票数量和捐赠额占总销售额的比例。程序的输出形式如下所示：

```
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
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
```

注意：第1列中的数据是左对齐的；右边列中的数字是右对齐的，并且保留两位精度。

输入　程序的输入包括电影名称、成人票价、儿童票价、销售的成人票数量、销售的儿童票数量和捐赠额占总销售额的比例。

输出　输出的形式如上。

问题分析和算法设计

为了计算总捐赠额（Amount Donated）和净销售额（Net Sale），必须首先知道总销售额（Gross Amount）。成人票价乘以销售的成人票数量与儿童票价乘以销售的儿童票数量的和就等于总销售额。即:

```
grossAmount = adultTicketPrice * noOfAdultTicketsSold
              + childTicketPrice * noOfChildTicketsSold;
```

接下来需要确定捐赠额占总销售额的比例，然后通过在总销售额中减掉捐赠额得到净销售额。下面给出了捐赠额和净销售额的计算公式。通过分析可以得到如下的算法：

1. 输入电影名称
2. 输入成人票价
3. 输入儿童票价
4. 输入销售的成人票数量
5. 输入销售的儿童票数量
6. 输入捐赠额占总销售额的比例
7. 应用下面公式计算总销售额：

```
grossAmount = adultTicketPrice * noOfAdultTicketSsold
              + childTicketPrice * noOfChildTicketsSold;
```

8. 应用下面公式计算捐赠额：

```
amountDonated = grossAmount * percentDonation / 100;
```

9. 应用下面公式计算净销售额：

```
netSale = grossAmount - amountDonated;
```

变量　由上面讨论可知，需要定义变量分别存储电影名称、成人票价、儿童票价、销售的成人票数量、销售的儿童票数量、捐赠额占总销售额的比例、总销售额、捐赠额以及净销售额。因此，这些变量的定义如下所示：

```
string      movieName;
double      adultTicketPrice;
double      childTicketPrice;
int         noOfAdultTicketsSold;
int         noOfChildTicketsSold;
double      percentDonation;
double      grossAmount;
double      amountDonated;
double      netSaleAmount;
```

因为movieName定义为string类型变量，所以需要包含头文件string。所以，在程序中需要包含下面的语句：

```
#include <string>
```

格式化输出 在输出中，第1列数据是左对齐的，第2列数字是右对齐的。因此，在输出第1列数据时，需要使用left控制符；在输出第1列数字时，需要使用right控制符。第1列与第2列输出数据间以点填充，这可以通过使用setfill控制符来完成。在总销售额、捐赠额和净销售额的输出行中，符号$和数字以空格符分隔。因此，在输出符号$前，需要使用setfill控制符将分隔符设置为空格符。下面的代码可以完成相应的输出：

```
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 经过上面的讨论，主要算法如下所示：

1. 定义变量。
2. 将输出的浮点数设置为两位小数固定小数点格式；不足两位小数的数字，尾部补零。因此，还需要包括头文件iomanip。
3. 提示用户输入电影名称。
4. 输入（读入）电影名称。由于电影名称可能由不止一个单词组成（也就是可能含有空格符），应该在程序中使用getline函数来读入字符串。需要特别注意的是，由于getline函数同时读取换行符，所以在一行内输入电影名称后，需要按两次回车键。
5. 提示用户输入成人票价。
6. 输入（读入）成人票价。
7. 提示用户输入儿童票价。
8. 输入（读入）儿童票价。
9. 提示用户输入成人票的销售数量。
10. 输入（读入）成人票的销售数量。
11. 提示用户输入儿童票的销售数量。
12. 输入（读入）儿童票的销售数量。
13. 提示用户输入捐赠额占总销售额的比例。
14. 输入（读入）捐赠额占总销售额的比例。
15. 计算总销售额。

16. 计算捐赠额。
17. 计算净销售额。
18. 输出结果。

完整的程序代码清单

```
#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;
}
```

程序运行结果 在本程序运行结果中，用户输入的数据加有阴影。

```
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
```

在该输出中，前6行为后6行的输出请求提供了必要的数据。

3.8 程序范例：学生成绩

编写一个程序，该程序读入学生ID和5门课程的考试分数。程序输出学生ID，5门考试分数以及平均考试分数。输出的考试分数保留两位精度。假定学生ID是一个字符。

需要读入的数据存储到名字为test.txt的文件中，该文件存储在A驱动器的软盘中。

输入 一个含有学生ID和5门考试分数的文件。

输出 学生ID，5门考试分数以及平均考试分数，并存储到文件中。

问题分析和算法设计

将5门考试分数加起来再除以5，就得到了5门考试的平均分数。输入数据的方式是：学生ID后跟5门考试分数。因此，必须先读入学生ID再读入5门考试分数。通过问题分析可以得到如下的算法：

1. 读入学生ID和5门考试分数
2. 输出学生ID和5门考试分数
3. 计算平均成绩
4. 输出平均成绩

变量 程序需要读入学生ID和5门考试分数。因此，需要1个变量存储学生ID，需要5个变量分别存储5门考试分数。为了计算平均分数，需要将5门考试分数加起来并除以5。因此，需要变量存储平均考试分数。而且，由于输入数据在文件中，所以需要一个ifstream变量来打开该输入文件；由于输出的数据也要写入到一个文件中，所以需要一个ofstream变量来打开输出文件。所以，程序中至少需要定义如下变量：

```
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
```

在上面讨论的基础上，得到主要算法如下所示：

主要算法

1. 定义变量。
2. 打开输入文件。
3. 打开输出文件。
4. 通过使用fixed和showpoint控制符，以固定小数点格式输出数字。而且，为了输出有两位小数的数字，需要设置两位精度，不足两位小数的数字通过在尾部添0补齐。
5. 读入学生ID。
6. 输出学生ID。
7. 读入5门考试分数。
8. 输出5门考试分数。
9. 计算平均分数。
10. 输出平均分数。
11. 关闭输入文件和输出文件。

完整的程序代码清单

```
//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;
}
```

程序运行结果

输入文件（输入文件 a:test.txt 中的内容是）:

```
T 87 89 65 37 98
```

输出文件（输出文件 a:testavg.out 中的内容是）:

```
Student ID: T
Test scores:     87   89   65   37   98
Average test score:   75.20
```

注意：如果用标准 C++ 编写程序代码，需要将下面语句：

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

替换成为：

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

或许还要将语句：

```
outFile << fixed << showpoint;
```

替换成为：

```
outFile.setf(ios::fixed, ios::floatfield);
outFile.setf(ios::showpoint);
```

3.9 小结

1. C++ 中的流是从源到目标的一系列字符的有限序列。
2. 输入流是从源到计算机的流。
3. 输出流是从计算机到目标的流。
4. cin 代表标准输入，是输入流对象，通常初始化为标准输入设备——键盘。
5. cout 代表标准输出，是输出流对象，通常初始化为标准输出设备——屏幕。
6. 双目运算符>>和输入流对象，如cin，结合使用时被称为流析取运算符。>>左边的操作对象必须是输入流变量，如 cin，右边的操作数必须是变量。
7. 双目运算符<<和输出流对象，如cout，结合使用时被称为流插入运算符。<<左边的操作对象必须是输出流变量，如 cout，右边的操作数必须是变量。
8. 当使用运算符>>将数据读入变量时，将略掉数据前的所有空白符。
9. 为了使用 cin 和 cout，程序中必须包括头文件 iostream。
10. get 函数用于逐个读入字符，并且不会略掉任何空白字符。
11. ignore 函数用于在一行中略掉某些数据。
12. putback 函数用于将 get 函数读入的最后一个字符退回到输入流中。
13. peek 函数返回输入流中的当前字符，但是并不将该字符从输入流中移走。
14. 试图将非法数据读入变量将导致输入流进入错误状态。
15. 一旦输入流进入错误状态，可以使用 clear 函数将输入流恢复成正常状态。
16. setprecision 控制符用来设置输出浮点数的精度。
17. fixed 控制符将浮点数以固定小数点格式输出。
18. showpoint 控制符用来指定浮点数必须含有小数点和小数尾部的零。
19. setw 控制符用来规定按照指定列数（域宽）输出数据，默认是右对齐输出方式。
20. 如果在 setw 控制符中指定的输出列宽小于数据实际需要的输出宽度，输出数据不会被截短，而是按照实际宽度输出。
21. 为了在标准 I/O 中使用 get，ignore，putback，peek，fill，clear，setf 和 unsetf 流函数，程序中必须要包含头文件 iostream。
22. 为了在程序中使用 setprecision，setw，setfill 和 setiosflags 控制符，程序中必须要包含头文件 iomanip。
23. 头文件 fstream 中包含 ifstream 和 ofstream 的定义。
24. 对于文件 I/O，必须使用语句 #include <fstream>将头文件 fstream 包含在程序中。而且还要做下面的事情：为输入文件定义 ifstream 类型的变量，为输出文件定义 ofstream 类型的变量，使用 open 语句打开输入文件和输出文件，将<<，>>，get，ignore，peek，putback 和 clear 与文件流变量联合使用以完成特定的功能。
25. 关闭文件需要调用 close 函数。例如关闭 ifstream 类型变量 inFile，需要使用语句 "inFile.close( );"，关闭 ofstream 类型变量 outFile，需要使用语句 "outFile.close( );"。

3.10 练习

1. 判断下面说法的正误。

   a. 析取运算符>>在输入流中搜索下一个数据项时，将略掉该数据前的所有空白字符。

   b. 在语句“cin >> x;”中，x必须是变量。

   c. 在语句“cin >> x >> y;”中，需要在同一行中输入x和y的值。

   d. 语句“cin >> num;”和语句“mun >> cin;”是等同的。

   e. 可以通过键入键盘上的Enter（回车）键生成换行符。

   f. ignore函数的作用是略掉某一行中的一些输入数据。

2. 假设x和y是int类型变量，ch是char类型变量。考虑下面的输入数据：

   ```
   5    28    36
   ```

   在下面各语句执行以后，变量x，y和ch中的值（如果存在）分别是多少（每条语句的输入数据相同）？

   a. `cin>>x>>y>>ch;`

   b. `cin>>ch>>x>>y;`

   c. `cin>>x>>ch>>y;`

   d.
   ```
   cin>>x>>y;
   cin.get(ch);
   ```

3. 假设x和y是int类型变量，ch是char类型变量。假设输入的数据如下所示：

   ```
   13   28   D
   14   E    98
   A    B    56
   ```

   在下面各语句执行以后，变量x，y和ch中的值（如果存在）分别是多少（每条语句的输入数据相同）？

   a.
   ```
   cin>>x>>y;
     cin.ignore(50,'\n');
     cin>>ch;
   ```

   b.
   ```
   cin>>x;
     cin.ignore(50,'\n');
     cin>>y;
     cin.ignore(50,'\n');
     cin.get(ch);
   ```

   c.
   ```
   cin>>y;
     cin.ignore(50,'\n');
     cin>>x>>ch;
   ```

   d.
   ```
   cin.get(ch);
     cin.ignore(50,'\n');
     cin>>x;
     cin.ignore(50,'E');
     cin>>y;
   ```

4. 假设输入的数据如下所示：

   ```
   46   A     49
   ```

C++代码段如下所示：

```
int x = 10, y = 18;
char z = 'A';
cin>>x>>y>>z;
cout<<x<<" "<<y<<" "<<z;
```

输出的结果是什么？

5. 编写一条C++语句，该语句使用setfill控制符生成一个有35个星号的输出行。例如：

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

6. 下面程序的功能是从一个名为input.dat的文件中读入两个数，并将这两个数的和写入到一个名为output.dat的文件中。但是，本程序没能完成上述功能。改写本程序，使其能够完成上述功能。

```
#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. 什么情况会导致输入流进入错误状态？当输入流进入错误状态时，会产生哪些后果？

8. 某个程序从一个名为inputFile.dat的文件中读入数据，在进行某些计算后，再将计算结果写到一个名为outFile.dat的文件中。回答下面问题：

   a. 在程序执行以后，文件inputFile.dat中的内容是什么？

   b. 在程序执行以后，文件outFile.dat中的内容是什么？假设在程序执行前该文件中没有任何数据。

   c. 在程序执行以后，文件outFile.dat中的内容是什么？假设在程序执行前该文件中含有100个数字。

   d. 如果在程序执行前文件outFile.dat并不存在，程序将会怎样？

3.11 编程练习

1. 考虑下面不完整的程序：

```
#include <iostream>

int main()
{
    ...
}
```

   a. 编写一条语句将头文件fstream包含在程序中。

   b. 给出语句定义ifstream类型变量inFile和ofstream类型变量outFile。

   c. 程序从文件inData.txt中读取数据，并把输出数据输出到文件outData.dat中。编写语句打开这两个文件，并分别将inFile关联到inData.txt上，将outFile关联到outData.dat上。

   d. 假设文件inData.txt包含下面的数据：

```
56   38
A
7    8
```

编写语句使得在程序执行以后，文件outData.dat中的数据如下。如果需要，可以定义其他变量。这些语句应该是通用的，即在输入文件的内容改变后，重新运行（并不重新编辑或编译）程序，可以输出相应的结果：

```
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. 编写语句关闭输入文件和输出文件。

f. 编写一个可以测试a~e部分的语句的程序。

2. 编写一个程序，该程序提示用户输入一个小数，四舍五入到小数点后两位数，并输出该数。
3. 某足球场经理希望编写一个程序，该程序用于在每场比赛后计算总售票收入。球票共分为4种：B（Box），S（Sideline），P（Premium）和G（General Admission）。在每场比赛结束以后，数据以下面的格式存储到文件中：

```
ticketPrice     numberOfTicketsSold
...
```

范例数据如下所示：

```
250  5750
100  28000
 50  35750
 25  18750
```

第1行数据的意思是价值为$250的球票总共售出了5 750张。计算并输出销售出的球票总数以及总销售额。输出数据采用两位小数格式。

4. 编写一个用于计算财产税的程序。财产税按照财产总价值的92%计税。例如，如果财产的总价值为$100 000，那么需要计税的部分为$92 000。假设财产税税率为1.05%。程序提示用户输入财产的总价值，并以以下格式将结果输出到文件中（下面给出了输出数据范例）：

```
Assessed Value:                  100000.00
Taxable Amount:                   92000.00
Tax Rate for each $100.00:            1.05
Property Tax:                       966.00
```

输出的数字采用两位小数格式（注意左边一列数据采用左对齐方式，右边一列数据采用右对齐方式）。

5. 编写一个程序，该程序可以将华氏温度换算成相应的摄氏温度。摄氏温度与华氏温度的换算公式为：

```
C = (5 / 9) (F - 32)
```

程序将提示用户输入华氏温度，并输出该华氏温度和相应的摄氏温度。

6. 编写一个程序，该程序用来计算和打印员工的月工资单。工资总数减去下面一些抵扣项就得到净工资数。

```
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
```

程序将提示用户输入工资总数和员工姓名。输出的数据将存储到文件中。输出数据中数字采用两位小数形式。输出数据范例如下所示：

```
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 1 列是左对齐的，而第 2 列是右对齐的。

第4章　控制结构 I（选择）

本章要点：

- 了解控制结构
- 了解关系运算符和逻辑运算符
- 理解怎样使用和计算逻辑（布尔）表达式
- 了解怎样在程序中使用 if，if…else 和 switch 来构造选择程序结构
- 理解怎样使用 assert 函数来终止程序运行

在第2章中将程序定义为语句的序列，通过执行这些语句可以完成特定的功能。到现在为止，在本书中见到的程序都很简单。在这些程序的执行过程中，计算机从第一条语句开始，按顺序执行到最后一条语句。在本章和第5章中，将了解怎样使用控制结构来改变程序的顺序执行。程序可以根据条件判断执行或是循环执行某些语句。

4.1　控制结构

计算机可以按三种方式执行程序：顺序（Sequence）执行、选择（Selection）执行（也称为分支）、循环（Repetition）执行（反复执行某些语句），如图4.1所示。第2章和第3章中出现的程序都是采用简单的顺序执行方式。在这些程序中，计算机从第一条语句开始，按顺序执行到最后一条语句。既没有选择执行，也没有循环执行。控制结构可以改变程序的执行顺序。在选择结构中，程序可以根据某个条件选择执行某些语句；在循环结构中，程序可以按指定的次数执行某些语句。

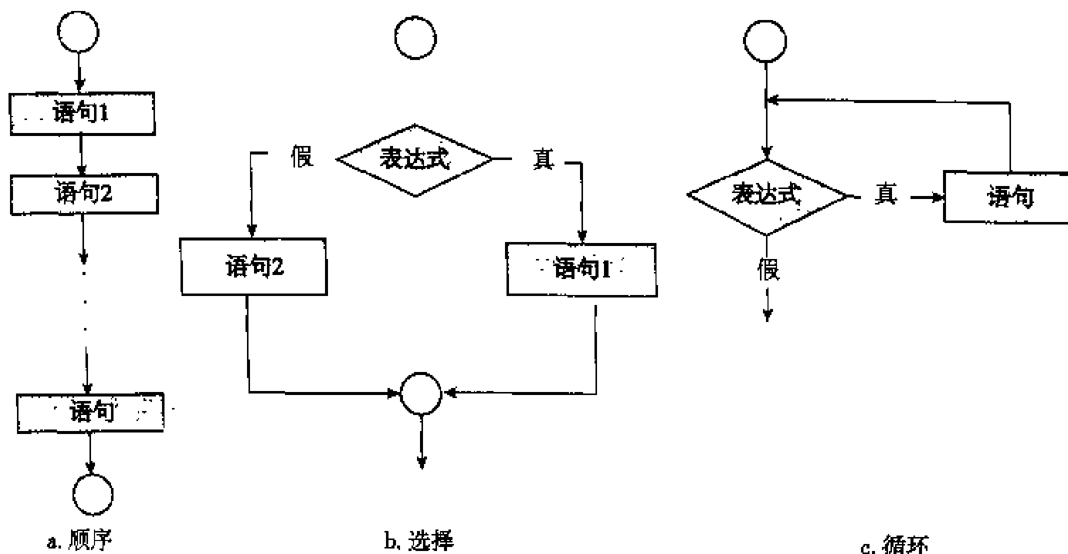


图4.1　程序执行顺序

在说明选择结构和循环结构之前，有必要先对条件语句性质及使用方法做简单的介绍。考虑下面三条语句：

1. if（分数大于或等于 90 分）
   成绩是 A
2. if（工作小时数小于或等于 40）

```
wages = rate * hours
```

否则

```
wages = (rate * 40) + 1.5 * (rate * (hours - 40))
```

3. if（气温高于 70 华氏度并且没有雨）
   很适合打高尔夫球

这些都是条件语句的例子。从这些语句中不难看出，只有在满足特定的条件下，某些语句才会得以执行。如果 if 后面表达式的值为真，那么就称该条件满足。例如，对于语句 1 中表达式：

分数大于或等于 90 分

只有当分数大于或等于 90 分时，表达式的值才为真（true）；否则为假（false）。例如，当分数为 95 分时，该表达式的值为真；当分数为 86 分时，该表达式的值为假。

如果程序具备判断条件表达式真假的能力，那么计算机将会发挥更大的作用。另外，在某些情况下，并不能只根据一个条件就判断出整个表达式的真假。例如在上面语句 3 中，必须同时满足气温高于 70 华氏度和没有雨两个条件，表达式的值才为真，适合进行的活动才是打高尔夫球。

正如在上面例子中看到的，只有计算机具备在程序运行过程中判断出条件表达式真假的能力，选择语句和循环语句才可以真正地发挥作用。接下来的几节中将介绍怎样在 C++ 中使用和计算条件表达式。

4.2 关系运算符

为了可以做出判断，程序必须能够表示条件和做出比较。例如，活期存款账户的利息和服务费取决于每个月月末账户上的存款余额。如果该存款余额低于规定的最小存款余额，那么利息率将降低并要收取一定的服务费。因此，为了确定适用的利息率，就必须要表示出规定的最小存款余额（条件），并且将其与储户的账户余额做出比较。保险费的确定也需要表示条件和做出比较两个过程。例如，为了确定人寿保险的保险费，就必须要知道投保人是否吸烟。不吸烟者（条件）的保险费要低于吸烟者的保险费。上面两例都涉及到做出比较。可以有几种形式的比较，例如：比较是否与特定条件相等、比较是否大于或者小于特定条件。

在 C++ 中，条件表示为逻辑表达式。值为 true 或是 false 的表达式称为逻辑（布尔）表达式，true 和 false 称为逻辑（布尔）值。假设 I 和 J 都是整数，考虑下面的表达式：

```
I > J
```

如果该表达式是逻辑表达式，那么当 I 大于 J 时该表达式的值为 true；否则该表达式的值为 false。可以将符号>看做一个运算符，该运算符（在本例中）有两个整型操作数，并且产生逻辑类型的运算结果。这里，运算符>与产生整型或者浮点型结果的运算符 + 和 - 十分类似。实际上，运算符>称为关系运算符。因为只有当 I 和 J 间的“大于”关系成立时，表达式 I>J 的值才是 true。关系运算符在程序中用来做比较。

C++ 中共有 6 种关系运算符，这些关系运算符用来表示条件和做出比较。表 4.1 列出了这些关系运算符。

表 4.1　C++ 中的关系运算符

运算符	说明
==	等于
!=	不等于

（续表）

运算符	说明
<	小于
<=	小于或等于
>	大于
>=	大于或等于

注意： 在 C++ 中，由两个等号组成的符号 == 称为恒等运算符（Equality Operator）。前面已经提到，= 称为赋值运算符。需要特别注意的是：恒等运算符（==）用来判断两个表达式的值是否相等；而赋值运算符（=）用来给变量赋值。

所有的关系运算符都是双目运算符，也就是说有两个操作数。因为比较的结果不是 true 就是 false，所以关系表达式的结果不是 true 就是 false。

4.2.1　关系运算符和简单数据类型

三种简单数据类型（整型、浮点型和枚举型）数据都可以作为关系运算符的操作数。例如，下面的表达式中有整型和浮点型两种类型数据。

表达式	含义	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 小于或等于 7.5	true

注意： 浮点数的恒等比较运算通常是依赖于特定机器的。很有可能在某些机器上，表达式：

```
6.8 + 3.1 == 2.7 + 7.2
```

的计算结果是 false。

如果在关系表达式中使用 char 类型数据，那么将按照这些 char 类型数据在机器编码中的值来计算关系表达式的值。表 4.2 说明了含有 ASCII 码的关系表达式的值是如何确定的。

表 4.2　使用 ASCII 码排列顺序和关系运算符来计算表达式的值

表达式	表达式的值	解释
' ' < 'a'	true	' '（空格符）的 ASCII 码值是 32，'a' 的 ASCII 码值是 97。因为 32 < 97 的值为 true，所以 ' ' < 'a' 的值为 true
'R' > 'T'	false	'R' 的 ASCII 码值是 82，'T' 的 ASCII 码值是 84。因为 82 > 84 的值为 false，所以 'R' > 'T' 的值为 false
'+' < '*'	false	'+' 的 ASCII 码值是 43，'*' 的 ASCII 码值是 42。因为 43 < 42 的值为 false，所以 '+' < '*' 的值为 false
'6' <= '>'	true	'6' 的 ASCII 码值是 54，'>' 的 ASCII 码值是 62。因为 54 <= 62 的值为 true，所以 '6' <= '>' 的值为 true

如果比较的是不同类型的数据，则可能会产生难以预料的结果。例如，下面表达式将一个整数与一个字符做比较：

```
8 < '5'
```

因为这种比较结果难以预料，所以应该尽量避免在程序的关系表达式中比较不同类型的数据。在某些机器上，上面表达式中的 8 将与 '5' 的编码值 53 做比较。也就是说，8 与 53 做比较。

诸如4 < 6和'R' > 'T'都是逻辑表达式。当C++计算逻辑表达式时，如果计算结果是true，将返回整数值1；否则，将返回整数值0。在C++中，任何非0数值，都被视为true。

注意： 第2章中介绍了bool（布尔）数据类型，bool数据类型只有两个数值：true和false。在C++中，true和false都是保留字。标识符true的值为1，false的值为0。逻辑表达式计算结果为true时，表达式的值实际上是1；逻辑表达式计算结果为false时，表达式的值实际上是0。虽然如此，但出于对程序可读性方面上的考虑，建议读者在逻辑表达式中使用标识符true和false，而不是使用1和0。

4.2.2　关系运算符和string类型

在string类型的变量上也可以进行关系比较运算。在比较string类型变量时，从第一个字符开始，按照字符的ASCII值，逐个字符逐个字符地进行比较。在遇到第一个不匹配字符或是在比较完最后一个字符后两个string变量完全相等的情况下，比较过程才会结束。考虑下面的语句：

```
string str1 = "Hello";
string str2 = "Hi";
string str3 = "Air";
string str4 = "Bill";
string str5 = "Big";
```

应用上述变量定义，表4.3说明了逻辑表达式的计算过程。

表4.3　计算含有string类型变量的逻辑表达式的值

表达式	值
str1 < str2	true
str1 > "Hen"	false
str3 < "An"	true
str1 == "hello "	false
str3 <= str4	true

如果所比较的两个字符串的长度不一样，那么这种逐个字符比较的过程在比较完较短字符串中最后一个字符之后结束（如果前面部分完全相等）。计算结果是短字符串小于长字符串。例如：

表达式	值
str4 >= "Billy"	false
str5 <= "Bigger"	true

4.3　逻辑运算符和逻辑表达式

本节将介绍怎样构造和计算由多个逻辑表达式组成的复杂逻辑表达式。逻辑（布尔）运算符用来连接多个逻辑表达式。C++中有三个逻辑运算符，如表4.4所示。

表4.4　C++中的逻辑运算符

运算符	说明
!	非
&&	与
||	或

逻辑运算符只能有逻辑类型的操作数，并且计算结果也是逻辑类型。运算符!是单目运算符，所以只能有一个操作数。运算符&&和||都是双目运算符。表4.5，表4.6和表4.7分别定义了这些运算符的运算规则。

表4.5定义了运算符!（非）的运算规则。表4.5说明在使用! 运算符时，!true是false，!false是true。置于逻辑表达式前面的!，将会使逻辑表达式的值变为相反的值。

表4.5　!（非）运算符

表达式	!表达式
true (非0)	false (0)
false (0)	true (1)

例4.1

表达式	值	解释
!('A' > 'B')	true	因为'A' > 'B'的值是false，所以!('A' > 'B')的值是true
!(6 <= 7)	false	因为6 <= 7的值是true，所以!(6 <= 7)的值是false

表4.6定义了运算符&&（与）的运算规则。从这张表中可以看出，只有当表达式1和表达式2的值同时为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)	true (非0)	true (1)
true (非0)	false (0)	false (0)
false (0)	true (非0)	false (0)
false (0)	false (0)	false (0)

例4.2

表达式	值	解释
(14 >= 5) && ('A' < 'B')	true	因为(14 >= 5)的值是true，('A' < 'B')的值也是true，true && true的值还是true，所以整个表达式的值也是true
(24 >= 35) && ('A' < 'B')	false	因为(24 >= 35)的值是false，('A' < 'B')的值是true，false && true的值是false，所以整个表达式的值是false

表4.7定义了运算符||（或）的运算规则。从这张表中可以看出，只要表达式1和表达式2中有一个值为true，那么表达式1||表达式2的值就是true；否则，表达式1||表达式2的值是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)	true (非0)	true (1)
true (非0)	false (0)	true (1)
false (0)	true (非0)	true (1)
false (0)	false (0)	false (0)

例4.3

表达式	值	解释
(14 >= 5) || ('A'> 'B')	true	因为(14 >= 5)的值是true，('A' > 'B')的值是false，true || false的值是true，所以整个表达式的值是true
(24 >= 35) || ('A'> 'B')	false	因为(24 >= 35)的值是false，('A' > 'B')的值也是false，false || false的值是false，所以整个表达式的值是false
('A' <= 'a') || (7 != 7)	true	因为('A' <= 'a')的值是true，(7 != 7)的值是false，true || false的值是true，所以整个表达式的值是true

4.3.1　运算符优先级

复杂的逻辑表达式计算起来十分困难。考虑下面的逻辑表达式：

```
11 > 5 || 6 < 15 && 7 >= 8
```

这个逻辑表达式的计算结果取决于是先计算运算符||，还是先计算运算符&&。如果先计算||，整个表达式的值是false；如果先计算&&，整个表达式的值是true。

为了可以计算复杂的逻辑表达式，必须先规定各种运算符的优先级。因为复杂的逻辑表达式中可能会含有算术运算符、关系运算符和逻辑运算符三种运算符，例如表达式5 + 3 <= 9 && 2 > 3，所以必须先规定好所有C++运算符的优先级。表4.8中给出了一些C++运算符的优先级，包括算术运算符、关系运算符和逻辑运算符（如果想知道所有C++运算符的优先级，请参阅附录B）。

表4.8　运算符的优先级

运算符	优先级别
!, +, -（单目运算符）	1
*, /, %	2
+, -	3
<, <=, >=, >	4
==, !=	5
&&	6
||	7
=（赋值运算符）	8

注意： 在C++中，&和|也是运算符。而且，这两个运算符与运算符&&和||的含义大相径庭。如果将&&写成&，则是输入错误，并且会产生非常奇怪的结果。同样，将||写成|，也会产生非常奇怪的结果。

应用表4.8中的运算符优先级规则计算表达式值时，关系运算符和逻辑运算符是从左向右计算的。因为关系运算符和逻辑运算符从左向右计算，所以称这些运算符的结合律是从左向右的。

例4.4说明了由变量组成的逻辑表达式的计算过程。

例4.4　假设变量的定义如下所示：

```
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	解释
!found	false	因为found的值是true，所以!found的值是false
x > 4.0	true	因为x的值是5.2，并且5.2 > 4.0的值是true，所以表达式x > 4.0的值是true
!num	false	因为num的值是1，是非0数值，num的值是true，所以!num的值是false
!found && (x >= 0)	false	在本表达式中，!found的值是false。而且x的值是5.2，并且5.2 >= 0的值是true，所以x >= 0的值是true。因此，表达式!found && (x >= 0)的值等于false && true，也就是false
!(found && (x >= 0))	false	在本表达式中，found && (x >= 0)的值是true && true，也就是true。因此，表达式!(found && (x >= 0))的值是!true，也就是false

x + y <= 20.5	true	因为x + y = 5.2 + 3.4 = 8.6，而且8.6 <= 20.5，所以表达式x + y <= 20.5的值是true
(n >= 0) && (n <= 100)	true	这里n的值是20。因为20 >= 0的值是true，所以n >= 0的值是true。而且，因为20 <= 100的值是true，所以n <= 100的值是true。因此，表达式(n >= 0) && (n <= 100)的值是true && true，也就是true
('A' <= ch && ch <= 'Z')	true	在本表达式中，ch的值是'B'。因为'A' <= 'B'的值是true，所以'A' <= ch的值是true。同样，因为'B' <= 'Z'的值是true，所以ch <= 'Z'的值是true。因此，表达式('A' <= ch && ch <= 'Z')的值是true && true，也就是true
(a + 2 <= b) && !flag	true	这里a + 2 = 5 + 2 = 7并且b等于8。因为7 < 8的值是true，所以表达式a + 2 < b的值是true。同样，因为flag的值是false，所以!flag的值是true。因此，表达式(a + 2 <= b) && !flag的值是true && true，也就是true

正如例4.5中所示，可以编写C++程序来计算例4.4中表达式的值。

例4.5　本例计算并输出例4.4中逻辑表达式的值。注意在本程序的输出结果中，如果逻辑表达式的值是true，输出的结果是1；如果逻辑表达式的值是false，输出的结果是0。记住，如果逻辑表达式的值是true，那么它的结果是1；如果逻辑表达式的值是false，那么它的结果是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;
}
```

输出结果

```
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
```

可以在程序中使用括号来指明运算符的优先级。括号的优先级高于所有运算符的优先级。按运算符优先级规则，表达式：

```
11 > 5 || 6 < 15 && 7 >= 8
```

的值与：

```
11 > 5 || (6 < 15 && 7 >= 8)
```

的值相等。

在第二个表达式中，11 > 5 的值是 true，6 < 15 的值是 true，7 >= 8 的值是 false。用逻辑值代替子表达式后，11 > 5 || (6 < 15 && 7 >= 8)就等于true || (true && false) = true || false = true。因此，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)
```

```
     (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  (因为true && false的结果是false)
=    false
```

因此，最终逻辑表达式的值是 false，也就是 0。

4.3.2 优化（短路）计算

C++采用高效的算法来计算表达式的值。这里，通过下面的语句来说明这种高效算法。

```
1.(x > y) || (x == 5)
2.(a == b) && (x >= 7)
```

在第一条语句中，运算符 || 的两个操作数分别是表达式(x > y)和 (x == 5)。只要(x > y)和 (x == 5)两个表达式中，有一个表达式的值为true，那么整个表达式的值就是true。在这种优化（短路）算法下，计算机从左向右计算表达式的值。只要能够确定表达式的最终计算结果，计算过程就立即结束。例如，在上面第一个表达式中，如果子表达式(x > y) 的值是true，那么整个表达式的值就是true。因为无论是true || true 还是 true ||false，最终结果都是 true。因此，也就无需计算子表达式(x == 5)的值。

同样在第二条语句中，运算符&&的两个操作数分别是表达式(a == b)和 (x >= 7)。假如操作数(a == b)的计算结果是false，那么整个表达式的计算结果就是false。因为无论是false && true还是false && false，最终结果都是 false。

逻辑表达式的优化（短路）计算：计算机从左向右地计算逻辑表达式，只要能够确定整个表达式的最终结果，就立即停止计算过程。

例 4.7　考虑下面的表达式：

```
1.(5 >= 3) || (x == 5)
2.(2 == 3) && (x >= 7)
```

在第一个表达式中，因为(5 >= 3)的值是true，并且逻辑运算符是||，所以整个表达式的值是true。计算机并不去计算子表达式(x == 5)的值。同样在第二个表达式中，因为(2 == 3)的值是 false，并且逻辑运算符是 &&，所以整个表达式的值是 false。计算机并不去计算子表达式(x >= 7)的值。

在C++ 中，可以通过下面两种方式中的任何一种来处理逻辑（布尔）表达式：使用 int 类型的变量或者使用 bool 类型的变量。下面几节将详细介绍这两种方法。

4.3.3　int 数据类型和逻辑（布尔）表达式

早期版本的C++ 不提供内建的逻辑（布尔）数据类型的值，例如：true 和 false。因为逻辑表达式的值不是 1 就是 0，所以逻辑表达式的值存储在 int 类型变量中。因此，可以使用 int 数据类型来存储逻辑表达式的值。

前面已经提过，所有非 0 数值都被视为 true。现在，考虑下面变量定义：

```
int legalAge;
int age;
```

和赋值语句：

```
legalAge = 21;
```

如果将 legalAge 视为逻辑变量，那么经过上面语句赋值以后，legalAge 的值是 true。

再来看赋值语句：

```
legalAge = (age >= 21);
```

如果 age 的值大于或等于 21，那么 legalAge 的值就是 1；如果 age 的值小于 21，那么 legalAge 的值就是 0。

4.3.4　bool 数据类型和逻辑（布尔）表达式

较新版本的C++ 提供内建的bool数据类型，它有两个值：true 和 false。因此，可以在逻辑表达式中使用 bool 数据类型。注意，在 C++ 中，true，false 和 bool 都是保留字。并且，标识符 true 的值是 1，标识符 false 的值是 0。现在，考虑下面的变量定义：

```
bool legalAge;
int age;
```

语句：

```
legalAge = true;
```

将 bool 类型变量 legalAge 的值初始化为 true。再来看赋值语句：

```
legalAge = (age >= 21);
```

如果age的值大于或等于21，那么legalAge的值就是true；如果age的值小于21，那么legalAge的值就是false。例如，如果age的值等于25，那么赋给legalAge的值就是true，也就是1；同样如果age的值等于16，那么赋给legalAge的值就是false，也就是0。

注意： 从前面章节可知，既可以使用int类型变量也可以使用bool类型变量来存储逻辑表达式的值。为了清楚起见，本书中使用bool类型变量存储逻辑表达式的值。

有时，逻辑表达式表现的并不是我们所期望的。例如，假设num是int类型变量。现在要编写一个逻辑表达式，如果num的值在0和10之间，包括0和10，那么该表达式的值是true；否则，该表达式的值是false。下面的表达式，看上去会将num与0和10做比较，并可以得到预想的值。

```
0 <= num <= 10
```

虽然这是一条合法的C++语句，但是并不会得到预想的结果。当变量num的值界于0和10之间时，该表达式的值真的会为true吗？当变量num的值大于10时会怎么样？假设num = 5，则：

```
  0 <= num <= 10
= 0 <= 5 <= 10
= (0 <= 5) <= 10                    (因为关系运算符从左向右计算)
= 1 <= 10                           (因为0 <= 5的值是true，也就是1)
= 1 (true)
```

现在假设num = 20，则：

```
  0 <= num <= 10
= 0 <= 20 <= 10
= (0 <= 20) <= 10                   (因为关系运算符从左向右计算)
= 1 <= 10                           (因为0 <= 20的值是true，也就是1)
= 1 (true)
```

很明显，这并不是我们期待的结果。因为num = 20，并不界于0和10之间，所以表达式0 <= 20 <= 10的值不应该是true。注意，无论num的值是多少，该表达式的值都是true。这是因为子表达式0 <= num的值不是0就是1，0 <= 10的值是true，1 <= 10的值也是true。那么，表达式0 <= num <= 10究竟出了什么问题呢？是缺少逻辑运算符&&，正确的C++语句应该是：

```
0 <= num && num <= 10
```

编写逻辑表达式时应该格外谨慎。当编写复杂的逻辑表达式时，必须正确使用逻辑运算符。

4.4 选择：if和if…else

虽然只有两个逻辑值，true和false，但是它们在程序中的用处却很大。使用这两个逻辑值，可以使程序能够做出判断选择，进而可以改变程序的执行过程。本章中的剩余部分，将着重介绍怎样在程序中使用选择语句。在C++中，有两种选择语句，或者称为分支控制结构：if语句和switch结构。本节将主要介绍怎样使用if和if…else语句在程序中建立单路、双路和多路选择。switch结构将放在本章中的后面介绍。

4.4.1 单路选择

当储户的活期储蓄账户余额低于一定数量时，银行将给它们发送通知。也就是说，如果活期储蓄账户余额低于某个最少余额时，银行将给储户发送通知；否则，银行不会采取任何行动。同样，如果某个险种的投保人不吸烟，那么保险公司将在保险费上给他10%的优惠。这两种情况都涉及到单路选择。在C++中，可以使用if语句构造单路选择。构造单路选择的语法是：

```
if(expression)
   statement
```

注意该语法的组成元素：以保留字if开头，接下来的表达式用括号括起来，最后是执行语句。if后面的表达式有时候被称为决策表达式（Decision Maker）或条件表达式，因为它决定是否执行if中的语句。该表达式通常是逻辑表达式，如果该表达式的值是true，就执行表达式后面的语句；如果值是false，计算机就将跳过表达式后面的语句，而去执行下面的语句。表达式后面的语句有时候被称为动作语句（Action Statement）。图4.2说明了if语句（单路）的执行过程。

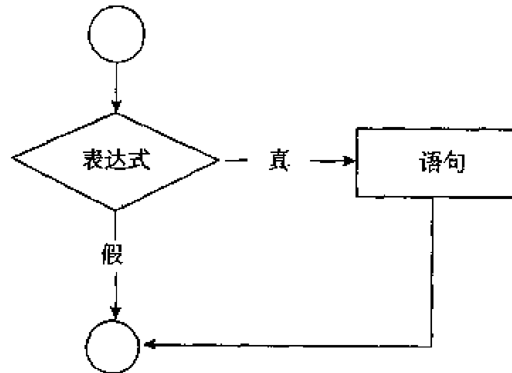


图4.2　单路选择

例4.8

```
if(score >= 90)
   grade = 'A';
```

在上述代码中，如果表达式(score >= 90)的值是真（true），则执行赋值语句“grade = 'A';”。如果表达式的值是假（false），则执行if结构后面的语句（如果存在）。例如，如果变量score的值是95，赋给变量grade的值是A。

例4.9　下面的C++程序用来计算整数的绝对值。

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

int main ()
{
  int number;

  cout<<"Please enter an integer--> ";          //Line 1
  cin>>number;                                   //Line 2
  if(number < 0)                                 //Line 3
    number = -number;                            //Line 4

  cout<<endl<<"The absolute value is "
      <<number<<endl;                            //Line 5
  return 0;
}
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
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行语句将输入的整数读入到变量number中。第3行语句检查number是否是负数。如果number是负数，第4行语句将number转变成正数。第5行语句输出number，这时number中的值就是原来的值的绝对值。

例4.10 考虑下面语句：

```
if score >= 90          //syntax error
      grade = 'A';
```

这条语句说明了一种错误使用if语句的情况。漏掉逻辑表达式的括号，将导致语法错误。

将分号置于if语句表达式的右括号后面（也就是在语句之前），将导致语义错误。如果分号紧跟在表达式的右括号后面，if语句将会执行一条空语句。

例4.11 考虑下面的C++语句：

```
if(score >= 90);        //Line 1
      grade = 'A';      //Line 2
```

这条语句看上去很像是单路选择语句。但是由于表达式后面有一个分号（见Line 1），if语句在遇到该分号时就已经终止。if结构中的动作语句将是空操作，而且第2行语句根本就不是if语句的一部分。因此，无论表达式的值是什么，第2行语句总要执行。

4.4.2 双路选择

前面一节介绍了怎样在程序中实现单路选择。但是在另一些情况下，程序需要在两段被选代码中选择执行。例如，如果兼职员工加班工作，那么将使用加班工资计算公式来计算月工资单；否则，将按照正常公式来计算月工资单。这是一个双路选择的例子。C++提供了if…else语句，使程序能够在两段被选代码中选择执行，也就是实现双路选择。双路选择的语法如下：

```
if(expression)
   statement1
else
   statement2
```

现在让我们花一些时间来仔细研究这个语法。它以保留字if开头，后面跟着用括号括起来的表达式，接着是语句1（statement1），然后是保留字else，最后是语句2（statement 2）。语句1和语句2可以是任意合法的C++代码。在双路选择中，如果表达式的值是真（true），则执行语句1；如果表达式的值是假（false），则执行语句2。图4.3说明了if…else语句的执行过程（双路选择）。

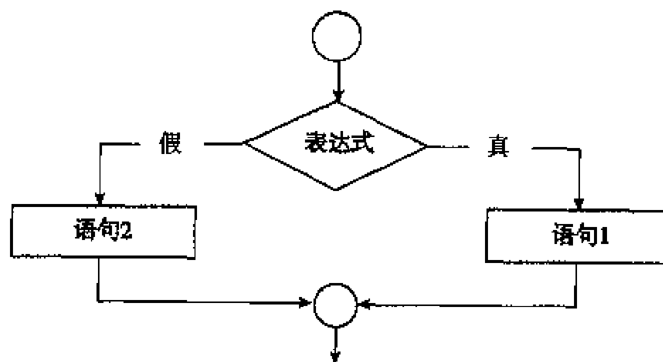


图4.3 双路选择

例4.12　考虑下面语句：

```
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
```

如果工作小时（hours）的值大于40.0，则工资（wages）中将包括加班费。假设hours是50。第1行表达式的值是true，所以程序执行第2行中的语句。在另一种情况下，假设hours的值是30，或者是任何小于等于40的数，第1行表达式的值则是false。这时，程序将略过第2行中的语句，转去执行第4行中的语句，也就是执行保留字else后面的语句。

在双路选择语句中，将分号置于表达式后面和语句1前面将会导致语法错误。如果if语句以分号结束，语句1也就不再是if语句的一部分，并且else语句也将成为单独的语句。在C++中，没有单独的else语句。也就是说，不能将其从if语句中分离出来。

例4.13　下面语句是一个语法错误语句的例子：

```
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if语句中表达式的右括号后面有一个分号（第1行），所以else成为一条单独的语句。if语句（第1行）后面的分号结束了if语句，所以第2行语句将else从句从整个if语句中分离出来。也就是说，else成了独立的语句。由于C++中并不存在else语句，所以该代码出现语法错误。

例4.14　下面的程序用来计算员工的周工资。如果工作时间（hours）超过40小时，工资（wages）中将包括加班费。

```
//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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
Line 2: Enter working hours and rate: 56.45 12.50

Line 9: The wages are $808.44
```

第1行语句将输出的浮点数设置为如下格式：固定小数点、小数点后面两位小数、必须带有小数点和小数部分，不足两位小数的添0。第2行语句提示用户输入工作的小时数和工资率。第3行语句将用户输入的数据分别读入到变量hours和rate中。第4行语句检查变量hours中的值是否大于40.0。如果变量hours中的值大于40.0，则执行第5行语句来计算工资，其中包括加班费；否则，执行第7行语句来计算工资。第9行语句输出计算出来的工资。

让我们再看一些if语句的例子，从中发现可能出现的语义错误。

例4.15 考虑下面的语句：

```
if(score >= 90)
      grade = 'A';
      cout << "The grade is " << grade << endl;
```

上述语句中含有语义错误。if语句只对第一个语句“grade = 'A';”起作用。无论表达式(score >= 90)是true还是false，cout语句都会执行。

例4.16中说明了另一种常犯的错误。

例4.16 考虑下面语句：

```
if(score >= 60)
      cout << "Passing" << endl;
      cout << "Failing" << endl;
```

如果表达式(score >= 60)的值是false，输出的结果将是Failing。也就是说在这种情况下，这套语句的作用与含有else的if语句相同。程序只执行if后面的第二条语句。例如，如果score的值是50，这些语句输出的结果是：

```
Failing
```

然而，如果表达式score >= 60的值是true，程序将既输出Passing，又输出Failing，这种结果显然是不能令人满意的。例如，如果score的值是70，则会输出如下结果：

```
Passing
Failing
```

下面是正确的代码，程序根据score的值只输出一个结果，Passing或者Failing。

```
if(score >= 60)
      cout << "Passing" << endl;
else
      cout << "Failing" << endl;
```

4.4.3 复合（块）语句

if语句和if…else语句一次仅能控制一条语句。然而，有时需要在if后面表达式的值是true的情况下，执行多于一条的语句。C++提供了一种称为复合语句或块语句的语句结构，来支持一次执行多条语句。复合语句的形式如下所示：

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

也就是说，复合语句由括在一对大括号中的语句序列组成。在一个if或者if…else语句中，复合语句的使用方法与单独一条语句的使用方法没有什么不同。因此，可以根据简单的双路选择语句代码，如：

```
if(age > 18)
      cout<<"Eligible to vote."<<endl;
else
      cout<<"Not eligible to vote."<<endl;
```

写出复杂一些的语句代码，如：

```
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;
}
```

复合语句十分有用，本章后面的许多程序将使用到这种复合语句。

4.4.4　多重选择：嵌套if

前面章节中已经介绍了怎样在程序中实现单路选择和双路选择。然而在有些时候，会出现需要在两种以上被选答案中做出选择的情况。例如，假设活期账户余额多于$50 000时，利息率是7%；余额在$25 000和$49 999.99之间时，利息率是5%；余额在$1 000和$24 999.99之间时，利息率是3%；否则，利息率是0%。对于这个特定问题，有4种备选可能——也就是说，有4条选择路径。通过在程序中使用多个if…else语句，可以实现多重选择。也就是说，动作语句本身也将是if语句或者if…else语句。一条控制语句出现在另一条控制语句中的语句结构，称为嵌套（Nested）语句。

假设所有变量的定义都是正确的，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
```

这些语句说明了怎样使用嵌套 if…else 语句结构在程序中实现多重选择。

要想正确使用嵌套 if…else 结构，必须首先清楚一个重要问题：程序怎样知道每个 else 与哪个 if 匹配？注意，在 C++ 中，没有单独的 else 语句，每个 else 语句必须有一个相应的 if 语句与之相匹配。else 语句和 if 语句的匹配规则如下。

else 语句和 if 语句的匹配规则 在嵌套的 if 语句结构中，C++ 将 else 语句与上面最近的未与 else 语句匹配的 if 语句相匹配。

在前面的代码中，应用上述匹配规则，则有：第 3 行中的 else 语句与第 1 行中的 if 语句相匹配；第 6 行中的 else 语句与第 4 行中的 if 语句相匹配；第 9 行中的 else 语句与第 7 行中的 if 语句相匹配；第 12 行中的 else 语句与第 10 行中的 if 语句相匹配。

为了避免过多的行缩进，一些程序员喜欢按下面风格编写代码：

```
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;
```

下面例子会帮助你了解怎样使用嵌套 if 结构来实现多重选择。

例 4.17 假设所有变量的定义都是正确的，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
```

在本 C++ 代码中，第 4 行中的 else 语句与第 2 行中的 if 语句相匹配；第 6 行中的 else 语句与第 1 行中的 if 语句相匹配。注意，第 4 行中的 else 语句不能与第 1 行中的 if 语句相匹配。如果第 4 行中的 else 语句与第 1 行中的 if 语句相匹配，那么第 2 行中的 if 语句就成为第 1 行中 if 语句的动作语句部分，而第 6 行中的 else 语句将独立出来。注意在本代码中，第 2 行至第 5 行语句都是第 1 行中 if 语句条件为 true 的动作语句部分。

例 4.18 假设所有变量的定义都是正确的，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
```

在本 C++ 代码中，第 4 行中的 else 语句与第 2 行中的 if 语句相匹配。注意，对于第 4 行中的 else 语句来说，最近的未匹配 else 的 if 语句是第 2 行中的 if 语句。在本代码中，第 1 行中的 if 语句没有 else，是单路选择语句。

例 4.19　假设所有变量的定义都是正确的，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
```

这段代码十分拙劣。根据 else 与 if 的匹配规则，第 4 行中的 else 语句与第 2 行中的 if 语句相匹配。然而，这种匹配并不是程序实际所需要的。假设GPA是3.8。第1行if语句中表达式的值是true，所以执行该 if 的动作语句部分。因为 GPA 是 3.8，第二个 if 语句中表达式的值是 false，所以执行该 if 语句的 else 部分，结果如下所示：

```
Current GPA below graduation requirement.
See your academic advisor.
```

然而，如果一个学生的GPA是3.8，那么他不仅可以毕业，而且学习成绩还不算低。实际上，当GPA是 3.8 时，程序什么信息也不应该输出。当 GPA 小于 2.0 时，程序应该输出下面的信息：

```
Current GPA below graduation requirement.
See your academic advisor.
```

当 GPA 大于等于 3.9 时，程序应该输出下面的信息：

```
Dean's Honor List.
```

为了正确地完成功能，第 4 行中的 else 语句必须与第 1 行中的 if 语句相匹配。这时，需要使用如下的复合语句：

```
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
```

在本例题的这种情况，总的规则是块（也就是在大括号）中 if 不能与块外 else 语句相匹配。第 4 行中的 else 语句不能与第 2 行中的 if 语句相匹配，因为第 2 行中的 if 语句被括在大括号中，所以第 4 行中的 else 语句不能到大括号里去找与之相匹配的 if 语句。因此，第 4 行中的 else 语句与第 1 行中的 if 语句相匹配。

4.4.5　if…else 语句和系列 if 语句的比较

考虑下面两个 C++ 代码段，这两个代码段的作用是相同的：

(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;
```

代码段(a)由if···else语句组成，代码段(b)由一系列if语句组成。这两个代码段的功能都是相同的。如果month的值是3，那么这两个代码段都将输出March。如果month的值是1，则在代码段(a)中，第1行if语句中表达式的值是true，与之对应的第2行语句被执行。而该if结构的余下部分，也就是else语句部分，将被跳过。所以，计算机并不去计算其他if语句中表达式的值。但是在代码段(b)中，计算机必须计算每个if语句中表达式的值，因为这些if语句并没有else语句。因此，代码段(b)的执行速度比代码段(a)慢。

4.4.6 使用伪代码进行程序开发、测试和调试程序

开发程序的方法有很多。其中一种方法就是混合使用C++语言和人类自然语言来描述程序，这种混合使用C++语言和人类自然语言的代码被称为伪代码（Pseudocode或者Pseudo）。在编写正式的C++代码前，伪代码在描述程序轮廓和精练程序上有着十分重要的作用。特别是当构建涉及复杂嵌套结构的程序代码时，使用伪代码可以很快地确定出程序的正确结构，从而避免出现常见错误。

现在，让我们看一个十分有用的代码段——判断两个整数中的最大值。当x和y都是整数时，可以很快地写出下面的伪代码段：

```
a. if (x > y) then
       x is larger
b. if (y > x) then
       y is larger
```

如果(a)中的表达式的值是true，那么x是最大值。如果(b)中的表达式的值是true，那么y是最大值。但是，使用这两个语句来判断两个整数的最大值时，计算机需要计算两个表达式的值：

`(x > y)`和`(y > x)`

即使第一个表达式的值是true。必须计算两个表达式的值，有时会浪费计算机时间。让我们重写这段伪代码：

```
if (x > y) then
    x is larger
else
    y is larger
```

在这段代码中，只需要计算一个表达式的值。这段代码看上去很好，让我们将其改写成相应的C++代码

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

int main()
{
      if (x > y)
```

在将伪代码程序转换成C++程序时，会立即意识到程序中并没有存放x和y值的变量。这些变量没有经过定义，特别是对程序设计的初学者来说，这是一个很常见的错误。在查看伪代码后，发现该程序中一共需要三个变量。这里使用能表示自身含意的标识符名称。

```
#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;
}
```

编译上述代码将会产生常见的语法错误（见第2行）。注意，分号不能出现在if…else语句的条件表达式后面。然而，即使在改正这个错误后，程序仍然不能正确运行，因为变量中并没有值。程序中的变量没有经过初始化，这是另外一个常见错误。而且，程序中没有输出语句，所以也不会看到程序的运行结果。

因为程序中有很多错误，应该对整个程序进行代码走查。应该尽可能地选用多个试验数据，验证程序在各种不同情况下运行结果是否正确。例如，在一个变量值是0的情况下，或是在一个正数、一个负数的情况下，或是在两个都是负数的情况下，或是在两个数相等的情况下，程序是否都能够正常运行？仔细查看上面的代码，会发现程序根本没有考虑到两个变量相等的情况。综合考虑上述问题，改写后的程序代码如下所示：

```
//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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
Enter any two integers: 78 90
The two integers entered are 78 and 90
The larger number is 90
```

在上面的程序代码编写过程中不难看出，在计算机上键入程序前，最好先用笔和纸写出代码。虽然这种用笔和纸写出的最初代码，可能并不能真正成功地在计算机上运行，但是这是编写程序的良好开端。因为在纸面上更容易找出错误并易于改进程序，特别是对大型程序来说更是如此。

4.4.7 输入失败和if语句

第3章中已经介绍过，当计算机试图读入非法的输入数据时，输入流将进入到错误状态中。一旦输入流进入错误状态，所有后续的使用到该输入流的输入语句将不会被执行。而计算机将继续执行下面的语句，这将会导致错误的运行结果。如果使用if语句检查输入流变量状态，就可以在输入流进入错误状态时，执行相应的指令终止程序执行。

除了试图读入非法数据以外，其他事件也可以使输入流进入到错误状态中。另两个常见的导致输入失败的原因是：

- 试图打开一个不存在的输入文件
- 试图在输入文件结束标志后面继续读取数据

一种解决输入失败的方法是检查输入流变量的状态。可以在if语句的逻辑表达式中检查输入流变量状态。当在if语句的逻辑表达式中检查输入流变量时，如果输入成功，那么表达式的值将是true；如果输入失败，那么表达式的值将是false。

下面语句：

```
if(cin)
  cout << "Input is OK." << endl;
```

当从标准输入设备成功地输入数据时输出：

```
Input is OK.
```

同样，如果infile是一个ifstream变量，则语句：

```
if(!infile)
  cout << "Input failed." << endl;
```

当从输入文件流变量infile输入数据失败时输出：

```
Input failed.
```

假设输入流变量试图打开一个文件，然后读入数据到程序中。如果输入文件并不存在，可以联合使用输入流变量的值和return语句来终止程序运行。

注意，函数main中的最后一条语句是：

```
return 0;
```

这条语句的作用是在程序运行结束时，向操作系统返回数值0。返回值0意味着程序正常终止，并且在执行过程中没有出现异常。return语句还可以向操作系统返回除0以外的其他int类型数值。然而，如果return语句返回0以外的其他数值，就意味着程序执行过程中出现了某种异常。

return语句可以出现在程序中的任何位置。当执行到return语句时，程序立即从该return语句出现的函数中跳出来。对于函数main来说，当执行到return语句时，程序立即终止。所以当输入流出现错误时，可以利用return语句这个特性来终止执行main函数。这个技巧在程序打开输入文件时显得格外有用。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并不存在。打开该文件的操作将会失败，导致输入流进入错误状态。作为逻辑表达式，文件流变量infile的值是false。由于infile的值是false，if语句中条件表达式!infile的值是true，所以执行if的动作语句。该语句向屏幕输出信息：

```
Cannot open input file. The program terminates.
```

return语句终止程序执行，并且向操作系统返回数值1。

注意：*某些系统需要在open函数中使用第二个参数ios::nocreate，指定在打开的文件不存在时，使open函数出错（而不是创建新文件）。在这种情况下，open函数应该写成：*

```
infile.open("a:inputdat.dat", ios::nocreate);
```

现在，让我们使用输入失败处理技术来改写第3章程序范例中的程序“学生成绩”。注意该程序使用输入文件中的数据来计算平均考试成绩，并将计算结果输出到另一个文件中。除了增添了在输入文件不存在情况下终止程序执行的语句外，程序中的其他代码与第3章中的程序代码相同。

```
//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：将其改写成为标准C++格式的程序，需要将下面语句：

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

改写为：

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

此外，还可能需要通过使用函数setf或者控制符setiosflags来设置控制符fixed和showpoint（详细信息，请参阅第3章中的相关部分）。

4.4.8 恒等运算符（==）和赋值运算符（=）之间容易产生的混淆

注意，当if语句中决定条件表达式的值是true时，执行if语句中的动作语句。并且，条件表达式通常是逻辑表达式。但是，C++允许在if条件表达式中使用任何类型的表达式，只要该表达式的值可以计算为true或false。考虑下列语句：

```
if (x = 5)
    cout << "The value is five." << endl;
```

该if语句的条件表达式是x = 5。因为表达式中有赋值运算符=并且在表达式后面没有分号，所以表达式x = 5被称为赋值表达式。

该表达式的计算过程如下。首先计算=右边的表达式的值，计算结果是5。然后，数值5赋给变量x。同时，数值5也就是整个表达式的值，即赋值表达式的值。因为5为非零值，所以该表达式的值为true，if结构的动作语句输出：

```
The value is five.
```

不论经验有多么丰富，几乎所有的程序员都会将=误用为==。容易误用的原因之一是，许多其他的程序设计语言将=用做恒等关系运算符。这样一来，使用过其他程序设计语言的程序员就容易将=误用为==。容易误用的另一个原因就是粗心，即在键入程序代码时误写。

尽管赋值表达式可以用在恒等表达式出现的地方而不产生错误，但是这样做通常会产生很多严重问题。例如，假设汽车保险费折扣基于驾驶员的驾驶记录。驾驶记录的值为1，表示该驾驶员驾车时从未出现过事故，因而可以获得25%的折扣。语句：

```
if (drivingcode == 1)
    cout << "The discount on the policy is 25%." << endl;
```

只有当drivingcode的值是1时，才会输出：

```
The discount on the policy is 25%.
```

而语句：

```
if (drivingcode = 1)
    cout << "The discount on the policy is 25%." << endl;
```

不管drivingcode的值是多少，总会输出：

```
The discount on the policy is 25%.
```

因为赋值号右边的表达式的值是1，是非零值。因此该if语句中表达式的值是true，输出如下一行文字："The discount on the policy is 25%."。并且变量drivingcode中的值也变为1。假设在if语句执行以前，变量drivingcode中的值是4，那么在if语句执行以后，不仅输出是错的，而且变量drivingcode中的值也将变为1。

将==误写成=的危害很大。因为它并不会导致语法错误，所以编译器也不会发出警告信息。但是，它确实是一个逻辑错误。

注意：将==误写成=还会导致一系列严重的错误，特别是在循环语句中。第5章将讨论循环结构。

如果在程序中将赋值运算符=误写成恒等运算符==，也会导致错误。例如，假设x，y和z都是int类型变量。语句：

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

将表达式y + z的值赋给变量x。而语句：

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

将表达式y + z的值与变量x的值做比较，并且变量x中原有的值不变。但是，如果在程序中的其他部分用到了变量x的值（本来是想让变量x的值等于表达式y + z的值），则会发生逻辑错误，并且编译器不会发出任何警告信息。编译器报告语法错误，而不会警告任何逻辑错误。因此，在使用赋值运算符和恒等运算符时应该格外小心。

4.4.9 条件运算符（?:）

注意：读者可以跳过本节内容，并对后续学习不会有任何影响。

应用C++条件运算符可以将某些if…else语句改写成更精练的形式。条件运算符，记为“?:”，是三目运算符，也就是说有三个操作数。条件运算符的使用语法是：

```
expression1 ? expression2 : expression3
```

这种结构的语句称为条件表达式。条件表达式的计算规则如下：如果expression1的值是非零数（也就是true），那么整个条件表达式的值就是expression2的值；否则，整个条件表达式的值就是expression3的值。

考虑下列语句：

```
if (a >= b)
    max = a;
else
    max = b;
```

可以使用条件运算符将这个if…else语句改写成下面形式：

```
max = (a >=b) ? a : b;
```

4.5 switch结构

前面已经讲过，C++中有两种选择结构，或者分支结构。第一种选择结构，是通过if或者if…else语句来实现的，并且通常要计算逻辑表达式的值。第二种选择结构，并不需要计算逻辑表达式的值，被称为switch结构。C++的switch结构使计算机能够在多个分支中选择执行。

使用switch语句的语法是：

```
switch(expression)
{
case value1: statements1
             break;
case value2: statements2
             break;
        ...
case valuen: statementsn
             break;
default: statements
}
```

在C++中，switch，case，break和default都是保留字。在switch结构中，首先计算表达式（expression）的值。然后根据表达式的值，执行相应的关键字case后面的语句指定的动作。注意，在语法模板中，加有阴影的部分是可选的。

虽然并不强制要求如此，但是表达式通常只是一个标识符。但是，无论是表达式还是标识符，它们的值只能是整型数。switch结构根据表达式的值决定执行哪条语句。每个case后面的值，在整个switch结构中只能出现一次。一个case标号后面可以跟一条或者多条语句，而且不需要使用花括号将它们括起来使其成为复合语句。每个case语句的后面不必跟有break语句。图4.4说明了switch语句的执行过程。

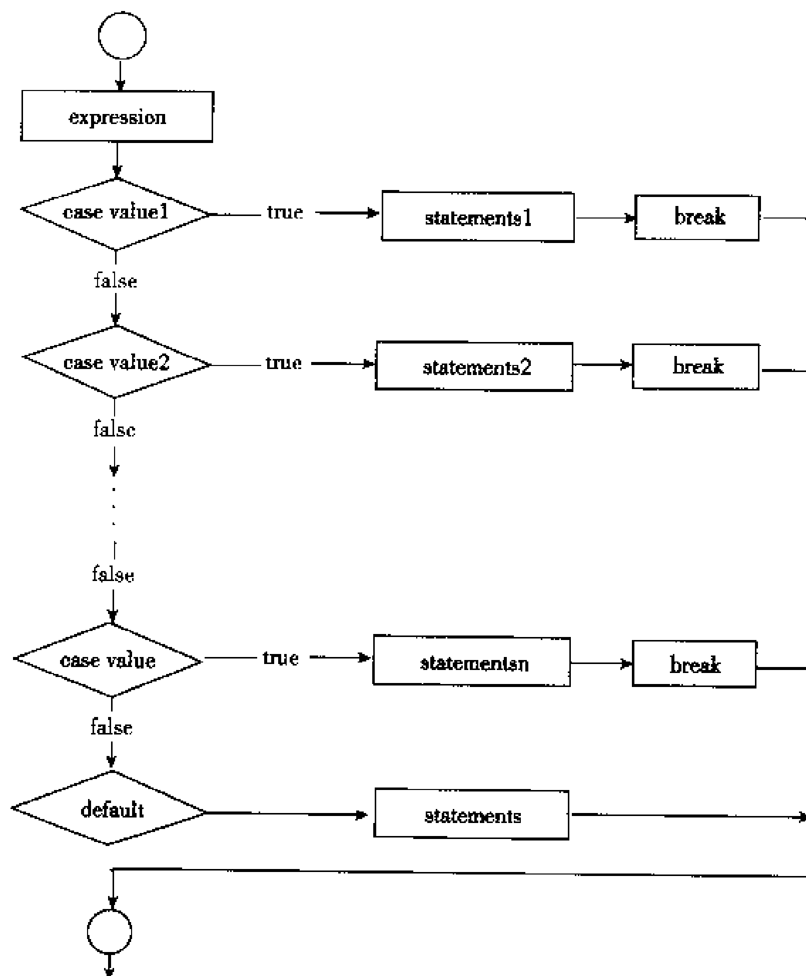


图4.4 switch语句

switch语句的执行规则如下所示：

1. 当表达式expression的值与某个case值（也称为标号）相匹配，则从该case后面的语句开始执行，直到遇到break语句或是整个switch结构执行完毕。
2. 如果表达式expression的值不与任何case值相匹配，则执行default后面的语句。如果switch结构中没有default标号，而且表达式expression的值不与任何case值相匹配，那么程序将跳过整个switch结构。
3. break语句可以使程序立即跳出switch结构。

例4.20 考虑下列语句：

```
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.";
}
```

在本例中，switch语句中的表达式是一个变量标识符。grade是char类型变量，所以也是int类型数值。变量grade的所有可能取值是'A', 'B', 'C', 'D'和'F'。根据grade的值，每一个case标签后面对应一个动作语句。如果grade的值是'A'，输出的结果是：

```
The grade is A.
```

例4.21 下列程序说明了break语句的作用。程序要求用户输入0到10中间的一个整数。

```
//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;
}
```

程序运行结果

这些输出结果是通过多次运行上面程序而得到的。在每个程序运行结果中，用户输入的数据都加有阴影。

程序运行结果 1

```
Enter an integer between 0 and 10: 0

The number you entered is 0
Hello there. I am Mickey.
Out of switch structure.
```

程序运行结果 2

```
Enter an integer between 0 and 10: 1

The number you entered is 1
Hello there. I am Mickey.
Out of switch structure.
```

程序运行结果 3

```
Enter an integer between 0 and 10: 3

The number you entered is 3
I am Mickey.
Out of switch structure.
```

程序运行结果 4

```
Enter an integer between 0 and 10: 4

The number you entered is 4
Mickey.
Out of switch structure.
```

程序运行结果 5

```
Enter an integer between 0 and 10: 5

The number you entered is 5
How are you?
Out of switch structure.
```

程序运行结果 6

```
Enter an integer between 0 and 10: 7

The number you entered is 7
are you?
Out of switch structure.
```

程序运行结果 7

```
Enter an integer between 0 and 10: 9

The number you entered is 9
Out of switch structure.
```

程序运行结果 8

```
Enter an integer between 0 and 10: 10

The number you entered is 10
Have a nice day.
Out of switch structure.
```

程序运行结果 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.
```

通过在表达式中使用不同的变量a的值在程序中进行代码走查，有助于观察break语句的作用。如果a的值是0，那么switch语句中表达式的值与case值0相匹配。“case 0:”后面的所有语句都将被执行，直到遇到break语句为止。

break语句首次出现在第10行，正好在“case 5:”的前面。虽然switch表达式的值与case值1，2，3，4都不匹配，但是这些case后面的语句都将被执行。

当switch语句中表达式的值与一个case值相匹配时，所有的后续语句都将被执行，直到遇到break语句为止。同样，如果a的值是3，它将与case值3相匹配，所有后续语句都将被执行，直到遇到第10行中的break语句为止。如果a的值是9，它将与case值9相匹配。在这种情况下，没有动作语句，因为在第16行中只有break语句跟在case值9的后面。

例4.22　虽然switch语句中的case值（标号）只能是整数，但是switch语句中的表达式可以写得很复杂。例如，考虑下列的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;
}
```

假设score是整型变量，并且取值空间在0和100之间。如果score的值是75，则score / 10 = 75 / 10 = 7，所以grade的值是'C'。如果score的值在0和59之间，那么grade的值将是'F'。因为score的值在0和59之间，score / 10的值将是0，1，2，3，4或者5，这些值对应的grade都是'F'。因此，在该switch结构中，case 0，case 1，case 2，case 3，case 4和case 5对应相同的动作语句。除了可以在每个case值0，1，2，3，4和5后面各使用一个“grade = 'F';”语句和break语句外，还可以通过将所有case集中写在一块，然后加上共同的动作语句的方法来简化程序代码(上面的代码中使用的就是这种简化表达方法)。case值9和10，采用了同样的简化方法。

switch语句中的表达式除了可以是标识符和复杂表达式以外，还可以是逻辑表达式。考虑下面语句：

```
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;
}
```

如果变量 age 的值是 25，表达式 age >= 18 的值是 1，也就是 true。因为表达式的值是 1，所以执行 case 1 后面的语句。如果变量 age 的值是 14，表达式 age >= 18 的值是 0，也就是 false，所以执行 case 0 后面的语句。

也可以在 case 中使用 true 和 false 分别代替 1 和 0，改写后的代码如下所示：

```
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;
}
```

正如在上面的例题中见到的，switch语句是解决多重选择的有效方法。在本章后面的程序范例中，还可以见到switch语句的使用实例。虽然没有固定的准则来评判在实现多重选择时if…else结构和switch结构孰优孰劣，但是仍然还有一些标准可以参考。如果多重选择涉及到的数据范围较大，那么既可以使用if…else结构也可以使用switch结构。但是，在使用switch结构时，需要将数据范围转换成有限的整数集。

例如在例4.22 中，变量 grade 的值取决于变量 score 的值。如果 score 的值在 0 和 59 之间，grade 的值就是 'F'。因为 score 是 int 类型变量，所以共有 60 个数值对应着 grade 的值 'F'。如果将这 60 个值全部列在 case 中，那么整个 switch 结构就会变得很长。然而，通过将变量 score 除以 10，将这 60 个值转化为 6 个值：0，1，2，3，4 和 5。

如果数据范围是无限集，并且不能将其转换成为有限的整数集，那么就只能使用 if…else 结构。例如，如果 score 是 double 类型变量，那么 0 到 60 之间的数值集将是无限集。但是在这种情况下，仍然可以使用表达式 static_cast<int>(score) / 10 将无限集转换成只有 6 个整数的有限集。

4.6　使用 assert 函数终止程序

程序中很可能会出现一些很难捕捉到的错误。例如，使用前面介绍过的程序设计技巧就很难捕捉到除数为0的错误。C++中包含了一个预定义函数assert，该函数的作用是在发生某种错误时终止程序的执行。在发生除数为0错误的情况下，可以使用assert函数来终止程序。assert函数可以将发生的错误类型和位置信息报告给用户。

考虑下列语句：

```
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')
```

在语句 1 中，如果变量 denominator 的值是 0，逻辑上不应该进行除法运算。但是在程序执行中，计算机将试图计算该表达式的值。如果denominator的值是0，程序将终止运行，并且提示发生了非法操作。

语句2的作用是在hours大于0并且rate大于0小于等于15.50时，计算wages的值。语句3的作用是在ch是大写字母时执行某种操作。

在执行上面各条语句时，必须满足某种条件。如果条件不满足，应该立即终止执行程序，并指出程序中的什么地方发生了何种错误。可以在程序中使用输出语句和return语句来处理上述情况。但是，C++还提供了一种更加有效的函数——assert函数，来处理某种条件不被满足的情况。

使用assert函数的语法如下：

```
assert(expression);
```

这里的表达式expression是任意的逻辑表达式。如果表达式的值是true，则执行下面的语句；如果表达式的值是false，则终止程序执行并提示程序中的哪个位置上出现了错误。

函数assert定义在头文件cassert中。因此，如果想使用assert函数，就必须在程序中包含：

```
#include <cassert>
```

注意： *在标准C++中，包含函数assert的头文件的名字是assert.h。*

assert函数所在的语句通常被称为assert（断言）语句。

现在再来看前面的代码，可以使用assert函数改写语句1（quotient = numerator / denominator;）。因为只有在denominator非零的情况下，才可以计算quotient的值，所以应该在赋值语句前使用assert语句：

```
assert(denominator);
quotient = numerator / denominator;
```

现在，如果变量denominator的值是0，assert语句将终止程序，并且提示类似如下的信息：

```
Assertion failed: denominator, file c:\temp\assert
function\assertfunction.cpp, line 20
```

该信息表明denominator发生了错误。提示信息中还包括了出现错误的源代码所在文件的文件名及出错代码的位置。

还可以使用函数assert改写语句2：

```
assert(hours > 0 && (0 < rate && rate <=15.50));
if(hours > 0 && (0 < rate && rate <=15.50))
      wages = rate * hours;
```

如果该assert语句中的表达式的值是false，程序将被终止，并且提示类似如下的信息：

```
Assertion failed: hours > 0 && (0 < rate && rate <=15.50), file
c:\temp\assertfunction\assertfunction.cpp, line 26
```

在程序开发和测试过程中，assert语句的使用对提高代码质量起到很大作用。正如上面所见，assert语句不仅可以终止程序执行，还可以指出发生错误的表达式、包含错误源代码的文件名和发生错误的行号。

虽然assert语句在程序开发中有很重要的作用，但是在程序开发结束并且投入使用后，如果由于某种原因仍然出现assert错误，用户将不知道该怎么办。因此，在程序开发和测试结束后，应该删除或取消所有的assert语句。但是，在大型程序中，删除所有开发中使用的assert语句不仅费力而且有时几乎是不可能的。而且，如果程序还需要修改，则应该保留所有的assert语句。所以较好的做法是保留并取消掉所有assert语句，而不是将它们删除。可以使用下面的预处理指令来取消掉程序中所有的assert语句：

```
#define NDEBUG
```

预处理指令#define NDEBUG必须置于预处理指令#include <cassert>之前。

4.7　程序范例：有线电视公司的计费程序

本程序用于计算某地有线电视公司的客户账单。客户分为两种类型：家庭客户和商业客户。有线电视收费标准也有两种：家庭客户收费标准和商业客户收费标准。对于家庭客户，收费标准如下所示：

- 账单处理费：$4.50
- 基本服务费：$20.50
- 频道租用费：$7.50/频道

对于商业客户，收费标准如下所示：

- 账单处理费：$15.00
- 基本服务费：前10个节点$75.00，以后每个节点$5.00
- 频道租用费：$50.00/频道（节点数目不限）

程序要求用户输入账号（整数）和客户代码。假设R或者r代表家庭客户，B或者b代表商业客户。

输入　客户账号、客户代码、用户租用的频道数量。如果是商业客户，还要提供接入的节点数目。
输出　客户账号和账单总额。

问题分析和算法设计

程序的功能是计算和打印账单。为了计算账单总额，需要知道该账单所对应的客户（客户账号）、客户类别（客户代码，家庭还是商业）和租用的频道数量。对于商业用户来说，还需要知道接入的节点数目。除此之外，计算账单所需要的其他数据，如账单处理费和每个频道的租用费都是固定不变的。账单中的数字应该保留两位有效数字，这也是金融处理上的通常要求。经过上面的分析，设计算法如下所示：

1. 将数字精度设为两位有效数字。
2. 提示用户输入账号和客户类别。
3. 根据客户类别、租用的频道数量、基本服务费计算客户账单：
   a. 如果客户类别是R或者r：
      (1) 提示用户输入租用频道的数量
      (2) 计算账单
      (3) 打印账单
   b. 如果客户类别是B或者b：
      (1) 提示用户输入接入的节点数目和租用频道的数量
      (2) 计算账单
      (3) 打印账单

变量　由于程序需要用户输入客户账号、客户代码、用户租用的频道数量以及商业客户的节点数目，所以需要定义变量来存储这些信息。而且，程序需要计算账单总额，所以需要一个变量存储账单总额。因此，为了可以计算和打印账单，程序中至少需要如下的变量定义：

```
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
```

命名常量 由上面的问题分析可知，计算账单所需要知道的账单处理费、基本服务费和每个频道的租用费对于同种类型客户来说，都是固定不变的。虽然这些数值对于程序来说都是常量，但是还是有可能在以后变动。为了便于以后程序修改，应该将它们定义为命名常量而不是在程序中直接使用它们的值，基于上面的问题分析，需要在程序中定义如下的命名常量：

```
      //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 程序中使用到几个公式来计算账单总额。对于家庭客户的账单，只需要知道租用的频道数量和该账单所对应的客户即可。下面是计算家庭客户的账单总额的公式：

```
amountDue = rBillProcessingFee + rBasicServiceCost +
            numberOfPremiumChannels * rCostOfaPremiumChannel;
```

对于商业用户来说，需要知道接入的节点数目、租用的频道数量以及该账单所对应的客户。如果接入的节点数目小于10，基本服务费则是固定不变的；如果接入的节点数目超过10，则10个节点以外的节点必须单独计费。下面的语句用来计算商业用户的账单总额：

```
if(numberOfBasicServiceConnections <= 10)
  amountDue = bBillProcessingFee + bBasicServiceCost +
              numberOfPremiumChannels * bCostOfaPremiumChannel;
else
  amountDue = bBillProcessingFee + bBasicServiceCost +
              (numberOfBasicServiceConnections - 10) *
              bBasicConnectionCost +
              numberOfPremiumChannels * bCostOfaPremiumChannel;
```

主要算法

经过上面的讨论，主要算法设计如下所示：

1. 通过使用控制符fixed和showpoint，将输出的浮点数设置成为固定小数点、两位有效数字、不足两位小数添0补齐的格式。注意为了可以使用这些控制符，必须在程序中包含头文件iomanip。
2. 提示用户输入客户账号。
3. 读入客户账号。
4. 提示用户输入客户代码。
5. 读入客户代码。
6. 如果客户代码是r或者R：
   a. 提示用户输入租用的频道数目
   b. 读入租用的频道数目
   c. 计算账单总额
   d. 打印客户账号和账单总额

7. 如果客户代码是 b 或是 B：
   a. 提示用户输入接入节点的数目
   b. 读入接入节点的数目
   c. 提示用户输入租用的频道数目
   d. 读入租用的频道数目
   e. 计算账单总额
   f. 打印客户账号和账单总额
8. 如果客户代码不是 r，R，b，B，则输出错误信息。

在第 6 步和第 7 步，程序使用 switch 语句来计算客户账单。

完整的程序代码清单

```
#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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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: 16
Enter the number of premium channels: 8

Account number = 12345
Amount due = $520.00
```

4.8 小结

1. 控制结构可以改变程序执行顺序。
2. 最常见的两种控制结构是选择和循环。
3. 选择语句用来在程序中实现判断。
4. 关系运算符有：==（恒等），<（小于），<=（小于或者等于），>（大于），>=（大于或者等于）和!=（不等于）。
5. 如果在关系运算符==，<=，>=，!=中间存在空格，那么将产生语法错误。
6. 字符比较实际上是比较其对应编码值的大小。
7. 逻辑表达式的值是 1（或者是非零值）或者是 0。逻辑值 1（或者非零值）被视为 true；逻辑值 0 被视为 false。
8. 在 C++ 中，可以使用 int 类型变量来存储逻辑表达式的值。
9. 在 C++ 中，可以使用 bool 类型变量来存储逻辑表达式的值。
10. 在 C++ 中，逻辑运算符有：!（非），&&（与）和 ||（或）。
11. 在 C++ 中，有两种类型的选择结构。
12. 单路选择的形式是：

```
if(expression)
   statement
```

    如果表达式 expression 的值是 true，则执行 statement 语句；否则，计算机则执行 if 结构后面的语句。
13. 双路选择的形式是：

```
if(expression)
   statement1
else
   statement2
```

    如果表达式 expression 的值是 true，则执行 statement1；否则，则执行 statement2。
14. if 和 if…else 中的表达式通常是逻辑表达式。
15. 如果在单路选择 if 语句的动作语句（statement）前出现分号，将会导致语义错误。在这种情况下，if 语句中实际动作是空语句。
16. 如果在双路选择 if 语句的动作语句（statement1）前面存在分号，将会导致语法错误。
17. C++ 中没有单独存在的 else 语句，每一个 else 必须与某个 if 相对应。
18. else 与前面最近一个未与 else 相匹配的 if 匹配。
19. 在大括号中括起来的语句序列称为复合语句或者块语句。复合语句可以视为单一语句来使用。
20. 通过在 if 语句中使用输入流变量，可以判断输入流的状态。
21. 将赋值运算符误写成恒等运算符将产生语义错误。这种错误由于不易发现，所以通常会导致更严重的错误。
22. switch 语句用来处理多路选择。
23. 如果在 switch 结构中执行 break 语句，程序将立即跳出该 switch 结构。
24. 如果程序中的某些条件没能得到满足，可以使用 assert 函数来终止程序运行。

4.9 练习

1. 判断下面说法的正误。

   a. 逻辑表达式的值不能赋给 int 类型变量。

   b. 在单路选择中，如果在if语句的条件表达式后面出现分号，则if语句中的条件表达式的值将是true。

   c. 每一个if语句都有与之相对应的else语句。

   d. 下面if语句中的表达式永远是true。

   ```
   if(score = 30)
      grade = 'A';
   ```

   e. 如果 ch < 'A' 或者 ch >= 'Z'，那么下面表达式的值永远是false。

   ```
   (ch >= 'A' && ch <= 'Z')
   ```

   f. 假设输入的数字是5，则下面输出语句将输出：Num is zero

   ```
   cin>>num;
   if(num > 5)
     cout<<num;
     num = 0;
   else
     cout<<"Num is zero";
   ```

   g. switch语句中表达式的值应该是简单数据类型。

   h. 只有当x是负数时，表达式!(x > 0)的值才是true。

   i. 在C++中,!和!=都是逻辑运算符。

   j. 程序中语句的执行顺序被称为控制流。

2. 选择最佳答案。

   a.
   ```
   if (6 < 2 * 5)
          cout << "Hello";
          cout << " There";
   ```
   将输出：

   (i) `Hello There`　(ii) `Hello`　(iii) `Hello` `There`　(iv) `There`

   b.
   ```
   if ('a' > 'b' || 66 > static_cast<int>('A'))
            cout << "#*#";
   ```
   将输出：

   (i) #*#　(ii) # * #　(iii) *　(iv) 以上全不是

   c.
   ```
   if (7 <= 7)
        cout << 6 - 9 * 2 / 6;
   ```
   将输出：

   (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;
  }
```

将输出：

(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<<"$";
```

将输出：

(i) *　(ii) &　(iii) $　(iv) 以上全不是

3. 下面 C++ 代码的输出结果是什么？

```
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++ 代码以完成如下功能：如果变量 gender 的值是 'M'，将输出 Male；如果变量 gender 的值是 'F'，将输出 Female；如果变量 gender 的值是其他字符，将输出 invalid gender。
5. 改正下面代码中的错误，使其输出正确的信息。

```
if(score >= 60)
  cout<<"You pass."<<endl;
else;
  cout<<"You fail."<<endl;
```

6. 判断下面的 switch 语句是否合法。如果不合法，说明原因。假设 n 和 digit 都是 int 类型变量。

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. 假设输入数据是5。下面的C++语句执行以后，alpha中的值是多少？

```
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. 假设输入数据是3。下面的C++语句执行以后，beta中的值是多少？

```
cin>>beta;
switch(beta)
{
case 3: beta = beta + 3;
case 1: beta++;  break;
case 5: beta = beta + 5;
case 4: beta = beta + 4;
}
```

9. 假设输入数据是6。下面的C++语句执行以后，a中的值是多少？

```
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. 改正下面程序中的错误，使其能够正确地编译和运行。

```
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. 下面的程序中含有错误。改正程序中的错误，使该程序能够正确运行并输出 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。程序将输出该数字并说明该数字是正数、负数还是零。
2. 编写一个程序提示用户输入三个数字。程序将按照从小到大顺序输出这三个数字。
3. 编写一个程序提示用户输入一个界于0和35之间的数字。如果该数字小于等于9，程序将输出这个数字；否则将根据规则输出 A 到 Z 中的某个字母。对应规则是：10对应 A，11 对应 B，12对应C，…，35对应Z（提示：当该数字大于等于10时，使用类型强制转换符 static_cast<char>( )）。
4. 从纽约到巴黎的国际长途电话计费标准如下：连接费是$1.99，前三分钟的通话费是$2.00，以后每分钟的通话费是$0.45。编写一个程序提示用户输入通话持续的分钟数，计算并输出通话费用。要求输出结果保留两位有效数字。

5. 在直角三角形中,斜边长度的平方等于两个直角边的平方和。编写一个程序提示用户输入三角形三个边的长度，判断该三角形是否是直角三角形，并输出判断结果。
6. 一元二次方程 $ax^2 + bx + c = 0$，$a \neq 0$ 的求根公式是：

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

在这个公式中，$b^2 - 4ac$ 被称为判别式。如果 $b^2 - 4ac = 0$，那么该方程只有单（重）根；如果 $b^2 - 4ac > 0$，那么该方程有两个实根；如果 $b^2 - 4ac < 0$，那么该方程有两个复根。编写一个程序提示用户输入 a（x^2 的系数）的值，b（x 的系数）的值，以及 c（常数项）的值，此外，如果 $b^2 - 4ac \geq 0$，则计算并且输出一元二次方程的根（提示：可以使用头文件中 pow 函数来计算算术平方根。第 3 章中介绍了如何使用 pow 函数）。
7. 编写一个程序提示用户输入笛卡儿（Cartesian）平面中的点的 x-y 坐标。程序将输出信息说明该点是在原点上或是在 x（或者 y）轴上，还是在某个象限中。例如：

```
(0,0) is the origin
(4,0) is on the x-axis
(0,-3) is on the y-axis
(-2,3) is in the second quadrant
```

8. 编写一个程序模拟计算器功能。程序的输入是两个整数和运算符,程序的输出是含有输入整数和运算符的表达式和计算结果（除法中如果遇到除数为 0 的情况，应该输出相应的错误信息）。例如：

```
3 + 4 = 7
13 * 5 = 65
```

9. 改写练习题 8 中的程序，使之可以计算浮点数（输出的数字保留两位有效数字）。
10. 本市某银行在月末清算所有客户的账目。银行提供两种类型的账户：储蓄账户和支票账户。每个客户的账户中必须保持一个最小账户余额。如果某个客户的储蓄账户余额低于最小账户余额,则将对该客户收取 $10.00 的服务费；如果支票账户余额低于最小账户余额，则将对该客户收取 $25.00 的服务费。如果客户的账户余额高于最小账户余额，则按照下面规则支付利息：

    a. 储蓄账户的利息率是 4%。

    b. 支票账户的余额如果不高于最小账户余额 $5 000，利息率是 3%；否则，利息率是 5%。

    编写一个程序，该程序读入客户账号（int 类型）、账户类别（char 类型，储蓄账户是 S，支票账户是 C）、需要保持的最小账户余额和当前余额。程序将输出客户账号、账户类别、当前余额和相应的信息。使用下面的数据，分 5 次验证程序的正确性：

```
46728 S 1000 2700
87324 C 1500 7689
79873 S 1000 800
89832 C 2000 3000
98322 C 1000 750
```

11. 根据例 1.2（第 1 章）中的算法编写程序，计算销售人员的月工资。
12. 编写一个程序,该程序用来计算并打印移动电话服务供应商的客户账单。该公司提供两种类型的服务：普通服务和优惠服务。根据不同的服务类别来计算服务费。服务计费标准是：

    普通服务：　$10.00，同时前 50 分钟免费；超过 50 分钟的部分，按照每分钟 $0.20 计费。

    优惠服务：　$25.00，同时：

    a. 6:00 a.m.到 6:00 p.m.之间的电话，前 75 分钟免费；超过 75 分钟的部分，按照每分钟 $0.10 计费。

b. 6:00 p.m.到6:00 a.m.之间的电话，前100分钟免费；超过100分钟的部分，按照每分钟 $0.05 计费。

该程序将提示用户输入账号、服务代码（char类型）以及通话时间。服务代码R或者r表示普通服务；服务代码P或者p表示优惠服务；其他字符将被视为非法输入。程序将输出账号、服务代码、通话时间（以分钟计）以及客户的话费总额。

对于优惠服务，客户既有可能在白天打电话也可能在晚上打电话。因此，需要用户分别输入白天的通话时间和晚上的通话时间。

第5章　控制结构Ⅱ（循环）

本章要点：

- 了解循环控制结构
- 理解怎样使用计数控制、标志控制和EOF控制来构建循环结构
- 了解break和continue语句
- 理解怎样构建和使用嵌套控制结构

第4章介绍了选择结构，以及怎样在程序中实现选择结构。本章将介绍循环结构，以及怎样在程序中实现循环结构。

5.1　为什么需要循环

假设要计算5个数字的平均数。根据目前学到的知识，计算过程如下（假设所有变量都经过正确定义）：

```
cin >> num1 >> num2 >> num3 >> num4 >> num5; //read five numbers
sum = num1 + num2 + num3 + num4 + num5;     //add the numbers
average = sum / 5;                          //find the average
```

现在，再假设要计算100个、1 000个或者更多个数字的平均数。在这种情况下，需要分别定义这些变量，将它们列到cin语句的输入变量列表中，甚至还需要列到输出语句的输出变量列表中。这将是一件十分耗费时间和精力的事。而且，为了计算不同数目的数字平均值，还需要重新改写程序。

假设要计算下面数字的和：

```
5 3 7 9 4
```

考虑下面语句，其中sum和num是int类型变量：

```
1.sum = 0;
2.cin >> num;
3.sum = sum + num;
```

语句1将变量sum初始化为0。现在再来看语句2和语句3：语句2将5存储到变量num里，语句3将num中的值加到sum里。在执行完语句3后，sum中的值是5。

让我们重复执行语句2和语句3。在执行完语句2（也就是在程序读入下一个数字后）：

```
num = 3
```

在执行完语句3后：

```
sum = sum + num = 5 + 3 = 8
```

到目前为止，变量sum中存储了两个数字的和。让我们再次执行语句2和语句3（第3次）。在执行完语句2后（也就是在程序读入下一个数字后）：

```
num = 7
```

在执行完语句 3 后：

```
sum = sum + num = 8 + 7 = 15
```

到目前为止，变量 sum 中存储了 3 个数字的和。如果再执行语句 2 和语句 3 两次，变量 sum 中将存储了全部 5 个数字的和。

如果要计算 10 个数字的和，需要执行 10 次语句 2 和语句 3。如果要计算 100 个数字的和，需要执行 100 次语句 2 和语句 3。但是，无论在哪种情况中，都无需定义很多变量。这种程序代码可以计算任意多的数字的和。

在很多情况下，需要重复执行程序中的某一部分代码。例如，对于某个班级中的所有学生来说，计算课程成绩的程序代码都是相同的。C++ 提供了三种重复机制，或者说循环机制，来在满足某种条件的情况下，循环执行某些语句。本章将介绍全部这三种循环结构。下一节将首先讨论第一种循环结构——while 循环结构。

5.2　while 循环（重复）结构

在前面一节中，已经见到了在某些情况下，有重复执行程序中某些语句的需求。一种重复执行某些语句的方法是，在程序中重复键入这些语句。例如，如果要重复 100 次执行某些语句，就需要在程序中键入 100 条同样的语句。但是这种方法不仅复杂，而且有时难以实现。令人高兴的是，C++ 提供了实现重复，或者说是循环的更好方法。正如前面提到的，C++ 提供了三种在满足某些条件的情况下实现循环的机制。本节将介绍第一种循环结构—— while 循环。

while 语句的结构是：

```
while(expression)
    statement
```

在 C++ 中，while 是保留字。当然，这里的 statement 可以是简单语句，也可以是复合语句。expression 为循环的条件表达式，而且通常是逻辑表达式。这里 statement 也被称为循环体。注意，括住条件表达式的括号也是该语句语法中不可缺少的一部分。图 5.1 说明了 while 循环的执行过程。

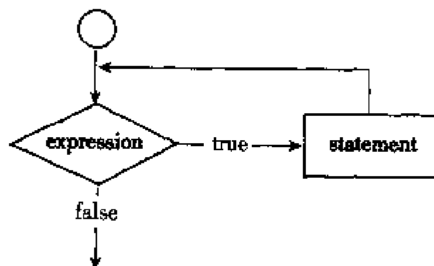


图 5.1　while 循环

expression 是 while 循环的入口条件。如果该表达式的初始值是 true，则执行循环体。然后，再次计算循环条件 expression，如果为 true，则循环体被再次执行。循环体中的语句将反复执行，直到 expression 的值不再是 true 为止。无休止循环被称为无限循环（Infinite Loop）。为了避免无限循环，必须保证在循环体中包含出口条件，即使 while 中 expression 的最终值为 false。

例 5.1　考虑下面的 C++ 程序段：

```
i = 0;                  //Line 1
while(i <= 20)          //Line 2
```

```
{
   cout<<i<<" ";          //Line 3
   i = i + 5;             //Line 4
  }
```

该程序段的输出是：

```
0 5 10 15 20
```

在第1行中，变量i的初始值是0。现在，计算while语句中的条件表达式：i <= 20。因为表达式 i <= 20的值是true，所以执行while语句中的循环体。该while语句中的循环体由第3行和第4行语句组成。第3行语句输出变量i的值，即0。第4行语句将变量i的值变成5。在执行完第3行和第4行中的语句后，while语句中的条件表达式的值被重新计算（第2行）。因为i的当前值是5，所以条件表达式i <= 20的值是true，所以再次执行while语句中的循环体。这个计算条件表达式值并执行循环体中语句的过程将反复执行，直到条件表达式i <= 20（第2行）的值不再是true为止。表达式中的变量i（第2行），被称为循环控制变量。

在上面的例子中需要注意：

a. 变量i中的值变成25之后，并不输出i的当前值，因为此时入口条件的值是false。

b. 如果循环体中漏掉语句：

```
i = i + 5;
```

该循环将变成无限循环，将不断地输出一行又一行的0。

c. 必须在执行循环体之前，初始化循环控制变量i。如果漏掉（第1行中的）语句：

```
i = 0;
```

该循环可能不会执行（注意，C++并不自动初始化变量）。

d. 在上面的程序段中，如果互相交换循环体中两个语句的位置，结果将大不一样。例如，考虑下面语句：

```
i = 0;
while(i <= 20)
{
  i = i + 5;
  cout<<i<<" ";
}
```

现在的输出结果是：

```
5 10 15 20 25
```

这应该是一个语义错误，因为在条件i <= 20时，不应该输出大于20的i。

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
}
```

本例题和例5.1有一些区别。在第1行中，变量i的值是20。因为i是20，所以while语句中的条件表达式i < 20（第2行）的值是false。由于初始入口条件i < 20的值是false，故while语句的循环体永远不会被执行。因此，该程序不会输出任何i值，并且i中的值是20。

当一个程序需要反复校验某些数据，或者程序需要处理大量数据时，使用键盘键入所需数据是不可取的。在这种情况下，可以将数据存储到外部设备，如磁盘上，然后使程序从该设备中读取数据。为了简便起见，假定磁盘（软盘或硬盘）中的数据没有任何错误。循环可以分为4种情形，分别在下面4小节中加以讨论。

5.2.1　情形1：计数器控制的while循环

假定已经准确地知道将要读入的数据（或数据项）的数目。在这种情况下，可以采用一种名为计数器控制的while循环（Counter-controlled while loop）结构来处理。假设文件中有N个数据项。可以使用计数器（在while语句之前初始化为0）来记录已经读入的数据项个数。在执行while语句的循环体之前，将计数器中的当前数值与N做比较。如果计数器中的值小于N，则执行while语句的循环体。该循环体将反复执行，直到计数器中的值大于或等于N为止。在while语句的循环体中，计数器的值在每读入一个数据项后都增加1。在这种情况下，while循环的形式如下所示：

```
counter = 0;
while(counter < N)
{
      .
      .
      .
      counter++;
      .
      .
      .
}
```

可以采用多种方法来确定N（文件中数据项的个数）的值。程序提示输入文件中数据项个数，再使用cin语句读入这个数字。此外，还可以指定文件中第一个数据项记录文件中的数据项的个数。这样，程序员就无须知道需要处理的数据项个数。当数据文件不是由程序员创建时，很适于采用这种方法来处理。考虑下面例5.3。

例5.3　假设输入的数据是：

```
12 8 9 2 3 90 38 56 8 23 89 7 2 8 3 8
```

第1个数字12，用来指定需要处理的数据集中含有的数据项的个数。假设要计算这12个数据项的和及平均值。注意，只有12个数据项能被读入，其余的数据项不做处理。完整的程序如下所示：

```
//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;                                  //Line 1
      cin>>limit;                                  //Line 2
```

```
        sum = 0;                                    //Line 3
        counter = 0;                                //Line 4

        while(counter < limit)                      //Line 5
        {
            cin>>number;                            //Line 6
            sum = sum + number;                     //Line 7
            counter++;                              //Line 8
        }

        cout<<"Line 9: The sum of "<<limit
            <<" numbers = "<<sum<<endl;             //Line 9

        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;
}
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
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
```

该程序的执行过程如下：第1行中的语句提示用户输入需要处理的数据。第2行中的语句从输入数据中读入第1个数据项，并将其存储到变量limit中。limit中的值表示需要处理的数据个数。第3行和第4行中的语句分别将变量sum和counter初始化为0。第5行中的while语句检查counter中的值，以确定已经读入的数据个数。如果counter小于limit，while语句执行下一个循环。第6行中的语句读入下一个数字，并将其存储到变量number中。第7行语句将变量number中的值加到变量sum中去。第8行语句将counter中的值加1。第9行语句输出所有数字的和。第10行到第13行语句输出平均值。

注意，本程序的第3行将变量sum初始化为0。在第7行，也就是读入number以后，程序将以前读入的所有number值的和与当前的number值相加。第1个读入的number值将与0相加（因为sum初始化为0）。为了计算平均值，将sum除以counter。如果counter中的值是0，将0作为除数将终止程序并且发出错误信息。因此，在用counter除number之前，必须首先检查counter是否为0。

5.2.2　情形2：结束标记控制的while循环

在很多情况下，事先并不知道读入的数据个数。但是，却知道最后要读入的是一个很特殊的数据值，称为结束标记（Sentinel）。在这种情况下，首先在while语句前读入第一个数据项。如果该数据值不等于结束标记，则执行while语句的循环体。只要读入的值不是结束标记，while语句将一直执行下去。这种while循环被称为结束标记控制的while循环（Sentinel-controlled while loop）。在这种情形下，while循环的形式如下所示：

```
cin>>variable;
while(variable != sentinel)
{
```

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

例 5.4 假设程序要读入若干个整数并且计算它们的平均值，而且事先并不知道需要读入的数据个数。假设数字 -999 标志输入数据的结束，程序代码如下所示：

```
//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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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
```

程序的执行过程如下：第 1 行语句提示用户输入第 1 个数字。第 2 行语句读入第 1 个数字并将其存储到变量 number 中。第 3 行 while 语句检查 number 是否等于结束标记。如果 number 不等于结束标记，则执行 while 语句的循环体。第 4 行语句将 number 中的值加到 sum 中去。第 5 行语句将 count 中的值加 1。第 6 行语句读入下一个数字，并将其存储到变量 number 中。第 4 行到第 6 行语句将反复执行，直到程序读入结束标记。第 7 行语句输出所有数字的和，第 8 行到第 10 行语句输出这些数字的平均值。

为了进一步弄清程序的执行过程，可以使用下面数据进行代码走查：

```
89 23 64 78 23 09 45 78 34
0 34 87 23 8 3 5 -999
```

接下来，考虑下一个标记控制的while循环的例子。在本例中，程序提示用户输入需要处理的数据值。因此，用户可以通过输入结束标记值来终止程序运行。

例5.5　电话数字

下面程序读入字母A到Z，并且输出相应的电话按键数字。本程序用到了结束标记控制的while循环。用户可以通过输入结束标记值来终止程序运行。这同时也是一个嵌套控制结构的例子，其中包括if…else，switch和while循环嵌套。

```
//*********************************************************
// 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;
}
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
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
```

本程序的执行过程如下：第2行和第3行中的语句提示用户该怎样做。第4行中的语句提示用户输入一个字母，第5行中的语句读入该字母，并将其存储到变量letter中。第7行中的while语句检查该字母是否为Q或者Z。如果该字母不是Q或者Z，则执行while语句中的循环体。第8行语句将用户输入的字母输出。第10行中的if语句检查用户输入的字母是否为大写字母。该if语句的动作语句部分是switch语句（第11行）。如果用户输入的是大写字母，if语句（第10行）中条件表达式的值是true，执行switch语句。如果用户输入的不是大写字母，则执行else（第27行）语句。第12到第26行中的语句确定相应的电话按键数字。

一旦当前输入的字符被处理，第29行和第30行语句再一次提示用户下一步该怎么做。第31行语句提示用户输入一个字母，第32行语句读入该字母，并将其存储到变量letter中（注意：第29行与第2行语句很相似，第30行至第33行语句与第3行至第6行语句完全相同）。在第33行语句执行以后，程序的控制权返回到while循环的开头处，并再次执行该循环。当用户输入Q或者Z后，程序终止。

注意：在例5.5的程序中，可以只使用一个switch语句来改写第10行到第28行语句（见本章后面的“编程练习4”）。

5.2.3　情形3：标志控制的while循环

标志控制的while循环使用布尔变量来控制循环。假设found是布尔变量。标志控制的while循环的形式如下所示：

```
found = false;

while(!found)
{
    .
    .
    .
    if(expression)
       found = true;
    .
    .
    .
}
```

变量found，用来控制while循环的执行，称为标志变量。

5.2.4　情形4：EOF控制的while循环

如果数据文件需要经常改动（例如经常需要增删数据），最好不要使用标记控制的while循环结构。因为在这种情况下，标记值本身有可能被删除，或者在标记值之后还有新增需要处理的值。特别是在由非程序员输入数据的情况下更是如此。另外，有时候程序员本身也不知道标记值应该是什么。在这种情况下，可以使用EOF控制的while循环（EOF-controlled while loop）结构。

到目前为止，我们已经学会使用输入流变量，如cin和析取运算符>>，读入数据并存储到变量中。然而，输入流变量还可以在下面情况下返回逻辑值：

1. 如果程序读到输入文件的结束处，输入流变量返回逻辑值false。
2. 如果程序读入了任何错误数据（例如将char类型数值读入到int类型变量中），输入流进入到输入状态中。一旦输入流进入错误状态，任何使用该输入流的后续操作都将是空操作。也就是说，这些操作将没有任何作用。遗憾的是，计算机并不能终止程序并给出相应的错误信息。程序将继续执行下去，并且毫无声息地忽略掉所有使用到该输入流的操作。在这种情况下，输入流变量返回逻辑值false。
3. 除了(1)和(2)中所述情况，输入流变量将返回逻辑值true。

可以通过检查输入流变量的返回值来判断程序是否遇到了文件的结束标志。因为输入流变量将返回逻辑值true或者false，所以在while循环中，可以将其作为逻辑表达式来使用。

下面是EOF控制的while循环的形式。

```
cin>>variable;
while(cin)
{
      .
      .
      .
      cin>>variable;
      .
      .
      .
}
```

5.2.5 eof函数

为了便于检查输入流变量，如cin，以判断程序是否遇到了文件结束标志，C++提供了eof函数。该函数可以用来检查输入流变量是否处于文件结束状态。与第3章中讨论的其他I/O函数，如get，ignore和peek一样，函数eof是istream类型变量的成员函数。

使用eof函数的语法如下所示：

```
istreamVar.eof()
```

这里istreamVar是输入流变量，例如cin。

假设有如下的变量定义：

```
ifstream infile;
```

进一步假设程序中使用变量infile打开文件。考虑下面的表达式：

```
infile.eof()
```

这是一个逻辑（布尔）表达式。在程序读到文件结束标志时，表达式的值是true；否则，表达式的值是false。

这种判断文件结束状态的方法（即使用eof函数的方法）特别适合输入文件是文本文件的情形。而前一种判断文件结束状态的方法则特别适合输入文件中含有数字的情形。

假设有下面的变量定义：

```
ifstream  infile;
char ch;

infile.open("inputDat.dat");
```

下面的while循环将一直进行下去，直到遇到文件结束标志为止。

```
infile.get(ch);
while(!infile.eof())
{
      cout<<ch;
      infile.get(ch);
}
```

只要程序没有遇到输入文件的文件结束标志，表达式：

```
infile.eof()
```

的值就是false，所以while语句中条件表达式：

```
!infile.eof()
```

的值就是true。当程序试图读取超过文件结束标志以外的数据时，表达式：

```
infile.eof()
```

的值变成true，所以while语句中表达式：

```
!infile.eof()
```

的值就变成false，循环结束。

注意：在DOS环境中，文件结束标志的输入方法是Ctrl+Z（按住Ctrl键的同时按下Z键）。在UNIX环境中，文件结束标志的输入方法是Ctrl+D（按住Ctrl键的同时按下D键）。

5.3 程序范例：活期账户余额

当地一家银行需要编写一个程序，该程序用来在月底计算客户的活期储蓄账户余额。文件中数据的存储格式如下所示：

```
467343 23750.40
W 250.00
D 1200
W 75.00
I 120.74
.
.
.
```

第1行中的数据是账号和月初账户余额。下面的每一行都有两个数据项：交易代码和交易金额。交易代码W或w代表取款，D或d代表存款，I或i代表银行付的利息。程序在完成每笔交易后，更新账户余额。无论在一个月中的任何时候，只要客户的账户余额低于$1 000.00，银行就要收取$25.00的服务费。程序将输出下面信息：账号、月初账户余额、月末账户余额、银行支付的利息、存款总金额、存款的笔数、取款总金额、取款的笔数以及服务费（如果存在）。

输入　上述格式的数据文件。

输出　输出的格式如下所示：

```
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

输入文件的第一行数据就是账号和月初账户余额。所以，程序首先要读入账号和月初账户余额。此后文件中的每一条数据项的形式如下所示：

transactionCode（交易代码）　transactionAmount（交易金额）

为了计算每个月末的账户余额，程序需要处理每一条交易记录，其中包括：交易代码和交易金额。账户中的余额开始是月初账户余额，然后在每笔交易后都更新该余额。如果交易代码是D，d，I或者i，则在账户余额中加上该交易金额；如果交易代码是W或者是w，则在账户余额中减去该交易金额。因为程序要分别输出存款和取款的总金额，所以还需要分别记录存款和取款的金额。经过上述讨论，可以得到以下算法：

1. 定义变量
2. 初始化变量
3. 读入账号和月初余额
4. 读入交易金额和交易代码
5. 分析交易代码并对变量做相应的更新

6. 重复第4步和第5步，直到所有数据都处理完为止
7. 输出结果

变量　程序需要输出账号、月初账户余额、月末账户余额、银行支付的利息、存款总金额、存款的笔数、取款总金额、取款的笔数以及服务费（如果存在）。需要定义变量存储这些信息。到目前为止，至少需要定义如下变量：

```
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
```

因为程序需要从输入文件中读入数据，并向输出文件中写入数据，所以需要在程序中定义输入流变量和输出流变量。在输入文件第一行以后，每一行数据都是由交易代码和交易金额组成；需要在程序中定义变量来存储这些信息。

只要账户余额低于最小账户余额，则需要对该账户收取服务费。在一个月中可能进行多次存取操作，所以可能出现以下情形：某次取款后账户余额低于最小账户余额，接下来的存款后账户余额又高于最小账户余额，随后的取款后账户余额再次低于最小账户余额。需要注意的是，在这种情况下，只收取一次服务费。

为了实现这一功能，需要在程序中使用布尔变量 isServiceCharged。该变量的初始值是 false，但是一旦账户余额低于最小账户余额，它的值就变成 true。在收取服务费之前，程序需要检查变量 isServiceCharged 的值。如果账户余额小于最小账户余额，并且 isServiceCharged 中的值是 true，则需要收取服务费。所以程序中需要定义如下变量：

```
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
```

命名常量　因为最小账户余额和服务费是固定不变的，所以需要在程序中定义两个命名常量来存储它们。

```
const double minimumBalance = 1000.00;
const double serviceCharge = 25.00;
```

问题分析和算法设计（续）

因为本程序比前面遇到的程序都复杂，所以在动手编写主要算法之前，需要将前面7步算法细化。

1. **定义变量** 根据前面的讨论定义各变量。
2. **初始化变量** 在每次取款之后，需要更新取款总金额并将取款笔数加1。在第一次存款之前，存款总金额是0，存款笔数也是0。因此，变量amountDeposited和numberOfDeposits的初始值都是0。同样，变量amountWithdrawn，numberOfWithdrawals和interestPaid的初始值也将是0。正如前面讨论，变量isServiceCharged将初始化为false。当然，也可以在定义这些变量的同时将它们初始化。

   在第一次存款、取款或是付息之前，账户的余额与月初账户余额相同。因此，在将月初账户余额从文件读入到变量beginningBalance之后，需要将变量accountBalance初始化为beginningBalance中的值。

   既然要从文件中读取数据，就需要打开输入文件。如果输入文件并不存在，输出相应信息并且终止程序。因为输出的数据要存储到文件中，所以要打开输出文件。假设输入文件是A驱动器中软盘上的文件Ch5_money.txt，并且将输出数据存储到A驱动器中软盘上的文件Ch5_money.out中。下面的代码用来打开这两个文件：

   ```
   infile.open("a:Ch5_money.txt"); //open the input file

   if(!infile)
   {
      cout<<"Cannot open input file"<<endl;
      cout<<"Program terminates!!!"<<endl;
      return 1;
   }
   ```

3. **读入账号和月初余额** 这可以通过下面语句来完成：

   ```
   infile >> acctNumber >> beginningBalance;
   ```

4. **读入交易代码和交易金额** 这可以通过下面语句来完成：

   ```
   infile >> transactionCode >> transactionAmount;
   ```

5. **分析交易代码并且更新相应变量** 如果transactionCode是'D'或者'd'，则需要通过加上transaction Amount来更新变量accountBalance中的值，而且还需要将numberOfDeposits加上1。如果transactionCode是'I'或者'i'，则需要通过加上transactionAmount来更新变量accountBalance和interestPaid中的值。如果transactionCode是'W'或者'w'，则需要通过减去transactionAmount来更新变量accountBalance中的值，而且还需要将numberOfWithdrawals加上1。如果账户余额低于最小账户余额，而且没有收取服务费，则需要在账户余额中减掉服务费，并且标记已经收取过服务费。可以通过下面的switch语句完成这些功能。

   ```
   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. **重复第4步和第5步，直到输入数据结束为止**　因为输入文件中的输入项个数未知，所以应该选用EOF控制的while循环。
7. **输出计算结果**　这可以通过输出语句来完成。

根据上面的讨论，程序主要算法如下所示。

主要算法

1. 定义并初始化变量。
2. 打开输入文件。
3. 如果输入文件不存在，终止程序。
4. 打开输出文件。
5. 通过控制符fixed，showpoint将输出数据的格式设置为带有两位有效数字的固定小数点格式，不足两位小数的部分添零补齐。
6. 读入accountNumber和beginningBalance。
7. 将accountBalance初始化为beginningBalance。
8. 读入transactionCode和transactionAmount。
9. 当（文件没有结束时）:
   a. 如果transactionCode是'D':
      (1) 将transactionAmount加到accountBalance中
      (2) 将numberOfDeposits加1
   b. 如果transactionCode是'I':
      (1) 将transactionAmount加到accountBalance中
      (2) 将transactionAmount加到interestPaid中
   c. 如果transactionCode是'W':
      (1) 从accountBalance中减去transactionAmount
      (2) 将numberOfWithDrawals加1
      (3) 如果（accountBalance < minimumBalance && !isServiceCharged）为true:

(i) 从 accountBalance 中减去 serviceCharge

(ii) 将 isServiceCharged 设置为 true

d. 如果 transactionCode 是'D'，'d'，'I'，'i'，'W'或'w'以外的其他字母，输出相应的错误信息。

10. 输出结果。

因为要从输入文件中读入数据，所以必须要包括头文件 fstream。因为要使用控制符 setprecision，所以必须要包括头文件 iomanip。如果输入文件并不存在，需要在屏幕上输出相应的信息，所以还要包括头文件 iostream。

完整的程序代码清单

```
//*******************************************************
// 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 9.a
    case 'd': accountBalance = accountBalance
                             + transactionAmount;
              amountDeposited = amountDeposited
                              + transactionAmount;
              numberOfDeposits++;
              break;
    case 'I':                                       //Step 9.b
    case 'i': accountBalance = accountBalance
                             + transactionAmount;
              interestPaid = interestPaid
                           + transactionAmount;
              break;
    case 'W':                                       //Step 9.c
    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;
}
```

程序运行结果 （输出文件 A:Ch5_money.out 的内容）

```
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：如果使用标准C++编写上面的程序，需要将下面语句：

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

改写成：

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

可能还需要将语句：

```
outp << fixed << showpoint;
```

改写成：

```
outp.setf(ios::fixed, ios::floatfield);
outp.setf(ios::showpoint);
```

5.4　程序范例：Fibonacci数列

到目前为止，已经遇到了许多循环语句的例子。这些while循环主要用来读入数据，但是除此之外，它还有别的用处。在C++中，while循环用于在满足特定条件之前，重复执行某些语句。下面的程序使用while循环来计算Fibonacci（斐波那契）数列。

考虑下面的数字序列：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

如果给定该序列中的前两项（比如说a_1和a_2），则该序列中的第n项a_n（n >= 3）的计算公式是：

$$a_n = a_{n-1} + a_{n-2}$$

因此：

$$\begin{aligned} a_3 &= a_2 + a_1 \\ &= 1 + 1 \\ &= 2 \\ a_4 &= a_3 + a_2 \\ &= 1 + 2 \\ &= 3 \end{aligned}$$

以此类推。

这种数列称为Fibonacci数列。在上面的数列中，$a_2 = 1$并且$a_1 = 1$。但是，只要任意给定两个数，应用上面的计算公式，都可以计算出数列中第n个数字a_n（n >= 3）。通过这种方法计算出来的数字，称为第n个Fibonacci数字。假设$a_2 = 6$并且$a_1 = 3$。则：

$a_3 = a_2 + a_1 = 6 + 3 = 9$; $a_4 = a_3 + a_2 = 9 + 6 = 15$

下面，编写一个程序，在给定前两个数字时，该程序能计算出第n个Fibonacci数字。

输入　Fibonacci数列中的前两个数字以及需要计算Fibonacci数字的项数。

输出　第n个Fibonacci数字。

问题分析和算法设计

为计算a_{10}，即数列中第10个Fibonacci数字，则必须先计算a_9和a_8，这又要计算a_7和a_6，以此类推。因此，为了计算a_{10}，必须先计算a_3，a_4，a_5，…，a_9。根据上述讨论可以得到下面的算法：

1. 读入前两个Fibonacci数字。
2. 读入要计算的Fibonacci数字项数。也就是读入该数字在Fibonacci数列中的位置n。
3. 通过将前两个Fibonacci数字相加，计算出下一个Fibonacci数字。
4. 重复第3步，直到计算出第n个Fibonacci数字。
5. 输出第n个Fibonacci数字。

变量　因为在计算当前的Fibonacci数字之前，必须知道该数字的前两个数字，所以需要如下变量：两个变量（比方说previous1和previous2）来存储当前Fibonacci数字的前两个数字；一个变量（比方说current）来存储当前Fibonacci数字。算法中第3步的执行次数，取决于需要计算的Fibonacci数字的位置。例如，如果要计算第10个Fibonacci数字，那么就需要执行上述第3步8次（因为，用

户已经输入了前两个数字)。因此，需要一个变量来存储第3步需要执行的次数。另外，还需要一个循环控制变量来记录已经执行第3步次数。因此，需要定义如下5个变量：

```
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Fibonacci数字，需要将previous1和previous2中的数值相加，结果存储到current中。为了计算第4个Fibonacci数字，需要将第2个Fibonacci数字（也就是previous2）和第3个Fibonacci数字（也就是current）相加。因此，在计算第4个Fibonacci数字时，则不需要第1个Fibonacci数字。除了再定义其他变量（可能需要定义很多变量来计算所需的Fibonacci数字）外，可以将previous2中的数字存储到previous1中，将current中的数字存储到previous2中。这样，就可以使用变量current来存储下一个Fibonacci数字。重复这个过程，直到所需的Fibonacci数字计算出来为止。最开始的previous1和previous2中的数字是由用户输入的。根据上面讨论，则只需要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. 将前两个数字读入（输入）到previous1和previous2中。
3. 输出前两个Fibonacci数字（回显输入）。
4. 提示用户输入需要计算的Fibonacci数字项数。
5. 将输入的Fibonacci数字项数输入到nthFibonacci中。
6. 因为已经知道了数列中前两个Fibonacci数字，所以可以计算第3个Fibonacci数字。将变量counter初始化为3，用来记录计算过的Fibonacci数字的个数。
7. 应用下面公式，计算下一个Fibonacci数字：

   ```
   current = previous2 + previous1;
   ```

8. 将previous2中的值赋给previous1。
9. 将current中的值赋给previous2。
10. counter加1。
11. 重复第7步到第10步，直到计算出所需的Fibonacci数字为止。下面的while循环，执行第7步到第10步，并且计算出第n个Fibonacci数字（nthFibonacci）。

    ```
    while(counter <= nthFibonacci)
    {
        current = previous2 + previous1;
        previous1 = previous2;
        previous2 = current;
        counter++;
    }
    ```

12. 输出第n个Fibonacci数字，也就是current中的数字。

完整的程序代码清单

```
//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 8
        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
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影（本程序共执行三次）。

程序运行结果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.
```

程序运行结果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.
```

程序运行结果 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

前面一节中讨论的while循环可以满足大多数程序对循环功能的需要。本节中将要介绍C++中的for循环，这是一种特殊形式的 while 循环。它的主要用途是简化计数器控制的 while 循环。因此，for 循环也被称为计数(Counted)或者是索引(Indexed)for 循环。

for 语句的语法如下：

```
for(initial statement; loop condition; update statement)
   statement
```

括号内括住的初始化语句(initial statement)、循环条件(loop condition)和更新语句(update statement)，称为 for 循环控制语句，控制着 for 语句的循环体。图 5.2 说明了 for 循环的执行过程。

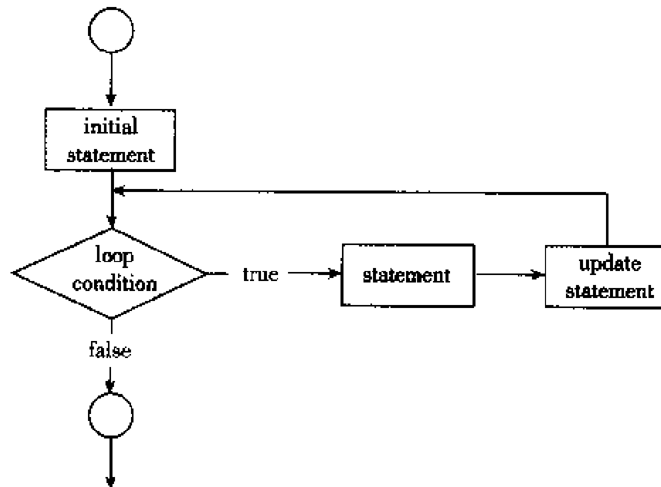


图 5.2 for 循环

for 循环的执行过程如下所示：

1. 执行初始化语句。
2. 计算循环条件。如果循环条件的值是 true：
   (1) 执行 for 循环语句 statement
   (2) 执行更新语句(括号中的第三个表达式)
3. 重复第 2 步，直到循环条件的值为 false 为止。

初始化语句通常用于初始化变量(称为 for 循环控制或索引变量)。

在 C++ 中，for 是保留字。

注意：正如初始化语句的称谓所示，该语句是 for 循环中的第一条语句，并且只执行一次。

例 5.6 下面的 for 循环语句输出前 10 个正整数：

```
for(i = 1; i <= 10; i++)
    cout << i << " ";
```

初始化语句 i = 1;，将 int 类型变量 i 初始化为 1。接下来计算循环条件，i <= 10，因为 i <= 10 的值是 true，所以执行输出语句，输出 1。然后执行更新语句，i++ 将 i 的值更新为 2。再次计算循环条件，还是 true，以此类推。当 i 的值变成 11 时，循环条件的值是 false，for 循环终止，程序随后执行 for 循环结构后面的语句。

for 循环中既可以包括简单语句也可以包括复合语句。

下面的例子进一步说明了 for 循环的执行过程。

例 5.7

1. 下面 for 循环输出正文行和星号行各 5 次：

```
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;
```

这个 for 循环将输出正文行 5 次，但只输出星号 1 次。注意，这里的 for 循环只控制第 1 条 cout 语句，因为这两条 cout 语句并不是复合语句。for 循环共执行 5 次，故第 1 条 cout 语句也执行 5 次。在执行 for 循环后，第 2 条 cout 语句只执行 1 次。

3. 下面的 for 循环将执行 5 条空语句：

```
for(i = 1; i <= 5; i++);            //Line 1
   cout << "*" << endl;             //Line 2
```

for 语句（在 cout 语句之前）后面分号结束了 for 循环。for 循环的循环语句是空语句。

通过这个例子不难看出，要谨慎使用 for 循环。

下面是使用 for 循环的一些注意事项：

- 如果循环条件最初是 false，则不会执行循环体。
- 当执行更新语句时，改变循环变量（通过初始化语句初始化）中的值，并最终使循环变量中的值变为 false。如果循环条件永远是 true，for 循环将永远执行下去。
- C++ 允许使用 double 类型（或其他实数类型）的小数作为循环控制变量。因为不同的计算机系统对这些小数的解释不同，所以要尽量避免使用非 int 类型的数据作为循环控制变量。
- 如果 for 语句后面（在循环体之前）出现分号，则是语义错误。在这种情况下，for 循环的循环语句将是空语句。
- 在 for 语句中，如果循环条件被略掉，则默认为 true。
- 在 for 语句中，可以将三个语句全部略掉——初始化语句、循环条件和更新语句。下面是合法的 for 循环：

```
for( ; ; )
   cout << "Hello" << endl;
```

下面是其他 for 循环的使用范例。

例 5.8

1. 可以以递减的顺序使用循环变量。例如，考虑下面的 for 循环：

```
for(i = 10; i >= 1; i--)
    cout << " " << i;
```

输出的结果是：

```
10 9 8 7 6 5 4 3 2 1
```

在这个 for 循环中，变量 i 的初始值是 10。在每次循环以后，i 递减 1。只要 i >= 1，循环就会一直进行下去。

2. 可以使循环控制变量以固定的步长增加（或减少）。在下面的 for 循环中，变量的初始值是 1。在每次循环以后，该变量递增 2。本 for 循环输出前 10 个正奇数。

```
for(i = 1; i <= 20; i = i + 2)
    cout << " " << i;
```

例 5.9　考虑下面的范例，这里 i 是 int 类型变量：

1.
```
for(i = 10; i <= 9; i++)
    cout << i << " ";
```
在本 for 循环中，初始化语句将变量 i 初始化为 10。因此，循环条件（i <= 9）的值是 false，不会执行循环体。

2.
```
for(i = 9; i >= 10; i--)
    cout << i <<" ";
```
在本 for 循环中，初始化语句将变量 i 初始化为 9。因此，循环条件（i >= 10）的值是 false，不会执行循环体。

3.
```
for(i = 10; i <= 10; i++)
    cout << i << " ";
```
在本 for 循环中，cout 语句只执行一次。

4.
```
for(i = 1; i <= 10; i++);
    cout << i << " ";
```
本 for 循环并没有作用在 cout 语句上。for 语句后面的分号结束了 for 循环，for 循环的循环语句是空语句。cout 语句自己只执行一次。

5.
```
for(i = 1; ; i++)
    cout << i << " ";
```
在本 for 循环中，因为 for 语句中缺少循环条件，所以默认的循环条件永远为 true。这是一个无限循环。

例 5.10　在本例中，for 循环读入 5 个数字，并计算它们的和与平均值。考虑下面的程序代码，其中 i, newNum，sum 和 average 都是 int 类型变量。

```
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;
```

在前面的 for 循环中，读入的每一个 newNum 值都加在 sum 中。变量 sum 在 for 循环前初始化为 0。因此，在程序读入第一个数字并将它与 sum 相加时，变量 sum 的值就是第一个数字的值。

在下面的C++程序中，建议读者走查程序代码的每一步。

例 5.11　下面 C++ 程序的功能是计算前 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;
}
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
Line 1: Enter the number of positive integers to be added:100
Line 6: The sum of the first 100 positive integers is 5050
```

第 1 行语句提示用户输入需要计算的正整数个数。第 2 行语句将用户输入的个数存储到变量 N 中，第 3 行语句将 sum 初始化为 0。第 4 行中的 for 循环将执行 N 次。在 for 循环中，counter 的初始值是 1，并且在每次循环之后都增加 1。因此，counter 的值的范围从 1 到 N。每一次执行循环时，counter 中的值都加在 sum 上。因此，在整个 for 循环执行后，sum 中包含了前 N 个数的和，在本例中运行结果是前 100 个正整数的和。

注意，将一个控制结构置于另一个控制结构中称为嵌套。下面的程序范例将说明一个简单的 for 循环嵌套。

5.6　程序范例：数字分类

程序读入一些整数，并分别输出这些整数中奇数的个数和偶数的个数。同时，还将输出零的个数。

程序将读入 20 个整数。但是经过简单修改，本程序也可以处理任何个数的整数。实际上，还可以通过修改本程序，使之能够处理用户指定个数的整数。

输入 20个整数——正数、负数或零。

输出 零的个数、偶数的个数和奇数的个数。

问题分析和算法设计

在读入一个数字后，程序将判断它是奇数还是偶数。假设输入的数字存储在变量number中。将number除以2，并检查余数。如果余数是0，number就是偶数。将偶数的个数加1，并检查number是否为0。如果number的值是0，将零的个数加1，否则，将奇数的个数加1。

程序中使用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，则余数是1；如果number是负数，则余数是-1。如果number是偶数，无论是正数还是负数，余数都将是0。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
```

应用上面讨论的方法，对所有输入的整数进行逐个分析。

经过上面的讨论，得到算法如下所示：

1. 对于每一个整数：
   a. 读入该数字
   b. 分析该数字
   c. 增加相应的奇数、偶数或零的个数
2. 输出结果。

变量 既然要计算奇数、偶数和零的个数，就需要三个int类型变量——比方说odds，evens和zeros，来分别记录这三种整数的个数。还需要一个变量，比方说number，读入并存储需要分析的整数。另外，还需要一个变量，比方说counter，来记录已经分析的整数个数。程序中需要的变量如下所示：

```
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，需要将变量zeros，evens和odds初始化为0。可以在定义这些变量的时候，初始化它们。

主要算法

1. 分析变量。
2. 提示用户输入20个整数。
3. 对于每一个整数：
   a. 将该整数读入到number中
   b. 输出该整数
   c. 如果number是偶数：
      {
        (1) 将偶数个数加1
        (2) 如果number是0，将零的个数加1
      }
      否则：
        将奇数的个数加1
4. 输出结果。

在动手编写程序之前，让我们对第1步到第4步做更仔细地研究。这将有助于将其改写成C++语句。

1. 初始化变量。可以在定义变量 zeros，evens 和 odds 时将其初始化。
2. 使用输出语句提示用户输入 20 个整数。
3. 在第 3 步，可以使用 for 循环来处理和分析这 20 个整数。本步骤的伪代码如下所示：

```
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。

完整的程序代码清单

```
//**********************************************************
// 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;                                  //Step1
   int odds = 0;                                   //Step1
   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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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    9    0   12   67   54
There are 13 evens, which also includes 6 zeros
Total number of odds are: 7
```

建议读者使用上面的输入数据，对程序进行代码走查。

5.7 do…while 循环（重复）结构

本节将讨论第三种循环或者重复结构，称为do…while循环结构。do…while循环结构的形式如下所示：

```
do
  statement
while(expression);
```

当然，这里的语句 statement 既可以是简单语句也可以是复合语句。如果是复合语句，这些语句必须用括号括起来。图 5.3 说明了 do…while 循环的执行过程。

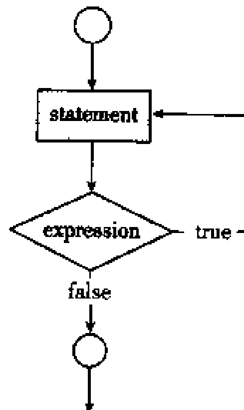


图 5.3 do…while 循环

在C++中，do是保留字。

在do…while循环结构中，首先执行语句statement，然后再计算表达式expression的值。如果表达式的值是true，再次执行语句statement。只要do…while语句中表达式expression的值是true，就会执行语句statement。为了避免无限循环，必须保证循环体中含有最终使表达式expression的值变为false的语句。

例5.12

```
i = 0;
do
{
   cout<<i<<" ";
   i = i + 5;
}
while(i <= 20);
```

这段代码的输出是：

```
0 5 10 15 20
```

在输出20以后，语句：

```
i = i + 5;
```

将变量i的值改为25，所以i <= 20变成false，结束该do…while循环。

因为while循环和for循环都有入口条件，所以如果该入口条件不能得到满足，那么就不会执行循环体。但是do…while循环，只有出口条件，所以不管怎样总会执行至少一次。

例5.13 考虑下面两个循环：

(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，while循环不会有任何输出。在(b)中，do…while循环将输出数字11。

do…while循环适合用于在循环体中语句执行以前不需要对条件进行检查的情形。例如，医院监视设备程序需要先读入数据后，然后将其与出口条件做比较。下面的程序说明了这种情况。

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;
int main ()
{
   int pressure;

   do
       cin>>pressure;
   while((pressure > low) && (pressure < high));

   cout<<"Help"<<endl;

   return 0;
}
```

三种循环结构在C++中各有各自的用处。虽然可以互相替代，但是这样做通常会显得十分笨拙。例如，可以使用do…while循环结构来改写程序AVG2（见例5.4）中的while循环结构。如果这样做，将会是以下形式：

```
if(number != -999)
  do
  {
     .
     .
     .
  }
  while(number != -999);
```

也就是用：

```
if(expression)
  do
    action
  while(expression);
```

来代替简单些的：

```
while(expression)
   action
```

5.8 break和continue语句

break和continue语句可以改变程序的执行顺序。switch结构中的break语句，可以使程序立即跳出该switch结构。同样，也可以在while，for和do…while循环中使用break语句。如果在循环结构中执行break语句，将导致程序立即跳出该循环结构。break语句主要有两个用途：

1. 提早跳出循环
2. 跳过switch结构的剩余部分

在执行完break语句后，程序将继续执行该结构后面的第一条语句。

在循环结构中使用break语句，就可以不必使用特定（标志）变量。下面的C++代码段中使用了这种方法（假定所有变量都经过正确定义）。

```
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;
   }
}
```

这里的while循环结构用来计算一些正数的和。如果输入的数据中含有负数，程序将终止运行并且输出相应的错误信息。本while循环结构中使用变量isNegative来标记输入的数字是否为负数。变量isNegative在while循环执行以前，初始化为false。在将num加到sum上之前，检查num是否为负数。如果num是负数，将在屏幕上输出错误信息，并且将isNegative的值设为true。在下一次循环中，while语句中表达式的值将是false，因为! isNegative的值是false（isNegative的值是true，故! isNegative的值是false）。

下面的while循环中没有使用变量isNegative：

```
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;
}
```

在这种形式的while循环里，如果发现输入数据中含有负数，if语句中表达式的值就变成true；在输出相应的错误信息后，break语句终止该循环（在执行完循环语句中的break语句后，该循环中的其他语句也将被略掉）。

continue语句被用在while，for和do…while循环结构中。当执行到循环中的continue语句后，程序将跳过该循环中的剩余语句，去执行下一次循环。在while和do…while循环结构中，在执行完continue语句后将立即计算循环条件表达式的值。在for结构中，在执行完continue语句后将立即执行更新语句，然后再执行循环条件。

在前面的程序段中，如果遇到了负数，整个while循环终止。如果要跳过该负数继续读入下一个数字，而不是终止整个while循环，则需要用continue语句代替break语句。见下面的例子。

```
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: 前面已经说明，这三种循环结构在C++中各有各的作用，并且通常可以互相替换。然而，continue语句的执行在while，do…while结构中与在for结构中略有不同。当在while和do…while结构中执行continue语句时，不会执行更新语句。而在for结构中执行continue语句时，将总会执行更新语句。

5.9 嵌套控制结构

本节将简要回顾在第4章和本章中到目前为止接触到的控制结构。前面已经介绍过，将一个控制结构置于另一个控制结构中，会有很重要的作用。嵌套结构使程序变得更加简练、高效。考虑下面程序:

```
#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 循环中不使用嵌套的 if 语句（只使用 if 语句序列，没有 else 语句或类似 switch 结构），那么程序将变成什么样呢？

考虑下面嵌套的例子。假设要创建下面的图案：

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

很明显，需要输出 5 行星号。第 1 行一个星号，第 2 行两个星号，以此类推。因为要输出 5 行星号，所以要使用下面的 for 语句：

```
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 i 的值是 2，以此类推。可以在嵌套的 for 循环中使用 i 作为限制条件，来控制每行中星号的个数。经过考虑，代码如下所示：

```
for(i = 1; i <= 5 ; i++)
{
   for(j = 1; j <= i; j++)
     cout<<"*";
   cout<<endl;
}
```

通过代码走查，可以看出：for 循环开始时，i = 1。当 i = 1 时，内部的 for 循环输出一个星号并将光标移到下一行。这时 i 变成 2，内部的 for 循环输出两个星号并将光标移到下一行，以此类推。这个过程一直持续到 i 变成 6，并停止循环。

如果将第一行 for 语句替换成下面的代码，将会输出何种图案呢？

```
for(i = 5; i >= 1; i--)
```

学习更多的嵌套 for 循环用法，请参见本章练习 30。

5.10 小结

1. 在 C++ 中有三种循环（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 是保留字。
4. 在 while 语句中，条件表达式 expression 外面的括号很重要，它们标志着表达式的开始和结束。
5. 动作语句 statement 称为循环体。
6. while 语句的循环体中，必须含有最终可以使条件表达式的值变为 false 的语句。
7. 计数器控制的 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，必须在循环之前将计数器变量初始化，而且在循环体中必须含有可以改变计数器变量值的语句。
9. 结束标记是标志输入数据结束的特殊值。结束标记必须与所有数据相似，但又必须与所有数据有区别。

10. 结束标记控制的 while 循环使用结束标记来控制 while 循环。在读到结束标记以前，while 循环将反复执行。
11. EOF 控制的 while 循环将反复执行，直到遇到文件结束标记为止。
12. 在 DOS 环境中，文件结束标记是 Ctrl+Z（按住 Ctrl 键的同时，按下 Z 键）。在 UNIX 环境中，文件结束标记是 Ctrl+D（按住 Ctrl 键的同时，按下 D 键）。
13. for 循环可以简化计数器控制的 while 循环。
14. 在 C++ 中，for 是保留字。
15. for 循环的语法是：

```
for(initialize statement; loop condition; update statement)
   statement
```

    这里的语句 statement，称为 for 循环的循环体。
16. 如果在 for 循环后面（循环体的前面）出现分号，则是语义错误。在这种情况下，for 循环的动作语句将是空语句。
17. do…while 语句的语法是：

```
do
      statement
while(expression);
```

    这里的语句 statement 称为 do…while 循环的循环体。
18. while 循环和 for 循环的循环体有可能不被执行，但是 do…while 循环的循环体至少被执行一次。
19. 在循环体中执行 break 语句，将导致立即终止该循环。
20. 在循环体中执行 continue 语句，程序将跳过该循环体的其余部分，并且转入下一次循环。
21. 当在 while 循环或者 do…while 循环中执行 continue 语句时，循环体中的更新语句将不会被执行。
22. 当在 for 循环中执行 continue 语句时，程序将转去执行更新语句。

5.11 练习

1. 判断下面说法的正误。
   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 循环的循环体很有可能一次也不执行。
   c. 在无限 while 循环中，while 条件表达式的初始值为 false。但是在第一次循环之后，它的值永远为 true。
   d. 当 J > 10 时，while 循环：

```
J = 0;
while(J <= 10)
  J++;
```

   将终止执行。
   e. 结束标记控制的 while 循环中实际上是事件控制的 while 循环，当遇到特殊值时终止执行。
   f. 循环是一种控制结构，这种结构可以使某些语句反复执行。
   g. 从一个未知长度的文件中读取数据，使用 EOF 控制的 while 循环是一个好的选择。
   h. 当 while 循环终止时，程序首先返回 while 循环前面的那条语句，然后再跳转到 while 循环后面的那条语句。

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，即循环终止时变量 ch 的值是多少？

```
ch = 'D';
while('A' <= ch && ch <= 'Z')
     ch = static_cast<char>(static_cast<int>(ch) + 1);
```

6. 假设输入的数据是38　45　71　4　-1。下面C++代码的输出是什么？假定所有变量都经过正确定义。

```
cin>>sum;
cin>>num;
for(j = 1; j <= 3; j++)
{
   cin>>num;
   sum = sum + num;
}
cout<<"Sum = "<<sum<<endl;
```

7. 假设输入的数据是38　45　71　4　-1。下面C++代码的输出是什么？假定所有变量都经过正确定义。

```
cin>>sum;
cin>>num;
while(num != -1)
{
   sum = sum + num;
   cin>>num;
}
cout<<"Sum = "<<sum<<endl;
```

8. 假设输入的数据是38　45　71　4　-1。下面C++代码的输出是什么？假定所有变量都经过正确定义。

```
cin>>num;
sum = num;
while(num != -1)
{
```

```
    cin>>num;
    sum = sum + num;
}
cout<<"Sum = "<<sum<<endl;
```

9. 假设输入的数据是38 45 71 4 -1。下面C++代码的输出是什么？假定所有变量都经过正确定义。

```
sum = 0;
cin>>num;
while(num != -1)
{
    sum = sum + num;
    cin>>num;
}
cout<<"Sum = "<<sum<<endl;
```

10. 改正下面程序代码的错误，使其计算10个数的和。

```
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. 假设输入的数据是：

```
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. 假定：

```
for(i = 12; i <= 25; i++)
    cout << i;
```

a. 输出的第 7 个整数是 ________。

b. 该语句共产生 ________ 行输出。

c. 如果将 i++ 写成 i--，将导致编译错误。这个说法正确吗？

14. 假定下面的代码被正确地置于程序中，说出它输出的内容和形式是什么？

```
num = 0;
for(i = 1; i <= 4; i++)
{
        num = num + 10 * (i - 1);
        cout<<num<<" ";
}
```

15. 假定下面的代码被正确地置于程序中，说出它输出的内容和形式是什么？

```
j = 2;
for(i = 0; i <= 5; i++)
{
        cout<<j<<" ";
        j = 2 * j + 3;
}
cout<<j<<" "<<endl;
```

16. 假定下面的代码被正确地置于程序中：

```
s = 0;
for(i = 0; i < 5; i++)
{
   s = 2 * s + i;
   cout<<s<<" ";
}
```

a. 变量 s 中的最终值是多少？

(i) 11 (ii) 4 (iii) 26 (iv) 以上都不是

b. 如果在 for 循环控制表达式的后面存在分号，那么变量 s 中的最终值是多少？

(i) 0 (ii) 1 (iii) 2 (iv) 5 (v) 以上都不是

c. 如果 for 循环控制表达式中 i 的初值是 5 而不是 0，那么变量 s 中的最终值是多少？

(i) 0 (ii) 1 (iii) 2 (iv) 以上都不是

17. 给出下面每一条语句的输出：

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. 编写一个程序，计算 1 到 100 中所有 3 的倍数的数的和。

19. 下面程序的输出是什么？

```
#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<<"OFTEN ";
             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. 假设输入的数据是 5　3　8。下面代码的输出是什么？假定所有变量都经过正确定义。

```
cin>>a>>b>>c;
for(j = 1; j < a; j++)
{
   d = b + c;
   b = c;
   c = d;
   cout<<c<<"  ";
```

```
}
cout<<endl;
```

21. 下面的 C++ 代码段的输出是什么？假定所有变量都经过正确定义。

```
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. 下面的程序中含有至少 5 个错误。请纠正这些错误。

```
#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. 下面各说法中哪些只对 while 循环适用？哪些只对 do…while 循环适用？哪些对两者都适用？

a. 被视为条件循环

b. 循环体至少执行一次

c. 在进入循环体之前，首先计算逻辑表达式的值

d. 循环体可以一次也不执行

24. 下面每个循环各会执行多少次？每个循环的输出结果是什么？

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. 下面的循环用来读入数字，直到遇到结束标记（本例中是-1）为止。程序将计算除了结束标记之外所有数字的和。如果输入的数据如下所示：

```
12  5  30  48  -1
```

下面的程序将不能完成上述的功能。请改正它。

```
#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. 如果输入的数据如练习25中所示，下面两段代码也不能完成设想的功能。请改正它。

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. 使用 while 循环结构和 do…while 循环结构改写下面的程序段，使其输出结果相同。

```
for(number = 1; number <= 10; number++)
    cout << setw(3) << number;
```

28. 使用 while 循环结构和 do…while 循环结构改写下面的程序段，使其输出结果相同。

```
j = 2;
for(i = 1; i <= 5; i++);
{
   cout<<setw(4)<<j;
   j = j + 5;
}
cout<<endl;
```

29. 下面程序的输出结果是什么？

```
#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. 对下面程序段做代码走查，弄清嵌套 for 循环的执行过程，并分别指出每段代码的输出结果。

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. 如果一个整数各位的数字之和可以被9整除，那么该整数就可以被9整除。编写一个程序提示用户输入一个整数。然后程序输出该整数并告之用户该整数是否可以被9整除。程序通过判断输入整数各位数字之和是否为9的整数倍来判断该整数是否可以被9整除。
2. 编写一个程序提示用户输入一个整数。程序将分别输出该整数每个位上的数字，并且输出这些数字的和。例如，程序输出整数3456每个位上的数字3 4 5 6；输出整数8030每个位上的数字8 0 3 0；输出整数2345526每个位上的数字2 3 4 5 5 2 6；输出整数4000每个位上的数字4 0 0 0；输出整数-2345每个位上的数字2 3 4 5。
3. 编写一个程序提示用户输入一个整数。程序将整数每个位上的数字完全颠倒过来。例如，如果输入的整数是12345，那么程序将输出54321。程序应将5000输出为0005，将980输出为089。
4. 重写例5.5程序电话数字。只用一个switch循环结构改写第10行到第28行中的语句，使程序可以计算出输入字母所对应的按键数字。
5. 电话数字程序只能将大写字母转换成相对应的按键数字。重写该程序，使其既可以将大写字母转换成相对应的按键数字，又可以将小写字母转换成相对应的按键数字。如果输入的字符既不是大写字母又不是小写字母，程序必须输出相应的错误信息。
6. 编写一个程序提示用户输入一些整数。该程序将分别计算出这些整数中奇数之和与偶数之和，并输出它们。

7. 编写一个程序提示用户输入一个正整数。程序将输出信息指出该正整数是否为质数（提示：偶数中只有2是质数；如果一个奇数不能被任何一个小于或等于它平方根的奇数整除，那么该奇数就是质数）。
8. 使用while循环编写程序，使其完成下面功能：
   a. 提示用户输入两个整数：firstNum和secondNum（firstNum一定要小于secondNum）。
   b. 输出所有界于firstNum和secondNum之间的奇数。
   c. 输出所有界于firstNum和secondNum之间偶数的和。
   d. 输出所有界于1到10之间的数字，以及它们的平方。
   e. 输出所有界于firstNum和secondNum之间奇数的平方和。
   f. 输出所有大写字母。
9. 使用for循环改写练习8。
10. 使用do…while循环改写练习8。
11. 出于有助于学生更好学习的需要，某大学的教务办公室需要知道每个男同学和女同学在每门课程中的成绩。所有男同学和女同学在这些课程中的GPA成绩都存在文件中。其中，字母m代表男同学，字母f代表女同学。文件中每一行只有一条记录，每一条记录都由字母和GPA成绩组成。文件中记录的条数未知。编写一个程序计算并输出全部同学（包括男同学和女同学）的平均GPA成绩。输出结果保留两位小数。
12. 如果利息按年增加，计算公式如下所示。其中p0是最初的账户余额，INT是年利息率，p1，p2和p3分别是第1年、第2年和第3年末的账户余额。则有：

```
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
```

以此类推。

假设不对退休金账户上的钱征收利息税。如果某人在16岁生日时以10%的利息率为自己存入一笔钱到这类账户，并且他以后没有再往里存钱或者从中取钱。在他60岁生日时，偶然地知道了这个账户。这笔钱以每年10%的速率增长。编写程序读入最初的账户余额，据此计算出在60岁生日时的账户余额是多少。

如果将这笔钱又存了一年。从61岁生日开始，决定提取这笔钱的年利息收益。也就是说，只提取利息，而不提取本金。在他的余生中，每个月可以从这笔钱中得到多少利息收益？

程序将读入整数数据，并用下面试验数据验证该程序的正确性：

$1 700，$3 600和$8 500

（注意：使用循环结构来计算最后的账户余额，而不要使用预定义函数pow）。

程序将做必要的计算，并输出以下数据：

16岁时的账户余额：________

利息率：________

60岁时的账户余额：________

61岁时的账户余额：________

每月的利息收入：________

第6章　用户自定义函数 I

本章要点：

- 了解标准（预定义）函数，并学会怎样在程序中使用它们
- 了解用户自定义函数
- 了解带有返回值的函数，包括函数的实参和形参
- 学会怎样创建并使用带有返回值的用户自定义函数

第2章已经描述过，C++程序其实就是由一系列函数组成的。main是其中的一个函数。第1章到第5章中只接触到了main函数；所有程序指令被包装在一个函数中。这种技术只适用于短小的程序。正如你将会见到的那样，对于大型程序来说，将所有的程序指令包装在一个函数中是不切实际的（虽然可能）。必须要将复杂问题拆分成若干小问题，使每个小问题都易于处理。本章将先介绍预定义函数，然后再介绍用户自定义函数。

让我们用汽车制造厂来打个比方。制造汽车时，使用的并不是生铁等原材料，而是将已经生产好的元部件组装起来。这些元部件有些是本工厂自己生产的，有些则是其他工厂生产的。

函数的作用正像这些元部件一样。使用函数可以将复杂程序拆分成若干个易于实现的子程序。在程序中使用的函数还有其他的优点：

1. 当编写某个子程序时，可以只专注于该子程序涉及到的问题，便于创建、调试和完善。
2. 可以由不同的人编写程序的不同部分，以便于分工协作。
3. 如果某个函数需要在程序不同部分中多次使用，或者在不同的程序中使用，可以只编写一次该函数，然后在需要时调用它。

函数通常也称为模块。函数很像是一些小程序；可以将它们组装成大程序。当介绍到用户自定义函数时，这一点就很明显了。而对于预定义函数来说，这一点并不是很明显，因为这些函数的代码对我们来说是不可见的。但是，因为预定义函数已经存在，所以先让我们学习这些函数，以便于在程序中使用它们。在程序中使用预定义函数，只需要知道怎样正确地使用它们即可。

6.1　标准（预定义）函数

在代数课程中，函数是由参数域和定义在这个参数域上的规则组成的。当选取某一参数时，函数值是惟一确定的。因此，如果 $f(x) = 2x + 5$，则：

```
f(1) = 7, f(2) = 9, f(3) = 11
```

这里1，2，3都是f的参数，而7，9，11则是对应函数f的值。

C++中的函数概念与代数中的函数概念相似。C++函数包括预定义函数（有时称为标准函数）和用户自定义函数两种。本节中将主要介绍预定义函数。一些预定义的数学函数有：abs(x)，sqrt(x)和pow(x, y)。

幂函数，pow(x, y)，用来计算 x^y。也就是说，pow(x, y) = x^y。例如，pow(2, 3) = 8.0，pow（2.5, 3）= 15.625。因为pow(x, y)的返回值是double类型，所以说pow是double类型函数。而且，x和y称为函数pow的参数。函数pow有两个参数。

平方根函数，sqrt(x)，用来计算非负数x（即x >= 0.0）的算术平方根。例如，sqrt(2.25)的值是1.5。sqrt是double类型函数，并且只有一个参数。

基数函数，floor(x)，将返回最大值不大于x的整数。例如，floor(48.79)的值是48.0。floor是double类型函数，并且只有一个参数。

在C++中，所有预定义函数都被分装在不同的库中。例如，头文件iostream中包含了I/O函数；头文件cmath中包含了数学函数。表6.1中列出了一些预定义函数，同时列出了包含该函数的头文件名、函数的作用、参数类型和函数类型。函数类型就是函数的返回值类型（附录F中列出了其他的预定义函数。附录E中列出了标准C++头文件）。

表6.1　预定义函数

函数名	标准头文件名	作用	参数类型	函数类型
abs(x)	<cstdlib>	返回参数的绝对值：abs(-7) = 7	int	int
ceil(x)	<cmath>	返回不小于x的最小整数：ceil(56.34) = 57.0	double	double
cos(x)	<cmath>	返回角度x的余弦值：cos(0.0) = 1.0	double（弧度）	double
exp(x)	<cmath>	返回e^x，其中e = 2.718：exp(1.0) = 2.71828	double	double
fabs(x)	<cmath>	返回参数的绝对值：fabs(-5.67) = 5.67	double	double
floor(x)	<cmath>	返回不大于x的最大整数：floor(45.67) = 45.00	double	double
pow(x,y)	<cmath>	返回x^y；如果x是负数，y必须是整数：pow(0.16，0.5) = 0.4	double	double
tolower(x)	<cctype>	如果x是大写字母，返回相应的小写字母；否则，返回x本身	int	int
toupper(x)	<cctype>	如果x是小写字母，返回相应的大写字母；否则，返回x本身	int	int

如果要在程序中使用预定义函数，必须使用include预处理指令将包含该函数的头文件包含到程序中。例如，要使用函数pow，程序中必须包含：

```
#include <cmath>
```

注意：在标准C++中，包含数学函数的头文件是math.h。

例6.1　本程序将说明预定义函数的使用方法。

```
//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;
}
```

输出

```
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
```

该程序的运行过程如下：第1行语句输出字母'a'相对应的大写字母'A'。注意函数toupper将返回int类型变量。因此，表达式toupper('a')的值是65，就是ASCII码中的'A'。为了输出字母A而不是65，需要在第1行语句中使用cast运算符。在第4行语句中，使用函数pow计算u^v。在C++术语中，这被称为调用带有参数u和v的pow函数。在这种情况下，u和v的值被传递给函数pow。其他语句的含义与之相似。

注意：如果在程序中使用标准C++头文件，需要将语句：

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

改写为：

```
#include <iostream.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
```

6.2 用户自定义函数

程序中函数的使用，极大地增强了程序的可读性。正如例6.1所示，通过使用函数大大地降低了main函数的复杂性。另外，如果某个函数通过调试，那么该函数就可以在程序中（或者不同的程序中）重复使用，而不需要重新编写该函数的代码。例如，在例6.1中，函数pow使用了三次。

C++并没有提供所有的所需函数，C++语言设计者也不可能知道用户的所有需求，所以必须学会编写自己的函数。

C++中的用户自定义函数分为两大类：

- 有数据类型的函数，称为带有返回值的函数（Value-returning Function）。
- 没有数据类型的函数，称为无返回值函数（Void Function）。

本章余下部分将讨论带有返回值的函数。带有返回值的函数中的诸多概念也同样适用于无返回值函数。第7章将讨论无返回值函数。

6.3　带有返回值的函数

前面一节介绍了一些预定义的 C++ 函数，如 pow，abs，islower 和 toupper。这些函数都是带有返回值的函数。为了能在程序中使用这些预定义函数，必须要知道包含这些函数的头文件名，而且还要使用预处理指令 include 将这些头文件包含到程序中。除此之外，还需要知道：

1. 函数名
2. 参数的个数（如果存在）
3. 每个参数的数据类型
4. 函数返回值的类型，也称为函数类型

因为函数返回值是惟一的，所以有三种很自然的方式来使用函数返回值：

- 存储返回值，便于以后进一步计算
- 在某些计算中直接使用返回值
- 输出返回值

这说明函数返回值不是用于赋值语句中，就是用于输出语句中，如 cout。也就是说，带有返回值的函数将用于表达式中。

在介绍带有返回值的用户自定义函数语法之前，让我们先来看一下关于函数的另外一个属性。函数除了上述 4 个属性之外，还有第 5 个属性（既包括带有返回值的函数，也包括无返回值的函数）。

5. 完成特定功能的程序代码

前面的 4 个属性称为函数的函数头（Function Header）；第 5 个属性称为函数的函数体（Function Body）。这 5 个属性合起来称为函数定义。例如，对于函数 abs 来说，函数头的形式是：

```
int abs(int number)
```

而函数 abs 的定义如下所示：

```
int abs(int number)
{
      if(number < 0)
         number = -number;
      return number;
}
```

在函数 abs 的函数头中定义的变量称为函数 abs 的形参（Formal Parameter）。因此，abs 的形参是 number。

例 6.1 的程序中含有三个使用函数 pow 的语句。在 C++ 术语中这称为函数 pow 被调用了三次。在本章的后面部分，将讨论在调用函数时会发生哪些事情。

假定 pow 函数的函数头是：

```
double pow(double base, double exponent)
```

从 pow 函数的函数头可知，pow 函数的参数是 base 和 exponent。考虑下面语句：

```
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1行中，调用了函数pow，并且该函数的参数是u和v。这里，变量u和v中的值传递给函数pow。实际上，程序将u中的值拷贝到base中，将v中的值拷贝到exponent中。第1行pow函数中的变量u和v称为该函数调用的实参。在第2行中，调用了函数pow，并且该函数的参数是2.0和3.2。在这次函数调用中，数值2.0被拷贝到base中，数值3.2被拷贝到exponent中。而且，在这次pow函数的调用中，实参分别是2.0和3.2。同样，在第3行中，pow函数的实参是u和7。变量u中的值将被拷贝到base中，7.0将被拷贝到exponent中。

形参　函数头中定义的参数。
实参　函数调用时变量或表达式。

对于预定义函数，只需要关心前4个属性。软件供应商不会给出实际的源代码，也就是函数体。否则，软件的成本将会很高。

语法：带有返回值的函数
带有返回值的函数的语法是：

```
functionType functionName(formal parameter list)
{
      statements
}
```

在语法中，functionType是函数返回值的类型，也被称为带有返回值函数的函数类型。用花括号括起来的语句构成了函数的函数体。

语法：形参列表（formal parameter list）
形参列表的形式是：

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

函数调用
调用带有返回值函数的语法是：

```
functionName(actual parameter list)
```

语法：实参列表（actual parameter list）
实参列表的语法是：

```
expression or variable, expression or variable, ...
```

因此，调用带有返回值的函数，必须给出函数名和括号内的实参列表（如果存在）。

函数的形参列表可以为空。但是，即使形参列表为空，括号仍然是必需的。在形参列表为空的情况下，带有返回值函数的函数头的形式为：

```
functionType functionName()
```

或者：

```
functionType functionName(void)
```

如果形参列表为空，在函数调用中实参列表也应该为空。在这种情况下（即形参列表为空），在实际的函数调用中，仍然需要写出括号。因此，形参列表为空的带有返回值函数的调用形式是：

```
functionName()
```

在函数调用中，实参的个数及类型应该与形参的个数及类型相匹配。也就是说，实参与形参应该一对一地匹配（第 7 章中将讨论默认参数）。

前面已经提过，应该在表达式中调用带有返回值的函数。表达式既可以是赋值语句的一部分，也可以是输出语句的一部分。如果在程序中调用某个函数，那么该函数的函数体就会被执行。

6.3.1　return 语句

带有返回值的函数通过 return 语句将计算结果返回给调用它的表达式。

语法：return 语句

return 语句的句法是：

```
return expr;
```

这里，expr 可以是变量、常量或者是表达式。程序首先计算 expr 的值，然后将该值返回。expr 的值的类型必须与函数类型相同。

在 C++ 中，return 是保留字。

当执行到函数中的 return 语句时，该函数立即终止执行，并将程序的控制权返回给调用函数。因此，如果执行到 main 函数中的 return 语句，则整个程序将终止。

为把上述讨论应用到实际中去，让我们编写一个函数来计算两个数中的最大值。因为函数要比较两个数的大小，所以该函数有两个参数，并且这两个参数都是数字。假定这两个数字的数据类型是浮点数——也就是 double 类型。函数的名字叫 larger。现在惟一要做的事情就是编写该函数的函数体。因此，根据函数的语法，编写的函数如下所示：

```
double larger(double x, double y)
{
    double max;
    if(x >= y)
      max = x;
    else
      max = y;

    return max;
}
```

也可以将该函数写成如下形式：

```
double larger(double x, double y)
{
    if(x >= y)
      return x;
    else
      return y;
}
```

第一种形式的函数 larger 需要定义另外的变量 max（称为局部变量，这里的 max 对于函数 larger 来说是局部的）；而第二种形式则不需要这样。

注意：1. 在函数定义中，x 和 y 是形参。

2. return 语句可以出现在程序中的任何位置。但需要注意的是，一旦执行 return 语句，该语句后面的所有语句都将被略掉。因此，最好是在函数计算出返回值后立即使用 return 语句将其返回给调用函数。

既然已经编写好larger函数，下面的C++代码将说明怎样在main函数中使用该函数。

```
int main()
{
   double one, two, maxNum;                        //Line 1

   cout<<"Larger of 5 and 6 is "
       <<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)，是函数调用，其中5和6是实参。

2. 在第6行中的表达式larger(one, two)，是函数调用，其中one和two是实参。
3. 在第7行中的表达式larger(one, 29)，也是函数调用，其中one和29是实参。
4. 在第8行中的表达式larger(38.45, 56.78)，是函数调用，其中38.45和56.78是实参。在这条语句中，函数larger的返回值被赋给变量maxNum。

注意： 在函数调用中，只需要给出实参，而不需要指定它们的数据类型。

一旦函数被编写出来，就可以在程序中的任何地方使用它。函数larger的功能是比较两个数字，并返回最大的数字。现在让我们动手编写计算三个数字最大值的函数，该函数的名字为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之间的最大值，然后再将其与x做比较。最后，return语句将返回第二次比较后的最大值。在函数头中，x，y和z都是形参。

6.3.2 函数原型

既然已经知道怎样编写以及在程序中使用函数，下一个需要弄清的问题就是用户自定义函数应该在程序中出现的位置。例如，函数larger应该放在main函数的前面还是后面？函数larger应该放在函数compareThree的前面还是后面？与变量必须在使用前定义很相似，如果要在main中使用larger，逻辑上需要把larger放在main前面。

但在实际编程中，C++程序员习惯将main函数置于所有自定义函数的前面。而这种组织结构会导致编译错误，因为编译器是以函数在源程序中出现的次序为顺序对这些函数进行编译的。例如，如果将函数main置于larger之前，那么在编译main时，标识符larger是未定义的。为了避免出现这种未定义标识符的编译错误，可以将函数原型置于所有函数定义（包括main函数）之前。

函数原型（Function Prototype） 没有函数体的函数头。

语法：函数原型

带有返回值函数的函数原型的语法是：

```
functionType functionName(parameter list);
```

(注意在函数原型后面有分号)。

函数 larger 的函数原型是:

```
double larger(double x, double y);
```

注意:在函数原型中,可以不必指出参数列表中的变量名。但是,必须要指出每个参数的数据类型。

函数 larger 的函数原型还可以写成:

```
double larger(double, double);
```

最终程序

到现在为止,完全可以编写出一个完整的程序,并将其编译和运行。下面的程序中使用函数larger, compareThree 和 main,来计算两个(三个)数字中的最大值。

```
//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));
}
```

程序运行结果 在本程序运行中,用户输入的数据加有阴影。

```
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：在本程序中，函数larger和compareThree的函数原型出现在main函数的前面。因此，可以在程序中的任何位置上使用函数larger和compareThree。

注意：带有返回值的函数使用return语句来返回函数值。考虑下面的return语句：

```
return x, y;//only the value of y will be returned
```

这是一个合法的return语句。你或许会认为该return语句将返回x和y的值，然而实际情况并非如此。需要注意的是，return语句将只返回一个值，即使它包含一个以上的表达式。如果return语句中包含一个以上的表达式，函数将只返回最后一个表达式的值。因此，在上面的return语句中，将返回y的值。下面的程序进一步说明这个问题。

```
//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

```
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
```

在例6.2的程序中的函数，将返回布尔类型的数据。

例6.2 回文数字

本例中的函数，isNumPalindrome，用来判断输入的整数是否为回文数字。如果输入的整数是回文数字，返回true；否则，返回false。如果一个数字从正的方向读和从反的方向读的结果是相同的，那么该数字就是回文数字（Palindrome Number）。例如，整数5，44，434，1881和789656987都是回文数字。

假设num是整数。如果num < 10，那么它一定是回文数字，所以该函数将返回true，现在再来考虑num >= 10的情况。判断num是否是回文数字，首先要比较num中的第一位数字和最后一位数字。如果第一位数字和最后一位数字不相同，那么它就不是回文数字，函数将返回false。如果相同，去掉num中的第一位数字和最后一位数字，将新的数字存储到变量num中。然后再对num中的新值重复上述过程，直到num >= 10为止。

例如，假设输入的数字是18281。因为该数字中第一位数字和最后一位数字相同，所以去掉第一位数字和最后一位数字后，得到新值828。再将上述过程应用于数字828上。第一位数字和最后一位数字再次相同。在去掉数字828第一位数字和最后一位数字后，只剩下了数字2，小于10。因此，18281是回文数字。

为了去掉num的第一位数字和最后一位数字，首先需要知道不大于该数字的10的最高次方数，称为pwr。不大于18281的10的最高次方数是4，也就是，pwr是4。18281 % 10^{pwr} = 8281，于是第一位数字就被去掉了。同样，8281 / 10 = 828，最后一位数字也被去掉了。因此，可以使用求余运算符将第一位数字去掉，这里除数是10^{pwr}。为了去掉最后一位数字，可以将该数字除以10。在下一次循环中，将pwr中的值减少2。经过上面讨论，得到下面的算法。

1. 如果num < 10，该数字是回文数字，函数将返回true。
2. 假设num是整数，并且num >= 10。为了判断num是否是回文数字：
   a. 找到不大于num的10的最高次方数pwr。例如，不大于434的10的最高次方数是2；不大于789656987的10的最高次方数是8。
   b. 当num大于或等于10时，比较num的第一位数字和最后一位数字。

b.1. 如果num的第一位数字和最后一位数字不同，num不是回文数字，返回false。

b.2. 如果num的第一位数字和最后一位数字相同：

b. 2.1. 去掉num的第一位数字和最后一位数字。

b. 2.2. 将pwr减2。

c. 返回true。

根据上面的算法，编写程序如下：

```
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中，头文件cmath中的函数pow用来计算不大于num的10的最高次方数pwr。因此，必须保证在程序中包含头文件cmath。

6.3.3 执行过程

前面已经说明，C++程序是函数的集合。注意，函数可以按任意的顺序在程序中出现。但是需要记住的是，所有标识符必须在定义之后才能使用。编译器按照从上到下的顺序编译整个程序。因此，如果main函数出现在所有用户自定义函数之前，那么它将被首先编译。然而，如果main函数出现在程序的尾部（或中间），所有在main函数之前出现的函数将按照先后顺序在main函数之前编译。

函数原型出现在所有的函数定义之前，所以编译器将首先编译它们。这样，编译器才能正确地解释函数调用。然而在程序运行时，无论main函数出现在程序中的什么位置，计算机总是首先执行main函数中的第一条语句。其他的函数只有在调用时才被执行。

函数调用将程序的控制权转交给被调用函数中的第一条语句。通常，在被调用函数最后一条语句执行完后，控制权立即返回给调用函数中的调用语句。如果是带有返回值的函数还要返回返回值。因此，对于带有返回值的函数来说，在函数执行完后，控制权返回调用函数，函数的返回值将取代函数调用语句。程序将在函数调用点后继续执行。

6.4　程序范例：最大值

在本程序范例中，函数larger用来找出一系列数字中的最大值。为了简便起见，程序将从10个数字中找出最大值。经过简单修改，该程序就可以处理任何个数的数字。

输入　10 个数字。

输出　10 个数字中的最大值。

问题分析和算法设计

假设输入的数据是：

```
15 20 7 8 28 21 43 12 35 3
```

首先读入第1个数字。因为这是到目前为止读入的惟一一个数字，可以假定它就是最大值，并存储到变量max中。读入第2个数字，并将其存储到变量num中。现在比较max和num，将两者中的最大值存储到max中。这时，max中的值就是前两个数字中的最大值了。读入第3个数字。将其与max进行比较，两者中的最大值存储到max中。这时，max中的值就是前三个数字中的最大值。读入下一个数字，将其与max比较，两者中的最大值存储到max中。重复这一过程，直到所有数字都被处理完为止。经过上述讨论，得出以下算法：

1. 读入第1个数字。因为这是到目前为止读入的惟一一个数字，所以它就是当前的最大值。将其存储到变量max中。
2. 对于剩下的每一个数字：
   a. 读入下一个数字。将它存储到变量num中。
   b. 比较num和max。如果max < num，那么num就是新的最大值，所以将max中的值更新为num中的值。如果max >= num，则丢弃num中的值；也就是，什么也不做。
3. 因为现在max中的值是最大值，所以输出max。

为了计算两个数字中的最大值，本程序中使用了函数larger。

完整的程序代码清单

```
//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;
}
```

程序运行结果 *在本程序运行中，用户输入的数据加有阴影。*

```
Enter 10 numbers.
10 56 73 42 22 67 88 26 62 11
The largest number is 88
```

6.5 程序范例：有线电视公司的计费程序

在第4章中已经编写过有线电视公司的计费程序。在那个程序中，所有的程序指令都被封装在main函数中。现在，我们使用用户自定义函数来重写该程序，这将有助于进一步理解结构化程序的设计。问题分析的过程将重点说明怎样将复杂程序分解成小的子程序。它同时说明在解决特定的子问题时，可以只将注意力放在问题的某个部分上。

本程序的输入输出与前面程序相同。

问题分析和算法设计

既然有两种类型的客户：家庭客户和商业客户，程序中应该包含两个独立的函数：一个用来计算家庭客户的账单，另一个用来计算商业客户的账单。这两个函数将分别计算两种类型的账单总额，并将其返回给main函数。main函数负责输出账单总额。可以通过调用函数residential来计算家庭用户的账单，通过调用函数business来计算商业用户的账单。计算这两种账单的公式跟第4章程序中的一样。

如同在第4章程序中，诸如家庭用户账单处理费、家庭用户基本服务费等数据，定义为命名常量。

函数residential 为了计算家庭用户账单，必须知道客户租用的频道数量。根据租用的频道数量，可以计算出账单总额。在计算账单总额后，该函数将使用return语句返回账单总额。该函数可以用下面4个步骤来描述该函数：

a. 提示用户输入租用的频道数量
b. 读入租用的频道数量
c. 计算账单
d. 返回账单总额

该函数中（步骤a）提示用户输入租用的频道数量，然后读入租用的频道数量（步骤b）。计算账单所需的其他费用项目，诸如基本服务费和账单处理费定义为命名常量（在函数main定义的前面）。因此，在计算账单总额的过程中，该函数不需要在main函数那里得到任何数据。因此，该函数没有参数。

局部变量（函数residential） 由前面的讨论可知，函数residential中需要变量来存储租用的频道数和账单总额。因此，需要在该函数中定义两个局部变量：

```
int noOfPChannels;      //number of premium channels
double bAmount;         //billing amount
```

函数 residential 的定义为：

```
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 为了计算商业用户账单，需要知道接入的节点数目和租用的频道数目。根据这些数据可以计算出账单总额，该函数使用return语句返回账单总额。可以用下面6个步骤来描述该函数：

a. 提示用户输入接入的节点数目
b. 读入接入的节点数目
c. 提示用户输入租用的频道数目
d. 读入租用的频道数目
e. 计算账单
f. 返回账单总额

该函数中（步骤a和步骤c）提示用户输入接入的节点数目和租用的频道数目，然后读入接入的节点数目和租用的频道数目（步骤b和步骤d）。计算账单所需的其他费用项目，诸如基本服务费和账单处理费定义为命名常量（在函数main定义的前面）。因此，在计算账单总额的过程中，该函数不需要在main函数那里得到任何数据。因此，该函数没有参数。

局部变量（函数business） 由前面的讨论可知，函数business中需要变量来存储接入的节点数目、租用的频道数目和账单总额。因此，需要在该函数中定义三个局部变量：

```
int noOfBasicServiceConnections;
int noOfPChannels;      //number of premium channels
double bAmount;         //billing amount
```

函数 business 的定义为：

```
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. 通过控制符 fixed 和 showpoint，将输出的浮点数格式设置为固定小数点、带有小数点和小数部分尾部补 0。
2. 通过控制符 setprecision，将输出的浮点数格式设置为两位小数。
3. 提示用户输入账号。
4. 读入账号。
5. 提示用户输入客户类别。
6. 读入客户类别。
7. a. 如果客户类别是 R 或者 r：
   (1) 调用函数 residential 来计算账单
   (2) 打印账单
   b. 如果客户类别是 B 或者 b：
   (1) 调用函数 business 来计算账单
   (2) 打印账单
   c. 如果客户类别不是 R，r，B，b，则输出错误信息。

完整的程序代码清单

```
//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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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：如果使用标准C++头文件，需要将下面语句：

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

替换为：

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

同样，需要将下面语句：

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

替换为：

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

6.6 小结

1. 函数很像是小型程序，又称为模块。
2. 使用函数可以将大程序分解成易于处理的小程序。
3. C++系统提供了标准（预定义）函数。
4. 使用标准函数，必须：

(1) 知道包含该函数的头文件名

(2) 将头文件包含在程序中

(3) 知道函数的名称和类型，参数的个数和类型

5. 有两种类型的用户自定义函数：带有返回值的函数和无返回值的函数。
6. 定义在函数头中的变量称为形参。
7. 在函数调用中的表达式、变量和常量值称为实参。
8. 在函数调用中，实参的个数和类型必须和形参的个数和类型相匹配。
9. 调用函数，必须提供该函数的函数名和实参列表。
10. 带有返回值的函数将返回返回值。因此，带有返回值的函数可以用在表达式和输出语句中。
11. 用户自定义函数的语法是：

```
functionType functionName(formal parameter list)
{
      statements
}
```

12. functionType functionName(formal parameter list)称为函数头。在大括号中括起来的语句称为函数体。
13. 函数头和函数体合起来称为函数定义。
14. 虽然函数可以没有参数，但是函数头和函数调用时的括号却是必需的。
15. 可以通过使用空括号或者在括号中指定 void 类型来说明形参列表为空。
16. 带有返回值的函数通过 return 语句返回返回值。
17. 在一个函数中可以有多于一个的return语句。然而，如果在函数中执行return语句，那么剩下的语句将被略掉，并且立即跳出函数。
18. return 语句只能返回一个值。
19. 函数原型是没有函数体的函数头；函数原型以分号为结束标志。
20. 函数原型声明函数类型以及函数中参数的个数和类型。
21. 在函数原型中，形参列表中的变量名称是可选的。
22. 函数原型帮助编译器解释每一个函数调用。
23. 在程序中，函数原型置于所有函数定义之前，包括 main 函数定义。
24. 如果在程序中使用函数原型，那么用户自定义函数可以出现在程序中的任何位置。
25. 程序运行时，总是从 main 函数中的第一条语句开始执行。
26. 用户自定义函数只有在被调用时才能执行。
27. 函数调用将控制权从调用函数传递给被调用函数。
28. 在函数调用语句中，只需要给出实参，而不需要指定参数类型和函数类型。
29. 当程序从被调用函数退出时，控制权将返回给调用函数。

6.7 练习

1. 判断下面语句的正误。

   a. 在程序中使用标准函数，只需要知道函数名和怎样使用该函数。

   b. 带有返回值的函数只能有一个返回值。

   c. 可以在每次函数调用中使用不同的参数值。

   d. 当在用户自定义函数中使用 return 语句时，函数立即终止执行。

   e. 带有返回值的函数只能返回整数值。

2. 下面C++程序的输出结果是什么（注意函数sqrt的功能是计算平方根。例如，sqrt(16.0) = 4.0。函数sqrt包含在头文件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. 在下面的函数头中哪些是合法的？如果不合法，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. 考虑下面语句：

```
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);
```

下面语句中哪些是合法的？如果不合法，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. 考虑下面函数：

```
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;
}
```

下面各程序段的输出结果是什么？

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. 考虑下面函数原型：

```
int test(int, char, double, int);
double two(double, double);
char three(int, int, char, double);
```

回答下面问题：

a. 函数 test 中有几个参数？函数 test 的类型是什么？

b. 函数 two 中有几个参数？函数 two 的类型是什么？

c. 函数 three 中有几个参数？函数 three 的类型是什么？

d. 调用函数test共需要几个实参？每个参数的数据类型是什么？调用函数test时，这些参数的使用顺序是什么？

e. 编写 C++ 语句输出函数 test 的返回值，实参是 5，5，7.3 和 'z'。

f. 编写 C++ 语句输出函数 two 的返回值，实参是 17.5 和 18.3。

g. 编写 C++ 语句输出函数 three 返回的字符（使用自己给定的实参）。

7. 考虑下面函数：

```
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);
}
```

下面 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. 考虑下面函数：

```
int secret(int one)
{
    int I;
    int prod = 1;

    for(I = 1; I <= 3; I++)
          prod = prod * one;
    return prod;
}
```

a. 下面C++语句的输出是什么？

(1) cout << secret(5) << endl;

(2) cout << 2 * secret(6) << endl;

b. 函数secret的作用是什么？

9. 下面程序的输出是什么？

```
#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. 下面程序的输出是什么？

```
#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6.2中所给函数isNumPalindrome（回文数字）编写程序。使用下面数字来验证程序的正确性：10，34，22，333，678，67876，44444和123454321。
2. 编写带有返回值的函数，isVowel。如果给定的字母是元音字母，该函数返回true；否则，返回false。
3. 编写程序提示用户输入一系列字符，并且输出其中元音字母的个数（使用编程练习题2中的函数isVowel）。
4. 考虑下面程序：

```
#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的函数定义。如果x大于y，返回x与y的和；否则，返回x减去2倍y。

b. 编写函数two的函数定义。

(1) 读入一个数字并将它存到变量 z 中。

(2) 通过将 a 中的值加到 z 上来更新 z 中的值。

(3) 将 one 函数调用的返回值赋给变量 first，其中 one 函数调用的参数是 6 和 8。

(4) 通过将 x 中的值加到 first 上来更新 first 中的值。

(5) 如果 z 大于 2 倍的 first 的值，返回 z；否则，返回 2 倍 first 减去 z 的差。

c. 编写 C++ 程序验证 a 和 b 中的函数（如果需要，可以在 main 函数中定义其他变量）。

5. 编写一个函数。该函数读入一个整数，然后将这个整数的每个位上的数字按相反的顺序输出。例如，reverseDigit(12345)的值是 54321。

6. 下面的公式用来计算笛卡儿(Cartesian)平面上的两点：(x_1, y_1)和(x_2, y_2)间的距离：

$$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

给定圆心和圆上一点的坐标，利用上述公式可以计算出圆的半径。编写程序提示用户输入圆心和圆上一点的坐标，然后该程序输出圆的半径、直径、周长和面积。程序中至少要包含下面函数：

a. distance：该函数有 4 个参数，这 4 个参数是平面上两个点的坐标，该函数将返回两点之间的距离。

b. radius：该函数有 4 个参数，这 4 个参数是圆心和圆上一点的坐标。该函数调用函数 distance 计算圆的半径，然后返回圆的半径。

c. circumference：该函数有一个参数，这个参数是圆的半径。该函数返回圆的周长（如果 r 是半径，那么周长就是 $2\pi r$）。

d. area：该函数有一个参数，这个参数是圆的半径。该函数返回圆的面积（如果 r 是半径，那么面积就是 πr^2）。

这里 π= 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，下面公式用来计算年末的估计人口数量：

$$p+\frac{B*P}{100}-\frac{D*P}{100}$$

人口自然增长率的计算公式是：

$$B-D$$

编写一个程序，该程序提示用户输入年初的人口数量、出生率和死亡率、年数 n。程序计算并输出第 n 年后的估计人口数量。程序中要包含下面函数：

a. growthRate：该函数有出生率和死亡率两个参数，将返回人口自然增长率。

b. estimatedPopulation：该函数有当前人口数量、人口自然增长率和年数 n 三个参数。它返回第 n 年后的估计人口数量。

程序将不接受负的出生率、负的死亡率、小于 2 的人口数量。

8. 重写第 4 章中编程练习第 12 题中的程序（移动电话服务供应商客户账单）。新程序使用下面的函数来计算客户账单（在本编程练习中，不必输出通话的分钟数）。

a. regularBill：该函数计算并且返回普通服务的账单。

b. premiumBill：该函数计算并且返回优惠服务的账单。

第7章　用户自定义函数Ⅱ

本章要点：

- 理解如何在程序中构建和使用无返回值函数
- 理解值参数和引用参数的区别
- 了解引用参数和带有返回值的函数
- 理解标识符的作用域
- 了解局部变量和全局变量的区别
- 了解静态变量
- 了解函数重载
- 了解带有默认参数的函数

在第6章中，已经为大家介绍了带有返回值的函数的使用方法。在本章中，将了解到用户自定义函数，特别是没有数据类型的C++函数，称为无返回值函数（Void Functions）。

7.1　无返回值函数

无返回值函数的结构与带有返回值函数的结构相似，都由函数头和函数体两部分组成。用户自定义函数既可以置于main函数之前，也可以置于main函数之后。但是，程序总是从main函数中的第一条语句开始执行。如果将用户自定义函数置于main函数之后，那么必须在main函数之前提供该函数的函数原型。因为无返回值函数的函数头部分中没有指定该函数类型，所以这种函数的函数体中的return语句没有意义。然而在无返回值函数中，可以使用不带值（没有返回值）的return语句来提前结束函数。与带有返回值的函数相同，无返回值函数可以带有形参也可以没有形参。

因为无返回值函数没有返回值，所以不能将其用于表达式中。调用无返回值函数的语句是一个单独的语句。因此，应该通过函数名和实参（如果需要），在一条单独语句中调用无返回值函数。

7.1.1　不带参数的无返回值函数

本节中将讨论不带形参的无返回值函数。

函数定义

定义不带形参的无返回值函数的语法是：

```
void functionName (void)
{
     statements
}
```

括号里面的void是可选的。在C++中，void是保留字。

注意： *与变量命名一样，也应该给函数名赋予有意义的名字。因此，在命名函数时，应该挑选有意义的名字。*

函数调用

函数调用的语法是:

```
functionName();
```

因为无返回值函数没有参数,所以不能将任何值传递给该函数(除非使用全局变量,将在本章的后面部分中介绍)。这种函数特别适合用于显示程序中的信息。考虑下面的程序(参见例7.1)。

例7.1 假定需要打印下面的标语来宣传春季销售大减价活动(相似的问题见第2章编程练习7)。

```
******************************
******************************
********** Annual  ***********
******************************
******************************
******* Spring Sale **********
******************************
******************************
```

这个标语的前两行是星号。在输出含有文字Annual的星号行后,再打印两行星号。在输出含有文字Spring Sale的星号行后,还需要再打印两行星号。可以编写单独的函数来打印两行星号,然后在需要时调用它。完整的程序代码是:

```
#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,调用函数printStars输出前两行星号。第2行语句输出含有文字Annual的一行星号。在第3行中,再次调用函数printStars输出两行星号。在第4行语句输出含有文字Spring Sale的一行星号,这也是第6行输出。第5行语句,再次调用函数printStars输出两行星号。

注意：main函数中的语句printStarts( );是函数调用。

在上面的程序中，可以使用下面的函数来取代函数printStars：

