www.Linuxidc.com

数据结构 C++语言描述

[美] William Ford, William Topp 著

刘卫东 沈官林 译

严蔚敏 审校

清华大学出版社

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

前　言

本书从面向对象(object-oriented)的角度来讲述数据结构的基础知识。数据结构是计算机科学专业的一门核心课程,它的研究对象为问题求解方法、程序设计方法及一些典型数据结构的算法。

本书使用通用的C++语言作为算法描述语言。它的类和面向对象结构可以有效地实现数据结构的算法。虽然目前有多种面向对象的语言,但由于C++起源于广泛流行的C语言,因而在这些语言中占有优势。抽象数据类型(ADT)定义了数据组织和数据处理运算,本书将围绕这一概念来讨论每一种数据结构,并采用C++语言中的类来表示ADT,在对象中有效地使用这些结构。

本书结构

《数据结构C++语言描述》围绕多数据集类——表、树、集合、图和字典来组织数据结构的学习。本书包括数据结构基本内容和面向对象程序设计方法两部分,给出了许多完整程序或程序段例子,并引入了描述算法复杂度的大O方法。

第1章至第11章给出了初级数据结构课程(CS 2)的内容,第12章介绍继承和抽象类,第13和14章介绍非线性结构及其排序和查找算法。这几章覆盖了后续的数据结构与算法课程(CS 7)和高级程序设计课程的内容。另外,本书还介绍了模板和运算符扩充,以支持样板结构,并使用C++语言创建数据结构和简化数据结构的使用。

本书可作为计算机专业学生学习数据结构和面向对象程序设计方法的教材,也可供计算机专业工作者自学提高时使用。

各章简介

本书的大部分章节介绍抽象数据类型及它们作为C++的类的应用,包括类及其主要方法的声明(declaration)。除少数情况下只给出类的简单定义和方法外,大多数情况下均给出了类的完整定义,并用程序给出了类的完整实现。

第1章　概述

本章引入抽象数据类型(ADT)的概念及其基本性质,即数据封装、信息隐藏、继承性

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

和多态性,并简单介绍了用 C++进行面向对象程序设计的方法。第 12 章中将对继承性和多态性进行全面介绍。

第 2 章　基本数据类型

程序设计语言提供最基本的数值和字符类型——整型、浮点型、字符型和用户定义的枚举型,并用这些基本类型组合产生了数组、记录、串和文件类型。本章以 C++为例描述了这些基本数据类型的 ADT。

第 3 章　抽象数据类型和类

本书描述了抽象数据结构及其用 C++类的表示。本章定义了类的基本概念及其建立和使用的方法。

第 4 章　集合类

集合是具有增加、删除、修改数据项功能的存储类。本书重点是对集合类的研究。本章给出了不同集合类型的例子。另外,还引入了衡量算法复杂度的大 O 方法。本书使用该方法来比较不同的算法。本书最后以一个 SeqList 类的研究结束,这是一个普通表结构的典型示例。

第 5 章　栈和队列

本章讨论栈和队列,它们分别以后进先出(LIFO)和先进先出(FIFO)顺序处理数据。另外,还讨论具有优先级的队列,这是一种特殊的队列,它每次从队列中删除权限最高的元组。

第 6 章　抽象运算

抽象数据类型定义了初始化和处理数据的方法。本章扩展 C++语言定义的运算符(如+,-,*,<<等)来应用于抽象数据类型,这被称为运算符重载,它重新定义标准运算符使之适用于 ADT 的抽象运算。最后,用有理数类作为例子说明如何进行运算符重载及类型转换,并介绍了用友函数重载标准 C++的输入/输出运算符的方法。

第 7 章　形式数据类型

C++使用模板机制提供支持不同数据类型的模板函数和类。模板为数据结构提供了强大的概括能力。本章以基于模板的 Stack 类及其对中缀表达式求值的应用为例说明了这些概念。

第 8 章　类和动态存储

动态数据结构使用运行时才由系统分配的内存,这可以让用户定义没有大小限制的结构,增强了类的可用性。当然,在使用它们时需要特别小心。本章介绍复制构造函数、重载赋值运算和析构函数方法,用它们可以正确地复制和赋值动态数据,并在对象被删除时释放内存空间。我们用 Array,String 和 Set 类说明动态数据的强大功能。这些类的使用将贯穿于本书。

第 9 章　链表

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

由于许多应用可用表来实现，用表来存储和查找数据是贯穿本书的一个论题。本章介绍可动态处理的链表。建立链表的一种方法是，首先定义一个基本的结点类，然后创建函数来增加或删除表中的数据项；另一种方法是建立一个包含遍历机制的链表类。类 LinkedList 用来实现类 SeqList 和类 Queue。这些方法提供了开发数据结构的有力工具。

第 10 章　递归

递归在计算机科学和数学中都是一个重要的问题求解工具。本章介绍递归并给出它的多个应用，如数学公式计算、组合算法、遍历迷宫和猜谜。最后，以 Fibonacci 队列为例，比较了递归算法、交互算法和直接计算三种算法。

第 11 章　树

链表是从表头顺序存取的结点集，这种数据结构称为线性表。在许多应用中，对象中的一个成员可产生多个后代，不具备线性性质。本章介绍一种基本的非线性结构——树，它的所有结点都由称为根(root)的结点产生的，树是用来描述继承关系，如计算机文件系统和商业报表的理想结构。首先集中讨论二叉树，即每个结点最多有两个后代。我们给出类 TreeNode 来实现二叉树及其应用，包括前序、中序和后序遍历算法。最后，在应用举例中，给出了以类 BinSTree 实现的二叉查找树的结构，它可用于高效地存取大批量数据。

第 12 章　继承和抽象类

继承是面向对象程序设计的基本概念。本章讨论继承的主要性质，给出其在 C++ 中的应用，并引入虚函数作为使用继承的工具。另外，它还给出了只有虚函数的基类。虚函数是面向对象程序设计的基础，在本书的后续章节中将会用到。本章引入迭代算子的概念，定义了对本书中出现的不同的表进行遍历的一个通用的算法。最后，给出了以继承和虚函数实现的异构数组及链表的例子。

第 13 章　高级非线性结构

本章继续讨论二叉树，并介绍其它的非线性结构，描述了基于数组的树，即将数组看成完全二叉树。本章对堆进行了研究，用它来实现堆排序及有权队列。尽管在一般情况下，二叉查找树是实现表的最好结构，但也存在一些不足。数据结构提供不同的深度平衡结构来保证最快的查找时间。通过继承，派生了一个新的查找树类，称为 AVL 树。本章最后介绍图的基本性质及关于图的一些传统算法。

第 14 章　集合数据的组织

本章介绍一般的数据集合的查找和排序算法，包括传统的以树组为基础的选择排序、冒泡排序和插入排序算法，以及著名的快排(QuickSort)算法。本书主要讨论的是存放在内存中的内部数据，但大量数据可能存放在外存上，需要相应的查找和排序算法。为此，我们为文件直接查找给出类 BinFile，并用它实现了外部索引顺序查找及外部归并排序算法。

背景知识

本书假定读者已修完程序设计方面的先修课，并熟悉基本的C++语言。第2章给

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

出了 C++ 的初级数据结构，用几个完整的例子给出了它们的应用。该章可作为学习本书所需 C++ 知识的概述。对感兴趣的读者，作者还给出 C++ 语言中初级类型的定义、数组、流程控制、I/O、函数和指针等的句法。并分别给出了一些例子、程序和习题。

补充材料

本书用到的所有类和程序的完整的源代码可通过 Internet 的 ftp 从作者所在的 Pacific 大学得到。本书所用 C++ 代码已在最新的 Borland 编译器上测试过。除了极少数例外，这些程序也可在用 Symantec C++ 的 Macintosh 系统和用 GNU C++ 的 UNIX 系统上编译和运行。

在 Internet 上，用 ftp 联结 ftp.cs.uop.edu，联上后，用 anonymous 用户名登录，口令是用户自己的 Internet 电子邮件地址。所有的软件均在/pub/C++ 目录下。

需要上述 C++ 辅导材料的读者，可和作者直接联系，通过电子邮件和普通邮件均可。电子邮件地址：billf @ uop. edu。通信地址：Bill Topp, 456 S. Regent, Stockton, CA 95204。

教师指导书给出了每章的讲课要点、大多数习题的答案和考试的样题，还给出了多数上机题的解题思路。需要教师指导书的任课教师可联络本地 Prentice Hall 公司的代表。

（使用本书中文版作为教材授课的教师，请联络 Prentice Hall 公司北京代表处，电子邮件地址：ssbj@bupt.edu.cn，通信地址：北京 100086，知春里 28 号开源商务写字楼 102 室。目前教师指导书仅能免费提供英文版。）

致谢

作者在准备《数据结构 C++ 语言描述》的过程中，得到许多朋友、学生和同事的支持。Pacific 大学慷慨地提供了许多资源支持完成此项工作。Prentice Hall 出版公司派出精干队伍完成本书的设计和出版。我们要特别感谢编辑 Elizabeth Jones，Bill Zobrist 和 Alan Apt，Bayani de Leon。本书由 Prentice Hall 和 Spectrum 出版服务公司共同完成，作者也得到 Spectrum 的 Kelly Ricci 和 Kristin Miller 的大力支持。

学生们通过直接和间接的反馈给我们的初稿提出了许多有价值的意见，我们的编审对初稿从内容到组织方式都给出了许多指导和建议。在此，我们要特别列出他们的姓名，Georgia 大学的 Hamid R. Arabnia；Florida 技术学院的 Rhoda A. Baggs；Michigan - Ann Arbor 大学的 Sandra L. Bartlett；美国海岸警卫队学院的 Richard T. Close；美国空军学院的 David Cook；Cotonsville（Baltimore 县）社会学院的 Charles J. Dowling；Mankato 州立大学的 David J. Haglin；California 州立大学 Chicago 分校的 Jim Murphy 和 Herbert Schildt。作者的两位同事，Texas-El Paso 大学的 Ralph Ewton 和 Pacific 大学的 Douglas Smith 对本书的出版做出了巨大的贡献。他们敏锐的洞察力和对作者的全力支持是无法估量的，并极大地提高了本书的质量。

William Ford
William Topp

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

目　录

第 1 章　概述 …… 1
1.1　抽象数据类型 …… 2
1.2　C++类和抽象数据类型 …… 5
1.3　C++应用中的对象 …… 6
1.4　对象设计 …… 8
1.5　类继承的应用 …… 16
1.6　面向对象程序设计 …… 17
1.7　程序测试与维护 …… 23
1.8　C++程序设计语言 …… 24
1.9　抽象基类及多态性* …… 25
书面作业 …… 26

第 2 章　基本数据类型 …… 29
2.1　整型 …… 30
2.2　字符类型 …… 33
2.3　实数类型 …… 34
2.4　枚举类型 …… 36
2.5　指针 …… 37
2.6　数组类型 …… 39
2.7　文本串及变量 …… 43
2.8　记录 …… 48
2.9　文件 …… 49
2.10　数组和记录的应用 …… 53
书面作业 …… 60
上机题 …… 66

第 3 章　抽象数据类型和类 …… 69
3.1　用户类型类 …… 70
3.2　类的举例 …… 77
3.3　对象和信息传递 …… 83
3.4　对象数组 …… 84
3.5　多构造函数 …… 85
3.6　应用举例：三角矩阵 …… 88
书面作业 …… 96
上机题 …… 100

第 4 章　群体类 …… 109
4.1　线性群体 …… 111
4.2　非线性群体 …… 116
4.3　算法分析 …… 118
4.4　顺序查找与折半查找 …… 122
4.5　基本的顺序表类 …… 128
书面作业 …… 136
上机题 …… 140

第 5 章　栈和队列 …… 143
5.1　栈 …… 144
5.2　类 Stack …… 146
5.3　表达式求值 …… 153
5.4　队列 …… 159
5.5　类 Queue …… 161
5.6　优先级队列 …… 171
5.7　实例研究：事件驱动模拟 …… 179
书面作业 …… 190
上机题 …… 193

第 6 章　抽象操作 …… 197
6.1　运算符重载 …… 198
6.2　有理数 …… 203
6.3　有理数类 …… 204
6.4　作为成员函数的有理数运算 …… 206
6.5　作为友元函数的有理数流运算符 …… 207
6.6　有理数的转换 …… 209
6.7　有理数的使用 …… 211
书面作业 …… 215
上机题 …… 222

第 7 章　形式数据类型 …… 225
7.1　模板函数 …… 226
7.2　模板类 …… 229
7.3　表的模板类 …… 231
7.4　中缀表达式求值 …… 233
书面作业 …… 240

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

上机题 …… 241

第 8 章　类和动态存储 …… 245

8.1　指针与动态数据结构 …… 247

8.2　动态申请对象 …… 248

8.3　赋值与初始化 …… 252

8.4　安全数组 …… 257

8.5　串类 …… 263

8.6　模式匹配 …… 273

8.7　整型集合 …… 278

书面作业 …… 288

上机题 …… 297

第 9 章　链表 …… 301

9.1　结点类 …… 304

9.2　构造链表 …… 308

9.3　设计链表类 …… 321

9.4　类 LinkedList …… 324

9.5　LinkedList 类的实现 …… 331

9.6　用链表实现集合 …… 337

9.7　实例研究：打印缓冲池 …… 343

9.8　循环表 …… 349

9.9　双向链表 …… 354

9.10　实例研究：窗口管理 …… 360

书面作业 …… 367

上机题 …… 374

第 10 章　递归 …… 379

10.1　递归的概念 …… 380

10.2　设计递归函数 …… 386

10.3　递归代码和运行时堆栈 …… 390

10.4　用递归进行问题求解 …… 392

10.5　递归评估 …… 410

书面作业 …… 414

上机题 …… 417

第 11 章　树 …… 421

11.1　二叉树结构 …… 426

11.2　设计 TreeNode 函数 …… 430

11.3　树扫描算法的使用 …… 434

11.4　二叉搜索树 …… 445

11.5　二叉搜索树的使用 …… 451

11.6　BinSTree 的实现 …… 455

11.7　实例研究：索引(Concordance) …… 464

书面作业 …… 469

上机题 …… 475

第 12 章　继承和抽象类 …… 477

12.1　继承概述 …… 478

12.2　C++中的继承 …… 480

12.3　多态性和虚函数 …… 487

12.4　抽象基类 …… 495

12.5　迭代算子 …… 498

12.6　有序表 …… 511

12.7　异构表 …… 514

书面作业 …… 521

上机题 …… 530

第 13 章　高级非线性结构 …… 533

13.1　基于数组的二叉树 …… 534

13.2　堆 …… 541

13.3　Heap 类的实现 …… 546

13.4　优先级队列 …… 554

13.5　AVL 树 …… 560

13.6　AVL 树类 …… 564

13.7　树迭代算子 …… 574

13.8　图 …… 579

13.9　Graph 类 …… 581

书面作业 …… 599

上机题 …… 606

第 14 章　群体数据的组织 …… 611

14.1　数组排序的基本算法 …… 612

14.2　快速排序(QuickSort) …… 617

14.3　哈希法(Hashing) …… 627

14.4　哈希表类 …… 632

14.5　搜索方法的性能 …… 640

14.6　二进制文件和外部数据操作 …… 641

14.7　辞典 …… 661

书面作业 …… 668

上机题 …… 673

附录　部分书面作业答案 …… 679

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第1章 概述

1.1 抽象数据类型
1.2 C++类和抽象数据类型
1.3 C++应用中的对象
1.4 对象设计
1.5 类继承的应用
1.6 面向对象程序设计
1.7 程序测试与维护
1.8 C++程序设计语言
1.9 抽象基类和多态性
书面作业

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

本书使用面向对象的程序设计语言 C++ 介绍数据结构和算法。我们把每种数据结构均视为抽象类型，它不但定义了数据的组织方式，还给出了处理数据的运算。这种结构，称为**抽象数据类型**(Abstract Data Type, ADT)，是一种描述用户和数据之间接口的抽象模型。本书用 C++ 语言来表示每一种抽象数据结构，即用**类**(class)来表示 ADT，在具体应用中用**对象**(object)来存储和处理数据。

本书介绍 ADT 的基本概念及其相关属性——数据封装和信息隐藏，并通过一系列实例来阐述 ADT 的设计方法和定义数据组织及运算的一般途径。

C++ 类的创建是我们学习数据结构的基础，本书第 3 章将对其进行详细讨论。在本章中，我们讲述 C++ 类的概况及其如何表示 ADT；在带星号(*)的选读节中，给出了一些 C++ 类的实例；概述了对象设计的基本内容及其继承性，这是面向对象程序设计的基石；还综述了本书所用到的程序设计方法。继承性和多态性扩充了面向对象程序设计的能力，使其可用于开发基于类库的大型软件系统。我们将在第 12 章中进一步阐述继承性和多态性，并有选择地将其用在一些高级数据结构中。

本章是对书中所用概念的预习，在正式学习关键数据结构和面向对象这些概念之前，读者就可以逐渐熟悉它们。

1.1 抽象数据类型

数据抽象是程序设计的中心内容。这种抽象被称为抽象数据类型(ADT)，它定义了数据取值范围和表现结构，以及对数据的操作集。也就是说，ADT 给出了一种用户定义的数据类型，其运算符指明了用户如何操作数据。ADT 与具体应用无关，这可使程序员把注意力集中在数据及其操作的理想模型上。

例 1.1

1. 小型公司维护进货信息的程序。每项进货由一条包括货号、当前库存、单价及进货级别的数据记录描述，其操作应包括修改上述不同数据，并在库存量低于一定数量时修改进货级别。数据抽象应该描述一条包括上述信息域的记录及可供库房经理维护进货记录的一系列操作。这些操作应包括在售出货物时修改库存量、调价时修改价格及在库存小于应再进货级别时给出需进货的信息。

数据

货　号	当前库存	单　价	进货级别

操作

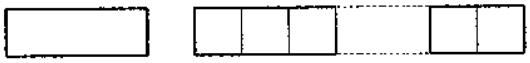
修改库存	UpdateStockLevel
调整单价	AdjustUnitPrice
需订货信息	ReorderItem

2. 掷骰子的游戏程序。在设计中，骰子可描述成下述 ADT：其数据包括被掷骰子数目、掷出骰子的总点数和每个骰子的点数；操作包括掷骰子、返回该次投掷的骰子的总点数以及打印所掷每个骰子的点数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

数据



总点数　　单个骰子点数表

操作

掷骰子　　Toss

求骰子总点数　Total

打印点数　　DisplayToss

ADT 描述规范

下面给出一种描述 ADT 的规范。它包括由 ADT 名称组成的头，对数据类型的描述及操作列表。对每种操作，我们用 input(输入)来指定由用户给定的输入值，precondition(前提)表示该操作可执行前必须具有的数据，process(加工)表示由该操作完成的动作。执行操作后，用 output(输出)来表示返回给用户的值，用 postcondition(结果)来表示在数据内部所作的任何改变。大多数 ADT 都有 initialize(初始化)操作来对数据赋初值。在C++语言环境下，初始化操作称为构造函数(Constructor)。我们用它来简化从 ADT 到它在 C++中的表示的转变。

综上所述，ADT 的描述规范为：

```
ADT    ADT 名称      is
   Data
     描述数据的结构
   Operations
      构造函数
        Initial values:     用来初始化对象的数据
        Process:            初始化对象
      操作 1
        Input:              用户输入的数值
        Preconditions:      系统执行本操作前数据所必需的状态
        Process:            对数据进行的动作
        Output:             返回给用户的数据
        Postconditions:     系统执行操作后数据的状态
      操作 2
              ......
      操作 n
              ......
   end ADT   ADT 名称
```

例　1.2

1. 抽象数据类型 Dice 的数据包括每次所掷骰子数 N，所掷出的总点数和一个有 N 项的存放每个骰子被掷出点数的表。

```
ADT    Dice    is
Data
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

该次投掷骰子的个数。它是一大于或等于1的整数。

该次掷出的总点数。它是一个整数。如果掷N个骰子,则该值在N与6N之间。

该次投掷所掷出的每个骰子的点数表。该表的每个数值均为从1到6的整数。

Operations

Constructor

Initial values:	被掷骰子个数
Process:	初始化数据,给定每次投掷骰子的个数

Toss

Input:	无
Preconditions:	无
Process:	掷骰子并计算总点数
Output:	无
Postconditions:	所掷骰子总点数及每个骰子的点数

Total

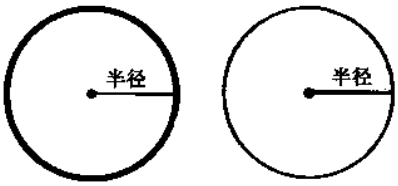
Input:	无
Preconditions:	无
Process:	检索该次投掷的总点数数据项
Output:	返回该次投掷总点数
Postconditions:	无

DiaplayToss

Input:	无
Preconditions:	无
Process:	打印该次掷出的各骰子的点数
Output:	无
Postconditions:	无

end ADT Dice

2、圆是平面上与圆心等距的所有点的集合。从图形显示角度看,圆的抽象数据类型包括圆心和半径,而从计量的角度看,其抽象数据类型只需要半径。我们以从计量角度看的圆为例给出圆的ADT。它包括求面积和求周长的操作。本书不打算过细讨论操作的数学细节。在下节中,我们用该ADT来说明其C++类的定义和程序中对象的使用。



周长　　面积

ADT Circle is

Data

非负实数,给出了圆的半径

Operations

Constructor

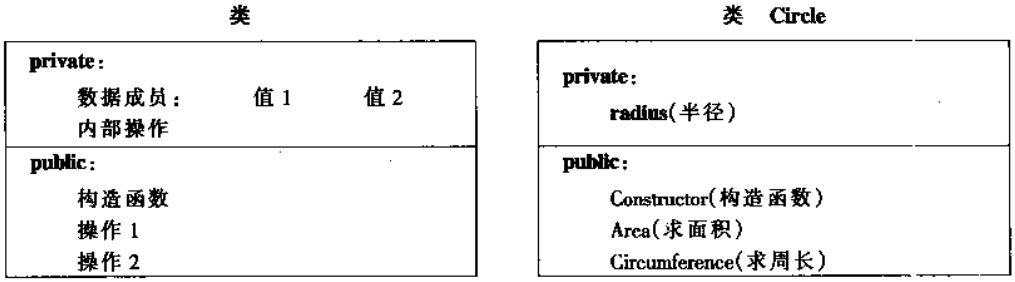
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        Initial values:      圆的半径
        Process:             给圆的半径赋初值
        Area
        Input:               无
        Preconditions:       无
        Process:             计算圆的面积
        Output:              返回面积值
        Postconditions:      无
        Circumference
        Input:               无
        Preconditions:       无
        Process:             计算圆的周长
        Output:              返回圆的周长
        Postconditions:      无
end ADT Circle
```

1.2 C++类和抽象数据类型

C++语言使用用户定义的**类**(Class)类型来表示抽象数据结构。类由多个存放数据值的**成员**和加工数据的运算组成。这些运算也称为"**方法**",因为它们定义了存取数据的方法。类型为类的变量称为**对象**。类可分为两个部分,其**公共**(public)部分描述用户使用类的界面,它使用户不必了解对象的内部细节而使用对象;其**私有**(private)部分由帮助实现数据抽象的数据和内部操作组成。例如,表示 ADT 圆的类中包含一个私有数据成员——radius(半径);其**公共**成员包括构造函数和计算面积和周长的方法。



数据封装和信息隐藏

类通过把数据和方法包装在一起并将它们视为整体来**封装**(encapsulates)信息。类在结构上隐藏了应用细节并严格限制对其数据和操作的外部访问。我们把类的这种特性称为**信息隐藏**(information hiding),它保护了数据的完整性。

类通过其私有和公共部分来控制外部应用对它的访问。私有部分的成员由类内部的方法在内部使用,与外界环境隔离。数据通常定义在类的私有部分以防止外界不必要的访问。它只提供其公共成员与外部环境打交道,并供用户使用。

例如,在类 Circle 中,半径是一私有成员,它仅供类 Circle 的 3 种方法使用。构造函数

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

可对其赋初值,其它两种方法用它来进行计算,如面积 = (π * radius^2)。这些方法是公共成员,可被外部程序调用。

消息传递

在应用中,对象的公共成员可由外部程序调用。这种调用由控制各对象相互作用的**主控模块**(主程序或子程序)来完成,控制码指挥对象用某种方法或运算访问数据。这种指挥每个对象活动的过程称为**消息传递**。消息的发送者将消息传递给接收对象请求它完成一项任务。

必要时,发送者也发送接收对象要用到的信息。这些信息作为运算的输入数据与消息一起传递。任务完成后,接收对象返回信息给发送者(输出结果)或给其它对象传递消息,要求执行其它任务。在接收对象完成运算时,它也可能修改某些内部数据值,此时,该对象发生**状态转换**,产生新的结果。

1.3 C++应用中的对象*

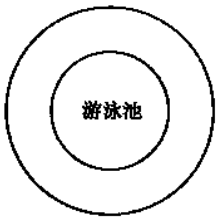
抽象数据类型给出了数据及可对数据进行的操作的综合描述。在声明 C++类时一般不定义成员函数,这叫**类声明**(class declaration),是 ADT 的一种具体表示,方法的具体定义在独立于声明之外的**类实现**(class implementation)中给出。

我们通过一个完整的程序来说明 C++类的实现和对象的应用。该程序用来计算一个圆形水池的池壁造价。程序定义了类 Circle,并给出了定义和使用对象的方法、类中公共和私有部分的定义及如何使用 C++函数来定义操作。主程序首先声明了对象,然后调用其操作来完成计算。另外,主程序还负责应用中的所有消息传递。

应用:类 Circle

我们用类 Circle 来描述一个圆形游泳池及其边上的过道。通过计算周长和面积,可以求出建造过道及围上栅栏的造价。题目如下:

一圆形游泳池如下图所示。现需在其周围建一圆形过道,并在其四周围上栅栏。栅栏价格为每英尺 3.5 美元,过道造价为每平方英尺 0.5 美元。过道宽度为 3 英尺,游泳池半径由键盘输入。要求编程计算并输出过道和栅栏的造价。



游泳池外环

我们声明对象 Pool 为游泳池本身,PoolRim 为池及其周围的过道。定义好对象后,调用构造函数为其赋初值。对 Pool 来说,主程序将读入的半径作为参数传递给它,而 PoolRim 的半径为 Pool 的半径加 3。

在对象名称后加点(.)和操作名称可调用类运算。例如,Pool.Area()计算游泳池的面

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

积，PoolRim.Circumference()计算过道的周长。

栅栏安装在过道的周围，可调用计算过道周长的运算来计算栅栏的造价。

```
FenceCost = PoolRim.Circumference() * 3.50
```

过道面积为外环面积减去游泳池面积，则其造价为：

```
ConcreteCost = (PoolRim.Area()-Pool.Area()) * 0.50
```

程序 1.1　类 Circle 的创建及使用

程序 1.1 是上述问题的具体实现。程序中提供了注释以帮助读者理解程序。类 Circle 的声明给出了 ADT Circle 的表示以及可控制对成员的访问的私有及公共调用。

主程序要求用户输入游泳池的半径，并赋值给对象 Pool，对象 PoolRim 的半径为该值加 3 英尺。最后，程序输出栅栏和过道的造价。

类 Circle 在主程序外部定义。读者也许注意到限定词 const，它限定的函数成员不改变数据成员的值，而且在函数的声明和定义中都要用到。建筑材料的价格用常量给出。

```
# include <iostream.h>
const float PI = 3.14152;
const float FencePrice = 3.50;
const float ConcretePrice = 0.50;
// 声明类 Circle 及其数据和方法
class Circle
{
    private:
        // 定义数据成员 radius 为浮点数
        float radius;
    public:
        // 构造函数
         Circle(float r);
        // 计算圆的周长和面积的函数
        float Circumference(void) const;
        float Area(void) const;
};
// 类的实现
// 构造函数用类初始化数据成员 radius
Circle::Circle(float r): radius(r)
{ }
// 计算圆的周长
float Circle::Circumference(void) const
{
    return 2 * PI * radius;
}
// 计算圆的面积
float Circle:Area(void) const
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    return PI * radius * radius;
}

void main()
{
    float radius;
    float FenceCost, ConcreteCost;
    // 设定浮点数输出时只显示小数点后两位
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    // 提示用户输入半径 radius
    cout << "Enter the radius of the pool:";
    cin >> radius;
    // 定义 Circle 对象
    Circle Pool(radius)
    Circle PoolRim(radius+3);
    // 计算栅栏造价并输出
    FenceCost = PoolRim.Circumference() * FencePrice;
    cout << "Fencing Cost is $" << FenceCost << endl;
    // 计算过道造价并输出
    ConcreteCost = (PoolRim.Area() - Pool.Area()) * ConcretePrice;
    Cout << "Concrete Cost is $" << ConcreteCost << endl;
}

/*
<程序 1.1 的运行结果>
Enter the radius of the pool: 40
Fencing Cost is $945.60
Concrete Cost is $391.12
*/
```

1.4 对象设计

本书用类和对象讲述数据结构。我们从由简单数据成员和操作所定义的类开始。为描述更复杂的结构,类可包含本身就是对象的数据成员。这种通过复合方法产生的类,可以访问组成它的对象的成员函数。对象复合扩充了数据封装和信息隐藏的概念,实现了代码复用。面向对象程序设计语言也支持从其它类通过继承派生出新的类,这就使设计者可通过细化某个类来创建新类,并复用已开发的代码。继承是面向对象程序设计语言C++的一个基本工具,我们将在第12章详细介绍,并用它来强化高级数据结构的设计和实现。

对象及其复合

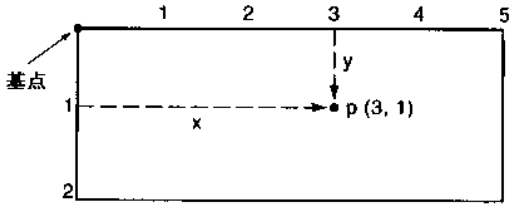
几何图形由构成线、长方形等的点集组成。点和一系列公理构成了几何学的基础。本节中,我们把点定义成为一个原始的对象以描述线和长方形,并用它们来说明对象及其复合。

点是平面上的位置。我们用点在坐标系里的坐标(x,y)来表示点这个对象,x和y分

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

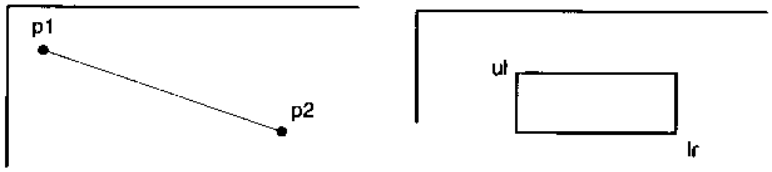
www.Linuxidc.com

别表示点到原点的水平和垂直距离。例如，点 P(3,1)表示从原点往右 3 个单位、往下 1 个单位。



线由点组成，两点决定一条直线。根据这一性质，可给出由起点 P1 和终点 P2 定义的线段的模型，如图 1.1(A)所示。

矩形是邻边正交的四边形。它也可用两点，即其左上点(UL)和右下点(LR)决定。如图 1.1(B)所示。



(A) 线段 (P1,P2)　　(B) 矩形 (ul,lr)

图 1.1　线段和矩形

我们用上述性质来定义类点(Point)、线段(Line)和矩形(Rectangle)。类线段和矩形的数据成员是点的对象。在使用其它类的对象创建类时，复合是一重要的工具。请注意每个类都定义了方法 Draw 来在平面上显示该图形。类 Point 还有取得该点坐标的两个函数 Getx，Gety。

类 Point	类 Line	类 Rectangle
private: x y <坐标值>	private: Point p1, p2	private: Point ul, lr
public: Constructor Draw	public: Constructor Draw	public: Constructor Draw

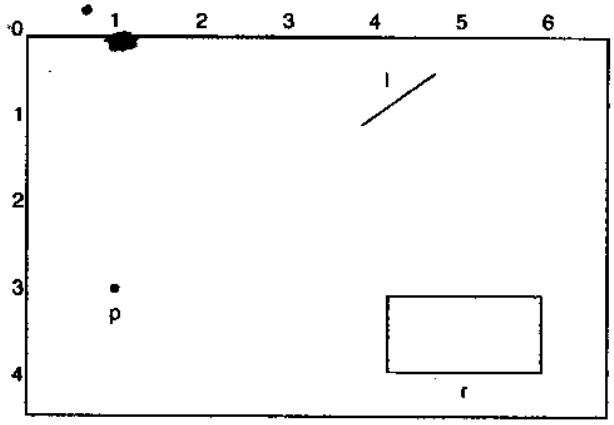
例 1.3

定义几何对象，并根据指定参数，画出几何图形。

```
1. Point p(1,3);            // 定义点 p(1,3)
2. Point p1(4,2), p2(5,1);
   Line  l(p1,p2);          // 定义线段，起点为 p1(4,2)，终点为 p2(5,1)
3. Point p1(4,3), p2(6,4);
   Rectangle  r(p1,p2);  // 定义矩形，左上点为 p1(4,3)，右下点 p2(6,4)
4. 用每个类中的方法 Draw 可在平面上画出如下图形。
   p.Draw();        l.Draw();        r.Draw();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



Draw 方法

C++的几何类*

下面是 C++对类 Point 和 Line 的定义。注意类 Line 的构造函数的参数是决定线段的两点的坐标值。每个类都有 Draw 函数，它可在作图区域上画出图形。

类 Point 的说明

声明

```
class Point
{
    private:
        float x, y;                 // 点的水平及垂直位置
    public:
        Point ( float h, float v);  // 将 h 赋值给 x,v 赋值给 y
        float GetX(void) const;     // 返回 x 座标(水平位置)
        float GetY(void) const;     // 返回 y 座标(垂直位置)
        void Draw(void) const;      // 在(x,y)处画一个点
};
```

类 Line 通过复合引用了两个 Point 的对象。两点均由构造函数初始化。

类 Line 的说明

声明

```
class Line
{
    private:
        Point   p1,   p2;           // 线段的两个端点
    public:
        Line (Point a, Point b);    // 将 a 赋值给 p1,b 赋值给 p2
        void Draw(void) const;      // 画出该线段
};
```

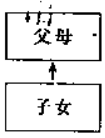

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

757698

对象和继承

继承是一个直观概念，现实生活中存在许多继承的例子。我们每个人都从父母身上继承了诸如人种、肤色及眼睛颜色等特征，我们也可以从这些特征上想象父母的大致情况。这种关系可用如下两个对象的图示表示。



动物学对动物种属的继承作了更深入的研究。图 1.2 给出了一种动物的种属图，从哺乳动物到狗再到苏格兰牧羊长毛狗。哺乳动物是热血动物，有毛，且用母乳哺育幼仔。狗是食肉、具有特定骨架、群居的犬齿类哺乳动物，而苏格兰牧羊长毛狗是有长鼻子、带红白斑、并具有放牧本能的狗。

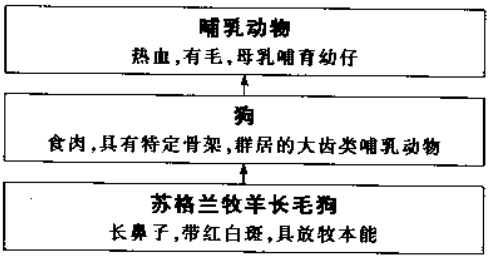


图 1.2　动物继承链

在种属链中，子类继承其父类的所有属性。如狗具有所有哺乳动物的属性，也具有将它与猫、象等区分开的属性。图 1.2 的从底到顶的继承顺序可用如下方式表示：

苏格兰牧羊长毛狗"是"(is a)狗，　　狗"是"(is a)哺乳动物。

在链中，"哺乳动物"是"狗"的**基类**，"狗"被称为**派生类**。用动物学中的概念，我们也可将基类和派生类分别称为父类和子类，并可得知，子类可从父类和祖父类中继承属性。

程序设计中的继承

面向对象的程序设计提供使派生类可以从基类继承数据和操作的机制，我们把这种机制称为**类继承**。它允许派生类使用在基类中预先定义的数据和操作。派生类还可增加新的操作或重写基类的某些操作来建立自己处理数据的方法，就如孩子从父母处继承一幢房子或一辆汽车后，可以使用房子和汽车，也可以将房子改建以适应自己的生活环境。

我们用依顺序次序存储信息的线性表 SeqList 来说明类继承，它是一种重要而又常见的数据结构，经常用来表示诸如库存记录、日程安排、供货的类型和数量等信息。(我们定义另外一种按元素大小排列的表为有序表，)有序表继承了线性表的所有基本操作，并提供自己的插入操作，以保证其元素按升序排列。

在一有 N 个元素的线性表中，元素占据从 0 至 N-1 的位置。第一个位置是表头，最后一个位置是表尾。图 1.3 给出了一个有 6 个整数的无序表。

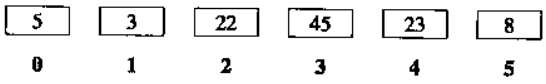


图 1.3　无序的线性表

SeqList 的基本操作包括 Insert(插入)——即在表尾增加一个新元素(图 1.4)和 Delete

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(删除),即删除表中第一个和键相匹配的元素。另一个删除函数 DeleteFront,即删除表中第一个元素(图 1.5)。该结构用 ListSize 表示表的大小,并提供操作 Find 来查找表中的元素。为便于管理数据,还提供了用以测试表是否为空表和用来删除整个表的操作 ClearList。

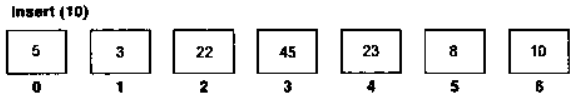


图 1.4 插入元素 10

SeqList 提供操作 GetData 让用户访问表中某个指定位置的元素。例如,求表中的最大值,可从位置 0 开始扫描表直到 ListSize,在每个位置,如果当前值(由 GetData 得到)大于当前最大值,则修改当前最大值为当前值。如在位置 2,当前值为 22,当前最大值为 3,则将

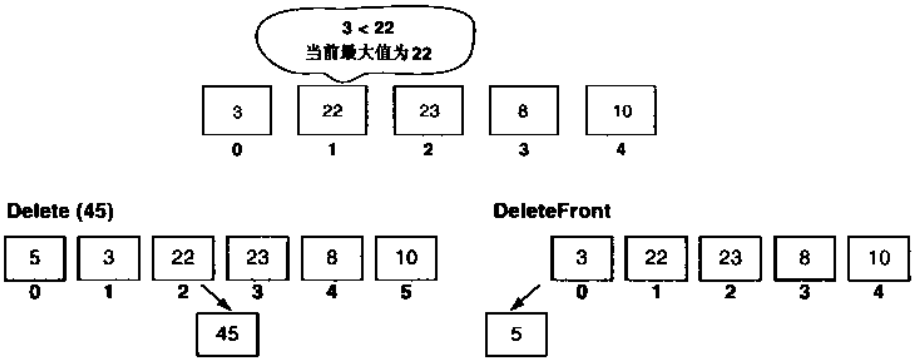


图 1.5 删除元素 45 和删除表头

当前最大值改为 22。最后,可得出 23 是该表的最大值。

ADT SeqList **is**

Data

表示表中当前所含元素个数(表的大小)的非负整数;
各数据元素组成的表。

Operations

Constructor

Initial values: 无
Process: 将表的大小置为 0

ListSize

Input: 无
Preconditions: 无
Process: 得到表的大小
Output: 表的大小
Postconditions: 无

ListEmpty

Input: 无
Preconditions: 无
Process: 检查表的大小
Output: 若表为空表(其大小为 0),则返回 TRUE;否则,返回 FALSE
Postconditions: 无

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

ClearList

Input:	无
Preconditions:	无
Process:	删除表中所有元素并将表的大小置为 0
Output:	无
Postconditions:	表成为空表

Find

Input:	要在表中查找的元素
Preconditions:	无
Process:	为匹配元素扫描表
Output:	若匹配失败,则返回 FALSE;否则,返回 TRUE 和第一次匹配成功的元素
Postconditions:	无

Insert

Input:	要插入表中的元素
Preconditions:	无
Process:	将元素加在表尾
Output:	无
Postconditions:	表增加了一个新元素,其大小加 1

Delete

Input:	要从表中删除的"值"
Preconditions:	无
Process:	扫描表并从表中删除第一个和值相等的元素 若不存在和值相等的元素,则不做任何操作
Output:	无
Postconditions:	如果存在等值元素,则表中元素减少一个,且其大小减 1

DeleteFront

Input:	无
Preconditions:	表必须非空
Process:	删除表中第一个元素
Output:	返回被删除的元素值
Postconditions	表中元素少了一个且表的大小减 1

GetData

Input:	表中的位置(pos)
Preconditions:	若 pos 小于 0 或大于表的大小 -1,则出错
Process:	获取表中 pos 位置处的元素值
Output:	位于位置 pos 处的元素值
Postconditions:	无

end ADT SeqList

有序表和继承

有序表是一种特殊的表,其元素按升序排列。作为抽象数据类型,这种特殊表保留了 SeqList 中除 Insert 之外的绝大部分操作,它的插入必须维持表的升序排列(图 1 6 所示)。

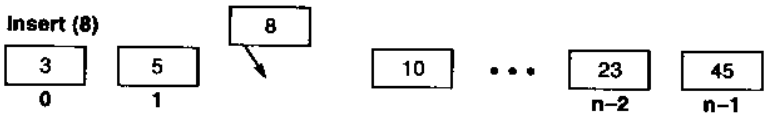


图 1.6 有序表中插入 8

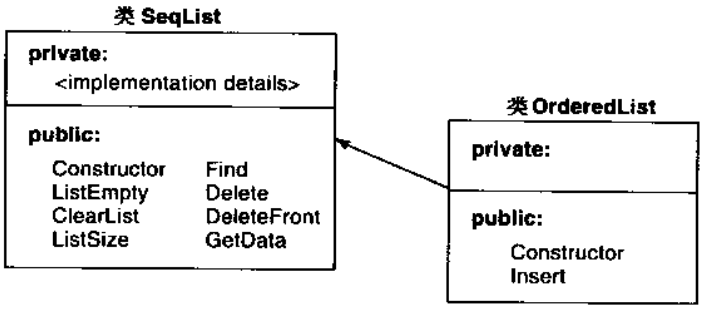
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

操作 ListSize，ListEmpty，ClearList，Find，GetData 与表中元素顺序无关，操作 Delete 和 DeleteFront 删除表中元素但不影响排列顺序。下述 ADT 反映了有序表 OrderedLis. 和 SeqList 操作的相似性。