```
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
    }
}
```

在函数定义中，外部的for循环（第1行）执行了两次。每执行一次外部for循环（第1行），就执行30次内部的for循环（第2行），结果是输出一行30个星号。第3行语句输出每个星号。第4行语句将光标移到输出设备中下一行的开始处。因为外部for循环执行两次，本函数将输出两行星号，每行中有30个星号。

很明显，在函数定义中使用for循环的printStars函数，比前面的那种printStars函数更容易修改。例如，如果要输出5行星号而不是2行星号，那么只需要将（在第1行的外部for循环中的）2改成5就可以了。同样，如果每行要输出40个星号而不是30个星号，那么只需要将（在第2行的内部for循环中的）30改成40就可以了。而且你很快就会发现，在定义中使用for循环的printStars函数，经过简单的修改就可以建立起与调用函数（例如函数main）的数据通信联系。

在前一个程序中，函数printStars将总是输出两行星号，每行中有30个星号。现在假设要输出下面的图案（星号三角形）：

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

假设要编写类似于printStars那样的函数。然而，如果该图案中有20行星号，那么在函数printStars中就要有20行输出语句。而且每次调用printStars时，该函数都将输出相同行数的星号；这种函数的灵活性很差。然而，如果能告诉函数printStars要输出多少行星号，那么函数的灵活性将大大增强。必须在调用函数与被调函数之间建立起数据通信联系。

7.1.2　带有参数的无返回值函数

前面一节讨论了不带参数的无返回值函数，并且指出了这种函数的局限性。而且，不带参数的无返回值函数既不能传入信息也不能传出信息。可以通过使用参数来建立调用函数和被调函数之间的联系。本小节将讨论带有参数的无返回值函数。

函数定义

定义带有参数的无返回值函数的语法是：

```
void functionName(formal parameter list)
{
     statements
}
```

形参列表(Formal Parameter List)

形参列表的语法是：

```
dataType& variable, dataType& variable, ...
```

在形参列表中，必须同时指出数据类型和变量名称。在dataType后面的&号有特别的意义。

函数调用

函数调用的语法是：

```
functionName(actual parameter list);
```

实参列表(Actual Parameter List)

实参列表的语法是：

```
expression or variable, expression or variable, ...
```

与带有返回值的函数一样，在函数调用中实参的个数和数据类型必须与形参的个数和数据类型相匹配。实参和形参必须一一对应。函数调用将使被调函数的函数体得以执行（带有默认参数的函数将在本章后面讨论）。

例7.2

```
void funexp(int a, double b, char c, int& x)
{
    .
    .
    .
}
```

函数funexp有4个参数。

参数提供了调用函数（例如main函数）和被调函数之间的数据通信联系。它使函数在每次被调用时都可以带有不同的数据。通常有两种类型的形参：值参数（Value Parameter）和引用参数（Reference Parameter）。

值参数 接收相应实参内容的拷贝的形参。

引用参数 接收相应实参位置（内存地址）的形参。

如果在函数的形参列表中将&置于形参类型后面，那么该类型后面的变量就成为引用参数。

例7.3

```
void expfun(int one, int& two, char three, double& four);
```

函数expfun有4个参数：(1) one是int类型值参数；(2) two是int类型引用参数；(3) three是char类型值参数；(4) four是double类型引用参数。

在考虑带有参数的无返回值函数范例前，让我们先来弄清值参数和引用参数的区别。从值参数的定义可知，如果形参是值参数，那么相对应的实参值将被拷贝到形参中。也就是说，值参数有自己数据拷贝。因此，在函数执行过程中，值参数（形参）使用自己内存空间中的数据进行计算。在得到数据拷贝以后，值参数与实参再也没有任何联系。

在形参是引用参数的情况下，形参接收的是相应实参的地址。也就是说，引用参数中存储的是实参的内存地址。在函数执行过程中，程序使用存储在引用参数中地址所对应的内存空间中的实参来进行计算。对引用参数所做的任何改动将立即导致引用参数所对应的实参的值发生变化。例7.5

和例7.6清楚地说明了在程序执行过程中值参数和引用参数的区别。注意，第6章中的例题中只用到了值参数。

我们现在来编写C++程序来输出前面小节中的图案（星号三角形）。

例7.4 下面的C++程序输出星号三角形：

```
//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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
Enter the number of star lines (1 to 20) to be printed-> ■
```

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

在本程序中，main 函数中的语句（见第 10 行）：

```
printStars(numberOfBlanks, counter);
```

是函数调用。标识符 numberOfBlanks 和 counter 都是实参。

函数printStars的执行过程如下。函数printStars有两个参数。函数printStars的作用是输出一行星号，并且星号前面有一定数量的空格。每一行中空格的个数和星号的个数是通过参数传递给函数printStars的。第一个参数blanks，用来指定输出的空格个数。第二个参数starsInLine，用来指定每行中输出的星号个数。例如，如果参数blanks的值是30，那么函数printStars中第一个for循环（第13行）将执行30次，所以输出30个空格。同样，因为要在星号前输出空格，所以函数printStars中第二个for的每一次循环（第15行）都将输出' *'（第16行）——也就是一个空格和一个星号的串。在函数main中，用户先要输入需要输出的星号行数（第1行）（在本程序中，限定最多可以输出20行星号）。因为程序只限定输出20行，所以第3行到第6行中的while循环保证输入的行数在1和20之间。

函数 main 中的 for 循环（第 9 行）调用函数 printStars（第 10 行）。这个 for 的每一次循环通过实际变量 numberOfBlanks 和 counter 来指定每一行中应该输出的空格个数和星号个数。每次调用函数 printStars 都将比上次调用该函数减少一个空格并增加一个星号。例如，main 函数中 for 的第一次循环，指定输出 30 个空格和 1 个星号（通过参数 numberOfBlanks 和 counter 传递给函数 printStars）。然后，for 循环：

- 将空格的数量减 1。这是通过第 11 行语句来实现的：

  ```
  numberOfBlanks--;
  ```

- 在for循环的末尾，将下一次循环中输出的星号个数加1。这是通过执行第9行for语句中的更新语句，counter++ 来实现的。该语句将变量 counter 的值加 1。

也就是说，第二次调用函数 printStars 时，参数是 29 个空格和 2 个星号。

7.1.3 引用参数

在数据拷贝之后，值参数和实参不再有任何联系。所以不能通过使用值参数来改变调用函数中实参的值。在被调函数的执行过程中，对形参的任何改动都不会影响到实参，实参也不会知道形参做了哪些改动。因此，不能通过无返回值函数的值参数将任何信息传递到函数外面。无返回值函数的值参数只在形参和实参之间提供了单向联系。因此，使用值参数的函数有很大的局限性。另一方面，因为引用参数接受的是地址（实参的内存地址），所以引用参数可以向调用函数返回一个或者多个值，并且可以改变实参中的值。

在下面三种情形中，引用参数显得十分有用：

- 要从函数中返回多于一个值
- 实参值本身需要改动
- 传递地址可以节省拷贝大量数据所需的内存空间和时间

前两种情形将出现在本书后面的各个章节中。在第9章和第12章介绍完数组和类之后，将介绍第三种情形。

注意，如果将 & 置于函数形参列表中的形参类型之后，该类型后面的变量就变成了引用参数。

注意：（常量引用参数）可以通过关键字const将引用参数（形参）定义为常量。第12章将讨论常量引用参数。在此之前讨论的引用参数都是本章中定义的非常量（non-constant）引用参数。从引用参数的定义不难看出，常量值和表达式值不能传递给非常量引用参数。因此，如果形参是非常量引用参数，那么在函数调用中，与它相对应的实参必须是变量。

例7.5　考虑下面的程序。假定学生的课程成绩是由课程分数（值在0和100之间）决定的。本程序有三个函数：main，getScore和printGrade。这些函数的定义如下所示：

1. main
   a. 读入课程分数
   b. 输出课程成绩
2. getScore
   a. 提示用户输入课程分数
   b. 读入课程分数
   c. 输出课程分数
3. printGrade
   a. 计算课程成绩
   b. 输出课程成绩

完整的程序代码如下所示：

```
//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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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
```

本程序的运行过程如下：本程序的第1行输出1行提示信息（见程序运行结果）。第2行语句调用函数 getScore，实参是 courseScore（main 函数中定义的变量）。因为函数 getScore 的形参 score 是引用参数，所以程序将变量 courseScore 的内存地址（即内存中位置）传递给 score。因此 score 与 courseScore 引用相同的内存位置。对 score 的值做的修改也就是对 courseScore 的值做的修改。

调用 getScore 后，程序控制权将传递给函数 getScore，即执行第4行语句。该语句输出第2行信息（见程序运行结果），提示用户输入课程分数 score。第5行语句将用户输入的分数（在本程序运行结果中是85）存储到变量 score 中。这时，变量 score 和 courseScore 中的值同为85。接下来，第6行语句输出 score 中的值，见程序运行结果。在第6行语句执行之后，程序控制权返回给函数 main。接下来执行第3行语句。该语句调用函数 printGrade，实参是 courseScore。因为函数 printScore 的形参是值参数，所以程序将相应实参 courseScore 中的值拷贝到参数 score 中。因此，score 中的值是85。在将 courseScore 中的值拷贝到 score 中之后，score 和 courseScore 之间再也不存在任何联系。现在程序执行第7行语句，该语句输出第4行信息（在程序运行结果中标记为第7行）。第8行中的 if…else 语句计算并输出课程成绩。因为第7行输出语句中不含有换行符和控制符 endl，所以该 if…else 语句的输出结果在同一行（第4行输出）中（见程序运行结果）。在 if…else 语句执行之后，程序的控制权返回给函数 main。因为接下来要执行的语句是函数 main 中的最后一条语句，return 语句，所以程序运行终止。

在本程序中，函数 main 首先调用函数 getScore，该函数读入用户输入的课程分数。函数 main 接下来调用函数 printGrade，该函数根据课程分数计算并输出课程成绩。函数 getScore 读入课程分数，然后函数 printGrade 根据该分数，计算课程成绩。因为程序后面要用到函数 getScore 中读入的成绩，所以函数 getScore 必须将该数值传递到函数外面。因此，存储该数值的形参应该是引用参数。

下面是值参数和引用参数属性的总结。

7.2　值参数、引用参数和内存地址

当调用函数时，程序将在函数数据区域内为该函数的形参和函数体中定义的变量（称为局部变量）分配存储空间。注意，如果是值参数，程序将实参的值拷贝到相应的形参的内存单元中；如果是引用参数，程序将实参的地址传递给相应的形参。也就是说，形参的值是内存地址。程序使用形参值所指向的内存地址中的数据来进行计算。也就是说，在函数参数是引用参数的情况下，实参和形参指向相同的内存地址。因此在程序运行过程中，对形参值所做的修改也同时反映在实参上。

注意： 流变量（例如 ifstream 和 ofstream）应该作为引用参数传递给函数。这样，在打开、读取或者输出流文件后，可以将输入/输出流的状态传递到函数外面。

因为参数传递是任何程序设计语言的基本部分，所以例 7.6 到例 7.8 将进一步阐述该概念。

例 7.6　下面程序说明了值参数和引用参数的使用方法：

```
//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
```

让我们对该程序进行代码走查。为了引述方便，程序中每一行都有编号。在每一条语句执行前/后，都将输出变量的当前值。

在第1行语句执行前，程序只为main函数中的变量分配了内存空间，变量没有经过初始化。在第3行语句执行后，各变量中的值如图7.1所示。

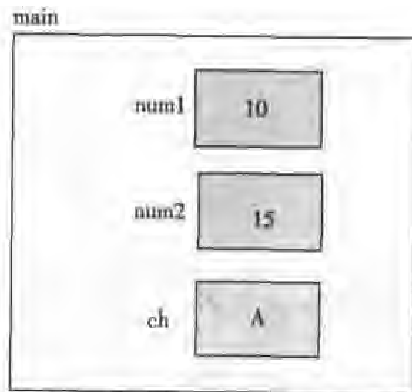


图7.1 第3行语句执行后各变量中的值

第4行的输出结果如下所示：

```
line 4: Inside main: num1 = 10; num2 = 15; and ch = A
```

第5行中的语句调用函数funOne。在函数funOne中有三个参数和一个局部变量。程序给函数funOne中的参数和局部变量分配内存空间。因为形参b是引用参数，它将接收相应实参num2的内存地址；另外两个形参是值参数，所以将拷贝相应的实参的值。在第9行语句执行之前，各变量中的值如图7.2所示。

在第9行语句（one = a;）执行后，各变量中的值如图7.3所示。

在第10行语句（a++;）执行后，各变量中的值如图7.4所示。

在第11行语句（b = b * 2;）执行后，各变量中的值如图7.5所示（注意变量b改变了num2中的值）。

在第12行语句（v = 'B';）执行后，各变量中的值如图7.6所示。

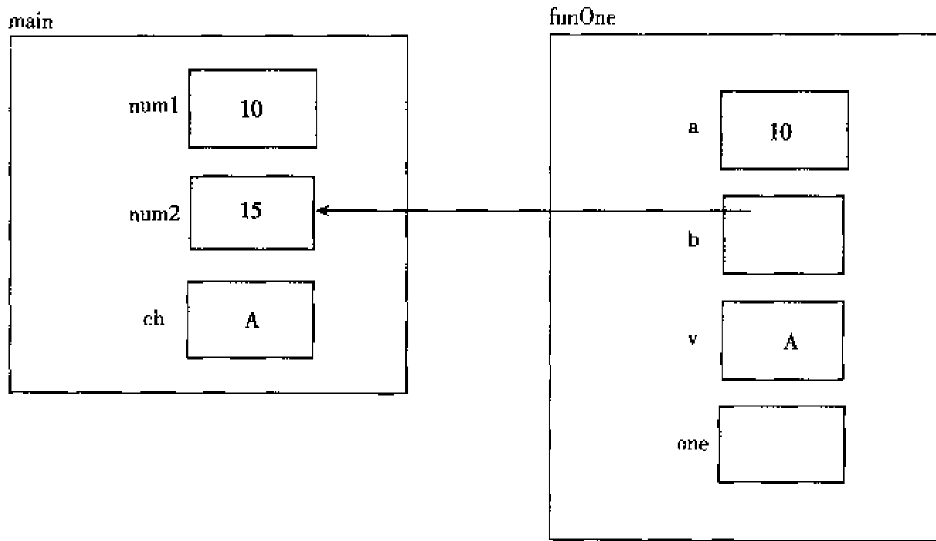


图7.2　第9行语句执行之前各变量中的值

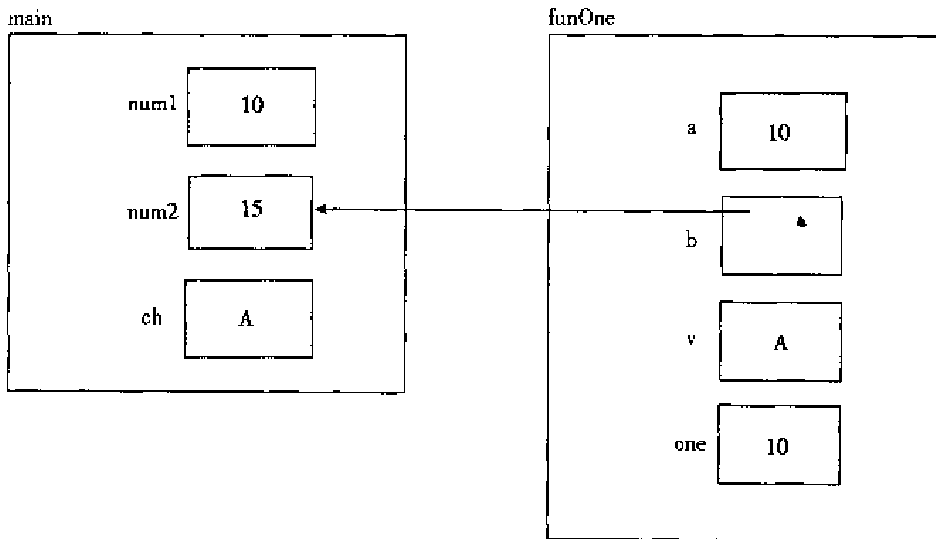


图7.3　第9行语句执行后各变量中的值

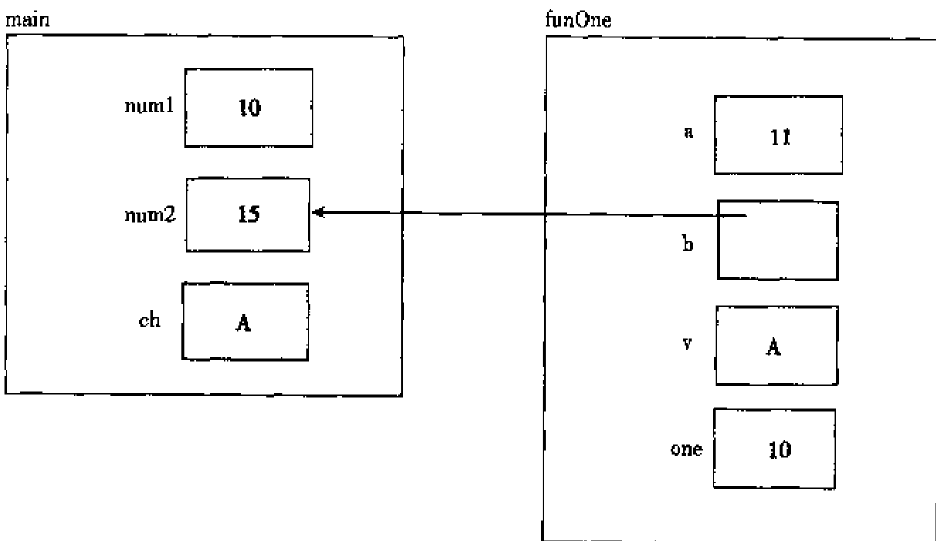


图7.4　第10行语句执行后各变量中的值

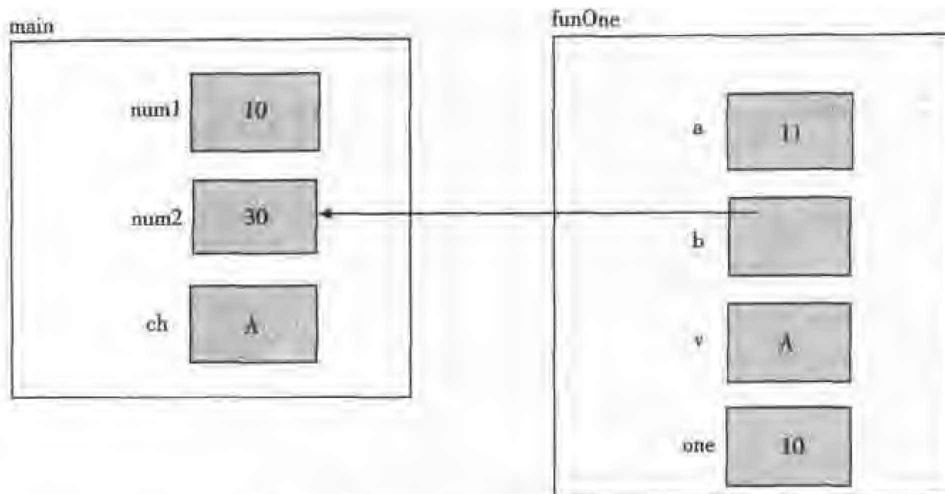


图 7.5 第 11 行语句执行后各变量中的值

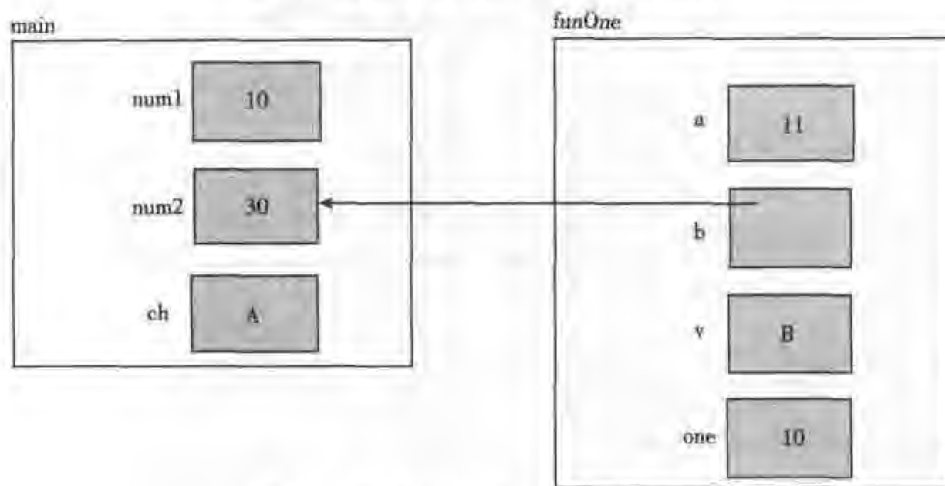


图 7.6 第 12 行语句执行后各变量中的值

第 13 行语句输出：

```
line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10
```

第 13 行语句执行后，程序控制权返回到第 6 行中，并且释放所有分配给函数 funOne 中变量的内存空间。此时，函数 main 中各变量的值如图 7.7 所示。

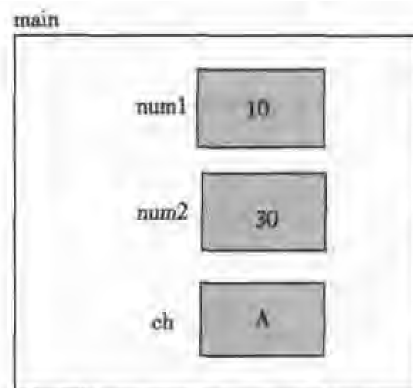


图 7.7 当程序控制权返回到第 6 行语句后各变量中的值

第6行语句输出：

```
line 6: After funOne: num1 = 10, num2 = 30, and ch = A
```

第7行语句调用函数funTwo。函数funTwo中有三个参数：x，y和w。这里x和y是引用参数，w是值参数。因此，x接收相应的实参num2的地址；w接收相应的实参ch的地址；变量y将数值25拷贝到它的内存单元中。在第14行语句执行之前，各变量中的值如图7.8所示。

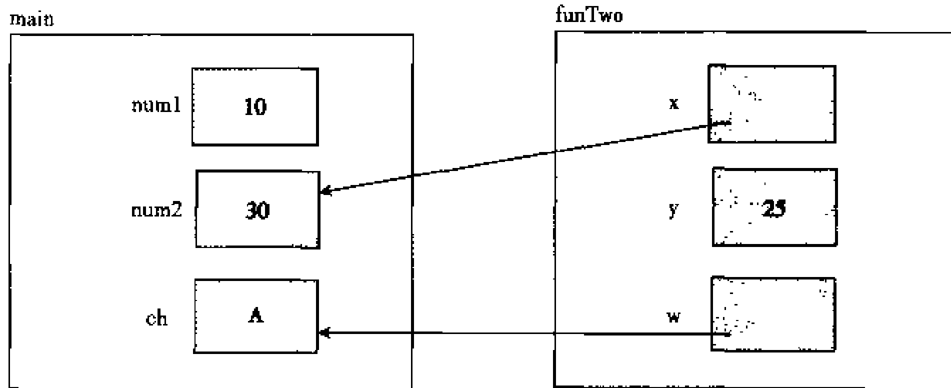


图7.8　第14行语句执行之前各变量中的值

在第14行语句（x++;）执行以后，各变量中的值如图7.9所示（注意变量x改变num2中的值）。

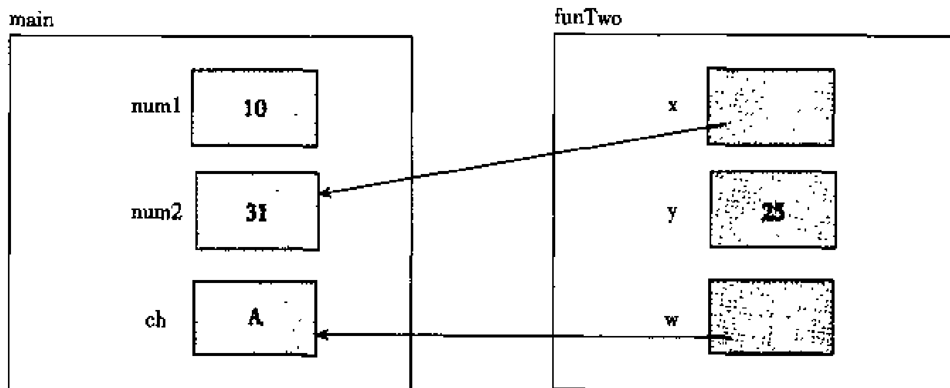


图7.9　第14行语句执行之后各变量中的值

在第15行语句（y = y * 2;）执行后，各变量中的值如图7.10所示。

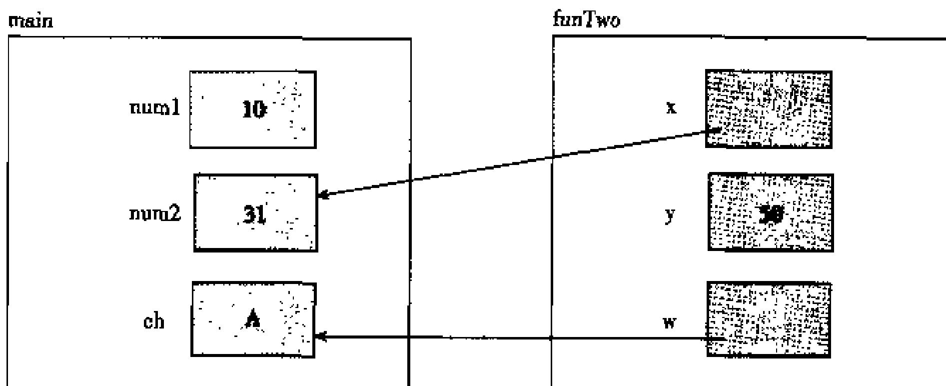


图7.10　第15行语句执行后各变量中的值

在第 16 行语句（w = 'G';）执行后，各变量中的值如图 7.11 所示（注意变量 w 改变了 ch 中的值）。

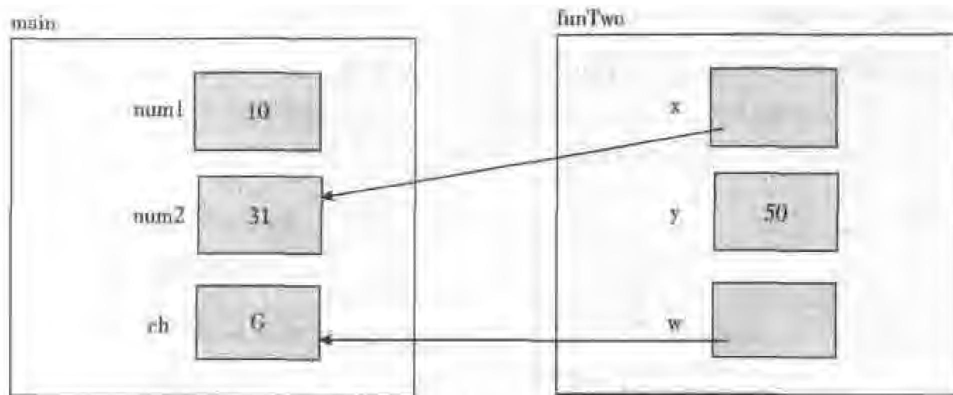


图 7.11 第 16 行语句执行后各变量中的值

第 17 行语句输出：

```
line 17: Inside funTwo: x = 31, y = 50, w = G
```

第 17 行语句执行后，程序控制权返回到第 8 行中，并且释放所有分配的内存。给函数 funTwo 中变量的内存空间。此时，函数 main 中各变量的值如图 7.12 所示。

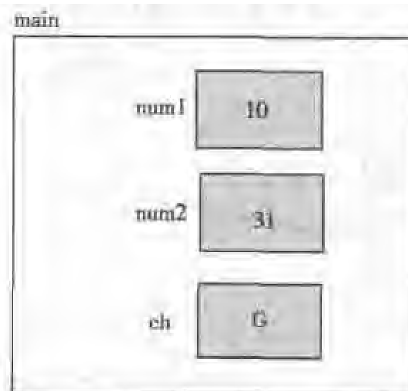


图 7.12 当程序控制权返回到第 8 行后各变量中的值

第 8 行语句输出：

```
line 8: After funTwo: num1 = 10, num2 = 31, and ch = G
```

在第 8 行语句输出之后，程序终止。

例 7.7 本例将说明怎样在函数中使用引用参数。

```
//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中的两个参数都是引用参数，函数doubleFirst中的两个参数都是值参数。函数main（第2行）中的语句：

```
addFirst(num, num);
```

将num的引用同时传递给函数addFirst两个形参first和second。这是因为该函数的两个形参所对应的实参完全相同。也就是说，变量first和second指向同一个num的内存单元。图7.13说明了这种情况。

对变量first所做的任何修改将立即反映到second和num中。同样，对变量second所做的任何修改也将立即反映到first和num中。这是因为这三个变量指向相同的内存单元（注意num中的初始值是5）。

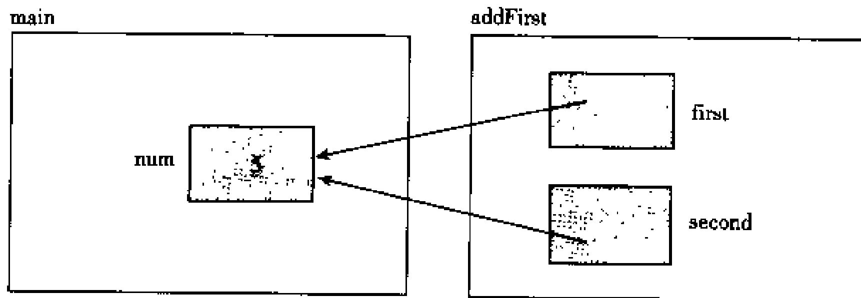


图7.13 函数addFirst的参数

函数doubleFirst的形参都是值参数。所以函数main（第4行）中的语句：

```
doubleFirst(num, num);
```

将num中的值同时拷贝到one和two中。这是因为该函数的两个形参所对应的实参完全相同。图7.14说明了这种情况。

因为one和two都是值参数，所以对one中的值所做的修改不会影响到two和num中的值。同样，所以对two中的值所做的修改也不会影响到one和num中的值（注意，在函数doubleFirst执行之前，num中的值是14）。

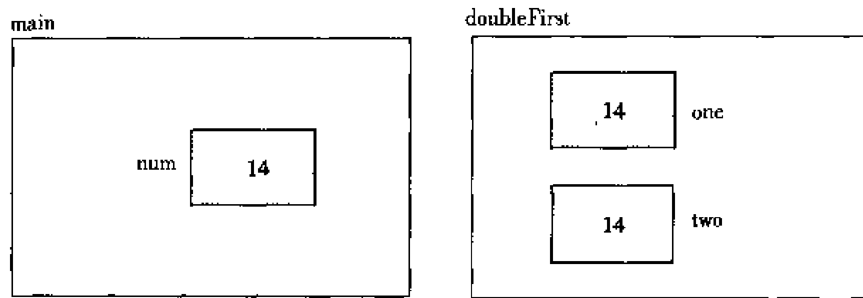


图7.14 函数doubleFirst的参数

函数squareFirst的形参ref是引用参数，val是值参数。变量ref接收的是相对应实参num的内存地址；而变量val拷贝的是相对应实参num中的值。num和ref指向相同的内存单元。图7.15说明了这种情况。

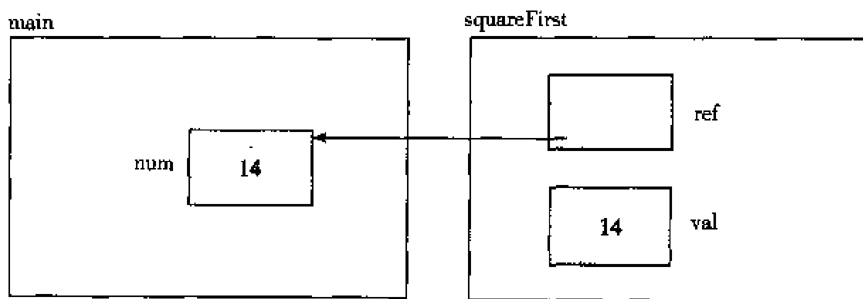


图7.15 函数squareFirst的参数

对ref所做的任何修改将立即反映到num上，而对val所做的任何修改将不会影响到num（注意，在函数squareFirst执行之前，num中的值是14）。

建议读者对例7.7中的程序进行代码走查。程序中每行语句都带有编号，以便于引述。输出结果显示了程序中各语句的执行顺序。

例7.8 本例说明了定义在所有函数之外的变量的使用方法。该例同时也说明了当一个用户自定义函数的引用参数传递给另一个函数时，引用参数对相对应的实参的影响。

```
//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;                                              //Line 1
   num2 = 20;                                              //Line 2
   t = 15;                                                 //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
```

本程序中的变量t定义在所有函数的前面。因为在任何函数中都没有定义标识符t，所以t在程序中的任何位置上都可以使用。该程序由两个无返回值函数组成：funOne，有两个形参，并且都是引用参数；funTwo，有一个引用参数和一个值参数。

在第5行中，函数main调用函数funOne，实参是num1和t。因此，funOne中形参a，接收的是num1的地址；形参x，接收的是t的地址。对变量a所做的任何修改将立即影响到num1；对变量x所做的任何修改也将立即影响到t。因为t在整个程序中都是可用的，所以函数funOne第16行中的语句可以直接修改t中的值（见第17行输出结果）。所以，既可以通过t的引用修改t中的值，也可以直接修改t中的值。

在第7行中，函数main调用了函数funTwo，实参是num2和num1。num2的地址传递给引用参数u，所以u和num2指向相同的内存单元。因为num1要传递给v，而且v是值参数，所以num1中的值被拷贝到v中。

在第20行中，函数funTwo调用函数funOne。传递给funOne的实参分别是aTwo（函数funTwo中的局部变量）和u（函数funTwo的形参）。因为函数funOne的两个参数a和x都是引用参数，所以a和x接收的都是相对应实参的地址。现在，函数funTwo中的参数u传递给x。因为u本身是引用参数，并指向num2，所以u和x同时指向num2。对x的任何修改将立即反映到num2中。也就是说，u，x和num2将指向相同的内存单元。

本例的输出结果将再次说明各语句的执行顺序。

7.3 引用参数和带有返回值的函数

在第6章中讨论带有返回值的函数的时候，只介绍了怎样使用值参数。引用参数也可以使用在带有返回值的函数中，虽然这种方法并不是推荐使用的。根据定义，带有返回值的函数只能返回一个值；这个值是通过return语句返回的。如果函数需要返回多个值，应该使用无返回值函数，并且通过相应的引用变量返回这些值。

7.4 标识符的作用域

第6章和本章前面几节的许多程序范例中，都出现了用户自定义函数。标识符可以定义在函数头中、块结构中或者块结构外。这里自然会遇到一个问题：在程序中的任何位置上，都可以使用这些标识符吗？答案是否定的。标识符的使用必须遵守一定的规则。标识符作用域（Scope）指的是程序中标识符的

有效使用范围。注意，标识符是C++中变量或者函数的名字。本节将讨论标识符的作用域。首先，让我们先定义两个在本书中广泛使用的术语：

局部标识符（Local Identifier） 在函数（或块结构）中定义的标识符

局部标识符不可以使用在函数（或块结构）的外部。

全局标识符（Global Identifier） 在所有函数外面定义的标识符

在C++中，不可以嵌套定义函数。也就是说，不可以在一个函数内部定义另一个函数。

通常，标识符的使用规则如下所示：

1. 在函数或者块结构中使用全局标识符（如变量）的条件是：
   a. 该标识符定义在函数定义或者块结构之前
   b. 函数名称与该标识符名称不相同
   c. 函数中所有参数名称与该标识符名称不相同
   d. 所有的局部标识符名称（例如局部变量）与该标识符名称不相同
2. （嵌套块结构中）在一个块中定义的标识符可以使用在：
   a. 从块结构中定义该标识符的位置开始，一直到块结构结束的部分
   b. 该部分的嵌套块结构中，这些嵌套块结构中的标识符名称必须与该标识符名称不相同
3. 函数名的作用域规则与定义在任何函数（块结构）外的标识符的作用域规则相同。也就是说，函数名的作用域规则与全局变量的作用域规则相同。

在使用范例解释作用域规则之前，先来关注定义在for语句中的标识符的作用域。C++允许程序员在for结构的初始化语句中定义变量。例如，在下面的for语句：

```
for(int count = 1; count < 10; count++)
    cout << count << endl;
```

定义了变量count，并将其初始化为1。变量count的作用域只限定在for的循环体内部。

注意： 上面提到的定义在for语句中的变量作用域规则不适用于标准C++。在标准C++中，定义在初始化语句中的变量的作用域是：从该变量定义开始，一直到直接包含该for循环的函数（块结构）结束为止（为确保正确使用，请参阅编译器文档）。

下面的C++程序有助于理解标识符的作用域：

```
#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域（可见性）。

表7.1　标识符的作用域（可见性）

标识符	在one中的可见性	在two中的可见性	在three中的可见性	在four中的可见性	在main中的可见性
rate（在main前）	Y	Y	Y	Y	Y
z（在main前）	Y	Y	N	N	N
t（在main前）	Y	Y	Y	Y	Y
main	Y	Y	Y	Y	Y
main的局部变量	N	N	N	N	Y
one（函数名）	Y	Y	N	N	Y
x（one的形参）	Y	N	N	N	N
y（one的形参）	Y	N	N	N	N
w（在函数two前）	N	Y	Y	Y	N
two（函数名）	Y	Y	Y	Y	Y

（续表）

标识符	在one中的可见性	在two中的可见性	在three中的可见性	在four中的可见性	在main中的可见性
a（two的形参）	N	Y	N	N	N
b（two的形参）	N	Y	N	N	N
x（two的形参）	N	Y	N	N	N
two的局数变量	N	Y	N	N	N
three（函数名）	Y	Y	Y	Y	Y
one（three的形参）	N	N	Y	Y	N
y（three的形参）	N	N	Y	Y	N
z（three的形参）	N	N	Y	Y	N
ch（three的局部变量）	N	N	Y	Y	N
a（three的局部变量）	N	N	Y	N	N
x（four的局部变量）	N	N	N	Y	N
a（four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中包含一个名为one的形参。同样，在函数three中的块结构four中也不能使用int类型变量a，因为块结构four中含有一个名为a的标识符。

1. 第2章中已经提过，C++不能自动地对变量进行初始化。然而，某些编译器可以将全局变量初始化为自己定义的默认值。例如，某些编译器将int，char和double类型的全局变量初始化为0。
2. 在C++中，::被称为作用域运算符（Scope Resolution Operator）。通过作用域运算符，在函数（块结构）中可以使用定义在该函数（块结构）前面的全局变量，即使该函数（块结构）中已经含有与之同名的标识符。例如，在上面的范例中，通过使用作用域运算符，函数main可以使用::z来引用全局变量z。同样，假设全局变量t定义在函数（例如funExample）的前面。函数funExample可以通过使用作用域运算符来引用全局变量t，即使在函数funExample中已含有一个名为t的标识符。通过作用域运算符，函数funExample通过::t来引用全局变量t。同样，在上面的范例中，通过使用作用域运算符，函数three可以调用函数one。
3. C++还允许在函数中使用定义在函数后面的全局变量。在这种情况下，函数中必定不能含有与该全局变量同名的标识符。在上面的范例中，全局变量w定义在函数one的后面。因为函数one中不含有名为w的标识符，所以在函数one中可以使用w，但是需要将w说明为外部变量（External Variable）。在函数one中将w说明为外部变量的语句是：

```
extern int w;
```

在C++中，extern是保留字。在上面语句中，extern说明w是一个定义在程序某处的外部变量。因此，虽然函数one中说明了变量w，但是在调用函数one的时候，并不真正给w分配存储空间。在C++中，外部变量说明还有其他用途，但在本书中不做介绍。

7.5 全局变量的副作用

C++程序可以使用全局变量。但是，必须要清楚使用全局变量的副作用。使用全局变量的函数独立性差，通常不能使用在其他的程序中。而且，如果多个函数都使用到某个全局变量，一旦出现差错，就很难发现问题是由哪个函数引起的。在程序中的某个部分引起全局变量的错误，很容易误以为是由另一部分引起的。本书建议读者尽量不要使用全局变量，而是使用适当的参数。

7.6 静态变量和自动变量

到目前为止讨论的变量，遵守两个简单的规则：

1. 为全局变量分配的存储空间在程序执行过程中始终存在。
2. 在函数（块）中定义的变量，在执行到该函数（块）时分配空间，在执行结束时释放所占用的空间。例如，当调用某函数时，为其形参和局部变量分配存储空间；在函数执行完毕后，释放所分配的存储空间。

如果一个变量的存储空间，在函数（块）执行时分配，在结束时释放，那么该变量就称为自动变量（Automatic Variable）。如果一个变量的存储空间在程序执行过程中始终存在，那么该变量就称为静态变量（Static Variable）。全局变量是静态变量，默认时定义在函数（块）中的变量是自动变量。也可以通过使用保留字 static，将定义在函数（块）中的变量指定为静态变量。定义静态变量的语法是：

```
static dataType identifier;
```

语句：

```
static int x;
```

将 x 定义为 int 类型静态变量。

定义在函数（块）中的变量是该函数（块）的局部变量，作用域仅限于本函数（块）内部。

静态变量通常在定义时初始化。语句：

```
static int x = 0;
```

将 x 定义为 int 类型变量，并初始化为 0。

例 7.9 下面程序将说明静态变量的使用方法：

```
//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 static 变量，初始值是 0；y 是自动变量，初始值是 10。函数 main 调用函数 test 5 次。每次调用函数 test 时，都为变量 y 分配存储空间，并且在函数执行结束时释放该空间。因此，每次调用函数 test 时，输出的 y 的值都相同。但是，因为 x 是静态变量，所以分配给 x 的存储空间在整个程序执行过程中始终存在。变量 x 只初始化一次，值是 0，以后每次调用函数 test 时，都是使用 x 中的当前值。

因为分配给静态变量的存储空间在整个程序执行过程中始终存在，所以在程序中的同一函数的多次调用中可以使用同一个静态变量的值。尽管全局变量也可以用来在同一函数的多次调用之间共享数据，而局部的静态变量却可以防止别的函数使用其中的数据。

在介绍另一些程序范例之前，需要注意与函数有关的另一个概念：函数重载（Function Overloading）。

7.7 函数重载简介

注意：*在第 13 章之前，本节中介绍的知识并不是必需的。所以，在阅读第 13 章之前，可以略过本节。*

在 C++ 中，许多函数可以使用同一个名字。在 C++ 术语中，这被称为函数名重载。因此，重载函数意味着创建许多同名函数。然而，如果许多函数有同一个名字，则每个函数必须有不同的形参列表。执行时提供的实参列表决定实际调用的函数。

假定要编写判断两个数中最大值的函数。这两个数可以分别是整数、浮点数、字符或者字符串。可以分别编写如下所示的函数：

```
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 用来判断两个整数中的最大值，函数 largerChar 用来判断两个字符中的最大值，其他函数的功能类似。所有函数都执行相似的操作。除了给每个函数起不同的函数名以外，还可以给这几个函数起相同的函数名，比方说 larger。也就是说，可以重载函数 larger。因此，可以将前面的函数原型改写成：

```
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，如果函数调用语句是 larger(5, 3)，则执行第一个函数；如果函数调用语句是 larger('A', '9')，则执行第二个函数。

函数重载用于在不同类型数据集上进行相似的操作。当然，必须分别给出这些函数的定义。

7.8 带有默认参数的函数

注意：*在第 12 章之前，本节讲述的内容不是必需的，可以略过。*

本节将讨论带有默认参数的函数。前面已经讲过，实参的个数必须与形参的个数相同。但是，在带有默认参数的函数里，C++ 允许实参个数少于形参个数。可以在函数名第一次出现在程序里的时候，例如在函数原型中，指定默认参数。通常，使用带有默认参数的函数时，应该遵守下面的规则：

- 如果函数的某个参数有默认值，而在函数调用时又没有给该参数指定新值，那么该参数将使用其默认值。
- 所有的默认参数必须出现在函数参数列表中的最右边。
- 如果一个函数有多于一个的默认参数，而且在函数调用时对其中某个有默认值的参数没有指定新值，那么在实参列表中必须略掉该参数右边的所有的参数值。
- 默认值可以是常量、全局变量或函数调用。
- 在函数调用时，可以对有默认值的参数指定新值，而不使用它们的默认值。
- 不能将常量指定为引用参数的默认值。

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，其中参数z，u，v和w是默认参数。如果在函数调用时没有对z，u，v和w指定值，那么它们将使用默认值。

假设有下面的变量定义：

```
int a, b;
char ch;
double d;
```

下面的函数调用都是合法的：

```
1.funcExp(a, b, c);
2.funcExp(a, 15, 34.6, 'B', 87, ch);
3.funcExp(b, a, 14.56, 'D');
```

在语句1中，使用了z，u，v和w的默认值。在语句2中，'B'取代了z的默认值，87取代了u的默认值，ch取代了v的默认值，参数w使用了自己的默认值。在语句3中，'D'取代了z的默认值，u，v和w分别使用了自己的默认值。

下面的函数调用都是非法的：

```
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的值被略掉，所以它后面所有的值必须要都被略掉。在语句2中，因为v的值被略掉，所以w的值也要被略掉。

下面是一些非法的带有默认参数函数的函数原型：

```
1.void funcOne(int x, double z = 23.45, char ch, int u = 45);
2.void 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有默认值，所以在z后面的所有参数都应该有默认值。在语句2中，因为第一个参数有默认值，所以所有参数都应该有默认值。在语句3中，因为y是引用参数，所以常量不能赋给y。

例7.10进一步说明了带有默认参数函数的使用方法。

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函数置于所有的用户自定义函数之前，所以需要在函数原型中指定自定义函数的默认参数值。并且，只需要在函数原型中指定一次（不需要在函数定义中指定）。

7.9 程序范例：数字分类

在本范例中，使用用户自定义函数来改写第5章中的程序。该程序用来计算输入的整数序列中的奇数个数和偶数个数。

主要算法与前面相同：

1. 将变量 zeros，odds 和 evens 初始化为 0。
2. 读入一个数字。
3. 如果该数字是偶数，将偶数数量加 1；如果该数字是 0，将 0 的数量加 1；否则，将奇数的数量加 1。
4. 对输入的数列中的每个数字重复上面的过程。

本程序主要由以下几个部分组成：初始化变量、读入并将数字分类、输出结果。为了简化 main 函数并进一步说明函数传递过程，本程序包括：

- 函数 initialize，用来初始化变量，如 zeros，odds 和 evens。
- 函数 getNumber，用来读入数字。
- 函数 classifyNumber，用来判断数字是奇数还是偶数（以及是否为0），并且将对应的计数变量加 1。
- 函数 printResults，用来输出结果。

下面分别介绍各函数。

initialize 函数 initialize 负责初始化变量。需要初始化的变量有：zeros，odds 和 evens。前面已经讲过，这些变量的初始值都是0。很明显，该函数应该有3个初始值。因为这些初始化变量的值必须可以传递到该函数的外部，所以这些参数必须是引用参数。该函数定义为：

```
void initialize(int& zeroCount, int& oddCount, int& evenCount)
{
      zeroCount = 0;
      oddCount = 0;
      evenCount = 0;
}
```

getNumber 函数 getNumber 读入数字，并将其传递给函数 main。因为只需要传递一个数字，所以该函数只有一个参数。因为还要将读入的数字传递到函数外面，所以应该使用引用参数。该函数定义为：

```
void getNumber(int& num)
{
      cin>>num;
}
```

也可以将函数 getNumber 改写成带有返回值的函数。请参见本程序范例后面的注释。

classifyNumber 函数 classifyNumber 的功能是判断输入的数字是奇数还是偶数。如果该数字是偶数，还要判断该数字是否为0。该函数同时还要对相应的计数变量 zeros，odds 或 evens 做修改。函数必须能够接收需要分析的数字，所以该数字应该作为该函数的参数。函数还要对计数变量（定义在函数 main 中的 zeros，odds 和 evens）做相应的修改，所以这些变量也应该作为该函数的参数。因此，该函数共有 4 个参数。

因为输入的数字只做分析使用，所以只需要传递它的值。因此，对应于该变量的形参应该是值参数。函数在分析输入的数字后，还要增加相应的计数变量中的值，如 zeros，odds 或 evens。因此，这些形参对应的变量应该是引用变量。分析数字和增加相应计数变量中的值的算法与前面程序中的算法相同。该函数定义为：

```
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 函数printResults将输出最终结果。为了输出最终结果，函数必须能够存取定义在函数main中的变量zeros，odds和evens中的值。因此，该函数应该有3个参数。因为只需要输出变量中的值，所以形参应该是值参数。该函数定义为：

```
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;
}
```

下面给出该程序的主要算法，并且说明函数main怎样调用这些函数。

主要算法

1. 调用函数initialize来初始化变量。
2. 提示用户输入20个数字。
3. 对于输入序列中的每个数字：
   a. 调用函数getNumber读入数字
   b. 输出数字
   c. 调用函数classifyNumber分析数字，并且增加相应的计算变量
4. 调用函数printResults输出最终结果。

完整的程序代码清单

```
//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);                            //Step 4
    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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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： 在本程序中，因为默认的是从标准输入设备（键盘）中输入数据，并且函数getNumber只接收一个数值，所以可以将函数getNumber改写成带有返回值的函数。如果写成带有返回值的函数，函数getNumber的定义是：

```
int getNumber()
{
    int num;
    cin>>num;
    return num;
}
```

在这种情况下，需要将函数main中的语句（函数调用）：

```
getNumber(number);
```

改写成为：

```
number = getNumber();
```

当然，还要对其函数原型做相应的修改。

7.10 程序范例：数据比较

本程序范例将说明：

- 在一个程序中怎样从多个文件中读入数据。
- 怎样将结果输出到文件中。
- 怎样生成条形（直方）图（Bar Graphs）。
- 怎样借助于函数和参数传递，使同一程序段可以处理不同（但相似）的数据集。
- 怎样使用结构化程序设计来解决实际问题，以及怎样进行参数传递。

本程序分为两个部分。首先介绍怎样从多个文件中读入数据；然后介绍怎样生成条形图。

某大学在夏季学期将选修某些新开设的课程的学生分成两组，分别由不同的教师授课。在学期期末，对这两组学生进行相同的考试，并将成绩分别存储在两个不同的文件中。这两个文件中的数据格式如下所示：

```
courseNo  score1, score2, ..., scoreN -999
courseNo  score1, score2, ..., scoreM -999
.
.
.
```

现在编写程序，计算选修每门课程的每组学生的平均成绩。输出的形式如下所示：

```
Course#   Group#   Course Average
   A        1          80.50
            2          82.75
   B        1          78.00
            2          75.35
```

输入 因为选修某门课程的两组学生的成绩存储在不同的文件中，所以输入数据也分别在两个文件中。

输出 如上所示。

问题分析和算法设计

从两个文件中读取数据很简单。假设第一组学生的数据存储在文件a:group1.txt中，第二组学生的数据存储在文件a:group2.txt中。在处理完某门课程的第一组数据之后，需要处理该课程的第二组数据，直到所有的课程成绩都处理完为止。对于每门课程来说，处理过程分为两步：

1. a. 计算某门课程的总成绩
   b. 计算选修该门课程的学生人数
   c. 用总人数除总成绩，计算出该课程的平均成绩
2. 输出结果

我们要比较每组相应课程的平均成绩，每个文件中的数据按照课程ID排序。为了保证比较的是相同课程的平均成绩，需要先比较课程ID。如果课程ID不同，要输出错误信息并终止程序。

经过上述讨论，现在编写函数calculateAverage，来计算课程的平均成绩。还需要编写函数printResult，来输出计算结果。通过传递相应的参数，使用相同的calculateAverage函数和printResult函数分别计算和输出每门课程的两组平均成绩（在程序的第二部分中，将修改函数printResult）。

由上面讨论可以得到下面的算法：

1. 初始化变量。
2. 第一组和第二组输入课程ID。
3. 如果输入的课程ID不同，输出错误信息并且终止程序。
4. 计算第一组和第二组的课程平均成绩。
5. 按照前面给出的形式输出结果。
6. 对每门课程重复上面的第2步至第6步。
7. 输出最终结果。

变量（函数main）

由上面讨论可知，需要在程序的main函数中定义如下变量：

```
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
```

下面我们分别讨论函数calculateAverage和函数printResult。然后，在函数main中调用这两个函数。

calculateAverage　该函数用来计算课程的平均成绩。因为在函数main中需要打开输入数据所在文件，所以需要将该文件与文件流变量ifstream联系起来。该函数还要将计算出来的课程平均成绩传递给函数main。因此，该函数有两个参数，而且每个参数都是引用参数。

为了计算课程平均成绩，必须先计算该门课程的总成绩和选修该课程的学生总数。因此，需要定义变量来存储课程成绩、课程总成绩和学生总数。当然，必须将存储课程总成绩和学生总数的变量初始化为0。

局部变量（函数calculateAverage）　由上面讨论可知，在函数calculateAverage需要定义以下3个变量：

```
double totalScore;     //to store the sum of score
int numberOfStudent;   //to store the number of students
int score;             //to read and store a course score
```

由上面讨论，可以得出函数calculateAverage的算法：

a. 定义变量
b. 将totalScore初始化为0.0
c. 将numberOfStudent初始化为0
d. 读入（下一个）课程成绩
e. 在totalScore中加上由步骤d读入的课程成绩
f. 将numberOfStudent加1
g. 重复步骤d，步骤e和步骤f，直到输入的课程成绩是-999为止
h. courseAvg = totalScore / numberOfStudent;

可以使用while循环结构重复步骤d，步骤e和步骤f。
现在，我们开始动手编写函数calculateAverage。

```
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输出每组成绩的课程ID，小组编号以及课程平均成绩。结果将输出到文件中。所以必须将4个参数传递到该函数中：与输出文件相联的ofstream类型文件流变量、小组编号、课程ID和该组的课程平均成绩。ofstream类型变量应该作为引用参数传递。因为该函数只用到了其他3个参数的值，所以剩下的3个参数应该是值参数。而且，从输出结果来看，只需要在第一组成绩前输出课程ID。该算法的伪代码如下所示：

```
if(group number == 1)
   print course ID
else
   print a blank

print group number and course average
```

函数printResult的定义如下所示：

```
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
```

在设计和定义完函数calculateAverage和printResults之后，我们现在开始设计函数main的算法。在动手之前，还要注意：虽然两个输入文件中的数据都按照课程ID排序，但是很有可能其中一个文件中的课程ID比另一个文件中的课程ID少。我们并不需要提前检查到底会不会出现这种不能处理的情况。但是，必须保证在输出最终结果（第一组和第二组的平均成绩）之前，检查这种错误。

主要算法：函数 main

1. 定义变量（局部变量）。
2. 打开文件。
3. 如果打开文件失败，输出错误信息并且终止程序。
4. 打开输出文件。
5. 通过控制符fixed和showpoint将输出的小数格式设置为：固定小数点、显示小数和小数尾部的0。通过精度设定，使输出数字有两位小数。
6. 将第一组的平均成绩初始化为0.0。
7. 将第二组的平均成绩初始化为0.0。
8. 将课程数目初始化为0。
9. 输出标题。
10. 读入第一组成绩的课程ID到courseId1。
11. 读入第二组成绩的课程ID到courseId2。
12. 对于第一组和第二组中的每门课程：

    a. `if(courseId1 != courseId2)`
    ```
    {
          cout<<"Data error: Course IDs do not match.\n");
          return 1;
    }
    ```
    b. `else`

    {

    (1) 计算第一组的平均成绩（调用函数calculateAverage，传递相应的参数）

    (2) 计算第二组的平均成绩（调用函数calculateAverage，传递相应的参数）

    (3) 输出第一组的平均成绩（调用函数printResults，传递相应的参数）

    (4) 输出第二组的平均成绩（调用函数printResults，传递相应的参数）

    (5) 更新第一组的平均成绩

    (6) 更新第二组的平均成绩

    (7) 将课程数目加1

    }

    c. 读入第一组成绩的课程ID到courseId1

    d. 读入第二组成绩的课程ID到courseId2

13. a. if(第一组数据没有结束，但是第二组数据结束)

    输出“Ran out of data for group 2 before group1”

    b. else

    if(第一组数据结束，但是第二组数据没有结束)

    输出“Ran out of data for group1 before group2”

c. else

输出第一组和第二组的平均成绩

14. 关闭输入和输出文件。

完整的程序代码清单

```
//Program: Comparison of class averages

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

 //function prototypes
void calculateAverage(ifstream& 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
   ifstream group1;       //input stream variable for group 1
   ifstream 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;
}
```

输出

```
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
```

输入第一组数据

```
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
```

输入第二组数据

```
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 -999A 80 100 50 10 32 90 89 100 23 50 -999
```

条形图 (Bar Graph)

在商业中，公司的执行官们大都喜欢看可视化的统计结果，例如条形图、又称直方图。现在很多软件包都可以分析数据，并将其显示成可视化图形，例如条形图和饼状图。本程序的第二部分将主要介绍怎样将计算结果显示成如下所示的条形图：

```
Course      Course Average
 ID     0   10   20   30   40   50   60   70   80   90   100
        |....|....|....|....|....|....|....|....|....|....|
 A      ***************
        ############
 B      *************
        #####################
Group 1 -- ****
Group 2 -- #####
```

图中的每个符号 (* 或者 #) 表示 2 分。如果平均成绩低于 2 分，一个符号也不会输出。

因为现在要求以条形图方式输出结果，所以需要修改函数 printResult。

printBar　函数printBar的功能是输出课程ID和课程平均成绩条形图，输出结果存储在文件中。该函数有4个参数：与输出文件相联系的ofstream类型文件流变量、小组编号、课程ID和课程平均成绩。输出条形图时，可以使用循环将每2分表示成一个符号。例如，如果平均成绩是78.45分，则需要输出39个符号来表示该分数。可以使用整数除法来计算要输出的符号个数。

```
numberOfSymbols = static_cast<int>(average) / 2;
```

例如，static_cast<int>(78.45) / 2 = 78 / 2 = 39。

经过上面讨论，printResult的函数定义为：

```
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
```

在程序中使用函数printHeading，来打印输出的前两行。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
```

如果用新的printResult函数取代前面的原有函数，则同时还需要在程序中包含函数printHeading，以及用于提示小组输出符号的Group 1 -- **** 和 Group 2 -- #### 的输出语句。在输入数据不变的情况下，重新运行程序后，输出的结果如下所示：

输出

```
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：比较上面两种输出方法，你认为哪种更好一些？

7.11 小结

1. 没有数据类型的函数称为无返回值函数。
2. 可以在无返回值函数中使用没有返回值的return语句。在希望提前跳出函数的情况下，通常会使用这种return语句。
3. 无返回值函数的函数头以void开头。
4. 在C++中，void是保留字。
5. 无返回值函数既可以有参数，也可以没有参数。
6. 调用无返回值函数的语句是单独的调用语句。
7. 调用无返回值函数，需要在单独的调用语句中提供该函数的函数名及实参。
8. 形参分为两种：值参数和引用参数。
9. 值参数接收的是相应实参的值的拷贝。
10. 引用参数接收的是相应实参的内存地址或内存单元。
11. 值参数的实参可以是表达式、变量或常量值。
12. 常量值不能传递给引用参数。
13. 与引用参数相对应的实参必须是变量。
14. 将 & 置于形参的类型后，表示该形参是引用参数。
15. 流变量应该通过引用参数传递给函数。
16. 如果希望某个形参可以改变实参中的值，需要在函数头中将该形参定义为引用参数。
17. 标识符的作用域是指程序中可以使用该标识符的范围。
18. 定义在函数（块结构）以内的变量称为局部变量。
19. 定义在所有函数（块结构）以外的变量称为全局变量。
20. 函数名的作用域与定义在所有函数（块结构）以外变量（全局变量）的作用域相同。
21. 仔细阅读本章（标识符的作用域一节）中的作用域规则。
22. C++不允许嵌套函数定义。
23. 自动变量的存储空间，在函数（块结构）执行开始时分配，并在执行后释放。
24. 静态变量的存储空间在整个程序执行过程中始终存在。
25. 全局变量默认为静态变量。
26. C++支持函数重载。
27. 如果函数重载，那么具有相同名字的所有函数必须有不同的形参列表。
28. C++允许函数有默认参数。
29. 如果在函数调用时，没有给有默认值的参数指定实参，那么该参数将使用默认值。
30. 所有的默认参数必须在函数参数列表的右侧。
31. 假设一个函数有多个默认参数。在函数调用中，如果没有给其中一个有默认值的参数指定实参，则必须略掉该参数右边的所有实参。

32. 默认参数值可以是常量、全局变量或者函数调用。
33. 在函数调用中可以为默认参数指定实参，而不使用它们的默认值。
34. 不能将常量值指定为引用参数的默认值。

7.12 练习

1. 判断下面说法的正误。
   a. 函数改变引用参数的值的同时也改变实参的值。
   b. 变量名不能传递给值参数。
   c. C++ 函数可以没有参数，但是空参数列表外面的括号仍然是必需的。
   d. 在 C++ 中，形参和实参的名字必须是相同的。
   e. 只要引用参数的值改变，对应的实参中的值也将改变。
   f. 在 C++ 中，函数可以嵌套定义。也就是说，一个函数可以定义在另一个函数的函数体中。
   g. 在程序中使用全局变量是良好的程序编写风格，要优于使用局部变量，因为这样可以避免定义额外的变量。
   h. 在程序中，使用全局常量与使用全局变量同样危险。
   i. 静态变量的存储空间在函数调用过程中始终存在。
2. 在下面的程序代码中，分别指出：
   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. 解释实参和形参的不同点。
   b. 解释值参数和引用参数的不同点。
   c. 解释局部变量和全局变量的不同点。
4. 下面程序的输出结果是什么？

```
#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. 假定输人值是：

```
7  3  6  4
2  6  3  5
```

给出下面程序的输出结果：

```
#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. 下面程序的输出结果是什么？

```
#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. 在下面程序中，给出前面有下划线标记的语句的执行顺序（逻辑执行顺序）。

```
    #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. 下面代码段的输出结果是什么（注意：alpha 和 beta 都是 int 类型变量）？

```
alpha = 5;
beta = 10;
if(beta >= 10)
{
    int alpha = 10;
    beta = beta + alpha;
    cout<<alpha<<' '<<beta<<endl;
}
cout<<alpha<<' '<<beta<<endl;
```

9. 如果将例 7.8 程序中的第 7 行语句替换成下面语句：

```
funTwo(t, num1);
```

那么该程序的输出结果是什么？并且指出每条语句执行后各变量中的值。

10. 考虑下面程序，它的输出结果是什么？如例 7.6 一样，说出每行语句执行后各变量中的值。

```
#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 = '*');
```

下面的函数调用中，哪些是合法的?

(1) testDefaultParam(5);

(2) testDefaultParam(5,8);

(3) testDefaultParam(6, '#');

(4) 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;
}
```

下列各函数调用的输出结果是什么?

(1) defaultParam(6);

(2) defaultParam(3,4);

(3) defaultParam(3, 0, 2.8);

7.13　编程练习

1. 考虑函数 main 的函数定义：

```
int main()
{
    int x, y;
    char z;
    double rate, hours;
    double amount;
    .
    .
    .
}
```

编写下面的函数定义：

a. 函数 initialize，将变量 x 和 y 初始化为 0，将 z 初始化为空格字符。

b. 函数 getHoursRate，提示用户输入工作小时和每小时的工资率。用这两个值初始化 main 函数中的变量 hours 和 rate。

c. 带有返回值的函数 payCheck，根据工作的小时数和工资率计算并返回员工工资总额。工作小时数和工资率分别存储在 main 函数中的变量 hours 和 rate 中。计算工资总额的公式如下：前 40 小时，按照正常的工资率计算；超过 40 小时的部分，按照 1.5 倍工资率计算。

d. 函数 printCheck，输出工作小时数、工资率和工资总额。

e. 函数 funcOne，提示用户输入一个数值。然后将变量 x 值改变为变量 x 值乘以 2，加上变量 y 中的值，最后减去输入的数值。

f. 函数 nextChar，将读入下一个字符并存储到变量 z 中。

g. 完成函数 main 的定义，以检验上面函数的正确性。

2. 例 7.5 中的函数 printGrade 是 void 类型函数，该函数用来计算并且输出课程成绩。课程分数作为参数传递给函数 printGrade。将函数 printGrade 改写成为带有返回值的函数，使之计算并返回课程成绩，课程成绩应在 main 函数中输出，并且将函数名改为 calculateGrade。

3. 在本练习中，将修改本章中的程序范例：数字分类。该程序从标准输入设备（键盘）中读入数据，并将结果输出到标准输出设备（屏幕）上。该程序只能处理20个数字。改写该程序，使之符合以下要求。

   a. 程序从一个未知长度的文件中读入数据；也就是说，程序预先并不知道文件中有多少个数字。

   b. 将程序运行结果输出到文件中。

   c. 修改函数 getNumber，使之能够从输入文件（在函数 main 中打开）中读入数字，并将结果输出到输出文件（在函数 main 中打开）中，并发送数字到 main 函数。每行最多输出 10 个数字。

   d. 使程序可以计算输入数字的总和以及平均数。

   e. 改写函数 printResult，使之将最终结果输出到输出文件（在函数 main 中打开）中。除了输出每种类型数字的个数，新的 printResult 函数还要输出总和及平均数。

4. 重写第 5 章中编程练习 11 中的程序，使函数 main 仅由各种函数组成。程序中需要包含以下函数：

   a. 函数 openFiles：该函数用来打开输入文件和输出文件，并将输出数字的格式设置为固定小数点的两位小数，显示小数点和小数部分尾部的 0。

   b. 函数 initialize：该函数用来初始化变量 countFemale，countMale，sumFemaleGPA 和 sumMaleGPA。

   c. 函数 sumGrades：该函数用来计算男学生和女学生 GPA 的和。

   d. 函数 averageGrade：该函数用来计算男学生和女学生 GPA。

   e. 函数 printResults：该函数用来输出相关计算结果。

   f. 不要定义任何全局变量。使用适当参数在函数之间传递数据。

5. 编写一个程序，该程序读入月－日－年形式的日期，并计算出该日期是该年中的第几天。例如日期 1-1-02，是该年中的第 1 天；日期 12-25-02，是该年中的第 359 天。程序必须能够判断该年是否是闰年。如果某个年份可以被 4 整除，但不能被 100 整除，那么该年就是闰年。例如，1992 和 2008 可以被 4 整除，但不能被 100 整除，所以都是闰年。但是，如果某个年份既可以被 100 整除，又可以被 400 整除，那么该年也是闰年（1600 年和 2000 年）。例如，1800 不是闰年，因为它不可以被 400 整除。

6. 编写程序读入学生姓名和考试分数。该程序计算每个学生考试的平均分数，并给出相应的成绩。考试分数与考试成绩的对应关系是：90~100，A；80~89，B；70~79，C；60~69，D；0~59，E。程序中需要包括下面的函数：

   a. 无返回值函数 calculateAverage，用来计算每个学生 5 门考试的平均成绩。使用循环结构读入每门成绩，并且求出总成绩（该函数并不输出考试的平均分数，这项功能由 main 函数完成）。

   b. 带有返回值的函数 calculateGrade，用来计算并且返回学生成绩（该函数并不输出考试成绩，这项功能由 main 函数完成）。

   使用下面数据测试程序。数据从文件输入，并将计算结果输出到文件中。程序中不要使用全局变量，使用适当的参数在函数之间传递数据。

```
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
```

示例输出

输出应采用下面的形式（计算结果输出到右边两列中，最后一行中应该输出班级的平均成绩）：

```
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. 编写程序处理文本文件。该程序读入文本文件并且将结果输出到文件中。该程序用来计算文本文件中的字数（word）、行数和段数。输入的文本文件各段之间有空行。

   程序中需要包含下面的函数：

   a. initialize：该函数用来初始化函数 main 中的所有变量。

   b. processBlank：该函数用来读写空格符。当遇到一个非空格字符（空白字符除外）时，函数将该行中的字数加 1。函数 updateCount 用来在一行结束后，将字数计数器清零。该函数在处理完空格符后终止。

   c. copyText：该函数用来读写非空格字符。当遇到空格符时，该函数终止。

   d. updateCount：该函数使用在每一行结束处。该函数用来更新总字数，将行数加 1，将每行中的字数计数器清零。如果某一行中没有任何字，将段数加 1。空行（段间）用来分隔段落，所以不应该算做行数。

   e. printTotal：该函数用来输出字数、行数和段数。

   程序将从文本文件中读入数据并且将结果输出到文件中。程序中不要使用全局变量，使用适当的参数在函数之间传递数据。使用如下形式的 main 函数测试你编写的程序：

```
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　用户自定义简单数据类型、名字空间和string类型

本章要点：

- 理解怎样创建和使用简单数据类型（枚举类型）
- 理解怎样使用typedef语句
- 了解名字空间机制
- 了解string数据类型，理解怎样使用各种函数来处理string类型数据

在第2章中已经介绍过，C++的简单数据类型分为三类：整型、浮点型和枚举型。在前面几章中，接触的主要是整型和浮点型数据类型。在本章中，将主要介绍枚举类型enum。因为所有使用ANSI/ISO标准头文件的C++程序都要使用语句using namespace std（在第2章中曾讨论过），所以本章最后一部分将主要介绍该语句的作用。实际上，你将了解到namespace机制。此外，还可以了解到string类型和string类型的函数。

8.1　枚举类型

第2章将数据类型定义为数据集及定义在数据集上的操作。例如，int数据类型的数据集由从 -2 147 483 648到2 147 483 647的整数组成，在这个数据集上定义的操作是算术运算（+，-，*，/和%）。因为程序的主要目标是操纵数据，所以数据类型概念成为所有程序设计语言的基础。数据类型的定义说明了哪些数据是合法的，以及在这些数据上的哪些操作是合法的。系统据此提供内建的机制来检查是否发生错误。

到目前为止，接触到的数据类型仅包括int，bool，char和double。虽然使用这几种数据类型可以解决绝大部分的问题，但是还是有某些问题不能仅用这些数据类型来解决。C++提供给用户建立自己数据类型的机制，这将大大地提高程序设计语言的灵活性。

在本节中，将学会怎样创建自己的简单数据类型和枚举类型。在接下来的几章中，还将了解到怎样创建复杂数据类型。

在定义枚举数据类型之前，必须知道：

- 数据类型的名字
- 数据类型的数据集
- 在该数据集上进行的运算

在定义新的简单数据类型时，C++只允许指定数据类型的名字和数据集，而不可以指定数据集上进行的运算。禁止用户定义自己的运算可以防止出现潜在的系统错误。

枚举数据类型中的数据必须是标识符。

定义枚举数据类型的语法是：

```
enum typeName{ value1, value2, ...};
```

这里，value1，value2…都是标识符，称为枚举分量（Enumerator）。在C++中，enum是保留字。

在将所有数据值列在花括号中的时候，同时也就指定这些值的先后顺序，即value1 < value2 < value3 <…。也就是说，枚举数据类型的数据集是顺序集。赋给这些枚举分量的值默认从0开始。也就是说，赋给value1的默认值是0，赋给value2的默认值是1，以此类推（也可以在定义枚举数据类型时赋给枚举分量其他的值，而不使用默认值）。

例8.1 语句：

```
enum colors{ brown, blue, red, green, yellow} ;
```

定义了一个新的数据类型——colors。该数据类型的值包括：brown，blue，red，green和yellow。

例8.2 语句：

```
enum standing{ freshman, sophomore, junior, senior} ;
```

将standing定义成枚举类型。standing中的值有：freshman，sophomore，junior和senior。

例8.3 考虑下面语句：

```
enum grades{ 'A', 'B', 'C', 'D', 'F'} ; //Illegal enumeration type
enum places{ 1st, 2nd, 3rd, 4th} ;      //Illegal enumeration type
```

上面两个都是非法的枚举类型定义，因为它们中的值都不是标识符。下面是合法枚举类型的定义：

```
enum grades{ A, B, C, D, F} ;
enum places{ first, second, third, fourth} ;
```

如果某个标识符已经被一个枚举类型使用，那么该标识符不能再被处于同一函数（块结构）的另一个枚举类型使用。本规则也同样适用于定义在任何函数之外的枚举数据类型。例8.4说明了这个规则。

例8.4 考虑下面语句：

```
enum mathStudent{ John, Bill, Cindy, Lisa, Ron} ;
enum compStudent{ Susan, Cathy, John, William} ;   //Illegal
```

假设上面两个语句出现在同一程序的同一函数中。第二条语句compStudent，不会通过编译，因为John已经被前面的枚举类型mathStudent使用了。

8.1.1 定义变量

在定义了数据类型之后，就可以使用该数据类型来定义变量。定义枚举类型变量的方法与定义其他类型变量没有什么不同：

```
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;
```

将popularSport和mySport定义为sports类型变量。

8.1.2 赋值

当变量定义好之后，就可以将值存储到变量里面了。根据前面的定义，语句：

```
popularSport = football;
```

将 football 存储到 popularSport 中。语句：

```
mySport = popularSport;
```

将 popularSport 中的值拷贝到变量 mySport 中。

8.1.3 枚举数据类型上的运算

所有的算术运算符都不可以使用在枚举类型的变量上。因此，下面的语句都是非法的：

```
mySport = popularSport + 2;          //Illegal
popularSport = football + soccer;    //Illegal
popularSport = popularSport * 2;     //Illegal
```

同样，增量运算符和减量运算符也不可以使用在枚举类型的变量上。因此，下面的语句也是非法的：

```
popularSport++;  //Illegal
popularSport--;  //Illegal
```

如果确实需要将 popularSport 中的值加 1，可以使用如下的强制转换运算符：

```
popularSport = static_cast<sports>(popularSport + 1);
```

如果程序中出现了类似上面的使用枚举变量的语句，那么编译器假定用户已经理解这样做的用意。因此，上面的语句可以通过编译，并且在程序执行时将枚举分量中的下一个值赋给变量popularSport。考虑下面语句：

```
popularSport = football;
popularSport = static_cast<sports>(popularSport + 1);
```

在执行完第二条语句后，popularSport 中的值是 hockey。同样，语句：

```
popularSport = football;
popularSport = static_cast<sports>(popularSport - 1);
```

的作用是将 basketball 存储到变量 popularSport 中。

关系运算符

由于枚举数据类型是有序的数据集，所以关系运算符可以使用在枚举数据类型的数据上。假设前面数据类型 sports 和变量 popularSport 及 mySport 的定义不变，那么：

```
football <= soccer 的值是 true
hockey > basketball 的值是 true
baseball < football 的值是 false
```

假设：

```
popularSport = soccer;
mySport = volleyball;
```

那么：

```
popularSport < mySport 的值是 true
```

8.1.4 枚举数据类型和循环结构

注意，实际上枚举类型是整数类型。通过类型强制转换运算符，可以增量、减量和比较枚举类型中的值。因此，可以在循环结构中使用枚举类型数据。假设前面的mySport定义不变，考虑下面的for循环：

```
for(mySport = basketball; mySport <= soccer;
                          mySport = static_cast<sports>(mySport + 1))
...
```

本for循环将执行5次。

在循环中使用枚举类型数据可以增强程序的可读性。

8.1.5　枚举数据类型的输入/输出

因为C++只定义了内建的数据类型，如int，char，double等的输入输出，所以枚举类型数据不能（直接地）输入输出。但是，可以间接地输入输出枚举变量。例8.5说明了这种情况。

例8.5　假设有下面语句：

```
enum courses{ algebra, basic, pascal, cpp, philosophy, analysis,
             chemistry, history};
courses registered;
```

第一条语句定义了枚举数据类型courses，第二条语句定义了courses类型变量registered。可以借助于char类型变量来读入（即输入）枚举类型数据。在本例中，通过读入一个或者两个字符就可以区分某些courses枚举类型的数据值。例如，可以通过读入一个字符来区分algebra和basic，通过读入两个字符来区分algebra和analysis。从键盘上读入这些数值时，只需要读入一个或者两个字符，然后通过选择结构就可以将该值存储到变量registered中。因此，需要定义两个char类型的变量。

```
char ch1, ch2;
cin >> ch1 >> ch2; //read two characters
```

下面的switch语句将相应的值赋给变量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，也可以间接地输出枚举类型数据：

```
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：如果试图直接输出枚举变量中的值，计算机会输出与该值对应的整数值。例如，假设"registered = algebra;"，下面语句将会输出 0，因为（默认的）赋给 algebra 的值是 0：

```
cout << registered << endl;
```

相似地，下面的语句将输出 4：

```
cout << philosophy << endl;
```

8.1.6 函数和枚举类型

与其他简单数据类型一样，枚举类型的数据也可以作为参数——值参数或者引用参数，传递给函数。同样，函数也可以返回枚举类型的返回值。利用这一特性，可以编写函数来输入和输出枚举类型数据。

下面的函数从键盘读入数据，并且返回枚举类型数据。假定枚举数据类型 courses 的定义不变。

```
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;
}
```

下面是函数输出枚举类型的数值:

```
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　在定义枚举数据类型同时定义变量

在前面的小节中，都是先定义枚举数据类型，然后再定义该类型的枚举变量。C++允许在一步中同时完成这两个过程。也就是说，可以在定义枚举数据类型的同时定义该类型的枚举变量。例如，语句:

```
enum grades{ A, B, C, D, F} courseGrade;
```

定义了枚举数据类型 grades 和该类型的变量 courseGrade。

相似地，语句:

```
enum coins{ penny, nickel, dime, halfDollar, dollar} change, usCoins;
```

定义了枚举数据类型 coins 和两个该类型变量 change 和 usCoins。

8.1.8　匿名数据类型

如果在定义数据类型的同时定义了变量，但是却又没有给出数据类型的名字，那么该数据类型就称为匿名类型（Anonymous Type）。下面语句定义了一个匿名数据类型:

```
enum {basketball, football, baseball, hockey} mySport;
```

该语句定义了变量 mySport，但是没有给出数据类型的名字。

然而，创建匿名数据类型有一些缺点。首先，因为这种数据类型没有名字，所以不能将匿名数据类型的数据传递给函数，函数也不能返回该匿名数据类型的值。其次，虽然一个匿名数据类型中的值可以用在另一个匿名数据类型中，但是这两个匿名数据类型的变量却被视为不同类型的变量。考虑下面的语句:

```
enum {English, French, Spanish, German, Russian} languages;
enum {English, French, Spanish, German, Russian} foreignLanguages;
```

虽然变量languages和foreignLanguages的取值范围是相同的，但是编译器将这两个变量视为两个不同类型的变量。因此，下面的语句是非法的：

```
languages = foreignLanguages;    //Illegal
```

虽然使用匿名数据类型来定义变量为程序设计提供了某些方便，但是要谨慎使用这种方法。为了避免错误，先定义枚举数据类型，然后再定义变量。

下面将介绍C++中的typedef语句。

8.1.9 typedef 语句

在C++中，可以通过typedef语句来创建已定义好的数据类型的同义词或者别名。typedef语句的通用语法是：

```
typedef existingTypeName  newTypeName;
```

在C++中，typedef是保留字。需要注意的是，typedef语句并不创建新的数据类型，而只是为已有的数据类型创建别名。

例8.6 语句：

```
typedef int integer;
```

创建了数据类型int的一个别名integer。类似地，语句：

```
typedef double real;
```

创建了数据类型double的一个别名real。语句：

```
typedef double decimal;
```

创建了数据类型double的另一个别名decimal。

使用typedef语句，可以创建自己的布尔数据类型，见例8.7所示。

例8.7 第4章中说明，C++中逻辑（布尔）表达式的值是1或0，实际上就是int类型的数据。作为逻辑值，1代表true，0代表false。考虑下面语句：

```
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数据类型创建了一个别名：Boolean。第2行和第3行语句定义了命名常量True和False，并分别将其初始化为1和0。第4行语句定义了一个Boolean类型变量flag。因为flag是Boolean类型变量，所以下面的语句是合法的：

```
flag = True;
```

8.2 程序范例：剪刀、石头、布的游戏

每个人都对剪刀、石头、布（Paper）的游戏很熟悉，孩子们经常玩这种游戏。该游戏有两个游戏者，每个人从三个物体，剪刀、石头、布中任意选取一个。如果第一个人选择石头，而第二个人选择布，则第二个人获胜，因为布可以包住石头。该游戏的规则如下所示：

- 如果两个人选了相同的物体，游戏平局。
- 如果一个人选了石头，而另一个人选了剪刀，则选了石头的人获胜，因为剪刀剪不动石头。
- 如果一个人选了石头，而另一个人选了布，则选了布的人获胜，因为布可以包住石头。
- 如果一个人选了剪刀，而另一个人选了布，则选了剪刀的人获胜，因为剪刀可以剪破布。

编写一个交互式程序，允许两个人同时玩该游戏。

输入　该程序有两种类型的输人：

- 参加游戏的人
- 选择的物体

输出　每个游戏者及他们的选择，游戏的获胜者。在每次游戏结束后，要输出游戏的总局数和每个游戏者获胜的局数。

问题分析和算法设计

该游戏有两个游戏者，每个人通过键盘输入选择的物体。游戏者按下 R 或 r 表示选择石头，P 或 p 表示选择布，S 或 s 表示选择剪刀。当第一个游戏者输入选择时，另一个游戏者在一旁等待。当两个人都输入选择，并且两个选择都合法后，程序输出每个游戏者的选择，并且说明谁是这次游戏的胜者。本游戏一直进行，直到一个游戏者退出游戏为止。在游戏结束后，程序将输出游戏的总局数和每个游戏者获胜的局数。经过讨论，得到下面的算法：

1. 提供简要的游戏说明及怎样玩该游戏
2. 询问用户是否参与游戏
3. 读人两个游戏者的选择
4. 如果选择合法，输出每个游戏者的选择及游戏的胜者
5. 更新游戏总局数和每个游戏者获胜的局数
6. 当游戏者同意继续玩该游戏时，重复第 2 步到第 5 步
7. 输出游戏的总局数和每个游戏者获胜的局数

这里使用枚举类型来表示物体：

```
enum objectType{ Rock, Paper, Scissors} ;
```

变量（main 函数）　很明显，程序的 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
```

该程序中包含如下几个函数，下面将仔细讨论每个函数：

- displayRules：该函数用来显示关于游戏和规则的简单介绍。
- validSelection：该函数用来检查游戏者输入的选择是否合法。合法的选择有 R，r，P，p，S 和 s。
- retrievePlay：因为枚举类型数据不能直接读人，该函数用来将输入的选择（R，r，P，p，S 或 s）转换成相应的枚举值，并返回该枚举值。

- gameResult：该函数用来输出每个游戏者的选择以及本次游戏的胜者。
- convertEnum：该函数被函数 gameResult 调用，输出相应的枚举类型数值。
- winningObject：该函数用来判断并输出获胜的物体。
- displayResults：该函数负责在游戏结束后显示最终结果。

函数 displayRules 该函数仅由输出游戏和规则的解释语句组成，没有参数。该函数的定义是：

```
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 该函数用来检查游戏者输入的选择是否合法。

```
if selection is 'R' or 'r' or 'S' of 's' or 'P' or 'p', then
  it is a valid selection;
otherwise the selection is invalid.
```

这里，我们使用 switch 语句来检查输入数据的合法性。该函数的定义是：

```
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 因为程序不能直接读入枚举类型数据，该函数用来将输入的选择（R，r，P，p，S或s）转换成相应的枚举值，并返回该枚举值。因此，该函数有一个char类型的参数。该函数是带有返回值的函数，返回 objectType 类型的数据值。该函数的伪代码算法是：

```
if selection is 'R' or 'r')
      return Rock;
if selection is 'P' or 'p')
      return Paper;
if selection is 'S' or 's')
      return Scissors;
```

函数 retrievePlay 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　该函数用来判断本次游戏是否是平局；如果不是平局，判断出谁是胜者。该函数输出每个游戏者的选择以及本次游戏的胜者。很明显，该函数包含三个参数：第一个游戏者的选择、第二个游戏者的选择和游戏胜者。该函数的伪代码是：

```
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.
   }
```

函数 gameResult 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 因为程序不能直接输出枚举类型数据，所以编写函数 convertEnum 来输出 objectType 枚举类型的值。该函数输出与 objectType 类型值对应的字符串。该函数的伪代码是：

```
if object is Rock
   output  "Rock"
if object is Paper
   output  "Paper"
if object is Scissors
   output  "Scissors"
```

函数 convertNum 的定义是：

```
void convertEnum(objectType object)
{
   switch(object)
   {
   case Rock:  cout<<"Rock";
               break;
   case Paper: cout<<"Paper";
               break;
   case Scissors: cout<<"Scissors";
   }
} //end convertEnum
```

函数 winningObject 该函数需要知道游戏者选择的物体，再根据游戏规则判断胜者。例如，如果一个游戏者选择 Rock，而另一个游戏者选择 Paper，则选择了 Paper 的人获胜。也就是说，获胜的物体是 Paper。函数 winningObject 判断输人的两个物体，并且返回获胜的物体。很明显，该函数有两个 objectType 类型的参数，并且返回 objectType 类型的返回值。该函数的定义是：

```
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 在游戏结束后，该函数输出最终结果——即游戏的总局数和每个游戏者获胜的局数。游戏的总局数存储在变量 gameCount 中，第一个游戏者获胜的局数存储在变量 winCount1 中，第二个游戏者获胜的局数存储在变量 winCount2 中。该函数有三个与这些变量相对应的参数。函数 displayResults 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，我们可以写出函数 main 的算法。

主要算法

1. 定义变量。
2. 初始化变量。
3. 显示规则。
4. 提示用户进入游戏。
5. 得到用户是否决定参加游戏的决定。
6. while(决定参加)

   {

   a. 提示第一个游戏者输入选择

   b. 读入第一个游戏者的选择

   c. 提示第二个游戏者输入选择

   d. 读入第二个游戏者的选择

   e. 如果两个选择都是合法的

     {

     (1) 将游戏总局数加 1

     (2) 宣布游戏的胜者

     (3) 将获胜者的获胜局数加 1

     }

   f. 询问用户是否还要继续玩游戏

   g. 得到用户的决定

     }

7. 输出游戏结果。

完整的程序代码清单

```
#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 名字空间

1998年7月，ANSI/ISO标准C++正式出台。现在，绝大多数编译器都支持ANSI/ISO标准C++（为保证绝对兼容，请参阅编译器的说明文档）。标准C++不支持ANSI/ISO标准C++中的部分特性，本章的后面部分将主要讨论这些不兼容的特性。除非特殊指出，在本书的后面章节中给出都是两种标准都支持的C++语法。首先，我们将讨论ANSI/ISO标准C++的名字空间机制。

如果程序中包含头文件，例如 iostream，那么该头文件中的全局标识符就成为整个程序的全局标识符。因此，如果程序中包含一个与头文件中某个标识符同名的标识符，那么编译器将报出语法错误（例如“标识符重复定义”）。这种情况也可能发生在使用第三方库文件的情况中。为了避免出现错误，第三方软件供应商通常在自己的全局标识符前面加上特殊的符号。在第 2 章中已经介绍过，因为编译器生产厂商通常在全局标识符前面加上下划线（_），所以建议读者不要在自己程序中的标识符前面加下划线。

ANSI/ISO 标准 C++ 试图采用名字空间（namespace）机制来解决全局标识符名字重复问题。

语句 namespace 的语法是：

```
namespace namespace_name
{
   members
}
```

这里 member 通常是命名常量、变量定义、函数或其他的 namespace。注意，namespace_name 是 C++ 标识符。

在 C++ 中，namespace 是保留字。

例 8.8　语句：

```
namespace globalType
{
   const int n = 10;
   const double rate = 7.50;
   int count = 0;
   void printResult();
}
```

定义 globalType 是一个拥有 4 个成员的 namespace：命名常量 n 和 rate，变量 count 和函数 printResult。

名字空间成员的作用域只限于名字空间内部。通常，可以通过以下两种方式来使用名字空间的成员。使用名字空间成员的语法是：

```
namespace_name::identifier
```

例如，可以通过下面语句使用名字空间 globalType 中的成员 rate：

```
globalType::rate
```

还可以通过下面的语句使用名字空间 globalType 中的成员 printResult（是一个函数）：

```
globalType::printResult();
```

在 C++ 中，:: 被称为作用域运算符。因此，可以通过在 namespace_name 后面加上作用域运算符，再加上成员名称的方法来使用名字空间中的成员。也就是说，需要在成员名称前面加上 namespace_name 和作用域运算符。

ANSI/ISO 标准 C++ 提供 using 语句来简化名字空间成员的使用。使用 using 语句的语法是：

(a) 简化使用所有名字空间成员的语法是：

```
using namespace namespace_name;
```

(b) 简化使用某个特定名字空间成员的语法是：

```
using namespace_name::identifier;
```

例如，using 语句：

```
using namespace globalType;
```

简化对名字空间 globalType 中所有成员的使用。语句：

```
using globalType::rate;
```

简化名字空间 globalType 中成员 rate 的使用。

在 C++ 中，using 是保留字。

在通常情况下，using语句应该置于名字空间定义之后。例如使用名字空间globalType中成员的代码是：

```
namespace globalType
{
   const int n = 10;
   const double rate = 7.50;
   int count = 0;
   void printResult();
}

using namespace globalType;
```

如果程序中使用了using语句，那么在使用名字空间成员时，无须再将namespace_name和作用域运算符置于名字空间成员之前。但是如果名字空间成员和程序中的全局标识符具有相同的名字，则必须在名字空间成员前面加上namespace_name和作用域运算符。同样，如果名字空间成员和函数（块结构）中的某个标识符名字相同，则必须在名字空间成员前面加上 namespace_name 和作用域运算符。

例 8.9 至例 8.12 有助于理解名字空间机制。

例 8.9 考虑下面的 C++ 代码：

```
#include <iostream>
using namespace std;
.
.
.
int main()
{
      .
      .
      .
}
.
.
.
```

在本例中，可以在不使用前缀 std::的情况下，使用头文件 iostream 中的全局变量，如 cin，cout 和 endl。一个很明显的限制是，在使用这些（头文件 iostream 中的）全局标识符的函数（块结构）中，不能含有与之同名的标识符。

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);
   .
   .
   .
}
```

本函数中使用了头文件 cmath 中的函数 pow。

例 8.11　考虑下面的 C++ 代码:

```
#include <iostream>
.
.
.
int main()
{
   using namespace std;
   .
   .
   .
}
.
.
.
```

在本例中，函数main可以在不在标识符名字前面使用前缀std::的情况下，使用头文件iostream中的全局标识符。因为 using 语句出现在函数 main 中，所以如果其他函数要使用头文件 iostream 中的全局标识符，就必须在这些标识符名字前面加前缀 std::，除非这些函数中也使用相同的 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
{
   .
   .
   .
}
```

在本C++程序中：

1. 如果要在函数main中使用第2行中定义的变量t，则必须在该变量前面加前缀::(即::t)。因为函数main中已有一个与之同名的变量t(定义在第9行)。
2. 如果要在函数main中使用(第5行中定义的)名字空间exp中的成员t，则必须在该变量前面加前缀exp::(即exp::t)。因为已经有一个名为t的全局变量并且函数main中也有一个名为t的变量。
3. 如果要在函数main中使用(第6行中定义的)名字空间exp中的成员u，则必须在该变量前面加前缀exp::(即exp::u)，因为已经有一个名为u的全局变量(定义在第3行)。
4. 可以在函数main中采用以下两种方式使用名字空间exp中的成员x(定义在第4行)：x或exp::x。因为程序中既没有名为x的全局标识符，函数中也没有名为x的局部标识符。
5. 正如上面程序中所示的那样，作为名字空间成员的函数，如printResult，其函数定义通常写在名字空间定义之外。在函数printResult的函数定义里，函数头中的函数名称既可以写成printResult，又可以写成exp::printResult(因为程序中没有名为printResult的全局变量)。

注意： 系统提供的头文件中的标识符，如iostream，cmath和iomanip都定义在名字空间std中。因此，为了简化对这些头文件中标识符的使用，通常在我们编写的程序中加入下面语句：

```
using namespace std;
```

8.4　string数据类型

第2章中已经对string数据类型做了一些简单介绍。在ANSI/ISO标准C++颁布之前，标准C++库并不提供string数据类型。编译器生产厂商通常提供自己的string数据类型，所以各种编译器在string数据类型的使用语法和语义上通常有很大差别。

string数据类型是一种用户自定义的数据类型，由C++标准库来支持，而不是C++语言本身的一部分。在使用string数据类型之前，需要在程序中包含头文件string，如下所示：

```
#include <string>
```

注意，C++中的字符串由零个或多个字符组成，并且所有字符都由双引号括住。

语句：

```
string name = "William Jacob";
```

将name定义为string类型变量，并初始化为"William Jacob"。name中的第一个字符W的位置是0；第二个字符i的位置是1，以此类推。也就是说，string类型变量中第一个字符的位置是0而不是1。

变量name中(理论上)可以存储任何长度的字符串。

第3章中介绍了string数据类型的I/O操作；第4章介绍了string数据类型上的关系运算。建议读者重读第3章和第4章，以便了解string数据类型的I/O操作和关系运算。

除此之外，ANSI/ISO标准C++还在string数据类型上定义了双目运算符+(字符串连接运算符)和数组下标运算符[]。现在将介绍这两个运算符。

假设有下面的变量定义：

```
string str1, str2, str3;
```

语句：

```
str1 = "Hello There";
```

将字符串"Hello There"存储到变量str1中。语句：

```
str2 = str1;
```

将变量 str1 中的值拷贝到变量 str2 中。

如果 str1 = "Sunny"，语句：

```
str2 = str1 + " Day";
```

将字符串 "Sunny Day" 存储到变量 str2 中。

假设 str1 = "Hello"，str2 = "There"。语句：

```
str3 = str1 + " " + str2;
```

将字符串 "Hello There" 存储到变量 str3 中。该语句与下面语句的作用相同：

```
str3 = str1 + ' ' + str2;
```

类似地，语句：

```
str1 = str1 + " Mickey";
```

在 str1 后面追加字符串 " Mickey"，更新了 str1 中的值。因此，str1 中的新值是 "Hello Mickey"。

注意： 字符串连接运算符 + 的两个操作数中，必须至少有一个 string 类型变量。例如，下面的语句都是非法的：

```
str1 = "Hello " + "there!";        //Illegal
str2 = "Sunny day" + '!';          //Illegal
```

如果 str1 = "Hello there"，那么语句：

```
str1[6] = 'T';
```

将 str1 字符串中的 t 改写成 T。注意，因为 string 类型变量中第一个字符的位置是 0，而 t 是 str1 中第 7 个字符，所以它的位置是 6。

在 C++ 中，[ ] 称为数组下标运算符。

正如前面所示，通过下标，可以对字符串中的元素进行读写访问。

例 8.13　下面程序显示了上面各语句的运行结果。

```
//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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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
```

在输入第29行语句中的数据后，必须按下Enter键两次。因为getline函数将读取一个换行符。

该程序的输出结果很清楚，所以将程序运行解释作为练习留给读者。

8.4.1 其他 string 类型数据上的操作

ANSI/ISO 标准 C++ 中含有许多对 string 数据类型进行操作的函数。本书中关注的 5 个函数分别是：length，size，find，substr 和 swap，本节中将介绍这些函数。

string 数据类型还有一个数据类型 string::size_type 和一个命名常量 string::npos 与之相关。

string::size_type：无符号整数类型。

string::npos：数据类型 string::size_type 的最大值，在许多机器上该数值是 4294967295。

函数 length

函数 length 返回当前字符串中字符的个数，返回值是无符号整数。调用函数 length 的语法是：

```
strVar.length()
```

这里 strVar 是 string 类型变量。函数 length 没有参数。

注意函数 length 的语法，strVar 和 length 之间的点（英文句号）很重要；它将变量名与 length 分离开来。而且，虽然 length 是一个无参函数，但是空括号仍然是必不可少的。因为 length 是一个带有返回值的函数，所以该函数调用必须出现在表达式中。

考虑下面的语句：

```
string firstName;
string name;
string str;

firstName = "Elizabeth";
name = firstName + " Jones";
str = "It is sunny.";
```

语句	结果
`cout<<firstName.length()<<endl;`	输出 9
`cout<<name.length()<<endl;`	输出 15
`cout<<str.length()<<endl;`	输出 12

因为函数 length 返回无符号整数，所以该返回值可以被赋给整型变量。在通常情况下，存储 length 函数返回值的变量的数据类型应与 string::size_type 的数据类型相一致。这有助于程序员弄清该返回值到底是无符号 int 类型，还是无符号 long 类型。

假设前面的变量定义不变，语句：

string::size_type len;

语句	结果
`len = firstName.length();`	len 中的值是 9
`len = name.length();`	len 中的值是 15
`len = str.length();`	len 中的值是 12

例 8.14 下面程序说明了 length 函数的用法。

```
//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
        <<"\" = "<<firstName.length()<<endl;                   //Line 7

    cout<<"Line 8: Length of \""<<name
        <<"\" = "<<name.length()<<endl;                        //Line 8

    cout<<"Line 9: Length of \""<<str
        <<"\" = "<<str.length()<<endl;                         //Line 9

    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
```

该程序的输出结果很清楚，所以将程序运行解释作为练习留给读者。

函数 size

一些人不喜欢使用length一词，而是喜欢使用size一词。因此，为了可以兼容使用这两个词，string数据类型提供了一个名为size的函数。函数size与length的作用相同。调用函数size的语法是：

```
strVar.size()
```

这里strVar是string类型的变量。与函数length一样，size函数没有返回值。

函数 find

函数find用于在一个字符串中查找特定子字符串出现的位置，并返回无符号的整型数值（与string::size_type的类型相同）。调用函数find的语法是：

```
strVar.find(strExp)
```

这里，strVar是string类型变量，strExp是sring类型的表达式。strExp既可以是字符串，也可以是单个字符。如果查找成功，函数find返回该子字符串在srtVar中首次出现的位置。为了便于查找成功，提供的子字符串必须十分精确。如果查找失败，该函数返回一个特殊值string::npos（“not a position within the string”）（该数值被描述为“not a valid position”可能更为恰当，因为字符串运算不允许任何字符串

有如此的长度）。因为函数 find 返回无符号整数，所以返回值可以存储在整型变量中（通常与 string::size_type 的数据类型相同）。

假设 str1 和 str2 是 string 类型变量。下面对函数 find 的调用都是合法的：

```
str1.find(str2)
str1.find("the")
str1.find('a')
str1.find(str2 + "xyz")
str1.find(str2 + 'b')
```

考虑下面语句：

```
string   sentence;
string   str;
string::size_type position;

sentence = "It is cloudy and warm.";
str = "cloudy";
```

语句	结果
`cout<<sentence.find("is")<<endl;`	输出 3
`cout<<sentence.find("and")<<endl;`	输出 13
`cout<<sentence.find('s')<<endl;`	输出 4
`cout<<sentence.find('i')<<endl;`	输出 3
`cout<<sentence.find(str)<<endl;`	输出 6
`cout<<sentence.find("the")<<endl;`	输出 `string::nops` 中的值
`position = sentence.find("warm");`	将 17 赋给 `position`

注意查找是区分大小写字母的。因此，在字符串变量 sentence 中，i（小写字母 i）的位置是 3。

例 8.15　下面程序说明了定义在 string 数据类型上的函数 find 的使用方法。

```
//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
```

该程序的输出结果很清楚，所以将程序运行解释作为练习留给读者。

函数 substr

函数 substr 用来返回指定的子字符串。调用函数 substr 的语法是：

```
strVar.substr(expr1, expr2)
```

这里，expr1 和 expr2 是无符号整数类型的表达式。expr1 用来指定子字符串在整个字符串中的起始位置；expr2 用来指定返回的子字符串的长度。

考虑下面语句：

```
string   sentence;
string   str;

sentence = "It is cloudy and warm.";
```

语句	结果
`cout<<sentence.substr(0,5)<<endl;`	输出：It is
`cout<<sentence.substr(6,6)<<endl;`	输出：cloudy
`cout<<sentence.substr(6,16)<<endl;`	输出：cloudy and warm.
`cout<<sentence.substr(3,6)<<endl;`	输出：is clo
`str = sentence.substr(0,8);`	str = "It is cl"
`str = sentence.substr(2,10);`	str = " is cloudy"

例 8.16　下面的程序说明了定义在 string 数据类型上的函数 substr 的使用方法。

```
//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"
```

该程序的输出结果很清楚，所以将程序运行解释作为练习留给读者。

函数swap

函数swap用来交换两个string类型变量中的字符串。调用函数swap的语法是：

```
strVar1.swap(strVar2);
```

这里，strVar1和strVar2都是string类型变量。如果执行该语句，将互换strVar1和strVar2中的内容。假设有下面的语句：

```
string str1 = "Warm";
string str2 = "Cold";
```

在下面语句执行之后，str1中的值是"Cold"，str2中的值是"Warm"。

```
str1.swap(str2);
```

注意： 附录F（头文件string）中列出了其他定义在string数据类型上的函数，如empty，clear，erase，insert和replace。

8.5 程序范例：Pig Latin字符串

Pig Latin字符串是故意颠倒英语字母顺序拼凑而成的行话。在本程序范例中，我们将编写一个程序提示用户输入一个字符串，该程序将输出该字符串的Pig Latin形式。将字符串转换成相应的Pig Latin形式的规则如下所示：

1. 如果字符串以元音字母开头，则在该字符串末尾加上字符串 "-way"。例如，字符串 "eye" 的 Pig Latin 形式是 "eye-way"。
2. 如果字符串以非元音字母开头，则首先在该字符串末尾加上 "-"。然后每次轮转一个字母；也就是说，将该字符串中第一个字母移到整个字符串的末尾，直到第一个字母是元音字母为止。然后，在整个字符串的末尾加上字符串 "ay"。例如，字符串 "There" 的 Pig Latin 形式是 "ere-Thay"。
3. 对于类似 "by" 这种形式的没有元音字母的字符串，字母 y 将被视为元音字母。也就是说，对于本程序来说，元音字母包括：a，e，i，o，u，y，A，E，I，O，U 和 Y。因此，"by" 的 Pig Latin 形式是 "y-bay"。
4. 对于类似 "1234" 这种形式的没有元音字母的字符串，其 Pig Latin 形式是 "1234-way"。也就是说，不含有任何元音字母的字符串的 Pig Latin 形式是在其末尾追加字符串 "-way"。

输入 本程序的输入是一个字符串

输出 本程序的输出是字符串的 Pig Latin 形式

问题分析和算法设计

假设 str 是一个字符串。为了将 str 转换成相应的 Pig Latin 形式，首先要检查该字符串的第一个字母 str[0]。如果 str[0]是元音字母，则在其末尾加上 "-way"。即：

```
str = str + "-way"
```

如果字符串 str 中的第一个字母 str[0]，不是元音字母。首先要在该字符串末尾加上 "-"。然后，将字符串 str 中的第一个字母移动到该字符串的末尾。这时，原来的第二个字母变成了第一个字母。这个检查第一个字母是否是元音字母，如果不是将其移动到字符串末尾的过程将一直进行下去，直到该字符串的首字母变成元音字母或者检查完整个字符串为止（也就是说，该字符串中不含有任何元音字母）。

本程序中包含以下函数：函数 isVowel，用来判断一个字母是否是元音字母；函数 rotate，用来将 str 中的第一个字母移动到整个字符串末尾；函数 pigLatinString，用来将 str 转换成相应的 Pig Latin 形式。经过讨论，得到下面的算法：

1. 读入 str
2. 通过函数 pigLatinString，将 str 转换成相应的 Pig Latin 形式
3. 输出 str 的 Pig Latin 形式

在编写主要算法之前，先来详细讨论每个函数。

函数 isVowel 该函数以一个字符作为参数。如果该字符是元音字母，该函数返回 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　该函数以一个字符串作为参数。该函数将字符串的第一个字母移动到整个字符串的末尾。该函数抽取从字符串中位置1（字符串中的第二个字符）到末尾的子字符串，然后将原字符串中第一个字符追加到新字符串的末尾。新字符串作为返回值返回到调用函数中。函数rotate的函数定义是：

```
string rotate(string pStr)
{
   int len = pStr.length();

   string rStr;

   rStr = pStr.substr(1,len - 1) + pStr[0];

   return rStr;
}
```

函数 pigLatinString　该函数以一个字符串pStr作为参数。该函数返回pStr的Pig Latin形式。假设pStr就是要转换成Pig Latin形式的字符串。这里有三种可能的情况：pStr[0]是元音字母；虽然pStr[0]不是元音字母，但是pStr中含有元音字母；pStr中不含有任何元音字母。假设pStr[0]不是元音字母，需要将pStr的第一个字母移动到pStr的末尾。这个过程一直会持续下去，直到pStr中的第一个字母变成元音字母或者将pStr中的所有字母都检查完毕（即pStr中不含有任何元音字母）为止。经过讨论，得到下面的算法：

1. 如果pStr[0]是元音字母，将"-way"加到pStr的末尾。
2. 如果pStr[0]不是元音字母。
3. 将pStr的第一个字母移动到pStr的末尾。原来pStr中的第二个字母变成了新的pStr中的第一个字母。现在，pStr中可能含有也可能不含有元音字母。我们定义一个布尔变量foundVowel。如果pStr中含有元音字母，foundVowel中的值就是true；否则，foundVowel中的值就是false。
   a. len表示字符串变量pStr的长度。
   b. 将foundVowel初始化为false。
   c. 如果pStr[0]不是元音字母，通过调用函数rotate将pStr中的第一个字母移动到pStr的末尾。
   d. 重复步骤c，直到pStr中的第一个字母变成元音字母或者pStr中的所有字母都检查完毕。
4. 将pStr转换成Pig Latin形式。
5. 返回pStr。

函数pigLatinString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[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
}
```

主要算法

1. 输入字符串
2. 调用函数 pigLatinString，将字符串转换成相应的 Pig Latin 形式
3. 输出该字符串的 Pig Latin 形式

完整的程序代码清单

```
#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
}
```

程序运行结果 在这些程序运行中，用户输入的数据加有阴影。

程序运行结果1

```
Enter a string: eye

The pig Latin form of eye is: eye-way
```

程序运行结果 2

```
Enter a string: There

The pig Latin form of There is: ere-Thay
```

程序运行结果 3

```
Enter a string: why

The pig Latin form of why is: y-whay
```

程序运行结果 4

```
Enter a string: 123456

The pig Latin form of 123456 is: 123456-way
```

8.6 小结

1. 枚举数据类型是有序数值集。
2. C++ 保留字 enum 的作用是创建枚举类型数据。
3. enum 的语法是：

```
enum typeName{ value1, value2, ...};
```

   这里 value1，value2，…是标识符，即 value1<value2<…。
4. 在枚举类型数据上不允许使用算术运算符。
5. 在枚举类型数据上可以使用关系运算符。
6. 程序不能直接输入或者输出枚举类型数据。
7. 枚举类型数据既可以作为值参数又可以作为引用参数传递给函数。
8. 函数可以返回枚举类型数据。
9. 匿名类型是没有名字的数据类型；匿名类型可以用来定义变量。
10. C++ 保留字 typedef 的作用是创建前面定义过的数据类型的同义词或别名。
11. 匿名数据类型变量不能作为参数传递给函数。
12. 名字空间（namespace）机制是 ANSI/ISO 标准 C++ 的新特性。
13. 名字空间成员通常是命名常量、变量、函数或者其他的名字空间。
14. 名字空间成员的作用域在本名字空间内部。
15. 在名字空间外部使用名字空间成员的一个方法是在成员名字前面加上名字空间名字和作用域运算符。
16. 在 C++ 中，namespace 是保留字。
17. 为了使用名字空间机制，在程序中使用的必须是 ANSI/ISO 标准 C++ 头文件——也就是说，不能使用带有扩展名 h 的头文件。
18. 使用 using 语句可以简化名字空间成员的使用。
19. 在 C++ 中，using 是保留字。
20. 保留字 namespace 必须出现在 using 语句中。
21. 如果不通过 using 语句方式使用名字空间成员，则必须将名字空间名字和作用域运算符置于名字空间成员之前。
22. 为了不通过名字空间名字来使用标准头文件中定义的标识符，必须在程序中包含所有必须的头文件以及下面的语句：

```
using namespace std;
```

23. 字符串是由零个或多个字符组成的序列。
24. C++ 中的字符串括在双引号之间。
25. 为了使用 string 类型，程序中必须包括头文件 string。该程序中使用到的其他头文件，也必须是 ANSI/ISO 标准 C++ 头文件。
26. 赋值运算符可以同 string 类型一同使用。
27. 字符串连接运算符 + 用来连接两个 string 类型的字符串。在连接运算中，运算符 + 的操作数中至少有一个是 string 类型变量。
28. 关系运算符可以使用在 string 类型数据上。
29. 在字符串中，第一个字符的位置是 0，第二个字符的位置是 1，以此类推。
30. 字符串的长度是指字符串中字符的个数。
31. 在 C++ 中，[ ]被称为数组下标运算符。
32. 通过在数组下标运算符中指定字符位置，就可以对字符串中的每个字符进行读写操作。
33. 函数 length 返回当前字符串中字符的个数。调用函数 length 的语法是：

```
strVar.length()
```

这里 strVar 是 string 类型变量。
34. 函数 size 返回当前字符串中字符的个数。调用函数 size 的语法是：

```
strVar.size ()
```

这里 strVar 是 string 类型变量。函数 size 和函数 length 的作用相同。
35. 函数 find 用来在字符串中查找特定子字符串首次出现的位置，并且返回一个无符号的整型数值（与 string :: size_type 的类型相同）。调用函数 find 的语法是：

```
strVar.find (strExp)
```

这里的 strVar 是一个 string 类型变量，strExp 是一个字符串表达式。
36. 函数 find 的参数（也就是 strExp）也可以是单个字符。
37. 如果查找成功，函数 find 返回子字符串首次匹配时，第一个字符在整个字符串中的位置。如果查找不成功，函数 find 返回 npos 值。
38. 函数 substr 返回字符串中特定的子字符串。调用函数 substr 的语法是：

```
strVar.substr (expr1, expr2)
```

这里 expr1 和 expr2 是无符号的整型表达式。表达式 expr1 用于指定字符串中某个位置（子字符串的开始位置）；表达式 expr2 用于指定返回的子字符串长度。
39. 函数 swap 用于交换两个字符串变量中的内容。调用函数 swap 的语法是：

```
strVar1.swap (strVar2);
```

其中 strVar1 和 strVar2 是 string 类型变量。该语句用来交换 strVar1 和 strVar2 中的值。

8.7 练习

1. 判断下面语句的正误。

a. 下面是合法的 C++ 枚举数据类型：

```
enum romanNumerals( I, V, X, L, C, D, M);
```

b. 考虑下面定义：

```
enum cars{ Ford, GM, Toyota, Honda};
cars domesticCars = Ford;
```

语句：

```
domesticCars = domesticCars + 1;
```

将 domesticCars 的值设置成 GM。

c. 函数的返回值可以是枚举类型。

d. 可以从标准输入设备中直接输入枚举类型数据。

e. 仅可以用在枚举类型数据上的两个算术运算符是增量运算符和减量运算符。

f. 枚举类型取值域中的值称为枚举分量。

g. 下面位于同一 C++ 函数（块结构）中的语句是合法的：

```
enum mathStudent{ Bill, John, Lisa, Ron, Cindy, Shelly};
enum historyStudent{ Amanda, Bob, Jack, Tom, Susan};
```

h. 下面语句创建匿名类型：

```
enum {A, B, C, D, F} srudentGrade;
```

i. 在使用扩展名为 h 的头文件的程序中也可以使用名字空间机制。

j. 假设 str = "ABCD"，在语句 "str[1] = 'a';" 执行过后，str 中的值变为 "aBCD"。

k. 假设 str = "abcd"，在语句 "str = str + "ABCD";" 执行过后，str 中的值变为 "ABCD"。

2. 编写 C++ 程序以完成以下功能：

a. 定义一个枚举类型 bookType，其中枚举分量有 Math，CSC，English，History，Physics 和 Philosopy。

b. 定义一个 bookType 类型的变量 book。

c. 将 Math 赋给变量 book。

d. 将枚举分量中的下一个值赋给 book。

e. 输出变量 book 中的值。

3. 给定：

```
enum currencyType{ Dollar, Pound, Frank, Lira, Mark};
currencyType currency;
```

下面哪些语句是合法的？

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;
```

选择正确的答案。

a. `static_cast<int>(wheat)`的值是 0　　(i) `true`　　(ii) `false`

b. static_cast<cropType>(static_cast<int>(wheat) - 1) 的值是：

(i) true (ii) false

c. rye > wheat (i) true (ii) false

d. for(crop = wheat; crop <= oats; ++crop)

cout<<"*";

输出：***** (i) true (ii) false

5. 下面程序的输出结果是什么？

```
    #include <iostream>                                   //Line 1
    int main()                                            //Line 2
    {
    cout<<"Hello World! "<<endl;                          //Line 3
    return 0;                                             //Line 4
}
```

6. 下面程序中的错误在哪里？

```
#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. 下面程序中的错误在哪里？

```
#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. 下面程序中的错误在哪里？

```
#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 = "ABCDEFGHIJK";
   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，其中的枚举分量值有scalene，isosceles，equilateral和noTriangle。
   b. 编写函数 triangleShape。该函数有三个数字类型参数，每个参数表示三角形中一个边的长度。该函数将返回三角形的形状（注意：在三角形中，两边之和要大于第三边）。
   c. 编写程序提示用户输入三角形各边的长度，并输出三角形的形状。
2. 重新编写第4章中编程练习12（移动电话服务供应商客户账单）。使程序中所有命名常量都定义在一个名字空间中。
3. 程序范例"Pig Latin 字符串"将字符串转换成相应的Pig Latin形式，但是该程序每次只能处理一个单词。重新编写该程序，使其可以处理未定长度的文本。如果其中的某一个单词后面有标点符号，则将该标点符号置于转换后的Pig Latin形式字符串的后面。例如，Hello 的 Pig Latin 形式是ello-Hay。假如规定文本中可以包含以下标点符号：,（逗号）、.（句号）、?（问号）、;（分号）和:（冒号）。
4. 编写一个程序，该程序用来计算美国联邦税。确定美国联邦税可计税收入的方法如下：对于未婚者，收入中$4 000以下部分不计税；对于已婚者，收入中的$7 000以下部分不计税。每个人还可以将总收入中最高为6%部分支付养老金，这部分也不计税。
   计税规则如下：如果可计税收入在：

   - $0到$15 000之间，需要缴纳可计税收入的15%。

- \$15 001到\$40 000之间，需要缴纳\$2 250再加上可计税收入中超过\$15 000以上部分的25%。
- \$40 000以上，需要缴纳\$8 460再加上可计税收入中超过\$40 000以上部分的35%。

程序提示用户输入下面信息：

- 婚姻状况
- 如果婚姻状况是married（已婚），提示用户输入14岁以下孩子的个数
- 总收入（如果婚姻状况是married，并且夫妇双方都有收入，则输入双方的总收入）
- 总收入中支付养老金的百分比

程序中至少应该包含以下函数：

a. 函数getDate：该函数提示用户输入所需的数据

b. 函数taxAmount：该函数用于计算并返回缴税金额

为计算计税收入，总收入应减去标准免税额、支付的养老金额以及每个14岁以下孩子的\$1 500的免税额之和。

5. 如果三个整数a，b，c满足等式$a^2+b^2=c^2$，那么这组整数就被称为是一组勾股数（Pythagorean triple）。例如，整数3，4，5是一组勾股数，因为$3^2+4^2=5^2$。这里采用下面的方法来构造勾股数。假设m和n是整型变量，如果满足：$a=m^2-n^2$，$b=2mn$，$c=m^2+n^2$，则a，b，c就是一组勾股数。编写程序提示用户输入m和n的值，并输出与m和n相对应的勾股数。
6. （分数计算器）编写一个程序，该程序允许用户对分数进行算术计算。分数的形式是a/b，这里a和b都是整数并且b不等于0。该程序允许用户通过菜单选择运算符（+，-，*或/）并可以输入分数的分子和分母。程序中至少要包含以下函数：

   a. 函数menu：该函数用来告诉用户程序的作用，解释怎样输入数据，并且允许用户选择运算符。

   b. 函数addFractions：该函数有4个参数，分别代表两个分数的分子和分母。该函数将两个分数相加，并且返回和的分子和分母。

   c. 函数subtractFractions：该函数有4个参数，分别代表两个分数的分子和分母。该函数将两个分数相减，并且返回差的分子和分母。

   d. 函数multiplyFractions：该函数有4个参数，分别代表两个分数的分子和分母。该函数将两个分数相乘，并且返回积的分子和分母。

   e. 函数divideFractions：该函数有4个参数，分别代表两个分数的分子和分母。该函数将两个分数相除，并且返回商的分子和分母。

   输出范例：

   ```
   3 / 4 + 2 / 5 = 23 / 20
   2 / 3 * 3 / 5 = 6 / 15
   ```

   注意：不需要简化最后的结果。

第9章 数组和字符串

本章要点：

- 了解数组
- 理解怎样定义数组以及怎样操纵数组中的数据
- 理解“数组下标越界”的含义
- 理解数组处理中的一些限制
- 理解怎样将数组作为参数传递给函数
- 了解 C-string
- 理解怎样使用字符串函数处理 C-string
- 理解怎样向 C-string 输入数据以及怎样从 C-string 输出数据
- 了解平行数组（Parallel array）
- 理解怎样操纵二维数组中的数据
- 了解多维数组

前面章节中介绍的主要是简单数据类型。第2章中讲过，C++的数据类型可以分为三类，其中之一就是构造数据类型。第9章及后面的几章将主要讨论构造数据类型。

注意，简单数据类型变量在某一时刻只能存储一个值。相反地，构造类型数据项是由一个或者多个其他类型的数据项组成的。也就是说，简单数据类型是构造类型的构成元素。下面首先讨论的构造类型是数组，其他构造类型将在第11章和第12章中介绍。

在正式定义数组之前，先来考虑下面的问题。假定要编写一个C++程序，该程序读入5个数字，计算这5个数字的和，并且以相反的顺序输出这5个数字。

第5章已经介绍了怎样读入数字，求和，以及输出数字。与前面遇到的问题不同的是，这里要求以相反的顺序输出这些数字。在输出第5个数字之前，不能输出前4个数字，以此类推。这意味着在以相反的顺序输出所有数字之前，需要先将这些数字存储起来。根据以前学过的知识，可以写出以下程序来完成这个功能。

```
//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;
}
```

上面的程序是正确的。但是如果要读入100（或者更多）个数字，并且以相反的顺序输出这些数字，就必须要在程序中定义100个变量并使用多个cin和cout语句。因此，对于处理大量数据，这种类型的程序是不能令人满意的。

注意上面程序的一些特点：

1. 在以相反的顺序输出数字之前，必须首先定义存储这些数字的变量。
2. 所有变量的类型相同——int类型。
3. 变量定义的方式说明，存储这些数字的变量名中除了最后一个字符（数字）外，其余部分都是相同的。

在这些特点中，不难看出：总共需要定义5个变量（特点1）。如果可以将变量名中的最后一个字符（数字）存储到计数器变量中，并且使用for循环从0到4记数这5个变量的输入和输出，则会很方便。最后，由于所有变量的类型都相同，最好可以在一条简单的语句中定义多个变量，并指定这些变量的类型。

在C++中，通过一种称为数组的数据结构可以完成上述功能。

9.1　数组

数组（Array）是由固定数目元素组成的数据结构，并且所有元素的类型都相同。一维数组是数组中所有元素以列表形式排列。本节将主要讨论一维数组。二维数组和多维数组将在本章中的后面部分介绍。

一维数组的定义形式如下所示：

```
dataType arrayName[intExp];
```

这里，intExp是任意值为正数的int型表达式。intExp用来指定数组中元素的个数。

例9.1　语句：

```
int num[5];
```

定义一个有5个元素的数组num。数组元素分别为num[0]，num[1]，num[2]，num[3]和num[4]，并且每个元素都是int类型。图9.1说明了数组num的组成。

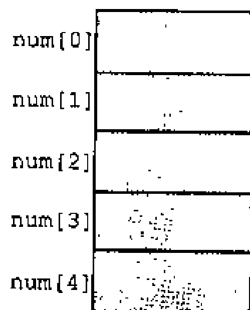


图9.1　数组num

9.1.1 访问数组元素

访问数组元素的语法形式是：

```
arrayName[ indexExp]
```

这里，indexExp是非负的int型表达式，称为下标。下标用于指定数组中元素的位置。

在C++中，[ ]是一个运算符，称为数组下标运算符。在C++中，数组下标从0开始。

考虑下面语句：

```
int list[ 10] ;
```

该语句定义了一个有10个元素的int型数组list。数组元素分别为list[0]，…，list[9]，即定义了10个变量（如图9.2所示）。

赋值语句：

```
list[ 5]  = 34;
```

将34存储到list[5]中，它是数组list中的第6个元素（如图9.3所示）。

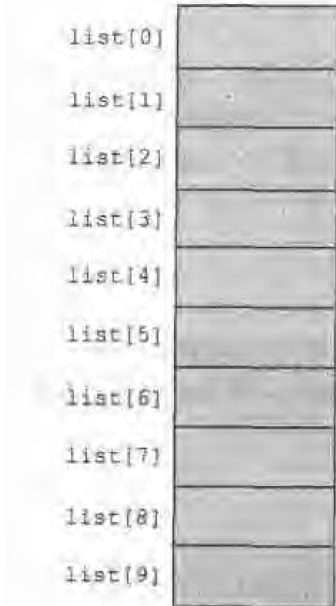


图9.2　数组list

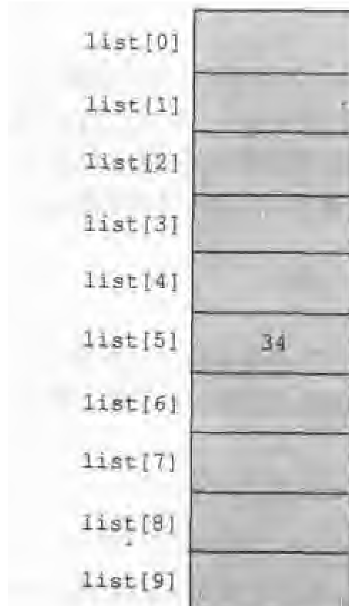


图9.3　在语句“list[5] = 34;”执行后的数组list

假设i是int型变量，则赋值语句：

```
list[ 3]  = 63;
```

与下面语句的作用相同：

```
i = 3;
list[ i]  = 63;
```

如果i是4，赋值语句：

```
list[ 2 * i - 3]  = 58;
```

将58存储到list[5]中，因为2 * i - 3的值是5。程序首先计算数组下标表达式的值，该值用来指定数组元素的位置。

接下来，考虑下面语句：

```
list[ 3]  = 10;
list[ 6]  = 35;
list[ 5]  = list[ 3]  + list[ 6] ;
```

第1条语句将10存储到list[3]中，第2条语句将35存储到list[6]中，第3条语句将list[3]和list[6]中的值相加，结果存储到list[5]中（如图9.4所示）。

list[0]	
list[1]	
list[2]	
list[3]	10
list[4]	
list[5]	45
list[6]	35
list[7]	
list[8]	
list[9]	

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　也可以按照以下方式定义数组：

```
const int arraySize = 10;
int list[ arraySize] ;
```

也就是说，可以先定义命名常量，然后使用命名常量的值来定义数组，指定元素个数。

注意： *在定义数组之前，必须要知道数组中元素的个数。例如，不能用以下方式来定义数组：*

```
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
```

在程序执行后，第2行语句提示用户输入数组元素的个数。第3行语句将数组元素的个数存储到变量arraySize中。在编译器编译第1行语句时，变量arraySize中的值未知。也就是说，在整个编译过程中，变量arraySize中的值始终是未知的。因此，在编译器编译第5行语句时，数组中的元素个数未知，编译器将不知道应该给数组分配多少内存空间。在第14章，将介绍怎样在程序执行的过程中指定数组元素的个数，以及怎样使用指针来定义数组。在程序运行中使用指针创建的数组称为动态数组。但是，目前在定义数组时，必须指定数组元素的个数。

9.1.2　处理一维数组

在一维数组上的基本运算包括初始化、输入数据、输出数据和查找最大/最小元素。此外，如果数组是数字类型，还可以计算数组中所有元素的和以及平均值。每一种运算都涉及到逐一访问数组元素，这很容易通过循环来实现。例如，假设有下面定义语句：

```
int list[100];          //list is an array of the size 100
int i;
```

下面伪代码中的for循环的功能是从第1个元素开始逐一访问数组list中的所有元素。

```
for (i = 0; i < 100; i ++)        //Line1
    process list[i]               //Line2
```

如果要向数组list中输入数据，那么应该在第2行语句中包含输入语句，如cin语句。例如，下面语句从键盘中读入100个数字，并将这些数字存储到数组list中。

```
for(i = 0; i < 100; i++)          //line1
    cin >> list[i];               //line2
```

同样，如果要从数组list中输出数据，那么应该在第2行语句中包含输出语句。例9.3进一步说明了怎样处理一维数组。

例9.3 本例说明了怎样使用循环结构来处理数组。本例中的定义语句如下所示：

```
double sale[10];
int index;
double largestSale, sum, average;
```

第1条语句定义了一个有10个元素的double型数组。其他定义语句很简单，不需要再做解释。

a. 初始化数组：下面的循环语句将数组sale中的每个元素都初始化为0.0。

```
for(index = 0; index < 10; index ++)
    sale[index] = 0.0;
```

b. 将数据读入到数组中：下面的循环语句将数据读入到数组中。为了简便起见，假设所有的数据都从键盘读入。

```
for(index = 0; index < 10; index ++)
    cin >> sale[index];
```

c. 输出数组元素：下面的循环语句用于输出数组中元素。为了简便起见，假设所有的数据都输出到屏幕上。

```
for(index = 0; index < 10; index ++)
    cout << sale[index] <<" ";
```

d. 计算数组中各元素的和与平均数：正如数组sale的名字含义所示，该数组存储的是一些销售数据。而通常情况下，需要计算总的销售数量和平均销售数量。下面的C++代码用于计算数组sale中各元素的和与平均数。

```
sum = 0;
for(index = 0; index < 10; index++)
    sum = sum + sale[index];

average = sum / 10;
```

e. 数组中的最大元素：现在要讨论查找数组中最大元素（也就是具有最大值的元素）的算法。在通常情况下，用户更关心的是数组中最大元素的位置。当然，如果知道元素的位置（即数组中最大元素的下标），可以很容易地确定最大元素中的值。所以，下面将要讨论确定数组中最大元素下标的算法。在本例中，将要确定数组sale中最大销售量的下标。

假定数组sale中最大元素的下标存储在变量maxIndex中。解决该问题的算法也很简单。首先，假定sale中第一个元素是该数组中的最大元素，所以先将maxIndex初始化为0。然后，将maxIndex

所指的元素与数组中所有其他元素做比较。一旦发现数组中存在比maxIndex所指的元素大的元素，立即更新变量maxIndex，使maxIndex指向新的最大元素。用C++实现的上述算法如下所示：

```
maxIndex = 0;
for(index = 1; index < 10; index++)
    if(sale[maxIndex] < sale[index])
       maxIndex = index;

largestSale = sale[maxIndex];
```

下面，通过举例来研究该算法的执行过程。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

现在我们来查找数组sale中的最大元素。在for循环之前，maxIndex的初始值是0。在for循环中，index被初始化为1。下面列出了每次for循环中maxIndex，index，sale[maxIndex]和sale[index]的值：

index	maxIndex	sale[maxIndex]	sale[index]	sale[maxIndex] < sale[index]
1	0	12.50	8.35	12.50 < 8.35的值是false
2	0	12.50	19.60	12.50 < 19.60的值是true; maxIndex = 2
3	2	19.60	25.00	19.60 < 25.00的值是true; maxIndex = 3
4	3	25.00	14.00	25.00 < 14.00的值是false
5	3	25.00	39.43	25.00 < 39.43的值是true; maxIndex = 5
6	5	39.43	35.90	39.43 < 35.90的值是false
7	5	39.43	98.23	39.43 < 98.23的值是true; maxIndex = 7
8	7	98.23	66.25	98.23 < 66.65的值是false
9	7	98.23	35.64	98.23 < 35.64的值是false

在for循环执行之后，maxIndex = 7，即数组sale中最大元素的下标。因此，largestSale = sale[maxIndex] = 98.23。

注意： 根据查找数组中最大元素算法，可以编写出查找数组中最小元素的算法(参见本章末的编程练习2)。

既然已经知道怎样定义和处理数组，现在可以重写本章开头处讨论的程序。注意，本程序读人5个数字，求和，并以相反的顺序输出这些数字。

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;
}
```

程序运行结果 *在本程序运行中，用户输入的数据加有阴影。*

```
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

考虑下面定义：

```
double num[10];
int i;
```

如果i = 0，1，2，3，4，5，6，7，8或者9，那么称num[i]是合法的。

如果数组下标index >= 0并且index <= arraySize - 1，称数组下标在界内（In Bounds）；如果index < 0或者index > arraySize - 1，称数组下标越界（Out of Bounds）。

遗憾的是，C++并没有提供防范数组下标越界的机制。也就是说，C++并不检查下标值是否在0到arraySize - 1的合法范围之内。如果指定的下标值在合法范围之外，同时程序试图访问该下标值指定的内存单元，这将导致访问或修改并非你想存取的内存单元中的内容。因此，如果在程序执行过程中发生了数组下标越界，将会产生难以预料的后果。程序员必须保证数组下标在合法范围之内。

下面的循环语句将会导致数组下标越界：

```
for(i = 0; i <= 10; i++)
    list[i] = 0;
```

这里假定数组list中只有10个元素。当i = 10时，循环测试条件i <= 10的值是true，所以执行下面的循环体。这将导致将数值0存储到list[10]中。而在逻辑上，list[10]并不存在。

9.1.4 在定义时初始化数组

与所有的简单数据类型相同，数组也可以在定义时初始化。例如，下面的C++语句定义了一个由5个元素组成的数组list，并且同时完成初始化过程：

```
double sales[5] = {12.25, 32.50, 16.90, 23, 45.68};
```

花括号中各数据项之间以逗号分隔。这里sales[0] = 12.25，sales[1] = 32.50，sales[2] = 16.90，sales[3] = 23.00，sales[4] = 45.68。

如果在定义数组时进行初始化，则不需要指定数组中元素的个数。数组中元素的个数取决于花括号中初始值的个数。但是，必须要将方括号置于数组的名称后面。所以，上面的语句等价于：

```
double sales[] = {12.25, 32.50, 16.90, 23, 45.68};
```

虽然在定义时初始化数组可以不指定数组元素的个数，但是指定数组元素个数却是一个良好的习惯。

在定义时部分地初始化数组

在定义数组时进行初始化，可以不必给出所有数组元素的初始值，这被称为在定义时部分地初始化数组。但是，在定义时部分地初始化数组有一些注意事项。下面的举例有助于理解这些注意事项。

语句：

```
int list[10] = {0};
```

将 list 定义为有 10 个元素的 int 型数组，并将所有元素都初始化为 0。语句：

```
int list[10] = {8, 5, 12};
```

将 list 定义为有 10 个元素的 int 型数组，并将 list[0]初始化为 8，将 list[1]初始化为 5，将 list[2]初始化为 12，将其余的所有元素初始化为 0。也就是说，在数组初始化语句中没有指定值的所有数组元素，都将自动初始化为 0。这里，必须要注意数组定义中是否指定了数组元素个数。例如，语句：

```
int list[ ] = {5, 6, 3};
```

将 list 定义为有 3 个元素的 int 型数组，并将 list[0]初始化为 5，将 list[1]初始化为 6，将 list[2]初始化为 3。而语句：

```
int list[25] = {4, 7};
```

将 list 定义为有 25 个元素的 int 型数组，并将 list[0]初始化为 4，将 list[1]初始化为 7，将其余的所有元素初始化为 0。

9.1.5　数组处理中的一些限制

假设 x 和 y 是两个相同类型的数组，并且这两个数组中数组元素的个数相同，比如说 25。再假设 x 已经初始化，现在要将数组 x 中的数值拷贝到数组 y 中。下面的语句是非法的：

```
y = x; //illegal
```

要将一个数组中的数值拷贝到另一个数组中，必须分别拷贝每个数组元素——也就是说，每次只能拷贝一个数组元素。这可以通过下面的循环语句来完成：

```
for(j = 0; j < 25; j++)
    y[j] = x[j];
```

与此类似，数组比较、将数据输入到数组中和输出数组中的数据，必须通过逐一地访问数组元素来完成。因此，下面语句是非法的：

```
cin>>x;         //illegal
cout<<y;        //illegal
if(x <= y)      //illegal
.
.
.
```

也就是说，C++不允许在数组上进行整体操作（Aggregate Operation）。数组上的整体操作是指将整个数组作为一个整体来处理的任何操作。

9.1.6 数组作为函数参数

既然已经知道怎样处理数组，那么很自然地会产生一个疑问：怎样将数组作为参数传递给函数？

只能通过引用参数传递 在C++中，数组只能通过引用参数传递。

因为数组只能通过引用参数传递，所以在定义数组类型形参时不需使用符号&。

在将一维数组定义为形参时，通常可以不指定数组元素的个数。如果在将一维数组定义为形参时指定了数组元素个数，编译器也将忽略所指定的数组元素个数。

下面函数的参数可以是任意的int类型数组：

```
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，所以该函数可以将任何具有5个元素的int类型数组初始化为0。但是如果数组元素个数改变，就必须编写另外的函数。然而，如果在函数头中增加一个形参（如size），使用size中的值来控制for循环的次数（即用size取代5），并且在函数调用的同时提供实际数组名和size，就可以初始化任意元素个数的数组了。改写后的函数如下所示：

```
void initialize(int list[], int size)
{
      int count;
      for(count = 0; count < size; count++)
          list[count] = 0;
}
```

函数initialize中的第一个参数是任意元素个数的int型数组。在调用函数initialize时，实际数组的元素个数作为第二个参数传递给函数initialize。

常量数组作为形参

前面已经讲过，当形参是引用参数时，只要形参中的值发生改变，对应的实参中的值也跟着改变。虽然数组总是作为引用参数传递给函数，但是还是有办法防止函数改动实参中的数值。这可以通过在形参定义中使用保留字const来实现。考虑下面函数：

```
void example(int x[], const int y[], int sizeX, int sizeY)
{
      .
      .
      .
}
```

这里的函数example可以修改数组x中的值，但不可以修改数组y中的值。任何试图修改y的语句将会导致编译错误。如果不希望函数改动数组中的值，那么一个很好的程序设计技巧是将数组定义成为常量形参。

例9.5 下例说明了怎样编写函数进行数组处理以及怎样将数组定义成为形参。

```
      //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可以正常工作，数组y中元素的个数不能少于数组x中元素的个数。

数组的基地址

数组的基地址（Base Address）是指数组中第一个元素的（内存空间）地址。例如，如果list是一个一维数组，那么list的基地址是元素list[0]的地址。

在将数组作为参数传递给函数时，实际上是将数组的基地址传递给形参。

如果允许数组作为值参数传递给函数，那么计算机就要为形参（数组）的每个元素分配存储空间，并将实际参数拷贝到相应的形参中。如果数组中的元素很多，这种函数传递方式不仅要占用大量内存空间，而且要耗费大量的计算机时间在拷贝数据上。

函数不能返回数组类型的返回值

C++不允许函数返回数组类型的返回值。前面介绍的函数sumArray和largestElement，返回int类型的数值。

例9.6 下面的程序说明了在函数调用中数组是怎样作为实参传递给函数的。

```
//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 definitions of the functions initializeArray, fillArray,
//and so on here. Example 9-6 gives the definitions of these
//functions.
```

程序运行结果　*在本程序运行中，用户输入的数据加有阴影。*

```
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: 33 77 25 63 56 48 98 39 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
```

程序的运行结果简单明了。第 1 行中的语句定义了一个有 10 个元素的数组 listA，并将该数组中每个元素初始化为 0。第 2 行中的语句定义了一个有 10 个元素的数组 listB。第 4 行中的语句调用函数 printArray，输出 listA 中的数值。第 11 行中的语句调用函数 fillArray 将数据输入到数组 listA 中。第 16 行中的语句调用函数 sumArray 输出数组 listA 中所有元素的和。第 18 行中的语句输出数组 listA 中最大元素中的值。

9.1.7　整数类型和数组下标

在学习本节之前，需要有第 8 章中枚举数据类型和 typedef 语句两个小节的知识。

除了整数之外，C++允许任何整数类型用做数组下标。这种灵活性大大地增强了程序的可读性。考虑下面语句：

```
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;
```

下面语句将 Red 颜色油漆的销售量增加 75.69：

```
paintSale[Red] = paintSale[Red] + 75.69;
```

显而易见，上面代码比使用整数作为数组下标的代码更加直观。基于这个原因，应该在程序中尽可能地使用枚举类型或者其他整数类型作为数组下标。

定义数组的其他方法

假设一个班级有 20 个学生，需要编写程序记录他们的学习成绩。因为在不同的学期学生人数可能发生变化，所以除了在定义数组时指定数组元素个数之外，还可以用下面方法来定义数组：

```
const int noOfStudents = 20;
int testscore[noOfStudents];
```

另一种定义数组的方法是：

```
const int size = 50;               //Line 1
typedef double list[size];         //Line 2

list a;                            //Line 3
list mylist;                       //Line 4
```

第 2 行语句定义了数据类型 list，该数据类型是一个有 50 个元素的 double 类型数组。第 3 行和第 4 行语句定义了两个变量 a 和 mylist。这两个变量都是有 50 个元素的 double 类型数组。当然，上面语句的作用等价于：

```
double a[50];
double mylist[50];
```

9.2 C-string（字符数组）

之所以到目前为止还没有讨论字符串数组，是因为字符串数组具有特殊性，处理字符串数组与处理其他类型数组的方法有很大不同。C++ 提供了许多（预定义）函数来处理字符数组。

字符数组 所有元素都是 char 类型的数组。

使用最为广泛的字符集是 ASCII 和 EBCDIC。ASCII 字符集中的第一个字符是不可打印的空字符（Null Character）。前面已经讲过，在 C++ 中，空字符用 '\0'（反斜线后面加数字 0）来表示。

语句：

```
ch = '\0';
```

将空字符存储到 ch 中，这里 ch 是 char 类型变量。

以后将会发现，空字符在处理字符数组中起到十分重要的作用。因为空字符的编码值是 0，所以在 char 类型数据集里空字符小于其他所有字符。

字符数组常被称为 C-string，但是字符数组和 C-string 之间还存在细微的差别。字符串是零个或多个字符的序列，并且字符串必须由双引号括起来。在 C++ 中，C-string 是以空字符结尾的，即 C-string 中的

最后一个字符总是空字符。字符数组可以不含有空字符，但是C-string最后一个字符却必须是空字符。换句话说，在C-string中空字符只可以出现在最后一个位置上。C-string也存储在一维字符数组内。

在本节中，除非特殊声明，字符串指的就是C-string。下面是字符串的举例：

```
"John L. Johnson"
"Hello there."
```

从字符串的定义中可以很明显地分辨出'A'和"A"的差别。前一个是字符'A'，而后一个则是字符串"A"。由于字符串是以空字符结尾的，所以"A"由两个字符'A'和'\0'组成。类似地，"Hello"由6个字符组成：'H'，'e'，'l'，'l'，'o'和'\0'。存储'A'只需要一个char类型存储空间，而存储"A"却需要两个char类型存储空间——一个存储'A'，另一个存储'\0'。类似地，如果要在计算机内存中存储字符串"Hello"，则需要6个char类型存储空间。

考虑下面语句：

```
char name[16];
```

该语句将数组name定义为有16个元素的char类型数组。因为字符串是以空字符结尾的，而name中只有16个元素，所以在name中存储字符串最多可以有15个字符。如果将一个有10个字符的字符串存储到name中，那么name的前11个元素中存有数据，而后5个元素中没有数据。

语句：

```
char name[16] = {'J', 'o', 'h', 'n', '\0'};
```

定义了一个有16个元素的char类型数组，并将字符串"John"存储到char中。C++允许在char类型数组定义时，以字符串表示法的形式初始化数组。因此，上面语句等价于：

```
char name[16] = "John";          //Line A
```

前面已经讲过，如果在定义数组的同时初始化数组，则可以不必指定数组中元素的个数。语句：

```
char name[] = "John";            //Line B
```

定义了一个足够多元素的数组（在这里是5个元素）并且将"John"存储到里面。上面两条语句虽然都是将字符串"John"存储到字符数组中，但是语句行A中定义的name有16个元素，而语句行B中定义的name有5个元素。

其他类型数组上的规则也同样适用于字符类型数组。考虑下面语句：

```
char studentName[26];
```

假设要将"Lisa L. Johnson"存储到studentName中。因为数组不支持如赋值和比较等整体操作，所以下面语句是非法的：

```
studentName = "Lisa L. Johnson"; //Illegal
```

C++提供了一系列字符串操作的函数，这些函数都包含在头文件cstring中。其中经常会用到的三个函数是：strcpy（字符串拷贝，将一个字符串拷贝到另一个字符串变量中，即赋值）、strcmp（字符串比较，用来比较字符串）和strlen（字符串长度，用来求出字符串的长度）。表9.1中总结了这些函数。

表 9.1 函数 strcpy，strcmp 和 strlen

函数	作用
strcpy(s1,s2)	将字符串 s2 拷贝到字符串变量 s1 中 s1 的长度不应小于 s2 的长度
strcmp(s1,s2)	如果 s1 小于 s2，返回一个负数；如果 s1 等于 s2，返回 0；如果 s1 大于 s2，返回一个正数
strlen(s)	返回字符串 s 的长度，末尾的空字符不计算在内

为了使用这些函数，必须使用include语句将头文件cstring包含到程序中。因此，程序中必须包含下面语句：

```
#include <cstring>
```

9.2.1 字符串比较

在C++中，字符串比较是通过逐个比较每个字符的编码值来完成的。假定使用的是ASCII字符集。

1. 字符串"Air"小于字符串"Boat"，因为"Air"中第一个字符'A'小于"Boat"中第一个字符'B'。
2. 字符串"Air"小于字符串"An"，因为虽然两个字符串的第一个字符相同，但是"Air"中第二个字符'i'小于"An"中第二个字符'n'。
3. 字符串"Bill"小于字符串"Billy"，因为虽然两个字符串中前5个字符都相同，但是"Bill"中第5个字符'\0'（空字符）小于"Billy"中第5个字符'y'（因为在C++中，字符串是以空字符结尾的）。
4. 字符串"Hello"小于字符串"hello"，因为"Hello"中第一个字符'H'小于"hello"中第一个字符'h'。

正如上面所示，函数strcmp将第一个字符串参数与第二个字符串参数逐个字符地进行比较。

例9.7 假如有下面语句：

```
char studentName[ 21] ;
char myname[ 16] ;
char yourname[ 16] ;
```

下面语句说明了字符串函数是如何工作的：

语句	结果
`strcpy(myname,"John Robinson");`	Myname = John Robinson
`strlen("John Robinson");`	返回字符串"John Robinson"的长度，13
`int len;`	
`len = strlen ("Sunny Day");`	将9存储到len中
`strcpy(yourname,"Lisa Miller");`	yourname = Lisa Miller
`strcpy(studentName,yourname);`	studentName = Lisa Miller
`strcmp("Bill", "Lisa");`	返回一个小于0的值
`strcpy(yourname, "Kathy Brown");`	yourname = Kathy Brown
`strcpy(myname, "Mark G. Clark");`	myname = Mark G. Clark
`strcmp(myname, yourname);`	返回一个大于0的值

注意： 在本章中，我们将C-string定义成零个或多个字符的序列。C-string括在一对双引号中，同时C-string是以空字符结尾的，所以虽然字符串"Hello"在双引号里面只有5个字符，但是在C-string "Hello"中有6个字符。因此，要想将"Hello"存储到计算机内存中，至少需要有6个元素的字符数组。字符串长度指的是括在双引号内字符串的实际长度。例如，字符串"Hello"的长度是5。因此，虽然在逻辑意义上C-string由零个或多个字符组成，但是在物理意义（即，将C-string存储到计算机内存中）上C-string至少由一个字符组成。这里必须要注意的是，计算机内存中C-string末尾的空格符在比较C-string时起到了很关键的作用，特别是在如"Bill"和"Billy"这样的字符串比较中。

9.2.2 输入输出字符串

前面已经讲过，应用在数组上的大多数规则对于字符串同样适用。在数组上不能进行例如赋值和比较这种整体操作。甚至数组的输入和输出都是基于元素的。然而，C++允许在字符数组上进行输入和输出字符串（即字符数组）整体操作。

为了讨论方便，定义下面的字符数组：

```
char name[ 31] ;
```

9.2.3　字符串输入

因为在字符串输入时允许整体操作，所以语句：

```
cin >> name;
```

将输入的字符串存储到name中。输入的字符串长度必须小于或等于30。如果输入的字符串长度是4，计算机将这4个字符连同空字符'\0'存储到内存中。如果输入的字符串长度大于30，因为C++不提供数组下标检查机制，所以计算机继续将多出来的部分存储到name后面的内存单元中，这会导致严重的错误！因为存储在name后面单元中的数据将会丢失。

注意，析取运算符>>将会略掉输入字符串中的所有前置空白字符，并且在遇到字符串后的第一个空白字符或者非法数据时停止向当前变量中读入数据。因此，不能使用析取运算符>>来读取含有空白字符的字符串。例如，如果姓和名之间以空格符作为分隔符，就不能使用析取运算符读入到name中。

怎样将含有空格符的字符串读入到字符数组中呢？再一次，函数get派上了用场。在前面章节介绍的函数get（第3章）每次只能读入一个字符，但是函数get也可以用来读入字符串。可以使用带有两个参数的get函数来读入字符串。第一个参数是字符串变量；第二个参数用来指定读入到字符串变量中的字符个数。

用输入流变量cin的get函数读入字符串的语法是：

```
cin.get(str, m+1);
```

本语句将m个字符，或者在遇到换行符之前的所有字符读入到str中。换行符本身并不存储到str中。如果输入的字符串少于m个字符，则输入在遇到换行符时终止。

考虑下面语句：

```
char str[ 31] ;
cin.get(str, 31);
```

如果输入是：

```
William T. Johnson
```

那么"William T. Johnson"将被存储到str中。如果输入是：

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

那么"Hello there. My name is Mickey"将被存储到str中。

现在假设有下面的语句：

```
char str1[ 26] ;
char str2[ 26] ;
char discard;
```

和两行输入字符串：

```
Summer is warm.
Winter will be cold.
```

进一步假设要将第1行字符串存储到str1中，将第2行字符串存储到str2中。str1和str2最多可以存储长度为25的字符串。因为第1行中字符的个数是15，所以在遇到'\n'时终止读入。为了输入第2行字

符串，必须读掉第1行末尾的换行符。下面的语句序列用于将第1行字符串存储到str1中，将第2行字符串存储到str2中：

```
cin.get(str1, 26);
cin.get(discard);
cin.get(str2, 26);
```

9.2.4 字符串输出

字符串输出是在字符数组上进行的另一种整体操作。可以通过输出流变量，如cout和插入运算符来输出字符串。例如，语句：

```
cout << name;
```

将name中的内容输出到屏幕上。插入运算符将连续输出name中的字符，直到遇到空字符（'\0'）为止。因此，如果name的长度是4，上面语句只输出4个字符。如果name中不含有空字符，该语句将输出很奇怪的字符串。因为插入运算符将继续从与name毗邻的内存单元中输出数据，直到遇到'\0'为止。

9.2.5 在执行时指定输入输出文件

第3章已经介绍了怎样从文件中读人数据。在此后的章节中，在打开输入文件时总是将文件的名字包含在open语句中。而这样一来，程序总是从同一个文件中读取数据。在实际应用中，数据很可能存储在不同存储介质上的不同文件中。而且为了比较方便，有时希望分别处理不同的文件或者将输出数据存储到不同的文件中。为了可以有效地完成这种任务，用户一般希望在运行时指定输入/输出文件的名字，而不是在编写代码时指定文件名。C++允许用户在程序运行时指定输入/输出的文件名。

考虑下面语句：

```
ifstream infile;
ofstream outfile;

char fileName[ 51] ;    //assume the file name is at most
                        //50 characters long
```

下面语句提示并且允许用户在程序执行时指定输入/输出文件名：

```
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
```

string数据类型和输入/输出文件

在第8章中，我们讨论了string数据类型。这里需要指出的是：string数据类型的值不是以空字符('\0')结尾的。string类型变量当然也可以用来读入并存储输入/输出的文件名。但是，函数open的参数必须是以空字符结尾的字符串——即C-string。因此，如果用string类型变量来读入输入/输出文件的名，并使用该变量来打开文件，那么必须首先将变量中的值转换成C-string（即以空字符结尾的字符串）。头文件string中包含了函数c_str，该函数将string类型值转换成为以空字符结尾的字符数组（即C-string）。使用函数c_str的语法是：

```
strVar.c_str()
```

这里strVar是string类型变量。

下面语句说明了怎样使用string类型变量在程序执行过程中读入输入/输出文件的名字，并且打开这些文件：

```
ifstream infile;
string   fileName;

cout<<"Enter the input file name: ";
cin>>fileName;

infile.open(fileName.c_str());    //open the input file
```

当然，必须要在程序中包含头文件string。输出文件也要遵守相同的规则。

9.3 平行数组

如果两个（或多个）数组的对应单元中存储相关的信息，就称这两个（或多个）数组是平行数组（Parallel Array）。

为了可以在学期期末将学生的考试成绩公布于众，需要记录下学生的考试分数和ID号码。假设某个班级中有50个学生，并且他们的ID号码的长度是5位。因为有50个学生，所以需要50个变量来存储他们的ID号码，还需要50个变量来存储他们的成绩。可以定义两个数组：int类型的studentId和char类型的courseGrade。每个数组各有50个元素。studentId[0]和courseGrade[0]中要存储第一个学生的ID号码和考试成绩，studentId[1]和courseGrade[1]中要存储第二个学生的ID号码和考试成绩，以此类推。

语句：

```
int studentId[50];
char courseGrade[50];
```

定义了上述的两个数组。

9.4 二维数组和多维数组

本章的下面部分将讨论二维数组和多维数组。

本章前面几节介绍了怎样使用一维数组来处理数据。如果数据以列表形式出现,则可以使用一维数组。但是，有时数据以二维表的形式出现。例如，本地某经销商库存的每种颜色的汽车数量记录。该经销商销售的汽车共有6款，每款车型都有5种颜色。

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9.6 库存表

上面的数据以二维表的形式存在。该表有30个数据项，每个数据项都是一个整数。因为所有的数据项都是相同的数据类型，所以可以定义一个有30个元素的int类型的一维数组。该一维数组的前5个元素存储表中第1行中的数据，接下来的5个元素存储表中第2行中的数据，以此类推。也就是说，可以将二维表形式的数据存储在一维数组里。

使用上述方法处理一维数组元素的算法显得有些复杂。因为你必须要留意哪里是二维表一行数据的末尾，哪里是下一行数据的开始。这就需要正确地计算数组元素的下标。在C++中，可以使用二维数组来简化处理以二维表形式给出的数据。本节将首先讨论怎样定义二维数组，然后再介绍怎样操作二维数组中的数据。

二维数组 以行和列（即二维）形式排列的固定数目元素的集合，每个元素的类型都相同。

定义二维数组的语法是：

```
dataType arrayName[intExp1][intExp2];
```

这里intExp1和intExp2是值为正整数的表达式。两个表达式intExp1和intExp2，分别用来指定数组中行和列的数目。

语句：

```
double sales[10][5];
```

定义了一个10行5列的二维数组sales，该数组中每个元素都是double类型。跟一维数组类似，行下标分别是0…9，列下标分别是0…4，如图9.7所示。

sales	[0]	[1]	[2]	[3]	[4]
[0]					
[1]					
[2]					
[3]					
[4]					
[5]					
[6]					
[7]					
[8]					
[9]					

图9.7 二维数组sales

9.4.1 访问数组元素

要想访问二维数组的元素，必须要给出两个下标：一个行下标和一个列下标。

访问二维数组元素的语法是：

```
arrayName[indexExp1][indexExp2]
```

这里indexExp1和indexExp2是值为非负整数的表达式。indexExp1指定行下标，indexExp2指定列下标。

语句：

```
sales[5][3] = 25.75;
```

将25.75存储到数组sales中行下标为5，列下标为3的元素中（第6行和第4列）（如图9.8所示）。

sales

	[0]	[1]	[2]	[3]	[4]
[0]					
[1]					
[2]					
[3]					
[4]					
[5]				25.75	
[6]					
[7]					
[8]					
[9]					

sales[5][3]

图9.8　sales[5][3]

假设：

```
int i = 5;
int j = 3;
```

则前面的语句：

```
sales[5][3] = 25.75;
```

与语句：

```
sales[i][j] = 25.75;
```

的作用相同。

也就是说，数组下标可以是变量。

9.4.2　在定义时初始化二维数组

与一维数组相同，二维数组也可以在定义时初始化。下面的语句有助于理解这一过程。

考虑下面语句：

```
int board[4][3] = {{2, 3, 1},
                   {15, 25, 13},
                   {20, 4, 7},
                   {11, 18, 14}};
```

该语句定义了一个4行3列的二维数组board。第1行中的元素分别是2，3和1；第2行中的元素分别是15，25和13；第3行中的元素分别是20，4和7；第4行中的元素分别是11，18和14。图9.9说明了数组board。

board

	[0]	[1]	[2]
[0]	2	3	1
[1]	15	25	13
[2]	20	4	7
[3]	11	18	14

图9.9　二维数组board

在定义时初始化数组需要注意：

1. 所有在一行中的元素用花括号括住，并且用逗号分隔。
2. 所有行都用花括号括住。
3. 对于数字类型数组，如果某一行中存在没有指定数值的元素，那么这些没有指定数值的元素将初始化为 0。此时，至少应有一个值用来初始化二维数组。

9.4.3 二维数组和枚举数据类型

注意：在学习本章之前，需要有第 8 章中的知识。

可以使用枚举类型作为数组下标。考虑下面语句：

```
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] ;
```

在上面的语句中，定义了 carType 和 colorType 两个枚举类型和一个 6 行 5 列的二维数组 inStock。假设每一行对应于一款车型，每一列对应于一种颜色。也就是说，第 1 行对应于 GM 汽车，第 2 行对应于 Ford 汽车，以此类推。类似地，第 1 列对应于颜色 Red，第 2 行对应于颜色 Brown，以此类推。inStock 中每一个数据项都表示某款特定颜色汽车的库存数量（如图 9.10 所示）。

inStock	[Red]	[Brown]	[Black]	[White]	[Gray]
[GM]					
[Ford]					
[Toyota]					
[BMW]					
[Nissan]					
[Volvo]					

图 9.10 二维数组 inStock

语句：

```
inStock[ 1][ 3]  = 15;
```

与下面的语句等价（如图 9.11 所示）：

```
inStock[ Ford][ White]  = 15;
```

第二种语句的含义很明显——颜色为 White 的 Ford 汽车的库存数量是 15。本例说明了使用枚举类型数据可以提高程序的可读性并且易于实现。

instock[Ford][white]

inStock	[Red]	[Brown]	[Black]	[White]	[Gray]
[GM]					
[Ford]				15	
[Toyota]					
[BMW]					
[Nissan]					
[Volvo]					

图 9.11　inStock[Ford][White]

9.4.4　处理二维数组

可以通过三种方式处理二维数组：

1. 处理整个数组
2. 处理数组中的某一行，称为行处理
3. 处理数组中的某一列，称为列处理

初始化和输出都是处理整个二维数组的例子。在某一行（列）中查找最大元素和求某一行（列）中所有元素的和都是行（列）处理的例子。为讨论方便，定义如下的变量：

```
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 9.12 说明了数组 matrix。

matrix	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]						
[2]						
[3]						
[4]						
[5]						
[6]						

图 9.12　二维数组 matrix

因为二维数组中每个元素的类型都相同，所以行元素和列元素的类型也是相同的。这就意味着二维数组中的每一行和每一列都是一维数组。因此，处理二维数组中某一行或某一列的算法，与处理一维数组的算法相同。我们将使用上面定义的二维数组 matrix 做进一步说明。

假设要处理数组 matrix 中行下标为 5 的一行（即数组 matrix 中的第 6 行）元素。数组 matrix 中行下标为 5 的一行元素是：

```
matrix[5][0], matrix[5][1], matrix[5][2], matrix[5][3], matrix[5][4]
matrix[5][5]
```

这些元素的第一个下标（行下标）都是 5；第二个下标（列下标）的变化范围是从 0 到 5。因此，可以使用下面的 for 循环（伪代码）来处理行下标是 5 的一行元素：

```
for(col = 0; col < columes; col++)
    process matrix[5][col]
```

很明显，该 for 循环等价于下面的 for 循环：

```
row = 5;
for(col = 0; col < columes; col++)
    process matrix[row][col]
```

类似地，假设要处理 matrix 中列下标为 2 的一列，即 matrix 的第 3 列。该列中的元素有：

```
matrix[0][2], matrix[1][2], matrix[2][2], matrix[3][2],  matrix[4][2],
matrix[5][2], matrix[6][2]
```

这里第二个下标（列下标）是 2；第一个下标（行下标）的变化范围是从 0 到 6。在这里，使用下面的 for 循环（伪代码）来处理 matrix 中列下标为 2 的一列：

```
for(row = 0; row < rows; row++)
    process matrix[row][2]
```

很明显，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]
```

下面将讨论几个特定的算法。

初始化

假设要将行下标为 4 的一行元素，即第 5 行，初始化为 0。正如前面介绍过的，下面的 for 循环将行下标为 4 的一行元素初始化为 0：

```
row = 4;
for(col = 0; col < columes; col++)
    matrix[row][col] = 0;
```

还可以将第一个下标（行下标）也放入一个 for 循环中，这样就可以将整个 matrix 初始化为 0。通过使用下面的嵌套 for 循环，将 matrix 中的每个元素都初始化为 0。

```
for(row = 0; row < rows; row++)
    for(col = 0; col < columns; col++)
        matrix[row][col] = 0;
```

输出

通过使用嵌套 for 循环，可以输出 matrix 中的每个元素。下面的嵌套 for 循环将按行输出 matrix 中元素的值。

```
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循环将数据输入到行下标为4的行中，也就是matrix中的第5行：

```
row = 4;
for(col = 0; col < columns; col++)
    cin >> matrix[row][col];
```

类似地，也可以将行下标放入for循环中，把所有输入数据读入到整个matrix数组中：

```
for(row = 0; row < rows; row++)
    for(col = 0; col < columns; col++)
        cin >> matrix[row][col];
```

按行计算元素的和

下面的for循环计算matrix中行下标为4的所有元素的和：

```
sum = 0;
row = 4;
for(col = 0; col < columns; col++)
    sum = sum + matrix[row][col];
```

再一次，可以将行下标放入循环中，分别计算每一行元素的和。下面的C++代码分别计算每一行元素的和：

```
//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;
}
```

按列计算元素的和

与计算每一行中元素的和相同，下面的嵌套for循环分别计算每一列元素的和：

```
//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

前面已经讲过，在二维数组中还可以进行查找每一行、每一列和两个对角线中最大元素的操作。下面给出实现这些操作的C++代码。

下面的for循环用来查找行下标是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++代码用来查找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(row = 1; row < rows; row++)
        if(largest < matrix[row][col])
            largest = matrix[row][col];

    cout<<"Largest element of col "<<col+1<<" = "<<largest<<endl;
}
```

对角线倒置

假设matrix是方形数组，即数组的行数和列数相同，所以matrix有主对角线和次对角线。为了更具体一些，假设有下面语句：

```
const int row = 4;
const int columns = 4;
```

matrix中主对角线元素是matrix[0][0]，matrix[1][1]，matrix[2][2]和matrix[3][3]；反对角线元素是matrix[0][3]，matrix[1][2]，matrix[2][1]和matrix[3][0]。

下面编写C++代码，将matrix两条对角线中的元素倒置。

假设数组matrix如图9.13所示。

在两条对角线中的元素倒置之后，matrix如图9.14所示。

很明显，为了倒置主对角线元素，需要：

1. 交换matrix[0][0]和matrix[3][3]中的值
2. 交换matrix[1][1]和matrix[2][2]中的值

为了倒置反对角线元素，需要：

1. 交换matrix[0][3]和matrix[3][0]中的值
2. 交换matrix[1][2]和matrix[2][1]中的值

matrix	[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9.13　二维数组matrix

matrix	[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9.14　倒置后的数组matrix

下面的for循环用于倒置对角线元素：

```
//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： 虽然二维数组可以是任何行数和列数的，但是，上面的代码只能转换方形数组对角线中的值。

9.4.5 将二维数组作为参数传递给函数

可以将二维数组作为参数传递给函数，这是通过引用参数的形式来实现的，即将数组的基地址（实际参数中第一个元素的地址）传递给形参。如果matrix是一个二维数组，那么matrix[0][0]就是该数组的第一个元素。

C++采用按行存放的形式将二维数组存放在计算机内存中。也就是说，首先存放第1行元素，然后存放第2行元素，然后再存放第3行元素，以此类推。

在将一维数组定义成函数的形参时，可以不必指定数组的大小。由于C++按行存放二维数组，为了正确计算某一元素的位置，编译器必须要知道每一行从哪里开始，到哪里结束。因此，在将二维数组定义成函数的形参时，可以不必指定第一维的大小，但是必须指定第二维的大小；也就是说，必须指定列数。

考虑下面的函数定义：

```
void example(int table[][5], int rowsize)
{
    .
    .
    .
}
```

该函数的第一个参数是一个二维数组，该数组没有指定行数，但是指定的列数是5。在函数调用时，实参的列数必须和形参的列数匹配。

9.4.6 字符串数组

假定要将一个名字序列按照字母表的顺序排序。因为每个名字都是一个字符串，所以采用数组来存储这个名字序列是很方便的。在C++中，字符串只可以存储在string数据类型中或者字符数组中（C-string）。而且需要注意的是，在标准C++的某些编译器中string数据类型不可用。本节将分别介绍这两种处理字符串的方法。

字符串数组和string数据类型

用string数据类型来处理字符串序列是很方便的。假设该序列中最多有100个名字。可以定义一个有100个元素的string类型数组：

```
string list[100];
```

因为string数据类型支持赋值、比较、输入和输出等基本操作，所以可以采用本章开始部分中介绍的处理一维数组的方法来处理list中的数据。

字符串数组和C-string（字符数组）

假设一个序列中有100个字符串，每个字符串（例如名字）的最大长度是15个字符。可以定义一个100行16列的二维数组来存储这些字符串：

```
char list[100][16];
```

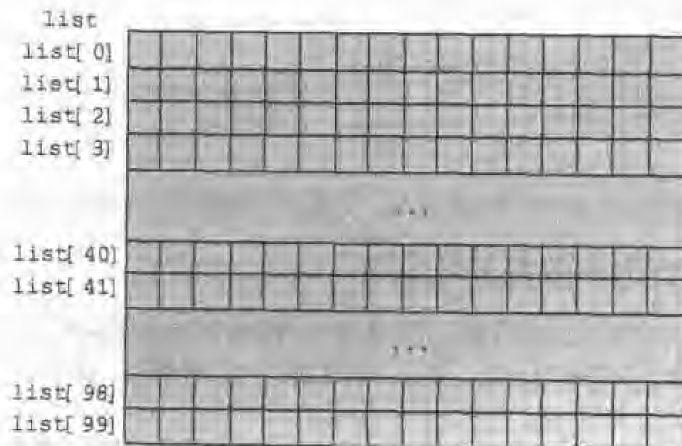


图9.15 字符串数组list

这里，每个list[j](0 <= j <=99)存储的都是一个最大长度为15的字符串。下面的语句将"Snow White"存储到list[1]中（如图9.16所示）：

```
strcpy(list[1], "Snow White");
```

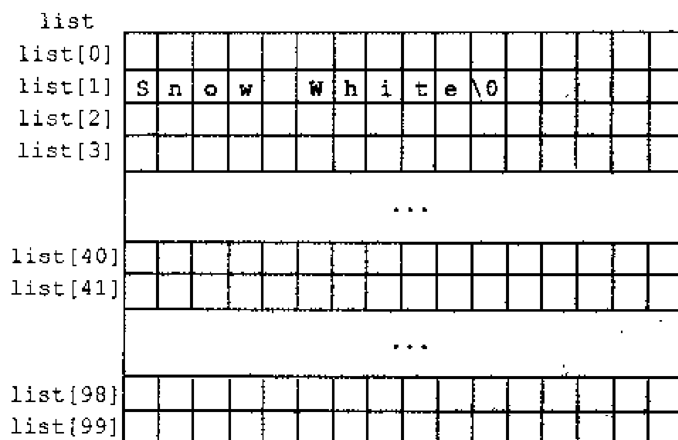


图9.16　数组list中的list[1]

假设要按行输入字符串并存储到list中，下面的for循环可以完成这项任务：

```
for(j = 0; j < 100; j++)
    cin.get(list[j], 16);
```

下面的for循环按行输出字符串：

```
for(j = 0; j < 100; j++)
    cout << list[j] << endl;
```

还可以使用其他的字符串函数（如strcmp和strlen）和for循环来处理list。

注意： 因为在string数据类型上已经定义了许多操作，例如赋值、连接和关系运算，所以如果你使用标准C++头文件，并且编译器支持string数据类型，那么建议使用string数据类型来处理字符串序列。

9.4.7　定义二维数组的另一种方法

注意： 略过本小节内容，不会影响后面章节的学习。

如果知道程序中使用的二维表的行数和列数，则可以先用typedef定义一个二维数组数据类型，然后用该数据类型来定义数组。例如，考虑下面语句：

```
const int numberOfRows = 20;
const int numberOfColumns = 10;

typedef int tableType[numberOfRows][numberOfColumns];
```

上面的语句定义了一个二维数组数据类型tableType。现在我们用该数据类型来定义数组。语句：

```
tableType matrix;
```

定义了一个20行10列的二维数组matrix。

也可以在函数定义中使用该数据类型来定义形参，如下面代码：

```
void initialize(tableType table)
{
      int row;
```

```
    int col;

    for(row = 0; row < numberOfRows; row++)
        for(col = 0; col < numberOfColumns; col++)
            table[row][col] = 0;
}
```

该函数可以将任何tableType类型的变量作为参数，这些变量都是二维数组，并初始化数组中所有元素值为0。

通过定义数据类型，在定义二维数组变量、形参或者在函数调用传递实参时，不必再对列数做检查。

9.5 多维数组

在本章中，我们将数组定义为同一类型固定数目的元素集合。一维数组中的元素以一维列表的形式排列；二维数组中的元素以二维表的形式排列。还可以定义三维或者更多维数的数组。在C++中，对数组维数没有限制。下面是数组的通用定义。

数组 以*n*维表形式排列的固定数目元素的集合，称为*n*维数组。

定义*n*维数组的语法是：

```
dataType arrayName[intExp1][intExp2]...[intExpn];
```

这里intExp1，intExp2，…，intExpn都是值为正整数的常量表达式。

访问n维数组元素的语法是：

```
arrayName[indexExp1][indexExp2]...[indexExpn]
```

这里indexExp1，indexExp2，…，和indexExpn都是值为非负整数的表达式。indexExpi用来指定第i维元素的位置。

例如，语句：

```
double carDealers[10][5][7];
```

将carDealers定义成为三维数组。第一维的大小是10，第二维的大小是5，第三维的大小是7。第一维下标的变化范围是从0到9，第二维下标的变化范围是从0到4，第三维下标的变化范围是从0到6。数组carDealers的基地址是第一个元素的地址——即carDealers[0][0][0]的地址。数组carDealers中元素的个数是10 * 5 * 7 = 350。

语句：

```
carDealers[5][3][2] = 15564.75;
```

将数值15564.75存储到元素carDealers[5][3][2]中。

可以使用for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。

在将多维数组定义成函数的形参时，可以不指定该数组中第一维的大小，但是必须指定该数组中其他维的大小。多维数组只可以作为引用参数传递给函数，并且函数不能返回一个数组类型的返回值。C++对数组下标不做检查。

9.6　程序范例：代码校验

当消息在通道中传递时，通常采用位序列，即0和1的序列的形式。由于传送通道中噪声等影响，传送的消息有可能发生错误。也就是说，送到目标地址中的信息同开始发送时可能不一样，某些位发生了改变。校验传递信息的方法有许多，其中一种就是传送两次信息。在目标地址，逐位比较这两份信息。如果每个位都相同，就认为传送的信息没有发生错误。

现在编写一个程序，检查信息在传送过程中有没有发生错误。为了简便起见，假设用来表示信息的密码是一个数字（0到9）序列，并且信息的最大长度是250个数字，并且信息中的第一个数字表示信息的长度。例如，如果密码是：

```
7 9 2 7 8 3 5 6
```

那么该信息的长度是7个数字，该信息将被传送两次。

上面的信息传送为：

```
7 9 2 7 8 3 5 6 7 9 2 7 8 3 5 6
```

输入　含有密码及其拷贝的文件。

输出　密码、密码拷贝和一条信息——如果接收的代码没有错误，则以下面的形式输出：

```
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

因为要将密码和其拷贝逐位地做比较，所以要先读入密码，并将其存储到数组中。然后读入密码拷贝中的第一个数字，并与密码中的第一个数字做比较，以此类推。如果一旦发现对应的数字不相同，就立即输出一条信息告诉用户传送有误。因为信息的最大长度是250个数字，所以应该定义一个有250个元素的数组。密码及其拷贝中的第一个数字，记录的都是该代码的长度。经过讨论，得出下面的算法：

1. 打开输入文件和输出文件。
2. 如果输入文件不存在，退出程序。
3. 读入密码的长度。
4. 如果密码的长度大于250，终止程序，因为代码的最大长度是250个数字。
5. 读入密码，并将其存储到数组中。
6. 读入密码拷贝的长度。
7. 如果密码和其拷贝的长度相同，进一步比较两个代码；否则，输出错误信息。

为了简化函数main，编写函数readCode读入密码，函数compareCode比较代码。

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，将布尔变量lengthCodeOK（引用参数）的值置为false，函数终止。返回到调用函数的lengthCodeOK中的值表明密码是否读取成功。如果代码长度小于250，函数readCode读入该代码，并将其存储到数组中。因为输入的数据

存储在文件中，而该文件在函数main中打开，所以与输入文件相关联的输入流变量必须通过函数传递给函数readCode。而且，在读入密码长度和代码本身之后，函数readCode必须将这些数据返回给函数main。因此，函数readCode有4个参数：一个输入文件流变量、存储密码的数组、代码长度和bool类型参数lengthCodeOK。函数readCode的定义如下所示：

```
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];
}
```

函数compareCode 该函数逐位比较密码及其拷贝。因此，该函数必须要访问存储密码的数组和存储密码长度的变量。密码拷贝及其长度存储在输入文件中。因此，与输入文件相关联的输入流变量必须作为参数传递给该函数。函数compareCode接下来将密码及其拷贝做比较，并且输出相应的信息。因为输出数据要存储到文件中，所以与该输出文件相关联的输出流变量也必须作为参数传递给该函数。因此，函数compareCode有4个参数：一个输入文件流变量、一个输出文件流变量、存储密码的数组和密码的长度。经过上面讨论，得出函数compareCode的算法如下所示。

a. 定义变量。
b. 将bool变量codeOK的值置为true。
c. 读入密码拷贝的长度。
d. 如果密码与其拷贝的长度不相同，则输出相应的错误信息并且终止该函数。
e. 对于输入文件中的每个数字：
  e.1. 读入密码拷贝中的下一个数字。
  e.2. 将密码及其拷贝中相应的数字输出。
  e.3. 如果相应的数字不相同，输出错误信息，并且将bool变量codeOK的值置为false。
f. 如果bool变量codeOK的值是true
    输出信息，说明密码传送无误。
  否则
    输出错误信息。

根据上面的算法，函数compareCode的定义是：

```
void compareCode(ifstream& infile, ofstream& outfile,
                 int list[], int length)
{
           //Step a
  int length2;
  int digit;
  bool codeOk;
```

```
    int count;

    codeOk = true;                                          //Step b

    infile>>length2;                                        //Step c

    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++)                 //Step e
    {
        infile>>digit;                                      //Step e.1
        outfile<<setw(7)<<list[count]<<setw(20)<<digit;     //Step e.2
        if(digit != list[count])                            //Step e.3
        {
          outfile<<"          code digit not the same"<<endl;
          codeOk = false;                                   //Step e.3
        }
        else
           outfile<<endl;
    }

    if(codeOk)                                              //Step f
        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. 定义变量
2. 打开文件
3. 调用函数 readCode 读入密码
4. if（密码的长度 <= 250）

   调用函数 compareCode 比较两个代码。

   else

   输出相应的错误信息。

完整的程序代码清单

```
//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<<"          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;
}
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
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
```

输出文件中的数据: (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.
```

9.7 程序范例：文本处理

（行和字母计数）我们现在来编写一个程序，该程序读入给定文本，原样输出该文本，并且输出行数以及文本中每个字符出现的次数。大写字母和小写字母将被视为相同的字母。

因为总共有26个字母，所以需要一个有26个元素的数组来存储字母出现次数。我们还需要一个变量来存储行数。

文本存储在文件 textin.txt（假定在软盘驱动器 A 中）中。输出数据将存储在文件 textout.out 中。

输入 含有需要处理文本的文件。

输出 含有文本、行数和每个字母在文本中出现的次数的文件。

问题分析和算法设计

根据输出的要求，很明显我们必须按照原样输出文本。也就是说，如果原文本中含有空白字符，那么输出的文本中也要含有空白字符。而且，还要记录文本中的行数。因此，必须要知道每行到哪里结束，也就是说必须要捕获换行字符。这种需要告诉我们，不能使用析取运算符来处理输入文件。因为还需要记录字母出现的次数，所以要用函数 get 来读入文本。

让我们先来确定程序中的变量，这将有助于简化后面的讨论。

变量 因为需要记录行数和字母出现的次数，所以需要定义一个存储行数的变量和存储26个字母出现次数的变量。我们定义了一个有26个元素的数组来存储字母出现次数。因为需要在输入文件中逐个地读入字符，所以需要一个存储依次读入的字符的变量。因为需要从文件中读取数据，并将结果输出到文件中，所以需要一个打开输入文件的输入流变量和一个打开输出文件的输出流变量。下面语句定义了函数 main 中所需的所有变量：

```
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
```

在上面定义中，letterCount[0]存储字母 A 出现的次数，letterCount[1]存储字母 B 出现的次数，以此类推。很明显，变量 lineCount 和数组 letterCount 必须初始化为 0。

程序的算法是：

1. 定义变量。
2. 打开输入文件和输出文件。
3. 初始化变量。
4. 当输入文件中还有未处理的字符时：
   - 4.1 对于每一行中的每个字符：
     - 4.1.1 读入并且写出字符
     - 4.1.2 将相应的字符数加 1
   - 4.2 将行数加 1
5. 输出行数和字母数。
6. 关闭文件。

为了简化函数 main，我们将之分为 4 个函数：

- 函数 initialize
- 函数 copyText
- 函数 characterCount
- 函数 writeTotal

下面，将详细的讨论每一个函数。在讨论完这些函数后，将描述函数 main 的算法。

函数 initialize　该函数将变量 lineCount 和数组 letterCount 初始化为 0。因此，该函数有两个参数：一个对应于变量 lineCount，另一个对应于数组 letterCount。很明显，对应于变量 lineCount 的参数应该是引用参数。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;
}
```

函数 copyText　该函数从输入文件中读取一行文本，并将其输出到输出文件中。只要遇到非空格字符，该函数调用函数 characterCount 来更新相应字母的个数。很明显，该函数有四个参数：一个输入文件流变量、一个输出文件流变量、一个 char 变量和一个存储字母个数的数组。

注意，虽然函数 copyText 并不记录字母出现的次数，但是仍然将数组 letterCount 传递给它。之所以这样做，是因为函数 copyText 要调用函数 characterCount，而该函数要使用数组 letterCount 来更新相应字母的个数。因此，必须将数组 letterCount 传递给函数 copyText，以便它可以将数组 letterCount 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
}
```

函数 characterCount　该函数用来增加字母出现的次数。为了可以正确地增加相应字母的个数，必须要知道字母是什么。因此，函数 characterCount 有两个参数：一个是 char 类型变量，一个是记录字母个数的数组。该函数的伪代码如下所示：

a. 将字母转换成大写字母。

b. 找到与该字母对应的数组下标。

c. 如果下标合法，增加相应的字母个数。在这里，必须保证字符是字母。该程序只记录字母的个数，而其他字符（如逗号、破折号和句号）都将被忽略掉。

根据算法，函数的定义如下所示：

```
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]++;
}
```

函数 writeTotal 该函数输出行数和字母数。该函数有三个参数：一个输出文件流变量、行数和记录字母数的数组。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;
}
```

函数 main 的算法描述如下所示。

主要算法

1. 定义变量。
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. 初始化变量，如 lineCount 和数组 letterCount。
6. 读入第一个字符。
7. while（不是输入文件结束时）:
   7.1 处理下一行，调用函数 copyText
   7.2 将行数加 1（将变量 lineCount 中的值加 1）
   7.3 读入下一个字符
8. 输出行数和字母数，调用函数 writeTotal。
9. 关闭文件。

完整的程序代码清单

```
//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;
}
```

9.8 小结

1. 如果某个类型变量在某一时刻只能存储一个值，那么该变量就是简单变量。
2. 构造类型数据由许多数据项组成的。
3. 数组是由固定数目元素组成的构造类型数据。所有元素的类型都相同，可以通过数组中的相对位置来访问数组元素。
4. 一维数组中的元素成列表形式排列。
5. C++ 对数组下标不做检查。
6. 在 C++ 中，数组下标从 0 开始。
7. 数组下标可以是任何值为非负整数的表达式。数组下标的值总要小于数组元素的个数。
8. 除了字符数组（C-string）的输入/输出操作外，数组上没有其他的整体操作。
9. 可以在定义时初始化数组。如果初始化中给定的值少于数组元素的个数，则多出的元素将被自动赋值为 0。
10. 数组的基地址是数组中第一个元素的地址。例如，如果 list 是一维数组，list 的基地址是 list[0]的地址。
11. 如果将一维数组定义成函数的形参，则在定义形参时可以不必指出数组元素个数。如果指定了形参的数组元素个数，那么编译器将忽略掉该数值。
12. 在函数调用语句中，如果实参是数组，那么只需要提供数组的名字。
13. 如果将数组作为函数参数，那么数组只能作为引用来传递。
14. 因为数组参数只能作为引用传递给函数，所以在将数组定义成形参时，不必在数据类型后面加&。
15. 函数不能返回数组类型的返回值。
16. 虽然数组只能作为引用参数传递给函数，但是在将数组定义成形参时，可以在数据类型前使用保留字 const 来防止函数修改数组中的内容。
17. 数组中的每个元素可以作为参数传递给函数。
18. 在 C++ 中，字符串是任意的括在双引号中的字符序列。
19. 在 C++ 中，C-string 是以空字符结尾的。
20. 在 C++ 中，空字符表示为 '\0'。
21. 在 ASCII 码字符集中，空字符的编码值是 0。
22. C-string 存储在字符数组中。
23. 可以在定义时使用字符串来初始化字符数组。
24. C++ 惟一允许的整体操作是 C-string 的输入/输出操作。

25. 头文件cstring中包含了许多作用在C-string上的函数定义。
26. 常用的C-string函数包括：strcpy（字符串拷贝）、strcmp（字符串比较）和strlen（字符串长度）。
27. C-string的比较是逐个字符比较的。
28. 因为字符串存储在字符数组中，所以可以通过数组下标来访问字符串中的某个字符。
29. 平行数组用来存储相互关联的信息。
30. 二维数组是元素成二维表形式排列的数组。
31. 为了访问二维数组中的某个元素，必须提供一对下标：一个是行位置，另一个是列位置。
32. 在二维数组中，行号的变化范围是0到rowsize - 1，列号的变化范围是0到columnsize-1。
33. 如果matrix是二维数组，那么matrix的基地址是matrix[0][0]的地址。
34. 在行处理中，每次处理二维数组中的一行元素。
35. 在列处理中，每次处理二维数组中的一列元素。
36. 在将二维数组定义为函数的形参时，可以不必指定第一维的大小，但是必须指定第二维的大小。
37. 当二维数组作为实参传递给函数时，实参的列数必须和形参的列数相匹配。
38. 在内存中，C++按行存储二维数组。

9.9 练习

1. 判断下面说法的正误。
   a. double数据类型属于简单数据类型。
   b. 一维数组属于构造数据类型。
   c. 数组可以通过值参数和引用参数两种方式传递给函数。
   d. 函数可以返回数组类型的返回值。
   e. 数组的大小必须在编译时确定。
   f. int类型数组上惟一允许的整体操作是增量运算和减量运算。
   g. 假设：

   ```
   int list[10];
   ```

   则语句：

   ```
   list[5] = list[3] + list[2];
   ```

   更新了该数组中第5个元素中的值。
   h. 如果数组下标越界，程序终止并报出错误。
   i. C++允许在字符串上进行某些整体操作。
   j. 语句：

   ```
   char names[16] = "John K. Miller";
   ```

   将names定义成一个有15个元素的数组，因为"John K. Miller"中含有14个字符。
   k. 语句：

   ```
   char str = "Sunny Day";
   ```

   将str定义成没有指定长度的字符串。
   l. 二维数组可以通过值参数或引用参数两种方式传递给函数。
2. 假设：

```
char string15[16];
```

判断下面语句是否合法。如果语句不合法，请说明原因。

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. 假设：

```
char str1[15];
char str2[15] = "Good day";
```

判断下面各语句是否合法。如果语句不合法，请说明原因。

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 that str2."<<endl;
```

4. 假设：

```
char name[8] = "Shelly";
```

如果下面语句输出Shelly，请在后面标记Yes；否则，请在后面标记No，并解释为什么没有输出Shelly。

a. `cout<<name;`

b.
```
for(int j = 0; j < 6; j++)
    cout<<name[j];
```

c.
```
int j = 0;
while(name[j] != '\0')
    cout<<name[j++];
```

d.
```
int j = 0;
while(j < 8)
    cout<<name[j++];
```

5. 假设：

```
char str1[21];
char str2[21];
```

a. 编写C++语句，将"Sunny Day"存储到str1中。

b. 编写C++语句，将str1的长度存储到变量length中。

c. 编写C++语句，将str1中的值拷贝到str2中。

d. 编写C++语句，如果str1的长度小于或者等于str2，输出str1中的值；否则，输出str2中的值。

6. 编写C++语句完成下面功能：

a. 将数组alpha定义成有15个元素int类型数组。

b. 输出alpha第10个元素中的值。

c. 将35存储到alpha第15个元素中。

d. 将alpha中第6个元素与第13个元素的和存储到第9个元素中。

e. 将alpha中第8个元素的值乘以3，减去57，存储到第4个元素中。

f. 输出alpha中各元素的值，每行5个。

7. 考虑下面函数头：

```
void funcOne(int alpha[], int size);
int funcSum(int x, int y);
void funcTwo(const int alpha[], int beta[]);
```

和定义：

```
int list[50];
int Alist[60];
int num;
```

编写C++语句完成下面功能：

a. 调用函数funcOne，实参分别是list和50。

b. 输出函数funcSum的返回值，实参分别是50和list中第4个元素的值。

c. 输出函数funcSum的返回值，实参分别是list中第13个和第10个元素的值。

d. 调用函数funcTwo，实参分别是list和Alist。

8. 假设list是有5个元素的int类型数组。在下面代码执行完后，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list是有5个元素的int类型数组。在下面代码执行完后，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. 假设有下面定义：

```
char name[21];
char yourName[21];
char studentName[31];
```

判断下面语句是否合法。如果语句不合法，请说明原因。

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;
}
```

13. 考虑下面定义：

```
const int carTypes = 5;
const int colorTypes = 6;

double sales[ carTypes] [ colorTypes] ;
```

a. 数组 sales 中有多少个元素？

b. 数组 sales 中有多少行？

c. 数组 sales 中有多少列？

d. 根据 carTypes 统计销售数量，需要进行哪种操作？

e. 根据 colorTypes 统计销售数量，需要进行哪种操作？

14. 编写 C++ 语句完成下面功能：

a. 定义一个有 10 行 20 列的 int 类型数组 alpha。

b. 将 alpha 初始化为 0。

c. 将 1 存储到第 1 行各元素中，将 2 存储到其余的各行元素中。

d. 将 5 存储到第 1 列各元素中，并且要保证后面各列元素中的值都是前面一列中元素值的 2 倍。

e. 按行输出 alpha 中的值。

f. 按列输出 alpha 中的值，即数组中的每一列元素在屏幕上单独占一行。

15. 考虑下面定义：

```
int beta[ 3] [ 3] ;
```

在下面各语句执行之后，beta 中各元素中的值是多少？

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;
```

9.10 编程练习

1. 编写 C++ 程序，该程序定义一个有 50 个元素的 double 类型数组 alpha。然后初始化该数组，使得：前 25 个元素中的值分别是其下标的平方，后 25 个元素的值分别是其下标的 3 倍。以每行 10 个元素的格式输出该数组。

2. 编写C++函数，smallestIndex。该函数的两个参数分别是int类型数组及其大小。该函数的返回值是数组中最小元素的值。然后，编写程序来验证函数是否正确。
3. 编写一个程序读入含有学生成绩的文件，学生成绩的范围是0~200。该程序计算下面各成绩区间中学生的人数：0~24，25~49，50~74，75~99，100~124，125~149，150~174和175~200。输出成绩区间及该区间内学生的人数（使用下面测试数据运行程序：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. 在体操和跳水比赛中，每个选手的得分是去掉一个最高分和一个最低分后各分数之和。编写一个程序，该程序允许用户输入8个裁判打分，并输出选手的得分。输出结果保留两位小数（裁判打分在1分和10分之间，1分是最低分，10分是最高分）。例如，裁判的打分分别是：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。
5. 编写一个程序，该程序提示用户输入一个字符串。然后，该程序按大写字母格式输出该字符串（使用字符数组来存储字符串）。
6. 某位历史教师需要一个判断对错的程序。学生的ID和答案存储在文件中。该文件中的第一个数据项存储的是考试的正确答案，形如：

```
TFFTFFTTTTFFTFTFTFT
```

该文件中其余的数据项的格式是：学生ID，后面跟一个空格，然后是学生的答案。例如，数据项：

```
ABC54301 TFTFTFTT TFTFTFFTTFT
```

表明，学生ID是ABC54301，第1题的答案是T，第2题的答案是F，以此类推。该学生没有回答第9题（答案为空格符）。考试共有20道题，该班有150多名学生。回答正确记2分，回答错误记-1分，不回答记0分。编写程序来处理学生的答案。该程序输出学生ID，试题答案，正确率和考试成绩。假定正确率和考试成绩的对应关系是：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，该程序允许用户输入在某地选举中5位候选人的名字以及他们的得票数。然后程序输出候选人的名字、得票数和得票的百分比。程序应输出选举的获胜者。一个输出的范例为：

```
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，该程序提示用户输入学生姓名及考试分数。该程序输出（假定每个班级中最多有50个学生）：

   (1) 班级平均分

   (2) 考试分数低于班级平均分的学生姓名

   (3) 最高分以及获得最高分的学生姓名
9. 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. 编写函数 setZero，该函数可以将任何 int 类型一维数组初始化为 0。

b. 编写函数 inputArray，该函数提示用户输入 20 个元素，并将这些元素存储到 alpha 里。

c. 编写函数 doubleArray，该函数初始化 beta，使 beta 中的元素的值是 alpha 中对应元素的值的二倍。要保证该函数不会修改 alpha 中的值。

d. 编写函数 copyGamma，该函数将 gamma 中的值拷贝到 inStock 中第 1 行，inStock 中其余各行元素的值是前一行元素相应值的三倍。要保证该函数不会修改 gamma 中的值。

e. 编写函数 copyAlphaBeta，该函数将 alpha 中的元素存储到 inStock 的前 5 行中，并将 beta 中的元素存储到 inStock 的后 5 行中。要保证该函数不会修改 alpha 和 beta 中的值。

f. 编写函数 printArray，该函数可以输出任何 int 类型的一维数组的值。每行输出 15 个元素。

g. 编写函数setInStock，该函数提示用户输入10个数值，并将其存储到inStock的第1列中。inStock 其余各列中各元素的值是两倍的前一列对应元素的值与 delta 中对应元素的差。

h. 编写 C++ 语句，调用 a 到 g 中的函数。

i. 编写 C++ 语句，测试函数 main 中所调用的 a 到 g 中的函数。

10. 编写一个程序，该程序使用一个二维数组存储一年中每个月的最高温度和最低温度。该程序输出一年中的平均最高温度、平均最低温度、最高温度和最低温度。程序中要包含下面函数：

    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：该函数计算并返回一年中的平均最高温度。

    c. 函数 averageLow：该函数计算并返回一年中的平均最低温度。

    d. 函数 indexHighTemp：该函数返回存储一年中最高温度元素的下标。

    e. 函数 indexLowTemp：该函数返回存储一年中最低温度元素的下标。

    （这些函数都是特定参数）

11. 编写一个程序，该程序读入一些正整数，并且输出每个正整数出现的次数。假设最多有 100个正整数，-999 标志输入结束。输出数据应该按照升序排列。例如，对于输入数据：

```
15 40 28 62 95 15 28 13 62 65 48 95 65 62 65 95 95
```

输出的结果是：

```
Number    Count
13        1
15        2
28        2
40        1
48        1
62        3
65        3
95        4
```

12. （客机座位分配）编写一个程序，该程序用于客机座位分配。该客机有 13 排，每排 6 个座位。第 1 排和第 2 排是一等舱，第 3 排到第 13 排是经济舱。第 1 排到第 7 排禁止吸烟。该程序提示用户输入下面信息：

a. 座位类型（一等舱或者经济舱）。

b. 对于经济舱，乘客是否吸烟。

c. 期望的座位。

该程序以下面的形式输出座位分配图：

```
           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     *
```

这里，*表示该座位还没有分配出去；X表示该座位已经分配出去。该程序为菜单驱动，并能显示用户的选择，以及能允许用户做适当选择。

13. 魔术方阵

a. 编写函数 createArithmeticSeq，该函数提示用户输入两个数字，first 和 diff。然后，该函数创建一个有 16 个元素的一维数组，该数组中存储以 first 为首项，diff 为差的等差数列。最后，该函数要输出这个等差数列数组。例如，如果 first = 21，diff = 5，则等差数列是：

```
21 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96
```

b. 编写函数 matricize，该函数的参数是一个有 16 个元素的一维数组和一个 4 行 4 列的二维数组（其他数值，诸如数组大小，也要作为参数传递给该函数）。该函数将一维数组中的值存储到二维数组中。例如，A 是一维数组，B 是二维数组，在将 A 中元素存储到 B 中后，数组 B 是：

```
21 26 31 36
41 46 51 56
61 66 71 76
81 86 91 96
```

c. 编写函数 reverseDiagonal，该函数反置两个对角线中的元素。例如，经过反置后的上面的二维数组变为：

```
96 26 31 81
41 71 66 56
61 51 46 76
36 86 91 21
```

d. 编写函数 magicCheck，该函数的参数是：一个有 16 个元素的一维数组、一个 4 行 4 列的二维数组、一维数组的大小、二维数组的大小。该函数将一维数组中各个元素加起来，除以 4，计算出魔数 magicNumber。然后，该函数将二维数组中各行、各列、各对角线上的元素分别加起来，与 magicNumber 做比较。如果每一行、每一列和每一对角线中元素的和都与 magicNumber 相等，函数输出 It is a magic square；否则，输出 It is not a magic number。不需要输出每一行、每一列和每一对角线中元素的和。

e. 编写函数printMatrix，该函数输出二维数组中各个元素，数组中的每一行在屏幕上占一行。输出的数组格式应该尽量与实际的方阵相似。

f. 上面a到e中各函数应该适用于任意大小的数组。

g. 使用下面函数main来检查a到e中各函数的正确性：

```
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10章 递 归

本章要点：

- 了解递归定义
- 研究递归算法
- 了解递归函数
- 理解怎样根据递归算法实现递归函数

在前面的章节里，通常使用循环技术来设计问题的解决方案。但是对于某些问题，如果采用循环技术，解决方案会变得十分复杂。这一章将引入另外一种解决问题的技术，称为递归，并且给出了几个说明递归的范例。

10.1 递归定义

通过调用自身来解决问题的过程称为递归。递归是解决某些复杂问题的十分有效的方法。我们来考虑一个大家都非常熟悉的问题。

在数学中，整数的阶乘定义如下所示：

```
0! = 1                                   (10.1)
n! = n × ( n-1 )!     if n > 0           (10.2)
```

在这个定义中，0! 定义为1。如果n是比0大的整数，首先计算（n-1)!，然后再乘以n。为了计算（n-1）!，再次应用定义。如果（n-1）>0，应用公式10.2；否则，应用公式10.1。这样，对于大于0的整数n，计算n!时，先要计算（n-1)!(还是阶乘），再将计算结果乘以n。

应用这个定义计算3!，$n = 3$，由于$n > 0$，应用公式10.2：

```
3! = 3 × 2!
```

接下来，计算2!，$n = 2$，由于$n > 0$，应用公式10.2：

```
2! = 2 × 1!
```

现在，计算1!，$n =1$，由于$n > 0$，再次应用公式10.2：

```
1! = 1 × 0!
```

最后，应用公式10.1得到0! =1。把0! 替换到1! 中，得到1! =1，2! = 2 × 1! = 2 × 1 = 2。再依次得到3! = 3 × 2! = 3 × 2 = 6。

公式10.1的定义是直接可以计算出来结果，公式右边不包含阶乘符号。公式10.2的定义给出了一种自身简化形式。公式10.1和公式10.2的阶乘定义称为递归定义。公式10.1称为基本实例（基本实例的结果可以直接获得），公式10.2称为递归归纳。

递归定义　定义本身定义为一种含有自身简化的形式。

从前面的例子（阶乘）可以清楚地看到：

1. 每个递归定义必须有一个（或者多个）基本实例。

2. 递归归约最终归结到基本实例。

3. 基本实例停止递归。

计算机科学中的递归定义与上面的递归定义很相似,下面讨论递归算法和递归函数。通过调用自身简化的新的拷贝解决问题的算法称为递归算法。递归算法必须有一个或者多个基本实例,并且递归归约最终归结到基本实例。调用自身的函数称为递归函数。在当前函数调用结束之前,递归函数体中包含调用自身函数的语句。递归算法通过递归函数实现。

接下来,编写一个实现阶乘的递归函数。

```
int fact(int num)
{
   if(num == 0)
      return 1;
   else
      return num * fact(num - 1);
}
```

图10.1追踪下面语句的执行过程:

```
cout<<fact(4)<<endl;
```

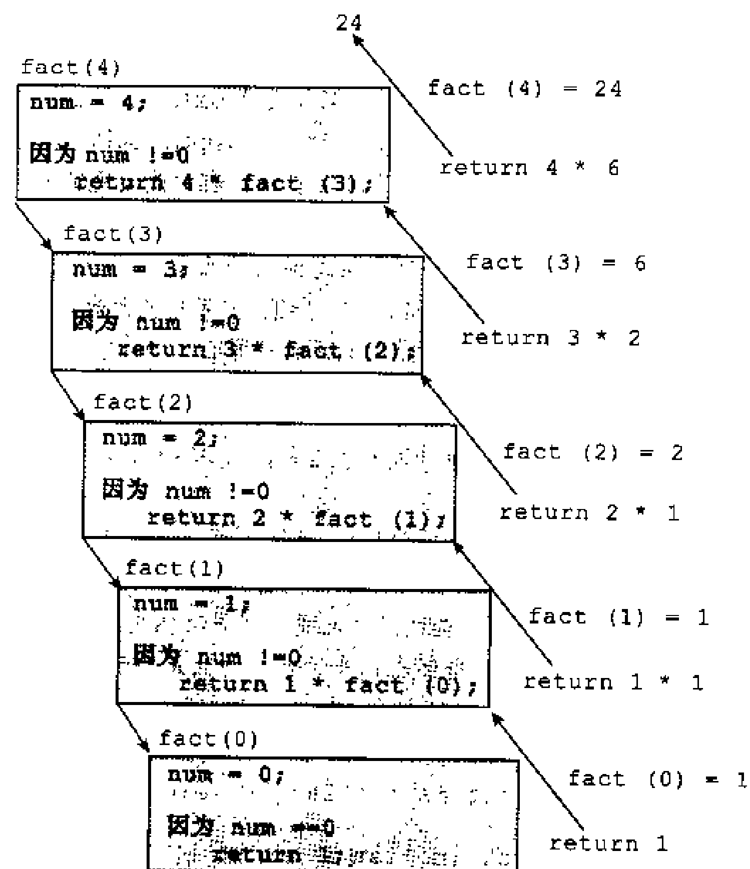


图10.1 函数fact(4)的执行过程

该cout语句的输出是24

在图10.1中,向下的箭头表示对函数fact的连续调用,向上的箭头表示向调用函数返回值。

在递归函数的举例中,包括阶乘函数,注意以下几点:

● 逻辑上,可以认为递归函数有无限的自身拷贝。

- 每个递归调用都有自身的代码、参数和局部变量。
- 完成某个递归调用后，控制返回到先前的调用环境。在控制返回到先前的调用环境之前，当前的递归调用必须已经执行完毕。先前调用函数将从递归调用的返回点开始执行。

在最后一条语句中执行递归调用的递归函数称为尾递归函数（Tail Recursive Function），fact函数是尾递归函数。

10.2 使用递归解决问题

例10.1到例10.3说明了如何在C++中通过递归函数来实现递归算法。

例10.1 查找数组的最大元素

在第9章，我们曾经使用循环算法来查找数组中的最大元素。在本例中，使用递归算法来查找数组中的最大元素。考虑图10.2给出的表。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	5	8	2	10	9	4	

图10.2 含6个元素的表

图10.2中最大的元素是10。

假设list是该数组的名称，并且list[a]…list[b]代表数组元素list[a]，list[a+1]，…，list[b]。例如，list[0]…list[5]代表数组元素list[0]，list[1]，list[2]，list[3]，list[4]，list[5]。类似地，list[1]…list[5]代表list[1]，list[2]，list[3]，list[4]，list[5]。为了编写查找数组中最大元素的递归算法，首先考虑递归形式。

如果list的长度是1，list中只有一个元素，当然它就是最大的元素。假设list的长度大于1，为了在list[a]…list[b]中寻找最大的元素，首先要在list[a+1]…list[b]中寻找最大的元素，并且将这个最大元素与list[a]进行比较。这样，list[a]…list[b]中最大元素的计算公式为：

```
maximum(list[a], largest(list[a+1]…list[b]))
```

应用上面公式在图10.2中查找最大元素，其中有6个元素，list[0]…list[5]，最大的元素是：maximum(list[0], largest(list[1]…list[5]))。

这样list中的最大的元素是list[0]和list[1]…list[5]中最大值之间的较大值。因为表的长度大于1，需要再一次应用上面的公式。list[1]…list[5]中的最大的元素是：maximum(list[1], largest(list[2]…list[5]))，以此类推。每次应用上面的公式在子表中查找最大元素，将下次调用的子表长度减1。最终，子表的长度是1，只包含惟一一个数据元素，当然也就是子表中的最大的元素。从这时开始，将函数值逐层返回给上层递归函数。与上述过程相对应的递归算法的伪代码如下所示：

```
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
```

在数组中查找最大元素算法的C++函数如下所示：

```
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 10.3 给出的表。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	5	10	12	8			

图 10.3 4个元素的表

跟踪下面语句的执行过程：

```
cout<<largest(list,0,3);
```

这里，upperIndex = 3，而且表中有4个元素。图 10.4 跟踪了 largest(list, 0, 3)的执行过程。

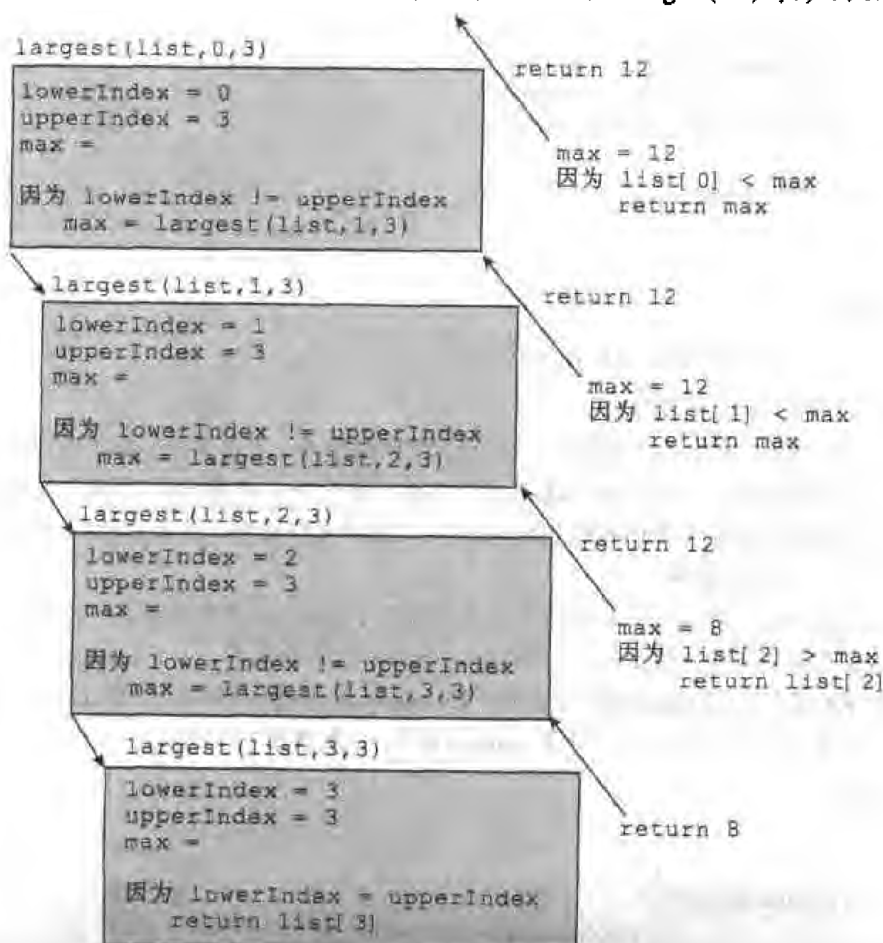


图 10.4 largest(list, 0, 3)的执行过程

表达式 largest(list, 0, 3)的返回值是12，它是list中的最大的元素。

下面的C++程序使用函数 largest 来查找表中的最大元素：

```
//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;
   }
}
```

程序运行结果

```
The largest element in int Array: 76
```

例 10.2 Fibonacci（斐波那契）数列

在第5章中，我们已设计用来计算某个Fibonacci数列的程序。在本章中，编写递归函数rFibNum来计算某个Fibonacci数列。rFibNum函数有3个参数：前两个参数用来指定Fibonacci数列中的前两个数字，参数n用来指定需要计算的Fibonacci数列的第n个数字。函数rFibNum返回Fibonacci数列中的第n个Fibonacci数字。

在Fibonacci数列中，第3个Fibonacci数字是前两个Fibonacci数字的和；Fibonacci数列中的第4个数字是第2个和第3个Fibonacci数字的和。所以，为了计算第4个Fibonacci数字，需要第2个Fibonacci数字和第3个Fibonacci数字（前两个Fibonacci数字的和）相加。下面是计算第n个Fibonacci数字的递归算法。这里，a表示第1个Fibonacci数字，b表示第2个Fibonacci数字，n表示Fibonacci数列的第n个。

$$\text{rFibNum}(a,b,n)=\begin{cases} a & \text{if } n=1 \\ b & \text{if } n=2 \\ \text{rFibNum}(a,b,n-1)+\text{rFibNum}(a,b,n-2) & \text{if } n>2 \end{cases} \quad (10.3)$$

假设要计算：

```
recFibNumber(2,5,4)
```

这里 $a=2$，$b=5$，$n=4$。也就是说，要计算第4个Fibonacci数字，该Fibonacci数列的第1个数字是2，第2个数字是5。因为 n 是4>2，所以：

```
1. rFibNum(2, 5, 4) = rFibNum(2, 5, 3) + rFibNum(2, 5, 2)
```

接下来要计算rFibNum(2, 5, 3)和rFibNum(2, 5, 2)。首先来计算rFibNum(2,5,3)。这里，$a=2$，$b=5$，$n=3$。因为 n 是3，所以：

```
1.a rFibNum(2, 5, 3) = rFibNum(2, 5, 2) + rFibNum(2, 5, 1)
```

这条语句需要计算rFibNum(2, 5, 2)和rFibNum(2, 5, 1)。在rFibNum(2, 5, 2)中，$a=2$，$b=5$，$n=2$。所以，由方程10.3给出的定义得到：

```
1.a.1 rFibNum(2, 5, 2) = 5
```

在rFibNum(2, 5, 1)中，$a=2$，$b=5$，$n=1$。所以，由方程10.3给出的定义得到：

```
1.a.2 rFibNum(2, 5, 1) = 2
```

把rFibNum(2, 5, 2)和rFibNum(2, 5, 1)的值代入（1.a）中得到：

```
rFibNum(2, 5, 3) = 5 + 2 = 7
```

接下来，计算rFibNum(2, 5, 2)。在（1.a.1）中，rFibNum(2, 5, 2) = 5，把rFibNum(2, 5, 3)和rFibNum(2, 5, 2)的值代入(1)中得到：

```
rFibNum(2, 5, 4) = 7 + 5 = 12
```

下面的递归函数实现了这个算法：

```
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);
}
```

跟踪下面语句的执行过程：

```
cout<<rFibNum(2,3,5)<<endl;
```

在上面的语句中，第1个数字是2，第2个数字是3，要计算Fibonacci数列中的第5个Fibonacci数字。图10.5跟踪了表达式rFibNum(2, 3, 5)的执行过程。返回值是13，13是第5个Fibonacci数字。该Fibonacci数列的第1个数字是2，第2个数字是3。

使用rFibNum函数的C++程序如下所示：

```
//Chapter 10: 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);
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

程序运行结果1

```
Enter first Fibonacci number: 2
Enter second Fibonacci number: 5
Enter desired Fibonacci number: 6
Fibonacci number at position 6 is: 31
```

程序运行结果2

```
Enter first Fibonacci number: 3
Enter second Fibonacci number: 4
Enter desired Fibonacci number: 6
Fibonacci number at position 6 is: 29
```

程序运行结果3

```
Enter first Fibonacci number: 12
Enter second Fibonacci number: 18
Enter desired Fibonacci number: 15
Fibonacci number at position 15 is: 9582
```

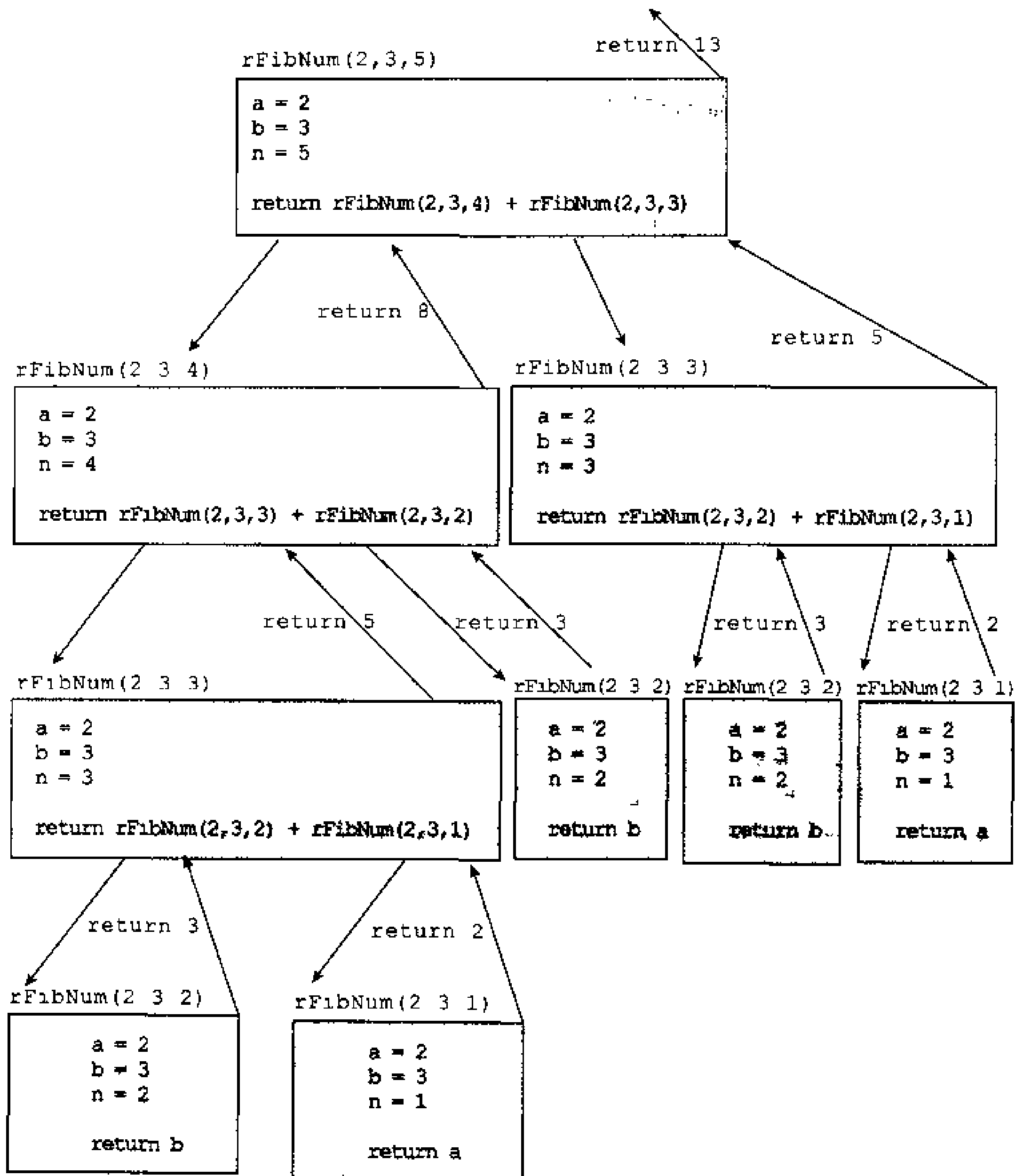


图 10.5 recFibNumber(2, 3, 5)的执行过程

例 10.3 Hanoi 塔

在 19 世纪，一个名为 Hanoi 塔的游戏在欧洲广为流行。该游戏曾是 Brahama 寺庙僧侣们的一项工作。传说在开创世界之初，Brahama 寺庙的僧侣拥有 3 根钻石的柱子，其中 1 根柱子上有 64 个金子做的盘子。64 个盘子从下到上按照由大到小的顺序叠放。僧侣的工作是把这 64 个盘子从第 1 根柱子上移动到第 3 根柱子上，移动的规则如下所示：

1. 每次只能移动一个盘子。
2. 移动的盘子必须放在其中一根柱子上。
3. 大盘子在移动过程中不能放在小盘子之上。

僧侣们被告知一旦他们把所有的盘子从第1根柱子移动到第3根柱子，整个世界也就到了末日。我们的目标是编写一个程序，该程序可以打印出将盘子从第1根柱子转移到第3根柱子的移动顺序。图10.6说明了3个盘子的Hanoi塔问题。

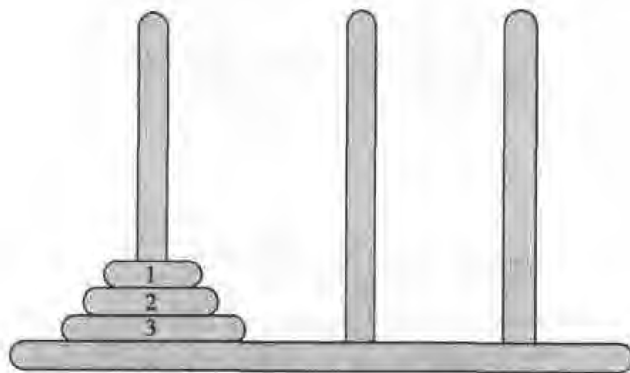


图10.6 3个盘子的Hanoi塔问题

像以前一样，我们用递归来思考。首先考虑第1根柱子上只有1个盘子的情况，这样盘子就可以从第1根柱子直接移动到第3根柱子上。再考虑第1根柱子上有2个盘子的情况。首先，要把第1个盘子从柱子1移动到柱子2。然后，把第2个盘子从柱子1移动到柱子3上。最后，把第1个盘子从柱子2移动到柱子3上。接下来考虑第1根柱子包含3个盘子的情况，这样一直推广到64个盘子的情况（实际上，可以推广到任意数目的盘子）。

假设柱子1上有3个盘子。为了把盘子3移动到柱子3，前两个盘子必须先移动到柱子2。然后，才能将盘子3从柱子1移动到柱子3上。为了把前两个盘子从柱子2移动到柱子3上，使用相同的策略。这一次要把柱子1作为中间柱子，图10.7说明了3个盘子的Hanoi塔问题的解决方案。

我们把这个问题推广到64个盘子的情形。开始时第1根柱子上有64个盘子。盘子64不可能从柱子1移动到柱子3，除非上面的63个盘子放在第2根柱子上。所以，首先把上面的63个盘子从柱子1移动到柱子2，然后再把盘子64从柱子1移动到柱子3。现在，前面的63个盘子都在柱子2上。为了把盘子63从柱子2移动到柱子3，首先要把前62个盘子从柱子2移动到柱子1，接着再把盘子63从柱子2移动到柱子3。按照相似的过程移动剩下的62个盘子。经过上面讨论，得到该递归算法的伪代码如下：假设第1根柱子上有n个盘子，并且$n \geq 1$。

1. 以柱子3作为中间柱子，把前n-1个盘子从柱子1移动到柱子2。
2. 把盘子n从柱子1移动到柱子3。
3. 以柱子1作为中间柱子，把前n-1个盘子从柱子2移动到柱子3。

以上的递归算法的C++函数实现如下所示：

```
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);
   }
}
```

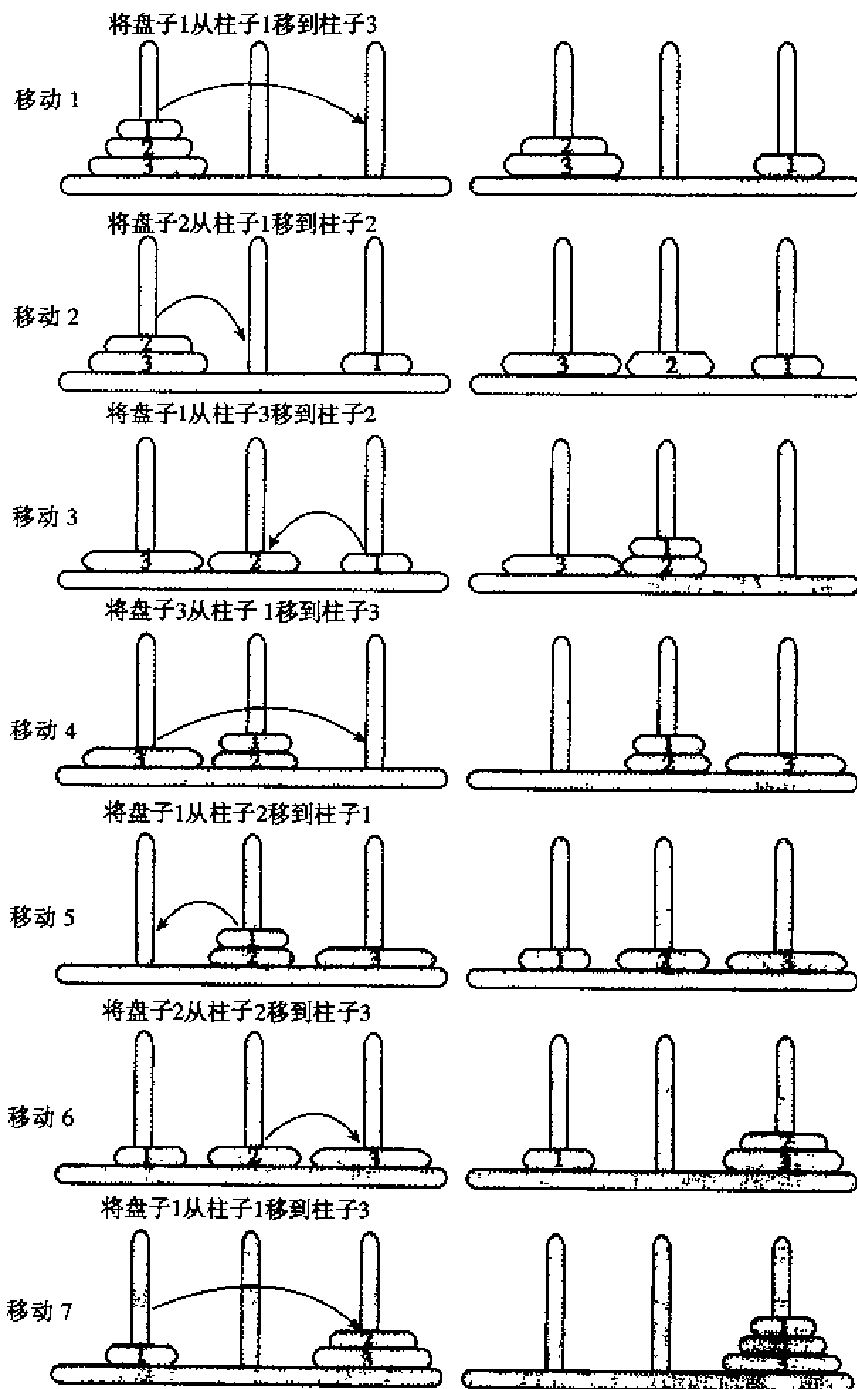


图 10.7 3个盘子的Hanoi塔问题的解决方案

10.2.1 Hanoi塔：分析

如果柱子1上有3个盘子，将全部3个盘子从柱子1移动到柱子3需要的移动次数是 $2^3 - 1 = 7$。类似地，如果柱子1上有64个盘子，将全部64个盘子从柱子1移动到柱子3需要的移动次数是 $2^{64} - 1$，并且：

```
2^10 = 1024 ≈ 1000 = 10^3,
2^64 = 2^4 * 2^60 ≈ 2^4 * 10^18 = 1.6 * 10^19
```

一年近似等于3.2*10^7秒，假设僧侣每秒移动一个盘子并且从不休息，则有：

```
1.6 * 10^19 = 5 * 3.2 * 10^18 = 5 * (3.2 *10^7) * 10^11
            = (3.2 *10^7) * (5 * 10^11)
```

把全部的64个盘子从柱子1移动到柱子3所需要的时间粗略估计是5 * 10^{11}年。据估计，宇宙现在的年龄大约是150亿=(1.5 * 10^{10})年。而5 * 10^{11} = 50 * 10^{10} ≈ 33 *(1.5 * 10^{10})，也就是说宇宙必须还得至少存在33倍于现有年龄的时间。

假设计算机每秒可以完成10亿=10^9次移动，那么计算机一年能完成的移动次数是：

```
(3.2 *10^7) * 10^9 = 3.2 * 10^16
```

所以，计算机完成2^{64}次移动需要的时间是：

```
2^64 ≈ 1.6 * 10^19 = 1.6 * 10^16 * 10^3 = (3.2 * 10^16) * 500
```

以每秒10亿次的移动速度计算，计算机大约要500年才能完成2^{64}次的移动。

10.3 递归与迭代的比较

在第5章中设计了一个计算Fibonacci数列的程序，该程序使用循环进行计算。换句话说，程序使用迭代控制结构反复执行一系列语句。也就是说，迭代控制结构使用while，for或者do…while等循环，反复执行一系列语句。在例10.2中设计了一个计算Fibonacci数列的递归函数，该程序通过递归调用反复执行一系列语句。此外，还可以在递归调用中使用选择控制结构来控制重复调用。

类似地，在第9章中使用迭代控制结构来计算表中的最大元素。在本章中，使用递归方法来计算表中的最大元素。此外，在本章的开始部分，我们编写了计算非负整数阶乘的函数。使用迭代控制结构也可以编写计算非负整数阶乘的算法。给出计算阶乘的递归算法只是为了说明递归的实现方法。

这样，我们有两种解决问题的方法：迭代和递归。一个很自然的问题是迭代和递归，哪种方法更好一些？很难简单地回答这个问题。除了考虑问题的性质以外，另一个选择问题解决方法的关键因素是效率。

例7.6（第7章）跟踪了问题的解决过程。当调用函数时，需要为函数的形参和局部变量分配存储空间；当函数终止时，释放存储空间。

本章在跟踪递归函数的执行过程时，也指出了每个递归调用都有自己的参数和局部变量。递归调用需要系统为它的形参和局部变量分配存储空间，当函数终止时释放存储空间。这样，无论是在存储空间上还是在时间上，递归函数都存在一定的执行开销。所以递归函数执行速度比迭代慢。在处理速度较慢的计算机上，特别是在存储空间有限的计算机上，可以很明显地发现递归函数的执行速度较慢。

今天的计算机不仅处理速度很快，而且存储器价格也很便宜。所以，执行起递归函数来也难以觉察执行速度有什么不同。考虑到当前计算机的能力，选择迭代还是递归，主要取决于问题的性质。当然，对于诸如导弹控制系统来说，效率绝对是关键。所以，效率还将是确定问题解决方案时需要考虑的因素。

一般来说，如果使用迭代来解决问题更清晰易懂，则应选择效率更高的迭代。另一方面，对于某些问题来说，使用递归解决起来更加清晰且易于构造，如Hanoi塔问题（实际上，使用迭代方法解决Hanoi塔问题很困难）。考虑到递归的高效性，如果问题的定义是递归的，则应考虑使用递归方法来解决。

10.4 程序范例：将二进制数转化成十进制数

第1章中曾经提到，计算机的语言称为机器语言，是0和1的序列。当在键盘上键入字符A时，在计算机中存储的是01000001。在ASCII码中字符A的编码值是65。实际上，与字符A的十进制表示65相对应的二进制表示是01000001。

在日常生活中，我们使用的数字系统是十进制系统，或者称为基于10的数字系统。而计算机使用的数字系统是二进制系统，或者称为基于2的数字系统。在本程序范例和下一个程序范例中，将分别讨论怎样把二进制数转化成十进制数和怎样把十进制数转化成二进制数。

把二进制数转化成十进制数 将二进制数转化成十进制数，首先要找出二进制数中每一位数字位的权值。在二进制数中，每一位数字位的权值由右向左依次赋值。最右边数字位的权值是0，其左侧相邻的数字位的权值是1，再左边的数字位的权值是2，依次类推。考虑二进制数1001101，每一位的权值如下：

```
权值    6   5   4   3   2   1   0
        1   0   0   1   1   0   1
```

根据二进制数中每一位数字的权值计算相应的十进制数。我们把每一位上的数字与对应的2的权值次幂相乘，然后再将所有的乘积相加。例如对于上面的二进制数来说，相应的十进制数是：

$$1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$
$$= 64 + 0 + 0 + 8 + 4 + 0 + 1$$
$$= 77$$

在编写将二进制数转化成十进制数的程序时，需要注意两点：(1) 必须知道二进制数每一位的权值；(2) 权值由右向左赋值。因为我们无法预知二进制数有多少位，所以必须由右向左逐位赋值。处理完一位后，将权值加1就得到紧靠它左边的数字位的权值。而且，还要将每一位数字从二进制数中提取出来，并且与2的权值次幂相乘。可以使用求余运算来取出每一位数字位。考虑下面以伪代码的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
}
```

上面的算法中假设decimal和weight的初始值为0。根据该算法，C++递归函数如下所示：

```
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);
   }
}
```

在上面函数中，decimal和weight是引用参数，相应的实参初始化为0。取出最右边的一位数字后，函数更新十进制数和二进制数的下一位权值。假设decimalNumber和bitWeight是int型变量，考虑下面语句：

```
decimalNumber = 0;
bitWeight = 0;
binToDec(1101,decimalNumber,bitWeight);
```

图10.8跟踪最后一条语句的执行过程。图中给出了每次函数调用前变量decimalNumber和bitWeight的值。

```
binToDec(1101, decimalNumber, bitWeight)
```

在图10.8中，向下的箭头代表依次调用的函数。因为binToDec函数的最后的一条语句是函数调用，所以这条语句执行后整个函数结束。当语句：

```
binToDec(1101 ,decimalNumber ,bitWeight);
```

执行完后，变量decimalNumber中的值是13。

图10.8 语句binToDec(1101, decimalNumber, bitWeight)的执行过程

下面的C++程序用来测试函数binToDec：

```
//Chapter 10: 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)
{
   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);
   }
}
```

程序运行结果 *在本程序运行中，用户输入的数据加有阴影。*

```
Enter number in binary: 11010110

Binary 11010110 = 214 decimal
```

10.5 程序范例：将十进制数转化成二进制数

在上一个程序范例中，讨论并设计了将二进制数转化成十进制数的程序。在本程序范例中，将讨论和设计使用递归将十进制的非负整数转化成相应的二进制数的程序。首先定义几个术语。

假设x是整数，将x除以2后的余数称为x的最右位（Rightmost bit）。这样，33的最右位是1，因为33 % 2等于1；28的最右位是0，因为28 % 2等于0。

首先，让我们借助于举例来说明将十进制数转化成相应的二进制数的算法。

假设要找到35的二进制表示形式。首先，35除以2，商是17，余数即35的最右位是1；接下来，17除以2，商是8，余数即17的最右位是1；然后，8除以2，商是4，余数即8的最右位是0。这个过程一直持续到商变成0为止。

在输出35最右位之前，必须先输出17的最右位；在输出17最右位之前，必须先输出8的最右位，依次类推。也就是说，35的二进制表示形式是17（35除以2的商）的二进制表示形式附加上35的最右位。

这样，如果把十进制数num转化成相应的二进制数，先要把num/2转化成相应的二进制数，然后在num/2的二进制表示形式之后附加num的最右位。

经过上面讨论可以得出下面的递归算法，其中binary(num)代表num的二进制表示形式：

1. binary(num) = num　　　　if num = 0
2. binary(num) = binary(num/2)　　　　后跟 num%2 if num > 0

下面的递归函数实现了上面的算法：

```
void decToBin(int num, int base)
{
    if(num > 0)
    {
        decToBin(num/base, base);
        cout<<num % base;
    }
}
```

图10.9跟踪了下面语句的执行过程：

```
decToBin(13,2);
```

这里，num =13，base =2。

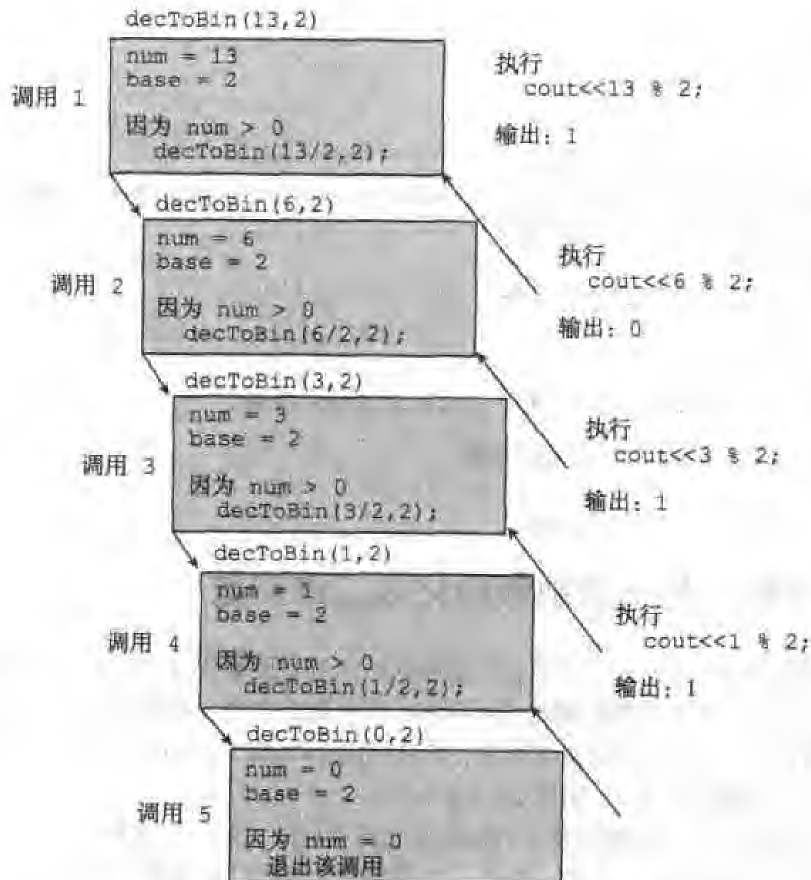


图10.9　语句decToBin(13, 2)的执行过程

因为第5次调用中的if语句失败，该调用不输出任何信息；第4次调用产生第1个输出，打印1；第3次调用产生第2个输出，打印1；第2次调用产生第3个输出，打印0；第1次调用产生第4个输出，打印1。这样，语句decToBin(13, 2)最终的输出是1101。

测试函数decToBin的C++代码如下所示：

```
//Chapter 10: 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;
   }
}
```

程序运行结果 *在本程序运行中，用户输入的数据加有阴影。*

```
Enter number in decimal : 57

Decimal 57 = 111001 binary
```

10.6 小结

1. 通过调用自身的新的拷贝解决问题的过程称为递归。
2. 递归定义通过在定义中含有自身拷贝的形式来定义问题。
3. 每个递归定义有一个或者多个基本实例。
4. 递归算法通过分解成自身简化来解决问题。
5. 每个递归算法有一个或者多个基本实例。
6. 基本实例的答案是直接获得的。

7. 调用自身的函数称为递归函数。
8. 递归算法通过递归函数实现。
9. 每个递归函数有一个或者多个基本实例。
10. 递归总是可以把问题简化成自身拷贝的形式。
11. 递归归约最终归结到基本实例。
12. 基本实例终止递归。
13. 逻辑上，递归函数有可以无限的自身拷贝，每次调用的递归函数都有自身函数体的拷贝。
14. 每个递归调用都有自身的参数和局部变量拷贝。
15. 在控制返回到前一次的调用之前，本次递归调用必须执行完毕，前一次的调用从递归调用的返回点处开始执行。
16. 最后一条语句执行递归调用的递归函数称为尾递归函数。

10.7 练习

1. 判断下面说法的正误。
   a. 每个递归定义必须有一个或者多个基本实例。
   b. 每个递归函数必须有一个或者多个基本实例。
   c. 递归归约可终止递归。
   d. 对于递归归约，问题的答案是直接获得的。
   e. 递归函数都具有返回值。
2. 考虑下面的递归函数：

```
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);
}
```

   回答下面的问题：
   a. 指出基本实例。
   b. 指出递归归约。
   c. 下面语句的输出是什么？

```
FuncRec(5, 'A');
```

3. 考虑下面的函数：

```
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. 考虑下面的函数:

```
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;`

10.8 编程练习

1. 编写递归函数vowels，该函数返回给定字符串中元音字母的个数，同时编写此递归函数的测试程序。
2. 编写返回int类型数组中元素之和的递归函数，同时编写此递归函数的测试程序。
3. 回文是指顺序和倒序相同的字符串。例如，"madam"，"madam I'm adam"都是回文。编写检查给定字符串是否是回文的递归函数。程序必须包含返回值，如果字符串是回文，返回true；否则，返回false。要求使用适当的参数，不能使用任何全局变量。
4. 编写通过递归函数反向输出字符串的程序，程序中必须包含反向输出字符串的递归函数。要求使用适当的参数，不能使用任何全局变量。
5. 编写递归函数reverseDigits。该函数以整数作为参数，并返回与给定整数各位数字顺序相反的整数。同时编写此递归函数的测试程序。
6. 编写递归函数power，函数有两个整型的参数x和y，其中x不等于0。函数返回x^y，应用以下的递归定义计算x^y。

   如果$y \geq 0$，

$$power(x,y)=\begin{cases}1 & \text{如果 } y=0\\ x & \text{如果 } y=1\\ x*power(x,y-1) & \text{如果 } y>1\end{cases}$$

   如果$y<0$，

$$power(x,y)=\frac{1}{power(x,-y)}$$

   同时编写此递归函数的测试程序。

7.（最大公因数）给定两个整数 x 和 y，通过下面的递归定义可以找出 x 和 y 的最大公因数。

$$gcd(x,y)=\begin{cases} x & \text{如果 } y=0 \\ gcd(y, x\%y) & \text{如果 } y\neq 0 \end{cases}$$

注意：上面定义中，% 是求余运算符。

编写递归函数gcd，函数以两个整数作为参数并且返回这两个整数的最大公因数，同时编写此递归函数的测试程序。

8. 在本章将十进制数转化成二进制数的程序范例中，我们学习了怎样把十进制数转化成相等的二进制数。计算机科学家感兴趣的还有另外两种数字系统，八进制和十六进制。实际上，在C++中可以指示计算机以八进制或者十六进制存储数据。

八进制数字系统中包含的数字有0，1，2，3，4，5，6，7。十六进制数字系统中包含的数字有0，1，2，3，4，5，6，7，8，9，A，B，C，D，E，F。十六进制中的A代表十进制中的10，B代表十进制中的11，依次类推。

把十进制的正整数转化成相等的八进制或者十六进制数的算法与前面讨论的把十进制的正整数转化成相应的二进制数的算法相同。不过要将十进制的数除以8或者16，假设a_b代表数a是b进制的数，例如，75_{10}说明75是十进制的数，83_{16}说明83是十六进制的数，那么：

$$753_{10} = 1361_8$$
$$753_{10} = 2F1_6$$

十进制数转化成二进制，八进制，十六进制数的方法可以扩展到任意进制。如果要把十进制的数n转化成相应的b进制数，其中b是界于2和36之间的数。在十进制数转化成二进制数的算法中使用n除以b代替n除以2。

例如二十进制中的数字包含0，1，2，3，4，5，6，7，8，9，A，B，C，D，E，F，G，H，I，J。使用递归函数实现把十进制数转化成给定b进制数的程序，其中b是界于2和36之间的数。程序应该提示用户输入十进制数和需要转化的进制。

使用下面的数据测试程序：

将9098转化为二十进制
将692转化为二进制
将753转化为十六进制

第11章 结　　构

本章要点：

- 了解结构
- 了解关于结构的各种操作
- 理解怎样使用结构操作数据
- 理解结构和函数之间的关系
- 理解结构中数组的用法
- 理解怎样创建结构数组

在第9章中介绍了怎样使用数组组织相同数据类型的数据，以及怎样处理数组中的数据和执行表等操作，例如查找和排序。

注意： 本章可以被跳过并对后面学习没有影响。

本章将介绍怎样将不同数据类型的数据组织在一起。C++提供了一种将不同类型的数据元素组织在一起的构造类型，称为结构（某些语言称为记录）。数组是包含相同数据类型的数据结构，而结构是包含不同数据类型的数据结构。本章讲述的结构类似于C语言中的结构。第12章将讨论另外一种被称为类的构造数据类型。

11.1 结构

假设要编写处理学生数据的程序。学生记录中包含了与学生相关的各种数据项，如学生姓名，学生ID，GPA，选择的课程，课程成绩。然而，这些数据项的数据类型是不同的。例如，学生姓名是字符串类型，而GPA是浮点类型的。由于这些数据是不同的类型，不能用数组把与学生相关的各种数据项组织起来。C++提供了一种可以将各种不同类型数据组织起来的数据类型，称为结构（struct）。将各种不同类型数据项组织在一起有很多优点。例如，只使用一个变量作为参数将所有数据项传递给函数。

结构　用名字访问的固定数目的相关数据元素的集合，这些数据元素可以是不同类型。

结构中的元素称为结构的成员。C++中定义结构的语法是：

```
struct structName
{
   dataType1  identifier1;
   dataType2  identifier2;
       .
       .
       .
   dataTypen  identifiern;
};
```

在C++中，struct是保留字。虽然括在大括号中，结构成员并不构成复合语句。要十分注意结构定义后面的分号，分号也是结构定义语法中的一部分。

语句：

```
struct employeeType
{
   string firstName;
   string lastName;
   string address1;
   string address2;
   double salary;
   string deptID;
};
```

上面的语句定义了有6个成员的结构employeeType，其中成员firstName，lastName，address1，address2，deptID都是string类型，成员salary是double类型。

像任何类型定义一样，结构只是类型定义，而不是变量定义。因此，结构只是定义了一种数据类型，而并不涉及存储分配。

定义了一种数据类型后，就可以定义这种数据类型的变量。在下面的代码中，先定义了一种结构类型studentType，然后定义了该类型的变量。

```
struct studentType
{
   string   firstName;
   string   lastName;
   char     courseGrade;
   int      testScore;
   int      programmingScore;
   double   GPA;
};

//variable declaration
studentType newStudent;
studentType student;
```

上面的语句定义了两个studentType结构变量newStudent和student。为这两个变量分配的存储空间可以存储firstName，lastName，courseGrade，testScore，programmingScore和GPA（参见图11.1）。

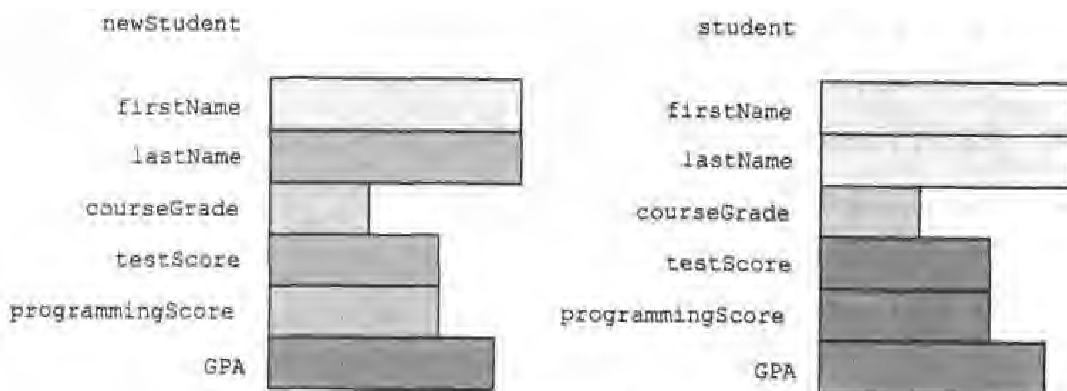


图11.1　结构newStudent和student

11.1.1　访问结构成员

在数组中，可以通过数组名称和下标来访问数组元素，数组名称和下标用方括号隔开。在结构中，可以通过用结构变量名称和成员名称来访问结构成员，这两个名称之间用圆点隔开。访问结构成员的语法是

```
structVariableName.memberName
```

变量 structVariableName.memberName 与其他变量相似。例如 newStudent.courseGrade 是 char 类型的变量，newStudent.firstName 是 string 类型的变量，等等。因此，对结构成员可以像对任何普通变量一样进行各种操作。例如，在赋值语句和输入输出语句中使用结构成员。

在 C++ 中，圆点（.）是运算符，称为成员访问运算符（Member Access Operator）。

假设要把结构 newStudent 中的成员 GPA 初始化为 0.0。使用下面的语句可以完成这项任务：

```
newStudent.GPA = 0.0;
```

类似地，语句：

```
newStudent.firstName = "John";
newStudent.lastName = "Brown";
```

将 "John" 和 "Brown" 分别存储在成员 firstName 和 lastName 中。

在前面的 3 条语句的执行后，结构 newStudent 中的值如图 11.2 所示。

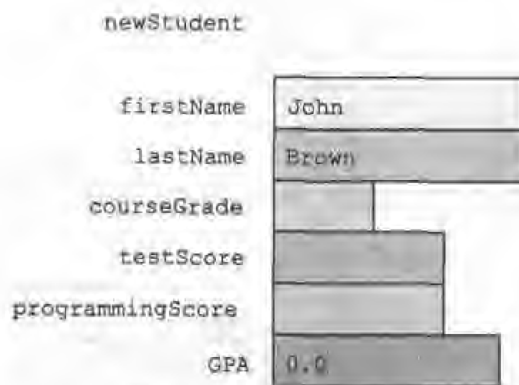


图 11.2　结构 newStudent

语句：

```
cin>>newStudent.firstName;
```

从标准输入设备中读入下一个字符串并且存储在 newStudent .firstName 中。

语句：

```
cin>>newStudent.testScore>>newStudent.programmingScore;
```

从键盘读两个整数并且分别存储在 newStudent.testScore 和 newStudent.programmingScore 中。

假设 score 是 int 型变量。语句：

```
score = newStuent.testScore + newStudent.programmingScore;
```

将 newStuent.testScore 和 newStudent.programmingScore 相加，并把结果存储在 score 中。

下面语句用来确定课程成绩，并将其存储在 newStudent.courseGrade 中：

```
newStudent.courseGrade:
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';
```

11.1.2 赋值

使用赋值语句可以将结构变量赋值给相同类型的结构变量。假设结构newStudent中的内容如图11.3所示。

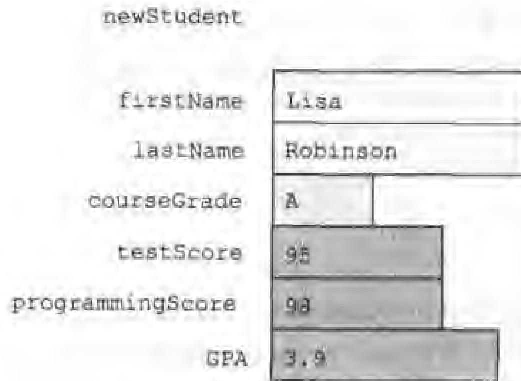


图 11.3 结构 newStudent

语句：

```
student = newStudent;
```

将结构 newStudent中的值拷贝到结构 student中。赋值语句执行完后，结构 student中的值如图11.4所示。

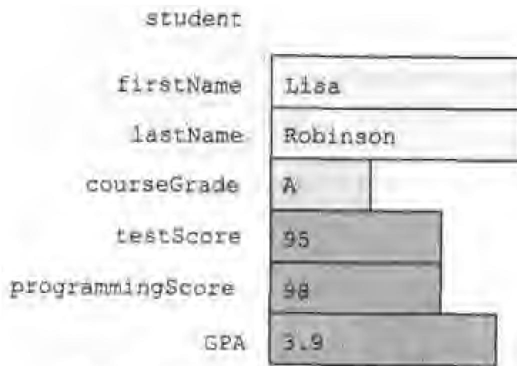


图 11.4 结构 student = newStudent

实际上，赋值语句：

```
student = newStudent;
```

等价于下面的语句：

```
student.firstName = newStudent.firstName;
student.lastName = newStudent.lastName;
student.courseGrade = newStudent.courseGrade;
student.testScore = newStudent.testScore;
student.programmingScore = newStudent.programmingScore;
student.GPA = newStudent.GPA;
```

11.1.3 比较（关系运算）

为了比较结构变量，必须逐一比较结构成员。跟数组一样，在结构类型数据上不能进行关系运算。例如前面提到的结构变量 newStudent 和 student，现在要确认 student 和 newStudent 是否代表同一个学生。如果它们有相同的firstName和lastName，则认为student和newStudent代表同一个学生。为了比较student和 newStudent 的值，必须逐一比较它们的成员：

```
if(student.firstName == newStudent.firstName &&
   student.lastName == newStudent.lastName)
.
.
.
```

尽管可以用赋值语句将一个结构类型变量的值拷贝到同类型的另一个结构变量中，但是不能在结构变量上进行任何关系运算。因此，下面的语句是非法的：

```
if(student == newStudent)     //illegal
.
.
.
```

11.1.4 输入/输出

不能在结构变量上执行输入/输出操作。对于结构类型变量来说，每次只能输入一个数据成员，同样，每次也只能输出一个数据成员。

我们已经知道了怎样向结构变量中输入数据，下面将了解怎样输出结构变量。语句：

```
cout<<newStudent.firstName<<" "<<newStudent.lastName<<" "
    <<newStudent.courseGrade<<" "<<newStudent.testScore<<" "
    <<newStudent.programmingScore<<" "<<newStudent.GPA<<endl;
```

输出结构变量 newStudent 的内容。

11.1.5 结构变量和函数

数组只能作为引用参数传递给函数，而且函数也不能返回数组类型的值。但是：

- 结构变量既可以作为引用参数又可以作为值参数传递给函数
- 函数可以返回结构类型的值

下面的函数读入并存储学生姓名，考试分数，程序设计分数和GPA。该函数同时计算出学生的课程成绩，并将其存储在成员 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。函数readIn将相应的信息存储在变量newStudent中。类似地，我们还可以编写输出结构变量内容的函数。例如，下面的函数用于将studentType类型结构变量的内容输出到屏幕上。

```
void printStudent(studentType student)
{
      cout<<student.firstName<<" "<<student.lastName<<" "
          <<student.courseGrade<<" "<<student.testScore<<" "
          <<student.programmingScore<<" "<<student.GPA<<endl;
}
```

11.1.6 数组和结构的比较

由上面的讨论可以知道，数组和结构既有联系又有区别。表11.1总结了这些异同点。

表11.1 数组和结构的比较

	整体运算	数组	结构
1	算术	No	No
2	赋值	No	Yes
3	输入/输出	No（除了字符串）	No
4	比较	No	No
5	参数传递	只能引用	引用或传值
6	函数返回值	No	Yes

11.1.7 含数组的结构

表（list）是由相同数据类型的一系列元素组成的数据结构。因此，表有两个相关的属性：表元素值和数组长度。由于表元素值和表长度都跟表相关，所以可以将这两个属性定义在一个结构中。

```
const arraySize = 1000;

struct listType
{
   int listElem[arraySize];      //array containing the list
   int listLength;               //length of the list
}
```

下面为在表中使用顺序查找算法来查找给定元素的函数。如果查找成功，函数返回查找项在表中的位置；否则，函数返回-1。

```
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;
}
```

在函数中，由于listLength是结构list的成员，我们可以通过list.listLength访问listLength。同样，可以通过list.listElem[loc]访问list中的元素。

注意，seqSearch函数的形参list是常量引用参数，说明list接收的是相应实参的地址。但是，list不能够修改实参，常量引用参数将在第12章中详细讨论。

同样，我们可以重新编写排序、二分查找及其他表处理函数。

11.1.8 结构数组

假设某个公司有50个全职员工。我们需要打印出他们的月收入和跟踪到目前为止的总收入，首先定义员工结构。

```
struct employeeType
{
    string   firstName;
    string   lastName;
    int      personID;
    string   deptID;
    double   yearlySalary;
    double   monthlySalary
    double   yearToDatePaid;
    double   monthlyBonus;
};
```

每个员工结构有以下成员：姓名，个人ID，部门ID，年薪，月薪，年收入和月奖金。因为有50个员工，并且每个员工的数据类型都是相同的，所以，可以使用有50个元素的数组来处理所有员工的数据。语句：

```
emplyeeType employees[50];
```

定义了有50个employeeType类型元素的employees数组。每个employees数组元素都是一个结构。例如，图11.5说明了employees[2]。

假设有以下变量定义：

```
int counter;
```

再假设每个员工的初始数据：姓名，个人ID，部门ID，年薪都在文件中提供。即这些员工的数据存储在文件employee.dat中。下面的C++代码从文件中读取数据到员工数组。假设yearToDatePaid的初始值是0，并根据工作成效确定月奖金。

```
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;
}
```

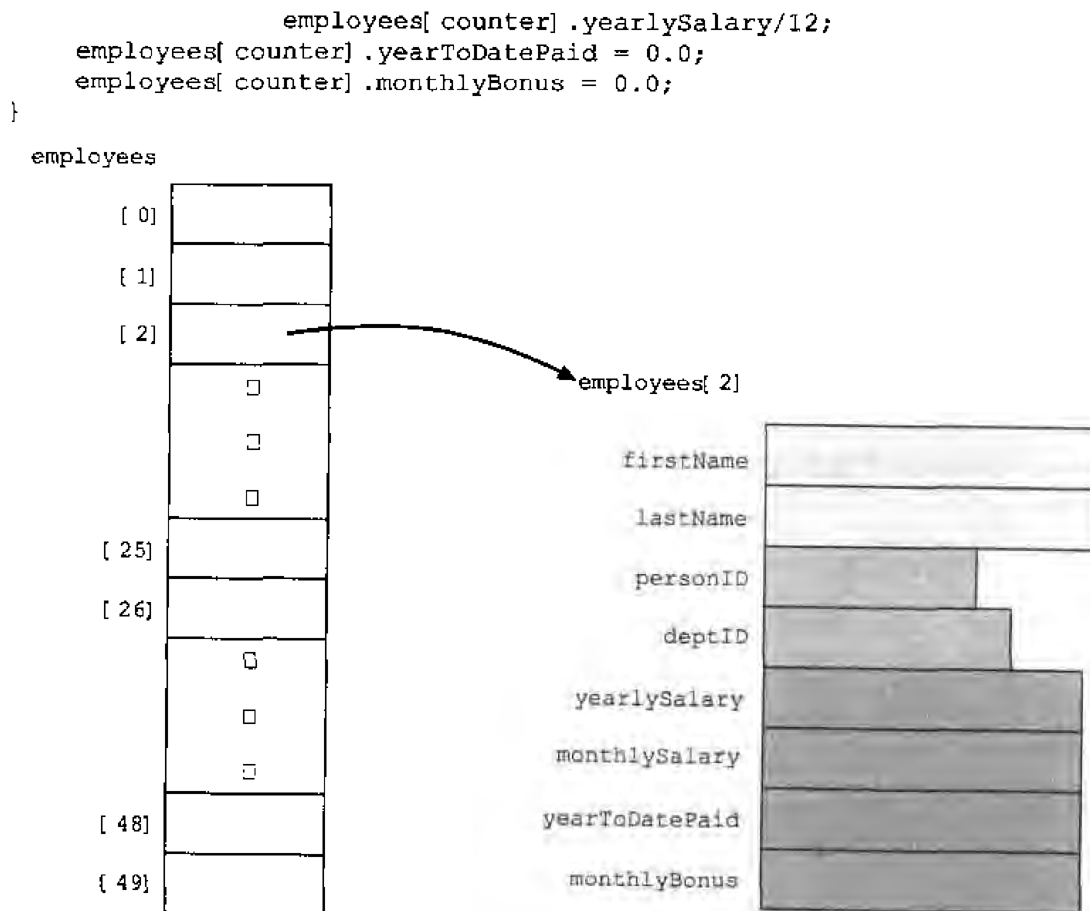


图 11.5 employees 结构数组

假设月奖金已经存储在每个员工结构中，我们要计算月收入并且更新yearToDatePaid总额，下面的循环计算并且打印每个员工的月收入。

```
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;
}
```

11.1.9 嵌套结构

前面介绍了怎样联合使用结构和数组这两个数据结构来组织信息。结构的成员可以是数组，数组元素的类型也可以是结构。在这一小节中，将介绍另一种数据组织形式——嵌套结构。

考虑下面的员工结构：

```
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;
};
```

正如所见到的，该结构中封装了许多信息。这个结构共有22个成员，结构中的某些成员相对于其他成员来说访问得更加频繁一些，而且某些成员之间的关系更紧密一些。此外，某些成员具有相同的基本结构，例如，hireday 和 quitday 都是日期类型数据。重新组织该结构如下所示：

```
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;
};
```

上面代码分别将员工名字、地址、联系方式分类定义成单独结构。另外，还定义了 dateType 结构。重新构造员工结构如下所示：

```
struct employeeType
{
        nameType        name;
        string          emplID;
        addressType     address;
        dateType        hiredate;
        dateType        quitdate;
        contactType     contact;
        string          deptID;
        double          salary;
};
```

这个员工结构的信息比前一个员工结构更易于管理。某些结构可以应用于另一个结构的构造中。假设要定义顾客结构，每个顾客有姓名、地址、联系方式等属性。可以使用 nameType，addressType 和 contactType 结构来定义顾客结构成员。

下面，定义 employeeType 类型变量，并讨论怎样访问变量成员：

```
     //variable declaration
employeeType newEmployee;
employeeType employees[100];          //declare 100 employees' recordes
```

语句：

```
newEmployee.salary = 45678.00;
```

将 newEmployee 的工资赋值为 45 678.00。语句：

```
newEmployee.name.first = "Mary";
newEmployee.name.middle = "Beth";
newEmployee.name.last = "Simmons";
```

上面代码分别将变量 newEmployee 的成员 first，middle，last 赋值为 "Mary"，"Beth"，"Simmons"。注意 newEmployee 有一个名为 name 的成员。可以通过 newEmployee.name 来访问成员 name。同样，注意 newEmployee.name 是有 3 个成员的结构。这里，我们使用成员访问规则来访问 newEmployee.name 结构中的成员 first。因此，newEmployee.name.first 是存储人的名字（first name）的成员。

语句：

```
cin >> newEmployee.name.first;
```

读取一个字符串并将其存储在 newEmployee.name.first 中。语句：

```
newEmployee.salary = newEmployee.salary * 1.05;
```

用于更新 newEmployee 的工资。

for 循环：

```
for(j = 0; j < 100; j++)
```

```
cin>>employees[ j] .name.first>>employees[ j] .name.middle
   >>employees[ j] .name.last;
```

读取 100 个员工名字并将之存储到数组 employees 中。因为 employees 是结构数组，所以可以使用数组下标访问该数组中的元素。例如，employees[50]是 employees 数组中的第 51 个元素（数组元素的下标从 0 开始）。因为 employees[50]是结构，所以应该使用成员访问规则来访问特定的结构成员。

11.2 程序范例：销售数据分析

某公司有 6 个销售人员。这些销售人员每个月都出去销售该公司的产品。在每个月的月末，将每个销售人员的销售额，销售人员 ID 和月份记录在文件中。在年终，公司经理希望看到以下形式的年度销售报告。

```
----------- 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
```

在报告中，QT1 代表第 1 季度（1 月到 3 月）；QT2 代表第 2 季度（4 月到 6 月）；QT3 代表第 3 季度（7 月到 9 月）；QT4 代表第 4 季度（10 月到 12 月）。

销售人员 ID 存储在一个文件中，销售数据则存储在另一个文件中。销售数据的格式如下所示：

```
salesPersonID  month  saleAmount
.
.
.
```

销售数据并没有按照销售人员 ID 排序。也就是说，没有特定的顺序。

现在编写程序，该程序按照上面指定的格式输出数据。

输入 第 1 个文件包含销售人员 ID；第 2 个文件包含销售数据。

输出 包含上面指定格式的年度销售报告的文件。

问题分析和算法设计

根据上面的问题需求可知，对于每个销售人员来说，记录的主要数据包括：销售人员 ID，季度销售额，年度销售额。由于这些数据的类型不同，应该使用结构将这些数据组织起来。结构定义如下所示：

```
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个销售人员，应该定义一个有6个元素的结构数组。每个数组元素都是salesPersonRec类型的结构。数组定义如下所示：

```
salesPersonRec  salesPersonList[ noOfSalesPersons] ;
```

这里，noOfSalesPersons的值是6。

因为程序要计算出该公司每个季度的销售额，所以应该定义一个有4个元素的数组来存储这些销售额数据。注意，这些数据要用做决定销售额最大的季度，所以应该定义下面的数组：

```
double totalSaleByQuarter[ 4] ;
```

图11.6说明了数组salesPersonList的结构。

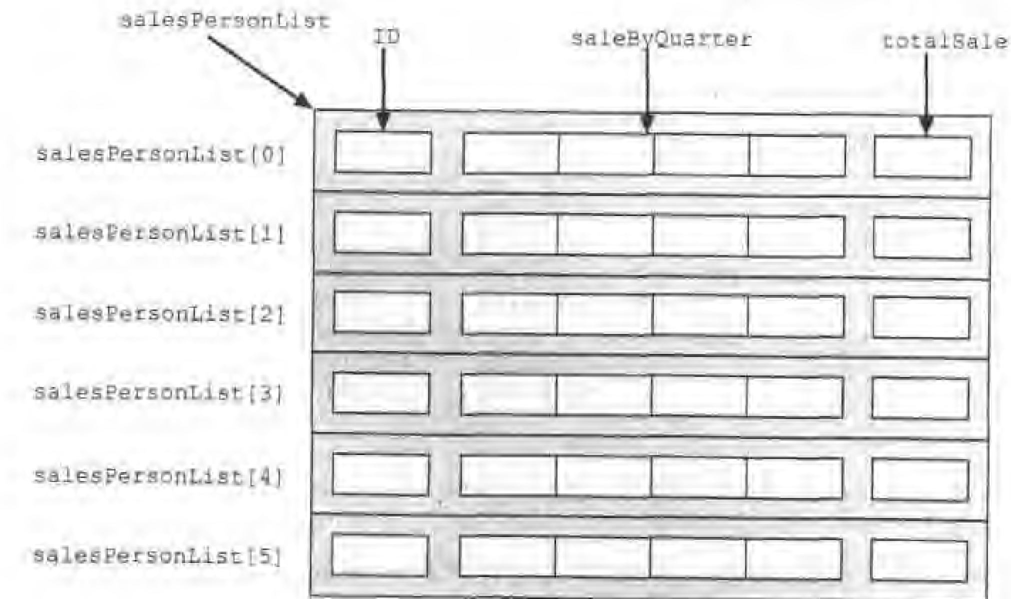


图11.6 数组salesPersonList

首先，程序要将销售人员ID读到数组salesPersonList中，并将每个销售人员的季度销售额和年总销售额初始化为0。完成这一步后，数组salesPersonList中的内容如图11.7所示。

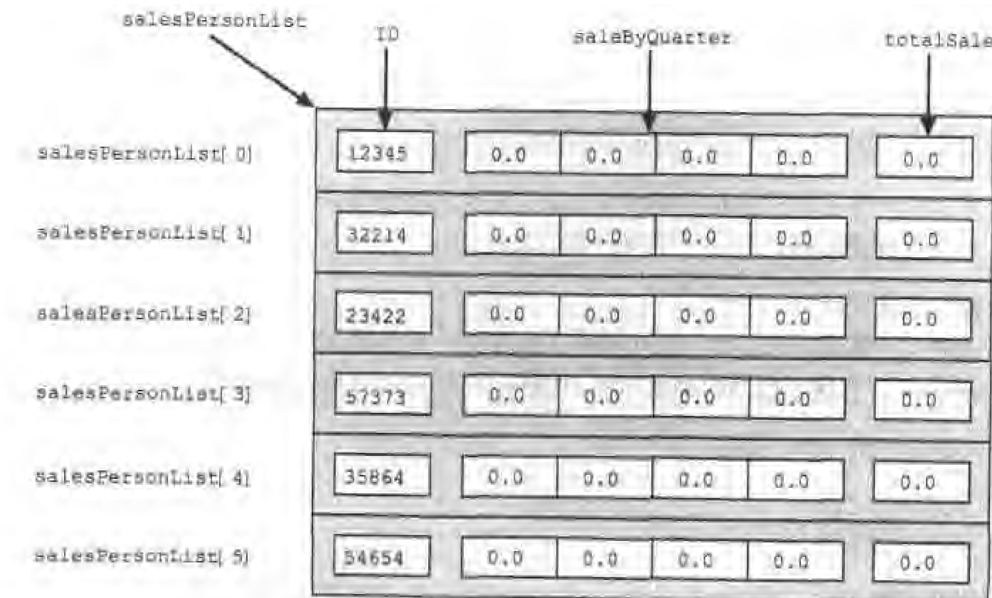


图11.7 初始化后的数组salesPersonList

接下来要处理销售额数据，对于包含销售数据文件中的每条记录：

1. 读取销售人员ID，月份和月销售额。
2. 在数组salesPersonList中查找销售人员，以确定其在数组中的存储位置。
3. 判断该月份所属的季度。
4. 累加月销售额以更新季度销售额，一旦包含销售数据的文件处理完毕：
   a. 计算销售人员年度销售总额
   b. 计算季度销售总额
   c. 打印报告

经过讨论，得出算法如下所示：

1. 初始化数组salesPersonList。
2. 处理销售数据。
3. 计算销售人员年度销售总额。
4. 计算季度销售总额。
5. 打印报告。
6. 计算并打印最大个人销售额及相应销售人员ID。
7. 计算并打印最大季度销售额及相应季度。

为了减少主程序的复杂性，应该为上面的每一步编写一个单独的函数。

函数initialize 该函数从输入文件中读取销售人员ID，并将之存储到数组salesPersonList中。将每个销售人员的季度销售总额和年度销售总额初始化为0。函数定义如下所示：

```
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 该函数从输入文件中读取销售数据，并在数组salesPersonList中存储相应的信息。该函数的算法是：

1. 读取销售人员ID，月份和月销售额。
2. 查找数组salesPersonList，确定销售人员在数组中的存储位置（由于销售人员的ID是无序的，应该使用顺序查找方法）。
3. 确定月份所属的季度。
4. 累加月销售额以更新季度销售额。

假设读取的记录是：

```
57373 2 350
```

销售人员该条记录的含义是：ID是57373，2月份，销售额是350。假设数组salesPersonList的内容如图11.8所示。

图 11.8 数组 salesPersonList

因为ID57373对应的数组元素是salesPersonList[3]，而且2月份属于第1季度，所以应该将350加到354.80上得到704.80。处理完这一条记录后，数组salesPersonList中的内容如图11.9所示。

图 11.9 处理完记录 57373 2 350 后的数组 salesPersonList

函数getData的定义如下所示：

```
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 该函数用来计算每季度的总销售额。为了计算某个季度的总销售额，要将该季度每个销售人员的销售额加起来。该函数应该能够访问数组salesPersonList和totalSaleByQuarter。而且，该函数还要知道每个数组的行数。因此，该函数有3个参数。函数定义如下所示：

```
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 totalSalesByPerson 该函数用来计算每个销售人员的年销售额。为了计算某个销售人员的年销售额，要将该销售人员4个季度销售额加起来。该函数应该能够访问数组 salesPersonList。而且，该函数还要知道数组的长度。因此，该函数有两个参数。函数定义如下所示：

```
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 该函数用来按照指定格式打印年度销售报告。函数算法的伪代码如下所示：

a. 打印标题，即前3行的输出。
b. 打印每个销售人员的数据。
c. 打印表格的最后1行。

注意，下面的两个函数产生最后两行的输出。

函数 *printReport* 应该能够访问数组 *salesPersonList* 和 *totalSaleByQuarter*。而且，要将输出存储到文件中，该函数还应该能够访问与输出文件相关的 ofstream 类型变量。因此，该函数有4个参数，分别是数组 salesPersonList，数组 totalSaleByQuarter，数组的长度，ofstream 类型变量。函数定义如下所示：

```
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<<"_______________________________________________"
           <<"_______________"<<endl;

    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 该函数用来输出产生最大销售额的销售人员名字。因此，该函数要查看每个销售人员的总销售额并找出最大的销售额。因为每个销售人员的总销售额存储在数组 salesPersonList 中，所以函数应该能够访问数组 salesPersonList。因为要将输出结果存储到文件中，函数应该能够访问与输出文件相关的 ofstream 类型变量。因此，该函数有 3 个参数，分别是数组 salesPersonList，数组长度和输出文件。

查找最大销售额的算法与在数组中查找最大元素的算法（第9章中已经讨论）类似，函数定义如下所示：

```
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 该函数用来输出产生最大销售额的季度。因此，该函数查看每季度的销售额并找出最大的季度销售额。由于每季度的销售额存储在数组totalSaleByQuarter中，所以函数应该能够访问数组totalSaleByQuarter。因为输出的结果要存储在文件中，所以函数应该能够访问与输出文件相关的ofstream类型变量。因此，该函数有两个参数，分别是数组totalSaleByQuarter和输出文件。

查找最大销售额的算法与在数组中查找最大元素的算法（第9章中已经讨论）类似，函数定义如下所示：

```
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;
}
```

为了增强程序的灵活性，应该在程序执行过程中提示用户指定输入和输出文件。下面为函数main的算法。

主要算法

1. 定义变量。
2. 提示用户输入包含销售人员 ID 的文件名。

3. 读取输入的文件名。
4. 打开输入文件。
5. 如果输入文件不存在，退出程序。
6. 调用函数 initialize 初始化数组 salesPersonList。
7. 关闭包含销售人员 ID 的文件。
8. 提示用户输入包含销售数据的文件名。
9. 读取输入的文件名。
10. 打开输入文件。
11. 如果输入文件不存在，退出程序。
12. 提示用户输入输出文件的名称。
13. 读取输出文件的名称。
14. 打开输出文件。
15. 通过控制符 fixed 和 showpoint，将输出的小数的格式设置为固定小数点格式，显示小数末尾的0。由于要以两个小数位输出浮点数，所以设置精度为两位小数。
16. 调用函数 getData，处理销售数据。
17. 调用函数 saleByQuarter 计算季度销售额。
18. 调用函数 totalSaleByPerson 计算每个销售人员的年度总销售额。
19. 调用函数 printReport 以表格的格式打印报告。
20. 调用函数 maxSaleByPerson 打印产生最大年度个人销售额的销售人员名字。
21. 调用函数 maxSaleByQuarter 打印产生最大季度销售额的季度。
22. 关闭文件。

完整的程序代码清单

```
//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 : ";           //Step 2
   cin>>inputfile;                                      //Step 3
   cout<<endl;

   infile.open(inputfile);                              //Step 4

   if(!infile)                                          //Step 5
   {
      cout<<"Cannot open input file."<<endl;
      return 1;
   }

   initialize(infile, salesPersonList, noOfSalesPerson); //Step 6

   infile.close();   //reclaim input file stream variable; Step 7

   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;
   }

   cout<<"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;
}
```

程序运行结果

输入文件：销售人员 ID

```
12345
32214
23422
57373
35864
54654
```

输入文件：销售数据

```
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
```

11.3 小结

1. 结构是固定数目元素的集合。
2. 结构的元素可以是不同类型。
3. 定义结构的语法是:

```
struct structName
{
    dataType identifier;
    dataType identifier;
         .
         .
};
```

4. 在C++中，struct是保留字。
5. 在C++中，定义结构时并不涉及存储分配，仅当定义结构变量时才涉及存储分配。
6. 结构中的元素称为结构成员。
7. 可以通过名字访问结构成员。
8. 通过点(.)运算符来访问结构成员。如果employeeType是一个结构，employee是employeeType类型的变量，name是employee的成员，表达式employee.name访问成员name。employee.name是变量，可以像其他变量一样进行操作。

9. 在C++中，点（.）运算符称为成员访问运算符。
10. 可以在结构上使用的运算符是赋值运算符和成员访问运算符。
11. 在结构上不允许算术运算和关系运算。
12. 结构可以作为函数的值参数或者引用参数。
13. 函数能够返回结构类型的值。
14. 结构可以嵌套。

11.4 练习

1. 判断下面说法的正误。
   a. 结构是包含相同数据类型的元素的数据结构。
   b. 函数不能返回结构类型的值。
   c. 结构的成员不能是另一个结构。
   d. 在结构上只允许赋值运算和成员访问运算。
   e. 结构的成员可以是数组。
   f. 在C++中，结构允许某些整体操作（即，将整个结构作为一个整体来操作）。
   g. 因为结构中的成员有限，所以结构允许关系运算。
2. 考虑下面的语句：

```
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;
```

判断下面语句的正误。如果语句错误，请说明原因。

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. 考虑下面的语句（nameType 的定义在练习 2 中）：

```
struct employeeType
{
   nameType name;
   int   performanceRating;
   int   pID;
   string dept;
   double salary;
};
employeeType employees[ 100] ;
employeeType newEmployee;
```

判断下面语句的正误。如果语句错误，说明原因。

a. `newEmployee.name = "John  Smith";`

b. `cout<<newEmployee.name;`

c. `employees[ 35]  = newEmployee;`

d.
```
if(employees[ 45] .pID   == 555334444)
     employee[ 45] .performanceRating = 1;
```

e. `employees.salary = 0;`

4. 假设有练习 2 和练习 3 中的定义，编写完成以下功能的 C++ 语句：

a. 在 newEmployee 中存储以下信息：

```
name: Mickey  Doe
pID: 111111111
performanceRating: 2
dept: ACCT
salary: 34567.78
```

b. 在 employees 数组中，将每个 performanceRating 初始化为 0。

c. 把 employees 数组中的第 20 个元素的值拷贝到 newEmployee 中。

d. 通过将 5 735.87 加到原来的值上，更新 employees 数组中的第 15 个员工的工资。

11.5　编程练习

1. 编写读取学生名字和考试分数的程序。该程序输出每个学生的名字、考试分数和相应的考试成绩。程序能够查找并打印最高考试分数的学生的名字和考试分数。

学生的数据存储在 studentType 类型的结构中。此结构有 3 个成员，string 类型的学生名字 studentName，int 类型的考试分数 testScore（取值范围 0~100）和 char 类型的课程成绩 grade。假设某个班有 20 个学生，使用有 20 个 studentType 类型元素的数组。

程序应该至少包含下面的函数：

a. 把学生的数据读到数组中的函数。

b. 将学生分数转换成相应成绩的函数。

c. 查找最高分数的函数。

d. 打印具有最高分数学生名字的函数。

程序输出学生的名字必须是左对齐的。此外，函数main除了定义变量、打开输入和输出文件外，应该只包含函数调用的语句。

2. 定义menuItemType结构。该结构有两个成员，string类型的menuItem和double类型的menuPrice。
3. 编写一个程序。该程序可以辅助当地餐馆自动管理早餐付账系统。程序的功能如下所示：

   a. 向顾客显示餐馆提供的早餐品种。

   b. 允许顾客在菜单中选择多种早餐。

   c. 计算和打印账单。

   假设餐馆提供以下的早餐品种（每种早餐的价格列在早餐种类的右边）：

```
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
```

   程序使用menuItemType类型的menuList结构数组。menuItemType的定义在编程练习2中，该程序必须包含以下的函数：

   - 函数getData：把数据存储到数组menuList中。
   - 函数showMenu：显示餐馆提供的早餐品种，并提示顾客怎样选择早餐。
   - 函数printCheck：计算和打印账单（注意账单应该包含5%的税）

   一个范例输出如下所示：

```
Welcome to Johnny's Restaurant
Bacon  and Egg             $2.45
Muffin                     $0.99
Coffee                     $0.50
Tax                        $0.20
Amount Due                 $4.14
```

   输出格式要求如下：小数要有两位有效数字，早餐名称应该是左对齐的，并且顾客每种早餐只能购买一份。

4. 修改编程练习3中的程序，使得顾客可以购买多份同一品种的早餐。一个范例输出如下所示：

```
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。函数main只包含变量定义和函数调用。程序读取一段文字，输出该段文字中的所有字符和每个字符的出现次数，见下面函数printResult的说明（程序中不可以使用全局变量，通过参数在函数之间传递信息。使用结构来存储信息）。程序中应该至少包含以下的函数：

   - 函数openFile：该函数用于打开输入和输出文件。函数以文件流作为参数（当然是以引用的形式传递）。如果文件不存在，该函数给出相应的消息并退出程序。该程序提示用户输入文件名和输出文件名。

- 函数 count：在使用 openFile 函数打开文件后，该函数统计每个大写字母 A~Z 和小写字符 a~z 的出现次数。统计信息存储在结构数组中，数组名和文件流变量应该作为参数传递给该函数。
- 函数 printResult：该函数用来打印大写字母和小写字母的数目以及每个大写字母 A~Z 和小写字符 a~z 出现的百分比数。百分比数的格式如 25%。统计信息来源于结构数组，数组必须作为参数传递。

第12章　类和数据抽象

本章要点：

- 了解类
- 了解类的公有成员、私有成员、受保护成员
- 理解怎样实现类
- 了解类的构造函数和析构函数
- 了解抽象数据类型（ADT）
- 理解怎样使用类来实现抽象数据类型
- 了解信息隐藏
- 理解怎样在C++中实现信息隐藏

在第11章中，介绍了怎样使用结构将不同类型的数据元素组合在一起。第11章中结构的定义与C语言中结构的定义十分类似。但是，C++中结构成员既可以是数据元素也可以是函数。C++提供另外一种构造数据类型，称为类（Class）。本章首先介绍类和类的实现方法，然后讨论结构和类的相似点与不同点。

注意： 本章不需要第11章作为预备知识。实际上，结构和类有类似的功能，结构和类的比较在本章的后面部分讨论。

12.1　类

第1章介绍了一种称为面向对象程序设计（OOD）的解决问题方法。在面向对象程序设计方法中，第一步是定义对象。对象是包含数据和数据操作的独立单元。在C++中，在独立单元中包含数据和数据操作的机制，称为类。我们已经掌握了怎样在计算机中存储和操作数据，以及怎样构造自己的函数，下面就可以了解怎样构造对象了。本章和后面的几章使用面向对象的方法设计和编写程序。这一章，首先讨论类的定义和类在程序中的应用。

类是固定数目元素的组合，组成类的元素称为类的成员。

定义类的语法是：

```
class classIdentifier
{
    classMembersList
};
```

classMembersList中包含各种变量定义和（或者）函数定义。即类的成员可以是变量（存储数据）或者函数。

- 如果类的成员是变量，成员变量的定义和其他的变量的定义很相似。但是，在类定义中，不能在变量定义的同时对变量进行初始化。
- 如果类的成员是函数，一般只用函数原型定义成员函数。

- 如果类的成员是函数，成员函数可以访问类中的任何成员，包括数据成员和成员函数。当编写成员函数定义时，可以直接访问类中的任何数据成员，惟一的约束条件是在使用标识符之前首先要定义标识符。

在 C++ 中，class 是保留字。类定义只是定义了一种数据类型，不涉及存储分配，只是完成了类的声明。注意右花括号后的分号也是语法的一部分，缺少分号将导致语法错误。

类的成员有 3 种：私有成员、公有成员和受保护成员。本章将主要讨论前两种，私有成员和公有成员。下面是私有成员和公有成员的一些性质：

- 在默认情况下，类的所有成员都是私有成员。
- 如果类的成员是私有的，在类的外面不能够访问该类的私有成员（例 12.1 说明了这种情况）。
- 在类的外面可以访问类的公有成员（例 12.1 说明了这种情况）。
- 使用标号 public 和冒号来指明类的公有成员。

在 C++ 中，public，private，protected 都是保留字，称为成员存取说明符（Member Access Specifier）。

假设要在程序中定义描述一天时间的类，称这个类为 clockType。此外，在计算机存储中，用 3 个 int 型变量表示时间。这 3 个变量分别表示小时、分钟和秒。可以在时间上执行下面的操作：

1. 设置时间。
2. 返回时间。
3. 打印时间。
4. 时间增加一秒种。
5. 时间增加一分钟。
6. 时间增加一小时。
7. 比较两个时间是否相等。

根据以上的讨论确定 clockType 类有 10 个成员，3 个数据成员和 7 个成员函数（如图 12.1 所示）。

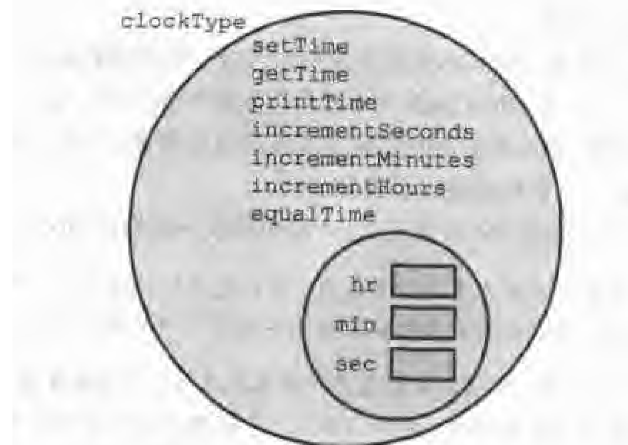


图 12.1　类 clockType 和它的成员

在 clockType 类中有些成员是私有成员，有些成员是公有成员。确定成员是公有成员还是私有成员的主要依据是成员的属性。一般的规律是：将所有需要在类的外面访问的成员声明为公有成员，将不需要被用户直接访问的成员声明为私有成员。例如，用户应该能够设置时间和打印时间。所以，将该设置时间和打印时间的成员函数声明为公有成员函数。

类似地，也应该将增加时间和比较时间是否相等的成员函数声明为公有成员函数。此外，为了防止直接访问数据成员hr，min，sec，应该将这些成员变量声明为私有成员变量。需要注意的是，如果用户能够直接访问数据成员，像setTime这样的成员函数就不需要了。这一章的后半部分（从信息隐藏开始），解释为什么有些成员是公有成员，而另一些成员是私有成员。

下面的语句定义了clockType类：

```
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;
};
```

在定义中：

- clockType类有7个成员函数：setTime，getTime，printTime，incrementSeconds，incrementMinutes，incrementHours，equalTime；有3个数据成员：hr，min，sec。
- clockType类中的3个数据成员是私有成员，在类的外面不能够访问这些私有成员（例12.1说明了这个情况）。
- clockType类的7个成员函数：setTime，getTime，printTime，incrementSeconds，incrementMinutes，incrementHours，equalTime，可以直接访问数据成员（hr，min，sec）。换句话说，数据成员不需要作为参数传递给成员函数。
- 成员函数equalTime的参数otherClock是常量引用参数。调用函数equalTime时，参数otherClock接收实际参数的地址，但是otherClock并不能修改实际参数的值。也可以将otherClock声明为值参数。但是如果这样做，otherClock需要拷贝实参的值给形参，这将导致性能降低（本章的后面部分“形参和类对象”一节对此做出了解释）。
- 成员函数printTime和equalTime后面的const说明函数不能修改clockType类型的成员变量。

注意：可以以任意的顺序声明私有成员和公有成员。可以先声明私有成员，然后再声明公有成员。这一章的后面部分中的“类的公有成员和私有成员的顺序”一节讨论了这个问题。

注意：在clockType类的定义中，所有的数据成员都是私有成员，所有的成员函数都是公有的成员函数。但是，成员函数也可以是私有的。例如，如果一个函数只是需要被该类中另外的函数调用，用户不需要访问这个函数，这样的函数可以定义为私有的。类似地，类的数据成员也可以是公有的。

注意，我们没有编写clockType类的成员函数定义。这样，你可以知道怎样在类定义中简洁地声明成员函数。

函数setTime的作用是将3个数据成员设置为指定的值。这些指定的值作为参数传递给函数setTime。函数printTime的作用是打印时间，即hr，min，sec的值。函数incrementSeconds的作用是使时间增加一秒钟。函数incrementMinutes的作用是使时间增加一分钟。函数incrementHours的作用是使时间增加一小时。函数equalTime的作用是判断两个时间是否相等。

注意，尽管做比较需要有两个时间，但是函数 equalTime 只有一个参数。本章后面的“成员函数实现”一节中，举例说明了这种情况。

变量（对象）声明

定义了类以后可以声明（定义）这个类的变量。在C++的术语中，类的变量称为类对象（Class Object）或者类实例（Class Instance）。为了熟悉术语，从现在开始，类的变量称为类对象，或者更简单一点，称为对象。

声明对象的语法和声明任何变量的语法相同，下面的语句声明了 clockType 类的两个对象。

```
clockType   myClock;
clockType   yourClock;
```

每个对象有10个成员：7个成员函数和3个数据成员。每个对象独立地为hr，min，sec分配存储空间。

实际上，每个对象只对数据成员分配存储空间。C++编译器为每个成员函数只产生一个物理拷贝。属于同一个类的所有对象都使用相同的成员函数拷贝。所以，在画类图时，应该画出该类的所有成员。而在画对象图时，只画出该对象中的成员变量。图 12.2 说明了赋值后的对象 myClock 和 yourClock。

图 12.2　myClock 和 yourClock 对象

访问类的成员

在定义了类对象以后，就可以访问类的公有成员。访问类成员的语法是：

```
classObjectName.memberName
```

在 C++ 中，点（.）称为成员访问运算符。例 12.1 说明了怎样访问类的成员。

例 12.1　*考虑下面的语句：*

```
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，在语法上是正确的。

第 1 条语句，“myClock.setTime(5, 2, 30);”，执行成员函数 setTime。5，2，30 作为参数传递给函数 setTime。函数用这些值分别将 myClock 的数据成员 hr，min，sec 的值设置为 5，2，30。类似地，第 2 条语句执行函数 printTime 并输出 myClock 的 3 个数据成员的值。第 3 条语句，使用变量 x，y，z 来设置 yourClock 的 3 个数据成员的值。

第 4 条语句，执行函数 equalTime 并且比较 myClock 和 yourClock 中的 3 个数据成员的值。因为 equalTime 是 myClock 对象的成员函数，所以它可以直接访问 myClock 的 3 个数据成员。在这种情况下，只需要一个 yourClock 对象就可以进行比较。这样就解释了为什么函数 equalTime 只有一个参数。

对象只能够访问类的公有成员。因此，下面的语句是不合法的。因为hr和min是类clockType的私有成员，所以myClock和yourClock对象不能够直接访问这些私有成员：

```
myClock.hr = 10;                              //illegal
myClock.min = yourClock.min;                  //illegal
```

12.1.1 类的内建运算符

类不支持C++中大多数的内建运算符。对象不支持算术运算（除非重载运算符，参见第15章）。例如，不能使用加号运算符对clockType类型的两个对象相加。同样，不能使用关系运算符比较两个对象是否相等（除非重载运算符，参见第15章）。

在类对象上的两个合法的内建运算符是：成员访问运算符（.）和赋值运算符（=）。我们已经知道了怎样使用对象的名字、成员访问运算符和成员的名字来访问类的成员。下面，通过举例说明怎样使用赋值语句的过程。

赋值运算符和类

假设myClock和yourClock是前面定义的clockType类的对象，并假设myClock和yourClock对象的内容如图12.3所示。

图12.3 myClock和yourClock对象

语句：

```
myClock = yourClock;                // Line A
```

将yourClock对象的内容拷贝到myClock对象中。即：

1. 把yourClock.hr的值拷贝到myClock.hr中
2. 把yourClock.min的值拷贝到myClock.min中
3. 把yourClock.sec的值拷贝到myClock.sec中

换句话说，将yourClock对象的3个数据成员中的值拷贝到myClock对象的相应的数据成员中，赋值语句对成员逐个进行拷贝。在执行完Line A的语句后，myClock和yourClock对象的内容如图12.4所示。

图12.4 执行完语句myClock =yourClock后的myClock和yourClock对象

12.1.2 类的作用域

类对象可以是动态的（每当程序执行到对象声明时创建类对象，当程序退出包含该对象的块结构时释放类对象）或者是静态的（只创建一次，当程序执行到对象声明时创建类对象，当程序终止时释放类

对象)。可以定义对象数组，对象和其他的变量一样具有相同的作用域。类的成员和结构的成员一样具有相同的作用域，类成员是类的局部变量。通过对象名字和成员访问运算符(.)可以访问类的成员。

12.1.3 函数和类

以下的规则描述了函数和类的关系：

- 对象可以作为参数向函数传递，也可以作为函数的返回值。
- 对象既可以作为值参数传递给函数，又可以作为引用参数传递给函数。
- 如果类对象作为值参数传递给函数，则实参的数据成员的内容拷贝到相应的形参的数据成员中。

引用参数和类对象

当变量以值参数传递给函数时，实参的值拷贝到形参中。需要为形参分配存储空间并完成实参的拷贝，对象可以作为值参数传递给函数。

假设类有很多数据成员，并需要大量的空间来存储数据。如果将该类的对象作为值参数传递给形参，程序需要给相应的形参拷贝大量数据，编译器需要为这些形参分配大量的存储空间。这样，需要花费大量的存储空间和计算机时间来完成从实参向形参的数据拷贝。

另一方面，如果将该类的对象作为引用参数传递参数，形参只接收实参的地址。所以，采用引用参数传递类对象的效率很高。如果以引用参数的形式传递参数，当形参改变时，实参也相应改变。但是，有时候并不希望函数改变数据成员的值。在这种情况下，C++仍然可以通过引用参数来传递参数。为了防止函数改变数据成员的值，可以在形参前冠以保留字const。考虑下面的函数定义：

```
void testTime(const clockType& otherClock)
{
      clockType dClock;
      ...
}
```

函数testTime中有一个引用参数otherClock。参数otherClock用保留字const声明。这样，在调用函数testTime时，形参otherClock接收实参的地址，但是otherClock并不改变实参的内容。例如，当下面的语句执行后，myClock的内容不变：

```
testTime(myClock);
```

一般地，如果要将类对象作为值参数使用，可以像上面所示，利用保留字const，将其声明为引用参数来使用。

如果形参是值参数，在函数定义中，可以改变形参的值。也就是说，可以使用赋值语句来改变形参的值。当然，这样对实参没有影响。但是，当形参是常量引用参数时，不能在函数内使用赋值语句改变参数的值，也不能用任何函数改变此参数的值。所以，在testTime函数定义中，不能改变otherClock的值。例如，下面语句在函数testTime中是不合法的：

```
otherClock.setTime(5, 34, 56); //illegal
otherClock = dClock;           //illegal
```

12.1.4 成员函数的实现

在定义clockType类时，只包含了成员函数的函数原型。为了使函数能够正常工作，必须编写代码实现相关的算法。一种实现函数的方法是，在类中提供函数定义而不是函数原型。但是，这样的类定义会变得很长，以至于难以理解。在类的定义中，只提供函数原型而不提供函数定义的另一个理由与信息隐藏有关。即我们需要隐藏数据操作的细节。在以后的章节中将讨论这个问题(请参见“信息隐藏”一节)。

下面来编写clockType类成员函数的定义。即，编写setTime，getTime，printTime，incrementSeconds，equalTime等函数的定义。因为标识符setTime，printTime等是类的局部标识符，不能在类的作用域范围之外直接引用。为了引用这些标识符，应该使用作用域运算符（::）。函数头应该由函数所在类的名字、作用域运算符、函数的名字依次组成。例如，函数setTime的定义如下所示：

```
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setTime函数检查hours，minutes和seconds的值的合法性。如果这些值超出了范围，数据成员hr，min，sec将初始化为0。下面将解释在访问clockType类型的对象时，成员函数setTime的工作过程。

成员函数setTime是有3个参数的无返回值函数。所以：

- 应该在一条单独的语句中调用该函数
- 调用函数时必须提供3个参数

此外，因为setTime是clockType类的成员函数，所以正如setTime定义中所示，它可以直接访问数据成员hr，min，sec。

假设myClock是clockType类的对象，在myClock对象中有如图12.5所示3个数据成员。

考虑下面的语句：

```
myClock.setTime(3, 48, 52);
```

对象myClock访问成员函数setTime。SetTime函数体的3个变量hr，min，sec是myClock对象的3个数据成员。因此，程序通过函数setTime将3，48，52赋给myClock对象的3个数据成员。语句执行完后，myClock对象如图12.6所示。

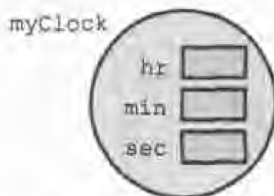


图12.5 myClock对象

图12.6 执行完语句myClock.setTime(3, 48, 52)后的对象myClock

下面给出类clockType中其他成员函数的定义。这些函数定义简单，易于理解：

```
void clockType::getTime(int& hours, int& minutes, int& seconds)
{
```

```
        hours = hr;
        minutes = min;
        seconds = sec;
}

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 的定义中可以清楚看到，在成员函数中可以调用其他的成员函数。

函数 equalTime 的定义如下所示：

```
bool clockType::equalTime(const clockType& otherClock) const
{
   return(hr == otherClock.hr
          && min == otherClock.min
          && sec == otherClock.sec);
}
```

下面介绍函数equalTime的工作过程。

假设myClock和yourClock是类clockType的对象。进一步假设myClock和yourClock对象的内容如图12.7所示。

图12.7 myClock和yourClock对象

考虑下面的语句：

```
if(myClock.equalTime(yourClock))
 .
 .
 .
```

在表达式myClock.equalTime(yourClock)中，myClock对象访问成员函数equalTime。如图12.8所示，参数yourClock的值传递给形参otherClock。

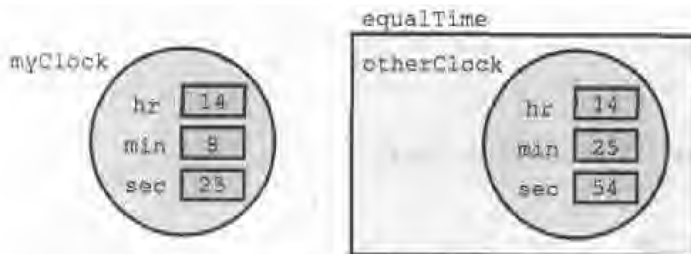


图12.8 myClock对象和otherClock参数

otherClock的数据成员hr，min，sec的值分别是14，25，54。换句话说，当执行函数equalTime时，otherClock.hr的值是14，otherClock.min的值是25，otherClock.sec的值是54。函数equalTime是myClock的成员函数。当执行函数equalTime时，equalTime函数体中的变量hr，min，sec是myClock对象的数据成员。也就是说，myClock中的hr成员和otherClock.hr比较，myClock中的min成员和otherClock.min比较，myClock中的sec成员和otherClock.sec比较。

在equalTime的定义中，再一次清楚地说明了为什么该函数只有一个参数。

在了解了类的定义、类的成员访问、类的成员函数的实现之后，将编写一个使用类的程序。下面是一个使用类clockType的简单函数。

```
//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
```

函数需要用户输入3个数，分别是小时、分钟、秒。假设输入是5 23 59，程序代码走查的输出如下所示：

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
```

用户输入的数据加有阴影。

下面的例子中包含了类的定义、成员函数的定义和函数main，构成了一个完整的程序。

例 12.2

```
//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);
}
```

程序运行结果 *在本程序运行中，用户输入的数据加有阴影。*

```
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
```

yourClock的值在输出的第2行(Line5:)中。这个值是机器相关的，在你的系统上可能会得到不同的值。

12.1.5 类的公有成员和私有成员的顺序

在C++中，公有成员和私有成员的声明没有固定的顺序，可以以任何顺序声明成员。不过要注意的是，类中默认的成员声明是私有成员。如果要声明类的公有成员，必须使用标号public；如果要在公有成员后声明私有成员，必须在私有成员声明开始处使用标号private。

可以采用如例12.3，例12.4和例12.5所示的三种方式之一定义clockType类。

例 12.3

与前面的定义相同。由于完全相同的缘故，这里也包括了类定义。

```
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 12.4

```
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 12.5

```
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 12.5 中，由于标识符 hr，min，sec 没有使用任何的成员访问说明符，所以它们是私有成员。一般来说，先列出所有的公有成员，然后是私有成员。这样，可以把注意力集中在公有成员上。

12.1.6 构造函数

在例12.2的程序中，在打印yourClock的值时，如果没有调用setTime函数，则会输出一些奇怪的数字。这是由于C++并不自动初始化变量。因为类的私有成员在类的范围之外不能够访问，如果用户忘记了调用setTime函数初始化这些变量，程序将会产生错误的结果。

可以通过使用构造函数(Constructor)来保证类中数据成员的初始化。有两种类型的构造函数：含有参数的构造函数和不含参数的构造函数。不含参数的构造函数称为默认构造函数，构造函数有下面的性质：

- 构造函数的名字和类的名字相同。
- 尽管构造函数是函数，但是它没有类型，即它不具有返回值，也不是void函数。
- 类可以有多个构造函数。但是，类的所有的构造函数都必须具有相同的名字。
- 如果类有多个构造函数，这些函数必须具有不同的参数列表。
- 当对象被创建时，构造函数自动执行。由于构造函数没有类型，所以不能被其他函数调用。
- 当声明对象时，通过传递给对象的参数值的类型来确定选择执行哪个构造函数。

下面扩展clockType类的定义，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;
};
```

clockType类的定义包含两个构造函数，其中一个有3个参数，另一个没有参数。下面编写这些构造函数的定义：

```
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;
}
```

从这些构造函数定义不难看出，默认的构造函数将数据成员hr，min，sec初始化为0。带有参数的构造函数将实参值赋给相应的形参。此外，还可以调用setTime函数来实现带有参数的构造函数：

```
clockType::clockType(int hours, int minutes, int seconds)
{
      setTime(hours, minutes, seconds);
}
```

12.1.7　调用构造函数

当声明对象时，构造函数将自动执行。类可以拥有多个构造函数，包括默认构造函数。下面将讨论怎样调用特定的构造函数。

调用默认构造函数

假设类定义包含默认的构造函数，调用默认构造函数的语法是：

```
className   classObjectName;
```

例如，语句：

```
clockType   yourClock;
```

将yourClock定义为clockType类型的对象。这种情况下，执行默认构造函数，将yourClock对象的数据成员初始化为0。

注意：如果声明了对象并想执行默认的构造函数，在对象声明语句中不需要对象名字后面的空括号。事实上，如果不小心使用了空括号，编译器将会输出语法错误的信息。例如，下面的语句是不合法的：

```
clockType   yourClock();  //illegal
```

调用带有参数的构造函数

假设类定义包含带有参数的构造函数，调用带有参数的构造函数的语法是：

```
className   classObjectName(argument1, argument2, ...);
```

其中，argument1, argument2等是变量或者表达式。

注意：

- 参数的数目和类型应该匹配一个构造函数的形参（按照参数顺序）。
- 如果参数的类型不符合任何构造函数的形参，C++应用类型转换寻找最接近的构造函数。例如，整数值可以转换成小数部分是0的浮点型的值。在转换过程中的任何歧义都会导致编译时的错误。

考虑语句：

```
clockType   myClock(5, 12, 40);
```

定义了 clockType 类型的对象 myClock。该对象定义指定了3个 int 类型参数，这与带有参数的构造函数的形参相匹配。所以，执行 clockType 类的带有参数的构造函数，将对象 myClock 的3个数据成员分别设置为5，12和40。

例12.6进一步说明了构造函数是怎样执行的。

例12.6 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;
};
```

该类中共有5个成员函数，其中4个是构造函数；该类中还包括4个数据成员。假设类 testClass 中成员函数的定义如下所示：

```
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;
}
```

考虑下面的语句：

```
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，因为没有传递任何值，所以执行默认构造函数，故对象 one 的 4 个数据成员 x，y，z，ch 分别初始化为 0，0，0，'*'。对于对象 two，因为传递给对象 two 的是 int 类型的 5 和 6，所以执行带有 int 类型参数的构造函数，故对象 two 的数据成员 x，y，z，ch 分别初始化为 5，6，0，'*'。

传递给对象 three 的参数分别是 5，4.5，7。因为这些参数匹配 d 行构造函数的参数，所以执行 d 行的构造函数。根据 d 行的构造函数的定义，对象 three 的数据成员 x，y，z，ch 分别初始化为 5，7，4.5，'*'。

传递给 four 对象的参数是 4，9，12。testClass 类不包含带有 3 个整型参数的构造函数，但是它包含带有 3 个参数分别是整型，双精度型，整型的构造函数（d 行）。由于整数值可以转换成小数部分是 0 的浮点型的值，在 four 对象中，执行 d 行的构造函数。对象 four 的数据成员 x，y，z，ch 分别初始化为 4，12，6.0，'*'。

传递给对象 five 的参数分别是 3.4 和 'D'。这些参数符合 e 行的构造函数的参数，所以执行 e 行的构造函数。根据 e 行的构造函数的定义，将对象 five 的数据成员 x，y，z，ch 分别初始化为 0，0，3.4，'D'。

注意：如果传递给对象的值不符合任何构造函数的参数，并且不能进行可能的类型转换，将会产生编译错误。

下面的程序用来测试前面定义的对象：

```
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
```

正如前面解释，第1行至第5行的语句声明并且初始化对象one，two，three，four，five。第6行至第10行语句输出这些对象的值，即这些对象的数据成员的值。

构造函数和默认参数

构造函数也可以有默认参数。在这种情况下，声明构造函数的默认形参的规则与普通函数中声明默认形参的规则相同。而且，在将实参传递给带有默认参数的构造函数时，应该遵循带有默认参数的函数参数传递规则（第7章讨论了带有默认参数的函数）。应用默认参数的定义规则，在clockType类的定义中，可以使用下面的构造函数来代替已有的两个构造函数（在函数原型中，形参的名字是可选的）：

```
clockType clockType(int = 0, int = 0, int = 0);
```

在函数实现中，该构造函数的定义与带有参数的构造函数的定义相同。

因此，我们可以将没有参数的构造函数或者所有参数带有默认参数的构造函数称为默认构造函数。

例 12.7 应用默认的构造函数，编写类 testClass 的定义如下所示：

```
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所示：

```
testClass::testClass(int a, double c, int  b, char d)
{
      x = a;
      y = b;
      z = c;
      ch = d;
}
```

现在可以定义对象 one，two，three，four，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，我们希望使用 z 和 ch 的默认值，而替换掉 x 和 y 的默认值。因为 z 的声明在 y 之前，为了应用 z 的默认值并且替换 y 的默认值，必须像上面所示来指定 z 的默认值。类似地，在

类 five 的声明中，如果希望使用 x 和 y 的默认值，而替换掉 z 和 ch 的默认值，因为 x 出现在 z 和 ch 之前，并且 y 出现在 ch 之前，必须像上面所示来指定 x 和 y 的默认值。

使用默认参数可以简化构造函数的声明和定义。这样，可以有效地利用带有参数的构造函数。

12.1.8　对象数组和构造函数

如果类有构造函数并且定义了类对象的数组，类必须有默认的构造函数。默认的构造函数用于初始化数组中的每个类对象。如果声明了具有100个类对象的数组，不能为每个类对象指定不同的构造函数。使用以前的 clockType 类的定义，语句：

```
clockType   clocks[ 100] ;
```

定义了有 100 个 clockType 类型元素的数组 clocks（如图 12.9 所示）。

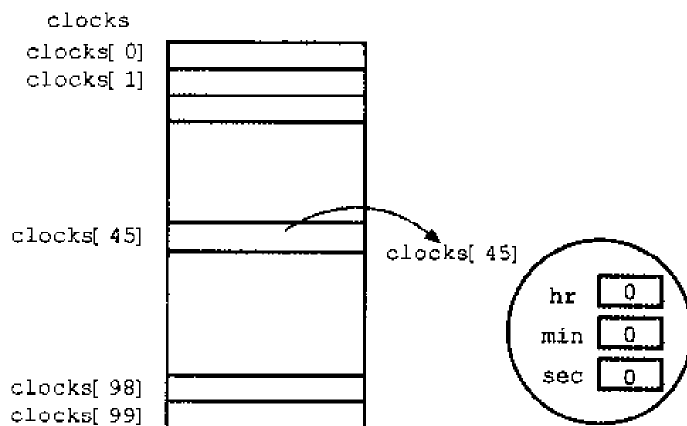


图 12.9　clocks 对象数组

每个数组元素的 3 个数据成员都初始化为 0。

12.1.9　析构函数

像构造函数一样，析构函数（Destructor）也是函数，析构函数没有类型，它既没有返回值，也不是 void 函数。但是，每个类只能有一个析构函数，并且析构函数没有参数。析构函数的名字由字符'~'和类的名字组成。例如，类 clockType 析构函数的名字是：

```
~clockType();
```

程序在退出类对象的作用域（即类对象被释放）时，自动执行类的析构函数。下面章节将讨论析构函数的应用。

12.2　数据抽象、类和抽象数据类型

你曾经想过汽车发动机是怎样工作的吗？我们中的大多数人只是想知道怎样发动汽车和驾驶汽车，并不关心汽车发动机的复杂工作原理。汽车发动机的设计细节和汽车使用的分离，汽车制造商使驾驶员可以将注意力集中在汽车驾驶上。在我们的日常生活中有许多相似的例子。在许多时候，我们只关心怎样使用某个东西，而不需要知道它是怎样工作的。

将设计细节和使用分离的技术称为抽象（Abstraction）。换句话说，抽象只注意汽车发动机做什么，而不关心发动机怎样工作。这样，抽象是一个逻辑特性和实现细节分离的过程。驾驶汽车是逻辑

特性，发动机的构造是实现细节。我们用抽象的观点来看待发动机，对发动机的具体的实现细节不感兴趣。

抽象也可以应用到数据处理上。本章的前面部分定义了clockType数据类型，clockType类有3个数据成员和下面的基本操作：

1. 设置时间
2. 返回时间
3. 打印时间
4. 时间增加一秒钟
5. 时间增加一分钟
6. 时间增加一小时
7. 比较两个时间是否相等

我们实际上并没有在类clockType定义时实现这些操作，而是在后面实现这些操作。数据抽象定义为数据的逻辑特性和实现细节分离的过程：类clockType的定义和它的基本操作属于逻辑特性；clockType在计算机中的存储方式以及实现这些操作的算法属于实现细节。

抽象数据类型(Abstract data type, ADT) 只确定逻辑特性而没有实现细节的数据类型。

像其他的数据类型一样，抽象数据类型有3个相关的属性：抽象数据类型的名称，称为类型名称；属于抽象数据类型的一系列值，称为域(Domain)；以及数据的一系列操作。根据这些规定，可以定义clockType抽象数据类型如下所示：

```
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12.8 表定义为相同类型的一系列的值。因为表中的所有的值具有相同的类型，表示和处理表的传统的方法是使用数组。可以将表的抽象数据类型定义如下所示：

```
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.
```

接下来一个很明显的问题是：如何在程序中实现抽象数据类型？为了实现抽象数据类型，必须描述数据并且编写算法来完成操作。

在前面说明中，使用类将数据和函数组织在一起。而且，在我们的类定义中只包含了操作的说明，实现操作的函数单独实现。由此不难看出，类是实现抽象数据类型的一种便利的方法。实际上，C++ 类为处理抽象数据类型做了特殊的设计。

下面的类 listType 定义了表的抽象数据类型，假设表的每个元素是 int 型的数据（如图 12.10 所示）。

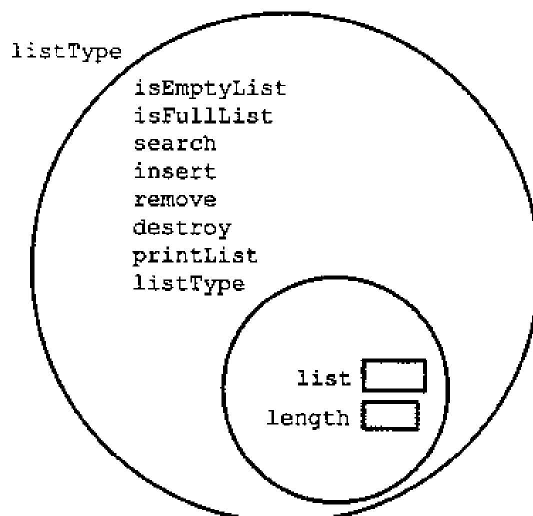


图 12.10 类 listType

```
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;
};
```

12.3 结构和类的比较

第 11 章将结构定义为固定成员的集合，结构中的成员可以是不同数据类型。这种结构定义中的成员只包括了数据成员。然而，C++ 中的结构与类很相似。和类相同，结构的成员还可以是函数，包括构造函数和析构函数。结构和类的惟一不同之处在于，在默认的情况下，结构的所有成员是公有成员，而类的所有成员是私有成员。可以在结构中使用标号 private 来声明私有成员。

在 C 语言中，结构的定义类似于第 11 章中给出的 C++ 的结构的定义。因为 C++ 是从 C 进化来的，C++ 中完全接受标准的 C 语言的结构。但是，在 C++ 中结构的定义被扩展，可以包括成员函数、构造函数和析构函数。由于类在语法上是独立的，特别是在设计抽象数据类型时，类的定义与 C 语言的结构定义完全不同。

C++中结构和类具有相同的功能。但是，大多数程序员只使用类C的结构形式，即不在结构中包含成员函数。也就是说，如果类的所有的数据成员都是公有成员，不包含任何成员函数，那么一般使用结构来组合这些成员。实际上，本书正是这样做的。

12.4 信息隐藏

在前面一节的程序中，定义了用来实现时间的类clockType。接着，我们编写了使用clockType类的程序。实际上，我们将类clockType的定义和实现操作的成员函数定义，以及函数main一起构成完整的程序。即类clockType的定义和具体实现细节都写在了同一程序中。

在程序中，将对象的描述和实现细节写在同一程序中好吗？当然不好。理由是，首先，如果这样做，对象的私有数据成员就不再是对象私有的。用户可以通过实现数据操作的函数可以直接访问数据成员。这样，用户可以以他喜欢的方式修改操作。假设几个程序员在某个工程项目中使用同一个对象，如果他们可以直接访问对象的内部，则不能保证每个程序员都以完全相同的方式使用对象。为了防止发生这种情况，必须隐藏具体的实现细节。用户只应该知道对象是做什么的，不应该知道对象是怎么做的。隐藏实现细节可防止用户在程序中修改这些代码。而且，隐藏细节能够保证在整个工程项目中，对象以完全相同的方式使用。此外，经过编码、调试、测试过的对象可保证没有错误，可以使用在程序中的任何部分。

本节将用类clockType来举例说明，怎样隐藏对象的实现细节。

为了可以在程序中使用clockType类，用户必须定义clockType类型的对象，并且知道哪些操作是允许的，以及这些操作的功能是什么。所以用户必须可以访问说明细节。由于用户并不关心实现细节，我们应该将实现的细节放在单独的文件中，称为实现文件（Implementation File）。因为类说明细节可能很长，我们必须可使用户能够不在程序中直接包含类的说明细节。但是，为了让用户正确地调用函数，用户还必须要看到类的说明细节。所以可把类的说明细节放在单独的文件中，包含类的说明细节的文件称为头文件。

实现文件包含实现对象操作的成员函数定义。实现文件中除了包含C++语句外，还包含别的东西（像预处理命令）。因为C++只有一个函数main，所以实现文件不包含函数main。只有用户程序中包含函数main。因为实现文件不包含函数main，实现文件不产生执行代码。实际上，实现文件产生所谓的目标代码。用户连接由实现文件产生的目标代码和使用此类的程序目标代码，从而最终生成可执行代码。

头文件的扩展名是h，而实现文件的扩展名是cpp。假设clockType类的说明细节在文件clock中，那么这个文件的完整名字应该是clock.h；如果clockType类的实现细节放在文件clockImp中，那么这个文件的完整名字应该是clockImp.cpp。

文件clockImp.cpp中只包含了函数定义，而不包含类定义。为了解决未声明的标识符的问题（像函数名和变量名），应该使用include语句在clockImp.cpp中包含头文件clock.h。任何应用clockType类的程序，都必须包含定义clockType类的头文件：

```
#include "clock.h"
```

注意头文件clock.h用双引号括住，而不是尖括号。头文件clock.h是用户定义的头文件。所有用户定义的头文件都用双引号括住，而系统提供的头文件（像iostream）都用尖括号括住。

下面是clockType类的说明文件和实现文件：

```
//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，描述用户使用clockType类的程序文件：

```
//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
.
.
.
```

12.5 可执行代码

前面一节讨论了怎样隐藏类的实现细节。为了使用对象，在程序的执行过程中必须能够访问对象的实现细节（即实现对象操作的算法）。本节将讨论用户程序怎样访问对象的实现细节。这里，用clockType类举例说明。

正如上面所述，为了使用clockType类，必须在程序中使用include语句包含头文件clock.h。例如，下面的程序段包含了头文件clock.h：

```
//Program test.cpp
#include "clock.h"
.
.
.
int main()
{
    .
    .
    .
}
```

程序test.cpp只包含相应头文件，并不包含实现文件。为了构造运行test.cpp程序的可执行代码，需要下面的步骤：

1. 单独编译文件clockImp.cpp产生目标文件clockImp.obj。该目标文件中包含不可执行形式的机器语言代码。假设命令cc（UNIX系统上C++编译命令）在操作系统命令行上调用C++的编译程序或者连接程序，或同时调用两者，则命令：

   ```
   cc -c clockImp.cpp
   ```

   产生目标文件clockImp.obj。

2. 为了产生源文件test.cpp的可执行代码，我们编译源文件test.cpp得到目标文件test.obj，接着连接test.obj和clockImp.obj产生可执行文件test.exe。在UNIX操作系统命令行上使用下面的命令可产生可执行文件test.exe：

```
cc test.cpp clockImp.obj
```

注意： 1. 为了产生源文件的目标文件，可以在UNIX操作系统命令行上使用命令选项 -c。例如，为了产生源文件exercise.cpp的目标文件，在操作系统命令行上用下面的命令：

```
cc -c exercise.cpp
```

2. 为了使源文件连接所需的目标文件，在操作系统命令行上需要列出所有有关的目标文件。例如，为了连接源文件test.cpp与A.obj和B.obj，应该使用命令：

```
cc test.cpp  A.obj  B.obj
```

3. 如果修改源文件，必须重新编译源文件。
4. 如果修改源文件影响到其他的文件，其他的源文件必须重新编译和重新连接。
5. 用户必须能够访问头文件和目标文件。访问头文件的目的是查看该对象有哪些操作以及怎样使用这些操作。访问目标文件的目的是用户可以将程序和目标文件连接在一起生成可执行代码。用户不需要访问包含实现细节的源文件。

第1章中曾经提到，诸如Visual C++，C++ Builder，CodeWarrior等软件开发工具将编辑器、编译器、连接器集成在同一个环境中。通过一个命令，就可以编译程序和并将一些必需的文件连接到程序中。此外，这些系统以项目的形式管理多个文件。这样，一个项目可包含多个文件，它们称为项目文件。这些系统通常有build，rebuild，make命令（查阅所在系统的文档）。当在项目上使用这些命令时，系统自动编译和连接所有文件从而生成可执行代码。当项目的一个或者多个文件修改后，可使用这些命令重新编译和连接文件。

例12.9进一步说明怎样设计并实现类。例12.9中personType类的设计十分有用，我们将在后续的章节中继续使用这个类。

例 12.9　人最通常的属性是姓名；对人的名字最常见的操作是设置名字和打印名字。下面的语句定义了具有这些属性的类（如图12.11所示）：

```
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
};
```

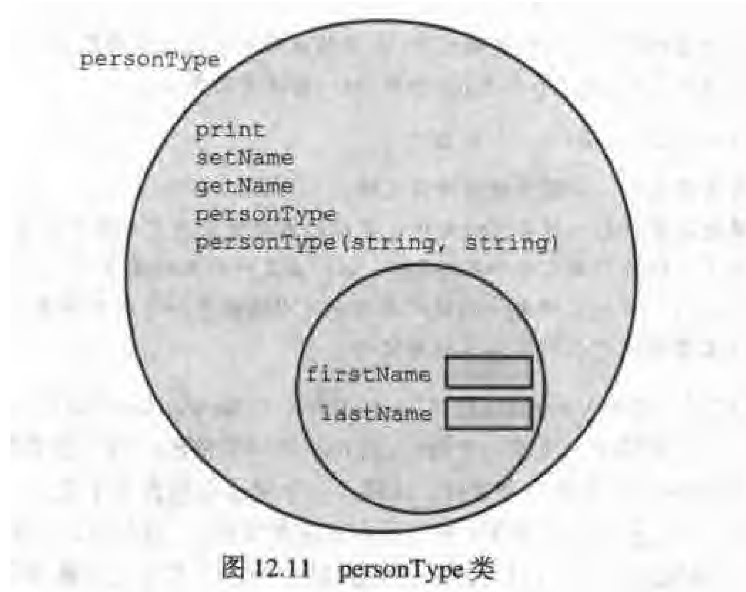


图 12.11 personType 类

下面给出 personType 类的成员函数定义：

```
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 = "";
}
```

我们可以使用一个带有默认参数的构造函数来取代personType类的所有的构造函数；该函数的实现细节作为练习留给读者。

12.6 程序范例：糖果自动销售机

人们通常从糖果自动销售机中购买糖果。假设某体育馆新买来一部糖果自动销售机，但是该糖果自动销售机由于缺少销售处理程序而不能正常工作。假设糖果自动销售机可以销售糖果、薯片、口香糖、饼干。请编写程序使糖果自动销售机能够正常工作。

程序完成以下工作：

1. 向顾客显示糖果自动销售机销售的食品品种
2. 顾客能够做出购买选择
3. 向顾客显示食品的价格
4. 接受顾客的付费
5. 给出食品

输入　选择食品及其价格

输出　选择的食品

问题分析和算法设计

糖果自动销售机有两个主要的部分：内置的收银机和销售食品的自动售货机。

收银机　首先讨论收银机的属性。收银机有一些库存现金，它可以接收顾客的付费。如果顾客的付费超过食品的价格，它能够找给顾客零钱。为了简便起见，假设顾客的付费恰好等于食品的价格。收银机能够在任何时候向糖果自动销售机拥有者显示收银机中现有的钱数。下面的类定义了收银机的属性（如图12.12所示）：

```
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
};
```

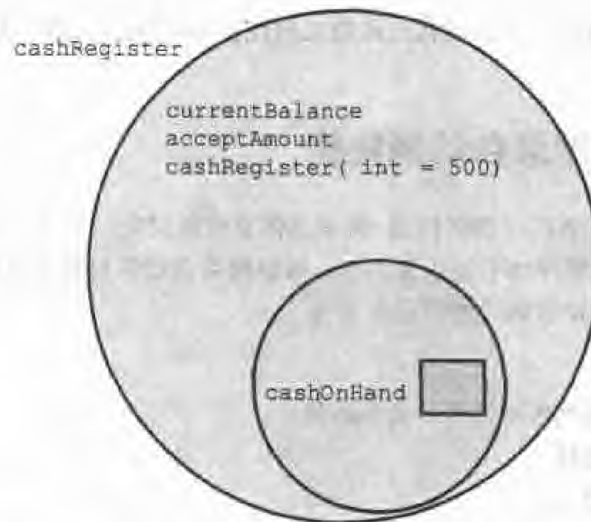


图 12.12 cashRegister类

下面给出cashRegister类中成员函数的定义，这些函数的定义很简单。函数currentBalance用于显示收银机的钱数，即返回私有数据成员cashOnHand的值，所以它的定义是：

```
int cashRegister::currentBalance()
{
      return cashOnHand;
}
```

函数acceptAmount用于接收顾客输入的付费，将其与收银机现有的钱数相加，然后更新收银机的钱数。所以该函数的定义是：

```
void cashRegister::acceptAmount(int amountIn)
{
      cashOnHand += amountIn;
}
```

在类cashRegister中，定义了带有默认参数的构造函数。所以，如果在对象定义时没有指定任何参数值，数据成员cashOnHand初始化为默认的值。因为已经在类定义的构造函数原型中指定了默认的参数值，所以在构造函数的定义中不需要在函数头中指出默认参数值。构造函数的定义如下所示：

```
cashRegister::cashRegister(int cashIn)
{
     if(cashIn >= 0)
        cashOnHand = cashIn;
     else
        cashOnHand = 500;
}
```

注意，在构造函数的定义中检查参数cashIn的合法性。如果cashIn的值小于0，数据成员cashOnHand赋值为500。

自动售货机 如果用户选择的食品没有销售完，自动售货机应该给出用户选择的食品。自动售货机应该显示食品的数目和价格。下面的dispenserType类定义了自动售货机的属性（如图12.13所示）：

```
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
};
```

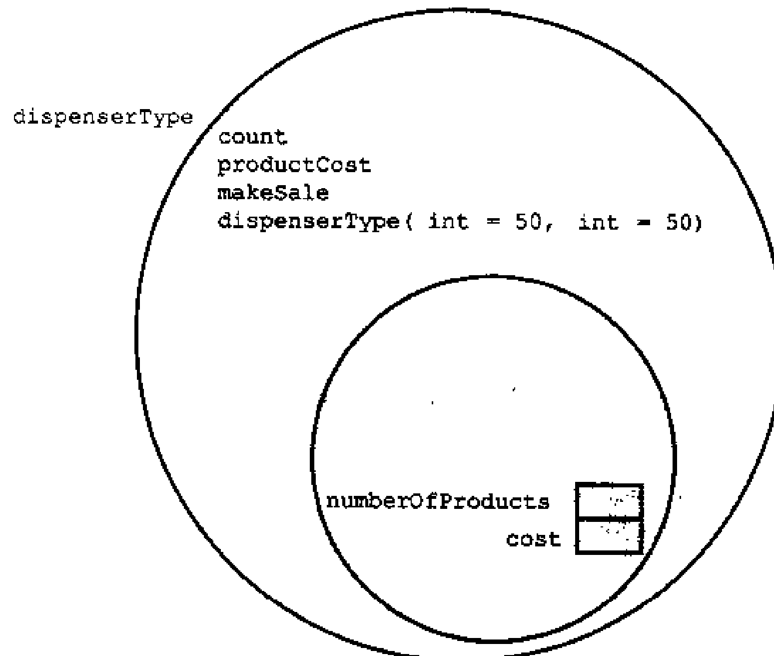


图 12.13　dispenserType 类

因为糖果自动销售机销售 4 种食品，应定义 4 个 dispenserType 类型的对象，例如语句：

```
dispenserType chips(100,65);
```

定义了 dispenserType 类型的对象 chips，并将 chips 的数目设置为 100，每盒 chip 的价格是 65 美分（如图 12.14 所示）。

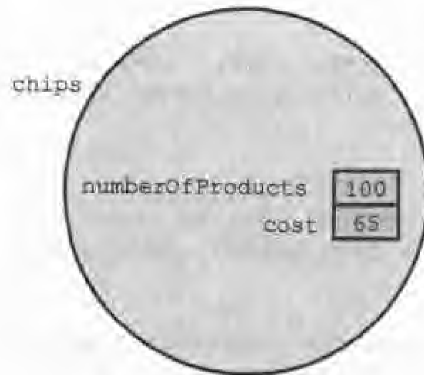


图 12.14 chips 对象

下面讨论实现类 dispenserType 操作的成员函数的定义。

函数 count 返回特定食品的数目。因为自动售货机现有食品的数目存储在私有数据成员 numberOfProducts 中，所以函数返回私有数据成员 numberOfProducts 的值。该函数的定义如下所示：

```
int dispenserType::count()
{
      return numberOfProducts;
}
```

函数 productCost 返回食品的价格，因为食品的价格存储在私有数据成员 cost 中，所以函数返回私有数据成员 cost 的值。函数定义如下所示：

```
int dispenserType::productCost()
{
      return cost;
}
```

当销售了一种食品后，自动售货机的食品数目减 1。所以函数 makeSale 使自动售货机中的食品数目减 1。即私有数据成员 numberOfProducts 的值减 1。该函数的定义如下所示：

```
void dispenserType::makeSale()
{
      numberOfProducts--;
}
```

在构造函数的定义中检查参数的合法性。如果参数值小于 0，数据成员赋值为默认值，构造函数的定义如下所示：

```
            //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

在程序执行时，必须能够完成以下的功能：

1. 显示糖果自动销售机销售的食品品种
2. 显示怎样选择购买特定的食品
3. 显示怎样终止程序

另外，必须在处理顾客的每次选择时都显示这些提示信息（除非终止程序）。这样，如果顾客购买多种食品，不需要记住应该怎样做。一旦顾客做出了适当的选择，糖果自动销售机必须执行相应的操作。如果顾客选择了购买某种食品而且该种食品没有售完，糖果自动销售机应该显示食品的价格并且提示顾客付费。如果付费不少于食品的价格，糖果自动销售机应该给出食品并且显示相应的信息。

根据上面的讨论，得出算法如下所示：

1. 提示顾客选择
2. 得到顾客的选择
3. 如果顾客选择是正确的，并且自动售货机中顾客选择的食品没有销售完，售出顾客选择的食品

我们将该程序分解为 3 个函数：showSelection，sellProduct 和 main。

函数 showSelection　显示用户选择和购买食品时必需的帮助信息，它应该包含以下的输出语句（假设糖果自动销售机销售 4 种食品）：

```
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
```

函数 showSelection 的定义如下所示：

```
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
```

函数 sellProduct　该函数要销售顾客选中的食品。所以，它必须能够访问存储食品的自动售货机。首先函数检查自动售货机中的食品是否销售完。如果已经销售完，则应该通知顾客相应的信息。如果顾客所选择的食品没有销售完，通知顾客付费购买食品。

如果顾客所付的购买食品的费用不足，函数 sellProduct 通知顾客接着付剩余的费用。如果顾客还是没有付购买食品所需的足够的费用，函数退回顾客的付费。如果顾客的付费够了，糖果自动销售机接收顾客的付费，销售食品。销售食品意味着自动售货机的食品数目减 1，并且在收银机钱数上加上售出食品的价格（由于程序不返回顾客购买食品的余额，收银机和顾客的付费相加得到收银机现有的钱数）。

从上面的讨论中可以看出，函数sellProduct能够访问存储食品的自动售货机（使自动售货机的食品数目减1并且显示食品价格）和收银机（更新收银机现有钱数）。所以函数有两个参数，分别传递自动售货机和收银机，这两个参数都是引用参数。

该函数算法的伪代码形式如下所示：

a. 如果自动售货机的食品没有销售完：
  i. 提示顾客对选定食品付费
  ii. 得到顾客的付费
  iii. 如果顾客的付费少于食品的价格：
    (1) 提示顾客接着付剩余的费用
    (2) 计算顾客的整个付费
  iv. 如果顾客的付费不少于食品的价格：
    (1) 更新收银机的钱数
    (2) 销售食品，使自动售货机中的食品数目减1
    (3) 显示相应的信息
  v. 如果顾客的付费少于食品的价格，退还顾客的付费

b. 如果自动售货机的食品销售完，通知顾客食品已销售完。

函数sellProduct的定义如下所示：

```
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
```

函数 main　函数 main 的算法如下所示：

1. 创建收银机，即定义 cashRegister 类的对象。
2. 创建自动售货机，即定义 dispenserType 类的 4 个对象并且将其初始化。例如语句：

   ```
   dispenserType  candy(100,50);
   ```

   创建了装载糖果的自动售货机 candy 对象。糖果的数目是 100，每个糖果的价格是 50 美分。
3. 定义其他必须的变量。
4. 调用函数 showSelection 显示食品选择。
5. 得到顾客的选择。
6. 当顾客的选择不是退出程序（选择 9 退出程序）：
   a. 调用函数 sellProduct 销售食品
   b. 调用函数 showSelection 显示选择
   c. 得到选择

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
```

完整的程序代码清单

```
//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;
   }
   else
      cout<<"Sorry this item is sold out."<<endl;
}//end sellProduct
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
*** Welcome to Shelly's Candy Shop ***
To Select an item , enter
1 for Candy
2 for Chips
3 for Gum
4 for Cookies
9 to exit
1
Please deposit 50 cents
50
Collect your item at the bottom and enjoy
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

*** Welcome to Shelly's Candy Shop ***
To Select an item, enter
1 for Candy
2 for Chips
3 for Gum
4 for Cookies
9 to exit
3
Please deposit 45 cents
```

```
45
Collect your item at the bottom and enjoy
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

*** Welcome to Shelly's Candy Shop ***
To Select an item, enter
1 for Candy
2 for Chips
3 for Gum
4 for Cookies
9 to exit
9
```

12.7 小结

1. 类是具有固定数目元素的集合。
2. 类的组成元素称为类的成员。
3. 类成员通过名字访问。
4. 在C++中，class是保留字。
5. 类的成员分为3类：私有成员、受保护成员、公有成员。
6. 不能在类的范围以外访问类的私有成员。
7. 可以在类的范围之外访问类的公有成员。
8. 在默认情况下，类的所有成员是私有成员。
9. 类的公有成员用标号public声明。
10. 类的私有成员用标号private声明。
11. 类的成员可以是函数或者是变量。
12. 如果类的成员是函数，通常只声明函数原型。
13. 如果类的成员是变量，其声明与其他的变量声明一样。
14. 在类的定义中，声明变量时不能进行初始化。
15. 在C++中类只是类型定义，不涉及存储分配，仅当声明类的变量时，才分配存储空间。
16. 在C++中，类变量称为类对象或者对象。
17. 调用类的程序（软件）称为类的客户程序。
18. 用类的变量名字、点运算符（.）和成员名字访问类成员。
19. 在类中内建的操作是赋值和成员访问。
20. 类可以作为值参数或者引用参数传递给函数。
21. 函数的返回值可以是类类型的值。
22. 当声明对象时，构造函数保证数据成员被初始化。
23. 构造函数的名字和类的名字相同。
24. 类可以有多个构造函数。
25. 没有参数的构造函数称为默认的构造函数。
26. 类对象进入其作用域（类对象被创建）时，构造函数自动执行。
27. 类对象退出其作用域（类对象被释放）时，析构函数自动执行。
28. 类只有一个析构函数，并且析构函数没有参数。
29. 析构函数的名字由（~）和类的名字组成（二者之间没有空格）。

30. 构造函数和析构函数是没有类型的函数，它们没有返回值，也不是void函数，不能被其他函数调用。
31. 只有逻辑特性而没有实现细节的数据类型称为抽象数据类型（ADT）。
32. 在C++中，类为处理抽象数据类型做了专门的设计。
33. 为了使用抽象数据类型，必须描述数据并实现相应操作的算法。

12.8 练习

1. 判断下面说法的正误。
   a. 类的数据成员必须是同一类型。
   b. 类的成员函数必须是公有成员。
   c. 类可以有多个构造函数。
   d. 类可以有多个析构函数。
   e. 构造函数和析构函数都可以有参数。
2. 在下面的类定义中找出语法错误。

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. 考虑下面的声明：

```
class xClass
{
 public:
     void func();
     void print() const;
     xClass ();
     xClass (int, double);
 private:
     int u;
     double w;
 };

 xClass x;
```

a. 类x有几个成员？

b. 类x有几个私有成员？

c. 类x有几个构造函数？

d. 编写成员函数func，将u和w的值分别设置为10和15.3。

e. 编写成员函数print，打印u和w的值。

f. 编写xClass类的默认构造函数的定义，其将私有数据成员初始化为0。

g. 编写打印对象x的数据成员值的C++语句。

h. 编写C++语句，声明xClass类型的对象t，并将t的数据成员分别初始化为20和35.0。

4. 考虑下面的类定义：

```
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. 当有下面声明时，给出其所执行的构造函数的行号：

(i) `CC one;`

(ii) `CC two(5, 6);`

(iii) `CC three(3.5, 8);`

b. 编写第1行的构造函数的定义，其使私有数据成员初始化为0。

c. 编写第2行的构造函数的定义，其使私有数据成员u初始化为相应的参数值，v初始化为0。

d. 编写第3行、第4行的构造函数的定义，它们使私有数据成员初始化为相应的参数值。

5. 考虑下面的类定义：

```
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. 为testClass类，编写成员函数的定义。

b. 编写测试程序，测试testClass类的各种操作。

12.9 编程练习

1. 编写一个程序，该程序将输入的罗马数字转换成相应的十进制数字形式。该程序包含romanType类，romanType类的对象应该有以下功能：

a. 存储罗马数字形式的数。

b. 把存储的罗马数转换成相应的十进制数字形式。

c. 按用户的要求，以罗马数字形式或者十进制形式打印数。罗马数字相应的十进制值如下所示：

```
M       1000
D        500
C        100
L         50
X         10
V          5
I          1
```

d. 用下面的罗马数：MCXIV，CCCLIX，MDCLXVI测试编写的程序。

2. 设计和实现描述一周中某一天的dayType类。dayType类存储某天是星期几的信息，例如把星期天存储为Sun。在dayType类型的对象上可以执行下面的操作：

a. 设置某一天

b. 打印某一天

c. 返回某一天

d. 返回下一天

e. 返回前一天

f. 通过对现有的日期加上一定天数，计算和返回相应的日期应该是星期几。例如如果现在是星期一，加4天，返回星期五。类似地，如果今天是星期二，加13天，返回星期一。

g. 添加适当的构造函数。

3. 对于编程练习2中的dayType类，编写实现操作的成员函数的定义。
4. 例12.9定义了存储人的姓名的personType类。类的成员函数仅包含打印和设置人的名字。在已有的基础上重新定义personType类，使之可以：
   a. 设置人的姓（last name）
   b. 设置人的名(first name)
   c. 存储和设置人的中间名(middle name)
   d. 检查给定的姓是否与这个人的姓相同
   e. 检查给定的名是否与这个人的名相同
5. a. 一本书的属性包括书名、作者、出版社、国际标准图书编号（ISBN）、价格、出版日期。设计定义书的抽象数据类型bookType类。
   (i) 类bookType应包含书的下面的信息：书名、作者（最多四个）、出版社、国际标准图书编号、价格、库存数。为了知道作者人数，另有一个信息表示作者数。
   (ii) 类bookType应包含实现各种操作的成员函数，例如，针对书名的一般操作有显示书名，设置书名，检查是否有相同的书名。类似地，针对库存数的一般操作有显示库存数，设置库存数，更新库存数，返回库存数。针对书的作者、出版社、国际标准图书编号、价格添加类似操作。并添加合适的构造函数和析构函数（如果需要）。
   b. 编写bookType类的成员函数的定义。
   c. 编写使用bookType类的程序，并且测试bookType类对象的各种操作。声明有100个bookType类型的元素的数组，执行以下操作：通过书名检索书籍、通过国际标准图书编号检索书籍、更新库存数。
6. 设计memberType类。
   a. 每个memberType类的对象包含人名、成员ID、购买的图书数量、支付的费用。
   b. 每个memberType类的对象包含各种成员函数来完成各种操作。例如，修改、设置、显示人名。类似地，更新、修改、显示购买图书的数量和支付的费用。
   c. 添加相应的构造函数和析构函数（如果需要）。
   d. 编写memberType类的成员函数的定义。
7. 用编程练习5和编程练习6中设计的类，编写一个模拟书店的程序。书店有两种类型的顾客：书店会员和普通购书者。书店的每个会员每年支付$10的会员费，并且购书享受5%的折扣。
   对于每个书店会员，书店记录其购买的书的数量和总的支付费用。会员每购买的第11本书，将收取上10本书的平均费用作为本书折扣的方式。然后，将会员总的支付费用重新设置为0。
   编写能够处理1 000种图书和500个书店会员的程序。为了使程序可以高效运行，程序给出用户购书的选择菜单。换句话说，程序应该可以自动完成其他功能。

第 13 章　继承和组成

本章要点：

- 了解继承
- 了解派生类和基类
- 理解怎样重定义基类中的成员函数
- 了解基类和派生类的构造函数
- 理解怎样构造派生类的头文件
- 了解三种继承类型：公有、受保护和私有
- 了解组成
- 熟悉面向对象程序设计的三条基本原则

在第 12 章中介绍了类、抽象数据类型和 C++ 实现抽象数据类型的方法。使用类可以把数据和操作封装在单独的单元中。因此，一个对象成为一个独立的实体。操作可以直接访问数据，但是对象的内部状态却不能够被直接操作。

除了实现抽象数据类型，类还有其他的特征。例如，可以从现有类的基础上创建新的类。这个重要的特征鼓励了代码复用。在 C++ 中类之间可以有多种关系，其中两种常见的关系是：

- **继承**（" is-a" 关系）
- **组成**（"has-a" 关系）

13.1　继承

假设要设计partTimeEmployee类来实现和处理临时员工的属性。临时员工的主要属性有姓名、单位时间工资和工作时间。在例 12.9（第 12 章）中设计实现了一个描述人名字的类。每个员工都是人，所以我们想通过添加成员（数据或者函数）扩展 personType 类，而不是重新设计 partTimeEmployee 类。

当然，我们并不想改变 personType 类，即编辑 personType 类、添加或者删除其成员。实际上，我们不对 personType 类做任何改变，而是通过添加必需的成员来创建 partTimeEmployee 类。例如，personType 类已经含有存储人的姓名的数据成员，partTimeEmployee类可以不必包含这两个成员。实际上，可以从 personType 类继承这些数据成员（在例 13.1 中设计了这样的类）。

在第 12 章中，深人研究和设计实现描述一天的某个时间的 clockType 类。clockType 类有 3 个数据成员，分别存储小时、分钟和秒。有些应用程序除了存储小时、分钟和秒之外，还要求存储时区。这种情况下，扩展 clockType 类的定义，创建 extClockType 类来存储时区。即通过添加数据成员 timeZone 和操作时间的必要成员函数（参见本章结束部分的编程练习 1），派生出 extClockType 类。在 C++ 中，完成这项任务的技术是继承。继承是 " is-a" 关系，例如，每个员工都是人。

继承（Inheritance）从现有类的基础上创建新类，创建的新类称为派生类（Derived class），现有类称为基类（Base class）。派生类继承基类的属性。这样，可以避免重新构造含有基类信息的完整的新类，充分利用继承来减小软件的复杂度。

每个派生类还可以成为它的派生类的基类，继承分单继承和多继承。在单继承(Single inheritance)中，派生类从一个基类派生；在多继承(Multiple inheritance)中，派生类从多个基类派生。本章的重点是单继承。

基类和派生类的继承关系可以看做是树形结构，或者是层次结构。考虑图13.1的继承关系的树形结构图。

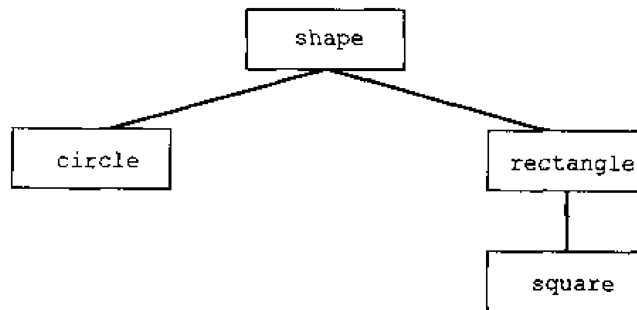


图 13.1 继承层次

在该图中，shape是基类，circle类和rectangle从shape类派生，square类从rectangle类派生。每一个circle和rectangle都是一种shape，每一个square是一种rectangle。

假设定义了shape类，为了说明circle类从shape类派生，通常使用下面的语法来定义：

```
class circle: public shape
{
    .
    .
    .
};
```

类头部的保留字public称为成员访问说明符。它的作用是指定shape类的所有的公有成员在circle类中仍为公有成员。换句话说，shape类的公有成员变成circle类的公有成员。

考虑下面的circle类的定义：

```
class circle: private shape
{
    .
    .
    .
};
```

在上面的定义中，shape类的所有的公有成员在circle类中成为私有成员。所以，任何circle类型的对象都不能直接访问这些成员。前面的circle的定义等同于：

```
class circle: shape
{
    .
    .
    .
};
```

如果类的定义中不使用成员访问说明符public或者private，则基类的公有成员在派生类中成为私有成员。

定义派生类的语法如下所示：

```
class className: memberAccessSpecifier baseClassName
{
```

```
    member list
};
```

其中 memberAccessSpecifier（成员访问说明符）是 public，private 或者 protected。当没有指定 memberAccessSpecifier 时，默认的是 private（私有）继承。本章的后面将讨论 protected（受保护）继承。

牢记下面的关于基类和派生类的规则：

1. 基类的私有成员是基类所私有的，派生类的成员不能直接访问基类的私有成员。当编写派生类的成员函数定义时，不能够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成员或者私有成员，也就是说基类的公有成员可以变成派生类的公有或者私有成员。
3. 派生类可以包含自己的数据成员或者函数成员。
4. 派生类可以重定义基类的公有成员函数。也就是说，派生类中可以有和基类同名的成员函数，甚至参数个数和类型也可以相同。但是，重定义的成员函数只适用于派生类对象，不适用于基类对象。
5. 基类的所有数据成员也是派生类的数据成员。类似地，基类的成员函数（除非重定义）也是派生类的成员函数（在派生类中访问基类的成员时，遵照规则 1）。

下面几节将讨论关于继承的两个重要的问题。第一个是派生类中基类成员函数的重定义问题。在讨论这个问题时，顺便介绍怎样在派生类中访问基类的私有成员。第二个是构造函数问题。派生类的构造函数不能直接访问基类的私有数据成员。因此，在执行派生类的构造函数时，必须保证能够初始化从基类继承的私有数据成员。

13.1.1　基类成员函数的重定义

假设 derivedClass 类从 baseClass 类派生，而且 derivedClass 和 baseClass 有一些共同的数据成员。derivedClass 既包含 baseClass 类的数据成员，又包含其自身的数据成员。假设 baseCalss 包含的成员函数 print 用于输出 baseClass 中数据成员的值。现在 derivedClass 除了包含 baseClass 类的数据成员以外，还包含其他的数据成员。如果希望在 derivedClass 类中包含用于输出 derivedClass 类数据成员的函数，当然可以任意定义函数名称。但是在 derivedClass 类中，也可以把函数名定义为 print（和 baseClass 类的函数名称相同）。这种情况称为基类成员函数的重定义（Redefining）。下面使用具体的范例来说明怎样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，派生类中相应的成员函数必须与基类成员函数名称相同、参数数目相同、参数类型相同。换句话说，派生类中重定义的函数有和基类函数相同的参数名称和参数列表。但是，如果派生类函数和相应的基类函数只是函数名称相同，而参数列表不同，也是合法的。这被称为派生类函数重载（Overloading）。

考虑下面的类定义（参见图 13.2）：

```
class baseClass
{
public:
    void print() const;

private:
    int u;
    int v;
    char ch;
};
```

baseClass 类有 4 个成员，假设 baseClass 类的 print 成员函数的定义是：

```
void baseClass::print() const
{
      cout<<"Base Class: u = "<<u<<", v = "<<v
          <<", ch = "<<ch<<endl;
}
```

接着考虑下面的类定义（参见图 13.3）：

```
class derivedClass: public baseClass
{
public:
    void print() const;

private:
    int first;
    double second;
};
```

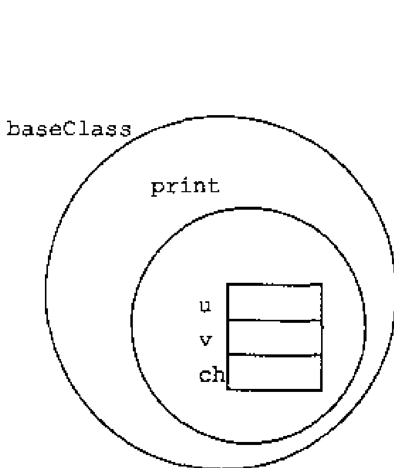


图 13.2 baseClass 类

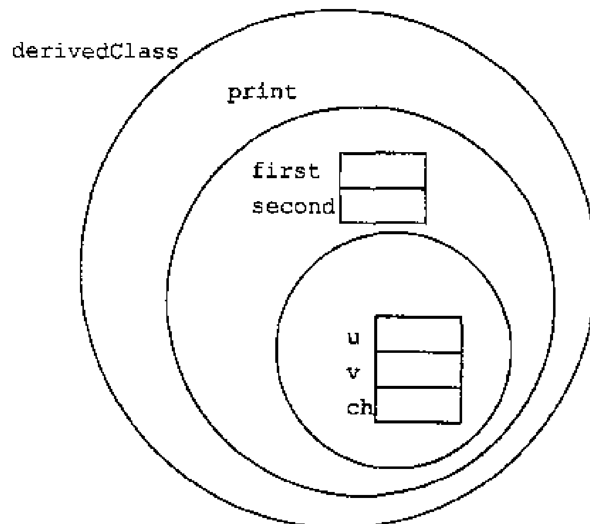


图 13.3 derivedClass 类

从 derivedClass 类的定义可以看出，derivedClass 类继承于 baseClass 类，并且是公有继承。所以 baseClass 类的所有公有成员在 derivedClass 类中仍然是公有成员。derivedClass 类重定义了 print 函数。

下面编写 derivedClass 类中成员函数 print 的定义。

derivedClass 类有 5 个数据成员：u，v，ch，first，second。derivedClass 类的函数 print 用于输出这些数据成员的值。在编写 derivedClass 类的 print 函数的定义时要注意：

- u，v，ch 是 baseClass 类的私有成员，不能在 derivedClass 类中直接访问。所以在 derivedClass 类的 print 函数定义中不能直接访问 u，v，ch。
- 在derivedClass类中，可以通过baseClass类的公有成员函数访问baseClass类的数据成员u，v，ch。

所以，derivedClass 类的 print 函数要首先调用 baseClass 类的 print 函数打印 u，v，ch 的值，然后再输出 first 和 second 的值。

为了在 derivedClass 类的 print 函数中调用 baseClass 类的 print 函数，需要使用下面的语句：

```
baseClass::print();
```

这样就保证了调用的是baseClass类的print函数，而不是derivedClass类的print函数。

derivedClass类的print成员函数的定义如下所示：

```
void derivedClass::print() const
{
       baseClass::print();
       cout<<"Derived Class: first = "<<first
           <<", second = "<<second<<endl;
       cout<<"___________________________________"
           <<"________"<<endl;
}
```

考虑下面的语句：

```
baseClass    baseObject;
derivedClass derivedObject;
```

第1条语句声明了baseObject是baseClass类的对象，baseObject对象有3个数据成员：u，v，ch。第2条语句声明了derivedObject是derivedClass类的对象，derivedObject有5个数据成员：u，v，ch，first，second（参见图13.4）。

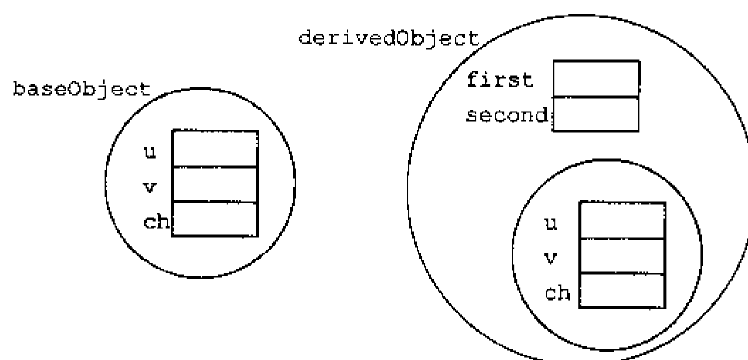


图 13.4　baseObject 和 derivedObject 对象

假设baseObject对象的数据成员u，v，ch的值分别是5，6，*。derivedObject对象的数据成员u，v，ch，first，second的值分别是12，76，&，32和16.38（参见图13.5）。

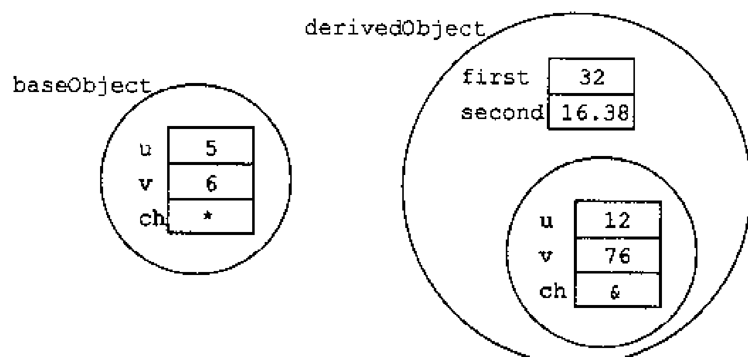


图 13.5　baseObject 和 derivedObject 对象的内容

考虑下面的语句：

```
baseObject.print();
derivedObject.print();
```

在第 1 条语句中，执行 baseClass 类的 print 函数；在第 2 条语句中，执行 derivedClass 类的 print 函数。如果在派生类中重定义了基类的成员函数，重定义函数只适用于派生类的对象。这样，第 1 条语句的输出是：

```
Base Class: u = 5, v = 6, ch = *
```

第 2 条语句的输出是：

```
Base Class: u = 12, v = 76, ch = &
Derived Class: first = 32, second = 16.38
```

13.1.2 基类和派生类的构造函数

派生类有自己的私有数据成员和自己的构造函数。构造函数一般用于初始化数据成员。在声明派生类的对象时，此对象继承了基类的成员。但是，派生类的对象不能直接访问基类的私有成员。同样地，派生类的成员函数也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初始化派生类的私有数据成员。当声明了派生类的对象时，它必须能够自动执行基类的某个构造函数。由于构造函数不能像其他函数一样被调用，所以在执行派生类的构造函数时必须要能够触发基类的某个构造函数的执行。实际上C++也是这样做的。C++在派生类构造函数的函数头中指定调用基类构造函数。

下面，让我们借助于范例加深对上面说明的理解。首先，定义基类和派生类，每个类都有自己的构造函数。

考虑下面的类定义（参见图 13.6）：

```
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;
};
```

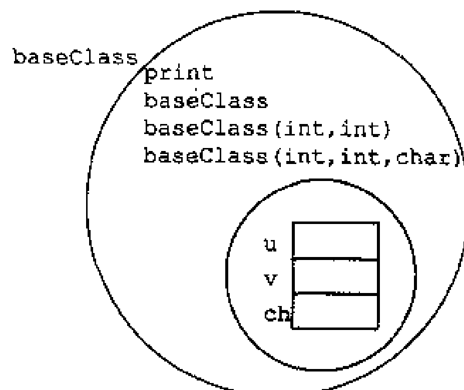


图 13.6 具有构造函数的 baseClass 类

baseClass 类有 3 个构造函数，1 个成员函数 print，3 个数据成员。假设 baseClass 类的成员函数和构造函数的定义如下所示：

```
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;
}
```

现在考虑下面的类定义（参见图 13.7）:

```
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 类从 baseClass 类派生，并且是公有继承，derivedClass 有 5 个数据成员 u，v，ch，first，second。数据成员 u，v，ch 从 baseClass 类中继承。

首先编写 derivedClass 类的 print 成员函数的定义：

```
void derivedClass::print()
{
       baseClass::print();
       cout<<"Derived Class: first = "<<first
           <<", second = "<<second<<endl;
```

```
        cout<<"_______________________________________"
            <<"__________"<<endl;
    }
```

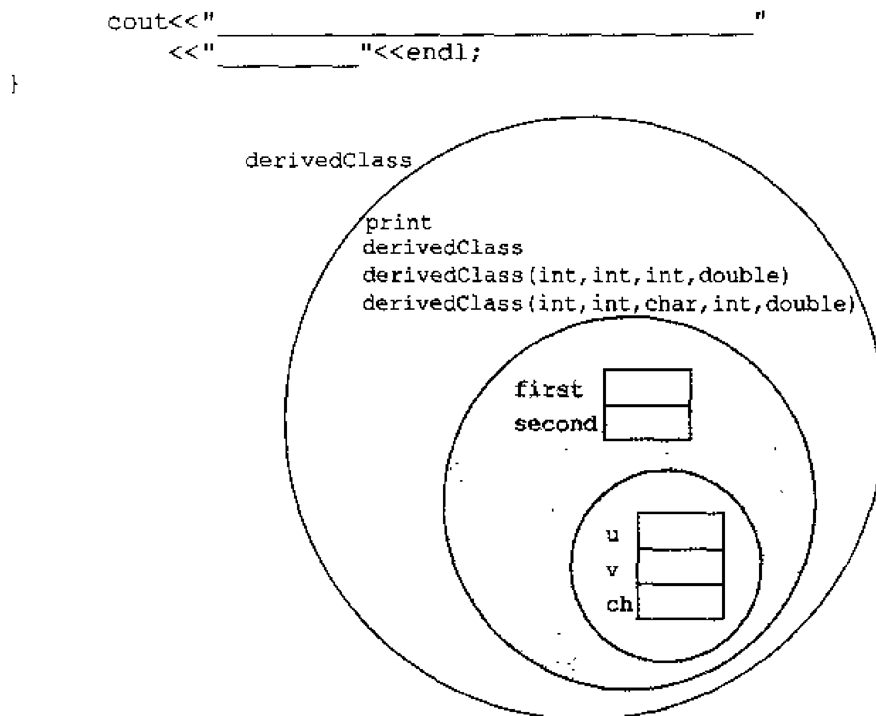


图 13.7 具有构造函数的 derivedClass 类

接着编写derivedClass类构造函数的定义。在派生类构造函数定义的函数头部指定了所调用的基类构造函数。

首先编写derivedClass类默认构造函数的定义。如果派生类中包含默认构造函数而且在对象声明时没有指定值，则执行默认的构造函数初始化对象。因为baseClass类也包含默认构造函数，所以在实现derivedClass类的默认构造函数时，不需要指定基类中的任何构造函数。

```
derivedClass::derivedClass()     //default constructor
{
      first = 0;
      second = 0;
}
```

接着讨论怎样编写带有参数的构造函数定义。为了触发执行基类的带有参数的构造函数，应该在派生类构造函数定义的函数头的部分指定调用的基类构造函数名称。

考虑下面的derivedClass类的构造函数的定义：

```
            //derivedClass constructor Line 3
derivedClass::derivedClass(int x, int y, int one, double two)
       : baseClass(x,y)
{
      first = one;
      second = two;
}
```

在派生类构造函数定义的函数头部分指定了使用baseClass类中带有两个参数的构造函数。当执行derivedClass类的这个构造函数时，将触发执行baseClass类中带有两个int类型参数的构造函数。

接着考虑下面的derivedClass类的构造函数的定义：

```
            //derivedClass; constructor Line 4
derivedClass::derivedClass(int x, int y, char w,
```

```
                                int one, double two)
        :baseClass(x,y,w)
{
        first = one;
        second = two;
}
```

在派生类构造函数定义的函数头部分指定了 baseClass 类中带有 3 个参数的构造函数。当执行 derivedClass 类的这个构造函数时，将触发执行 baseClass 类中带有 3 个参数的构造函数。这 3 个参数的类型分别是 int，int，char。

下面的程序验证了上述概念。

```
#include <iostream>
#include "baseClass.h"
#include "derivedClass.h"
using namespace std;

int main()
{
      baseClass baseObject1;
      baseClass baseObject2(5,6);
      baseClass baseObject3(4,8,'K');

      derivedClass derivedObject1;
      derivedClass derivedObject2(12,13,45,8.5);
      derivedClass derivedObject3(21,23,'Y',76,64.58);

      cout<<"Line 1: baseObject1: ";
      baseObject1.print();                         //main Line 1
      cout<<"Line 2: baseObject2: ";
      baseObject2.print();                         //main Line 2
      cout<<"Line 3: baseObject3: ";
      baseObject3.print();                         //main Line 3

      cout<<"Line 4: -*-*-*-*-*-*-*-*-*-*"
          <<"-*-*-*-*-*-*-*-*-*-*-*-"<<endl;       //main Line 4
      cout<<"Line 5: Derived Class Objects."
          <<endl;                                  //main Line 5
      cout<<endl;

      cout<<"Line 6: derivedObject1: "<<endl;
      derivedObject1.print();                      //main Line 6
      cout<<"Line 7: derivedObject2: "<<endl;
      derivedObject2.print();                      //main Line 7
      cout<<"Line 8: derivedObject3: "<<endl;
      derivedObject3.print();                      //main Line 8

      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
______________________________________________
```

建议读者对上面程序进行代码走查。

注意：假设 baseClass 有私有数据成员和构造函数。derivedClass 类从 baseClass 类派生，而且 derivedClass 类没有数据成员。因此，derivedClass 中的数据成员都继承于 baseClass。构造函数不能像其他函数一样被调用，derivedClass 的成员函数不能直接访问 baseClass 的数据成员。虽然 derivedClass 没有数据成员，但是它必须有相应的构造函数。这样，才能保证 derivedClass 对象能够初始化。derivedClass 的构造函数只是用来触发执行 baseClass 的构造函数。所以，在编写 derivedClass 的构造函数定义时，应该在其函数头部中指定要调用的基类构造函数。注意，derivedClass 构造函数的函数体是空的，只包含一对花括号。

例 13.1 假设需要定义实现员工属性的类。员工分为全职员工和兼职员工。对于兼职员工来说，是根据工作时间和每小时工资数来支付工资。假设要定义类，存储兼职员工的信息，包括：员工名字、每小时工资数、工作时间。该类应该还可以输出员工的名字和工资。因为每个员工都是人，例 12.9（第 12 章）定义了 personType 类存储员工的名字和对名字的相应操作，可以基于 personType 类定义 partTimeEmployee 类（参见图 13.8），还可以重定义输出相应信息的 print 函数。

```
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
};
```

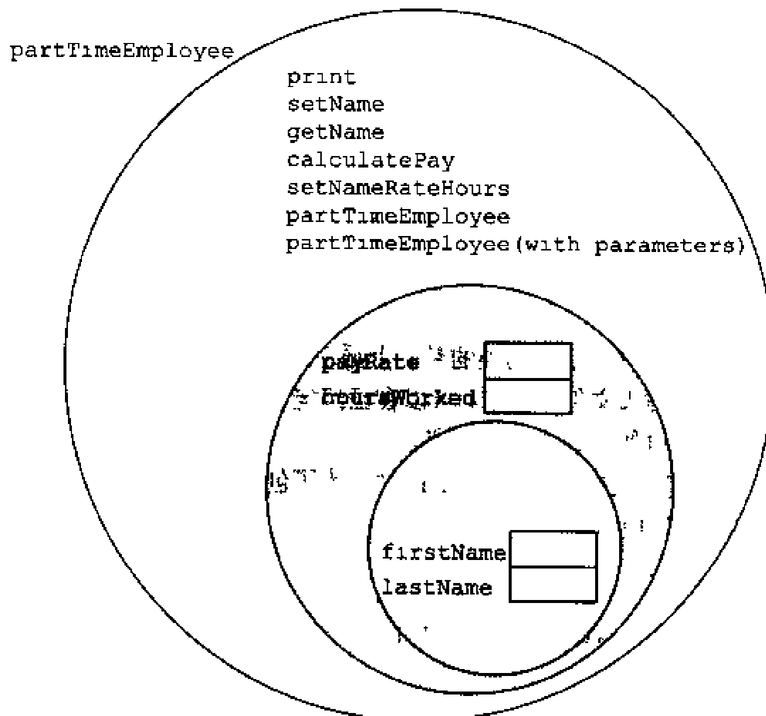


图 13 8　partTimeEmployee 类

partTimeEmployee 类有两个构造函数：默认构造函数和带有参数的构造函数。这两个构造函数都可以被一个带有默认参数的构造函数代替。实现的细节留做练习。

partTimeEmployee 类的成员函数的定义如下所示：

```
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;
}
```

在带有参数的构造函数的定义中，函数头部调用基类中带有参数的构造函数。

```
partTimeEmployee:: partTimeEmployee()   //default constructor
{
    payRate = 0;
    hoursWorked = 0;
}
```

在默认构造函数定义中，因为函数头部不调用基类的构造函数，所以执行基类的默认构造函数。

13.1.3 派生类的头文件

前面章节中介绍了怎样在已经定义的类的基础上派生出新类。为了定义新类,需要创建新的头文件。已经定义好的基类放在基类的头文件中。为了可以在已经定义好的类的基础上派生新类，派生类的头文件中应该包含命令告诉计算机如何查找基类定义。

假设personType类的定义放在头文件person.h中。为了创建partTimeEmployee类，在该类的头文件ptEmployee.h中必须包含预处理命令：

```
#include "person.h"
```

具体地说，头文件ptEmployee.h应该如下所示：

```
//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;
};
```

可以将成员函数的定义放在另一个单独的文件中。如果用户程序包含系统提供的头文件，例如iostream，需要使用尖括号包含头文件。如果包含用户定义的头文件，需要用双引号包含头文件。

13.1.4 头文件的多重包含

前面部分讨论了怎样创建派生类的头文件。为了在程序中包含头文件，需要使用预处理命令。在程序被编译之前，预处理程序首先处理该程序。考虑下面的头文件：

```
//Header file test.h
const int One = 1;
const int Two = 2;
```

假设为了可以使用标识符 One 和 Two，头文件 testA.h 中包含头文件 test.h。具体地说，假如头文件 testA.h 类似于：

```
//Header file testA.h
#include " test.h"
.
.
.
```

考虑下面的程序代码：

```
//Progam headerTest.cpp
#include  "test.h"
#include  "testA.h"
.
.
.
```

当编译 headerTest.cpp 程序时，预处理程序首先处理此程序。预处理程序首先包含头文件 test.h，然后再包含 testA.h。当包含头文件 testA.h 时，因为程序已包含了预处理命令 #include "test.h"，所以程序中的头文件 test.h 被包含了两次。第二次包含头文件 test.h 时会导致编译时错误，例如出现标识符 One 被重复声明的警告信息。之所以出现这样问题，是因为在第一次包含头文件 test.h 时，已经定义了变量 One 和 Two。可以在头文件中使用特定的预处理命令来避免在同一个程序中多次包含同一个文件。下面用预处理命令重新编写头文件 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 的含义是“如果没有定义 H_test”

b. #define H_test 的含义是“定义 H_test”

c. #endif 的含义是“end if ”

H_test 是预处理程序标识符。

这些命令的意思如下：如果标识符 H_test 没有定义，必须首先定义标识符 H_test，并且使得 #define 和 #endif 之间的语句通过编译程序。如果在程序中已经包含了头文件 test.h，语句 #ifndef 失败，跳过 #ifndef 到 #endif 之间的语句。实际上，所有的头文件都是用类似的预处理命令编写的。

13.1.5　C++ 流类

第 3 章详细描述了怎样使用标准输入输出设备对文件进行读写操作。特别是使用对象 cin，析取运算符>>，get 和 ignore 函数从标准输入设备中读取数据；使用对象 cout，插入运算符<<向标准输出设备输出。为了使用 cin 和 cout 对象，程序必须包含头文件 iostream，此文件中包含了 istream 和 ostream 类的定义。此外为了使用文件输入输出，程序必须包含 fstream 头文件，它使用 ifstream 类对象进行文件输入，使用 ofstream 类对象进行文件输出。本节简要描述相关的流类以及在 C++ 中的实现。

在 C++ 中，用图 13.9 所示的继承机制来实现流。

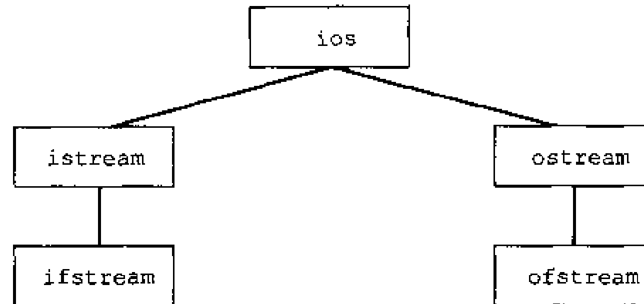


图13.9 C++的流继承

图13.9说明了前面章节所遇到的流类。从图中可以看出，ios类是所有流类的基类，istream和ostream类直接从ios类派生。ifstream类继承于istream类，ofstream类继承于ostream类。此外，用多继承机制，iostream类（不要和头文件iostream混淆）从istream类和ostream类派生，fstream类从ifstream类和ofstream类派生（本书不讨论istream类和fstream类）。

ios类包含了格式化标志和访问及修改这些标志的成员函数。为了确定输入输出状态，ios类包含一个整数状态字，此整数状态字提供流状态的实时报告。

istream和ostream类负责提供在内存和设备之间传输数据的操作。istream类定义了析取运算符>>和get，ignore等函数。ostream类定义了cout对象使用的插入运算符<<。

ifstream类从istream类派生，提供文件输入操作。类似地，ofstream类从ostream类派生，提供文件输出操作。ifstream类对象用于文件输入，ofstream类对象用于文件输出。头文件fstream包含ifstream类和ofstream类的定义。

13.1.6 类的受保护成员

类的私有成员是类私有的，在类的范围之外不能访问类的私有成员。只有类的成员函数可以访问私有成员。前面已经讲过，派生类不能访问基类的私有成员。但是，在有些时候派生类必须访问基类的私有成员。如果把私有成员改成公有成员，那么任何函数都可以访问此成员。类的成员分为三类：公有成员（public）、私有成员（private）和受保护成员（protected）。为了在派生类中访问基类的私有成员，并保证在类的范围之外不能访问类的私有成员，可以使用成员访问说明符protected来声明基类成员。受保护成员的可访问性界于公有成员和私有成员之间。派生类可以直接访问基类的受保护成员。

总之，如果派生类需要访问基类的成员，必须在基类中使用成员访问说明符protected来声明成员。

13.1.7 公有继承、受保护继承和私有继承

假设B类从A类派生，B类不能直接访问A类的私有成员，即A类的私有成员在B类中是隐藏的。B类能访问A类的公有成员和受保护成员吗？本节将给出派生类访问基类成员时应该遵守的一般规则。

考虑下面的语句：

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

在该语句中，成员访问说明符memberAccessSpecifier为public，private或protected。

1. 如果成员访问说明符是public，即公有继承，则：

a. A 的公有成员在 B 中仍然是公有成员，可以在 B 类中直接访问。
b. A 的受保护成员在 B 中是受保护成员，能够被 B 类的成员函数（和友元函数）直接访问。
c. A 的私有成员在 B 中是隐藏的，它们不能被 B 类的成员函数访问，但 B 类的成员函数（和友元函数）可以通过 A 类中的公有成员和受保护成员来访问它们。

2. 如果成员访问说明符是 protected，即受保护继承，则：
a. A 的公有成员在 B 中成为受保护成员，能够被 B 类的成员函数（和友元函数）访问。
b. A 的受保护成员在 B 中仍然是受保护成员，能够被 B 类的成员函数（和友元函数）访问。
c. A 的私有成员在 B 中是隐藏的，它们不能被 B 类的成员函数访问，但 B 类的成员函数（和友元函数）可以通过 A 类中的公有成员和受保护成员来访问它们。

3. 如果成员访问说明符是 private，即私有继承，则：
a. A 的公有成员在 B 中成为私有成员，能够被 B 类的成员函数（和友元函数）访问。
b. A 的受保护成员在 B 中成为私有成员，能够被 B 类的成员函数（和友元函数）访问。
c. A 的私有成员在 B 中是隐藏的，它们不能被 B 类的成员函数访问，但 B 类的成员函数（和友元函数）可以通过 A 类中的公有成员和受保护成员来访问它们。

注意：友元函数将在第 15 章中介绍。

例 13.2 说明了派生类的成员函数可以直接访问基类的受保护成员。

例 13.2　（在派生类中访问受保护成员）

考虑下面的 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 类包含 char 类型的受保护数据成员 bCh，double 类型的私有数据成员 bX。该类定义中还包含重载的成员函数 setData。在这两个成员函数中，一个用于给数据成员 bCh 和 bX 赋值，而另一个只用于给私有数据成员 bX 赋值。bClass 类包含带有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所示：