```
ADT  OrderedList  is
   Data
     < 同 ADT SeqList >
   Operations
       Constructor          < 执行基类的 Constructor >
       ListSize             < 同 ADT SeqList >
       ListEmpty            < 同 ADT SeqList >
       ClearList            < 同 ADT SeqList >
       Find                 < 同 ADT SeqList >
       Delete               < 同 ADT SeqList >
       DeleteFront          < 同 ADT SeqList >
       GetData              < 同 ADT SeqList >
     Insert
       Preconditions:       无
       Input:               要插入表中的元素。
       Process:             在表中可保持元素有序的位置插入该元素。
       Output:              无
       Postconditions:      表增加一个新元素且其大小加 1。
end ADT OrderedList
```

类 OrderedList 是从类 SeqList 派生的，它继承了基类的许多操作，只是将其 Insert 操作改写为按有序的次序插入元素。



软件复用

数据结构的面向对象方法提供了对代码的复用，即可将过去开发和测试过的代码"植入"新的应用中。我们在类的复用中已学习过代码的复用，继承也是实现这一目的的有力工具。例如，实现类 OrderedList 仅需我们写构造函数和 Insert 的代码，所有其它操作都可从类 SeqList 中得到。代码复用是面向对象方法的巨大优势，它可节约软件开发时间，并提高了应用和系统之间的整体性。例如，在系统升级时，为使应用能继续运行，可将原操作系统定义为基类，升级后的系统作为具有新功能的派生类运行。

类 SeqList 和 OrderedList 的说明*

我们将在第 4 章深入讨论类 SeqList，本节我们给出它的说明，以使读者可将类及其方

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

法与一般 ADT 联系起来。同时,我们定义类 OrderedList 来说明类继承。表中元素的类型为 DataType,可以是用户定义的任何类型。

类 SeqList 的说明

声明

```
class SeqList
{
    private:
    // 存放表的数组及表中当前元素的个数
      DataType listitem[ARRAYSIZE];
      int size;
    public:
    // 构造函数
      SeqList(void);
    // 访问表的方法
      int ListSize(void) const;
      int ListEmpty(void) const;
      int Find (DataType& item) const;
      DataType GetData (int pos) const;
    // 修改表的方法
      void Insert (const DataType& item);
      void Delete (const DataType& item);
      DataType DeleteFront (void);
      void ClearList(void);
}
```

说明

在函数 ListSize,ListEmpty,Find 和 GetData 的定义后面都有单词"const",这些函数叫常量函数,因为它们并不改变表的状态。函数 Insert,Delete 在参数表中出现单词"const",C++用这种方法来表示虽然传递的是参数的地址,但并不允许改变参数值。

C++声明派生类的语法很简单。在头部,类名的后面用冒号(:)表示其基类。下面是类 OrderedList 的声明。详情将在第 12 章讲述继承时讨论。

类 OrderedList 的说明

声明

```
class OrderedList: public SeqList            // 从类 Seqlist 中继承
{
    public:
        OrderedList(void);                   // 初始化基类来创建一个空表
        void Insert ( const dataType& item); // 按保序往表中插入元素
};
```

说明

函数 Insert 覆盖了基类中的同名函数。它遍历从基类中继承的表,将元素插入到可保持其升序排列的位置。

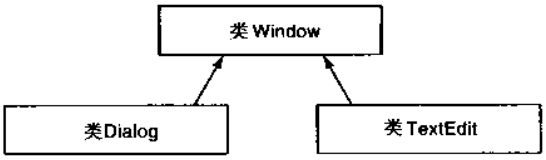
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

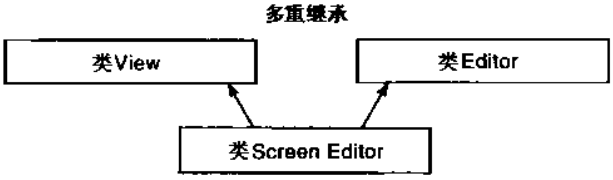
1.5 类继承的应用

类继承概念在图形用户界面(GUI)程序设计和数据库系统中有着重要的应用。图形上的应用集中在诸如窗口、菜单、对话框等对象上,最基本的窗口是一具有适用于所有类型窗口的数据和操作的数据结构。这些操作包括打开窗口、创建或修改窗口标题,建立卷滚条和拖动区等。其它的窗口,如对话框、菜单、文本窗口等可继承这些基本结构。GUI的应用实际上就由这些窗口组成。例如,下图中的类 Dialog 和 TextEdit 即是从类 Window 中派生出来的。

图 1.7 给出了一个包含对话框和文本框的 GUI 应用。



上述有关窗口的例子给出的都是单重继承,即一个派生类有且仅有一个基类,对**多重继承**来说,派生类来自两个或更多个基类。某些 GUI 应用就用到了这个性质。例如,字处理程序包括编辑器(Editor)和用来在窗口内显示文本的浏览器(View)。编辑器读入字符流并通过插入和删除字符串和相应的格式信息来改变字符流;浏览器则负责将文本用给定字体和窗口信息显示在屏幕上。这样,屏幕编辑器(Screen Editor)可定义为以类 Editor 和 View 为基类的派生类。



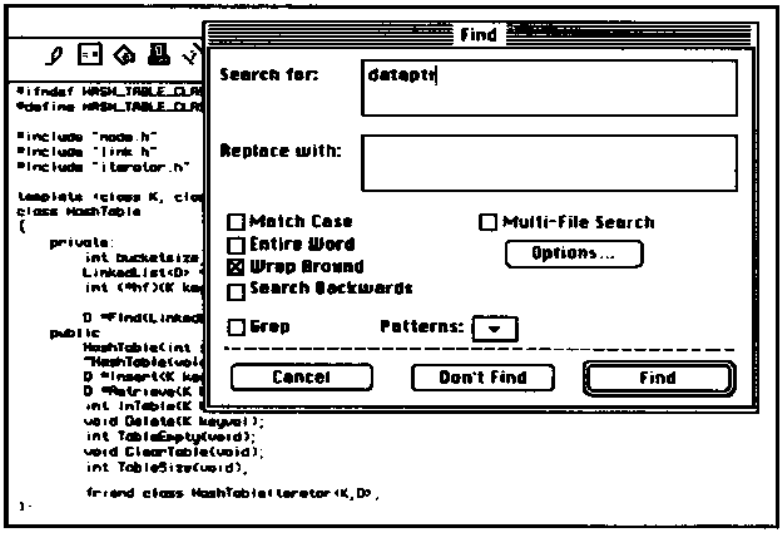


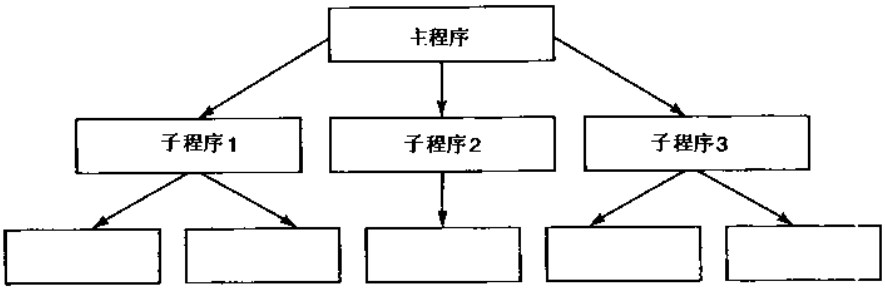
图 1.7 GUI 窗口应用

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

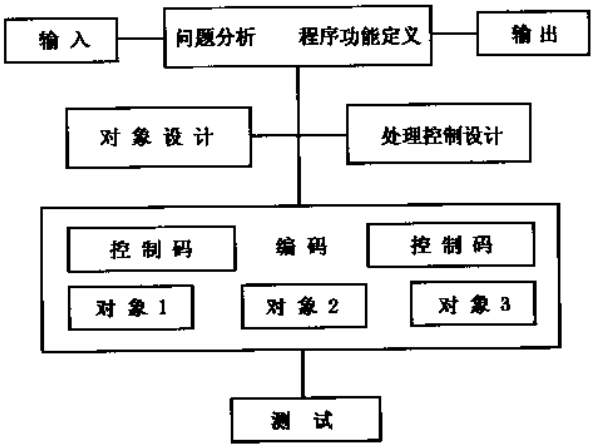
1.6 面向对象程序设计

大型软件系统复杂度的不断增加，对软件系统设计方法也提出了新的要求。传统的模块化设计方法要求有一位软件负责人，他了解整个系统并负责分解任务。这种自顶向下的设计方法将系统视为分层的子程序集合。在最顶层，主程序通过顺序调用子程序来完成计算并返回信息。子程序也可进一步分解为完成更小任务的子程序。显然，自顶向下设计的精髓对所有系统都是必需的，但当问题的复杂性超出个人能够弄清楚系统调用关系的能力之外时，自顶向下的方法可能失败。而且，上层子程序的简单改动，即可造成底层程序代价高昂的修改。



面向对象程序设计提出了一种新的系统设计模型。它将系统看成通过交互作用来完成任务的对象的集合，每个对象用自己的方法管理数据。

程序设计应给出一个具有良好的可读性和可维护性的体系结构，当然也须具有良好的可扩充性。组织得好的系统应是易于理解、开发和排错的系统。所有的设计方法都试图通过分解和控制的原则来控制软件系统的复杂性。自顶向下程序设计方法将系统视为函数模块的层次集合，面向对象程序设计方法以对象作为设计的基础。软件设计并不存在万能的规则，它应该是包含创作自由和灵活性的人类活动。本书中，我们讨论一种常用的**软件开发方法**，它将软件开发过程划分为明显的阶段，即问题分析和功能定义、对象及其处理设计、编码、测试和维护。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

问题分析和功能定义

程序开发从用户提出需解决的问题开始，这些问题常常并不明确。对系统可使用的数据(输入)和应提供的信息(输出)都没有精确的定义。在本阶段，程序员和用户共同分析问题，确定输入和输出数据及其格式，设计计算的算法。本阶段应在程序设计之前完成。

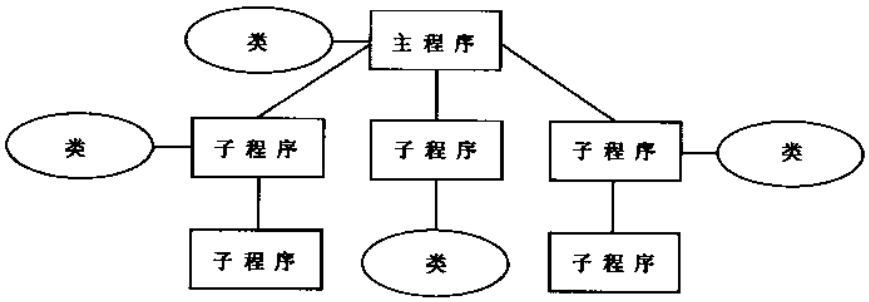
设计

程序设计阶段应给出所有对象的描述，这是构成程序的主要部分。同时，设计阶段也描述对象间相互作用的控制。

对象设计阶段明确所有在程序中将用到的对象，并给出每个类的定义，设计出一些小程序对类进行测试。类可独立于系统之外测试是面向对象程序设计的一大特色。

处理控制设计阶段完成程序的框架设计，它使用自顶向下方法创建主程序和子程序来控制对象间的相互作用。

主控模块对应于 C++程序中的 main 函数，它控制程序的数据流。通过自顶向下程序设计方法，可将系统分解为顺序执行的子程序。主程序和子程序可用结构树来描述其模块间的自顶向下调用关系。主模块为树的根。每个模块用矩形表示，模块用到的类用椭圆表示。对每个模块，分别列出其函数名、输入和输出参数，并对其处理加以简单的描述。



函数名

输入参数：　　<输入参数描述>

处理：　　<函数处理算法描述>

返回参数：　　<列出函数的返回值>

编码

本阶段完成实现程序设计的主程序和子程序的编码。

测试

由于对象设计阶段已完成对象的实现和测试，测试阶段的注意力应集中在对控制模块的设计上。我们要通过控制模块测试每个对象的相互作用来验证程序的编码。

程序设计举例：点数图

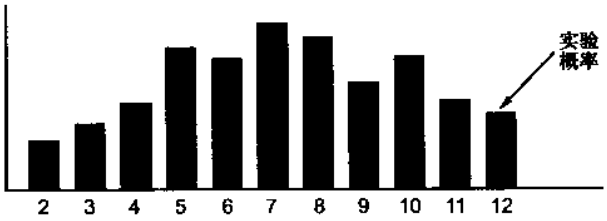
我们用记录骰子点数的图为例来说明面向对象程序设计。下面我们分别描述该程序生命周期的各个阶段。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

问题分析 假定事件为投掷两个骰子。每次投掷得到的点数之和为 2 至 12 之间。重复投掷，我们可得到点数和分别为 2,3,…,11,12 的实验概率，并画出用柱图表示的每个可能点数的概率图。

注意： 实验概率用随机数发生器来模拟，对应于每个点数的随机数发生的几率表示实验概率。例如，若重复投掷 100,000 次骰子，点数和为 4 的情况发生了 10,000 次，则掷出 4 点的实验概率是 0.10。



只有在问题明确之后，才有可能得到有效的解决办法。解决办法应建立在对输入数据、输出数据和中间计算的正确理解之上。在问题分析阶段，用户对系统提出一系列的要求，包括输入数据的控制，给出要用到的算法和公式，并描述预期的输出。

程序定义 程序首先请求用户输入投掷骰子的次数 N。由于掷骰子是一个随机输出过程，我们用随机数发生器来模拟 N 次投掷的结果。程序对每种投掷的结果 S(2 < = S < = 12)保留一个次数记录。S/N 即为每种投掷结果的实验概率，输出时用它来确定图形中柱的高度。这样，就得到一个用柱图输出的结果。

对象设计 程序用类 Line 来创建图形的坐标轴，用类 Rectangle 来显示每个柱。这些类都已在 1.4 节中介绍过，而掷骰子可用类 Dice 中处理两个骰子的方法。下面我们给出类 Dice 的定义。它的实现和测试随类 Line 和类 Rectangle 的实现和测试一起由程序 1.2 给出。

```
#include "random.h"
class Dice
  {
  private:
      // 数据成员
         int diceTotal;          // 两个骰子的点数和
         int diceList[2];        // 两个骰子各自的点数
      // 用随机数发生器模拟掷骰子
         RandomNumber rnd;
  public:
      // 构造函数
         Dice(void);
      // 处理掷骰子进程的方法
         void Toss(void);
         int Total(void) const;
         void DisplayToss(void) const;
  };
```

过程控制设计 主程序调用三个子程序来实现点数图。函数 SimulateDieToss 用类中

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

的方法掷 N 次骰子，DrawAxe 调用类 Line 的方法 Draw 来画图形的坐标轴，Plot 通过画矩形来产生柱图。为计算柱图中每个柱的相对高度，Plot 函数调用 MAX 来确定所有发生的情况中的最大次数。图 1.8 给出了本程序的结构树。下面是结构树中每个控制模块的描述。

Main

传递参数：

无

过程：

提示用户输入模拟投掷骰子的次数。调用函数 SimulateDieToss 来完成投掷并记录每种结果(2≤Total≤12)发生的次数。调用函数 DrawAxe 画坐标轴，再调用函数 Plot 产生柱图。

返回参数：

无

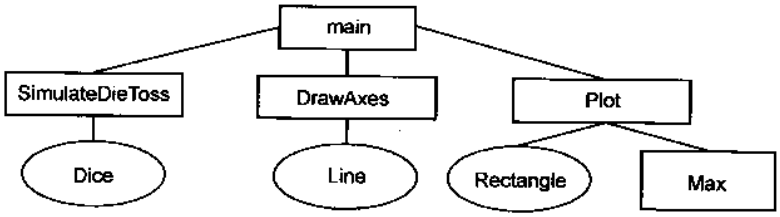


图 1.8　点数图程序的结构树

模块名：SimulateDieToss

模块参数：

tossTotal　　数组 tossTotal 记录每种投掷结果发生的次数。当 2 < = i < = 12 时，tossTotal[i]为投掷出点数和为 i 的次数。

tossCount　　模拟投掷的次数。

处理过程：

创建对象 Dice，用它来掷指定次数的骰子，并将每种结果发生的次数记录在数组 tossTotal 的相应元素中。

返回参数：

记录有每种结果发生次数的数组 tossTotal。

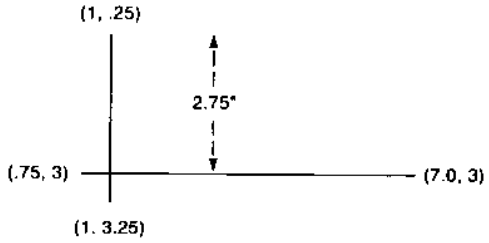
模块名：DrawAxe

模块参数：

无

处理过程：

创建两个 Line 型对象，一个为垂直坐标轴(y 轴)，另一个为水平坐标轴(x 轴)。y 轴起始点坐标为(1.0,3.25)，终止点坐标为(1.0,0.25)，x 轴为(0.75,3.0)至(7.0,3.0)。柱的最高高度为 2.75。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

返回参数：
无

模块名：Max

模块参数：

a	存放 long 型数据的数组
n	数组 a 中的数据个数

处理过程：
取得数组 a 中元素的最大值

返回参数：
数组中的最大值

模块名：Plot

模块参数：

tossTotal	存放每种结果出现次数的数组，由函数 SimulateDieToss 得到

处理过程：
取得数组 tossTotal 中下标从 2 至 12 的元素的最大值(maxTotal)，这是出现最多的结果。然后，用数组中每个元素除以该值(tossTotal[i]/maxTotal)得到柱图的相对高度。将 x 轴的长度 6 等分为 23 等份，然后在相应位置上画出矩形，每个矩形的高度为 tossTotal[i]/maxTotal * 2.75，其中 2 < = i < = 12。

返回参数：
无

编码*：将控制模块编码实现程序的功能。程序 1.2 可实现点数图程序。

程序 1.2　点数图

主程序请求用户输入模拟投掷的次数。我们给出投掷 500 000 次的运行结果。程序结束前等待用户键盘输入或单击鼠标来结束程序。类 Line 和 Rectangle 的定义在文件“figures.h”中，类 Dice 的定义在文件“dice.h”中。图形例程可在“graphlib.h”中找到。