```
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;
}
```

接着，使用公有继承从 bClass 类派生 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 int 类型的私有数据成员 dA。并包含一个有三个参数的函数 setDate 和函数 print。下面编写函数 setData 的定义。因为 bCh 是 bClass 类的受保护数据成员，所以可以在 setData 函数定义中直接访问。但是由于 bX 是 bClass 类的私有数据成员，所以不能被该类中函数 setData 直接访问。这样，setData 函数必须使用 bClass 类的 setData 函数来给 bX 赋值。dClass 类的 setData 函数的定义如下所示：

```
void dClass::setData(char ch, double v, int a)
{
      bClass::setData(v);

      bCh = ch;     //initialize bCh using the assignment
                    //statement
      dA = a;
}
```

注意函数 setData 调用 bClass 类中带有一个参数的 setData 函数来给数据成员 bX 赋值。然后，直接给变量 bCh 赋值。下面编写 print 函数的定义。

bClass 类有一个 print 函数（它没有像 setData 函数一样进行重载）。我们不想重复打印 bCh 的值。所以，应该首先调用 bClass 类的 print 函数，然后输出 dA 的值。print 函数定义如下所示：

```
void dClass::print() const
{
      bClass::print();

      cout<<"Derived class dA = "<<dA<<endl;
}
```

下面的程序说明了 bClass 和 dClass 类对象的工作过程。假设 bClass 的类定义在头文件 protectMembClass.h 中，dClass 的类定义在头文件 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
```

在编写 dClass 类成员函数的定义时，可以直接访问受保护数据成员 bCh。但是，dClass 类对象不能直接访问 bCh。所以，下面的语句是非法的（实际上是语法错误）:

```
dObject.bCh = '&';    //Illegal
```

13.2　组成

组成（Composition）是在两个类之间建立起联系的另一种方法。在组成中，一个类的一个或多个成员是另一个类的对象。组成是 "has-a" 关系，例如 "每个人都有一个出生日期"。

在例 12.9 中定义了 personType 类，personType 类存储人的姓名。假设想要查找一个人的其他信息，例如：个人 ID 和出生日期等。因为每个人都有个人 ID 和出生日期，所以可以定义一个 personalInfo 类，该类中的一个成员是 personType 类型的对象。可以在 personalInfo 类中添加存储个人 ID 和出生日期信息的成员。

首先定义 dateType 类，该类只存储个人的出生日期。然后，使用 personType 类和 dateType 类来构造 personalInfo 类。这样，就说明了怎样使用两个已经存在的类来定义新类的组成方法。

为了定义 dateType 类，需要三个成员来分别存储月、日、年。数据操作包括设置个人的出生日期和输出出生日期。下面的语句定义了 dateType 类（参见图 13.10）:

```
class dateType
{
public:
    void setDate(int month, int day, int year);
```

```
        //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
};
```

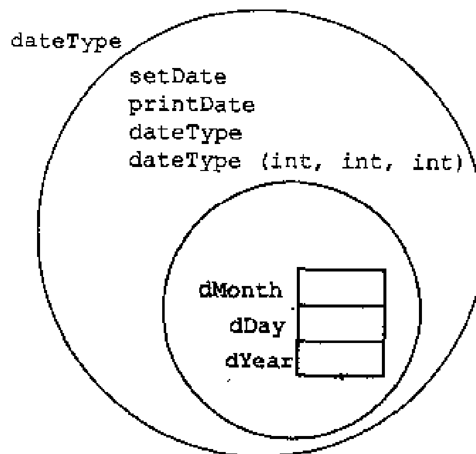


图 13.10 dateType 类

dateType类的定义包含两个构造函数：默认构造函数和带有参数的构造函数。可以使用一个带有默认参数的构造函数来代替这两个构造函数。在本章后面的编程练习 2 中，实现了该替换。

dateType 类的成员函数的定义如下所示：

```
void dateType::setDate(int month, int day, int year)
{
```

```
        dMonth = month;
        dDay = day;
        dYear = year;
}
```

在setDate函数定义中，在将出生日期存储到数据成员之前，没有进行日期的合法性检验。即，没有保证月份界于1和12之间，年份大于0和日的正确（例如，1月的天数是界于1和31之间的）。在本章后面的编程练习2中，需要重新编写setDate函数定义，使之可以在存储出生日期之前、对出生日期进行合法性检验。其他的成员函数定义如下所示：

```
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;
}
```

与setDate函数的情况一样，在编程练习2中，需要重新编写构造函数的定义，使之可以在将日期存储到数据成员之前，增加对月、日、年的合法性检验。

```
dateType:: dateType()  //default parameter
{
      dMonth = 1;
      dDay = 1;
      dYear = 1900;
}
```

下面给出personalInfo类的定义（参见图13.11）：

```
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;
};
```

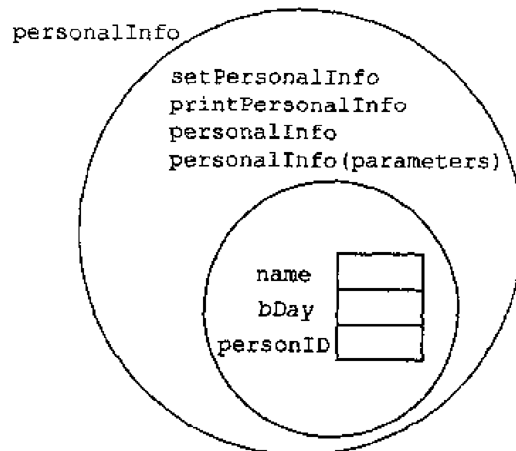


图 13.11 personalInfo 类

在给出 personalInfo 类的成员函数定义之前，首先讨论怎样调用 bDay 和 name 的构造函数。

类对象被创建时，构造函数将自动执行。假设有下面的语句：

```
personalInfo  student;
```

当对象 student 被创建时，它的两个成员对象 bDay 和 name 也同时被创建，并执行各自的某个构造函数。所以在编写 student 类的构造函数定义时，必须要知道怎样向它的成员对象（bDay 和 name）的构造函数传递参数。构造函数没有类型，而且不能像其他函数那样被调用。因此，应该在类的构造函数定义的函数头部指定其成员对象（例如 bDay）构造函数的参数。而且，类的成员对象是以声明的顺序被构造（初始化）的（而不是在执行类的构造函数时，按照在参数列表中的顺序构造的）。并且要注意的是，在类的构造函数执行之前，已经完成其成员对象的构造（初始化）过程。因此，这里先初始化对象 name，然后是对象 bDay，最后才是对象 student。

下面的语句说明了怎样向成员对象的构造函数传递参数：

```
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所示：

```
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;
}
```

因为在上面的代码中，没有在类的默认构造函数的函数头部分向成员对象传递任何参数，所以执行成员对象bDay和name的默认构造函数。

注意：在继承的情况下，使用类的名称调用基类的构造函数；而在组成的情况下，使用成员对象的名字调用自身的构造函数。

13.3 面向对象程序设计（OOD）和面向对象编程（OOP）

在本书的前11章中，使用的是自顶向下的程序设计方法，又称为结构化程序设计方法。问题被划分成模块，每个模块解决整个问题的特定部分。然后，确定所需的数据并编写操作数据的函数。函数和数据是分离的，函数被动地操纵数据，所以结构化程序设计有一定的限制。在结构化程序设计中，函数依赖于数据。而且，设计出来的函数只能使用在问题的某个部分中。在一个程序中编写的函数很难在另一个程序中使用。基于上述原因，对于大型软件的开发来说，结构化程序设计的效率并不高。

第12章引入了类，了解了定义类和实现类的方法。在12章的后面部分，主要讨论了问题的数据需求及数据上的逻辑操作。通过类，可以把数据和数据上的操作组合在一个独立的单元中。也就是说，将数据和操作封装在独立的单元中。借助于类，可以将数据和数据上的操作算法分离开来。但是，实现数据操作的函数可以直接访问数据。本章介绍了通过继承（和组成）在现有的类的基础上创建新的类。此外，对象还具有隐藏实现细节的能力。这些是面向对象程序设计的一些特征。

下面是面向对象程序设计的三个基本特征：

- **封装** 把数据和数据上的操作组合在一个独立单元中的能力

- **继承** 在现有的对象的基础上创建新的对象的能力
- **多态** 使用相同的表达式指定不同操作的能力

在面向对象程序设计中，对象是基本的实体；在结构化程序设计中，函数是基本的实体。在面向对象程序设计中，调试对象；在结构化程序设计中，调试函数。在面向对象程序设计中，程序是相互关联的对象集合；在结构化程序设计中，程序是相互关联的函数的集合。而且，面向对象程序设计鼓励代码复用。因为对象是独立的实体，所以只要一个对象是正确的，就可以在任何程序中重复使用。

在创建对象之前，必须要清楚应该怎样描述数据并编写操作数据的函数。因此，需要有第2章到第9章的预备知识。无论是结构化程序设计还是面向对象程序设计，都需要先掌握前9章的内容。

C++通过类来支持面向对象程序设计。在本章和第12章中我们已经接触到面向对象程序设计的前两个特性：封装和继承。在第15章中，将讨论面向对象程序设计的第三个特性：多态。多态函数或者多态操作有多种形式。

在C++中，函数名称和运算符都可以重载。函数重载的例子是：当函数调用时，通过使用的参数来选用相应的操作。例如，如果两个操作数都是整数，它们的除法运算将产生整数结果；否则，将产生小数结果。假设某个类有多个构造函数。在声明对象时，如果没有向对象传递参数，则执行默认的构造函数；否则，则执行带有参数的构造函数。不过，所有构造函数的名称都相同。

C++还支持参数多态。利用参数多态机制，可以在定义函数时不指定参数类型，而是在使用时实例化。模板（将在第15章中讨论）支持参数多态机制。C++在继承中提供了虚函数来支持多态机制，这样允许运行时选择合适的成员函数（第14章将讨论虚函数）

现在有很多种面向对象语言，包括：Ada，Modula-2，Object，Pascal，Turbo Pascal，Eiffel，C++，Java以及Smalltalk。早期的面向对象语言有1967年开发的Simula语言。很多面向对象的术语都来源于Smalltalk语言。面向对象语言在20世纪70年代的Xerox研究中心得到较大的发展。此后，面向对象语言中又增添了许多“富于想像的”词汇，例如：方法、消息传递等。

从设计过程来看，面向对象方法是一种自然的、直观的方法。当我们考虑一个对象时，立刻会想到它能做什么。例如一提起汽车，就会想到关于汽车的各种操作，例如：开动和驾驶汽车。当程序员考虑表时，通常会想到表的操作，例如：查找、排序、插入。面向对象支持抽象数据类型的创建和使用，在C++中，通过使用类实现抽象数据类型。

在声明类的变量时，创建对象。对象之间通过函数调用交互。每个对象有内部状态和外部状态。对象的私有成员组成了对象的内部状态，对象的公有成员组成了对象的外部状态。只有对象自身才可以操作对象的内部状态。

13.3.1 标识类、对象和操作

本书的前11章，都是在问题的分析阶段分析问题、确定数据、设计算法。通过编写函数来操作数据，来减小函数main的复杂程度。在第12章，开始介绍面向对象技术。首先确定问题所涉及的对象，对象的设计和实现独立于主程序。在面向对象程序设计中，最难的部分是确定对象和类。在这一节中，将介绍确定类和对象的简单方法。

首先描述整个问题，确定所有的名词和动词。从名词列表中选择对象，从动词列表中选择操作。

例如，要编写计算和打印圆柱体体积和表面积的程序。可以将该问题描述如下所示：

Write a **program** to *input* the **dimensions** of a **cylinder** and *calculate* and *print* the **surface area** and **volume**
（*编写***程序***输入***圆柱体**的**尺寸**，*计算*和*打印*圆柱体的**表面积**和**体积**）

在上面的描述中：名词是粗体字，动词是斜体字。从名词：程序、尺寸、圆柱体、表面积和体积中，可以定义圆柱体是cylinderType类。cylinderType类可以创建各种尺寸的圆柱体对象。尺寸、表面积和体积都是圆柱体的属性。

在确定类之后，接下来要确定三件事情：

- 该类对象可以执行的操作
- 可以在该类对象上进行的操作
- 该类对象必须维护的信息

在问题描述的动词中选择该类对象可以执行的操作，或者可以在该类对象上进行的操作。例如在圆柱体的问题描述中，动词有：编写、输入、计算、打印。圆柱体可以执行的操作是：输入、计算和打印。

在cylinderType类中，尺寸数据描述了圆柱体的属性。属性包括：圆柱体底的center（圆心）和radius（半径），以及圆柱体的height（高）。可以通过构造函数或者普通函数对这些属性赋值。

动词计算用于确定圆柱体的体积和表面积。从这里可以定义操作cylinderVolume和cylinder-SurfaceArea。类似地，动词打印用于在输出设备上显示圆柱体的体积和表面积。在本章结束部分的编程练习5中，设计并实现了圆柱体属性的类。

从问题描述中的名词和动词来标识类并不是惟一的面向对象设计技术，还有一些其他面向对象设计技术。然而，该方法对于本书中的编程练习是足够了。

13.4　程序范例：成绩单

此程序范例进一步说明继承和组成的概念。

某地一所大学的期中考试快要到了，教务处要准备学生的成绩单。由于某些学生还没有付清学费，所以：

1. 对于已付学费的学生，成绩单上同时显示成绩和GPA。
2. 对于没有付费的学生，不打印成绩。成绩单上包括由于没有付清学费而保留成绩的信息，并显示需要付的学费。

教务处需要编写程序，分析学生的数据并打印出相应的成绩单。在输入文件中，数据的存储格式如下所示：

```
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行中的数据是招收的学生人数和每个学分的学费。下面是学生的数据。

一个输入的范例是：

```
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行数据说明：招收了3个学生，每个学分的学费是$345。接下来给出了Lisa Miller的课程数据：Lisa Miller的学号是890238，已付了学费，学习了4门课。数学的课程编号是MTH345，该课程有4个学分。她期中考试的成绩是A。

对于每个学生，期望的输出如下所示：

```
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
```

从输出可以看出，课程按照课程编号进行排序。为了计算GPA，假设成绩A是4分，成绩B是3分，成绩C是2分，成绩D是1分，成绩F是0分。

输入　包含前面给定格式数据的文件。假设输入文件的名称是"a:stData.txt "

输出　包含前面给定格式输出的文件

问题分析和算法设计

首先确定程序的主要组成部分：大学里有学生，每个学生学习课程。这样，程序的主要组成部分就是学生和课程。

下面首先描述课程。

课程　课程的主要属性是课程名称、课程编号、课程学分。虽然学生的成绩不是课程的属性，但是为了简便起见，课程中包含学生的课程成绩。

需要对课程类型对象执行的基本操作如下所示：

1. 设置课程信息
2. 打印课程信息
3. 显示课程学分
4. 显示课程编号
5. 显示学生的课程成绩

下面以抽象数据类型定义了课程类（参见图13.12）：

```
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
};
```

下面讨论实现 courseType 类上实现操作的函数的定义。函数 setCourseInfo 按照参数值设置 courseType 类的私有数据成员。它的定义如下所示：

```
void courseType::setCourseInfo(string cName, string cNo,
                               char grade, int credits)
{   courseName = cName;
    courseNo = cNo;
    courseGrade = grade;
    courseCredits = credits;
}
```

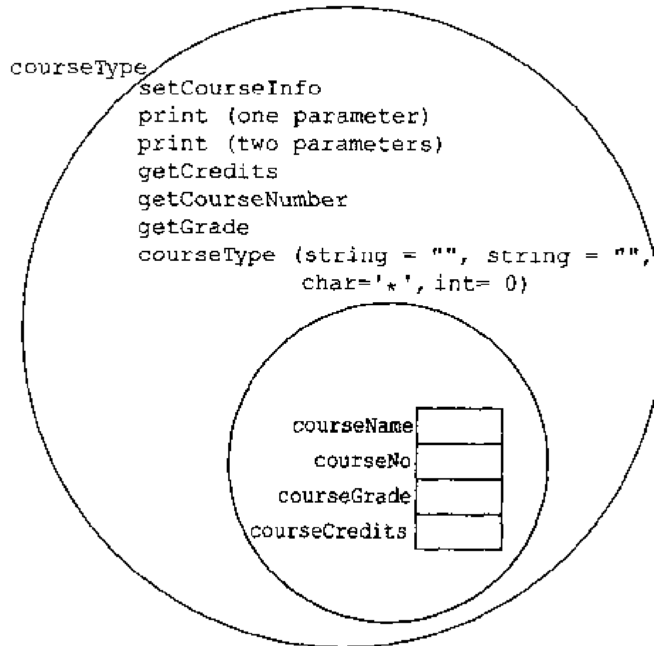


图 13.12　courseType 类

带有一个参数的函数 print 的作用是在屏幕上打印课程信息。如果 bool 类型参数 isGrade 是 true，屏幕上打印课程成绩；否则，打印三个星号。采用左对齐的方式，而不是默认的右对齐的方式打印课程名称和课程编号。因此，需要使用控制符 left。在打印成绩和课程学分之前，需要取消该控制符。下面的步骤描述了该函数的实现：

1. 设置控制符 left
2. 打印课程编号
3. 打印课程名称
4. 取消控制符 left
5. 打印课程学分
6. 如果 isGrade 是 true：
　　输出课程成绩
　否则
　　输出三个星号

print 函数的定义如下所示：

```
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;
}
```

带有两个参数的函数 print 的作用是将课程信息发送到文件中，文件作为该函数的参数。这个函数与前面定义的函数 print 的定义相同。该函数 print 的定义如下所示：

```
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;
}
```

定义的构造函数带有默认值。如果在声明courseType类对象时没有指定参数值，构造函数用默认值初始化对象。使用默认值将对象的数据成员初始化如下：课程编号初始化为空格，课程名称初始化为空格，课程成绩初始化为 *，课程学分初始化为 0。如果 courseType 对象声明时指定了值，则用指定值初始化对象。构造函数的定义如下所示：

```
courseType::courseType(string cName, string cNo,
                       char grade, int credits)
{
      courseName = cName;
      courseNo =  cNo;
      courseGrade = grade;
      courseCredits = credits;
}
```

其他的函数定义如下所示：

```
int courseType::getCredits()
{
      return courseCredits;
}

char courseType::getGrade()
{
      return courseGrade;
}

void courseType::getCourseNumber(string& cNo)
{
      cNo = courseNo;
}
```

下面描述学生部分。

学生　学生的主要属性是学生姓名、学生学号、选择的课程数目、选择的课程、每门课程的成绩。因为每个学生都需要付学费，学生属性中还应该包含说明是否付清学费的数据成员。

每个学生都是人，而且每个学生都选择课程。我们已经定义了 personType 类处理人的姓名。而且，前面还设计了处理课程信息的类。从personType类可以得到说明学生信息的studentType类，studentType 类的一个成员是 courseType 类型的。根据需要，可以添加更多的成员函数。

对studentType类的对象进行的基本操作如下所示：

1. 设置学生信息
2. 打印学生信息
3. 计算总的学分
4. 计算GPA
5. 计算需要付的学费
6. 因为成绩单需要以升序打印课程，按照课程编号排列课程

下面抽象数据类型定义了studentType类。假设学生每个学期最多只能选6门课（参见图13.13）：

```
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
};
```

studentType
print
setName
getName
setInfo
print (one parameter)
print (two parameters)
getHoursEnrolled
getGpa
billingAmount
studentType
sortCourses
sID
numberOfCourses
isTuitionPaid
coursesEnrolled
firstName
lastName

图 13.13　studentType 类

注意用于对coursesEnrolled数组排序的成员函数sortCourse是studentType类的私有成员函数。这是由于此函数只访问内部数据，而且用户不需要访问此成员函数。

下面讨论 studentType 类中实现操作的函数定义。

函数 setInfo 使用参数初始化 studentType 类的私有数据成员。该函数调用函数 sortCourses 按照课程编号对coursesEnrolled数组排序。studentType类从personType类派生，存储学生姓名的变量是personType类的私有数据成员。所以，调用 personType 类的 setName 成员函数，通过向此函数传递相应的参数来初始化学生姓名。函数 setInfo 的定义如下所示：

```
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
}
```

默认的构造函数把私有数据成员初始化为默认值。注意私有数据成员coursesEnrolled是courseType类型的，而且是数组，所以自动执行courseType类的默认构造函数，初始化整个数组。

```
studentType::studentType()
{
        numberOfCourses = 0;
        sId = 0;
        isTuitionPaid = false;
}
```

带有一个参数的函数print的作用是在屏幕上打印成绩单。如果学生已经付清了学费，屏幕上将显示成绩和GPA。如果学生没有付清学费，在成绩的位置显示三个星号，不显示GPA，并说明由于没有付清学费而保留成绩的信息，以及需要付的学费。实现该函数。需要的步骤如下所示：

1. 输出学生的名字
2. 输出学生的学号
3. 输出学生选择的课程数目
4. 输出标题：CourseNo　CourseName　Credits　Grade
5. 打印每门课程的信息
6. 打印总学分
7. 以带有小数点和小数尾部的0，固定小数点格式输出GPA和需要支付的学费，输出的小数保留两位精度
8. 如果isTuitionPaid是true：
   输出GPA
   否则
   输出需要付的学费和保留成绩的信息

```
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;
}
```

带有两个参数的函数print用于向文件输出信息。该函数的大部分与只有一个参数的print函数相同。因为personType类没有向文件发送输出的函数，所以首先使用personType类的成员函数getName返回姓名，然后打印姓名。该函数的定义和前面的print函数的定义相同：

```
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
```

```
        {
            outp<<"*** Grades are being held for not paying "
                <<"the tuition. ***"<<endl;
            outp<<"Amount Due: $"<<billingAmount(tuitionRate)
                <<endl;
        }

        outp<<"-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*"
            <<"-*-*-*-*-"<<endl<<endl;
}
```

函数getHoursEnrolled的作用是计算和返回学生选课的总学分。课程的学分用来计算GPA和支付的学费。该函数根据学生选择的每门课程学分计算出总学分。因为课程的学分是courseType类对象的私有数据成员，需要使用courseType类的成员函数getCredits计算总学分。该函数的定义如下所示：

```
int studentType::getHoursEnrolled()
{
        int totalCredits = 0;
        int i;

        for(i = 0; i < numberOfCourses; i++)
            totalCredits += coursesEnrolled[i].getCredits();

        return totalCredits;
}
```

如果学生没有付清学费，函数billingAmount根据选课的总学分计算并返回需要支付的学费。该函数的定义如下所示：

```
double studentType::billingAmount(double tuitionRate)
{
        return tuitionRate * getHoursEnrolled();
}
```

下面讨论函数getGpa，该函数的作用是计算学生的GPA。GPA的计算方法为累加所选课程成绩相对应的分数与相应课程学分的乘积，然后除以选择课程的总学分。该函数的定义如下所示：

```
double studentType::getGpa()
{
    int i;
    double sum = 0.0;

    for(i = 0; i < numberOfCourses; i++)
    {
        switch(coursesEnrolled[i].getGrade())
        {
        case 'A': sum += coursesEnrolled[i].getCredits() * 4;
                  break;
        case 'B': sum += coursesEnrolled[i].getCredits() * 3;
                  break;
        case 'C': sum += coursesEnrolled[i].getCredits() * 2;
                  break;
        case 'D': sum += coursesEnrolled[i].getCredits() * 1;
                  break;
        case 'F': sum += coursesEnrolled[i].getCredits() * 0;
                  break;
        default: cout<<"Invalid Course Grade"<<endl;
```

```
        }
    }

    return sum / getHoursEnrolled();
}
```

函数sortCourses按照课程编号对coursesEnrolled数组排序。这里使用选择排序的算法。由于需要比较课程编号，课程编号是string类型，而且是courseType类的私有数据成员，首先把课程编号存储在局部变量中。该函数的定义如下所示：

```
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

已经设计了courseType类和studentType类，下面使用这两个类来编写程序。

限制程序最多只能处理10个学生。注意，该程序稍加修改就可以处理任意多个数目学生的成绩。

因为函数print为了打印最终的成绩单，已经做了必要的计算。所以，函数main只需要做少量的事情。实际上，函数main只需要声明存储学生数据的对象，向对象输入数据，打印成绩单。因为输入存储在文件中，而且输出需要发送到文件中，所以使用流变量访问输入输出文件。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，退出程序
4. 打开输出文件
5. 得到登记的学生数目和每个学分的学费
6. 加载学生数据
7. 打印成绩单

变量 程序最多只能处理10个学生，所以需要定义用来存储学生数据的一个有10个元素studentType类型数组。该程序还要存储登记的学生数目和每个学分的学费。因为需要从文件中读取数据并将输出存储到文件中，所以需要定义两个流变量来访问输入和输出文件。这样，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
```

函数 getStudentData 函数有三个参数，一个参数指定输入文件，一个参数指定studnetList数组，另一个参数指定登记的学生数目。该函数定义的伪代码形式如下所示：

1. 读入学生的姓名、学生学号、是否付了学费。
2. 如果isPaid是'Y'
   设置isTuitionPaid为true
   否则
   设置isTuitionPaid为false
3. 读入学生选课的数目。
4. 对于每门课程：
   读入课程名称、课程编号、课程学分、课程成绩
   将课程信息存储到courseType类型对象中
5. 将数据存储到studentType类型对象中。

需要几个局部变量读取和存储数据，函数getStudentData的定义如下所示：

```
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
}
```

函数printGradeReports　该函数的作用是打印成绩单。对于每个学生，调用studentType类的print函数打印成绩单。函数 printGradeReports 的定义如下所示：

```
void printGradeReports(ofstream& outfile,
                       studentType studentList[],
                       int numberOfStudents,
                       double tuitionRate)
{
    int count;

    for(count = 0; count < numberOfStudents; count++)
        studentList[ count] .print(outfile,tuitionRate);
}
```

完整的程序代码清单

```
//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

```
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
```

13.5 小结

1. 继承和组成是关联两个类或多个类的有效方法。
2. 继承是"is-a"关系。
3. 组成是"has-a"关系。
4. 在单继承中，派生类仅从一个现有类派生，此现有类称为基类。
5. 在多继承中，派生类从多个基类派生。
6. 基类的私有成员是基类私有的，派生类不能访问基类的私有成员。
7. 基类的公有成员在派生类中可以是公有成员，或者私有成员。
8. 派生类可以重定义基类的成员函数，但是重定义仅适用于派生类的对象。
9. 可以在派生类构造函数的函数头部中指定调用基类的构造函数。
10. 当初始化派生类的对象时，首先执行基类的构造函数。
11. 回顾本章给出的继承规则。
12. 在组成中，类的成员是另一个类的对象。
13. 在组成中，可以在类的构造函数的函数头部中指定调用的成员对象构造函数。
14. 面向对象程序设计的三个基本特征是：封装、继承和多态。
15. 标识类、对象、操作的一个简单的方法是从所描述的问题中，确定名词和动词。然后从名词中选择类，从动词中选择操作。

13.6 练习

1. 判断下面说法的正误。
   a. 可以在派生类构造函数的函数头部中指定调用的基类构造函数。
   b. 派生类的构造函数用类的名称调用基类的构造函数。
   c. 假设x，y是类。x的一个数据成员是y类的对象，x和y都有构造函数。在x的构造函数中使用y类对象名字来调用y类的构造函数。
   d. 派生类必须具有构造函数。

2. 描绘一个多个类继承于一个类的继承关系图。
3. 假设 employeeType 类从 personType 类派生（参见例 13.9）。试着在 employeeType 类中添加几个数据成员和成员函数。
4. 说明类的私有成员和受保护成员的不同点。
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类的私有成员是xClass的公有成员。正确还是错误？

b. 判断下面说法的正误。如果错误，请说明原因。

(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. 假设练习6中的声明不变。

a. 编写yClass类默认构造函数的定义，该函数将yClass类的私有数据成员初始化为0。

b. 编写xClass类默认构造函数的定义，该函数将xClass类的私有数据成员初始化为0。

c. 编写yClass类成员函数two的定义，该函数将私有数据成员a初始化为two函数的第1个参数值，私有数据成员b初始化为two函数的第2个参数值。

8. 下面的代码有什么错误？

```
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. 考虑下面的代码:

```
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. 编写 two 类函数 setData 的定义。

b. 编写 two 类函数 print 的定义。

10. 下面 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. 下面C++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;
}
```

13.7 编程练习

1. 在第12章中，定义了实现一天中时间的类clockType。但是，对于某些应用程序，除了存储小时、分钟、秒以外，还需要存储时区。从clockType类派生出extClockType类，添加存储时区的成员变量，并添加相应的成员函数和构造函数。最后，编写用于测试extClockType类的程序。
2. 在本章中，设计了实现日期的dateType类。但是在其成员函数setDate和带有参数的构造函数中，在将日期存储到数据成员之前，并没有对日期进行合法性检查。重新编写成员函数setDate和构造函数的定义，使其在将日期存储到数据成员之前，对月、日、年进行合法性检查。并使用带有默认参数的构造函数来代替默认的构造函数和带有参数的构造函数。并添加成员函数isLeapYear，该函数用于检验给定年份是否为润年，并且编写一个测试程序来测试修改过的类。
3. x-y平面上的任意点可以用x坐标和y坐标来表示。设计一个pointType类，使其能够存储和处理x-y平面上的点。在点上可以执行的操作有：显示点、设置点的坐标、打印点的坐标、返回点的x坐标、返回点的y坐标。同时编写程序来测试点上的各种操作。
4. 每个圆都有圆心和半径。给定半径，可以确定圆的面积和周长；给定圆心，可以在x-y平面上确定圆的位置。圆心是x-y平面上的一个点。设计circleType类可以存储圆心和半径。因为圆心是x-y平面上的一个点，在编程练习3中已设计了获取点属性的类。所以可以从pointType类派生circleType类。在圆上可以执行的操作有：设置圆的半径、输出圆的半径、计算和输出圆的面积和周长，并且实现对圆心的操作。
5. 每个圆柱体都有底和高，圆柱体的底面是一个圆。设计cylinderType类，可以获得圆柱体的属性并且可以执行对圆柱体的一些操作。从编程练习4设计的circleType类派生cylinderType类。在圆柱体上可以执行的操作有：计算和输出圆柱体的体积、计算和输出圆柱体的表面积、设置圆柱体的高，设置圆柱体的底面半径、设置圆柱体的底面圆心。
6. 可以使用类来实现存储家庭成员、朋友和商业伙伴信息的通信录。可以在通信录上存储的信息包括：名字、地址、电话号码、出生日期。该程序最多只能存储500条信息。
   a. 定义能够存储街道地址、城市、省、邮政编码的addressType类。使用合适的函数来存储、输出地址。使用构造函数自动初始化数据成员。
   b. 使用personType类、dateType类、addressType类定义extPersonType类。为extPersonType类添加数据成员，把通信录中人员分为家庭成员、朋友和商业伙伴三类。添加存储电话号码的数据成员。添加（或者重载）输出和存储适当信息的函数。使用构造函数自动初始化数据成员。
   c. 使用前面定义的类，定义addressBookType类。该类对象最多可以处理500条信息。程序能够执行下面的操作：
      (i) 从磁盘向通信录装载数据
      (ii) 按照姓对通信录排序
      (iii) 按照姓查找人
      (iv) 输出指定人的地址、电话号码、出生日期等信息
      (v) 输出在指定两个日期之间出生的所有人的名字
      (vi) 输出在两个姓之间的所有人的名字

(vii)按照用户需求，输出所有家庭成员、朋友和商业伙伴的名字

7. 在编程练习2中定义和实现的存储日期的dateType类只可以进行有限的操作。重新定义dateType类，使其还可以完成以下的操作：
   a. 设置月
   b. 设置日
   c. 设置年
   d. 返回月
   e. 返回日
   f. 返回年
   g. 判断某年是否是闰年
   h. 返回某个月的天数。例如，如果日期是3-12-2002，返回的天数是31，因为3月有31天
   i. 返回某年中已过的天数。例如，如果日期是3-18-2002，2002年已经度过的天数是77天。注意返回的天数包含当前天
   j. 返回某年剩下的天数。例如，如果日期是3-18-2002，2002年剩下的天数是288天
   k. 增加日期加上一个固定天数的功能。例如，如果日期是3-18-2002，增加25天，新的日期是4-12-2002
8. 编写编程练习7中实现dateType类新增操作的函数定义。
9. 编程练习7中定义的dateType类以数字的形式输出日期。某些应用程序需要以其他形式输出日期，例如：March 24,2002。派生一个extDateType类能够以该形式输出日期。向extDateType类添加数据成员使月份可以以字符串形式存储。添加成员函数，使月份以字符串形式输出并后跟年份，例如：March 2002。编写extDateType类中实现该操作的函数定义。
10. 使用extDateType类（编程练习9）和dayType类（第12章中的编程练习2）定义calendarType类。对于给定的年份和月份，可以输出该月的月历。为了输出月历，需要知道月份的第一天为星期几和该月的总天数。这样，需要用dayType类存储月份的第一天。很明显，可以使用extDateType的类对象来存储日期，其中月份和年份由用户输入，而天被设置为1。这样，calenderType类有两个数据成员：dayType类对象，extDateType类对象。

    设计calendarType类，使程序能够输出从1500年1月1日开始的任何月份的日历。注意1500年1月1日是星期一。为了计算某个月份的第一天是星期几，需要在1500年1月1日上面增加相应的天数。

    calendarType类包含以下的操作：
    a. 调用firstDayOfMonth函数，确定要打印的月份的第一天
    b. 设置月份
    c. 设置年份
    d. 返回月份
    e. 返回年份
    f. 输出特定月份的日历
    g. 添加适当的构造函数，初始化数据成员
11. a. 编写实现calendarType类（设计参考编程练习10）中操作的成员函数定义。
    b. 编写测试程序，输出指定年月的日历。例如，September 2002的日历是：

```
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14章 指针、类、表及虚函数

本章要点：

- 了解指针数据类型和指针变量
- 理解声明和使用指针变量的方法
- 了解地址和递引用（Dereference）运算符
- 认识动态变量
- 理解怎样用 new 和 delete 运算符来生成动态变量
- 了解指针运算
- 认识动态数组
- 了解数据的浅（Shallow）拷贝和深（Deep）拷贝
- 认识带有指针数据成员的类的特性
- 理解怎样使用动态数组来处理表
- 了解虚函数
- 理解地址运算符与类之间的关系

在第2章中，C++的数据类型分成了三类：简单数据类型、构造数据类型及指针数据类型。到目前为止，已经介绍了前两种数据类型。本章要讨论的是第三种数据类型，也就是指针数据类型。首先，要了解怎样声明指针变量（简称为指针）及通过指针来操作数据。在说明动态数组及表时将会用到这些概念。表将在16章中讨论。

14.1 指针数据类型与指针变量

指针数据类型的值是计算机内存的地址。像其他很多种语言一样，C++中指针类型数据没有名字，这是因为指针数据类型的值是地址（内存位置）。指针变量的内容是地址，也就是内存地址。

指针变量 变量的内容是地址——内存地址。

14.1.1 声明指针变量

因为指针类型数据没有名字，所以指针变量与其他变量的声明不太一样。声明指针变量时，也就确定了内存单元中指针所指的值的数据类型。C++中，通过在数据类型与变量名之间使用星号（*）来声明指针变量。声明指针变量的语法为：

```
dataType *identifier;
```

例如下面的语句：

```
int *p;
char *ch;
```

上面语句中p和ch都是指针变量。p（在正确赋值后）的内容（地址）所指内存单元的值的类型为int，即p所指内存单元的值的类型是int，ch所指内存单元的值的类型是char。一般来说，p称为*int*型指针变量，ch称为char型指针变量。

在讨论指针是怎样工作之前，先考虑下面三个语句：

```
int *p;
int* p;
int * p;
```

的作用完全一样。

即，只要星号（*）出现在数据类型名与变量名之间即可。考虑下面语句：

```
int* p,q;
```

此语句中只有p是指针变量，而q不是，q是int变量。为避免混淆，字符*最好靠近变量名，上面语句可写为：

```
int *p,q;
```

当然，语句：

```
int *p,*q;
```

将p和q都声明成int型指针变量。

到现在为止，你已知道怎样声明指针。下面我们将讨论怎样将指针指向内存空间，及怎样操作内存单元中的数据。

由于指针的值是内存地址，所以只要指定了内存空间存储的数据的类型，就可以使用指针存储其内存空间的地址。例如，如果p是int类型的指针，它就能存储int类型内存空间的任何地址。C++提供了两种指针运算符——取地址运算符（&）和递引用运算符（*）。下面两节将说明这两种运算符。

14.2　取地址运算符

以C++中，&称为取地址运算符（Address of Operator）。取地址运算符是单目运算符，返回操作数的地址。例如，考虑下面语句：

```
int x;
int *p;
```

语句：

```
p = &x;
```

将x的地址赋给p，也就是说，x与p所指向的内存单元为同一个内存单元。

14.3　递引用运算符

在此之前的各章节中都将星号（*）作为双目乘法运算符。在C++中，*也可作为单目运算符。在作为单目运算符时，*被称为递引用运算符（Dereferencing Operator）或间接运算符（Indirection Operator）。例如，下面的语句：

```
int x = 25;
int *p;
p=&x;     //store the address of x in p
```

语句：

```
cout<<*p<<endl;
```

输出p所指内存单元所存储的值，即x的值。语句：

```
*p=55;
```

将 55 存储到 p 所指的内存单元中，即 x 中。

考虑下面语句：

```
int *p;
int num;
```

在上面的语句中，p 是 int 型指针变量，num 是 int 型变量，如图 14.1 所示。

假设内存地址为 1200 的内存单元分配给 p，内存地址 1800 分配给 num，语句：

```
num = 78;
```

将 78 存储在 num 中——也即是内存单元 1800 中，如图 14.2 所示。

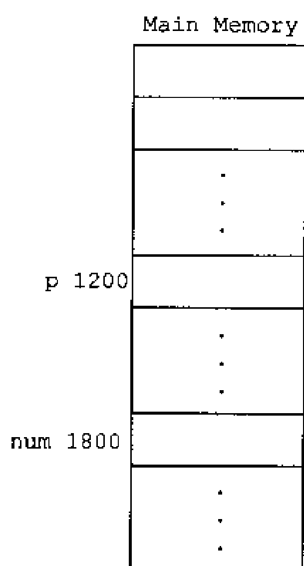


图 14.1　内存中的 p 与 num

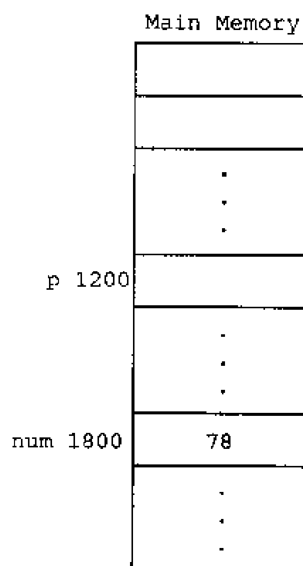


图 14.2　语句"num = 78;"执行后的 num

语句：

```
p = &num;
```

将 num 的地址（也就是 1800）存储到 p 中。此语句执行后，*p 和 num 都指向内存单元 1800 中的内容——也就是 num，如图 14.3 所示。

赋值语句：

```
*p=24;
```

改变了内存单元 1800 中的内容，当然也改变了 num 的内容。如图 14.4 所示。

综上所述：

1. &p，p 和 *p 的含义不同。
2. &p 表示 p 的地址——即 1200（如图 14.4 所示）。
3. p 表示 p 的内容（图 14.4 中是 1800）。
4. *p 表示 p 所指（即 p 中的地址）的内存单元（图 14.4 中是 1800）的内容（图 14.4 中是 24）。

例 14.1　考虑下面语句：

```
int *p;
int  x;
```

假设p和x的内存分配如图14.5所示。

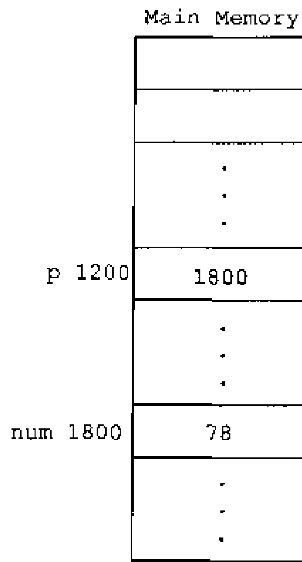


图14.3 "p=#"执行后的p

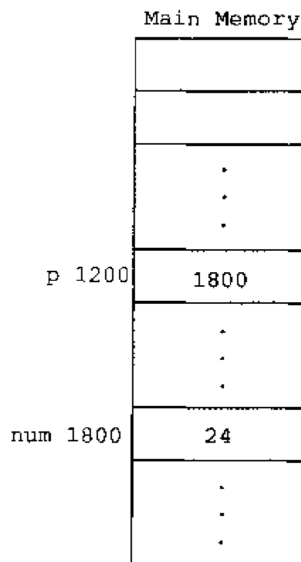


图14.4 "*p=24;"执行后的*p与num

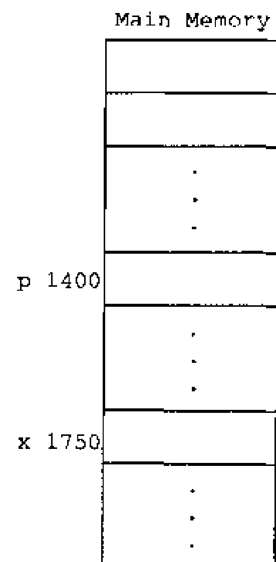


图14.5 内存中的p和x

&p, p, *p, &x和x的值如下所示：

	值
&p	1400
p	???（未知）
*p	不存在（未定义）
&x	1750
x	???（未知）

假设语句按如下顺序执行：

```
x = 50;
p=&x;
*p=38;
```

在每条语句执行之后，都将给出&p, p, *p, &x及x中的值。语句：

```
x = 50;
```

执行后，&p, p, *p, &x及x中的值是：

	值
&p	1400
p	???（未知）
*p	不存在（未定义）
&x	1750
x	50

语句：

```
p = &x;
```

执行后，&p, p, *p, &x及x中的值是：

	值
&p	1400
p	1750
*p	50
&x	1750
x	50

注意，在语句“p = &x;”执行后，p 中存储了 x 的地址，*p 和 x 为相同的内存单元，也就是 x。因此 *p 的值是 50。

语句：

```
*p=38;
```

执行后，&p，p，*p，&x 及 x 中的值是（由于 *p 和 x 引用相同的内存空间，x 的值也变为 38）：

	值
&p	1400
p	1750
*p	38
&x	1750
x	38

应该注意例 14.1 中的如下内容：

1. 声明：

```
int *p;
```

仅给 p 而不是 *p 分配内存单元，下面将讨论怎样给 *p 分配内存单元。

2. 假设有下面的语句：

```
int *p;
int x;
```

那么：

a. p 是指针变量

b. p 所指的内存单元的内容的类型是 int 型

c. 内存单元 x 存在并且是 int 型，因此赋值语句：

```
p = &x;
```

是合法的。此赋值语句执行后，*p 才有意义。

例 14.2　下面的程序说明了指针变量是怎样工作的：

```
//Chapter 14: Example 14-2

#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;
}
```

程序运行结果

```
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
```

上述程序运行情况如下：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。由于p存储了x的地址，*p与x的值是一样的，如第3行的输出所示。第4行中的语句将*p的值改为58，第5行中的语句输出*p与x的值，它们仍然相同。第6行与第10行之间的语句输出了p的地址、p的值、*p的值、x的地址以及x的值。注意p的值与x的地址一样，因为在第2行的语句中，x的地址存储在p中（注意在第6行、第7行和第9行中输出的p的地址、p的值及x的地址都与机器有关。如果在自己的机器上运行这个程序，可能会得到不同的值）。

14.4 类、结构及指针变量

在前面的章节中，已经介绍了声明及使用简单数据类型（如int和char）指针变量的方法。也可将指针声明为指向其他数据类型，例如类。现在学习类和结构的指针（回想一下，类和结构具有相同的功能，它们的不同之处仅仅在于：默认情况下，类的成员是private，结构的成员是public。因此，下面的讨论对二者都适用）。

考虑下面的结构声明：

```
struct studentType
{
    char name[26];
    double gpa;
    int sID;
    char grade;
```

```
};

studentType    student;
studentType    *studentPtr;
```

在上面的声明中，student是studentType类型的对象，studentPtr是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存储在student对象的gpa成员中：

```
(*studentPtr).gpa=3.9;
```

表达式(*studentPtr).gpa结合使用了指针递引用和类成员访问运算符。在C++中，点运算符的优先级比递引用要高。因此，括号就显得十分重要。为了简化通过指针访问类和结构的成员，C++提供了另外一种成员访问运算符 ->。运算符 -> 包含了两个相连的符号：连字符和大于号。

使用运算符 -> 访问类（结构）成员的语法是：

```
pointerVariableName->classMemberName
```

因此，语句：

```
(*studentPtr).gpa=3.9;
```

与语句：

```
studentPtr->gpa = 3.9;
```

相同。

通过指针使用->运算符访问类（结构）的成员，可以避免使用括号和递引用运算符。由于错误很难避免，而漏掉括号会导致程序异常终止或得出错误的结果。在本书中，通过指针访问类（结构）成员时使用箭头符号。

例14.3说明了怎样通过指针操作成员函数。

例 14.3 *考虑下面的类：*

```
class classExample
{
public:
    void setX(int a);
      //Function to set the value of the data member x
      //Post: x = a;
    void print() const;
      //Function to output the value of x

private:
    int x;
};
```

成员函数的定义如下所示：

```
void classExample::setX(int a)
{
     x = a;
}
```

```
void classExample::print() const
{
     cout<<"x = "<<x<<endl;
}
```

考虑下面的函数 main:

```
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 行中的语句将 cExpPtr 声明为 classExample 类型指针，第 2 行中的语句将 cExpObject 声明为 classExample 型对象，第 3 行中的语句将 cExpObject 的地址存储到 cExpPtr 中（如图 14.6 所示）。

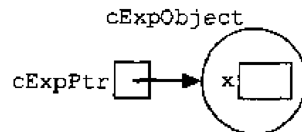


图 14.6 “cExpPtr = &cExpObject;”执行后的 cExpObjext 和 cExpPtr

在第 4 行语句中，使用指针 cExpPtr 访问成员函数 setX 来设置数据成员 x 的值（如图 14.7 所示）。

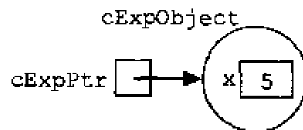


图 14.7 “cExpPtr->setX(5);”执行后的 cExpObject 与 cExpPtr

在第 5 行语句中，使用指针 cExpPtr 访问成员函数 print 来输出 x 的值，如上所示。

14.5 初始化指针变量

C++不自动初始化变量，如果你不想指针变量乱指，就必须对其初始化。若指针变量用常量值 0 初始化，则称为空指针（Null Pointer）。因此语句 p = 0;将空指针存储到 p 中。也就是说，p 不指向任何对象。有些程序员用命名常量 NULL 来初始化指针变量。下面两句完全一样：

```
p = NULL;
p = 0;
```

注意：只有 0 能直接赋给指针变量。

14.6 动态变量

在前面的章节中，已经学习了怎样声明指针变量，怎样在指针变量中存储变量地址，以及怎样使用指针操作数据。但是，你所知道的仅仅是使用指针操作由其他变量获得的内存空间。也就是说，指针操作的是已经存在的内存空间的数据。既然使用变量同样可以访问这些内存空间，那么使用指针有什么好处呢？在本节中，将进一步说明指针的作用，尤其是在程序执行期间，使用指针分配与释放内存。

程序运行时创建的变量称为动态变量(Dynamic Variable)。在C++中，可以使用指针创建动态变量。C++提供了new和delete两种运算符分别创建、删除动态变量。当程序需要新的变量时，使用new创建；当程序中不再需要某个动态变量时，使用delete删除。

在C++中，new与delete都是保留字。

运算符new有两种形式：一种是分配单个变量，另一种是分配变量数组。运算符new的使用语法如下所示：

```
new dataType;                //to allocate a single variable
new dataType[ intExp] ;      //to allocate an array of variables
```

其中，intExp可以是任何正整数类型的表达式。运算符new分配指定类型变量的内存，并返回其指针——也就是所分配内存的地址。注意，所分配的内存未初始化。

考虑下面的声明：

```
int  *p;
char *q;
int  x;
```

语句：

```
p = &x;
```

只是将x的地址存储在p中，而并不分配新的内存。再考虑下面语句：

```
p = new int;
```

在程序运行期间，此语句在内存中创建了新的变量，并将所分配的内存地址存放在p中。可通过指针递引用——*p，来访问该内存。同样，语句：

```
q = new char[ 16] ;
```

创建了有16个成员的char型数组，并将数组的起始地址存放在q中。

由于动态变量未命名，不能直接对它们进行访问，只能通过由new返回的指针对它们间接访问。下面的语句说明了这个概念：

```
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
                        //memory pointed to by str
```

若不再需要某个动态变量，可以将它删除，即释放其内存空间。可以使用C++运算符delete来删除动态变量，运算符delete的使用语法有两种形式：

```
delete pointerVariable;          //to destory a single dynamic variable
detele [] pointerVariable;       //to destory a dynamically created array
```

因此，语句：

```
delete p;
delete [] name;
```

的作用是释放由p和name指定的内存空间。

14.7 指针变量的运算

可以在指针变量上进行的运算是赋值及关系运算和一些受限的算术运算。指针变量的值可赋给同类型的另一指针变量，两个同类型的指针可以比较是否相等，等等。指针变量可以加上或减去某个整数值，一个指针变量的值可以减去另一个指针变量的值。

例如，有下面的语句：

```
int *p,*q;
```

语句：

```
p = q;
```

将q的值拷贝到p中。在该语句执行后，p和q都指向同一内存单元，对*p的任何改动都会影响到*q中的值，反之亦然。

如果p和q有相同的值，即它们指向同一内存单元，则表达式：

```
p == q
```

的值是true。同样，如果p与q指向不同的内存单元，则表达式：

```
p != q
```

的值是true。

指针变量上的算术运算不同于数字类型数据上的算术运算，下面的语句说明了指针变量的增量和减量运算：

```
int *p;
double *q;
char *chPtr;
studentType *stdPtr;  //studentType is as defined before
```

前面已经讲过，系统一般为int型变量分配4个字节的内存单元，为double型变量分配8个字节的内存单元，为char型变量分配1个字节的内存单元，为studentType型变量分配39个字节的内存单元。

因为p是int型指针，所以语句：

```
p++; 或 p = p+1;
```

将p中的值增加4个字节（指向下一个整型单元）。同样，语句：

```
q++;
chPtr++;
```

将q中的值增加8个字节，将chPtr中的值增加1个字节。语句：

```
stdPtr++;
```

将stdPtr中的值增加39个字节。

增量运算符根据指针变量所指的数据类型的大小，增加指针变量的值，减量运算符也是如此。

语句：

```
p = p+2;
```

将p中的值增加8个字节。

因此，在指针变量加上一个整数值时，指针变量的值增加了它所指的数据类型大小的整数倍。同样，指针变量减去一个整数值时，指针变量的值减少了它所指的数据类型大小的整数倍。

注意： 指针运算非常危险。进行指针运算时，程序可能会意外访问其他变量的内存空间，并有可能在没有警告的情况下改变这些变量的内容，并且使得程序员不知道哪里出了毛病。当指针变量试图访问一个既不是某个变量的内存空间，也不是某个合法的内存空间时，如系统空间，有些系统可能会给出相应的错误信息。因此，要谨慎使用指针运算。

14.8　动态数组

在第9章中，已经讨论了怎样声明和使用数组。因为这些数组的大小在编译时就固定下来，所以它们被称为静态数组。静态数组的缺点是每次执行程序时，数组的大小固定，所以不能使用同一数组处理不同长度的数据集。可以声明一个足够大的数组来解决这个限制，使它能够处理各种数据集。但是，如果数组很大而数据集中的数据很少，会造成内存空间的浪费。反之，如果能在程序运行期间提示用户输入数组大小，并根据输入的大小来创建数组，将会很灵活。尤其是在数组大小无法预测时，这种方法的优点更为明显。在本节中，将要说明怎样在程序运行期间创建、使用动态数组。

程序运行期间创建的数组称为动态数组（Dynamic Array）。可以使用new运算符的第二种形式来创建动态数组。

语句：

```
int *p;
```

声明了p是int型的指针变量。语句：

```
p = new int[10];
```

分配了10个连续的int型内存单元，并将其起始地址存储在p中。也就是说，运算符new创建了10个元素的int型数组，它返回该数组的起始地址。赋值运算符将起始地址存储在p中。例如，语句：

```
*p = 25;
```

将25存储在第一个内存单元中，语句：

```
p++;    //p points to the next array component
*p = 35;
```

将35存储在第二个内存单元中。所以，使用增量和减量运算符，就可以访问数组的每个成员。但是，经过几次增量运算之后，可能会丢失起始成员的地址。C++允许使用数组下标来访问这些成员。如下语句：

```
p[0] = 25;
p[1] = 35;
```

将25和35存储到数组的第一个和第二个成员中。即p[0]为数组的第一个成员，p[1]为数组的第二个成员，以此类推，p[i]为数组的第（i+1）个成员，而p仍然指向数组的第一个成员。下面的for循环将每个数组成员初始化为0：

```
for(j=0; j<10; j++)
    p[j]=0;
```

这里j是int型变量。

当用数组下标访问p所指的数组时，p始终指向第一个内存单元（数组的第一个成员）。在此，p为程序运行期间创建的数组，称为动态数组。

注意：语句：

```
int list[10];
```

将list声明为有10个元素的数组。当然，list本身是变量，它存储的值是数组的基地址，也就是第一个成员的地址。由于list是指针类型的值，所以它是指针变量。然而，由于list始终指向数组中的第一个元素，所以不能在list上使用增量或减量运算符。任何对list的增量或减量运算都会导致编译错误。如果p是int型指针变量，语句：

```
p = list;
```

将list值拷贝给p，可以对p进行增量或减量运算。

数组名是常量指针。

例14.4 下面的例子说明怎样得到用户自定义数组的长度，以及怎样在程序执行期间创建动态数组。考虑下面语句：

```
int *intList;                          //Line 1
int arraySize;                         //Line 2

cout<<"Enter the array size: ";        //Line 3
cin>>arraySize;                        //Line 4
cout<<endl;                            //Line 5

intList = new int[arraySize];          //Line 6
```

第1行中定义了一个int型指针intList；第2行中定义了一个int型变量arraySize；第3行提示用户输入数组的长度；第4行将用户输入的数组长度值赋给变量arraySize；第6行创建长度为arraySize的int型动态数组，并将这个数组的初始指针存储到变量intList中。从这以后，就可以把intList视为普通数组。例如，可以使用数组下标来处理intList中的元素，而且还可以将intList作为函数的参数值来传递。

函数和指针

指针变量既可以作为值参数又可以作为引用参数传递给函数。与其他变量声明机制一样，可以在函数头中将指针声明为值参数。为了将形参声明为引用参数，则应在函数头中的形参前加上&。也就是说，

若要将某个指针变量形参声明为引用参数，必须使用&。即，应该在数据类型名称和标识符名称之间使用*，将该标识符定义为指针；同时还要使用&，将该标识符定义为引用参数。一个显而易见的问题是，应该在数据类型名和标识符之间以何种顺序排列*和&来声明一个指针作为引用参数？在C++中，在函数头中声明一个指针作为引用参数时，符号*应在符号&之前。下面的举例说明了这一点：

```
void example(int* &p, double *q)
{
        .
        .
        .
}
```

在上面举例中，p和q都是指针：参数p是引用参数，参数q是值参数。

指针和函数返回值

在C++中，函数可以返回一个指针类型的值。例如，下面函数的返回值：

```
int* testExp(...)
{
        .
        .
        .
}
```

就是一个int型指针。

14.9 浅拷贝、深拷贝及指针

在前面的章节中，我们已经讨论了指针运算。而且，如果不小心，一个指针可能会指向其他（完全不相关）的内存地址。这可能会导致难以预料的或错误的结果。这里，我们讨论指针的另一种特性。为了便于讨论，将使用图表来说明指针和与其相关的内存。

考虑下面语句：

```
int *p;

p = new int;
```

第一条语句将p声明为一个int型的指针变量。第二条语句分配了int型的内存，并将其内存地址存储到变量p中。使用图14.8来说明这种情况：

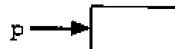


图 14.8 指针p和它所指向的内存

上图中的方块代表分配的内存（在本例中，类型为int），p和箭头代表p指向分配的内存。现在，考虑下面的语句：

```
*p=87;
```

该语句将87存储到p所指向的内存中，图14.9说明了这种情况：

图 14.9 将87存储到p所指向的内存中

考虑下面语句：

```
int *first;
int *second;

first = new int[10];
```

前面两条语句将first和second声明为int型指针变量，第三条语句创建了具有10个元素的数组，并将数组的基地址存储在first中，如图14.10所示。

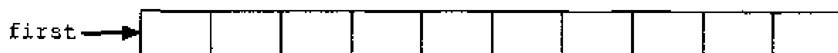


图 14.10　first指针及它所指的数组

假设first所指的数组中已存储数据，数组中的数据如图14.11所示。

图 14.11　指针 first 与它所指的数组

下面，考虑语句：

```
second = first;      //Line A
```

该句将first中的值拷贝到second中。在本语句执行后，first与second指向相同的数组，如图14.12所示。

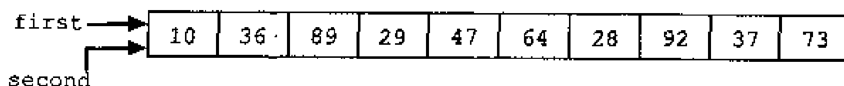


图 14.12　"second = first;" 执行后的 first 与 second

然后，执行下面的语句：

```
delete [] second;
```

该语句执行后，second所指的数组被删除，结果如图14.13所示。

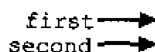


图 14.13　"delete [] second;" 执行后的 first 与 second

由于first与second指向同一个数组，在语句：

```
delete [] second;
```

执行后，first也变为无效。如果后面的程序仍然试图访问first所指的内存，则程序要么访问的是错误的内存地址，要么因错误而终止，这种情况就是浅拷贝。关于浅拷贝（Shallow Copy），正式的说法是，两个或多个相同类型的指针指向同一内存地址，即它们指向同一个数据。

另一方面，假设用下面语句替代Line A（即second = first;）：

```
second = new int[10];

for(int j = 0; j < 10; j++)
    second[j] = first[j];
```

第一条语句创建了有10个元素的int型动态数组，数组的基地址存储到second中。第二条语句将first所指数组中的内容拷贝到second所指的数组中，如图14.14所示。

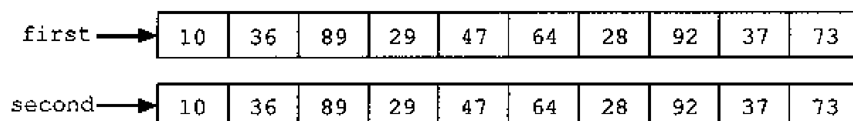


图 14.14　first 与 second 分别指向各自的数据

first 与 second 分别指向各自的数据，如果 second 删除自己的内存空间，对 first 不会有一点影响，这种情况就是深拷贝（Deep Copy）。深拷贝是指两个或多个指针分别指向各自的数据。

经过上面讨论，你应该清楚什么时候使用浅拷贝，什么时候使用深拷贝。

14.10　类与指针：一些特性

如果变量是类的指针，前面的章节中已经讨论过怎样使用箭头符号访问类成员。类也可以有指针类型的数据成员，本节将讨论这种类的一些特性。为了便于说明，使用下面的类：

```
class pointerDataClass
{
public:
    ...

private:
    int x;
    int lenP;
    int *p;
};
```

并考虑下面语句（如图 14.15 所示）：

```
pointerDataClass objectOne;
pointerDataClass objectTwo;
```

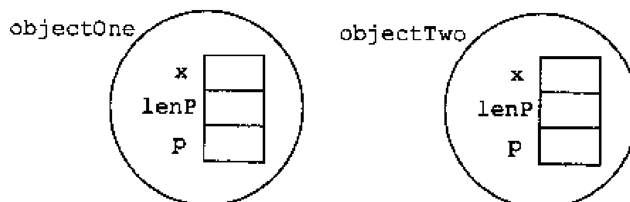


图 14.15　对象 objectOne 与 objectTwo

14.10.1　析构函数

对象 objectOne 有一个指针数据成员 p。假设 p 在程序运行期间创建了一个动态数组，当 objectOne 超出作用域（被释放）时，它的所有成员都被删除，但 p 创建了动态数组，而动态数组只能由 delete 运算符删除。因此，如果指针 p 没有使用 delete 运算符删除动态数组，即使不能被访问，动态数组的内存空间将仍然标记为已分配。假如 objectOne 如图 14.16 所示，怎样保证 p 被删除时，由 p 所创建的动态数组也被删除？

如前述，如果类有析构函数，当类对象超出其作用域（被释放）时，析构函数将会自动执行（见第 12 章）。因此，可以在析构函数中加入必要的代码，以保证 objectOne 超出作用域时，由 p 创建的内存会被释放。例如，可以将类 pointerDataClass 的析构函数定义为：

```
pointerDataClass::~pointerDataClass()
{
```

```
    delete [] p;
}
```

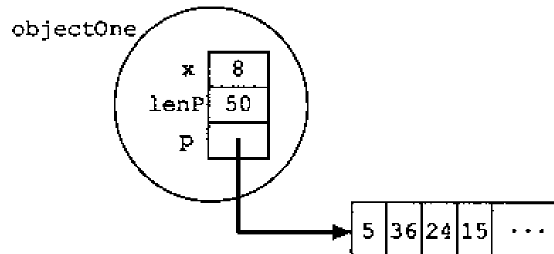


图 14.16　对象 objectOne 及其数据

当然，必须在类定义时将析构函数定义为该类一个成员。本节的剩余部分都假设pointerDataClass的析构函数的定义如上，即析构函数删除p所指的内存空间。包含了析构函数的pointerDataClass类的定义如下所示：

```
class pointerDataClass
{
public:
    ~pointerDataClass();
      ...

private:
    int x;
    int lenP;
    int *p;
};
```

注意：*为使析构函数正确工作，指针p一定要有合法的值。如果p未正确初始化（即p的值无效），程序执行后，要么将异常终止，要么将删除不相干的内存空间。因此，使用指针时，一定要多加小心。*

14.10.2　赋值运算符

本节将介绍带有指针数据成员的类，在使用内建的赋值运算符时的限制。假设 objectOne 与 objectTwo 如图 14.17 所示。

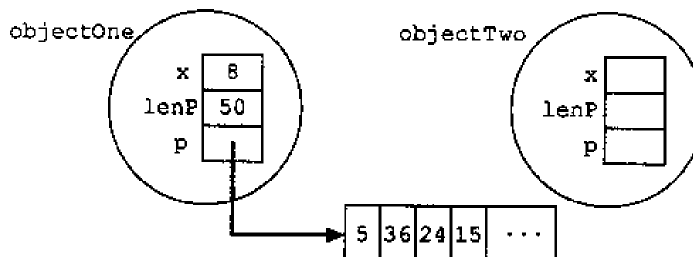


图 14.17　对象 objectOne 与 objectTwo

前面已经讲过，赋值运算符是类内建的运算符之一。例如，语句：

```
objectTwo = objectOne;
```

将 objectOne 中的数据成员拷贝到 objectTwo 中。即将 objectOne.x 拷贝到 objectTwo.x 中，将 objectOne.p 拷贝到 objectTwo.p 中。由于 p 是指针，这种基于成员方式的拷贝会导致数据的浅拷贝，objectTwo.p 与 objectOne.p 会指向同一内存空间，如图 14.18 所示。

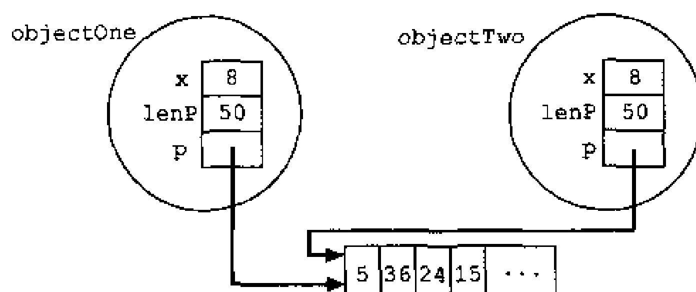


图 14.18 "objectTwo = objectOne;"执行后的对象 objectOne 与 objectTwo

如果objectTwo.p删除了它所指的内存空间，会造成objectOne.p非法。当pointerDataClass的对象超出作用域（被释放），其析构函数删除p所指的内存空间时，经常会发生上述情况。为避免带有指针数据成员的类出现数据的浅拷贝，C++允许程序员扩展赋值运算符的定义，这个过程称为赋值运算符的重载。第 15 章将讲述怎样实现运算符重载。一旦赋值运算符正确重载，对象 objectOne 和 objectTwo 都会有自己的数据、如图 14.19 所示。

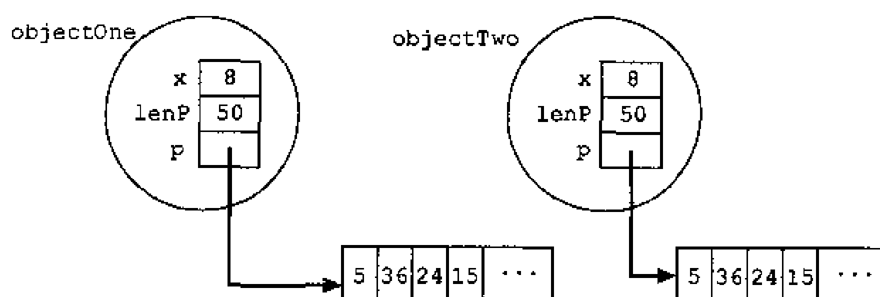


图 14.19 对象 objectOne 和 objectTwo

14.10.3 拷贝构造函数

声明某个类的对象时，可以用已经存在的对象的值对其进行初始化。例如，考虑下面语句：

```
pointerDataClass objectThree(objectOne);
```

它声明了对象objectThree，并将objectOne的成员的值拷贝到objectThree相应的成员中，完成objectThree的初始化。这种初始化方式，被称为默认的基于成员的初始化，由被称为拷贝构造函数（Copy Constructor）的构造函数实现，默认的拷贝构造函数由编译器提供。和赋值运算符一样，由于类pointerDataClass有指针数据成员，这种默认的初始化方法也会导致数据的浅拷贝，如图 14.20 所示。

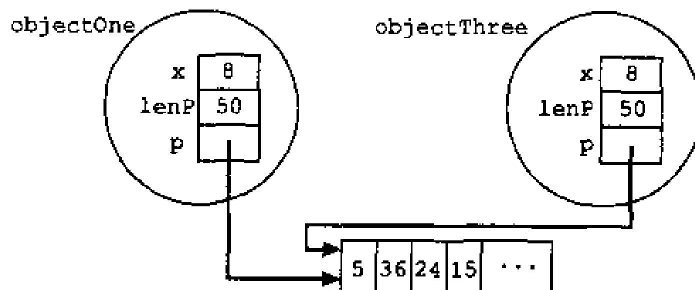


图 14.20 对象 objectOne 和 objectThree

在说明怎样解决这个问题之前，让我们先来考虑另外一个问题，它也会导致数据的浅拷贝。

前面说过，类的对象可以作为值参数或者引用参数传递给函数。类 pointerDataClass 的析构函数删除指针 p 所指的内存空间。假设 objectOne 如图 14.21 所示。

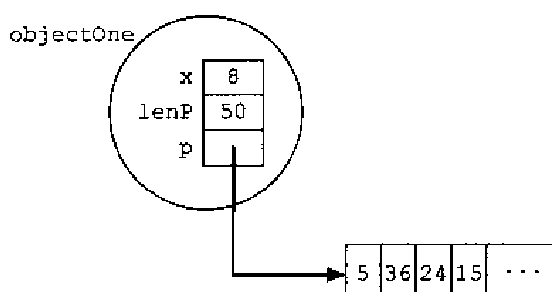


图 14.21 对象 objectOne

考虑下面的函数原型：

```
void destroyList(pointerDataClass paramObject);
```

函数 destroyList 有一个值参数 paramObject，再考虑下面语句：

```
destroyList(objectOne);
```

在上面语句中，objectOne 作为参数传给函数 destroyList。由于 paramObject 是值参数，拷贝构造函数将 objectOne 数据成员的值拷贝给 paramObject 相应的成员。和前面的例子一样，paramObject.p 和 objectOne.p 指向同一内存空间，如图 14.22 所示。

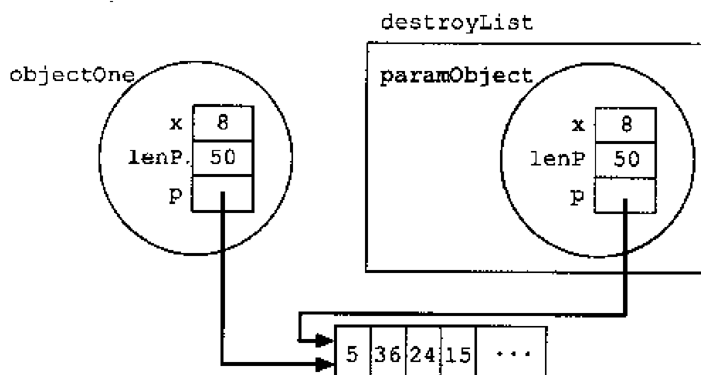


图 14.22 对象 objectOne 与 paramObject 的指针数据成员指向同一数组

由于 objectOne 作为值参数传递，paramObject 的数据成员都有自己的数据拷贝。特别是，paramObject.p 也有自己的内存空间。我们怎样保证事实确实如此呢？

如果类有指针数据成员：

- 对象声明期间，如果使用另一个对象的值对对象进行初始化，并且采用默认的基于成员的数据拷贝方式，将会导致数据的浅拷贝。
- 如果将对象作为值参数传递，并且采用默认的基于成员的数据拷贝方式，将会导致数据的浅拷贝。

在这两种情况下，为了强制对象有自己的数据拷贝，必须使用自己的拷贝构造函数定义来重载编译器提供的拷贝构造函数定义。要完成这个工作，常用的方法是在类的定义中加入拷贝构造函数，并编写拷贝构造函数的定义。这样，在执行拷贝构造函数时，系统会执行用户定义的而非编译器提供的拷贝构造函数。也就是说，在类 pointerDataClass 中包含拷贝构造函数可以解决浅拷贝问题。例 14.5 说明了这个问题。

在下面两种情况下，将自动执行拷贝构造函数：

- 使用其他对象的值声明和初始化对象时

● 对象作为值参数传递给函数时

因此，在正确定义了类 pointer DataClass 的拷贝构造函数后，objectOne.p 和 objectThree.p 都有自己的数据拷贝。如图 14.23 所示，objectOne.p 和 paramObject.p 都有自己的数据拷贝。

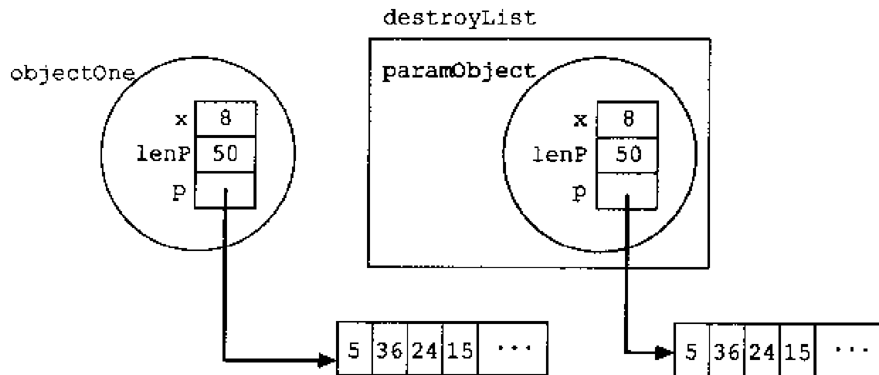


图 14.23　对象 objectOne 和 paramObject 的指针数据成员都有自己的数据

当函数 destroyList 执行完毕，形参 paramObject 将退出其作用域，paramObject 的析构函数删除 paramObject.p 所指的内存空间。这个删除对 objectOne 没有影响。

在类定义中包含拷贝构造函数的一般语法如下所示：

```
className(const className& otherObject);
```

例 14.5 说明怎样在类中包含拷贝构造函数，以及它是怎样工作的。

例 14.5

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 deallocate 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
};
```

假如类 pointerDataClass 的成员定义如下所示：

```
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 main 函数（假设类 pointerDataClass 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
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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
```

main函数的执行情况如下：第1行语句创建了一个pointerDataClass型对象one，并在对象one中创建了一个有5个元素的数组，并将数组的基地址存储在one.p中，one.p指向数组。第2行到第5行

的语句先将数据输入到 one 中，再输出 one 的数据。第 6 行语句创建对象 two，并用 one 的值初始化 two。拷贝构造函数将 one.x 拷贝给 two.x，并在 two 中创建了一个有 5 个元素的数组，数组的基地址存储在 two.p 中，然后将数组 one.p 内容拷贝给 two.p，one.p 和 two.p 都有自己的数据拷贝。第 8 行到第 13 行的语句完成了这个功能。例如，第 8 行语句输出 two 的数据，第 10 行语句删除 two.p 所指的内存空间，第 12 行语句输出 one 的值。同样，第 15 行到第 19 行说明如何将对象 one 作为值参数传递给函数 testCopyConst，拷贝构造函数使得形参 temp 拥有自己的数据拷贝。在函数 testCopyConst 中，temp 的值被改变，但对 one 没有影响。函数 testCopyConst 结束时，类 pointerDataClass 的析构函数删除分配给 temp.p 的内存空间，不会对 one.p 产生影响。

对于有指针数据成员的类，一般要做三件事：

1. 在类中包含析构函数
2. 重载类的赋值运算符
3. 包含拷贝构造函数

第 15 章将讨论赋值运算符的重载。在那以前，在遇到带有指针数据成员的类时，我们只实现上面列举的两个函数：析构函数和拷贝构造函数。

14.11　基于数组的表

本章前面的部分描述了怎样用指针创建动态数组，第 9 章简要介绍了怎样用循环处理数组中的元素。本节将讨论怎样用数组处理表。首先，我们有如下定义：

表（List）　相同类型元素的集合
表的长度是表中元素的个数。可以在表上进行的部分操作如下所示：

1. 创建表，将表初始化为空
2. 判断表是否为空
3. 判断表是否为满
4. 确定表大小
5. 删除或清空表
6. 判断某个数据是否与表中的某元素相同
7. 在表的指定位置插入一个元素
8. 删除表指定位置的元素
9. 替换表中指定位置的元素
10. 检索表中指定位置的元素
11. 在表中搜索指定的元素

在讨论怎样使用这些操作之前，首先要确定在计算机中怎样存储表。由于表中所有元素的类型完全一样，一个有效而又简便的处理表方法是将它存在数组中。通常在开始时，数组中可以存储的元素个数比它实际存储的表中元素的个数多。此后，表可以在一定范围内适当地增加元素。因此，我们一定要知道数组是否满？也就是说，要能跟踪数组中存储的表元素的个数。C++ 允许程序员创建动态数组，从而可以让用户指定数组大小。即可以在声明表对象时，指定数组大小。要想用数组维护和处理表，必须要使用下面三个变量：

- 存储表元素的数组

- 表明表长度的变量（目前数组中表元素的数目）
- 表明数组大小的变量（数组中能存储的表元素的数目）

既然已经知道可对表进行的操作，以及表在内存中的存储，就可以将表作为ADT（抽象数据类型）来定义类。为了说明这个问题，假设表中的元素都是int型，在第15章讨论类模板时，我们将去掉此限制，设计一个能处理各种类型表的通用类。

下面类将list定义为ADT：

```
class arrayListType
{
public:
    bool isEmpty();
      //Returns true if the list is empty;
      //otherwise, returns false.
    bool isFull();
      //Returns true if the list is full;
      //otherwise, returns false.
    int listSize();
      //Returns the size of the list, that is, the number
      //of elements currently in the list.
    int maxListSize();
      //Returns the maximum size of the list, that is, the
      //maximum number of elements that can be stored in
      //the list.
    void print() const;
      //Outputs the elements of the list.
    bool isItemAtEqual(int location, int item);
      //If the item is the same as the list element at the
      //position specified by the location, returns true;
      //otherwise, returns false.
    void insertAt(int location, int insertItem);
      //Inserts an item in the list at the specified location.
      //The item to be inserted and the location are passed
      //as parameters to the function.
      //If the list is full or the location is out of range,
      //an appropriate message is displayed.
    void insertEnd(int insertItem);
      //Inserts an item at the end of the list. The item to
      //be inserted is specified by the parameter insertItem.
    void removeAt(int location);
      //Removes the item from the list at the specified
      //position. The location of the item to be removed is
      //passed as a parameter to this function.
    void retrieveAt(int location, int& retItem);
      //Retrieves the element from the list at the position
      //specified by the location. The item is returned via
      //the parameter retItem.
    void replaceAt(int location, int repItem);
      //Replaces the elements in the list at the position
      //specified by the location. The item to be replaced is
      //specified by the parameter repItem.
      //If the list is full, an appropriate message is
      //displayed.
    void clearList();
      //All elements from the list are removed. After this
      //operation, the size of the list is zero.
```

```
    int seqSearch(int item);
      //Searches the list for a given item. If the item is
      //found, returns the location in the array where the
      //item is found; otherwise, returns -1.
    void insert(int insertItem);
      //The item specified by the parameter insertItem is
      //inserted at the end of the list. However, the list
      //is first searched to see whether the item to be
      //inserted is already in the list. If the item is
      //already in the list, an appropriate message is output.
    void remove(int removeItem);
      //Removes an item from the list. The item to be removed
      //is specified by the parameter removeItem.
    arrayListType(int size = 100);
      //constructor
      //Creates an array of the size specified by the
      //parameter size. The default array size is 100.
    arrayListType (const arrayListType& otherList);
      //copy constructor
    ~arrayListType();
      //destructor
      //Deallocates the memory occupied by the array.

protected:
    int *list;  //array to hold the list elements
    int length; //stores the length of the list
    int maxSize; //stores the maximum size of the list
};
```

注意，之所以将类arrayListType的数据成员声明为protected，是因为我们要求此类的派生类能实现某些特殊的表，如一个有序表。下面是这些函数的定义。

length是zero时，表示表是空的；length等于maxSize时，表示表是满的。因此，isEmpty和isFull函数定义如下所示：

```
bool arrayListType::isEmpty()
{
      return (length == 0);
}
bool arrayListType::isFull()
{
      return (length == maxSize);
}
```

类的数据成员length存储目前表中元素的数目。同样，数据成员maxSize是数组的大小，它决定了能存储的表元素的个数，maxSize指定了表中可以存储的最多元素数目。因此，函数listSize和maxListSize的定义是：

```
int arrayListType::listSize()
{
      return length;
}

int arrayListType::maxListSize()
{
      return maxSize;
}
```

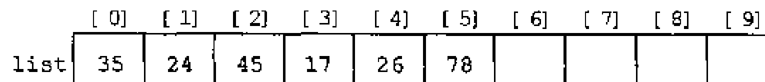
成员函数print输出表中的元素，这里假设输出到标准输出设备。

```
void arrayListType::print() const
{
      int i;

      for(i = 0; i < length; i++)
          cout<<list[i]<<" ";

      cout<<endl;
}
```

函数isItemAtEqual的定义如下所示：

```
bool arrayListType::isItemAtEqual(int location, int item)
{
      return(list[location] == item);
}
```

函数insertAt在表中指定位置上插入一个元素。插入的元素及要插入的位置作为参数传递给此函数。若要在表中间插入元素，则必须为这个新元素腾出位置。也就是说，要将数组中的一些元素向数组后面移动一个位置，图14.24说明了这个概念。



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
list	35	24	45	17	26	78				

图14.24 数组表

目前表中有6个元素，所以length是6。在插入一个新元素之后，表的length是7。如果要将数据插入到位置6上，则只需将元素拷贝到list[6]中即可。如果要将元素插入到位置3上，则必须先要将数组中的list[3]、list[4]和list[5]向后移动一位，以腾出位置。即，按顺序将list[5]拷贝到list[6]、将list[4]拷贝到list[5]，将list[3]拷贝到list[4]。然后，再将新元素拷贝到list[3]中。

当然，对于一些特例，如在已满的表中插入元素也应该可以处理。某些情况可以由其他成员函数实现。

函数insertAt的定义如下所示：

```
void arrayListType::insertAt(int location, int insertItem)
{
   int  i;

   if(location < 0 || location >= maxSize)
      cout<<"The position of the item to be inserted "
          <<"is out of range"<<endl;
   else
      if(length >= maxSize)  //list is full
         cout<<"Cannot insert in a full list"<<endl;
      else
      {
         for(i = length; i > location; i--)
             list[i] = list[i - 1]; //move the elements down

         list[location] = insertItem;   //insert the item at the
                                        //specified position

         length++;      //increment the length
      }
} //end insertAt
```

函数 insertEnd 的功能可用函数 insertAt 来完成，但 insertEnd 不需要移动元素。因此，下面直接给出它的定义：

```
void arrayListType::insertEnd(int insertItem)
{

   if(length >= maxSize)  //the list is full
      cout<<"Cannot insert in a full list"<<endl;
   else
   {
      list[ length] = insertItem; //insert the item at the end
      length++;   //increment the length
   }
} //end insertEnd
```

函数 removeAt 与函数 insertAt 的功能相反。removeAt 删除表中指定位置的元素。要将删除元素的位置作为参数传递给该函数。并且，在删除之后，将表的长度减 1。若删除的是表中间的元素，还要将后面的元素依次前移，以免在存储表的数组中留下空位置。图 14.25 说明了这个概念。

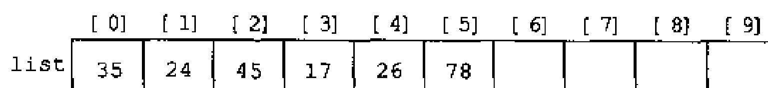


图 14.25　数组表

表中元素的个数是 6，也就是 length 为 6。删除一个元素之后，length 是 5。假如要删除的元素在位置 3 上。显然，要依次把 list[4]移到 list[3]，把 list[5]移到 list[4]。

函数 removeAt 的定义如下所示：

```
void arrayListType::removeAt(int location)
{
   int i;

   if(location < 0 || location >= length)
      cout<<"The location of the item to be removed "
          <<"is out of range"<<endl;
   else
   {
      for(i = location; i < length - 1; i++)
          list[ i] = list[ i+1] ;

      length--;
   }
} //end removeAt
```

函数 retrieveAt 的定义比较简单，它的参数是需要检索元素的位置及结果。同样，函数 replaceAt 的定义也比较简单，下面是它们的定义：

```
void arrayListType::retrieveAt(int location, int& retItem)
{
   if(location < 0 || location >= length)
      cout<<"The location of the item to be retrieved is "
          <<"out of range"<<endl;
   else
      retItem = list[ location] ;
} //end retrieveAt
```

```
void arrayListType::replaceAt(int location, int repItem)
{
   if(location < 0 || location >= length)
      cout<<"The location of the item to be replaced is "
          <<"out of range"<<endl;
   else
      list[location] = repItem;
} //end replaceAt
```

函数 clearList 将表中的所有元素全部删除。由于数据成员 length 指明了表中元素的个数，通过将 length 置为 0 即可删除全部元素。函数的定义如下所示：

```
void arrayListType::clearList()
{
      length = 0;
} //end clearList
```

下面讨论构造函数和析构函数。构造函数创建由用户指定大小的数组，并将表的长度初始化为 0，maxSize 由用户所指定的数组的大小来确定。数组的大小作为参数传递给构造函数，默认的数组大小是 100。析构函数释放分配给数组的内存空间。构造函数和析构函数的定义如下所示：

```
arrayListType::arrayListType(int size)
{
      if(size < 0)
      {
         cout<<"The array size must be positive. Creating "
             <<"an array of size 100. "<<endl;

         maxSize = 100;
      }
      else
         maxSize = size;

      length = 0;
      list = new int[maxSize];
}

arrayListType::~arrayListType()
{
     delete [] list;
}
```

14.11.1 拷贝构造函数

前面已经说明，将对象作为值参数传递给函数，或者使用一个对象对另一个同类型对象声明初始化时，要调用拷贝构造函数。构造函数将实际对象的数据成员拷贝到形参中来创建新的对象。拷贝构造函数定义如下所示：

```
arrayListType::arrayListType(const arrayListType& otherList)
{
   int j;

   maxSize = otherList.maxSize;
   length = otherList.length;
   list = new int[maxSize];       //create the array

   if(length != 0)                //if otherList is not empty
```

```
        for(j = 0; j < length; j++)   //copy otherList
            list [ j] = otherList.list[ j] ;
} //end copy constructor
```

14.11.2　搜索

下面要讨论的搜索算法称为顺序或线性搜索。

考虑图 14.26 中有 7 个元素的表。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	
list	35	12	27	18	45	16	38		...

图 14.26　有 7 个元素的表

假如要判断 27 是否在表中，顺序搜索的过程如下：首先，将 27 与 list[0]比较，即比较 27 与 35。由于 list[0]不等于 27，再比较 27 与 list[1]（表的第二个元素 12），list[1]不等于 27，继续将 27 与下一个元素比较，即比较 27 和 list[2]。由于 list[2]等于 27，搜索停止。这个举例演示了一次成功的搜索。

现在来搜索 10。和前面一样，先从表的第一个元素——list[0]开始搜索。这次，10 与表中的每一项进行比较，直到最后。这个举例演示了一次失败的搜索。

总之，若在表中找到与搜索内容相同的元素，就立即停止，并报告“成功”（这种情况下，一般还要报告搜索到的元素在表中的位置）。否则，在被搜索的元素与表中每个元素比较之后，就应该停止搜索并报告“失败”。

假设存放表的数组名为 list，可将前面的讨论转换成顺序搜索算法，算法如下所示：

```
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;
```

下面的算法可在表中执行顺序搜索。为了便于说明，假设表中的元素都是 int 型。

```
int arrayListType::seqSearch(int item)
{
      int loc;
      bool found = false;

      for(loc = 0; loc < length; loc++)
          if(list[ loc] == item)
      {
              found = true;
              break;
      }

      if(found)
         return loc;
      else
         return -1;
} //end search
```

在了解了顺序搜索算法的实现之后，下面给出 insert 和 remove 的函数定义。前面说过，如果表未满，insert 函数在表末尾插入一个新元素。remove 函数从非空表中删除一个元素。

14.11.3 插入

函数 insert 在表中插入一个新元素。由于在表中不允许元素重复，要先搜索表以确定待插入的元素是否已经存在。确定一个元素是否在表中的函数为前面已经介绍过的 seqSearch。如果元素未在表中，就将它插在表的末尾，表的 length 加 1，要插入的元素作为参数传递给函数。函数的定义如下所示：

```
void arrayListType::insert(int insertItem)
{
    int loc;

    if(length == 0)                     //list is empty
       list[ length++] = insertItem;    //insert the item and
                                        //increment the length
    else
       if(length == maxSize)
          cout<<"Cannot insert in a full list."<<endl;
       else
       {
          loc = seqSearch(insertItem);
          if(loc == -1)    //the item to be inserted
                           //does not exist in the list
             list[ length++] = insertItem;
          else
             cout<<"the item to be inserted is already in "
                 <<"the list. No duplicates are allowed."<<endl;
       }
} //end insert
```

14.11.4 删除

函数 remove 从表中删除一个元素，待删除的元素作为参数传递给该函数。删除元素时，此函数要调用成员函数 seqSearch 以确定要删除的元素是否在表中。如果有，就删除它并将表长度减 1。若在表中找到了要删除的元素，函数 seqSearch 返回它的下标，函数 removeAt 根据返回的下标删除表中的元素。函数 remove 的定义如下所示：

```
void arrayListType::remove(int removeItem)
{
    int loc;

    if(length == 0)
       cout<<"Cannot delete from an empty list."<<endl;
    else
    {
       loc = seqSearch(removeItem);

       if(loc != -1)
          removeAt(loc);
       else
          cout<<"The item to be deleted is not in the list."
              <<endl;
    }
} //end remove
```

下面的程序用来测试表的一些操作：

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

using namespace std;

void testCopyConstructor(arrayListType testList);

int main()
{
    arrayListType list;                                   //Line 1
    int num;                                              //Line 2

    cout<<"Line 3: Enter numbers ending with -999"
        <<endl;                                           //Line 3

    cin>>num;                                             //Line 4

    while(num != -999)                                    //Line 5
    {
          list.insert(num);                               //Line 6
          cin>>num;                                       //Line 7
    }

    cout<<"Line 8: The list you entered is: "
        <<endl;                                           //Line 8
    list.print();                                         //Line 9
    cout<<"Line 10: The list size is: "
        <<list.listSize()<<endl;                          //Line 10

    cout<<"Line 11: Enter the item to be deleted: ";      //Line 11
    cin>>num;                                             //Line 12
    cout<<endl;                                           //Line 13

    list.remove(num);                                     //Line 14

    cout<<"Line 15: After removing "<<num
         <<" the list is: "<<endl;                        //Line 15
    list.print();                                         //Line 16
    cout<<"Line 16: The list size is: "
        <<list.listSize()<<endl;                          //Line 17

    //test copy constructor

    testCopyConstructor(list);                            //Line 18

    cout<<"Line 19: The list after the copy "
        <<"constructor."<<endl;                           //Line 19
    list.print();                                         //Line 20
    cout<<"Line 21: The list size is: "
        <<list.listSize()<<endl;                          //Line 21

    return 0;                                             //Line 22
}

void testCopyConstructor(arrayListType testList)
{
```

```
    cout<<"Line 23: Inside the function "
        <<"testCopyConstructor."<<endl;                    //Line 23

    testList.print();                                      //Line 24

    cout<<"Line 25: The list size is: "

        <<testList.listSize()<<endl;                       //Line 25
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
Line 3: Enter numbers ending with -999
23 16 15 25 35 46 14 32 98 7 -999
Line 8: The list you entered is:
23 16 15 25 35 46 14 32 98 7
Line 10: The list size is: 10
Line 11: Enter the item to be deleted: 98

Line 15: After removing 98 the list is:
23 16 15 25 35 46 14 32 7
Line 16: The list size is: 9
Line 23: Inside the function testCopyConstructor.
23 16 15 25 35 46 14 32 7
Line 25: The list size is: 9
Line 19: The list after the copy constructor.
23 16 15 25 35 46 14 32 7
Line 21: The list size is: 9
```

这个输出的过程很简单，详细内容可做为练习留给读者。

14.12 继承、指针和虚函数

注意： 读者可以跳过本节而不会影响下面内容的学习。

前面已经说明，类对象既可以作为值参数又可以作为引用参数传递给函数。形参与实参必须相匹配。但是，对于类，C++允许用户将派生类的对象传递给基类类型的形参。

首先，先讨论形参是引用参数或指针的情况。为了说明这种情况，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;
};
```

类 baseClass 有三个成员，类 derivedClass 从 baseClass 派生，并有三个自己的成员，两个类中都有成员函数 print。假设两个类的成员函数的定义如下所示：

```
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 的形参 p 的类型是 baseClass。在调用函数 callPrint 时，参数既可以使用 baseClass 类型的对象，也可以使用 derivdeClass 类型的对象。而且，在 callPrint 函数中调用成员函数 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
```

第1行到第5行的语句很简单，主要看第6行和第7行。第6行语句调用函数callPrint并将one作为参数，它产生输出的第5行。第7行语句也调用函数callPrint，但将对象two作为参数，它产生输出的第6行。尽管传递给该函数的参数的类型不同，但是第6行与第7行的输出却是相同的变量x中的值（由于在第7行中，对象two作为参数传给callPrint，第7行输出本应该和第2行及第3行输出一样）。这是因为在第6行和第7行中，执行的都是baseClass的成员函数print。之所以这样，是因为callPrint函数体中成员函数print的绑定是在编译时进行的。由于callPrint的形参p是baseClass类型，所以对于语句p.print();与编译器关联的是baseClass的print函数。更加具体地说，在编译时绑定（Compile-time Binding）机制中，调用指定函数所必须的代码是由编译器指定的。编译时绑定也称为静态绑定（Static Binding）。

在第7行语句中，实参是derivedClass类型。因此，当执行到函数callPrint的函数体时，逻辑上执行的应该是对象two的print函数。但事实并非如此。所以，一个很自然的问题是：在程序执行期间，C++是怎样避免这种问题发生的呢？在C++中，是通过虚函数（Virtual Function）机制来解决这个问题的。虚函数的绑定发生在程序执行期间，而不是在编译时。这种绑定被称为运行时绑定（Run-time Binding）。在编译时，编译器并不产生调用指定函数的代码，而只是提供必要的信息，使得运行时系统能产生实际的代码来调用相应的函数。运行时绑定也称为动态绑定（Dynamic Binding）。

在C++中，虚函数通过保留字virtual声明。使用虚函数机制，重新定义前面的类如下所示：

```
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;
};
```

注意仅需要在基类中声明virtual函数。

成员函数print的定义和前面一样，执行改变后的程序，得到的输出如下所示。

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
```

该输出表明第7行语句，执行的是derivedClass类的print函数（见输出的最后两行）。

若形参是某个类的指针，而派生类对象的指针作为实参传递，前面的讨论也适用。为说明这个特性，假设有前面的类（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 14: 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
```

上面的例子说明，如果形参，也就是 p，是引用参数或指针，并且在基类中使用了虚函数，就可以有效地将派生类的对象作为实参传给 p。但是，如果 p 是值参数，即使 p 使用了虚函数，这种机制也无法将派生类的对象作为实参传递给 p。回想一下，若形参是值参数，则实参的值拷贝给形参。因此，若形参是 class 类型，则实际对象的数据成员拷贝给形参相应的数据成员。

假设我们有前面的类——baseClass 和 derivedClass，考虑下而的函数定义：

```
void callPrint(baseClass p)        //p is a value parameter
{
```

```
   p.print();
}
```

进一步假设有下面的声明：

```
derivedClass two;
```

对象two有两个数据成员x和a，x由基类继承而来。下面的函数调用：

```
callPrint(two);
```

由于形参p是值参数，two的数据成员拷贝给p的数据成员。然而，p是baseClass的对象，它只有一个数据成员。因此，two的数据成员中只有x拷贝给p中的数据成员x。同时，语句：

```
p.print();
```

执行的是类baseClass中的成员函数print。

下面程序的输出进一步说明了这种情况（和前面一样，假定类baseClass的定义在头文件baseClass.h中，类derivedClass的定义在头文件derivedClass.h中）。

```
//Chapter 14: 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
```

仔细查看程序中第6行和第7行语句的输出（输出的最后两行）。在第7行中，由于形参p是值参数，two的数据成员拷贝给p的数据成员。然而，由于p是baseClass类型的对象，只有一个数据成员。所以，只有two的数据成员x拷贝给p的数据成员x。而且，函数callPrint中的语句p.print( );执行的是baseClass的print函数，而不是derivedClass的print函数。因此，最后一行只输出了x的值（two的数据成员）。

注意：若形参是派生类类型，则基类的对象不能传递给它。

14.12.1 类与虚析构函数

有指针数据成员的类都要有析构函数。当类的对象超出作用域（被释放）时，析构函数自动执行。如果对象创建了动态对象，析构函数可以用来释放动态对象的存储空间。如果派生类对象传递给基类类型的形参，无论是作为引用参数还是作为值参数，自动执行的都是基类的析构函数。然而，逻辑上，当派生类的对象超出作用域时，执行的应该是派生类的析构函数。

为解决这个问题，基类的析构函数必须是虚函数。基类的虚析构函数自动使派生类的析构函数都为虚函数。若派生类的对象传递给基类类型的形参，当对象超出作用域时，执行派生类的析构函数。派生类的析构函数执行之后，执行基类的析构函数。因此，删除派生类对象时，也删除了派生类对象中的基类部分（从基类继承来的成员）。

如果基类包含了虚函数，基类的析构函数同时也要设为虚函数。

14.13 取地址运算符和类

本章已经使用取地址运算符&将变量的地址存储在指针变量中，取地址运算符还可用来创建对象的别名。考虑下面语句：

```
int x;
int &y = x;
```

第一个语句将x声明为int型变量，第二个语句声明了x的别名y，x和y引用同一内存单元。因此，y就像是常量指针变量。语句：

```
y = 25;
```

将y与x的值设为25。同样，语句：

```
x = 2 * x + 30;
```

将同时更新x和y的值。

也可以使用取地址运算符来返回类的私有数据成员地址。但若不小心的话，可能导致程序出现严重的错误。我们借助于下面举例来说明这个问题。

考虑下面类的定义：

```
//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
```

成员函数的实现如下所示：

```
//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;
}
```

由于函数addressOfX返回的是int型内存单元的地址，所以语句：

```
return x;
```

返回x的地址。

首先，我们借助于使用类testAddress的一个简单程序说明产生错误的原因。然后，说明应该怎样改正错误。

```
//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，语句：

```
int &y = a.addressOfX();
```

执行以后，y 成了对象 a 的私有数据成员 x 的别名。因此，语句：

```
y = 25;
```

将改变 x 中的值。

在第 12 章中讨论过，类的私有数据成员不能在类外面访问。但是，程序员可以通过返回它们的地址来访问这些私有数据成员。解决这个问题的一个方法就是，不向用户提供类的私有数据成员地址。但是，正如在下一章中将会看到的，有时又必须要返回私有数据成员的地址。怎样才能防止程序直接访问私有数据成员呢？可以通过在函数返回类型之前加 const 来解决这个问题。通过这种方法，既可以返回私有数据成员的地址，同时又可以防止程序员能够直接访问私有数据成员。使用这种机制重写的类 testAddress 如下所示：

```
#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
```

在实现文件中，函数 addressOfX 的定义如下所示：

```
const int& testAddress::addressOfX()
{
   return x;
}
```

这样同样的程序将会产生编译错误。

14.14　小结

1. 指针变量的值是其他变量的地址。
2. 在 C++ 中，指针类型数据没有名字。

3. 指针变量是在数据类型和变量之间使用 * 来声明，例如语句：

```
int *p;
char *ch;
```

声明 p 和 ch 为指针变量。p 的值指向 int 型内存空间，ch 的值指向 char 型内存空间。通常，p 称为 int 型指针变量，ch 称为 char 型指针变量。
4. 在 C++ 中，& 称为取地址运算符。
5. 取地址运算符返回操作数的地址。例如，p 是整型指针变量，num 是整型变量，语句：

```
p = &num;
```

将 p 的值设为 num 的地址。
6. 作为单目运算符时，* 称为递引用运算符。
7. 由指针变量指示的内存单元可通过递引用运算符 * 访问。例如，p 是整型指针变量，语句：

```
*p = 25;
```

将 p 所指的内存单元的值设为 25。
8. 访问指针所指的对象的成员，可使用成员访问运算符 ->。
9. 指针变量可用 0（整数 0），NULL 或同类型变量的地址来初始化。
10. 惟一可以直接赋给指针变量的数值是 0。
11. 在指针变量上只允许算术运算：增量（++）、减量（--）、指针变量加整数、指针变量减整数，以及两指针相减。
12. 指针算术运算不同于一般的算术运算。指针加上整数时，指针所加的值是指针所指的对象大小的整数倍。同样，指针变量减去整数时，减去的也是指针所指对象大小的整数倍。
13. 指针变量可用关系运算符比较（只有在指针指向相同类型数据时比较才有意义）。
14. 指针变量的值可赋给同类型的另一指针变量。
15. 程序执行时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 用来释放给动态变量分配的内存空间。
18. 在 C++ 中，new 和 delete 都是保留字。
19. 运算符 new 有两种形式：创建单个动态变量及创建动态变量数组。
20. 若 p 是 int 型指针，语句：

```
p = new int;
```

在内存中分配 int 型空间，并将分配的内存空间地址存储在 p 中。
21. 运算符 delete 有两种形式：释放单个动态变量的内存空间及释放动态变量数组的内存空间。
22. 若 p 是 int 型指针，语句：

```
delete p;
```

释放 p 所指的内存空间。
23. 数组名是常量指针，它始终指向同一内存单元，即数组第一个成员的位置。
24. 可以使用 new 运算符创建动态数组。例如，p 是 int 型指针，语句：

```
p = new int[10];
```

创建了有 10 个元素的动态数组。数组的基地址存在 p 中，我们称 p 为动态数组。

25. 可以使用数组下标来访问动态数组中的元素。例如，假设p是有10个元素的动态数组，p[0]为数组的第一个元素，p[1]为第二个元素，等等。p[i]为数组的第i+1个元素。
26. 程序执行时创建的数组称为动态数组。
27. 若p是动态数组，语句：

```
delete [] p;
```

释放分配给p的内存空间。
28. 浅拷贝中，两个或多个同类型指针指向相同的内存空间，也就是指向同一数据。
29. 深拷贝中，两个或更多同类型的指针都有各自的数据。
30. 若类有析构函数，当对象超出作用域时，析构函数自动执行。
31. 若类有指针数据成员，内建的赋值运算符提供了数据的浅拷贝。
32. 当一个对象用另一个对象声明和初始化时，或者对象作为参数传值时，执行拷贝构造函数。
33. 表是同类元素的集合。
34. 表可进行的一般操作有：创建表、判断表是否为空、判断表是否为满、确定表的大小、删除或清空表、判断表中是否有某项、在表的指定位置插入一个元素、在表的指定位置删除某元素、用某项替换表中指定位置的元素、在表中指定的位置检索某元素，以及搜索表以寻找给定的元素。
35. C++ 允许给基类类型的形参传递派生类的对象。
36. 虚函数的绑定是在执行期间进行而不是在编译时进行的，这称为动态或执行时绑定。
37. 在 C++ 中，虚函数用保留字 virtual 声明。
38. 取地址运算符可用来返回类的私有数据成员的地址。

14.15　练习

1. 判断下面说法的正误。
   a. 在 C++ 中，pointer 是保留字。
   b. 在 C++ 中，指针变量通过保留字 pointer 声明。
   c. 语句 delete p；释放变量指针 p。
   d. 语句 delete p；释放 p 所指的动态变量。
   e. 有下面的声明：

   ```
   int list[10];
   int *p;
   ```

   语句：

   ```
   p = list;
   ```

   在 C++ 中是合法的。
   f. 有下面的声明：

   ```
   int *p;
   ```

   语句：

   ```
   p  = new int[50];
   ```

   动态分配了有 50 个元素的 int 型数组，并且 p 包含了数组的基地址。
   g. 取地址运算符返回操作数的地址和值。
   h. 若 p 是指针变量，语句 p = p*2;在 C++ 中是合法的。

2. 声明如下：

```
int x;
int *p;
int *q;
```

判断下面语句是否合法。如果不合法，请说明原因。

a. `p = q;`

b. `*p = 56;`

c. `p = x;`

d. `*p = *q;`

e. `q = &x;`

f. `*p = q;`

3. 下面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. 下面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. 有如下声明：

```
int num = 6;
int *p = &num;
```

下面哪些语句使num的值加1？

a. `p++;`

b. `(*p)++;`

c. `num++;`

d. `(*num)++;`

6. 下面代码的输出是什么？

```
int *p;
int *q;
p = new int;
q = p;
*p = 46;
*q = 39;
cout<<*p<<" "<<*q<<endl;
```

7. 下面代码的输出是什么?

```
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. 下面的代码有什么错误?

```
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. 下面代码的输出是什么?

```
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. 下面代码的输出是什么?

```
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. 解释浅拷贝与深拷贝的区别。
12. 下面的代码有什么错误?

```
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. 下面代码的输出是什么?

```
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. 拷贝构造函数的目的是什么?
15. 哪两种情况下会执行拷贝构造函数?

16. 若类有指针数据成员，应当做哪三件事？

17. 有类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. 若classA的定义如下，练习17中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. 编译时绑定与运行时绑定有什么区别？

14.16 编程练习

1. 用动态数组重做第9章编程练习5。
2. 用动态数组重做第9章编程练习6。
3. 用动态数组重做第9章编程练习7，必须询问候选人的数目，然后创建相应的数组存放数据。

第15章　重载与模板

本章要点：

- 了解重载
- 了解运算符重载的限制
- 了解指针 this
- 了解友元（friend）函数
- 理解类的成员和非成员（nonmember）
- 理解怎样重载各种运算符
- 了解模板
- 理解怎样构造函数模板和类模板

第12章中已经讨论过，在C++中，类是怎样将数据和对数据的操作结合起来作为一个整体的。将数据和对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。第12章定义了抽象数据类型（ADT），并说明了在C++中类是怎样实现ADT的。第13章讨论了怎样通过继承机制，从已有的类中派生出新类。继承是OOD的第二个原则，它鼓励代码重用。

本章的第一个主题是运算符重载，第二个主题是模板。模板允许程序员为相关的函数和类写出通用代码。可以使用函数模板来简化函数重载。

15.1　为什么需要重载运算符

第12章定义并应用了类clockType，并且说明了怎样在程序中用clockType表示一天中的时间。我们再来看一下类clockType的几个特点。

考虑下面语句：

```
clockType myClock(8,23,34);
clockType yourClock(4,5,30);
```

第一个语句声明了myClock，并分别将myClock的数据成员hr，min和sec初始化为8，23和34。第二个语句声明了clockType类型对象yourClock，并分别将yourClock的数据成员hr，min和sec初始化为4，5和30。

现在考虑下面语句：

```
myClock.printTime();
myClock.incrementSeconds();
if(myClock.equalTime(yourClock))
.
.
.
```

第一个语句以hr:min:sec的形式打印myClock的值，第二个语句将myClock中sec的值加1，第三个语句检查myClock的值是否与yourClock的值相等。

这些语句能正常工作。然而，如果能使用插入运算符<<输出myClock的值，增量运算符++按秒增加myClock的值，就能极大地增加程序的灵活性。为了更明确地说明这一点，我们用下面的语句代替前面的语句：

```
cout<<myClock;
myClock++;
if(myClock == yourClock)
.
.
.
```

前面已经说过，类中内建的操作只有赋值运算符和成员访问运算符，其他运算符都不能直接用于类的对象。不过，C++允许程序员扩展很多运算符的定义。因此，关系运算符、算术运算符、数据输出时的插入运算符，以及数据输入时的析取运算符等，都能用于类。在C++的术语中，这称为运算符重载。除了运算符重载，本章还将讨论第7章已简要介绍过的函数重载。

15.2 运算符重载

回忆一下，算术运算符/是怎样工作的。如果运算符/的两个操作数都是整数，结果就是整数；否则，结果就是浮点数。同样，流插入运算符<<和流析取运算符>>也都是重载过的：运算符>>可作为流析取和右移运算符，<<可作为流插入和左移运算符。这些是运算符重载的例子。

另一个重载的运算符是+和-，对于整数运算、浮点运算，及指针运算来说+和-的结果各不相同。

C++允许用户重载大部分运算符，以使运算符在应用中的使用效率更高。但它不允许用户创建新的运算符。现有的大部分运算符都可以通过重载来对类的对象进行操作。

如果要重载运算符，就必须编写函数定义（函数头和函数体）。重载运算符的函数名的写法是保留字operator后面跟要重载的运算符。例如，重载运算符>=的函数名是：

```
operator>=
```

运算符函数 重载运算符的函数。

15.2.1 运算符函数的语法

运算符的计算结果是值。因此，运算符函数是带有返回值的函数。

运算符函数的函数头的语法为：

```
returnType operator operatorSymbol(formal parameter list)
```

在C++中，operator是保留字。

前面说过，类内建的运算符只有赋值（=）和成员访问运算符。对于类的对象，要使用其他运算符，就必须明确地重载它们。运算符重载为用户自定义的数据类型提供了简洁的表达方式。

重载类的运算符需要：

1. 在类的定义中包含运算符重载函数的声明语句
2. 编写运算符函数的定义

如果在类定义中包含了运算符函数，必须要遵循一定的规则，这些规则将在本节稍后的“运算符函数作为成员函数和非成员函数”小节中说明。

15.2.2 重载运算符：一些限制

重载运算符时要牢记以下几点：

1. 不能改变运算符的优先级。
2. 不能改变运算符的结合律（Associativity）（例如，算术运算符的结合律是自左向右，这点不能改变）。
3. 重载运算符不能使用默认参数。
4. 不能改变运算符所需的参数个数。
5. 不能创建新运算符，只能重载已存在的运算符。不能重载下面运算符：

```
.    .*    ::    ?:    sizeof
```

6. 重载运算符不能改变原有运算符的含义。如重载后的析取运算符>>仍用于输入操作。
7. 不论是用户定义的数据类型的对象，还是用户定义类型对象和内建类型对象的结合，它们的运算符都可以重载。

15.2.3 this 指针

对于给定的对象，成员函数能（直接）访问本类的数据成员。有时，成员函数要访问的是对象的整体，而不是单独的数据成员。定义成员函数时，尤其是没有将对象作为参数传递时，怎样才能将涉及的对象作为一个整体（单独的单元）来引用呢？每个对象都隐含有一个指向自己的指针，指针名为this。在C++中，this是保留字，用户可使用指针this。当对象调用成员函数时，成员函数可引用对象的this指针。例如，假设test是类，并有one成员函数，one的定义如下所示：

```
test test::one()
{
    .
    .
    .
    return *this;
}
```

若x和y都是test类型的对象，那么语句：

```
y = x.one();
```

将对象x的值拷贝给对象y，即x的数据成员拷贝给y相应的数据成员。x调用函数one时，成员函数one中定义的this指针指向对象x。因此，this表示x的地址，*this表示x的值。

下面的例题说明了this指针是怎么工作的。

例 15.1　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;
};
```

假设类 thisPointerClass 的成员函数定义如下所示：

```
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 更新了 x，y 和 z 的值，用指针 this 返回了整个对象的值，即返回了数据成员 x，y 和 z 的值。

```
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
```

大部分输出结果很明显，不需要解释。第5行语句计算表达式object1.updateXYZ()，该表达式更新object1中数据成员的值。object1的值在函数updateXYZ的定义中由指针this返回。然后，赋值运算符将其拷贝给object2。

下面的例题进一步说明this指针是怎么工作的。

例 15.2　例12.9（第12章）在程序中设计实现人名的类。这里扩展类personType的定义，将人的名和姓分开设置，然后返回整个对象。类personType扩展的定义如下所示：

```
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personType的定义中，使用有默认参数的构造函数代替默认的构造函数和有参数的构造函数。

函数print，setTime，getName和构造函数与以前一样（见例12.9）。函数setFirstName和setLastName的定义如下所示：

```
personType& personType::setLastName(string last)
{
      lastName = last;

      return *this;
}

personType& personType::setFirstName(string first)
{
      firstName = first;

      return *this;
}
```

考虑下面的函数main：

```
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行、第2行、第3行语句声明并初始化对象student1，student2和student3，student2和student3初始化为空字符串。第5行语句输出student1的值（见相应输出）。第7行语句执行过程如下。在语句：

```
student2.setFirstName("Shelly").setLastName("Malik");
```

中，先执行表达式：

```
student2.setFirstName("Shelly")
```

这是因为点运算符自左向右结合。此表达式将名字设为"Shelly"并返回对象student2的引用。下一个执行的表达式是：

```
student2.setLastName("Malik")
```

它将student2的姓设为"Malik"。第9行语句输出student2的值。第11行语句将student3的名字设为"Chelsea"，第13行语句输出student3的值。注意该输出，它只有名，而没有姓，这是因为还没有给student3的姓赋值。此时student3的姓仍与在第3行中声明时设置的一样，仍然为空。第15行中设置了student3的姓，第16行输出了student3的值。

15.2.4 类的友元函数

在类的作用域范围之外定义的函数称为友元函数（Friend Function）。友元函数是类的非成员函数，但它能访问类的私有数据成员。若要将函数设置为类的友元函数，必须在函数原型（在类的定义中）的前面加保留字friend（friend只用在类定义中的函数原型前，而不用在友元函数的定义前）。

考虑下面语句：

```
class classIllusFriend
{
    friend void two(...);
        .
        .
        .
};
```

在类classIllusFriend的定义中，函数two声明为类的友元，它是类的非成员函数。在编写函数two的定义时，classIllusFriend类型的任何对象——或是作为函数two的局部变量或是作为形参。在函数two内都能访问该对象的私有成员（例15.3说明了这种情况）。而且，由于友元函数不是类的成员，它的声明可放在private，protected或是public中。

友元函数的定义

在编写友元函数的定义时，在友元函数的函数头前不加类名和作用域运算符。前面说过，在定义友元函数时，也不用在函数头前加关键字friend。classIllusFriend中函数two的定义如下所示：

```
void two(...)
{
    .
    .
    .
}
```

当然，友元函数的定义也可以放在实现文件中。

下一节当讨论为某些类重载运算符时，将说明成员函数和非成员函数（友元函数）的区别。

下面的例题说明了友元函数怎样访问类的私有成员。

例 15.3 考虑下面的类：

```
class classIllusFriend
{
    friend void two(classIllusFriend cLFObject);

public:
    void print();
    void setx(int a);

private:
    int x;
};
```

在类classIllusFriend的定义中，two声明为友元函数。假设类classIllusFriend成员函数的定义如下所示：

```
void classIllusFriend::print()
{
      cout<<"In class classIllusFriend: x = "<<x<<endl;
}

void classIllusFriend::setx(int a)
{
      x = a;
}
```

下面是函数two的定义：

```
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包括形参cLFObject和局部变量localTwoObject，它们都是classIllusFriend类型。在第3行语句中，对象localTwoObject访问它的私有数据成员并将其x设为45。如果函数two不是类classIllusFriend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。同样，在第7行语句中，形参cLFObject访问它的私有成员并将其设为88。与上面一样，如果函数two不是类classIllusFriend的友元函数，此语句也会导致语法错误。第6行语句通过直接访问x，输出localTwoObject的私有成员x的值。同样，第10行语句直接访问并输出cLFObject中x的值。函数two也通过使用函数print打印x的值（见第4行和第8行语句）。

考虑下面main函数的定义：

```
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
```

在大多数情况下，该输出很简单，很容易读懂。例如，第17行语句调用函数two（一个类ClassIllusFriend的友员函数），并通过对象aObject作为实参。注意，函数two生成了输出的最后6行语句，其中包含两行空语句。

15.2.5 运算符函数作为成员函数和非成员函数

本章开头已说明，类定义中包含运算符函数时应遵循一定的规则。本节将说明这些规则。

大部分运算符函数都可以是成员函数或非成员函数——即类的友元函数。用运算符函数作为类的成员和非成员，一定要记住以下几点：

1. 重载运算符( ),[ ]，->或 = 的函数一定要声明为类的成员。
2. 假定某个类，例如OpOverClass，重载运算符op，则：
   a. 若op最左边的操作数是一个不同类型的对象（不是OpOverClass类型），重载运算符op的函数一定要作为非成员——即类OpOverClass的友元。
   b. 若重载运算符op的函数是OpOverClass类的成员，当op用于OpOverClass类型的对象时，op最左边的操作数必须是OpOverClass类型。

类定义中包含运算符函数时，一定要遵循上面的规则。

本章的后面将会看到，为类重载的插入运算符<<和析取运算符>>函数一定要作为非成员——类的友元函数。

除了前面提到的运算符，重载其他运算符时都既可以作为成员函数又可以作为非成员函数。下面的讨论说明了这两种函数的区别。

为便于讨论，使用下面的类说明运算符重载：

```
class OpOverClass
{
        .
        .
        .
private:
   int a;
   int b;
};
```

类 OpOverClass 有两个 int 型私有成员函数 a 和 b。我们将在类 OpOverClass 中增加运算符函数，以便可以重载这些运算符。

有下面的语句：

```
OpOverClass x;
OpOverClass y;
OpOverClass z;
```

即 x，y 和 z 是 OpOverClass 类型的对象。

15.2.6 重载双目运算符

在C++中，有双目和单目运算符，还有一个三目运算符。本节和下面的几节将讨论怎样重载各种双目和单目运算符。首先说明怎样重载双目运算符。

假设为类 OpOverClass 重载双目运算符 +。此运算符可作为成员函数或友元函数重载，下面分别用这两种方式重载此运算符。

作为成员函数重载 +

运算符 + 作为类 OpOverClass 的成员函数重载，重载 + 的函数名是：

```
operator+
```

由于 x 和 y 是 OpOverClass 类型的对象，可以进行下面的操作：

```
x + y
```

编译器将此语句解释为：

```
x.opertor+(y)
```

此表达式中，函数 operator+ 只有一个参数 y。

由于 operator+ 是类 OpOverClass 的成员，x 是 OpOverClass 的对象，在上面的语句中，operator+ 直接访问 x 的私有成员。因此，operator+ 的第一个操作数是调用 operator+ 的对象，第二个操作数作为参数传递给运算符函数。

假设 operator+ 的功能是将两个对象中对应的数据成员相加，则表达式：

```
x + y
```

的结果是 OpOverClass 类型的对象，即运算符函数 operator+ 的返回值是 OpOverClass 类型。

下面为类 OpOverClass 编写重载 operator+ 的运算符函数的原型与定义。

在类 OpOverClass 定义中，operator+ 的函数原型是：

```
OpOverClass operator+(const opOverClass& ) const;
```

下面讨论怎么为类 OpOverClass 编写 operator+ 的定义。

函数 operator+ 求两个对象相对应的数据成员的和。例如，表达式：

```
x+y
```

的结果是：x 的数据成员 a 与 y 的数据成员 a 相加；x 的数据成员 b 与 y 的数据成员 b 相加。

由于 operator+ 是类 OpOverClass 的成员，编写函数 operator+ 的定义时，类名（OpOverClass）与作用域运算符::必须出现在函数头中。为类 OpOverClass 定义的函数 operator+ 如下：

```
OpOverClass OpOverClass::operator+
                        (const OpOverClass& otherObject) const
{
      OpOverClass temp;

      temp.a = a + otherObject.a;
      temp.b = b + otherObject.b;

      return temp;
}
```

在函数定义中，我们将对象的数据成员与 otherObject 相应的数据成员相加，结果存储在局部变量 temp 中。最后，返回对象 temp 的值。注意函数 operator+ 的返回值类型是 OpOverClass。

同样，我们可以重载其他双目算术运算符。

作为成员函数重载双目（算术）运算符的一般语法

下面说明作为类的成员函数重载双目（算术）运算符的一般形式。

函数原型（包含在类的定义中）：

```
returnType operator op(const className&) const;
```

op 表示被重载的双目运算符，returnType 是函数返回值的类型，className 是正被重载的运算符的类名。

函数定义：

```
returnType className::operator op
                     (const className& otherObject) const
{
    //algorithm to perform the operation

    return (value);
}
```

下面的例子说明了怎样重载和使用双目运算符。

例 15.4 *作为成员函数，为类 OpOverClass 重载运算符 + 和 *。*

```
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;
};
```

print，operator+，operator*和构造函数的定义如下所示：

```
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 = (276, 450)
```

函数 main 很简单，很容易读懂。例如，第 6 行语句输出 u 的值。同样，第 9 行语句输出 v 的值。第 11 行语句计算 u 和 v 的和，并将结果赋给 w1；第 15 行语句将 u 和 v 相乘的结果赋给 w2。

作为成员函数重载关系运算符

现在说明怎样作为类的成员函数重载关系运算符。为了便于说明，为类 OpOverClass 重载恒等运算符 ==。

由于关系运算符的结果是 true 或 false，所以函数 operator== 的返回值是 bool 类型。

类 OpOverClass 定义中包含的函数原型语法是：

```
bool operator==(const OpOverClass&) const;
```

如果两个对象（同一类型）相应的数据成员相同，一般来说，这两个对象也相同。因此，函数 operator== 的定义如下所示：

```
bool OpOverClass ::operator==
            (const OpOverClass& right) const
{
    return(a == right.a && b == right.b);
}
```

作为成员函数重载双目关系运算符的一般语法

下面说明作为类的成员函数重载双目关系运算符的一般形式。

函数原型（包含在类的定义中）：

```
bool operator op(const className&) const;
```

op 代表要重载的关系运算符，className 是重载运算符 op 的类名。

函数定义：

```
bool className::operator op(const className& right) const
{
      //Compare and return the value
}
```

下面的例题说明了关系运算符的重载和使用。

例 15.5 在本例中，作为类 OpOverClass 的成员函数重载了运算符 == 和!=。

类 OpOverClass 和函数的定义如下所示：

```
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 15-4
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
```

作为非成员函数的双目运算符

假定运算符 + 作为类 OpOverClass 的非成员函数重载。并假设要执行下面的语句：

```
x + y
```

在本例中，表达式将被编译为：

```
operator+(x, y)
```

函数 operator+ 有两个形参。很明显，在上面表达式中，operator+ 既不是对象 x 的成员，也不是对象 y 的成员。需要相加的对象作为参数传给函数 operator+。

若要将运算符函数operator+作为非成员函数包含在类的定义中，保留字friend一定要出现在函数头的前面（即作为友元函数）。当然，函数 operator+ 一定要有两个形参。

由此，若 operator+ 作为非成员函数包含在类 OpOverClass 的定义中，则在 OpOverClass 的定义中，它的原型为：

```
firend OpOverClass operator+(const OpOverClass&, const OpOverClass&);
```

函数 operator+ 的定义如下所示：

```
OpOverClass operator+(const OpOverClass& firstObject,
                      const OpOverClass& secondObject)
{
      OpOverClass temp;

      temp.a = firstObject.a + secondObject.a;
      temp.b = firstObject.b + secondObject.b;

      return temp;
}
```

在上面的定义中，将 firstObject 和 secondObject 中相对应的数据成员相加，结果存储在 temp 中。前面说过，类的私有成员对类来说是局部变量，不能从类外访问。按照这个原则，operator+ 不是类的成员，在函数 operator+ 的定义中，由于 a 是 firstObject 的私有成员，表达式 firstObject.a 是非法语句。然而，由于operator+ 声明为OpOverClass 的友元函数，在它的定义中，可以访问 OpOverClass 类对象的私有成员。注意在函数头中，在函数名 operator+ 之前，没有类名（OpOverClass）和作用域运算符，这是因为函数 operator+ 不是类的成员。

作为非成员函数重载双目（算术）运算符的一般语法

下面说明作为类的非成员函数重载双目运算符的一般形式。

函数原型（包含在类定义中）：

```
firend returnType operator op(const className&,
                              const className&);
```

op代表被重载的双目运算符，returnType是函数返回值的类型，className是需要重载运算符的类名。

函数定义：

```
returnType operator op(const className& firstObject,
                       const className& secondObject)
{
   //algorithm to perform the operation
   return (value);
}
```

作为非成员函数重载关系运算符

下面说明怎样作为非成员函数为类重载关系运算符。为简单起见，为类OpOverClass重载恒等运算符==。

很明显，函数operator==的返回类型是bool型。

函数原型（包含在类OpOverClass的定义中）：

```
friend bool operator==(const OpOverClass&,
                       const OpOverClass&);
```

函数定义：

```
bool operator==(const OpOverClass& firstObject,
                const OpOverClass& secondObject)
{
      return(firstObject.a == secondObject.a &&
             firstObject.b == secondObject.b);
}
```

作为非成员函数重载双目关系运算符的一般语法

这里讨论作为类的非成员函数重载双目关系运算符的一般形式。

函数原型（包含在类定义中）：

```
friend bool operator op(const className&, const className&);
```

op代表重载的关系运算符，className是需要重载运算符op的类名。

函数定义：

```
bool operator op (const className& firstObject,
                  const className& secondObject)
{
      //Compare and return the value
}
```

15.2.7　重载流插入和流析取运算符

重载流插入运算符（<<）和流析取运算符（>>）的函数必须是类的非成员函数，其原因有以下几点。

考虑下面的表达式：

```
cin>>x
```

在此表达式中，运算符>>最左边的操作数（cin）是istream型对象，而不是OpOverClass类型对象。由于>>最左边不是OpOverClass类型，重载的析取运算符函数必须是类OpOverClass的非成员函数。

同样，为类OpOverClass重载流插入运算符的函数也必须是类OpOverClass的非成员函数。

重载流插入运算符

下面说明为类重载流插入运算符<<的一般语法。

函数原型（包含在类的定义中）：

```
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;
}
```

在上面的函数定义中：

- 两个参数都是引用参数。
- 第一个参数（osObject）是一个 ostream 类型对象的引用。
- 第二个参数通常是特定类的常量引用。因为（参见 12 章）若将一个对象作为参数传递给类，最有效的方法是通过传递引用。在这种情况下，形参不需要拷贝实参的数据成员。在上面的定义中，类名前之所以有 const，是因为我们只要输出对象的数据成员，不想修改对象的数据成员。
- 函数的返回类型是 ostream 对象的引用。

由于如下原因，运算符<<的重载函数的返回值必须是 ostream 对象的引用。

假定类 OpOverClass 重载了运算符<<，语句：

```
cout << x;
```

等同于语句：

```
operator << (cout, x);
```

这当然是合法的语句，因为两个实参都是对象，而非对象的值。该函数的第一个参数cout是ostresm类型；第二个参数 x 是 OpOverClass 类型。

现在，考虑下面语句：

```
cout << x << y;
```

上面语句等同于语句：

```
operator<<(operator<<(cout, x), y);  //Line A
```

由于运算符<<的结合性是自左向右，上面语句中首先执行表达式：

```
cout<<x
```

即表达式：

```
oprerator<<(cout,x)
```

在计算完上面表达式之后，输出 x 的值，无论函数 operator<<返回的是什么，它都会成为下一个运算符<<左边的参数（作为函数 operator<<的第一个参数），对象 y 的值的输出过程也是一样的。由于 operator<<左边的参数必须是 ostream 类型的对象，表达式：

```
cout<<x
```

必须返回 cout 类型对象，以便输出 y 的值。

因此，函数 operator<<的返回值必须是 ostream 类型对象的引用。

重载流析取运算符（>>）

下面是为类重载析取运算符>>的一般语法。

函数原型（包含在类的定义中）：

```
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;
}
```

在上面的函数定义中：

- 两个参数都是引用参数。
- 第一个参数（isObject）是istream对象的引用。
- 第二个参数通常是某个特定类的引用。
- 函数的返回类型是istream类型对象的引用。

与前面的原因一样（重载插入运算符<<），函数operator>>的返回类型必须是istream对象的引用。使得类似于下面的语句可以成功执行：

```
cin>>x>>y;
```

下面的例子用来说明为类OpOverClass重载插入和析取运算符。

例15.6 类OpOverClass与运算符函数的定义如下所示：

```
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 15-4

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;
}
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
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 行和第 2 行语句将 u 和 v 声明为 OpOverClass 类型对象，并将其初始化。第 3 行语句用 cout 和插入运算符输出 u 的值，第 5 行语句使用 cin 和析取运算符将数据输入到 v 中，第 7 行语句用 cout 和插入运算符输出 v 的值。第 8 行语句将 u 和 v 相加，并使用 cout 输出结果。同样，第 9 行语句将 u 和 v 相乘并输出结果。输出结果表明流插入和流析取运算符重载成功。

15.2.8　重载赋值运算符

赋值运算符（ = ）是类所允许的重载运算符之一。赋值运算符基于成员（member-wise）拷贝类的数据成员。例如，语句：

```
x = y;
```

等同于语句：

```
x.a = y.a;
x.b = y.b;
```

第 14 章讲过，当类没有指针数据成员时，内建的赋值运算符可以正常使用。因此，为避免有指针数据成员的类的浅拷贝，必须要明确重载赋值运算符。

前面讲过，若要重载类的赋值运算符 =，运算符函数 operator = 必须是类的成员。

为类重载赋值运算符 = 的一般语法

为类重载赋值运算符 = 的一般语法说明如下。

函数原型（包含在类的定义中）：

```
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
- 形参通常是某个特定类的常量引用
- 函数的返回类型是对某个特定类的常量引用

下面解释为什么函数 operator = 的返回类型必须是类的引用。

假定类 OpOverClass 已重载赋值运算符 =，语句：

```
x = y;
```

等同于语句：

```
x.operator = (y);
```

也就是说对象 y 是函数 operator = 的实参。

再看语句：

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

由于运算符 = 的结合律是自右向左，此句等同于语句：

```
x.operator=(y.operator=(z));            //Line B
```

显然，首先执行表达式：

```
y.operator=(z)
```

也就是表达式：

```
y = z
```

表达式：

```
y.operator=(z)
```

的返回值成为函数 operator = 的参数，以便对 x 赋值（见上面语句 Line B）。由于函数 operator = 的形参是引用参数，表达式：

```
y.operator=(z)
```

一定要返回对象的引用，而不是它的值。就是说，它要返回对象 y 的引用，而不是 y 的值。由于这个原因，重载的赋值运算符 = 函数的返回值必须是类的引用。

考虑下面语句：

```
x = x;
```

x 试图给 x 赋值，即这是一个自赋值语句。为了避免浪费计算机时间，此类语句一定要避免。

operator = 的函数体确实能防止这类赋值，原因如下所述。

在 operator = 的函数体中，if 语句：

```
if(this != &rightObject)  //avoid self-assignment
{
       //algorithm to copy rightObject into this object
}
```

语句：

```
x = x;
```

编译为：

```
x.operator=(x);
```

对象 x 调用函数 operator =，operator = 函数体的指针 this 指向对象 x，并且 x 也是 operator = 的参数，形参 rightObject 也引用对象 x。因此，表达式：

```
this != &rightObject
```

this 是 x 的地址，&rightObject 也是 x 的地址。表达式的值为 false，if 语句里面的内容被跳过。

下面的例子说明怎样重载赋值运算符。

例 15.7　*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;
};
```

类 cAssignmentOprOverload 成员函数的定义如下所示:

```
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
}
```

重载的赋值运算符函数工作过程如下。第1行语句检查对象是否自我赋值。第2行语句检查list是否为非空。如果是非空的，则释放list所占有的内存。第4行和第5行语句将otherList的数据成员maxSize和length的值分别赋给list的相应数据成员maxSize和length。若otherList不是NULL而且非空，第6行到第10行的语句创建动态数组list，并将otherList拷贝给list。若otherList是NULL，list初始化为NULL。由于函数operator =的返回类型是引用，第13行语句返回此表的地址。

下面的函数测试类cAssignmentOprOverload：

```
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;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
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intList1的大小为10，第15行和第16行语句创建大小为50（默认值）的intList2和intList3。第20行至第22行语句将数据输入到intList1。第24行语句输出intList1。第25行语句先将intList1拷贝到intList2，然后将intList2拷贝到intList3。第27行语句输出intList2（见相应程序运行结果中的输出）。第28行语句删除intList2。第31行语句输出intList2，intList2为空（见相应程序运行结果中的输出）。删除intList2之后，程序输出intList1和intList3的内容（见相应程序运行结果中的输出）。从程序运行结果中可以清楚地看到，intList2的删除不影响intList1和intList3，这是因为intList1和intList3各自都有自己的数据。

15.2.9 重载单目运算符

重载单目运算符与重载双目运算符很相似，它们惟一的区别在于双目运算符有两个操作数，单目运算符有一个操作数。因此，为一个类重载单目运算符：

1. 若运算符函数是类的成员，则它没有参数。
2. 若运算符函数是非成员——类的友元函数，它有一个参数。

下面说明怎样重载增量与减量运算符。

15.2.10 重载增量运算符和减量运算符

增量运算符（++）有两种形式：前置增量(++u)和后置增量(u++)。这里，u是int型变量。使用前置增量++u时，在使用u值之前，u的值加1。使用后置增量u++时，先在表达式中使用u的值，然后再将u的值加1。

重载前置增量运算符

重载前置增量运算符的方法比较简单。在函数定义中，先增加对象的值，然后用this指针返回对象的值。例如，假设需要为类OpOverClass重载前置增量运算符，并且运算符函数operator++是类的成员。运算符函数operator++没有参数，因此可以用this返回增加过值的对象（为便于说明，假定增量运算符将类OpOverClass的数据成员a和b的值加1）。

```
OpOverClass OpOverClass::operator++()
{
      //increment the object
```

```
    ++a;
    ++b;

   return *this;     //return the incremented value of the object
}
```

假设 x 是 OpOverClass 类型的对象，语句：

```
++x;
```

将 x 的值加 1，与 x 相关的 this 指针返回增加过的 x 的值。同时，由于++x 不是出现在表达式中，由 *this 返回的值将丢失。如果 y 也是 OpOverClass 类型的对象，语句：

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

将 x 的值加 1，this 指针返回增加过的 x 的值，并存储在 y 中。

作为成员函数重载前置增量运算符的一般语法

下面说明作为成员函数重载前置增量运算符的一般语法。

函数原型（包含在类的定义中）：

```
className operator++();
```

函数定义：

```
className className::operator++()
{
     //increment the value of the object by 1
   return *this;
}
```

重载前置增量运算符的函数也可以作为类 OpOverClass 的非成员函数。下面说明这种用法。

由于运算符函数 operator++ 是类 OpOverClass 的非成员函数，它有一个参数，且是 OpOverClass 类型的对象（为便于说明，假定增量运算符使类 OpOverClass 的数据成员 a 和 b 的值增加 1）。

```
OpOverClass operator++(OpOverClass& incObj)
{
      //increment the object
   (incObj.a)++;
   (incObj.b)++;

   return incObj;     //return the incremented value of the object
}
```

作为非成员函数重载前置增量运算符 ++ 的一般语法

下面说明作为非成员函数重载前置增量运算符 ++ 的一般语法。

函数原型（包含在类的定义中）：

```
friend className operator++(className&);
```

函数定义：

```
className operator++(className& incObj)
{
     //increment incObj by 1
   return incObj;
}
```

重载后置增量运算符

现在讨论怎样重载后置增量运算符。在介绍前置增量运算符时，我们首先讨论的是作为类的成员重载此运算符。

下面为类 opOverClass 重载后置增量运算符。不管是前置增量还是后置增量，运算符函数的名字完全一样——operator++。为了区分前置增量和后置增量运算符重载，在运算符函数的函数头部增加了一个（int 型）伪参数。因此，类 OpOverClass 的后置增量运算符的函数原型是：

```
OpOverClass operator++(int);
```

编译器将语句：

```
x++;
```

编译为：

```
x.operator++(0);
```

可以看出，函数执行时带一个参数。参数 0 仅用来区分前置、后置增量运算符函数。

后置增量运算符在表达式中先使用对象的值，再将对象的值加 1。所以，函数的执行步骤是：

1. 存储对象的值——假设在 temp 中。
2. 将对象的值加 1。
3. 返回 temp 中存储的值。

类 OpOverClass 的后置增量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
}
```

作为成员函数重载后置增量运算符的一般语法

下面说明作为成员函数重载后置增量运算符的一般语法。

函数原型（包含在类的定义中）：

```
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
}
```

后置增量运算符也可作为类的非成员函数重载。此时，运算符函数 operator++ 有两个参数。作为类 OpOverClass 的非成员重载后置增量运算符，其定义如下所示：

```
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
}
```

作为非成员函数重载后置增量运算符++的一般语法

下面说明作为非成员函数重载后置增量运算符++的一般语法。

函数原型（包含在类的定义中）：

```
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
}
```

减量运算符可用类似的方法重载，具体的细节留给读者作为练习。

下面是类OpOverClass的定义，运算符函数包含在其中。由于某些运算符函数既可以作为成员又可以作为非成员重载，下面给出了类OpOverClass两个相等的定义。在第一个定义中，增量、减量、算术及关系运算符作为成员函数重载；在第二个定义中，增量、减量、算术及关系运算符作为非成员函数重载。

类OpOverClass的定义如下所示：

```
//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;
};
```

下面也是类 OpOverClass 的定义，但算术、增量、减量及关系运算符作为类的非成员函数重载：

```
//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;
};
```

类 OpOverClass 的重载函数的定义留给读者作为练习（见本章后面的编程练习 1 和编程练习 2）。

15.2.11　运算符重载：成员与非成员的比较

前面的章节讨论并说明了怎样重载运算符。一些运算符必须作为类的成员函数重载，另一些必须作为类的非成员（友元）重载。既可以作为成员又可作为非成员函数的运算符的情况又怎么样呢？例如，双目算术运算符 + 既可作成员又可作非成员函数重载。若作为成员函数重载运算符 +，则 + 可直接访问其中一个对象的数据成员，并且仅需将另一个对象作为参数传递。若作为非成员重载 +，必须将两个对象作为参数传递。因此，作为非成员重载 +，需要额外的内存和计算机时间以获得数据的拷贝。因此，为了提高效率，只要有可能，就应作为成员函数重载运算符。

15.2.12　类和指针数据成员（再叙）

在第 14 章中，已讨论了有指针数据成员的类的特性。既然已经讨论了怎样重载各种运算符，为了避免由于这些特性造成错误，我们再次回忆一下有指针数据成员的类的特性。

前面已经讲过，类内建的运算符只有赋值运算符和成员访问运算符。赋值运算符提供了基于成员来拷贝数据，也就是一个对象的数据成员拷贝给同类型的另一个对象相应的数据成员。我们已经知道，如果类有指针数据成员，则这种成员方式拷贝可能会出问题。当对象超出作用域以及类的对象作为参数传值时，释放与指针数据成员相关的动态内存就会出现问题。为解决这些问题，有指针数据成员的类必须：

1. 明确重载赋值运算符
2. 带有拷贝构造函数
3. 带有析构函数

15.2.13　运算符重载：结束语

下面通过三个范例来说明运算符重载。在考虑这些范例之前，应该记住：假如为类（比如说 OpOverClass）重载运算符 op，只要对 OpOverClass 类型的对象使用运算符 op，则该类的重载运算符 op 的函数体就会执行。

15.3 程序范例：clockType

第12章定义了类clockType，用来在程序中实现时间功能。通过相应函数实现输出时间、增加时间、比较两个时间是否相等。该例重新定义了类clockType，为了便于输入输出，重载了流插入和流析取运算符；为了比较，重载了关系运算符；为了按秒增加时间，重载了增量运算符。使用类clockType的程序要求用户以hr:min:sec的形式输入时间。

修改过的类clockType的定义如下所示：

```
//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
```

下面编写的是函数定义，以实现类clockType的操作。注意类clockType仅重载了前置增量运算符。为了一致，也应当重载后置增量运算符，这一步留给读者作为练习（见本章末尾的练习3）。

先编写函数operator++的定义。将时间增加一秒的算法如下所示：

a. 将时间增加一秒。

b. 如果秒>59。

  b.1. 将秒设置为0

  b.2. 将分钟增加1

  b.3. 如果分钟>59

    b.3.1. 将分钟设置为0

    b.3.2. 将小时加1

    b.3.3. 如果小时>23

      b.3.3.1 将小时设为0

c. 返回对象的值。

函数operator++的定义如下所示：

```
      //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
}
```

函数operator==的定义很简单。如果两个时间的小时、分钟和秒均相同，则两个时间相等。因此，函数operator==的定义如下所示：

```
      //overload the equality operator
bool clockType::operator==(const clockType& otherClock) const
{
      return(hr == otherClock.hr && min == otherClock.min
```

```
            && sec == otherClock.sec);
}
```

下面给出的是函数 operator<=定义。当下述条件满足时，则第一个时间小于或等于第二个时间。

1. 第一个时间的小时小于第二个时间的小时。
2. 第一个时间与第二个时间的小时数相等，但第一个时间的分钟数小于第二个时间。
3. 第一个时间与第二个时间的小时与分钟数都相等，但第一个时间的秒钟数小于或等于第二个时间。

函数 operator<= 的定义如下所示：

```
    //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));
}
```

构造函数和函数 setTime 的定义与第 12 章中的一样。为了完整，这里也将它们包含进来。

```
      //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;
}
```

现在开始讨论函数 operator<<，时间输出的形式必须是：

```
hh:mm:ss
```

以此格式输出时间的算法与第 12 章中 clockType 的 printTime 函数体中的一样。在以上面的格式输出时间之后，还必须返回 ostream 对象。因此，函数 operator<<的定义如下所示：

```
      //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
}
```

下面讨论函数 operator>>的定义。程序输入的格式是：

```
hh:mm:ss
```

也就是，输入小时后跟着冒号，然后是分钟，然后是冒号，再后是秒。很明显，输入时间的算法是：

a. 读取输入，这是一个数，将它存储在数据成员 hr 中。
b. 读取下一个输入，这是一个冒号，将它丢弃。
c. 读取下一个输入，这是一个数，将它存储在数据成员 min 中。
d. 读取下一个输入，这是一个冒号，将它丢弃。
e. 读取下一个输入，这是一个数，将它存储在数据成员 sec 中。
f. 返回 istream 对象。

显然，需要一个 char 型局部变量来读取冒号。
函数 operator>>的定义如下所示：

```
      //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
}
```

下面的测试程序用来测试类 clockType：

```
//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;
}
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
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
```

15.4　程序范例：复数

如果数的形式为$a+ib$，且$i^2=-1$，a和b是实数，则此数称为复数（Complex Numbers）。a称为$a+ib$的实部，b称为虚部。复数也可用有序数对（a，b）表示。复数的加法和乘法定义如下所示：

```
(a + ib) + (c + id) = (a + c) + i(b + d)
(a + ib) * (c + id) = (ac - bd) + i(ad + bc)
```

若使用有序数对，规则可写为：

```
(a,b) + (c,d) = ((a + c), (b + d))
(a,b) * (c,d) = ((ac - bd), (ad + bc))
```

C++没有内建操作复数的数据类型。此范例将构造数据类型complexNumber，可以用它来处理复数。为了便于输入输出，重载了流插入和流析取运算符。同时，还重载了+和*来执行复数的加法和乘法。若x和y是复数，可以直接对表达式$x+y$和$x*y$求值。

```
//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
```

下面开始编写函数的定义以实现类complexType的各种操作。因为很多函数的定义都很简单，所以这里只讨论流插入运算符<<和流析取运算符>>的重载函数定义。

复数的输出形式是：

```
(a, b)
```

这里，a是实部，b是虚部。显而易见，算法是：

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)

因此，函数operator<<的定义如下所示：

```
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
}
```

下一步讨论重载流析取运算符>>函数的定义。

输入的形式是：

```
(3, 5)
```

这里，复数的实部是3，虚部是5。显而易见，读复数的算法是：

a. 读并丢弃左括号

b. 读并存储实部

c. 读并丢弃逗号

d. 读并存储虚部

e. 读并丢弃右括号

按照这些步骤，函数operator>>的定义如下所示：

```
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所示：

```
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;
}
```

下面的程序说明了类 complexType 的使用方法：

```
//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;
}
```

程序运行结果 *在本程序运行中，用户输入的数据加有阴影。*

```
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)
```

可以扩展此数据类型以处理复数的减法和除法。

下面定义的类称为newString，并重载其赋值和关系运算符。也就是说，当声明一个newString类型的变量时，可以直接使用赋值运算符将一个字符串拷贝给另一个字符串，也可以使用关系运算符比较两个字符串。

在讨论类newString之前，先来了解一下重载的运算符[ ]。前面说过，可以使用运算符[ ]访问数组的成员。为了在newString类型的字符串中访问每个字符，必须重载类newString的运算符[ ]。

15.5 重载数组下标运算符

前面讲过，重载运算符[ ]的函数必须是类的成员函数。而且，由于数组能声明为常量或非常量（Nonconstant），重载的运算符[ ]要能处理这两种情况。

对于非常量数组，作为类成员声明运算符函数 operator[ ]的语法是：

```
Type& operator[] (int index);
```

对于常量数组，作为类成员声明运算符函数 operator[ ]的语法是：

```
const Type& operator[] (int index) const;
```

这里，Type 是数组元素的数据类型。

假如 classTest 是带有数组数据成员的类，重载运算符函数[ ]的 classTest 的定义如下所示：

```
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 是数组元素的数据类型。

为 classTest 重载运算符[ ]函数的定义如下所示：

```
   //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
}
```

考虑下面语句：

```
classTest list1;
classTest list2;
const classTest list3;
```

在语句：

```
list1[2]=list2[3];
```

中，执行了非常量数组的运算符函数 operator[ ]的函数体。在语句：

```
list1[2]=list3[5];
```

中，首先执行的是常量数组的运算符函数operator[ ]的函数体，这是因为list3是常量数组。然后，执行的是非常量数组的运算符函数operator[ ]的函数体，并完成该赋值语句。

15.6 程序范例：newString

第9章讨论了C-string。

1. C-string是一个或多个字符的序列
2. C-string用双引号括住
3. C-string以空字符('\0')结尾
4. C-string存储在字符数组中

在本程序范例中，字符串都是指C-string。

允许对字符串进行的全部操作是输入和输出。若需要其他操作，程序员应包含头文件cstring，cstring中包含了大量操作字符串的函数的说明。

最初，C++并不提供处理字符串的内建数据类型。但在最近的版本中，C++提供了字符串类以处理和操作字符串。

本程序范例的目的是定义自己的类来处理字符串并进一步说明运算符重载。明确地说，要重载赋值运算符、关系运算符，及为了便于输入输出而重载流插入和流析取运算符。此类名为newString。下面先给出类newString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
```

类 newString 有两个私有数据成员：一个存储字符串，一个存储字符串的长度。

下一步给出了函数定义以实现newString的操作。因为要用到函数assert，实现文件中包含了头文件cassert。函数 assert 的说明见第 4 章或附录 F 中的头文件 cassert。

```
//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;
}
```

考虑函数 operator>>定义中的语句：

```
is>>setw(81)>>temp;
```

由于 temp 声明为大小是 81 的数组，temp 能存储的字符串的最大长度是 80。此语句（输入语句）中的控制符 setw 确保读入 temp 的字符不超过 80 个。

大多函数都浅显易懂，这里只介绍重载的转换构造函数、赋值运算符及拷贝构造函数。

转换构造函数（Conversion Constructor）是单参数函数，它将参数转换为构造函数所在的类对象。在本程序范例中，转换构造函数将字符串转换为 newString 类型的对象。

注意，在此所重载的赋值运算符只显式地用于 newString 类型的对象。但是，重载的赋值运算符也可用于存储一个字符串到 newString 对象。考虑变量声明：

```
newString str;
```

及语句：

```
str = "Hello there";
```

编译器将此句转换为：

```
str.operator=("Hello there");
```

1. 首先，编译器自动调用转换构造函数创建newString类型的对象，以临时存储字符串"Hello there"。
2. 然后，编译器调用赋值运算符，将临时 newString 对象赋给 str 对象。

因此，若将一个字符串存储到 newString 类型的对象中，不需要明确重载赋值运算符。

下面的 C++ 程序用以测试类 newString 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;
```

程序运行结果 *在本程序运行中，用户输入的数据加有阴影。*

```
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
```

该程序的工作过程如下。第1行语句输出str1，str2和str3的值，注意str3的值打印在***和###之间。由于str3为空，***和###之间没有打印任何内容，见相应的输出。第2行到第5行语句比较str1和

str2，并输出结果。第7行语句将最小长度为7的字符串输入到str1中。第8行语句输出str1中的值。注意下面的语句（见第9行）：

```
str4 = str3 = "Birth Day";
```

由于赋值运算符的结合律是自右向左，所以先执行语句str3 = "Birth Day";再执行语句str4 = str3;第10行语句输出str3和str4的值，第14行、第16行、第18行语句使用数组下标运算符[ ]来分别处理str3中的字符。其他的语句都很容易理解。

15.7 函数重载

前面章节讨论了运算符重载，运算符重载为程序员提供了简明的表示法，使得用户定义的数据类型与内建类型的运算符完全一样。运算符的参数类型决定采取什么操作。同运算符重载一样，C++允许程序员重载函数名。第7章已经介绍了函数重载，为了在下面的讨论中便于理解，现在复习一下这个概念。

前面说过，类可以有多个构造函数，但所有的构造函数都必须有相同的名字，即类名。这是函数重载的一个例子。回忆一下，多个重载函数有相同的函数名，但参数不同。参数的类型决定选择执行哪个函数。

为使重载函数能够运行，必须给出每个函数的定义。下一节将说明怎样用单一代码段重载函数，而把生成各个函数功能代码的工作交给编译器去做。

15.8 模板

模板是C++中的利器。为一系列相关函数编写的一段独立代码，称为函数模板（Function Template）；为一系列相关类编写的一段独立代码，则称为类模板（Class Template）。模板的语法如下所示：

```
template <class Type>
   declaration;
```

其中Type是数据类型，declaration是函数或类的声明。在C++中，template是保留字。模板头部中的class用来引用任意的用户定义类型或是内建类型，Type是模板的形参。与变量是函数的参数类似，类型（数据类型）是模板的参数。

15.8.1 函数模板

第7章介绍了函数重载。经过重载函数larger可以分别用于计算两个整数、字符、浮点数或字符串中的较大值。为了能够重载函数larger，必须为下面4种数据类型分别编写函数定义：int型、char型、double型和string型。并且，每个函数的函数体非常相似。C++通过提供函数模板简化了重载函数的过程。

函数模板的语法是：

```
template <class Type>
function definition;
```

其中Type作为模板的形参，用于确定函数的参数类型及返回值类型，还用于在函数中声明变量。语句：

```
template <class Type>
Type larger(Type x, Type y)
{
   if(x >= y)
      return x;
```

```
    else
        return y;
}
```

定义了函数模板larger，它返回两个数据项中较大的一个。在函数头中，形参x和y的类型是Type。函数调用时，它由实际参数的类型确定。语句：

```
cout << larger(5, 6) << endl;
```

调用了函数模板larger。由于5和6是int类型，数据类型int取代Type并且编译器为其产生相应代码。若忽略函数模板定义的函数体，函数模板就可以作为原型使用。

下面的例子说明了函数模板的使用。

例15.8 本例中，用函数模板larger来确定两个数据项中较大的一个。

```
#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
```

15.8.2 类模板

类模板与函数模板相似，类模板是为一系列相关类编写的同一个代码段。例如，第 12 章中定义了作为抽象数据类型（ADT）的表，表的元素类型是 int，如果元素类型由 int 变为 char，double 或 string，就必须为每个数据类型编写单独的类。在很多地方，表的操作及实现这些操作的算法是相同的。使用类模板，可以创建一般类 listType，编译器为特定的应用产生相应源代码。

类模板的语法是：

```
template<class Type>
class declaration
```

因为根据参数类型产生特定的类，类模板又被称为参数化类型（Parameterized Type）。

下面的语句定义 listType 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 的定义是一般性的定义，仅包含表的基本操作。该类模板将存储表元素的数组和表长度声明为 protected，是为了可从此表中派生出特定的表，并可以增添和重写操作。

下一步，我们描述一个特定表。假定需要创建一个表处理整数数据。语句：

```
listType<int> intList;        //Line 1
```

将intList声明为有100个成员的表，每个成员都是int型。同样，语句：

```
listType<newString> stringList;        //Line 2
```

将stringList声明为有100个成员的表，每个成员都是newString类型。

在第1行和第2行语句中，listType<int>与listType<newString>称为类模板listType<elemType>的模板实例化（Template Instantiation）简称为实例化。其中elemType是模板头中的类参数。模板实例化可用内建或用户定义的类型创建。

类模板的函数成员可看做函数模板。因此，在给出类模板函数成员的定义之后，必须要给出函数模板的定义。例如，类listType的成员insert的定义如下所示：

```
template<class elemType>
void listType<elemType>::insert(elemType newElement)
{
   .
   .
   .
}
```

在成员函数的函数头中，类名由参数elemType确定。

第1行语句将intList声明为有100个元素的表。编译器为intList生成代码时，用int代替在类listType的定义中出现的elemType。类listType成员函数定义中的模板参数（例如，insert定义中的elemType），也被int代替。

类模板的头文件与实现文件

到现在为止，我们都是将类定义（在头文件中）和成员函数定义（在实现文件中）分别放在单独的文件中。目标代码从实现文件中生成并和用户代码连接。但是，这种类和成员函数分别定义的机制并不适于类模板。因为，给函数传递参数是在运行时发生的，而给模板传递参数是在编译时发生的。由于类的实参是在用户编写的代码中指定的，并且若没有实参传递给模板的话，编译器不能实例化函数模板。因此，不能没有用户的代码而单独编译实现文件。

可以有几种方案来解决这个问题。可以将类定义和函数模板定义直接放在用户的代码中，也可将类定义和函数模板定义放在同一个头文件中。另一种解决方法是将类定义和函数定义放在各自的文件中（和多数用法一样），但在头文件的结尾，加上包含实现文件的预处理指令。无论哪种方法，函数定义和用户的代码都被一起编译。为便于说明，本书将函数定义和类定义放在同一个头文件中。

15.8.3 基于数组的表（再叙）

在第14章中，我们设计了类arrayListType来处理存储在数组中的表。然而，第14章中设计的arrayListType类只能处理int型元素的表。在介绍完怎样使用类模板创建一般（类属）模板之后，本节将重新设计类arrayListType以使其能处理任意类型的表。本章已经讨论过怎样重载赋值运算符，由于类arrayListType有一个指针数据成员。因此，除了第14章讨论过的操作外，我们还将重载赋值运算符。

下面的类模板将基于数组的表定义为抽象数据类型（ADT）：

```
template <class elemType>
class arrayListType
{
public:
```

```
 const arrayListType<elemType>&
               operator=(const arrayListType<elemType>&);
   //overload the assignment operator

 bool isEmpty();
   //Returns true if the list is empty;
   //otherwise, returns false
 bool isFull();
   //Returns true if the list is full;
   //otherwise, returns false
 int listSize();
   //Returns the size of the list, that is, the number
   //of elements currently in the list
 int maxListSize();
   //Returns the maximum size of the list, that is, the
   //maximum number of elements that can be stored in
   //the list
 void print() const;
   //Outputs the elements of the list
 bool isItemAtEqual(int location, const elemType& item);
   //If the item is the same as the list element at the
   //position specified by the location, returns true;
   //otherwise, returns false
 void insertAt(int location, const elemType& insertItem);
   //Inserts an item in the list at the specified location.
   //The item to be inserted and the location are passed
   //as parameters to the function.
   //If the list is full or the location is out of range,
   //an appropriate message is displayed.
 void insertEnd(const elemType& insertItem);
   //Inserts an item at the end of the list. The parameter
   //insertItem specifies the item to be inserted.
   //If the list is full, an appropriate message is
   //displayed.
 void removeAt(int location);
   //Removes the item from the list at the position
   //specified by the location. If the location is out
   //of range, an appropriate message is printed.
void retrieveAt(int location, elemType& retItem);
  //Retrieves the element from the list at the
  //position specified by the location. The item
  //is returned via the parameter retItem.
  //If the location is out of range, an appropriate
  //message is printed.
void replaceAt(int location, const elemType& repItem);
  //Replaces the elements in the list at the position
  //specified by the location. The item to be replaced
  //is specified by the parameter repItem.
  //If the location is out of range, an appropriate
  //message is printed.
void clearList();
  //All elements from the list are removed. After this
  //operation, the size of the list is zero.
int seqSearch(const elemType& item);
  //Searches the list for a given item.
  //If the item is found, returns the location in the array
  //where the item is found; otherwise, returns -1.
```

```
    void insert(const elemType& insertItem);
      //The item specified by the parameter insertItem is
      //inserted at the end of the list. However, first the list
      //is searched to see whether the item to be inserted is
      //already in the list. If the item is already in the list
      //or the list is full, an appropriate message is output.
    void remove(const elemType& removeItem);
      //Removes an item from the list. The parameter removeItem
      //specifies the item to be removed.

    arrayListType(int size = 100);
      //constructor
      //Creates an array of the size specified by the parameter
      //size. The default array size is 100.
    arrayListType (const arrayListType<elemType>& otherList);
      //copy constructor
    ~arrayListType();
      //destructor
      //Deallocates the memory occupied by the array.

protected:
    elemType *list;    //array to hold the list of elements
    int length;   //to store the length of the list
    int maxSize;  //to store the maximum size of the list
};
```

实现类arrayListType的操作的函数定义与第14章中的基本一样，只不过实现这些操作的函数是函数模板而已。例如，函数insertAt，seqSearch与remove的定义如下所示：

```
template <class elemType>
void arrayListType<elemType>::insertAt
                   (int location, const elemType& insertItem)
{
   int  i;

   if(location < 0 || location >= maxSize)
      cout<<"The position of the item to be inserted "
          <<"is out of range"<<endl;
   else
      if(length >= maxSize)  //list is full
         cout<<"Cannot insert in a full list"<<endl;
      else
      {
         for(i = length; i > location; i-)
             list[ i] = list[ i - 1] ; //move the elements down

         list[ location] = insertItem;  //insert the item at the
                                        //specified position

         length++;      //increment the length
      }
} //end insertAt

template <class elemType>
int arrayListType<elemType>::seqSearch(const elemType& item)
{
      int loc;
```

```
        bool found = false;

        for(loc = 0; loc < length; loc++)
            if(list[loc] == item)
            {
               found = true;
               break;
            }

        if(found)
           return loc;
        else
           return -1;
} //end seqSearch


template<class elemType>
void arrayListType<elemType>::remove(const elemType& removeItem)
{
        int loc;
        if(length == 0)
           cout<<"Cannot delete from an empty list."<<endl;
        else
        {
           loc = seqSearch(removeItem);

           if(loc != -1)
              removeAt(loc);
           else
              cout<<"The item to be deleted is not in the list."
                     <<endl;
        }

} //end remove
```

由于要重载类 arrayListType 的赋值运算符，下面给出了重载赋值运算符的函数定义：

```
template <class elemType>
const arrayListType<elemType>& arrayListType<elemType>::operator=
                 (const arrayListType<elemType>& otherList)
{
        if(this != &otherList)        //avoid self-assignment
        {
           if(maxSize != otherList.maxSize)
                cout<<"Cannot copy. The two arrays are of "
                    <<"different sizes"<<endl;
           else
           {
                int j;

                maxSize = otherList.maxSize;
                length = otherList.length;
                list = new elemType[maxSize];     //create the array

                if(length != 0)          //if otherList is not empty
                   for(j = 0; j < length; j++)  //copy otherList
                       list[j] = otherList.list[j];
           }
```

```
        }

        return *this;
}
```

类arrayListType的其他函数模板定义留给读者作为练习（见本章末尾编程练习14）。

下面的例子说明了怎样使用类模板arrayListType来处理各种表。例15.9的程序处理了一个数字表和一个字符串表。

例15.9 下面的程序用于测试基于数组的表上的各种操作。

```
#include <iostream>

#include "myString.h"
#include "arrayListType.h"

using namespace std;

int main()
{
    arrayListType<int> intList(100);                            //Line 1
    arrayListType<newString> stringList;                        //Line 2

    int counter;                                                //Line 3
    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(counter, number);                      //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: Enter the item to be deleted: ";           //Line 14
    cin>>number;                                                //Line 15
    intList.remove(number);                                     //Line 16
    cout<<"Line 17: After removing "<<number
        <<" the list is:"<<endl;                                //Line 17
    intList.print();                                            //Line 18
    cout<<endl;                                                 //Line 19

    newString str;                                              //Line 20

    cout<<"Line 21: Processing the string list"
        <<endl;                                                 //Line 21

    cout<<"Line 22: Enter 5 strings: ";                        //Line 22
    for(counter = 0; counter < 5; counter++)                    //Line 23
    {
```

```
        cin>>str;                                        //Line 24
        stringList.insertAt(counter, str);               //Line 25
    }

    cout<<endl;                                          //Line 26
    cout<<"Line 27: The list you entered is: "
        <<endl;                                          //Line 27
    stringList.print();                                  //Line 28
    cout<<endl;                                          //Line 29

    cout<<"Line 30: Enter the string to be deleted: ";   //Line 30
    cin>>str;                                            //Line 31
    stringList.remove(str);                              //Line 32
    cout<<"Line 33: After removing "<<str
        <<" the list is:"<<endl;                         //Line 33
    stringList.print();                                  //Line 34
    cout<<endl;                                          //Line 35

    int intListSize;                                     //Line 36

    cout<<"Line 37: Enter the size of the integer "
        <<"list: ";                                      //Line 37
    cin>>intListSize;                                    //Line 38

    arrayListType<int> intList2(intListSize);            //Line 39

    cout<<"Line 40: Processing the integer list"
        <<endl;                                          //Line 40
    cout<<"Line 41: Enter "<<intListSize
        <<" integers: ";                                 //Line 41

    for(counter = 0; counter < intListSize; counter++)   //Line 42
    {
        cin>>number;                                     //Line 43
        intList2.insertAt(counter, number);              //Line 44
    }

    cout<<endl;                                          //Line 45
    cout<<"Line 46: The list you entered is: "<<endl;    //Line 46
    intList2.print();                                    //Line 47
    cout<<endl;                                          //Line 48

    return 0;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
List 5: Processing the integer list
List 6: Enter 5 integers: 23 78 56 12 79
List 11: The list you entered is: 23 78 56 12 79

Line 14: Enter the item to be deleted: 56
Line 17: After removing 56 the list is:
23 78 12 79

Line 21: Processing the string list
Line 22: Enter 5 strings: hello sunny warm winter summer
```

```
Line 27: The list you entered is:
hello sunny warm winter summer

Line 30: Enter the string to be deleted: hello
Line 33: After removing hello the list is:
sunny warm winter summer

Line 37: Enter the size of the integer list: 7
Line 40: Processing the integer list
Line 41: Enter 7 integers: 23 67 77 10 12 89 34

Line 46: The list you entered is:
23 67 77 10 12 89 34
```

上面程序的执行过程如下。第1行语句声明intList为arrayListType类型的对象，intList的数据成员list是有100个元素的数组，并且每个元素的类型是int。第2行语句声明stringList是arrayListType类型对象，stringList的数据成员list是有100个元素（默认大小）的数组，并且每个元素类型是newString。第6行语句提示用户输入5个整数，第8行语句从输入流中读取下一个数字。在第9行语句中，成员函数insertAt将数字存储到intList中。在第12行语句中，intList的成员函数print输出intList中的元素。第14行语句提示用户输入要从intList中删除的数字，第15行语句从输入流中读取要删除的数字，第16行语句用intList的成员函数remove从intList中删除数字。

第21行到第35行语句与第5行到第19行的执行过程一样，这些语句处理的是字符串表。

第37行语句提示用户输入整数表的大小，第38行语句将用户输入的表大小存储到变量intListSize中。第39行语句将intList2声明为arrayListType类型的对象，intList2表的数据成员list是表元素数目为intListSize的数组。这说明在程序运行时可以指定存储数据的数组的大小。其他语句的含义都很简单。

15.9 小结

1. 运算符对于不同的数据类型有不同的含义，称为重载。
2. 在C++中，>>可作为流析取运算符或者右移运算符，<<可作为流插入运算符或者左移运算符，它们都是运算符重载的例子。
3. 重载运算符的函数称为运算符函数。
4. 运算符函数头的语法是：

```
returnType operator operatorSymbol (parameters)
```

5. 在C++中，opreator是保留字。
6. 运算符函数是带有返回值的函数。
7. 除了赋值运算符和成员访问运算符外，使用其他运算符操作类对象，则运算符必须重载。赋值运算符默认的执行方式是基于成员方式的拷贝（Member-wise Copy）。
8. 有指针数据成员的类，必须明确重载赋值运算符。
9. 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相同运算符。
10. 重载的运算符不能改变优先级，不能改变结合律，不能使用默认的参数，运算符操作数的数目不能改变。并且运算符的含义与内建数据类型相应的运算符的含义一样。
11. 不能创建新运算符，只能重载现有的运算符。
12. 大部分C++运算符都能重载。

13. 不能重载的运算符是：.,.*,::,?:和 sizeof。
14. 指针 this 将对象作为一个整体来引用。
15. 重载运算符( ),[ ]，->和 = 的函数必须是类的成员。
16. 友元函数是类的非成员（Nonmember）。
17. 友元函数之前要有 friend。
18. 在 C++ 中，friend 是保留字。
19. 若运算符函数是类成员，则最左边的操作数必须是运算符所在类的对象（或类对象的引用）。
20. 双目运算符函数作为类的成员时有一个参数；作为类的非成员时有两个参数。
21. 重载类的流插入运算符<<和流析取运算符的函数>>必须是类的友元函数。
22. 为类重载前置增量元素运算符（++）时，若运算符函数是类的成员，则它没有参数。同样，为类重载前置减量运算符（--）时，若运算符函数是类的成员，它也没有参数。
23. 为类重载后置增量运算符（++）时，若运算符函数是类的成员，则它必须要有一个 int 型参数。用户不必指定参数的值，函数头中的伪参数仅帮助编译器生成正确的代码。后置减量运算符也是一样的。
24. 拷贝构造函数用一个对象的值初始化另一个同类的对象，被拷贝的对象必须作为引用参数传递。
25. 如果形参是值参数，在执行拷贝构造函数时，将实参的值传递给形参。
26. 转换构造函数是单参数函数。
27. 转换构造函数将自己的形参转换为构造函数类的对象，编译器隐含调用此构造函数。
28. 有指针数据成员的类必须重载赋值运算符，并且要包含拷贝构造函数和析构函数。
29. 在 C++ 中，函数名可被重载。
30. 重载函数的各个实例有不同的参数列表。
31. 在 C++ 中，template 是保留字。
32. 有了模板，可以为一系列相关函数写一段单独代码——称为函数模板。
33. 有了模板，可以为一系列相关类写一段单独代码——称为类模板。
34. 模板的语法是：

```
template <class Type>
declaration;
```

   其中，Type 是用户定义的标识符，它作为参数传递类型（数据类型），declaration 是类或函数，模板头中的 class 用来引用任意用户定义或内建的数据类型。
35. 类模板称为参数化类型。
36. 类模板中，参数 Type 用来指定由通用类模板生成特定类类型。
37. 参数 Type 将用在每个类模板的类头和成员函数定义中。
38. 假设 cType 是类模板，func 是 cType 的成员函数。func 的函数定义的函数头是：

```
template <class Type >
funcType cType<Type>::func(parameters)
```

   其中 funcType 是函数类型，如 void。
39. 假设 cType 是类模板，并可将 int 作为参数，则语句：

```
cType<int> x;
```

   声明 x 是 cType 类型的对象，且传递给类 cType 的类型是 int。

15.10 练习

1. 判断下列说法的正误。
   a. 在C++中，对用户定义的数据类型，所有运算符都能被重载。
   b. 在C++中，对内建类型，运算符不能被重定义。
   c. 重载运算符的函数称为运算符函数。
   d. C++允许用户创建自己的运算符。
   e. 运算符的优先级不能改变，但结合律可以改变。
   f. 重载函数的每个实例都有相同数目的参数。
   g. 若类仅有int型数据成员，则不需重载关系运算符。
   h. 类模板的成员函数是函数模板。
   i. 编写友元函数的定义时，关键字friend必须出现在函数头中。
   j. 模板提供了软件重用的能力。
   k. 由于两个运算符有相同的符号，所以重载前置增量运算符和后置增量运算符的函数头相同。
2. 什么是友元函数？
3. 假定用户定义的类mystery重载了运算符<<，为什么运算符<<必须作为友元函数重载？
4. 假定为类strange作为成员函数重载了双目运算符+，则函数operator+有几个参数。
5. 在什么时候类要重载赋值运算符并需要定义拷贝构造函数？
6. 考虑下面的声明：

```
class strange
{
.
.
.
};
```

   a. 编写一个声明语句，表明在类strange中重载了运算符>>。
   b. 编写一个声明语句，表明在类strange中重载了运算符=。
   c. 编写一个声明语句，表明在类strange中作为成员函数重载了双目运算符+。
   d. 编写一个声明语句，表明在类strange中作为成员函数重载了运算符==。
   e. 编写一个声明语句，表明在类strange中作为成员函数重载了后置增量运算符++。
7. 假定练习6中的声明不变。
   a. 编写一个声明语句，表明在类strange中作为友元函数重载了双目运算符+。
   b. 编写一个声明语句，表明在类strange中作为友元函数重载了运算符==。
   c. 编写一个声明语句，表明在类strange中作为友元函数重载了后置增量运算符++。
8. 找出下面代码中的错误：

```
class mystery                          //Line 1
{
    ...
    bool operator <= (mystery);        //Line 2
    ...
};

bool mystery::<=(mystery rightObj)     //Line 3
{
    ...
}
```

9. 找出下面代码中的错误：

```
class mystery                                    //Line 1
{
    ...
    bool operator <= (mystery, mystery);         //Line 2
    ...
};
```

10. 找出下面代码中的错误：

```
class mystery                              //Line 1
{
    ...
    friend operator+ (mystery);            //Line 2
      //overload binary +
    ...
};
```

11. 作为成员函数重载类的前置增量运算符时需要几个参数？
12. 作为友元函数重载类的前置增量运算符时需要几个参数？
13. 作为成员函数重载类的后置增量运算符时需要几个参数？
14. 作为友元函数重载类的后置增量运算符时需要几个参数？
15. $a+ib$ 是复数。$a+ib$ 的共轭复数是 $a-ib$，$a+ib$ 的绝对值是 $\sqrt{a^2+b^2}$。扩展前面程序范例中复数类 complexType 的定义，作为成员函数重载运算符~和!，使得运算符~返回复数的共轭复数，而运算符!返回复数的绝对值。编写这两个运算符函数的定义。
16. 重做练习 15，将运算符~和!作为非成员函数重载。
17. 找出下面代码中的错误：

```
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. 编写一个声明语句，声明 sObj 是 strange 类型的对象，私有数据成员 a 和 b 为 int 型。
   b. 编写一个声明语句，表明在类 strange 中作为成员函数重载了运算符 ==。
   c. 如果假定相应的数据成员相等，则两个 strange 类型的对象相等。为类 strange 写出函数 operator== 的定义，它作为成员函数重载。

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;
}
```

再假设有下面的声明：

```
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. 为类newString重载运算符+以实现字符串的连接。例如，如果s1是"Hello"，s2是"there"，语句：

```
s3=s1+s2;
```

将 "Hello there" 赋值给 s3，其中 s1，s2 和 s3 是 newString 对象。

b. 为类 newString 重载运算符 += 以实现下述字符串的连接：假设 s1 是 "Hello"，而 s2 是 "there"，语句：

```
s1+=s2;
```

应将 "Hello there" 赋值给 s1，其中 s1 和 s2 是 newString 对象。

15.11 编程练习

1. a. 编写增量、减量、算术及关系运算符的重载函数的定义，它们都作为类 OpOverClass 的成员。
   b. 编写一个测试程序，测试类 OpOverClass 的各种操作。
2. a. 编写增量、减量、算术及关系运算符的重载函数的定义，它们都作为类OpOverClass的非成员。
   b. 编写一个测试程序，测试类 OpOverClass 的各种操作。
3. a. 作为类 clockType 的成员重载后置增量运算符函数，以扩展类 clockType 的定义。
   b. 按照 a 的要求，编写为类 clockType 重载的后置增量运算符函数的定义。
4. a. 在类 clockType 中，增量与关系运算符是作为成员函数重载。重写类 clockType 的定义，将这些运算符及后置增量运算符作为非成员函数重载。
   b. 按照 a 部分的设计，写出类 clockType 的成员函数及非成员函数的定义。
   c. 编写一个测试程序，测试按照 a 和 b 部分设计的类的各种操作。
5. a. 扩展类complexType的定义，使之能进行减法和除法运算，作为成员函数重载类的减法和除法运算符。

   若(a, b)和(c, d)是复数：

   $(a, b) - (c, d) = (a-c, b-d)$

   若(c, d)非零：

   $(a, b) / (c, d) = ((ac + bd) / (c^2 + d^2), (-ad + bc) / (c^2 + d^2))$

   b. 按照 a 部分的要求，编写运算符 - 和 / 的重载函数的定义。
   c. 编写一个测试程序，测试类 complexType 的各种操作，计算结果保留两位有效数字。
6. a. 重写类 complexType 的定义，算术运算符和关系运算符作为非成员函数重载。
   b. 按照 a 部分的要求，编写类 complexType 的非成员函数的定义。
   c. 编写一个测试程序，测试类 complexType 的各种操作，计算结果保留两位有效数字。
7. a. 按如下要求扩展 newString 的定义：
   i. 重载运算符 + 及 += 以实现字符串连接操作。
   ii. 增加函数 length 并返回字符串的长度。
   b. 编写函数定义以实现 a 部分要求的操作。
   c. 编写一个测试程序，测试类 newString 对象的各种操作。
8. a. 按照编程练习 7 的要求重写类 newString 的定义，并且将关系运算符作为非成员函数重载。
   b. 按 a 部分的要求编写类 newString 的定义。
   c. 编写一个测试程序，测试类 newString 的各种操作。
9. 有理分数的形式是a / b，其中a和b是整数且$b \neq 0$。在此练习中，“分数”都是指有理分数。假设a / b和c / d是分数，分数的算术运算按如下规则定义：

   $a / b + c / d = (ad + bc) / bd$

   $a / b - c / d = (ad - bc) / bd$

   $a / b \times c / d = ac / bd$

   $(a / b) / (c / d) = ad / bc$，其中$c/d \neq 0$

   分数按下面规则比较：若ad op bc，则a / b op c / d，其中op是关系运算符。例如，若$ad < bc$，则$a / b < c / d$。

设计类fraction，它可实现分数的算术和关系运算。重载算术和关系运算符以使相应的符号能实现相应的运算功能。同时，为了便于输入输出，重载流插入和流析取运算符。

用类fraction编写一个能实现分数运算的C++程序。

测试下面的操作。假定x，y和z是fraction类型的对象，如果输入是2/3，语句：

```
cin>>x;
```

应将2/3存储在x中，语句：

```
cout<<x+y<<endl;
```

应以分数的形式输出x+y的值。

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

应以分数的形式存储x和y的和到z中。结果不必简化成最小项形式。

10. 前面曾说明，在C++中不做数组下标越界检查。但程序执行时，数组下标越界会造成严重问题。注意在C++中，数组下标由0开始。

设计并实现类myArray，使它可以避免数组下标越界，并且允许用户指定该数组的下标以任意的整数开始，包括正数和负数。每个myArray类型的对象都是int型数组。程序运行期间，如果访问的数组成员下标越界，程序必须终止并给出相应的错误信息。考虑下面语句：

```
myArray<int> list(5);                      //Line 1
myArray<int> myList(2,13);                 //Line 2
myArray<int> yourList(-5,9);               //Line 3
```

第1行语句将list声明为有5个成员的数组，成员的类型是int型，它的成员是：list[0]，list[1]，…，list[4]；第2行语句将myList声明为有11个成员的数组，成员的类型是int型，它的成员是：myList[2]，myList[3]，…，myList[12]；第3行语句将yourList声明为有14个成员的数组，成员的类型是int型，它的成员是：yourList[-5]，yourList[-4]，…，yourList[0]，…，yourList[8]。编写一个程序测试类myArray。

11. 编程练习10仅能处理int型数组，用模板重新设计类myArray，使该类能处理任何数据类型的数组。

12. 设计一个类，它能进行各种矩阵运算。矩阵是按行和列排列的一系列数据，因此矩阵中的每个元素都有一个行位置和一个列位置。若$\boldsymbol{A}$是有5行6列的矩阵，则称矩阵$\boldsymbol{A}$的大小是5×6，有时表示为$\boldsymbol{A}_{5\times 6}$。显而易见，用二维数组存储矩阵很方便。若两个矩阵的大小相同，则它们可以相加减。假设$\boldsymbol{A} = [a_{ij}]$和$\boldsymbol{B} = [b_{ij}]$是两个大小为$m \times n$的矩阵，这里$a_{ij}$表示在$\boldsymbol{A}$中的第$i$行第$j$列的元素。$\boldsymbol{A}$与$\boldsymbol{B}$的和与差为：

$$\boldsymbol{A} + \boldsymbol{B} = [a_{ij} + b_{ij}]$$
$$\boldsymbol{A} - \boldsymbol{B} = [a_{ij} - b_{ij}]$$

只有当$\boldsymbol{A}$的列数等于$\boldsymbol{B}$的行数时，$\boldsymbol{A}$与$\boldsymbol{B}$的乘法（$\boldsymbol{A}*\boldsymbol{B}$）才有意义。如果$\boldsymbol{A}$的大小是$m \times n$，$\boldsymbol{B}$的大小是$n \times t$，那么$\boldsymbol{A}*\boldsymbol{B} = [c_{ik}]$的大小是$m \times t$，元素$c_{ik}$由下面公式给出：

$$c_{ik} = a_{i1}b_{1k} + a_{i2}b_{2k} + \cdots + a_{in}b_{nk}$$

设计并实现类matrixType，使它能存储任意大小的数组。分别重载运算符+，-和*以实现矩阵加、减和乘运算，并重载运算符<<以输出矩阵。同时，编写一个程序测试矩阵的各种运算。

13. a. 在第12章的编程练习1中，我们定义了一个romanType类来使用罗马数字，并且还使用函数romanToDecimal将罗马数字转换成相等的十进制数。

修改类 romanType 的定义，将数据成员声明为 protected。使用编程练习 7 中的类 newString 来处理字符串。为便于输入输出，重载流插入和流析取运算符，流插入运算符以罗马数字格式输出罗马数字。

而且，还要包含一个成员函数 decimalToRoman，它将十进制数（十进制数必须是正整数）转换为相等的罗马数字格式。编写成员函数 decimalToRoman 的定义。

为简单起见，假定字符 I 只可以出现在 V 和 X 的前面。例如，4 表示为 IV，9 表示为 IX，39 表示为 XXXIX，49 表示为 XXXXIX，40 表示为 XXXX，190 表示为 CLXXXX，等等。

b. 从类 romanType 中派生一个类 extRomanType，并完成下面的工作。在类 extRomanType 中，重载算术运算符 +，-，*，/，使罗马数字能进行算术运算。同时，为类 extRomanType 作为成员函数重载前置、后置增量与减量运算符。

罗马数字相加（减、乘或除）时，用它们的十进制数相加（减、乘和除），然后将结果转换成罗马数字格式。相减时，若第一个数字小于第二个数字，输出信息 "Because the first number is smaller than the second, the numbers cannot be subtracted"。同样，相除时，分子必须大于分母。增量和减量运算符采用类似的约定。

c. 按照上面 b 部分的要求编写运算符重载函数的定义。

d. 用下面的程序测试类 extRomanType（包含相应的头文件）：

```
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. 在本章例 15.9 中，类模板 arrayListType 设计为可以处理程序中的表。为便于说明，例中仅包含了操作 insertAt，seqSearch，remove 及重载赋值运算符的函数模板。编写其他函数模板的定义，以完成类模板 arrayListType 的设计，并编写程序测试表中的各种操作。

15.（股票市场）编写一个程序帮助当地股票交易公司完成自动交易。公司仅在股票市场投资，每个交易日结束时，公司希望生成并邮寄其股票清单，以使投资者知道他们持有的股票当天的交易情况。假设公司投资了 10 种不同的股票，要求输出两个清单，一个按股票代号排序，另一个按照从最高到最低的收益百分比排列。

数据从文件中输入，其格式如下所示：