```
# include <iostream.h>
# include "figures.h"
# include "dice.h"
# include "grahlib.h"
// 将两个骰子掷 tossCount 次，掷得点数为 2 的次数存入 tossTotal[2]中，
// 点数为 3 的次数存入 tossTotal[3]，……
void SimulateDieToss(long tossTotal[],long tossCount)
{
    long tossEvent;
    int i;
    Dice D;
    // 将点数记录清零
    for (i = 2;i < = 12;i + + )
        tossTotal[i] = 0;
    // 掷骰子 tossCount 次
    for (tossEvent = 1;tossEvent < = tossCount;tossEvent + + )
    {
      D.Toss()                          // 掷骰子
      tossTotal[D.Total()] + + ;        // 将对立点数的计数器加 1
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

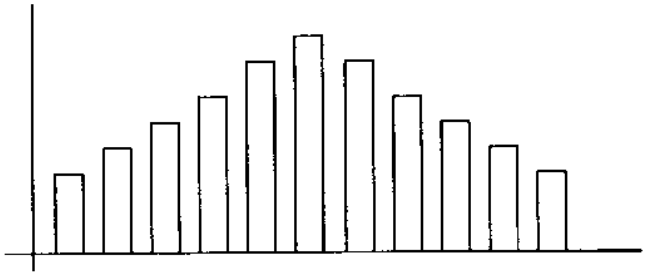
www.Linuxidc.com

```
    }
}
// 在 n 元数组 a 中找最大值
long Max(long a[], int n)
{
    long lmax = a[0];                    // 保存最大值的变量
    int i;
    for (i = 1;i < n;i + + )
        if (a[i] > lmax)
            lmax = a[i];
    return lamx;
}
// 画两根轴线
void DrawAxes(void)
{
    const float verspan = 3.0;
    Line VerticalAxis(Point(1.0 vertspan + 0.25),Point(1.0,0.25));
    VerticalAxis.Draw();
    Line HorizontalAxis(Point(0.75,vertspan),Point(7.0,vert-span));
    HorizontalAxis.Draw();
}
    // 画柱图
void Plot(long tossTotal[])
{
    const float vertspan = 3.0,scaleHeight = 2.75;
    float x, rectHeight,dx;
    long mxTotal;
    int i
    // 在数组 tossTotal 中找最大值(仅限元素 2-12 中)
    maxTotal = Max(&tossTotal[2],11);
    // 生成矩形
    dx = 6.0/23.0;
    x = 1.0 + dx
    // 循环 11 次,每次循环决定当前柱高,并在相应位置、高度、宽度创建一个矩形对象,
    // 画出该柱并移到下一位置
    for (i = 2;i < = 12;i + + )
    {
      rectHeight = (float(tossTotal[i]/maxTotal) * scaleHeight;
      Rectangle CurrRect(Point(x,vertspan-rectHeight),
                         Point(x + dx,verspan));
      CurrRect.Draw();
      x + = 2.0 * dx;
    }
}
    void main(void)
{
    long numTosses;
    long tossTotal[13];
    // 提示用户输入骰子次数
    cout < < "Enter the number of tosses: ";
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    cin > > numTosses;
    SimulateDieToss(tossTotal,numTosses);      // 掷骰子
    InitGraphics();                            // 初始化图表系统
    DrawAxes();                                // 画轴线
    Plot(tossTotal);                           // 画柱图
    ViewPause();                               // 等待按键或按鼠标
    ShutdownGraphics();                        // 关闭图形系统
}

    /*
<程序 1.2 运行结果>
    Enter the numer of tosses: 500000
*/
```



1.7 程序测试与维护

使用面向对象程序设计方法产生的系统，可对对象进行独立测试，也可复用已经写好的类。由于整个系统是从我们有充分信心的小系统逐步扩充而成的，这就降低了构建复杂软件系统的出错风险。测试贯穿在整个软件系统的开发过程中。

对象测试

类类型是一自包含的数据结构，它能和外部程序块双向交换信息。我们可用开发小程序调用类的每个公共方法来对类进行测试。

控制模块测试

程序必须用精心选择的测试数据进行彻底的测试。在这之前，程序员可通过给其他程序员讲解整个程序的设计和实现过程来验证程序的正确性。这个过程经常可发现错误概念、澄清程序的逻辑过程，并给出对测试的建议。

现在的编译器支持源程序级排错，允许跟踪单条指令，并在给定的断点暂停。在这个受控执行过程中，可显示变量值，允许在错误发生前后比较“程序快照”。

程序最终的正确性测试应是用精心选择的测试用例运行程序。通过测试，程序员可获得对程序正确性的信心。程序员还应该用不正确的输入来测试代码的“健壮性”，这是对程序识别错误数据并返回错误信息的能力的测试。

测试数据的选择方法多种多样，一种是不值得提倡的“凑合法”，即程序员只用简单的

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

数据运行几遍程序,便认为程序正确。更合理一些的方法是选择可测试程序不同算法的一系列输入数据。这些数据应包括简单数据、典型数据、测试程序的特殊情况的特殊数据和检验程序响应错误输入能力的非法数据。

完整的结构化测试关系到程序的逻辑结构。这种方法认为:如果程序还有代码没被执行到,就没有达到完全测试。它要求程序员选择数据测试程序中的不同算法:每个条件语句、每个循环结构、每个函数调用,等等。某些编译器还可提供程序执行期间每个函数执行情况的记录。

程序维护和文档

为满足不断增加的需求,经常需要对程序进行修改。面向对象的程序设计方法简化了程序维护工作,类继承允许对现存软件进行复用。但这些优势只有在具备良好的设计文档时才能更好地发挥出来。文档中对类和控制模块的描述可帮助用户正确理解程序及对程序的正确使用。大的软件系统通常还有用户手册、安装手册等一本或多本手册对其重要特性进行说明。

对象说明和控制模块结构图是极好的程序文档工具。在程序的源代码中,可用注释来解释每个函数和类的功能。当然,在算法的微妙之处也应加上注释进行说明。

1.8 C++程序设计语言

本书使用面向对象程序设计语言C++为读者介绍数据结构。虽然目前有多种面向对象程序设计语言,但C++由于起源于广泛使用的C语言和具有数量很多的编译器而在这些语言中占有优势。

C语言作为用于系统程序设计的结构化语言,最早开发于70年代早期。它既能调用低级的系统例程,又具备高级语言的程序结构。几年以后,大多数计算机平台都有了高速又高效的C编译器。UNIX操作系统的绝大部分由C语言写成,C语言也是UNIX环境的主力语言。C++语言是由Bell实验室的Bjarne Stroustrup作为对C语言的扩充而开发的。以C语言为基础意味着C++语言不用从头开发,并拥有一批数量庞大的程序员。C++最初被称为"带类的C",并在80年代早期提供给用户使用。在调用C编译器来产生机器代码之前,先用翻译器将带类的C语言源程序翻译成"纯粹"的C语言源程序。

C++这个名称是1983年由Rich Mascitti新创的。使用C的增量算符"++"来反映该语言来自C语言,又是对C语言的扩充。许多人问C++是否应和C语言兼容,尤其是在C++扩充了许多不用C表示的结构和例程之后。事实上,C++应继续保持其扩充的C语言的身份。现存的C语言开发的软件和C的库函数将使C语言和C++语言的联系更加紧密,C++的定义保证了普通C和C++的兼容性。

70年代时许多C++结构的思想来自Simula 67和Algol 68。这两种语言引入了严格的类型检查、类的概念和模块化思想。美国国防部据此设计的Ada语言,在编译器中实现了这些关键优势。Ada通过生成方法可生成类的实例,C++则用模板结构实现了同样的目的。

C++的流行及其在许多平台上的移植呼唤着C++标准的制定。AT&T承担了此项

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

工作，它联合 C++ 编译器的作者、C 语言程序员和用户共同制定 C++ 的标准。AT&T 根据其在 UNIX 上的成功经验，协调用户意见，公布了 ANSI C++ 标准。预计 ANSI(美国国家标准局)C++ 标准将成为 ISO(国际标准组织)C++ 标准的一部分。

1.9 抽象基类及多态性*

类继承和抽象基类相结合，就产生了一个重要的数据结构工具。这些抽象基类给出了独立于类数据及其内部操作的与用户的公共接口。类的公共接口定义了存取其数据的方法。尽管类的内部实现改变后，用户也可保持原有的公共接口不变。面向对象的语言通过定义每个公共操作的名称和参数的基类来实现这项功能。抽象基类仅提供有限的实现细节，而将注意力集中在公共操作的定义上。C++ 抽象基类将某些操作定义为纯粹的虚函数。下面我们给出抽象基类 List 的定义，它给出了一些表的操作。关键字"virtual"及将操作赋值为 0 标识了纯粹的虚函数。

```
template <class T>
class List
{
    protected:
        // 表中元素个数。由派生类修改
           int size;
    public:
        // 构造函数
           List(void);
        // 访问表的方法
           virtual int ListSize(void) const;
           virtual int ListEmpty(void) const;
           virtual int Find (T& item) = 0 ;
        // 修改表的方法
           virtual void Insert( const T& item ) = 0 ;
           virtual void Delete( const T& item ) = 0 ;
           virtual void ClearList( void ) = 0 ;
};
```

该抽象基类着重描述表的最基本的性质。在后述章节中，我们以它为基础描述了一系列的数据集类(表结构)。以该抽象类为基类要求这些数据集实现类 List 的最基本的操作。本书在第 12 章以从类 List 中派生出的类 SeqList 为例来详细说明这一过程。

多态性和动态绑定

C++ 中继承可通过多种方式实现。上面已介绍了用抽象基类的虚函数实现的方式。虚函数还可通过允许继承族谱中两个或多个对象定义名称相同但完成不同任务的函数的方式来支持继承。我们把面向对象的这种特性称为**多态性**，它允许不同类的对象响应相同的消息。接受消息的对象在运行时动态确定。例如，在多系统环境中，系统管理员可利用多态性来进行文件备份。假设该系统管理员有两个磁带设备，一个是 1/2"磁带，另一个为 1/4"磁带，我们用派生于同一基类 Tape 的两个类 HalfTape 和 QuarterTape 分别管理这两个设备。类 Tape 有虚操作 Backup，它定义了磁带驱动器的公共操作。派生类根据不

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

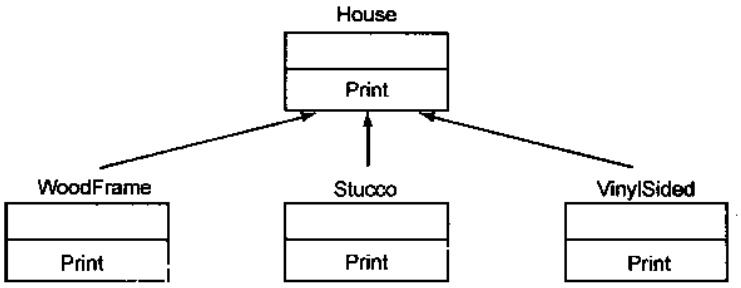
同的磁带类型来实现不同的 Backup 操作。系统管理员备份时，两个驱动器可对相应硬件的相应 Backup 消息执行各自的操作。HalfTape 的对象执行到 1/2"磁带的备份，QuarterTape 的对象执行到 1/4"磁带的备份。

多态性这一概念是面向对象程序设计的基础。专业人员经常将面向对象程序设计说成"实时多态下的继承"，C++通过**动态绑定**和**虚函数**来支持这种结构，这些概念在第 12 章中讨论。现在，我们把注意力集中到不带程序设计细节的概念上。

动态绑定允许系统中不同对象用自己的方式响应同一消息。这正如专业油漆师油漆不同的房子。我们用类 House 来描述房子，除了相同的任务外，油漆不同类型的房子需要不同的技术。油漆木结构的房子和油漆水泥结构或塑料结构的房子是有区别的。在面向对象程序设计中，对不同类型房子的油漆工作由继承同一基类 House 的不同派生类来完成，每个派生类都有各自的 Paint 操作。类 House 的 Paint 操作以虚函数的形式给出。假定对象 BigWoody 是 WoodFrame 类型，那我们可通过显式调用它的 Paint 操作来直接完成对木结构房子的油漆，我们称这种方式为**静态绑定**。

```
BigWoody.Paint();          // 静态绑定
```

然而，如果油漆队的队长手中有一张需要油漆的房子的地址表，他安排他的队员分别去油漆这些房子。此时，他给出的消息并没有和具体的房子联系起来，只给出了这些房子的地址，队员找到房子后再根据房子的类型选择正确的 Paint 操作。这种方式就是**动态绑定**方式。如下所示：




```
(位于地址 414 的房子).Paint();        // 动态绑定
```

即进程根据给定地址的房子来调用相应的 Paint 操作。如果在地址 414 的房子是木结构，则执行类 WoodFrame 的操作 Paint()。

在 C++中使用继承结构时，动态加载到对象的操作均定义为**虚成员函数**。首先生成一段代码，它创建一张表来指定对象的虚函数的位置。然后将对象和这张表联系起来，程序运行时，系统根据具体的对象，再查找这张表即可执行正确的函数。

多态概念是面向对象程序设计的基础，在许多数据结构中，我们都用到这个概念。

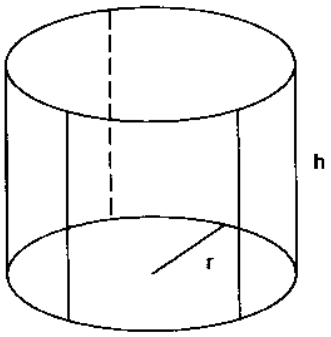
书面作业

1.1 说明数据的封装性和数据隐藏性的区别。

1.2 (a) 设计一个 ADT 来说明圆柱。其数据包括圆柱的半径和高度；操作包括构造函数，求面积函数和求体积函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



(b) 设计一个 ADT 来说明电视机。数据包括音量和频道;操作包括开、关电视、调节音量和转换电视频道。

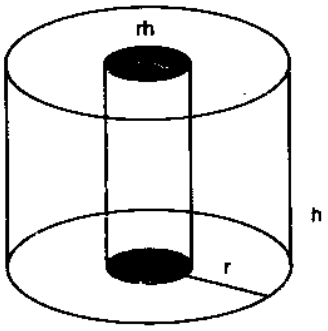
(c) 设计一个 ADT 来说明球。数据包括球的半径和重量(单位:磅);操作包括返回球的半径和重量。

(d) 设计 ADT 来说明例 1.1 的第 2 部分。

1.3 一个空心圆环,其外圆半径为 r,内圆半径为 rh,高为 h。

(a) 用 1.2 (a) 设计的圆柱 ADT 求圆环的体积。

(b) 用本书中设计的圆 ADT 来计算圆环的表面积(提示:包括内环的表面积)。



1.4 为习题 1.2 (b) 的电视机 ADT 给出几条消息。并说明每条消息的接受者应完成什么动作。

1.5 设计一个类 Cylinder 来实现习题 1.2 (a) 的 ADT。

1.6 画出包括下列各项目的继承链: 运输工具、汽车、卡车、敞篷车、福特、挂车。

1.7 说明软件开发的一般流程。

1.8 给出 C++广为流行的三个原因。

1.9 说明 C++与 C 的关系。

1.10 给出一些常用的 C++编译器。哪些你曾经用过?指出这些编译器是否提供集成开发环境。

1.11 (a) 解释多态的意义。

(b) 某图形系统在基类 TWindow 中封装了窗口操作,其派生类实现了主程序窗口、对话框和控制。每个类都有初始化窗口中各成员的操作 SetupWindow。可以在基类中说明 SetupWindow 来使用多态性吗?