```
symbol openingPrice closingPrice todayHigh todayLow prevClose volume
```

例如，有如下数据：

```
MSMT 112.50 115.75 116.50 111.75 113.50 6723823
CBA 67.50 75.50 78.75 67.50 65.75 378233
.
.
.
```

第一行说明股票代号为MSMT，当天开盘价是112.50，收盘价是115.75，当天最高价是116.50，最低价是111.75，昨天收盘价是113.50，当前持股为6 723 823。

按股票代号排序的列表格式如下所示：

```
*********  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
-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

分两步编写此程序。第一步（a部分），设计并实现股票对象。第二步（b部分），设计并实现一个对象来处理股票表。

a.（股票对象）设计并实现股票对象。能表示股票的各种特征的对象称为stockType类。

股票的主要成员是股票代码、股票价格、持股的数目。而且，需要输出一天的开盘价、最高价、最低价、前一天价格及收益/损失百分比，这是股票的全部特征。因此，股票对象应存储上述所有信息。

每个股票对象应能进行下述操作：

i. 设置股票信息

ii. 打印股票信息

iii. 显示不同的价格

iv. 计算并打印收益/损失百分比

v. 显示持股的数目

a.1. 股票表一般按照股票代码排序，重载关系运算符以通过代码来比较两个股票对象。

a.2. 为便于输出，重载插入运算符<<。

a.3. 由于数据存储在文件中，为便于输入，重载流析取运算符>>。

例如，假设infile是ifstream对象，输入文件用对象infile打开。且myStock是一个股票对象。语句：

```
infile>>myStock;
```

从输入文件中读数据并存储在对象myStock中（注意此语句读取并将数据存储在myStock对象相应的成员中）。

b. 在上面已设计并实现类 stockType 来处理股票对象的基础上，下面开始创建股票对象表。
   实现股票对象表的类称为 stockListType。
   类 stockListType 必须从类 listType 中派生，在前面的练习中已经实现了类 listType。然而，类 stockListType 是一个非常明确的类，用来创建股票对象表。因此，类 stockListType 不是模板。
   增加或者重写类 listType 的操作，以实现股票表所必须的操作。
   下面的语句从类 listType 中派生出类 stockListType：

```
class stockListType: public listType<stockType>
{
    member list
};
```

   在类 listType 中，存放表元素、表长度及 max listSize 的数据成员声明为 protected。因此，在类 stockListType 中，这些成员能被直接访问。
   由于公司还要求按照收益/损失百分比给出表顺序，还必须对它们排序。但是，并不需要按照收益/损失百分比对表进行物理排序，提供此成员的逻辑序列就可以了。
   为完成此工作，增加一个数组数据成员，它用来存放按照收益/损失百分比排序的股票表的下标索引，这个数组称为 sortIndicesGainLoss。打印按照收益/损失百分比排序的表时，使用 sortIndicesGainLoss。数组 sortIndicesGainLoss 中的元素指出下一个要打印的股票表的成员（通过该成员的索引）。
c. 使用这两个类，编写一个程序实现公司对股票数据分析的自动化。

第16章　链　　表

本章要点：

- 了解链表
- 了解链表的基本属性
- 掌握链表的插入和删除操作
- 理解怎样创建并处理链表
- 理解怎样构造双向链表

前面已经说明怎样用数组顺序地组织并处理数据，它被称为顺序表（Sequential List）。你已经掌握了顺序表的几种操作，如排序、插入、删除及查找。如果表中的数据没有排序，那么查找数据时会花费很长时间，特别是当表很大时更是如此。如果表中的数据经过排序，就可以使用二分法来改进查找算法。然而在这种情况下，由于需要移动数据，插入和删除操作会变得非常费时，特别是当表很大时更是如此。同时，由于运行时数组大小固定，只有在还有空余空间的情况下，才能增加新的数据项。也就是说，在数组中组织数据时，还有一些限制。

本章可以解决上述问题。第14章说明了怎样用指针来动态分配和释放内存（变量）。本章使用指针来组织和处理表中的数据，这被称为链表（Linked List）。回忆一下，当数据存储在数组中时，数据成员在内存中是连续存放的。

16.1　链表

链表是成员的集合，成员称为节点（Node）。每个节点（除了最后一个）都包含下一节点的地址。因此，链表中的每个节点都有两个成员：一个用来存储相应的信息（数据），另一个用来存储地址，称为表中下一节点的链接（Link）。链表中第一个节点的地址存储在单独的单元中，称为链表的头（Head）或起始（First）。图16.1显示了节点的结构。

图16.1　节点的结构

链表　一个由称为节点的数据项（Item）组成的表。其中节点的顺序由地址决定，这些地址称为链接，存储在每个节点中。

图16.2的表是一个链表的例子。

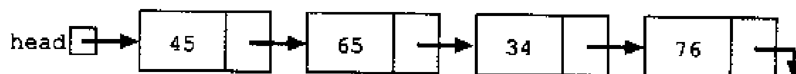


图16.2　链表

最后一个节点中向下的箭头表明此链接域为NULL。每个节点中的箭头表明此节点所存储的下一节点的地址。采用这种表示法是为便于理解，假定第一个节点的内存地址是1200，第二个节点的内存地址是1575，则有图16.3。

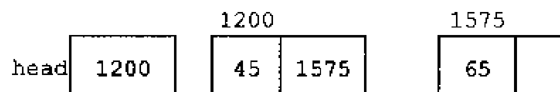


图16.3 链表及链接的值

头节点的值是1200，第一个节点的数据部分存储的是45，第1个节点链接部分存储的是1575，这是第2个节点的地址。只要不会造成歧义，链表的图都将采用箭头表示法。

由于链表的每个节点都有两个成员，需要将每个节点都声明为class或struct。每个节点的数据类型决定于具体的应用——即要处理的数据类型；节点的链接部分是指针，指针变量的数据类型是节点自身的类型。对于前面的链表，节点的定义如下所示（假设数据类型是int）：

```
struct nodeType
{
      int  info;
      nodeType *link;
};
```

变量声明如下所示：

```
nodeType *head;
```

16.1.1 链表：一些属性

为了更好地理解链表和节点的概念，下面说明链表的一些重要属性。

考虑图16.4中的链表。

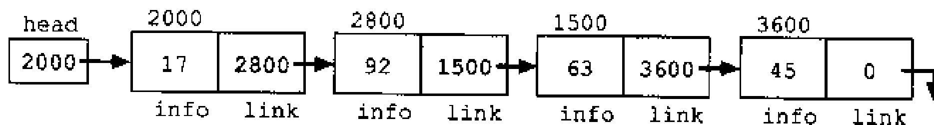


图16.4 有4个节点的链表

此链表有4个节点，第1个节点的地址存储在指针head中。每个节点有两个成员：info存储信息；link存储下一节点的地址。为简单起见，假设info是int型。

假设第1个节点的存储单元地址是2000，第2个节点的存储单元地址是2800，第3个节点的存储单元地址是1500，第4个节点的存储单元地址是3600。因此，head的值是2000，第1个节点link成员的值是2800，第2个节点link成员的值是1500，等等。最后一个节点link成员中的0表明其值是NULL，在图中用向下的箭头表示。每个节点上方的数字是节点的地址。下面的表说明了此链表。

	值	
head	2000	
head->info	17	由于head是2000，在单元2000的节点的info是17
head->link	2800	
head->link->info	92	由于head->link是2800，在单元2800的节点的info是92

假设current是一个指针，并且与指针head的类型一样。语句：

```
current = head;
```

将head的值赋给current（如图16.5所示）。

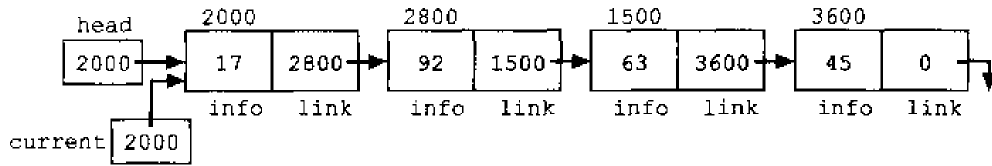


图 16.5　current = head;执行后的链表

显而易见，在图 16.5 中有：

	值
current	2000
current->info	17
current->link	2800
current->link->info	92

考虑下面语句：

```
current = current->link;
```

该语句将 current->link 的值，也就是 2800，赋给 current。因此，该语句执行后，current 指向链表的第二个节点（操作链表时，经常使用此类语句，从而使指针指向链表的下一个节点），如图 16.6 所示。

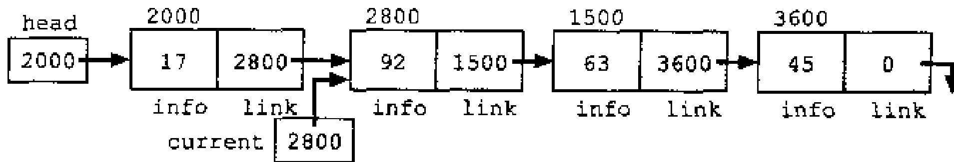


图 16.6　"current = current->link;"执行后的链表

在图 16.6 中有：

	值
current	2800
current->info	92
current->link	1500
current->link->info	63

最后要注意的是，在图 16.6 中有：

	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

从现在开始，每当用到链表时，都采用箭头表示法来存取链表成员。

遍历链表

链表有如下基本操作：查找链表以确定某元素是否在链表中，在链表中插入元素，从链表中删除元素。这些操作都需要遍历链表，也就是给出指向链表第一个节点的指针，然后访问链表的每个节点。

假设指针head指向链表的第一个节点，最后一个节点的链接是NULL。由于使用head遍历链表会导致头节点的丢失，所以不能用它进行遍历。之所以会产生这个问题，是因为只有一个方向有链接。指针head包含了第一个节点的地址，第一个节点包含了第二个节点的地址，第二个节点包含了第三个节点的地址，依次类推。若将head移到第二个节点，第一个节点就会丢失（除非有存储指向此节点的指针），若一直将head向下一个节点移动，整个链表的节点都会丢失（除非在移动head前保存所有节点的地址，但这不太现实，因为它需要额外的计算机时间及内存空间来存放链表）。

因此，head总是用来指向第一个节点，现在的问题是必须使用同类型的另一指针来遍历链表。假定指针current的类型与head一样，下面的代码可以遍历链表：

```
current = head;
while(current != NULL)
{
      //Process current
      current = current->link;
}
```

例如，假设head指向一个数字链表，下面的代码可以输出每个节点存放的数据：

```
current = head;
while(current != NULL)
{
      cout<<current->info<<" ";
      current = current->link;
}
```

16.1.2 插入及删除节点

本节讨论怎样在链表中插入或删除一个节点。考虑下面节点的定义，为简单起见，假设info类型是int型。在下一节中，将会讨论作为抽象数据类型（ADT）的链表，它使用模板。

```
struct nodeType
{
      int  info;
      nodeType *link;
};
```

还将用到下面的变量声明：

```
nodeType *head, *p, *q, *newNode;
```

插入

考虑图16.7中的链表。

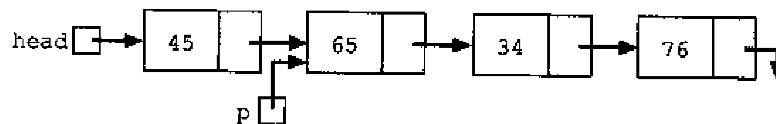


图16.7 插入节点之前的链表

假设p指向info为65的节点，在p之后要插入一个info为50的新节点。下面的语句创建并将50存储在新节点的info域中：

```
newNode = new nodeType;   //create newNode
newNode->info = 50;       //store 50 in the new node
```

第一条语句（newNode = new nodeType;）在内存中创建了一个新节点，并将新节点的地址存放在newNode中。第二条语句（newNode->info=50;）将50存放在新节点的info域中（如图16.8所示）。

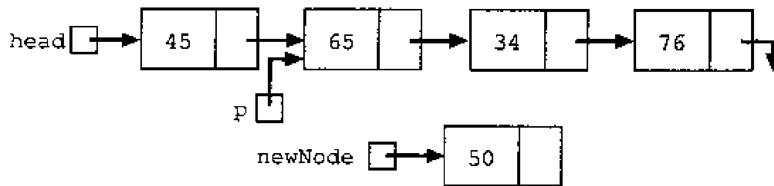


图16.8 创建newNode并存储50

下面的语句将节点插入到链表中的指定位置：

```
newNode->link = p->link;
p->link = newNode;
```

第一条语句（newNode->link = p->link;）执行后，结果显示在图16.9中。

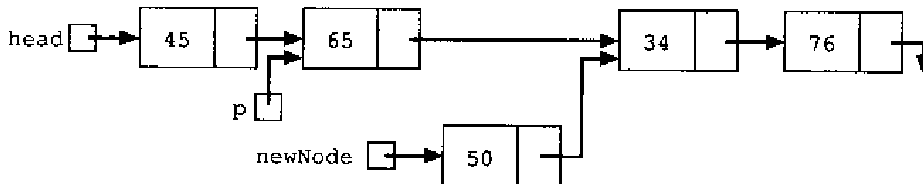


图16.9 "newNode->link = p->link;"执行后的链表

第二条语句（p->link = newNode;）执行后，结果显示在图16.10中。

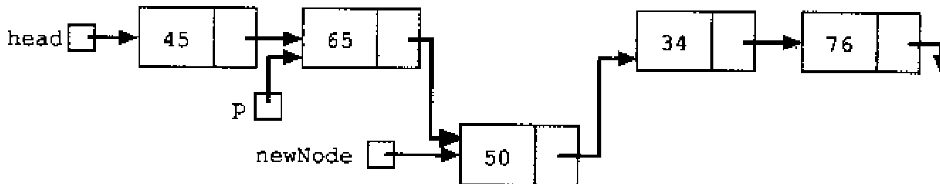


图16.10 "p->link = newNode;"执行后的链表

注意，插入节点的语句顺序非常重要。这是因为在链表中插入newNode时，只使用了一个指针p来调整节点的链接。现在改变语句的执行顺序如下所示：

```
p->link = newNode;
newNode->link = p->link;
```

图16.11显示了这些语句执行后的结果。

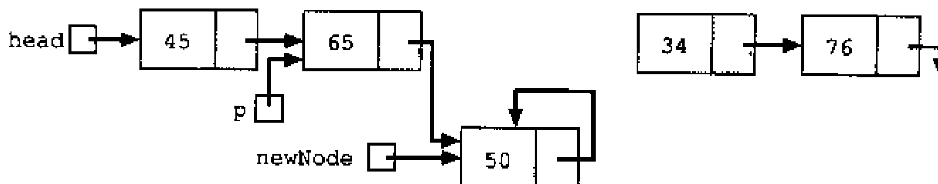


图16.11 "p->link = newNode; newNode->link = p->link;"执行后的链表

从图16.11可以看出，newNode指向自己，链表的其他部分丢失。

使用两个指针，在一定程度上可以简化插入代码。假设q指向info是34的节点（如图16.12所示）。下面的语句在p和q之间插入newNode：

```
newNode->link = q;
p->link = newNode;
```

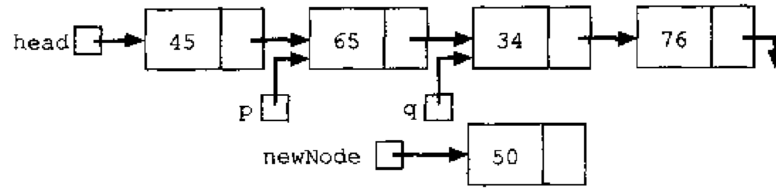


图 16.12　带指针 p 和 q 的链表

这些语句的执行顺序变得不再重要，为说明这个问题，假设按下面的顺序执行语句：

```
p->link = newNode;
newNode->link = q;
```

语句“p->link = newNode;”执行后，链表的结果显示在图 16.13 中。

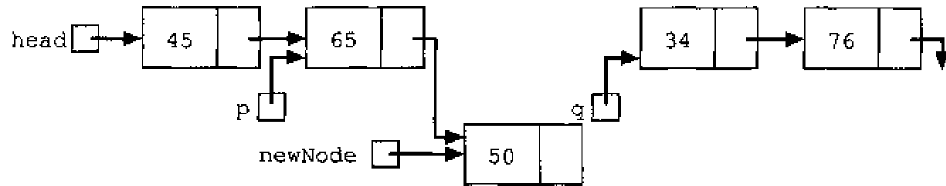


图 16.13　“p->link = newNode;”执行后的链表

由于有指针 q 指向其余的链表，所以链表的其余部分不会丢失。语句 newNode->link = q;执行后，链表如图 16.14 所示。

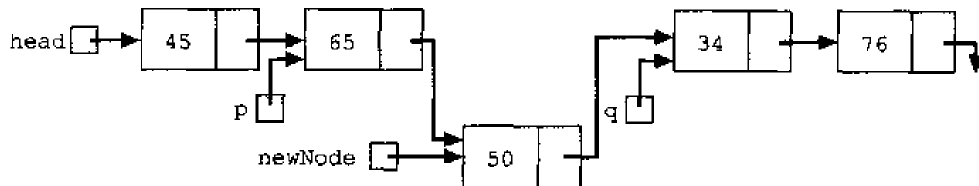


图 16.14　“newNode->link = q;”执行后的链表

删除

考虑图 16.15 中的链表。

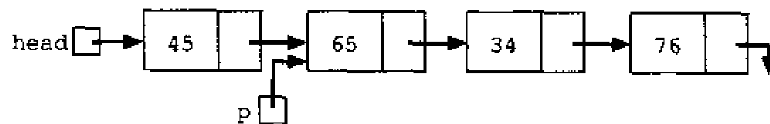


图 16.15　删除 info 是 34 的节点

假设要从链表中删除 info 是 34 的节点，下面的语句将从链表中删除该节点：

```
p->link = p->link->link;
```

图 16.16 显示了上述语句执行后链表的结果。

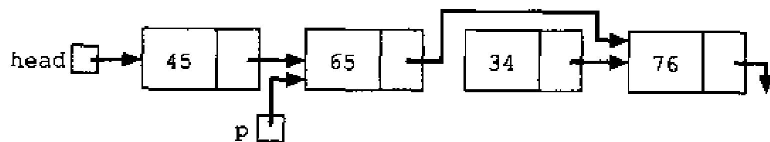


图 16.16　“p->link = p->link->link;”执行后的链表

从图16.16中可以看出，从链表中删除了info是34的节点，但是为此节点分配的内存仍然存在。也就是说，出现了悬挂节点。为了释放内存，还需要一个指向此节点的指针。下面的语句从链表中删除节点并释放节点所占用的内存：

```
q = p->link;
p->link = q->link;
delete q;
```

语句“q = p->link;”执行后，链表如图16.17所示。

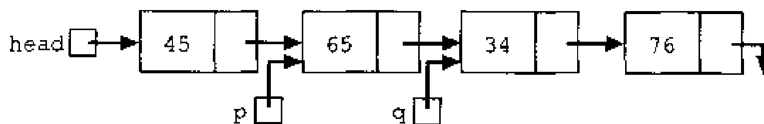


图16.17 “q = p->link;”执行后的链表

语句p->link = q->link;执行后，结果如图16.18所示。

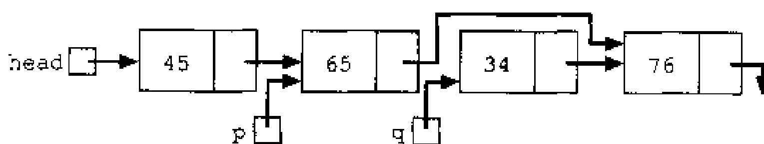


图16.18 “p->link = q->link;”执行后的链表

语句delete q;执行后，链表如图16.19所示。

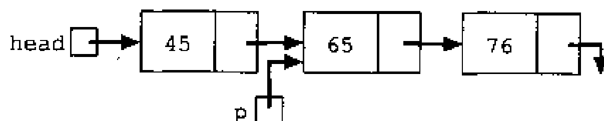


图16.19 “delete q;”执行后的链表

16.1.3 创建链表

知道怎样在链表中插入一个节点之后，再看怎样创建一个链表。首先要明确的是，如果读入的数据未排序，则链表也是未排序的。这种链表可通过两种方式建立：正向方式（Forward Manner）和逆向方式（Backward Manner）。正向方式是指新节点总是插入到链表的末尾；逆向方式是新节点总是插入到链表的最前端。下面分别考虑这两种方式。

正向创建链表

假设链表是常见的info-link形式，info是int型。假定需要处理下列数据：

```
2 15 8 24 34
```

创建链表需要3个指针：一个指向链表的第一个节点，它不能移动；一个指向链表的最后一个节点；一个指向新创建的节点。考虑下面的变量声明：

```
nodeType *first,*last,*newNode;
int num;
```

假设first指向链表的第一个节点。开始时，链表为空，first和last都为NULL。因此必须有下面的语句：

```
first = NULL;
last = NULL;
```

将first和last初始化为NULL。

考虑下面语句：

```
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
   }
```

下面开始执行这些语句。开始时，first和last都为NULL，链表如图16.20所示。

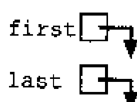


图16.20 空链表

语句1执行后，num是2。语句2创建一个节点，并将此节点的地址存储在newNode中。语句3将2存储到newNode的info域，而语句4将NULL存储在newNode的link域（如图16.21所示）。

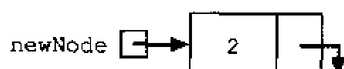


图16.21 info为2的newNode

由于first是NULL，执行语句5a和语句5b。执行结果如图16.22所示。

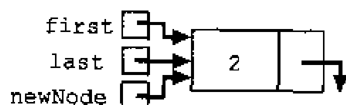


图16.22 插入newNode后的链表

现在重复语句1到语句6b。语句1执行后，num是15。语句2创建一个节点，并将此节点的地址存储在newNode中。语句3将15存储到newNode的info域，语句4将NULL存储在newNode的link域（如图16.23所示）。

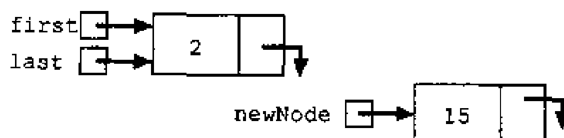


图16.23 info为15的newNode及链表

由于 first 不是 NULL，执行语句 6a 和语句 6b。执行结果如图 16.24 所示。

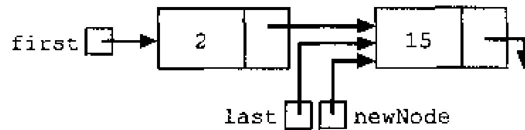


图 16.24　结尾插入 newNode 后的链表

现在再将语句 1 到语句 6b 重复 3 次。执行结果如图 16.25 所示。

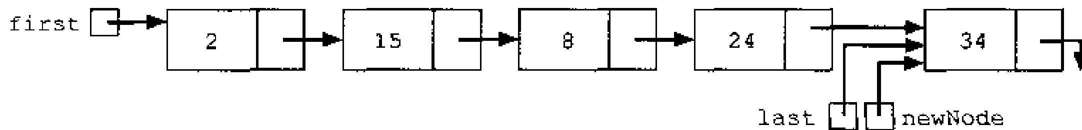


图 16.25　插入 8，24 及 34 后的链表

前面的语句可以放在循环中，在满足特定条件后执行该循环，从而建立链表。事实上，可以写一个建立链表的 C++ 函数。

假设要读取一个以 -999 结尾的整数表，下面函数 buildListForward，（采用正向方式）建立链表并返回所建链表的指针：

```
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
```

逆向创建链表

现在考虑逆向方式建立链表的情况，仍用前面所给的数据 2，15，8，24 和 34，链表如图 16.26 所示。

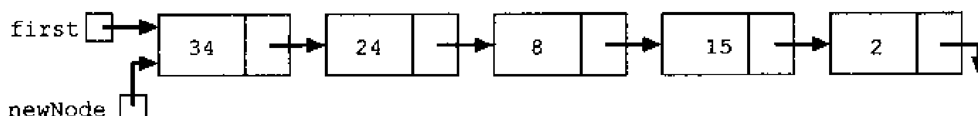


图 16.26 逆向创建的链表

由于新节点总是插入到链表的起始处，不需要知道链表的结尾，所以不再需要指针 last。同时，由于新节点插入到链表的起始点后，就成了链表的第一个节点，因此要更新 first 的值，以使其正确地指向链表的第一个节点。也就是说，建立链表需要两个指针：一个指向链表，一个创建新节点。初始时链表为空，因此指针 first 必须被初始化为 NULL。该算法的伪代码如下所示：

1. first 初始化为 NULL。
2. 对于表中的每一项：
   a. 创建新节点 newNode
   b. 将元素存入 newNode
   c. 在 first 前插入 newNode
   d. 更新指针 first 的值

下面的 C++ 函数采用逆向方式建立链表并返回链表的指针：

```
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
```

16.2 作为抽象数据类型的链表

前面的章节描述了链表的一般属性，以及怎样构造并操作链表。由于链表是非常重要的数据类型，本节主要将链表作为抽象数据类型（ADT）来讨论，而不讨论具体类型的链表，如 int 型或 char 型。本节用模板给出了链表的类属定义，在下一节及本书的后面部分还将用到它。本章结尾的程序范例也将用到链表的类属定义。

链表的基本操作是：

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. 计算链表长度
6. 删除链表
7. 在链表中查找给定的元素
8. 在链表中插入一个元素
9. 从链表中删除一个元素

此外，还要包括如下的操作。该操作在检查第一个节点的信息时，极为有用。

10. 检索第一个节点的信息

首先，要讨论的是任意链表——即链表可能排序或未排序，下一节将讨论已排序的链表。

若链表是未排序的，则可在开始或末尾插入新元素，这样的链表初始化时可以采用正向方式或者逆向方式。函数 buildListForward 将新元素插入到末尾，函数 buildListBackward 将新元素插入到开始。为说明这两种操作，下面编写两个函数：insertFirst 将 newitem 插入到链表的开始处，insertLast 将 newitem 插入到链表的末尾处。为使算法更有效，将在链表中保留两个指针：first 指向链表的第一个节点；last 指向链表的最后一个节点。

下面的类将链表作为 ADT 定义：

```
//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，类linkedListType的数据成员是protected，而不是private。这是由于要从这个类中派生其他类。在本章后面的“有序链表”一节及第17章中都要用到这个类。

类linkedListType的定义包括了重载赋值运算符的成员函数。对于含有指针数据成员的类，必须明确重载赋值运算符（见第14章和第15章）。出于同样的原因，类的定义也包括了拷贝构造函数。下一步将讨论成员函数的实现。

成员函数isEmptyList和isFullList相当简单。若链表为空，first是NULL。由于数据采取动态存储分配，（逻辑上）链表永远不会满（除非内存空间不足时，链表才会满）。因此，函数isFullList总是返回false。实现这些操作的函数定义如下所示：

```
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

默认的构造函数linkedListType非常简单，它将链表简单地初始化为空。记住，当声明linkedListType类型的对象且没有传递值时，默认的构造函数自动执行。

```
template<class Type>
linkedListType<Type>::linkedListType() //default constructor
{
      first = NULL;
      last = NULL;
}
```

删除链表

函数destroyList释放分配给每个节点的内存，它从第一个节点开始遍历链表，并调用运算符delete释放内存，释放内存时还需要一个临时指针。将整个链表删除以后，指针first和last必须设置为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将链表初始化为空。注意，声明链表对象时，默认的构造函数或拷贝构造函数已经初始化过链表。事实上，这个操作重新将链表初始化为空，因此它必须从链表中删除节点(若有的话)。destroyList操作可完成这个工作，它还将first和last重置为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用于打印每个节点中的数据。为了打印每个节点中的数据，必须从第一个节点开始遍历链表。由于指针first总是指向链表的第一个节点，因此需要另一个指针来遍历链表(若使用first遍历链表，则整个链表都会丢失)。

```
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

要想知道链表的长度（链表中有多少个节点），需要设置一个计数器（初始化为0）并遍历链表，每经过一个节点，计数器加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
```

检索第一个节点中的数据

函数retrieveFirst返回第一个节点中的info，它的定义很简单。

```
template<class Type>
void linkedListType<Type>::retrieveFirst(Type& firstElement)
{
    firstElement = first->info; //copy the info of the first node
} //end retrieveFirst
```

查找链表

成员函数search在链表中查找给定的元素。若找到该元素，它输出“Item is found in the list”，否则输出“Item is not in the list”。由于链表不是可随机访问的数据结构，必须从第一个节点开始顺序查找。

下面的步骤说明了这个函数的执行过程：

1. 将查找的元素和链表的当前节点做比较，如果当前节点的info与被查找的元素相同，停止查找。否则，将下一节点作为当前节点。
2. 重复步骤1，直到找到元素或到达链表末尾。

```
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
```

插入第一个节点

函数insertFirst将新元素插入到链表的起始处——即first所指的节点的前面。下面的步骤将实现这个函数:

1. 创建一个新节点
2. 将新元素存入新节点中
3. 在first前插入节点

```
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的定义很相似，这里是将新节点插入到last之后。函数insertLast的定义如下所示:

```
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

下一步讨论的是成员函数deleteNode的实现，它从链表中删除给定的info。下面几种情况必须加以考虑：

1. 链表为空。
2. 第一个节点是要删除的节点。这种情况需要调整指针first。
3. 带有给定info的节点在链表中。若要删除的节点是最后一个节点，还必须调整指针last。
4. 链表中没有给定info的节点。

若链表为空，可以简单打印一个信息说明链表是空的。若链表不为空，就要在链表中查找带有给定info的节点，如果存在这个节点，就将之删除。算法的伪代码如下所示：

```
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。

若链表为空，输出如伪代码所示的错误信息。

情况 2 链表非空，要删除的节点是第一个节点。

这种情况有两种可能：list只有一个节点，list有不止一个节点。在图16.27中，list只有一个节点。

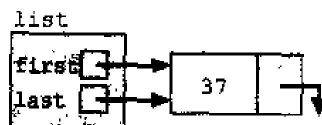


图 16.27 list只有一个节点

假设需要删除37。删除该节点以后，list变为空。因此，删除该节点后first和last都应该设置为NULL。现在考虑图16.28，list有不止一个节点。

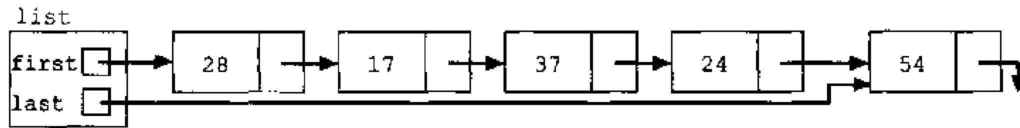


图 16.28 list有不止一个节点

假设要删除的节点是28。此节点删除后，第二个节点成为第一个节点。因此，此节点删除后，指针first的值发生变化，first包含的是info值为17的节点的地址。图16.29说明了删除28后的链表。

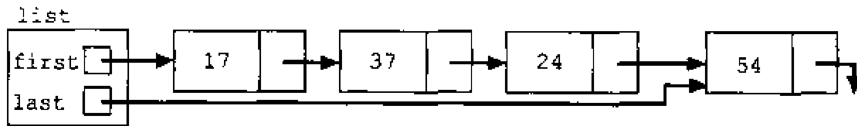


图 16.29 删除info值为28节点后的list

情况 3 要删除的节点不是第一个节点，而是在链表中间。

这种情况还有两种可能：(a) 删除的节点不是最后一个节点；(b) 删除的节点是最后一个节点。下面分别说明这两种情况。

情况 3a 删除的节点不是最后一个节点。

考虑图16.30中的链表。

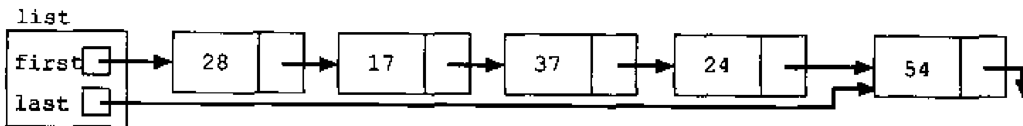


图 16.30 删除37之前的list

假设要删除的节点是37。删除此节点后，结果如图16.31所示（注意，删除37并不需要改变first和last的值，改变的是前一节点17的link域。删除之后，info值为17的节点包含值为24的节点的地址）。

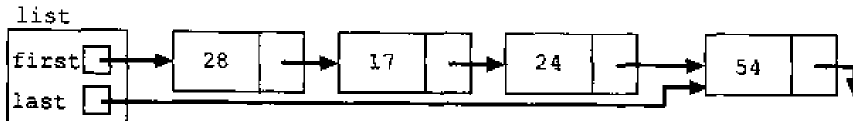


图 16.31 删除37之后的list

情况 3b 删除的节点是最后一个节点。

考虑图16.32中的链表，假设要删除的节点是54。

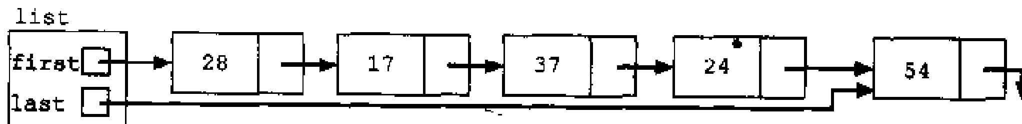


图 16.32 删除54之前的list

删除54后，info值为24的节点成为最后一个节点。因此，删除54需要改变指针last的值。删除54后，last包含的是info值为24的节点的地址。结果如图16.33所示。

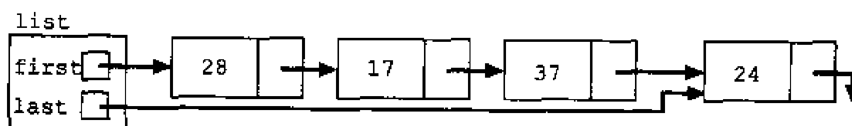


图 16.33 删除54之后的list

情况4　要删除的节点不在链表中。这种情况下，链表不需要调整，仅简单地输出一条错误信息，说明要删除的元素不在链表中。

从情况2，情况3和情况4可以看出，删除一个节点需要遍历链表。由于链表不是可随机访问的数据结构，所以必须顺序查找链表。情况1将单独处理，因为它不需遍历链表。遍历时从第二个节点开始顺序查找链表，若要删除的节点在链表的中间，则需要改变要删除节点之前的那个节点的link域，因此需要一个当前节点的指针。当在链表中查找给定的info时，需要两个指针：一个检查当前节点的info，一个保存当前节点之前的节点。若删除的是最后一个节点，则需要调整指针last。

函数deleteNode的定义如下所示：

```
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 else
} //end deleteNode
```

析构函数

在给出拷贝构造函数的定义之前,先看一下析构函数。析构函数的作用是当类的对象超出作用域时,释放链表节点的内存。由于链表节点的内存是动态分配的,重置指针 first 和 last 不会释放分配给链表节点的内存。所以必须从第一个节点开始遍历链表,并将每个节点都删除。现在可以清楚地看出,析构函数的定义与函数 destroyList 的定义很相似。前面讲过,当类的对象超出作用域时,析构函数自动执行。

```
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
```

拷贝构造函数

因为类 linkedListType 含有指针数据成员,类的定义应包含拷贝构造函数。记住,若形参是值参数,拷贝构造函数用自己数据的拷贝提供形参。当对象被声明并用另一个对象初始化时,将执行拷贝构造函数(更多的信息参见第 14 章)。

拷贝构造函数生成与原链表完全相同的拷贝,因此必须要从原链表的第一个节点开始遍历。与原链表相对应,需要:

1. 创建一个节点,称为 newNode
2. 将原链表中节点的 info 拷贝给 newNode
3. newNode 插入到新创建的链表末尾

拷贝构造函数的定义如下所示:

```
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

类linkedListType的重载赋值运算符函数定义与拷贝构造函数的定义非常相似。为了完整，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;
}
```

16.3 有序链表

前面已经讲过，在C++中怎样使用指针变量动态分配和释放内存，并说明了怎样创建并处理链表。本节讨论的是怎样创建有序链表并说明这种链表的一些操作。有序链表通常进行下面的操作：

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. 逆向打印链表
6. 删除链表
7. 在链表中查找给定的元素
8. 在链表中插入一个元素
9. 从链表中删除一个元素
10. 查找链表长度
11. 拷贝链表

有序链表的很多操作与上节讨论的一般链表的操作很类似。由于链表有序，需要改变查找、插入和删除操作的算法。因此，可从类 linkedListType 派生出作为 ADT 的类的定义。

```
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没有任何作用，所以将其忽略。

查找链表

首先讨论查找操作，实现查找操作的算法与前面讨论的一般链表的查找算法很类似。由于链表已经排过序，查找算法的效率可以大大提高。和前面一样，从链表的第一个节点开始查找。当链表中节点的info大于或等于要查找的元素，或者是已经查找了整个链表时，停止查找。

下面的步骤说明了此算法：

1. 将要查找的元素和链表中当前节点做比较，如果当前节点的info大于或等于查找的元素，停止查找；否则，将下一个节点作为当前节点。
2. 重复步骤1，直到链表中的元素大于或等于要查找的元素，或者链表中已没有需要比较的元素为止。

注意上面的循环并没有明确检查链表中是否有与要查找的元素相等的元素。因此，循环执行后，还必须检查要查找的元素是否与链表中的元素相等。

```
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

要在有序链表中插入一个元素，首先要找到插入的位置，然后再将元素插入到链表中。和前面一样，要找到新元素在链表中的位置，必须查找链表。这里用到两个指针，current和trailCurrent来查找链表。指针current指向与插入的元素相比较的节点，trailCurrent指向current之前的节点。由于链表有序，查找算法和前面一样。有下面几种情况：

1. 链表初始化为空。链表中只有含有新元素的节点，它是第一个节点。
2. 新元素小于链表中元素最小的节点，新元素放在链表的起始处。这种情况下调整链表的头指针first。
3. 元素插入到链表的中间。
   - 3a. 新元素大于链表中所有的元素。在这种情况下，新元素插入到链表的末尾。因此，current的值是NULL，新元素插入到trailCurrent之后。
   - 3b. 新元素插入到链表中间的某个地方。这种情况下，新元素插入到trailCurrent和current之间。

下面的语句可以实现3a和3b的情况，newNode指向新节点。

```
trailCurrnet->link = newNode;
newNode->link = current;
```

下面分别说明这些情况。

情况1 链表为空。

考虑图16.34中的链表。

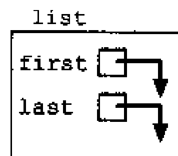


图16.34 空链表

假设需要在链表中插入27。为完成这个任务，需要创建一个节点，拷贝27给该节点，再将该节点的link域设为NULL，让first和last指向该节点。结果如图16.35所示。

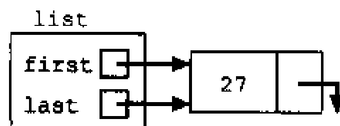


图16.35 插入27后的链表

注意，插入27后，first和last的值都发生改变。

情况2 链表非空，插入的元素小于链表中最小的元素。考虑图16.36中的链表。

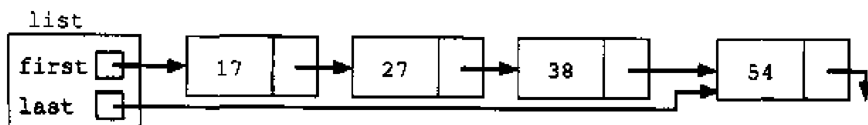


图16.36 插入10之前的非空链表

假设要插入10。将10插入链表后，info为10的节点成为list的第一个节点，这就要改变first的值。结果如图16.37所示。

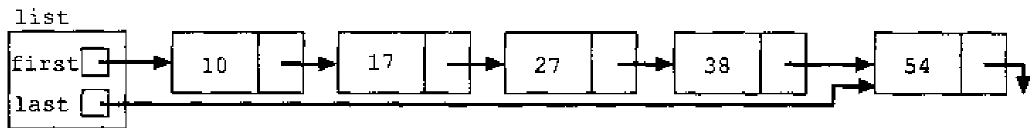


图16.37 插入10之后的链表

情况3 链表非空，插入的元素大于链表中的第一个元素。如前所述，这种情况还有两种可能。

情况3a 插入的元素大于链表中最大的元素，也就是说要插入到链表的末尾。考虑图16.38所示的链表。

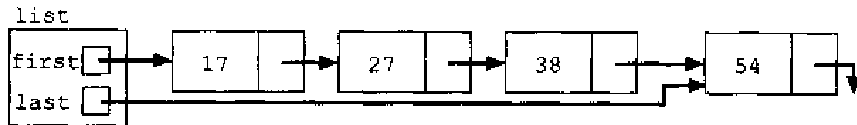


图16.38 插入65之前的链表

假设要在链表中插入65。插入65后，图16.39显示了插入后的链表（注意插入65需要改变last的值）。

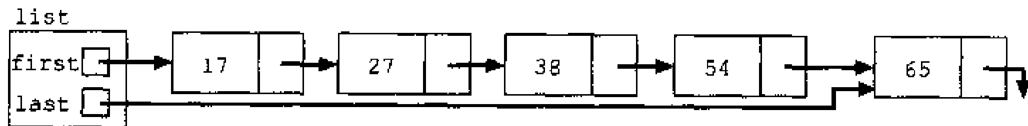


图16.39 插入65之后的链表

情况3b 元素插入到链表的中间某处。考虑图16.40所示的链表。

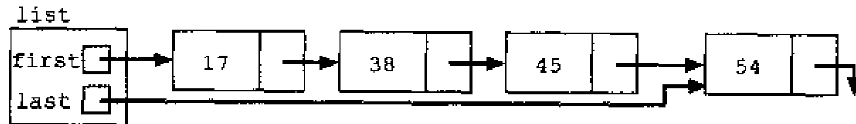


图16.40 插入27之前的链表

假设要在链表中插入27。显然，27要插入到17和38之间，这要改变info为17的节点的link值。链表插入27后，结果如图16.41所示。

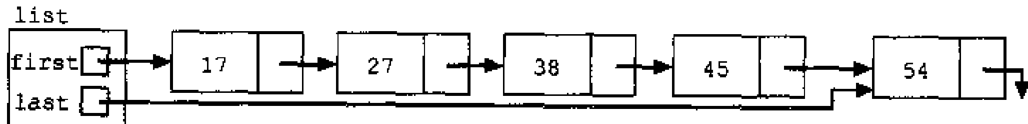


图16.41 插入27之后的链表

从情况3可以看出，要找到新元素插入的位置，必须先遍历链表。遍历链表时需要两个指针，current和trailCurrent。指针current用来遍历链表，并将要插入的元素和链表中节点的info域做比较。指针trailCurrent指向current之前的节点。例如，在情况3b中，查找停止时，trailCurrent指向节点17，current指向节点38，元素插入到trailCurrent后。在情况3a中，为找到65的位置而查找链表之后，trailCurrent指向节点54，current是NULL。

函数 insertNode 实现如下所示：

```
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

要从有序链表中删除一个给定的元素，首先要查找链表以确定要删除的元素是否在链表中。实现这个操作的函数和一般链表的删除操作一样。由于链表已经排序，可以改进实现算法。

与 insertNode 中的情况一样，查找链表需要两个指针，current 和 trailCurrent。和 insertNode 中查找情况类似，也有几种情况：

1. 链表初始化为空。这是一种错误的情况，不能从空链表中删除节点。
2. 要删除的节点为第一个节点。必须调整链表的头指针 first。
3. 要删除的元素在链表中某处，这种情况下，current 指向包含要删除元素的节点，trailCurrent 指向 current 所指节点之前的节点。
4. 链表非空，但要删除的元素不在链表中。

函数deleteNode的定义如下所示：

```
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
```

逆向打印链表

有序链表的节点（如前所构造的链表）是按升序排列。但有些应用会要求按降序打印数据，这时就需要从后向前打印链表。下面将讨论的是函数reversePrint，当给出链表的指针时，此函数能逆向打印链表中的元素。

考虑图16.42中的链表。

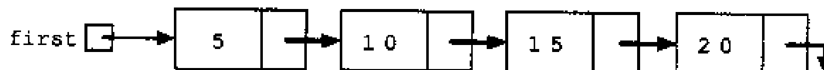


图16.42 链表

按照图16.42中的链表，输出的形式应当是：

```
20 15 10 5
```

由于只有一个方向的链接，所以不能从最后一个节点逆向遍历链表。下面将说明怎样利用递归逆向打印链表。

对于递归方式，在打印第一个节点的info之前，必须打印链表中其余所有的节点（第一个节点后面的部分）。同样，在将第二个节点后面的部分打印出来之前，不能打印第二个节点的info，依次类推。每次考虑一个节点的后面部分时，都将链表的大小减1，最后链表的大小减为0，这时停止递归。下面先用伪代码写出算法（假设current是指向链表的指针）：

```
if(current != NULL)
{
    reversePrint(current->link);  //print the tail
    cout<<current->info<<endl;    //print the node
}
```

在此，只有当指向链表的指针非空时，才打印链表。对于该递归调用，当current为NULL时（即指向最后一个节点的尾部），if语句为假，递归停止。注意由于打印语句（即打印节点的info）出现在递归调用的后面，当递归调用返回时，执行该打印语句。注意只有最后的语句执行完后，函数才会退出。

下面用C++编写上面的函数，以实现链表的逆向打印功能：

```
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);
```

其中first是nodeType<Type>类型的指针。

根据图16.42所示的链表跟踪这个语句，它是一个函数调用。由于形参是值参数，传递给形参的是实参的值，如图16.43所示。

printListReverse

在实现函数reversePrint之后，就可以编写函数printListReverse的定义。它的定义如下所示：

```
template<class Type>
void orderedLinkedListType<Type>::printListReverse() const
{
      reversePrint(first);
      cout<<endl;
}
```

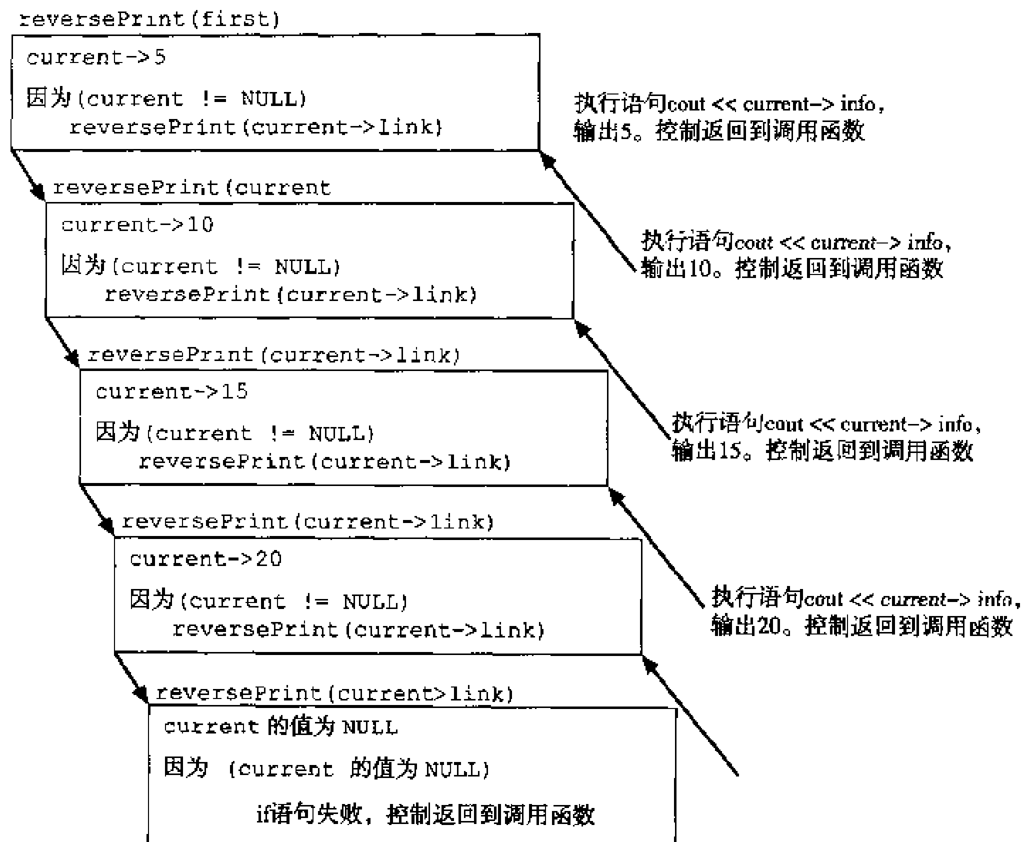


图 16.43 语句 “reversePrint(first);” 的执行

有序链表的头文件

为了完整，下面将说明怎样创建头文件以定义类orderedListType和此类链表的操作（假设类linkedListType的定义和实现操作的函数的定义都在头文件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
```

下面的程序测试有序链表中的各种操作：

```
//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;
}
```

程序运行结果　在本程序运行中，用户输入的数据加有阴影。

```
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
```

上面的输出很简单，分析留给读者作为练习。

16.4　双向链表

双向链表中每个节点有两个指针，一个指向下一个节点，另一个指向前一个节点。也就是说，每个节点都包含下一个节点的地址（除了最后一个节点），每个节点也都包含前一个节点的地址（除了第一个节点）（如图16.44所示）。

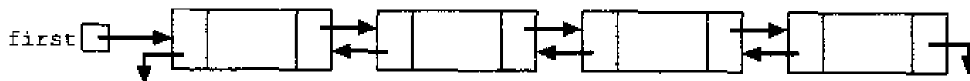


图16.44　双向链表

双向链表可以在两个方向上遍历。也就是说，可以从第一个节点开始遍历，若给出最后节点的指针的话，也可以从最后一个节点开始遍历。

双向链表的基本操作是：

1. 初始化链表
2. 删除链表
3. 检查链表是否为空
4. 检查链表是否为满
5. 在链表中查找给定的元素
6. 在链表中插入一个节点
7. 从链表中删除一个节点
8. 查找链表长度
9. 打印链表

下面说明有序双向链表的操作。在此，双向链表定义为ADT。

```
//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与前面介绍过的函数很类似。但是，这里由于每个节点有两个指针back和next，一些操作需要调整所有节点的这两个指针。对于插入和删除操作，因为无论以哪个方向遍历链表都可以，所以可以仅使用一个指针来遍历链表，我们可以将此指针称为current。可以通过current及current所指节点的back指针，来设置trailCurrent的值。除了拷贝构造函数和重载赋值运算符的函数之外，下面给出了所有函数的定义。拷贝构造函数和重载赋值运算符函数的定义留给读者作为练习。

默认的构造函数

默认的构造函数将双向链表初始化为空状态，它将first设置为NULL。

```
template<class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
      first= NULL;
}
```

isEmptyList

若链表为空，此操作返回true，否则返回false。若指针first是NULL，则链表为空。

```
template<class Type>
bool doublyLinkedList<Type>::isEmptyList()
{
    return(first == NULL);
}
```

删除链表

此操作删除链表中的所有节点，将链表置为空。它从第一个节点开始遍历链表并删除每个节点。

```
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

此操作将双向链表置为空状态，此工作由操作destroy完成。函数initializeList的定义如下所示：

```
template<class Type>
void doublyLinkedList<Type>:: initializeList()
{
   destroy();
}
```

链表长度

链表的长度是链表中节点的个数。此操作计算链表中节点的个数并返回结果。

```
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

此函数输出每个节点的info，它从第一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
```

查找链表

若查找项在链表中，则函数search输出“Item is found in the list”。查找算法和有序链表的查找算法完全一样。

```
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

由于是在双向链表中插入元素，所以需要调整特定节点的两个指针的值。和前面一样，需要找到新节点要插入的位置，创建节点，存储新元素，调整新节点及链表中特定节点的链接域。有下面 4 种情况：

1. 插入到空链表中
2. 插入到非空链表的起始处
3. 插入到非空链表的末尾处
4. 插入到非空链表的中间

情况 1 和情况 2 需要改变指针 first 的值。情况 3 和情况 4 很相似，下面说明情况 4。

考虑图 16.45 中的双向链表。

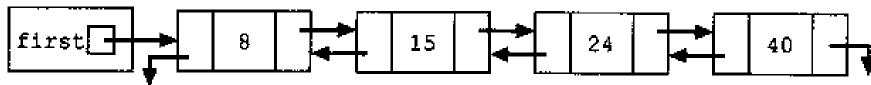


图 16.45 插入 20 之前的双向链表

假设要在链表中插入 20。插入 20 后，结果如图 16.46 所示。

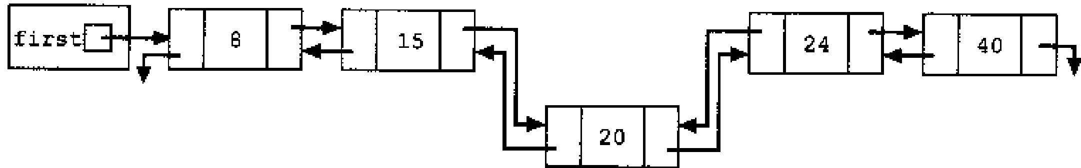


图 16.46 插入 20 之后的双向链表

从图 16.46 可以看出，节点 15 的 next 指针，节点 24 的 back 指针，以及节点 20 的 next 和 back 指针需要调整。

函数 insertNode 的定义如下所示：

```
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
        {
              //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

此操作从双向链表中删除给定的元素（若存在）。和前面一样，首先要查找链表以确定要删除的元素是否在链表中，查找算法也和前面一样。与操作 insertNode 类似，此操作也需要（若删除的元素在链表中）调整特定节点的两个指针。删除操作有下列几种情况：

1. 链表为空。
2. 要删除的元素是链表的第一个节点，这需要改变指针 first 的值。
3. 要删除的元素在链表中的某个地方。
4. 要删除的元素不在链表中。

下面将说明情况 3，考虑图 16.47 中所示的链表。

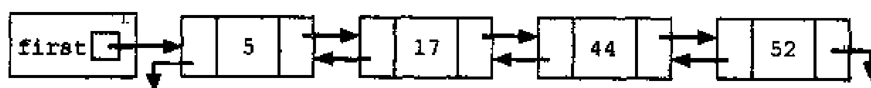


图 16.47 删除 17 之前的双向链表

假设要删除的元素是 17。首先，要用两个指针来查找链表并找到 info 值为 17 的节点，然后调整受影响的节点的链接（如图 16.48 所示）。

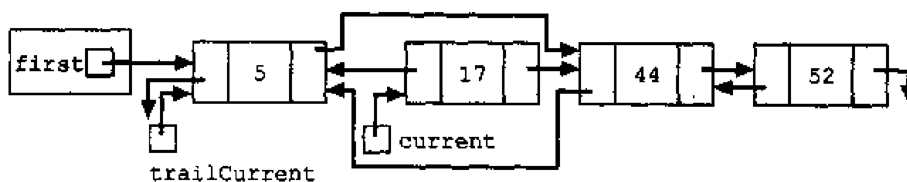


图 16.48 info 为 17 节点的前、后节点链接调整之后的链表

接下来删除 current 所指的节点（如图 16.49 所示）。

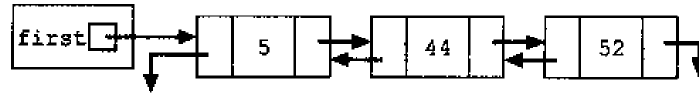


图 16.49 删除 info 为 17 的节点之后的链表

函数 deleteNode 的定义如下所示：

```
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 else
}//end deleteNode
```

16.5 程序范例：影碟店

对于家庭或个人来说，在周末或假期去影碟店租影碟看是一个不错的选择。一个新影碟店要开张了，但它还没有用来管理影碟和顾客的程序。该店经理希望有人能写一个程序，以使他的店正常运转起来。这个程序应当能完成下面的操作：

1. 出租影碟，也就是借出影碟
2. 收回或登记影碟
3. 创建该店所拥有的影碟的目录
4. 显示某个影碟的详细说明
5. 打印店中所有影碟的目录
6. 检查某个影碟是否在店中
7. 维护客户数据库
8. 打印每个客户租借的影碟的目录

下面为该影碟店编写一个程序。此例进一步说明了面向对象的设计方法，特别是继承和重载。

程序设计的需求指出影碟店有两个对象：影碟与顾客。下面将详细说明这两个对象，此外还需要维护两个表：

1. 店中影碟的表
2. 所有顾客的表

我们分两部分开发此程序。在部分 1 中设计、实现并测试影碟对象。在部分 2 中，设计和实现顾客对象，然后将之加入到部分 1 中的影碟对象。在完成部分 2 和部分 2 之后，就可以实现前面所列的所有操作。

部分 1：影碟对象

影碟对象 首先，确定与影碟有联系的属性（如图 16.50 所示）：

1. 电影的名字
2. 主角的名字
3. 制片人的名字
4. 导演的名字
5. 制片公司的名字
6. 店中拷贝的数目

图 16.50 影碟的数据成员

从表中可以得出能对影碟对象进行的操作是（如图16.51所示）：

a. 设置影碟信息如名称、主角、制片公司等。

b. 显示某个影碟的详细说明。

c. 检查店中拷贝数。

d. 借出（出租）影碟，也就是说，若拷贝数大于0，则将拷贝数减1。

e. 登记（归还）影碟。要收回影碟，首先要检查店中是否有这个影碟，如果有，将拷贝数加1。

f. 检查是否还有某个影碟，也就是说店中目前拥有的拷贝数大于0。

1. 设置影碟信息
2. 查找店内影碟的拷贝数量
3. 归还影碟
4. 借出影碟
5. 打印影碟名称
6. 检查影碟名称
7. 检查两个影碟是否相同
8. 更新店内影碟的拷贝数量
9. 设置店内影碟的拷贝数量

图16.51　影碟对象的操作

从影碟表中删除一个影碟需要查找表，也就是检查影碟的名称，从表中找到要删除的影碟。若两个影碟有一样的名称，则它们是相同的。

下面的类将影碟对象定义为ADT：

```
class videoType
{
      friend ostream& operator<<(ostream&, const videoType&);

public:
    void setVideoInfo(newString title, newString star1,
                      newString star2, newString producer,
                      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，还重载了类videoType的流插入运算符<<。注意类videoType处理字符串时，用到了第15章中设计的类newString。

下面分别编写类videoType中每个函数的定义。这些函数的定义非常简单易懂。

```
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;
}
```

影碟表 该程序要求维护一个店中所有影碟的表，并且在表中还能添加新的影碟。一般来说，不需要知道店中有多少影碟，而且增加或删除影碟会改变店中影碟的数目。因此，下面将用链表来创建影碟表（如图16.52所示）。

图16.52 影碟表

本章的前面部分定义了类linkedListType用来创建对象的链表，并定义了我们所要的一些基本操作，如在链表中插入或删除影碟。然而，另一些操作对影碟链表来说比较独特，如出租影碟、归还影碟、设置影碟的拷贝数，等等，这些操作在类 linkedListType 中没有。下面将从类 linkedListType 中派生类 videoListType 并增加这些操作（如图 16.53 所示）。

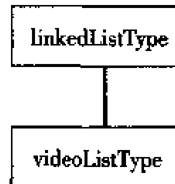


图 16.53　从 linkedListType 中派生 videoListType

类 videoListType 的定义如下所示：

```
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.
};
```

注意类 videoListType 是通过 public 继承从类 linkedListType 派生而来的，而且 linkedListType 是类模板，类 videoType 作为参数传递给了这个类。也就是说，类 videoListType 不是模板。由于要处理的是一个确定的数据类型，并不需要将类 videoListType 作为模板。因此，链表中每个节点的 info 的类型都是 videoType。通过类 videoType 的成员函数，可以访问 videoType 类型对象的一些成员，如 videoTitle 和 copiesInStock 等。

下面要给出的是实现类 videoListType 操作的函数的定义。

影碟链表的基本操作是归还和出租影碟，这两种操作都需要查找链表及在链表中找到归还的和出租的影碟位置。其他的操作，如查看某个影碟是否在店中，更新某个影碟的拷贝数等，也需要查找影碟链表。为了简化查找过程，下面编写一个函数来完成在影碟链表中查找某个特定影碟的工作。若找到了影碟，它将参数 found 设置为 true 并返回指向此影碟的指针，从而可以对影碟对象进行归还、出租等操作。注意函数 searchVideoList 是类 videoListType 的私有成员函数，这是因为它仅用来进行内部操作。首先要说明的是查找的顺序。

考虑图 16.54 中所示的影碟链表的节点。

图 16.54　影碟链表的节点

info 成员的类型是 videoType，它包含了影碟的必要信息。事实上，节点的 info 中有 7 个成员：videoTitle，movieStar1，movieStar2，movieProducer，movieDirector，movieProductionCo 及 copiesInStock（见类 videoType 的定义）。因此，影碟链表的节点有图 16.55 所示的形式。

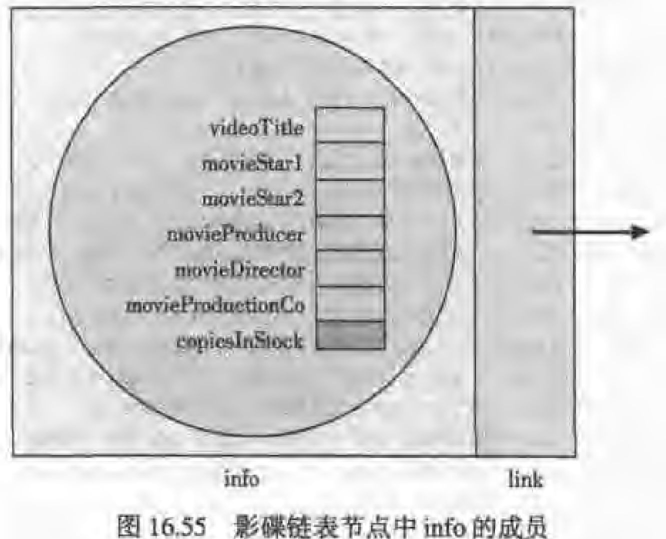


图 16.55　影碟链表节点中 info 的成员

这些数据成员都是 private，不能直接访问。类 videoType 的成员函数可以用来检查或设置某个成员的值。

假设指针 current 指向影碟链表中的某个节点（如图 16.56 所示）。

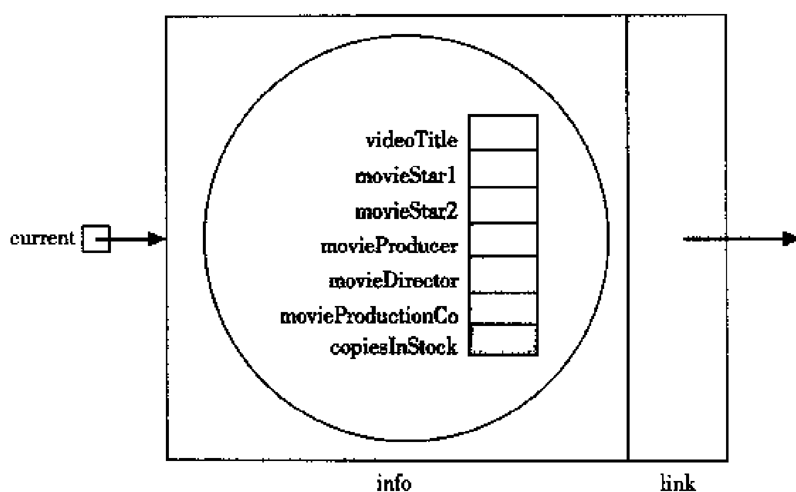


图16.56 指针current和影碟链表节点

则有：

```
current->info
```

存取节点的info部分。假设我们想知道此节点中存储的影碟名称是否与变量title所存储的名称相一致。如果一致的话，则表达式：

```
current->info.checkTitle(title)
```

的值是true；否则，值是false（注意成员函数checkTitle是带返回值的函数，见类videoType的声明）。

再举一个例子，假设我们希望将此节点的copiesInStock数据成员设为25。由于copiesInStock是私有数据成员，不能直接访问它。因此语句：

```
current->info.copiesInStock = 25;  //illegal
```

非法并会产生编译错误。应当按下面的方式使用成员函数setCopiesInStock：

```
current->info.setCopiesInStock(25);
```

既然已经知道了怎样访问存储在节点中的影碟的数据成员，下面将说明影碟链表的查找算法：

```
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
```

下面的函数定义实现了该查找算法：

```
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，将参数found设为true，并且参数current指向所要查找的节点。若查找失败，将found设为false，current将为NULL。

类videoListType其他函数的定义如下所示：

```
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;
       }
}
```

部分 2：顾客对象

顾客对象 顾客对象用来存储顾客的信息，如姓名、账号以及此顾客租借的影碟清单。因此，影碟店程序有两个对象：

1. 影碟对象，它包含影碟的必要信息，如前所述。
2. 顾客对象，它包含顾客的必要信息，下面将会说明。

   与两个对象相对应，需要两个链表：

   a. 店中所有影碟的链表（如前所述）

   b. 所有顾客的链表

下面，要说明的是顾客对象。
顾客的基本特征是：

1. 顾客的姓名
2. 顾客的账号
3. 所租借的影碟的清单

每个顾客都是一个人。在例 12.9（第 12 章）中已经设计了类 personType，并说明了对一个人的名字所能进行的操作。因此，可以从类 personType 中派生出类 customerType 并增加所需要的数据成员（如图 16.57 所示）。但是，首先还要重定义类 personType，以便利用面向对象设计所提供的新功能，如运算符重载等，然后再派生类 customerType。

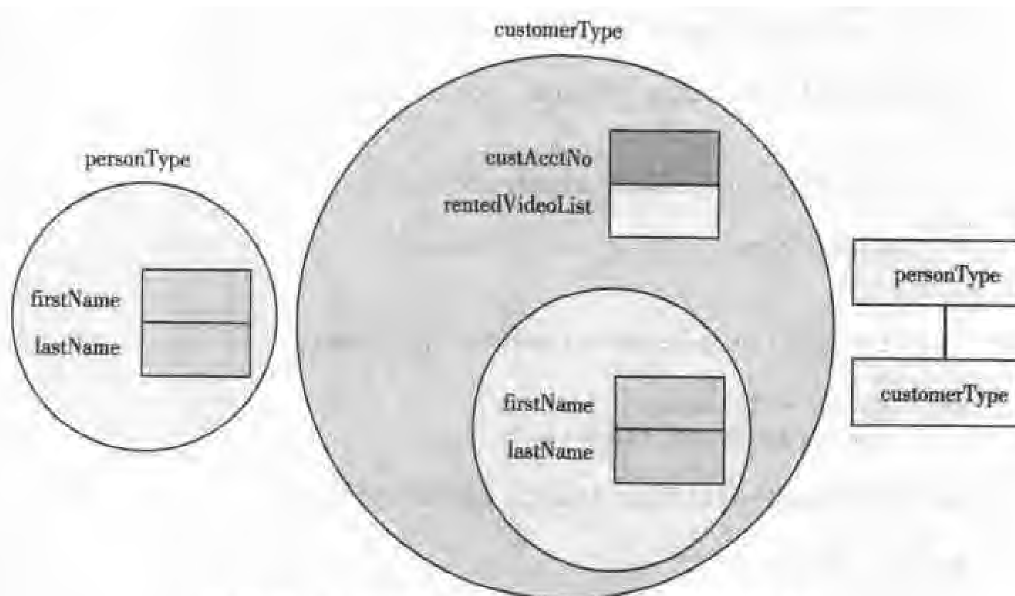


图 16.57 类 personType 和 customerType 及其数据成员

personType 类对象的基本操作是：

1. 打印姓名
2. 设置姓名
3. 显示姓名

类似地，customerType 类的对象的基本操作是：

1. 打印姓名、账号和租借的影碟的清单

2. 设置姓名和账号
3. 租借影碟，也就是将所租借的影碟加到清单中
4. 归还影碟，也就是从清单中删除租借的影碟
5. 显示账号

顾客对象的实现细节留给读者作为练习（见本章结尾的编程练习 5）。

主程序

下面编写一个主程序来测试影碟对象。假设影碟的数据都存储在一个文件中，程序需要打开文件并创建影碟店的影碟链表。输入文件的数据格式如下所示：

```
video title(电影名字)
movie star1
movie star2
movie producer
movie director
movie production co.
number of copies
  .
  .
  .
```

我们将写一个函数 createVideoList 来从输入文件中读取数据并创建影碟链表，还要编写一个函数 displayMenu 来显示用户不同的选择——如归还影碟或租借影碟。main 函数的算法如下所示：

1. 打开输入文件
   如果输入文件不存在，退出程序
2. 创建影碟链表（createVideoList）
3. 显示菜单（displayMenu）
4. While 循环

完成各种操作。

打开输入文件很简单，下面主要说明第2步和第3步，它们通过编写两个单独的函数：createVideoList 和 displayMenu 来完成。

createVideoList

此函数从输入文件中读入数据并创建影碟链表。

由于要从文件中读取数据，而输入文件已在 main 函数中打开，所以将输入文件的指针传递给此函数，同时还要将在main函数中声明的影碟链表指针传递给此函数。下一步要读取每一个影碟的数据并将此影碟插入到链表中。主要算法如下所示：

a. 读取数据并将其存储在影碟对象中
b. 将影碟对象插入到链表中
c. 对文件中每个影碟的数据重复上面步骤 a 和步骤 b

displayMenu

此函数告诉用户可做什么，它包含下面的输出语句：

a. 选择下面一项
b. 1：检查某个影碟是否在店中
c. 2：出租影碟

d. 3：归还影碟
e. 4：检查某个影碟是否在店中
f. 5：打印所有影碟的名称
g. 6：打印所有影碟的清单
h. 9：退出

main函数算法步骤4的伪代码如下所示：

```
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
```

主程序代码清单

```
#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;
}
```

16.6 小结

1. 链表是由一系列称为节点的元素构成的表，节点的顺序由每个节点中存储的地址（称为链接）决定。
2. 指向链表的指针——也就是指向链表中第一个节点的指针，存储在单独的单元中，称为链表的头（head）或起始（first）。

3. 链表是一个动态数据结构。
4. 链表的长度是链表中节点的数目。
5. 在链表中插入或删除节点不需要移动数据，仅需要调整指针。
6. （单向）链表只能按一个方向遍历。
7. 链表的查找是顺序的。
8. 链表的头指针总是指向链表的第一个节点。
9. 要遍历链表，必须要使用头指针以外的另一个指针，它初始化为指向链表的第一个节点。
10. 类对象作为值传递时，拷贝构造函数将实参的值拷贝到形参中。
11. 有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，实际链表元素在头和尾节点之间。
13. 在有头和尾节点的链表中，若链表中只有头和尾节点，则它是空的。
14. 在双向链表中，每个节点有两个链接：一个指向下一个节点，一个指向前一个节点。
15. 双向链表可从两个方向遍历。
16. 在双向链表中，插入和删除节点需要改变节点的两个指针。

16.7 练习

1. 判断下面说法的正误。
   a. 在链表中，元素的顺序是由存储元素的节点的创建顺序来决定。
   b. 在链表中，分配给节点的内存是顺序的。
   c. 单向链表可从两个方向遍历。
   d. 由于链表不是可随机访问的数据结构，所以节点只能插在链表的开始或结尾。
   e. 在有头和尾节点的链表中插入(和删除)元素要比普通链表简单，这是因为它不存在特殊情况。
   f. 不能使用头指针遍历链表。

考虑图16.58所示的链表。假设节点是常用的info-link形式，根据此链表来回答练习2至练习7。如果需要，还可以声明其他变量（假设List，p，s，A和B都是nodeType类型的指针）。

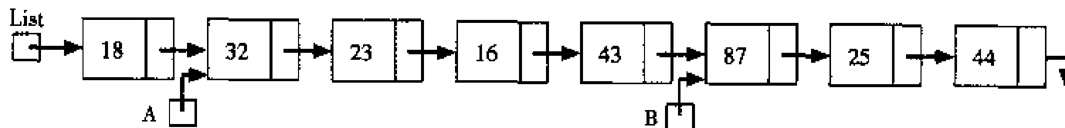


图16.58 练习2至练习7的链表

2. 下面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. 下面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. 判断下面的语句是否合法，如果非法，说明为什么。

   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. 写出能完成下面功能的C++语句。

   a. 让A指向包含info为23的节点。

   b. 让List指向包含16的节点。

   c. 让B指向链表的最后一个节点。

   d. 让List指向空链表。

   e. 将值为25的节点的值改为35。

   f. 创建info值为10的节点，并将其插入到A所指节点的后面。

   g. 删除info值为23的节点，并释放分配给此节点的内存空间。

6. 下面C++代码的输出是什么?

```
p = List;

while(p != NULL)
  cout<<p->info<<" ";
  p = p->link;
cout<<endl;
```

7. 如果下面C++代码合法，请说明输出；如果非法，解释原因。

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. 假设节点是常见的info-link形式，info是int型，请说明下面C++代码运行的结果（List和ptr是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. 考虑下面的C++语句（类linkedListType在本章中有定义）。

```
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();
```

上面程序段的输出是什么?

10. 假设输入数据是:

```
18 30 4 32 45 36 78 19 48 75 -999
```

下面C++代码的输出是什么(类linkedListType在本章中有定义)?

```
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;
```

16.8 编程练习

1. 第13章中的编程练习6最多只能处理500个条目。用链表重新编写此程序以处理任意数目的条目,并在程序中加入下列操作:
   a. 在地址簿中增加或删除一个新条目。
   b. 程序终止时,将地址簿中的数据写到磁盘上。
2. 通过增加下述操作来扩展类linkedListType:
   a. 在链表中找到并删除info值最小的节点(仅第一次查找时删除,只遍历链表一次)。
   b. 在链表中找到并删除给定的info值的所有出现(只遍历链表一次)。
3. 链表经常还需要一些其他操作,如divideMid和divideAt。
   a. divideMid:此操作将给定的链表分为两个(基本)相等的子表。例如,假设给定的表是13 72 89 65 34,则两个子表是13 72 89和65 34。类似,如果初始链表是12 67 34 65,则两个子链表是12 67和34 65。
   (i) 按如下方式增加类linkedListType的操作:

```
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.
```

考虑下面语句：

```
linkedListType<int> myList;
linkedListType<int> subList,
```

假设 myList 指向元素是 34 65 27 89 12 的链表，语句：

```
myList.divideMid(subList);
```

将 myList 分成两个子串：myList 指向元素为 34 65 27 的链表，subList 指向元素为 89 12 的子链表。

(ii) 编写实现操作 divideMid 的函数模板的定义。

b. divideAt：此操作按照给定的 info 分开链表。

假设 oldList 指向的链表有如下元素：

```
10 18 34 6 28 92 56 48
```

要将链表从 info 为 6 的节点处分开，则两个子链表是：

```
10 18 34 和 6 28 92 56 48
```

(i) 将下面的操作加入到类 linkedListType 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.
```

考虑下面语句：

```
linkedListType<int> myList;
linkedListType<int> otherList;
```

假设 myList 指向的链表有如下元素：

34 65 18 39 27 89 12。语句：

```
myList.divideAt(otherList,18);
```

将 myList 分为两个子链表：myList 指向的链表中的元素是 34 65，otherList 指向的链表中的元素是 18 39 27 89 12。

(ii) 编写实现操作 divideAt 的函数模板的定义。

4. a. 将下面的操作加到类 orderedLinkedListType 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
```

例如：考虑下面语句：

```
orderedLinkedListType<int> newList;
orderedLinkedListType<int> list1;
orderedLinkedListType<int> list2;
```

假设 list1 指向的链表有元素 2 6 7，list2 指向的链表中元素 3 5 8。语句：

```
newList.mergeLists(list1,list2);
```

以 2 3 5 6 7 8 的顺序创建了一个新链表，且对象 newList 指向此链表。同时，在上面的语句执行后，list1 和 list2 为空。

b. 编写函数模板 mergeLists 的定义以实现操作 mergeLists。

5.（程序范例：影碟店）

a. 完成并实现在该程序范例中定义的类 customerType。

b. 设计并实现类 customerListType 以创建并维护影碟店的顾客链表。

6.（程序范例：影碟店）按要求完成影碟店程序的设计和实现。

第17章 栈和队列

本章要点：

- 了解栈（Stack）
- 了解各种不同的栈操作
- 理解怎样用数组实现栈
- 理解怎样用链表实现栈
- 了解栈的应用
- 理解怎样使用栈来消除递归
- 了解队列
- 了解队列的各种操作
- 理解怎样用数组实现队列
- 理解怎样用链表实现队列
- 了解队列的应用

本章将讨论两种极为有用的数据结构：栈和队列。这两种数据结构在计算机科学领域中有着广泛的应用。

17.1 栈

假定有一个含几个函数的程序。为方便起见，假设这几个函数是A，B，C，D。现在假设函数A调用了函数B，函数B调用了函数C，而函数C调用了函数D。当函数D结束时，控制就交还给了函数C；当函数C结束时，控制交还给了函数B；而当函数B结束时，控制又交还给了函数A。你有没有想过，在程序执行时，计算机是如何追踪这些函数调用序列的？如果是递归函数，那么计算机又是如何追踪这些递归调用的呢？在第16章，我们设计了一个递归函数用来逆向打印链表。而逆向打印链表的非递归算法又该怎样编写呢？

本节将讨论一种称为栈（Stack）的数据结构。栈用来在计算机上实现函数调用。当然，也可以用栈来将递归算法转化为非递归算法，尤其是那些不是尾递归的递归算法。栈在计算机科学中还有许多其他应用。在介绍实现栈所必需的操作之后，我们将研究一些有关栈的应用。

栈是一个由同类元素组成的表，元素的插入和删除只能在表的一端进行。这一端称为栈顶。例如，你在自助餐厅中，要取走一叠碟子中的第二个碟子就必须先拿走第一个碟子。又比如，要想取出你最钟爱的计算机类图书（而它正压在你的数学和历史书下面），就必须先取出数学书和历史书。而后，那本计算机书才能变成最上面的书，即栈顶元素。图17.1给出了栈的一些举例。

位于栈底的元素在栈中存在的时间最久。而栈顶元素则是最近入栈的元素。由于栈的插入和删除操作在栈的一端（也就是栈顶）进行，所以栈遵守后进先出的规律。因而，栈又被称为后进先出（LIFO，Last In First Out）的数据结构。

栈 一种插入和删除均在一端进行的数据结构；一种后进先出（LIFO）的数据结构。

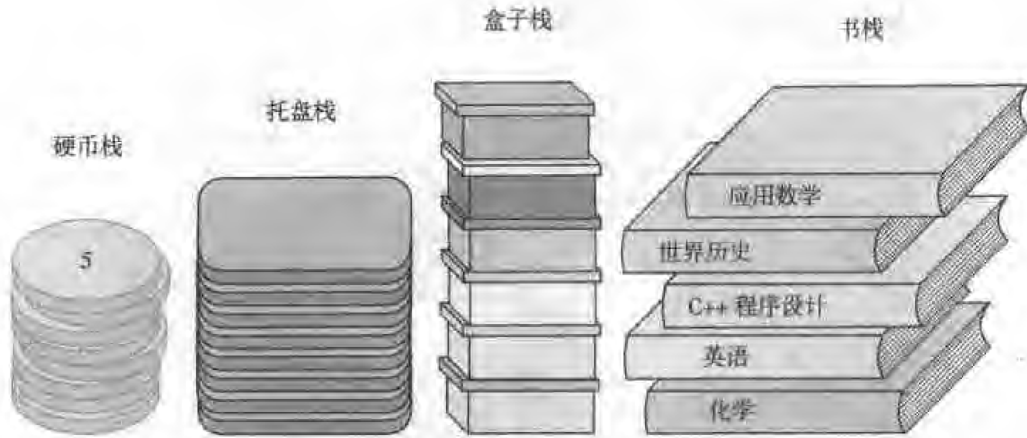


图 17.1　各种类型的栈

现在已经清楚栈是怎么回事了，让我们来看看在栈上可以执行哪些操作。由于可以向栈中加入新的元素，所以可以在栈上实现插入操作，又称为压栈（Push）。压栈操作将一个元素插入到栈顶。同样由于可以将栈顶的元素取出，所以可以在栈上实现取出操作，又称为退栈（Pop）。退栈操作将栈顶元素取出。图 17.2 至图 17.7 说明了压栈操作和退栈操作。

压栈操作和退栈操作以如下方式工作：假定地板上有几个盒子，需要将其放到桌面（桌面视为栈）上。最初，所有的盒子都在地板上，并且栈是空的（如图 17.2 所示）。

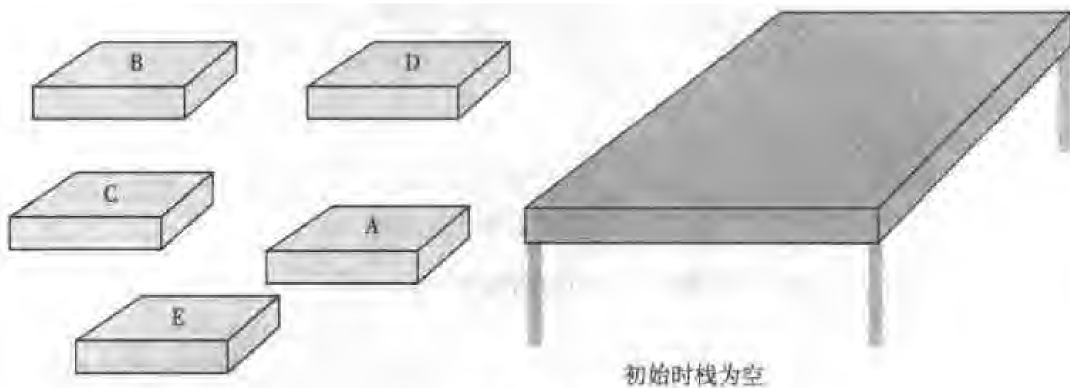


图 17.2　空栈

首先，我们将盒子 A 压入栈中。压栈操作后，栈的结构如图 17.3 所示。

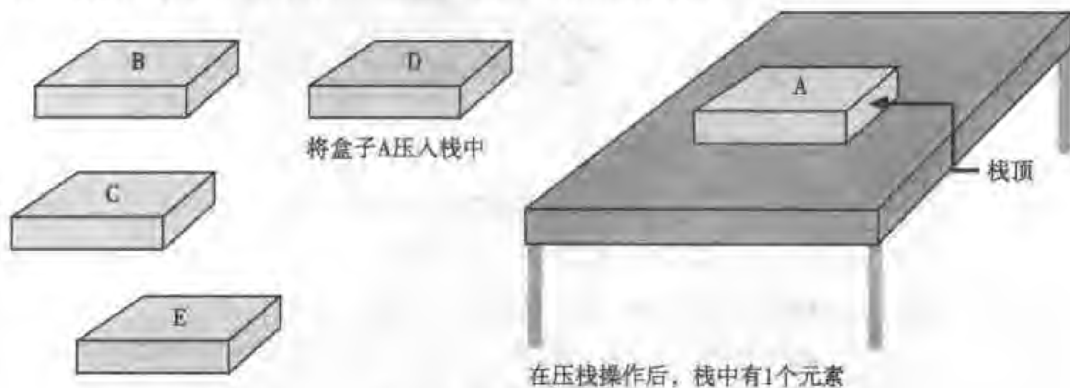


图 17.3　把盒子 A 压入后的栈

将盒子B压入栈中。压栈操作后，栈的结构如图17.4所示。

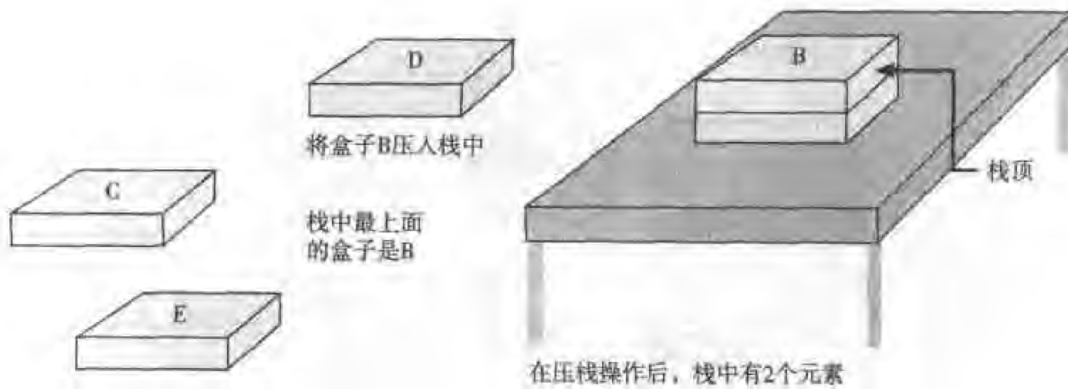


图17.4 把盒子B压入后的栈

接着，我们将盒子C压入栈中。压栈操作后，栈的结构如图17.5所示。

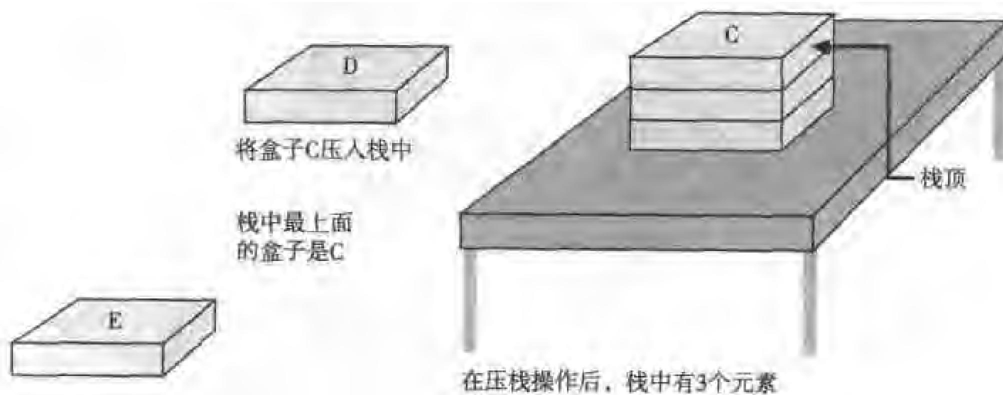


图17.5 把盒子C压入后的栈

然后，将盒子D压入栈中。压栈操作后，栈的结构如图17.6所示。

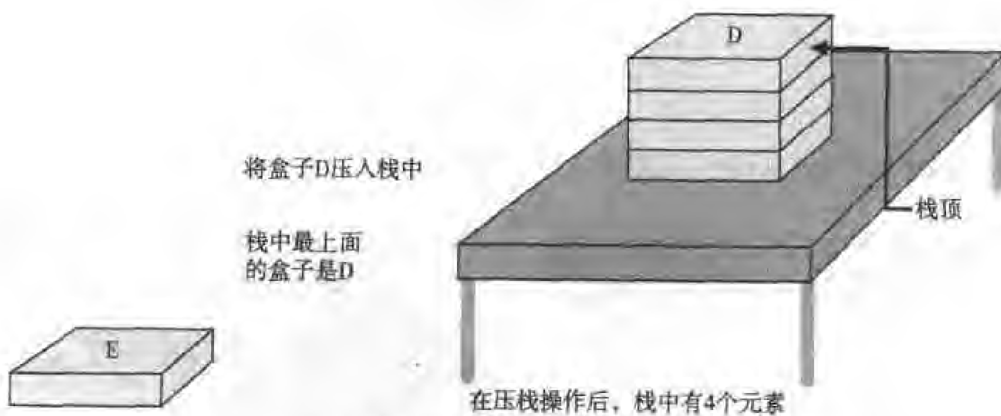


图17.6 把盒子D压入后的栈

接下来，我们进行退栈操作。在退栈操作后，栈的结构如图17.7所示。

只有当栈中有元素时才能从栈中取出元素；同样，只有栈中还有空闲空间时才可能压入元素。下面，除了压栈操作push及退栈操作pop外，接着要介绍的操作是isFullStack（检验栈是否满）和isEmptyStack（检验栈是否空）。当插入和删除元素时栈会随之变化。因此，在第一次使用之前，栈一定是空的。这样，就需要一个操作，用来初始化栈使之处于栈空的状态，我们称之为initializeStack。还有一种操作

叫destroyStack，该操作用来删除栈中所有元素使栈为空。总之，要成功实现一个栈，至少需要这6种操作（将在下一节讨论）。根据需求的不同，可能还需要其他栈操作。

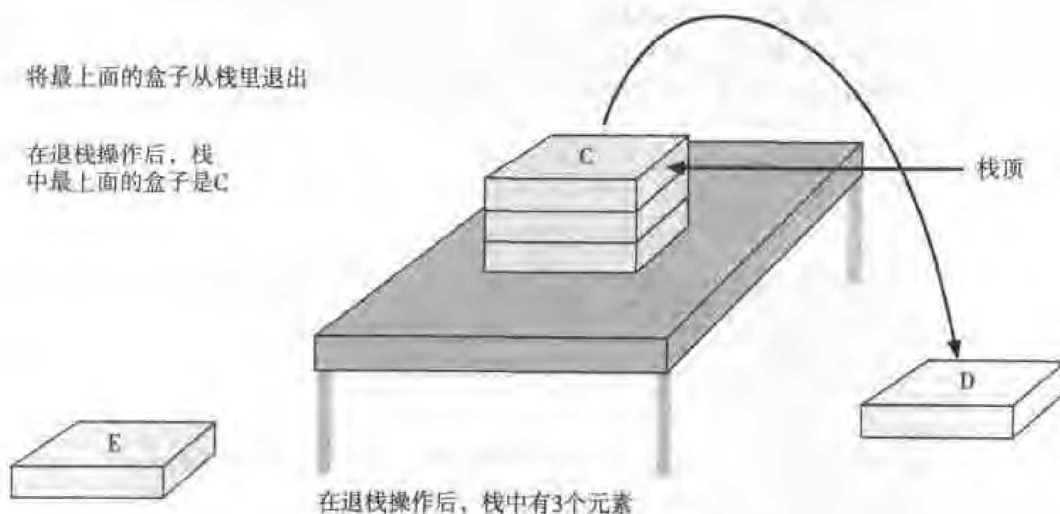


图 17.7　在退栈操作后的栈

17.1.1　栈操作

基本的栈操作有以下几种：

1. initializeStack: 初始化栈使之处于栈空的状态。
2. destroyStack: 删除栈中所有的元素使栈成为空栈。
3. isEmptyStack: 检验栈是否为空。如果栈是空的，将返回 true 值；否则，返回 false 值。
4. isFullStack: 检验栈是否为满。如果栈是满的，将返回 true 值；否则，返回 false 值。
5. push：向栈顶压入一个新的元素。该操作的输入是栈以及要加入的元素。执行该操作之前，栈必须存在且没有满。
6. pop：退出栈顶元素，并将该元素存入由 poppedElement 指定的空间。该操作的输入是栈以及用于存储栈顶元素的地址。执行该操作之前，栈必须存在且非空。

现在我们考虑抽象栈数据结构的实现。由于 C++ 并没有提供压栈操作和退栈操作所必需的函数，所以我们就必须为这些栈操作编写相应的函数实现。

由于栈中所有元素的类型都相同，因此既可以使用数组结构也可以使用链表结构来实现栈。这两种实现方式都非常有用，本章将对它们分别加以讨论。

17.2　使用数组实现栈

由于栈中所有元素的类型都相同，因而可以用数组来实现栈。可以将栈的第一个元素放入数组的第一个单元中，将栈的第二个元素放入数组的第二个单元中，以此类推。栈顶元素就是最后放入数组中的元素。

在这种实现方式中，栈的元素存放在数组中，而数组是一种随机存储方式的数据结构；也就是说，你可以直接访问数组中的任何一个元素。然而，根据定义我们可以知道，栈是一种只能在一端进行访问（压入或者退出）元素的数据结构，即后进先出的数据结构。因而，只能在栈顶访问元素，而不能在栈底或栈中间访问。栈的这种特征极为重要，必须在一开始就对它有明确认识。

要追踪数组中栈顶的位置，我们可以简单地声明另一个变量，称之为top。

下面要介绍类stackType，将栈定义为一种抽象数据类型（ADT）。通过使用指针，可以动态地分配数组，因而可以让用户指定数组大小（也就是栈大小）。假定栈默认的大小是100。由于类stackType中有一个指针数据成员（指向存储栈元素的数组），必须重载赋值运算符，并增加拷贝构造函数和析构函数。下面，将给出一个类属的栈定义。在具体应用中，可以在声明栈对象时指定栈元素的类型。

```
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++中数组下标从0开始，所以有必要区分top的值和top指定的数组元素位置。如果top是0，则栈是空栈；否则，栈非空并且top−1指定栈顶元素的位置。

图17.8说明了这种数据结构。stack是stackType类型的对象。注意top的取值范围是0到maxStackSize。如果top是非0，则top−1就是当前栈顶元素的下标。假定maxStackSize = 100。

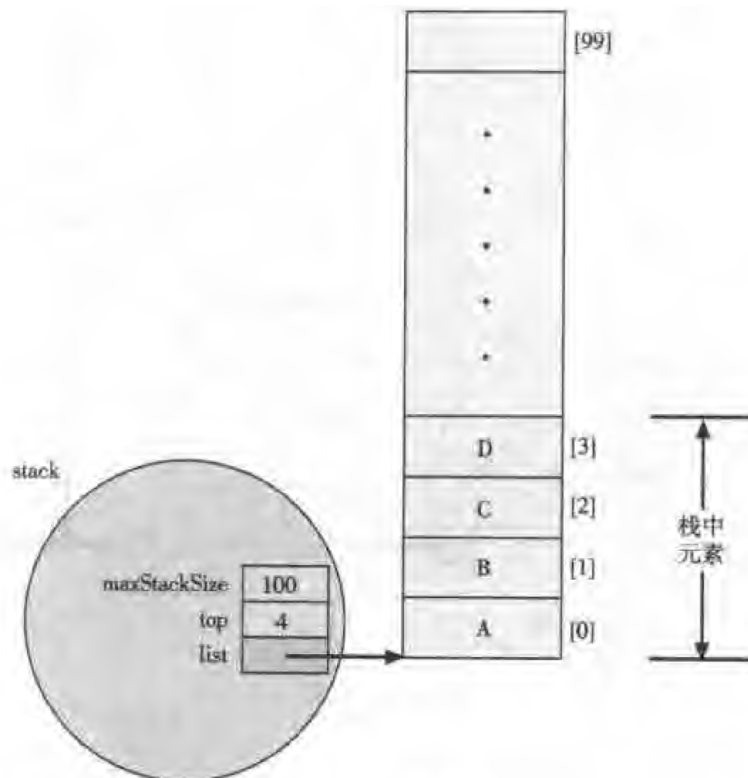


图 17.8 栈的示例

注意指针list指向数组（存放栈元素）的基地址，即第一个数组元素的地址。接下来将定义类stackType的成员函数，以实现栈的操作。

初始化栈

我们先来看一下栈操作initializeStack。由于top的值用来指示栈是否为空，所以可以简单的将top设为0就可完成栈的初始化工作（如图17.9所示）。

函数initializeStack的定义如下所示：

```
template<class Type>
void stackType<Type>::initializeStack()
{
      top = 0;
} //end initializeStack
```

删除栈

在栈的数组实现方式中，destroyStack操作同initializeStack操作很类似。如果我们设定top的值为0，则栈中的所有元素都被删除，尽管这些元素实际仍在栈中（它们被视为垃圾），top的值能说明栈是否为空。

```
template<class Type>
void stackType<Type>::destroyStack()
{
```

```
    top = 0;
} //end destroyStack
```

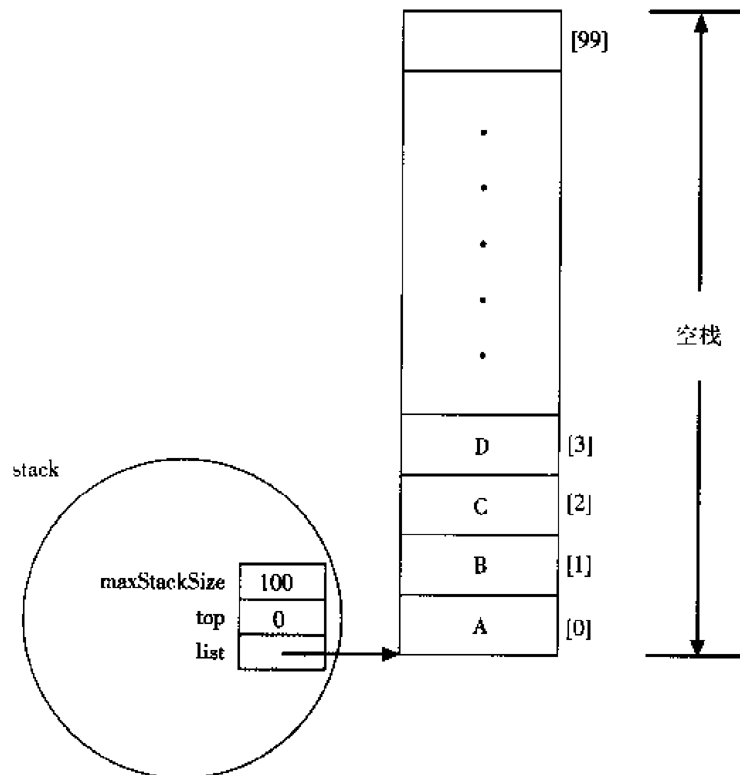


图 17.9 空栈

空栈

正如上面所示，top 的值可以说明栈是否为空。如果 top 是 0，则栈是空栈；否则，栈非空。函数 isEmptyStack 的定义如下所示：

```
template<class Type>
bool stackType<Type>::isEmptyStack()
{
      return(top == 0);
} //end isEmptyStack
```

满栈

下面，我们看一下 isFullStack 操作。如果 top 的值等于 maxStackSize，则栈是满的。函数 isFullStack 的定义如下所示：

```
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简单。带有参数的构造函数创建一个由用户指定大小的数组来存放栈元素，并将 top 设置为 0。如果用户没有指定栈的大小，构造函数创建一个默认值大小是 100 的数组。析构函数只是简单地释放动态数组（也就是栈）所占用的内存空间，并设置 top 为 0。构造函数和析构函数的定义如下所示：

```
    //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
```

拷贝构造函数

当一个栈对象被作为值参传递给一个函数时，程序就会调用拷贝构造函数。它将实参的数据成员拷贝到形参的相应数据成员中。拷贝构造函数的定义如下所示：

```
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

前面讲过，在包含指针数据成员的类中，赋值运算符必须显式地重载。对于类stackType而言，其赋值运算符（=）的重载函数定义如下所示：

```
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=
```

压栈

将一个元素压入栈中，可以分为两个步骤。top的值指明栈中元素的个数，而top-1则指定栈顶元素的位置。因此，压栈操作应该分为以下几个步骤：

1. 将newItem存放到由top指定的数组元素中
2. 将top增加1

图17.10和图17.11说明了压栈操作。

(a) 假定在执行压栈操作之前，栈的结构如图17.10所示。

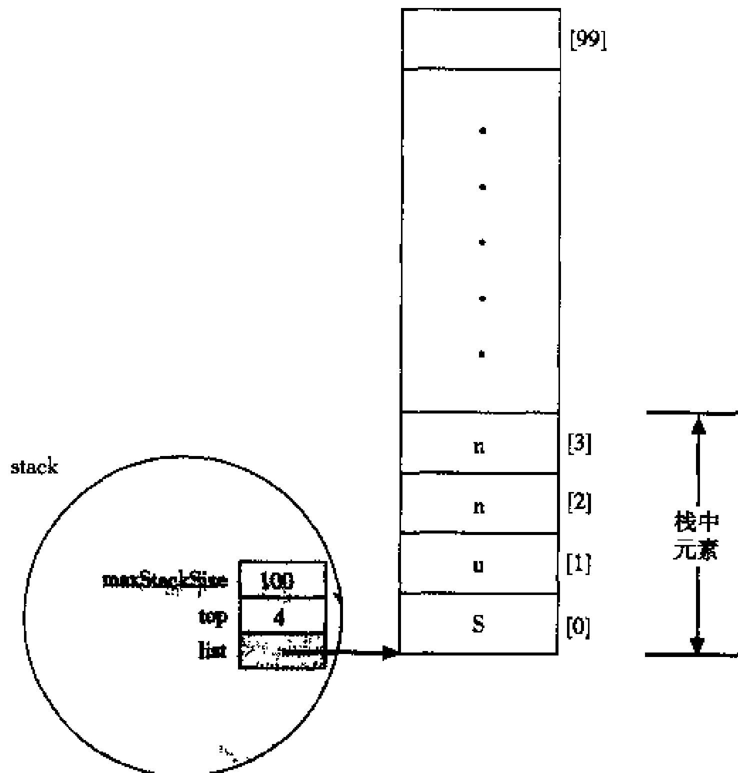


图17.10 压入y之前的栈

(b) 假定newItem的值是'y'。在压栈操作之后，栈的结构如图17.11所示。

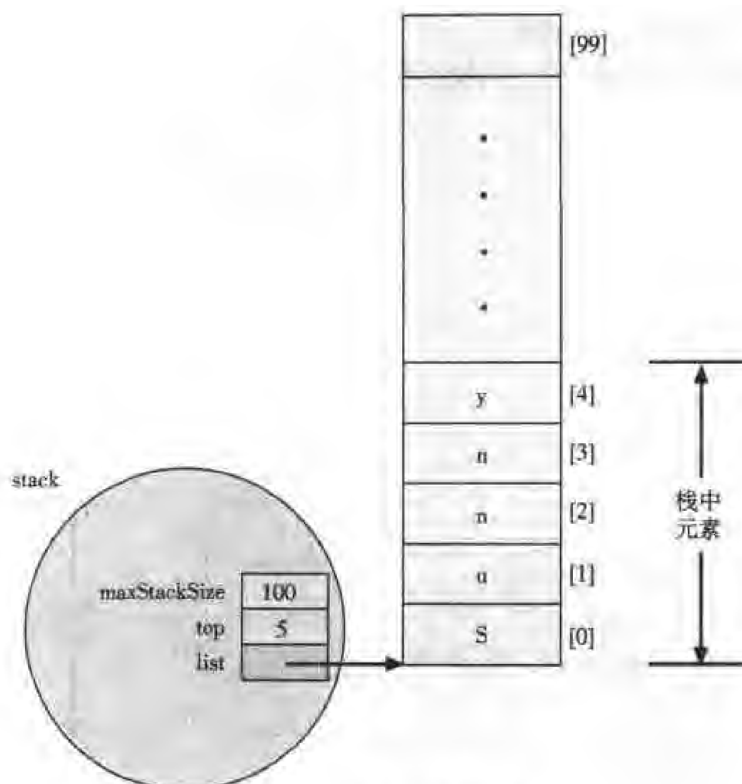


图 17.11　压入 y 之后的栈

根据前面的算法，push 函数的定义如下所示：

```
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
```

这里的 push 函数并没有在压入 newItem 之前检验栈是否已满。所以，在调用压栈函数之前，必须检查栈是否已满。这样，对压栈的调用（假设 stack 是类 stackType 的一个对象）应该是：

```
if(!stack.isFullStack())
   stack.push(newItem);
```

如果试图在已满的栈中加入新元素，将导致上溢（Overflow）错误。可以采用多种方法来检查上溢错误。上面是一种检查方法，还可以在 push 函数内部进行溢出检查。这种 push 函数的算法如下所示：

```
If the stack is full
      Overflow is true
Else
      Overflow is false
      Add the new item to the stack
      Increment top
```

退栈

将栈顶元素退出（删除），只需要做压栈的逆操作。所以，退栈操作的算法如下所示：

1. 将 top 减 1
2. 将栈顶元素存储到由 poppedItem 指定的内存单元

图 17.12 和图 17.13 说明了退栈操作：

(a) 假定在退栈操作执行之前，栈的结构如图 17.12 所示。

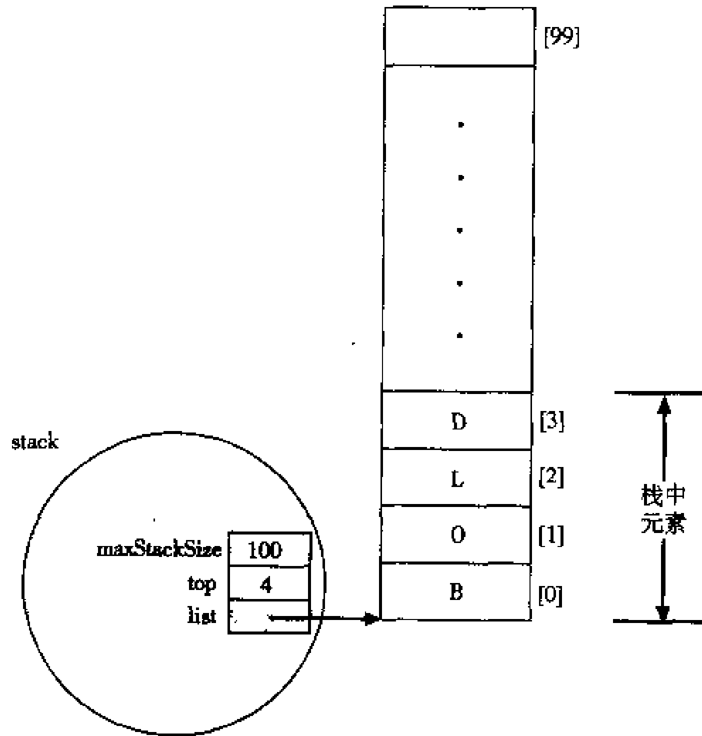


图 17.12 退出 D 之前的栈

(b) 在退栈操作执行之后，栈的结构如图 17.13 所示。

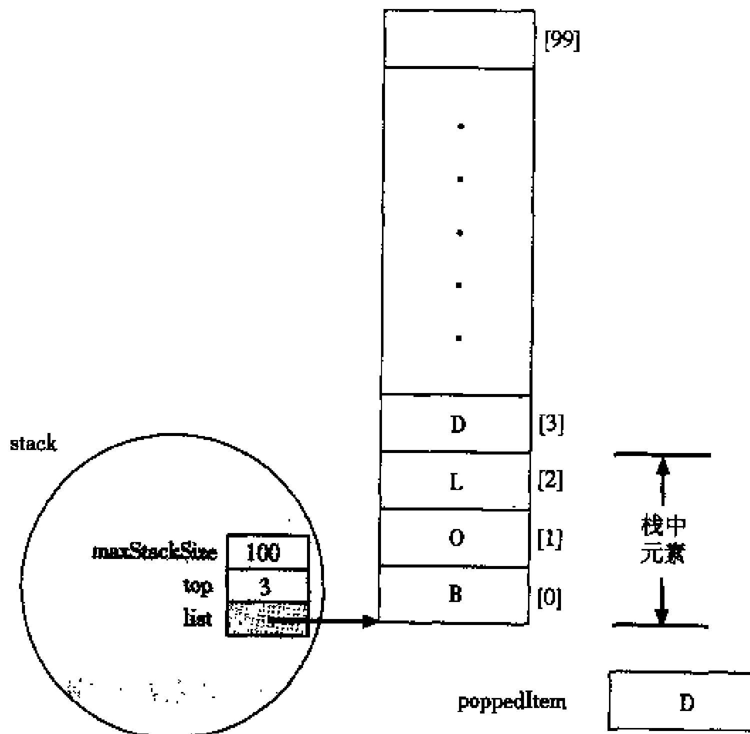


图 17.13 退出 D 之后的栈

根据上面的算法，pop 函数的定义如下所示：

```
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
```

与函数 push 类似，在调用函数 pop 之前必须检验栈是否为空（假设 stack 是类 stackType 的一个对象）。

```
if(!stack.isEmptyStack())
   stack.pop(poppedItem);
```

退出或者删除空栈中的元素将导致下溢（Underflow）错误。跟压栈类似，也可以采用多种方式对退栈进行出错检查。如果在函数 pop 内部进行下溢检查，伪代码的算法如下所示：

```
If the stack is empty
      Underflow is true
Else
      Underflow is false
      Decrement top
      Assign the top element of the stack to poppedItem
```

17.2.1　栈头文件

在介绍了如何实现栈操作之后，可以将类的定义以及栈操作的实现汇集起来形成一个栈的头文件。为了完整起见，下面给出头文件（为节省空间，只给出类的定义，而没给出说明文档）。假定包含类 stackType 的头文件名称是 myStack.h。本书中使用到栈的程序都将引用这个头文件。

```
//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17.1 在给出程序范例之前，让我们先编写一个使用类stackType，并测试某些栈操作的简单程序。程序及其输出如下所示：

```
//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.
```

建议读者对该程序进行代码走查。

17.3 程序范例：最高GPA

在本程序范例中，将编写一个C++程序读取一个数据文件，该文件内容包括每个学生的GPA以及学生的姓名。程序将输出最高的GPA以及获得这个GPA的所有学生名单。程序仅对输入文件扫描一次。

输入 程序读取数据文件，该文件包含每个学生的GPA及学生姓名。示例数据如下所示：

```
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GPA以及获得该GPA的所有学生姓名。例如，对于如上所示数据，最高GPA是3.9，获得该GPA的学生是Kathy和David。

程序分析和算法设计

首先，我们读取第一个GPA和学生姓名。由于这是读取的第一个数据项，因而也是当前最高的GPA。接下来，我们读取第二个GPA和学生姓名。然后，将这个（第二个）GPA与当前最高的GPA相比较，会出现三种情况：

1. 新GPA大于当前最高GPA。这种情况下，应该：
   (a) 更新当前最高GPA的值
   (b) 删除栈，即将栈中所有的学生姓名删除
   (c) 将获得当前最高GPA的学生姓名入栈
2. 新GPA等于当前最高GPA。这种情况下，将新的学生姓名压栈。
3. 新GPA小于当前最高GPA。这种情况下，丢弃新读入的学生姓名。

接下来，我们继续读取GPA和学生姓名，重复步骤1到步骤3，直到读到文件末尾。通过以上讨论，需要定义以下变量：

```
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，这里使用第15章设计的newString类存储学生姓名。这样，就可以使用赋值运算符将一个姓名存储到一个newString类型的变量中。

根据上面讨论，可以得到如下算法：

1. 声明变量。
2. 打开输入文件。
3. 如果输入文件不存在，退出程序。
4. 将输出的浮点数设置为带有小数点及小数末尾的0，小数点后保留两位小数。

5. 读取 GPA 和学生姓名。
6. highestGPA = GPA;
7. 初始化栈。
8. while（没到文件尾）

```
{
8.1 if (GPA > highestGPA)
    {
    8.1.1 clearstack(stack);
    8.1.2 push(stack, 学生姓名);
    8.1.3 highestGPA = GPA;
    }
8.2 else
    if( GPA 等于 highestGPA)
        push(stack, 学生姓名);
8.3 读入 GPA 和学生姓名;
}
```

9. 输出最高 GPA。
10. 输出获得最高 GPA 的学生名单。

完整的程序代码清单

```
//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:Ch17_HighestGPAData.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;
}
```

程序运行结果

输入文件（a:Ch17_HighestGPAData.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
```

17.4 使用链表实现栈

在栈的数组（线性）表示中，由于数组的大小是固定的，只有有限数目的元素可以被压入栈中。如果压入栈的元素个数超过了数组的大小，程序可能会出错终止。我们必须解决这种问题。

通过指针变量可以动态分配和释放内存，所以通过链表可以动态地组织数据（例如一个顺序表）。下面，将使用指针和链表来动态地实现一个栈。

回忆栈的线性表示：top的值表示栈中元素的个数，而top-1的值指向了栈顶元素。通过top，就可以找到栈顶元素，以及检验栈是否为空。

同线性表示类似，在链表表示中，top被用来确定栈顶元素的位置。然而，这里有一个细微的区别：前者的top给出的是数组下标；而后者的top给出的则是栈顶元素的地址（内存位置）。

下面语句定义了一个链表栈的抽象数据类型：

```
//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
};
```

下面给出了空链表栈和非空链表栈的示意图。

例 17.2

(a) 空栈：假定 stack 是 linkedStackType 类型的一个对象（如图 17.14 所示）。

图 17.14 空链表栈

(b) 非空栈（如图 17.15 所示）。

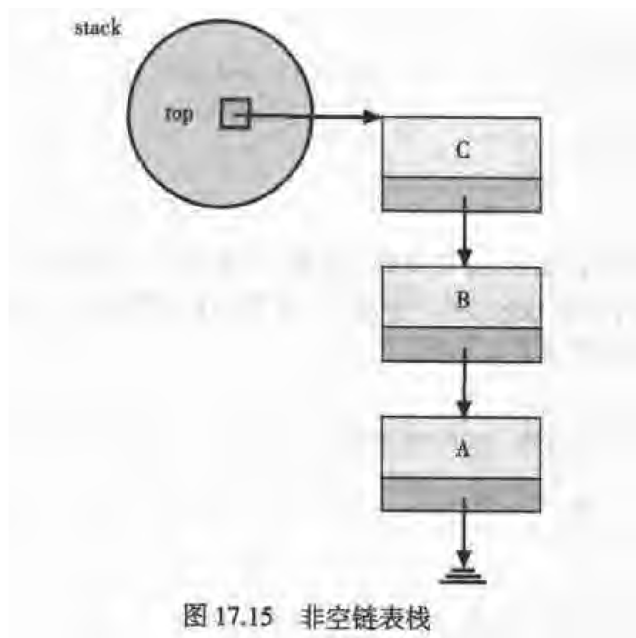


图 17.15 非空链表栈

在图 17.15 中，栈顶元素是 C。也就是说，最后一个入栈的元素是 C。

接下来，我们讨论如何定义实现链表栈操作的函数。

默认构造函数

首先考虑的操作是默认构造函数。当声明一个栈对象时，默认构造函数初始化栈使之处于栈空的状态。因而，此函数将 top 置为 NULL。函数的定义如下所示：

```
template<class Type> //default constructor
linkedStackType<Type>::linkedStackType()
{
      top = NULL;
}
```

删除栈

在栈的链表表示中，函数 destroyStack 要比在线性表示中多做一些工作。在数组表示中，仅把 top 的值置为 0 就删除了栈。而在链表表示中，栈内元素的内存是动态分配的。所以，需要将 top 的值置为 NULL，并释放所有栈元素所占用的内存空间。

```
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 将栈重新初始化为空的状态。由于栈中可能已有一些元素而使用的又是链表表示，因而需要释放所有栈元素所占用的内存。这个任务可以通过调用成员函数 destroyStack 来完成。注意函数 destroyStack 也会把 top 的值置为 NULL。函数定义如下所示：

```
template<class Type>
void linkedStackType<Type>:: initializeStack()
{
    destroyStack();
}
```

空栈和满栈

操作 isEmptyStack 和 isFullStack 很简单。如果 top 是 NULL，则栈是空的。注意，由于栈元素空间的分配和释放是动态的，所以栈永远也不会满（只有当内存耗尽时栈才会满）。因此，函数 isFullStack 永远都返回 false。这两个函数定义如下所示：

```
template<class Type>
bool linkedStackType<Type>::isEmptyStack()
{
       return(top == NULL);
}

template<class Type>
bool linkedStackType<Type>:: isFullStack()
{
    return false;
}
```

接下来，考虑操作 push 和 pop。根据图 17.15 所示，可以清楚地看到 newElement 将会插入到由 top 指定的链表的前面（当压栈时）。在退栈时，由 top 指定的节点将会被删除。在这两种情况下，都要修改指针 top 的值。

压栈（push）

考虑如图 17.16 所示的栈。

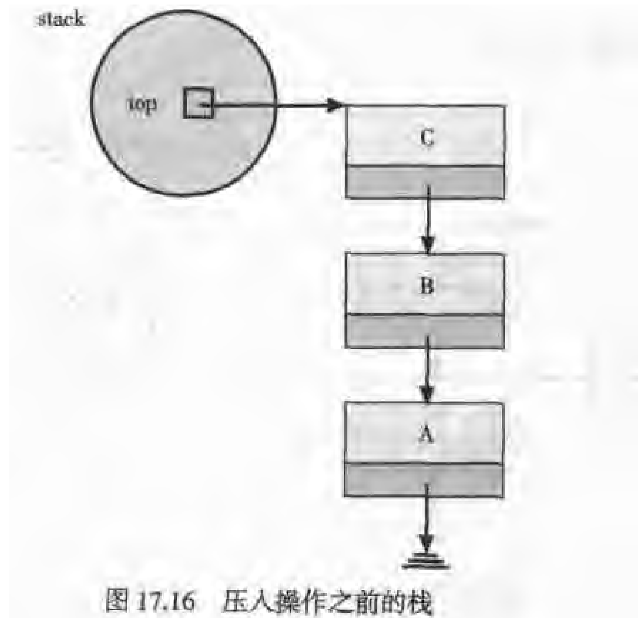


图 17.16 压入操作之前的栈

假设将压入栈的新元素是'D'。首先，给新节点分配内存空间。接下来，将'D'存入新节点，并将这一节点插入到链表的前面。最后，修改 top 的值。语句：

```
newNode = new nodeType<Type>;          //create the new node
newNode->info = newElement;
```

创建一个节点，将该节点的地址存储到变量 newNode 中，并将 newElement 的值存放到 newNode 的 info 域。这样，栈的结构就如图 17.17 所示。

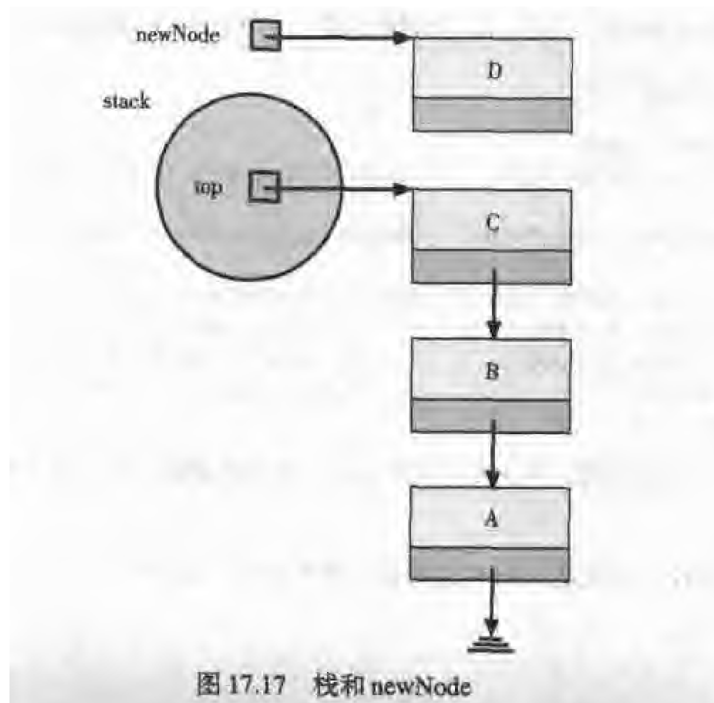


图 17.17 栈和 newNode

语句：

```
newNode->link = top;
```

将newNode插入到栈顶，如图17.18所示。

最后，语句：

```
top = newNode;
```

修改top的值，结果如图17.19所示。

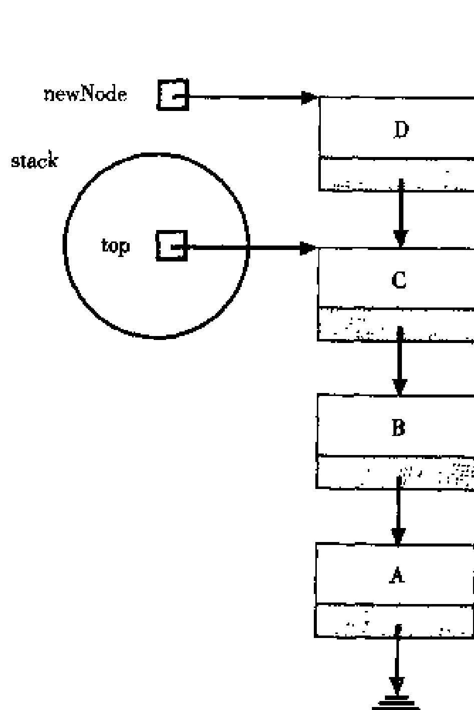


图17.18 语句newNode->link = top;执行后的栈

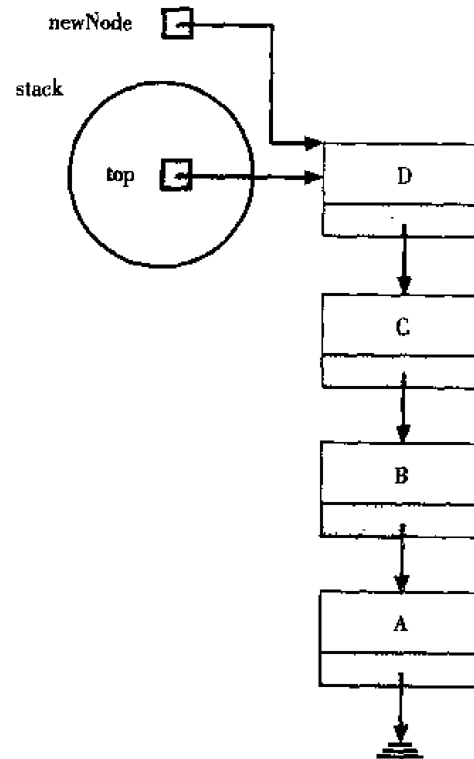


图17.19 语句top = newNode;执行后的栈

函数push的定义如下所示：

```
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
```

在将一个元素压入栈之前，不必检验栈是否满。因为从逻辑上讲，链表栈永远不会满。

退栈（pop）

现在考虑pop操作，也就是push的逆操作。考虑如图17.20所示的栈。

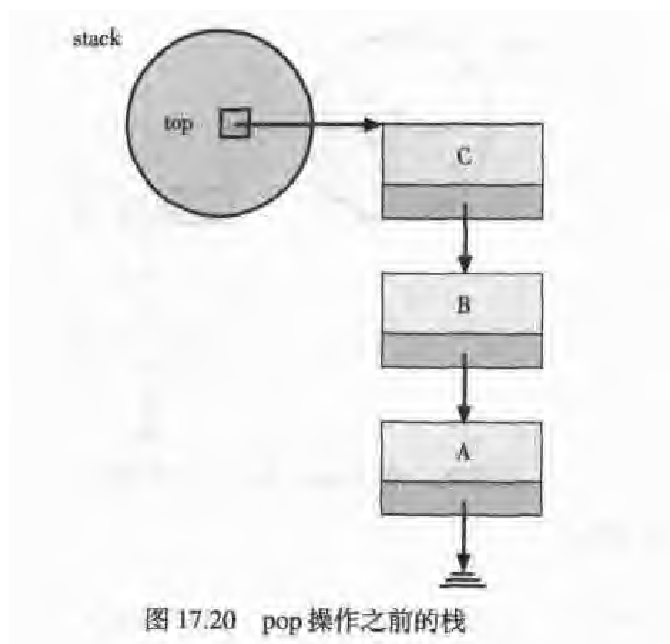


图 17.20 pop操作之前的栈

假定栈顶元素要存储在 poppedElement 中。则语句：

```
poppedElement = top->info;
```

将栈顶元素拷贝到 poppedElement 中。语句：

```
temp = top;
```

使指针 temp 指向栈顶元素，而语句：

```
top = top->link;
```

使得栈的下一个元素成为栈顶元素。此时的情形如图 17.21 所示。

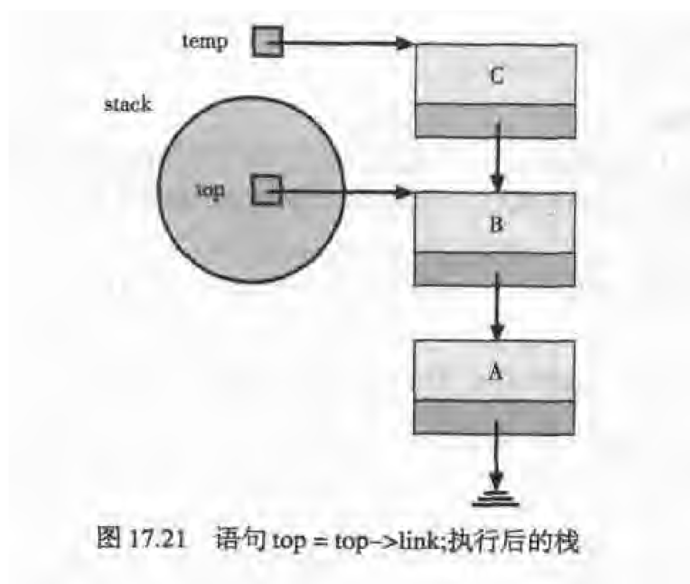


图 17.21 语句 top = top->link;执行后的栈

最后，语句：

```
delete temp;
```

释放指针 temp 所指的内存空间。图 17.22 显示了其结果。

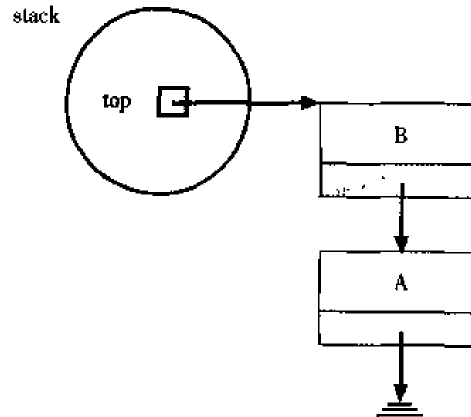


图 17.22 语句delete temp;执行后的栈

函数pop的定义如下所示：

```
template<class Type>
void linkedStackType<Type>::pop(Type& poppedElement)
{
   nodeType<Type> *temp;       //pointer to deallocate memory

   poppedElement = top->info; //copy the top element into
                              //poppedElement
   temp = top;                //set temp to point to the top node
   top = top->link;           //advance top to the next node
   delete temp;               //delete the top node
}//end pop
```

同栈的数组表示一样，在退出栈顶元素之前必须检查栈是否为空。因此，在调用pop之前要先调用isEmptyStack函数。这样，函数pop（假定stack是linkedStackType类型的对象）的调用形式是：

```
if(!stack.isEmptyStack())
    stack.pop(poppedElement);
```

构造函数和析构函数

我们已经讨论过了默认构造函数。要完成栈的操作，接下来要讨论拷贝构造函数和析构函数，以及赋值运算符的重载函数（这些函数同第16章中讨论过的链表的函数类似）。

```
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(=)

对于类linkedStackType而言，重载赋值运算符的函数定义如下所示：

```
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=
```

前面讨论的栈定义，以及实现栈操作的函数都是通用的。而且，如同栈的数组实现一样，应该将栈的定义连同实现栈操作的函数一起汇集到一个（头）文件中。这样，用户程序就可以使用include语句将此头文件包含进来。另外，在声明一个栈对象时，应该将栈元素数据类型作为一个参数传递给类型linkedStackType。例如，语句：

```
linkedStackType<int> stack;
```

将stack声明为linkedStackType类型的对象，并且栈中元素的数据类型是int。类似地，语句：

```
linkedStackType<newString> stringstack;
```

将stringstack声明为linkedStackType类型的对象，栈中元素的数据类型是newString。

为了检验链表栈的各种操作，我们在例17.3中编写了一个小程序。建议读者对该程序进行代码走查。

例17.3 假定类linkedStackType的定义以及实现栈操作的函数都包括在头文件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
```

从类 linkedListType 派生栈

如果将栈的push函数和第16章中讨论的通用链表的insertFirst函数相比较，会发现实现这些操作的算法是相似的。如果比较其他函数，如initializeStack和initializeList，isEmptyList和isEmptyStack等，就会发现类linkedStackType完全可以从类linkedListType派生而来。此外，函数pop可以使用上面的实现方法。

下面，定义从类 linkedListType 派生类 linkedStackType。同时也给出了相应实现栈操作的函数：

```
#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
```

17.5 栈应用：后缀表达式计算器

通常我们用于数学表达式的表示法（也就是我们在学校里学到的那种表示法）称为中缀表示法，这种表示法把运算符放在操作数中间。例如，表达式 $a+b$ 中运算符 + 就放在操作数 a 和 b 之间。在中缀表示法中，运算符是有优先级别的。也就是说，从左到右计算表达式，乘法和除法要比加法和减法有更高的优先级。如果想以不同的顺序计算表达式，就必须加上括号。例如，对于表达式 $a+b*c$，我们先对 b 和 c 进行 * 运算，然后再对 a 和 $b*c$ 的结果进行 + 运算。

在20世纪50年代，波兰数学家Lukasiewicz发现如果将运算符写在操作数之前（前缀表示或波兰表示，如 + a b），或者操作数之后（后缀表示或逆波兰表示，如 a b +），则可以不必指定运算符优先级。例如，表达式：

$a+b*c$

的后缀式表示是：

$a\ b\ c\ *\ +$

下例给出了中缀表达式和与之对应的后缀表达式。

例 17.4

中缀表达式	对应的后缀表达式
$a+b$	$a\ b+$
$a+b*c$	$a\ b\ c*+$
$a*b+c$	$a\ b*c+$
$(a+b)*c$	$a\ b+c*$
$(a-b)*(c+d)$	$a\ b-c\ d+*$
$(a+b)*(c-d/e)+f$	$a\ b+c\ d\ e/-*f+$

在Lukasiewicz的发现公布不久，计算机科学界就意识到后缀表示的一些重要应用。事实上，直到今天许多编译器也要先将数学表达式翻译成某种形式的后缀表达式，然后再将后缀表达式翻译成机器代码。可以使用如下算法计算后缀表达式：

按从左到右的顺序扫描表达式。当遇到某个运算符时，则退出该运算符所需个数的操作数，并计算该运算符的表达式的值，然后继续。

考虑下面的后缀表达式：

```
6 3 + 2 * =
```

下面利用栈和上述的算法计算该表达式的值。

1. 读取第一个符号6，它是一个数字。将这个数字压入栈中（如图17.23所示）。

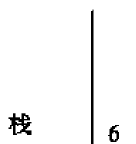


图 17.23 压入6后的栈

2. 读取下一个符号3，它是一个数字。将这个数字压入栈中（如图17.24所示）。

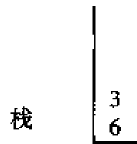


图17.24 压入3后的栈

3. 读取下一个符号+，这是一个运算符。由于这个运算符需要两个操作数，退栈两次。计算表达式，并将结果重新压栈（如图17.25所示）。

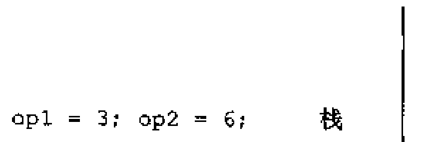


图17.25 退栈两次后的栈

执行运算：op2 + op1 = 6 + 3 = 9
将结果重新压栈（如图17.26所示）。

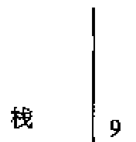


图17.26 压入op2 + op1结果（为9）后的栈

4. 读取下一个符号2，它是一个数字。将这个数字压入栈中（如图17.27所示）。

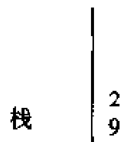


图17.27 压入2后的栈

5. 读取下一个符号*，这是一个运算符。由于这个运算符需要两个操作数，退栈两次。计算表达式，并将结果重新压栈（如图17.28和图17.29所示）。

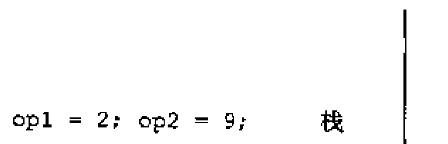


图17.28 退栈两次后的栈

执行运算：op2 * op1 = 9 * 2 = 18
将结果重新压栈（如图17.29所示）。

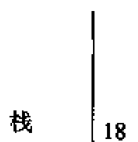


图17.29 压入op2 * op1结果（为18）后的栈

6. 读取下一个符号 =，这是一个等号，意味着表达式的结束。因此，输出结果。表达式的计算结果存储在栈中，所以弹出并输出栈顶元素，如图 17.30 所示。

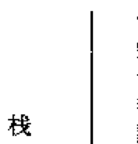


图 17.30　弹出最后元素后的栈

表达式的值为：6 3 + 2 * = 18

通过上面讨论可知，在读入一个非数字的符号时，会出现以下情形：

1. 读到的符号是以下几种之一：+，-，/，* 或 =。
   a. 如果符号是 +，-，* 或 /，即运算符，则应该进行运算。由于这些运算符都需要两个操作数，因而栈中至少应该有两个元素；否则，表达式有错误。
   b. 如果符号是 =（等号），表明表达式结束，并应该输出计算结果。这时，栈中应该有且仅有一个元素；否则，表达式有错误。
2. 读到的符号不是 +，-，/，* 或 = 之一。这种情况表明，表达式含有非法的运算符。

还有一点需要注意的是，由于运算符在操作数之后，所以在读到一个操作数（数字）时，需要先把它压入栈中。

考虑下表达式：

(i)　7 6 + 3 ; 6 - =

(ii)　14 + 2 3 * =

(iii) 14 2 3 + =

表达式 (i) 中含有非法运算符；表达式 (ii) 中的运算符 + 没有足够的操作数；而表达式 (iii)中含有过多的操作数。对于表达式(iii)，在读到等号（=）时，栈中将会有两个元素，而这个错误直到准备输出表达式的值时才会发现。

为了使输入更容易阅读，假定后缀式采用如下格式：

```
#6 #3 + #2 * =
```

表达式中每个数字之前都有一个 #。如果读到一个 #，则说明下一个输入符号是数字（即操作数）。如果读到的符号不是#，那么它或者是运算符（可能是非法的），或者是等号（说明表达式结束）。此外，假定表达式中只含有运算符 +，-，* 或 /。

本程序将输出完整的后缀表达式及其计算结果。如果表达式中含有错误，表达式将被丢弃。在这种情况下，程序将输出表达式本身及相应的错误信息。由于表达式中可能会含有错误，程序必须在处理下一个表达式之前清除栈内信息。就是说，应该重新初始化栈，即栈应该是空的。

主要算法

根据前面讨论，程序主要算法的伪代码如下所示：

```
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;
}
```

Evaluate函数 该函数用于计算表达式的值。一次运算需要两个操作数，而且操作数存放在栈中。因而，栈中必须包含两个以上的数字。如果栈中的数字不足两个，则表达式出错。这种情况下，程序丢弃整个表达式并输出相应的出错信息。该函数还将同时检查非法运算。函数的伪码如下所示：

```
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 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
  }
```

Discard 函数 在表达式中发现错误时调用该函数。此函数不断地输入，然后输出数据，直到遇到'1=1'（表达式的结束符号）为止。

```
while(ch != '=')
{
  read ch;
  output ch;
}
```

完整的程序代码清单

```
//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:Ch17_RpnData.txt");

   if(!infile)
   {
      cout<<"Cannot open input file. Program terminates!"<<endl;
      return 1;
   }

   outfile.open("a:Ch17_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 (!= '=')

       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(ifstream& in, ofstream& out, char& ch)
{
   while(ch != '=')
   {
      in.get(ch);
      out<<ch;
   }
}//end discard
```

程序运行结果

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
_________________________________
```

17.6 消除递归：逆向打印链表的非递归算法

在第16章，我们使用递归实现了逆向打印链表。在本节中，将学会使用栈来设计逆向打印链表的非递归算法。

考虑如图17.31所示的链表。

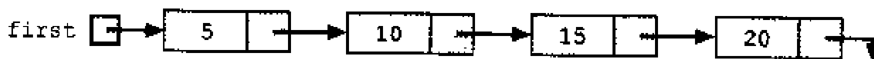


图 17.31 链表

要逆向打印链表，首先要访问链表的最后一个节点，这可以通过从链表首节点开始遍历链表来实现。然而，当访问到最后一个节点时，我们又如何访问它的前趋节点呢（尤其是当给定的链表是单向时）？当然可以使用相应的循环终止条件来再次遍历链表，但这种方法会极大地浪费运行时间，当链表规模较大时更是如此。另外，如果对每一个节点都进行这样的操作，程序运行起来会很慢。下面，将说明如何使用栈来有效地逆向打印链表。

当打印了某一节点的 info 域之后，要立即移动到该节点的前一节点。例如，在打印完20后，要移动到info为15的节点。因而，当第一次遍历整个链表时，应该记录下每一个节点的指针。例如，对于图17.31所示的链表，应该记录下info为5，10，15的节点指针。当打印完20后，应该回退到info为15

的节点；打印完15后，应该回退到info为10的节点，以此类推。因此，应该遵循后进先出规律，逐次将每个节点压入栈中。

由于通常链表中节点的个数是未知的，所以应该采用栈的链表存储方式。假定stack是类linkedListType的一个对象，current是与first同类型变量的指针。考虑如下语句：

```
current = first;                    //Line 1

while(current != NULL)              //Line 2
{
      stack.push(current);          //Line 3
      current = current->link;      //Line 4
}
```

当执行完第1行的语句后，current指向第1个节点（如图17.32所示）。

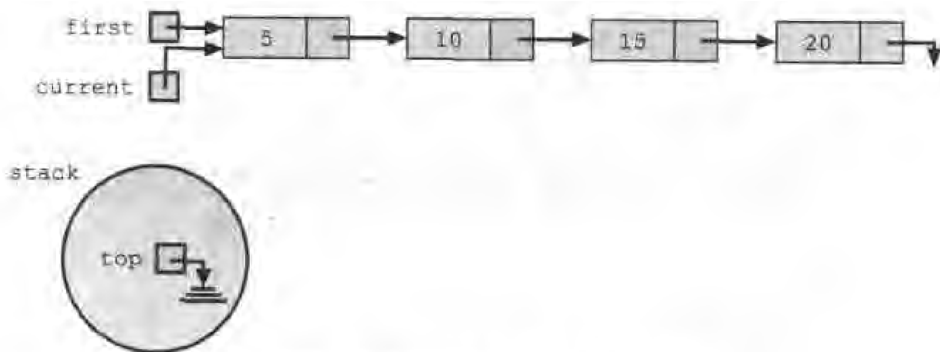


图17.32 语句current = first;执行后的链表

由于current非NULL，所以第3行和第4行语句得以执行（如图17.33所示）。

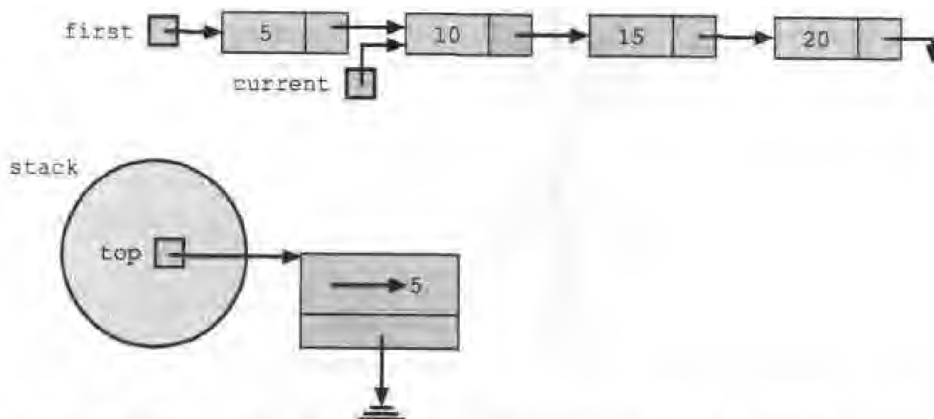


图17.33 语句stack.push(current);和current = current->link;执行后的栈和链表

当执行完第4行语句后，重新计算第2行的循环条件语句。由于current非NULL，循环条件的值是true，所以继续执行第3行语句和第4行语句（如图17.34所示）。

当执行完第4行语句后，重新计算第2行的循环条件语句。由于current仍非NULL，循环条件的值是true，所以继续执行第3行语句和第4行语句（如图17.35所示）。

当执行完第4行语句后，重新计算第2行的循环条件语句。由于current仍非NULL，循环条件的值是true，所以继续执行第3行语句和第4行语句（如图17.36所示）。

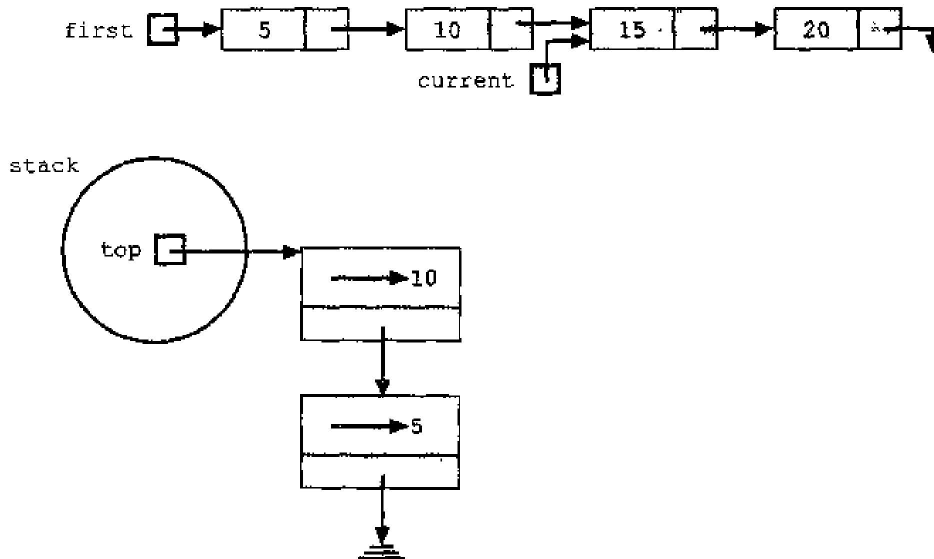


图 17.34 语句“stack.push(current);”和“current = current->link;”执行后的栈和链表

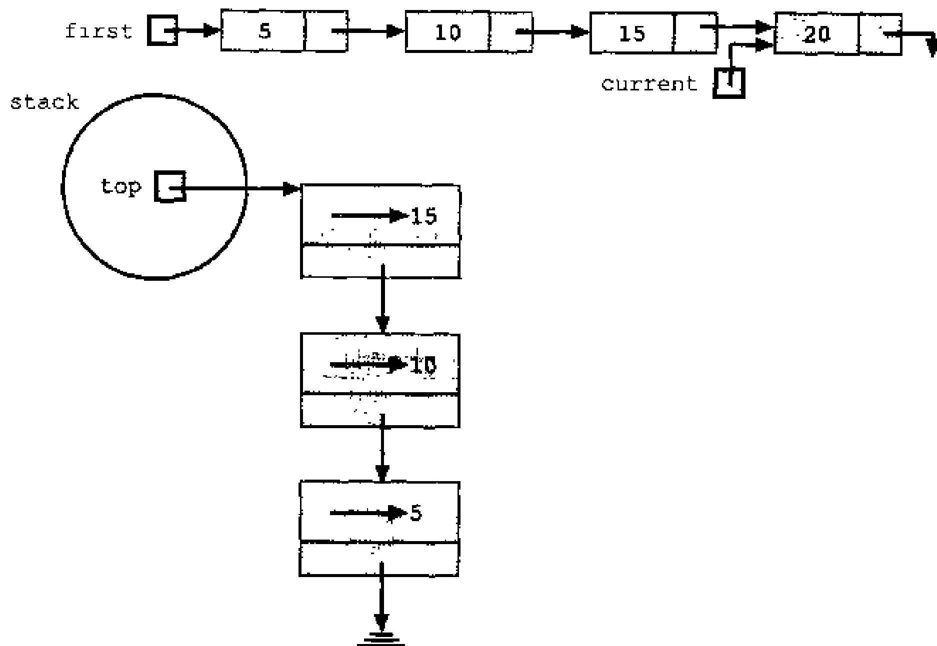


图 17.35 语句“stack.push(current);”和“current = current->link;”执行后的栈和链表

当执行完第4行语句后，重新计算第2行的循环条件语句。由于此时current为NULL，循环条件的值是false，while循环终止。根据图17.36，指向链表每一个节点的指针都被保存在栈中。栈顶元素存储的是指向链表中最后一个节点的指针，以此类推。下面，执行下列语句：

```
while(!stack.isEmptyStack())          //Line5
{
        stack.pop(current);           //Line6
        cout<<current->info<<" ";     //Line7
}
```

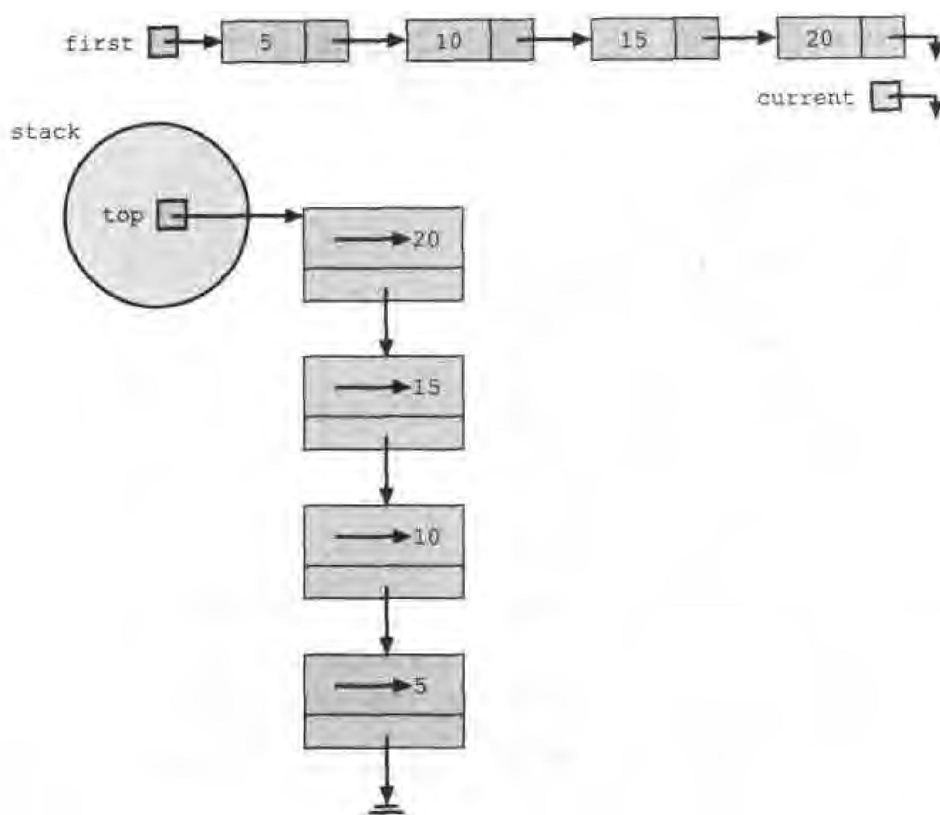


图 17.36　语句 "stack.push(current);" 和 "current = current->link;" 执行后的栈和链表

因为栈非空，所以在第 5 行的循环测试条件为 true。因此，执行第 6 行和第 7 行语句。在第 6 行语句执行后，current 指向链表的最后一个节点（如图 17.37 所示）。

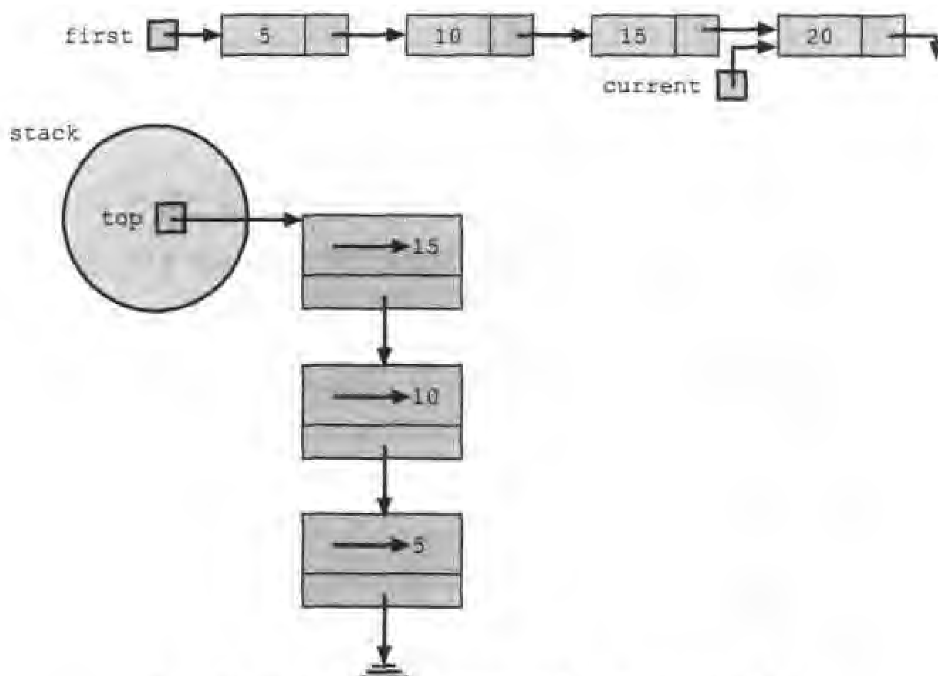


图 17.37　语句 "stack.pop(current);" 执行后的栈和链表

第 7 行语句输出 current->info，即 20。接下来，检查第 5 行的循环条件。由于循环条件的值为 true，执行第 6 行和第 7 行语句。在第 6 行语句执行后，结果如图 17.38 所示。

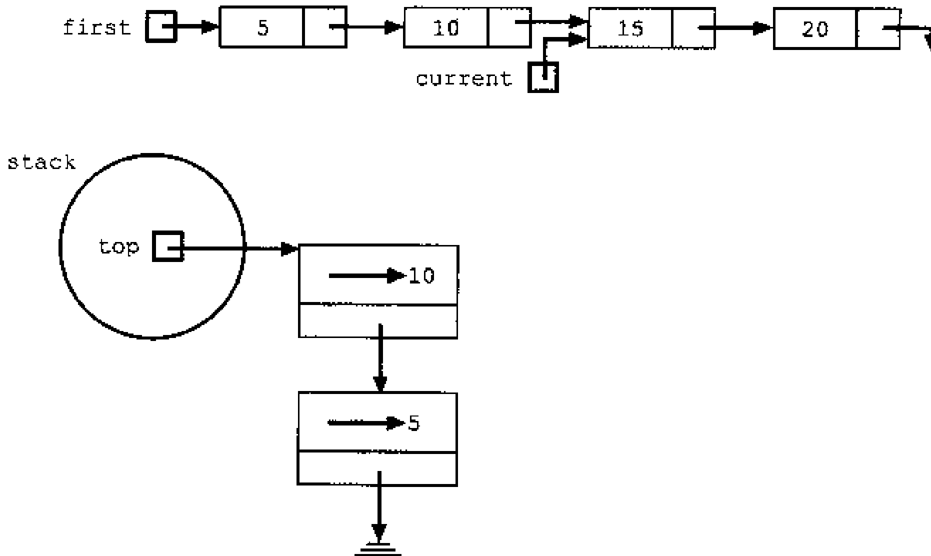


图 17.38　语句 "stack.pop(current);" 执行后的栈和链表

第7行语句输出 current->info，即15。接下来，继续检查第5行的循环条件。由于循环条件的值为true，执行第6行和第7行语句。在第6行语句执行后，结果如图17.39所示。

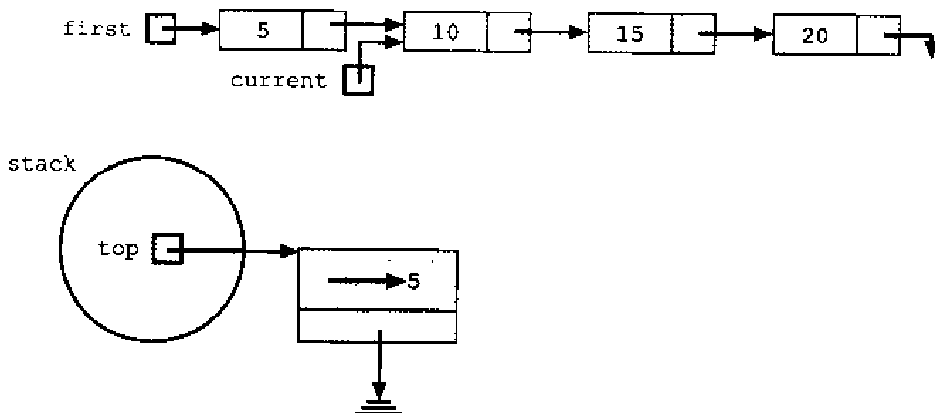


图 17.39　语句 "stack.pop(current);" 执行后的栈和链表

第7行语句输出 current->info，即10。接下来，继续检查第5行的循环条件。由于循环条件的值为true，执行第6行和第7行语句。在第6行语句执行后，结果如图17.40所示。

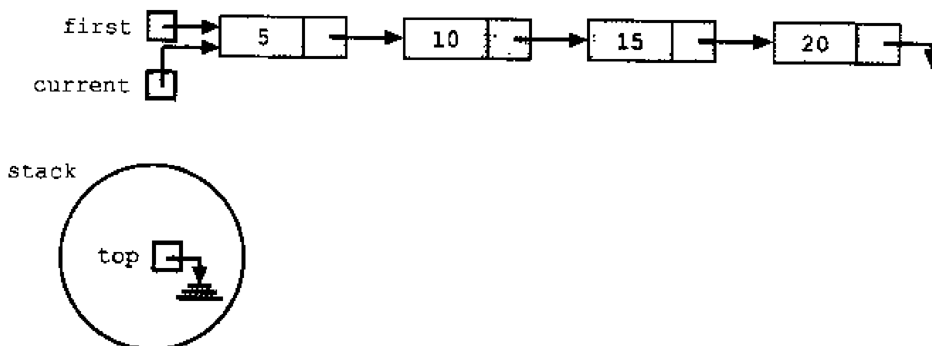


图 17.40　语句 "stack.pop(current);" 执行后的栈和链表

第7行的语句输出current->info，即5。接下来，继续检查第5行的循环条件。由于循环条件的值为false，while循环终止。第5行的while循环产生了如下输出：

```
20 15 10 5
```

17.7 队列

本节将讨论另一个重要的数据结构——队列(Queue)。在计算机科学中的队列同日常生活中的队列是相同的。我们经常可以在银行或者杂货铺中看到排队的顾客，在收费站前看到排队的汽车。与此相似，由于计算机发送打印请求的速度要比打印机的打印速度快，所以经常会有一个文件队列等待打印服务。通常，处理队列中元素的规则是：队首的用户首先得到服务，当有新的用户到来时，他将排在队尾。也就是说，队列是一种先进先出的数据结构。

队列在计算机科学中有极为广泛的应用。如果系统模型满足先进先出规律，就可以使用队列。在本节的最后，我们将讨论队列最广泛的应用之一，计算机模拟。当然，首先还是要开发实现队列所必需的工具。接下来的几节将讨论如何设计类来实现作为抽象数据类型（ADT）的队列。

队列是一个由同类型元素构成的集合。在这种集合中，在一端（称为队尾——back或rear）插入元素，在另一端（称为队首——front）删除元素。例如，在银行中排队的客户在等待存/取款或进行其他交易时，每个新来的客户都要从队尾加入到队列中。当一个出纳员准备好下一次服务时，则排在队首的客户获得服务。

当向队列中加入一个新元素时只能访问队尾，而当从队列中删除一个元素时只能访问队首。与栈一样，不能访问队列中间的元素，即便队列是以数组的形式存放的。

队列（queue） 队列是一种在一端（称为队尾——back或rear）插入元素，在另一端（称为队首——front）删除元素的数据结构；是一种先进先出（First In First Out，FIFO）的数据结构。

17.7.1 队列操作

在队列的定义中可以看到，队列的两种关键的操作是插入和删除。我们称插入操作为addQueue；称删除操作为deQueue。由于既不能从空队列中删除元素，也不能向满队列中插入元素，因而还需要另外两个队列操作才能实现addQueue和deQueue操作：isEmptyQueue（检查队列是否为空）和isFullqueue（检查队列是否为满）。

此外，还需要另一种操作initializeQueue，将队列初始化为空。与栈相似，还需要destroyQueue操作删除整个队列，使其为空。因而，队列的主要操作如下所示：

1. initializeQueue：此操作将队列初始化为空。
2. destroyQueue：此操作删除队列中所有元素，使其变为空。
3. isEmptyQueue：此操作检查是否队列为空。如果队列为空，返回true；否则，返回false。
4. isFullQueue：此操作检查是否队列为满。如果队列为满，返回true；否则，返回false。
5. addQueue：此操作向队尾插入一个新元素。此操作的输入是队列和新加入的元素。此操作的先决条件是，队列必须存在且非满。
6. deQueue：此操作从队列最前边删除一个元素并将它存放在由deqElement指定的内存单元中。此操作的输入是队列和队首元素将被存放的地址。此操作的先决条件是，队列必须存在且非空。

与栈一样，既可以使用数组结构存储队列，也可以使用链表结构存储队列。我们将分别论述这两种存储方式。由于元素的删除和插入要分别在队列的两端进行，所以需要两个指针分别记录队首和队尾，称之为front和rear。

17.7.2 使用数组实现队列

在给出作为抽象数据类型（ADT）的队列的类定义之前，首先要弄清实现队列需要多少个类成员。很显然，需要一个数组来存放队列元素，变量 front 和 rear 来记录队首和队尾，变量 maxQueueSize 来指定队列中可存储的最大元素个数。因而，实现队列至少需要 4 个数据成员。

在设计实现队列操作的算法之前，必须弄清如何使用 front 和 rear 访问队列元素。front 和 rear 如何表示队列空或队列满？假定 front 指定了队列中第一个元素的下标，而 rear 指定了队尾元素的下标。当向队列插入一个元素时，首先将 rear 移动到数组的下一位置，然后将新元素插入到 rear 所指的位置上。从队列中删除一个元素，首先取出 front 所指的元素，然后将 front 向数组的下一位置移动。总之，每个 deQueue 操作后，front 将改变；而每次 addQueue 操作后，rear 将改变。

再来看 deQueue 操作后 front 将如何改变，以及 addQueue 操作后 rear 将如何改变。假定用于存放队列元素的数组的大小是 100。

初始状态下，队列为空的。在操作：

```
addQueue(Queue, 'A');
```

之后，数组如图 17.41 所示。

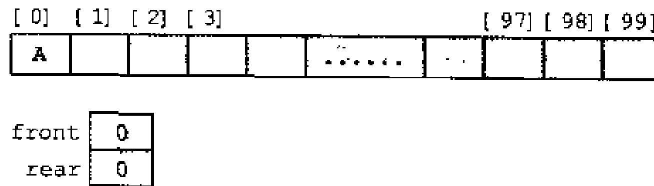


图 17.41 第一次 addQueue 操作之后的队列

在另两个 addQueue 操作：

```
addQueue(Queue, 'B');
addQueue(Queue, 'C');
```

之后，数组如图 17.42 所示。

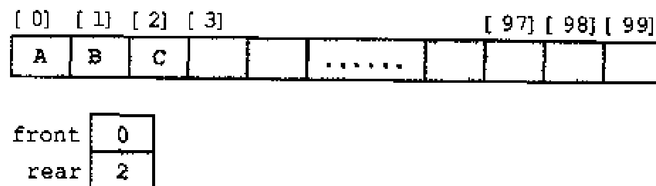


图 17.42 另外两次 addQueue 操作之后的队列

现在讨论 deQueue 操作：

```
deQueue(Queue,deqElement);
```

在此操作之后，数组队列如图 17.43 所示。

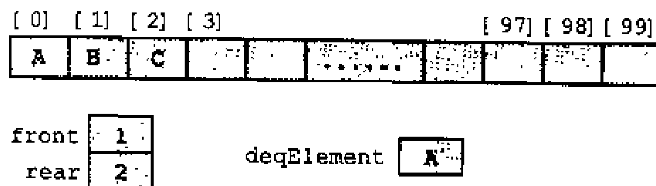


图 17.43 deQueue 操作之后的队列

上面这种设计的队列能正常工作吗？假设 A 代表将一个元素插入（即 addQueue）到队列中，D 代表从队列中删除（即 deQueue）一个元素。考虑如下操作序列：

```
AAADADADADADADADA...
```

按照这个操作序列，rear 最终将会指向数组的最后一个位置，让人误以为队列已满。然而，实际上这时队列中只有三个元素，数组的前部是空闲的（如图 17.44 所示）。

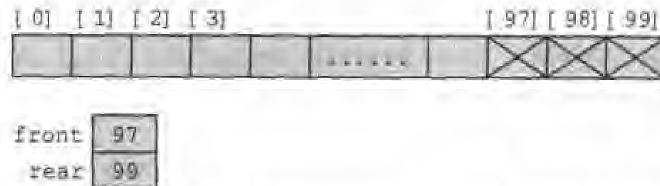


图 17.44　经过一系列的 AAADADADADADA…操作之后的队列

解决这种问题的一种方案是，当队尾溢出（即 rear 指向数组的最后位置）时，检查下标为 front 的数组元素的值。如果通过 front 的值表明数组前边还有空间，当溢出发生时，将队列中所有元素向数组的第一个位置移动。如果数组规模很小，这种方案还是可行的。但是，当数组很大时，程序运行起来会很慢。

另一种解决方案是：假定数组是环状的——即数组起始位置同数组结束位置相接（如图 17.45 所示）。

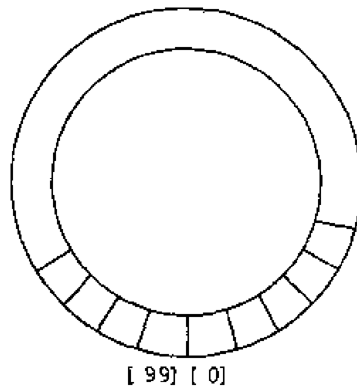


图 17.45　循环队列

尽管数组示意图的画法还与前面相同，但是我们将其视为环状来考虑。

假定如图 17.46 所示的队列。

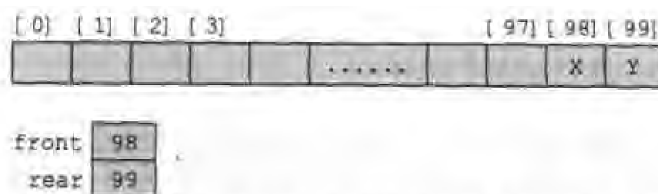
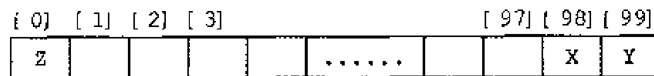


图 17.46　只包含两个（在第 98 和第 99 位置上）元素的队列

在操作：

```
addQueue(Queue,'z');
```

之后，队列如图 17.47 所示。



front 98
rear 0

图 17.47 经过一次 addQueue 操作之后的队列

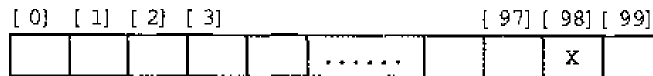
由于存储队列的数组是环状的，可以使用以下语句将指针 rear（front）移动到下一个数组位置：

```
rear = (rear + 1) % maxQueue;
```

如果 rear < maxQueue−1，则 rear + 1 <= maxQueue−1，这样(rear+1) % maxQueue = rear+1；如果 rear == maxQueue−1（即，rear 指向数组的最后位置），则 rear+1 == maxQueue，这时(rear+1) % maxQueue == 0。在这种情况下，rear 会被置为 0，也就是数组的第一个位置。

这种设计看起来能够工作得很好。在编写实现队列操作的算法之前，考虑下面两种情况。

情况 1 假定经过了某些操作后，存储队列的数组如图 17.48 所示。



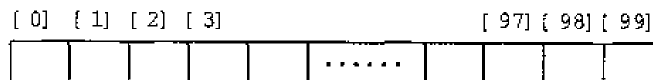
front 98
rear 98

图 17.48 只有一个元素的队列

在操作：

```
deQueue(Queue,deqElement);
```

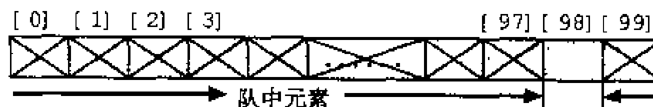
之后，数组如图 17.49 所示。



front 99
rear 98

图 17.49 在 deQueue 操作之后的队列

情况 2 下面，考查如图 17.50 所示的队列。



front 99
rear 97

图 17.50 包含 99 个元素的队列

在操作：

```
addQueue(Queue, 'Z');
```

之后，数组如图17.51所示。

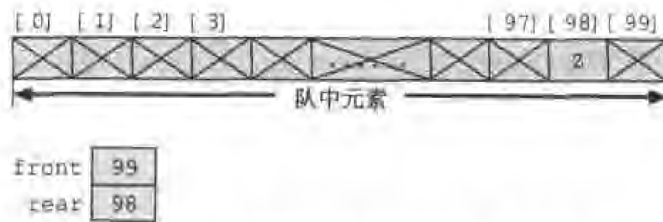


图17.51 在addQueue操作之后，队列满

图17.49和图17.51所示的队列中front和rear的值相同。然而，图17.49所示的数组表示的是一个空队列，而图17.51所示的数组表示的则是一个满队列。这种新的队列设计方案引出了一个新的问题——如何区分空队列和满队列。

这个问题有多种解决方法。其中之一是保留一个计数器。实现队列时，除了指针rear和front外，还需要另一个变量count。当插入元素时，count的值增量；而删除元素时，count的值减量。这种情况下，函数initializeQueue和destroyQueue将变量count初始化为0。当用户需要知道队列中元素的个数时，这种方法十分有效。

另一种方法是让front指向队列首元素数组下标的前驱位置，而不是指向首元素本身的下标。在这种情况下，假定rear仍然指向队列末尾元素数组下标，当front == rear时，表明队列是空的。使用这种方法，由front指向的内存单元（即真正队首元素的前一个内存单元）是保留的。如果下一个可用空间就是由front指向的特别保留的内存单元，队列就满了。最后，由于front所指的数组元素始终是空闲的，如果数组大小是100，则队列中最多可存储99个元素（如图17.52所示）。

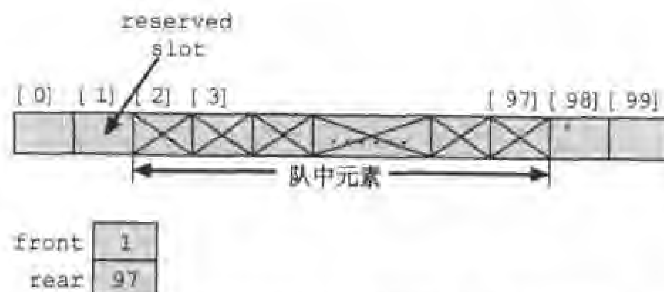


图17.52 用保留内存单元存储队列元素的数组

现在，我们使用第一种方法实现队列。即，使用变量count指示队列是空或满。

下面的类将队列作为抽象数据类型（ADT）定义。由于可以动态地分配数组，所以可以由用户指定数组大小。数组的默认大小是100。

这里只给出队列的定义，而相应的注释工作作为练习留给读者。队列类操作的注释与栈操作的注释相似。

```
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
    };
```

下面，考虑队列操作的实现。

初始化队列

首先考虑的操作是initializeQueue。此操作将队列初始化为空。第一个元素会加入到数组的第一个位置上。因而，将front初始化为0，rear初始化为maxQueueSize－1，count初始化为0，如图17.53所示。

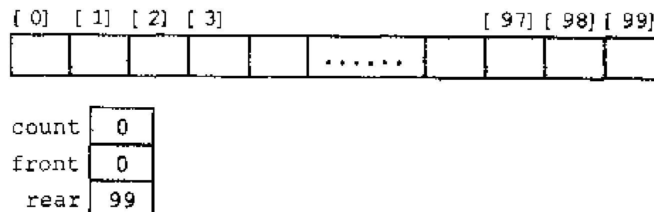


图17.53 空队列

```
template<class Type>
void queueType<Type>::initializeQueue()
{
       front = 0;
       rear = maxQueueSize - 1;
       count = 0;
}
```

删除队列

destroyQueue操作将队列中所有元素删除，使其处于空状态。由于队列存储于数组中，所以可以简单地通过将front、rear、count重新设置来删除队中的所有元素。在这种实现方式中，destroyQueue操作等同于initializeQueue操作。

```
template<class Type>
void queueType<Type>::destroyQueue()
{
       front = 0;
       rear = maxQueueSize - 1;
       count = 0;
}
```

空队列和满队列

如前所述，如果count == 0则队列是空的；如果count == maxQueueSize，则队列是满的。实现这两种操作的函数如下所示：

```
template<class Type>
int queueType<Type>::isEmptyQueue()
{
   return(count == 0);
}

template<class Type>
int queueType<Type>::isFullQueue()
{
   return(count == maxQueueSize);
}
```

addQueue 函数

下面，我们实现addQueue操作。由于rear指向队列的最后一个元素，要向队列中插入一个新元素，应该首先将rear指针向前移动一位，然后将要加入的元素存入到rear所指的位置上。此外，还要将count加1。所以，函数addQueue如下所示：

```
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的内部没有在插入元素前对队列是否已满进行判断，所以在调用函数addQueue之前必须调用isFullQueue来检查队列是否已满。假定queue是queueType类型的对象，则函数addQueue的调用形式如下所示：

```
if (!queue.isFullQueue())
    queue.addQueue(newElement);
```

deQueue 函数

要实现deQueue操作，就要访问front。由于front指向存储队首元素的数组位置，要删除队首元素必须访问front指向的元素，将count减1，并将front移动到队列下一个元素位置。这样函数deQueue如下所示：

```
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
}
```

deQueue函数内部并没有在删除元素前对队列是否为空进行检查，所以在调用函数deQueue之前必须调用isEmptyQueue来检查队列是否已空。函数deQueue的调用形式如下所示：

```
if (!queue.isEmptyQueue())
    queue.deQueue(deqElement);
```

构造函数和析构函数

要实现完整的队列操作，还要考虑构造函数和析构函数的实现。构造函数将用户指定的大小存储到变量maxQueueSize中，然后创建大小为maxQueueSize的动态数组。如果用户没有指定队列大小，构造函数将使用默认值，创建一个大小是100的动态数组。构造函数同时初始化变量front和rear指示队列为空。构造函数的定义如下所示：

```
    //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
}
```

由于存储队列元素的数组是动态创建的，因而当一个队列对象超出其作用域时，析构函数只是简单地将这个数组所占的内存释放掉。析构函数的定义如下所示：

```
template<class Type>
queueType<Type>::~queueType()   //destructor
{
   delete [] list;
}
```

拷贝构造函数和重载赋值运算符的实现作为练习留给读者（这些函数的定义类似于链表和栈）。

17.7.3 用链表实现队列

由于存储队列元素的数组大小是固定的，数组中只能够存储有限的队列元素。而且，用数组实现队列要求对数组，特别是front和rear所指定的元素加以特殊考虑。用链表实现队列简化了数组实现中的许多特殊情况。而且，由于存储队列元素的内存是动态分配的，所以队列永远不会满。本节将讨论队列的链表实现。

由于元素插入在一端（rear），元素删除在另一端（front），因此需要知道队尾和队首元素的位置。所以，需要两个指针front和rear，来维护队列。下面类定义链表队列作为抽象数据类型（ADT）。

在此，只给出了队列的定义，操作增加注释作为练习留给读者。

```
//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
};
```

实现队列操作的函数的定义相当简洁。如果front是NULL，则队列是空的。由于存储队列元素的内存是动态分配的，所以队列永远不会满，于是实现isFullQueue操作的函数总返回false（只有在内存耗尽时队列才会满）。

destroyQueue操作删除队列中所有元素，将队列置为空状态。由于存储队列元素的内存是动态分配的，所以此操作从第一个节点开始遍历存储队列的链表，逐一释放每个队列元素所占用的内存空间。

下面给出实现操作 isEmptyQueue，isFullQueue，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
}
```

initializeQueue操作将队列初始化为空的状态。如果队列中没有元素，则队列是空的。与栈相似，initializeQueue操作将队列重新初始化为空状态。注意，当声明队列对象时，构造函数初始化了队列。所以，此操作必须从队列中删除所有的元素。这项工作可以通过调用函数destroyQueue来实现。

```
template<class Type>
void linkedQueueType<Type>::initializeQueue()
{
   destroyQueue();
}
```

addQueue和deQueue操作

addQueue操作在队尾插入一个新元素。实现这个操作需要访问指针rear。类似地，deQueue操作删除队首元素，需要访问指针front。实现这些操作的函数如下所示：

```
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没有检查队列是否为空。因而，在调用deQueue之前，必须调用函数isEmptyQueue检查队列是否为空。函数deQueue的调用形式如下所示：

```
if (!queue.isEmptyQueue())
    queue.deQueue(deqElement);
```

默认构造函数的定义类似于initializeQueue函数。当队列对象超出其作用域时，析构函数将删除队列。即，释放队列元素所占据的内存空间。析构函数的定义类似于函数destroyQueue。另外，拷贝构造函数和赋值运算符的重载函数类似于栈中对应的函数。这些操作的实现作为练习留给读者。

17.7.4 从类linkedListType派生队列

从实现队列操作的函数定义中可以清楚地看到，队列的链表实现非常类似于前面以正向方式创建的链表（参见第16章）。addQueue操作类似于insertFirst；initializeQueue操作类似于initializeList；isEmptyQueue操作类似于isEmptyList；destroyQueue操作类似于destroyList。deQueue操作可按上面的方法来实现。指针front等同于first；指针rear等同于last。这种相似性表明，可以通过从linkedListType（见第16章）派生类来实现队列。

下面，从类linkedListType派生类linkedQueueType，并使用已定义的链表操作来实现队列操作：

```
//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 17.5 下面的程序测试队列的各种操作。该程序使用从类 linkedListType 派生的链表队列。

```
//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;
}
```

程序运行结果 *在本程序运行中用户的输入加有阴影。*

```
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
```

17.8 队列应用：模拟

一个系统模仿另一个系统行为的技术称为模拟。例如，使用包括风洞这样的物理模拟器来检测汽车车体的设计；使用飞行模拟器来训练飞行员。特别是当使用真实系统来测试造价过高或危险性过大时往往使用模拟技术。同样可以设计计算机模型来了解真实系统的行为（这里将简要介绍某些被计算机模型化的真实系统）。使用计算机模拟高造价或高危险性试验的行为通常比使用真实系统花费低廉，这是一种不必冒着生命危险就可获得必要信息的好办法。此外，对于难以构建数学模型的复杂系统，计算机模拟就特别有用。对于此类系统，计算机模型可以获得无限的精度。我们来看一下这类问题。

某地电影院的一个主管正在听取客户关于购票排队等待时间太久的投诉。该电影院目前只有一个售票口。附近一家新电影院就要开张了，这位主管担心会流失观众。他想雇佣更多的售票员，这样观众购票就不必等太久；但他又不想雇佣过多的售票员，以避免在淡季有人会无事可做而浪费时间和金钱。有件事情是他非常关心的，即观众平均等待时间。该主管希望有人来写一个程序来模拟电影院（排队的）行为。

在计算机模拟中，被研究的对象通常表示为数据。对于电影院问题，对象就是观众和售票员。售票员为观众提供服务，我们想要确定的是观众平均等待时间。对象行为通过编写算法进行实现；而算法在编程语言中又通过使用函数得以实现。因而，函数是用来实现对象行为的。在C++中，我们可以使用类将数据和操作封装在一个单元中。因而，对象可以表示成类。类的数据成员描述了对象的属性；成员函数描述了数据上的动作。如果改变了数据的值或者函数的定义（也就是修改了实现动作的算法），模拟结果就可能会发生改变。计算机模拟的主要任务就是产生现有系统的性能分析结果或者对指定系统的性能提出预测。

在电影院问题中，当售票员服务于某个观众时，其他观众必须等待。由于是以先来先服务原则对观众提供服务，而队列又是实现先进先出系统的有效方法，因而队列是计算机模拟技术中的重要数据结构。本节将讨论使用队列作为基本数据结构的计算机模拟技术。这种队列对象等待服务者服务的模拟系统，称为排队系统（Queuing Systems）。换句话说，排队系统由服务者（器）和等待服务的队列对象构成。我们在日常生活中经常会碰到各种各样的排队系统。例如，食品店系统和银行系统都是排队系统。另外，当向一个由多人共享的网络打印机发送一个打印请求时，该打印请求就被加入到队列中。在它之前的打印请求通常在该请求满足之前完成。因而，当有一个等待打印的文档队列时，打印机就成为一个服务者。

下面，我们描述一个可用于多种不同应用（例如，银行、食品店、打印机或者是很多人使用同一组处理器来执行其程序的主机环境）的排队系统。为了便于描述排队系统，我们使用术语服务者（Server）表示提供服务的对象。例如，在银行，出纳员是服务者；在食品店或电影院，售货员或售票员是服务者。我们将接受服务的对象称为客户（Customer），而服务的时间（即为一个客户服务所用的时间）称为一个事务处理时间（Transaction Time）。

因为排队系统由服务者和等待服务的对象队列组成，我们将模仿一个由多个服务者和等待服务的客户队列构成的系统。位于队首的客户等待下一个可用的服务者。当某个服务者空闲下来，队首客户立即到该服务者处接受服务。

当第一个客户到来时，所有的服务者是空闲的，于是该客户到第一个服务者前接受服务。当下一个客户到来时，如果还有服务者可用，则客户立即到该服务者前接受服务；否则，客户排队等候。要模仿一个排队系统，必须知道服务者的个数、客户期望到来时间、相邻客户到来时间间隔以及影响系统的事件个数。

现在，再来看电影院系统。假定服务者个数是1，平均每6分钟服务一个客户，平均每4分钟到来一个客户。系统的性能依赖于可用服务者的个数、服务于每个客户的时间，以及客户到来的频繁程度。如果对客户服务时间很长或客户到来很频繁，就需要更多的服务者。此类系统可归类为时间驱动模拟。在时间驱动模拟（Time-driven Simulation）模型中，可以通过一个计数器来实现时钟。也就是说，可以将计数器加1表示过了1分钟。这种模拟运行固定的时间。如果模拟运行100分钟，就让计数器从1增加到100（可以通过循环来实现）。

在本节讨论的模拟中，我们希望能够知道每个客户的平均等待时间。要计算这个平均值，需要将所有的客户等待时间加起来，然后再用这个总额除以到来的客户总数。当一个客户到来时，他排到队尾，则该客户的等待时间开始计时。如果队列是空且有空闲的服务者，客户立即接受服务，于是该客户的等待时间是0。否则，当客户到来时队列非空或者所有的服务者都忙着，客户就必须等待下一个可用的服务者，因而该客户的等待时间开始计时。可以通过给每个客户安装一个计时器来追踪客户等待时间。当客户到来时，将它的计时器置为0。此后，每个时钟单元将其加1。

假定服务者平均5分钟服务一个客户。当一个服务者变为空闲，并且客户等待队列非空时，位于队首的客户就可以上前接受服务。这样，必须追踪客户使用服务者的时间。当客户到达某服务者前时，事务处理时间被置为5。此后，每个时钟单元将其减1。当事务处理时间变为0时，服务者就标记为空闲。因而，要实现此排队系统的时间驱动计算机模拟，需要两种对象——服务者和客户。

下面，在设计实现模拟的主算法之前，先来设计实现两种对象（客户和服务者）的类。

17.8.1 客户

客户的属性包括：客户号码、到达时间、等待时间、事务处理时间，以及离开时间。如果知道到达时间、等待时间和事务处理时间，就可以将到达时间、等待时间、事务处理时间相加得到离开时间。实现客户的类为customerType。该类包含4个数据成员：customerNumber，arrivalTime，waitingTime和transactionTime，都是int数据类型。类型customerType的对象必须要实现的基本操作是：设置客户号码、到达时间、等待时间；按一个时钟单元增加等待时间；返回交易时间；返回客户号码。下面类customerType将客户作为抽象数据类型（ADT）实现（如图17.54所示）。

```
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);
        //The waiting time is set according to the parameter
        //Post: waitingTime = time
    void incrementWaitingTime();
        //The value of the data member waitingTime is
        //incremented by one time unit
        //Post: waitingTime++
    int getArrivalTime();
        //Return the value of the data member arrivalTime
    int getTransactionTime();
        //Return the value of the data member transactionTime
    int getCustomerNumber();
        //Return the value of the data member customerNumber
private:
    int customerNumber;
    int arrivalTime;
    int waitingTime;
    int transactionTime;
};
```