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第2章　基本数据类型

2.1　整型
2.2　字符类型
2.3　实数类型
2.4　枚举类型
2.5　指针
2.6　数组类型
2.7　文本串及变量
2.8　记录
2.9　文件
2.10　数组和记录的应用
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

本章介绍一些基本数据类型，包括数值、字符、用户定义的枚举型和指针。大多数程序设计语言都提供这些类型。每种基本数据类型都含有数据和操作，即抽象数据结构(ADT)的组件。本章我们给出了整型、字符型、实型、枚举型和指针类型的 ADT。程序设计语言通过不同的方式——包括二进制数和 ASCII 字符集表现的数据在计算机内实现 ADT。本书使用 C++语言来表示 ADT 的实现。

由于数、字符和浮点类型的对象无法再分为更简单的部分，我们称其为简单数据。反之，结构化数据类型由简单类型的元素及定义元素之间关系的规则组成，这些类型包括数组、字符串、记录、文件、表、栈、队列、树、图和表格，它们构成了本书的主要话题。多数程序设计语言提供程序结构或库函数来处理数组、字符串、记录和文件结构，因此我们称其为嵌入式结构化数据类型，并给出它们的抽象数据类型，讨论它们的 C++实现。

本章中，我们还将介绍嵌入式结构化类型的重要的顺序查询及交换排序算法，给出用 C++串的库函数实现的 C++串应用的例子。C++用继承族谱来实现文件，我们给出处理三种不同文件类型的应用。

2.1 整型

整数是由符号和一个或多个数字组成的正数或负数，又称为带符号的自然数。例如，以下是称为整型常量的一些整数值：

+35　-278　19(符号为+)　-28 976 510

用初等算术中定义的运算符可产生新的整数值，运算符带上一个操作数(单目操作符)或两个操作数(二目操作符)构成了整数表达式：

(单目+)　+35=35　(减法-)　73-50=23
(加法+)　4+6=10　(乘法*)　-3*7=-21

整数表达式和关系算符可产生 True 或 False 的结果：

(关系:小于)　5<7　True
(关系:大于等于)　10>=18　False

理论上，整数没有大小限制，在其 ADT 的定义中，我们暗示了这一事实。

我们提供原始类型的 ADT 时假定读者熟悉其每个操作的条件、输入和结果，例如，整数除的完整说明如下：

整数除：
　输入：　两个整数值 U 和 V
　条件：　除数 V 不能为 0
　加工：　U 和 V 进行整数除
　输出：　商值
　结果：　如果 V=0，则出错

但在下面的说明中，为简化起见，我们用 C++的运算符来代替通用的运算符，且只给出加工描述。

```
ADT    Integer    is
    Data
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
  有符号的自然数 N
Operations
  假设 U 和 V 为整型表达式,N 为整型变量
  赋值
    =       N=U     将表达式 U 的值赋给变量 N
  二目算术运算
    +       U+V     两个整数值相加
    -       U-V     两个整数值相减
    *       U*V     两个整数值相乘
    /       U/V     用整数除求商
    %       U%V     用整数除求余数
  单目算术运算
    -       -U      改变符号(单目负号)
    +       +U      +U 与 U 相同(单目加号)
  关系运算符
    (在给定条件下,关系表达式值为 True)
    ==      U==V    若 U 等于 V 则为 True
    !=      U!=V    若 U 不等于 V 则为 True
    <       U<V     若 U 小于 V 则为 True
    <=      U<=V    若 U 小于或等于 V 则为 True
    >       U>V     若 U 大于 V 则为 True
    >=      U>=V    若 U 大于或等于 V 则为 True
end  ADT  Integer
```

例 2.1

```
3+5            (表达式值为 8)
val=25/20      (val=1)
rem=25%20      (rem=5)
```

整数的机内存储

程序设计语言的类型说明和计算机硬件一起提供整数的实现。计算机系统用固定大小的内存块来存放整数,导致了整数值域为一有限区间,其值域与存储块大小相关。为满足某些标准,程序设计语言提供两种原始的固有整数类型,短整型和长整型。当需要很大的整型值时,必须由应用程序本身提供子程序来实现其运算,这些运算可将整数扩充到任意大小,但此时应用程序的运行效率可能会被这些例程严重降低。

在计算机内,整数以由 0 和 1 组成的二进制数存放。但在日常生活中,我们用 0,1,2,…,9 这 10 个数字组成的十进制系统来表示整数。一个十进制数由以 10 为幂的数字集 $d_0, d_1, d_2, \cdots$ 组成。例如,k 位数

$$N_{10} = d_{k-1} d_{k-2} \cdots d_j \cdots d_1 d_0 \qquad 0 \leqslant d_j \leqslant 9$$

表示

$$N_{10} = d_{k-1}(10^{k-1}) + d_{k-2}(10^{k-2}) + \cdots + d_j(10^j) + \cdots + d_1(10^1) + d_0(10^0)$$

下标 10 表示 N 写作十进制数,例如,四位数的十进制整数 2589 可表示为

$$2589_{10} = 2(10^3) + 5(10^2) + 8(10^1) + 9(10^0)$$

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

$$= 2(1000) + 5(100) + 8(10) + 9$$

二进制整数用数字 0 和 1 以及 2 的幂表示。($2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16$ 等)。例如,13_{10}可表示为

$$13_{10} = 1(2^3) + 1(2^2) + 0(2^1) + 1(2^0) = 1101_2$$

一个二进制数字称为位(bit)。通常,k 位二进制数可表示为

$$\begin{aligned} N_2 &= b_{k-1}b_{k-2}\cdots b_j\cdots b_1b_0 \\ &= b_{k-1}(2^{k-1}) + b_{k-2}(2^{k-2}) + \cdots + b_j(2^j) + \cdots + b_1(2^1) + b_0(2^0) \qquad 0 \leqslant b_j \leqslant 1 \end{aligned}$$

下面的 6 位二进制数是 42 的二进制表示,二进制数的十进制值可通过计算各项之和得到:

$$101010_2 = 1(2^5) + 0(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 0(2^0) = 42_{10}$$

例 2.2

计算下列二进制数的十进制值:

1. $110101_2 = 1(2^5) + 1(2^4) + 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0) = 53_{10}$
2. $10000110_2 = 1(2^7) + 1(2^2) + 1(2^1) = 134_{10}$

十进制数转换为等值的二进制数则可通过求小于或等于它的最大的 2 的幂数来进行,这可决定其二进制表示的首位,然后用余数将这个过程继续下去,直到其余数为 0,即可求出其二进制的等值数。例如,求十进制数 35 的二进制数值,小于等于 35 的最大的 2 的幂数是 $32 = 2^5$,这说明 35 可表示为 6 位二进制数:

$$35_{10} = 1(32) + 0(16) + 0(8) + 0(4) + 1(2) + 1(1) = 100011_2$$

纯二进制数是 2 的幂数的简单加法,没有符号,我们称其为无符号数,它们表示正数,负数用二进制补码或符号量表示。这两种表示格式均有符号位来表示该值的符号。

内存中的数据

整数在内存中以固定长度的二进制序列形式存放,常见的长度有 8,16,32 位。这些序列的长度以 8 位为单位测量,称为一个字节。

0	0	1	0	0	0	1	1

以字节表示的 35

表 2.1 给出了常见位数表示的有符号和无符号数的范围。

表 2.1 数值范围和位大小

大小	无符号数范围	有符号数范围
8(1 字节)	0 至 $255 = 2^8 - 1$	$-2^7 = -128$ 至 $127 = 2^7 - 1$
16(2 字节)	0 至 $65535 = 2^{16} - 1$	$-2^{15} = -32768$ 至 $32767 = 2^{15} - 1$
32(4 字节)	0 至 $4294967295 = 2^{32} - 1$	-2^{31} 至 $2^{31} - 1$

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

计算机内存是一通过地址 0,1,2,3 等来寻址的字节序列。在内存中,整数的地址是存放该整数的第一个字节的位置。图 2.1 给出了数 $87_{10} = 1010111_2$存放于地址为 3 的一个字节中,数 $500_{10} = 0000000111110100_2$ 存放于地址为 4 的两个字节中的内存视图。

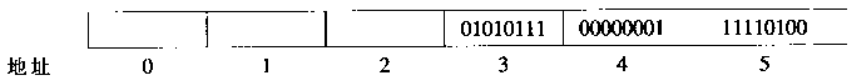


图 2.1 内存视图

C++整数表示法

C++中整数类型有整型(int)、短整型(short int)和长整型(long int)。短整型(short)可表示 16 位(2 字节)的整数值,范围为 -32768 至 32767。长整型(long)给出最宽的整数值,在多数计算机上为 32 位(4 字节),即 -2^{31}至 $2^{31}-1$。

一般整型(int)表示的整数,其位数与机器和编译器有关。典型的编译器使用 16 位或 32 位整数。在某些情况下,允许用户用选项来选择整数位数。整数类型说明了数据的值域及其算术和关系运算集,它的不同类型实现了不同范围整数值限制下的整数 ADT。

2.2 字符类型

字符数据包括大小写字母、数字、标点符号及特殊字符。计算机行业中不同的应用使用不同的字符表示。在字处理,文本输入和输出及数据通信上应用最广泛的是 128 元素的 ASCII 字符集。我们在字符 ADT 中使用 ASCII 字符集。与整数类似,ASCII 字符定义了一系列关系运算符来排序,字母字符的顺序为字典顺序,且所有大写字母小于所有小写字母:

T < W, b < d, T < b

```
ADT   character   is
    Data
      ASCII 字符集
    Operations
      赋值
        将字符值赋给字符变量
      关系运算
        用 ASCII 字典顺序定义的 6 个标准关系运算
end ADT charater
```

ASCII 字符

多数计算机系统采用 ASCII 标准编码模式来表示字符。ASCII 字符以 7 位整型码形式存放于 8 位数中。$2^7 = 128$ 个不同代码可分为 95 个可见字符和 33 个控制字符。控制字符用作数据通信和让设备完成某些控制功能,如将光标下移一行等。

表 2.2 给出了可见 ASCII 字符集,空格用◆表示。每个字符的十进制编码的十位数由表左边的列给出,个位数由表上的行给出。例如,字符"T"的 ASCII 码值为 84_{10},用二进

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

制表示为 01010100_2。

在 ASCII 码字符集中，十进制数字和字母字符所处的位置被精心安排，如表 2.3 所示，这极大地方便了大小写字母之间及 ASCII 数字代码（'0'…'9'）与其对应数字（0…9）之间的转换。

表 2.2　可见 ASCII 码字符集

	低　位									
高位	0	1	2	3	4	5	6	7	8	9
3			◆	!	"	#	$	%	&	'
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	{	|	}	~			

注：代码 00～31 和 127 为不可见的控制字符

表 2.3　ASCII 字符范围

ASCII 字符	十进制码	二进制码
空格	32	00100000
十进制数码	48～57	00110000～00111001
大写字母	65～90	01000001～01011010
小写字母	97～122	01100001～01111010

例　2.3

1. 数码'0'的 ASCII 码值为 48，数码范围为 48 至 57：

ASCII 数码：　'3'的值为 51(48＋3)

其对应的数字可由代码值减去'0'(ASCII 码值为 48)得到：

数字：　3＝'3'－'0'＝51－48

2. 将一字符从大写转换到小写，只需将该字符的 ASCII 码值加 32：

ASCII('A')＝65　　ASCII('a')＝65＋32＝97

C＋＋中用原始类型 **char** 来存放字符，ASCII 码的范围为 0 至 127；然而，与系统有关的扩展字符集经常使用范围里的其它值。用作整型时，即为该字符 ASCII 码值。

2.3　实数类型

整数类型，作为**离散类型**，表示可被记数的数据值，如－2，－1，0，1，2，3 等。但许多应

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

用需要小数数值，即实数。它可用由小数点分开整数和小数两部分的定点格式来表示：

9.6789　　-6.345　　+18.23

实数也可用科学记数法或浮点格式来表示，它包括尾数和指数两部分，指数部分表示10的多少次方，例如，6.02e23表示尾数为6.02，指数为23的实数。定点数只是浮点数中指数为0的特例。和整型与字符型一样，实数也可用抽象数据类型表示，除了用实数除代替整数除以外，标准的算术运算和关系运算都可用于实数类型。

```
ADT Real is
    Data
      用定点或浮点格式描述的数
    Operations
      赋值
        将一实数表达式值赋给实数变量
      算术运算符
        标准的二目和单目算术运算符，除法用实数除法
        没有其他的算术运算符
      关系运算符
        六个标准的关系运算均可用于实数
end ADT Real
```

实数表示法

和整数一样，实数的值域也是无限的，实数值在正负两个方向都是无限的，其小数部分也可将实数映射到坐标轴的连续的点上。但在计算机内，由于在有限的内存块上存放实数，故使得实数在范围上有限并在坐标轴上是离散的点。

多年来，计算机研究者使用了多种格式来存放浮点数，其中，IEEE格式是一广泛使用的标准。以读者熟悉的定点格式为例，它被分开为整数部分和小数部分两部分。小数部分是其数字乘以1/10，1/100，1/1000等等。一个十进制小数点将数分为两部分：

$$25.638 = 2(10^1) + 5(10^0) + 6(10^{-1}) + 3(10^{-2}) + 8(10^{-3})$$

与整数一样，定点实数也有相应的二进制表示，包括二进制整数部分、二进制小数部分和二进制小数点。二进制小数点后的小数部分值为对应数字乘以1/2，1/4，1/8等。这种表示的通常形式为：

$$N = b_n \cdots b_0 . b_{-1} b_{-2} \cdots = b_n(2^n) + \cdots + b_0(2^0) + b_{-1}(2^{-1}) + b_{-2}(2^{-2}) + \cdots$$

例如：

$$\begin{aligned} 1011.1101_2 &= 1(2^3) + 1(2^1) + 1(2^0) + 1(2^{-1}) + 1(2^{-2}) + 1(2^{-4}) \\ &= 8 + 2 + 1 + 0.5 + 0.25 + 0.0625 \\ &= 11.8125_{10} \end{aligned}$$

十进制和二进制浮点数之间的转换用的算法与整数转换算法相似。二进制转换成十

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

进制可通过将两部分转换成十进制之后得到。十进制转换成二进制要复杂一些，因为可能要用无限的二进制数来表示等值的十进制浮点数。但在计算机上，只用定长的浮点数，所以其表示也可截为有限长度了。

例 2.4

将二进制数转换为十进制数：

1. $0.01101_2 = 1/4 + 1/8 + 1/32 = 0.25 + 0.125 + 0.03125 = 0.40625_{10}$

将十进制数转换成二进制浮点数：

2. $4.3125_{10} = 4 + 0.25 + 0.0625 = 100.0101_2$

3. 十进制数 0.15 并没有一个等值的二进制小数部分，将其转换成二进制数需要无穷多位。由于计算机存储将其限制在固定长度，无限长度的尾数被截断，剩余部分的和近似等于十进制值：

$$0.15_{10} = 1/8 + 1/64 + 1/128 + 1/1024 + \cdots = 0.0010011001\cdots_2$$

多数计算机用科学记数法存放二进制实数，包括符号、尾数和指数。

$$N = \pm D_n D_{n-1} \cdots D_1 D_0 . d_1 d_2 \cdots d_n \times 2^e$$

C++支持三种实数类型，即 float，double 和 long double，long double 类型用于要求高精度计算的情况，本书不讨论。通常，浮点类型用 IEEE 32 位浮点格式实现，double 类型采用 64 位格式。

2.4 枚举类型

ASCII 字符集用整数来表示字符数据，类似的整数表示法可用来描述程序员定义的数据集合。例如，下述是一些天数为 30 天的月份：

April，June，September，November

月份的集合形成了枚举数据类型，这种类型中的元素中的顺序由列出它们的方式给定。例如：

头发颜色

```
black       // 第 1 个值
blond       // 第 2 个值
brunette    // 第 3 个值
red         // 第 4 个值
```

这种类型支持赋值操作及标准的关系函数。例如：

```
black < red        // black 在 red 之前列出
brunette > = blond// brunette 在 blond 之后列出
```

枚举类型包含数据和操作，也是 ADT。

```
ADT Enumerated is
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Data

用户定义的具有 N 个不同元素的表

Operations

赋值

枚举类型变量可被赋值为表中的任一元素

关系运算

6 个标准的关系运算，元素之间大小由在表中的位置决定

end ADT Enumerated

C++枚举类型的实现

C++具有可由名字常量引用的定义为不同整数值的枚举类型。

例 2.5

1. Boolean 类型可说明为枚举型，常量 False 的值为 0，True 的值为 1，变量 Done 定义为枚举型，其初值为 False：

```
enum Boolean{False,True};
Boolean Done = False;
```

2. 一年中的各月被定义为枚举类型。缺省情况下，一月(Jan)的值为 0。然而，该整数序列可用任一值开始，只需将该值赋给第一个元素。在下面的情况下，一月(Jan)的值为 1，月份对应的顺序为 1,2,…,12：

```
enum Month{Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov,
           Dec};
Month Mon = Dec;
```

2.5 指针

指针数据类型在任何程序设计语言中都是基本内容。所谓指针，实际上是一代表某一内容地址的无符号数，它也可用来得到存放于该地址的数据。指针所指地址内存放的数据的类型称为基类型，它在定义指针时使用。例如，我们可定义指向字符的指针、指向整数的指针等等。在图 2.2 中，两种情况下的指针 P 值(地址)都为 5000，然而，(A)中 P 所指数据为一字符，(B)中 P 所指数据为一短整数。

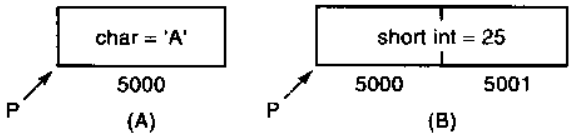


图 2.2 指针数据类型

指针可有效地用来存取表中的元素，同时，它也是开发动态数据结构如链表、树及图的基础。

指针 ADT

作为整数，指针可用来进行某些算术和关系运算。值得注意的是算术运算。指针可通过增加或减少一个整数值来指向内存中的新数据。加 1 操作将指针指向内存中的下一

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

个元素。例如，如果 p 指向一个字符对象，则 p + 1 指向内存中的下一字节。指针加 k(k > 0)即将指针右移 k 个数据位置。例如，若 p 指向 double 型数据，则 p + k 指向 p 右边 N = sizeof(double) * k 字节的 double 数据。

数据类型	当前地址	新地址
char	p = 5000	p + 1 = 5001
int(2 byte)	p = 5000	p + 3 = 5000 + 3 * 2 = 5006
double(4 byte)	p = 5000	p - 6 = 5000 - 6 * 4 = 4976

指针用地址操作返回内存中数据项的地址。相反，操作“ * ”返回指针所指数据的值。指针大小可通过比较其无符号整数值决定。

动态存储区是在程序执行时新申请的内存区，它用来区别静态存储区，即在程序开始执行之前(编译时)就已存在的存储区。动态存储区将在第 8 章讨论。以类型 T 为参数的操作符 New 可动态地为一个类型为 T 的数据申请内存；以指针为参数的 delete 操作可释放该指针申请的内存。

```
ADT Pointer is
    Data
      表示内存地址的无符号整数集，地址中存放基类型 T 的数据元素
    Operations
      假设 u 和 v 为指针表达式，i 为整数表达式，ptr 为指针变量，var 是一个类型为 T 的变量
      地址操作
        &        ptr = &var        将 var 的地址赋给 ptr
      赋值操作
        =        ptr = u           将指针 u 赋给 ptr
      求值操作
        *        var = * ptr       将指针 ptr 所指的类型为 T 的数据项的值赋给 var
      动态内存的申请与释放
        new      ptr = new T       为类型为 T 的元素申请动态内存区，并将申请到的地址赋
                                   给 ptr
        delete   delete ptr        释放在地址 ptr 处申请的动态内存
      算术操作
        +        u + i             指针从 u 右移 i 个数据元素的位置
        -        u - i             指针从 u 左移 i 个数据元素的位置
        -        u - v             返回两个指针之间的基类型的元素的个数
      关系操作
        通过比较指针的无符号整数值实现 6 个标准关系操作
end ADT Pointer
```

指针值

根据机器体系结构的不同，指针值是使用 16,32 或更多位的内存地址。例如，pc 是指向一个字符(1 字节)的指针，px 是指向短整数(2 字节)的指针。

```
char str[] = "ABCDEFG";
char * PC = str;          // PC 指向字符串 str
short X = 33;
short * PX = &X;          // PX 指向短整形 X
```

以下语句示意基本指针操作

```
cout << * PC << endl;          // 输出'A'
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

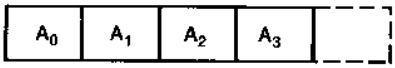
www.Linuxidc.com

```
PC + = 4;                        // 将 PC 右移 4 个字符
cout < < *PC < < endl;           // 输出'E'
PC - -;                          // 将 PC 左移 1 个字符
cout < < *PC < < endl;           // 输出'D'
cout < < *PX + 3 < < endl;       // 输出 36 = 33 + 3
```

指针操作 new 和 delete 在第 8 章讨论。

2.6 数组类型

数组是数据集的例子。一维数组是一具有有限个相同数据类型元素的顺序表(同构数组)。顺序即哪个是第 1,第 2,第 3 个元素,等等。与每个元素相关联的是标识每个元素在表中位置的**下标**(Index)。数组就是下标运算符,它允许在存储和检索数据项时对表中元素直接访问。



ADT Array is

Data

N 个类型相同的元素的集合,下标为 0 到 N - 1,它标识了元素在表中的位置,并可通过它直接存取元素,下标 0 标识表中的第 1 个元素,1 标识第 2 个元素,等等。

Operations

下标运算 [ ]

Input:	下标
Preconditions:	下标在 0 到 N - 1 之间
Process:	在赋值语句右边时,该操作从元素中检索数据;在左边时,返回存放着右边表达式所指值的数组元素的地址
Output:	若该操作在赋值语句右边,则该操作从数组中取得数据并将其返回给调用者
Postconditions:	若该操作在左边,则改变相应的数组元素

end ADT Array

内置的 C + + 数组类型

作为其基本语法的一部分,C + + 提供可定义一组相同类型元素的静态数组类型。在说明中,用常量 N 明确给出数组的大小,即其下标范围为 0 至 N - 1。例 2.6 说明了静态的 C + + 数组。在第 8 章中,我们还可用指针来产生动态数组。

例 2.6

1. 定义两个类型为 double 的数组。数组 X 有 50 个元素,数组 Y 有 200 个元素。

```
double  X[50],  Y[200];
```

2. 定义一个长整型数组 A,其大小由整型常量 ArraySize = 10 定义。

```
const  int  ArraySize = 10;
long  A[ArraySize];
```