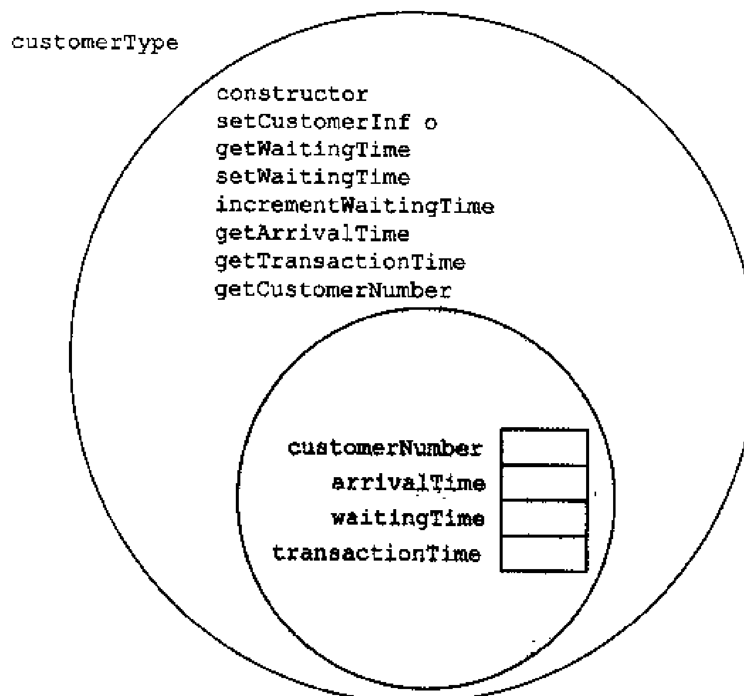


图17.54 类customerType

下面，给出类customerType的成员函数的定义。

构造函数使用参数来初始化变量customerNumber，arrivalTime，waitingTime和transactionTime。构造函数的定义如下所示：

```
customerType::customerType(int cN, int arrvTime, int wTime,
                           int tTime)
{
      customerNumber = cN;
      arrivalTime = arrvTime;
      waitingTime = wTime;
      transactionTime = tTime;
}
```

函数 setCustomerInfo 的定义类似于上面讨论过的构造函数。其定义如下所示：

```
void customerType::setCustomerInfo(int customerN, int inTime,
                                   int wTime, int tTime)
{
      customerNumber = customerN;
      arrivalTime = inTime;
      waitingTime = wTime;
      transactionTime = tTime;
}
```

函数 getWaitingTime 返回当前等待时间。其定义如下所示：

```
int customerType::getWaitingTime() const
{
    return waitingTime;
}
```

函数 incrementWaitingTime 增加 waitingTime 的值。其定义如下所示：

```
void customerType::incrementWaitingTime()
{
    waitingTime++;
}
```

函数 setWaitingTime，getArrivalTime，以及 getTransactionTime 的定义类似于函数 incrementWaitingTime 的定义：

```
void customerType::setWaitingTime(int time)
{
    waitingTime = time;
}


int customerType::getArrivalTime()
{
    return arrivalTime;
}

int customerType::getTransactionTime()
{
    return transactionTime;
}
```

函数 getCustomerNumber 返回客户号码，也就是返回数据成员 customerNumber 的值。

```
int customerType::getCustomerNumber()
{
```

```
        return customerNumber;
    }
```

17.8.2 服务者（器）

在某个给定的时间单元内，服务者要么空闲状态，要么忙于服务某一客户。我们使用一个字符串变量来标记服务者的状态。每个服务者有一个计时器，由于程序可能会需要知道哪个服务者正在向哪个客户提供服务，服务者还要存储它正服务的客户的信息。因而，有3个数据成员与服务者相关：status，transactionTime，currentCustomer。服务者必须实现的基本操作如下：检查服务者是否空闲；将服务者设置为空闲；将服务者设置为忙；设置事务处理时间（就是服务者给一个客户提供服务所用的时间）；返回事务处理剩余时间（用以确定是否要将服务者置为空闲）；如果每一时间单元后服务者是忙的，每个时间单元递减一次事务处理时间。下面的类serverType将服务者类作为抽象数据类型（ADT）实现（如图17.55所示）：

```
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;
};
```

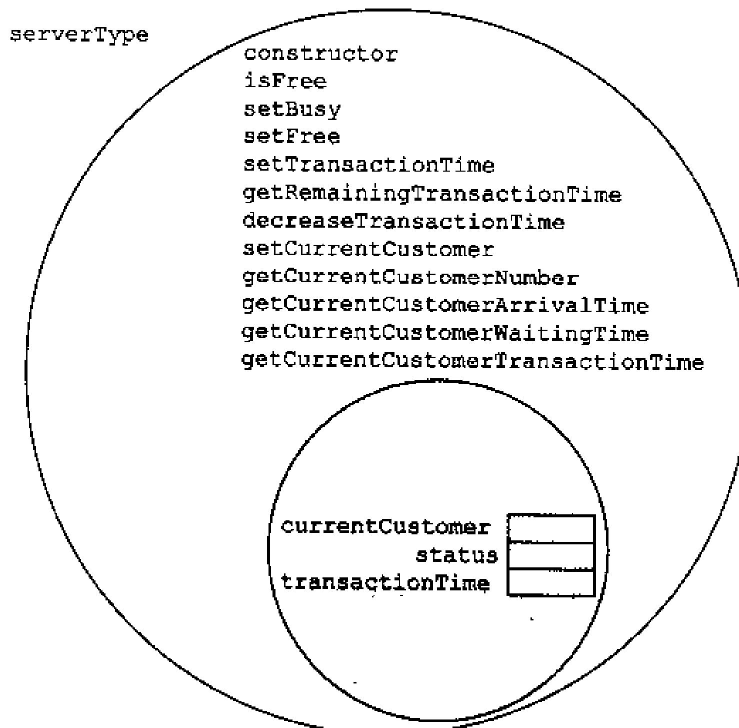


图 17.55　类 serverType

类 serverType 的成员函数定义很简单：

```
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();
}
```

由于要设计的是一个可以用于各种不同应用的模拟程序，所以还要设计两个类：一个创建和处理服务者表的类，一个创建和处理等待客户队列的类。下面两小节将讨论这两个类。

17.8.3 服务者（器）表

服务者表是一组服务者。在任何给定时刻，服务者不是处于空闲状态，就是处于忙状态。对位于队首的客户而言，需要从服务者表中找出一个空闲的服务者。如果所有服务者都是忙的，客户必须一直等待，直到其中一个服务者空闲。因此，实现服务者表类包括两个数据成员：一个记录服务者的个数，另一个维护服务者表。程序在执行时，根据客户指定的服务者个数，使用动态数组创建服务者表。服务者表必须实现的操作如下：返回一个空闲服务者ID；当一个客户准备好接受服务并且存在一个空闲服务者时，设置该服务者为忙；当模拟结束时，可能还有一些服务者忙，还要返回忙服务者的个数；每个时间单元过后，将每个忙服务者的transactionTime减去一个时间单元；还有如果某个服务者的transactionTime变为0，设置该服务者为空闲。下面的类serverListType将服务者表作为抽象数据类型（ADT）实现（如图17.56所示）：

```
class serverListType
{
public:
   serverListType(int num = 1);
      //constructor to initialize a list of servers
      //Post: numOfServers = 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;
};
```

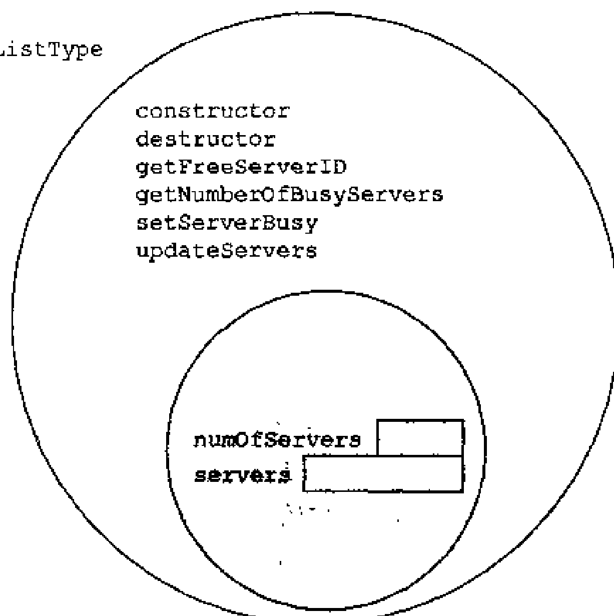


图 17.56 类 serverListType

下面是类 serverListType 的成员函数定义。构造函数和析构函数的定义很简单。

```
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 在服务者表中查找空闲服务者。如果找到，则返回该服务者 ID；否则，返回 -1，表示所有的服务者都忙。此函数定义如下所示：

```
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在服务者表中查找并确定忙服务者的个数。函数返回忙服务者的个数。其定义如下所示：

```
int serverListType::getNumberOfBusyServers()
{
      int busyServers = 0;

      int i;

      for(i = 0; i < numOfServers; i++)
          if(!servers[ i].isFree())
              busyServers++;

      return busyServers;
}
```

函数setServerBusy将一个服务者设置为忙状态。此函数是重载的。将被设为忙状态的服务者ID，即serverID，作为此函数的参数之一。该函数的一个重载函数根据参数tTime来设置服务者的事务处理时间，另一个重载函数则根据存储在对象cCustomer中的事务处理时间来设定服务者的事务处理时间。后面还要用到事务处理时间来计算平均等待时间。这两个函数的定义如下所示：

```
void serverListType::setServerBusy(int serverID,
                                   customerType cCustomer,
                                   int tTime)
{
   servers[ serverID] .setBusy();
   servers[ serverID] .setTransactionTime(tTime);
   servers[ serverID] .setCurrentCustomer(cCustomer);
}

void serverListType::setServerBusy(int serverID,
                                   customerType cCustomer)
{
      int time;

      time = cCustomer.getTransactionTime();

      servers[ serverID] .setBusy();
      servers[ serverID] .setTransactionTime(time);
      servers[ serverID] .setCurrentCustomer(cCustomer);

}
```

函数updateServers的定义很简单。从第一个服务者开始，依次查找忙服务者。当找到一个忙服务者时，将其transactionTime减1。如果transactionTime为0，则设置该服务者为空闲。如果transactionTime变为了0，则说明该服务者与客户的事务已经处理结束。这时将输出一条包括服务者ID，客户号码，客户离开时间的消息。函数updateServers是重载的。一个重载函数将消息输出到屏幕上，而另一个则输出到文件中。这些函数定义如下所示：

```
void serverListType::updateServers()
{
   int i;

   for(i = 0; i < numOfServers; i++)
      if(!servers[ i]..isFree())
      {
```

```
            servers[ i] .decreaseTransactionTime();

            if(servers[ i] .getRemainingTransactionTime() == 0)
            {
               cout<<"Server No: "<<(i+1)<<" Customer number "
                   <<servers[ i] .getCurrentCustomerNumber()
                   <<" departed at "<<endl
                   <<"             clock unit "
                   <<servers[ i] .getCurrentCustomerArrivalTime()
                     + servers[ i] .getCurrentCustomerWaitingTime()
                     + servers[ i] .getCurrentCustomerTransactionTime()
                   <<endl;
               servers[ i] .setFree();
            }
        }
}
void serverListType::updateServers(ofstream& outFile)
{
    int i;

    for(i = 0; i < numOfServers; i++)
        if(!servers[ i] .isFree())
        {
            servers[ i] .decreaseTransactionTime();

            if(servers[ i] .getRemainingTransactionTime() == 0)
            {
               outFile<<"Server No: "<<(i+1)<<" Customer number "
                      <<servers[ i] .getCurrentCustomerNumber()
                      <<" departed at "<<endl
                      <<"             clock unit "
                      <<servers[ i] .getCurrentCustomerArrivalTime()
                       + servers[ i] .getCurrentCustomerWaitingTime()
                       + servers[ i] .getCurrentCustomerTransactionTime()
                      <<endl;
               servers[ i] .setFree();
            }
        }
}
```

17.8.4 等待客户队列

当一个客户到来，他加入到队尾。当某个服务者可用时，则位于队首的客户离队开始接受服务。在每个时间单元过后，队列中每个客户的等待时间都加 1。本章设计的抽象数据结构 queueType 中包含了队列的绝大部分操作，只是缺少在每个时间单元过后将队列中所有客户等待时间增加的操作。我们将从类 queueType 派生一个新类 waitingCustomerQueueType，并加入实现客户队列所需的附加操作。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
};
```

下面给出成员函数的定义。构造函数和析构函数定义如下所示：

```
waitingCustomerQueueType::waitingCustomerQueueType(int size)
                          :queueType<customerType>(size)
{
}

waitingCustomerQueueType::~waitingCustomerQueueType()
{
}
```

函数updateWaitingQueue将队列中每个客户的等待时间增加一个时间单元。类waitingCustomerQueueType从类queueType派生。由于queueType的数据成员是私有的，故updateWaitingQueue不能直接访问队列元素。访问队列元素的惟一途径是使用deQueue操作。在增加了等待时间后，可以使用addQueue操作将元素重新放回队列。

addQueue操作将指定元素插入到队尾。如果我们对队列中每一个元素都执行deQueue，然后在执行addQueue操作，最终原来的队首元素会重新回到队首的位置上。假定在每个deQueue操作后面都有addQueue操作，如何判断队列中所有元素都经过处理了呢？我们不可能使用isEmptyQueue或isFullQueue，因为队列既不为空也不为满。

一种解决方法是创建一个临时队列。原有队列中的每个元素都要经过删除、处理，插入到临时队列中。当原始队列为空时，则说明队列中的所有元素都被处理过了。这时，可以将队列重新拷贝到原始队列中去。然而，这种方法需要使用额外的内存空间（可能会很大）。而且如果队列很大，在将元素从临时队列拷贝回原始队列中时会耗费大量的计算机时间。下面，再来看一下另一种解决方法。

在第二种方法中，在开始更新队列元素之前，可以插入一个等待时间是-1的伪客户。在处理过程中，当遇到等待时间是-1的客户时，就可以停下来不再对其进行更新操作。如果不处理等待时间为-1的客户，而将其删除，则在所有元素处理完后队列中将没有其他元素。这种方法不需要创建临时队列，所以也就不需要额外的计算机时间将元素拷贝回原始队列。这里，我们采用这种方法来更新队列。函数updateWaitingQueue定义如下所示：

```
void waitingCustomerQueueType::updateWaitingQueue()
{
    customerType cust;
    cust.setWaitingTime(-1);
    int wTime = 0;

    addQueue(cust);

    while(wTime != -1)
    {
        deQueue(cust);
        wTime = cust.getWaitingTime();

        if(wTime == -1)
            break;

        cust.incrementWaitingTime();
        addQueue(cust);
    }
}
```

17.8.5　主程序

要进行模拟，就必须获得下列信息：

- 模拟过程需要运行的时间单元数。假定每个时间单元是一分钟。
- 服务者个数。
- 服务一个客户所需的时间，即事务处理时间。
- 客户到来的大致时间间隔。

这些信息称为模拟参数。通过改变这些参数的值，就可以观察到系统性能的变化。可以使用函数 setSimulationParameters 提示客户输入这些值。

该函数的定义如下所示：

```
void setSimulationParameters(int& sTime, int& numOfServers,
                             int& transTime,
                             int& tBetweenCArrival)
{
      cout<<"Enter simulation time: "<<flush;
      cin>>sTime;
      cout<<endl;

      cout<<"Enter number of servers: "<<flush;
      cin>>numOfServers;
      cout<<endl;

      cout<<"Enter transaction time: "<<flush;
      cin>>transTime;
      cout<<endl;

      cout<<"Enter time between customer arrivals: "<<flush;
      cin>>tBetweenCArrival;
      cout<<endl;
}
```

当一个服务者变为空闲，并且客户队列非空时，队首客户就可以到这一服务者前接受服务。此外，当客户开始接受服务时，他的等待时间结束。客户的等待时间被加到总等待时间上。开始服务的主要算法如下所示（假设 serverID 指示了空闲服务者 ID）：

1. 将客户从队首删除

```
CustomerQueue.deQueue(customer);
```

2. 把当前客户的等待时间加到总等待时间上

```
totalWait = totalWait + customer.getWaitingTime();
```

3. 设置空闲服务者开始服务

```
serverList.setServerBusy(serverID, customer, transTime);
```

要进行模拟，就必须知道在给定时间内到来的客户数量以及每个客户的服务时间。我们使用统计学中的泊松（Poisson）分布，它说明 γ 事件在给定时间出现的概率为：

$$P(\gamma)=\frac{\lambda^{\gamma}e^{-\lambda}}{\gamma!},\ \gamma=0,1,2,\cdots,$$

其中 λ 是 γ 事件在那个时间出现的期望值。假定平均每4分钟到来一个客户。在一个4分钟时间间隔中，客户可能在任意一分钟到来。假设每一分钟到来都是可能的，则客户在某一分钟到来的期望值是1/4 = 0.25。下面，要确定客户是否会在某一给定分钟内到来。

既然 $P(0) = e^{-\lambda}$ 是在给定时刻，没有事件发生的概率。泊松分布的一个基本假设就是在一个短时间间隔内出现多个结果是可忽略的。为简单起见，假定一个给定时间单元中只有一个客户到来。因而，我们使用 $e^{-\lambda}$ 作为确认客户在给定时间单元是否到来的截止点。假定平均4分钟到来一个客户，则 $\lambda = 0.25$。可以使用一个算法生成一个介于0和1之间的数字。如果生成的数字大于 $e^{-0.25}$，就可以认为客户在特定时间单元内到达。例如，假定rNum是一个介于0和1之间的随机数字。如果rNum大于 $e^{-0.25}$，则客户在特定时间单元内到达。

现在，描述实现模拟的函数runSimulation。假设进行100个时间单元的模拟，而客户在时间单元93，96和100时到达。平均事务处理时间是5分钟——即5个时间单元。为简化起见，假定只有一个服务者且它在第97时间单元时为空闲，而在时间单元93前到来的客户都已处理完毕。当服务者在时间单元97变得空闲时，则在时间单元93到来的客户开始接受服务。由于该客户的服务需要5分钟完成，因此当模拟循环（即100个时间单元）结束时，该客户还处于服务中。此外，在96和100时间单元到来的客户还在队列中。为了简便，假定当模拟循环结束时，所有仍在服务者前的客户都被认为已服务。此函数的主要算法为：

1. 声明和初始化变量，例如模拟参数；客户ID；时钟；总等待时间和平均等待时间；到达客户数；已服务客户数；在等待队列中的剩余客户数；waitingCustomersQueue，以及服务者表。
2. 主循环为：

```
for(clock = 1; clock <= simulationTime; clock++)
{
```

2.1 更新服务者表。将每个忙服务者的事务处理时间减去一个时间单元。

2.2 如果客户等待队列非空，将每个客户的等待时间增加一个时间单元。

2.3 如果一个新客户到来，将客户数加1并将新客户插入到队列中。

2.4 如果某个服务器空闲且客户队列非空，从队首取出一个客户，并将它发送到该服务者前。

```
}
```

3. 输出结果。结果中包括：队列中剩余的客户个数、服务者前的客户个数、到达的客户个数，以及实际完成事务处理的客户个数。

在设计完函数runSimulation之后，函数main的定义就变得很简单，因为函数main只调用了一个函数——runSimulation。

在测试该模拟程序时，生成了下面结果。假定平均交易时间是5分钟，平均每3分钟到来一个客户，使用一个随机数生成器生成一个介于0和1之间的数，以确定客户是否在某指定时间段到来。

程序运行结果

程序运行结果1

```
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
Server No: 1 Customer number 7 departed at
             clock unit 47
Customer number 11 arrived at time unit 49
Customer number 12 arrived at time unit 51
Server No: 1 Customer number 8 departed at
             clock unit 52
Customer number 13 arrived at time unit 52
Customer number 14 arrived at time unit 53
Customer number 15 arrived at time unit 54
Server No: 1 Customer number 9 departed at
             clock unit 57
Customer number 16 arrived at time unit 59
Server No: 1 Customer number 10 departed at
             clock unit 62
Customer number 17 arrived at time unit 66
Server No: 1 Customer number 11 departed at
             clock unit 67
Customer number 18 arrived at time unit 71
Server No: 1 Customer number 12 departed at
             clock unit 72
Server No: 1 Customer number 13 departed at
             clock unit 77
Customer number 19 arrived at time unit 78
Server No: 1 Customer number 14 departed at
             clock unit 82
Server No: 1 Customer number 15 departed at
             clock unit 87
Customer number 20 arrived at time unit 90
Server No: 1 Customer number 16 departed at
             clock unit 92
Customer number 21 arrived at time unit 92
Server No: 1 Customer number 17 departed at
             clock unit 97

Simulation ran for 100 time units
Number of servers: 1
Average transaction time: 5
Average arrival time difference between customers: 4
Total wait time: 269
Number of customers completed transaction: 17
Number of customers left in servers: 1
Customers left in queue: 3
Average wait time: 12.81
************** END SIMULATION *************
```

程序运行结果2

```
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: 2 Customer number 4 departed at
             clock unit 18
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: 1 Customer number 8 departed at
             clock unit 47
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 *************
```

程序运行结果3 （为了节省空间，在输出中忽略了部分客户到达时间、离开时间的详细信息）

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

程序运行结果4 （为了节省空间，在输出中忽略了部分客户到达时间、离开时间的详细信息）

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

17.9 小结

1. 栈是一种插入和删除都在一端进行的数据结构。
2. 栈是一种后进先出（LIFO）的数据结构。
3. 栈的基本操作如下：将元素压入栈，将元素从栈中退出，初始化栈，删除栈，检查栈是否空，检查栈是否满。
4. 栈可以用数组也可以用链表实现。
5. 不能直接访问栈中间的元素。
6. 要使用栈顶元素，必须先将其从栈顶退出。
7. 栈分为数组栈和链表栈。
8. 后缀表达式不需要使用括号来指明运算符的优先级。

9. 在后缀表达式中，运算符在操作数之后。
10. 使用以下规则计算后缀表达式：
    a. 从左至右扫描表达式。
    b. 如果遇到运算符，回退该运算符所需数目的操作数，计算此表达式，然后继续。
11. 队列是在其一端插入元素而在另一端删除元素的数据结构。
12. 队列是一种先进先出（FIFO）的数据结构。
13. 队列的基本操作如下：将元素插入到队列、从队列中删除元素、初始化队列、删除队列、检查队列是否为空、检查队列是否为满。
14. 队列可以用数组也可以用链表实现。
15. 无法直接访问队列中间的元素。
16. 要使用队列首元素，必须先将其从队列中取出。
17. 队列分为数组队列和链表队列。

17.10 练习

1. 考虑下面语句：

```
stackType<int> stack;
int  x, y;
```

请写出下列代码段的输出结果：

```
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. 考虑下面语句：

```
stackType<int> stack;
int  x, y;
```

假定输入是：

```
14 45 34 23 10 5 -999
```

请写出下列代码段的输出结果：

```
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. 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. 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. 考虑下面语句：

```
stackType<int> stack;
queueType<int> queue;
int x, y;
```

   请写出下列代码段的输出结果：

```
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. 考虑下面语句:

```
stackType<int> stack;
queueType<int> queue;
int x, y;
```

假定输入是:

```
15 28 14 22 64 35 19 32 7 11 13 30 -999
```

请写出下列代码段的输出结果:

```
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，使用栈对象和队列对象作为参数。函数reverseStack使用队列来倒置栈上的元素。
8. 编写函数模板reverseQueue，使用栈对象和队列对象作为参数。函数reverseQueue使用栈来倒置队列上的元素。
9. 编写函数模板printListReverse，使用栈逆向打印队列元素。假定函数是第17章中定义的类linkedListType的成员函数。
10. 给类queueType增加操作queueCount，用来返回队列中元素的个数。编写此操作的函数模板。

17.11 编程练习

1. 如果两个栈中元素类型相同，元素个数相同，并且每个位置上相对应的元素都相同，称两个栈相同。为类stackType重载关系运算符==，使得当两个栈相同时返回true；否则返回false。请给出重载此运算符的函数模板定义。
2. 为类linkedStackType重复上面练习1。
3. a. 给类stackType增加下面操作：

   ```
   void reverseStack(stackType<Type> &otherStack);
   ```

   该操作将一个栈中元素逆向拷贝到另一栈中。
   考虑下面语句：

   ```
   stackType<int> stack1;
   stackType<int> stack2;
   ```

   语句：

   ```
   stack1.reverseStack(stack2);
   ```

   将stack1中的元素逆向拷贝到了stack2中。也就是说，stack1的栈顶元素是stack2的栈底元素，以此类推。stack2原有内容被删除，而stack1没有改变。

   b. 请给出实现操作reverseStack的函数模板。
4. 为类linkedStackType重复上面练习3a和3b。
5. 编写一个程序，该程序可以判断表达式中是否含有匹配的分组符号（在数学表达式中的圆括号和花括号），并输出相应的信息。例如，表达式{25 + (3-6) * 8}中就包含了匹配的分组符号。
6. 编写一个程序，以递增顺序打印某个正整数的所有质数因子（使用栈）。
7. 在第10章中，“程序范例：将二进制数转换成十进制数”中，使用递归将二进制数转化为相等的十进制数。编写一个程序，使用栈来完成相应的转换。
8. 在第10章中，“程序范例：将十进制数转换成二进制数”中，使用递归将十进制数转化为相等的二进制数。编写一个程序，使用栈来完成相应的转换。
9. 编写一个程序将中缀表达式转化为等价的后缀表达式。
   假设infx代表中缀表达式，pfx代表后缀表达式。

将中缀表达式 infx 转化为等价的后缀表达式 pfx 的规则是：

a. 初始化 pfx 为空的表达式并初始化栈。

b. 从 infx 取得下一个符号 sym。

b.1. 如果 sym 是操作数，将 sym 追加到 pfx。

b.2. 如果 sym 是 (，将 sym 压栈。

b.3. 如果sym是)，弹出并追加栈上所有元素到pfx直到遇到最近的左括号。弹出并丢弃该左括号。

b.4. 如果 sym 是一个运算符：

b.4.1 弹出并追加栈上所有在最近的左括号之上，且优先级大于或等于sym的运算符到pfx。

b.4.2 将 sym 入栈。

c. 处理完 infx，栈上可能会留下一些运算符。弹出并追加栈上所有运算符到 pfx。

在这个程序中，要考虑以下（双目）算术运算符：+，-，* 和 /。可以假定程序要处理的表达式都是无错的。

设计一个类来存储中缀和后缀表达式字符串。类必须包含下列操作：

1. getInfix：存储中缀表达式。
2. showInfix：输出中缀表达式。
3. showPostfix：输出后缀表达式。

还有一些可能需要的操作是：

4. convertToPostfix：将中缀表达式转换成后缀表达式，将后缀表达式存放在 postfixString 中。
5. precedence：比较两个运算符的优先级。如果第一个运算符的优先级高于或等于第二个运算符的优先级，返回 true；否则，返回 false。

要包含构造函数和析构函数，这样就能自动初始化和释放动态内存。

使用以下表达式测试该程序：

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，程序的输出格式如下所示：

```
Infix Expression: A + B - C;
Postfix Expression: AB+C-
```

10. 为类 queueType 编写赋值运算符和拷贝构造函数的重载函数的定义。然后，编写程序对之进行测试。
11. 为类 linkedQueueType 编写赋值运算符和拷贝构造函数的重载函数的定义。然后，编写程序对之进行测试。
12. 在本章讨论的队列的数组实现中，使用了一个特殊的数组单元（称为保留单元，用以区分队列空和队列满）。给出该类及各成员函数的完整定义。然后，编写程序对各种操作进行测试。
13. 给出函数 runSimulation 的定义，以完成计算机模拟程序（见“队列应用：模拟”一节）。使用不同的数据测试该程序。在本编程练习中，使用一个随机数生成器来判断在某一时间单元内是否会有客户到来。

第18章　查找和排序算法

本章要点：

- 了解各种查找算法
- 理解如何实现顺序查找算法或者折半查找算法
- 了解顺序查找算法或者折半查找算法的执行过程
- 了解渐进表示法（Asymptotic Notation）
- 了解各种排序算法
- 理解如何实现选择、插入、快速和归并排序算法
- 了解本章介绍的排序算法的执行过程

18.1　查找算法

第14章和第15章讨论了如何使用数组将数据组织到内存中，以及如何实现数据的基本操作。而第16章讨论了如何使用链表来组织数据。链表上最重要的操作就是查找。通过查找，可以：

1. 确认某一数据项是否在表中。
2. 如果数据是经过组织的（例如，经过排序），找出新的数据项应该插入的位置。
3. 查找要删除项的位置。

因而，查找算法的性能是很关键的。如果查找算法较慢，则需要花费大量的计算时间；如果查找算法较快，则可以很快完成任务。

第14章、第15章和第16章讨论了顺序查找算法的实现。本章将讨论其他的查找算法和排序算法。此外，还将给出一些算法分析。算法分析可以帮助程序员在特定的应用中选择查找算法。在描述这些算法之前，让我们先做下面讨论。

数据集中每一个数据项都有一个特殊成员，该成员用于在此数据集中惟一地标识该数据项。例如，在一个由学生记录组成的数据集中，学生ID就可以惟一地标识该校的每个学生。这个特殊成员称为该数据项的键（key）。数据项的键可以用于在数据集中做查找、排序、插入和删除等操作。例如，在数据集中查找指定数据项时，就要用查找项的键同数据集中各数据项的键进行比较。

本章在讨论查找和排序算法之后，还将分析这些算法。在算法分析中，键值比较是指用数据项的键同数据项集合中某项的键进行比较。此外，键值比较次数是指（在查找或排序算法中）数据项的键同数据项集合中数据项的键的比较次数。

在第14章和第15章，设计并实现了类arrayListType。该类用来实现表及基本的表的操作。由于本章还要引用这个类，为了便于引用，这里给出它的定义：

```
template<class elemType>
class arrayListType
{
public:
    const arrayListType<elemType>& operator=
                    (const arrayListType<elemType>& otherList);
      //Overload the assignment operator
```

```
bool isEmpty();
  //Returns true if the list is empty;
  //otherwise, returns false.
bool isFull();
  //Returns true if the list is full;
  //otherwise, returns false.
int listSize();
  //Returns the size of the list; that is, the number
  //of elements currently in the list.
int maxListSize();
  //Returns the maximum size of the list; that is, the
  //maximum number of elements that can be stored in
  //the list.
void print() const;
  //Outputs the elements of the list.
bool isItemAtEqual(int location, const elemType& item);
  //If the item is the same as the list element at the
  //position specified by the location, returns true;
  //otherwise, returns false.
void insertAt(int location, const elemType& insertItem);
  //Inserts an item in the list at the specified location.
  //The item to be inserted and the location are passed
  //as parameters to the function.
  //If the list is full or the location is out of range,
  //an appropriate message is displayed.
void insertEnd(const elemType& insertItem);
  //Inserts an item at the end of the list. The parameter
  //insertItem specifies the item to be inserted.
  //If the list is full, an appropriate message is
  //displayed.
void removeAt(int location);
  //Removes the item from the list at the specified
  //position. The location of the item to be removed is
  //passed as a parameter to this function.
void retrieveAt(int location, elemType& retItem);
  //Retrieves the element from the list at the position
  //specified by location. The item is returned via
  //the parameter retItem.
  //If the location is out of range, an appropriate
  //message is printed.
void replaceAt(int location, const elemType& repItem);
  //Replaces the elements in the list at the position
  //specified by location. The item to be replaced is
  //specified by the parameter repItem.
void clearList();
  //All elements from the list are removed. After this
  //operation, the size of the list is zero.
int seqSearch(const elemType& item);
  //Searches the list for a given item. If the item is found,
  //returns the location where in the array the item is found;
  //otherwise, returns -1.
void insert(const elemType& insertItem);
  //The item specified by the parameter insertItem is
  //inserted at the end of the list. However, first the
  //list is searched to see whether the item to be inserted
  //is already in the list. If the item is already in the
  //list, an appropriate message is output.
```

```
        void remove(const elemType& removeItem);
          //Removes an item from the list. The parameter removeItem
          //specifies the item to be removed.
        arrayListType(int size = 100);
          //constructor
          //Creates an array of the size specified by the parameter
          //size. The default array size is 100.
        arrayListType (const arrayListType<elemType>& otherList);
          //copy constructor
        ~arrayListType();
          //destructor
          //Deallocates the memory occupied by the array.

    protected:
        elemType *list; //array to hold the list elements
        int length;     //stores the length of the list
        int maxSize;    //stores the maximum size of the list
    };
```

18.1.1 顺序查找

在第14章和第15章中，讨论了基于数组的顺序查找（也称为线性查找）；在第16章中，讨论了基于链表的顺序查找。顺序查找既可以在数组上实现，又可以在链表上实现。查找总是从表中的第一个元素开始，直到找到该数据项或者搜索完整个表为止。

由于要知道顺序查找的性能（也就是该类查找的分析），为了方便和完整性起见，这里将给出基于数组的表的顺序查找算法。如果找到查找项，则返回它的下标；否则，返回-1。注意，下面给出的顺序查找算法不要求表元素以任何形式有序：

```
template<class elemType>
int arrayListType<elemType>::search(const elemType& item)
{
      int loc;
      bool found = false;
      for(loc = 0; loc < length; loc++)
          if(list[ loc] == item)
          {
              found = true;
              break;
          }
      if(found)
         return loc;
      else
         return -1;
} //end search
```

注意：顺序查找算法使用了迭代控制结构（for循环）进行比较。也可以编写一个递归算法实现顺序查找（见本章后的编程练习1）。

顺序查找分析

由于常用的查找算法很多，用户必须知道哪个算法是最适用的。本章还将讨论其他的查找算法，所以我们必须了解每种算法的基本原理。本节中，将分析顺序查找算法的平均效率和最低效率。

因为在循环以外的语句只执行一次，所以执行这些语句只需要花费很少时间。而for循环中的语句则要重复执行固定的次数。在每次循环中，主要是将查找项和表中的数据项相比较，可能还包括其他一些语句（包括其他比较语句）。很显然，循环将在查找项与表中的数据项完全匹配后结束。因此，循环

中其他语句的执行次数直接取决于键比较的次数。虽然不同的程序员在实现相同的查找算法时可能会略有差异，但是通常键比较语句的执行次数都相同。虽然计算机的性能会影响算法执行的时间，但是不会影响到键比较语句执行的次数。

因此，分析一个查找算法时，从键比较语句的执行次数入手是最有效的方法。进一步说，对于所有的查找算法，这个方法都适用。

假设表L的长度为n。下面讨论在L中查找某一项时顺序查找算法中键比较的执行次数。

如果查找项不在表中，算法会将查找项和表中的每一数据项做比较。也就是说，在查找失败时，比较次数为n。

假设查找项在表中。那么键比较语句的执行次数取决于查找项的位置。如果查找项是表的第一个元素，这是最好情况，只需做一次比较。但是，如果查找项是表尾元素，这是最坏情况，要执行n次比较。并非在每次查找时都会遇到最好情况或者最坏情况，所以考虑平均次数更有用。也就是说，需要确定顺序查找算法中键比较的平均次数。

确定顺序查找算法中键比较的平均次数：

1. 考虑所有可能的情况
2. 找出每种情况的比较次数
3. 求出总比较次数和，然后除以情况总数

如果查找项，即目标项（target），是表的第一个元素，则只需要比较一次。如果是第二个元素，则需要比较两次。以此类推，如果目标项是表中第k个元素，则需要比较k次。假设目标项是表中的任意一个元素，也就是说，表中元素等于目标项的机会均等。表中有n个元素，则平均比较次数为：

$$\frac{1+2+\cdots+n}{n}$$

已知：

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

所以，下面的公式给出了成功查找的平均次数：

$$\frac{1+2+\cdots+n}{n}=\frac{1}{n}\frac{n(n+1)}{2}=\frac{n+1}{2}$$

该公式指出，平均查找次数等于表长度的一半。如果表长度为1 000 000，则平均来说该算法要做500 000次比较。由此可见，顺序查找算法不适用于长度很长的表。

18.1.2 有序表

有序表是指表中所有元素都按照一定规则排序。有序表中的元素一般是升序排列。有序表中的某些操作和无序表是相同的，例如：确定表是否空或满、确定表长度、输出表、清空表。因此，可以使用继承机制，从前面章节的**arrayListType**类模板派生出所需的有序表类模板，将有序表定义为一个抽象数据类型（ADT）。依据表的存储方式（数组或链表），分别定义两个类模板。

类模板orderedArrayListType，定义了一个基于数组存储方式的抽象数据类型（ADT）有序表：

```
template <class elemType>
class orderedArrayListType: public arrayListType<elemType>
{
public:
    orderedArrayListType(int size = 100);
      //constructor
```

```
        ...
        //We will add the necessary members as needed.

private:
        //We will add the necessary members as needed.
}
```

第 16 章定义了下面的链表类模板来实现有序链表。本章节后面的部分也将引用这个类模板。

```
template<class elemType>
class orderedLinkedListType: public linkedListType<elemType>
{
public:
        ...

}
```

18.1.3 折半查找

正如前面所见，顺序查找算法的平均查找次数等于表长度的一半，所以不太适合很长的表。因此，我们将讨论另一种算法，即折半查找算法。折半查找是一种效率很高的算法，但是只能使用于有序表中。因此在本节中，假设所有表都已经排序。在本章后面，我们将讨论几种排序算法的性能和适用情况。

折半查找算法使用“拆分”技术来搜索表中元素。首先，将查找项和表正中间的元素相比较，如果查找项小于表正中间的元素，则把查找范围限制在表的前半部分；否则，把查找范围限制在表的后半部分。

考虑下面的有序表，其中 length = 12，如图 18.1 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 18.1 长度为 12 的表

假设要在表中查找值为 75 的元素。开始时，查找范围是整个的表（如图 18.2 所示）

查找范围

	[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

中间元素

图 18.2 查找范围是 list[0]…list[11]

首先，将 75 与表的中间元素 list[5]的值 39 相比较。由于 75 ≠ list[5]且 75 > list[5]，所以将查找范围限制在 list[6]…list[11]之间，如图 18.3 所示。

查找范围

	[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

中间元素

图 18.3 查找范围是 list[6]…list[11]

然后，在表 list[6]…list[11]中重复上述过程，此时 length = 6。

因为要频繁地确定表的中间元素，所以折半查找算法通常应用于数组存储的有序表。这样，可以通过将表头元素下标和表尾元素下标相加，除以 2 来确定表中间元素的下标。即：mid = (first + last) /2。初始时，first = 0 (因为 C++ 中数组下标从 0 开始，而 length 则表示表中元素的数目)，last = length - 1。

下面的 C++ 函数使用了折半查找算法。如果查找项在表中，则返回其下标；如果查找项不在表中，则返回 -1。

```
template<class elemType>
int orderedArrayListType<elemType>::binarySearch
                                         (const elemType& item)
{
      int first = 0;
      int last = length - 1;
      int mid;

      bool found = false;

      while(first <= last && !found)
      {
            mid = (first + last) / 2;

            if(list[mid] == item)
               found = true;
            else
               if(list[mid] > item)
                  last = mid - 1;
               else
                  first = mid + 1;
      }

      if(found)
         return mid;
      else
         return -1;
} // end binarySearch
```

在折半查找算法中，每次循环中都要进行两次比较。惟一一次例外是最后一次循环，只进行一次比较。

注意：在上面的折半查找算法中，使用 while 循环结构来比较查找项和表中元素。也可以利用递归来实现折半查找算法（见本章后面的编程练习 2）。

例 18.1　本例进一步说明了折半查找算法的工作过程。考虑图 18.4 中的表。

	[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 18.4　折半查找有序表

这个表的大小，即长度为 12。下面的表显示了每次循环中 first，last 和 mid 的值。表中同时显示了每次循环中查找项和表中元素的比较次数。

假设查找项是 89。

Iteration	first	last	mid	list[mid]	Number of Comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (found 为 true)

查找项在位置10上被找到，总比较次数是5。

下面查找34。

Iteration	first	last	mid	list[mid]	Number of 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 为 true)

这次查找项在位置4上被找到，总比较次数是7。

下面查找22。

Iteration	first	last	mid	list[mid]	Number of Comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	3	2	循环中止 (因为 first > last)		

这次查找失败，总比较次数是6。

折半查找算法的性能

假设L是一个有1 000个元素的有序表，要在表中查找x。因为L是经过排序的，可以利用折半查找算法来查找x。假设L如图18.5所示。

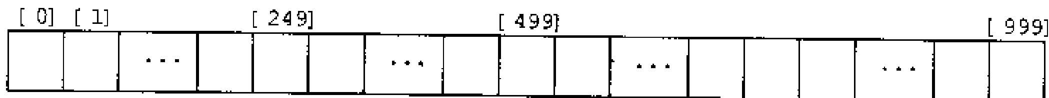


图18.5 表L

while语句的第一次循环在一个1 000个元素的表：L[0]…L[999]中查找x。该while循环中x和L[499]比较（如图18.6所示）。

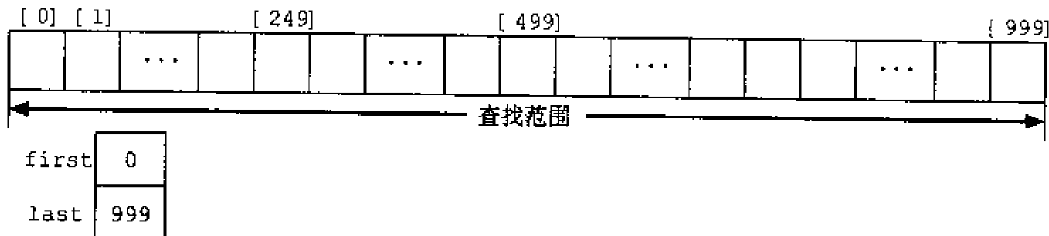


图18.6 查找表

假设x ≠ L[499]，如果x<L[499]，则下一次循环在L[0]…L[498]中查找x；否则，在L[500]…L[999]中查找x。假设x<L[499]，下面，while循环在L[0]…L[498]中，共计499个元素的表中查找x，如图18.7所示。

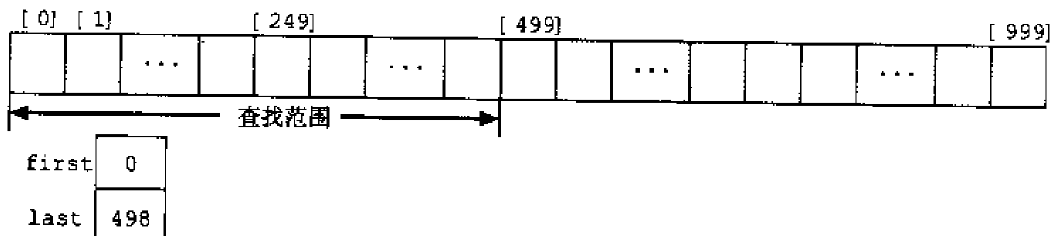


图18.7 第一次循环后查找表

这时，while循环将x和L[249]相比较。再次假设 $x \neq L[249]$，且 $x > L[249]$。下一步循环将在L[250]…L[498]中查找x，这个表共计249项，如图18.8所示。

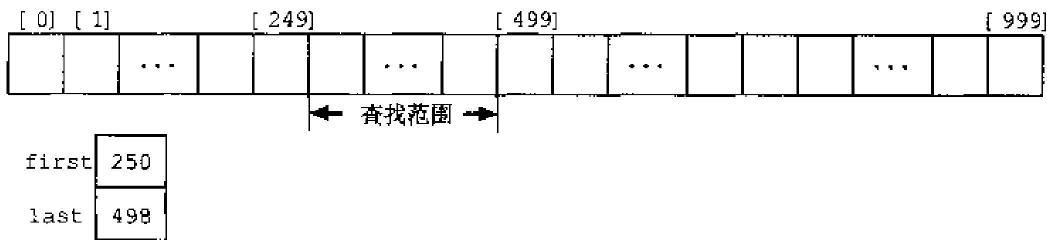


图 18.8 在第二次循环后查找表

从上面讨论可知，每次 while 循环都把查找范围缩减一半。因为 $1\,000 \approx 1\,024=2^{10}$，while 循环最多需要循环 11 次就可以确定 x 是否在 L 中。由于每次循环中都有两次比较，所以查找项 x 和 L 中的元素比较两次——折半查找最多需要比较 22 次。相比之下，如果使用顺序查找算法，平均需要做 500 次比较。

为了更好地理解折半查找与顺序查找的比较，假设表L的大小为1 000 000。由于 $1\,000\,000 \approx 1\,048\,576 = 2^{20}$，因此，while 循环最多需要循环 21 次就可以确定x是否在L中。由于每次循环中都有两次比较，因此，确定一个元素是否在表L中，折半查找最多需要比较 42 次。而相应的顺序查找算法，平均要做 500 000 次比较来决定一个元素是否在表 L 中。

注意：

$$40 = 2*20 = 2*\log_2 2^{20} = 2*\log_2(1\ 048\ 576) \approx 2*\log_2(1\ 000\ 000)$$

通常，如果 L 是一个大小为 n 的有序表，要确定一个元素是否在 L 中，折半查找最多需要做 $2*\log_2 n + 2$ 次比较。

不成功的查找

查找失败时，对一个长度为 n 的表，折半查找的次数大约是 $2*\log_2(n+1)$。

成功查找

查找成功时，对一个长度为 n 的表，折半查找平均要做 $\frac{2(n+1)}{n}\log_2(n+1)-3$ 次比较。

现在，我们已经知道了如何对一个有序表进行有效查找。在讨论排序算法前，让我们看看如何在一个有序表中插入一项。

18.1.4 有序表的插入

假设要在一个有序表中插入一项。插入后，表必须仍然是有序的。第 16 章描述了如何在一个已排序的链表中插入一项，本节将介绍如何在基于数组方式的有序表中插入一项。

在插入元素之前，先要找到插入位置。接着，将表中的元素依次移动一个位置，腾出空间留给插入项，然后插入元素。由于表以数组方式存储，所以可以使用一个类似于折半查找的算法，来确定插入项的位置。然后，可以利用函数 insertAt（类 arrayListType）将该项插入（注意，不能照搬前面的折半查找算法，因为该算法在插入项不在表中时，返回 -1。当然，可以另外编写一个函数，利用折半查找算法找到插入位置）。因此，插入算法的步骤如下所示：

1. 利用类似于折半查找的算法找到插入位置。
2. if 该项已经在表中：

    输出相关的信息

else
    使用函数 insertAt 将该项插入

下面函数 insertOrd 实现了上述算法：

```
template<class elemType>
void orderedArrayListType<elemType>::insertOrd(const elemType& item)
{
      int first = 0;
      int last = length - 1;
      int mid;

      bool found = false;

      if(length == maxSize)
         cout<<"Cannot insert into a full list."<<endl;
      else
      {
         while(first <= last && !found)
         {
               mid = (first + last) / 2;

              if(list[mid] == item)
                 found = true;
              else
                 if(list[mid] > item)
                    last = mid - 1;
                 else
                    first = mid + 1;
         } //end while

         if(found)
            cout<<"The insert item is already in the list. "
                <<"Duplicates are not allowed.";
         else
         {
            if(list[mid] < item)
                  mid++;
             insertAt(mid, item);
         }
      }
} //end insertOrd
```

如果将折半查找算法和 insertOrd 函数加到类 orderedArrayListType 中，则该类定义如下所示：

```
template<class elemType>
class orderedArrayListType: public arrayListType<elemType>
{
public:
    void insertOrd(const elemType&);
    int binarySearch(const elemType& item);
    orderedArrayListType(int size = 100);
};
```

类似地，可以编写一个函数，从一个有序表中删除一个元素。见本章后面的编程练习 6。

18.2 渐近表示法

假设一个算法完成一个任务需要$f(n)$步基本操作，n代表问题的规模。例如在查找算法中，n是表的大小，$f(n)$则是计数函数，即$f(n)$给出了查找算法中比较的次数。假设在某计算机上，执行一步操作需要花费c个运算时间单位，则在该计算机上执行$f(n)$步操作需要的时间为$cf(n)$。很显然，c是一个与具体计算机有关的常数。但是基本操作次数$f(n)$，对于所有计算机都是一样的。如果知道了问题规模与$f(n)$的关系，就可以知道算法的效率了。考虑下面的表：

n	$\log_2 n$	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65 536
32	5	160	1 024	4 294 967 296

上表说明了随着问题规模n的变化，某些函数的变化趋势。假设问题规模增加一倍，从上表中可以看出：如果基本操作的执行次数函数是$f(n) = n^2$，则执行的基本操作次数是原来的4倍；如果基本操作的执行次数函数是$f(n) = 2^n$，则执行的基本操作次数是原来的平方；如果基本操作的执行次数函数是$f(n) = \log_2 n$，则执行的基本操作次数变化得不那么明显。

假设一台计算机每秒钟执行十亿次基本操作，下表显示了执行$f(n)$步基本操作所需的时间。

n	$f(n) = n$	$f(n) = \log_2 n$	$f(n) = \log_2 n$	$f(n) = n^2$	$f(n) = 2^n$
10	0.01 µs	0.003 µs	0.03 µs	0.1 µs	1
20	0.02 µs	0.004 µs	0.09 µs	0.4 µs	1 ms
30	0.03 µs	0.005 µs	0.15 µs	0.9 µs	1 s
40	0.04 µs	0.005 µs	0.21 µs	1.6 µs	18.3 分钟
50	0.05 µs	0.006 µs	0.28 µs	2.5 µs	13 天
100	0.10 µs	0.007 µs	0.66 µs	10 µs	4×10^{13} 年
1 000	1.00 µs	0.010 µs	9.96 µs	1 ms	
10 000	10 µs	0.0.13 µs	130 µs	100 ms	
100 000	0.10 µs	0.017 µs	1.67 ms	10 s	
1 000 000	0.01 s	0.020 µs	19.92 ms	16.7 m	
10 000 000	0.10 s	0.0.23 µs	0.23 s	1.16 天	
100 000 000	1.00 s	0.0.27 µs	2.66 s	115.7 天	

上表中：1 µs = 10^{-6}秒，1 ms = 10^{-3}秒。

本节的余下部分使用了一种表示法，该表示法可以用来描述一个函数$f(n)$是如何随着n的无限增加而增长的。通过这种表示法，可以方便地描述算法的执行效率。首先，我们定义在本节标题中出现的术语“渐近”。

令f是关于n的一个函数。术语“渐近”表示随着n的无限增加，f的增长趋势。

考虑函数$g(n) = n^2$和$f(n) = n^2+4n+20$。很清楚，函数g不包括任何线性项，即n的一次项系数为0。考虑下面的表：

n	$g(n) = n^2$	$f(n) = n^2+4n+20$
10	100	160
50	2 500	2 720
100	10 000	10 420
1 000	1 000 000	1 004 020
10 000	100 000 000	100 040 020

显然，随着n的增大，$f(n)$中$4n+20$的变化并不明显，n^2才是主导变化项。当n的值很大时，可以通过$g(n)$来估计$f(n)$的值。在算法分析中，如果一个函数的复杂度可以由一个没有线性项的二次函数描述，则我们就说这个函数是$O(n^2)$的，称为n^2的Big-O。

假设f和g都是实值函数，并且f和g非负。

定义 如果存在正常数c和n_0，对所有的$n \geq n_0$，都有$f(n) \leq cg(n)$，则$f(n)$是$g(n)$的Big-O，记为$f(n)=O(g(n))$。

下表显示了一些算法分析中出现的Big-O函数。令$f(n)=O(g(n))$，其中n是表示问题的规模。

函数$g(n)$	$f(n)$的增长率
$g(n)=1$	增长率是不依赖于问题规模n的常数
$g(n)=\log_2 n$	增长率是$\log_2^n$的函数。因为对数函数增长很慢，函数f增长率也很慢
$g(n)=n$	增长率是线性函数，与问题规模n同阶
$g(n)=n*\log_2 n$	增长率大于线性函数
$g(n)=n^2$	增长率随问题规模的增大而迅速增长。当问题规模增加一倍时，增长率增长4倍
$g(n)=2^n$	增长率是指数增长。当问题规模增大一倍时，增长率成平方增加

注意：$O(1)<O(\log_2 n)<O(n)<O(n*\log_2 n)<O(n^2)<O(2^n)$

使用本节中介绍的表示法，下表总结了前面讨论的查找算法的分析结果。

一个长度为n的表的比较次数

算法	查找成功	查找失败
顺序查找	$\frac{n+1}{2}=O(n)$	$n=O(n)$
折半查找	$\frac{2(n+1)}{n}\log_2(n+1)-3=O(\log_2 n)$	$2*\log_2(n+1)=O(\log_2 n)$

18.2.1 基于比较的查找算法的下界

顺序查找和折半查找算法都要对查找项和表中的元素进行比较。因此，这类算法被称为基于比较的查找算法。本章前面的部分已经指出，顺序查找算法的复杂度为n，而折半查找算法的复杂度为$\log_2 n$，其中n是表的长度。一个很明显的问题是：能否找到一种复杂度小于$\log_2 n$的算法？在回答这个问题前，必须要知道基于比较的查找算法的比较次数的下界。

定理 设表L的长度为n，$n>1$。假设表L的元素已经排序，假如SRH(n)是需要比较的最少次数，在最坏情况下，使用基于比较的查找算法在L中查找元素x时，则SRH(n)$\geq \log_2(n+1)$。

推论 在基于比较的查找方法中，折半查找算法是最差情况下的最佳算法。

通过上面的讨论，可以得出：在采用基于比较的查找算法的前提下，要设计一个复杂度小于$\log_2 n$的查找算法是不可能的。

18.3 排序算法

上面章节讨论了表中的查找算法。顺序查找算法并不要求数据已经有序，但是在很长的表中使用这种算法的效率不高。相比较而言，在处理基于数组的表时，折半查找算法的效率非常高。但是，折半查找算法要求表中元素必须已经有序。

因为折半查找算法要求数据元素有序，而且在基于数组的表中查找的效率很高。本章的下面部分将集中讨论排序算法。因为排序算法很多，所以我们只能介绍一些常用的算法。为了比较这些算法的效率，

还要对这些算法进行分析。排序算法既可以应用于基于数组的表，也可以应用于链表。我们将具体区分这些算法是应用于数组表还是链表。

通常，实现这些算法的函数作为public成员包含在相应的类中（例如，对基于数组的表来说，它们是类 orderedArrayListType 的成员）。这样，算法就可以直接存取表元素。

假设在基于数组的表中实现了选择排序算法（将在下一节讨论）。下面代码说明了如何将其作为成员函数包含在类 orderedArrayListType 中：

```
template <class elemType>
class orderedArrayListType: public arrayListType<elemType>
{
public:
    void selectionSort();
    ...
};
```

18.3.1　选择排序：基于数组表

选择排序算法先通过找到表中未排序部分的最小元素，然后将其移动到未排序部分的最前端来排序一个表。选择排序算法第一次在整个表中查找最小元素，第二次从表的第二项开始查找最小元素，以此类推。在此描述的选择排序算法是用于基于数组的表。

例如，有如图 18.9 所示的表：

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
list	16	30	24	7	25	62	45	5	65	50

图 18.9　有 10 个元素的表

开始时，整个表是无序的。先要在表中查找最小元素，其下标是 7，如图 18.10 所示。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
list	16	30	24	7	25	62	45	5	65	50

图 18.10　无序表中的最小元素

因为它是整个表中的最小元素，所以要将其移到下标为 0 的位置上。因此，将 16（list[0]）和 5（list[7]）交换，如图 18.11 所示。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
list	16	30	24	7	25	62	45	5	65	50

图 18.11　交换元素 list[0]和 list[7]

交换后，结果如图 18.12 所示。

现在，无序表是 list[1]…list[9]。下面，再次查找最小元素，其下标是 3，如图 18.13 所示。

因为未排序部分的最小元素的下标是 3，所以要将其移到下标为 1 的位置上。因此，将 7（list[3]）和 30（list[1]）交换，如图 18.14 所示。

图 18.12 交换 list[0]和 list[7]后的表

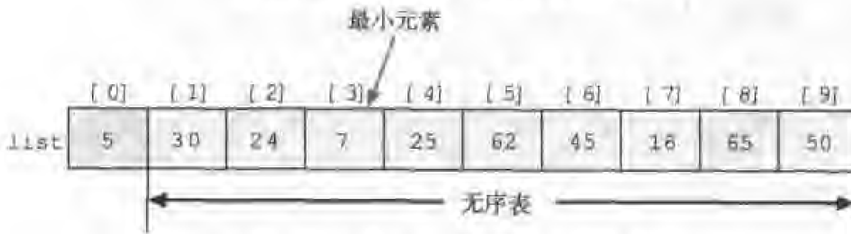


图 18.13 表中未排序部分的最小元素

图 18.14 交换 list[1]和 list[3]

交换后，结果如图 18.15 所示。

图 18.15 交换 list[1]和 list[3]后的表

现在，无序表是 list[2]…list[9]。再次重复上述过程，查找最小元素，将其移到无序表的第一个位置上。因此，选择排序包括以下步骤（在表的未排序部分）：

a. 找到最小元素

b. 将最小元素移到未排序部分的第一个位置上

初始情况下，整个表 list[0]…list[length–1]为无序表。步骤 a 和步骤 b 执行一次以后，无序表变为 list[1]…list[length–1]。再次执行步骤 a 和步骤 b 后，无序表变为 list[2]...list[length–1]，以此类推。可以通过使用 for 循环语句，实现步骤 a 和步骤 b 的循环执行：

```
for (index = 0; index < length - 1; index++)
{
      a.在 list[ index] 和 list[ length-1] 中找到最小元素的下标 smallestIndex。
      b.将最小元素 list[ smallestIndex] 和 list[ index] 交换。
}
```

第一次循环时，在 list[0]…list[length–1]中找最小元素，并将其和 list[0]交换。第二次循环时，在 list[1]…list[length–1]中找最小元素，并将其和 list[1]交换，以此类推。这个过程不断重复，直到表中未排序部分的长度为 1（注意，长度为 1 的表可以认为是有序的）。接下来就是怎样实现选择排序算法，即实现步骤 a 和步骤 b。

假设第一个元素的下标为 first，最后一个元素的下标为 last。下面的 C++ 函数返回表 list[first]…list[last]中最小元素的下标：

```
template<class elemType>
int orderedArrayListType<elemType>::minLocation(int first, int last)
{
       int loc, minIndex;

       minIndex = first;

       for(loc = first + 1; loc <= last; loc++)
           if(list[loc] < list[minIndex])
              minIndex = loc;

           return minIndex;
} //end minLocation
```

如果需要交换元素，下面的 C++ 函数 swap 可以实现该功能：

```
template<class elemType>
void orderedArrayListType<elemType>::swap(int first, int second)
{
       elemType temp;

       temp = list[first];
       list[first] = list[second];
       list[second] = temp;
} //end swap
```

下面给出完整的排序函数 selectionSort：

```
template<class elemType>
void orderedArrayListType<elemType>::selectionSort()
{
       int loc, minIndex;

       for(loc = 0; loc < length -1; loc++)
       {
           minIndex = minLocation(loc, length - 1);
           swap(loc, minIndex);
       }
}
```

实现选择排序算法的类 orderedArrayListType 定义如下所示：

```
template<class elemType>
class orderedArrayListType: public arrayListType<elemType>
{
public:
    void insertOrd(const elemType&);
    int binarySearch(const elemType& item);
    void selectionSort();

    orderedArrayListType(int size = 100);

private:
    void swap(int first, int second);
    int minLocation(int first, int last);
};
```

例18.2 下面程序测试选择排序算法。假设类orderedArrayListType的定义在头文件orderedArrayListType.h中。

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

using namespace std;

int main()
{
    orderedArrayListType<int> list;                      //Line 1
    int num;                                             //Line 2

    cout<<"Line 3: Enter numbers ending with -999"
        <<endl;                                          //Line 3

    cin>>num;                                            //Line 4

    while(num != -999)                                   //Line 5
    {
        list.insert(num);                                //Line 6
        cin>>num;                                        //Line 7
    }

    cout<<"Line 8: The list before sorting:"<<endl;      //Line 8
    list.print();                                        //Line 9
    cout<<endl;                                          //Line 10
    list.selectionSort();                                //Line 11

    cout<<"Line 12: The list after sorting:"<<endl;      //Line 12
    list.print();                                        //Line 13
    cout<<endl;                                          //Line 14

    return 0;
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
Line 3: Enter numbers ending with -999
34 67 23 12 78 56 36 79 5 32 66 -999
Line 8: The list before sorting:
34 67 23 12 78 56 36 79 5 32 66


Line 12: The list after sorting:
5 12 23 32 34 36 56 66 67 78 79
```

程序中的大部分都很简单，需要注意的是第6行语句调用了函数insert，它是类arrayListType的成员函数，而类arrayListType又是类orderedArrayListType的基类。类似地，第9行和第13行语句调用了类arrayListType的成员函数print。第11行语句调用了函数selectionSort对表进行排序。

注意： 1. 选择排序算法还有一个实现方式：在表中查找最大元素，并将其移到无序表中最后一个位置上。只要改动函数minLocation中if语句部分，并在函数selectionSort中传递相应参数给对应函数和swap函数就很容易实现。

2. 选择排序算法也可以用于链表。算法的绝大部分都相同，微小的改动工作留给读者作为练习。参见本章后面的编程练习7。

18.3.2 分析：选择排序

在查找算法中，我们关心的只是比较的次数。而排序算法既进行比较，又要进行元素移动。因此，进行排序算法分析时，既要注意比较次数，也不能忽视移动次数。让我们来看看选择排序算法的效率。

设表长为 n，每个 swap 函数执行 3 条赋值语句，函数 swap 本身执行 $n-1$ 次。因此，赋值语句执行的总次数为 $3(n-1)$。

比较语句在函数 minLocation 中。对一个长度为 k 的表，函数 minLocation 进行 $k-1$ 次比较。同时，函数 minLocation 执行 $n-1$ 次（通过函数 selectionSort）。第一次，函数 minLocation 在整个表中查找最小元素的下标，需要进行 $n-1$ 次比较。第二次，函数 minLocation 在长度为 $n-1$ 的子表中查找最小元素的下标，需要进行 $n-2$ 次比较，以此类推。所以，比较次数为：

$$
\begin{aligned}
(n-1)+(n-2)+\cdots+2+1 &= \frac{n(n-1)}{2} \\
&= \frac{1}{2}n^2 - \frac{1}{2}n \\
&= \frac{1}{2}n^2 + O(n) \\
&= O(n)^2
\end{aligned}
$$

不难得出，如果 $n=1\,000$，则比较次数是 $\frac{1}{2}(1\,000)^2 - \frac{1}{2}(1\,000) = 499\,500 \approx 500\,000$。

18.4 插入排序：基于数组表

前面一节介绍并分析了选择排序算法。显然如果表长 n=1 000，比较次数约为 500 000，这个数目很大。本节将介绍插入排序算法，试图提高效率，减少比较次数。

插入排序算法通过将每个元素移到相应的位置上来排序，如图 18.16 所示。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list	10	18	25	30	23	17	45	35

图 18.16 表

这个表长度为 8。在表中，list[0]，list[1]，list[2]，list[3]是顺序的，也就是说它们是表中有序部分（如图 18.17 所示）。

	有序表				无序表			
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list	10	18	25	30	23	17	45	35

图 18.17 表的有序和无序部分

接下来，考虑元素 list[4]，未排序部分的第一个元素。因为 list[4] < list[3]，所以要将元素 list[4] 移到相应的位置上。在该表中，元素 list[4]要移到 list[2]中（如图 18.18 所示）。

为了将 list[4]移到 list[2]，首先要将 list[4]拷贝到一个临时空间 temp 中（如图 18.19 所示）。

接下来，将 list[3]拷贝到 list[4]中，然后将 list[2] 拷贝到 list[3]中（如图 18.20 所示）。

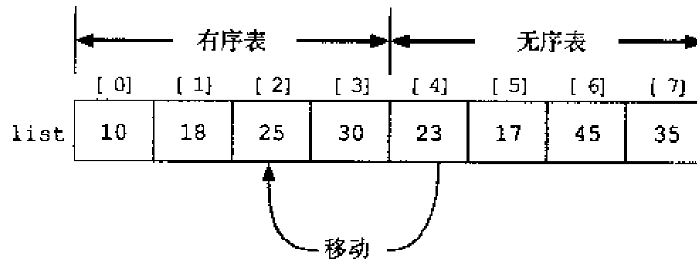


图 18.18 将 list[4]移到 list[2]中

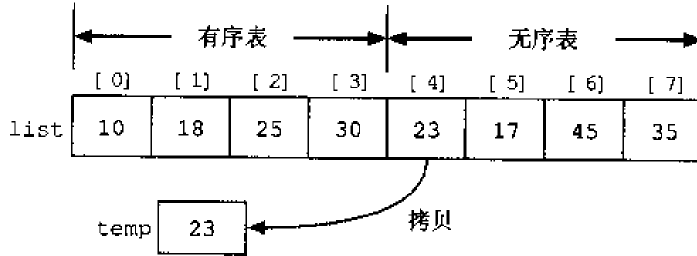


图 18.19 list[4]拷贝到 temp 中

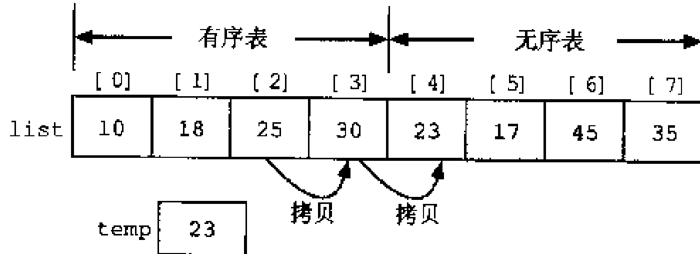


图 18.20 list[3]拷贝到 list[4]中，list[2] 拷贝到 list[3]中

将 list[3]拷贝到 list[4]中，然后将 list[2] 拷贝到 list[3]中后，表如图 18.21 所示。

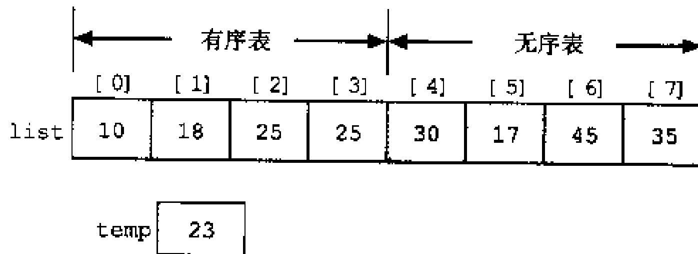


图 18.21 list[3]拷贝到 list[4]中，list[2] 拷贝到 list[3]中后的表

现在，将 temp 拷贝回 list[2]。最后表如图 18.22 所示。

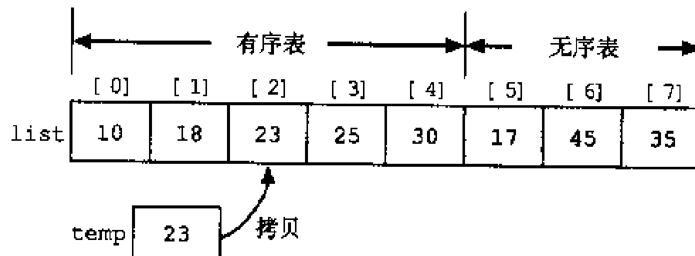


图 18.22 temp 拷贝回 list[2]后的表

现在，表中list[0]…list[4]已经有序，而list[5]…list[7]仍无序。我们在上图所示的表上重复上述过程，将未排序表中的第一个元素移动到有序表的合适位置上。

从上面的讨论中可以清楚地看到，在排序过程中，表被分为两个部分：前部和后部。表前部的元素有序，表后部元素逐一地移到表前部的合适位置上。这里，我们使用一个下标firstOutOfOrder，指向表后部的第一个元素。也就是说，firstOutOfOrder给出了无序表中第一个元素的位置。一开始，firstOutOfOrder被初始化为1。

上面讨论的算法伪码如下所示：

```
for(firstOutOfOrder = 1; firstOutOfOrder < length; firstOutOfOrder++)
  if(list[ firstOutOfOrder] is less than  list[ firstOutOfOrder - 1] )
  {
      copy list[ firstOutOfOrder]  into temp

      initialize location to firstOutOfOrder

      do
      {
           a. move list[ location - 1]  one array slot down
           b. decrement location by 1 to consider the next element of
              the sorted portion of the array
      }
      while(location > 0 && the element in the upper list at
                      location - 1 is greater than temp)
  }

copy temp into list[ location]
```

下面，跟踪该算法在图 18.23 所示表上的执行过程。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list	13	7	15	8	12	30	3	20

图 18.23　无序表

上图中表长度为8，即length = 8。首先，将firstOutOfOrder设置为1（如图 18.24 所示）。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list	13	7	15	8	12	30	3	20

↑ firstOutOfOrder（指向 list[1]）

firstOutOfOrder　1

图 18.24　firstOutOfOrder = 1

现在 list[firstOutOfOrder] = 7，list[firstOutOfOrder−1] = 13，7<13，所以if语句中的条件表达式的值为true，执行该if语句体：

```
temp = list[ firstOutOfOrder]  = 7
location = firstOutOfOrder = 1
```

下面，执行do…while循环。

```
list[1] = list[0] = 13              (将list[0] 拷贝到list[1]中)
location = 0                        (location减1)
```

因为 location = 0，do…while 循环终止。然后将 temp 拷贝到 list[location]，即 list[0]中。结果如图 18.25 所示。

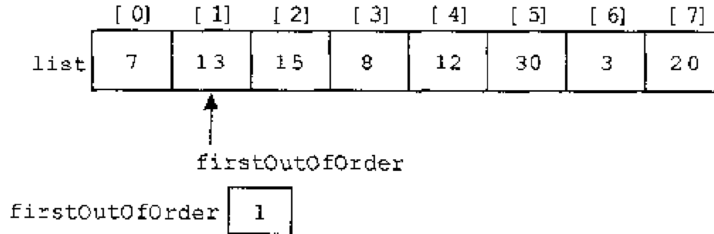


图 18.25 插入排序算法执行第一次迭代后的表

现在，假设有如图 18.26 所示的表。

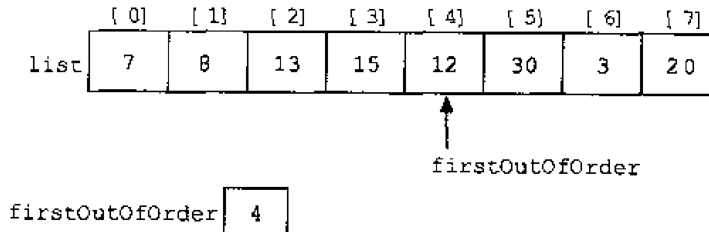


图 18.26 无序表中第一个元素的下标是 4

上面表中，list[0]…list[3]，即元素 list[0]，list[1]，list[2]，list[3]是有序的。所以 firstOutOfOrder = 4。因为 list[4]<list[3]，所以 list[4]（值为 12）要移动到一个合适的位置上。

与前面的做法相同：

```
temp = list[firstOutOfOrder] = 12
location = firstOutOfOrder = 4
```

首先，将 list[3] 拷贝到 list[4]中，并将 location 减 1。然后将 list[2]拷贝到 list[3]中，并将 location 减 1。现在，location 的值是 2。此时，表如图 18.27 所示。

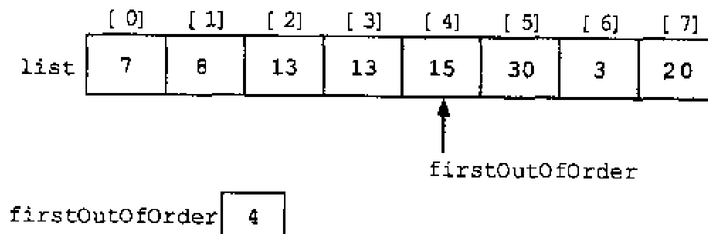


图 18.27 list[3]拷贝到 list[4]中，list[2]拷贝到 list[3]中后的表

因为 list[1]<temp，do…while 循环终止。这时 location = 2，所以将 temp 拷贝到 list[2]中。即：

```
list[2] = temp = 12
```

结果如图 18.28 所示。

上面过程不断重复，将未排序部分的元素归入到排序部分中。

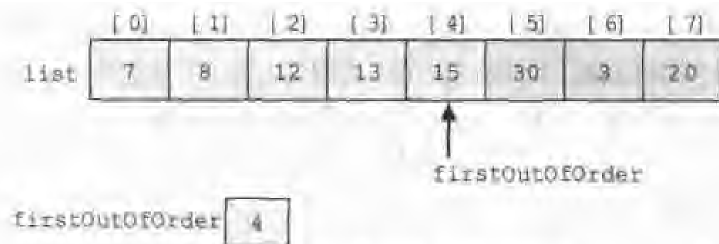


图 18.28　temp 拷贝到 list[2]后的表

下面的 C++ 函数实现了上面的算法：

```
template<class elemType>
void orderedArrayListType<elemType>::insertionSort()
{
   int firstOutOfOrder, location;
   elemType temp;

   for(firstOutOfOrder = 1; firstOutOfOrder < length;
                            firstOutOfOrder++)
      if(list[ firstOutOfOrder] < list[ firstOutOfOrder - 1])
      {
            temp = list[ firstOutOfOrder];
            location = firstOutOfOrder;

            do
            {
                  list[ location] = list[ location - 1];
                  location--;
            }while(location > 0 && list[ location - 1] > temp);

             list[ location] = temp;
      }
} //end insertionSort
```

18.5　插入排序：基于链表

插入排序算法同样也可以用于链表。因此，本节将讨论链表上的插入排序算法。考虑下面的图 18.29。

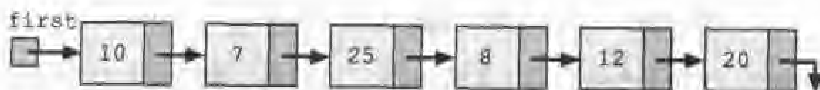


图 18.29　链表

在图 18.29 中，first 是指向链表中第一个元素的指针。

如果表存储在数组中，则可以通过下标在任意方向上移动元素。然而，在如图 18.29 所示的单向链表中，只能从第一个节点开始，向后面一个方向移动元素。因此，在链表中确定插入元素位置的过程如下：假设 firstOutOfOrder 是指向需要插入到有序部分中的节点的指针。lastInOrder 是指向已排序部分最后一个节点的指针，如图 18.30 所示。

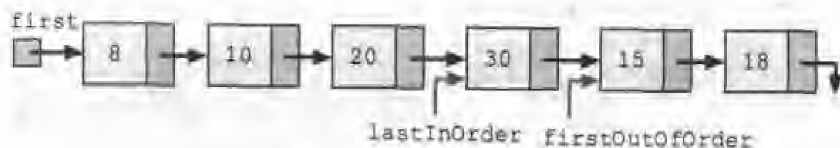


图 18.30　链表和指针 lastInOrder 和 firstOutOfOrder

首先，将 firstOutOfOrder->info 和第一个节点的 info 相比较。如果 firstOutOfOrder->info 小于 first->info，则要将节点 firstOutOfOrder 移到表的头节点前；否则，从第二个节点开始查找 firstOutOfOrder 要插入的节点位置。通常，我们使用两个指针：current 和 trailCurrent 来查找链表。指针 trailCurrent 指向 current 之前的那个节点。当然，还要考虑一些特殊情况，比如空链表，只有一个节点的链表，或者节点 firstOutOfOrder 已经在正确的位置上了。

上面讨论的算法如下所示：

```
if(firstOutOforder->info is less than first->info)
      move firstOutOfOrder before first
else
{
    set trailCurrent to first
    set current to the second node in the list first->link;

     //search the list
    while(current->info is less than firstOutOfOrder->info)
    {
          advance trailCurrent;
          advance current;
    }

    if(current is not equal to firstOutOfOrder)
    {       //insert firstOutOfOrder between current and trailCurrent
       lastInOrder->link = firstOutOfOrder->link;
       firstOutOfOrder->link = current;
       trailCurrent->link = firstOutOfOrder;
    }
    else       //firstOutOfOrder is already at the first place
       lastInOrder = lastInOrder->link;
}
```

用图 18.31 所示的链表来说明上述算法，这里需要考虑多种情况。

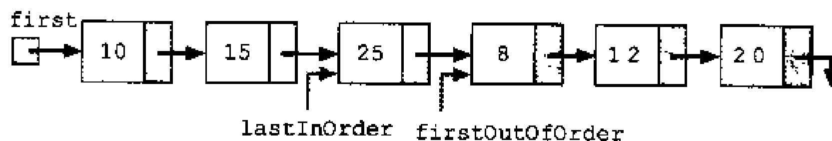


图 18.31 链表及指针 lastInOrder 和 firstOutOfOrder

情况 1 因为 firstOutOfOrder->info 小于 first->info，所以将节点 firstOutOfOrder 移动到 first 之前。结果如图 18.32 所示。

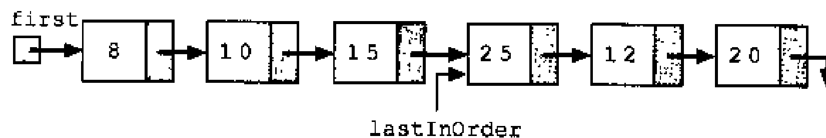


图 18.32 info 为 8 的节点移动到第一个位置上后的链表

情况 2 考虑如图 18.33 所示的链表。

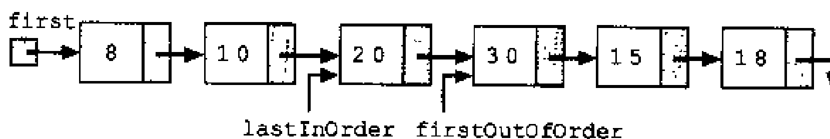


图 18.33 链表及指针 lastInOrder 和 firstOutOfOrder

因为firstOutOfOrder->info大于first->info，所以要在链表的有序部分中找到firstOutOfOrder要被移动到的位置。如前所述，利用指针trailCurrent和current来历遍链表。对图18.33中的链表，这两个指针所指的位置如图18.34所示。

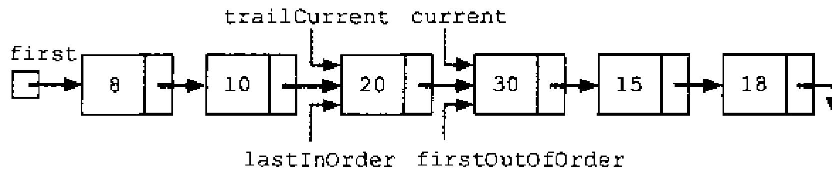


图 18.34 链表及指针 trailCurrent 和 current

因为current和firstOutOfOrder相等，所以节点firstOutOfOrder处于正确位置。链表无须改动，但要将lastInOrder指针指向下一个位置。

情况 3 考虑如图18.35所示的链表。

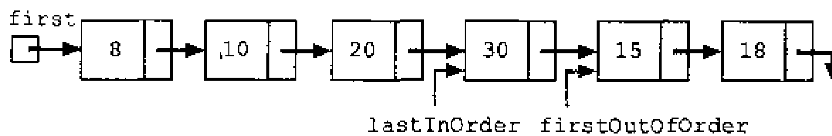


图 18.35 链表及指针 lastInOrder 和 firstOutOfOrder

因为firstOutOfOrder->info大于first->info，所以要在链表的有序部分中查找firstOutOfOrder要被移动到的位置。这里，同样需要使用指针trailCurrent和current。结果如图18.36所示。

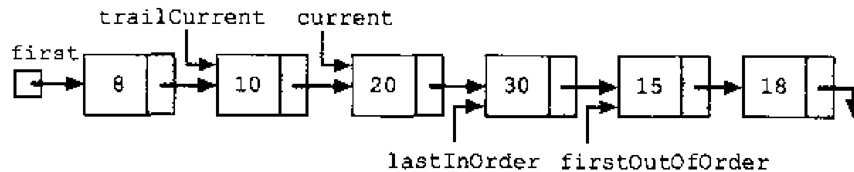


图 18.36 链表及指针 trailCurrent 和 current

很显然，firstOutOfOrder将被移动到trailCurrent和current之间。修改链表，结果如图18.37所示。

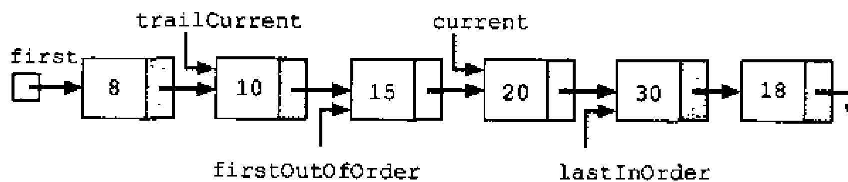


图 18.37 firstOutOfOrder 移动到 trailCurrent 和 current 之间后的链表

现在，编写C++函数，实现上面的算法：

```
template<class elemType>
void orderedLinkedListType<elemType>::linkedInsertionSort()
{
    nodeType<elemType> *lastInOrder;
    nodeType<elemType> *firstOutOfOrder;
    nodeType<elemType> *current;
    nodeType<elemType> *trailCurrent;

    lastInOrder = first;
```

```
    if(first == NULL)
       cout<<"Cannot sort an empty list"<<endl;
    else
       if(first->link == NULL)
          cout<<"List is of length 1. Already in order"<<endl;
       else
          while(lastInOrder->link != NULL)
          {
             firstOutOfOrder = lastInOrder->link;

             if(firstOutOfOrder->info < first->info)
             {
                lastInOrder->link = firstOutOfOrder->link;
                firstOutOfOrder->link = first;
                first = firstOutOfOrder;
             }
             else
             {
                trailCurrent = first;
                current = first->link;

                while(current->info < firstOutOfOrder->info)
                {
                    trailCurrent = current;
                    current = current->link;
                }

                if(current != firstOutOforder)
                {
                   lastInOrder->link = firstOutOfOrder->link;
                   firstOutOfOrder->link = current;
                   trailCurrent->link = firstOutOfOrder;
                }
                else
                   lastInOrder = lastInOrder->link;
        }
     } //end while
} //end linkedInsertionSort
```

插入排序算法的测试程序留给读者作为练习，参见本章编程练习 8 和编程练习 9。

18.5.1 分析：插入排序

可以看出，在插入排序算法中，平均比较次数和元素移动次数为：

$$\frac{1}{4}n^2+O(n)=O(n^2)$$

下表总结了插入排序算法和选择排序算法的部分指标。

对长度为 n 的表，在平均情况下的排序性能

算法	比较次数	交换次数
选择排序	$\frac{n(n-1)}{2}=O(n^2)$	$3(n-1)=O(n)$
插入排序	$\frac{1}{4}n^2+O(n)=O(n^2)$	$\frac{1}{4}n^2+O(n)=O(n^2)$

18.5.2　基于比较的排序算法的下界

前面小节讨论了选择排序算法和插入排序算法，并且指出在平均情况下，它们的复杂度均为$O(n^2)$。这两个算法都被称为基于比较的排序算法，因为它们都需要通过比较相应键来移动表项。在讨论其他的排序算法前，让我们来看看基于比较的排序算法的最佳情况。

利用比较树（Comparison Tree）图，可以清楚地看到基于比较的排序算法的执行过程。设L是一个有n个不同元素的表，这里n>0。对任意的j和k，$1 \leq j, k \leq n$，$j \neq k$，$L[j]<L[k]$或者$L[j]>L[k]$。因为每次比较都有两个可能结果，所以比较树是一个二叉树。在图中，把每次比较表示成为一个圆圈，称为节点（node）。节点标注为：$j:k$，表示$L[j]$和$L[k]$相比较。如果$L[j]<L[k]$，沿着左子树向下进行比较；否则，沿着右子树向下进行比较。图 18.38 是一个长度为 3 的表的比较树［在图 18.38 中，矩形框称为叶（leaf），代表了节点的最后顺序］。

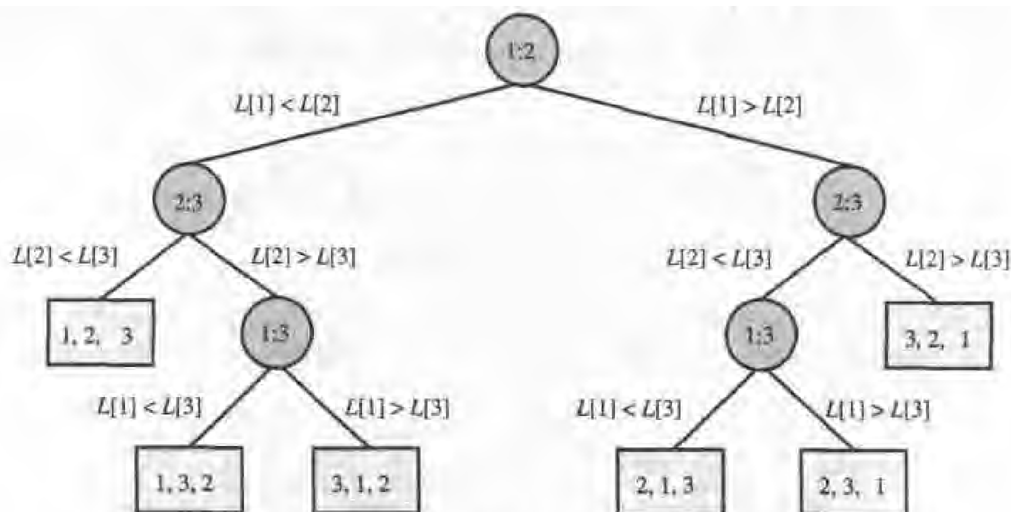


图 18.38　长度为 3 的表的比较树

图中顶部的节点称为根节点（root），连接两个节点之间的直线称为分支（branch），从节点x到节点y的一系列分支称为从x到y的路径（path）。

和从根节点到某个叶节点的路径相对应的是L中元素的一个惟一排列。之所以惟一，是因为排序算法仅仅比较和移动数据。进一步说，无论初始输入是什么，从根节点到叶节点的任何一条路径上的数据移动都是相同的。对一个有n个元素的表，其中n>0，共有$n!$种排列方式。其中，只有一种是L的正确排序序列。因此，比较树有$n!$个叶节点。

对于基于比较的排序算法的最差情况，我们不加证明给出如下结论：

定理　令L是一个有n个不同元素的表。任何通过比较键进行排序的算法，在最差情况下，至少进行$O(n*\log_2 n)$次比较。

正如前面分析的一样，选择和插入排序算法复杂度都是$O(n^2)$。本章剩余部分，将讨论平均复杂度为$O(n*\log_2 n)$的排序算法。

18.6　快速排序：基于数组表

上节指出，基于比较的排序算法复杂度的下界是$O(n*\log_2 n)$。选择和插入排序算法的复杂度都是$O(n^2)$。下面，将给出两种复杂度为$O(n*\log_2 n)$的算法。第一种叫做快速排序算法。

快速排序算法采用“拆分”技术来对表进行排序。表被分成两个子表，分别排序后再合并到一起，成为一个有序表。通常算法描述如下所示：

```
if(表长度大于1)
{
    a.把表分成两个子表，称为下子表(lowerSublist)和上子表(upperSublist)。
    b.对下子表进行快速排序。
    c.对上子表进行快速排序。
    d.将下子表和上子表合并成一个表。
}
```

这里的排序算法是基于数组的表，基于链表的快速排序算法与之类似，留给读者作为练习。见本章后面的编程练习11。

在快速排序中，合并子表(lowerSublist和upperSublist)与表的拆分相比要简单得多。因此，在快速排序中，所有的排序工作都集中在拆分表中。因为排序工作在表拆分过程中就已完成，所以首先详细描述表的拆分过程。

为了把表拆分成两个子表，先从表中选取一个元素，称为支点(pivot)。然后，对表中的元素重新排列，使得lowerSublist中的元素都小于该支点，而upperSublist中的元素都大于该支点。考虑如图18.39所示的表。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
list	45	82	25	94	50	60	78	32	92

图18.39 拆分前的表

选取支点的方法有多种。但是，选取支点要尽量使lowerSublist和upperSublist的大小相同。因此，我们选取表的中间元素作为支点。在本例中，中间元素为50，将其作为支点，如图18.40所示。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
list	32	25	45	50	82	60	78	94	92
	lowerSublist				upperSublist				

图18.40 拆分后的表

从图18.40可以看出，将表拆分成lowerSublist和upperSublist后，支点处于正确的位置上。这样，将lowerSublist和upperSublist分别排序后，合并就很容易了。

拆分算法如下所示：

1. 确定pivot，将其和表的第一个元素交换。假设下标smallIndex表示小于pivot的最后一个元素的下标，则将其初始化为表中的第一个元素。
2. 对表中的剩余元素(从第二个元素开始)：
   如果当前元素小于pivot，则：
   a. 将smallIndex加1
   b. 将smallIndex所指元素和当前元素交换
3. 将第一个元素，即pivot和smallIndex所指元素交换

步骤2可以用一个for语句来实现，循环从表中第二个元素开始。

用图18.41所示表使用上述算法。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
list	45	82	25	94	50	60	78	32	92

图18.41 使用排序算法前的表

1. a. pivot = 50

   b. swap(list[0], list[4]);

   c. smallIndex = 0;

2.1. index = 1（使用 for 循环控制变量），如图 18.42 所示。

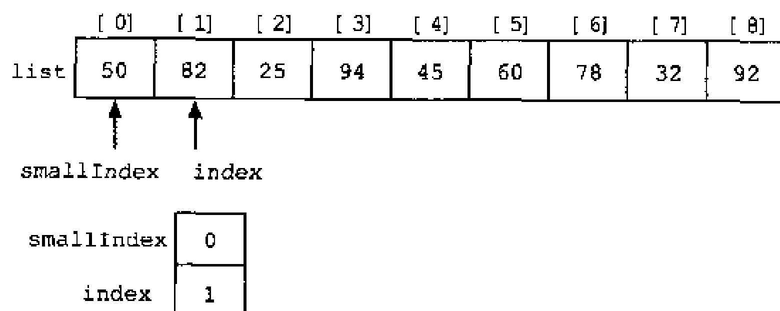


图 18.42　list，smallIndex 和 index

因为 list[index] > pivot，if 语句失败，进入 for 的下一次循环。

2.2. index = 2，如图 18.43 所示。

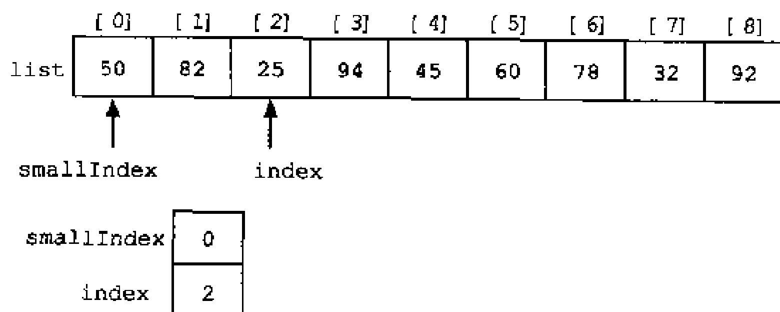


图 18.43　第 2 次循环前的表

因为 list[index] = 25 < pivot = 50，if 条件表达式的值为 true，smallIndex 加 1，即 smallIndex = 1 接下来，执行 swap(list[smallIndex], list[index])，即 swap(list[1], list[2])，结果如图 18.44 所示。

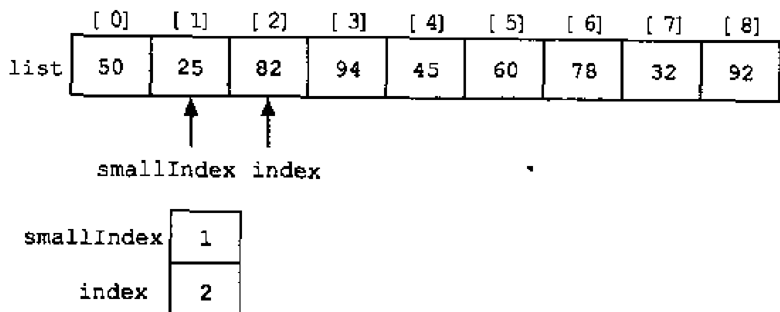


图 18.44　第 2 次循环后的表

2.3. index = 3，如图 18.45 所示。

因为 list[index] > pivot，if 语句失败。进入 for 语句的下一次循环。第 3 次循环后，表如图 18.45 所示。

2.4. index = 4，如图 18.46 所示。

因为 list[index] = 45 < pivot = 50，if 语句的条件表达式的值为 true，smallIndex 加 1，值为 2。接着，执行 swap(list[smallIndex], list[index])，即 swap(list[2], list[4])，表如图 18.47 所示。

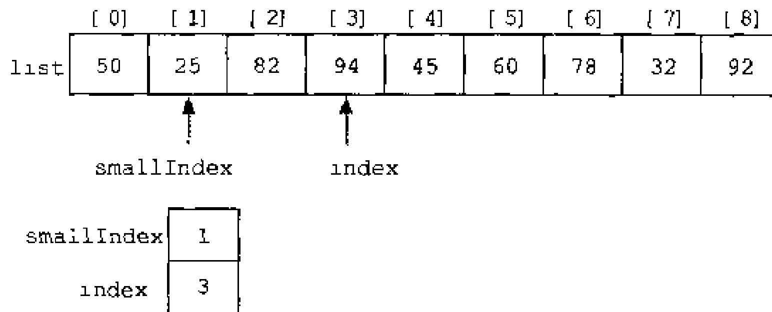


图 18.45 第3次循环前的表

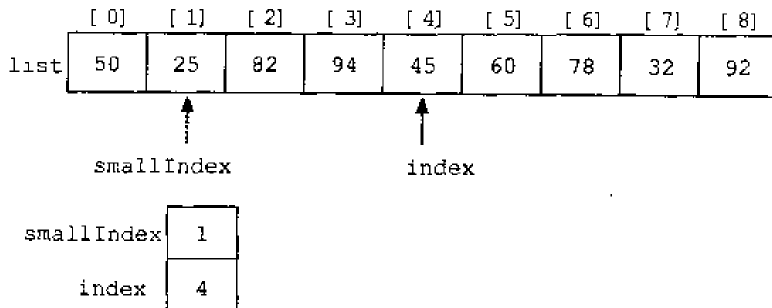


图 18.46 第4次循环前的表

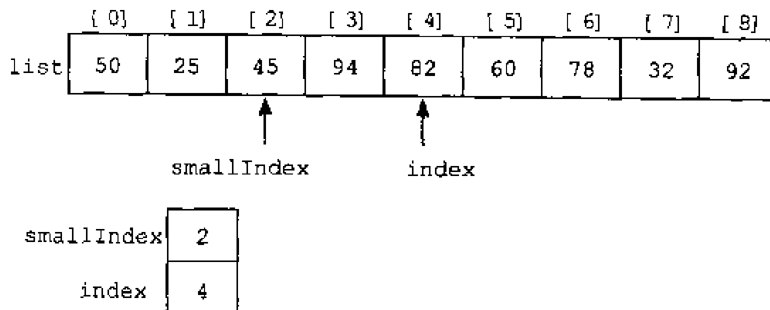


图 18.47 第4次循环后的表

2.5. index = 5，如图 18.48 所示。

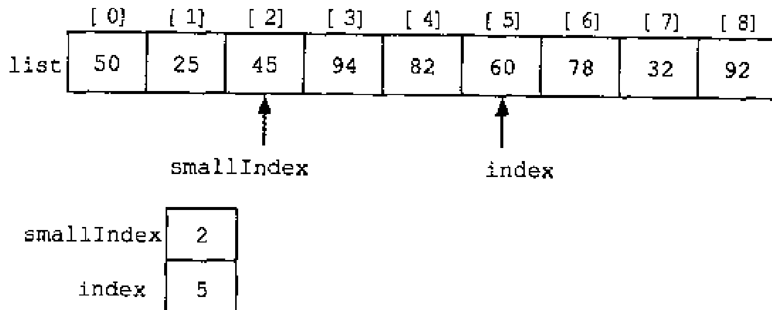


图 18.48 第5次循环前的表

因为 list[index] > pivot，if 语句失败。进入 for 语句的下一次循环。第5次循环后，表仍如图 18.48 所示。

2.6. index = 6，如图 18.49 所示。

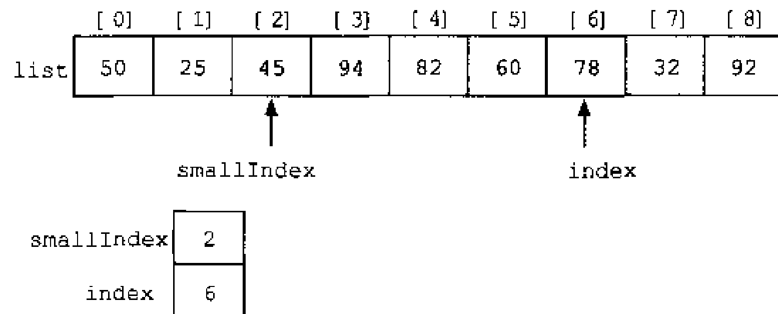


图 18.49　第 6 次循环前的表

因为 list[index] > pivot，if 语句失败。进入 for 语句的下一次循环。第 6 次循环后，表仍如图 18.49 所示。

2.7. index = 7，如图 18.50 所示。

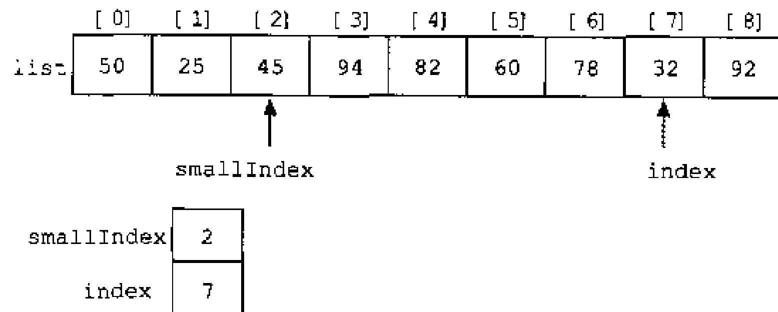


图 18.50　第 7 次循环前的表

因为 list[index] = 32 < pivot = 50，if 语句的条件表达式值为 true，smallIndex 加 1，值为 3。接着，执行 swap(list[smallIndex], list[index])，即 swap(list[3], list[7])。结果如图 18.51 所示。

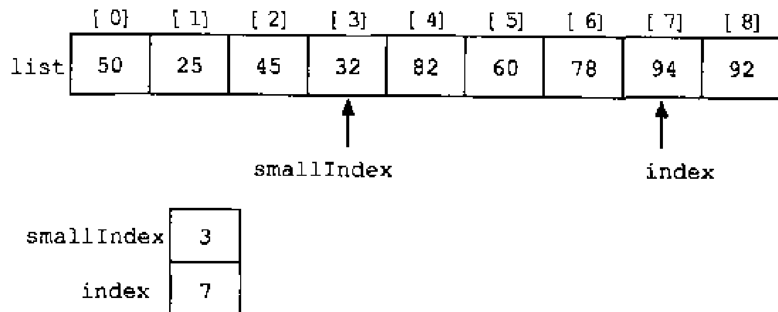


图 18.51　第 7 次循环以后的表

2.8. index = 8，如图 18.52 所示。

因为 list[index] > pivot，if 语句失败。进入 for 语句的下一次循环。第 8 次循环后，表仍如图 18.52 所示。

第 8 次循环后，for 语句循环终止。

3. 将 list[smallIndex]和 list[0]相交换，结果如图 18.53 所示。

现在，编写函数 partition，实现上述的拆分算法。在重新安排表中元素之后，函数返回 pivot 的位置，使得可以确定子表的开始位置和结束位置。由于该函数是类的成员函数，它可以直接访问数组表。这样，在拆分表时，只需要传递表的开头元素和结尾元素下标给函数就可以了。

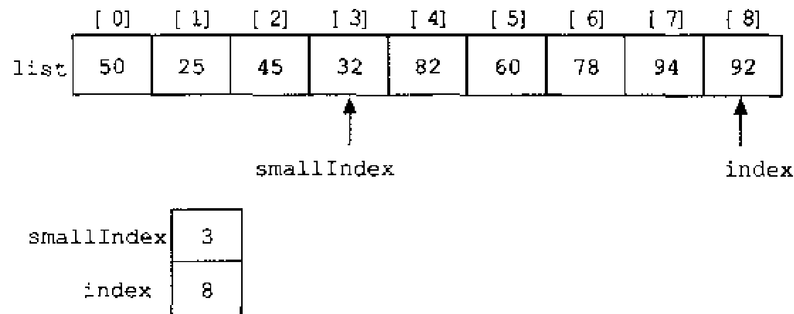


图 18.52 第8次循环前的表

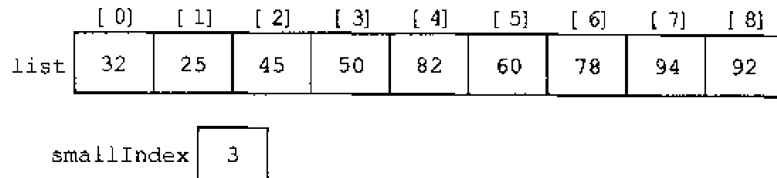


图 18.53 执行步骤3以后的表

```
template<class elemType>
int orderedArrayListType<elemType>::partition(int first, int last)
{
       elemType pivot;

       int index, smallIndex;

       swap(first, (first + last) / 2);

       pivot = list[ first];
       smallIndex = first;

       for(index = first + 1; index <= last; index++)
           if(list[ index] < pivot)
           {
               smallIndex++;
               swap(smallIndex, index);
           }

       swap(first, smallIndex);

       return smallIndex;
}
```

正如从函数partition的定义中看到的一样，表中的一些元素需要交换。下面函数swap完成这个功能:

```
template<class elemType>
void orderedArrayListType<elemType>::swap(int first, int second)
{
       elemType temp;

       temp = list[ first];
       list[ first] = list[ second];
       list[ second] = temp;
}
```

一旦表被拆分成lowerSublist和upperSublist，则需要再次利用快速排序算法来对子表进行排序。两个子表都要进行排序，最简单的方法是利用递归。实际上，本节给出的算法就是递归算法。如前面

所述，在表中元素重新排列以后，函数partition返回pivot的下标，以确定子表的开始元素和结束元素下标。

给出一个表的开始元素和结束元素下标，下面的recQuickSort实现了快速排序的递归算法：

```
template<class elemType>
void orderedArrayListType<elemType>::recQuickSort(int first, int last)
{
      int pivotLocation;

      if(first < last)
      {
         pivotLocation = partition(first, last);
         recQuickSort(first, pivotLocation - 1);
         recQuickSort(pivotLocation + 1, last);
      }
}
```

最后，编写快速排序函数：quickSort，其对初始表调用recQuickSort函数：

```
template<class elemType>
void orderedArrayListType<elemType>::quickSort()
{
      recQuickSort(0, length - 1);
}
```

快速排序算法的测试程序作为练习留给读者。参见本章后面的编程练习10。

18.6.1　分析：快速排序

下表总结了长度为 n 的表的快速排序算法的性能（本书不做证明）。

长度为 n 的表的快速排序算法分析

	比较次数	交换次数
平均情况	$(1.39)n*\log_2 n+O(n)=O(n*\log_2 n)$	$(0.69)*n*\log_2 n+O(n)=O(n*\log_2 n)$
最坏情况	$\frac{n^2}{2}-\frac{n}{2}=O(n^2)$	$\frac{n^2}{2}+\frac{3n}{2}-2=O(n^2)$

18.7　归并排序：基于链表

前面部分讨论了快速排序算法，并且指出在平均情况下快速排序算法的复杂度为$O(n*\log_2 n)$。但是，快速排序在最差情况下的复杂度也为 $O(n^2)$。本节将介绍一个复杂度总是 $O(n*\log_2 n)$的排序算法。

和快速排序算法一样，归并排序算法同样基于“拆分”技术。归并算法也将一个表分成两部分，并分别对子表进行排序，然后将其合并成为一个有序表。本节将介绍归并算法在链表上的应用。也可以采用同样的方法，对于基于数组的表进行归并排序，我们将其作为作业留给读者。

归并排序和快速排序的区别在于如何拆分表。如前所述，快速排序选取表中的一个元素，称为支点（pivot）。然后用它来拆分表，使得其中的一个子表元素都小于该支点，而另一个子表中的元素都大于该支点。而归并排序则将表拆分成两个大小几乎相同的子表。例如，对下面的表：

表：35　28　18　45　62　48　30　38

归并算法将其拆分成：

第一个子表：35　28　18　45
第二个子表：62　48　30　38

然后，使用同样的归并排序算法对这两个子表进行排序。假设已对子表进行了排序，即：

第一个子表：18 28 35 45
第二个子表：30 38 48 62

接下来，归并算法合并子表，把两个子表归并成一个表。

图 18.54 说明了归并算法的全过程。

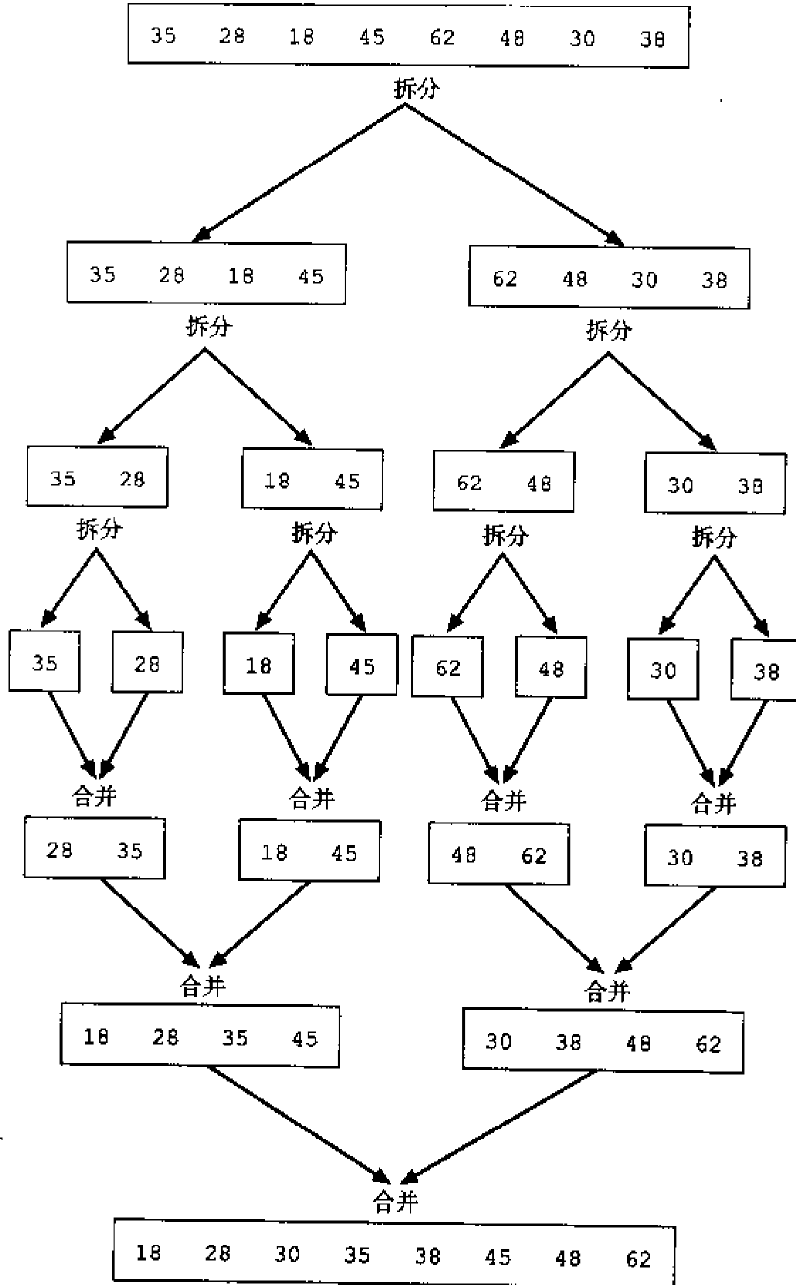


图 18.54 归并排序算法

从图 18.54 中可以看出，归并排序算法的大部分工作集中在归并有序子表上。

归并算法描述如下所示：

```
if 表的长度大于1
{
        a.将表拆分成两个子表
```

```
      b.对第一个子表进行归并排序
      c.对第二个子表进行归并排序
      d.将第一个子表和第二个子表归并
}
```

归并算法可以总结为：

1. 将表拆分成两个大小差不多的子表
2. 对子表归并排序
3. 将排序过的子表合并

18.7.1 拆分

因为数据存储在链表中，而我们事先并不知道链表的长度。并且，链表并非是一个可以任意访问的数据结构。因此，为了将链表拆成两个大小差不多的子表，首先要确定中间节点。

考虑图 18.55 中的链表。

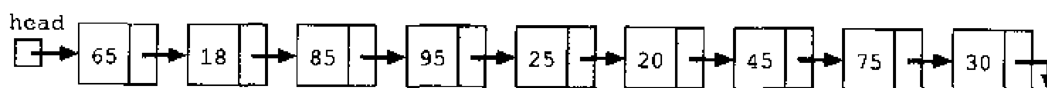


图 18.55　未排序的链表

为了确定中间节点，需要使用两个指针——middle 和 current 来历遍链表。指针 middle 被初始化为链表的头节点。因为该链表超过了两个节点，所以将 current 指向链表的第三个节点（我们只需要对长度大于 1 的表进行排序，因为长度等于 1 的链表相当于已经排过序。另外，如果链表仅有两个节点，只需将 current 设置为 NULL）。见图 18.56 中的链表。

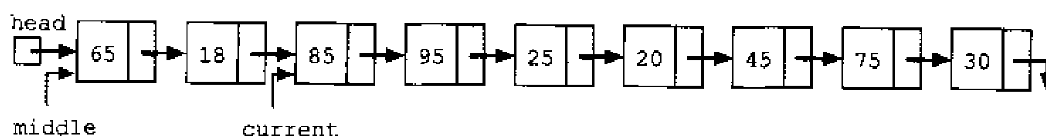


图 18.56　历遍链表前的 middle 和 current 指针

在将 middle 前移一个位置的同时，也要将 current 前移一个位置。当 current 前移后，如果它不等于 NULL，则要将其再次前移。在大多数情况下，将 middle 前移一个位置，就需要将 current 前移两个位置。最终，current 等于 NULL，middle 指向第一个子表的最后一个节点。对于图 18.56 中的链表，当 current 变成 NULL 时，middle 指向 info 为 25 的节点（见图 18.57）。

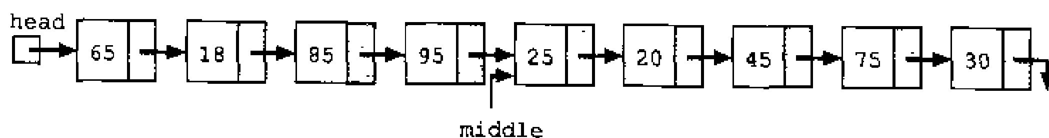


图 18.57　历遍链表后的 middle 指针

现在要把链表拆分开就很容易了。首先，使用一个新指针指向 middle 所指的节点，然后将 middle 指向的节点的 link 域改为 NULL。结果如图 18.58 所示。

由上面讨论可以编写一个 C++ 函数 divideList：

```
template<class elemType>
void orderedLinkedListType<elemType>::divideList(
                                        nodeType<elemType>* first1,
                                        nodeType<elemType>* &first2)
{
   nodeType<elemType>* middle;
```

```
    nodeType<elemType>* current;

    if(first1 == NULL)   //list is empty
       first2 = NULL;
    else
       if(first1->link == NULL)     //list has only one node
          first2 = NULL;
       else
       {
          middle = first1;
          current = first1->link;
          if(current != NULL)        //list has more than two nodes
             current = current->link;

          while(current != NULL)
          {
             middle = middle->link;
             current = current->link;
             if(current != NULL)
                current = current->link;
          } //end while

          first2 = middle->link;  //first2 points to the first
                                  //node of the second sublist
          middle->link = NULL;    //set the link of the last node
                                  //of the first sublist to NULL
       } //end else
} //end divide
```

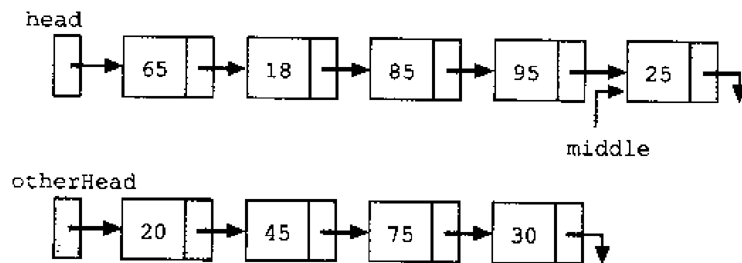


图 18.58 链表被拆分成两个子表

18.7.2 归并

在将子表排序之后，就可以将子表合并成一个链表了。合并的方法是比较子表中的元素，然后调整info较小的节点的指针。利用图18.59中的子表来说明这个过程。假设first1指针指向第一个子表的头节点，first2指针指向第二个子表的头节点。

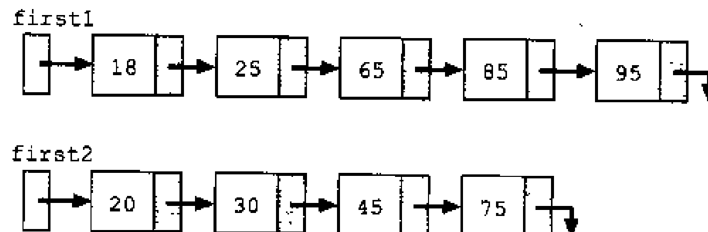


图 18.59 归并前的子表

首先，将两个子表的第一个位置上的info相比较，以确定归并后的链表的第一个元素。使用一个新指针 newHead，指向归并后的链表的头节点。再使用另一个指针 lastSmall，指向归并后链表的最后一个节点。头节点较小的那个子表的第一个位置上指针要后移一个节点。图18.59中的链表在设置了newHead和 lastSmall 并后移 first1 以后的情形如图 18.60 所示。

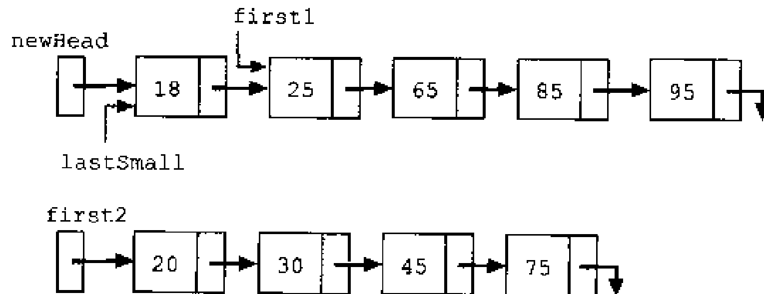


图 18.60　设置 newHead 和 lastSmall 并且后移 first1 以后的链表

在图 18.60 中，first1 指向要和第二个子表归并的第一个子表的头节点。所以将 first1 和 first2 指向的节点相比较，然后调整较小节点的 link，并将其移到归并链表的最后一个位置上。对于图 18.60 中的子表，调整后如图 18.61 所示。

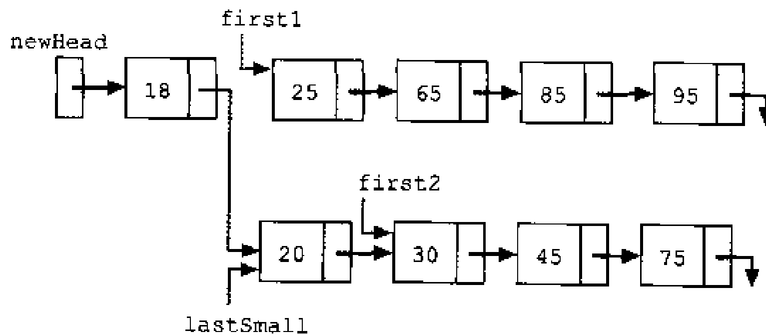


图 18.61　将 info 为 20 的节点移到归并链表的最后一个位置上

我们对余下的节点继续这个过程。每次将一个节点移动到归并链表的最后一个位置上，把 first1 或 first2 分别下移一个节点。最终，它们必有一个指向 NULL。如果是 first1 指向了 NULL，说明第一个子表已经没有元素了，所以将第二个子表的余下部分直接链接到归并链表的最后一个元素上。如果是first2 指向了NULL，说明第二个子表已经没有元素了，所以将第一个子表的余下部分直接链接到归并链表的最后一个元素上。

根据上面讨论，编写相应的 C++ 函数 mergeList。子表的头指针作为参数传递给该函数：

```
template<class elemType>
nodeType<elemType>* orderedLinkedListType<elemType>::mergeList
      (nodeType<elemType>* first1, nodeType<elemType>* first2)
{
   nodeType<elemType> *lastSmall; //pointer to the last node of
                                  //the merged list
   nodeType<elemType> *newHead;   //pointer to the merged list

   if(first1 == NULL)    //the first sublist is empty
      return first2;
   else
      if(first2 == NULL)    //the second sublist is empty
            return first1;
```

```
    else
    {
       if(first1->info < first2->info) //compare the first nodes
       {
          newHead = first1;
          first1 = first1->link;
          lastSmall = newHead;
       }
       else

          {
                newHead = first2;
                first2 = first2->link;
                lastSmall = newHead;
          }

       while(first1 != NULL && first2 != NULL)
       {
          if(first1->info < first2->info)
          {
                lastSmall->link = first1;
                lastSmall = lastSmall->link;
                first1 = first1->link;
          }
         else
         {
               lastSmall->link = first2;
               lastSmall = lastSmall->link;
               first2 = first2->link;
         }
      } //end while

      if(first1 == NULL)      //first sublist is exhausted first
         lastSmall->link = first2;
      else                    //second sublist is exhausted first
         lastSmall->link = first1;

      return newHead;

} //end mergeList
```

最后，我们给出递归函数recMergeSort，它调用了函数divide和mergeList对链表排序。第一个元素的指针作为参数传递给该函数：

```
template<class elemType>
void orderedLinkedListType<elemType>::recMergeSort(
                                    nodeType<elemType>* &head)
{
      nodeType<elemType> *otherHead;

      if(head != NULL)  //if the list is not empty
         if(head->link != NULL)  //if the list has more than one node
         {
            divideList(head, otherHead);
            recMergeSort(head);
            recMergeSort(otherHead);
            head = mergeList(head, otherHead);
```

```
        }
} //end recMergeSort
```

现在，我们可以给出函数 mergeSort，它同时也是类 orderedLinkedListType 的一个 public 成员函数。注意，函数 divideList，mergeList 和 recMergeSort 都是类 orderedLinkedListType 的 private 成员函数，因为它们只被 mergeSort 所调用。函数 mergeSort 只是调用 recMergeSort，first 指针作为参数传递给该函数。mergeSort 函数如下所示：

```
template<class elemType>
void orderedLinkedListType<elemType>::mergeSort()
{
      recMergeSort(first);
} //end mergeSort
```

如果将实现归并排序算法的函数添加到类 orderedLinkedListType 中，则类定义如下所示：

```
template<class elemType>
class orderedLinkedListType: public linkedListType<elemType>
{
public:
    void search(const elemType& 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 elemType& 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 elemType& 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;
      //Prints the list in the reverse order.
      //Because the original list is in ascending order, the
      //elements will be printed in descending order.

    void linkedInsertionSort();
    void mergeSort();

private:
    void reversePrint(nodeType<elemType> *current) const;
      //This function is called by the public member
      //function to print the list in the reverse order.

    void divideList(nodeType<elemType>* first1,
                    nodeType<elemType>* &first2);
    nodeType<elemType>* mergeList(nodeType<elemType>* first1,
                                  nodeType<elemType>* first2);
    void recMergeSort(nodeType<elemType>* &head);
};
```

我们将测试归并排序算法函数的程序留给读者作为练习。参见本章后面的编程练习 14。

18.7.3　分析：归并排序

假设 L 是一个有 n 个元素的链表，其中 $n>0$。令 $A(n)$ 为平均情况下的比较次数，$W(n)$ 为最坏情况下的比较次数，则它们如下所示：

$A(n) = n*\log_2 n - 1.26 = O(n*\log_2 n)$

$W(n) = n*\log_2 n - (n-1) = O(n*\log_2 n)$

18.8 程序范例：选举结果统计

某所大学即将选举出新一任学生会主席。为了保密，选举委员会主席希望使用计算机来统计选票。现需要一个统计选票并找出获胜者的程序。下面编写一个程序来帮助选举委员会。

这所大学分为4个区，每个区都有若干个系。选举时，4个区分别标注为选区1、选区2、选区3和选区4。选区中的每个系独立进行选举，最后将每个候选人获得的票数统计报告给选举委员会。选票报告格式如下所示：

```
firstName  lastName  regionNumber  numberOfVotes
```

选举委员会需要输出如下形式的表格：

```
                -----------Election Results-----------

                                                  Votes
    Candidate Name         Region1  Region2  Region3  Region4   Total
---------------------      -------  -------  -------  -------  ------
Buddy       Balto                0        0        0      272     272
Doctor      Doc                 25       71      156       97     349
Ducky       Donald             110      158        0        0     268

.
.
.

Winner: ???,  Votes Received: ???
Total votes polled: ???
```

候选人的名字必须按照字母顺序输出。

在该程序中，假设共计有6个候选人。当然，稍加修改，该程序就可以处理任意数目的候选人。

数据分别装在两个文件中。一个是candData.txt，由候选人的名字构成，未排序。另一个是voteData.txt，每一条记录的形式如下所示：

```
firstName  lastName  regionNumber  numberOfVotes
```

也就是说，voteData.txt文件的每条记录都是由候选人姓名、选区编号和候选人得票数组成。其中，每行只有一条记录。输入文件中的样例数据如下所示：

```
Ducky Donald 1 23
Peter Pluto 2 56
Doctor Doc 1 25
Peter Pluto 4 23
.
.
.
```

第一行数据说明：Ducky Donald 从选区1获得了23张选票。

输入 两个文件，一个包含候选人姓名，另一个包含上面所述的选举数据。

输出 将选举结果按上面所述的形式输出，同时输出获胜者。

问题分析和算法设计

从输出表格可以看出，程序要以选区来排列选票数据，并且计算出每个候选人的选票数和总选票数。另外，候选人的姓名要以字母顺序排序。

本程序的主要部分是处理候选人数据，所以首先设计一个类 candidateType 来表示候选人对象。每个候选人有一个名字和所得选票。因为有 4 个区，所以可以利用有 4 个元素的数组。例 12.9（第 12 章）中，我们设计了类 personType 来表示一个人的名字。类 personType 对象可以存储人的姓名。接下来，讨论运算符重载。我们设计类 personType，并且重载相应的运算符，使得两个人的名字可以比较。同样，还可以重载赋值运算符，使姓名赋值更加容易。另外，重载输入输出流运算符，可以方便输入输出。因为每个候选人都是一个人，所以可以从类 personType 来派生类 candidateType。

类 personType　类 personType 包括了一个人的姓名数据。因此类 personType 有两个数据成员：firstName 表示名，lastName 表示姓。它们被声明为 protected 数据成员，以适应扩充性的需要。类 personType 的定义如下所示（参见图 18.62）：

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

class personType
{
    friend istream& operator>>(istream&, personType&);
    friend ostream& operator<<(ostream&, const personType&);
public:
    const personType& operator=(const personType&);
      //overload the assignment operator

    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;

      //overload the relational operators
    bool operator==(const personType& right) const;
    bool operator!=(const personType& right) const;
    bool operator<=(const personType& right) const;
    bool operator<(const personType& right) const;
    bool operator>=(const personType& right) const;
    bool operator>(const personType& right) const;

protected:
    string firstName; //store the first name
    string lastName;  //store the last name
};
```

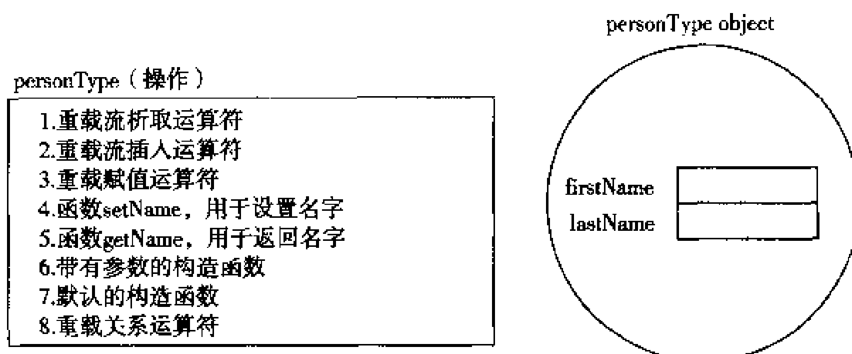


图 18.62　类 personType 中的操作和一个 personType 类对象

下面，给出了类 personType 中函数的定义：

```
void personType::setName(string first, string last)
{
      firstName = first;
      lastName = last;
}

void personType::getName(string& first, string& last)
{
      first = firstName;
      last = lastName;
}

     //constructor
personType::personType(string first, string last)

{
      firstName = first;
      lastName = last;
}

const personType& personType::operator=(const personType& right)
{
      if(this != &right)
      {
         firstName = right.firstName;
         lastName = right.lastName;
      }

      return *this;
}
```

如果两个人的 firstName 和 lastName 都相同，则名字相等。因此，运算符 "==" 重载如下所示：

```
bool personType::operator==(const personType& right) const
{

   return(firstName == right.firstName
          &&  lastName == right.lastName);
}
```

如果两个人的 firstName 或者 lastName 不同，则名字不等。因此，运算符 "!=" 重载如下所示：

```
bool personType::operator!=(const personType& right) const
```

```
{
    return(firstName != right.firstName
           || lastName != right.lastName);
}
```

类似地，其他运算符重载如下所示：

```
bool personType::operator<=(const personType& right) const
{
    return(lastName <= right.lastName ||
           (lastName == right.lastName &&
            firstName <= right.firstName));
}

bool personType::operator<(const personType& right) const
{
    return(lastName < right.lastName ||
           (lastName == right.lastName &&
            firstName < right.firstName));
}

bool personType::operator>=(const personType& right) const
{
    return(lastName >= right.lastName ||
           (lastName == right.lastName &&
            firstName >= right.firstName));
}

bool personType::operator>(const personType& right) const
{
    return(lastName > right.lastName ||
           (lastName == right.lastName &&
            firstName > right.firstName));
}
```

析取和插入运算符重载如下所示：

```
istream& operator>>(istream& isObject, personType& pName)
{
      isObject>>pName.firstName>>pName.lastName;

      return isObject;
}

ostream& operator<<(ostream& osObject, const personType& pName)
{
       osObject<<pName.firstName<<" "<<pName.lastName;

       return osObject;
}
```

类 candidateType　程序的主体部分是候选人，因此本节将主要讨论这个问题。每个候选人有一个姓名及相应所得选票。因为共有4个选区，所以要定义一个有4个元素的数组来记录每个选区的选票。需要一个数据成员来记录每个候选人的得票。可以从类personType派生出candidateType类。由于在类personType中数据成员是受保护的（protected），所以可以在类candidateType中直接访问。

一共有6个候选人。因此，要声明一个可以存储6个候选人的candidateType类型表。在本章中，我们已经定义了类orderedArrayListType来表示一个有序表。所以可以利用这个类来对候选人表进行操作。

由于需要在候选人表上进行排序和查找，因此需要在类candidateType上定义赋值运算和其他相关操作，这些操作用于查找表和对表进行排序。

包含候选人信息的文件中只有候选人姓名。因此，在类candidateType中，除了重载赋值运算符，使其可以将一个对象的值赋给另一个对象以外，还要为类candidateType重载赋值运算符，使得候选人的姓名（类personType对象）可以直接赋值给一个candidateType类的对象。也就是说，在类candidateType中，要重载赋值运算符两次：一个是针对类candidateType的对象，另一个是针对类candidateType和类personType的对象（如图18.63和图18.64所示）。

```
#include <string>
#include "personType.h"

using namespace std;

const int noOfRegions = 4;

class candidateType: public personType
{
public:
    const candidateType& operator=(const candidateType&);
      //Overload the assignment operator for objects of the
      //type candidateType.

    const candidateType& operator=(const personType&);
      //Overload the assignment operator for objects so that
      //the value of an object of the type personType can be
      //assigned to an object of the type candidateType.

    void setName(string first, string last);
      //Function to set the candidate name according to
      //the parameters.

    void getName(string& first, string& last);
      //Function to return the first name and the last name.

    void updateVotesByRegion(int region, int votes);
      //Function to update the votes of a candidate for a
      //particular region.

    void setVotes(int region, int votes);
      //Function to set the votes of a candidate for a
      //particular region.

    void calculateTotalVotes();
      //Function to calculate the total votes received by a
      //candidate. This function adds the votes received in
      //each region.

    int getTotalVotes();
      //Function to return the total votes received by
      //the candidate.

    void printData() const;
      //Function to output the candidate's name, the votes
      //received in each region, and the total votes received.
```

```
    candidateType();
      //Default constructor to initialize the votes received
      //in each region and the total votes received to zero.
      //overload the relational operators
    bool operator==(const candidateType& right) const;
    bool operator!=(const candidateType& right) const;
    bool operator<=(const candidateType& right) const;
    bool operator<(const candidateType& right) const;
    bool operator>=(const candidateType& right) const;
    bool operator>(const candidateType& right) const;

private:
    int votesByRegion[noOfRegions];
    int totalVotes;
};
```

candidateType（操作）

1. 重载赋值运算符
2. 函数setName，用于设置名字
3. 函数getName，用于返回名字
4. 函数updateVotesByRegion，用于更新某一选区的选票
5. 函数setVotes，用于设置某一选区的选票
6. 函数calculateTotalVotes，用于计算某一候选人的总得票数
7. 函数getTotalVotes，用来返回某一候选人的总得票数
8. 函数printData，用于输出相应数据
9. 默认的构造函数
10. 重载关系运算符

图 18.63　类 candidateType 中的操作

candidateType对象

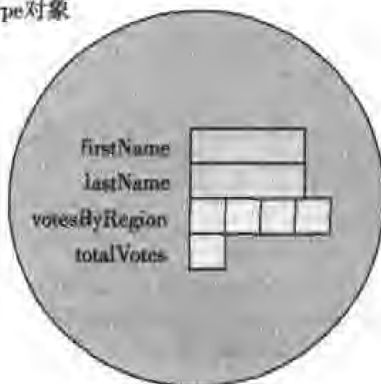


图 18.64　类 candidateType 的对象

类 candidateType 中的成员函数定义如下。

函数 setName 根据参数来设置候选人的名字。其定义如下所示：

```
void candidateType::setName(string first, string last)
{
      firstName = first;
      lastName = last;
}
```

为了设置一个选区的票数，函数setVotes把选区号和选票的数目作为参数。因为数组下标从0开始，选区1对应于数组中下标为0的元素，以此类推。因此，要设置正确的数组元素的值，要将选区号减1。函数setVotes的定义如下所示：

```
void candidateType::setVotes(int region, int votes)
{
      votesByRegion[ region - 1] = votes;
}
```

要增加一个选区的选票时，函数updateVotesByRegion将选区号和新增票数作为参数，并在原来的选票数上加上新增票数：

```
void candidateType::updateVotesByRegion(int region, int votes)
{
      votesByRegion[ region - 1] = votesByRegion[ region - 1]
                                   + votes;
}
```

下面直接给出了函数calculateTotalVotes，getTotalVotes，printData和getName的定义：

```
void candidateType::calculateTotalVotes()
{
      int i;

      totalVotes = 0;

      for(i = 0; i < noOfRegions; i++)
          totalVotes += votesByRegion[ i] ;
}

int candidateType::getTotalVotes()
{
     return totalVotes;
}

void candidateType::printData() const
{
      cout<<left
          <<setw(10)<<firstName<<" "
          <<setw(10)<<lastName<<" ";

      cout<<right;

      for(int i = 0; i < noOfRegions; i++)
            cout<<setw(8)<<votesByRegion[ i]<<"   ";
      cout<<setw(7)<<totalVotes<<endl;
}

candidateType::candidateType()
{
      for(int i = 0; i < noOfRegions; i++)
          votesByRegion[ i] = 0;

      totalVotes = 0;
}

void candidateType::getName(string& first, string& last)
{
```

```
        first = firstName;
        last = lastName;
    }
```

因为要比较候选人的姓名，如果两个人的姓名一样，则认为他们是同一个人，所以要为类candidateType重载关系运算符。这些重载关系运算符函数的定义，与类personType中的定义相似，定义如下所示：

```
bool candidateType::operator==(const candidateType& right) const
{
   return(firstName == right.firstName
          && lastName == right.lastName);
}

bool candidateType::operator!=(const candidateType& right) const
{
   return(firstName != right.firstName
          || lastName != right.lastName);
}

bool candidateType::operator<=(const candidateType& right) const
{
   return(lastName <= right.lastName ||
          (lastName == right.lastName &&
           firstName <= right.firstName));
}

bool candidateType::operator<(const candidateType& right) const
{
   return(lastName < right.lastName ||
          (lastName == right.lastName &&
           firstName < right.firstName));
}

bool candidateType::operator>=(const candidateType& right) const
{
   return(lastName >= right.lastName ||
          (lastName == right.lastName &&
           firstName >= right.firstName));
}

bool candidateType::operator>(const candidateType& right) const
{
   return(lastName > right.lastName ||
          (lastName == right.lastName &&
           firstName > right.firstName));
}

const candidateType& candidateType::operator=
                                   (const candidateType& right)
{
      if(this != &right)  //avoid self-assignment
      {
            firstName = right.firstName;
            lastName = right.lastName;

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

```
                votesByRegion[i] = right.votesByRegion[i];

            totalVotes = right.totalVotes;
        }

        return *this;
}

const candidateType& candidateType::operator=
                            (const personType& right)
{
        personType temp;

        temp = right;

        string firstN;
        string lastN;

        temp.getName(firstN, lastN);

        firstName = firstN;
        lastName = lastN;

        return *this;
}
```

主程序

在设计了类candidateType之后，可以开始设计主程序。

因为一共有6个候选人，所以要设计一个candidateList数组，包含6个类型为candidateType的元素。程序要做的第一件事情是从文件candData.txt中读取候选人的姓名，并将其存储到表candidateList中。然后对candidateList进行排序。

然后，从文件voteData.txt中提取选票数据。最后，计算出每个候选人的总得票数，并按规定格式打印表格。其算法如下所示：

1. 将候选人姓名读到candidateList中
2. 对candidateList排序
3. 处理选票数据
4. 计算每个人的总得票数
5. 输出结果

下面的语句创建了orderedArrayListType类型的对象candidateList：

```
orderedArrayListType<candidateType>   candidateList(noOfCandidates);
```

图18.65显示了candidateList对象。数组list中每一个元素都是candidateType类型对象。

在图18.65中，数组votesByRegion和变量totalVotes初始化为0，这是类candidateType默认构造函数所产生的。为了节省篇幅，我们在需要时将对象candidateList画成如图18.66所示的示意图。

函数fillNames 程序要做的第一件事情是从文件中读取候选人的名字到candidateList中。因此，我们用一个函数来实现这个功能。文件candData.txt在main函数中打开。输入文件的名字和candidateList作为参数传递给函数fillNames。由于数据成员list是对象candidateList的protected数据成员，无法直接访问。因此，要创建一个candidateType类型临时对象temp来存放候选人的名字，再用表的insertAt函数将每个候选人名字存入到对象candidateList中。函数fillNames实现如下所示：

```
void fillNames(ifstream& inFile,
               orderedArrayListType<candidateType>& cList)
{
      string firstN;
      string lastN;
      int i;
       candidateType temp;

      for(i = 0; i < noOfCandidates; i++)
      {
          inFile>>firstN>>lastN;
          temp.setName(firstN,lastN);
          cList.insertAt(i,temp);
      }
}
```

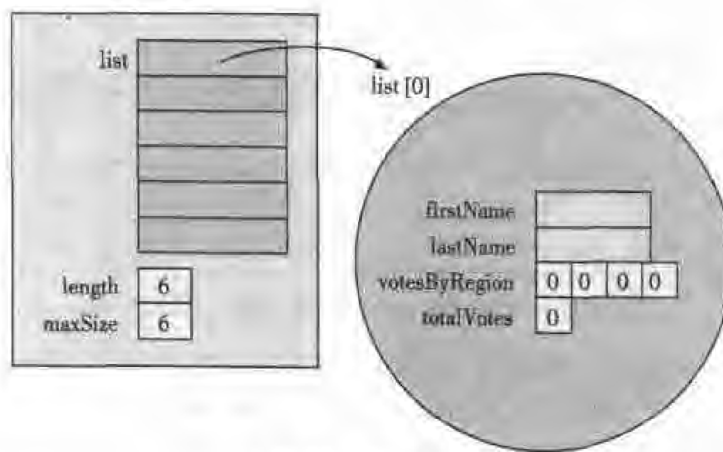


图 18.65　candidateList

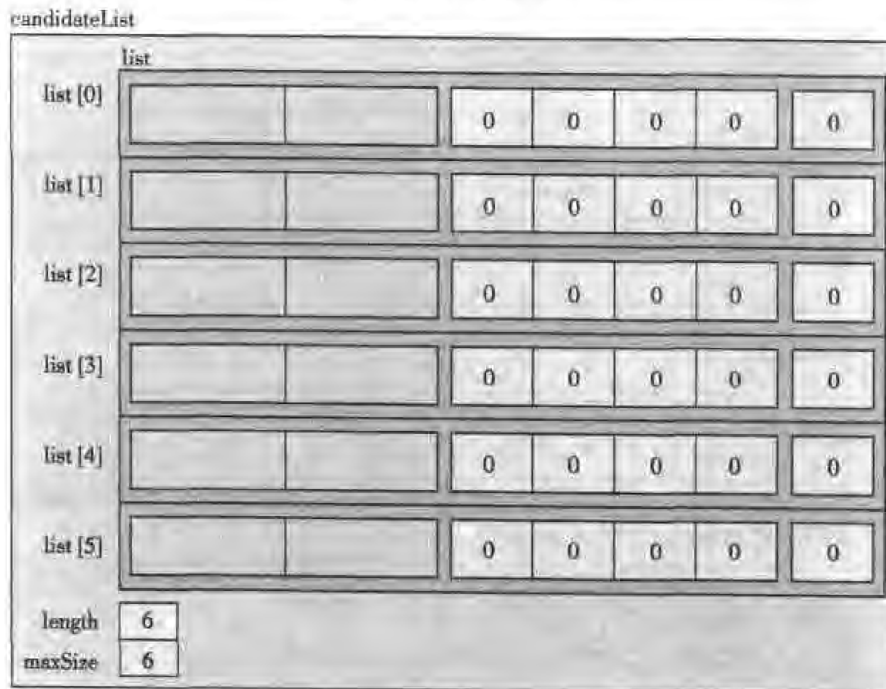


图 18.66　candidateList 对象

调用函数 fillNames 以后，对象 candidateList 如图 18.67 所示。

candidateList

list							
list [0]	Goldy	Goofy	0	0	0	0	0
list [1]	Monty	Mickey	0	0	0	0	0
list [2]	Ducky	Donald	0	0	0	0	0
list [3]	Peter	Pluto	0	0	0	0	0
list [4]	Doctor	Doc	0	0	0	0	0
list [5]	Buddy	Balto	0	0	0	0	0
length	6						
maxSize	6						

图 18.67　调用 fillNames 以后的 candidateList 对象

排序姓名　在读取了候选人的姓名后，要对 candidateList 中的 list 数组进行排序。可以使用本章讨论的任何基于数组的排序算法。因为 candidateList 是 orderedArrayListType 的一个对象，所有的排序算法都可以直接使用。为了方便叙述，这里采用了选择排序算法。下面语句实现了排序功能：

```
candidateList.selectionSort();
```

当这条语句执行完以后，candidateList 如图 18.68 所示。

candidateList

list							
list [0]	Buddy	Balto	0	0	0	0	0
list [1]	Doctor	Doc	0	0	0	0	0
list [2]	Ducky	Donald	0	0	0	0	0
list [3]	Goldy	Goofy	0	0	0	0	0
list [4]	Monty	Mickey	0	0	0	0	0
list [5]	Peter	Pluto	0	0	0	0	0
length	6						
maxSize	6						

图 18.68　执行 candidateList.selectionSort( );以后的 candidateList 对象

处理选票数据

处理选票数据是很容易的。文件 voteData.txt 中的每一行都为如下形式：

```
firstName   lastName   regionNumber   numberOfVotes
```

从文件 voteData.txt 中读取一条记录后，就要在 candidateList 类型的数组 list 中依据候选人来定位记录，然后更新相应的 regionNumber。

由于 votesByRegion 是数组 list 的每个元素的 private 数据成员。而 list 又是 candidateList 的 private 数据成员。因此，要更改候选人选票数，惟一的方法就是将候选人的记录拷贝到一个临时对象中，然后更新该临时对象，再将临时对象拷贝回 list 中，替换原来的对象。可以利用成员函数 retreiveAt 来拷贝临时对象。然后再利用 replaceAt 把临时对象拷贝回 list。假设下一条记录是：

```
Ducky  Donald  2  35
```

该条记录表明 Ducky Donald 在选区 2 获得了 35 张选票。假设在处理记录前，candidateList 如图 18.69 所示。

candidateList

list							
list [0]	Buddy	Balto	0	0	50	0	0
list [1]	Doctor	Doc	10	0	56	0	0
list [2]	Ducky	Donald	76	13	0	0	0
list [3]	Goldy	Goofy	0	45	0	0	0
list [4]	Monty	Mickey	80	0	78	0	0
list [5]	Peter	Pluto	100	0	0	20	0

length 6
maxSize 6

图 18.69　处理记录 Ducky Donald 2 35 前的 candidateList 对象

我们首先对 candidateList 中对应于 Ducky Donald 的记录做一个拷贝（如图 18.70 所示）。

图 18.70　temp 对象

接下来，下面的语句更改选票（这里 region = 2，votes = 35）：

```
temp.updateVotesByRegion(region, votes);
```

该语句执行以后，对象temp如图18.71所示。

图18.71 temp.updateVotesByRegion(region, votes);执行后的temp对象

现在，将temp拷贝回list中（如图18.72所示）。

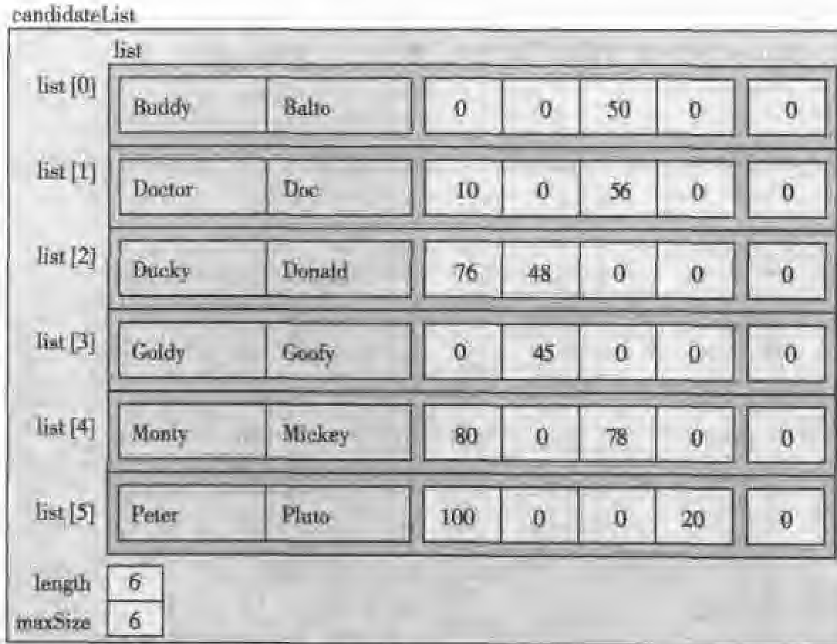


图18.72 拷贝回temp后的candidateList

因为list是排过序的。所以可以用折半查找算法在list中查找得票数需要更新的候选人的位置。同样，函数binarySearch是类orderedArrayListType的一个成员函数，可以使用这个函数来查找list。函数processVotes定义如下所示：

```
void processVotes(ifstream& inFile,
                  orderedArrayListType<candidateType>& cList)
{
      string firstN;
      string lastN;
      int region;
      int votes;
      int candLocation;

      candidateType temp;

      inFile>>firstN>>lastN>>region>>votes;
      temp.setName(firstN,lastN);
      temp.setVotes(region,votes);

      while(inFile)
      {
            candLocation = cList.binarySearch(temp);
            if(candLocation != -1)
            {
```

```
                    cList.retrieveAt(candLocation,temp);
                    temp.updateVotesByRegion(region,votes);
                    cList.replaceAt(candLocation,temp);
                }

                inFile>>firstN>>lastN>>region>>votes;

                temp.setName(firstN,lastN);
                temp.setVotes(region,votes);
        }
}
```

统计选票　在处理完选票数据以后，接下来就是统计每个人的得票总数。采用方法是将每个选区的选票相加。由于 votesByRegion 是 candidateType 的一个 private 数据成员，list 是 candidateType 的一个 protected 数据成员。因此，为了统计每个候选人的选票，需要用 retreiveAt 函数来获取每个人的数据到一个临时对象，并在临时对象中统计选票，再把临时对象拷贝回 candidateList 中。下面函数完成这个功能：

```
void addVotes(orderedArrayListType<candidateType>& cList)
{
        int i;

        candidateType temp;

        for(i = 0; i < noOfCandidates; i++)
        {
                cList.retrieveAt(i,temp);
                temp.calculateTotalVotes();
                cList.replaceAt(i,temp);
        }
}
```

图 18.73 显示了对每个候选人统计选票以后的 candidateList——即调用 addVotes 函数以后的 candidateList。

candidateList

list

list [0]	Buddy	Balto	0	0	0	272	272
list [1]	Doctor	Doc	25	71	156	97	349
list [2]	Ducky	Donald	110	158	0	0	268
list [3]	Goldy	Goofy	75	34	134	0	243
list [4]	Monty	Mickey	112	141	156	89	498
list [5]	Peter	Pluto	285	56	0	46	387

length: 6
maxSize: 6

图 18.73　调用函数 addVotes 以后的 candidateList

打印

为了完成程序，还需要增添打印功能。首先需要输出表头（前4行输出）。下面的函数实现这个功能：

```
void printHeading()
{
    cout<<"     ---------------Election Results---------------"
        <<"-----------"<<endl<<endl;
    cout<<"                                          Votes"<<endl;
    cout<<"    Candidate Name      Region1  Region2  Region3   "
       <<"Region4    Total"<<endl;
    cout<<"--------------------  ------  ------    "
       <<"------  ------  ------"<<endl;
}
```

现在，讨论函数printResults，它打印结果数据。假设变量sumVotes表示选举的总票数，变量largestVotes表示候选人的最大得票数，变量winLoc表示list中获胜者的下标。进一步假设temp是一个candidateType对象。该函数的算法如下所示：

i. 初始化sumVotes，largestVotes，winLoc为0。
ii. 对每个候选人：
  a. 提取其信息到temp中
  b. 打印候选人的姓名和相应信息
  c. 获取每个人的总得票数，更新sumVotes
  d. if(largestVotes < temp.getTotalVotes( ))

```
   {
       largestVotes = temp.getTotalVotes();
       winLoc = i ;  //This is the ith candidate
   }
```

iii. 输出最后结果

函数printResults定义如下所示：

```
void printResults(orderedArrayListType<candidateType>& cList)
{
   int i;

   candidateType temp;
   int largestVotes = 0;
   int sumVotes = 0;
   int winLoc = 0;

   for(i = 0; i < noOfCandidates; i++)
   {
       cList.retrieveAt(i,temp);
       temp.printData();

       sumVotes += temp.getTotalVotes();

       if(largestVotes < temp.getTotalVotes())
       {
          largestVotes = temp.getTotalVotes();
          winLoc = i;
       }
   }

   cList.retrieveAt(winLoc,temp);
```

```
    cout<<endl<<endl<<"Winner: ";

    string firstN;
    string lastN;

    temp.getName(firstN,lastN);
    cout<<firstN<<" "<<lastN;
    cout<<",  Votes Received: "<<temp.getTotalVotes()<<endl<<endl;
    cout<<"Total votes polled: "<<sumVotes<<endl;
}
```

全整的程序代码清单

```
#include <iostream>
#include <string>
#include <fstream>
#include "candidateType.h"
#include "orderedArrayListType.h"

using namespace std;


const int noOfCandidates = 6;

void fillNames(ifstream& inFile,
               orderedArrayListType<candidateType>& cList);
void processVotes(ifstream& inFile,
                  orderedArrayListType<candidateType>& cList);
void addVotes(orderedArrayListType<candidateType>& cList);

void printHeading();
void printResults(orderedArrayListType<candidateType>& cList);

int main()

{
  orderedArrayListType<candidateType> candidateList(noOfCandidates);

  candidateType temp;

  ifstream inFile;

  inFile.open("a:candData.txt");

  fillNames(inFile, candidateList);

  candidateList.selectionSort();

  inFile.close();

  inFile.open("a:voteData.txt");

  processVotes(inFile,candidateList);

  addVotes(candidateList);

  printHeading();

  printResults(candidateList);
```

```
  return 0;
}

//Place the definitions of the functions fillNames, processVotes,
//addVotes, printHeading, and printResults here.
```

程序输出结果

```
              ----------Election Results----------
                                                 Votes
    Candidate Name       Region1  Region2  Region3  Region4   Total
---------------------    -------  -------  -------  -------  ------
Buddy      Balto               0        0        0      272     272
Doctor     Doc                25       71      156       97     349
Ducky      Donald            110      158        0        0     268
Goldy      Goofy              75       34      134        0     243
Monty      Mickey            112      141      156       89     498
Peter      Pluto             285       56        0       46     387
Winner: Monty Mickey,  Votes Received: 498

Total votes polled: 2017
```

输入文件

candData.txt

```
Goldy Goofy
Monty Mickey
Ducky Donald
Peter Pluto
Doctor Doc
Buddy Balto
```

voteData.txt

```
Goldy Goofy 2 34
Monty Mickey 1 56
Ducky Donald 2 56
Peter Pluto 1 78
Doctor Doc 4 29
Buddy Balto 4 78
Monty Mickey 2 63
Ducky Donald 1 23
Peter Pluto 2 56
Doctor Doc 1 25
Peter Pluto 4 23
Doctor Doc 4 12
Goldy Goofy 3 134
Buddy Balto 4 82
Monty Mickey 3 67
Ducky Donald 2 67
Doctor Doc 3 67
Buddy Balto 4 23
Monty Mickey 1 56
Ducky Donald 2 35
Peter Pluto 1 27
Doctor Doc 2 34
Goldy Goofy 1 75
Peter Pluto 4 23
```

```
Monty Mickey 4 89
Peter Pluto 1 23
Doctor Doc 3 89
Monty Mickey 3 89
Peter Pluto 1 67
Doctor Doc 2 37
Buddy Balto 4 89
Monty Mickey 2 78
Ducky Donald 1 87
Peter Pluto 1 90
Doctor Doc 4 56
```

18.9 小结

1. 表是相同类型的元素集合。
2. 表长即为元素的个数。
3. 存储和处理表的一种方便形式是一维数组。
4. 顺序查找算法从第一个位置开始，在表中查找一个指定的数据项。该算法不断地将查找元素与表中元素相比较，直到找到匹配元素或整个表历遍完为止。
5. 在平均情况下，顺序查找算法查找次数为表长的一半。
6. 对长度为 n 的表，如果顺序查找算法成功，平均要做 $\frac{n+1}{2}=O(n)$ 次比较。
7. 顺序查找算法适用于较短的表。
8. 折半查找算法要比顺序查找算法快得多。
9. 折半查找算法要求表元素是有序的——即排过序。
10. 对一个长度为 1 024 的表，要查找一项，折半查找算法的循环迭代次数不超过 11，因此比较次数不超过 22。
11. 对一个长度为 n 的表，如果查找成功，平均情况下，折半查找算法要做：$\frac{2(n+1)}{n}\log_2(n+1)-3=O(\log_2 n)$ 次比较。
12. 选择排序算法通过查找最小（或者最大）元素，并将其移到表的第一个（或最后一个）位置上来排序表。
13. 对长度为 n 的表，这里 $n>0$，选择排序要做 $\frac{1}{2}n^2-\frac{1}{2}n$ 次比较，并进行 $3(n-1)$ 次赋值操作。
14. 对长度为 n 的表，这里 $n>0$，平均情况下，插入排序要进行 $\frac{1}{4}n^2+O(n)$ 次比较，并执行 $\frac{1}{4}n^2+O(n)$ 次赋值操作。
15. 设 L 是一个有 n 个元素的表。对于任何基于比较的排序算法，在最坏情况下，比较次数至少为 $O(n*\log_2 n)$ 次。
16. 快速排序和归并排序都要对表进行拆分。
17. 快速排序拆分表时，首先选取表中的一个元素，称为支点（pivot）。然后将表中的元素重新排列，使得其中一个子表中的所有元素都小于该支点（pivot），而另一个子表中的所有元素都大于或等于该支点（pivot）。
18. 快速排序时，排序工作在拆分表时完成。
19. 平均情况下，快速排序比较次数是 $O(n*\log_2 n)$。最坏情况下，快速排序的比较次数是 $O(n^2)$。
20. 归并排序对表的拆分是从中间进行的。
21. 归并排序时，排序工作在归并表时完成。
22. 归并排序的比较次数是 $O(n*\log_2 n)$。

18.10 练习

1. 判断下面说法的正误。
   a. 顺序查找要假设表是升序的。
   b. 折半查找要假设表是有序的。
   c. 折半查找在有序表上较快，在无序表上较慢。
   d. 折半查找在长表上较快，而顺序查找在短表上较快。
   e. 平均情况下，选择排序的比较次数少于插入排序。
   f. 快速排序总是比归并排序快。
2. 考虑下面的表：

   ```
   63  45  32  98  46  57  28  100
   ```

   使用本章介绍的顺序查找算法，发现下列各项是否在表中各需要比较多少次(注意，这里说的比较是指元素比较，而不是下标比较)？
   a. 90
   b. 57
   c. 63
   d. 120
3. 考虑下面的表：

   ```
   2 10 17 45 49 55 68 85 92 98 110
   ```

   使用本章介绍的折半查找算法，发现下列各项是否在表中各需要比较多少次？并指出每次循环后的 first，last 和 middle 的值和比较次数。
   a. 15
   b. 49
   c. 98
   d. 99
4. 假设有下面的表：

   ```
   5,18,21,10,55,20
   ```

   前三项是顺序的。使用本章介绍的插入排序算法来确定 10 的位置，需要比较多少次?
5. 假设有下面的表：

   ```
   7,28,31,40,5,20
   ```

   前三项是顺序的。使用本章介绍的插入排序算法来确定 5 的位置，需要比较多少次?
6. 假设有下面的表：

   ```
   28,18,21,10,25,30,12,71,32,58,15
   ```

   使用本章介绍的基于数组的插入排序算法对表排序。请指出经过 6 轮排序——即 for 语句循环执行 6 次以后的结果。
7. 针对本章讨论的插入排序算法。假设有下面的表：

   ```
   18,8,11,9,15,20,32,61,22,48,75,83,35,3
   ```

   使用插入排序算法对表排序需要多少次比较?
8. 归并排序算法和快速排序算法都通过拆分表来排序。解释二者的区别。

9. 假设有下面的表：

```
16,38,54,80,22,65,55,48,64,95,5,100,58,25,36
```

使用本章介绍的快速排序法对该表排序。把表的中间元素作为支点（pivot）。

a. 给出调用函数 partition 一次后的结果。

b. 给出调用函数 partition 二次后的结果。

10. 假设有下面的表：

```
18,40,16,82,64,67,57,50,37,47,72,14,17,27,35
```

使用本章介绍的快速排序法对该表排序。把 first，last 和 middle 的中值作为支点（pivot）。

a. pivot 的值是多少？

b. 给出调用函数 partition 一次后的结果。

11. 为基于数组的表编写实现查找和排序算法的类 orderedArrayListType 的定义。

12. 为链表编写查找和排序算法的类 orderedLinkedListType 的定义。

18.11 编程练习

1. 本章给出的顺序查找算法是非递归的。请给出顺序查找算法的一个递归版本（递归顺序查找）。
2. 本章给出的折半查找算法是非递归的。请给出折半查找算法的一个递归版本（递归折半查找）。
3. 本章中的顺序查找算法并不假设表已经排序。因此，它对有序表和无序表都适用。然而，如果表是有序的，可以适当地提高顺序查找算法的效率。例如，如果查找项不在表中，则在发现有一个表元素大于查找项的时候，就可以停止查找。请编写一个函数 seqOrdSearch，为有序表实现上述顺序查找算法。把这个函数添加到类 orderedArrayListType 中，并编写一个程序来测试它。
4. 编写一个程序统计折半查找的比较次数和顺序查找的比较次数：

   假设 list 是一个有 1 000 个元素的数组：

   a. 使用随机数生成器生成该表元素

   b. 使用任意的排序算法对表排序

   c. 表中查找方式如下：

      i. 使用折半查找算法查找

      ii. 使用折半查找算法，但是设置一个跳转语句，当表长减小到小于 15 的时候，使用顺序查找算法（使用编程练习 1 中的基于有序表的顺序查找算法）

   d. 打印步骤 c.i 和步骤 c.ii 的比较次数。如果项被找到，并打印其下标位置。
5. 编写一个程序测试函数 insertOrd，该函数的功能是在基于数组的有序表中插入一个数据项。
6. 编写一个函数 removeOrd，它从基于数组的有序表中删除一项。被删除的数据项作为参数传递给该函数。删除元素之后，表必须是有序的，并且元素之间不能有空项。把这个函数添加到类 orderedArrayListType 中，并且编写一个程序测试它。
7. 编写一个基于链表的选择排序算法，并测试它。
8. 编写一个程序，测试本章给出的基于数组表的插入排序算法。
9. 编写一个程序，测试本章给出的基于链表的插入排序算法。
10. 编写一个程序，检测本章给出的基于数组表的快速排序算法。
11. 编写一个基于链表的快速排序算法，并且测试它。
12. (C.A.R.Hoare)设 L 是一个大小为 n 的表。可以使用快速排序算法查找该表中第 k 个最小的元素，这里 $0 \leq k \leq n-1$，且无须对 L 进行彻底排序。编写一个 C++ 函数 kThSmallestItem，利用快速排序算法来查找 L 中第 k 个最小元素，不必对表进行彻底排序。

13. 对一个有10 000个元素的数组进行快速排序如下所示：
   a. 使用中间元素作为pivot排序。
   b. 使用起始元素、末尾元素和中间元素的中值作为pivot排序。
   c. 使用中间元素作为pivot排序。但是当子表小于20时，用插入排序法排序子表。
   d. 使用起始元素、末尾元素和中间元素的中值作为pivot排序。但是当子表小于20时，用插入排序法排序子表。
   e. 分别计算并输出以上4种方法耗费的CPU时间。
   为了获得当前的CPU时间，声明一个变量x，类型为clock_t。语句x = clock( );将当前CPU时间保存到x中。在程序执行一段时间后，再次检测当前的CPU时间，然后计算出CPU的执行时间。注意，为了使用clock_t类型和clock函数，在程序中必须包含头文件ctime。使用随机数函数来初始化表。
14. 编写一个程序，测试本章讨论的归并排序算法。
15. 在本章“程序范例：选举结果统计”中，类candidateType包含函数calculateTotalVotes，用于在处理完选票数据后计算每个候选人的总得票数。类candidateType的函数updateVotesByRegion仅用于更新一个特定选区的选票数。修改这个函数，使其可以更新候选人的总得票数。为此，在主程序中函数addVotes不再被需要。修改并运行程序，使其使用修改过的函数updateVotesByRegion。
16. 在本章“程序范例：选举结果统计”中，类orderedArrayListType对象candidateList用于处理选票数据。插入候选人数据及更新和统计选票等几个操作比较复杂。为了更新候选人的选票数，就要把其信息从candidateList中拷贝到一个临时的candidateType类型对象中，更新临时对象后，再将它替换原来的信息。这是因为数据成员list是candidateList的protected数据成员，并且list的每个元素又是private的。在此，请修改选举结果统计程序简化候选人数据的存取，可按下面方式从类orderedArrayListType中派生一个类candidateListType：

```
class candidateListType: public orderedArrayListType<candidateType>
{
public:
    candidateListType();
         //default constructor
    candidateListType(int size);
         //constructor
    void processVotes(string fName, string lName, int region,
                      int votes);
         //Function to update the number of votes for
         //a particular candidate for a particular region.
         //The name of the candidate, the region number, and
         //the number of votes are passed as parameters.
    void addVotes();
         //Function to find the total number of votes
         //received by each candidate.
    void printResult();
         //Function to output the voting data.
};
```

因为类candidateListType从类orderedArrayListType中派生而来，而list是类orderedArrayListType（从类arrayListType继承）的protected数据成员，所以list可以直接被类candidateListType中的成员访问。

请写出类candidateListType中的成员函数的定义，并且使用这个类来重写该程序，并再次运行该程序。

第19章 二 叉 树

本章要点：

- 了解二叉树
- 了解各种二叉树遍历算法
- 理解怎样在二叉搜索树中组织数据
- 理解怎样在二叉搜索树中插入、删除元素
- 了解非递归的二叉树遍历算法

19.1 二叉树简介

在组织数据时，程序员最优先考虑的就是能够快速地插入、删除和查找元素。我们已经知道了如何在数组中存储和处理数据。因为数组是随机访问数据结构，如果将数据组织适当（进行了排序），就可以使用折半查找算法等查找算法有效地查找、读取表中的元素。然而，在数组中存储数据有它自己的局限性。例如：元素的插入（尤其对于有序数组）和删除要耗费大量的计算机时间。特别是当表非常大时，由于每次操作都需要移动元素，耗时显得尤为明显。为了提高元素插入和删除的速度，可以使用链表。在链表中插入和删除元素并不需要移动数据，只需要简单地调整一下链表中的指针。然而链表也存在着只能顺序地处理数据的缺点。也就是说，为了插入、删除一个元素或者只是简单地查找链表中的某个元素，必须从链表的第一个节点开始查找。由于顺序查找的平均查找长度是整个链表长度的一半，所以顺序查找只适合非常小的链表。

本章将讨论如何动态地组织数据从而提高元素的插入、删除和查找效率。

为了方便讨论，首先介绍一些定义：

二叉树（binary tree） 一个二叉树T或者为空或者符合以下条件：

(i) T有一个称为根节点（root）的特殊节点。

(ii) T有两个节点集合：L_T和R_T，分别叫做T的左子树和右子树。

(iii) L_T和R_T均是二叉树。

二叉树可以用图形表示。假设T是有一个根节点A的二叉树。用L_A表示A的左于树，R_A表示A的右子树。L_A和R_A都是二叉树。假设B是L_A的根节点，C是R_A的根节点。B被称为A的左孩子（left child），C被称为A的右孩子（right chile），而A被称为B和C的父节点（parent）。

在二叉树图中，每个二叉树的节点都用一个圆表示，并用节点名标识。二叉树的根节点画在最顶部。根节点的左孩子（如果有的话）画在根节点的左下方。类似地，根节点的右孩子（如果有的话）画在根节点的右下方。于节点和父节点通过一条由父节点指向子节点的箭头连接。该箭头通常被称为有向边（Directed Edge）或有向分支（Directed Branch）（或者简单地称为分支）。由于已经画了L_A的根节点B，使用同样的方法来画L_A的剩余部分。R_A也用类似的方法生成。如果一个节点没有左孩子，就画一个从该节点出发指向左下方的箭头，箭头最后指向三条横线。即如果一个箭头指向三条横线就说明该节点的相应子树为空。

图 19.1 是一个二叉树的图例。二叉树的根节点为A，用L_A表示根节点的左子树，$L_A=\{B, D, E, G\}$。用R_A表示的根节点的右子树，$R_A=\{C, F, H\}$。A的左子树根节点（也就是L_A的根节点）为B。R_A

的根节点是C，其他类似。很明显，L_A和R_A都是二叉树。由于一个箭头指向三条横线意味着该子树为空，所以D的左子树为空。

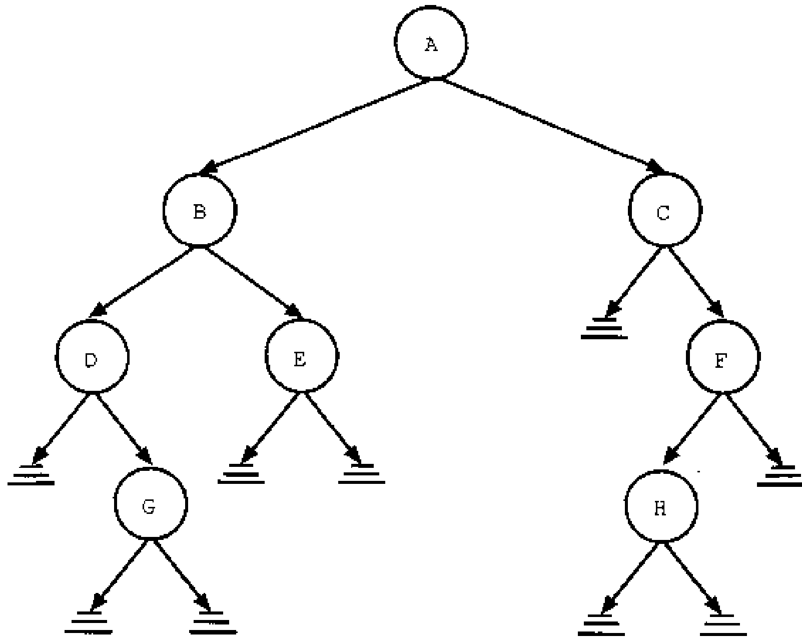


图 19.1　二叉树

在图 19.1 中，A的左孩子是B，右孩子为C。同样，F的左孩子是H，F没有右孩子。

例 19.1 到例 19.5 展示了一些非空二叉树。

例 19.1　该例展示了一个只有一个节点的二叉树，如图 19.2 所示。

在图 19.2 的二叉树中：

- 二叉树的根节点 = A
- L_A = 空
- R_A = 空

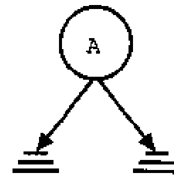


图 19.2　只有一个节点的二叉树

例 19.2　该例展示一个有两个节点的二叉树。考虑图 19.3 中的二叉树。

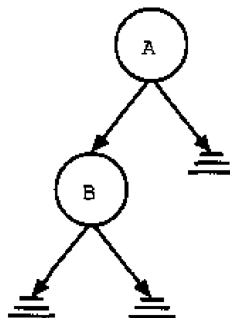


图 19.3　有两个节点的二叉树，根节点的右子树为空

在图 19.3 的二叉树中：

- 二叉树的根节点 = A
- $L_A = \{B\}$
- R_A = 空

- L_A的根节点 = B
- L_B = 空
- R_B = 空

例 19.3 该例展示了一个有两个节点的二叉树，如图 19.4 所示。

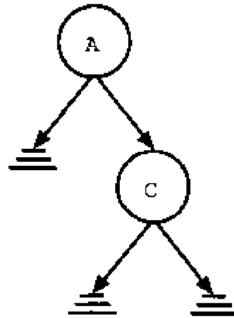


图 19.4 有两个节点的二叉树，根节点的左子树为空

在图 19.4 的二叉树中：

- 二叉树的根节点 = A
- L_A = 空
- $R_A = \{C\}$
- R_A的根节点 = C
- L_C = 空
- R_C = 空

例 19.4 该例展示了一个有三个节点的二叉树，如图 19.5 所示。

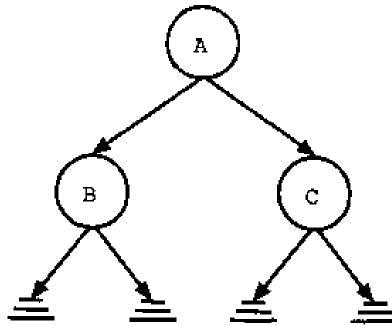


图 19.5 有三个节点的二叉树

在图 19.5 的二叉树中：

- 二叉树的根节点 = A
- $L_A = \{B\}$
- $R_A = \{C\}$
- L_A的根节点 = B
- L_B = 空
- R_B = 空
- R_A的根节点 = C
- L_C = 空
- R_C = 空

例 19.5 该例展示了其他情况的有三个节点的二叉树，如图 19.6 所示。

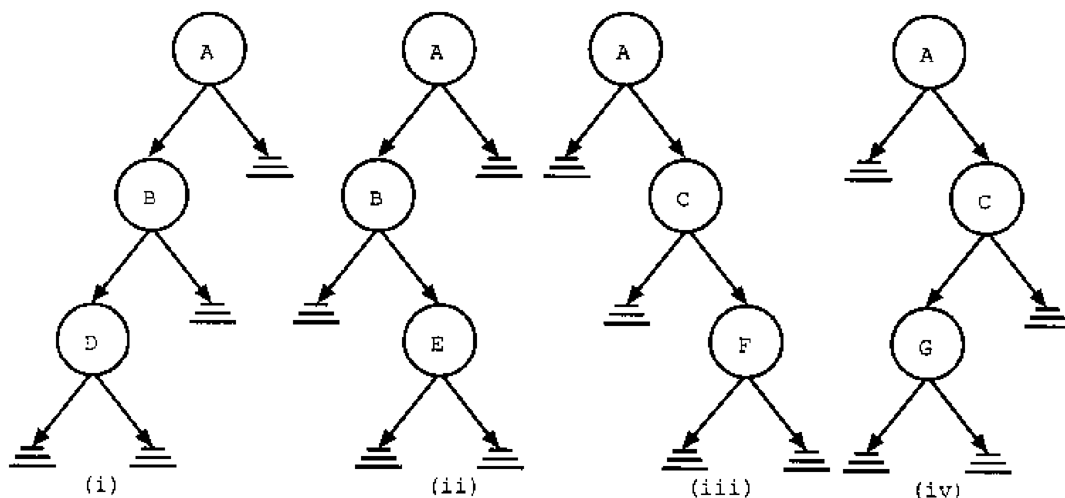


图 19.6 几个有三个节点的二叉树

正如前面范例所示，二叉树中每个节点最多有两个子节点。所以每个节点除了要保存自己的信息之外，还必须要跟踪它的左子树和右子树。这意味着每个节点都有两个指针：llink 和 rlink。指针 llink 指向该节点左子树的根节点；指针 rlink 指向该节点右子树的根节点。

下面的结构定义一个二叉树的节点：

```
template <class elemType>
struct nodeType
{
   elemType               info;
   nodeType<elemType>   *llink;
   nodeType<elemType>   *rlink;
};
```

根据节点的定义，可以看到对于每个节点：

1. 数据存储在 info 中
2. 指向左孩子的指针存储在 llink 中
3. 指向右孩子的指针存储在 rlink 中

另外，指向二叉树根节点的指针存储在一个 nodeType 类型指针变量中，通常称为 root。所以，一般来说一个二叉树就像图 19.7 所示的那样。

为了简便，我们将继续使用原来的图示表示二叉树。也就是说用圆圈表示节点，用左、右箭头表示链接。跟以前一样，箭头指向三条横线表示该子树为空。

在结束本节之前，我们定义一些术语。

如果一个二叉树节点既没有左孩子也没有右孩子，则称该节点为叶节点（leaf）。假设 U 和 V 是二叉树 T 中的两个节点，如果存在一个由 U 指向 V 的分支，则称 U 是 V 的父节点（parent）。从一个二叉树节点 X 到节点 Y 的路径（path）是指节点 X_0, X_1, …, X_n 序列，其中：

1. $X = X_0$, $X_n = Y$
2. 对所有 $i = 1, 2, \cdots, n$, X_{i-1} 都是 X_i 的父节点。也就是说从 X_0 到 X_1, X_1 到 X_2, …, X_{i-1} 到 X_i, …, X_{n-1} 到 X_n 各有一个分支。

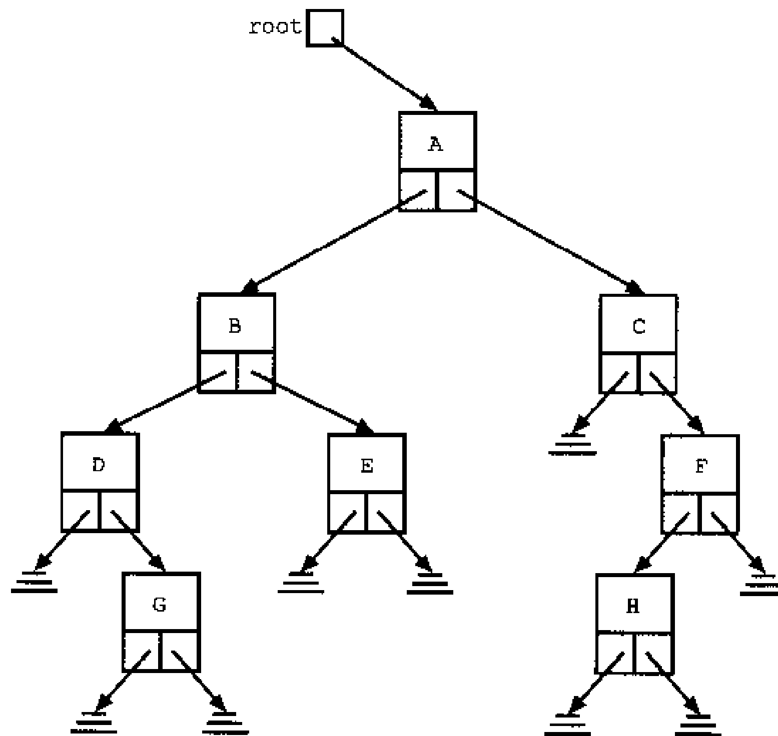


图19.7 二叉树

因为分支只是从父节点指向子节点，所以从上面的讨论中可以清楚地得出：在二叉树中，从根节点到每个节点都有一个惟一的路径。

节点的层数（Level of a Node） 在二叉树中，节点的层数是指从根节点到该节点所经过的分支个数。

很明显，二叉树的根节点的层数为0，根节点的子节点的层数为1。

二叉树的高度（Height of a Binary Tree） 二叉树的高度是指从根节点到叶节点的最长路径上的节点个数。

假设有一个指向二叉树根节点的指针p，接下来我们要描述一个用来计算二叉树高度的C++函数height。指向根节点的指针作为参数传递给函数height。

如果二叉树为空，则height为0。假设二叉树为非空。要获得二叉树的高度，可以首先获得左子树和右子树的高度，然后在两个子树高度的最大值上加1，得到二叉树的高度。由于左（右）子树也是二叉树，所以可以采用同样的方法获得左（右）子树的高度。因此计算二叉树高度的通用算法如下所示。假设height(p)表示根节点为p的二叉树的高度。

```
if(p is NULL)
   height(p) = 0
else
   height(p) = 1 + max(height(p->llink), height(p->rlink))
```

显而易见，这是一个递归算法。下面函数实现该算法：

```
template<class elemType>
int height(nodeType<elemType> *p)
{
   if(p == NULL)
      return 0;
   else
```

```
        return 1 + max(height(p->llink), height(p->rlink));
}
```

在函数 height 的定义中使用了函数 max 来计算两个整数中的最大值。函数 max 很容易实现。

类似地，可以实现获得二叉树中节点个数和叶节点个数的算法。

19.1.1 拷贝树

二叉树的拷贝是一个很有用的操作。二叉树是动态数据结构，也就是说二叉树节点的内存是在程序执行过程中分配和释放的。所以，如果使用根节点指针的值拷贝二叉树的话，只能得到数据的浅拷贝。要想获得与原二叉树完全相同的拷贝，就要创建同原二叉树节点同样多的节点。而且，拷贝的二叉树中节点的顺序必须同原二叉树中节点的顺序相同。

假设有一个指向二叉树根节点的指针。下面，描述二叉树的拷贝函数。该函数还将用在本章后面（见“实现二叉树”一节）描述的拷贝构造函数和重载赋值运算符的实现中。

```
template <class elemType>
void copyTree(nodeType<elemType>* &copiedTreeRoot,
              nodeType<elemType>* otherTreeRoot)
{
   if(otherTreeRoot == NULL)
      copiedTreeRoot = NULL;
   else
   {
      copiedTreeRoot = new nodeType<elemType>;
      copiedTreeRoot->info = otherTreeRoot->info;
      copyTree(copiedTreeRoot->llink, otherTreeRoot->llink);
      copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
   }
} //end copyTree
```

当重载赋值运算符和实现拷贝构造函数时，我们将使用函数 copyTree。

19.1.2 二叉树遍历

元素的插入、删除和查找都需要对二叉树进行遍历。所以二叉树中最常用的操作就是遍历二叉树，即访问二叉树中的每个节点。从二叉树的图示中可以看到，由于存在一个指向二叉树根节点的指针，所以每次遍历都必须从根节点开始。对于每个节点，都有两种选择：

- 先访问节点
- 先访问子树

不同的选择导致不同的二叉树遍历方式。

- 中序遍历
- 前序遍历
- 后序遍历

下面章节将会讨论这三种遍历方式。

中序遍历

在中序遍历中，二叉树的遍历方式如下所示：

1. 遍历左子树
2. 访问节点

3. 遍历右子树

前序遍历

在前序遍历中，二叉树的遍历方式如下所示：

1. 访问节点
2. 遍历左子树
3. 遍历右子树

后序遍历

在后序遍历中，二叉树的遍历方式如下所示：

1. 遍历左子树
2. 遍历右子树
3. 访问节点

很明显，每个遍历算法都是递归算法。

通过二叉树中序遍历生成的节点列表称为中序序列(Inorder Sequence)；通过二叉树前序遍历生成的节点列表称为前序序列(Preorder Sequence)；通过二叉树后序遍历生成的节点列表称为后序序列(Postorder Sequence)。

在给出每种遍历算法的C++代码之前，先描述图19.8中所示二叉树的中序遍历。为了简单起见，假设在访问节点时只输出节点中存储的数据。在"二叉树遍历与函数参数"一节中介绍了如何修改二叉树遍历算法，以便用户可以通过使用函数来指定在访问一个节点时对节点实施何种操作。