我们可通过范围为 0 至 ArraySize - 1 的数组下标来存取单个数组元素。A[i]表示下标为 i 的元素。

C + + 中,数组的索引运算由下标运算符([ ])完成。它是二目运算符,其左边是数组

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

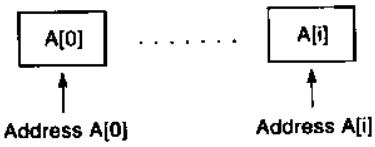
www.Linuxidc.com

名称，右边为元素在数组中的位置。赋值符号的两边均可通过该操作来存取数组元素。

```
A[i]=x;            //将 x 存放到数组中
t=A[i];            //从数组中检索数据 A[i]并赋给变量 t
A[i]=A[i+1]=x;     //将 x 赋给 A[i+1]，第二个赋值语句将 A[i+1]赋给 A[i]
```

一维数组的内部存储

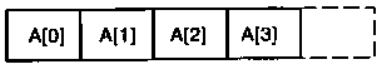
C++的一维数组 A 逻辑上在内存中连续存放，其每个元素的类型相同。



在 C++中，数组名为常量，且作为数组中第一个元素的地址。因此，在数组声明

```
Type A[ArraySize];
```

中，数组名 A 是一个常量，它存放第一个数组元素 A[0]的内存地址。元素 A[1]，A[2]等顺序存放在其之后。假设 sizeof(Type)=M，则整个数组 A 占据 M * ArraySize 个字节。



编译器建立一张内部向量表来维持数组属性记录。表中包括元素大小、数组起始地址、数组中元素个数等方面的信息。

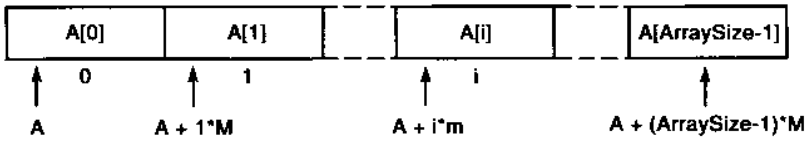
起始地址：　A
数组元素个数：　ArraySize
元素类型的大小：　M = sizeof(Type)

该表也被编译器用来实现标识元素在内存中的地址的存取函数。函数 ArrayAccess 用数组的起始地址及其数据类型的大小来将下标 I 映射到 A[I]的地址：

```
Address A[I]=ArrayAccess(A,I,M)
```

ArrayAccess 通过下面的算法实现：

```
ArrayAccess(A,I,M)=A+I*M;
```



例 2.7

假定 float 型用 4 字节存放(sizeof(float)=4)。数组 Height 起始内存位置为 20000。

```
float Height[35];
```

数组元素 Height[18]存放于下面地址：

20000 + 18 * 4 = 20072

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

数组越界

ADT array 设定，当数组大小为 N 时，下标范围为整数 0 至 N－1。实际上，大多数 C＋＋编译器在存取数组时并不生成检查数组是否越界的代码。例如，多数 C＋＋编译器会接受下面这段代码而并不报错：

```
int V = 20;
int A[20];        // 数组大小为 20，下标范围为 0～19。
A[V] = 0;         // 下标 V 超出下标上限。
```

数组占用用户数据区中内存空间。数组存取函数计算每个指定元素的地址，但通常并不检查是否越界。因此，C＋＋程序可能造成下标越界。此时，通过越界下标计算出的地址也可能仍在用户数据区中。处理时程序就可能覆盖其他变量，造成不必要的运行错误。图 2.3 给出了程序在内存中的视图。

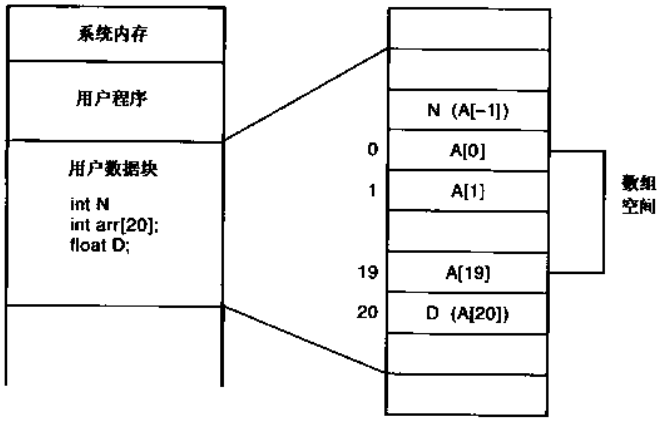


图 2.3　用户数据区中数组空间

也有一些编译器通过生成执行代码来检查数组下标是否在范围之内。由于这些额外代码减慢了程序执行，多数程序员只在程序开发时使用。一旦代码调试完毕，则将选项关闭并重新编译来生成更高效的代码。解决越界问题的另一方法是开发一种“安全数组”，它捕捉非法下标并打印出错信息。我们在第 8 章中介绍安全数组。

二维数组

二维数组，通常称为矩阵，是一种由多个一维数组合成的结构化数据类型。其元素通过行和列下标存取。例如，以下是一有 4 行 8 列 32 个元素的矩阵 T，值 10 可通过(行，列)对(1,2)存取，－3 可通过(2,6)存取。

列

行	0	1	2	3	4	5	6	7
0								
1			10					
2							－3	
3								

二维数组的概念可扩充为一般的多维数组，其元素可通过 3 个或更多下标存取。二

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

维数组在数据处理、多项式方程求解等多个领域内有着广泛的应用。

C++中，二维数组的说明应定义行数、列数及元素的数据类型。

```
<Type> T[行数][列数]
```

T的元素通过行、列下标存取：

```
T[i][j],0≤i≤行数-1,0≤j≤列数-1
```

例如，矩阵T是一4×8的整数数组：

```
int T[4][8];
```

其中T[1][2]=10,T[2][6]=-3;

我们可将二维数组看成一维数组的表，如T[0]是第0行，包括ColumnCount个单独的元素。当二维数组作为参数时，这个概念十分有用。T[][8]说明的是下述有8个元素的表。

	0	1	2	3	4	5	6	7
T[0]								
T[1]			10					
T[2]							-3	
T[3]								

二维数组的存放

二维数组可通过一次赋值一行元素来初始化。例如，数组T是一3×4的表：

```
int T[3][4] = {{20,5,-30,0},{-40,15,100,80},{3, 0, 0, -1}};
```

行	0	1	2	3
0	20	5	-30	0
1	-40	15	100	80
2	3	0	0	-1

作为数组，其元素按第1行、第2行、第3行的顺序存放，如图2.4所示：

按行有效的数组

行#0				行#1				行#2			
20	5	-30	0	-40	15	100	80	3	0	0	-1

图2.4 矩阵T的存储

为存取内存中的元素，编译器扩展内部向量表，增加了行数和列数等信息，编译器用新的存取函数MatrixAccess来返回元素的地址。

起始地址： T

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

行数：　　RowCount

列数：　　ColumnCount

类型大小：　　M = sizeof(Type)

行大小：　　RS = M * ColumnCount(一个整行的大小)

函数 MatrixAccess 读入行列下标对(I,J)；返回元素 T[i][j]的地址：

```
Address T[i][j] = MatrixAccess(T,I,J)
                = T+(I*RS)+(J*M)
```

(I*RS)给出了存放 I 行数据所需的字节数，(J*M)则给出了存放 I 行内头 J 个元素所需字节数。

例 2.8

T 为图 2.4 给出的 3×4 矩阵，假定整数的字节数为 2，并且矩阵存放在内存中 1000 的位置。

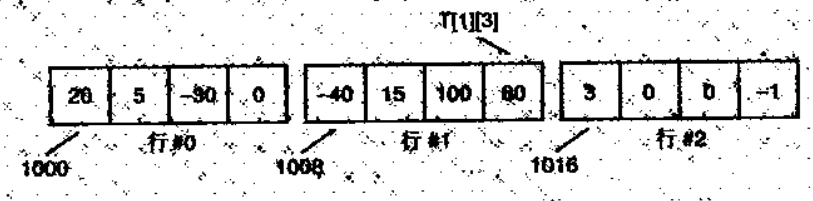
起始地址：　　1000

行数：　　3

列数：　　4

类型大小：　　2 = sizeof(int)

行大小：　　8 = 2 * 4(一整行的大小)



1. 各行存放的地址为：

行 0：　　地址 1000

行 1：　　地址 1000 + 1 * 8 = 1008(每行 8 个字节)

行 2：　　地址 1000 + 2 * 8 = 1016

2. T[1][3]的地址为：

ArrayAccess(T,1,3) = 1000 + (1 * 8) + (2 * 3) = 1014

2.7　文本串及变量

数组是一可存放多个同类元素的结构化数据类型。数组有一种特殊形式，它所存放的数据是组成名字、单词、句子等的字符数据。我们称这种结构为**串**，它将字符视为单元实体，并提供得到字符在串 1 内位置的操作。在许多字处理的应用中，串是非常重要的数据结构。在文本编辑处理、串查找、替换算法中，它都不可缺少。例如，语言学家可能需要

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

特定词汇在文件中出现的次数;程序员需要用查找/替换模式修改文件中的源代码,多数语言定义串结构并提供内置的操作及库函数来处理串。

为得到串的长度,可在串尾加0(NULL)或用一个单独的长度参数。下面是含6个字符的串STRING的表示法:

以NULL结束的串(NULL值为0)

S	T	R	I	N	G	NULL

给出长度参数的串

6	S	T	R	I	N	G
length						

许多操作将串作为基本处理对象。例如,求串长度、串拷贝、两个串连接,以及用插入、删除和串匹配操作来处理字符串,串也有用于串排序的比较操作,该操作使用ASCII码序。例如:

```
"Baker"小于"Martin"          //B在M之前
"Smith" 小于 "Smithsun"
"Barber" 小于 "barber"       //大写B在小写b之前
"123Stop" 小于 "AAA"         //数字在字母之前

ADT String is
    Data
      串是一具有一定长度的字符序列,串结构可由字符NULL结束,或另有一单独的长度参数
    Operations
      Length
        Input:            无
        Preconditions:    无
        Process:          对以NULL结束的串,计算直到NULL字符之前的字符个数;对有长
                          度参数的字符串,检索该长度值
        Postconditions:   无
        Output:           返回该串的长度
      Copy
        Input:            两个串STR1和STR2,STR2是源串,STR1是目的串
        Preconditions:    无
        Process:          从STR2拷贝字符到STR1
        Postconditions:   创建一个长度和数据与STR2相同的串STR1
        Output:           返回指向STR1的指针
      Concatenation
        Input:            两个串STR1和STR2,将STR2连接到STR1的尾部
        Preconditions:    无
        Process:          找到串STR1的尾部,将STR2的字符拷贝到STR1的尾部,并修改
                          串STR1的长度信息
        Postconditions:   STR1被改变了
        Output:           返回指向STR1的指针
      Compare
        Input:            两个串STR1和STR2
        Preconditions:    无
        Process:          比较两串的ASCII码值的大小
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    Postconditions:   无
    Output:           按下述规则返回值
                      若 STR1 小于 STR2：返回负数值
                      若 STR1 等于 STR2：返回 0
                      若 STR1 大于 STR2：返回正数值
  Index
    Input:            串 STR 和单字符 CH
    Preconditions:    无
    Process:          在 STR 中查找字符 CH
    Postconditions:   无
    Output:           返回串 STR 中第一次发现字符 CH 的位置。
                      若没有找到，则返回 0
  RightIndex
    Input:            串 STR 和单字符 CH
    Preconditions:    无
    Process:          在串 STR 中找 CH 出现的最后位置
    Postconditions:   无
    Output:           返回串 STR 中最后一次发现 CH 的位置
                      若没有找到，则返回 0
  Read
    Input:            字符读出的文件流和存放字符的串 STR
    Preconditions:    无
    Process:          从文件流中读入字符序列到串 STR 中
    Postconditions:   串 STR 指向读入的字符序列
    Output:           无
  Write
    Input:            存放输出字符的串和字符将被写入的流
    Preconditions:    无
    Process:          将字符串写入流中
    Postconditions:   无
    Output:           输出的流被改变了
end ADT String
```

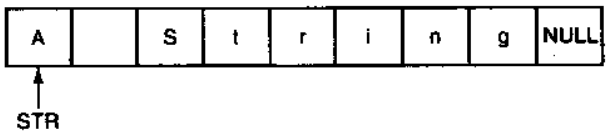
C++串

在第 8 章中给出了一个完整的 C++串类，包括扩充的比较操作和 I/O 操作。在本章中，我们用 C++NULL 结束的串和 C++的串库来实现串 ADT。

C++串是以 ASCII 值为 0 的空字符(NULL)结束的字符串，编译器以双引号内的一系列字符来标识字符串，字符串变量实际上是字符数组，它存放以 NULL 结束的字符序列。下述语句说明了一个字符数组并赋初值：

```
Char STR[9] = "A String";
```

串"A String"以 9 元素字符数组的形式存放在内存中



C++为字符流 cin(键盘)，cout(屏幕)，cerr(屏幕)和用户定义的文件流提供文本I/O操作。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

C++串库(string.h)包含许多灵活的字符串函数,它们直接实现了多数串 ADT 操作。表2.4列出了重要的 C++串函数。

表 2.4 C++的串函数及其举例

```
char s1[20] = "dir/bin/appl", s2[20] = "file.asm", s3[20];
char *p;

1. 串长度        int  strlen(char *s);
        cout << strlen(s1) << endl;                    // 输出 12
        cout << strlen(s2) << endl;                    // 输出 8

2. 串拷贝        char *strcpy(char *s1, char *s2);
      strcpy(s3,s1);                                   // s3 = "dir/bin/appl"

3. 串连接        char *strcat(char *s1, char *s2);
        strcat(s3,"/");
        strcat(s3,s2);                                 // s3="dir/bin/appl/file.asm"

4. 串比较        int strcmp(char *s1, char *s2);
        result = strcmp("baker","Baker");              // result > 0
        result = strcmp("12","12");                    // result = 0
        result = strcmp("Joe","Joseph");               // result < 0

5. 串定位       char *strchr(char *s, int c);
        p = strchr(s2,'.');
        if (p)                                         // p 指向 file 后面的'.'
        strcpy(p,".cpp");                              // s2 = "file.cpp"

6. 串右定位        char *strrchr(char *s, int c);
      p = strrchr(s1,'/');                             // p 指向 bin 后面的'/'
      if (p);                                          // 在 bin 后结束串,
      *p = 0;                                          // s2="dir/bin"

7. 读入串        StreamVariable >> s

8. 写串         StreamVariable << s                   // 若输入是"hello world"
        cin >> s1;                                     // s1 是 "hello"
        cout << s1;                                    // 输出也是 "hello"
```

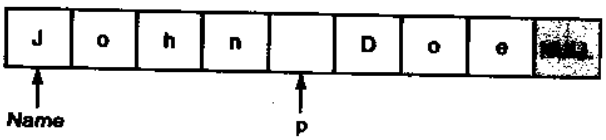
应用:姓名对换

我们给出一个串应用实例来说明 C++串的库函数。该程序用函数 strchr(),strcpy()和 strcat()将名字如"John Doe"拷贝成"Doe John"存放于串 Newname 中。下述语句描述了该算法:

```
Char    Name[10]="John Doe",Newname[30];
char    *p;
```

语句 1: p = strchr(Name, '');

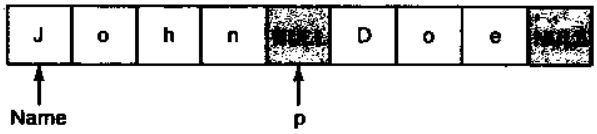
返回指向 Name 中第一个空格的指针 p。这样,姓的第一个字符从地址 p+1 开始。



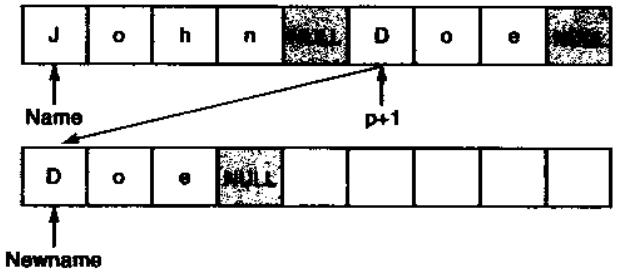
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

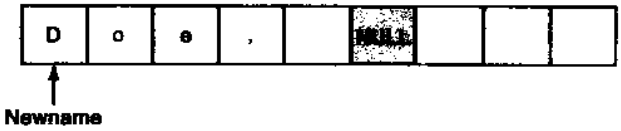
语句 2：*p = 0;　　// 将空格置换为 NULL



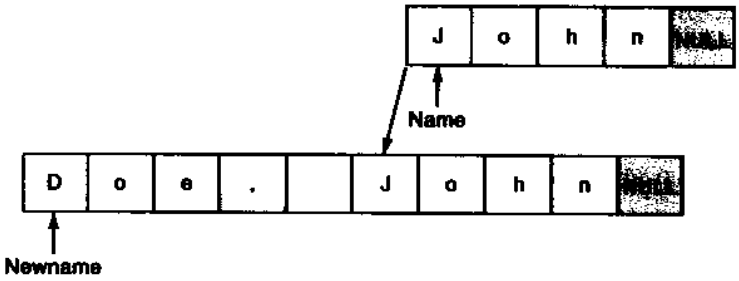
语句 3：strcpy(Newname, p+1);　　// 将姓拷贝到 Newname 串中



语句 4：strcat(Newname, ", ");　　// 在 Newname 后拼上','和空格



语句 5：starcat(Newname, Name);　　// 再在 Newname 后拼上名



程序 2.1　姓名对换

本程序用上述语句 1 到语句 5 来反转姓名。这些语句的编码均在函数 ReverseName 中。主程序循环读入三个串并分别输出其反转后的姓名来测试该算法。

```
#include <iostream.h>
#include <string.h>
// 将姓和名反转并在其中加入逗号,将结果送到 newName 中
void ReverseName(char *name,char *newName)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    char *p;
    // 在 name 中找第一个空格,并将其置换为 NULL
    p=strchr(name.'');
    *p=0;
    // 将姓拷贝至 newName,然后拼上","和名
    strcpy(newName,p+1);
    strcat(newName,",");
    strcat(newName,name);
    // 将 NULL 置换回原来空格
    *p='';
}
void main(void)
{
    char name[32],newName[32];
    int i;
    // 读入并处理三个串
    for (i=0;i<3;i++)
    {
        cin.getline(name,32,'\n');
        ReverseName(name,newName);
        cout << "Reversed name:"<< newName << endl << endl;
    }
}
/*
<程序 2.1 运行结果>
Abraham Lincoln
Reversed Name: Lincoln, Abraham
Debbie Rogers
Reversed Name: Rogers, Debbie
Jim Brady
Reversed Name: Brady, Jim
*/
```

2.8 记录

记录是将多个不同类型的元素组合成一个单独对象的结构。记录中的元素称为域。和数组一样,记录也可通过存取操作符直接存取各个域。例如,Student 是描述在校学生信息的记录结构。这些信息包括姓名(Name),住址(Address),年龄(Age),专业(Major)和等级平均分(GPA)。

Name	Address	Age	Major	GPA
串	串	整型	枚举型	实型

姓名和住址域中存放着串型数据,年龄和 GPA 是数字型,专业是枚举类型。假定 Tom 是一个学生,我们可以通过用存取符"."连接起来的记录名和域名来存取各个域:

```
Tom.Name    Tom.Age    Tom.GPA    Tom.Major
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

记录结构内各域类型可以不同，和数组不一样，记录描述单个值而不是一组值。

ADT Record is

Data

包含类型不同的各个域的元素，每个域都有名字，通过域名可直接存取域中的数据

Operations

存取操作符

Preconditions:	无
Input:	记录名(Record)和域。
Process:	存取域中的数据。
Output:	若检索数据，将域值返回给用户。
Postconditions:	若存储数据，则改变记录。

end ADT Record

C++结构

C++提供结构类型来表示记录。结构是从C语言中继承下来、为了可兼容性而保留的。C++将结构类型定义为一个特殊的类，类中所有的成员均是公共的。本书中只是在处理记录时才使用结构类型。

例 2.9