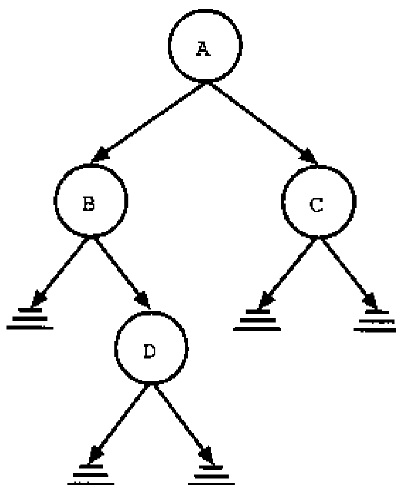


图19.8 中序遍历的二叉树

变量root中保存着指向图19.8所示的二叉树根节点的指针，即指向info为A的节点。所以，首先从A开始遍历。

1. 遍历A的左子树，也就是遍历$L_A=\{B, D\}$。
2. 访问节点A。
3. 遍历A的右子树，也就是遍历$R_A=\{C\}$。

注意，必须等到步骤1结束以后才能执行步骤2。

1. 遍历A的左子树，也就是遍历$L_A=\{B, D\}$。L_A是根节点为B的二叉树。所以可以对L_A应用同样的中序遍历算法。

1.1. 遍历 B 的左子树，也就是遍历 L_B = 空。

1.2. 访问 B。

1.3. 遍历 B 的右子树，也就是遍历 $R_B = \{D\}$。

与前面一样，只有步骤 1.1 完成后才能执行步骤 1.2。

1.1. 由于 B 的左子树为空，所以无须遍历。步骤 1.1 完成后，开始执行步骤 1.2。

1.2. 访问 B，将 B 输出到输出设备。显而易见，输出的第一个节点是 B。步骤 1.2 完成后，开始执行步骤 1.3。

1.3. 遍历 B 的右子树，也就是遍历 $R_B = \{D\}$。R_B 是二叉树。所以可以对 R_B 应用同样的中序遍历算法。

1.3.1. 遍历 D 的左子树，也就是遍历 L_D = 空。

1.3.2. 访问 D。

1.3.3. 遍历 D 的右子树，也就是遍历 R_D = 空。

1.3.1. 由于 D 的左子树为空，所以无须遍历。步骤 1.3.1 完成，开始执行步骤 1.3.2。

1.3.2. 访问 D。将 D 输出到输出设备。步骤 1.3.2 完成，开始执行步骤 1.3.3。

1.3.3. 由于 D 的左子树为空，所以无须遍历。步骤 1.3.3 完成。

步骤 1.3 完成。由于步骤 1.1，步骤 1.2 和步骤 1.3 都已完成，所以步骤 1 完成，开始执行步骤 2。

2. 访问 A。将 A 输出到输出设备。步骤 2 完成，开始执行步骤 3。

3. 遍历 A 的右子树，也就是遍历 $RA = \{C\}$。R_A 是根节点为 C 的二叉树。所以可以对 R_A 应用同样的中序遍历算法。

3.1. 遍历 C 的左子树，也就是遍历 L_C = 空。

3.2. 访问 C。

3.3. 遍历 C 的右子树，也就是遍历 R_C = 空。

3.1. 由于 C 的左子树为空，所以无须遍历。步骤 3.1 完成。

3.2. 访问 C，将 C 输出到输出设备。步骤 3.2 完成，开始执行步骤 3.3。

3.3. 由于 C 的右子树为空，所以无须遍历。步骤 3.3 完成。

步骤 3 完成，同时整个二叉树的遍历也就完成。

很明显，对该二叉树中序遍历会输出如下序列：

中序序列：*B D A C*

类似地，前序遍历和后序遍历分别会输出如下序列：

前序序列：*A B D C*

后序序列：*D B C A*

通过上面对中序遍历的简单描述，可以看到：算法访问完节点的左子树之后必须返回到节点本身。而链接只有一个方向，即从父节点指向左孩子或右孩子，没有从子节点指向父节点的指针。所以在移动到子节点之前，必须保存指向父节点的指针。一个比较方便的方法就是用一个递归中序算法。这是因为在递归调用结束后，控制会返回给调用函数（后面我们将讨论如何编写非递归遍历算法）。中序遍历算法的递归函数定义如下所示：

```
template<class elemType>
void inorder(nodeType<elemType> *p)
{
   if(p != NULL)
   {
```

```
        inorder(p->llink);
        cout<<p->info<<" ";
        inorder(p->rlink);
    }
}
```

为了对二叉树进行遍历，应该将二叉树的根节点作为参数传递给函数inorder。例如，如果root指向二叉树的根节点，那么函数的调用如下所示：

```
inorder(root);
```

类似地，我们还可以写出前序遍历和后序遍历函数，它们的定义如下所示：

```
template<class elemType>
void preorder(nodeType<elemType> *p)
{
    if(p != NULL)
    {
        cout<<p->info<<" ";
        preorder(p->llink);
        preorder(p->rlink);
    }
}

template<class elemType>
void postorder(nodeType<elemType> *p)
{
    if(p != NULL)
    {
        postorder(p->llink);
        postorder(p->rlink);
        cout<<p->info<<" ";
    }
}
```

19.1.3 实现二叉树

前面章节描述了关于二叉树的各种操作以及实现这些操作的函数。本节将描述作为抽象数据类型（ADT）的二叉树。在设计作为抽象数据类型的二叉树的类之前，先给出在二叉树上的典型操作：

1. 检查二叉树是否为空
2. 在二叉树中查找特定的元素
3. 在二叉树中插入一个元素
4. 从二叉树中删除一个元素
5. 获得二叉树的高度
6. 获得二叉树的节点数目
7. 获得二叉树的叶节点数目
8. 遍历二叉树
9. 拷贝二叉树

元素的查找、插入和删除操作都需要对二叉树进行遍历。然而，由于二叉树的节点并没有特定顺序，所以这些算法对于任意的二叉树效率并不高。在下一节我们将看到，二叉树的查找并不存在着标准。因此我们会针对特定的二叉树讨论这些算法。

下面的类定义了一个作为抽象数据类型的二叉树，只是缺少了查找、插入和删除操作。节点的定义也同以前一样。然而，为了完整和便于参考，我们同时给出了节点和类的定义：

```
      //Definition of the node
template <class elemType>
struct nodeType
{
   elemType             info;
   nodeType<elemType>  *llink;
   nodeType<elemType>  *rlink;
};

       //Definition of the class
template <class elemType>
class binaryTreeType
{
public:
    const binaryTreeType<type>& operator=
                              (const binaryTreeType<type>&);
      //overload the assignment operator
    bool isEmpty();
      //Returns true if the binary tree is empty;
      //otherwise, returns false

    void inorderTraversal();
      //Function to do an inorder traversal of the binary tree
    void preorderTraversal();
      //Function to do a preorder traversal of the binary tree
    void postorderTraversal();
      //Function to do a postorder traversal of the binary tree

    int treeHeight();
      //Returns the height of the binary tree
    int treeNodeCount();
      //Returns the number of nodes in the binary tree
    int treeLeavesCount();
      //Returns the number of leaves in the binary tree

    void destroyTree()
      //Deallocates the memory space occupied by the
      //binary tree
      //Post: root = NULL;

    binaryTreeType(const binaryTreeType<elemType>& otherTree);
      //copy constructor

    binaryTreeType();
      //default constructor

    ~binaryTreeType();
      //destructor

protected:
    nodeType<elemType>  *root;

private:
    void copyTree(nodeType<elemType>* &copiedTreeRoot,
```

```
                    nodeType<elemType>* otherTreeRoot);
          //Makes a copy of the binary tree to which
          //otherTreeRoot points. The pointer copiedTreeRoot
          //points to the root of the copied binary tree.
        void destroy(nodeType<elemType>* &p);
          //Function to destroy the binary tree to which
          //p points
          //Post: p = NULL

        void inorder(nodeType<elemType> *p);
          //Function to do an inorder traversal of the binary
          //tree to which p points
        void preorder(nodeType<elemType> *p);
          //Function to do a preorder traversal of the binary
          //tree to which p points
        void postorder(nodeType<elemType> *p);
          //Function to do a postorder traversal of the binary
          //tree to which p points

        int height(nodeType<elemType> *p);
          //Returns the height of the binary tree
          //to which p points
        int max(int x, int y);
          //Returns the larger of x and y
        int nodeCount(nodeType<elemType> *p);
          //Returns the number of nodes in the binary
          //tree to which p points
        int leavesCount(nodeType<elemType> *p);
          //Returns the number of leaves in the binary
          //tree to which p points

};
```

注意：binaryTreeType类包含了重载赋值运算符函数、拷贝构造函数和析构函数。这是因为binaryTreeType类包含了指针数据成员。回想一下对于包含指针数据成员的类必须做的三件事：明确重载赋值运算符、包含拷贝构造函数、包含析构函数。

binaryTreeType类定义中包含了几个私有成员函数。这些函数被用于实现类的公共成员函数。例如，为了实现中序遍历，函数inorderTraversal调用函数inorder并将指针root作为参数。而且指针root被声明为受保护成员，从而以此可以派生出特定的二叉树类。

接下来，我们给出binaryTreeType的成员函数定义。

如果root为NULL则二叉树为空，所以函数isEmpty定义如下所示：

```
template<class elemType>
bool binaryTreeType<elemType>::isEmpty()
{
      return (root == NULL);
}
```

默认构造函数将二叉树初始化为空，即将指针root设为NULL。所以默认构造函数定义如下所示：

```
template<class elemType>
binaryTreeType<elemType>::binaryTreeType()
{
      root = NULL;
}
```

其他函数的定义为：

```
template<class elemType>
void binaryTreeType<elemType>::inorderTraversal()
{
      inorder(root);
}

template<class elemType>
void binaryTreeType<elemType>::preorderTraversal()
{
      preorder(root);
}

template<class elemType>
void binaryTreeType<elemType>::postorderTraversal()
{
     postorder(root);
}

template<class elemType>
int binaryTreeType<elemType>::treeHeight()
{
      return height(root);
}

template<class elemType>
int binaryTreeType<elemType>::treeNodeCount()
{
      return nodeCount(root);
}

template<class elemType>
int binaryTreeType<elemType>::treeLeavesCount()
{
      return leavesCount(root);
}

template<class elemType>
void binaryTreeType<elemType>::inorder(nodeType<elemType> *p)
{
   if(p != NULL)
   {
      inorder(p->llink);
      cout<<p->info<<" ";
      inorder(p->rlink);
   }
}

template<class elemType>
void binaryTreeType<elemType>::preorder(nodeType<elemType> *p)
{
   if(p != NULL)
   {
      cout<<p->info<<" ";
```

```
        preorder(p->llink);
        preorder(p->rlink);
    }
}

template<class elemType>
void binaryTreeType<elemType>::postorder(nodeType<elemType> *p)
{
    if(p != NULL)
    {
        postorder(p->llink);
        postorder(p->rlink);
        cout<<p->info<<" ";
    }
}

template<class elemType>
int binaryTreeType<elemType>::height(nodeType<elemType> *p)
{
    if(p == NULL)
        return 0;
    else
        return 1 + max(height(p->llink), height(p->rlink));
}

template<class elemType>
int binaryTreeType<type>::max(int x, int y)
{
    if(x >= y)
        return x;
    else
        return y;
}
```

函数nodeCount和leavesCount的定义留给读者作为练习，见本章后面的编程练习1和编程练习2。

接下来给出函数copyTree，destroy，destroyTree，拷贝构造函数，析构函数的定义，同时还重载赋值运算符。

函数copyTree的定义和以前一样，但为binaryTreeType的成员函数。

```
template <class elemType>
void binaryTreeType<elemType>::copyTree
                    (nodeType<elemType>* &copiedTreeRoot,
                     nodeType<elemType>* otherTreeRoot)
{
    if(otherTreeRoot == NULL)
        copiedTreeRoot = NULL;
    else
    {
        copiedTreeRoot = new nodeType<elemType>;
        copiedTreeRoot->info = otherTreeRoot->info;
        copyTree(copiedTreeRoot->llink, otherTreeRoot->llink);
        copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
    }
} //end copyTree
```

为了删除二叉树，对于每个节点，首先要删除它的左子树，然后是右子树，接着是节点本身。必须使用delete运算符来释放分配给节点的内存。函数destroy的定义如下所示：

```
template <class elemType>
void binaryTreeType<elemType>::destroy(nodeType<elemType>* &p)
{
    if(p != NULL)
    {
        destroy(p->llink);
        destroy(p->rlink);
        delete p;
        p = NULL;
    }
}
```

为了实现函数destroyTree，可以使用函数destroy并将二叉树的根节点传递给它。函数destroyTree的定义如下所示：

```
template <class elemType>
void binaryTreeType<elemType>::destroyTree()
{
    destroy(root);
}
```

类对象按值传递时，拷贝构造函数会将实参的值拷贝给形参。由于binaryTreeType类中含有用于产生动态内存的指针数据成员，所以必须提供拷贝构造函数来避免数据的浅拷贝。拷贝构造函数使用函数copyTree生成一个与作为参数传递进来的二叉树完全一样的拷贝。

```
        //copy constructor
template <class elemType>
binaryTreeType<type>::binaryTreeType
              (const binaryTreeType<elemType>& otherTree)
{
    if(otherTree.root == NULL) //otherTree is empty
        root = NULL;
    else
        copyTree(root, otherTree.root);
}
```

析构函数的定义非常简单，当binaryTreeType类对象超出其作用域后，析构函数将释放分配给二叉树节点的内存。析构函数的定义使用了函数destroy。

```
        //destructor
template <class elemType>
binaryTreeType<elemType>::~binaryTreeType()
{
    destroy(root);
}
```

接下来讨论重载赋值运算符函数的定义，为了将一个二叉树的值传递给另一个二叉树，可使用函数copyTree生成一个与原二叉树完全一样的拷贝。重载赋值运算符函数的定义如下所示：

```
        //overloading the assignment operator
template<class elemType>
const binaryTreeType<elemType>& binaryTreeType<elemType>::operator=
                 (const binaryTreeType<type>& otherTree)
{
```

```
    if(this != &otherTree) //avoid self-copy
    {
       if(root != NULL)  //if the binary tree is not empty,
                         //destroy the binary tree
          destroy(root);
       if(otherTree.root == NULL) //otherTree is empty
          root = NULL;
       else
          copyTree(root, otherTree.root);
    }//end else

    return *this;
}
```

19.2 二叉搜索树

到目前为止，我们已经知道二叉树上的基本操作。这一节将讨论一种特殊的二叉树——二叉搜索树。考虑图 19.9 中的二叉树。

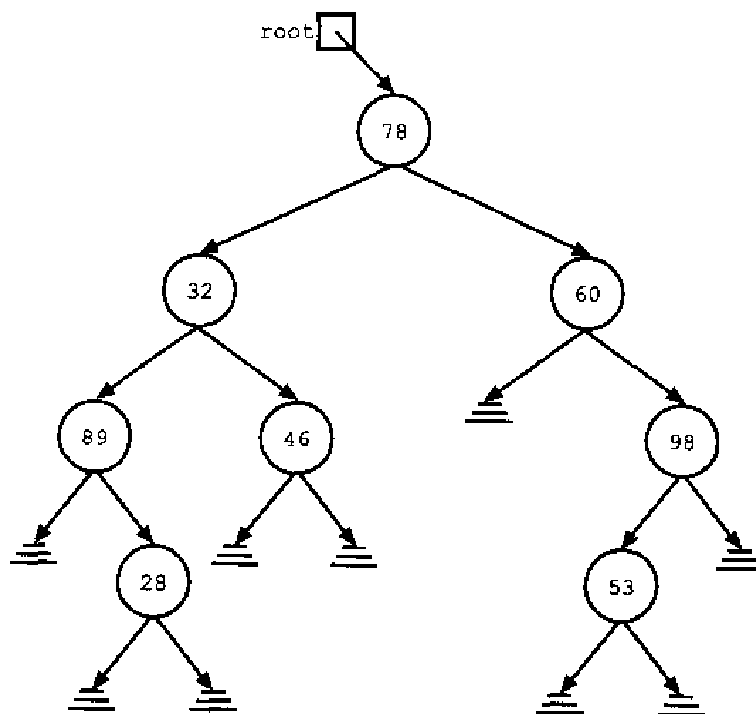


图 19.9 普通二叉树

假设现在要查找53是否在二叉树中。为了实现查找，可以使用任意一种前面介绍过的遍历算法来访问每个节点并将节点中存储的值与要查找的值做比较。然而这需要遍历二叉树的绝大部分节点，所以查找会很慢。由于二叉树上的查找不存在特定的规则，所以必须访问二叉树的每个节点，直到找到该元素或者遍历完整个二叉树为止。这类似于一般的链表，必须从第一个节点开始依次查找每个节点，直到找到该元素或者查找完整个链表为止。

另一方面，考虑图 19.10 中的二叉树。

在图 19.10 中的二叉树，每个节点中存储的数据：

- 比它的左孩子存储的数据大
- 比它的右孩子存储的数据小

图19.10中的二叉树存在着某种结构。假设要查找58是否在该二叉树中，像以前一样，必须从根节点开始查找。先用根节点存储的值与58比较，也就是用60与58比较。因为58 ≠ 60且58<60，说明58不可能在根节点的右子树中。所以如果58在二叉树中的话，它肯定在根节点的左子树中。顺着根节点指向左孩子的指针到达值为50的节点并应用同样的查找规则。由于58>50，所以必须顺着该节点的右孩子的指针查找值为58的节点，最后找到元素58。

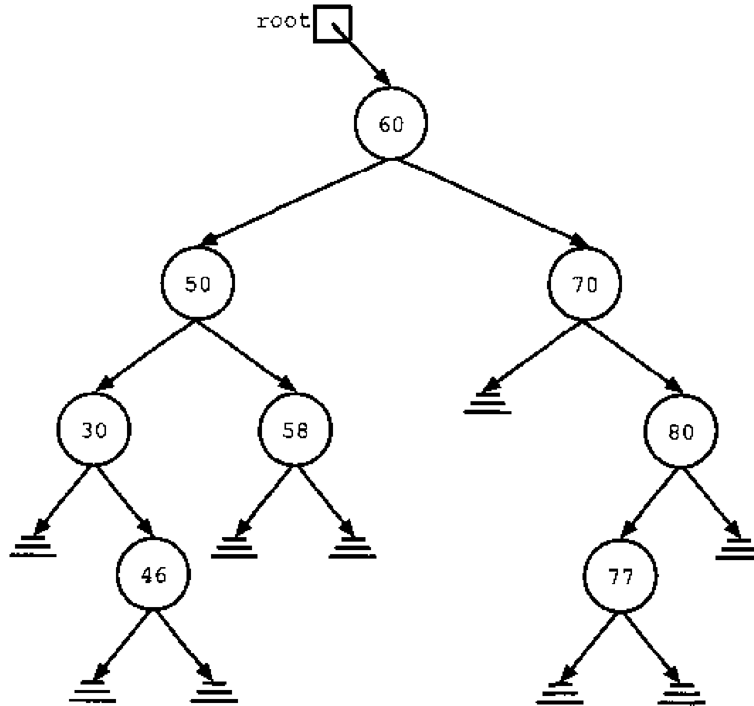


图19.10 二叉搜索树

上例说明每次向下移动到一个子节点就会使查找减少一个子树。如果二叉树组织得很好的话，那查找就与数组中的折半算法查找非常相似。

图19.10中的二叉树是一种特殊的二叉树——二叉搜索树（在下面的定义中，我们用术语“节点的键”来表示能够惟一标识元素的数据键）。

定义 二叉搜索树（Binary Search Tree）T，或者为空或者：

(i) T有一个称为根节点的特殊节点。

(ii) T有两个节点集合：L_T和R_T，分别称为T的左子树和右子树。

(iii) 根节点的键值大于左子树中任一节点的键值，小于右子树中任一节点的键值。

(iv) L_T和R_T均是二叉搜索树。

注意： 在定义中，由于根节点的左子树和右子树也是二叉搜索树，所以(iii)等价于：根节点的键值大于左孩子（如果存在的话）的键值，小于右孩子（如果存在的话）的键值。

下面给出了二叉搜索树中的典型操作：

1. 检验二叉搜索树是否为空
2. 在二叉搜索树中查找指定的元素
3. 在二叉搜索树中插入元素
4. 从二叉搜索树中删除元素

5. 获得二叉搜索树的高度
6. 获得二叉搜索树的节点数目
7. 获得二叉搜索树的叶节点数目
8. 遍历二叉搜索树
9. 拷贝二叉搜索树

显而易见，每个二叉搜索树都是二叉树。二叉搜索树高度与二叉树高度的计算方法相同。类似地，对于二叉搜索树，获得节点数目、获得叶节点数目、中序遍历、前序遍历、后序遍历的方法也和二叉树完全一样。因此，可以从二叉树直接继承这些操作。

下面的类通过扩展二叉树的定义将二叉搜索树定义为抽象数据类型（ADT）：

```
template <class elemType>
class bSearchTreeType: public binaryTreeType<elemType>
{
public:
    bool search(const elemType& searchItem);
      //Returns true if searchItem is found in the binary
      //search tree; otherwise, returns false
    void insert(const elemType& insertItem);
      //If no node in the binary search tree has the same info
      //as insertItem, a node with the info insertItem is
      //created and inserted in the binary search tree
    void deleteNode(const elemType& deleteItem);
      //If a node with the same info as deleteItem is
      //found, it is deleted from the binary search tree

private:
    void deleteFromTree(nodeType<elemType>* &p);
      //The node to which p points is deleted from the
      //binary search tree
};
```

接下来，我们逐一介绍这些操作。

查找

函数search用于在二叉搜索树中查找指定元素。如果在二叉搜索树中查找到该元素，函数返回true；否则，返回false。由于指针root指向二叉搜索树的根节点，所以必须从根节点开始查找。另外，由于root总是指向根节点，所以需要一个指针current遍历二叉搜索树。current初始值为root。

如果二叉搜索树非空，则首先将要查找的元素与根节点中info的值做比较。如果相同，则停止查找并返回true。否则，如果要查找的元素小于根节点中info的值，则顺着指针llink到达左子树；否则顺着rlink到达右子树。对下一个节点重复上述过程。如果要查找的节点在二叉搜索树中，则查找将在找到该节点时结束；否则，查找将在遇到一个空子树时结束。算法的伪代码如下所示：

```
if root is NULL
   Cannot search an empty tree, returns false.
else
{
    current = root;
    while(current is not NULL and not found)
      if(current->info is the same as the search item)
         set found to true;
      else
```

```
        if(current->info is greater than the search item)
            follow the llink of current
      else
            follow the rlink of current
}
```

上面的伪代码转换成C++函数如下所示：

```
template<class elemType>
bool bSearchTreeType<elemType>::search(const elemType& searchItem)
{
    nodeType<elemType> *current;
    bool found = false;

    if(root == NULL)
       cout<<"Cannot search the empty tree."<<endl;
    else
    {
       current = root;

       while(current != NULL && !found)
       {
             if(current->info == searchItem)
                found = true;
             else
                if(current->info > searchItem)
                   current = current->llink;
                else
                   current = current->rlink;
        }//end while
    }//end else

    return found;
}//end search
```

插入

一个二叉搜索树在插入一个元素后必须仍是二叉搜索树。为了插入一个新元素，首先要查找二叉树找到新元素应该插入的位置。查找的算法与函数search的算法类似。这里使用两个指针来遍历二叉树。其中一个指针为current，用来表示指向的当前节点。另一个为trailCurrent，指向current的父节点。由于不允许有重复的元素，所以查找一定在遇到一个空子树时结束。要插入的元素insertItem作为参数传递给函数insert。算法如下所示：

a. 创建一个新节点，将元素insertItem拷贝给新节点，并将新节点的llink和rlink设置为NULL。

b. if root为NULL，则树为空，root指向新节点。

```
else
{
    current = root;
    while(current is not NULL)    //search the binary tree
    {
          trailCurrent = current;
          if(current->info is the same as the insertItem)
             Error: Cannot insert duplicate
             exit
          else
```

```
            if(current->info > insertItem)
               Follow llink of current
            else
               Follow rlink of current
    }

    //insert the new node in the binary tree

    if(trailCurrent->info > insertItem)
       make the new node the left child of trailCurrent
    else
       make the new node the right child of trailCurrent
 }
```

上面的伪代码转换成C++函数后如下所示：

```
template<class elemType>
void bSearchTreeType<elemType>::insert(const elemType& insertItem)
{
   nodeType<elemType> *current;  //pointer to traverse the tree
   nodeType<elemType> *trailCurrent; //pointer behind current
   nodeType<elemType> *newNode;  //pointer to create the node

   newNode = new nodeType<elemType>;
   newNode->info = insertItem;
   newNode->llink = NULL;
   newNode->rlink = NULL;
   if(root == NULL)
      root = newNode;
   else
   {
      current = root;

      while(current != NULL)
      {
           trailCurrent = current;

           if(current->info == insertItem)
           {
              cout<<"The insert item already is in the list — ";
              cout<<"duplicates are not allowed."<<insertItem
                  <<endl;
              return;
           }
           else
              if(current->info > insertItem)
                 current = current->llink;
              else
                 current = current->rlink;
      }//end while

     if(trailCurrent->info > insertItem)
        trailCurrent->llink = newNode;
     else
        trailCurrent->rlink = newNode;
   }
}//end insert
```

删除

同前面一样，首先要在二叉搜索树中找到要删除的节点。为了能更好地理解删除操作，在介绍用来删除二叉搜索树中元素的函数前，先考虑图19.11中的二叉搜索树。

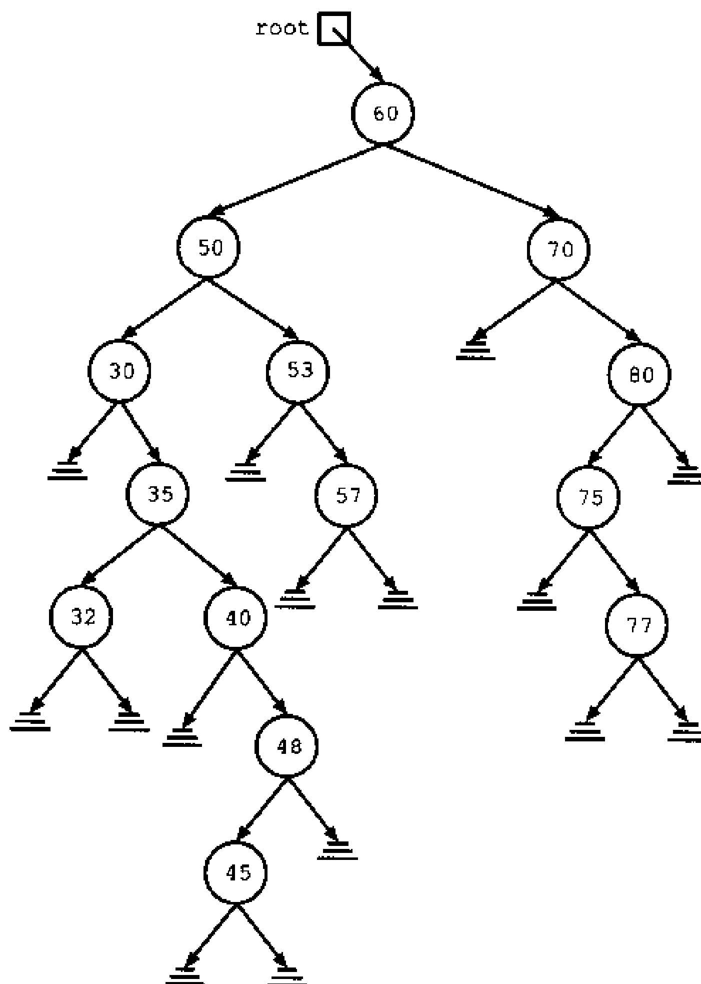


图19.11　删除节点前的二叉搜索树

在删除指定的元素后（如果该元素在二叉搜索树中的话），该树必须仍为二叉搜索树。删除操作有4种情况：

1. 要删除的节点既没有左子树也没有右子树，即要删除的节点是叶节点。
2. 要删除的节点没有左子树，即左子树为空但右子树非空。
3. 要删除的节点没有右子树，即右子树为空但左子树非空。
4. 要删除的节点左右子树均非空。

情况1　假设要从图19.11中所示的二叉搜索树中删除45。查找二叉搜索树找到含有45的节点。由于该节点是叶节点，并且是其父节点的左孩子，所以只需将其父节点的llink指针置为NULL，并释放分配给该节点的内存即可。在删除元素后，得到的二叉搜索树如图19.12所示。

情况2　假设要从图19.11中所示的二叉搜索树中删除30。在这种情况下要删除的节点没有左子树。由于30是其父节点的左孩子，所以将其父节点的llink指针指向30的右孩子，并释放分配给30的内存。在删除元素后，得到的二叉搜索树如图19.13所示。

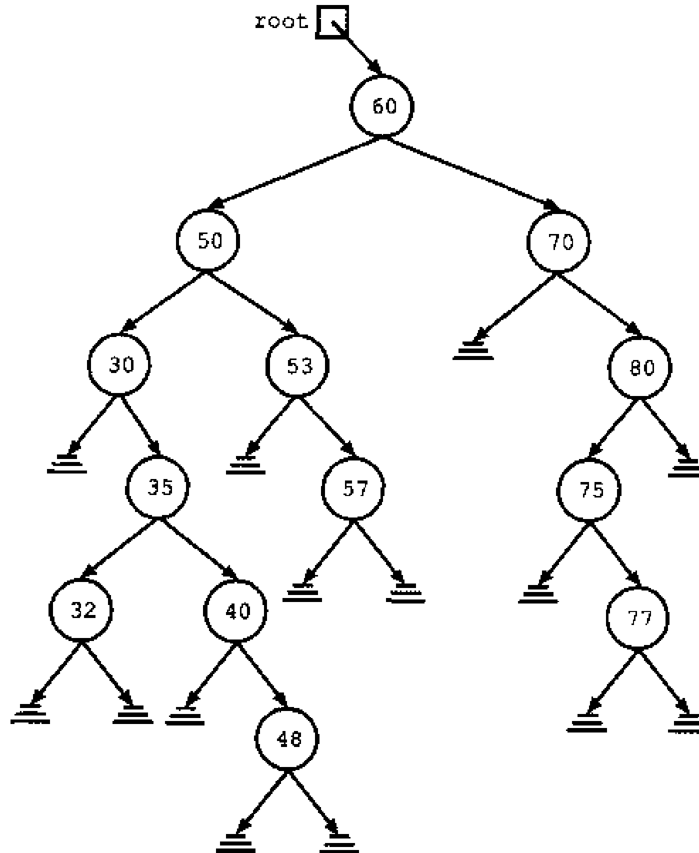


图19.12　删除45后的二叉搜索树

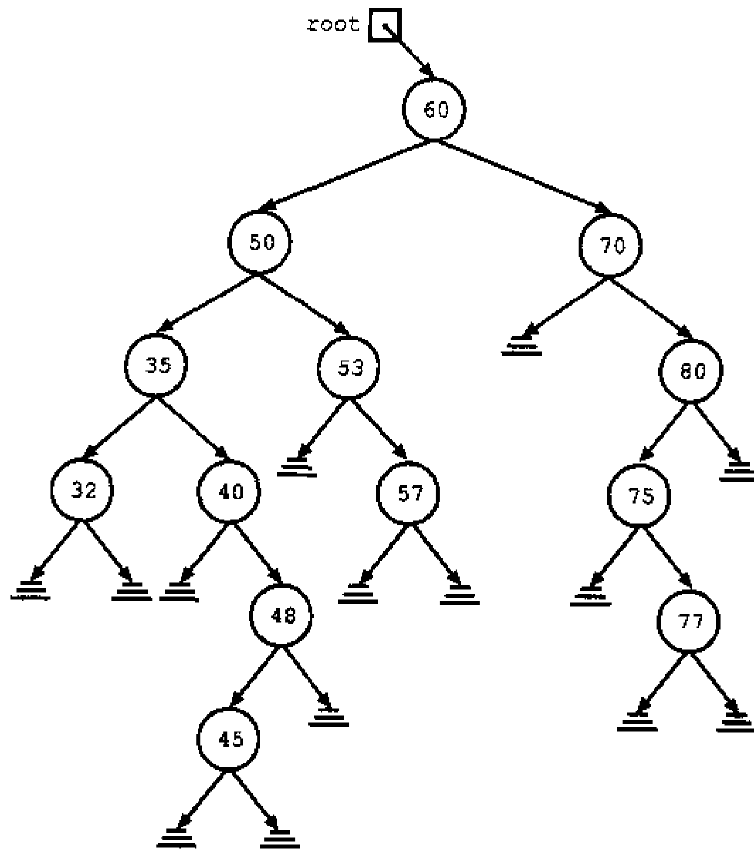


图19.13　删除30后的二叉搜索树

情况 3 假设要从图 19.11 中所示的二叉搜索树中删除 80。包含 80 的节点没有右子树，并且是其父节点的右孩子。所以将其父节点 70 的 rlink 指针指向 80 的左孩子，并释放分配给 80 的内存。在删除元素后，得到的二叉搜索树如图 19.14 所示。

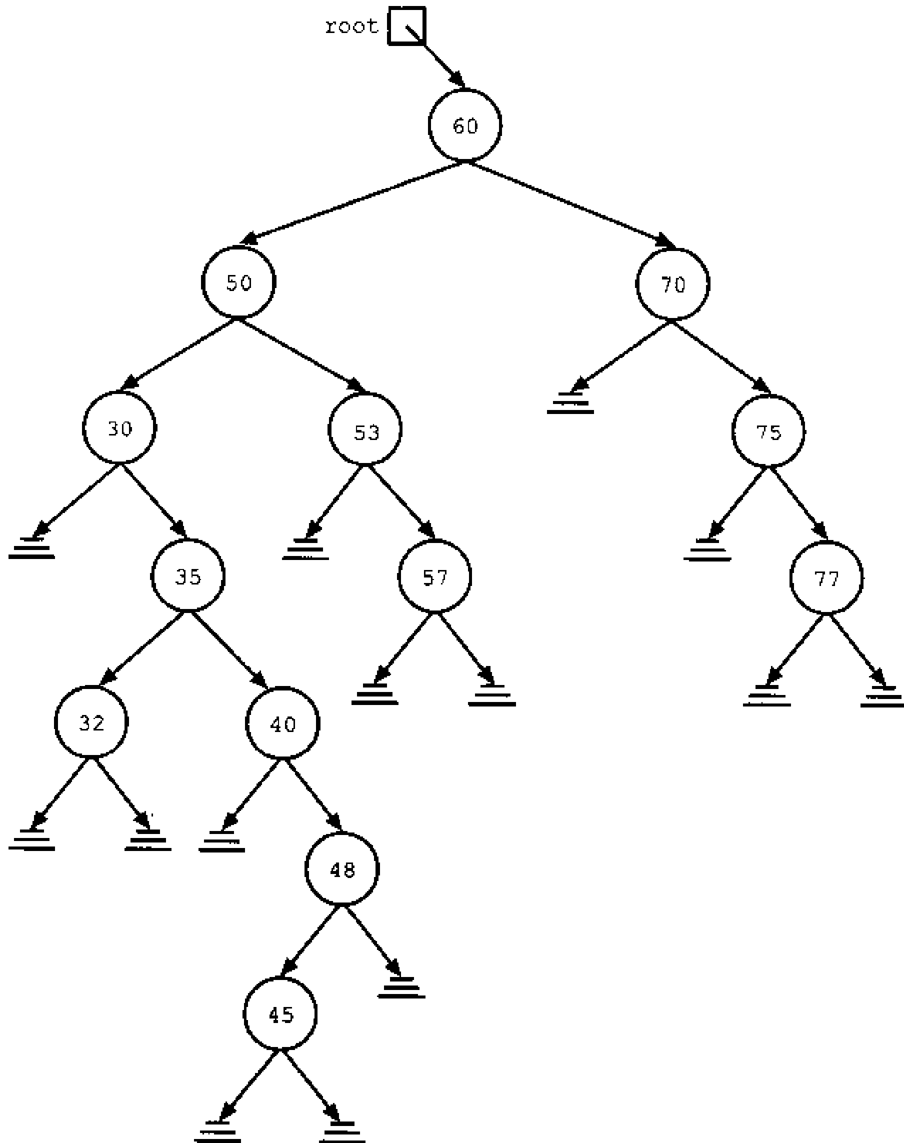


图 19.14 删除 80 后的二叉搜索树

情况 4 假设要从图 19.11 中所示的二叉搜索树中删除 50。info 为 50 的节点有一个非空的左子树和一个非空的右子树。首先要将这种情况简化为情况 2 或者情况 3。为了明确，假设要简化为情况 3，也就是说简化成要删除的节点没有右子树的情况。为此，首先找出 50 在二叉树中的直接前趋——48。这可通过首先移到 50 的左孩子，然后找到 50 的左子树中的最右节点来获得该节点。可以顺着 50 的左孩子节点的 rlink 指针查找。由于二叉搜索树是有穷的，所以最终能够到达一个没有右子树的节点。接下来，将要删除的节点和它的直接前趋的 info 交换。在这个二叉树中交换的是 48 和 50。这样就简化成了要删除的节点是没有右子树的情况。现在可以对要删除的节点应用情况 3 中的方法（注意：由于实际要从二叉搜索树中删掉直接前趋所在的节点，所以只需拷贝直接前趋的 info 至要删除的节点即可）在删除 50 后，得到的二叉搜索树如图 19.15 所示。

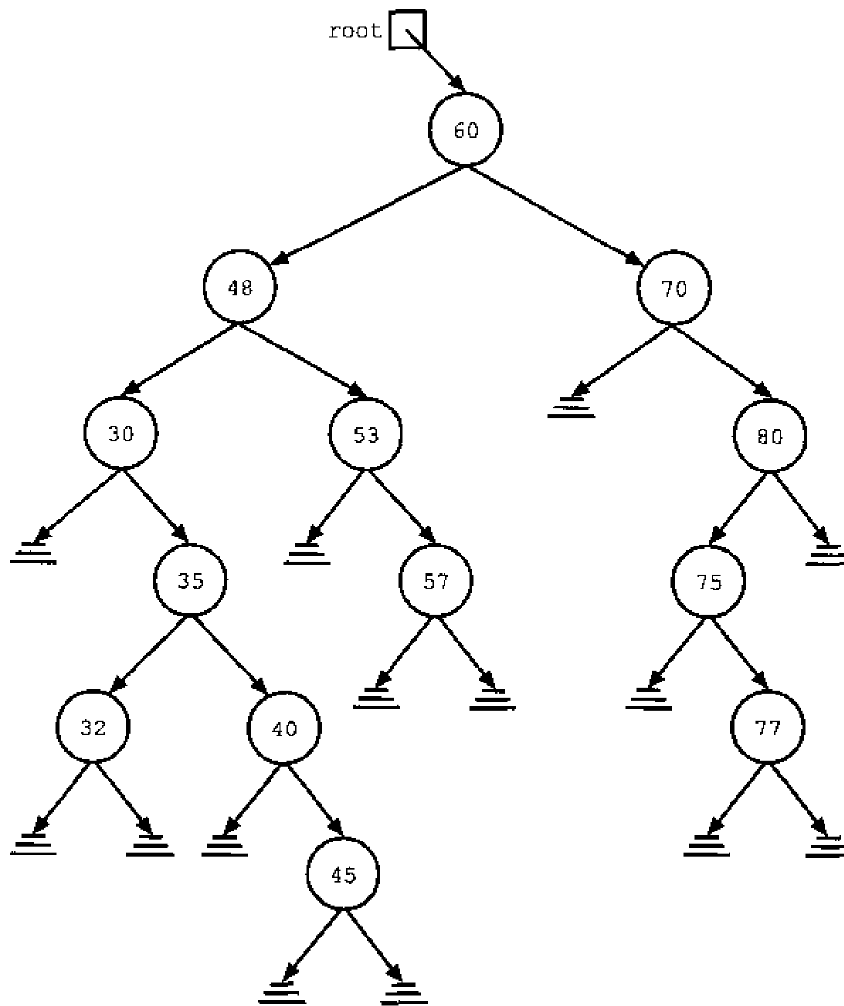


图 19.15 删除50后的二叉搜索树

很明显，不管是哪一种情况，得到的二叉树仍是二叉搜索树。

从以上讨论可以看出，为了从二叉搜索树中删除一个元素，必须：

1. 找到包含要删除元素的节点（如果存在的话）
2. 删除该节点

第2步可以用一个单独的函数 deleteFromTree 来完成。给定一个要删除节点的指针，这个函数能够针对以上4种情况删除该节点。

从以上的例子可以看出，无论什么时候要从二叉搜索树中删除一个节点，都要调整它的父节点的指针。由于调整是针对父节点的，所以必须使用相应父节点的指针来调用函数 deleteFromTree。例如，假设要删除的元素是35，它是其父节点的右孩子。假设 trailCurrent 指向35的父节点，该节点包含元素30。函数 deleteFromTree 的调用形式为：

```
deleteFromTree(trailCurrent->rlink);
```

当然，如果要删除的节点为根节点，则函数 deleteFromTree 的调用形式为：

```
deleteFromTree(root);
```

函数 deleteFromTree 定义如下所示：

```
template<class elemType>
void bSearchTreeType<elemType>::deleteFromTree(nodeType<elemType>* &p)
{
      nodeType<elemType> *current;     //pointer to traverse
                                       //the tree
      nodeType<elemType> *trailCurrent;   //pointer behind current
      nodeType<elemType> *temp;        //pointer to delete the node

     if(p == NULL)
        cout<<"Error: The node to be deleted is NULL"
            <<endl;
     else if(p->llink == NULL && p->rlink == NULL)
          {
             temp = p;
             p = NULL;
             delete temp;
          }
     else if(p->llink == NULL)
          {
             temp = p;
             p = temp->rlink;
             delete temp;
          }
     else if(p->rlink == NULL)
          {
             temp = p;
             p = temp->llink;
             delete temp;
          }
     else
     {
        current = p->llink;
        trailCurrent = NULL;

        while(current->rlink != NULL)
        {
              trailCurrent = current;
              current = current->rlink;
        } //end while

        p->info = current->info;

        if(trailCurrent == NULL) //current did not move;
                                 //current == p->llink; adjust p
           p->llink = current->llink;
        else
           trailCurrent->rlink = current->llink;

        delete current;
     }//end else
} //end deleteFromTree
```

接下来，描述函数 deleteNode。函数 deleteNode 首先在二叉搜索树中找出要删除元素的节点。要删除的元素 deleteItem 作为参数传递给该函数。如果在二叉搜索树中找到包含要删除的元素 deleteItem 的节点，函数 deleteNode 调用函数 deleteFromTree 来删除该节点。函数 deleteNode 的定义如下所示：

```
template<class elemType>
void bSearchTreeType<elemType>::deleteNode(const elemType& deleteItem)
{
      nodeType<elemType> *current;  //pointer to traverse the tree
      nodeType<elemType> *trailCurrent; //pointer behind current
      bool found = false;
      if(root == NULL)
         cout<<"Cannot delete from the empty tree."<<endl;
      else
      {
         current = root;
         trailCurrent = root;

         while(current != NULL && !found)
         {
               if(current->info == deleteItem)
                  found = true;
               else
               {
                  trailCurrent = current;

                     if(current->info > deleteItem)
                        current = current->llink;
                  else
                     current = current->rlink;
               }
         } //end while

         if(current == NULL)
            cout<<"The delete item is not in the list."<<endl;
         else
            if(found)
            {
               if(current == root)
                  deleteFromTree(root);
               else
                  if(trailCurrent->info > deleteItem)
                     deleteFromTree(trailCurrent->llink);
               else
                  deleteFromTree(trailCurrent->rlink);
            } //end if
      }
} //end deleteNode
```

19.2.1 二叉搜索树：分析

用T表示一个有n个节点（n>0）的二叉搜索树。假设现在要查找元素x是否在T中。查找的性能主要取决于T的形状。首先考虑最差的情况。在最坏的情况下，T是线性的。也就是说T是图19.16中所示的形状之一。

由于T是线性的，T的查找算法性能与线性链表相同。所以，在查找成功的情况下，查找算法平均要做$\frac{n+1}{2}$次比较。在查找不成功的情况下，做n次比较。

现在考虑一般情况。在查找成功的情况下，查找会在某个节点处停止。由于有n个元素。所以会有$n!$种可能的键值排列次序。假设所有的$n!$个次序都有可能。用$S(n)$表示在一般查找成功的情况下比较的次数，$U(n)$表示在一般查找失败情况下比较的次数。

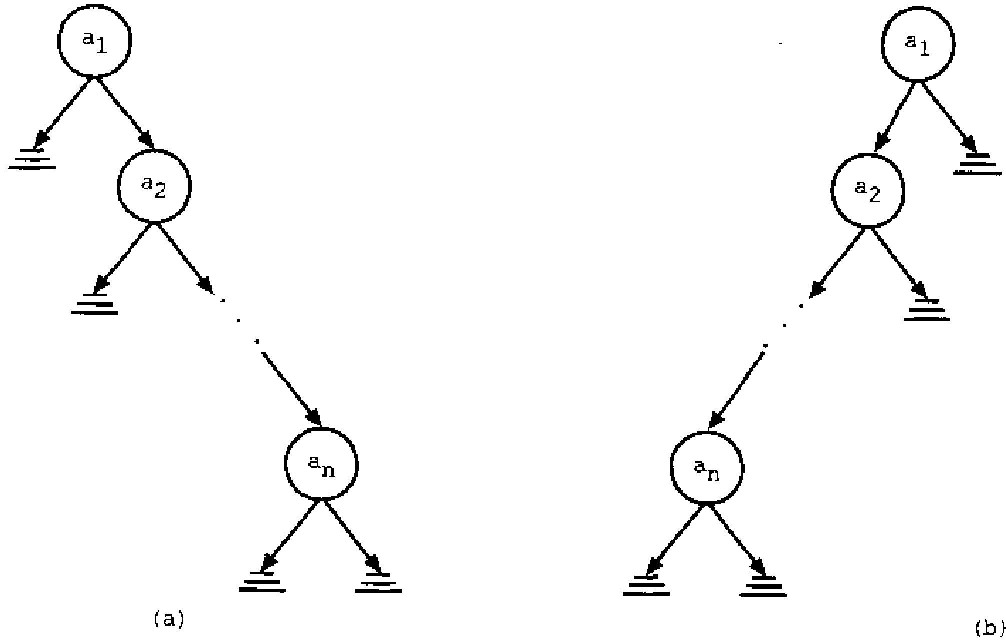


图 19.16 线性二叉树

查找x是否在T中所需的比较次数比将x插入到T中所需的比较次数多1。另外，将x插入到T中所需的比较次数与在查找失败情况下（即x不在T中）所需的比较次数相同。由此引出公式：

$$S(n)=1+\frac{U(0)+U(1)+\cdots+U(n-1)}{n} \tag{19.1}$$

同样：

$$S(n)=\left(1+\frac{1}{n}\right)U(n)-3 \tag{19.2}$$

对公式（19.1）和公式（19.2）求值可得：

$$U(n) \approx 2.77 \log_2 n$$

和

$$S(n) \approx 1.39 \log_2 n$$

由此可以得出如下结论。

定理 用T表示具有n（n>0）个节点的二叉搜索树。在对T进行查找时平均的节点访问次数约等于$1.39 \log_2 n$。

19.3 非递归二叉树遍历算法

前面几节主要介绍如何：

- 使用中序遍历、前序遍历和后序遍历来遍历二叉树
- 构造一个二叉树
- 将一个元素插入到二叉树中
- 从二叉树中删除一个元素

前面讨论的二叉树遍历算法（包括前序、中序、后序遍历）都是递归的。由于遍历二叉树是基本的操作，所以这一节将讨论非递归的中序、前序、后序遍历算法。

19.3.1 非递归的中序遍历

在中序遍历中，对于每个节点都是先访问左子树，然后是节点，最后访问右子树。因此对于中序遍历，访问的第一个节点是二叉树中的最左节点。例如，在图19.17的二叉树中，最左的节点的info为28。

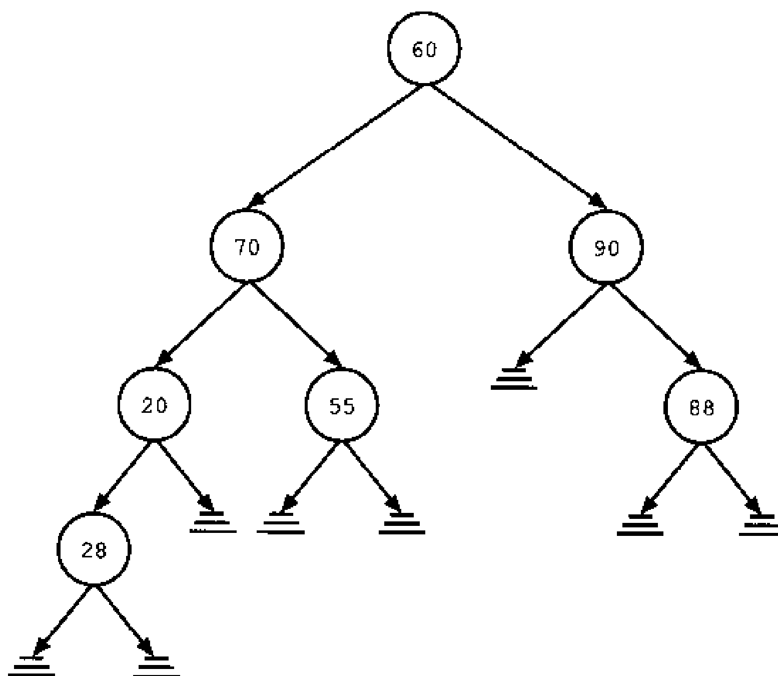


图19.17 二叉树：最左节点为28

为了获得二叉树的最左孩子，可以先从二叉树的根节点开始遍历，顺着每个节点的左指针向下直到某个节点的左指针为空为止。从这里开始，返回父节点，访问该节点，然后移到右孩子。由于节点链接是单向的，所以为了能够返回到某个节点，必须在移到其子节点前保存指向该节点的指针。另外，节点还必须能够按照遍历的顺序返回。因此节点在返回时应该按照后进先出方式，这可以用堆栈实现。所以使用堆栈来保存指针，算法如下所示：

```
1. current = root; //start traversing the binary tree at the root node
2. while(current is not NULL and stack is nonempty)
   if(current is not NULL)
   {
      push current onto stack;
      current = current->llink;
   }
   else
   {
      pop stack into current;
      visit current;        //visit the node
      current = current->rlink;  //move to the right child
   }
```

下面函数实现了二叉树的非递归中序遍历算法：

```
template <class elemType>
void binaryTreeType<elemType>::nonRecursiveInTraversal()
{
   stackType<nodeType<elemType>*> stack;
   nodeType<elemType> *current;
   current = root;

   while((current != NULL) || (!stack.isEmptyStack()))
      if (current != NULL)
      {
         stack.push(current);
         current = current->llink;
      }
      else
      {
         stack.pop(current);
         cout<<current->info<<" ";
         current = current->rlink;
      }

   cout<<endl;
}
```

19.3.2 非递归的前序遍历

在前序遍历中，对每个节点都是先访问节点，然后是左子树，最后访问右子树。与中序遍历一样，在访问节点之后并在移到其左子树之前，必须保存该节点的指针以便在访问完左子树之后还能够访问右子树。算法如下所示：

```
1. current = root; //start the traversal at the root node
2. while(current is not NULL or stack is nonempty)
   if(current is not NULL)
   {
      visit current node;
      push current onto stack;
      current = current->llink;
   }
   else
   {
      pop stack into current;
      current = current->rlink;  //prepare to visit the right subtree
   }
```

下面函数实现二叉树的非递归前序遍历算法：

```
template <class elemType>
void binaryTreeType<elemType>::nonRecursivePreTraversal()
{

   stackType<nodeType<elemType>*> stack;
   nodeType<elemType> *current;
```

```
    current = root;

    while((current != NULL) || (!stack.isEmptyStack()))
          if(current != NULL)
          {
             cout<<current->info<<" ";
             stack.push(current);
             current = current->llink;
          }
          else
          {
             stack.pop(current);
             current = current->rlink;
          }

    cout<<endl;
}
```

19.3.3 非递归的后序遍历

在后序遍历中，对于每个节点都是先访问左子树，然后是右子树，最后访问节点本身。与中序遍历一样，在后序遍历中，第一个访问的节点是二叉树的最左孩子。由于对于每个节点都是在访问节点之前访问其左右子树，所以必须标明节点是否已经访问了左子树和右子树。在访问左子树之后并在访问节点之前，必须先访问其右子树。因此，在从左子树返回之后，必须通知节点现在要访问的是右子树。而在访问完右子树之后，要通知节点现在要访问的是它自己。为了做到这点，除了保存节点的指针（为了移动到节点的右子树和节点本身）外，还要保存一个整数值，在移动到左子树之前将其置为1，在移动到右子树之前将其置为2。在堆栈弹出指针时，与该指针相关联的整数也要被弹出。这个整数值用来指示要访问的是节点的右子树还是节点本身。

算法如下所示：

```
1. current = root; //start the traversal at the root node
2. v = 0;
3. if current is NULL
   The binary tree is empty
4. if current is not NULL
   a. push current onto stack;
   b. push 1 onto stack;
   c. current = current->llink;
   d. while(stack is not empty)
      if(current is not NULL and v is 0)
      {
         push current and 1 onto stack;
         current = current->llink;
      }
      else
      {
         pop stack into current and v;
         if(v == 1)
         {
```

```
            push current and 2 onto stack;
            current = current->rlink;
            v = 0;
        }
        else
            visit current;
    }
```

上面算法用到两个（并行的）堆栈：一个保存指向节点的指针；另一个保存与该指针关联的整数值（1或2）。下面函数实现二叉树的非递归后序遍历算法：

```
template <class elemType>
void binaryTreeType<elemType>::nonRecursivePostTraversal()
{
   stackType<nodeType<elemType>*> stack;

   stackType<int> intStack;

   nodeType<elemType> *current = root;

   int v = 0;

   if(current == NULL)
      cout<<"The binary tree is empty"<<endl;
   else
   {
      stack.push(current);
      intStack.push(1);
      current = current->llink;
      while(!stack.isEmptyStack() && !intStack.isEmptyStack())
          if(current != NULL && v == 0)
          {
             stack.push(current);
             intStack.push(1);
             current = current->llink;
          }
          else
          {
             stack.pop(current);
             intStack.pop(v);

             if(v == 1)
             {
                stack.push(current);
                intStack.push(2);
                current = current->rlink;
                v = 0;
             }
             else
                cout<<current->info<<" ";
          }
   }

   cout<<endl;
}
```

19.4　二叉树遍历与函数参数

假设在二叉搜索树中存储了雇员的信息，到年底时要为每个雇员加薪或发奖金。这需要访问二叉搜索树中的每个节点并更新雇员的薪水。前面章节讨论了几种遍历二叉树的方法。然而在这些遍历算法中（中序遍历、前序遍历、后序遍历），为了简便和突出重点，在访问节点时我们只是输出该节点包含的数据。怎么才能通过一个遍历算法来访问并更新每个节点的数据呢？一种方法就是再创建一个二叉搜索树，该二叉搜索树中每个节点的数据都是对原二叉搜索树中节点数据的更新，然后再删除原二叉搜索树。这需要额外的计算时间和额外的内存，所以效率较低。另一个方法就是写一个单独的遍历算法来更新数据。这种方法需要不断地修改用以实现二叉搜索树的类的定义。然而，如果用户能够提供一个用以更新每个雇员数据的函数，并将该函数作为参数传递给遍历算法，就可以大大地提高程序的灵活性。这一节将介绍如何将一个函数作为参数传递给另一个函数。

在C++中，没有括号的函数名被认为是一个指向该函数的指针。将一个函数指定为另一个函数的形参的方法是：指明函数的类型，后面加上作为指针的函数名，最后是该函数的参数。例如下面的语句：

```
void fParamFunc1(void (*visit) (int));          //Line 1
void fParamFunc2(void (*visit) (elemType&));    //Line 2
```

第1行语句声明函数fParamFunc1的参数可以是任何拥有一个int类型值参数的void类型函数。第2行语句声明函数fParamFunc2的参数可以是任何拥有一个elemType类型的引用参数的void类型函数。

现在可以重写类binaryTreeType的中序遍历函数，也可以重载已有的中序遍历算法。为了进一步说明函数的重载，我们将重载中序遍历函数。因此，应在类binaryTreeType的定义中包含如下语句：

```
void inorderTraversal(void (*visit) (elemType&));
  //This function does an inorder traversal of the binary tree.
  //The parameter visit, which is a function, specifies the
  //action to be taken at each node.

void inorder(nodeType<elemType> *p, void (*visit) (elemType&));
  //This function does an inorder traversal of the binary
  //tree, starting at the node specified by the parameter p.
  //The parameter visit, which is a function, specifies the
  //action to be taken at each node.
```

这些函数的定义如下所示：

```
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal(
                        void (*visit) (elemType& item))
{
      inorder(root, *visit);
}

template <class elemType>
void binaryTreeType<elemType>::inorder(nodeType<elemType>* p,
                               void (*visit) (elemType& item))
{
      if(p != NULL)
      {
            inorder(p->llink, *visit);
            (*visit)(p->info);
            inorder(p->rlink, *visit);
      }
}
```

函数 inorder 定义中的语句：

```
(*visit)(p->info);
```

实现了对指针 visit 所指的拥有一个 elemType 类型的引用参数的函数的调用。

例 19.6 本例说明了如何将一个用户定义的函数作为参数传递给二叉树遍历算法。为了说明重点，只用到了中序遍历函数。

下面的程序使用了从binaryTreeType 类派生出类bSearchTreeType，用以构建一个二叉树。遍历函数包含在 binaryTreeType 类中，因此它们也被类 bSearchTreeType 所继承。

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

using namespace std;

void print(int& x);
void update(int& x);

int main()
{
      bSearchTreeType<int>  treeRoot;                     //Line 1

      int num;                                            //Line 2

      cout<<"Line 3: Enter numbers ending with -999"
          <<endl;                                         //Line 3
      cin>>num;                                           //Line 4

      while(num != -999)                                  //Line 5
      {
            treeRoot.insert(num);                         //Line 6
            cin>>num;                                     //Line 7
      }

      cout<<endl<<"Line 8: Tree nodes in inorder: "; //Line 8
      treeRoot.inorderTraversal(print);                   //Line 9
      cout<<endl<<"Line 10: Tree Height: "
          <<treeRoot.treeHeight()
          <<endl<<endl;                                   //Line 10
      cout<<"Line 11: ******* Update Nodes *******"
          <<endl;                                         //Line 11
      treeRoot.inorderTraversal(update);                  //Line 12

      cout<<"Line 13: Tree nodes in inorder after "
          <<"the update: "<<endl<<"          ";          //Line 13
      treeRoot.inorderTraversal(print);                   //Line 14
      cout<<endl<<"Line 15: Tree Height: "
          <<treeRoot.treeHeight()
          <<endl;                                         //Line 15

      return 0;                                           //Line 16
}

void print(int& x)                                        //Line 17
{
```

```
        cout<<x<<" ";                                    //Line 18
}

void update(int& x)                                      //Line 19
{
        x = 2 * x;                                       //Line 20
}
```

程序运行结果 在本程序运行中，用户输入的数据加有阴影。

```
Line 3: Enter numbers ending with -999
56 87 23 65 34 45 12 90 66 -999

Line 8: Tree nodes in inorder: 12 23 34 45 56 65 66 87 90
Line 10: Tree Height: 4

Line 11: ******* Update Notes *******
Line 13: Tree nodes in inorder after the update:
         24 46 68 90 112 130 132 174 180
Line 15: Tree Height: 4
```

第1行语句将treeRoot声明为一个二叉搜索树对象，其中的每个节点的数据类型均为int。第4行到第7行语句生成二叉搜索树。第9行语句使用了treeRoot的成员函数inorderTraversal来遍历二叉搜索树treeRoot。第9行中函数inorderTraversal的参数为函数print（在第17行定义）。因为函数print输出参数的值，所以第9行输出二叉搜索树中所有节点的数据。第10行语句输出了二叉搜索树的高度。

第12行语句使用成员函数inorderTraversal来遍历二叉搜索树treeRoot。在第12行中，函数inorderTraversal的实参为函数update（在第19行定义）。函数update将其参数的值增加一倍。所以第12行语句将二叉搜索树中的每个节点的值都乘以2。第14行和第15行语句输出二叉搜索树的所有节点和树的高度。

19.5 程序范例：影碟店

在第16章中，我们设计了一个影碟店处理影碟出租业务的程序。程序使用一个（无序）链表来维护影碟店中的影碟清单。由于链表的查找算法是顺序的，当链表非常大时，查找会非常费时。本章中介绍了如何用二叉树组织数据。如果二叉树被很好地构造（也就是说是非线性的），查找算法会有相当大的改进。另外，一般来说，在二叉树中插入和删除元素会比链表快。所以，在此要重新设计影碟店程序，改用二叉树来维护影碟清单。与第16章一样，用二叉树记录客户清单的设计留给读者作为练习。另外，程序使用了第15章中设计实现的newString类来操作字符串。

影碟对象 在第16章中，使用了一个链表来维护影碟店中的影碟清单。由于链表无序，为了查看影碟店中是否有某影碟，顺序查找算法使用了恒等运算符来做比较。然而，在使用二叉树的情况下，查找还需要其他的关系运算符。所以，需要重载其他的关系运算符。除了这些不同之外，videoType类和以前一样。但为了完整，这里仍给出了它的全部定义。

```
class videoType
{
    friend ostream& operator<<(ostream&, const videoType&);

public:
    void setVideoInfo(newString title, newString star1,
                      newString star2, newString producer,
```

```
                    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 checks out a video.
      //The number of copies in stock is decremented by 1.
    void checkIn();
      //This function checks in a video.
      //The number of copies in stock is incremented by 1.
    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
       //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;

        //overload relational operators
    bool operator==(const videoType&) const;
    bool operator!=(const videoType&) const;
```

```
    bool operator<(const videoType&) const;
    bool operator<=(const videoType&) const;

    bool operator>(const videoType&) const;
    bool operator>=(const videoType&) const;

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
};
```

videoType类的成员函数定义与第16章中的定义一样。由于要重载所有的关系运算符，所以这里只给出了这些运算符的定义。

```
    //Overload the relational operators
bool videoType::operator==(const videoType& right) const
{
    return (videoTitle == right.videoTitle);
}

bool videoType::operator!=(const videoType& right) const
{
    return (videoTitle != right.videoTitle);
}

bool videoType::operator<(const videoType& right) const
{
    return (videoTitle < right.videoTitle);
}

bool videoType::operator<=(const videoType& right) const
{
    return (videoTitle <= right.videoTitle);
}

bool videoType::operator>(const videoType& right) const
{
    return (videoTitle > right.videoTitle);
}

bool videoType::operator>=(const videoType& right) const
{
    return (videoTitle >= right.videoTitle);
}
```

影碟清单 这里使用二叉搜索树来维护影碟清单。所以可以从bSearchTreeType中派生videoBinaryTree类。videoBinaryTree类定义如下所示：

```
class videoBinaryTree: public bSearchTreeType<videoType>
{
public:
    bool videoSearch(newString title);
      //This function searches the list to see whether a
      //particular title, specified by the parameter
      //title, is in stock.
      //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 stock, false otherwise.
    void videoCheckOut(newString title);
      //This function checks out a video, that is, rents
      //a video.
      //Post: copiesInStock is decremented by 1.
    void videoCheckIn(newString title);
      //This function checks in a video returned by a customer.
      //Post: copiesInstock is incremented by 1.
    bool videoCheckTitle(newString title);
      //This function returns true if a particular video is
      //in stock, false otherwise.
    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 videos in
      //stock.

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, false otherwise. The parameter current points
      //to the node containing the video.
   void inorderTitle(nodeType<videoType> *p);
      //This function prints the titles of all videos
      //in stock.
};
```

videoBinaryTree类中成员函数的定义与第16章中的定义类似。为了完整起见，这里给出它们的全部定义：

```
bool videoBinaryTree::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 videoBinaryTree::videoCheckIn(newString title)
{
        bool found = false;
        nodeType<videoType> *location;

        searchVideoList(title,found,location);

        if(found)
              location->info.checkIn();
        else
              cout<<"Video not in stock "<<endl;
}

void videoBinaryTree::videoCheckOut(newString title)
{
        bool found = false;
        nodeType<videoType> *location;

        searchVideoList(title,found,location);

        if(found)
           location->info.checkOut();
        else
           cout<<"Video not in stock "<<endl;
}

bool videoBinaryTree::videoCheckTitle(newString title)
{
        bool found = false;
        nodeType<videoType> *location;

        searchVideoList(title,found,location);

        return found;
}

void videoBinaryTree::videoUpdateInStock(newString title, int num)
{
        bool found = false;
        nodeType<videoType> *location;

        searchVideoList(title,found,location);

        if(found)
           location->info.updateInStock(num);
```

```
      else
         cout<<"Video not in stock "<<endl;
}

void videoBinaryTree::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 videoBinaryTree::videoSearch(newString title)
{
      bool found = false;
      nodeType<videoType> *location;

      searchVideoList(title,found,location);

      return found;
}
```

函数searchVideoList使用的查找算法与本章给出的二叉搜索树的查找算法类似。如果在列表中查找到元素就返回true，同时还返回包含所找元素的节点的指针。该函数的定义如下所示：

```
void videoBinaryTree::searchVideoList(newString title,
                                      bool& found,
                                      nodeType<videoType>* &current)
{
    found = false;

    videoType temp;

    temp.setVideoInfo(title,"","","","","",0);

    if(root == NULL)  //tree is empty
       cout<<"Cannot search an empty list. "<<endl;
    else
    {
       current = root;   //set current to point to the root node
                         //of the binary tree
       found = false;    //set found to false

       while(!found && current != NULL) //search the tree
            if(current->info == temp)    //item is found
               found = true;
            else
               if(current->info > temp)
                  current = current->llink;
               else
                  current = current->rlink;
```

```
    } //end else
}
```

通过给定一个指向包含影碟信息的二叉树根节点指针，函数inorderTitle使用中序遍历算法打印出所有影碟的名称。注意这个函数仅仅输出影碟名称。该函数定义如下所示：

```
void videoBinaryTree::inorderTitle(nodeType<videoType> *p)
{
      if(p != NULL)
      {
         inorderTitle(p->llink);
         p->info.printTitle();
         inorderTitle(p->rlink);
      }
}
```

函数 videoPrintTitle 使用函数 inorderTitle 打印影碟店中所有影碟的名称，该函数定义如下所示：

```
void videoBinaryTree::videoPrintTitle()
{
      inorderTitle(root);
}
```

主程序

主程序和以前一样，这里只给出了程序的清单。并假设包含类 videoBinaryTree 定义的头文件为 videoBinaryTree.h。

```
#include <iostream>
#include <fstream>
#include "myString.h"
#include "binarySearchTree.h"
#include "videoType.h"
#include "videoBinaryTree.h"

using namespace std;

void createVideoList(ifstream& infile, videoBinaryTree& videoList);
void displayMenu();

int main()
{
   videoBinaryTree videoList;
   int choice;
   char ch;
   char title[50];

   ifstream infile;
   infile.open("a:videoDat.txt");

   if(!infile)
   {
      cout<<"Input file does not exist"<<endl;
      return 1;
   }
   createVideoList(infile, videoList);
   infile.close();

   displayMenu();
```

```
cout<<"Enter choice: ";
cin>>choice;
cout<<endl;

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<<"This video is not in stock"<<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<<"This video is not in stock"<<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<<"This video is not in stock"<<endl;

            break;
      case 5: videoList.videoPrintTitle();
            break;
      case 6: videoList.inorderTraversal();
            break;
      default: cout<<"Bad Selection"<<endl;
      }//end switch

      displayMenu();
      cout<<"Enter choice: ";
      cin>>choice;
      cout<<endl;
   }//end while

   return 0;
}

void createVideoList(ifstream& infile, videoBinaryTree& 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.insert(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 see whether a particular video is in stock"
        <<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;
}
```

19.6　小结

1. 一个二叉树或者为空或者有一个叫做根节点的特殊节点。如果树为非空，则有两个节点集合：左子树和右子树；左子树和右子树均为二叉树。
2. 二叉树的节点有两个链接。
3. 如果二叉树中的一个节点既没有左孩子也没有右孩子，则该节点为叶节点。
4. 如果存在一条从节点 U 到节点 V 的分支，则称 U 为 V 的父节点。
5. 从一个二叉树节点 X 到节点 Y 的路径是指节点 X_0，X_1，…，X_n 序列，并有：
   (a) $X=X_0$，$X_n=Y$
   (b) 对于所有 $i=1, 2, \cdots, n$，X_{i-1} 是 X_i 的父节点。也就是说从 X_0 到 X_1，X_1 到 X_2，…，X_{i-1} 到 X_i，…，X_{n-1} 到 X_n 各有一个分支。
6. 在二叉树中，节点的层数是指从根节点到该节点所经过的分支个数。
7. 二叉树的根节点的层数为0，根节点的子节点的层数为1。
8. 二叉树的高度是指从根节点到叶节点的最长路径上的节点个数。
9. 对于中序遍历，二叉树采用如下方式遍历：
   a. 遍历左子树
   b. 访问节点
   c. 遍历右子树
10. 对于前序遍历，二叉树采用如下方式遍历：
   a. 访问节点
   b. 遍历左子树
   c. 遍历右子树
11. 对子后序遍历，二叉树采用如下方式遍历：
   a. 遍历左子树
   b. 遍历右子树
   c. 访问节点

12. 二叉搜索树 T 或者为空或者：
   (i) T 有一个叫做根节点的特殊节点。
   (ii) T 有两个节点集合：L_T 和 R_T，分别叫做 T 的左子树和右子树。
   (iii) 根节点的键值大于左子树中任一节点的键值，小于右子树中任一节点的键值。
   (iv) L_T 和 R_T 均是二叉搜索树。
13. 要从二叉搜索树中删除一个左右子树均为非空的节点。首先要找到它的直接前趋，然后将直接前趋节点的 info 拷贝给该节点，最后删除直接前趋节点。

19.7 练习

1. 判断下面说法的正误。
   a. 一个二叉树不能为空。
   b. 根节点的层数为 0。
   c. 如果一个树只有一个节点，则由于该树的层数为 0，所以高度也为 0。
   d. 二叉树的中序遍历算法总是按升序输出数据。
2. 画出所有的 14 种有 4 个节点的二叉树。

图 19.18 中的二叉树适用于练习 3 至练习 8。

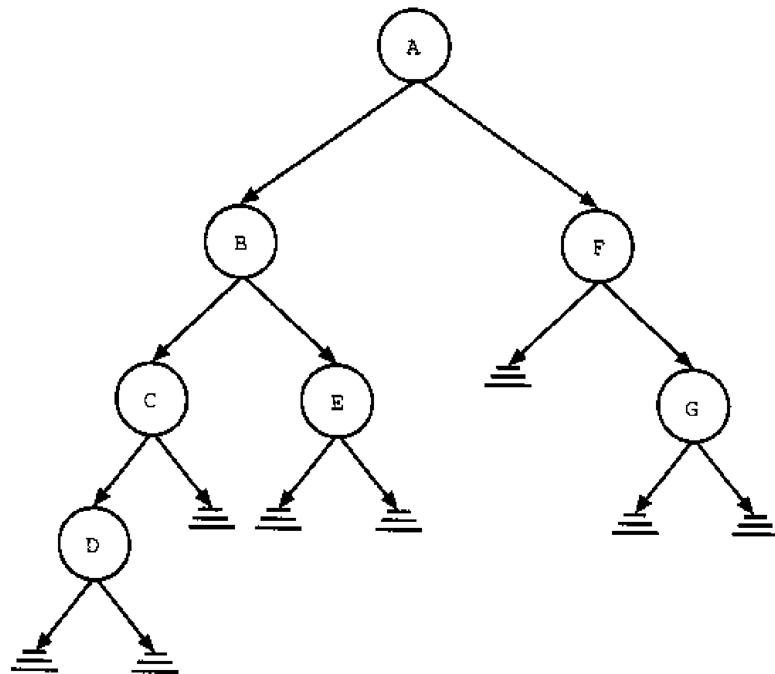


图 19.18 用于练习 3 至练习 8 的图

3. 找出 A 的左子树 L_A 中的所有节点。
4. 找出 A 的右子树 R_A 中的所有节点。
5. 找出 B 的右子树 R_B 中的所有节点。
6. 列出该二叉树的中序序列。
7. 列出该二叉树的前序序列。
8. 列出该二叉树的后序序列。

图 19.19 中的二叉树用于练习 9 至练习 13。

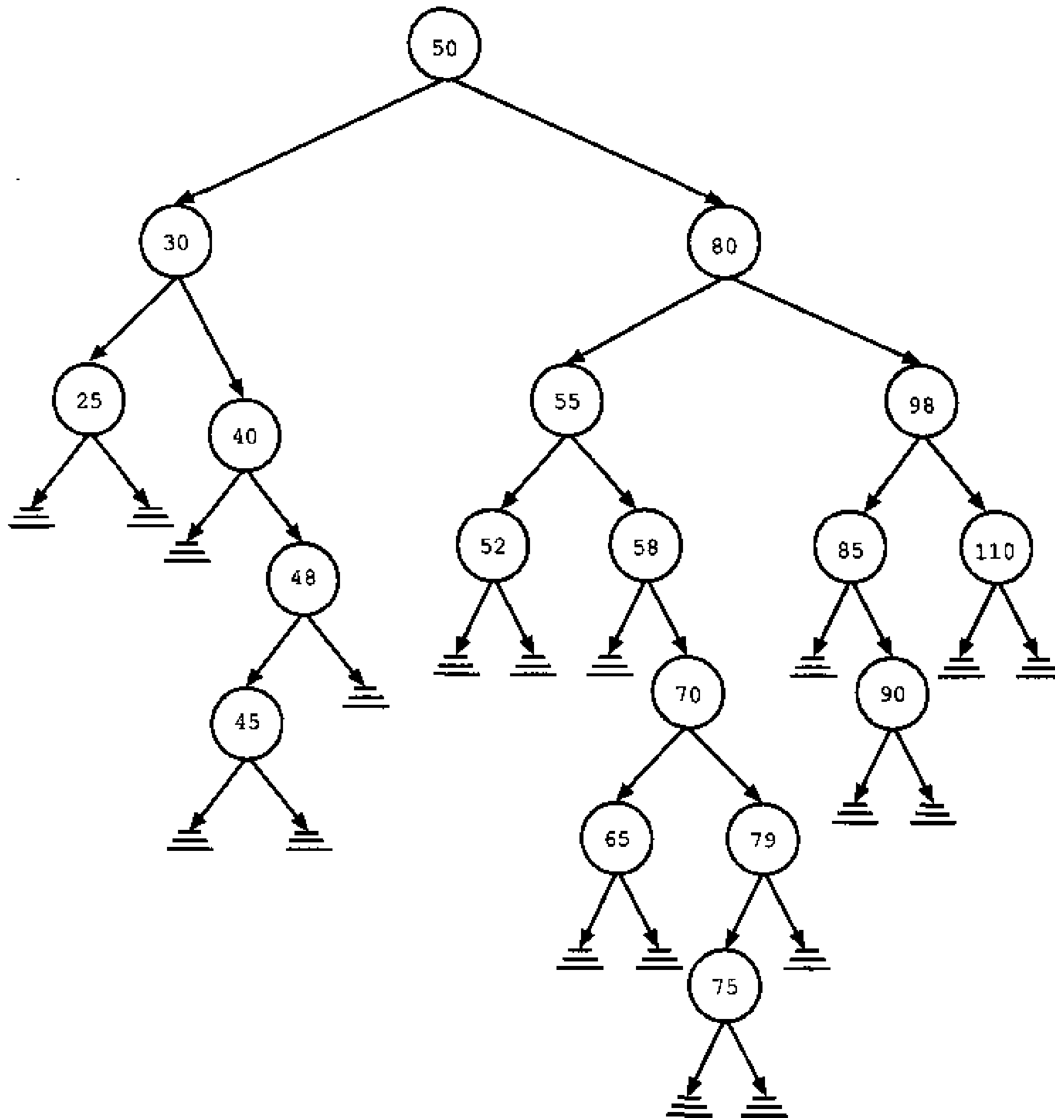


图 19.19 用于练习 9 至练习 13 的图

9. 列出从 info 为 80 的节点到 info 为 79 的节点的路径。
10. 要将一个info为35的节点插入到该二叉树。列出函数insert在插入该节点时所要访问的所有节点，并重画插入节点后的二叉树。
11. 删除节点 52 并重画该二叉树。
12. 删除节点 40 并重画该二叉树。
13. 删除节点 80 和 58 并重画该二叉树。
14. 假设有两个元素序列，分别对应着中序序列和前序序列，证明能够重新构建一个惟一的二叉树。
15. 下面列出了遍历二叉树所产生的两个序列：

    前序：ABCDEFGHIJKLM
    中序：CEDFBAHJIKGML

    画出该二叉树。
16. 给出前序序列和后序序列，说明可能无法重新构建二叉树。

19.8 编程练习

1. 写出函数 nodeCount 的定义，该函数返回二叉树中的节点数目。将该函数加入到 binaryTreeType 类中，并编写程序测试该函数。
2. 写出函数 leavesCount 的定义，该函数参数为指向二叉树根节点的指针并返回二叉树中的叶节点数目。将该函数加入到 binaryTreeType 类中，并编写程序测试该函数。
3. 写出函数 swapSubtrees 的定义，该函数交换二叉树中所有左子树和右子树。将该函数加入到 binaryTreeType 类中，并编写程序测试该函数。
4. 实现一个前序遍历算法，该算法以一个用户定义的函数作为参数，该函数指定访问节点时应进行的操作。
5. 实现一个后序遍历算法，该算法以一个用户定义的函数作为参数，该函数指定访问节点时应进行的操作。
6. （影碟店程序）在第 16 章中的编程练习 5 中，要求设计并实现一个类，该类通过链表维护客户数据。由于查找链表的算法是顺序的，所以查找非常费时。设计并实现一个使用二叉搜索树存储客户数据的 customerBTreeType 类。customerBTreeType 类必须从本章中的 bSearchTreeType 类中派生。
7. （影碟店程序）参考本章和编程练习 6，使用类来实现影碟数据、影碟列表数据、客户数据以及客户列表数据。设计并实现完整的影碟店程序。

第20章 图

本章要点：

- 了解图
- 了解图论的基本术语
- 理解如何在计算机内存中表示一个图
- 研究并实现各种图的遍历算法
- 学习如何实现最短路径算法
- 研究并实现最小生成树算法

前几章介绍了表示和操作数据的几种方式。本章将讨论如何实现和操作图，图在计算机科学中有着广泛的应用。

20.1 简介

1736年，人们提出了如下的问题：在Königsburg小镇（如今的Kaliningrad），Pregel（Pregolya）河流经Kneiphof岛时被分为两条分支，如图20.1所示。

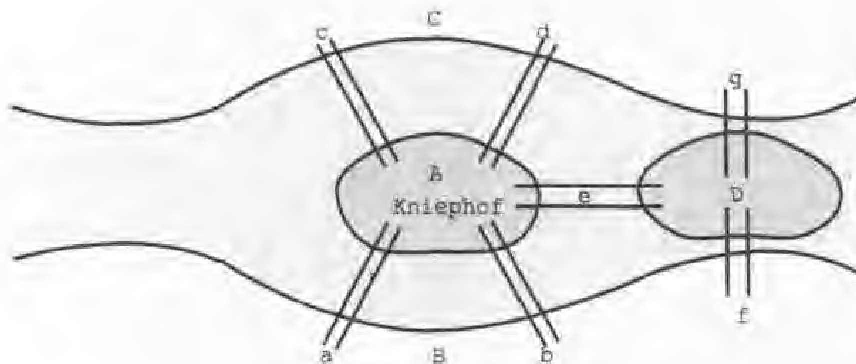


图20.1 Königsburg桥问题

如图所示，Pregel河有四块陆地（A，B，C，D）。这4块陆地通过7座桥连接起来，分别为a，b，c，d，e，f和g。Königsburg桥问题如下：从一个陆地出发，能否每座桥只走一次，最后返回到出发地。1736年，欧拉（Euler）用图（图20.2）表示出Königsburg桥问题，并说明该问题不可解。这标志着图论的诞生。

在过去的200多年，图论被广泛应用到不同的领域。图可用于表示电子电路、化合物、公路地图，等等。同时也可应用到电子电路分析、查找最短路径、项目规划、语言学、基因学、社会科学等。本章将介绍图及其在计算机科学中的应用。

20.1.1 图的定义和符号

为了方便讨论，我们从集合论中借鉴一些定义和术语。令X为一集合，如果a是X中的元素，则记为$a \in X$（符号“$\in$”表示“属于”）。如果集合Y中的每个元素同时都是X中的元素，则Y为X的子集，

记为 $Y \subseteq X$（符号“⊆”表示“是…的子集”）。集合 A 和 B 的交集（Intersection）中的元素既是 A 中的元素也是 B 中的元素，记为 $A \cap B$，即 $A \cap B = \{x \mid x \in A 且 x \in B\}$（符号“∩”表示“交集”）。集合 A 和 B 的并集（Union）中的元素或者是 A 中的元素或者是 B 中的元素，记为 $A \cup B$，即 $A \cup B = \{x \mid x \in A 或 x \in B\}$（符号“∪”表示“并集”）。对于集合 A 和 B，$A \times B$ 表示 A 和 B 中元素的所有有序偶集合，即 $A \times B = \{(a, b) \mid a \in A, b \in B\}$。

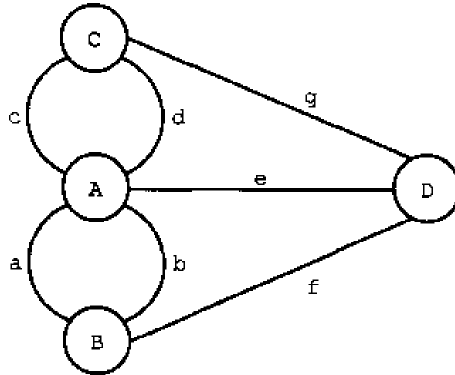


图 20.2　Königsburg 问题图示

一个图（Graph）G 由一对元素集合组成，记为：$G = (V, E)$，其中 V 为有穷的非空集合，称为 G 的顶点（Vertices）的集合。$E \subseteq V \times V$，所以 E 中的每个元素都是 V 中的元素对，E 被称为边（Edges）的集合。

$V(G)$ 表示图 G 的顶点的集合，$E(G)$ 表示图 G 的边的集合。

如果 $E(G)$ 的元素是有序对，则称 G 为有向图（Directed Graph）；否则，称 G 为无向图（Digraph）。在一个无向图中，(u, v) 和 (v, u) 表示同一条边。

若 G 是一个图，如果 $V(H) \subseteq V(G)$ 且 $E(H) \subseteq E(G)$，则称图 H 为图 G 的子图。图 H 中的每个顶点都是图 G 的顶点，图 H 中的每条边都是图 G 的边。

一个图可以用绘成图的形式表示：带有标号的圆圈表示顶点。在无向图中，边用直线表示；在有向图中，边用箭头表示。

例 20.1　图 20.3 给出了几个无向图的例子。

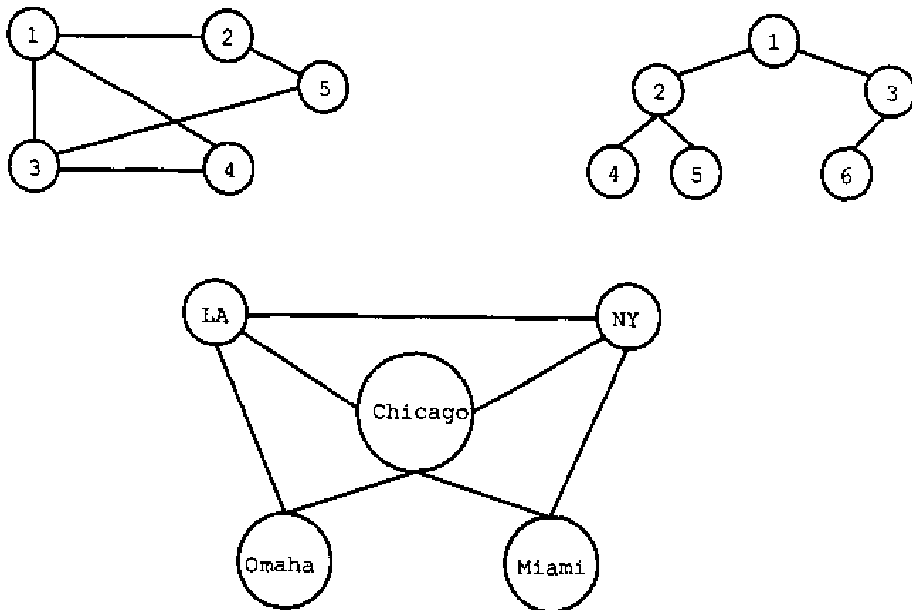


图 20.3　几种无向图

例 20.2 图 20.4 给出了几个有向图的例子。

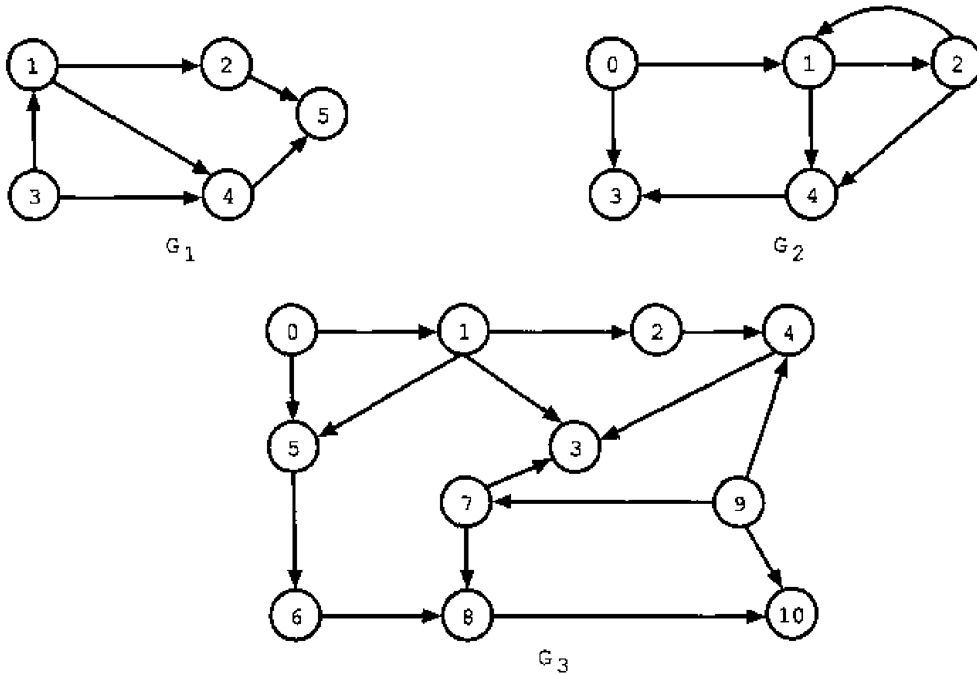


图 20.4 几种有向图

对于图 20.4 中的图，有：

$V(G_1)=\{1, 2, 3, 4, 5\}$ $E(G_1)=\{(1, 2),(1, 4),(2, 5),(3, 1),(3, 4),(4, 5)\}$

$V(G_2)=\{0, 1, 2, 3, 4\}$ $E(G_2)=\{(0, 1),(0, 3),(1, 2),(1, 4),(2, 1),(2, 4),(4, 3)\}$

$V(G_3)=\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ $E(G_3)=\{(0, 1),(0, 5),(1, 2),(1, 3),(1, 5),(2, 4),(4, 3),$
$(5, 6),(6, 8),(7, 3),(7, 8),(8, 10),(9, 4),(9, 7),(9, 10)\}$

若 G 为无向图，u 和 v 为 G 的两个顶点。如果两个顶点之间存在一条边，则称 u 和 v 为邻接点（Adjacent），即 $(u, v) \in E(G)$。只与一个单独顶点相关联的边称为环（Loop）。如果边 e_1 和 e_2 与同一个节点对 $\{u, v\}$ 相关联，则称 e_1 和 e_2 为平行边（Parallel Edges）。如果一个图既没有环也没有平行边，则称该图为简单图（Simple Graph）。若 $e=(u, v)$ 为图 G 的一条边，则称 e 与顶点 u 和 v 相关联（Incident）。如果在 u 和 v 之间存在一个顶点序列 $u_1, u_2, \cdots, u_n$，满足 $u=u_1, u_n=v$，且对于所有 $i=1, 2, \cdots, n-1$，(u_i, u_{i+1}) 为图的一条边，则称 u 和 v 之间存在一条路径（Path），u 和 v 是连通的（Connected）。如果一条路径除了第一个顶点和最后一个顶点外，所有的顶点都是惟一的，则称该路径为一条简单路径（Simple Path）。如果图 G 中的某条简单路径的第一个顶点与最后一个顶点相同，则称该路径为环路（Cycle）。如果图 G 中的任意两个顶点都是可连通的，则称图 G 是连通图。图 G 的最大可连通顶点子集称为图 G 的分量（Component）。

若图 G 是有向图，u 和 v 为 G 的两个顶点。如果从 u 到 v 之间存在一条边，即 $(u, v) \in E(G)$，则称 u 邻接到 v，v 邻接于 u。有向图 G 中路径和环路的定义与无向图类似。如果 G 中的任意两个顶点都是可连通的，则称 G 是强连通图（Strongly Connected）。

考虑图 20.4 中的有向图。在 G_1 中，1–4–5 为由顶点 1 到顶点 5 的一条路径。G_1 中不存在环路。G_2 中，1–2–1 是一条环路。G_3 中，0–1–2–4–3 是从顶点 1 到顶点 3 的一条路径。1–5–6–8–10 是从顶点 1 到顶点 10 的一条路径。G_3 中没有环路。

20.2　图的表示

为了能够通过程序处理和操作图，首先要能够存储图（即能够在计算机中表示）。图可以用多种方式表示，下面讨论两种最常用的方式：邻接矩阵（Adjacency Matrix）和邻接表。

20.2.1　邻接矩阵

设图 G 是有 n（$n>0$）个顶点的图。$V(G)=\{v_1, v_2, \cdots, v_n\}$。邻接矩阵 A_G 是一个二维 $n \times n$ 的矩阵，且如果 v_i 和 v_j 之间存在一条边，则 A_G 项(i, j)为 1，否则项(i, j)为 0。即：

$$A_G(i,j)=\begin{cases}1, & \text{如果 } (v_i,v_j)\in E(G)\\ 0, & \text{否则}\end{cases}$$

例 20.3　考虑图 20.4 中的有向图。有向图 G_1，G_2，G_3 对应的邻接矩阵分别如下所示：

$$A_{G1}=\begin{bmatrix}0&1&0&1&0\\0&0&0&0&1\\1&0&0&1&0\\0&0&0&0&1\\0&0&0&0&0\end{bmatrix}\qquad A_{G2}=\begin{bmatrix}0&1&0&1&0\\0&0&1&0&1\\0&1&0&0&1\\0&0&0&0&0\\0&0&0&1&0\end{bmatrix}$$

$$A_{G3}=\begin{matrix}0\\1\\2\\3\\4\\5\\6\\7\\8\\9\\10\end{matrix}\begin{bmatrix}0&1&0&0&0&1&0&0&0&0&0\\0&0&1&1&0&1&0&0&0&0&0\\0&0&0&0&1&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0\\0&0&0&1&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&0&0&0&1\\0&0&0&0&1&0&0&1&0&0&1\\0&0&0&0&0&0&0&0&0&0&0\end{bmatrix}$$

20.2.2　邻接表

设图 G 为有 n（$n>0$）个顶点的图，$V(G)=\{v_1, v_2, \cdots, v_n\}$。若用邻接表表示，对于每个顶点 v 都有一个链表相对应，该链表中的每个节点存放的是顶点 u，且$(v, u) \in E(G)$。由于有 n 个顶点，所以可采用大小为 n 的数组 A 存储顶点，$A[i]$为指向包含顶点 v_i 所有相邻顶点的链表的指针。因此，每个节点由两个部分组成：顶点、链接指针。顶点部分则保存的是与顶点 i 相邻接的顶点的索引。

例 20.4　对于图 20.4 中的有向图，图 20.5 给出了有向图 G_2 的邻接表：

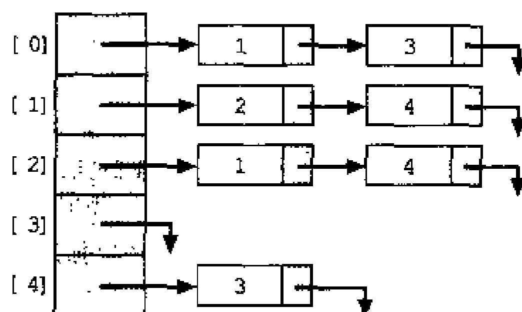


图 20.5　图 20.4 中有向图 G_2 的邻接表

图 20.6 给出了有向图 G_3 的邻接表。

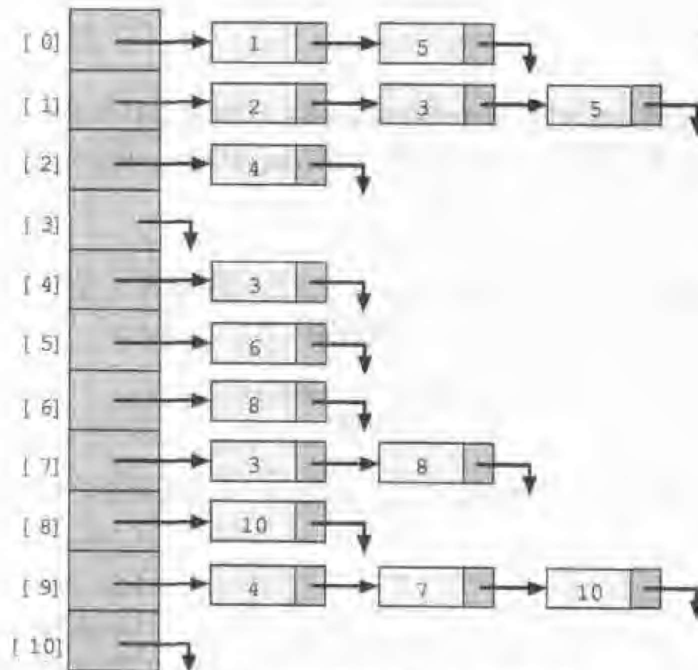


图 20.6　图 20.4 中有向图 G_3 的邻接表

图节点的定义

由于每个节点有两个部分组成，所以节点定义如下所示：

```
template <class vType>
struct nodeType
{
      vType vertex;
      nodeType<vType>  *link;
};
```

20.3　图操作

现在我们已经知道了怎样在计算机中表示图，下面将了解有关图的基本操作。图的常用操作有：

1. 创建图，即采用某种图的表示方法在计算机中存储图
2. 清空图，该操作将图置空
3. 打印图
4. 判定图是否为空
5. 遍历图

本章后面在讨论到具体的应用程序或图时，将会添加更多的操作。

采用何种方式在计算机中表示图，主要取决于具体的应用。为了说明问题，下面将采用邻接表来表示图。所以，对于每个顶点 v，与 v 相邻接的所有顶点[在有向图中也称为直接后继(Immediate Successor)]都存储在与 v 相关联的链表中。

为了处理链表中的数据，我们采用第 16 章中讨论的 linkedListType 类。图的遍历算法和在本节后面讨论的算法都需要对每个节点所对应的链表进行遍历，获取每个节点中存储的顶点。linkedListType 类中并没有包含能够对链表进行遍历的函数，也没有能够逐个获得节点中数据的函数(函数 print 只是简单地将数据输出到输出设备)。所以，首先要扩展 linkedListType 类的定义（使用继承），以便能够获得数组

中某个顶点的所有邻接顶点。这便简化了对于顶点的处理过程。在这里称该类为linkedListGraph，其定义如下所示：

```
template<class vType>
class linkedListGraph: public linkedListType<vType>
{
public:
      void getAdjacentVertices(vType adjacencyList[],
                               int& length);
      //The vertices adjacent to a given vertex from the
      //linked list are retrieved in the array adjacencyList.
      //The parameter length specifies the number of vertices
      //adjacent to a given vertex.
};
```

成员函数getAdjacentVertices的定义如下所示：

```
template<class vType>
void linkedListGraph<vType>::getAdjacentVertices(
                        vType adjacencyList[], int& length)
{
      nodeType<vType> *current;

      length = 0;
      current = first;

      while(current != NULL)
      {
            adjacencyList[length++] = current->info;
            current = current->link;
      }
}
```

接下来，我们要扩展类模板定义以包含常量表达式参数。

模板（再叙）

到目前为止，传给模板的都是数据类型。与类型（数据类型）相似，常量表达式也可以作为参数传递给模板。例如下面的类模板：

```
template<class elemType, int size>
class listType
{
public:
  .
  .
  .
private:
    int maxSize;
    int length;
    elemType listElem[size];
};
```

该类模板包含一个数组成员变量。数组的大小和类型都作为参数传递给该类模板。要创建一个有100个int型元素的表，可以采用如下的语句：

```
listType<int, 100> intList;
```

这样元素类型和数组大小都传递给了类模板listType。注意元素类型（int）和数组大小（100）都在类模板的名称后面的尖括号中。

由于上面的类定义中不再包含指针成员变量，所以无须包含析构函数。然而，我们添加默认构造函数来设置成员变量maxSize和length的值。

本章中，我们将采用模板的方法来声明数组数据成员变量，这特别适合于二维数组或非常大的数组。

下面介绍用以实现作为抽象数据类型（ADT）的图类，并提供关于图中操作的函数定义。

下面的类实现了作为抽象数据类型的图：

```
const int infinity = 10000000; //This will be used in later sections
                               //of this chapter, when we discuss
                               //weighted graphs

template<class vType, int size>
class graphType
{
public:
    bool isEmpty()
        //Returns true if the graph is empty; otherwise,
        //returns false
    void createGraph();
        //The graph is created using the adjacency list
        //representation
    void clearGraph();
        //The memory occupied by each vertex is deallocated
    void printGraph() const;
        //The graph is printed

    graphType();
        //default constructor
        //Post: The graph size is set to 0, that is, gSize = 0;
        //        gSize = 0; and maxSize = size

    ~graphType();
        //destructor
        //The storage occupied by the graph is deallocated

protected:
    int maxSize;  //maximum number of vertices
    int gSize;   //current number of vertices

    linkedListGraph<vType> *graph; //array of pointers
                    //to create the adjacency lists (linked lists)
};
```

注意：在本章的下面章节中，在使用C++描述图的算法时，总是假设图的n个顶点为数字0, 1,…, n-1。所以顶点数据类型为整型。为了使用户能够使用其他的方式指定顶点的类型，我们将继续使用模板，对算法的必要修改留给读者。

接下来讨论graphType类的成员函数定义。

如果一个图的顶点个数为零，即gSize等于零，则图为空。所以函数isEmpty定义如下所示：

```
template<class vType, int size>
bool graphType<vType, size>::isEmpty()
{
      return(gSize == 0);
}
```

函数createGraph的定义取决于数据输入的方式。为了说明问题，假设程序中的数据从文件中输入。程序提示用户指定输入文件，文件中的数据格式如下所示：

```
5
0 2 4 … -999
1 3 6 8 … -999
…
```

第一行指定了图中的顶点个数。其余行中的第一项指定了顶点，其余项（除了最后一项）指定了该顶点的邻接顶点。每一行以 -999 结束。

按照上述约定，函数 createGraph 定义如下所示：

```
template<class vType, int size>
void graphType<vType,size>::createGraph()
{
      ifstream infile;
      char fileName[50];

      vType vertex;
      vType adjacentVertex;

      if(gSize != 0)  //if the graph is not empty, make it empty
         clearGraph();

      cout<<"Enter the input file name: ";
      cin>>fileName;
      cout<<endl;

      infile.open(fileName);
      if(!infile)
      {
            cout<<"Cannot open the input file"<<endl;
            return;
      }

      infile>>gSize;  //get the number of vertices

      for(int index = 0; index < gSize; index++)
      {
            infile>>vertex;
            infile>>adjacentVertex;

            while(adjacentVertex != -999)
            {
                  graph[vertex].insertLast(adjacentVertex);
                  infile>>adjacentVertex;
            } //end while
      } //end for

      infile.close();
} //end createGraph
```

函数clearGraph通过回收每个链表占用的内存空间来清空图，同时将顶点数目置为0。

```
template<class vType, int size>
void graphType<vType, size>::clearGraph()
{
      int index;
```

```
	for(index = 0; index < gSize; index++)
		graph[index].destroyList();

	gSize = 0;
}
```

函数printGraph非常简单，其定义如下所示：

```
template<class vType, int size>
void graphType<vType, size>::printGraph() const
{
	int index;

	for(index = 0; index < gSize; index++)
	{
		cout<<index<<" ";
		graph[index].printList();
		cout<<endl;
	}
	cout<<endl;
} //end printGraph
```

默认构造函数和析构函数定义如下所示：

```
	//default constructor
template<class vType, int size>
graphType<vType, size>::graphType()
{
	maxSize = size;
	gSize = 0;

	graph = new linkedListGraph<vType>[maxSize];
}

	//destructor
template<class vType, int size>
graphType<vType, size>::~graphType()
{
	clearGraph();

	delete[] graph;
}
```

20.4　图遍历

图的操作往往需要对图进行遍历，本节将讨论图的遍历算法。

除了更复杂一些外，图的遍历与二叉树的遍历相似。二叉树没有环路，从根节点开始可以遍历整个二叉树。而图则不然，既可能存在环路又可能无法从某一个顶点遍历整个树(例如非连通图)。所以，必须跟踪每一个访问过的顶点，并从图中选择每一个没有访问过的节点进行遍历。这样，就可以保证遍历整个图。

图的最常用的两个遍历算法是：深度优先算法和广度优先算法。下面将详细讨论。为了方便，假设访问某个节点时只是输出该节点的索引。布尔型数组visited用于记录顶点是否被访问过。

20.4.1　深度优先算法

深度优先算法（Depth First Traversal）与二叉树的前序遍历算法类似，算法如下所示：

```
for each vertex, v, in the graph
  if v is not visited
     start the depth first traversal at v
```

为了便于参考，我们将图 20.4 中的图 G_3 重新给出（如图 20.7 所示）。

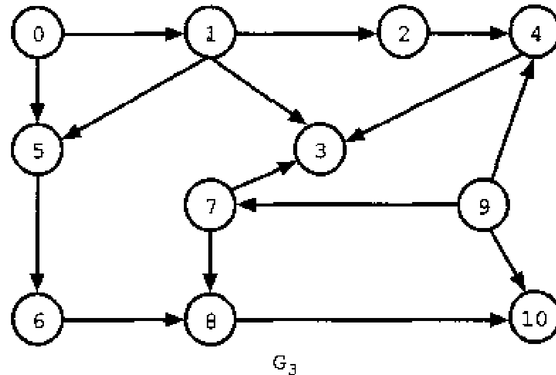


图 20.7　有向图 G_3

图 20.7 中有向图 G_3 顶点的深度优先遍历序列为：

```
0 1 2 4 3 5 6 8 10 7 9
```

对节点 v 进行深度优先遍历算法如下所示：

```
a. mark node v as visited
b. visit the node
c. for each vertex u adjacent to v
     if u is not visited
       start the depth first traversal at u
```

很明显，这是个递归算法。我们用递归函数 dft 来实现该算法。需要进行深度优先遍历的顶点和布尔型数组 visited 作为参数传递给该函数。

```
template<class vType, int size>
void graphType<vType,size>::dft(vType v, bool visited[])
{
      vType w;

      vType *adjacencyList;       //array to retrieve
                                  //the adjacent vertices
      adjacencyList = new vType[gSize];

      int alLength = 0;  //the number of adjacent vertices

      visited[v] = true;
      cout<<" "<<v<<" ";  //visit the vertex
      graph[v].getAdjacentVertices(adjacencyList, alLength);
              //retrieve the adjacent vertices into adjacencyList

      for(int index = 0; index < alLength; index++) //for each
```

```
    {                                       //vertex adjacent to v
          w = adjacencyList[ index] ;
          if(!visited[ w] )
             dft(w,visited);
    } //end for

    delete [] adjacencyList;
} //end dft
```

下面给出实现图中深度优先遍历算法的函数 depthFirstTraversal:

```
template <class vType, int size>
void graphType<vType, size>::depthFirstTraversal()
{
    bool *visited;     //array to keep track of the visited vertices
    visited = new bool[ gSize] ;

    int index;

    for(index = 0; index < gSize; index++)
        visited[ index] = false;

    for(index = 0; index < gSize; index++) //for each vertex
        if(!visited[ index] )          //that is not visited,
           dft(index,visited);         //do a depth first
                                       //traverssal

    delete [] visited;
} //end depthFirstTraversal
```

函数 depthFirstTraversal 实现了对整个图的遍历。函数 dftAtVertex 则对给定的顶点进行深度优先遍历，其定义如下所示：

```
template <class vType, int size>
void graphType<vType, size>::dftAtVertex(vertexType vertex)
{
      bool *visited;

      visited = new bool[ gSize] ;

      for(int index = 0; index < gSize; index++)
          visited[ index] = false;

      dft(vertex,visited);

      delete [] visited;

} //end dftAtVertex
```

20.4.2 广度优先遍历

图的广度优先遍历（Breadth First Traversal）算法与二叉树的按层遍历算法类似（从左向右访问每层节点）。只有在访问完第 i 层中的所有节点之后，才能访问第 i+1 层中的节点。

有向图 G_3 中顶点的广度优先遍历序列为：

```
0 1 5 2 3 6 4 8 10 7 9
```

与深度优先遍历一样，由于有可能无法从一个顶点遍历整个图，所以广度优先遍历也要从每一个没有访问过的顶点开始遍历。即，从第一个顶点开始，尽可能地遍历图；然后从下一个没有访问过的顶点开始广度优先遍历。我们使用队列来实现广度优先遍历，算法如下所示：

```
for each vertex v in the graph
    if v is not visited
       a.add v to the queue  //start the breadth first search at v
       b.mark v as visited
       c.while the queue is not empty
         c.1.remove vertex u from the queue
         c.2.retrieve the vertices adjacent to u
         c.3.for each vertex w that is adjacent to u
                 if w is not visited
                 c.3.1.Add w to the queue
                 c.3.2.mark w as visited
```

函数 breadthFirstTraversal 实现了该算法，其定义如下所示：

```
template <class vType, int size>
void graphType<vType, size>::breadthFirstTraversal()
{
   linkedQueueType<vType> queue;
   vType u;

   bool *visited;
   visited = new bool[gSize];

   for(int ind = 0; ind < gSize; ind++)
       visited[ind] = false;   //initialize the array
                               //visited to false

   vType *adjacencyList;
   adjacencyList = new vType[gSize];

   int alLength = 0;
   for(int index = 0; index < gSize; index++)
       if(!visited[index])
       {
         queue.addQueue(index);
         visited[index] = true;
         cout<<" "<<index<<" ";

         while(!queue.isEmptyQueue())
         {
            queue.deQueue(u);
            graph[u].getAdjacentVertices(adjacencyList, alLength);
            for(int w = 0; w < alLength; w++)
               if(!visited[adjacencyList[w]])
               {
                  queue.addQueue(adjacencyList[w]);
                  visited[adjacencyList[w]] = true;
                  cout<<" "<<adjacencyList[w]<<" ";
               } end if
         } //end while
      } //end if

      delete[] visited;
```

```
        delete[] adjacencyList;
} //end breadthFirstTraversal
```

在包含了上面图的遍历算法后，类 graphType 的定义如下所示：

```
template <class vertexType, int size>
class graphType
{
public:
    bool isEmpty();
        //Returns true if the graph is empty; otherwise,
        //returns false
    void createGraph();
        //The graph is created using the adjacency list
        //representation
    void clearGraph();
        //The memory occupied by each vertex is deallocated
    void printGraph() const;
        //The graph is printed

    void depthFirstTraversal();
        //This function performs the depth first traversal of
        //the entire graph
    void dftAtVertex(vertexType vertex);
        //This function performs the depth first traversal of
        //the graph at a node specified by the parameter vertex

    void breadthFirstTraversal();
        //This function performs the breadth first traversal of
        //the entire graph

    graphType();
        //default constructor
        //The graph size is set to 0, that is, gSize = 0;
        //maxSize = size

    ~graphType();
        //destructor
        //The storage occupied by the graph is deallocated

protected:
    int maxSize;     //maximum number of vertices
    int gSize;       //current number of vertices
    linkedListGraph<vertexType> *graph; //array of pointers
                     //to create the adjacency lists (linked lists)

private:
    void dft(vertexType v, bool visited[]);
      //This function performs the depth first traversal of
      //the graph at a particular node
};
```

在以后讨论图的算法时，将使用C++函数来实现特定算法，并采用继承的方法从graphType类中派生出新类。

20.5 最短路径算法

图论有很多应用。我们可以用图来表示化学物质之间的区别或显示空中航线，还可以表示城市、州、

国家的公路结构。两个顶点之间的边可以带有一个非负的实数，称为该边的权（Weight of the Edge）。如果一个图表示的是公路结构，权就表示两个地方之间的距离，或从一个地方到另一个地方的行程时间。这样的图就成为加权图（Weighted Graph）。

设G为加权图。u和v是图G中的两个顶点，P是G中由u到v的路径。路径P的权为路径P上所有边的权的总和，又称为通过P由u到v的权。

设G为代表公路结构的加权图，边的权表示的是行程时间。例如，某个商人在做月商务旅行计划时，他需要找到从他所在的城市到图中其他所有城市的最短路径（即，有着最小权的路径）。在现实中还有许多类似于这样的问题，即要找到从图中的某个顶点（源顶点）到其他每个顶点之间的最短路径。

本节将讨论由Dijkstra提出的最短路径算法（Shortest Path Algorithm），又称贪婪算法（Greedy Algorithm）。

假设G是有n（$n \geq 0$）个顶点的图，$V(G) = \{v_1, v_2, \cdots, v_n\}$。$W$是$n \times n$的二维矩阵，满足：

$$W(i,j)=\begin{cases} w_{ij}, & \text{如果}(v_i, v_j)\text{是}G\text{中的边，并且}w_{ij}\text{是边}(v_i, v_j)\text{的权} \\ \infty, & \text{如果从}v_i\text{到}v_j\text{没有边} \end{cases}$$

程序的输入是图与图相关联的加权矩阵。为了简化数据的输入，我们扩展了graphType类（通过继承），添加了用来创建图和与之关联的加权矩阵的函数createWeightedGraph，该派生类称为weightGraphType。同时还增加了用以实现最短路径算法的函数。weightedGraphType的类定义如下所示：

```
template <class vType, int size>
class weightedGraphType: public graphType<vType, size>
{
public:
    void createWeightedGraph();
        //This function creates the graph and the weight matrix
    void shortestPath(vType vertex);
        //This function determines the smallest weight from the
        //vertex, that is, the source to every other vertex
        //in the graph
    void printShortestDistance(vType vertex);
        //This function prints the shortest weight from the
        //source to the other vertices in the graph
protected:
    int weights[size][size];   //weight matrix
    int smallestWeight[size];  //smallest weight from
                               //source to vertices
};
```

函数createWeightedGraph的定义作为练习留给读者。下面将介绍最短路径算法。

20.5.1 最短路径

给定一个顶点vertex（源顶点），最短路径算法如下所示：

1. 初始化数组smallestWeight，使得smallestWeight[u] = weights[vertex,u]。
2. 令smallestWeight[vertex] = 0。
3. 查找距离vertex最近的没有确定最短路径的顶点v。
4. 标记v为相应已找到最短路径的顶点。
5. 对于G中的每一个从顶点vertex到其最短路径还未找到，并且存在边(v, w)的顶点w，如果从vertex通过v到w的路径权值小于它当前的权值，则更新w的权值为v的权值+边(v, w)的权值。

由于有 n 个顶点，所以要重复步骤 3 至步骤 5 共计 $n-1$ 次。

例 20.5 说明了最短路径算法。

例 20.5 假设图 G 如图 20.8 所示。

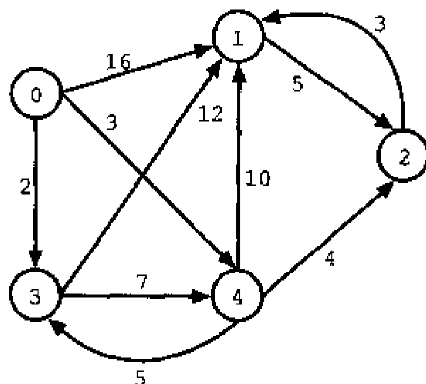
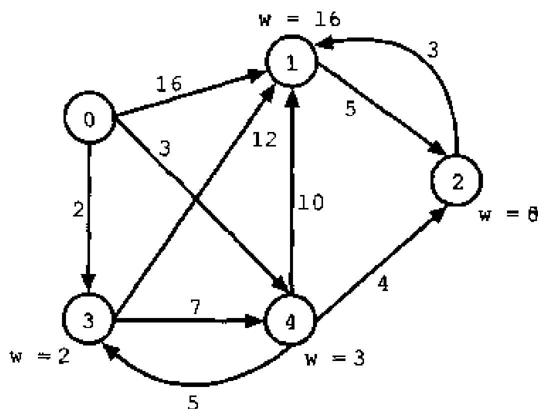


图 20.8 加权图 G

假设 G 的源顶点为 0，图给出了每条边的权值。在步骤 1 和步骤 2 执行之后，结果如图 20.9 所示（在图 20.9 中，符号 Θ 表示 ∞）。



	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	16	θ	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	F	F	F	F

图 20.9 执行完步骤 1 和步骤 2 后的图

首先选取顶点 3，并标记 weightFound[3]为 true。很明显，对于从 0 到 1，路径 0–3–1 的权小于路径 0–1。所以我们更新 smallestWeight[1]为 14。结果如图 20.10 所示（虚线表示从源顶点到该顶点的最短路径）。

现在选取顶点 4 并重复以前的步骤，设置 weightFound[4]为 true。很明显，路径 0–4–1 的权 13 小于当前顶点 1 的权值为 14，所以更新 smallestWeight[1]。类似地，更新 smallestWeight[2]。结果如图 20.11 所示。

接下来要选取的顶点为 2，设置 weightFound[2]为 true。很明显，路径 0–4–2–1 的权 10 小于当前 1 的权值 13，所以更新 smallestWeight[1]。结果如图 20.12 所示。

最后选取顶点 1，并标记路径。结果如图 20.13 所示。

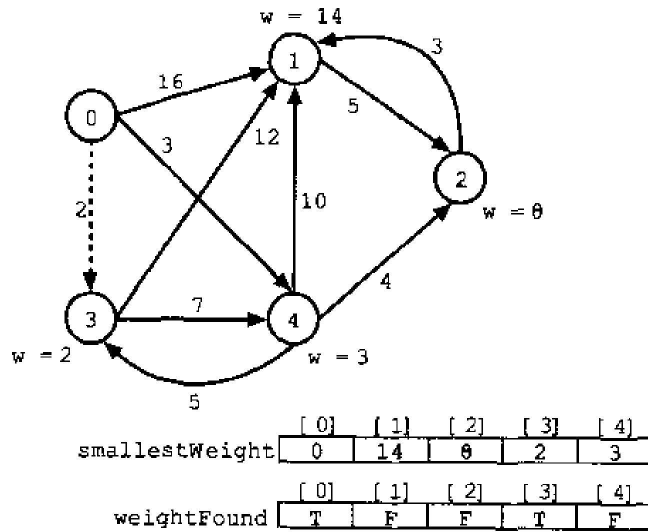


图20.10　经过步骤3，步骤4，步骤5，执行完第1次迭代后的图

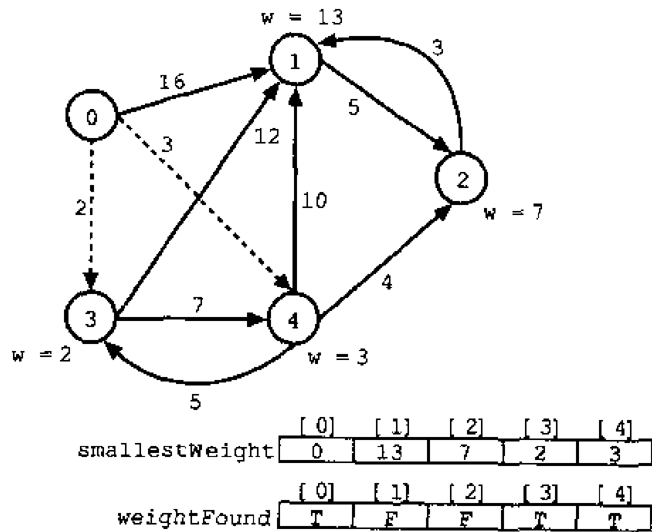


图20.11　经过步骤3，步骤4，步骤5，执行完第2次迭代后的图

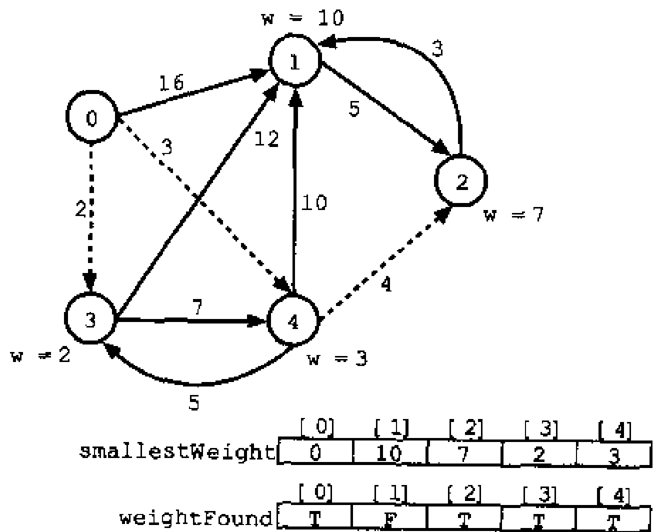
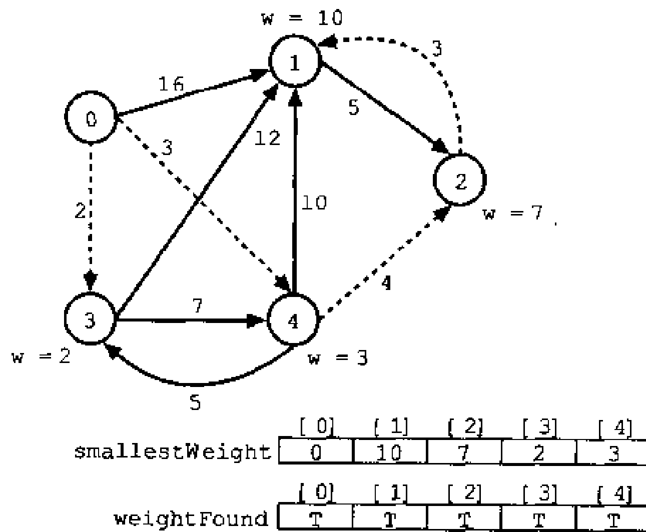


图20.12　经过步骤3，步骤4，步骤5，执行完第3次迭代后的图



	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	10	7	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	T	T	T	T

图 20.13 经过步骤3，步骤4，步骤5，执行完第4次迭代后的图

函数 shortestPath 实现了上面的算法，该函数定义如下所示：

```
template <class vType, int size>
void weightedGraphType<vType, size>::shortestPath(vType vertex)
{
   int i,j;
   int v;
   int minWeight;

   for(j = 0; j < gSize; j++)
       smallestWeight[j] = weights[vertex][j];

   bool weightFound[size];
   for(j = 0; j < gSize; j++)
       weightFound[j] = false;

   weightFound[vertex] = true;
   smallestWeight[vertex] = 0;

   for(i = 0; i < gSize - 1; i++)
   {
       minWeight = infinity;

       for(j = 0; j < gSize; j++)
           if(!weightFound[j])
               if(smallestWeight[j] < minWeight)
               {
                   v = j;
                   minWeight = smallestWeight[v];
               }

       weightFound[v] = true;

       for(j = 0; j < gSize; j++)
           if(!weightFound[j])
               if(minWeight + weights[v][j] < smallestWeight[j])
                   smallestWeight[j] = minWeight + weights[v][j];
   } //end for
} //end shortestPath
```

注意函数 shortestPath 只是记录了从源顶点到某个顶点最短路径的权值。可修改该函数，使其记录下从源顶点到某个顶点的最短路径。函数的修改留给读者作为练习。

函数 printShortestDistance 的定义非常简单：

```
template <class vType, int size>
void weightedGraphType<vType,size>::printShortestDistance (vType vertex)
{
        cout<<"Source vertex: "<<vertex<<endl;
        cout<<"Shortest distance from source to each vertex"<<endl;
        cout<<"Vertex Shortest_Distance"<<endl;

        for(int j = 0; j < gSize; j++)
            cout<<setw(4)<<j<<setw(12)<<smallestWeight[ j] <<endl;
        cout<<endl;
}
```

20.6　最小生成树

树中任意两个顶点有惟一路径存在，则该树为简单图。一棵指派了一个顶点作为根节点的树称为有根树（Rooted Tree）。如果树 T 中的每个边都赋予了权值，则称 T 为加权树（Weighted Tree）。如果 T 是一棵加权树，T 的权值（记为 $W(T)$）则为 T 中所有权值的总和。

如果树 T 是图 G 的一个子图，且 $V(T) = V(G)$，也就是说 G 的所有顶点也都是 T 的顶点，则称树 T 为图 G 的生成树（Spanning Tree）。

图 20.14 中的图 G 是一个有 7 个顶点的加权图。

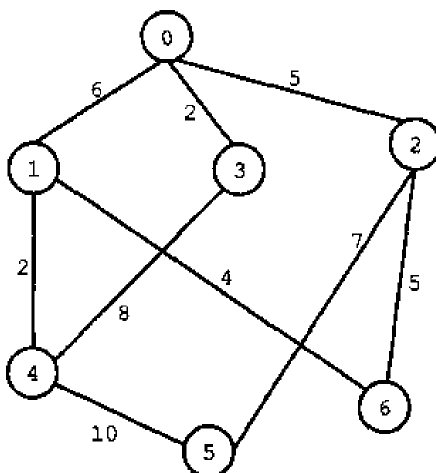


图 20.14　加权图 G

图 20.15 给出了 G 的生成树。

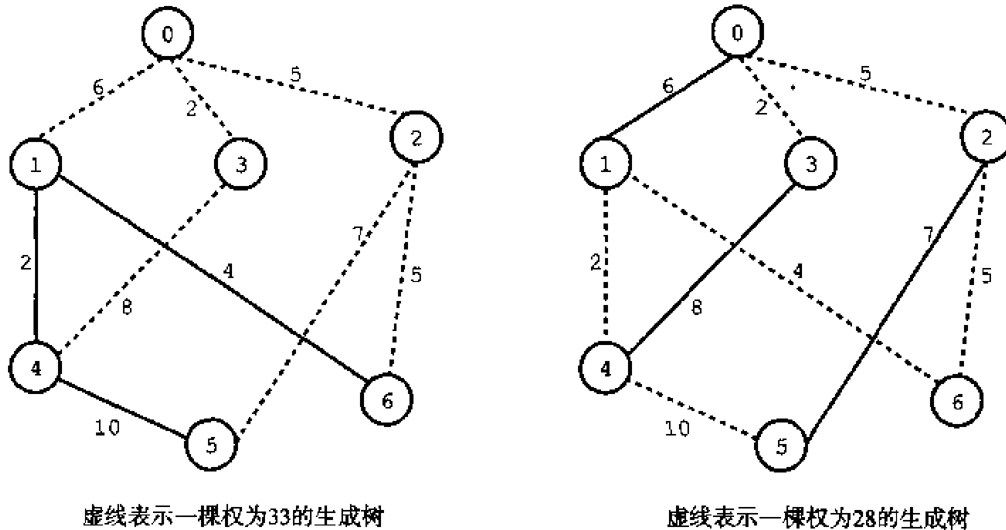
定理　一个图 G 当且仅当 G 为连通图时有生成树。

设 G 为加权图，G 的一棵最小生成树（Minimal Spanning Tree）是指 G 的权值最小的生成树。

查找图的最小生成树有两个著名的算法：Prim 算法和 Kruskal 算法。本节将讨论 Prim 算法。

Prim 算法通过迭代添加边直到获得最小生成树的方法来构造树。首先指定一个顶点为源顶点，每次迭代都将一个没有形成回路的新边加入到树中。

设 G 为加权树，$V(G)=\{v_0, v_1,\cdots, v_{n-1}\}$，顶点的数目 n 为非负整数。令 v_0 为源顶点。将构建的树记为 T，起初 $V(T)$ 只有源顶点，$E(T)$ 为空。在接下来的迭代中，对于不在 $V(T)$ 中的顶点，如果 T 中的顶点到该顶点之间存在边并且权值最小，则将该顶点加入到 $V(T)$ 中，同时将该边加入到 $E(T)$ 中。



虚线表示一棵权为33的生成树　　虚线表示一棵权为28的生成树

图 20.15　图 20.14 中图的生成树

Prim 算法如下所示（n 为 G 中的顶点个数）：

```
1. Set V(T) = {source}
2. Set E(T) = empty
3. for i = 1 to n
   3.1 minWeight = infinity;
   3.2 for j = 1 to n
      3.2.1 if v_j is in V(T)
         3.2.1.1 for k = 1 to n
            3.2.1.1.1 if v_k is not in T and weight[v_j, v_k] < minWeight
                      {
                         endVertex = v_k;
                         edge = (v_j, v_k);
                         minWeight = weight[v_j, v_k];
                      }
   3.3 V(T) = V(T) ∪ {endVertex};
   3.4 E(T) = E(T) ∪ {edge};
```

下面通过图 20.16 所示的图来说明 Prim 算法。

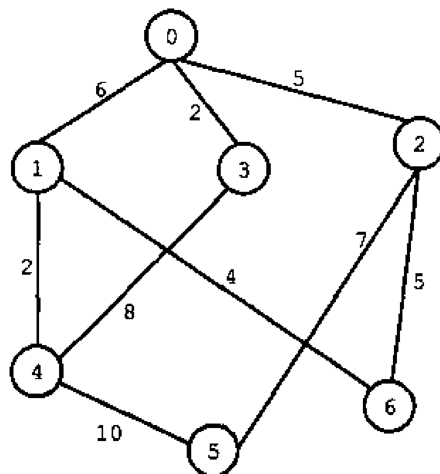


图 20.16　加权图 G

用N来表示G中不在T中的顶点集合。假设源顶点为0，执行完步骤1和步骤2后，$V(T)$，$E(T)$和N如图20.17所示。

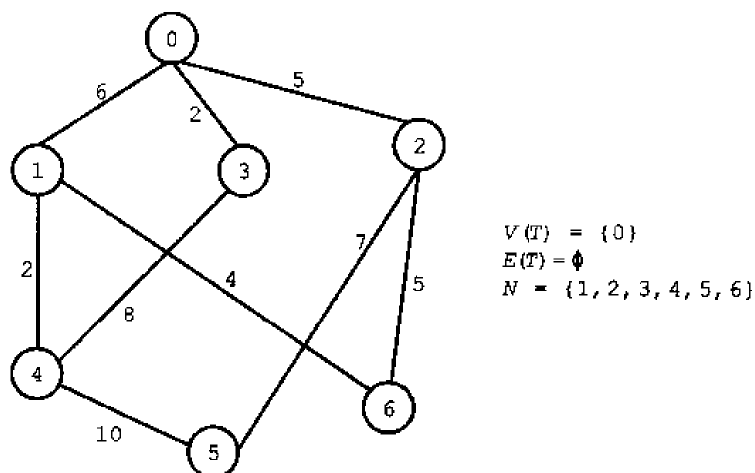


图20.17　经过步骤1和步骤2后的图G，$V(T)$，$E(T)$和N

步骤3.2首先检查以下的边：

边	边的权值
(0,1)	6
(0,2)	5
(0,3)	2

很明显，边(0, 3)的权值最小。所以将顶点3加入到$V(T)$中，同时将边(0, 3)加入到$E(T)$中。$V(T)$，$E(T)$和N结果如图20.18所示（属于T的边用虚线显示）。

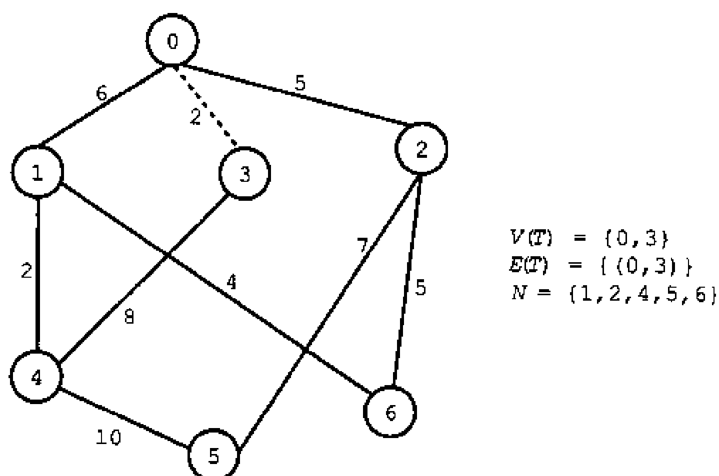


图20.18　经过步骤3.2的第1次迭代后的图G，$V(T)$，$E(T)$，N

接下来，步骤3.2检查以下边：

边	边的权值
(0,1)	6
(0,2)	5
(3,4)	8

很明显，边(0, 2)的权值最小。所以将顶点2加入到$V(T)$中，同时将边(0, 2)加入到$E(T)$中。$V(T)$，$E(T)$和N结果如图20.19所示。

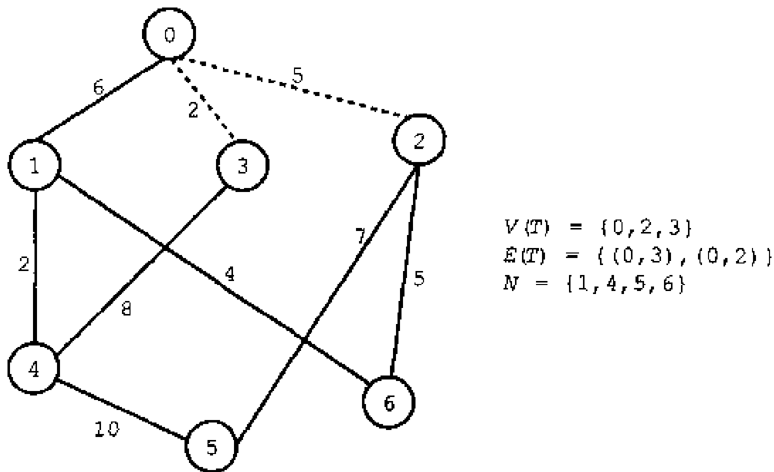


图20.19 经过步骤3.2的第2次迭代后的图G，$V(T)$，$E(T)$，N

接下来的迭代中，步骤3.2检查以下边：

边	边的权值
(0,1)	6
(2,5)	7
(2,6)	5
(3,4)	8

很明显，边(2, 6)的权值最小。所以将顶点6加入到$V(T)$中，同时将边(2, 6)加入到$E(T)$中。$V(T)$，$E(T)$和N结果如图20.20所示。

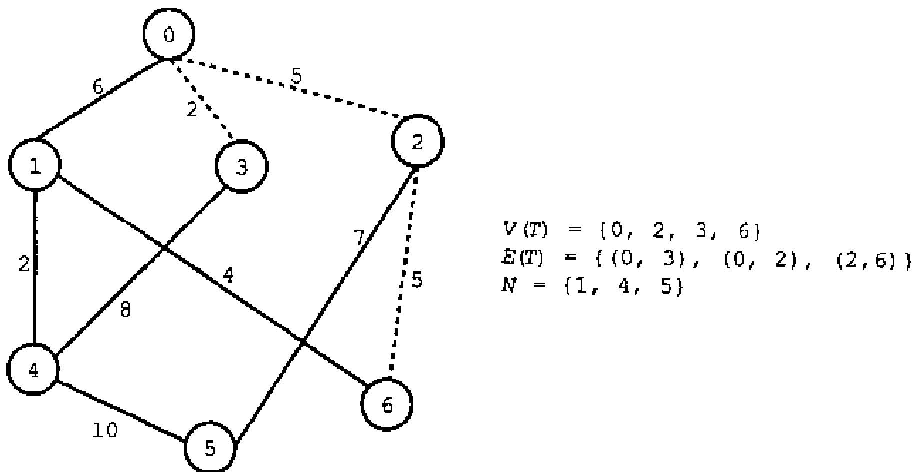


图20.20 经过步骤3.2的第3次迭代后的图G，$V(T)$，$E(T)$，N

接下来的迭代中，步骤3.2检查以下边：

边	边的权值
(0,1)	6
(2,5)	7
(3,4)	8
(6,1)	4

很明显，边(6, 1)的权值最小。所以将顶点 1 加入到 $V(T)$中，同时将边(6, 1)加入到 $E(T)$中。$V(T)$，$E(T)$和 N 结果如图 20.21 所示。

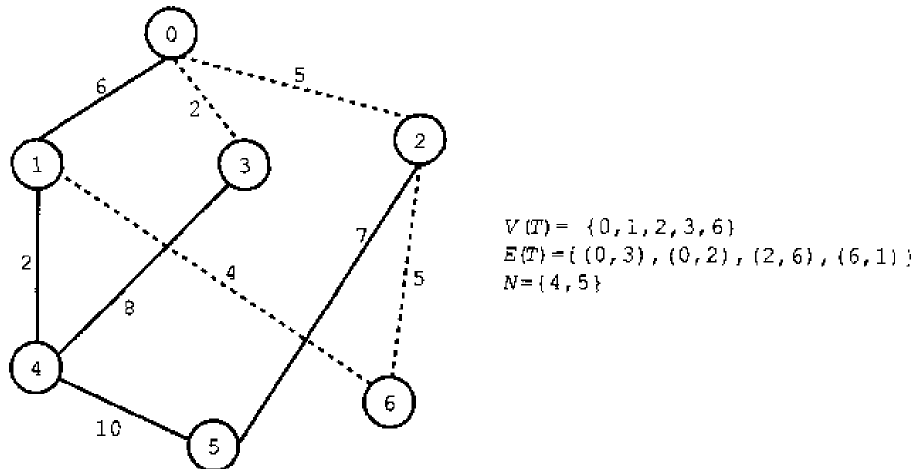


图 20.21　经过步骤 3.2 的第 4 次迭代后的图 G，$V(T)$，$E(T)$，N

接下来的迭代中，步骤 3.2 检查以下边：

边	边的权值
(1,4)	2
(2,5)	7
(3,4)	8

很明显，边(1, 4)的权值最小。所以将顶点 4 加入到 $V(T)$中，同时将边(1, 4)加入到 $E(T)$中。$V(T)$，$E(T)$和 N 结果如图 20.22 所示。

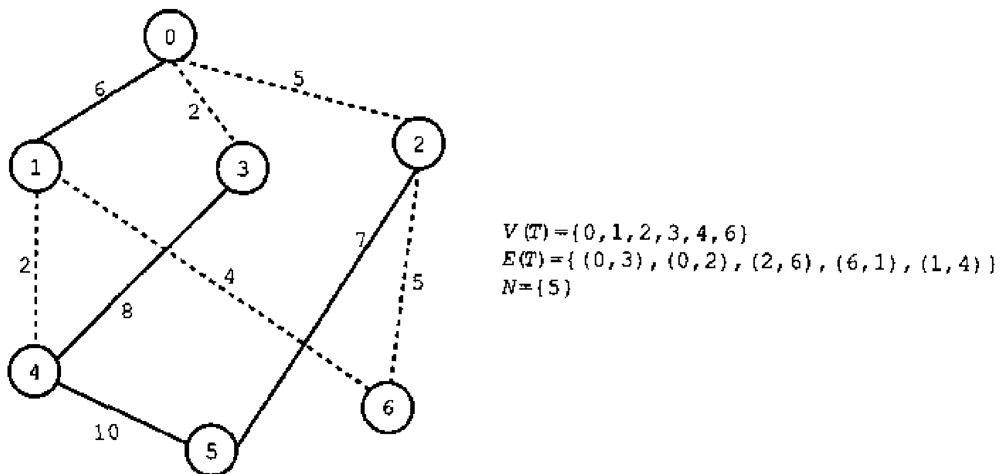


图 20.22　经过步骤 3.2 的第 5 次迭代后的图 G，$V(T)$，$E(T)$，N

接下来的迭代中，步骤 3.2 检查了以下边：

边	边的权值
(2,5)	7
(4,5)	10

很明显，边(2, 5)的权值最小。所以将顶点 5 加入到 $V(T)$中，同时将边(2, 5)加入到 $E(T)$中。$V(T)$，$E(T)$和 N 结果如图 20.23 所示。

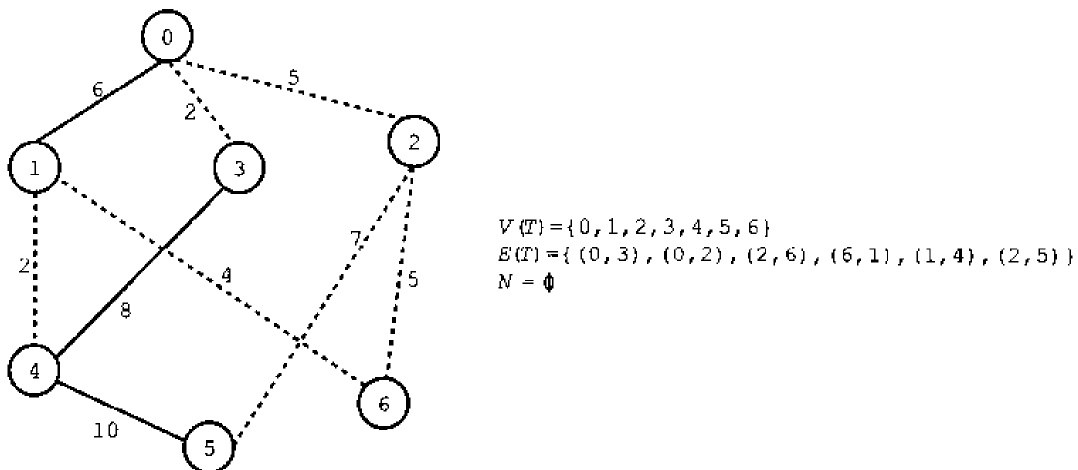


图 20.23 经过步骤 3.2 的第 6 次迭代后的图 *G*，*V*(*T*)，*E*(*T*)，*N*

虚线便是图 *G* 的最小生成树，权值为 25。

在给出 Prim 算法的定义之前，先将生成树定义为一个抽象数据类型（ADT）。

假设 msvt 是一个布尔型数组：如果顶点 v_j 在 *T* 中，则 msvt[*j*]为 true；否则，msvt[*j*]为 false。如果顶点 v_j 和 v_k 间有一条边相连，则数组 edges 的 edges[*j*] = k。假设边(v_i,v_j)在最小生成树内，则数组 edgeWeights 的 edgeWeights[*j*]为边(v_i, v_j)的权值。

根据这些约定，生成树类的定义如下所示：

```
template <class vType, int size>
class msTreeType: public graphType<vType, size>
{
public:
    void createSpanningGraph();
      //This function creates the graph and the weight matrix.
    void minimalSpanning(vType sVertex);
      //This function creates the edges of the minimal
      //spanning tree. The weight of the edges is also
      //saved in the array edgeWeights.
    void printTreeAndWeight();
      //This function outputs the edges of the minimal
      //spanning tree and the weight of the minimal
      //spanning tree.

protected:
    vType source;
    int weights[size][size];
    int edges[size];
    int edgeWeights[size];
};
```

函数 createSpanningGraph 的定义留给读者作为练习。该函数创建图及其加权矩阵。

函数 minimalSpanning 实现了 Prim 算法，定义如下所示：

```
template <class vType, int size>
void msTreeType<vType, size>::minimalSpanning(vType sVertex)
{
   int i,j,k;
   vType startVertex, endVertex;
   int minWeight;
```

```
    source = sVertex;

    bool mstv[size];

    for(j = 0; j < gSize; j++)
    {
       mstv[j] = false;
       edges[j] = source;
       edgeWeights[j] = weights[source][j];
    }

    mstv[source] = true;
    edgeWeights[source] = 0;

    for(i = 0; i < gSize - 1; i++)
    {
       minWeight = infinity;

       for(j = 0; j < gSize; j++)
          if(mstv[j])
             for(k = 0; k < gSize; k++)
                if(!mstv[k] && weights[j][k] < minWeight)
                {
                   endVertex = k;
                   startVertex = j;
                   minWeight = weights[j][k];
                }

       mstv[endVertex] = true;
       edges[endVertex] = startVertex;
       edgeWeights[endVertex] = minWeight;
    } //end for
} //end minimalSpanning
```

函数minimalSpanning使用了三层嵌套for循环。所以在最坏的情况下，本节给出的Prim算法的复杂度为$O(n^3)$。也可以设计出复杂度为$O(n^2)$的Prim算法，本章编程练习5便要求设计这样一个算法。

函数printTreeAndWeight非常简单，其定义如下所示：

```
template <class vType, int size>
void msTreeType<vType, size>::printTreeAndWeight()
{
   int treeWeight = 0;

   cout<<"Source Vertex: "<<source<<endl;
   cout<<"Edges    Weight"<<endl;

   for(int j = 0; j < gSize; j++)
   {
      treeWeight = treeWeight + edgeWeights[j];
      cout<<"("<<edges[j]<<", "<<j<<") "
          <<edgeWeights[j]<<endl;
   }

   cout<<endl;
   cout<<"Tree Weight: "<<treeWeight<<endl;
} //end printTreeAndWeight
```

20.7 小结

1. 图由两个集合组成：$G=(V, E)$。V为有穷的非空集合，称为G的顶点的集合；$E \subseteq V \times V$，称为边的集合。
2. 在无向图$G=(V, E)$中，集合E中的元素为无序对。
3. 在有向图$G=(V, E)$中，集合E中的元素为有序对。
4. 对于图G，如果图H中的所有顶点都是G中的顶点，且H的所有边也都是G的边，则称H为G的子图。
5. 在无向图中，如果两个顶点u和v之间存在一条边，则称这两个顶点为邻接顶点。
6. 只与一个顶点相关联的边称之为环。
7. 在无向图中，如果两个边e_1和e_2与同一对顶点$\{u, v\}$相关联，则称e_1和e_2为平行边。
8. 如果一个图既没有环也没有平行边，则该图称为简单图。
9. 若$e=(u, v)$为无向图G的一条边，则e与顶点u和v相关联。
10. 如果在u和v之间存在一个顶点序列u_1，u_2，…，u_n，满足$u=u_1$，$u_n=v$且对于所有$i=1$，2，…，$n-1$，(u_i, u_{i+1})为图的一条边，则称u和v之间存在一条路径。
11. 如果从顶点u到v之间存在着一条路径，则称u和v是可连通的。
12. 如果一条路径除了第一个顶点和最后一个顶点外，所有的顶点都是惟一的，则该路径为一条简单路径。
13. 如果G中的一个简单路径的第一个顶点与最后一个顶点相同，则称该路径为环路。
14. 如果无向图G中的任意两个顶点都是可连通的，则称G是连通图。
15. 无向图G的极大连通子图称为G的连通分量。
16. 若u和v为有向图G的两个顶点。如果从u到v之间存在一条边，即$(u, v) \in E(G)$，则称u邻接到v，v邻接于u。
17. 如果有向图G中的任意两个顶点都是可连通的，则称G是强连通图。
18. 设G为有n（$n>0$）个顶点的图。$V(G)=\{v_1, v_2, \cdots, v_n\}$。邻接矩阵$\boldsymbol{A}_G$是一个二维$n \times n$的矩阵，如果$v_i$和$v_j$之间存在一条边，则$\boldsymbol{A}_G$中$(i, j)$为1；否则，$(i, j)$为0。
19. 若用邻接表表示图，图中的每个顶点v都对应一个链表。对链表中每个节点存放的顶点u，u满足$(v, u) \in E(G)$。
20. 图的深度优先遍历与二叉树的前序遍历类似。
21. 图的广度优先遍历与二叉树的按层遍历类似。
22. 最短路径算法可以求出从一个顶点到图中其他顶点之间的最短路径。
23. 在加权图中，每条边都有一个非负的权值。
24. 路径P的权值为路径上所有边的权值总和，也称为通过P由v到u的权值。
25. 任意两个顶点都只有惟一路径的树称为简单图。
26. 一个指派了一个顶点作为根节点的树被称为有根树。
27. 如果树T中的每个边都赋予了权值，则称T为加权树。
28. 如果T是一个加权树，T的权值（记为$W(T)$）是T中所有边权值的总和。
29. 如果树T是图G的一个子图，且$V(T)=V(G)$，即G的所有顶点也都是T的顶点，则称T为G的生成树。

20.8 练习

图20.24中的图适用于练习1至练习4。

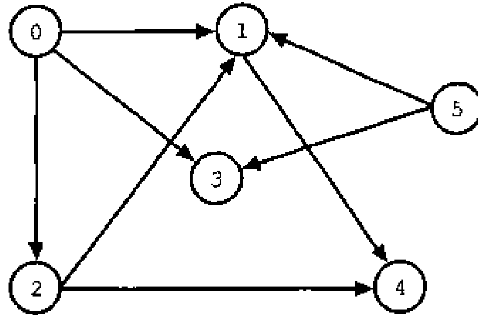


图 20.24　用于练习 1 至练习 4 的图

1. 给出该图的邻接矩阵。
2. 画出该图的邻接表。
3. 列出图的深度优先遍历的节点序列。
4. 列出图的广度优先遍历的节点序列。
5. 给出图 20.25 中图的加权矩阵。
6. 根据图 20.26，找出从节点 0 到其他每个节点的最短距离。

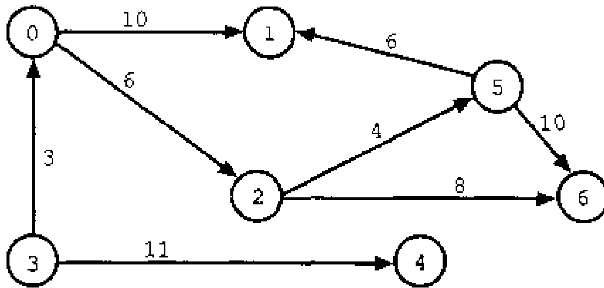


图 20.25　用于练习 5 的图

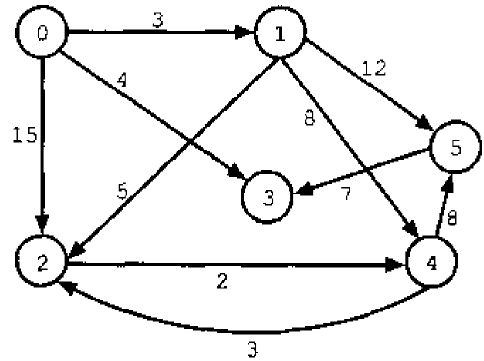


图 20.26　用于练习 6 的图

7. 找出图 20.27 的生成树。

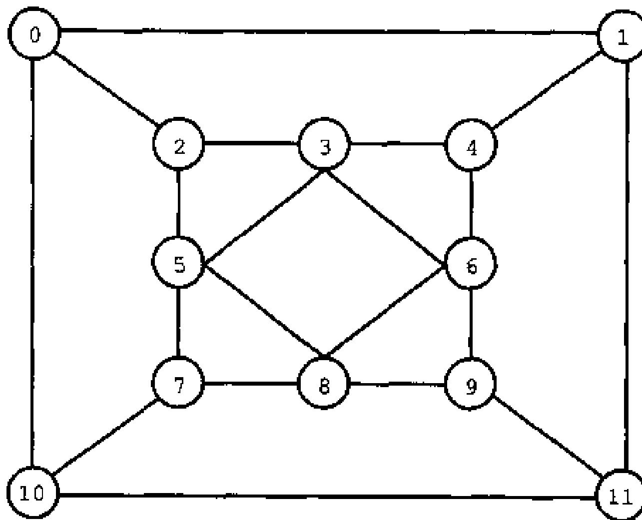


图 20.27　用于练习 7 的图

8. 找出图 20.28 的生成树。

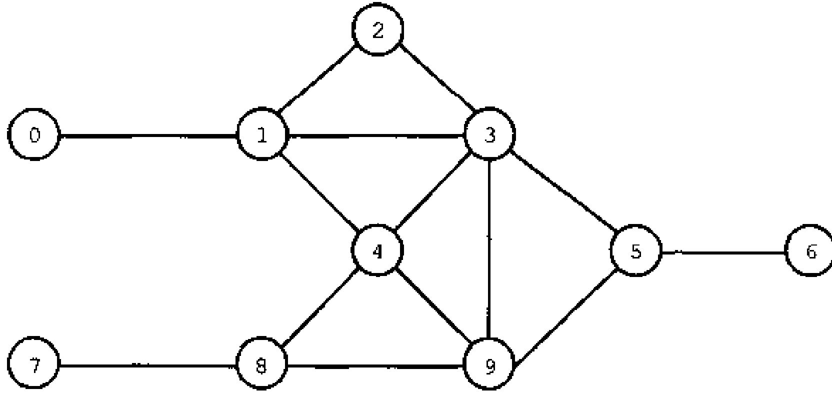


图 20.28　用于练习 8 的图

9. 使用本章给出的算法找出图 20.29 的最小生成树。

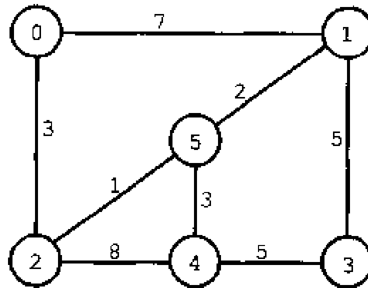


图 20.29　用于练习 9 的图

20.9 编程练习

1. 写出通过深度优先遍历算法输出图中所有节点的程序。
2. 写出通过广度优先遍历算法输出图中所有节点的程序。
3. 写出一个能够输出从给定节点到其他所有节点最短距离的程序。
4. 写出一个能够输出给定图的最小生成树的程序。
5. 本章中给出的最小生成树算法的复杂度为 $O(n^3)$，而下面算法的复杂度为 $O(n^2)$：

输入　一个有 n 个顶点的连通加权图 G，$G=(V, E)$。顶点为数字 0, 1,⋯, $n-1$，起始顶点为 s，加权矩阵为 W。

输出　一棵最小生成树：

```
Prim2(G, W, n, s)

a. Let T = (V,E),where E = φ.
b. for(j = 0; j < n; j++)
  {
        edgeWeight[j] = W(s, j);
        edges[j] = s;
        visited[s] = false;
  }
c. edgeWeight[s] = 0;
d. visited[s] = true;
```

```
e. while(not all nodes visited)
  {
   i. Choose the node that is not visited and has the smallest weight, and call it k.
   ii. visited[k] = true;
   iii. E = E ∪{ (k, edges[k])}
   iv. V = V ∪ {k}
   v. for each node j that is not visited
         if(W(k, j) < edgeWeight[k])
         {
            edgeWeight[k] = W(k, j);
            edges[j] = k;
         }
  }
f. Return T.
```

写出实现该算法的函数Prim2,并将该函数加入到msTreeType类中。最后编写一个程序测试该函数。

6. 对于图 G，$V(G) = \{v_1, v_2, \cdots, v_n\}$，其中 $n \geq 0$，$V(G)$的拓扑序列（Topological Ordering）是一个满足以下条件的线性顶点序列 v_{i1}，v_{i2}，…，v_{in}：如果 v_{ij} 是 v_{ik} 的前趋，$j \neq k$，$1 \leq j$，$k \leq n$，则 v_{ij} 先于 v_{ik}，即在该线性序列中$j<k$。假设 G 没有环路。下面的广度优先拓扑算法列出了图的拓扑序列。

   在广度优先拓扑序列中，首先要找到一个没有前趋的顶点，并将该顶点置于拓扑序列的首位。接下来，找出一个所有前趋顶点都已经在拓扑序列中的顶点 v，并将 v 放入拓扑序列中。使用数组 predCount 来记录图中每个顶点的前趋个数。开始时，将 predCount[j]初始化为顶点 v_j 的前趋顶点个数。然后，初始化广度优先遍历队列，将所有 predCount[k]为 0 的顶点放入该队列中。该算法如下所示：

   a. 创建一个数组 predCount，并将其初始化，使得 predCount[i]为顶点 v_i 的前趋顶点个数。
   b. 初始化队列 queue，将所有 predCount[k]为 0 的顶点 v_k 加入到 queue 中（很明显，queue 一定为非空，因为图没有环路）。
   c. while 队列 queue 非空：
      c.1. 删除队列的队首元素 u
      c.2. 将 u 放到数组 topologicalOrder[topIndex]中下一个可用位置上，并将 topIndex 加 1。
      c.3. 对于 u 的直接后继 w：
         c.3.1. 将 w 的前趋顶点个数减 1
         c.3.2. 如果 w 的前趋顶点个数为 0，则将 w 加入到 queue 中

   编写一个实现该广度优先拓扑序列算法的程序。

第21章　标准模板库

本章要点：

- 了解标准模板库（STL，Standard Template Library）
- 了解标准模板库的三个基本组成部分：容器、迭代器和算法
- 了解在程序中各种容器是如何操作数据的
- 了解迭代器的用途
- 学习各种标准算法

第15章详细介绍和说明了模板。有了模板类，我们就可以开发（并使用）类属（通用）代码来处理表。例如，使用类 listType 来处理整型表和字符串表。在第16章和第17章中，学习了三种最重要的数据结构：链表、栈和队列。在这几章中，我们借助于类模板开发类属代码来处理链表。此外，使用面向对象程序设计（OOP）的原则，开发了类属代码来处理有序表。此外，在第17章中，我们使用类模板开发类属代码实现栈和队列。因此，不难看出模板是一种提高代码复用的有效工具。

ANSI/ISO 标准 C++ 配备了一个标准模板库（STL）。此外，STL 提供了类模板来处理表（顺序的或是链接的）、栈和队列。本章将讨论 STL 的许多重要特征，并说明如何在程序中使用 STL 提供的工具。

21.1　STL 的组成部分

程序的一个重要目标就是操作数据并生成结果。要达到这个目标，就必须要具备以下能力：将数据存储到计算机内存中，访问特定数据，以及编写操作数据的算法。

例如，假设所有数据元素的类型都相同，并且可以大致估计出元素的个数，那么就可以使用一个数组来存储这些数据。我们可以使用下标来访问数组中的某个元素；还可以使用循环和数组下标遍历数组中的所有元素。还可以使用初始化数组、排序，或是在数组中查找某个元素等算法，来操作存储在数组中的数据。另一方面，如果不希望考虑数据的规模，那么可以使用链表来处理这些数据。如果要以后进先出（LIFO）的方式来处理数据，那么就可以使用栈。类似地，如果要以先进先出（FIFO）的方式来处理数据，那么就可以使用队列。

STL 使用以下功能来有效地操作数据。STL 有三个主要组成部分：

- 容器
- 迭代器
- 算法

容器和迭代器都是类模板。迭代器用来遍历容器中的所有元素；算法用来操作数据。以下几节将详细讨论这三个组成部分。

21.1.1　容器类型

容器用来管理某种类型的对象。STL 容器可以分为三类：

- 顺序容器
- 关联容器
- 容器适配器

21.1.2　顺序容器

在顺序容器中，每个对象都有一个确定的位置。三种预定义的顺序容器是：

- 向量（Vector）
- 双端队列（Deque）
- 表（List）

在讨论容器类型之前，先简要介绍向量顺序容器。向量容器很像数组，因此向量容器上的操作也与数组上的操作相似。同时，利用向量容器，我们可以描述一些所有容器上的共同属性。实际上，所有容器中通用操作的名字都相同。当然，对于某个容器来说，有其特有的操作。这些特有操作将在介绍具体某个容器时加以讨论。

21.1.3　顺序容器：向量

向量容器使用动态数组存储、管理对象。因为数组是一个随机访问数据结构，所以可以随机访问向量中的元素。在数组中间或是开始处插入一个元素是费时的，特别是在数组非常大的时候更是如此。然而，在数组末端插入元素却很快。

实现向量容器的类名是 vector（容器是类模板）。包含 vector 类的头文件名是 vector。所以，如果要在程序里使用向量容器，就要在程序中包含下面语句：

```
#include <vector>
```

此外，在定义向量类型对象时，必须指定该对象的类型，因为 vector 类是一个类模板。例如，语句：

```
vector<int> intList;
```

将 intList 声明为一个元素类型为 int 的向量容器对象。类似地，语句：

```
vector<string> stringList;
```

将 stringList 声明为一个元素类型为 string 的向量容器对象。

声明向量对象

vector 类包含了多个构造函数，其中包括默认构造函数。因此，可以通过多种方式来声明和初始化向量容器。表 21.1 描述了怎样声明和初始化指定类型的向量容器。

表 21.1　各种声明和初始向量容器的方法

语句	作用
`vector<elementType> vecList;`	创建一个没有任何元素的空向量 vecList（使用默认构造函数）
`vector<elementType> vecList(otherVecList)`	创建一个向量 vecList，并使用向量 otherVecList 中的元素初始化该向量。向量 vecList 与向量 otherVecList 的类型相同
`vector<elementType> vecList(size);`	创建一个大小为 size 的向量 vecList，并使用默认构造函数初始化该向量
`vector<elementType> vecList(n, elem);`	创建一个大小为 *n* 的向量 vecList，该向量中所有的 *n* 个元素都初始化为 elem
`vector<elementType> vecList(begin, end);`	创建一个向量 vecList，并初始化该向量（begin，end）中的元素。即，从 begin 到 end-1 之间的所有元素

在介绍了如何声明向量顺序容器之后，让我们开始讨论如何操作向量容器中的数据。首先，必须要知道下面几种基本操作：

- 元素插入
- 元素删除
- 遍历向量容器中的元素

假设vecList是一个向量类型容器。表21.2给出了在vecList中插入元素和删除元素的操作，这些操作是vector类的成员函数。表21.2还说明了如何使用这些操作。

表21.2 向量容器上的各种操作

语句	作用
`vecList.clear()`	从容器中删除所有元素
`vecList.erase(position)`	删除由position指定的位置上的元素
`vecList.erase(beg,end)`	删除从beg到end-1之间的所有元素
`vecList.insert(position, elem)`	将elem的一个拷贝插入到由position指定的位置上，并返回新元素的位置
`vecList.insert(position, n, elem)`	将elem的*n*个拷贝插入到由position指定的位置上
`vecList.insert(position, beg, end)`	将从beg到end-1之间的所有元素的拷贝插入到vecList中由position指定的位置上
`vecList.push_back(elem)`	将elem的一个拷贝插入到vecList的末尾
`vecList.pop_back()`	删除最后元素
`vecList.resize(num)`	将元素个数改为num。如果size( )增加，默认的构造函数负责创建这些新元素
`vecList.resize(num, elem)`	将元素个数改为num。如果size( )增加，默认的构造函数将这些新元素初始化为elem

注意：在表21.2中，参数position在STL术语中被称为迭代器。迭代器的作用类似于指针。通常，迭代器用来遍历容器中的元素。换句话说，通过迭代器我们可以遍历容器中的所有元素，并可以逐一地处理这些元素。下面，将介绍怎样在向量容器中声明迭代器，以及怎样操作容器中的数据。由于迭代器是STL的一个独立部分，所以我们将在“迭代器”一节中详细讨论它。

在向量容器中声明迭代器

vector类包含了一个typedef iterator，这是一个public成员。通过iterator，可以声明向量容器中的迭代器。例如，语句：

```
vector<int>::iterator intVecIter;
```

将intVecIter声明为int类型的向量容器迭代器。

因为iterator是一个定义在vector类中的typedef，所以必须使用容器名（vector）、容器元素类型和作用域运算符来使用iterator。

表达式：

```
++intVecIter
```

将迭代器intVecIter加1，使其指向容器中的下一个元素。表达式：

```
*intVecIter
```

返回当前迭代器位置上的元素。

注意，迭代器上的这些操作和指针（参见第14章）上的相应操作是相同的。运算符*作为单目运算符使用时，称为递引用运算符。

下面将讨论如何使用迭代器来操作向量容器中的数据。假设有下面语句：

```
vector<int> intList;                  //Line 1
vector<int>::iterator intVecIter;     //Line 2
```

第 1 行中的语句将 intList 声明为元素为 int 类型的向量容器。第 2 行中的语句将 intVecIter 声明为元素为 int 类型的向量容器的迭代器。

容器与函数 begin 和 end

所有容器都包含成员函数 begin 和 end。函数 begin 返回容器中第一个元素的位置；函数 end 返回容器中最后一个元素的位置。这两个函数都没有参数。在执行下面语句：

```
intVecIter = intList.begin();
```

迭代器 intVecIter 指向容器 intList 中第一个元素。

下面的 for 循环将 intList 中所有元素输出到标准输出设备上：

```
for (intVecIter = intList.begin(); intVecIter != intList.end();
                                    ++intVecList)
    cout<<*intVecList<<" ";
```

可以通过表 21.3 中给出的操作直接访问向量容器中的元素。

表 21.3　访问向量容器中元素的操作

表达式	作用
vecList.at(index)	返回由 index 指定的位置上的元素
vecList[index]	返回由 index 指定的位置上的元素
vecList.front()	返回第一个元素（不检查容器是否为空）
vecList.back()	返回最后一个元素（不检查容器是否为空）

表 21.3 说明：可以按照数组的方式来处理向量中的元素（注意，在 C++ 中，数组下标从 0 开始。类似地，向量容器中第一个元素的位置也是 0）。

向量类中还包含：返回容器中当前元素个数的成员函数，返回可以插入到容器中的元素的最大个数的成员函数等。表 21.4 描述其中一些操作（假设 vecCont 是向量容器）。

表 21.4　计算向量容器大小的操作

表达式	作用
vecCont.capacity()	返回不重新分配空间可以插入到容器 vecCont 中的元素的最大个数
vecCont.empty()	如果容器 vecCont 为空，返回 true；否则，返回 false
vecCont.size()	返回容器 vecCont 中当前元素的个数
vecCont.max_size()	返回可以插入到容器 vecCont 中的元素的最大个数

例 21.1 说明了如何在程序中使用向量容器，以及怎样操纵向量容器中的元素。

例 21.1

```
#include <iostream>

#include <vector>

using namespace std;

int main()
{
      vector<int>  intList;                          //Line 1
      int i;                                         //Line 2
```

```
        intList.push_back(13);                                  //Line 3
        intList.push_back(75);                                  //Line 4
        intList.push_back(28);                                  //Line 5
        intList.push_back(35);                                  //Line 6

        cout<<"Line 7: List Elements: ";                        //Line 7
        for(i = 0; i < 4; i++)                                  //Line 8
            cout<<intList[ i]<<" ";                             //Line 9
        cout<<endl;                                             //Line 10

        for(i = 0; i < 4; i++)                                  //Line 11
            intList[ i]  *= 2;                                  //Line 12
        cout<<"Line 13: List Elements: ";                       //Line 13
        for(i = 0; i < 4; i++)                                  //Line 14
            cout<<intList[ i]<<" ";                             //Line 15
        cout<<endl;                                             //Line 16

        vector<int>::iterator  listIt;                          //Line 17

        cout<<"Line 18: List Elements: ";                       //Line 18
        for(listIt = intList.begin(); listIt != intList.end();
                                      ++listIt)                 //Line 19
            cout<<*listIt<<" ";                                 //Line 20
        cout<<endl;                                             //Line 21

        listIt = intList.begin();                               //Line 22
        ++listIt;                                               //Line 23
        ++listIt;                                               //Line 24
        intList.insert(listIt,88);    //Insert 88 at the
                                      //position specified
                                      //by listIt               //Line 25

        cout<<"Line 25: List Elements: ";                       //Line 26
        for(listIt = intList.begin(); listIt != intList.end();
                                      ++listIt)                 //Line 27
            cout<<*listIt<<" ";                                 //Line 28
        cout<<endl;                                             //Line 29

        return 0;
}
```

输出

```
Line 7: List Elements: 13 75 28 35
Line 13: List Elements: 26 150 56 70
Line 18: List Elements: 26 150 56 70
Line 25: List Elements: 26 150 88 56 70
```

第 1 行语句声明了一个 int 型向量容器（简称为向量）intList。第 2 行语句将 i 声明为一个 int 类型变量。第 3 行到第 6 行语句通过 push_back 操作将数字 13，75，28，35 插入到 intList 中。第 8 行和第 9 行语句使用 for 循环和数组下标运算符[ ]输出 intList 中的元素。在程序输出中标记为第 7 行的输出，是程序中第 7 行到第 10 行语句的执行结果。第 11 行和第 12 行语句使用 for 循环将 intList 中所有元素加倍。第 14 行和第 15 行语句输出 intList 元素，在程序输出中标记为第 13 行的输出，是程序中第 13 行到第 16 行语句的执行结果。

第 17 行语句将 listIt 声明为一个向量迭代器，它可以用于所有类型为 int 的向量容器。第 19 行和第 20 行语句使用迭代器 listIt 输出 intList 中的元素。在第 22 行语句执行后，listIt 指向 intList 的第 1

个元素。第 23 行和第 24 行语句将 listIt 增量两次，执行了这些语句之后，listIt 指向 intList 中的第 3 个元素。第 25 行语句将 88 插入到 intList 中由迭代器 listIt 指定的位置。由于 listIt 指向位置为 2 的元素（intList 中的第 3 个元素），88 被插入到 intList 中第 2 个位置上。所以，88 为 intList 中的第 3 个元素。第 27 行和第 28 行语句输出了修改后的 intList。

21.1.4 所有容器公共的成员函数

前面一节介绍了向量容器，现在将介绍所有容器都具有的成员函数。例如，所有容器类都有一个默认的构造函数、一个或者多个带有参数的构造函数、一个析构函数、一个将元素插入到容器中的函数等。

类将数据和数据上的操作封装到一个单元中。由于每个容器都是一个类，所以容器中的各种操作也就成为类定义的一部分。数据上的操作由函数来实现，它们被称为这个类的成员函数。表 21.5 描述了所有容器公共的成员函数。即，这些函数为相应容器类模板的成员函数。

假设 ct，ct1 和 ct2 是相同类型的容器。表 21.5 标明了这些公共成员函数的名字，同时介绍怎样调用这些函数。

表 21.5 所有容器公共的操作

运算符函数	作用
默认构造函数	将对象初始化为空
带参数构造函数	除了默认的构造函数之外，每个容器都有带有参数的构造函数。我们将在讨论具体的容器时再介绍这些构造函数
拷贝构造函数	当对象作为值参数传递时，及用另一个相同类型的对象初始化对象时执行该函数
析构函数	当对象超出作用域时执行该函数
`ct.empty()`	如果容器 ct 为空，返回 true；否则，返回 false
`ct.size()`	返回在容器 ct 中当前的元素个数
`ct.max_size()`	返回可以插入到容器 ct 中的元素最大个数
`ct1.swap(ct2)`	交换容器 ct1 和 ct2 中的元素
`ct.begin()`	返回容器中指向第一个元素的迭代器
`ct.end()`	返回容器中指向最后一个元素的迭代器
`ct.rbegin()`	倒置开始位置。该函数返回容器 ct 中最后一个元素的指针，用来倒置处理 ct 中的元素
`ct.rend()`	倒置最后位置。该函数返回容器 ct 中第一个元素的指针
`ct.insert(position,elem)`	将 elem 插入到容器 ct 中由参数 position 指定的位置上。注意，这里 position 是迭代器
`ct.erase(begin, end)`	删除容器 ct 中从 begin 到 end−1 之间的所有元素
`ct.clear()`	删除容器中的所有元素。在调用该函数后，容器 ct 为空
`ct1 = ct2;`	将 ct2 中的所有元素拷贝到 ct1 中
`ct1 == ct2`	如果容器 ct1 和容器 ct2 相等，返回 true；否则，返回 false
`ct1 != ct2`	如果容器 ct1 和容器 ct2 不相等，返回 true；否则，返回 false
`ct1 < ct2`	如果容器 ct1 小于容器 ct2，返回 true；否则，返回 false
`ct1 <= ct2`	如果容器 ct1 小于或者等于容器 ct2，返回 true；否则，返回 false
`ct1 > ct2`	如果容器 ct1 大于容器 ct2，返回 true；否则，返回 false
`ct1 >= ct2`	如果容器 ct1 大于或者等于容器 ct2，返回 true；否则，返回 false

注意：因为上述操作是所有容器共有的，所以为了节省篇幅，在讨论某一具体容器时不再列出这些操作。

21.1.5 顺序容器公共的成员函数

前面一节描述了所有容器共有的成员函数。除了上述成员函数之外，表 21.6 描述了所有顺序容器（即向量、双端队列和表）公共的成员函数。假设 seqCont 是一个顺序容器。

表21.6 顺序容器公共的成员函数

表达式	作用
`seqCont.insert(position, elem)`	将elem的拷贝插入到由position指定的位置上。该函数返回新元素的位置
`seqCont.insert(position, n, elem)`	将*n*个elem的拷贝插入到由position指定的位置上
`seqCont.insert(position, beg, end)`	将从beg到end–1的所有元素的拷贝插入到由position指定的位置上
`seqCont.push_back(elem)`	将elem的拷贝插入到容器seqCont的末尾
`seqCont.pop_back()`	删除最后一个元素
`seqCont.erase(position)`	删除由position指定的元素
`seqCont.erase(beg, end)`	删除从beg到end–1之间的所有元素
`seqCont.clear()`	删除容器中的所有元素
`seqCont.resize(num)`	将容器中元素个数改为num。如果size()增加，新元素由默认的构造函数创建
`seqCont.resize(num,elem)`	将容器中的元素个数改为num。如果size()增加，新元素初始化为elem

21.1.6 copy算法

例21.1使用一个for循环输出向量容器中的元素。STL中的copy函数为输出容器中的元素提供了一个更便利的方法。copy函数是类属算法的一部分，可以使用在所有容器中。由于要频繁地输出容器中的元素，所以在继续讨论容器之前，先介绍该函数。

函数copy所做的工作不仅是输出容器中的元素。而且，它允许将元素从一个地方拷贝到另一个地方。例如，可以使用函数copy输出向量容器中的元素，或是将元素从一个向量容器拷贝到另一个向量容器中。函数模板copy的原型如下所示：

```
template<class inputIterator,class outputIterator>
outputIterator copy(inputIterator first1, inputIterator last,
                    outputIterator first2);
```

参数first1指定拷贝元素的起始位置，参数last指定拷贝元素的结束位置，参数first2指定要将元素拷贝到的位置。即，参数first1和last指定了源位置，参数first2指定了目标位置。

注意，拷贝的是first1到last–1之间的元素。

函数模板copy的定义包含在头文件algorithm中。因此，要使用函数copy，程序必须包含下面语句：

```
#include <algorithm>
```

函数copy的工作方式如下。考虑下面语句：

```
int intArray[] = {5, 6, 8, 3, 40, 36, 98, 29, 75}
```

这条语句创建了一个有9个元素的数组intArray：

```
intArray = {5, 6, 8, 3, 40, 36, 98, 29, 75};
```

这里intArray[0] = 5，intArray[1] = 6，依次类推。

语句：

```
vector<int> vecList(9);
```

创建了一个向量类型的空容器，该容器中有9个int型元素。

intArray实际上是一个指针，它指向该数组的基地址。因此，intArray指向了数组中的第一个元素，intArray+1指向数组中的第二个元素，依次类推。

下面考虑语句：

```
copy(intArray, intArray+9, vecList.begin( ));
```

该语句将从intArray（数组intArray中第一个元素）到intArray+9-1（intArray+8，数组intArray中最后一个元素）之间的所有元素拷贝到容器vecList中（注意：这里与形参first1对应的实参是intArray，与last对应的实参是intArray +9，与first2对应的是vecList.begin( )）。在上面语句执行之后：

```
vecList = {5, 6, 8, 3, 40, 36, 98, 29, 75}
```

下面考虑语句：

```
copy(intArray+1, intArray+9, intArray);
```

这里与形参first1对应的是实参intArray+1，即first1指向数组intArray中的第二个元素；与last对应的是实参intArray+9；与first2对应的是实参intArray，即first2指向数组intArray中的第一个元素。因此，该函数将第二个数组元素拷贝到第一个数组元素的位置上，第三个数组元素拷贝到第二个数组元素的位置上，依次类推。在上面语句执行后：

```
intArray = {6, 8, 3, 40, 36, 98, 29, 75, 75}
```

显然，数组intArray中的所有元素都向左移动了一个位置。

考虑下面语句：

```
copy(vecList.rbegin( )+2, vecList.rend( ), vecList.rbegin( ));
```

前面讲过，函数rbegin返回倒置之后的第一个元素的指针；它可以用来处理一个倒置之后的容器中的元素。因此，vecList.rbegin( )+2返回指向容器vecList中倒数第三个元素的指针。类似地，函数rend返回倒置之后的最后一个元素的指针。上面的语句将容器vecList中的元素向右移动了两位。执行了上面的语句之后，容器：

```
vecList = {5, 6, 5, 6, 8, 3, 40, 36, 98}
```

例21.2使用一个C++程序说明前面语句的执行结果。在讨论例21.2之前，先介绍一个特殊的迭代器，称为ostream迭代器。这种迭代器可以与函数copy联合使用，将元素从一个容器拷贝到输出设备中。

ostream迭代器和函数copy

for循环是输出容器中内容的一种方式。使用函数begin初始化for循环的控制变量，使用函数end设置边界条件，或使用函数copy输出容器中的内容。在这种情形下，ostream类型的迭代器就指定了目标（ostream迭代器将在本章后面讨论）。当创建一个ostream类型迭代器时，我们也同时指定了该迭代器输出的元素类型。

下面语句说明了怎样创建int类型的ostream迭代器：

```
ostream_iterator<int> screen(cout," ");          //Line A
```

该语句创建了ostream类型的迭代器screen，元素类型为int。迭代器screen有两个参数，对象cout和空格。因此，对象cout用来初始化迭代器screen；当该迭代器输出元素时，用空格分隔。

语句：

```
copy(intArray, intArray+9, screen);
```

将intArray中的元素输出到screen。

类似地，语句：

```
copy(vecList.begin(), vecList.end(), screen);
```

将容器 vecList 中的元素输出到 screen。

我们将频繁地使用函数copy和迭代器ostream输出容器中的元素。在详细讨论ostream迭代器之前，将继续使用类似于上面语句所创建的一个 ostream 迭代器。

当然，还可以直接在函数 copy 中指定 ostream 迭代器。例如，前面出现过的语句：

```
copy(vecList.begin(), vecList.end(), screen);
```

等价于下面语句：

```
copy(vecList.begin(), vecList.end(), ostream_iterator<int>(cout," "));
```

下面语句：

```
copy(vecList.begin(), vecList.end(), ostream_iterator<int>(cout, ", "));
```

输出 vecList 中的元素，并以逗号和空格分隔。

例 21.2 说明了如何在程序中使用函数 copy 和迭代器 ostream。

例 21.2

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>

using namespace std;

int main()
{
    int intArray[] = {5, 6, 8, 3, 40, 36, 98, 29, 75};       //Line 1

    vector<int> vecList(9);                                  //Line 2

    ostream_iterator<int> screen(cout, " ");                 //Line 3

    cout<<"Line 4: intArray: ";                              //Line 4
    copy(intArray, intArray + 9, screen);                    //Line 5
    cout<<endl;                                              //Line 6

    copy(intArray, intArray + 9, vecList.begin());           //Line 7

    cout<<"Line 8: vecList:";                                //Line 8
    copy(vecList.begin(), vecList.end(), screen);            //Line 9
    cout<<endl;                                              //Line 10

    copy(intArray+1, intArray + 9, intArray);                //Line 11
    cout<<"Line 12: After shifting the elements one "
        <<"position to the left, intArray: "<<endl;          //Line 12
    copy(intArray, intArray + 9, screen);                    //Line 13
    cout<<endl;                                              //Line 14

    copy(vecList.rbegin() + 2, vecList.rend(),
                          vecList.rbegin());                 //Line 15
    cout<<"Line 16: After shifting the elements down "
        <<"two positions, vecList:"<<endl;                   //Line 16
    copy(vecList.begin(), vecList.end(), screen);            //Line 17
```

```
    cout<<endl;                                              //Line 18

    return 0;
}
```

输出

```
Line 4: intArray: 5 6 8 3 40 36 98 29 75
Line 8: vecList:5 6 8 3 40 36 98 29 75
Line 12: After shifting the elements one position to the left,
intArray:
6 8 3 40 36 98 29 75 75
Line 16: After shifting the elements down two positions, vecList:
5 6 5 6 8 3 40 36 98
```

21.1.7　顺序容器：双端队列

本节将介绍顺序容器双端队列（Deque）。双端队列容器是由动态数组实现的，可在两端插入元素。因此，双端队列既可以在两端插入元素，又可以在中间插入元素。在双端队列的头和尾插入元素要比在中间插入快得多，因为在队列中间插入元素需要移位。

定义双端队列容器类的名字是 deque。类 deque 的定义，以及实现该类对象上各种操作的函数也都包含在头文件 deque 中。因此，如果在程序中使用双端队列容器，就要包含下面语句：

```
#include <deque>
```

类 deque 包含了多个构造函数，因此可以在声明时以多种方式初始化 deque 对象，如表 21.7 所示。

表 21.7　各种声明 deque 对象的方法

语句	作用
`deque<elementType> deq;`	创建一个不含任何元素的空双端队列 deq（使用默认构造函数）
`deque<elementType> deq(otherDeq);`	创建一个双端队列容器 deq，并使用 otherDeq 中的元素将其初始化，deq 和 otherDeq 的类型相同
`deque<elementType> deq(size);`	创建一个大小为 size 的双端队列 deq，并使用默认的构造函数将其初始化
`deque<elementType> deq(n,elem);`	创建一个大小为 *n* 的双端队列 deq，并使用 *n* 个 elem 元素拷贝将其初始化
`deque<elementType> deq(begin, end);`	创建一个双端队列容器 deq，并使用 begin 和 end-1 之间的元素将其初始化

除了所有容器都共有的操作之外（见表 21.6），表 21.8 描述了双端队列容器中元素的特有操作，及实现这些操作的函数名。同时，还说明了该如何使用每一个函数。假设 deq 是一个双端队列容器。

表 21.8　在 deque 对象上可执行的各种操作

表达式	作用
`deq.assign(n,elem)`	赋值 *n* 个 elem 的拷贝给双端队列
`deq.assign(beg,end)`	赋值从 beg 到 end-1 的所有元素给双端队列
`deq.push_front(elem)`	将 elem 插入到 deq 的开始处
`deq.pop_front()`	将 deq 中的第一个元素删除
`deq.at(index)`	返回由 index 指定的元素
`deq[index]`	返回由 index 指定的元素
`deq.front()`	返回第一个元素（不检查容器是否为空）
`deq.back()`	返回最后一个元素（不检查容器是否为空）

例 21.3 说明了怎样在程序中使用双端队列容器。

例 21.3

```
//Deque Example
#include <iostream>
#include <deque>
#include <algorithm>
#include <iterator>

using namespace std;

int main()
{
    deque<int>  intDeq;                                     //Line 1
    ostream_iterator<int> screen(cout, " ");                //Line 2

    intDeq.push_back(13);                                   //Line 3
    intDeq.push_back(75);                                   //Line 4
    intDeq.push_back(28);                                   //Line 5
    intDeq.push_back(35);                                   //Line 6

    cout<<"Line 7: intDeq: ";                               //Line 7
    copy(intDeq.begin(), intDeq.end(), screen);             //Line 8
    cout<<endl;                                             //Line 9

    intDeq.push_front(0);                                   //Line 10
    intDeq.push_back(100);                                  //Line 11

    cout<<"Line 12: After adding two more "
        <<"elements, one at the front "<<endl
        <<"         and one at the back, intDeq: ";        //Line 12
    copy(intDeq.begin(), intDeq.end(), screen);             //Line 13
    cout<<endl;                                             //Line 14

    intDeq.pop_front();                                     //Line 15
    intDeq.pop_front();                                     //Line 16

    cout<<"Line 17: After removing the first "
         <<"two elements, intDeq: ";                        //Line 17
    copy(intDeq.begin(), intDeq.end(), screen);             //Line 18
    cout<<endl;                                             //Line 19

    intDeq.pop_back();                                      //Line 20
    intDeq.pop_back();                                      //Line 21

    cout<<"Line 22: After removing the last "
        <<"two elements, intDeq = ";                        //Line 22
    copy(intDeq.begin(), intDeq.end(), screen);             //Line 23
    cout<<endl;                                             //Line 24

    deque<int>::iterator  deqIt;                            //Line 25
    deqIt = intDeq.begin();                                 //Line 26
    ++deqIt;                     //deqIt points to the
                                 //second element           //Line 27
    intDeq.insert(deqIt,444);    //Insert 444 at the
                                 //location deqIt           //Line 28
    cout<<"Line 29: After inserting 444, intDeq:  ";        //Line 29
    copy(intDeq.begin(), intDeq.end(), screen);             //Line 30
    cout<<endl;                                             //Line 31
```

```
    intDeq.assign(2,45);                                    //Line 32

    cout<<"Line 33: After assigning two "
        <<"copies of 45, intDeq: ";                         //Line 33
    copy(intDeq.begin(), intDeq.end(), screen);             //Line 34
    cout<<endl;                                             //Line 35

    intDeq.push_front(-10);                                 //Line 36
    intDeq.push_back(-999);                                 //Line 37

    cout<<"Line 38: After inserting two "
        <<"elements, one at the front "<<endl
        <<"         and one at the back, intDeq: ";         //Line 38
    copy(intDeq.begin(), intDeq.end(), screen);             //Line 39
    cout<<endl;                                             //Line 40

    return 0;
}
```

输出

```
Line 7: intDeq: 13 75 28 35
Line 12: After adding two more elements, one at the front
         and one at the back, intDeq: 0 13 75 28 35 100
Line 17: After removing the first two elements, intDeq: 75 28 35 100
Line 22: After removing the last two elements, intDeq = 75 28
Line 29: After inserting 444, intDeq:  75 444 28
Line 33: After assigning two copies of 45, intDeq: 45 45
Line 38: After inserting two elements, one at the front
         and one at the back, intDeq: -10 45 45 -999
```

第1行语句声明了一个int类型的双端队列容器intDeq。第2行语句将screen声明为一个ostream迭代器，并将其初始化为标准输出设备。第3行到第6行语句使用函数push_back在intDeq中插入4个数字——13，75，28和35。第8行语句输出intDeq中的元素。在程序输出中标有第7行的输出，是程序第7行到第9行语句的输出结果。

第10行语句在intDeq的头部插入0；第11行语句在intDeq的尾部插入100。第13行语句输出了修改后的intDeq。第15行和第16行语句使用pop_front函数删除了intDeq中的前两个元素。第18行语句输出了修改后的intDeq。第20行和第21行语句使用pop_back函数删除intDeq的最后两个元素。第23行语句输出了修改后的intDeq。

第25行语句将deqIt声明为一个双端队列迭代器，它可以用来处理所有int类型的双端队列容器。在执行了第26行语句后，deqIt指向intDeq中的第一个元素。第27行语句使deqIt指向intDeq的下一个元素。第28行语句把444插入到intDeq中由deqIt指定的位置。第30行语句输出intDeq。

第32行语句将45的两份拷贝赋给intDeq。在执行第32行语句后，intDeq中所有原有元素都被删除，只剩下两个45。第34行的输出语句说明了上述过程。在程序输出中标有第33行的输出，是程序中第33行到第35行语句的输出。

其他语句很简单，不再解释。

21.1.8 顺序容器：表

本节将描述顺序容器表（List）。表容器是用双向链表实现的。因此，表中的每个元素都既有指向其直接前驱的指针，又有指向其直接后继的指针（第一个元素和最后一个元素除外）。由于链表不是一个可以随机访问的数据结构，所以要访问表中的第5个元素，就必须先访问前4个元素。

包含表定义的类的名字是list。类list和表上各种操作的定义都包含在头文件list中。因此，要在程序中使用list，就必须在程序中包含下面语句：

```
#include <list>
```

与其他容器类相似，类list包含了多个构造函数，因此可以在声明时以多种方式初始化list对象，如表21.9所示。

表21.9 各种声明list对象的方法

语句	作用
`list<elementType> listCont;`	创建一个没有任何元素的空表listCont（使用默认构造函数）
`list<elementType> listCont(otherList);`	创建表listCont，并使用otherList中的元素将其初始化。listCont和otherList的类型相同
`list<elementType> listCont(size);`	创建一个大小为size的表listCont，并使用默认构造函数将其初始化
`list<elementType> listCont(n, elem);`	创建一个大小为n的表listCont，并使用n个elem元素将其初始化
`list<elementType> listCont(begin, end);`	创建一个表listCont，并使用begin和end-1之间的元素将其初始化

表21.5中描述了所有容器所共有的操作，表21.6描述了所有顺序容器所共有的操作。除了这些公共的操作外，表21.10描述了表容器所特有的操作。同时，给出了实现这些操作的函数名。

表21.10 表容器特有的操作

表达式	作用
`listCont.assign(n, elem)`	赋值n个elem的拷贝给表
`listCont.assign(beg, end)`	赋值从beg到end-1的所有元素给表
`listCont.push_front(elem)`	将elem插入到listCont的开始处
`listCont.pop_front()`	删除listCont中的第一个元素
`listCont.front()`	返回第一个元素（不检查容器是否为空）
`listCont.back()`	返回最后一个元素（不检查容器是否为空）
`listCont.remove(elem)`	删除所有等于elem的元素
`listCont.remove_if(oper)`	删除所有oper(elem)为true的元素
`listCont.unique()`	如果listCont中含有相同元素，删除重复的元素
`listCont.unique(oper)`	如果listCont中含有相同元素，删除所有oper(elem)为true的重复元素
`listCont1.splice(pos, listCont2)`	将listCont2中的所有元素移到listCont1中由迭代器pos指定的位置上。在该操作执行后，listCont2为空
`listCont1.splice(pos,listCont2,pos2)`	将listCont2中由pos2开始的所有元素移到listCont1中由迭代器pos指定的位置上
`listCont1.splice(pos, listCont2, beg, end)`	将listCont2中从beg到end-1之间的所有元素移到listCont1中由迭代器pos指定的位置上
`listCont.sort()`	将listCont中的所有元素按升序顺序排序
`listCont.sort(oper)`	将listCont中的所有元素按oper指定的顺序排序
`listCont1.merge(listCont2)`	假设listCont1和listCont2中所有元素都已经有序，该操作将listCont2中的所有元素移到listCont1中。在该操作执行后，listCont2为空，listCont1中元素仍然有序
`listCont1.merge(listCont2,oper)`	假设listCont1和listCont2中的所有元素按oper指定的顺序有序，该操作将listCont2中的所有元素移到listCont1中。在该操作执行后，listCont2为空，listCont1中元素仍然按oper指定的顺序排序
`listCont.reverse()`	将listCont中所有元素倒置

例 21.4 说明了怎样使用表容器上的各种操作。

例 21.4

```
//List Container Example

#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

using namespace std;

int main()
{
    list<int> intList1, intList2, intList3, intList4;     //Line 1

    ostream_iterator<int> screen(cout," ");               //Line 2

    intList1.push_back(23);                               //Line 3
    intList1.push_back(58);                               //Line 4
    intList1.push_back(58);                               //Line 5
    intList1.push_back(58);                               //Line 6
    intList1.push_back(36);                               //Line 7
    intList1.push_back(15);                               //Line 8
    intList1.push_back(93);                               //Line 9
    intList1.push_back(98);                               //Line 10
    intList1.push_back(58);                               //Line 11

    cout<<"Line 12: intList1: ";                          //Line 12
    copy(intList1.begin(),intList1.end(),screen);         //Line 13
    cout<<endl;                                           //Line 14

    intList2 = intList1;                                  //Line 15

    cout<<"Line 16: intList2: ";                          //Line 16
    copy(intList2.begin(),intList2.end(),screen);         //Line 17
    cout<<endl;                                           //Line 18

    intList1.unique();                                    //Line 19

    cout<<"Line 20: After removing the consecutive "
        <<"duplicates,"<<endl
        <<"            intList1: ";                       //Line 20
    copy(intList1.begin(),intList1.end(),screen);         //Line 21
    cout<<endl;                                           //Line 22

    intList2.sort();                                      //Line 23

    cout<<"Line 24: After sorting, intList2: ";           //Line 24
    copy(intList2.begin(),intList2.end(),screen);         //Line 25
    cout<<endl;                                           //Line 26

    intList3.push_back(13);                               //Line 27
    intList3.push_back(23);                               //Line 28
    intList3.push_back(25);                               //Line 29
    intList3.push_back(136);                              //Line 30
```

```
    intList3.push_back(198);                               //Line 31

    cout<<"Line 32: intList3: ";                           //Line 32
    copy(intList3.begin(),intList3.end(),screen);          //Line 33
    cout<<endl;                                            //Line 34

    intList4.push_back(-2);                                //Line 35
    intList4.push_back(-7);                                //Line 36
    intList4.push_back(-8);                                //Line 37

    cout<<"Line 38: intList4: ";                           //Line 38
    copy(intList4.begin(),intList4.end(),screen);          //Line 39
    cout<<endl;                                            //Line 40

    intList3.splice(intList3.begin(),intList4);            //Line 41

    cout<<"Line 42: After moving the elements of "
        <<"intList4 into intList3,"<<endl
        <<"          intList3: ";                          //Line 42
    copy(intList3.begin(),intList3.end(),screen);          //Line 43
    cout<<endl;                                            //Line 44

    intList3.sort();                                       //Line 45

    cout<<"Line 46: After sorting, intList3: ";            //Line 46
    copy(intList3.begin(),intList3.end(),screen);          //Line 47
    cout<<endl;                                            //Line 48

    intList2.merge(intList3);                              //Line 49

    cout<<"Line 50: After merging intList2 and intList3, "
        <<"intList2: "<<endl<<"          ";                //Line 50
    copy(intList2.begin(),intList2.end(),screen);          //Line 51
    cout<<endl;                                            //Line 52

    intList2.unique();                                     //Line 53

    cout<<"Line 54: After removing the consecutive "
        <<"duplicates, intList2: "<<endl
        <<"          ";                                    //Line 54
    copy(intList2.begin(),intList2.end(),screen);          //Line 55
    cout<<endl;                                            //Line 56

    return 0;
}
```

输出

```
Line 12: intList1: 23 58 58 58 36 15 93 98 58
Line 16: intList2: 23 58 58 58 36 15 93 98 58
Line 20: After removing the consecutive duplicates,
         intList1: 23 58 36 15 93 98 58
Line 24: After sorting, intList2: 15 23 36 58 58 58 58 93 98
Line 32: intList3: 13 23 25 136 198
Line 38: intList4: -2 -7 -8
Line 42: After moving the elements of intList4 into intList3,
         intList3: -2 -7 -8 13 23 25 136 198
```

```
Line 46: After sorting, intList3: -8 -7 -2 13 23 25 136 198
Line 50: After merging intList2 and intList3, intList2:
         -8 -7 -2 13 15 23 23 25 36 58 58 58 58 93 98 136 198
Line 54: After removing the consecutive duplicates, intList2:
         -8 -7 -2 13 15 23 25 36 58 93 98 136 198
```

程序中大部分语句都很简单。第 3 行到第 11 行语句将数字 23，58，58，58，36，15，93，98 和 58（按这个顺序）插入到 intList1 中。第 15 行语句将 intList1 中的元素拷贝到 intList2 中。在执行完这条语句后，intList1 和 intList2 相等。第 19 行语句删除所有连续出现的相同元素。例如，数字 58 连续出现了 3 次，操作 unique 删除了后两个 58。注意这个操作没有影响 intList1 尾部出现的 58。第 23 行语句对 intList2 进行排序。第 27 行到第 31 行语句将 13，23，25，136 和 198 插入到 intList3 中。类似地，第 35 行到第 37 行语句将 -2，-7，-8 插入到 intList4 中。第 41 行语句使用操作 splice 将 intList4 中的元素移到 intList3 首部。在执行了 splice 操作后，intList4 为空。第 45 行语句对 intList3 进行排序。第 49 行语句将 intList2 和 intList3 合并到 intList2 中。在 merge 操作后，intList3 为空。其他语句的含义类似。

例 21.2 到例 21.4 再次强调了迭代器在有效地处理容器中元素的重要性。在介绍关联容器之前，让我们先对迭代器进行详细地讨论。

21.2　迭代器

迭代器类似于指针。也就是说，迭代器指向容器（顺序的或关联的）中的元素。因此，通过迭代器，可以访问容器中的每个元素。

在迭代器上经常使用的两个操作是 ++（增量运算符）和 *（递引用运算符）。假设 cntItr 是某个容器的迭代器，语句：

```
++cntItr;
```

将 cntItr 加 1，使其指向容器中的下一个元素。类似地，语句：

```
*cntItr;
```

访问容器中由 cntItr 指向的元素。

21.2.1　迭代器的类型

迭代器有 5 种类型：

- 输入迭代器
- 输出迭代器
- 前向迭代器
- 双向迭代器
- 随机访问迭代器

下面将详细介绍上述 5 种迭代器。

输入迭代器

输入迭代器，用于读操作，依次访问每个元素，并返回元素的值。这种迭代器用于从输入流中读取数据。

假设 inputIterator 是一个输入迭代器，表 21.11 描述了 inputIterator 上的操作。

表 21.11 输入迭代器上的操作

表达式	作用
*inputIterator	访问 inputIterator 所指的元素
inputIterator->member	访问成员 member
++inputIterator	指向并返回下一个位置（前置增量）
inputIterator++	指向并返回下一个位置（后置增量）
inputIt1 == inputIt2	如果两个迭代器相等，返回 true；否则，返回 false
inputIt1 != inputIt2	如果两个迭代器不相等，返回 true；否则，返回 false

输出迭代器

输出迭代器，用于写操作，依次访问每个元素。这种迭代器用于向输出流中写入数据。

假设 outputIterator 是一个输出迭代器，表 21.12 描述了 outputIterator 上的操作。

表 21.12 输出迭代器上的操作

表达式	作用
*outputIterator = Value;	将 value 写到 outputIterator 指向的位置上
++outputIterator	指向并返回下一个位置（前置增量）
outputIterator++	指向并返回下一个位置（后置增量）

注意： 输出迭代器不能在某个范围上使用两次。因此，如果在同一个位置上写两次数据，不能保证新值一定会替代旧值。

前向迭代器

前向迭代器结合了所有输入迭代器的功能和几乎所有输出迭代器的功能。假设 forwardIterator 是一个前向迭代器，表 21.13 描述了 forwardIterator 上的操作。

表 21.13 前向迭代器上的操作

表达式	作用
*forwardIterator	访问 forwardIterator 所指的元素
forwardIterator->member	访问 forwardIterator 的成员 member
++forwardIterator	指向并返回下一个位置（前置增量）
forwardIterator++	指向并返回下一个位置（后置增量）
forwardIt1 == forwardIt2	如果两个迭代器相等，返回 true；否则，返回 false
forwardIt1 != forwardIt2	如果两个迭代器不相等，返回 true；否则，返回 false
forwardIt1 = forwardIt2	赋值

注意： 前向迭代器可以引用同一个集合中的同一元素，并且可以多次处理同一元素。

双向迭代器

双向迭代器既可以后向访问元素，又可以前向访问元素。假设biDirectionalIterator是一个双向迭代器。前向迭代器上的操作（参见表 21.13）同样也适用于双向迭代器。为了可以后向访问，要为 biDirectional-Iterator 定义减量运算。表 21.14 说明了双向迭代器上附加的操作。

表 21.14 双向迭代器上的附加操作

表达式	作用
--biDirectionalIterator	指向并返回前一个位置（前置减量）
biDirectionalIterator--	指向并返回前一个位置（后置减量）

注意： 双向迭代器可以在 list，set，multiset，map 和 multimap 等类型的容器上使用。

随机访问迭代器

随机访问迭代器是可以随机访问容器中元素的双向迭代器。这种迭代器可以在 vector，dequeue，

string 和数组类型容器上使用。在双向迭代器上定义的操作（如表 21.13 和表 21.14 所示）也适用于随机访问迭代器。假设 rAccessIterator 是一个随机访问迭代器，表 21.15 描述了随机访问迭代器上附加的操作。

表 21.15　随机访问迭代器上的附加操作

表达式	作用
`rAccessIterator[n]`	访问第 n 个元素
`rAccessIterator += n`	如果 n>=0，rAccessIterator 向前移动 n 个元素；否则，向后移动 n 个元素
`rAccessIterator -= n`	如果 n>=0，rAccessIterator 向后移动 n 个元素；否则，向前移动 n 个元素
`rAccessIterator + n`	返回下面第 n 个元素的迭代器
`n + rAccessIterator`	返回下面第 n 个元素的迭代器
`rAccessIterator - n`	返回前面第 n 个元素的迭代器
`rAccessIt1 - rAccessIt2`	返回 rAccessIt1 和 rAccessIt2 之间的元素个数
`rAccessIt1 < rAccessIt2`	如果 rAccessIt1 在 rAccessIt2 之前，返回 true；否则，返回 false
`rAccessIt1 <= rAccessIt2`	如果 rAccessIt1 在 rAccessIt2 之前或与之相等，返回 true；否则，返回 false
`rAccessIt1 > rAccessIt2`	如果 rAccessIt1 在 rAccessIt2 之后，返回 true；否则，返回 false
`rAccessIt1 >= rAccessIt2`	如果 rAccessIt1 在 rAccessIt2 之后或与之相等，返回 true；否则，返回 false

图 21.1 说明了迭代器的层次关系。

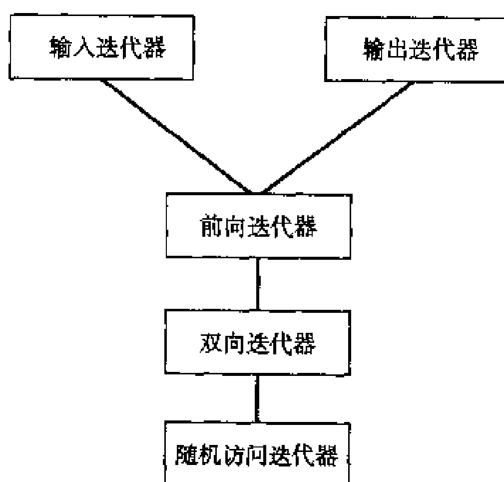


图 21.1　迭代器的层次关系

在介绍了不同类型的迭代器之后，下面说明如何为容器声明迭代器。

typedef iterator

每个容器（顺序的或关联的）都包含了一个 typedef iterator。因此，要用 typedef iterator 声明容器的迭代器。例如，语句：

```
vector<int>::iterator intVecIter;
```

将 intVecIter 声明为 int 类型的 vector 容器迭代器。

由于 iterator 是在容器（它是一个类，如 vector）中定义的，所以在使用 typedef iterator 时，必须使用合适的容器名称、容器元素类型和作用域运算符。

typedef const_iterator

由于迭代器的工作原理与指针相似，所以通过容器迭代器和递引用运算符*，就可以修改容器中的元素。然而，如果一个容器被声明为const，那么就必须防止迭代器修改该容器中的元素，尤其是要防止意外地修改。为处理这种情况，所有容器还包含了一个typedef const_iterator。例如，语句：

```
vector<int>::const_iterator intConstVecIt;
```

将intConstVecIt声明为元素是int类型的向量容器的迭代器。迭代器intConstVecIt用来处理类型为vector<int>的常量向量容器中的元素。

类型为const_iterator的迭代器是只读迭代器。

typedef reverse_iterator

所有容器还包含了typedef reverse_iterator。这种类型的迭代器用于逆向访问容器中的元素。

typedef const_reverse_iterator

这种类型的迭代器是只读迭代器，用来逆向访问容器中的元素。如果容器被声明为const并且需要逆向访问该容器，就可以使用这种迭代器。

除了上面4种typedef，所有容器还具有其他几种typedef，表21.16中描述了这些typedef。

表21.16 所有容器共有的typedef

typedef	作用
difference_type	同一容器上两个迭代器相减结果的类型
pointer	容器中元素类型的指针
reference	容器中元素类型的引用
const_reference	容器中元素类型的常量引用，常量引用是只读的
size_type	容器中用来统计元素个数的类型，该类型还可以用于除了表容器之外的顺序容器的检索
value_type	容器中元素的类型

21.2.2 流迭代器

除了上面介绍的迭代器之外，还有一种很有用的迭代器——流迭代器，它包括istream迭代器和ostream迭代器。本节将介绍这两种迭代器。

istream_iterator

istream迭代器用来从输入流向程序中输入数据。istream_iterator类包含了输入流迭代器的定义。通常istream迭代器的使用语法是：

```
istream_iterator<Type> isIdentifier(istream&);
```

其中Type既可以是内建类型，也可以是用户自定义类类型，并在该类上定义了一个输入流迭代器。标识符isIdentifier由构造函数初始化，其参数既可以是istream类对象（如cin），也可以是任何明确定义的istream子类型（如ifstream）。

ostream_iterator

ostream迭代器用来将程序中的数据输出到输出流中。这种迭代器在本章前面已有定义。但是，为了完整，这里再介绍一次。

ostream_iterator类包含了一个输出流迭代器的定义。通常ostream迭代器的使用语法是：

```
ostream_iterator<Type> osIdentifier(ostream&);
```

或者：

```
ostream_iterator<Type> osIdentifier(ostream&, char* deLimit);
```

其中Type既可以是内建类型，也可以是用户自定义类类型，并在该类上定义了一个输出流迭代器。标识符osIdentifier用构造函数初始化，其参数是ostream类对象（如cout），或是任何明确定义的ostream子类型（如ofstream）。在第二种形式的ostream迭代器声明中，可以通过在初始化构造函数中使用第二个参数（deLimit）来指定分隔符。

21.3　关联容器

本节讨论关联容器。关联容器中的元素是根据某个排序准则自动排序的。默认的排序准则是关系运算符<（小于）。用户也可以指定其他的排序准则。

由于关联容器中的元素自动排序，所以在向容器中插入元素时，该元素就会被插到适当的位置上。实现这种数据结构的便捷方式是使用二叉搜索树。事实上，关联容器的确也是这样实现的。因此，每个容器中的元素都有一个父节点（除了根节点外）和两个子节点。对每个元素，父节点的键值一定大于左孩子的键值而小于右孩子的键值。

STL 中预定义的关联容器是：

- 集合（Set）
- 多重集合（Multiset）
- 映射（Map）
- 多重映射（Multimap）

本书仅讨论关联容器集合和多重集合。

21.3.1　关联容器：集合和多重集合

如前面所述，集合和多重集合中的元素都是根据某个排序准则自动排序的。默认的排序准则是关系运算符<（小于），即元素按升序排列。用户也可以指定其他的排序准则。对于用户自定义的数据类型（如类），关系运算符必须重载。

容器集合和多重集合之间惟一的区别是：容器多重集合允许元素重复，而容器集合则不允许元素重复。

定义容器集合的类名是set；定义容器多重集合的类名是multiset。set和multiset的类定义，以及实现各种操作的函数定义都包含在头文件set中。因此，要使用这些容器，必须在程序中包含下面语句：

```
#include <set>
```

21.3.2　声明集合和多重集合关联容器

类set和multiset包含了多种构造函数来声明和初始化这些类型容器。本小节将讨论声明和初始化这些类型容器的方法。表21.17介绍了如何声明和初始化指定类型的set/multiset容器。

表 21.17　声明 set/multiset 容器的方法

语句	作用
`ctType<elmType> ct;`	创建一个空的set/multiset容器ct，排序准则是<
`ctType<elmType, sortOp> ct;`	创建一个空的set/multiset容器ct，排序准则由sortOp指定
`ctType<elmType> ct(otherCt);`	创建一个set/multiset容器ct，otherCt中的元素拷贝到ct中，排序准则是<。ct与otherCt的类型相同
`ctType<elmType, sortOp> ct(otherCt);`	创建一个set/multiset容器ct，otherCt中的元素拷贝到ct中，排序准则由sortOp指定。ct与otherCt的类型相同。注意，ct和otherCt的排序准则必须相同

（续表）

语句	作用
ctType<elmType> ct(beg, end);	创建一个set/multiset容器ct，从beg到end-1之间的元素被拷贝到ct中。beg和end都是迭代器
ctType<elmType, sortOp> ct(beg, end);	创建一个set/multiset容器ct，从beg到end-1之间的元素被拷贝到ct中，排序准则由sortOp指定。beg和end都是迭代器

如果不想使用默认的排序准则，就必须在声明容器时指定其他的排序准则。例如，考虑下面语句：

```
set<int>                              intSet;                    //Line1
set<int, greater<int> >               otherIntSet;               //Line2
multiset<string>                      stringMultiset;            //Line3
multiset<string, greater<string> >    otherStringMultiset;       //Line4
```

第1行语句将intSet声明为一个空的set容器，元素类型是int，使用的是默认排序准则。第2行语句将otherIntSet声明为一个空的set容器，元素类型是int，排序准则是大于（>）。即，容器otherIntSet中的元素将按降序排列。第3行和第4行语句与之类似。

第2行和第4行语句说明了怎样指定降序排序准则。

注意：在第2行和第4行语句中，在两个>符号之间（即greater<int>和>）的空格很重要。因为在C++中，>>是移位运算符。

21.3.3 集合/多重集合中元素的插入和删除

假设ct是set或multiset类型容器。表21.18介绍了集合中元素的插入和删除操作，并且给出了实现这些操作的函数名。

表21.18 集合中元素的插入和删除操作

表达式	使用
ct.insert(elem)	将elem的拷贝插入到ct中，并返回插入操作是否成功
ct.insert(position, elem)	将elem的拷贝插入到ct中，并返回elem被插入的位置。第一个参数position是迭代器，用来指定insert操作开始搜索的位置
ct.insert(beg, end);	将从beg到end-1之间的所有元素插入到ct中，beg和end都是迭代器
ct.erase(elem);	将所有值为elem的元素删除，并返回被删除的元素个数
ct.erase(position);	将由迭代器position指定的元素删除，不返回任何值
ct.erase(beg, end);	将从beg到end-1之间的所有元素删除，beg和end都是迭代器，不返回任何值
ct.clear();	将容器ct中所有元素删除。在该操作后，容器ct为空

例21.5说明了集合/多重集合容器中的各种操作。

例21.5

```
#include <iostream>
#include <set>
#include <string>
#include <iterator>
#include <algorithm>

using namespace std;

int main()
{
   set<int>  intSet;                                        //Line 1
   set<int, greater<int> >  intSetA;                        //Line 2
```

```
set<int, greater<int> >::iterator  intGtIt;           //Line 3
ostream_iterator<int> screen(cout, " ");                //Line 4

intSet.insert(16);                                      //Line 5
intSet.insert(8);                                       //Line 6
intSet.insert(20);                                      //Line 7
intSet.insert(3);                                       //Line 8

cout<<"Line 9: intSet: ";                               //Line 9
copy(intSet.begin(), intSet.end(), screen);             //Line 10
cout<<endl;                                             //Line 11

intSetA.insert(36);                                     //Line 12
intSetA.insert(84);                                     //Line 13
intSetA.insert(30);                                     //Line 14
intSetA.insert(39);                                     //Line 15
intSetA.insert(59);                                     //Line 16
intSetA.insert(238);                                    //Line 17
intSetA.insert(156);                                    //Line 18

cout<<"Line 19: intSetA: ";                             //Line 19
copy(intSetA.begin(), intSetA.end(), screen);           //Line 20
cout<<endl;                                             //Line 21

intSetA.erase(59);                                      //Line 22

cout<<"Line 23: After removing 59, intSetA: ";          //Line 23
copy(intSetA.begin(), intSetA.end(), screen);           //Line 24
cout<<endl;                                             //Line 25

intGtIt = intSetA.begin();                              //Line 26
++intGtIt;                                              //Line 27
++intGtIt;                                              //Line 28
++intGtIt;                                              //Line 29

intSetA.erase(intGtIt);                                 //Line 30

cout<<"Line 31: After removing the fourth element, "
    <<endl<<"          intSetA: ";                      //Line 31
copy(intSetA.begin(), intSetA.end(), screen);           //Line 32
cout<<endl;                                             //Line 33

set<int, greater<int> >  intSetB(intSetA);              //Line 34

cout<<"Line 35: intSetB: ";                             //Line 35
copy(intSetB.begin(), intSetB.end(), screen);           //Line 36
cout<<endl;                                             //Line 37

intSetB.clear();                                        //Line 38

cout<<"Line 39: After removing all the elements, "
    <<endl<<"          intSetB: ";                      //Line 39
copy(intSetB.begin(), intSetB.end(), screen);           //Line 40
cout<<endl;                                             //Line 41

multiset<string, greater<string> > namesMultiSet;       //Line 42
multiset<string, greater<string> >::iterator iter;      //Line 43
```

```
    ostream_iterator<string> pScreen(cout, " ");            //Line 44

    namesMultiSet.insert("Donny");                          //Line 45
    namesMultiSet.insert("Zippy");                          //Line 46
    namesMultiSet.insert("Goofy");                          //Line 47
    namesMultiSet.insert("Hungry");                         //Line 48
    namesMultiSet.insert("Goofy");                          //Line 49
    namesMultiSet.insert("Donny");                          //Line 50

    cout<<"Line 51: namesMultiSet: ";                       //Line 51
    copy(namesMultiSet.begin(), namesMultiSet.end(),
         pScreen);                                          //Line 52
    cout<<endl;                                             //Line 53

    return 0;
}
```

输出

```
Line 9: intSet: 3 8 16 20
Line 19: intSetA: 238 156 84 59 39 36 30
Line 23: After removing 59, intSetA: 238 156 84 39 36 30
Line 31: After removing the fourth element,
         intSetA: 238 156 84 36 30
Line 35: intSetB: 238 156 84 36 30
Line 39: After removing all the elements,
         intSetB:
Line 51: namesMultiSet: Zippy Hungry Goofy Goofy Donny Donny
```

第1行语句将intSet声明为一个集合容器。第2行语句将intSetA声明为一个集合容器，该集合中的元素按降序排列。第3行语句将intGtIt声明为一个set迭代器，该迭代器可以应用在任何元素类型为int并且按降序排列的集合容器上。第4行语句将screen声明为一个ostream迭代器，用来输出任何元素类型为int的容器。

第5行到第8行中的语句将16，8，20和3插入到intSet中，第10行语句输出intSet中的元素。在程序输出中的第9行，包含了程序第9行到第11行语句的输出结果。

第12行到第18行语句将36，84，30，39，59，238和156插入到intSetA中，第20行语句输出intSetA中的元素。在程序输出中的第19行包含了程序第19行到第21行语句的输出结果。注意，intSetA中的元素按降序排列。

第22行语句从intSetA中删除了59。执行了第26行语句后，intGtIt指向了intSetA中的第一个元素。第27行语句将intGtIt加1，使其指向intSetA中的下一个元素。执行了第29行语句后，intGtIt指向intSetA中的第4个元素。第30行语句删除了IntSetA中由intGtIt指定的元素。第34行到第41行语句的含义与之类似。

第42行语句将namesMultiset声明为一个multiset类型容器。namesMultiset中元素的类型是string，按降序排列。第43行语句将iter声明为一个multiset迭代器。

第45行到第50行语句将Donny，Zippy，Goofy，Hungry，Goofy和Donny插入到namesMultiset中。第52行语句输出了namesMultiset中的元素。

21.4 容器适配器

前面几节讨论了几种容器类型。除了在常规框架下使用的容器之外，STL还提供了适合在特殊情况下使用的容器，称为容器适配器（Container Adapter）。有三种容器适配器，它们是：

- 栈
- 队列
- 优先队列

容器适配器不支持迭代器。也就是说，迭代器不能和容器适配器一同使用。下面两节将介绍两种容器适配器：栈和队列。

21.4.1 栈

第 17 章详细地讨论了栈。由于栈是一种重要的数据结构，所以 STL 提供了实现栈的类。实现栈的类的名字为 stack，包含其定义的头文件名是 stack。STL 提供的 stack 类同第 17 章中讨论的栈在实现上有细微的差别。第 17 章中栈的 pop 操作删除栈顶元素，并通过引用参数返回栈顶元素；而 STL 提供的 stack 类的 pop 操作仅删除栈顶元素。表 21.19 定义了栈容器类支持的各种操作。

表 21.19 栈对象上的操作

操作	作用
size	返回栈中元素的实际个数
empty	如果栈为空，返回 true；否则，返回 false
push(item)	将 item 的拷贝压入栈中
top	返回栈顶元素，但并不将其从栈中删除。该操作为一个带有返回值的函数
pop	将栈顶元素删除

除了操作 size，empty，push，top 和 pop，栈容器类还提供了关系运算符来比较两个栈。例如，关系运算符 == 可用来判定两个栈是否相等。

例 21.6 中的程序说明了如何使用栈容器类。

例 21.6

```
#include <iostream>
#include <stack>

using namespace std;

int main()
{
    stack<int> intStack;                            //Line 1

    intStack.push(16);                              //Line 2
    intStack.push(8);                               //Line 3
    intStack.push(20);                              //Line 4
    intStack.push(3);                               //Line 5

    cout<<"Line 6: The top element of intStack: "
        <<intStack.top()<<endl;                     //Line 6

    intStack.pop();                                 //Line 7

    cout<<"Line 8: After the pop operation, "
        <<"the top element of intStack: "
        <<intStack.top()<<endl;                     //Line 8

    cout<<"Line 9: intStack elements: ";            //Line 9
    while(!intStack.empty())                        //Line 10
    {
```

```
            cout<<intStack.top()<<" ";                  //Line 11
            intStack.pop();                             //Line 12
        }

        cout<<endl;                                     //Line 13

        return 0;
}
```

输出

```
Line 6: The top element of intStack: 3
Line 8: After the pop operation, the top element of intStack: 20
Line 9: intStack elements: 20 8 16
```

该输出内容简单易懂，详细分析留给读者作为练习。

21.4.2 队列

第17章详细地讨论了队列。由于队列是一种重要的数据结构，所以STL提供了实现队列的类。实现队列的类的名字是queue，包含queue类定义的头文件名是queue。STL提供的queue类同第17章中讨论的队列在实现上有细微的差别。表21.20定义了队列容器类支持的各种操作。

表21.20 队列对象上的操作

操作	作用
size	返回队列中元素的实际个数
empty	如果队列为空，返回true；否则，返回false
push(item)	将item的拷贝插入队列中
front	返回队列中的第一个元素，但并不将其从队列中删除。该操作为带有返回值的函数
back	返回队列中的最后一个元素，但并不将其从队列中删除。该操作为带有返回值的函数
pop	将下一个元素从队列中删除

除了操作size，empty，push，front，back和pop，队列容器类还提供了关系运算符来比较两个队列。例如，关系运算符==可用来判定两个队列是否相等。

例21.7中的程序说明了如何使用队列容器类。

例21.7

```
#include <iostream>
#include <queue>

using namespace std;

int main()
{
        queue<int>  intQueue;                           //Line 1

        intQueue.push(26);                              //Line 2
        intQueue.push(18);                              //Line 3
        intQueue.push(50);                              //Line 4
        intQueue.push(33);                              //Line 5

        cout<<"Line 6: The front element of intQueue: "
            <<intQueue.front()<<endl;                   //Line 6

        cout<<"Line 7: The last element of intQueue: "
            <<intQueue.back()<<endl;                    //Line 7
```

```
        intQueue.pop();                                   //Line 8

        cout<<"Line 9: After the pop operation, "
            <<"the front element of intQueue: "
            <<intQueue.front()<<endl;                     //Line 9

        cout<<"Line 10: intQueue elements: ";             //Line 10

        while(!intQueue.empty())                          //Line 11
        {
            cout<<intQueue.front()<<" ";                  //Line 12
            intQueue.pop();                               //Line 13
        }

        cout<<endl;                                       //Line 14

        return 0;
}
```

输出

```
Line 6: The front element of intQueue: 26
Line 7: The last element of intQueue: 33
Line 9: After the pop operation, the front element of intQueue: 18
Line 10: intQueue elements: 18 50 33
```

该输出内容简单易懂，详细分析留给读者作为练习。

21.5　容器、相关的头文件以及迭代器

前面几节讨论了各种类型的容器。每个容器都是类，实现具体容器的类定义包含在头文件中。表21.21描述了容器、头文件以及容器支持的迭代器类型。

表21.21　容器、头文件以及容器支持的迭代器类型

顺序容器	**头文件**	**迭代器类型**
vector	<vector>	随机访问
deque	<deque>	随机访问
list	<list>	双向
关联容器	**头文件**	**迭代器类型**
map	<map>	双向
multimap	<map>	双向
set	<set>	双向
multiset	<set>	双向
适配器	**头文件**	**迭代器类型**
stack	<stack>	不支持
queue	<queue>	不支持
priority queue	<queue>	不支持

21.6　算法

一个容器上可以定义多种操作。有些操作与具体的容器有关，因此将其作为该容器定义的一部分提供（也就是作为实现该容器的类的成员函数）。然而，某些操作是所有容器共有的——例如查找（find）、排序（sort）和合并（merge）。可以将这些操作作为类属算法，应用在所有容器和内建数组类型上。特定容器上的算法通过迭代器来实现。

标准算法包含在头文件algorithm中。本节将介绍其中几种算法，并说明如何在程序中使用它们。由于算法是用函数实现的，所以在本节中，术语函数和算法的含义相同。

21.6.1 STL算法分类

前面几节介绍的顺序容器上的各种操作，如clear，sort，merge等，都是作为具体容器的类成员函数实现的。所有的这些算法和其他一些算法也可通过更加类属的形式来实现，这种形式被称为类属（Generic）算法，并可被应用在各种情况中。本节将讨论其中几种类属算法。

STL中包含查看容器中元素及其移动的算法，同时还包含某些执行特定计算的算法，例如计算数字容器中所有元素的和。此外，STL还包含了集合论中某些基本操作（如集合的“并”和“交”）的算法。前面已经遇到了一些类属算法，如copy，该算法将指定范围的元素拷贝到另一个容器或屏幕上。STL中的算法可分为以下几类：

- 非修改算法
- 修改算法
- 数字算法
- 堆算法

下面几节将分别介绍这4类算法。大多数类属算法都包含在头文件algorithm中；个别算法，如数字算法，包含在头文件numeric中。

非修改算法

非修改算法只访问容器中的元素，而不修改这些元素。表21.22描述了非修改算法。

表21.22 非修改算法

adjacent_find	find_end	max
binary_search	find_first_of	max_element
count	find_if	min
count_if	for_each	min_element
equal	includes	search
equal_range	lower_bound	search_n
find	mismatch	upper_bound

修改算法

正如修改算法的名字所示，这类算法通过重新排列、删除或改变元素的值来修改容器中的元素。表21.23描述了这类算法。

表21.23 修改算法

copy	prev_permutation	rotate_copy
copy_backward	random_shuffle	set_difference
fill	remove	set_intersection
fill_n	remove_copy	set_symmetric_difference
generate	remove_copy_if	set_union
generate_n	remove_if	sort
inplace_merge	replace	stable_partition
iter_swap	replace_copy	stable_sort
merge	replace_copy_if	swap
next_permutation	replace_if	swap_ranges
nth_element	reverse	transform
partial_sort	reverse_copy	unique
partial_sort_copy	rotate	unique_copy
partition		

修改算法中改变元素顺序，而不改变其值的算法，称为变异（Mutating）算法。例如，next_permutation，partition，prev_permutation，random_shuffle，reverse，reverse_copy，rotate，rotate_copy 和 stable_partition 都是变异算法。

数字算法

数字算法用于在容器元素上执行某些数字计算。表 21.24 定义了这些算法。

表 21.24 数字算法

accumulate	inner_product
adjacent_difference	partial_sum

堆算法

有一种用于对存储在数组中的数据进行排序的特殊排序算法，称为**堆排序算法**，被用来排序存储在数组中的数据。在堆排序算法中，包含数据的数组看做是一个二叉树。因此，堆是二叉树的数组表现形式。在一个堆中，第一个元素是最大的元素，第 *i* 个位置上的元素（若存在）大于第 2*i* 和 2*i*+1 个位置上的元素（若存在）。在堆排序算法中，首先将包含数据的数组转换成一个堆，然后用--种特殊的排序算法给数组排序。表 21.25 定义了堆排序算法所需要的基本算法。

表 21.25 堆算法

make_heap	push_heap
pop_heap	sort_heap

在本节结束之前，大多数STL算法都将做简单介绍。其中的大部分算法都给出了函数原型和算法功能的简要说明，然后通过一个 C++ 程序帮助学习如何使用这些算法。STL 算法的功能非常强大。而且，这些算法以通用形式给出，除了可以使用常规的操作容器以外，用户还可以指定操作准则。例如，常规的排序准则是升序，但用户可以自己指定准则，使得容器中的元素按降序排序。因此，每个算法都用重载函数来实现。在开始介绍这些算法之前，先讨论一下函数对象，它允许用户指定操作准则。

21.6.2 函数对象

为使类属算法具有灵活性，STL常使用函数重载机制为算法提供两种形式。算法的第一种形式使用的是常规操作来实现目标。在第二种形式中，算法可以根据用户指定的准则对元素进行处理。例如，算法adjacent_find搜索容器，返回第一个两元素相等的位置。在该算法的第二种形式中，可以指定准则（如小于）来查找第一个、第二个元素小于第一个元素的元素的位置。这种准则是通过函数对象传递的。也就是说，函数对象包含了一个可以通过函数调用运算符()使用的函数。实际上，函数对象是重载了函数调用运算符 operator()的类模板。

除了允许用户创建自己的函数对象以外，STL还提供了算术函数对象、关系函数对象和逻辑函数对象，如表 21.26 所示。STL 的函数对象包含在头文件 functional 中。

表 21.26 STL 算术函数对象

函数对象名	说明
plus<Type>	plus<int> addNum; int sum = addNum(12, 35); sum 的值是 47
minus<Type>	minus<int> subtractNum; int difference = subtractNum(56, 35); difference 的值是 21
multiplies<Type>	multiplies<int> multiplyNum; int product = multiplyNum(6, 3); product 的值是 18

（续表）

函数对象名	说明
divides<Type>	divides<int> divideNum; int quotient = divideNum(16, 3); quotient 的值是 5
modulus<Type>	modulus<int> remainder; int rem = remainder(16, 7); rem 的值是 2
negate<Type>	negate<int> num; int opposite = num(-25); opposite 的值是 25

例 21.8 说明了如何使用 STL 函数对象。

例 21.8

```
//Function Objects

#include <iostream>
#include <string>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <vector>
#include <functional>

using namespace std;

int funcAdd(plus<int>, int, int);

int main()
{
    plus<int> addNum;                                   //Line 1
    int num = addNum(34,56);                            //Line 2

    cout<<"Line 3: num = "<<num<<endl;                  //Line 3

    plus<string> joinString;                            //Line 4

    string str1 = "Hello ";                             //Line 5
    string str2 = "There";                              //Line 6

    string str = joinString(str1, str2);                //Line 7

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

    cout<<"Line 9: Sum of 34 and 26 = "
        <<funcAdd(addNum, 34, 26)<<endl;                //Line 9

    int list[8] = {1, 2, 3, 4, 5, 6, 7, 8};             //Line 10

    vector<int> intList(list, list + 8);                //Line 11
    ostream_iterator<int> screenOut(cout, " ");         //Line 12

    cout<<"Line 13: intList: ";                         //Line 13
    copy(intList.begin(), intList.end(), screenOut);    //Line 14
    cout<<endl;                                         //Line 15
```

```
    //accumulate function
    int sum = accumulate(intList.begin(),
                         intList.end(), 0);           //Line 16

    cout<<"Line 17: Sum of the elements of intList = "
        <<sum<<endl;                                  //Line 17

    int product = accumulate(intList.begin(),
                             intList.end(),
                             1, multiplies<int>());   //Line 18

    cout<<"Line 19: Product of the elements of intList = "
        <<product<<endl;                              //Line 19

    return 0;
}

int funcAdd(plus<int> sum, int x, int y)
{
    return sum(x,y);
}
```

输出

```
Line 3: num = 90
Line 8: str = Hello There
Line 9: Sum of 34 and 26 = 60
Line 13: intList: 1 2 3 4 5 6 7 8
Line 17: Sum of the elements of intList = 36
Line 19: Product of the elements of intList = 40320
```

表 21.27 列举了 STL 关系函数对象。

表 21.27　STL 关系函数对象

函数对象名	说明
equal_to<Type>	如果两个参数相等，返回 true；否则，返回 false。例如： equal_to<int> compare; bool isEqual = compare(5, 5); isEqual 的值是 true
not_equal_to<Type>	如果两个参数不相等，返回 true；否则，返回 false。例如： not_equal_to<int> compare; bool isNotEqual = compare(5, 6); isNotEqual 的值是 true
greater<Type>	如果第一个参数大于第二个参数，返回 true；否则，返回 false。例如： greater<int> compare; bool isGreater = compare(8, 5); isGreater 的值是 true
greater_equal<Type>	如果第一个参数大于或者等于第二个参数，返回 true；否则，返回 false。例如： greater_equal<int> compare; bool isGreaterEqual = compare(8, 5); isGreaterEqual 的值是 true
less<Type>	如果第一个参数小于第二个参数，返回 true；否则，返回 false。例如： less<int> compare; bool isLess = compare(3, 5); isLess 的值是 true

（续表）

函数对象名	说明
less_equal<Type>	如果第一个参数小于或者等于第二个参数，返回true；否则，返回false。例如： less_equal<int> compare; bool isLessEqual = compare(8, 15); isLessEqual的值是true

正如下面所示，容器中也可以使用STL关系函数对象。STL算法adjacent_find搜索容器，返回容器中相等的两个元素的位置。该算法的第二种形式允许用户指定比较准则。例如，考虑如下向量vecList:

```
vecList = {2, 3, 4, 5, 1, 7, 8, 9};
```

假设vecList中的元素按升序排列。为查看是否存在没有排序的元素，可以这样使用算法adjacent_find:

```
intItr = adjacent_find(vecList.begin(), vecList.end(), greater<int>( ));
```

其中intItr是vector类型的迭代器。函数adjacent_find从位置vecList.begin()（vecList中的第一个元素）开始，查找第一个后面元素小于前面元素的位置。该函数返回一个指向元素5的指针，并存储在intItr中。

例21.9中的程序说明了如何使用关系函数对象。

例21.9 该例显示了STL关系函数对象的作用。

```
//STL Predicates

#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>
#include <functional>

using namespace std;

int main()
{
    equal_to<int> compare;                              //Line 1
    bool isEqual = compare(6,6);                        //Line 2

    cout<<"Line 3: isEqual = "<<isEqual<<endl;          //Line 3

    greater<string> greaterStr;                         //Line 4

    string str1 = "Hello";                              //Line 5
    string str2 = "There";                              //Line 6

    if(greaterStr(str1,str2))                           //Line 7
       cout<<"Line 8: \""<<str1<<"\" is greater "
           <<"than \""<<str2<<"\""<<endl;               //Line 8
    else                                                //Line 9
       cout<<"Line 10: \""<<str1<<"\" is not "
          <<"greater than \""<<str2<<"\""<<endl;        //Line 10

    int temp[8] = {2, 3, 4, 5, 1, 7, 8, 9};             //Line 11
    vector<int> vecList(temp, temp+8);                  //Line 12
    vector<int>::iterator intItr1, intItr2;             //Line 13
    ostream_iterator<int> screen(cout, " ");            //Line 14
```

```
    cout<<"Line 15: vecList: ";                         //Line 15
    copy(vecList.begin(), vecList.end(), screen);       //Line 16
    cout<<endl;                                         //Line 17

    intItr1 = adjacent_find(vecList.begin(),
                            vecList.end(),
                            greater<int>());            //Line 18
    intItr2 = intItr1 + 1;                              //Line 19

    cout<<"Line 20: In vecList, the first set of "
        <<"out of order elements are: "<<*intItr1
        <<" "<<*intItr2<<endl;                          //Line 20
    cout<<"Line 21: In vecList, the first out of "
        <<"order element is at position: "
        <<vecList.end() - intItr2<<endl;                //Line 21

    return 0;
}
```

输出

```
Line 3: isEqual = 1
Line 10: "Hello" is not greater than "There"
Line 15: vecList: 2 3 4 5 1 7 8 9
Line 20: In vecList, the first set of out of order elements are: 5 1
Line 21: In vecList, the first out of order element is at position: 4
```

表 21.28 列举了 STL 逻辑函数对象。

表 21.28　STL 逻辑函数对象

函数对象名	作用
logical_not<Type>	如果操作数的值为 false，则返回 true；否则，返回 false。这是一个单目函数对象
logical_and<Type>	如果两个操作数的值均为 true，则返回 true；否则，返回 false。这是一个双目函数对象
logical_or<Type>	如果至少其中一个操作数的值为 true，则返回 true；否则，返回 false。这是一个双目函数对象

谓词

谓词是一种返回布尔值的特殊类型函数对象。谓词分为两种：一元谓词和二元谓词。一元谓词只在一个参数上检查某种特性；二元谓词在两个参数上检查某种特性。谓词的典型应用领域是指定查找或排序准则。在 STL 中，对相同的值来说谓词必须总是返回相同的结果。因此，修改内部状态的函数不是谓词。

21.6.3　插入迭代器

考虑下面语句：

```
int list[5] = {1, 3, 6, 9, 12};        //Line 1
vector<int> vList;                     //Line 2
```

第 1 行语句将 list 声明并初始化为一个有 5 个元素的数组。第 2 行语句将 vList 声明为向量。由于没有指定 vList 的大小，所以也就没有为 vList 中的元素分配存储空间。现在假设要将 list 中的元素拷贝到 vList 中去。

```
copy(list, list+5, vList.begin());
```

该语句不会执行，因为没有为vList分配存储空间，而copy函数使用赋值运算符将元素从源拷贝到目的。一种解决方法是使用for循环遍历整个list，并用vList的push_back函数拷贝list中的元素。此外，还有一种更好的方法，非常适用于没有为目的分配存储空间的情况。STL提供三种可以将元素插入到目的的迭代器，称为插入迭代器。它们分别是：back_inserter，front_inserter和inserter。

- **back_inserter** 这种插入迭代器使用容器的push_back操作替代赋值运算符。该迭代器的参数是容器本身。例如，对于前面的问题，我们可以按下面方式将list中的元素拷贝到vList中去：

  ```
  copy(list, list+5, back_inserter(vList));
  ```

- **front_inserter** 这种插入迭代器使用容器的push_front操作来替代赋值运算符。该迭代器的参数是容器本身。由于vector类不支持push_front操作，所以该迭代器不能用于vector容器。
- **inserter** 这种插入迭代器使用容器的insert操作来替代赋值运算符。该迭代器有两个参数：第一个参数是容器本身，第二个参数是容器的一个迭代器，用以指定开始插入的位置。

例21.10中的程序说明了插入迭代器的使用效果。

例21.10

```
//Inserters

#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>
#include <list>

using namespace std;

int main()
{

   int temp[8] = {1, 2, 3, 4, 5, 6, 7, 8};                  //Line 1

   vector<int>  vecList1;                                   //Line 2
   vector<int>  vecList2;                                   //Line 3

   ostream_iterator<int> screenOut(cout, " ");              //Line 4

   copy(temp, temp + 8, back_inserter(vecList1));           //Line 5

   cout<<"Line 6: vecList1: ";                              //Line 6
   copy(vecList1.begin(), vecList1.end(), screenOut);       //Line 7
   cout<<endl;                                              //Line 8

   copy(vecList1.begin(), vecList1.end(),
        inserter(vecList2, vecList2.begin()));              //Line 9

   cout<<"Line 10: vecList2: ";                             //Line 10
   copy(vecList2.begin(), vecList2.end(), screenOut);       //Line 11
   cout<<endl;                                              //Line 12

   list<int> tempList;                                      //Line 13
```

```
    copy(vecList2.begin(), vecList2.end(),
         front_inserter(tempList));                   //Line 14

    cout<<"Line 15: tempList: ";                      //Line 15
    copy(tempList.begin(), tempList.end(), screenOut);  //Line 16
    cout<<endl;                                       //Line 17

    return 0;
}
```

输出

```
Line 6: vecList1: 1 2 3 4 5 6 7 8
Line 10: vecList2: 1 2 3 4 5 6 7 8
Line 15: tempList: 8 7 6 5 4 3 2 1
```

21.6.4 STL算法

下面几节将介绍STL中的大多数算法。对于每种算法，我们都将给出有关该算法用途的简要说明，并用一个程序来说明如何使用该算法。在函数原型中，参数类型指明该算法适用的容器类型。例如，如果一个参数是randomAccessIterator类型，那么该算法就仅适用于随机访问类型容器，如向量。在下面，我们将使用缩写，如用outputItr表示输出迭代器，用inputItr表示输入迭代器，用forwardItr表示前向迭代器，等等。

21.6.5 函数fill和fill_n

函数fill用来向一个容器中填充元素；函数fill_n用来填充*n*个元素。被填充的元素以参数的形式传递给这两个函数。这两个函数被定义在头文件algorithm中，其函数原型是：

```
template <class forwardItr, class Type>
void fill(forwardItr first, forwardItr last, const Type& value);

template <class forwardItr, class size, class Type>
void fill_n(forwardItr first, size n, const Type& value);
```

函数fill的前两个参数是前向迭代器，指定向容器插入元素的起始位置和结束位置；第三个参数是填充的元素。函数fill_n的第一个参数是前向迭代器，指定向容器插入元素的起始位置；第二个参数指定填充的元素个数；第三个参数指定填充的元素。例21.11中的程序说明了如何使用这些函数。

例21.11

```
//STL functions fill and fill_n


#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main()
{
      vector<int> vecList(8);                          //Line 1
      ostream_iterator<int> screen(cout, " ");         //Line 2
```

```
        fill(vecList.begin(), vecList.end(), 2);            //Line 3

        cout<<"Line 4: After filling vecList with 2's: ";   //Line 4
        copy(vecList.begin(), vecList.end(), screen);       //Line 5
        cout<<endl;                                         //Line 6

        fill_n(vecList.begin(), 3, 5);                      //Line 7

        cout<<"Line 8: After filling the first three "
            <<"elements with 5's: "<<endl<<"        ";      //Line 8
        copy(vecList.begin(), vecList.end(), screen);       //Line 9
        cout<<endl;                                         //Line 10

        return 0;
}
```

输出

```
Line 4: After filling vecList with 2's: 2 2 2 2 2 2 2 2
Line 8: After filling first three elements with 5's:
        5 5 5 2 2 2 2 2
```

第1行和第2行语句将vecList声明为一个大小为8的顺序容器，将screen声明为一个ostream类型的迭代器，并初始化为用空格符分隔的cout。第3行语句使用函数fill将2填充到vecList中。即，vecList中的8个元素都是2。前面讲过，vecList.begin( )返回指向vecList中第一个元素的迭代器，vecList.end( )返回指向vecList中最后一个元素的迭代器。第5行语句使用copy函数输出vecList中的元素。第7行语句使用函数fill_n将5填充到vecList中。fill_n的第一个参数vecList.begin( )，指定拷贝的起始位置；fill_n的第二个参数3，指定填充的元素个数；第三个参数5，指定填充的字符。因此，5被拷贝到vecList中的前三个元素中。第9行语句输出了vecList中的元素。

21.6.6 函数generate和generate_n

函数generate和generate_n用来生成元素，并将其填充到一个序列中。这两个函数被定义在头文件algorithm中，其函数原型是：

```
template <class forwardItr, class function>
void generate(forwardItr first, forwardItr last, function gen);

template <class forwardItr, class size, class function>
void generate_n(forwardItr first, size n, function gen);
```

函数generate通过连续调用函数gen( )，在first…last-1的范围内填充数字。函数generate_n在first…first+n-1的范围内，即从first位置开始，连续*n*次调用函数gen( )，填充数字。注意，gen也可以是一个函数指针。如果gen是一个函数，它必须是一个不带参数的有返回值函数。例21.12中说明了如何使用这两个函数。

例21.12

```
//STL Functions generate and generate_n

#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>
```

```
using namespace std;

int nextNum();

int main()
{
    vector<int>  vecList(8);                              //Line 1

    ostream_iterator<int> screen(cout, " ");              //Line 2

    generate(vecList.begin(), vecList.end(), nextNum);    //Line 3

    cout<<"Line 4: vecList after filling with "
        <<"numbers: ";                                    //Line 4

    copy(vecList.begin(), vecList.end(), screen);         //Line 5
    cout<<endl;                                           //Line 6

    generate_n(vecList.begin(), 3, nextNum);              //Line 7

    cout<<"Line 8: vecList, after filling the first three "
        <<"elements "<<endl
        <<"        with the next number: ";               //Line 8
    copy(vecList.begin(), vecList.end(), screen);         //Line 9
    cout<<endl;                                           //Line 10

    return 0;
}
int nextNum()
{
    static int n = 1;

    return n++;
}
```

输出

```
Line 4: vecList after filling with numbers: 1 2 3 4 5 6 7 8
Line 8: vecList, after filling the first three elements
        with the next number: 9 10 11 4 5 6 7 8
```

上面程序包含了一个带有返回值的函数nextNum，该函数中有一个初始值为1的静态变量*n*。在每次调用该函数时，都会返回*n*的当前值，并将*n*的值加1。因此，第一次调用nextNum返回1，第二次调用返回2，以此类推。

第1行和第2行语句将vecList声明为一个大小为8的顺序容器，将screen声明为一个ostream类型的迭代器，并初始化为用空格符分隔的cout。第3行语句使用函数generate，通过连续调用函数nextNum填充vecList。注意，当执行完第3行语句后，nextNum中静态变量*n*的值是9。第5行语句输出vecList中的元素。第7行语句调用函数generate_n，通过连续三次调用函数nextNum填充vecList中的前三个元素。起始位置是vecList.begin()，即vecList中的第一个元素。generate_n的第二个参数指定（见第7行）填充的元素个数是3。第9行语句输出vecList中的元素。

21.6.7　函数find，find_if，find_end和find_first_of

函数find，find_if，find_end和find_first_of用来在一个指定区间中查找元素。这些函数被定义在头文件algorithm中。函数find和find_if的函数原型是：

```
template<class inputItr, class size, class Type>
inputItr find(inputItr first, inputItr last,
              const Type& searchValue);

template<class inputItr, class unaryPredicate>
inputItr find_if(inputItr first, inputItr last, unaryPredicate op);
```

函数find在first…last-1的范围内查找元素searchValue。如果找到searchValue，返回searchValue的位置；否则，返回last。函数find_if在first…last-1的范围内查找op(rangeElement)为true的元素。如果找到满足op(rangeElement)为true的元素，函数返回该元素的位置；否则，返回last。

例21.13中的程序说明了如何使用函数find和find_if。

例21.13

```
//STL Functions find and find_if

#include <iostream>
#include <cctype>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main()
{
   char cList[10] = {'a', 'i', 'C', 'd', 'e',
                     'f', 'o', 'H', 'u', 'j'};          //Line 1

   vector<char> charList(cList, cList + 10);            //Line 2

   ostream_iterator<char> screen(cout, " ");            //Line 3

   cout<<"Line 4: Character list: ";                    //Line 4
   copy(charList.begin(), charList.end(), screen);      //Line 5
   cout<<endl;                                          //Line 6

   vector<char>::iterator position;                     //Line 7

         //find
   position = find(charList.begin(),
                   charList.end(), 'd');                //Line 8

   if(position != charList.end())                       //Line 9
      cout<<"Line 10: Element found at position = "
          <<(position - charList.begin())<<endl;        //Line 10
   else                                                 //Line 11
      cout<<"Line 12: The element is not in the list"
          <<endl;                                       //Line 12

         //find_if
   position = find_if(charList.begin(),
                      charList.end(), isupper);         //Line 13

   if(position != charList.end())                       //Line 14
      cout<<"Line 15: First uppercase letter found "
         <<"at position = "
```

```
            <<(position - charList.begin())<<endl;          //Line 15
    else                                                    //Line 16
        cout<<"Line 17: The element is not in the list"
            <<endl;                                         //Line 17

    return 0;
}
```

输出

```
Line 4: Character list: a i C d e f o H u j
Line 10: Element found at position = 3
Line 15: First uppercase letter found at position = 2
```

第1行语句创建并初始化有10个元素的字符数组cList。第2行语句创建一个字符顺序向量charList，并用字符数组cList初始化。第3行语句创建一个ostream迭代器。第5行语句输出charList（在程序输出中标有第4行的输出是程序中第4行到第6行语句的输出结果）。第7行语句将position声明为vector<char>类型迭代器。第8行语句查找'd'首次出现的位置，返回一个迭代器，并存储在position中。第9行到第12行语句输出查询结果。由于'd'是charList中第4个字符，其位置是3（见程序输出中标有第10行的输出）。第13行语句使用函数find_if查找charList中的第一个大写字母。注意，头文件cctype中的函数isupper作为第三个参数传递给find_if（见第13行）。第14行到第17行语句输出了查询结果。charList中第一个大写字母'C'，它是charList中的第三个元素，其位置是2（见程序输出中标有Line 15的输出）。

下面介绍函数find_end和find_first_of。这两个函数都各有两种形式。函数find_end的原型是：

```
template <class forwardItr1, class forwardItr2>
forwardItr1 find_end(forwardItr1 first1, forwardItr1 last1,
                     forwardItr2 first2, forwardItr2 last2);

template <class forwardItr1, class forwardItr2,
          class binaryPredicate>
forwardItr1 find_end(forwardItr1 first1, forwardItr1 last1,
                     forwardItr2 first2, forwardItr2 last2,
                     binaryPredicate op);
```

这两种形式的函数find_end都用于在first1…last1-1的范围内查找最后一次出现子区间first2…last2-1的位置。如果查找成功，函数返回first1…last1-1中匹配的位置；否则返回last1。即，函数find_end返回子区间first2…last2-1在区间first1…last1-1中的出现位置。在函数第一种形式中，按相等比较元素；在第二种形式中，比较op(elementFirstRange,elementSecondRange)必须为true。

函数find_first_of的原型是：

```
template <class forwardItr1, class forwardItr2>
forwardItr1 find_first_of(forwardItr1 first1, forwardItr1 last1,
                          forwardItr2 first2, forwardItr2, last2);

template <class forwardItr1, class forwardItr2,
          class binaryPredicate>
forwardItr1 find_first_of(forwardItr1 first1, forwardItr1 last1,
                          forwardItr2 first2, forwardItr2 last2,
                          binaryPredicate op);
```

函数的第一种形式当区间first2…last2-1中的元素在区间first1…last1-1中首次出现时，返回其在first1…last1-1中的位置。第二种形式当op（elemRange1, elemRange2）为true时，返回first2…last2-1中的元素在first1…last1-1中首次出现的位置。如果没有匹配出现，两种形式都返回last1-1。

例21.14说明了如何使用函数find_end和find_first_of。

例21.14

```
//STL Functions find_end and find_first_of

#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main()
{
   int list1[10] = {12, 34, 56, 21, 34,
                    78, 34, 56, 12, 25};                  //Line 1
   int list2[2] = {34, 56};                               //Line 2
   int list3[3] = {56, 21, 35};                           //Line 3
   int list4[5] = {33, 48, 21, 34, 73};                   //Line 4

   vector<int>::iterator location;                        //Line 5

   ostream_iterator<int> screenOut(cout, " ");            //Line 6

   cout<<"Line 7: list1: ";                               //Line 7
   copy(list1, list1 + 10, screenOut);                    //Line 8
   cout<<endl;                                            //Line 9
   cout<<"Line 10: list2: ";                              //Line 10
   copy(list2, list2 + 2, screenOut);                     //Line 11
   cout<<endl;                                            //Line 12

      //find_end
   location = find_end(list1, list1 + 10,
                       list2, list2 + 2);                 //Line 13

   if(location != list1 + 10)                             //Line 14
      cout<<"Line 15: list2 is found in list1. The "
          <<"last occurrence of list2 in \nlist1 is at "
          <<"position: "<<(location - list1)<<endl;       //Line 15
   else                                                   //Line 16
      cout<<"Line 17: list2 is not in list1"<<endl;       //Line 17

   cout<<"Line 18: list3: ";                              //Line 18
   copy(list3, list3 + 3, screenOut);                     //Line 19
   cout<<endl;                                            //Line 20

   location = find_end(list1, list1 + 10,
                       list3, list3 + 3);                 //Line 21

   if(location != list1 + 10)                             //Line 22
      cout<<"Line 23: list3 is found in list 1. The "
          <<"last occurrence of list3 in \nlist 1 is at "
          <<"position: "<<(location - list1)<<endl;       //Line 23
   else                                                   //Line 24
      cout<<"Line 25: list3 is not in list1"<<endl;       //Line 25

      //find_first_of
```

```
   cout<<"Line 26: list4: ";                              //Line 26
   copy(list4, list4 + 5, screenOut);                     //Line 27
   cout<<endl;                                            //Line 28

   location = find_first_of(list1, list1 + 10,
                            list4, list4 + 5);            //Line 29

   if(location != list1 + 10)                             //Line 30
      cout<<"Line 31: The first element "<<*location
          <<" of list4 is found in \nlist1 at "
          <<"position: "<<(location - list1)<<endl;       //Line 31
   else                                                   //Line 32
      cout<<"Line 33: No element of list4 is in list1"
          <<endl;                                         //Line 33

   return 0;
}
```

输出

```
Line 7: list1: 12 34 56 21 34 78 34 56 12 25
Line 10: list2: 34 56
Line 15: list2 is found in list1. The last occurrence of list2 in
list1 is at position: 6
Line 18: list3: 56 21 35
Line 25: list3 is not in list1
Line 26: list4: 33 48 21 34 73
Line 31: The first element 34 of list4 is found in
list1 at position: 1
```

第1行到第4行语句创建并初始化int类型数组list1，list2，list3和list4。第5行和第6行语句声明了向量和ostream的迭代器。第8行和第11行语句输出了list1和list2的值（见程序输出中标有Line 7和Line 10的输出）。第13行语句使用函数find_end在list1中查找list2最后一次出现的位置。在list1中list2最后一次出现的起始位置是6（也就是第7个元素）。第14行到第17行的语句输出了查找结果（见程序输出中标有Line 15的输出）。第19行语句输出list3。第21行语句使用函数find_end在list1中查找list3最后一次出现的位置。由于list3没有在list1中出现，所以查找失败。

第27行语句输出list4。第29行语句使用函数find_first_of来查找list4中的元素在list1中的首次出现时，其在list1中的位置。list4中在list1中首次出现的元素是34，它在list1中的位置是1，也就是list1的第二个元素。第30行到第33行语句输出了查找结果。见程序输出中标有Line 31的输出。

21.6.8 函数remove，remove_if，remove_copy和remove_copy_if

函数remove用来从一个序列中删除指定元素。函数remove_if用来从一个序列中按照某个准则删除元素。函数remove_copy用来将一个序列中的元素拷贝到另一个序列中去，但不包含第一个序列中的指定元素。类似地，函数remove_copy_if将一个序列中的元素拷贝到另一个序列中去，但不包含第一个序列中的符合某个准则的元素。这些函数被定义在头文件algorithm中。

函数remove和remove_if的原型是：

```
template <class forwardItr, class Type>
forwardItr remove(forwardItr first, forwardItr last,
                  const Type& value);

template <class forwardItr, class unaryPredicate>
forwardItr remove_if(forwardItr first, forwardItr last,
                     unaryPredicate op);
```

函数remove将指定元素从区间first···last-1中删除。要删除的元素作为第三个参数传递给该函数。函数remove_if删除区间first···last-1中所有op(element)为true的元素。这两个函数都返回forwardItr，指向新区间最后一个元素后的位置。这些函数并不改变容器的大小。实际上，只是将未被删除的元素往容器开头处移动。例如，如果原序列是{3, 7, 2, 5, 7, 9}，删除7后序列变为{3, 2, 5, 9, 7, 9}。函数返回指向（5后面）元素9的指针。

例21.15中的程序进一步说明函数返回forwardItr的重要性（见第8行，第10行，第12行和第14行）。

下面是函数remove_copy和remove_copy_if的原型：

```
template<class inputItr, class outputItr, class Type>
outputItr remove_copy(intputItr first1, inputItr last1,
                      outputItr destFirst, const Type& value);

template<class inputItr, class outputItr, class unaryPredicate>
outputItr remove_copy_if(intputItr first1, inputItr last1,
                         outputItr destFirst,
                         unaryPredicate op);
```

除了由value指定的元素，函数remove_copy将区间first1···last1-1中其余所有元素拷贝到另一个序列中，起始位置是destFirst。类似地，除了op(element)为true的元素外，函数remove_copy_if将区间first1···last1-1中其余所有元素拷贝到另一个序列中，起始位置是destFirst。这两个函数都返回outputItr，指向了被拷贝的最后一个元素后的位置。

例21.15中的程序演示了如何使用函数remove，remove_if，remove_copy和remove_copy_if。

例21.15

```
//STL Functions remove, remove_if, remove_copy, and
//              remove_copy_if

#include <iostream>
#include <cctype>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

bool lessThanEqualTo50(int num);
int main()
{
   char cList[10] = {'A', 'a', 'A', 'B', 'A',
                     'c', 'D', 'e', 'F', 'A'};              //Line 1

   vector<char> charList(cList, cList + 10);                //Line 2
   vector<char>::iterator lastElem, newLastElem;            //Line 3

   ostream_iterator<char> screen(cout, " ");                //Line 4

   cout<<"Line 6: Character list: ";                        //Line 5
   copy(charList.begin(), charList.end(), screen);          //Line 6
   cout<<endl;                                              //Line 7

      //remove
   lastElem = remove(charList.begin(),
                     charList.end(), 'A');                  //Line 8
```

```
    cout<<"Line 9: Character list after removing A: ";          //Line 9
    copy(charList.begin(), lastElem, screen);                   //Line 10
    cout<<endl;                                                 //Line 11

       //remove_if
    newLastElem = remove_if(charList.begin(),
                            lastElem, isupper);                  //Line 12
    cout<<"Line 13: Character list after removing "
        <<"the uppercase letters: "<<endl;                      //Line 13
    copy(charList.begin(), newLastElem, screen);                //Line 14
    cout<<endl<<endl;                                           //Line 15

    int list[10] = {12, 34, 56, 21, 34,
                    78, 34, 55, 12, 25};                        //Line 16

    vector<int> intList(list, list + 10);                       //Line 17
    vector<int>::iterator  endElement;                          //Line 18

    ostream_iterator<int> screenOut(cout, " ");                 //Line 19

    cout<<"Line 20: intList: ";                                 //Line 20
    copy(intList.begin(), intList.end(), screenOut);            //Line 21
    cout<<endl;                                                 //Line 22

    vector<int> temp1(10);                                      //Line 23

          //remove_copy
    endElement = remove_copy(intList.begin(), intList.end(),
                             temp1.begin(), 34);                //Line 24
    cout<<"Line 25: temp1 list after copying all the "
       <<"elements of intList except 34: "<<endl;              //Line 25
    copy(temp1.begin(), endElement, screenOut);                 //Line 26
    cout<<endl;                                                 //Line 27

    vector<int> temp2(10, 0);                                   //Line 28

       //remove_copy_if
    remove_copy_if(intList.begin(), intList.end(),
                   temp2.begin(), lessThanEqualTo50);           //Line 29

    cout<<"Line 30: temp2 after copying all the elements of "
       <<"intList except \nnumbers less than or equal to 50: ";//Line 30
    copy(temp2.begin(), temp2.end(), screenOut);                //Line 31
    cout<<endl;                                                 //Line 32

    return 0;
}

bool lessThanEqualTo50(int num)
{
       return (num <= 50);
}
```

输出

```
Line 6: Character list: A a A B A c D e F A
Line 9: Character list after removing A: a B c D e F
Line 13: Character list after removing the uppercase letters:
```

```
a c e

Line 20: intList: 12 34 56 21 34 78 34 55 12 25
Line 25: temp1 list after copying all the elements of intList except
34:
12 56 21 78 55 12 25
Line 30: temp2 after copying all the elements of intList except
numbers less than or equal to 50: 56 78 55 0 0 0 0 0 0 0
```

第2行语句创建了一个char类型向量表charList，并用第1行创建的数组cList将其初始化，第3行语句声明了两个向量迭代器lastElem和newLastElem。第4行语句声明了一个ostream迭代器screen。第6行语句输出charList的值。第8行语句使用函数remove删除了charList中所有的'A'。函数返回指向新区间最后一个元素的下一个位置，并将其存储在lastElem中。第10行语句输出新区间中的元素（注意第10行语句输出了区间charList.begin()…lastElem-1中的元素）。第12行语句使用函数remove_if删除了charList中所有大写字母，并在newLastElem中保存函数remove_if执行后返回的指针。第14行语句输出新区间中的所有元素。

第17行语句创建了一个int类型向量intList，并用第16行创建的数组list将其初始化。第21行语句输出intList中的元素。第24行语句将intList中除34以外的所有元素拷贝到temp1中。表intList没有被修改。第26行语句输出了temp1中的元素。第28行语句创建了一个int类型向量temp2，它有10个元素，并且都初始化为0。第29行语句使用remove_copy_if拷贝了intList中所有大于50的元素。第31行语句输出了temp2中的元素

21.6.9　函数replace，replace_if，replace_copy和replace_copy_if

函数replace用一个新值替换指定区间内所有的指定元素。函数replace_if用一个新值替换指定区间内所有满足特定准则的元素。这两个函数的原型是：

```
template <class forwardItr, class Type>
void replace(forwardItr first, forwardItr last,
             const Type& oldValue, const Type& newValue);

template <class forwardItr, class unaryPredicate, class Type>
void replace_if(forwardItr first, forwardItr last,
                unaryPredicate op, const Type& newValue);
```

函数replace用newValue替换区间first…last-1中所有等于oldValue的元素。函数replace_if用newValue替换区间first…last-1中所有使op(element)为true的元素。

函数replace_copy是replace和copy的组合。类似地，函数replace_copy_if是replace_if和copy的组合。下面是这两个函数的原型：

```
template <class inputItr, calss outputItr, class Type>
outputItr replace_copy(forwardItr first, forwardItr last,
                       outputItr destFirst,
                       const Type& oldValue,
                       const Type& newValue);

template <class forwardItr, calss outputItr,
          class unaryPredicate , class Type>
outputItr replace_copy_if(forwardItr first, forwardItr last,
                          outputItr destFirst,
                          unaryPredicate op,
                          const Type& newValue);
```

函数replace_copy将区间first…last-1中所有元素都拷贝到目标容器起始位置destFirst处。如果该区间中存在等于oldValue的元素，则在目标容器中就用newValue将其替换。函数replace_copy_if将区间

first⋯last-1中所有元素都拷贝到目标容器起始位置destFirst处。如果该区间中存在使op(element)为true的元素，则在目标容器中就用newValue将其替换。这两个函数都返回outpurItr（指针），指向拷贝到目标容器中最后一个元素的下一个位置。

例21.16中的程序说明了如何使用函数replace，replace_if，replace_copy和replace_copy_if。

例21.16

```
//STL Functions replace, replace_if, replace_copy, and
//              replace_copy_if

#include <iostream>
#include <cctype>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

bool lessThanEqualTo50(int num);

int main()
{
   char cList[10] = {'A', 'a', 'A', 'B', 'A',
                     'c', 'D', 'e', 'F', 'A'};              //Line 1

   vector<char>  charList(cList, cList + 10);               //Line 2

   ostream_iterator<char> screen(cout, " ");                //Line 3

   cout<<"Line 4: Character list: ";                        //Line 4
   copy(charList.begin(), charList.end(), screen);          //Line 5
   cout<<endl;                                              //Line 6

         //replace
   replace(charList.begin(), charList.end(),
                'A', 'Z');                                  //Line 7

   cout<<"Line 8: Character list after replacing "
       <<"A with Z: "<<endl;                                //Line 8
   copy(charList.begin(), charList.end(), screen);          //Line 9
   cout<<endl;                                              //Line 10
     //replace_if
   replace_if(charList.begin(), charList.end(),
              isupper, '*');                                //Line 11
   cout<<"Line 12: Character list after replacing "
       <<"the uppercase \nletters with *: ";                //Line 12
   copy(charList.begin(), charList.end(), screen);          //Line 13
   cout<<endl<<endl;                                        //Line 14

   int list[10] = {12, 34, 56, 21, 34,
                   78, 34, 55, 12, 25};                     //Line 15

   vector<int> intList(list, list + 10);                    //Line 16

   ostream_iterator<int> screenOut(cout, " ");              //Line 17

   cout<<"Line 18: intList: ";                              //Line 18
```

```
    copy(intList.begin(), intList.end(), screenOut);        //Line 19
    cout<<endl;                                              //Line 20

    vector<int> temp1(10);                                   //Line 21

             //replace_copy
    replace_copy(intList.begin(), intList.end(),
                 temp1.begin(), 34, 0);                      //Line 22

    cout<<"Line 23: temp1 list after copying intList "
        <<"and \nreplacing 34 with 0: ";                     //Line 23
    copy(temp1.begin(), temp1.end(), screenOut);             //Line 24
    cout<<endl;                                              //Line 25

    vector<int> temp2(10);                                   //Line 26

             //replace_copy_if
    replace_copy_if(intList.begin(), intList.end(),          //Line 27
             temp2.begin(), lessThanEqualTo50, 50);          //Line 28

    cout<<"Line 29: temp2 after copying intList and "
        <<"replacing any numbers \nless than or equal to 50 "
        <<"with 50: ";                                       //Line 29
    copy(temp2.begin(), temp2.end(), screenOut);             //Line 30
    cout<<endl;                                              //Line 31

    return 0;
}

bool lessThanEqualTo50(int num)
{
        return (num <= 50);
}
```

输出

```
Line 4: Character list: A a A B A c D e F A
Line 8: Character list after replacing A with Z:
Z a Z B Z c D e F Z
Line 12: Character list after replacing the uppercase
letters with *: * a * * * c * e * *

Line 18: intList: 12 34 56 21 34 78 34 55 12 25
Line 23: temp1 list after copying intList and
replacing 34 with 0: 12 0 56 21 0 78 0 55 12 25
Line 29: temp2 after copying intList and replacing any numbers
less than or equal to 50 with 50: 50 50 56 50 50 78 50 55 50 50
```

第2行语句创建了一个char类型向量表charList，并用第1行中创建的数组cList将其初始化。第3行语句声明了一个ostream迭代器screen。第6行语句输出charList中元素的值。第7行语句使用函数replace将charList中所有的'A'替换成'Z'。第9行语句输出charList中元素的值。在程序输出中标有Line 8的输出包含了第8行到第10行的输出。第11行语句使用函数replace_if将表charList中所有大写字母替换成'*'。第14行语句输出charList中元素的值。在程序输出中标有Line 12的输出包含了第12行到第14行的输出。

第16行语句创建了一个int类型向量intList，并用第15行创建的数组list将其初始化。第19行语句输出intList中的元素。第21行语句声明了一个int类型向量temp1。第22行语句拷贝了intList中所

有元素到 temp1 中，并用 0 替换了 34。表 intList 没有改动。第 24 行语句输出了 temp1 中的元素。第 26 行语句创建了一个有 10 个元素的 int 类型向量 temp2。第 28 行语句使用 replace_copy_if 拷贝了 intList 中的所有元素到 temp2 中，并将小于 50 的元素替换为 50。第 30 行语句输出了 temp2 中的元素。在程序输出中标有 Line 29 的输出包含了第 29 行到第 31 行的输出。

21.6.10　函数 swap，iter_swap 和 swap_ranges

函数 swap，iter_swap 和 swap_ranges 的作用是交换元素。这些函数被定义在头文件 algorithm 中。这些函数的原型是：

```
template <class Type>
void swap(Type& object1, Type& object2);

template <class forwardItr1, class forwardItr2>
void iter_swap(forwardItr1 first, forwardItr2 second);

template <class forwardItr1, class forwardItr2>
forwardItr2 swap_ranges(forwardItr1 first, forwardItr1 last1,
                        forwardItr2 first2);
```

函数 swap 交换了 object1 和 object2 中的值。函数 iter_swap 交换了迭代器 first 和 second 所指元素的值。

函数 swap_ranges 用起始于 first2 的连续元素交换区间 first1…last-1 中的所有元素，返回一个迭代器，指向第二个区间中的最后一个元素的下一个位置。例 21.17 中的程序说明了如何使用这些函数。

例 21.17

```
//STL functions swap, iter_swap, and swap_ranges

#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>

using namespace std;

int main()
{
   char cList[10] = {'A', 'B', 'C', 'D', 'F',
                     'G', 'H', 'I', 'J', 'K'};             //Line 1

   vector<char> charList(cList, cList + 10);             //Line 2
   vector<char>::iterator  charItr;                      //Line 3

   ostream_iterator<char> screen(cout, " ");             //Line 4

   cout<<"Line 5: Character list: ";                     //Line 5
   copy(charList.begin(), charList.end(), screen);       //Line 6
   cout<<endl;                                           //Line 7
         //swap
   swap(charList[0], charList[1]);                       //Line 8

   cout<<"Line 9: Character list after swapping the "
        <<"first and second elements: "<<endl;           //Line 9
   copy(charList.begin(), charList.end(), screen);       //Line 10
   cout<<endl;                                           //Line 11
         //iter_swap
```

```
    iter_swap(charList.begin() + 2,
              charList.begin() + 3);                          //Line 12

    cout<<"Line 13: Character list after swapping the "
        <<"third and fourth elements: "<<endl;                //Line 13
    copy(charList.begin(), charList.end(), screen);           //Line 14
    cout<<endl;                                               //Line 15

    charItr = charList.begin() + 4;                           //Line 16
    iter_swap(charItr, charItr + 1);                          //Line 17
    cout<<"Line 18: Character list after swapping the "
        <<"fifth and sixth elements: "<<endl;                 //Line 18
    copy(charList.begin(), charList.end(), screen);           //Line 19
    cout<<endl<<endl;                                         //Line 20

    int list[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};           //Line 21

    vector<int> intList(list, list + 10);                     //Line 22

    ostream_iterator<int> screenOut(cout, " ");               //Line 23

    cout<<"Line 24: intList: ";                               //Line 24
    copy(intList.begin(), intList.end(), screenOut);          //Line 25
    cout<<endl;                                               //Line 26

        //swap_ranges
    swap_ranges(intList.begin(), intList.begin() + 4,
                intList.begin() + 5);                         //Line 27

    cout<<"Line 28: intList after swapping the first "
        <<"four elements with \nfour elements starting "
        <<"at the sixth element of intList"<<endl;            //Line 28
    copy(intList.begin(), intList.end(), screenOut);          //Line 29
    cout<<endl;                                               //Line 30

    swap_ranges(list, list + 10, intList.begin());            //Line 31

    cout<<"Line 32: list and intList after swapping "
        <<"the elements with each other "<<endl;              //Line 32
    cout<<"Line 33: list: ";                                  //Line 33
    copy(list, list+10, screenOut);                           //Line 34
    cout<<endl;                                               //Line 35
    cout<<"List 36: intList: ";                               //Line 36
    copy(intList.begin(), intList.end(), screenOut);          //Line 37
    cout<<endl;                                               //Line 38

    return 0;
}
```

输出

```
Line 5: Character list: A B C D F G H I J K
Line 9: Character list after swapping the first and second elements:
B A C D F G H I J K
Line 13: Character list after swapping the third and fourth elements:
B A D C F G H I J K
Line 18: Character list after swapping the fifth and sixth elements:
B A D C G F H I J K
```

```
Line 24: intList: 1 2 3 4 5 6 7 8 9 10
Line 28: intList after swapping the first four elements with
four elements starting at the sixth element of intList
6 7 8 9 5 1 2 3 4 10
Line 32: list and intList after swapping the elements with each other
Line 33: list: 6 7 8 9 5 1 2 3 4 10
List 36: intList: 1 2 3 4 5 6 7 8 9 10
```

第 2 行语句创建了一个向量表 charList，并用第 1 行中声明的 cList 将其初始化。第 6 行语句输出了 charList 中的值。第 8 行语句交换了 charList 中第 1 个元素和第 2 个元素的位置。第 12 行语句使用函数 iter_swap 交换了 charList 的第 3 个元素和第 4 个元素的位置（注意，charList 中第 1 个元素的位置是 0）。在第 16 行语句执行后，charItr 指向 charList 中第 5 个元素。第 17 行语句使用迭代器 charItr 交换了 charList 中第 5 个元素和第 6 个元素的位置。第 19 行语句输出了 charList 中元素的值（在程序输出中标有 Line 18 的输出包含了程序中第 18 行到第 20 行的输出）。

第 22 行语句创建了一个向量表 intList，并用第 21 行声明的数组将其初始化。第 25 行语句输出了 intList 中元素的值，第 27 行语句使用函数 swap_ranges 将 intList 中前 4 个元素与第 6 个元素到第 9 个元素交换位置。第 29 行语句输出了 intList 中的元素（在程序输出中标有 Line 28 的输出包含了程序中第 28 行到第 30 行的输出）。

第 31 行语句用向量 intList 中的元素交换了数组 list 中的元素。第 34 行语句输出了数组 list 中的元素，第 37 行语句输出了 intList 中的元素。

21.6.11　函数 search，search_n，sort 和 binary_search

函数 search，search_n，sort 和 binary_search 被用来查找元素和将元素排序，这些函数定义在头文件 algorithm 中。

函数 search 的原型是：

```
template <class forwardItr1, class forwardItr2>
forwardItr1 search(forwardItr1 first1, forwardItr1 last1,
                   forwardItr2 first2, forwardItr2 last2);

template <class forwardItr1, class forwardItr2,
          class binaryPredicate>
forwardItr1 search(forwardItr1 first1, forwardItr1 last1,
                   forwardItr2 first2, forwardItr2 last2,
                   binaryPredicate op);
```

函数 search 在区间 first1…last1–1 中查找子区间 first2…last2–1。该函数的第一种形式在两个区间上进行相等比较。在第二种形式中，比较 op(elemFirstRange, elemSecondRange)必须为 true。一旦发现了匹配的子区间，函数就返回该子区间在 first1…last1–1 中出现的位置；否则，返回 last1。

函数 search_n 的原型是：

```
template <class forwardItr, class size, class Type>
forwardItr search_n(forwardItr first, forwardItr last,
                    size count, const Type& value);

template <class forwardItr, class size, class Type,
          class binaryPredicate>
forwardItr search_n(forwardItr first, forwardItr last,
                    size count, const Type& value,
                    binaryPredicate op);
```

函数search_n在区间first…last-1上查找连续count个出现的value。第一种形式返回区间first…last-1中的某个位置，从该位置开始连续count个元素的值都等于value。第二种形式返回first…last-1中的某个位置，从该位置开始连续count个元素的值都使op(elemRange, value)为true。如果没有发现匹配，这两种形式就都返回last。

函数sort的原型是：

```
template <class randomAccessItr>
void sort(randomAccessItr first, randomAccessItr last);

template <class randomAccessItr, class compare>
void sort(randomAccessItr first, randomAccessItr last,
          compare op);
```

第一种形式的sort函数按照升序顺序将区间first…last-1中的元素重新排列。第二种形式按照op指定的准则将区间first…last-1中的元素重新排列。

函数binary_search的原型是：

```
template <class forwardItr, class Type>
bool binary_search(forwardItr first, forwardItr last,
                   const Type& searchValue);
template <class forwardItr, class Type, class compare>
bool binary_search(forwardItr first, forwardItr last,
                   const Type& searchValue, compare op);
```

在第一种形式的函数binary_search中，如果在区间first…last-1中找到了searchValue，返回true；否则，返回false。在第二种形式中，使用函数对象op指定查找准则。

例21.18说明了如何使用查找和排序函数。

例21.18

```
//STL Functions search, search_n, sort, and binary_search

#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main()
{
   int intList[15] = {12, 34, 56, 34, 34,
                      78, 38, 43, 12, 25,
                      34, 56, 62, 5, 49};                  //Line 1

   vector<int> vecList(intList, intList + 15);          //Line 2
   int list[2] = {34, 56};                              //Line 3

   vector<int>::iterator location;                      //Line 4

   ostream_iterator<int> screenOut(cout, " ");          //Line 5

   cout<<"Line 6: vecList: ";                           //Line 6
   copy(vecList.begin(), vecList.end(), screenOut);     //Line 7
   cout<<endl;                                          //Line 8

   cout<<"Line 9: list: ";                              //Line 9
```

```
    copy(list, list + 2, screenOut);                      //Line 10
    cout<<endl;                                           //Line 11

       //search
    location = search(vecList.begin(), vecList.end(),
                      list, list + 2);                    //Line 12

    if(location != vecList.end())                         //Line 13
       cout<<"Line 14: list found in vecList. The "
           <<"first occurrence of \n    list in vecList "
           <<"is at position: "
           <<(location - vecList.begin())<<endl;          //Line 14
    else                                                  //Line 15
       cout<<"Line 16: list is not in vecList"<<endl;     //Line 16

       //search_n
    location = search_n(vecList.begin(), vecList.end(),
                        2, 34);                           //Line 17
    if(location != vecList.end())                         //Line 18
       cout<<"Line 19: 2 consecutive occurrences of "
           <<"34 found in \n    vecList at position: "
           <<(location - vecList.begin())<<endl;          //Line 19
    else                                                  //Line 20
       cout<<"Line 21: 2 consecutive occurrences of "
           <<"34 not in vecList"<<endl;                   //Line 21

       //sort
    sort(vecList.begin(), vecList.end());                 //Line 22

    cout<<"Line 23: vecList after sorting:"
        <<endl<<"    ";                                   //Line 23
    copy(vecList.begin(), vecList.end(), screenOut);      //Line 24
    cout<<endl;                                           //Line 25

       //binary_search
    bool found;                                           //Line 26

    found = binary_search(vecList.begin(),
                          vecList.end(), 43);             //Line 27
    if(found)                                             //Line 28
       cout<<"Line 29: 43 found in vecList "<<endl;       //Line 29
    else                                                  //Line 30
       cout<<"Line 31: 43 not in vecList"<<endl;          //Line 31

    return 0;
}
```

输出

```
Line 6: vecList: 12 34 56 34 34 78 38 43 12 25 34 56 62 5 49
Line 9: list: 34 56
Line 14: list found in vecList. The first occurrence of
    list in vecList is at position: 1
Line 19: 2 consecutive occurrences of 34 found in
    VecList at position: 3
Line 23: vecList after sorting:
    5 12 12 25 34 34 34 34 38 43 49 56 56 62 78
Line 29: 43 found in vecList
```

第2行语句创建了一个向量表vecList，并用第1行创建的数组intList将其初始化。第3行语句创建了一个有两个元素的数组list，并将其初始化。第7行语句输出vecList中的元素。第12行语句使用函数search在vecList中查找子序列（第一次出现）。第13行到第16行语句输出了查找结果。见程序输出中标有Line 14的输出。

第17行语句使用函数search_n在vecList中查找连续两个34的位置。第18行语句输出了查找的结果。第22行语句使用函数sort来对vecList排序。第24行语句输出了vecList中的元素。在程序输出中标有Line 23的输出包含了程序中第23行到第25行语句的输出。

第27行语句使用函数binary_search在vecList中进行查找。第28行到第31行语句输出了查找结果。

21.6.12 函数adjacent_find，merge和inplace_merge

算法adjacent_find用来查找符合某一准则的连续元素的首次出现。实现该算法的函数的函数原型是：

```
template <class forwardItr>
forwardItr adjacent_find(forwardItr first, forwardItr last);

template <class forwardItr, class binaryPredicate>
forwardItr adjacent_find(forwardItr first, forwardItr last,
                         binaryPredicate op);
```

第一种形式的adjacent_find使用相等准则。即，它查找相同元素的首次连续出现。在第二种形式中，算法返回区间first…last−1中某个元素的迭代器，并且op(elem, nextElem)为true。其中elem是区间first…last−1中的一个元素，而nextElem是elem的下一个元素。如果没有找到匹配的元素，则两个算法都将返回last。

算法merge用于合并有序表，结果仍是一个有序表。注意，两个表的排序准则必须相同。例如，两个表都是按照升序或是降序排列。实现合并算法的函数原型是：

```
template <class inputItr1, class inputItr2,
          class outputItr>
outputItr merge(inputItr1 first1, inputItr1 last1,
                inputItr2 first2, inputItr2 last2,
                outputItr destFirst);
template <class inputItr1, class inputItr2,
          class outputItr, class binaryPredicate>
outputItr merge(inputItr1 first1, inputItr1 last1,
                inputItr2 first2, inputItr2 last2,
                outputItr destFirst, binaryPredicte op);
```

两种形式的merge算法都是将有序区间first1…last1−1和first2…last2−1合并。合并后的目标区间起始位置由迭代器destFirst指定。第一种形式使用小于运算符<来对元素排序；第二种形式使用二元谓词op对元素排序，也就是说，op(elemRange1, elemRange2)必须为true。两种形式都返回目标区间中最后一个拷贝进去的元素的下一个位置。注意，源区间没有被修改，目标区间不被源区间覆盖。

算法inplace_merge用来合并有序连续序列。实现这种算法的函数原型是：

```
template <class biDirectionalItr>
void inplace_merge(biDirectionalItr first,
                   biDirectionalItr middle,
                   biDirectionalItr last);

template <class biDirectionalItr, class binaryPredicate>
void inplace_merge(biDirectionalItr first,
                   biDirectionalItr middle,
```

```
                  biDirectionalItr last,
                  binaryPredicate op);
```

这两种形式都将连续序列first…middle-1和middle…last-1合并排序。合并后的元素覆盖了起始于first的两个区间。第一种形式使用小于准则将两个连续序列合并。第二种形式使用二元谓词op合并序列。也就是说，对于两个区间中的元素，op(elemSeq1, elemSeq2)必须为true。例如，假设：

```
vecList = {1, 3, 5, 7, 9, 2, 4, 6, 8}
```

其中vecList是向量容器，假设vecItr是一个指向元素2的向量迭代器。在下面语句执行后：

```
inplace_merge(vecList.begin(), vecItr, vecList.end());
```

vecList中元素的顺序如下所示：

```
vecList = {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

例21.19中的程序说明上述算法是怎样工作的。

例21.19

```
//STL Functions adjacent_find, merge, and inplace_merge

#include <iostream>
#include <functional>
#include <algorithm>
#include <iterator>
#include <vector>
#include <list>

using namespace std;

int main()
{
   int list1[10] = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};          //Line 1
   int list2[10] = {0, 1, 1, 2, 3, 4, 4, 5, 6, 6};          //Line 2
   int list3[5] = {0, 2, 4, 6, 8};                          //Line 3
   int list4[5] = {1, 3, 5, 7, 9};                          //Line 4

   list<int> intList(list2, list2 + 10);                    //Line 5
   list<int>::iterator listItr;                             //Line 6

   vector<int> vecList(list1, list1 + 10);                  //Line 7
   vector<int>::iterator intItr;                            //Line 8

   ostream_iterator<int> screen(cout, " ");                 //Line 9

   cout<<"Line 10: intList : ";                             //Line 10
   copy(intList.begin(), intList.end(), screen);            //Line 11
   cout<<endl;                                              //Line 12

         //adjacent_find
   listItr = adjacent_find(intList.begin(),
                           intList.end());                  //Line 13

   if(listItr != intList.end())                             //Line 14
      cout<<"Line 15: Adjacent equal elements "
          <<"are found "<<endl
          <<"         The first set of adjacent "
```

```
            <<"equal elements: "<<*listItr<<endl;            //Line 15
    else                                                     //Line 16
        cout<<"Line 17: No adjacent equal element "
            <<"found"<<endl;                                 //Line 17

    intList.clear();                                         //Line 18

            //merge
    merge(list3, list3 + 5, list4, list4 + 5,
          back_inserter(intList));                           //Line 19

    cout<<"Line 20: intList after merging list3 and "
        <<"list4: ";                                         //Line 20
    copy(intList.begin(), intList.end(), screen);            //Line 21
    cout<<endl;                                              //Line 22

            //adjacent_find; second form
    intItr = adjacent_find(vecList.begin(), vecList.end(),
                           greater<int>());                  //Line 23

    cout<<"Line 24: Last element of first sorted "
        <<"sublist: "<<*intItr<<endl;                        //Line 24
    intItr++;                                                //Line 25
    cout<<"Line 26: First element of second sorted "
        <<"sublist: "<<*intItr<<endl;                        //List 26

    cout<<"Line 27: vecList before inplace_merge: ";         //Line 27
    copy(vecList.begin(), vecList.end(), screen);            //Line 28
    cout<<endl;                                              //Line 29
            //inplace_merge
    inplace_merge(vecList.begin(), intItr,
                  vecList.end());                            //Line 30

    cout<<"Line 31: vecList after inplace_merge: ";          //Line 31
    copy(vecList.begin(), vecList.end(), screen);            //Line 32
    cout<<endl;                                              //Line 33

    return 0;
}
```

输出

```
Line 10: intList : 0 1 1 2 3 4 4 5 6 6
Line 15: Adjacent equal elements are found
         The first set of adjacent equal elements: 1
Line 20: intList after merging list3 and list4: 0 1 2 3 4 5 6 7 8 9
Line 24: Last element of first sorted sublist: 9
Line 26: First element of second sorted sublist: 0
Line 27: vecList before inplace_merge: 1 3 5 7 9 0 2 4 6 8
Line 31: vecList after inplace_merge: 0 1 2 3 4 5 6 7 8 9
```

第5行语句创建list<int>类型的intList，并用list2将其初始化。因此，intList是一个链表。第7行语句创建了int类型的向量vecList，并用list1将其初始化。第11行语句输出intList中的元素。第13行语句使用函数adjacent_find查找首次出现连续相同元素的位置。函数返回了一个指向首次出现连续相同元素位置的指针。如果查找成功，第14行到第17行语句输出这些连续相同的元素。注意，第15行语句输出的是*listItr（listItr所指的存储空间中的内容）。

第 18 行语句通过删除 intList 中的全部元素清除了 intList。第 19 行语句使用函数 merge 合并了 list3 和 list4。在第 19 行中，函数 merge 的第 5 个参数是对 back_inserter 的调用，将合并后的表存储到 intList 中。在程序输出中标有 Line 20 的输出包含了第 20 行到第 22 行语句的输出。

注意 vecList 是{1, 3, 5, 7, 9, 0, 2, 4, 6, 8}，它包含了两个有序子序列。第 23 行语句使用第二种形式的 adjacent_find 函数来查找第二个子序列的起始位置。注意，函数 adjacent_find 的第三个参数是二元谓词 greater，所以该函数返回 vecList 中前一个元素大于后一个元素的位置。返回的位置存储在迭代器 intItr 中，并指向了元素 9。第 25 行语句将 intItr 加 1，使它指向元素 0，即第二个子序列中的第一个元素。第 30 行语句使用函数 inplace_merge 和迭代器 intItr 合并了 vecList 有序序列。注意，结果序列仍存储在 vecList 中。在程序输出中标有第 27 行的输出包含了程序中第 27 行到第 29 行语句的输出；在程序输出中标有 Line 31 的输出包含了程序中第 31 行到第 33 行的输出。

21.6.13　函数 reverse，reverse_copy，rotate 和 rotate_copy

算法 reverse 用于将指定区间中所有元素的顺序倒置。实现算法 reverse 的函数原型是：

```
template <class biDirectionalItr>
void reverse(biDirectionalItr first, biDirectionalItr last);
```

区间 first…last–1 中的元素被倒置。例如，如果 vecList = {1, 2, 5, 3, 4}，那么倒置后 vecList = {4, 3, 5, 2, 1}。

算法 reverse_copy 在将指定区间的元素拷贝到目标区间的同时，倒置元素的顺序。源区间没有被修改。实现算法 reverse_copy 的函数原型是：

```
template <class biDirectionalItr, class outputItr>
outputItr reverse_copy(biDirectionalItr first,
                       biDirectionalItr last,
                       outputItr destFirst);
```

将区间 first…last–1 中的元素以相反的顺序拷贝到起始于 destFirst 的目标区间。函数返回拷贝到目标区间的最后一个元素的下一个位置。

算法 rotate 调换指定区间中元素的顺序，其函数原型是：

```
template <class forwardItr>
void rotate(forwardItr first, forwardItr newFirst,
            forwardItr last);
```

区间 first…newFirst–1 中的元素被调换到区间的末端。newFirst 指定的元素变成区间的第一个元素。例如，假设：

```
vecList = {3, 5, 4, 0, 7, 8, 2, 5}
```

假设，迭代器 vecItr 指向 0。那么在执行语句：

```
rotate(vecList.begin(), vecItr, vecList.end());
```

之后，vecList 变为：

```
vecList = {0, 7, 8, 2, 5, 3, 5, 4}
```

算法 rotate_copy 是 rotate 和 copy 的结合。也就是，源元素按调换了顺序拷贝到目的端。源端没有被修改。实现这个算法的函数原型是：

```
template <class forwardItr, class outputItr>
outputItr rotate_copy(forwardItr first,forwardItr middle,
```

```
                         forwardItr middle, last,
                         outputItr destFirst);
```

区间 first…last-1 中的元素以调换后的顺序拷贝到起始于 destFirst 的目标区间。即区间 first…last-1 中由middle指定的元素变成了目标区间的第一个元素。函数返回拷贝到目标区间的最后一个元素的下一个位置。

算法 reverse，reverse_copy，rotate 和 rotate_copy 包含在头文件 algorithm 中。例 21.20 中的程序说明了如何使用这些算法。

例 21.20

```
//STL Functions reverse, reverse_copy,
//              rotate, and rotate_copy

#include <iostream>
#include <algorithm>
#include <iterator>
#include <list>

using namespace std;

int main()
{
   int temp[10] = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};          //Line 1

   list<int> intList(temp, temp + 10);                      //Line 2
   list<int> resultList;                                    //List 3
   list<int>::iterator listItr;                             //Line 4

   ostream_iterator<int> screen(cout, " ");                 //Line 5

   cout<<"Line 6: intList: ";                               //Line 6
   copy(intList.begin(), intList.end(), screen);            //Line 7
   cout<<endl;                                              //Line 8

      //reverse
   reverse(intList.begin(), intList.end());                 //Line 9

   cout<<"Line 10: intList after reversal: ";               //Line 10
   copy(intList.begin(), intList.end(), screen);            //Line 11
   cout<<endl;                                              //Line 12

      //reverse_copy
   reverse_copy(intList.begin(), intList.end(),
                back_inserter(resultList));                 //Line 13

   cout<<"Line 14: resultList: ";                           //Line 14
   copy(resultList.begin(), resultList.end(), screen);      //Line 15
   cout<<endl;                                              //Line 16

   listItr = intList.begin();                               //Line 17
   listItr++;                                               //Line 18
   listItr++;                                               //Line 19

   cout<<"Line 20: intList before rotating: ";              //Line 20
   copy(intList.begin(), intList.end(), screen);            //Line 21
```

```
    cout<<endl;                                              //Line 22

        //rotate
    rotate(intList.begin(), listItr, intList.end());        //Line 23

    cout<<"Line 24: intList after rotating: ";              //Line 24
    copy(intList.begin(), intList.end(), screen);            //Line 25
    cout<<endl;                                              //Line 26

    resultList.clear();                                      //Line 27

        //rotate_copy
    rotate_copy(intList.begin(), listItr, intList.end(),
                back_inserter(resultList));                  //Line 28

    cout<<"Line 29: intList after rotating and "
        <<"copying: ";                                       //Line 29
    copy(intList.begin(), intList.end(), screen);            //Line 30
    cout<<endl;                                              //Line 31

    cout<<"Line 32: resultList after rotating and "
             <<"copying: ";                                  //Line 32
    copy(resultList.begin(), resultList.end(), screen);      //Line 33
    cout<<endl;                                              //Line 34

    resultList.clear();                                      //Line 35

    rotate_copy(intList.begin(),
                find(intList.begin(), intList.end(), 6),
                intList.end(),
                back_inserter(resultList));                  //Line 36

    cout<<"Line 37: resultList after rotating and "
        <<"copying: ";                                       //Line 37
    copy(resultList.begin(), resultList.end(), screen);      //Line 38
    cout<<endl;                                              //Line 39

    return 0;
}
```

输出

```
Line 6: intList: 1 3 5 7 9 0 2 4 6 8
Line 10: intList after reversal: 8 6 4 2 0 9 7 5 3 1
Line 14: resultList: 1 3 5 7 9 0 2 4 6 8
Line 20: intList before rotating: 8 6 4 2 0 9 7 5 3 1
Line 24: intList after rotating: 4 2 0 9 7 5 3 1 8 6
Line 29: intList after rotating and copying: 4 2 0 9 7 5 3 1 8 6
Line 32: resultList after rotating and copying: 0 9 7 5 3 1 8 6 4 2
Line 37: resultList after rotating and copying: 6 4 2 0 9 7 5 3 1 8
```

21.6.14　函数 count，count_if，max_element，min_element 和 random_shuffle

算法 count 在指定区间上统计指定值出现的次数。实现该算法的函数原型是：

```
template <class inputItr, class Type>
iterator_traits<inputItr>::difference_type
        count(inputItr first, inputItr last, const Type& value);
```

函数 count 返回参数 value 指定的值在区间 first…last-1 中的出现次数。

算法 count_if 在指定区间上统计符合某个准则的指定值出现的次数。实现该算法的函数原型是：

```
template <class inputItr, class unaryPredicate>
iterator_traits<inputItr>::difference_type
    count_if(inputItr first, inputItr last, unaryPredicate op);
```

函数 count_if 返回区间 first…last-1 中使 op(elemRange)为 true 的元素个数。

算法 max 用来判断两个数值中的较大值。该算法有如下两种形式，函数原型是：

```
template <class Type>
const Type& max(const Type& aVal, const Type& bVal);

template <class Type, class compare>
const Type& max(const Type& aVal, const Type& bVal, compare comp);
```

在第一种形式中，比较准则是大于运算符；在第二种形式中，比较准则由 comp 指定。

算法 max_element 用来在指定区间中查找最大元素。该算法有如下两种形式，函数原型是：

```
template <class forwardItr>
forwardItr max_element(forwardItr first, forwardItr last);

template <class forwardItr, class compare>
forwardItr max_element(forwardItr first, forwardItr last,
                       compare comp);
```

在第一种形式中，对区间first…last-1中的元素使用的比较准则是大于运算符；在第二种形式中，比较准则由 comp 指定。两种形式都返回指向区间 first…last-1 中值最大的元素的迭代器。

算法 min 用来判断两个数值中的较小值。该算法有如下两种形式，函数原型是：

```
template <class Type>
const Type& min(const Type& aVal, const Type& bVal);

template <class Type, class compare>
const Type& min(const Type& aVal, const Type& bVal, compare comp);
```

在第一种形式中，比较准则是小于运算符；在第二种形式中，比较准则由 comp 指定。

算法 min_element 用来在指定区间中查找最小元素。该算法有如下两种形式，函数原型是：

```
template <class forwardItr>
forwardItr min_element(forwardItr first, forwardItr last);

template <class forwardItr, class compare>
forwardItr min_element(forwardItr first, forwardItr last,
                       compare comp);
```

在第一种形式中，比较准则是小于运算符；在第二种形式中，比较准则由 comp 指定。两种形式都返回指向区间 first…last-1 中值最小的元素的迭代器。

算法 random_shuffle 用来将指定区间上的元素按随机顺序排列。该算法有如下两种形式，函数原型是：

```
template <class randomAccessItr>
void random_shuffle(randomAccessItr first,
                    randomAccessItr last);

template <class randomAccessItr, class randomAccessGenerator>
void random_shuffle (randomAccessItr first,
                     randomAccessItr last,
                     randomAccessGenerator rand);
```

第一种形式使用统一分布式随机数生成器对区间first…last-1中的元素进行排列。第二种形式使用随机数生成函数对象或函数指针对区间 first…last-1 中的元素进行排列。

例 21.21 说明如何使用这些函数。

例 21.21

```
//STL Functions count, count_if, max_element,
//              min_element, random_shuffle

#include <iostream>
#include <cctype>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

void doubleNum(int num);

int main ()
{
      char cList[10] = {'Z', 'a', 'Z', 'B', 'Z',
                        'c', 'D', 'e', 'F', 'Z'};          //Line 1

      vector<char> charList(cList, cList + 10);          //Line 2

      ostream_iterator<char> screen(cout, " ");          //Line 3

      cout<<"Line 4: charList: ";                        //Line 4
      copy(charList.begin(), charList.end(), screen);    //Line 5
      cout<<endl;                                        //Line 6

            //count
      int noOfZs = count(charList.begin(), charList.end(),
                         'Z');                           //Line 7

      cout<<"Line 8: Number of Z\'s in charList = "
          <<noOfZs<<endl;                                //Line 8

            //count_if
      int noOfUpper = count_if(charList.begin(),
                        charList.end(), isupper);        //Line 9

      cout<<"Line 10: Number of uppercase letters in "
          <<"charList = "<<noOfUpper<<endl;              //Line 10

      int list[10] = {12, 34, 56, 21, 34,
                      78, 34, 55, 12, 25};               //Line 11

      ostream_iterator<int> screenOut(cout, " ");        //Line 12

      cout<<"Line 13: list: ";                           //Line 13
      copy(list, list + 10, screenOut);                  //Line 14
      cout<<endl;                                        //Line 15
            //max_element
      int *maxLoc = max_element(list, list+10);          //Line 16

      cout<<"Line 17: Largest element in list = "
```

```
        <<*maxLoc<<endl;                                //Line 17

          //min_element
    int *minLoc = min_element(list, list + 10);         //Line 18

    cout<<"Line 19: Smallest element in list = "
        <<*minLoc<<endl;                                //Line 19

          //random_shuffle
    random_shuffle(list, list + 10);                    //Line 20

    cout<<"Line 21: list after random shuffle: ";       //Line 21
    copy(list, list+10, screenOut);                     //Line 22
    cout<<endl;                                         //Line 23

    return 0;
}

void doubleNum(int num)
{
    cout<<2 * num<<" ";
}
```

输出

```
Line 4: charList: Z a Z B Z c D e F Z
Line 8: Number of Z's in charList = 4
Line 10: Number of uppercase letters in charList = 7
Line 13: list: 12 34 56 21 34 78 34 55 12 25
Line 17: Largest element in list = 78
Line 19: Smallest element in list = 12
Line 21: list after random shuffle: 34 21 12 56 34 55 12 25 78 34
```

该输出非常简单，详细分析留给读者作为练习。

21.6.15 函数 for_each 和 transform

算法for_each对指定区间中的每个元素使用指定的函数进行访问及处理，所用的函数作为参数传递给该函数。实现该算法的函数原型是：

```
template <class inputItr, class function>
function for_each(inputItr first, inputItr last, function func);
```

对区间 first…last-1 中的每个元素都使用由参数 func 指定的函数。函数 func 可以修改元素。函数 for_each 的返回值通常被忽略掉。

算法 transform 有如下两种形式，函数原型是：

```
template <class inputItr, class outputItr,
          class unaryOperation>
outputItr transform(inputItr first, iniputItr last,
                    outputItr destFirst,
                    unaryOperation op);

template <class inputItr1, class inputItr2,
          class outputItr, class binaryOperation>
outputItr transform(inputItr1 first1, iniputItr1 last,
                    inputItr2 first2,
                    outputItr destFirst,
                    binaryOperation bOp);
```

函数 transform 的第一种形式有 4 个参数。通过对区间 first…last-1 中的每个元素使用 op 操作，该函数在目标端从destFirst开始创建了一个元素序列。该函数返回最后一个拷贝到目标端的元素的下一个位置。

函数 transform 的第二种形式有 5 个参数。通过对区间 first1…last-1 中的元素和起始于 first2 区间上的相应元素使用二元操作 bOp(elemRange1, elemRange2)，并将所产生的元素序列存放到起始于 destFirst 的目标端。该函数返回最后一个拷贝到目标端的元素的下一个位置。

例 21.22 说明了如何使用这些函数。

例 21.22

```
//STL Functions for_each and transform

#include <iostream>
#include <cctype>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

void doubleNum(int& num);

int main()
{
      char cList[5] = {'a', 'b', 'c', 'd', 'e'};             //Line 1
      vector<char> charList(cList, cList + 5);               //Line 2

      ostream_iterator<char> screen(cout, " ");              //Line 3

      cout<<"Line 4: cList: ";                               //Line 4
      copy(charList.begin(), charList.end(), screen);        //Line 5
      cout<<endl;                                            //Line 6

      transform(charList.begin(), charList.end(),
                charList.begin(), toupper);                  //Line 7

      cout<<"Line 8: cList after changing all lowercase"
          <<" letters to \n        uppercase: ";             //Line 8
      copy(charList.begin(), charList.end(), screen);        //Line 9
      cout<<endl;                                            //Line 10


      int list[7] = {2, 8, 5, 1, 7, 11, 3};                  //Line 11

      ostream_iterator<int> screenOut(cout, " ");            //Line 12

      cout<<"Line 13: list: ";                               //Line 13
      copy(list, list + 7, screenOut);                       //Line 14
      cout<<endl;                                            //Line 15

      cout<<"Line 16: The effect of for_each "
          <<"function: ";                                    //Line 16
      for_each(list, list + 7, doubleNum);                   //Line 17
      cout<<endl;                                            //Line 18

      cout<<"Line 19: list after a call to for_each "
          <<"function: ";                                    //Line 19
```

```
        copy(list, list + 7, screenOut);                    //Line 20
        cout<<endl;                                          //Line 21

        return 0;
}

void doubleNum(int& num)
{
        num = 2 * num;
        cout<<num<<" ";
}
```

输出

```
Line 4: cList: a b c d e
Line 8: cList after changing all lowercase letters to
        uppercase: A B C D E
Line 13: list: 2 8 5 1 7 11 3
Line 16: The effect of for_each function: 4 16 10 2 14 22 6
Line 19: list after a call to for_each function: 4 16 10 2 14 22 6
```

第7行语句使用函数transform把charList中所有小写字母转换成相应的大写字母。在程序输出中标有Line 8的输出包含了程序中第8行到第10行语句的输出。注意，函数transform的第4个参数（第7行）是头文件cctype中的函数toupper。

第17行语句调用函数for_each使用函数doubleNum处理表中的所有元素。函数doubleNum有一个int类型的引用参数num。此外，该函数将num的值加倍，然后输出num的值。由于num是一个引用参数，所以会改变实际参数的值。在程序输出中标有Line 16的输出包含了函数doubleNum（for_each的第三个参数）中cout语句产生的输出（见第17行）。第20行语句输出list中元素的值。在程序输出中标有Line 19输出包含了程序中第19行到第20行语句的输出。

21.6.16 函数includes，set_intersection，set_union，set_difference和set_symmetric_difference

本节将介绍集合论中includes、set_intersection，set_union，set_difference和set_symmetric_difference等操作。这些算法都假定指定区间内的元素已经有序。

算法includes用来判定某一区间中的元素是否在另一个区间中出现。该算法有如下两种形式，函数原型是：

```
template <class inputItr1, calss inputItr2>
bool includes(inputItr1 first1, inputItr1 last1,
              inputItr2 first2, inputItr2 last2);

template <class inputItr1, calss inputItr2,
          class binaryPredicate>
bool includes(inputItr1 first1, inputItr1 last1,
              inputItr2 first2, inputItr2 last2,
              binaryPredicate op);
```

两种形式的函数includes都假设区间first1…last1-1和区间first2…last2-1中的元素按照同一标准排序。如果子区间first2…last2-1在区间first1…last1-1中，函数返回true。第一种形式的函数假设两个区间中的元素都按升序排列；第二种形式的函数使用操作op指定元素的顺序。

例21.23说明了函数includes是怎样工作的。

例21.23

```
//STL function includes
//This function assumes that the elements in the given ranges
//are ordered according to some sorting criterion

#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
      char setA[5] = {'A', 'B', 'C', 'D', 'E'};           //Line 1
      char setB[10] = {'A', 'B', 'C', 'D', 'E',
                       'F', 'I', 'J', 'K', 'L'};          //Line 2
      char setC[5] = {'A', 'E', 'I', 'O', 'U'};           //Line 3

      ostream_iterator<char> screen(cout, " ");           //Line 4

      cout<<"Line 5: setA: ";                             //Line 5
      copy(setA, setA + 5, screen);                       //Line 6
      cout<<endl;                                         //Line 7

      cout<<"Line 8: setB: ";                             //Line 8
      copy(setB, setB + 10, screen);                      //Line 9
      cout<<endl;                                         //Line 10

      cout<<"Line 11: setC: ";                            //Line 11
      copy(setC, setC + 5, screen);                       //Line 12
      cout<<endl;                                         //Line 13

      if(includes(setB, setB + 10, setA, setA + 5))       //Line 14
          cout<<"Line 15: setA is a subset of setB"
              <<endl;                                     //Line 15
      else                                                //Line 16
          cout<<"Line 17: setA is not a subset of setB"
              <<endl;                                     //Line 17

      if(includes(setB, setB + 10, setC, setC + 5))       //Line 18
          cout<<"Line 19: setC is a subset of setB"
              <<endl;                                     //Line 19
      else                                                //Line 20
          cout<<"Line 21: setC is not a subset of setB"
              <<endl;                                     //Line 21

      return 0;
}
```

输出

```
Line 5: setA: A B C D E
Line 8: setB: A B C D E F I J K L
Line 11: setC: A E I O U
Line 15: setA is a subset of setB
Line 21: setC is not a subset of setB
```

该程序非常简单，详细分析留给读者作为练习。

算法set_intersection用来查找在两个区间上都出现的元素。该算法有如下两种形式，函数原型是：

```
template <class inputItr1, calss inputItr2,
          class outputItr>
outputItr set_intersection(inputItr1 first1, inputItr1 last1,
                           inputItr2 first2, inputItr2 last2,
                           outputItr destFirst);

template <class inputItr1, calss inputItr2,
          class outputItr, class binaryPredicate>
outputItr set_intersection(inputItr1 first1, inputItr1 last1,
                           inputItr2 first2, inputItr2 last2,
                           outputItr destFirst,
                           binaryPredicate op);
```

两种形式的函数都使用区间first1…last1-1和区间first2…last2-1中共同出现的元素创建一个有序元素序列，并将其存储到容器中起始于destFirst的位置上。这两种形式的函数都返回一个迭代器，指向拷贝到目标区间中的最后一个元素的下一个位置。第一种形式的函数假设两个区间中的元素按升序排序；第二种形式的函数假设两个区间中的元素按op指定的操作排序。源区间中的元素没有被修改。

假设：

```
setA[5] = {2, 4, 5, 7, 8};
setB[7] = {1, 2, 3, 4, 5, 6, 7};
setC[5] = {2, 5, 8, 8, 15};
setD[6] = {1, 4, 4, 6, 7, 12};
setE[7] = {2, 3, 4, 4, 5, 6, 10};
```

则：

```
AintersectB = {2, 4, 5, 7}
AintersectC = {2, 5, 8}
DintersectE = {4, 4, 6}
```

注意，尽管8在setC中出现两次，由于8仅在setA中出现一次，所以8在AintersectC中仅出现一次。但是，由于4在setD和setE中都出现了两次，所以4在DintersectE中出现两次。

算法set_union用来查找包含在两个区间中的元素。该算法有如下两种形式，函数原型是：

```
template <class inputItr1, calss inputItr2,
          class outputItr>
outputItr set_union(inputItr1 first1, inputItr1 last1,
                    inputItr2 first2, inputItr2 last2,
                    outputItr destFirst);

template <class inputItr1, calss inputItr2,
          class outputItr, class binaryPredicate>
outputItr set_union(inputItr1 first1, inputItr1 last1,
                    inputItr2 first2, inputItr2 last2,
                    outputItr result,
                    binaryPredicate op);
```

两种形式的函数都使用区间first1…last1-1或区间first2…last2-1中出现过的元素创建一个有序元素序列，并将其存储到容器中起始于destFirst的位置上。这两种形式的函数都返回一个迭代器，指向拷贝到目标区间中最后一个元素的下一个位置。第一种形式的函数假设两个区间中的元素按升序排序；第二种形式的函数假设两个区间中的元素按op指定的操作排序。源区间中的元素没有被修改。

假设前面定义的setA，setB，setC，setD和setE不变，则：

```
AunionB = {1, 2, 3, 4, 5, 6, 7, 8}
AunionC = {2, 4, 5, 7, 8, 8, 15}
BunionD = {1, 2, 3, 4, 4, 5, 6, 7, 12}
DunionE = {1, 2, 3, 4, 4, 5, 6, 7, 10, 12}
```

注意，8在setC中出现了两次，所以它在AunionC中也出现两次。4在setD和setE中出现了两次，所以它在DunionE中也出现两次。

例21.24说明了函数set_union和set_intersection是怎样工作的。

例21.24

```
//STL set theory functions set_union and set_intersection
//These functions assume that the elements in the given ranges
//are ordered according to some sorting criterion

#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
      int setA[5] = {2, 4, 5, 7, 8};                    //Line 1
      int setB[7] = {1, 2, 3, 4, 5, 6, 7};              //Line 2
      int setC[5] = {2, 5, 8, 8, 15};                   //Line 3
      int setD[6] = {1, 4, 4, 6, 7, 12};                //Line 4
      int AunionB[10];                                  //Line 5
      int AunionC[10];                                  //Line 6
      int BunionD[15];                                  //Line 7
      int AintersectB[10];                              //Line 8
      int AintersectC[10];                              //Line 9

      int *lastElem;                                    //Line 10

      ostream_iterator<int> screen(cout, " ");          //Line 11

      cout<<"Line 12: setA = ";                         //Line 12
      copy(setA, setA + 5, screen);                     //Line 13
      cout<<endl;                                       //Line 14

      cout<<"Line 15: setB = ";                         //Line 15
      copy(setB, setB + 7, screen);                     //Line 16
      cout<<endl;                                       //Line 17

      cout<<"Line 18: setC = ";                         //Line 18
      copy(setC, setC + 5, screen);                     //Line 19
      cout<<endl;                                       //Line 20

      cout<<"Line 21: setD = ";                         //Line 21
      copy(setD, setD + 6, screen);                     //Line 22
      cout<<endl;                                       //Line 23

      lastElem = set_union(setA, setA + 5,
                           setB, setB + 7,
                           AunionB);                    //Line 24
```

```
    cout<<"Line 25: Set AunionB: ";              //Line 25
    copy(AunionB, lastElem, screen);             //Line 26
    cout<<endl;                                  //Line 27

    lastElem = set_union(setA, setA + 5,
                         setC, setC + 5,
                         AunionC);                //Line 28

    cout<<"Line 29: Set AunionC: ";              //Line 29
    copy(AunionC, lastElem, screen);             //Line 30
    cout<<endl;                                  //Line 31

    lastElem = set_union(setB, setB + 7,
                         setD, setD + 6,
                         BunionD);                //Line 32

    cout<<"Line 33: Set BunionD: ";              //Line 33
    copy(BunionD, lastElem, screen);             //Line 34
    cout<<endl;                                  //Line 35

    lastElem = set_intersection(setA, setA + 5,
                                setB, setB + 7,
                                AintersectB);     //Line 36

    cout<<"Line 37: Set AintersectB: ";          //Line 37
    copy(AintersectB, lastElem, screen);         //Line 38
    cout<<endl;                                  //Line 39

    lastElem = set_intersection(setA, setA + 5,
                                setC, setC + 5,
                                AintersectC);     //Line 40

    cout<<"Line 41: Set AintersectC: ";          //Line 41
    copy(AintersectC, lastElem, screen);         //Line 42
    cout<<endl;                                  //Line 43

    return 0;
}
```

输出

```
Line 12: setA = 2 4 5 7 8
Line 15: setB = 1 2 3 4 5 6 7
Line 18: setC = 2 5 8 8 15
Line 21: setD = 1 4 4 6 7 12
Line 25: Set AunionB: 1 2 3 4 5 6 7 8
Line 29: Set AunionC: 2 4 5 7 8 8 15
Line 33: Set BunionD: 1 2 3 4 4 5 6 7 12
Line 37: Set AintersectB: 2 4 5 7
Line 41: Set AintersectC: 2 5 8
```

该程序非常简单，详细分析留给读者作为练习。

算法set_difference用来查找只在其中一个区间出现，同时不在另一个区间出现的元素。该算法有如下两种形式，函数原型是：

```
template <class inputItr1, calss inputItr2,
          class outputItr>
```

```
outputItr set_difference(inputItr1 first1, inputItr1 last1,
                         inputItr2 first2, inputItr2 last2,
                         outputItr destFirst);
template <class inputItr1, calss inputItr2,
          class outputItr, class binaryPredicate>
outputItr set_difference(inputItr1 first1, inputItr1 last1,
                         inputItr2 first2, inputItr2 last2,
                         outputItr destFirst,
                         binaryPredicate op);
```

两种形式的函数都使用只在有序区间first1…last1-1中出现，而不在有序区间first2…last2-1中出现的元素创建一个有序序列，并将其存储到容器中起始于destFirst的位置上。这两种形式的函数都返回一个迭代器，指向拷贝到目标区间中最后一个元素的下一个位置。第一种形式的函数假设两个区间中的元素按升序排序；第二种形式的函数假设两个区间中的元素按op指定的操作排序。源区间中的元素没有被修改。

假设：

```
setA[5] = {2, 4, 5, 7, 8};
setC[5] = {1, 5, 6, 8, 15};
setD[5] = {2, 5, 5, 6, 9};
setE[5] = {1, 5, 7, 9, 12};
```

则：

```
AdifferenceC = {2, 4, 7}
DdifferenceE = {2, 5, 6}
```

注意，由于5在setD中出现了两次，而在setE中仅出现了一次，所以5在DdifferenceE中只出现一次。

算法set_symmetric_difference有如下两种形式，函数原型是：

```
template <class inputItr1, calss inputItr2,
          class outputItr>

outputItr set_symmetric_difference(inputItr1 first1,
                                   inputItr1 last1,
                                   inputItr2 first2,
                                   inputItr2 last2,
                                   outputItr destFirst);
template <class inputItr1, calss inputItr2,
          class outputItr, class binaryPredicate>
outputItr set_symmetric_difference(inputItr1 first1,
                                   inputItr1 last1,
                                   inputItr2 first2,
                                   inputItr2 last2,
                                   outputItr destFirst,
                                   binaryPredicate op);
```

两种形式都使用只在有序区间first1…last1-1中出现，但不在有序区间first2…last2-1中出现，或只在有序区间first2…last2-1中出现，但不在有序区间first1…last1-1中出现的元素创建一个有序序列，并将其存储到容器中起始于destFirst的位置上。这两种形式的函数都返回一个迭代器，指向拷贝到目标区间中最后一个元素的下一个位置。第一种形式的函数假设两个区间中的元素按升序排序；第二种形式的函数假设两个区间中的元素按op指定的操作排序。源区间中的元素没有被修改。由set_symmetric _difference创建的元素序列包含了在range1_union_range2中的元素，但不包含在range1_intersection _range2中的元素。

假设:

```
setB[7] = {3, 4, 5, 6, 7, 8, 10};
setC[5] = {1, 5, 6, 8, 15};
setD[5] = {2, 5, 5, 6, 9};
```

注意，BdifferenceC = {3, 4, 7, 10}，CdifferenceB = {1, 15}，因此:

```
BsymDiffC = {1, 3, 4, 7, 10, 15}
```

现在 DdifferenceC = {2, 5, 9}，CdifferenceD = {1, 8, 15}，因此:

```
DsymDiffC = {1, 2, 5, 8, 9, 15}
```

例 21.25 说明了函数 set_difference 和 set_symmetric_difference 是怎样工作的。

例 21.25

```
//STL set theory functions set_difference and
//                         set_symmetric_difference
//These functions assume that the elements in the given ranges
//are ordered according to some sorting criterion

#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   int setA[5] = {2, 4, 5, 7, 8};                              //Line 1
   int setB[7] = {3, 4, 5, 6, 7, 8, 10};                       //Line 2
   int setC[5] = {1, 5, 6, 8, 15};                             //Line 3

   int AdifferenceC[5];                                        //Line 4
   int BsymDiffC[10];                                          //Line 5

   int *lastElem;                                              //Line 6

   ostream_iterator<int> screen(cout, " ");                    //Line 7

   cout<<"Line 8: setA = ";                                    //Line 8
   copy(setA, setA + 5, screen);                               //Line 9
   cout<<endl;                                                 //Line 10

   cout<<"Line 11: setB = ";                                   //Line 11
   copy(setB, setB + 7, screen);                               //Line 12
   cout<<endl;                                                 //Line 13

   cout<<"Line 14: setC = ";                                   //Line 14
   copy(setC, setC + 5, screen);                               //Line 15
   cout<<endl;                                                 //Line 16

   lastElem = set_difference(setA, setA + 5,
                             setC, setC + 5,
                             AdifferenceC);                    //Line 17

   cout<<"Line 18: AdifferenceC: ";                            //Line 18
   copy(AdifferenceC, lastElem, screen);                       //Line 19
```

```
    cout<<endl;                                         //Line 20

    lastElem = set_symmetric_difference(setB, setB + 7,
                                        setC, setC + 5,
                                        BsymDiffC);     //Line 21

    cout<<"Line 22: BsymDiffC: ";                       //Line 22
    copy(BsymDiffC, lastElem, screen);                  //Line 23
    cout<<endl;                                         //Line 24

    return 0;
}
```

输出

```
Line 8: setA = 2 4 5 7 8
Line 11: setB = 3 4 5 6 7 8 10
Line 14: setC = 1 5 6 8 15
Line 18: AdifferenceC: 2 4 7
Line 22: BsymDiffC: 1 3 4 7 10 15
```

该程序非常简单，详细分析留给读者作为练习。

21.6.17 函数accumulate，adjacent_difference，inner_product和partial_sum

算法accumulate，adjacent_difference，inner_product和partial_sum都是数字函数，因此只能操作数字类型数据。这些函数都各有两种形式。第一种形式使用常用操作来操作数据。例如，算法accumulate通常计算指定区间中所有元素的和，在第二种形式中，可以指定对区间中的元素所进行的操作。例如，在使用accumulate操作时可以不去计算指定区间中所有元素的和，而是计算指定区间中所有元素的乘积。下面给出这类算法的函数原型，并做简要说明。这些算法包含在头文件numeric中。

```
template <class inputItr, class Type>
Type accumulate( inputItr first, inputItr last, Type init);

template <class inputItr, class Type, calss binaryOperation>
Type accumulate( inputItr first, inputItr last,
                 Type init, binaryOperation op);
```

第一种形式算法accumulate将区间first…last-1中所有元素同由参数init指定的初始值相加。例如，如果init的值是0，那么该算法返回的就是所有元素的和。在第二种形式中，可以指定对区间中元素所进行的操作，如乘法。假设init的值是1，并且指定的操作是乘法，该算法返回的就是这个区间上所有元素的乘积。

下面，描述算法adjacent_difference，其函数原型是：

```
template <class inputItr, class outputItr>
outputItr adjacent_difference(inputItr first, inputItr last,
                              outputItr destFirst);

template <class inputItr, class outputItr,
          class binaryOperation>
outputItr adjacent_difference(inputItr first, inputItr last,
                              outputItr destFirst,
                              binaryOperation op);
```

第一种形式adjacent_difference函数创建了一个元素序列，其第一个元素与区间first…last-1中第一个元素相同，其他元素是当前元素与前一个元素之差。例如，如果区间中的元素是：

```
{2,5,6,8,3,7}
```

那么函数 adjacent_difference 所创建的序列是：

```
{2,3,1,2,-5,4}
```

其第一个元素与原序列中第一个元素相同。第二个元素等于原序列中第二个元素与第一个元素之差。类似地，第三个元素等于原序列中第三个元素与第二个元素之差，以此类推。

第二种形式 adjacent_difference 函数，对区间中的每个元素使用操作 op。结果序列拷贝到起始于 destFirst 的目标区间上。例如，如果序列是{2, 5, 6, 8, 3, 7}，操作是乘法，那么结果序列就是{2, 10, 30, 48, 24, 21}。

这两种形式的函数都返回拷贝到目标区间上的最后一个元素的下一个位置。

算法 inner_product 操作两个区间中的元素。其函数原型是：

```
template <class inputItr1, class inputItr2, class Type>
Type inner_product(inputItr1 first1, inputItr1 last,
                   inputItr2 first2, Type init);

template <class inputItr1, class inputItr2, class Type,
          class binaryOperation1, class binaryOperation2>

Type inner_product(inputItr1 first1, inputItr1 last,
                   inputItr2 first2, Type init,
                   binaryOperation1 op1, binaryOperation2 op2);
```

第一种形式的 inner_product 函数用区间 first1…last-1 中的元素和以 first2 开始的区间中相对应的元素相乘，结果加到由参数 init 指定的变量中。具体地说，假设 elem1 依次表示第一个区间中的元素，elem2 依次表示以 first2 起始的第二个区间中的元素。第一种形式的 inner_product 函数在所有相对应元素上计算：

```
init = init + elem1 * elem2
```

例如，假设两个区间是{2, 4, 7, 8}和{1, 4, 6, 9}，并且 init 是 0，函数计算并返回：

```
0 + 2*1 + 4*4 + 7*6 + 8*9 = 132
```

在第二种形式的 inner_product 函数中，用 op1 指定的操作替换默认的加法操作，用 op2 指定的操作替换默认的乘法操作。第二种形式的 inner_product 函数实际计算是：

```
init = init op1 (elem1 op2 elem2);
```

算法 partial_sum 有两种形式，函数原型如下所示：

```
template <class inputItr, class outputItr>
outputItr partial_sum(inputItr first, inputItr last,
                      outputItr destFirst);

template <class inputItr, class outputItr,
          class binaryOperation>
outputItr partial_sum(inputItr first, inputItr last,
                      outputItr destFirst, binaryOperation op);
```

第一种形式的 partial_sum 函数创建了一个元素序列，其中的每个元素都是区间 first…last-1 中在它前面所有元素（包括该元素本身）之和。例如，新序列中的第一个元素与区间 first…last-1 中第一个元素相同，第二个元素是区间 first…last-1 中前两个元素之和，第三个元素是区间 first…last-1 中前三个元素之和，以此类推。例如，对于序列：

{1,3,4,6}

函数 partial_sum 生成如下序列：

{1,4,8,14}

在第二种形式的 partial_sum 函数中，用 op 指定的操作替换默认的加法操作。例如，如果序列是：

{1,3,4,6}

并且操作是乘法，则 partial_sum 生成如下序列：

{1,3,12,72}

所创建的序列拷贝到起始于 destFirst 的目标区间中，并返回所拷贝的最后一个元素的下一个位置。例 21.26 说明了本节介绍的函数是怎样工作的。

例 21.26

```
//Numeric algorithms: accumulate, adjacent_difference
//                    inner_product, and partial_sum

#include <iostream>
#include <algorithm>
#include <numeric>
#include <iterator>
#include <vector>
#include <functional>

using namespace std;

void print(vector<int> vList);

int main()
{
      int list[8] = {1, 2, 3, 4, 5, 6, 7, 8};          //Line 1

      vector<int> vecList(list, list + 8);             //Line 2
      vector<int> newVList(8);                         //Line 3


      cout<<"Line 4: vecList: ";                       //Line 4
      print(vecList);                                  //Line 5

            //accumulate function
      int sum = accumulate(vecList.begin(),
                           vecList.end(), 0);          //Line 6

      cout<<"Line 7: Sum of the elements of vecList = "
          <<sum<<endl;                                 //Line 7

      int product = accumulate(vecList.begin(),
                               vecList.end(),
                               1, multiplies<int>());  //Line 8

      cout<<"Line 9: Product of the elements of "
          <<"vecList = "<<product<<endl;               //Line 9

            //adjacent_difference function
      adjacent_difference(vecList.begin(),
```

```
                        vecList.end(),
                        newVList.begin());              //Line 10

    cout<<"Line 11: newVList: ";                        //Line 11
    print(newVList);                                    //Line 12

    adjacent_difference(vecList.begin(), vecList.end(),
                        newVList.begin(),
                        multiplies<int>());             //Line 13


    cout<<"Line 14: newVList: ";                        //Line 14
    print(newVList);                                    //Line 15

        //inner_product function
    sum = inner_product(vecList.begin(), vecList.end(),
                        newVList.begin(), 0);           //Line 16

    cout<<"Line 17: Inner product of vecList "
         <<"and newVList: "<<sum<<endl;                 //Line 17

    sum = inner_product(vecList.begin(), vecList.end(),
                        newVList.begin(), 0,
                        plus<int>(), minus<int>());     //Line 18

    cout<<"Line 19: Inner product of vecList and "
        <<"newVList, using - for *: "<<sum<<endl;       //Line 19

        //partial_sum function
    partial_sum(vecList.begin(), vecList.end(),
                newVList.begin());                      //Line 20

    cout<<"Line 21: newVList with partial sum: ";       //Line 21
    print(newVList);                                    //Line 22

        //partial_sum: the default + is replaced by *
    partial_sum(vecList.begin(), vecList.end(),
              newVList.begin(), multiplies<int>());     //Line 23

    cout<<"Line 24: newVList with partial "
         <<"multiplication: "<<endl<<"          ";      //Line 24
    print(newVList);                                    //Line 25

    return 0;
}

void print(vector<int> vList)
{
    ostream_iterator<int> screenOut(cout, " ");         //Line 26

    copy(vList.begin(), vList.end(), screenOut);        //Line 27
    cout<<endl;                                         //Line 28
}
```

输出

```
Line 4: vecList: 1 2 3 4 5 6 7 8
Line 7: Sum of the elements of vecList = 36
Line 9: Product of the elements of vecList = 40320
```

```
Line 11: newVList: 1 1 1 1 1 1 1 1
Line 14: newVList: 1 2 6 12 20 30 42 56
Line 17: Inner product of vecList and newVList: 1093
Line 19: Inner product of vecList and newVList, using - for *: -133
Line 21: newVList with partial sum: 1 3 6 10 15 21 28 36
Line 24: newVList with partial multiplication:
         1 2 6 24 120 720 5040 40320
```

该程序非常简单，详细分析留给读者作为练习。

21.7 小结

1. STL的三个主要组成部分是：容器、迭代器和算法。
2. STL容器是类。
3. 迭代器用来遍历容器中的每个元素。
4. 算法用来操作容器中的元素。
5. 容器的主要种类有顺序容器、关联容器和容器适配器。
6. 三种预定义的顺序容器是向量、队列和表。
7. 向量容器使用动态数组存储和管理数据。
8. 由于数组是一种随机访问数据结构，所以向量中的元素可以随机访问。
9. 实现向量容器的类的名字是vector。
10. 在向量容器中插入元素使用的操作是insert和push_back。
11. 从向量容器中删除元素使用的操作是pop_back，erase和clear。
12. 可以使用typedef iterator声明一个向量容器的迭代器，其为vector类的public成员。
13. 在所有容器上都共有的成员函数是默认构造函数，带参数构造函数，拷贝构造函数，析构函数，empty，size，max_size，swap，begin，end，rbegin，rend，insert，erase，clear和关系运算符函数。
14. 函数begin返回一个指向容器中第一个元素的迭代器。
15. 函数end返回一个指向容器中最后一个元素的迭代器。
16. 除了列在上面第13条中的成员函数之外，所有顺序容器还具备的成员函数是insert，push_back，pop_back，erase，clear和resize。
17. 算法copy用来将指定区间中的元素拷贝到另外一个区间上。
18. 如果在函数copy中使用ostream迭代器，可以输出容器中的元素。
19. 当创建了一个ostream类型的迭代器后，也即指定了该迭代器输出的元素类型。
20. 双端队列容器使用动态数组实现，可以在该动态数组的两端插入元素。
21. 双端队列的两端都可以扩展。
22. 实现双端队列的类是deque。
23. 除了所有容器都有的操作外，还可以在双端队列中使用的操作是assign，push_front，pop_front，at，数组下标运算符[]，front和back。
24. 表容器使用双向链表实现。因此，表中的每个元素（第一个元素和最后一个元素除外）都含有指向其直接前驱和直接后继的指针。
25. 实现表的类是list。
26. 除了所有容器都共有的操作外，还可以在表容器中使用的操作是assign，push_front，pop_front，front，back，remove，remove_if，unique，splice，sort，merge和reverse。
27. 5种迭代器是：输入迭代器、输出迭代器、前向迭代器、双向迭代器和随机访问迭代器。

28. 输入迭代器用来从一个输入流输入数据。
29. 输出迭代器用来把数据输出到一个输出流。
30. 前向迭代器可以引用同一集合中的同一元素，并且可以处理同一元素多次。
31. 双向迭代器既是前向迭代器，又可以后向遍历元素。
32. 双向迭代器可以在 list，set，multiset，map 和 multimap 类型的容器上使用。
33. 随机访问迭代器是可以随机访问容器中元素的双向迭代器。
34. 随机访问迭代器可以在 vector，dequeue，string 和数组类型的容器上使用。
35. 关联容器中的元素自动按照某种排序准则存储。默认的排序准则是关系运算符小于号（<）。
36. 在STL中预定义的关联容器是集合(set)、多重集合(multiset)、映射(map)和多重映射(multimap)。
37. set 类型的容器中不允许有重复元素。
38. multiset 类型的容器中允许有重复元素。
39. 定义集合容器的类是 set。
40. 定义多重集合容器的类是 multiset。
41. 包含集合类和多重集合类容器定义及其操作的头文件是 set。
42. 操作 insert，erase 和 clear 可用来在集合中插入和删除元素。
43. 大多数类属算法包含在头文件 algorithm 中。
44. STL 算法主要种类有：非修改型、修改型、数字型和堆。
45. 非修改算法不修改容器中的元素。
46. 修改算法通过重排列、删除以及改变元素的值来修改容器中的元素。
47. 只改变元素的顺序，而不改变元素值的修改算法，又称为变异算法。
48. 数字算法用来对容器中元素执行数字计算。
49. 函数对象是一种类模板，它重载了函数调用运算符 operator( )。
50. 预定义的算术函数对象是 plus，minus，multiplies，divides，modulus 和 negate。
51. 预定义的关系函数对象是 equal_to，not_equal_to，greater，greater_equal，less 和 less_equal。
52. 预定义的逻辑函数对象是 logical_not，logical_and 和 logical_or。
53. 谓词是特殊类型的函数对象，它返回布尔类型的值。
54. 一元谓词在单个参数上检查某种特性；二元谓词在两个参数上检查某种特性。
55. 谓词通常用来指定查找或排序准则。
56. 在 STL 中，谓词总是为相同的值返回相同的结果。
57. 修改内部状态的函数不是谓词。
58. STL 提供三种插入迭代器—— back_inserter，front_inserter 和 inserter 用来在目标中插入元素。
59. back_inserter 使用容器的 push_back 操作替换赋值运算符。
60. front_inserter 使用容器的 push_front 操作替换赋值运算符。
61. 由于向量类不支持 push_front 操作，所以上面迭代器不能用于向量容器。
62. inserter 迭代器使用容器的 insert 操作替换赋值运算符。
63. 函数 fill 用来填充一个容器，而函数 fill_n 用来填充 *n* 个元素。
64. 函数 generate 和 generate_n 用来生成元素并填充一个序列。
65. 函数 find，find_if，find_end 和 find_first_of 用来在指定区间中查找元素。
66. 函数 remove 用来从一个序列中删除某个特定元素。
67. 函数 remove_if 用来从一个序列中按照某个准则删除元素。
68. 函数 remove_copy 将一个序列中的元素拷贝到另一个序列中，但不包含前面序列中的指定元素。
69. 函数remove_copy_if按照某个准则，将一个序列中的元素拷贝到另一个序列中，但不包含前一个序列中的指定元素。

70. 函数 swap，iter_swap 和 swap_ranges 用来交换元素。
71. 函数 search，search_n，sort 和 binary_search 用来查找元素。
72. 函数 adjacent_find 用来查找满足某个特定准则的第一次连续出现的元素的位置。
73. 算法 merge 合并两个有序表。
74. 算法 inplace_merge 用来合并两个有序的连续序列。
75. 算法 reverse 倒置指定区间中元素的顺序。
76. 算法 reverse_copy 倒置指定区间中的元素，并拷贝到目标区间中。源区间中的元素没有被修改。
77. 算法 rotate 用于调换指定区间中的元素。
78. 算法 rotate_copy 按调换后的顺序把指定区间中的元素拷贝到目标区间中。
79. 算法 count 在指定区间中统计指定值出现的次数。
80. 算法 count_if 在指定区间中统计满足特定准则的值出现的次数。
81. 算法 max 用来判定两个值中较大的一个。
82. 算法 max_element 用来判定指定区间中最大的元素。
83. 算法 min 用来判定两个值中较小的一个。
84. 算法 min_element 用来判定指定区间中最小的元素。
85. 算法 random_shuffle 用来将指定区间中的元素按随机顺序排列。
86. 算法 for_each 用来访问并用一个函数处理指定区间中的每个元素，处理函数作为参数传递给该函数。
87. 算法 transform 通过对指定区间中的每个元素使用某个特定操作，来创建一个元素序列。
88. 算法 includes 用来判定一个区间中的元素是否在另一个区间中出现。
89. 算法 set_intersection 用来查找两个区间中的共同元素。
90. 算法 set_union 用来查找包含在两个区间中的元素。
91. 算法 set_difference 用来查找在一个区间中出现但不在另一个区间中出现的元素。
92. 算法 set_symmetric_difference 用于查找在第一个区间中出现，但不在第二个区间中出现，或在第二个区间中出现，但不在第一个区间中出现的元素。
93. 算法 accumulate，adjacent_difference，inner_product 和 partial_sum 是数字函数，用于操作数字数据。

21.8 练习

1. STL 的三个主要组成部分是什么？
2. STL 容器和 STL 迭代器之间的区别是什么？
3. STL 容器和 STL 算法之间的区别是什么？
4. set 和 multiset 之间的区别是什么？
5. 什么是 STL 函数对象？
6. 假设 vecList 是一个向量容器，并且：

```
vecList = {12, 16, 8, 23, 40, 6, 18, 9, 75}
```

给出下面语句执行后的 vecList：

```
copy(vecList.begin() + 2, vecList.end(), vecList.begin());
```

7. 假设 vecList 是一个向量容器，并且：

```
vecList = {12, 16, 8, 23, 40, 6, 18, 9, 75}
```

给出下面语句执行后的 vecList：

```
copy(vecList.rbegin() + 3, vecList.rend(), vecList.rbegin());
```

8. 假设intList是表容器，并且：

```
intList = {3, 23, 23, 43, 56, 11, 11, 23, 25}
```

给出下面语句执行后的intList：

```
intList.unique();
```

9. 假设intList1和intList2是表容器，并且：

```
intList1 = {3, 58, 78, 85, 6, 15, 93, 98, 25}
intList2 = {5, 24, 16, 11, 60, 9}
```

给出下面语句执行后的intList1：

```
intList1.splice(intList1.begin(), intList2);
```

10. 假设charList是向量容器，并且：

```
charList = {a, A, B, b, c, d, A, e, f, K}
```

再假设：

```
lastElem = remove_if(charList.begin(), charList.end(), islower);
ostream_iterator<char> screen(cout, " ");
```

这里lastElem是char类型的向量容器迭代器。下面语句的输出是什么？

```
copy(charList.begin(), lastElem, screen);
```

11. 假设intList是向量容器，并且：

```
intList = {18, 24, 24, 5, 11, 56, 27, 24, 2, 24}
```

再假设：

```
vector<int>::iterator lastElem;
ostream_iterator<int> screen(cout, " ");
vector<int> otherList(10);

lastElem = remove_copy(intList.begin(), intList.end(), otherList.begin(), 24);
```

下面语句的输出是什么？

```
copy(otherList.begin(), lastElem, screenout);
```

12. 假设intList是向量容器，并且：

```
intList = {2, 4, 6, 8, 10, 12, 14, 16}
```

下面语句执行后，result的值是什么？

```
result = accumulate(intList.begin(), intList.end(), 0);
```

13. 假设intList是向量容器，并且：

```
intList = {2, 4, 6, 8, 10, 12, 14, 16}
```

下面语句执行后，result的值是什么？

```
result = accumulate(intList.begin(), intList.end(), 0, multiplies<int>());
```

14. 假设setA，setB，setC和setD的定义如下所示：

```
int setA[] = {3, 4, 5, 8, 9, 12, 14};
int setB[] = {2, 3, 4, 5, 6, 7, 8};
```

```
int setC[] = {2, 5, 5, 9};
int setD[] = {4, 4, 4, 6, 7, 12};
```

再假设有如下声明：

```
int AunionB[10];
int AunionC[9];
int BunionD[10];
int AintersectB[4];
int AintersectC[2];
```

在下面语句执行后，AunionB，AunionC，BunionD，AintersectB 和 AintersectC 中的值是什么？

```
set_union(setA, setA +7, setB, setB + 7, AunionB);
set_union(setA, setA +7, setC, setC + 4, AunionC);
set_union(setB, setB +7, setD, setD + 6, BunionD);
set_intersection(setA, setA +7, setB, setB + 7, AintersectB);
set_intersection(setA, setA +7, setC, setC + 4, AintersectC);
```

21.9 编程练习

1. 重写第 16 章中的“程序范例：影碟店”程序，使用 STL 的 list 类处理影碟表。
2. 重写第 16 章中的编程练习 5，使用 STL 的 list 类来处理客户租用的影碟表及影碟店会员表。
3. 重写第 16 章中的编程练习 6，使用 STL 的 list 类处理影碟表、客户租用的影碟表以及影碟店会员表。
4. 重写第 17 章中的“后缀表达式计算器”程序，使用 STL 的 stack 类来计算后缀表达式值。
5. 重写第 17 章中的编程练习 9，使用 STL 的 stack 类将中缀表达式转换成后缀表达式。
6. 重写第 17 章中的“模拟”程序，使用 STL 的 queue 类来维护等待客户表。

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 运算符优先级

（由高到低）

下表给出了C++中运算符的优先级（由高到低）和结合律。

运算符	结合律
::（双目作用域运算符）	由左向右
::（单目作用域运算符）	由右向左
()	由左向右
[] -> .	由左向右
++ --（作为后缀运算符）	由右向左
typeid dynamic_cast	由右向左
static_cast const_cast	由右向左
reinterpret_cast	由右向左
++ --（作为前缀运算符） ! +（单目） -（单目）	由右向左
~ &（取地址） *（递引用）	由右向左
new delete sizeof	由右向左
->* -- .*	由左向右
* / %	由左向右
+ -	由左向右
<< >>	由左向右
< <= > >=	由左向右
== !=	由左向右
&	由左向右
^	由左向右
|	由左向右
&&	由左向右
||	由左向右
?:	由右向左
= += -= *= /= %=	由右向左
<<= >>= &= |= ^=	由右向左
throw	由右向左
,（顺序运算符）	由左向右

附录C 字 符 集

C.1 ASCII（American Standard Code for Information Interchange，美国标准信息交换码）

下表给出了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	sp	!	"	#	$	%	&	'
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		

表中第1列数字0~12指定ASCII字符集中字符编码值的左边数字，第2行中的数字0~9指定编码值的右边数字。例如，位于第6行、第5列的字符A的编码值是65（ASCII字符集中第66个字符）。注意，编码值为32的字符是空格符。

ASCII字符集中前32个字符（编码值为00～31的字符）和编码值为127的字符是不可打印字符。下表给出了这些字符的缩写和表示的意义。

nul	空字符	ff	换页	can	取消
soh	序始	cr	回车	em	载终
stx	文始	so	移出	sub	取代
etx	文终	si	移入	esc	扩展
eot	送毕	del	转义	fs	卷隙
enq	询问	dc1	机控1	gs	群隙
ack	确认	dc2	机控2	rs	录隙
bel	告警	dc3	机控3	us	原隙
bs	退格	dc4	机控4	sp	空格
ht	横表	nak	否认	del	删除
lf	换行	syn	同步		
vt	纵表	etb	组终		

C.2 EBCDIC（Extended Binary Coded Decimal Interchange Code，扩展二进制－十进制交换码）

下表给出了EBCDIC字符集。

	EBCDIC									
	0	1	2	3	4	5	6	7	8	9
6					b̲					
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

表中第1列数字6~24指定EBCDIC字符集中字符编码值左边的数字，第2行中的数字0~9指定字符编码值右边的数字。例如，位于第19行、第3列的字符A在EBCDIC字符集中的编码值是193（是EBCDIC字符集中第194个字符）。注意，编码值为64的字符表示空格符。该表未列出所有EBCDIC字符集中的字符。实际上，编码值为00~63和250~255之间的字符为不可打印的控制符。

附录D 运算符重载

下表列出了可以重载的运算符。

可以重载的运算符

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td></tr>
<tr><td>!</td><td>&&</td><td>||</td><td>=</td><td>==</td><td><</td><td><=</td><td>></td></tr>
<tr><td>>=</td><td>!=</td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td></tr>
<tr><td>|=</td><td>&=</td><td><<</td><td>>></td><td>>>=</td><td><<=</td><td>++</td><td>--</td></tr>
<tr><td>->*</td><td>,</td><td>-></td><td>[]</td><td>()</td><td>~</td><td>new</td><td>delete</td></tr>
</table>

下表列出了不可重载的运算符。

不可以重载的运算符

.	.*	::	?:	sizeof

附录E　ANSI/ISO标准C++和标准C++中头文件命名规则

本书中的所有程序都使用ANSI/ISO标准C++编写，（在前几个章中）同时也给出了标准C++编写的代码。从本书中的程序可以看出，标准C++的头文件有扩展名.h，而ANSI/ISO标准C++中的头文件没有扩展名。另外，在ANSI/ISO标准C++中某些头文件在原有头文件的名字（比如math.h）前面添加了字母c。C++语言起源于C语言，所以某些头文件是从C语言引入的，例如math.h，stdlib.h和string.h。另一些头文件是专为C++设计的，如iostream.h，iomanip.h和fstream.h。我们知道，将一个头文件包含在程序中，头文件中的全局标识符也就成为程序中的全局标识符。在ANSI/ISO标准C++中，为了使用名字空间（Namespace）机制，所有头文件都进行了修改，使得标识符的声明只在一个名为std的名字空间中。在ANSI/ISO标准C++中，抛弃了那些专为C++设计的头文件的.h扩展符。而那些从C引入到C++的头文件的.h扩展符也被抛弃并在名字前添加字母c。

下面是标准C++和ANSI/ISO标准C++中最常用的头文件。

标准C++头文件	ANSI/ISO标准C++头文件
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

可以使用下面的语句将头文件（比如iostream）包含到程序中：

```
#include <iostream>
```

注意，在使用标识符（cin，cout，endl等）时，可以在程序中加入下面语句：

```
using namespace std;
```

或者在标识符前加上前缀std::。

附录F 头 文 件

C++标准库提供了许多预定义函数、命名常量和特殊的数据类型。本附录将介绍一些最常用的库函数，及命名常量。要想获得更多的信息，请参阅系统提供的文档。标准C++的头文件名在括号中给出。

F.1 头文件cassert（assert.h）

下表描述了函数assert。函数说明在头文件cassert（assert.h）中。

assert(expression)	expression可为任意的int型表达式 expression通常为逻辑表达式	● 如果expression的值为非0（true），程序继续执行 ● 如果expression的值为0（false），程序立即中断执行。并显示expression，源代码所在的文件名和所在行

注意：*若要禁止所有的assert语句，可以在指令#include<cassert>前面添加指令#define NDEBUG。*

F.2 头文件cctype（ctype.h）

下表给出了头文件cctype（ctype.h）中的函数。

函数名和参数	参数类型	函数返回值
isalnum(ch)	ch为char类型	函数返回一个整数： ● 如果ch为字母或数字，即（'A'~'Z', 'a'~'z', '0'~'9'），函数返回一个非0值（true） ● 否则返回0（false）
iscntrl(ch)	ch为char类型	函数返回一个整数： ● 如果ch为控制字符（在ASCII码中编码值为0~31或127），函数返回一个非0值（true） ● 否则返回0（false）
isdigit(ch)	ch为char类型	函数返回一个整数： ● 如果ch为数字（'0'~'9'），函数返回一个非0值（true） ● 否则返回0（false）
islower(ch)	ch为char类型	函数返回一个整数： ● 如果ch为小写字母（'a'~'z'），函数返回一个非0值（true） ● 否则返回0（false）
isprint(ch)	ch为char类型	函数返回一个整数： ● 如果ch为可打印字符或空格，函数返回一个非0值（true） ● 否则返回0（false）
ispunct(ch)	ch为char类型	函数返回一个整数： ● 如果ch为标点符号，函数返回一个非0值（true） ● 否则返回0（false）
isspace(ch)	ch为char类型	函数返回一个整数： ● 如果ch为空白字符（空格符、换行符、制表符、回车符、换页符），函数返回一个非0值（true） ● 否则返回0（false）

（续表）

函数名和参数	参数类型	函数返回值
isupper(ch)	ch 为 char 类型	函数返回一个整数: ● 如果 ch 为大写字母（'A'~'Z'），函数返回一个非 0 值（true） ● 否则返回 0（false）
tolower(ch)	ch 为 char 类型	函数返回一个字符值: ● 如果 ch 为大写字母，函数返回 ch 相应的小写字母 ● 否则返回 ch
toupper(ch)	ch 为 char 类型	函数返回一个字符值: ● 如果 ch 为小写字母，函数返回 ch 相应的大写字母 ● 否则返回 ch

F.3 头文件 cfloat（float.h）

第2章给出了浮点类型的最大值和最小值，并且说明这些值是与系统有关的。这些最大和最小值保存在命名常量中。头文件 cfloat 中包含许多这样的命名常量。下表列出了部分常量:

命名常量	描述
FLT_DIG	float 类型值的有效数的近似位数
FLT_MAX	float 类型的最大正值
FLT_MIN	float 类型的最小正值
DBL_DIG	double 类型值的有效数的近似位数
DBL_MAX	double 类型的最大正值
DBL_MIN	double 类型的最小正值
LDBL_DIG	long double 类型值的有效数的近似位数
LDBL_MAX	long double 类型的最大正值
LDBL_MIN	long double 类型的最小正值

下面程序可以输出所在系统上的这些命名常量的值:

```
#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;
}
```

如果要使用标准 C++ 的头文件，可将语句：

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

替换为：

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

F.4 头文件 climits（limits.h）

第 2 章给出了 int 型的最大值和最小值，并且说明这些值是与系统有关的。这些最大值和最小值保存在命名常量中。头文件 climits 中包含许多这样的命名常量。下表列出了部分常量。

命名常量	描述
CHAR_BIT	一个字节中的位数
CHAR_MAX	char 类型的最大值
CHAR_MIN	char 类型的最小值
SHRT_MAX	short 类型的最大值
SHRT_MIN	short 类型的最小值
INT_MAX	int 类型的最大值
INT_MIN	int 类型的最小值
LONG_MAX	long 类型的最大值
LONG_MIN	long 类型的最小值
UCHAR_MAX	unsigned char 类型的最大值
USHRT_MAX	unsigned short 类型的最大值
UINT_MAX	unsigned int 类型的最大值
ULONG_MAX	unsigned long 类型的最大值

下面程序可以输出所在系统上的这些命名常量的值：

```
#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;
}
```

如果要使用标准C++的头文件，可将语句：

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

替换为：

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

F.5 头文件cmath（math.h）

下表给出了部分数学函数。

函数名和参数	参数类型	函数返回值
acos(x)	x为浮点类型的表达式，$1.0 \leqslant x \leqslant 1.0$	x的反余弦，值在0.0和π之间
asin(x)	x为浮点类型的表达式，$-1.0 \leqslant x \leqslant 1.0$	x的反正弦，值在$-\pi/2$和$\pi/2$之间
atan(x)	x为浮点类型的表达式	x的反正切，值在$-\pi/2$和$\pi/2$之间
ceil(x)	x为浮点类型的表达式	大于等于x的最小整数
cos(x)	x为浮点类型的表达式，x表示弧度	弧度x的余弦值；如：$x=90$, $\cos(x)=0$
cosh(x)	x为浮点类型的表达式	x的双曲余弦值
exp(x)	x为浮点类型的表达式	求e的x次幂（e=2.718…）
fabs(x)	x为浮点类型的表达式	x的绝对值
floor(x)	x为浮点类型的表达式	小于等于x的最大整数
log(x)	x为浮点类型的表达式，且$x>0.0$	x的自然对数（基于e）
log10(x)	x为浮点类型的表达式，且$x>0.0$	x的常用对数（基于10）
pow(x,y)	x，y均为浮点类型的表达式，如果$x=0.0$，y必须为正，如果$x \leqslant 0.0$，则y必须为整数	x的y次幂
sin(x)	x为浮点类型的表达式，x表示弧度	弧度x的正弦值，如：$x=90$, $\sin(x)=1$
sinh(x)	x为浮点类型的表达式	x的双曲正弦值
sqrt(x)	x为浮点类型的表达式，且$x \geqslant 0.0$	x的平方根
tan(x)	x为浮点类型的表达式，x表示弧度	弧度x的正切值，如：$x=45$, $\tan(x)=1$
tanh(x)	x为浮点类型的表达式	x的双曲正切值

F.6 头文件cstddef（stddef.h）

头文件cstddef中定义了下面的符号常量：

NULL：与系统相关的空指针（通常为0）

F.7 头文件cstring（string.h）

下表给出了部分字符串函数。

（续表）

函数名和参数	参数类型	函数返回值
strcat(destStr, srcStr)	destStr 和 srcStr 均为以 NULL 终止的字符数组；destStr 的大小必须能够容纳源字符串	返回 destStr 的首地址；srcStr（包括 NULL 字符）被连接到 destStr 的末尾
strcmp(str1, str2)	str1 和 str2 均为以 NULL 终止的字符数组	返回值为： ● 如果 str1 < str2，则返回<0 的 int 值 ● 如果 str1 = str2，则返回 0 ● 如果 str1 > str2，则返回>0 的 int 值
strcpy(destStr, srcStr)	destStr 和 srcStr 均为以 NULL 终止的字符数组	返回 destStr 的首地址；srcStr 被拷贝到 destStr
strlen(str)	str 为以 NULL 终止的字符数组	返回 str 的长度（不包括'\0'）

F.8　头文件 string

不要将头文件 string 与头文件 cstring 混淆。string 头文件提供了数据类型 string。与 string 类型相关连的是数据类型 string::size_type 和一个命名常量 string::npos。它们定义如下所示：

```
string::size_type        一个无符号整数
string::npos             string::size_type 类型的最大值
```

下表给出了部分与 string 类型相关的函数。除非特别说明，str，str1 和 str2 都是 string 类型的变量（对象）。在 string 变量中第一个字符位置为 0，第二个字符位置为 1，以此类推。

函数名称和参数	参数类型	作用和函数返回值
str.c_str( )	无	与 str 中字符对应的以 NULL 字符结尾的 C-string 的首地址
getline(istreamVar, str)	istreamVar 为输入流（istream 类型或 ifstream 类型）str 为 string 对象（变量）	从 istreamVar 中读入字符（直到读入换行符为止）并存储在 str 中（换行符被读入但不存储在 str 中）。函数的返回值通常被忽略
str.empty( )	无	如果 str 为空（即 str 中字符个数为 0），返回 true；否则返回 false
str.length( )	无	返回 string 中字符个数，其为 string::size_type 类型
str.size( )	无	返回 string 中字符个数，其为 string::size_type 类型
str.find(strExp)	str 为 string 对象，strExp 为 string 表达式。strExp 可以是一个字符	find 函数在 str 中查找由 strExp 指定的字符串或字符的第一次出现的位置。如果查找成功，则返回开始匹配的位置，否则，函数返回特殊值 string::npos
str.substr(pos, len)	两个无符号整数，pos 和 len。pos 表示（str 中子字符串的）起始位置，len 表示（str 中子字符串的）长度。pos 的值必须小于 str.length( )	返回一个包含 str 中从 pos 开始的子字符串的临时字符串对象。子字符串的长度最长为 len，如果 len 过大，则子字符串到 str 的末尾
str1.swap(str2);	一个 string 类型的参数；str1 和 str2 都是 string 类型的对象	str1 和 str2 的内容相交换
str.clear( );	无	从 str 中删除所有字符
str.erase( );	无	从 str 中删除所有字符

（续表）

函数名称和参数	参数类型	作用和函数返回值
str.erase(m);	一个 string::size_type 类型的参数	删除 str 中从位置 m 开始的所有字符
str.erase(m, n);	两个参数均为 int 类型	从位置 m 开始，在 str 中删除接下来的 n 个字符。如果 n 大于 str 的长度，则删除所有从 m 开始的字符
str.insert(m, c);	参数 m 为 string::size_type 类型，c 为字符	将字符 c 插入到 str 中 m 处
str.insert(m, n, c);	参数 m 和 n 都是 string::size_type 类型，c 为字符	将 n 个字符 c 插入到 str 中 m 处
str1.insert(m, str2);	参数 m 为 string::size_type 类型	将 str2 中所有字符插入到 str1 中 m 处
str1.replace(m, n, str2);	参数 m 和 n 都是 string::size_type 类型	从 m 开始，用 str2 中的所有字符替换 str1 中接下来的 n 个字符，如果 n 大于 str1 的长度，则 str1 从 m 到末尾所有字符都被替换

附录G　系统中数据类型长度

下面程序给出了所在系统上内建类型所占用的存储空间大小（程序的输出为程序运行时所在系统的内建类型的大小）。

```
#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 参考文献

1. G. Booch, *Objected-Oriented Analysis and Design*, Second Edition,Addison-Wesley, 1995.
2. E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms C++*, Computer Science Press, 1997.
3. R. Johnsonbaugh, *Discrete Mathematics*, Fifth Edition, Prentice-Hall, Upper Saddle River, NJ, 2001.
4. N.M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*,Addison-Wesley, Reading, MA, 1999.
5. D.E.Knuth, *The Art of Computer Programming*,Vols. 1-3,Addison-Wesley, 1973, 1969, 1973.
6. S.B. Lippman and J. Lajoie, *C++ Primer*,Third Edition,Addison-Wesley, Reading, MA, 1998.
7. E.M. Reingold and W.J. Hensen, *Data Structures in Pascal*, Little Brown and Company, Boston, MA, 1986.
8. R. Sedgewick, *Algorithms in C*,Third Edition,Addison-Wesley, Reading, MA, Parts 1-4, 1998; Part 5, 2002.
9. B. Stroustrup, *The Design and Evolution of C++*,Addison-Wesley, Reading, MA, 1994.

附录I　部分参考答案

第1章

1. a 正确；b 错误；c 错误；d 错误；e 错误；f 错误；g 错误；h 正确；i 正确；j 正确
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. 高级程序设计语言中的指令更接近于如英语等自然语言，因此比机器语言易学、易记。

第2章

1. a 错误；b 错误；c 错误；d 正确；e 正确；f 错误；g 正确；h 正确；i 错误；j 正确；k 错误
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; 正确
9. a和c合法
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正确
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 正确；b 正确；c 错误；d 错误；e 正确；f 正确

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 非法的数据会导致输入流进入错误状态。
  b 当输入流进入错误状态时，下面所有与该输入流有关的输入都会被忽略掉。无论变量中的值是多少，程序都将继续进行。

第4章

1. a 错误；b 错误；c 错误；d 正确；e 错误；f 错误；g 错误；h 错误；i 错误；j 正确
2. a i；b i；c ii；d i；e iii.
5. 漏掉else的分号，正确语句是：

```
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 错误；b 正确；c 错误；d 正确；e 正确；f 正确；g 正确
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 无穷循环

c 无穷循环

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 错误；b 正确；c 正确；d 正确；e 错误

3. a 非法，没有指定函数返回值的类型。

b 合法。

c 非法，没有指定参数b的类型。

d 非法，在函数名后面没有括号。

5. a 4

b 26

c 10 4 0

d 0

8. a (i) 125; (ii) 432

b 函数计算x^3，在此x是函数的参数。

第7章

1. a 正确；b 错误；c 正确；d 错误；e 正确；f 错误；g 错误；h 错误；i 正确

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), (iv)正确

第8章

1. a 正确；b 错误；c 正确；d 错误；e 错误；f 正确；g 正确；h 正确；i 错误；j 正确；k 错误

2. a 正确；b 错误；c 正确；d 错误

6. 名字空间机制不能与标准C++格式头文件（带有扩展名.h）一起使用。所以，应该去掉第1行中的.h。

8. 第3行中漏掉了关键字namespace。

第9章

1. a 正确；b 正确；c 错误；d 错误；e 正确；f 错误；g 错误；h 错误；i 正确；j 错误；k 错误

2. a 合法。

b 合法。

c 不合法，C-strings上不能使用赋值运算符。

d 合法。

e 合法。

f 不合法，C-strings上不能使用关系运算符。

g 合法。

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 合法。

b 合法。

c 合法。

d 不合法，C-strings 上不能使用赋值运算符。

e 不合法，C-strings 上不能使用关系运算符和赋值运算符。

f 合法。

g 合法。

h 合法。

11. List elements: 11 16 21 26 30

13. a 30

b 5

c 列

14. a int alpha[10][20];

b
```
for(j = 0; j < 10; j++)
    for(k = 0; k < 20; k++)
        alpha[j][k] = 0;
```

c
```
for(j = 0; j < 10; j++)
{
    for(k = 0; k < 20; k++)
        cout<<alpha[j][k]<<" ";
    cout<<endl;
}
```

15. a beta is initialized to zero.

d First row of beta: 0 2 0
  Second row of beta: 2 0 2
  Third row of beta: 0 2 0

第10章

1. a 正确；b 正确；c 错误；d 错误；e 错误

2. a 基本实例：
```
if(u == 0)
    cout<<v;
else if(u == 1)
    cout<<static_cast<char>(static_cast<int>(v) + 1);
```

b 递归归约：funcRec(u – 1, v);

c B

第11章

1. a 错误；b 错误；c 正确；d 正确；e 正确；f 正确；g 错误。
2. a 非法。newEmployee的成员name是一个结构。应该分别指定name的成员来存储"John Smith"。例如：

```
newEmployee.name.first = "John";
newEmployee.name.first = "Smith";
```

b 非法。newEmployee的成员name是一个结构，而在结构上并没有整体输入操作。正确的语句是：

```
cin >> newEmployee.name.first;
cin >> newEmployee.name.last;
```

c 合法。

d 合法。

e 非法。employees是一个数组，而在数组上并没有整体赋值操作。

第12章

1. a 错误；b 错误；c 正确；d 错误；e 错误
4. a (i) 第1行的构造函数。
   (ii) 第3行的构造函数。
   (iii) 第4行的构造函数。

b
```
CC::CC()
{
    u = 0;
    v = 0;
}
```

c
```
CC::CC(int x)
{
    u = x;
    v = 0;
}
```

第13章

1. a 正确；b 正确；c 正确；d 错误
4. 类的私有成员是私有的，不能被其派生类的成员函数直接访问；类的受保护成员可以被其派生类的成员函数直接访问。
5. a 语句：

```
class bClass public aClass
```

应为：

```
class bClass: public aClass
```

b }后面缺少分号。

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14章

1. a 错误；b 错误；c 错误；d 正确；e 正确；f 正确；g 错误；h 错误
2. a 合法。
   b 合法。
   c 非法。因为p是指针变量，x是int类型变量，所以x的值不能赋给p。
   d 合法。
   e 合法。
   f 非法。因为*p是int类型变量，q是指针变量，所以q的值不能赋给*p。
5. b和c。
7. 78 78
8. 第5行中的语句将p的值拷贝到q中。在该语句执行之后，p和q指向相同的内存单元。第7行中的语句将q所指的内存空间释放，这导致p和q中的值无效。因此，第8行中的语句输出的值不可预料。
9. 4 4 5 7 10 14 19 25 32 40
12. 第6行中的语句将p的值拷贝到q中。在该语句执行之后，p和q指向相同的数组。第7行中的语句将p所指的数组内存空间释放，这导致q中的值无效。因此，第9行中的语句输出的值不可预料。
16. 含有指针数据成员的类应该包含析构函数、重载赋值运算符、并且明确提供拷贝构造函数。
17.
```
ClassA x: 4

ClassA x: 6
ClassB y: 5
```
19. 在编译时绑定中，编译器生成调用函数的必要代码；在运行时绑定中，系统生成调用函数的必要代码。

第15章

1. a 错误；b 正确；c 正确；d 错误；e 错误；f 正确；g 错误；h 正确；i 错误；j 正确；k 错误
5. 当类含有指针数据成员时。
7.
```
a friend strange operator+(const strange&, const strange&);
b friend bool operator==(const strange&, const strange&);
c friend strange operator++(strange&, int);
```

9. 在第2行的语句中，在关键字bool前缺少关键字friend。正确的语句应该是：

```
friend bool operator <= (mystery, mystery);          //Line 2
```

11. 无。
12. 两个。
17. 第4行错误。只可以在内建的数据类型或是用户自定义的数据类型上进行模板实例化。所以，必须将尖括号内的类型换成内建的数据类型或是用户自定义的数据类型。
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16章

1. a 错误；b 错误；c 错误；d 错误；e 正确；f 正确
3. a 正确；b 正确；c 错误；d 错误；e 正确
4. a 合法。
   b 合法。
   c 合法。
   d 非法（B是指针，而*List是结构）。
   e 合法。
   f 非法（B是指针，而A->link->info是int类型）。
   g 合法。
   h 合法。
   i 合法。
6. 这是一个无限循环，将不断打印18。
7. a 这是一段非法代码。语句s->info = B;非法，因为B是指针，而s->info是int类型。
   b 这是一段非法代码。在语句s = s->link;执行之后，s为NULL，所以s->info不存在。

第17章

1.

```
x = 3
y = 9
7
13
4
7
```

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);
    }
}
```

第18章

1. a 错误; b 正确; c 错误; d 错误; e 错误; f 错误
2. a 8; b 6; c 1; d 8
3. b

迭代	first	last	mid	list [mid]	比较次数
1	0	10	5	55	2
2	0	4	2	17	2
3	3	4	3	45	2
4	4	14	4	49	1 (found 为 true)

在位置4上找到该元素，总比较次数是7次。

d

迭代	first	last	mid	list [mid]	比较次数
1	0	10	5	55	2
2	6	10	8	92	2
3	9	10	9	98	2
4	10	10	10	110	2
5	11	10	循环停止		

查找失败，总比较次数是8次。

4. 3

6. 10, 12, 18, 21, 25, 28, 30, 71, 32, 58, 15

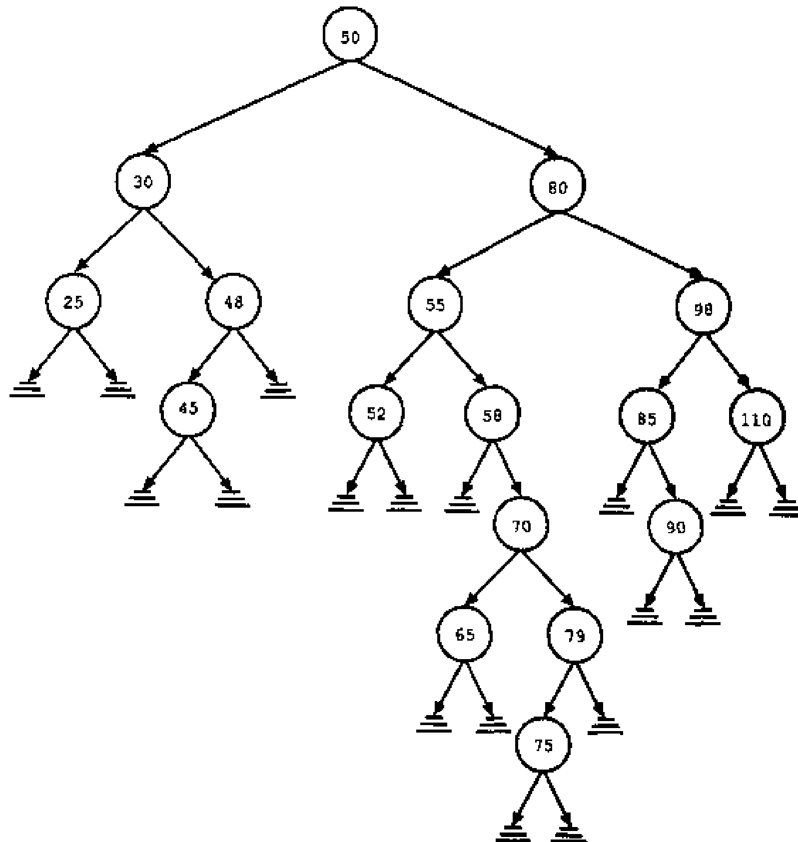
9. a. 36, 38, 32, 16, 40, 28, 48, 80, 64, 95, 54, 100, 58, 65, 55

11.
```
template<class elemType>
```

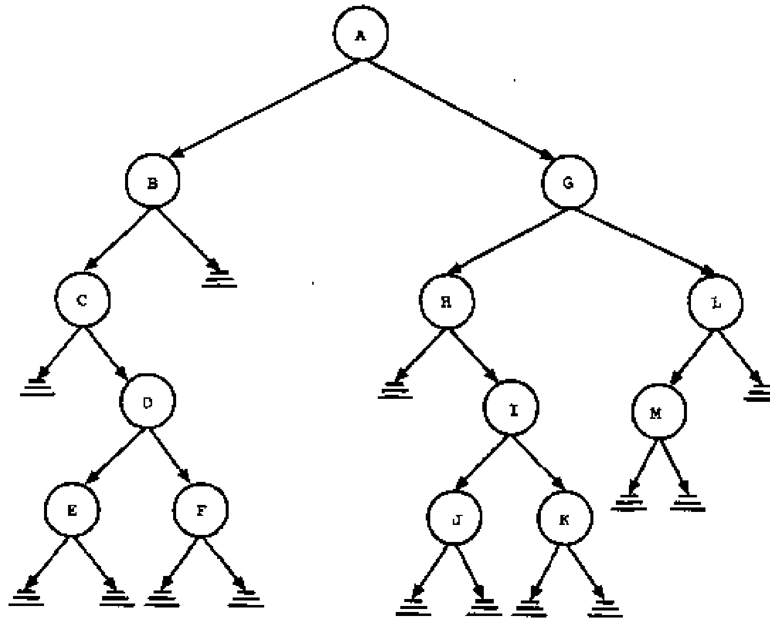
```
class orderedArrayListType: public arrayListType<elemType>
{
public:
    int binarySearch(const elemType& item);
    void insertOrd(const elemType&);

    void selectionSort();
    void insertionSort();
    void quickSort();

    orderedArrayListType(int size = 100);

private:
    void recQuickSort(int first, int last);
    int partition(int first, int last);
    void swap(int first, int second);
    int minLocation(int first, int last);
};
```

第19章

1. a 错误; b 正确; c 错误; d 错误
3. L_A = {B, C, D, E}
4. R_A = {F, G}
5. R_B = {E}
6. D C B E A F G
7. A B C D E F G
8. D C E B G F A
9. 80-55-58-70-79
10. 50-30-40
12.

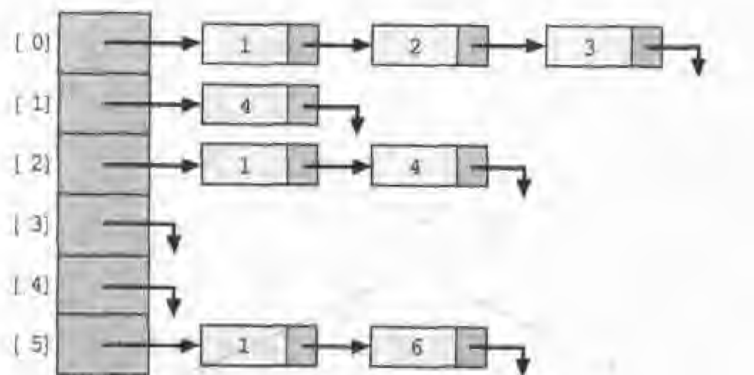


15.



第20章

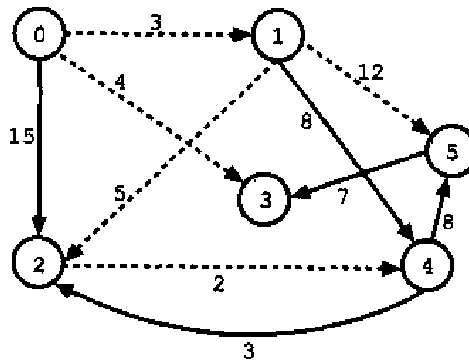
2.



3. 0 1 4 2 3 5

4. 0 1 2 3 4 5

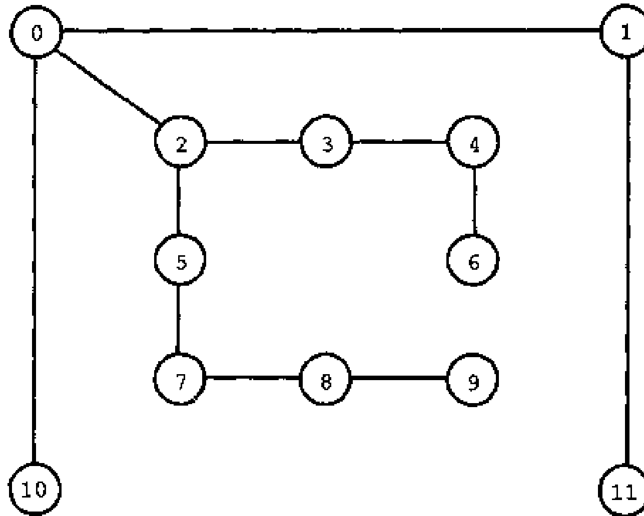
6.




```
Source Vertex: 0
Shortest Distance from Source to each Vertex
Vertex  Shortest_Distance
   0           0
   1           3
```

```
2          8
3          4
4         10
5         15
```

7.



第21章

1. STL的三个主要组成部分是：容器、迭代器和算法。
2. 容器用于存储数据；算法用于操作存储在容器中的数据。
4. set不允许元素重复；multiset允许元素重复。
6. vecList = {8, 23, 40, 6, 18, 9, 75, 9, 75}
10. A B A K
11. 18 5 11 56 27 2
13. 0