```
Struct Student
{
    int id;
    char name[30];
}
Student S={555,"Davis,Samuel"};
cout<<S.id<<""<<S.name<<endl;
```

2.9 文件

本书的大部分内容集中在内部数据结构上，访问的是驻留在内存中的信息。然而，对许多应用来说，数据常常存放在磁盘等外部存储设备上。设备(物理文件)以字符流形式存放信息，由操作系统提供设备与内存交换数据的操作，这使我们可以读写在外部设备上永久存放的数据。外部设备上存储的数据及转换操作定义了(逻辑)文件数据结构。它在存放大量信息方面比传统的只在内存驻留信息方法占有很大的优势。

程序设计语言提供高级文件处理操作，将程序员从使用低级文件系统调用中解放出来。文件操作使用逻辑上和文件相连的数据流，这个数据流与文件连接起来，对输入来说，该流使数据从外部设备流到内存上(图2.5所示)；同样的程序使用输出数据流可将信息输出到文件中。

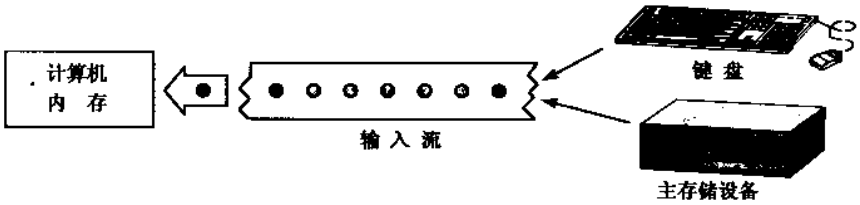


图2.5 输入数据流

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

为文件定义 ADT 是有用的。数据由表示文本数据的字符序列或表示二进制数据的字节序列组成，数据以用换行符隔开的 ASCII 字符序列形式存放。ADT 的操作由原始的 I/O 操作派生而来。其输入操作 Read 从字符流中得到字符序列，相关的输出操作 Write 插入一个字符序列到流中，另有操作 Get 和 Put 处理单个字符的 I/O。

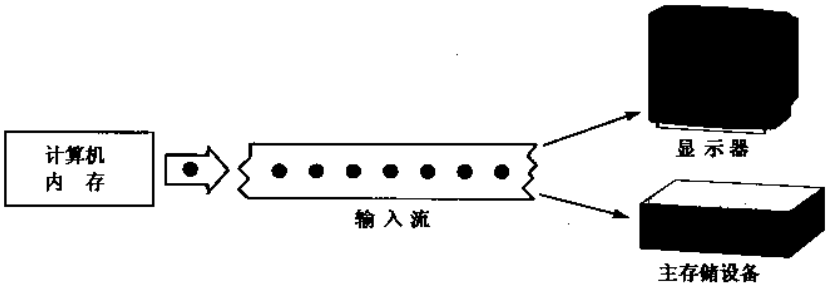


图 2.6 输出数据流

流维护一个表明在流中当前位置的**文件指针**。Input 将文件指针前移到流中下一个未读的数据项，Output 将文件指针指向下一个输出位置。查找操作(Seek)可移动文件指针的位置，该操作假定我们已经访问文件中所有字符且可移动至文件头、文件尾或文件中某一位置。一般来说，查找操作用在磁盘文件上。

文件通常和数据流以下述三种模式之一连接：只读，只写或可读写。只读和只写模式分别指定文件用作输入和输出，可读写模式允许数据流双向流动。

ADT File is

Data

外部文件标识及其数据流的方向，从文件中读出或写入文件的字符序列

Operations

Open

Input:	文件名及数据流方向
Preconditions:	对于输入来说，外部文件必须存在
Process:	建立流与文件的连接
Output:	指示操作是否成功的标志
Postconditons:	数据可通过流在外部文件和内存中流动

Close

Input:	无
Preconditions:	无
Process:	取消流和文件的连接
Output:	无
Postconditions:	数据无法通过流在外部文件和内部文件中流动

Read

Input:	存放数据块的数据；计数器 N
Preconditions:	必须用“只读”或“读写”方式打开流
Process:	从流中输入 N 个字符到数组中，遇文件结束符停止
Output:	返回读入字符的个数
Postconditions:	文件指针前移 N 个字符

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Write
  Input:              数组及计数器 N
  Preconditions:      必须用"只读"或"读写"方式打开流
  Process:            从数组中输出 N 个字符到流
  Output:             返回被写字符的个数
  Postconditions:     流中存放有输出的数据,文件指针前移 N 个字符
end ADT File
```

C++流的层次结构

C++为文件处理提供用类层次结构实现的流 I/O 系统。图 2.7 给出了部分流。C++的流对应于图中类的对象。每个流都标识了文件及其数据流的输入输出方向。

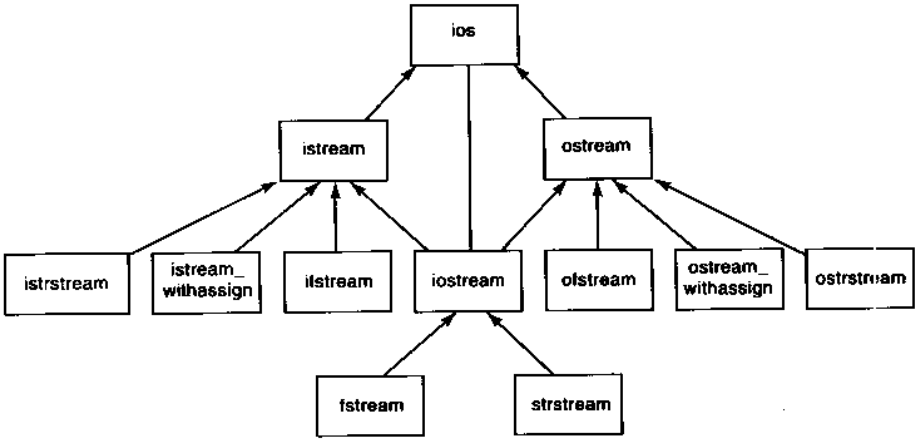


图 2.7　流类层次结构

类层次结构中的根类是 ios,它包括所有派生类的数据和操作,该类包含有标明流的特殊属性及可影响其 I/O 格式方法的标志。例如:

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

设置实数的显示方法为定点格式而不是科学记数法。

istream和ostream提供基本的输入和输出操作,用作I/O流层次结构中其余类的基类。

类 istream-withassign 是 istream 的变种,它允许对象赋值,预定义的对象 cin 是该类的一个对象,而预定义的对象 cout 和 cerr 是类 ostream-withassign 的对象。程序运行时,打开这 3 个流作为键盘输入和屏幕输出,其说明包括在文件 <iostream.h> 中。

类 ifstream 是从 istream 中派生出来的,用于磁盘文件输入;类似地,ofstream 用于磁盘文件输出。这些类在文件 <fstream.h> 中定义。所有类都有将文件与流连接的 open 操作和将文件与流分离的 close 操作。

磁盘文件有两种类型,文本文件和二进制文件。文本文件中存放可打印的 ASCII 字符而二进制文件中存放纯粹的二进制数据。程序 2.2 给出了一个文本文件 I/O 的实例。第 14 章将给出一个二进制文件类,它用来实现外部查找和排序算法。

类 fstream 允许用户创建和维护需同时读写的文件,它在第 14 章与相关的应用一起讨论。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类 istrstream 和 ostrstream 实现以数组为基础的 I/O，它们在文件 < strstream.h > 中说明。这里，数据是从数组中读入或写到数组中，而不是外部设备上。文本编辑器经常使用以数组为基础的 I/O 来完成复杂的格式化操作。

程序 2.2　文件 I/O

本程序给出了 C++ 流的使用，包括文本文件和以数组为基础的 I/O。程序分别使用了定义在 < iostream.h > 中的 cout 和 cerr，定义在 < fstream.h > 中的文本文件输入和定义在 < strstream.h > 中以数组为基础的文件输出。程序打开文件，并按名字值的格式读入每个有变量/值的行，通过用基于数组的流操作，这对名字和值以格式"name = value"的形式写入数组 outputstr，然后用 cout 语句打印输出。例如，输入行为：

```
start 55
stop 8.5
```

将以下述串的形式输出：

```
start = 55     stop = 8.5
```

```
#include <iostream.h>
#include <fstream.h>
#include <strstream.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
    // 包含有名字及值的文本文件
    ifstream fin;

    // 标识符存放于 name 中，对立值存放于 outputstr 中
    char name[30],outputstr[256];

    // 定义一个使用 outputstr 的基于数组的输出流
    ostrstream outs(outputstr,sizeof(outputstr));

    double value;

    // 打开 names.dat 用作输入，并确保其存在
    fin.open("names.dat",ios::in | ios::nocreate);
    if (! fin)
    {
        cerr << "cloud not open 'names.dat'" << endl;
        exit(1);

    // 读入名字及对立值，并用格式'name:value'写到 outs 中
    while(fin >> name)
    {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        fin > > value;
        outs < < name < < "=" < < value < < "";
    }

    // 以 Null 结束输出串
    outs < < ends;

    outs < < outputstr < < endl;
}

/*
<文件"name.dat">

start 55
breakloop 255.39
stop 23

<程序 2.2 运行结果>

start = 55    breakloop = 255.39    stop = 23
*/
```

2.10 数组和记录的应用

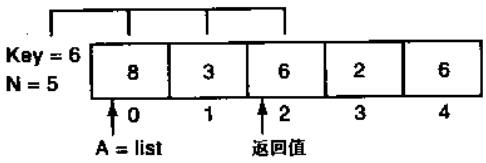
大多数程序设计语言都提供固有的数组和记录数据结构。本章介绍这两种结构的 ADT 并讨论其 C++ 的实现，并用它们来实现本书中的一些重要算法。数组是表的基础，在许多应用中，我们用查找和排序例程在以数组表示的表中查找元素及对数据排序。本章介绍易于编码及理解的顺序查找和交换排序算法。

顺序查找

顺序查找用一称为 key 的目标值来查找表中的元素。该算法从用户给出的起点开始，遍历表中的后续元素，并将元素值与 key 值比较，直到元素值与 key 值相等或到表尾。如果找到 key 值，函数返回该元素在表中的位置；否则，返回 -1。函数 SeqSearch 需要 4 个参数，即表的起始地址，查找的起始位置、元素个数及 key 值。例如，数组 A 中存放以下整数组成的一个表：

A:　　8　3　6　2　6

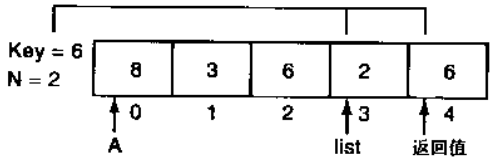
1. key = 6，start = 0，n = 5，则从表头开始查找，返回第一次遇到元素 6 的位置。



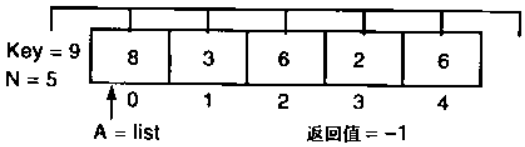
2. key = 6，start = 3，n = 2，从 A[3]开始对表进行查找，返回第一次遇到元素 6 的位置。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



3. key = 9, start = 0, n = 5,从第一个元素开始查找表中值为 9 的元素,由于该值不存在,返回值为 - 1。



顺序查找算法运用于定义了算符"= = "的元素类型,完整的通用查找算法要求样板及算符重载。这些将在第 6 及第 7 章中讨论。以下是整数数组的顺序查找算法的实现。

顺序查找算法:

```
int SeqSearch(int list[], int start, int n, int key)
{
    for(int i = start;i < n; i + + )
        if (list [i] = = key)
            return i;
        }
        return - 1;
}
```

程序 2.3　重复查找

本程序通过对 key 在表中出现的次数进行计数来检验顺序查找算法主程序。首先读入 10 个整数到数组 A,然后读入 key 值。

程序用不同的起点重复调用 SeqSearch。开始时,起点为 0,即数组的起始位置。每次调用 SeqSearch 后,如果发现 key 值,则记录发现次数的计数器加 1;否则结束查找并输出计数器值。发现 key 值时,函数返回值表明了它在表中的位置,则下次调用 SeqSearch 将从紧靠其右边的元素开始。

```
# include <iostream.h>
// 在 n 元整数数组中查找与 key 值相等的元素,返回指向该元素的指针,若没有找到,
// 则返回 NULL。
int SeqSearch(int list[], int start, int n, int key)
{
    for(int i = start;i < n; i + + )
        if (list[i] = = key)
        return i;       // 返回与 key 相等的元素的下标
        return - 1;     // 找不到 key 值,则返回 - 1。
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

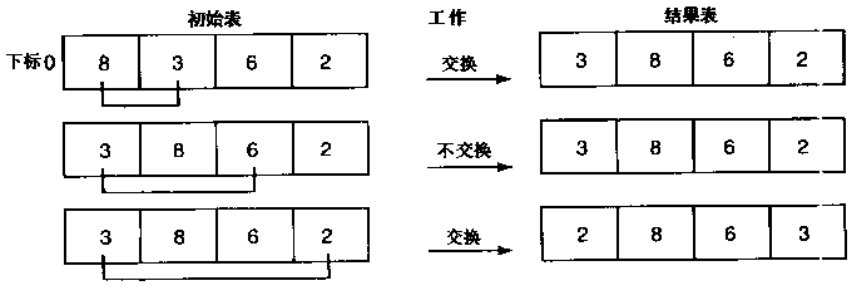
```
void main(void)
{
    int A[10]
    int key, count = 0, pos;
    // 提示用户输入 10 个整数组成一个数组,然后再输入想查找的 key 值
    cout << "Enter a list of 10 integers: ";
    for (pos = 0; pos < 10; pos++)
        cin >> A[pos];
    cout << "Enter a key: ";
    cin >> key;
    // 从第一个数组元素开始查找
    pos = 0;
    // 从数组中顺序查找 key 值
    while ((pos = SeqSearch(A,pos,10,key)) ! = -1)
    {
        count++;
        // 找到后指针后移
        pos++;
    }
    cout << key << " occurs " << count
         << (count ! = 1 ? " times" : " time")
         << " in the list." << endl;
}
/*
<程序 2.3 运行结果>
Enter a list of 10 integers: 5 2 9 8 1 5 8 7 5 3
Enter a key: 5
5 occurs 3 times in the list
*/
```

交换排序

表中元素的顺序对许多应用来说很重要。例如,为能尽快查到进货记录,可能要将记录对货号进行排序;字典内单词要按字母排序;学校对学生记录按社会保险号排序等等。

我们介绍将元素按升序排列的算法 ExchangeSort 来创建有序表。我们以表 8,3,6,2 来说明该算法。其结果为有序表 2,3,6,8。

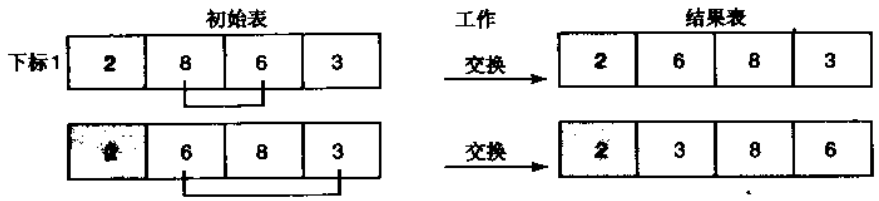
下标 0:考虑整个表 8,3,6,2,将下标为 0 的元素依次和下标为 1,2,3 的元素进行比较,对每一次比较,若后续元素比下标 0 的元素小,则互相交换这两个元素,所有比较完成后,最小的元素放在下标 0 的位置上。



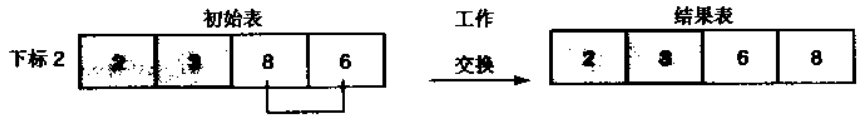
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

下标 1：由于最小元素已经放在下标 0 的位置，则只需考虑子表 8,6,3，即从下标 1 到表尾的元素。下标 1 的元素与后续下标为 2 和 3 的元素进行比较，每次比较，如果后续元素比下标 1 的元素小，则互相交换两个元素，所有比较完成后，第二小的元素放在下标 1 的位置上。



下标 2：考虑子表 8,6。对下标为 2 和 3 的两个元素的子表进行上述过程。只进行一次比较即可得出交换两个元素的结果。



在下标 3 中我们只剩下一个元素，表已经排列好了。

最终有序表

2	3	6	8

C++ 的 ExchangeSort 函数使用嵌套循环。假设表的大小由 n 给出。外部循环下标 i 递增范围是 0 到 n-2。每一个下标 i 都要与下标为 j=i+1,i+2…,n-1 的后续元素相比较。如果 list[i]>list[j]则交换两个元素。

程序 2.4　对表排序

本程序演示排序算法。15 个 0 到 99 之间的的整数组成一个表。ExchangeSort 对表进行排序，用函数 Swap 交换两个数组元素。程序在排序前后分别打印出表的内容。

```
#include <iostream.h>

// 互换两个整型变量 x 和 y 的值
void Swap(int & x, int & y)
{
    int temp = x;     // 保存 x 的初始值
    x = y;            // 将 x 换成 y 值
    y = temp;         // 将 x 的初始值赋给 y
}

// 将 n 元整数数组按升序排序。
void ExchangeSort(int a[], int n)
{
    int i, j;
    // 循环 n-1 遍，使 a[0],…,a[n-2]值按升序排列
    // a[0],...,a[n-2].
    for (i = 0; i < n-1; i++)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 将 a[i+1],...,a[n-1]中的最小值置于 a[i]中
          for (j = i+1; j < n; j++)
            // 若 a[i]>a[j], 则互换两值
                if (a[i] > a[j])
                    Swap(a[i], a[j]);
}
// 遍历全数组,并打印每个元素的值
void PrintList(int a[], int n)
{
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;
}
void main(void)
{
    int list[15] = {38,58,13,15,51,27,10,19,12,86,49,67,84,60,25}
    int i;
    cout << "Original List \n";
    PrintList(list,15)
    ExchangeSort(list,15)
    cout << endL << "Sorted List" << endL;
    PrintList(list,15)
}
/*
<程序 2.4 运行结果>
Original List
38  58  13  15  51  27  10  19  12  86  49  67  84  60  25

Sorted List
10  12  13  15  19  25  27  38  49  51  58  60  67  84  86
*/
```

统计 C++保留字

2.8 节中讨论了 C++中以结构形式实现的记录类型,为说明记录类型,我们编写一个程序计算在 C++程序中保留关键字"else","for","if","include"和"while"出现的次数,程序中使用字符串变量以及记录数组。

程序的主要数据结构是结构 KeyWord,其域由串变量 keyword 和整数 count 组成

```
struct KeyWord
{
    char keyword[20];
    int count;
};
```

程序建立了包含 5 个保留关键字的表放在数组 KeyWordTable 中,对表中每一项指定保留关键字和计数值,对其进行初始化,例如,第一对数组初始化值{"else",0}使得表项 KeyWordTable[0]中包含串"else",其计数值为 0:

```
KeyWord KeyWordTable[ ] =
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{
    {"else", 0}, {"for", 0}, {"if", 0}, {"include", 0}, {"while", 0}
}
```

程序用函数 GetWord 从文件中逐个读出单字，单字就是以字母开头后跟以任意数量的字母或数字的任何字符序列，例如，如果文件中有这么一行：

"Expression: 3 + 5 = 8 (N1 + N2 = N3)"

GetWord 会从中提取出单字"Expression"，"N1"，"N2"和"N3"，并舍弃其他符号，函数 SeqSearch 对表进行扫描以寻找关键字的匹配值，若找到，函数返回匹配记录的索引，然后将其 count 域值增 1。

程序 2.5　统计保留关键字

本程序读取其自身的源代码作为输入用一个循环读出每个单字，并调用SeqSearch以确定输入是否与 KeyWordTable 中的保留关键字相匹配，若是，则将记录的 count 域值增 1，输入完成以后，打印出每个关键字的出现次数，程序中用来动态计算数组 KeyWordTable 中元素个数的语句很有趣，它用的是以下表达式：

sizeof(KeyWordTable) / sizeof(KeyWord)

这个表达式提供了一种与机器无关的计算数组元素个数的方法，如果其它关键字被加入到表中，再编译时会产生新的元素计数值。

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>        // 使用串函数时必须 include
#include <ctype.h>         // 提供 isalpha 和 isdigit 两个函数
#include <stdlib.h>
// 说明 KeyWord 的结构
struct KeyWord
{
        char keyword[20];
        int  count;
};
// 说明并初始化 KeyWord Table 表
KeyWord KeyWordTable[] =
{
    {"else", 0}, {"for", 0}, {"if", 0},{"include", 0}, {"while", 0}
};
// 查找字符串的顺序查找算法
int SeqSearch(KeyWord *tab, int n, chr *word)
{
    int i;
    // 扫描数组。将 word 和当前记录中的 keyword 进行比较
    for (i = 0; i < n; i++, tab++)
        if (strcmp(word,tab->keyword) == 0)
          return i;       // 若两值匹配,返回其下标值
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    return -1;          // 否则,返回-1
}
// 从文件中摘录以字母开头并后跟字母或数字的词
int GetWord(ifstream& fin, char w[])
{
    char c;
    int i = 0;
    // 跳过非字母的输入
    while(fin.get<c> && !isalpha(c))
        ;
    // 若到文件尾,则返回0(未找到词)
    if (fin.eof())
        return 0;

    // 保存词的第一个字母
    w[i++] = c;
    // 保存该词的后续字母和数字,并用NULL结束该串
    while(fin.get(c) && (isalpha(c) || isdigit(c)))
    w[i++] = c;
    w[i] = '\0';
    return 1;        // 返回1(找到一个词)
}
    void main(void)
{
    const int MAXWORD = 50      // 每个词的最大长度
    // 说明关键字表的大小并赋初值
    const int NKEYWORDS = sizeof(KeyWordTable)/sizeof(KeyWord);
    int n;
    char word[MAXWORD], c;
    ifstream fin;
    // 打开文件:若出错则退出
    fin.open("prg2_5.cpp",ios::in | ios::nocreate);
    if (! fin)
    {
        cerr << "Could not open file 'prg2_5.cpp'" << endl;
        exit (1);
    }
    // 从文件中摘词直到文件结束
    while(GetWord(fin,word))
        // 若该词在关键字表中,则将其计数器加1
        if ((n = SeqSearch(KeyWordTable, NKEYWORDS, word)) ! = -1)
            KeyWordTable[n].count++;
    // 扫描关键字表并输出其域值
    for (n = 0; n < NKEYWORDS; n++)
        if (KeyWordTable[n].count > 0)
        {
            cout << KeyWordTable[n].count;
            cout << " " << KeyWordTable[n]. keyword << endl;
        }
    fin.close();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
/*
<程序 2.5 运行结果>
1 else
3 for
9 if
7 include
4 while
*/
```

书面作业

2.1 计算下列各二进制数的十进制值：

(a) 101　(b) 1110　(c) 110111　(d) 1111111

2.2 将下列十进制数改写为二进制形式：

(a) 23　(b) 55　(c) 85　(d) 253

2.3 在现代计算机系统中，地址通常是以 16 位或 32 位二进制值的形式在硬件一级实现的。以二进制的形式处理地址比将其转化为十进制显得更直接了当，因为二进制数书写起来比较麻烦，所以我们使用十六进制。十六进制是整数的一种重要表达方法，它与二进制数之间的转换非常容易。大多数系统软件都是以十六进制的形式处理机器地址。

十六进制数以 16 为基数，其数字为十进制的 0－15，前 10 个数字取自十进制数：0，1，2，3，…，9。10－15 之间的数字由 A，B，C，D，E 和 F 表示。16 的幂为 $16^0=1, 16^1=16, 16^2=256, 16^3=4096$，等等。若采用位置计数法(positional nctation form)，17E，48 和 FFFF8000 都是十六进制数的例子。将十六进制数转换为十进制数时，只需将其按 16 的幂次展开，就像将二进制数按 2 的幂次展开一样。

例如，按 16 的幂次展开十六进制数 $2A3F_{16}$，将其转换为十进制数。

$$
\begin{aligned}
2A3F_{16} &= 2(16^3) + A(16^2) + 3(16^1) + F(16^0) \\
&= 2(4096) + 10(256) + 3(16) + 15(1) \\
&= 8192 + 2560 + 48 + 15 = 10185_{10}
\end{aligned}
$$

将以下十六进制数转换为十进制：

(a) 1A　(b) 41F　(c) 10EC　(d) FF　(e) 10000

将下列十进制数转换为十六进制：

(f) 23　(g) 87　(h) 115　(i) 255

2.4 引入十六进制数的主要原因是它们与二进制数存在直接对应关系。他们可以紧凑地表达二进制数据和内存地址。0－15 范围内的十六进制数字可以用 4 位二进制数字表达。下表列出的二进制与十六进制数字之间的对应关系。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

十六进制	二进制	十六进制	二进制
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

以十六进制方式表示二进制数的方法如下：从二进制数的最右边一位开始，每4位一组，将其划分为若干组，如有必要，在最后一组的左侧添0。将每一组中的4位写成十六进制数字，例如：

$111100011101110_2 = 0111\ \ 1000\ \ 1110\ \ 1110 = 78EE_{16}$

若要将十六进制数转换为二进制数，只需逆向进行上述过程，将每个十六进制数字写成4位二进制数字即可。见下面的例子：

$A789_{16} = 1010\ \ 0111\ \ 1000\ \ 1001 = 1010011110001001_2$

将以下二进制数转换为十六进制：

(a) 1100　　(b) 1010 0110　　(c) 1111 0010

(d) 1011 1101 1110 0011

将十六进制数转换为二进制：

(e) 0610_{16}　　(f) $AF20_{16}$

2.5 C++中允许程序员以十六进制的方式进行输入和输出。只要将控制符"hex"，放在流中，输入和输出方式就变为十六进制的。这种方式将一直保持有效，除非用控制符"dec"使之返回到十进制方式，例如：

```
cin >> hex >> t >> dec >> u;    // t 按十六进制读入;u 按十进制
<input 100 256> t = 100 and u = 256_10
cout << hex << 100 << t << u;   // output is 64 100 100
cout << dec << 100 << t << u;   // output is 100 256 256
```

现有如下声明和可执行语句：

```
int i, j, k;
cin >> i;
cin >> hex >> j >> dec;
cin >> k;
```

(a) 如果<输入>为50 50 32，则以下语句的输出是什么？

```
cout << hex << i << "" << j << "" << dec << k << endl;
```

(b) 如果<输入>为32 32 64，则以下语句的输出是什么？

```
cout << dec << i << "" << hex << j << "" << k << endl;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

2.6 以整数 ADT 的形式写出运算符%和比较运算符!=的完整说明。

2.7 Boolean(布尔)型定义了具有值 True(真)或 False(假)的数据。某些程序设计语言定义了初级类型布尔型，并提供一系列用于处理数据的内置函数。C++中每个关系表达式都与一个布尔值相关联。

(a) 定义一个布尔型 ADT 以描述数据域及其操作。

(b) 用 C++语言构件描述以上 ADT 的一种实现。

2.8 (a) 十进制数 78 与什么 ASCII 字符相对应？

(b) 二进制数 1001011_2 与什么 ASCII 字符相对应？

(c) 字符"*"，"q"以及回车的 ASCII 代码分别是什么？用十进制和二进制数分别给出答案。

2.9 以下代码段打印出什么内容？

```
cout << char(86) << "" << int('q') << "" <<
   char(int("0") + 8) << endl;
```

2.10 试解释为什么运算符%(求余数)不以实数 ADT 的形式给出。

2.11 将下列定点二进制数转换成十进制：

(a) 110.110

(b) 1010.0101

(c) 1110.00001

(d) 11.111...111...(提示：用几何级数的求和公式。)

2.12 将下列定点十进制数转换为二进制：

(a) 2.25

(b) 1.125

(c) 1.0875

2.13 (a) 在实数 ADT 中，是否存在最小的正实数？为什么？

(b) 在计算机上实现实数时，是否存在最小的正实数？为什么？

2.14 IEEE 浮点格式单独存储数的符号位，并将指数和尾数以无符号数的形式存放。任一浮点数的规范化的格式表示是唯一的。

规范化格式：将浮点数调整为其二进制小数点左边仅有 1 个非零数字：

$$N = \pm 1.d_1d_2\cdots d_{n-1} \times 2^e$$

浮点数 0.0 在存储时，其符号位，指数和尾数都为 0。下面是将两个二进制数转换为规范化格式表示的例子

二进制数	规范化格式
1101.101×2^1	1.1011010×2^4
0.0011×2^6	1.1×2^3

32 位浮点数则用内部 IEEE 格式存储：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

符号位　　最左边的 1 位用来表示符号。符号位为 0 表示"+"，符号位为 1 表示"-"。

指数　　指数按 8 位存储，为确保所有指数都以正数(无符号数)形式存储，IEEE 用"盈 127"计数法表示指数，将实际指数增加 127，得到 Exp_s。

$$Exp_s = Exp + 127$$

实际指数范围	存储指数范围
$-127 \le Exp \le 128$	$0 \le Exps \le 255$

尾数　　假设以规范化格式存储数，头 1 个数字被隐藏。小数部分以 23 位尾数形式保存。实际精度为 24 位。

符号位	指数	尾数
1 bit	8 bits	23 bits

例如，求 -0.1875 的内部表示：

规范化格式　　$(-1)\ 1.100 \times 2^{-3}$

符号位　　1

指数　　$Exp_s = -3 + 127 = 124 = 01111100_2$

尾数　　<1>1000000…0

-0.1875 = 10111110010000000000000000000000

用 32 位 IEEE 浮点计数法表示下面的数：

(a) 7.5　　(b) -1/4

以下 32 位 IEEE 格式化的数所对应的十进制值是多少？每个数均以十六进制的形式给出。

(c) C1800000　　(d) 41E90000

2.15 (a) 按日历顺序列出一年中名字带"r"的月份，此为枚举类型。

(b) 用 C++ 实现枚举类型。

(c) 在 C++ 实现中，哪个月份与整数 4 对应？十月(October)在什么位置？

(d) 按字母顺序写出枚举值，有没有在两个表中具有相同位置的月份？

2.16 在枚举类型的 ADT 中增加后继和前趋操作，使用完整的说明，后继返回表中下一项，而前趋则返回前一项，要注意定义在表的尽头会发生什么。

2.17 现有如下声明和语句，请指出这些语句执行完后 X，Y 和 A 的内容：

```
int X = 4, Y = 7, * PX = &X, * PY;
double a[ ] = {2.3, 4.5, 8.9, 1.0, 5.5, 3.5}, * PA = A;
PY = &Y; (* PX) - - ; * PY + = * PX; PY = PX;
* PY = 55; * PA + = 3.0; PA + + ; * PA + + = 6.8;
PA + = 2;
* + + PA = 3.3;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

2.18 (a) A 的声明如下：

```
short A[5];
```

数组 A 被分配了多少字节？如果数组 A 的地址是 6000，请计算 A[3]和 A[1]的地址。

(b) 假设有如下声明

```
long A[ ] = {30, 500000, -100000, 5, 33};
```

如果一个长字(long word)是 4 个字节且 A 的地址为 2050。

地址 2066 的内容是什么？将地址 2050 和地址 2062 的内容加倍，写出数组 A。A[3]的地址是什么？

2.19 假设 A 是一个 m×n 的数组，其行下标为范围为 0 到(m-1)，列下标范围为 0 到(n-1)，假设元素按列存放，编写一个存取函数以计算 A[row][col]的地址。

2.20 A 的声明如下：

```
short A[5][6];
```

(a) 数组 A 被分配了多少字节？

(b) 若数组 A 的地址为 A=1000，请计算 A[3][2]和 A[1][4]的地址。

(c) 数组的哪一项位于地址 1020？1034 呢？

2.21 (a) 声明一个初始值是你的名字的串 NAME。

(b) 现有以下串变量声明

```
char s1[50], s2[50];
```

并有输入语句如下

```
cin >> S1 >> S2;
```

对于输入行“George flies!”，S1 和 S2 的值分别是多少？

若输入以下文本(< >是空格符，1 是行尾)，则 S1 和 S2 的值分别是多少？

```
Next1
< > < > < > < > < >Word
```

2.22 假设有串声明如下：

```
char S1[30] = "Stockton, CA", S2[30] = "March 5, 1994", *p;
char S3[30];
```

(a) 执行完以下各语句后，*p 的值分别是多少？

```
p = strchr(S1, 't');
p = strchr(S1, 't');
p = strchr(S2, '6');
```

(b) 执行完下列语句后 S3 的值是多少？

```
strcpy(S3, S1);
strcat(S3, ", ");
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
strcat(S3, S2);
```

(c) 函数调用 strcmp(S1,S2)的返回值是多少?

(d) 函数调用 strcmp(&S1[5], "ton")的返回值是多少?

2.23 函数

```
void strinsert(char * s, char * t, int i);
```

将串 t 插入到 s 中，起始位置为下标 i,如果 i 大于 s 的长度，则不进行插入,用 C++库函数 strlen, strcpy 和 strcat 实现 strinsert,有必要声明一个临时串变量来保存 s 中从下标 i 到下标 strlen(s)-1 的原始字符,可以假定这个尾巴不超过 127 个字符。

2.24 函数

```
void strdelete(char * s, int i, int n);
```

删除 s 中从下标 i 开始的连续 n 个字符。如果 i 大于或等于 s 的长度，则不删除任何字符,如果 i+n 大于或等于 s 的长度，则从下标 i 开始将串的尾部删除,用 C++库函数 strlen 和 strcpy 实现 strdelete。

2.25 对使用 NULL 终结的串的一种替代方法是将字符计数值放在字符数组的头一个元素处,这被称为字节计数格式，这种串通常被称为 Pascal 串，因为 Pascal 程序设计系统使用这种串格式。

(a) 假设以字节计数格式存储串，试实现函数 strcat。

(b) 函数 PtoCStr 和 CtoPStr 进行两种串格式之间的原地转换:

```
void PtoCStr(char * s); // 将 s 从 Pascal 格式转换为 C++格式
void CtoPStr(char * s); // 将 s 从 C++格式转换为 Pascal 格式
```

试实现这两个函数。

2.26 用完整的说明将赋值运算符"="加到记录 ADT 中,要仔细定义在赋值时发生的动作。

2.27 复数的格式是 x + iy，其中 $i^2 = -1$，复数在数学、物理和工程中都有重要的应用，其算术运算规则如下:

令 u = a + ib, v = c + id

$$u + v = (a + c) + i(b + d)$$

$$u - v = (a - c) + i(b - d)$$

$$u * v = (ac - bd) + i(ad + bc)$$

$$u/v = \frac{ac + bd}{c^2 + d^2} + i\left(\frac{bc - ad}{c^2 + d^2}\right)$$

用记录结构表示一个复数:

```
struct Complex
{
    float real;
    float imag;
};
```

实现下列进行复数操作的函数:

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Complex cadd(Complex& x, Complex& y); // x + y
Complex csub(Complex& x, Complex& y); // x - y
Complex cmul(Complex& x, Complex& y); // x * y
Complex cdiv(Complex& x, Complex& y); // x / y
```

2.28 将操作 FileSize 增加到流 ADT 中，它必须返回文件中的字符个数。要仔细说明使这一操作有意义的前提条件。(提示：cin/cout 如何?)

2.29 仔细区分文本文件和二进制文件。你认为有没有可能设计出这样一个程序：它以文件名为输入参数即可确定它是文本文件或二进制文件?

上机题

2.1 编写函数

```
void BaseOut(unsigned int n, int b);
```

以 b 为基数输出 n，$2 \leqslant b \leqslant 10$，分别以 2，4，5，8 和 9 为基数打印出 $2 \leqslant n \leqslant 50$ 范围内的每个数。

2.2 编写函数

```
void OctIn(unsigned int& n);
```

读入一个八进制数并将其赋值给 n，在主程序中用 OctIn 读入以下八进制数并打印出与其等值的十进制数：

7，177，127，7776，177777

2.3 编写一个程序，声明 3 个整数变量 i，j，k。按十进制输入 i 的值，按十六进制输入 j 和 k 的值，用十六进制和十进制两种方式打印出 3 个变量的值。

2.4 测试你的机器上实数表示的颗粒度，方法是对 $D = 1/10, 1/100, 1/1000 \cdots, 1/10^n$ 分别计算 $1 + D$，直到 $1 + D == 1.0$ 为止，如果你可以访问数种体系结构的机器，请在其他机器上做一下测试。

2.5 设有如下枚举类型：

```
enum DaysOfWeek {Sun, Mon, Tues, Wed, Thurs, Fri, Sat};
```

编写函数

```
void GetDay(DaysOfWeek& day);
```

以字符串方式从键盘读入某天的名称并将相应的枚举值赋给这一天，再编写函数

```
void PutDay(DaysOfWeek day)
```

将枚举值输出到屏幕上，设计一个主程序测试这两个函数。

2.6 输入一系列单字，直到文件尾，将每个字转换为 Pig - latin 语，如果单字以辅音字母开头，则将其第一个字符移到最后一个位置并在结尾增加“ay”。如果单字是以元音字母开头的，则直接在尾部加上“ay”。例如：

输入：this is simple
输出：histay isay implesay

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

2.7 文本串可以用表映射的方法进行加密，即将字母表中的每个字母唯一映射到另一个字母。例如，映射

```
abcdefghijklmnopqrstuvwxyz = = > ngzqtcobmuhelkqdawxfyivrsj
```

将“encrypt”映射为“tkzwsdf”。

编写一个程序，读入文本，一直到文件尾，并输出加密后的内容。

2.8 编写程序建立一个与上机题 2.7 中相反的映射，输入一个加密后的文件并打印其解密后的内容。

2.9 编写一个导致一个或多个数组越界的程序，使情况糟糕到导致程序“死机”，假设你不知道问题出在哪里，用你能使用的任一调试器诊断问题的起因。

2.10 修改交换排序，使之能以降序对表排序，编写一个与程序 2.4 类似的主程序对新算法进行测试。

2.11 假设有记录声明如下：

```
struct Month
{
    char name[10];          // 该月的名称
    int monthnum;           // 该月的天数
};
```

(a) 编写函数

```
void SortByName(Month months[ ], int n);
```

通过对名字进行比较(使用 C++ 函数 strcmp)，对元素类型为 Month 的数组进行排序，另外再编写一个函数

```
void SortByDays(Month months[ ], int n);
```

通过比较各个月份的天数对表进行排序。编写一个主程序，声明一个包含一年中所有月份的数组并用以上两个函数分别对它进行排序，打印出每个排好序的表。

(b) 注意通过比较月份的天数对月份表排序时有可能出现天数相等，当发生这种情况时，排序算法可以用“辅键”解决问题，编写函数

```
void Sort2ByDays(Month months[ ], int n);
```

用以下方法对表进行排序：先比较天数，如果天数相同，则通过名称比较决定其次序，在主程序中用此函数打印出按天数排序的一年中所有的月份表。

2.12 编写一个程序，读入一个文本文件并打印出标点符号(.,!?)出现次数的计数值。

2.13 用 cin.getline 读入一行字符，该字符行以单字符函数名开头，后跟以一系列“x”，中间插入字符“+”或“-”，一行的结尾可以不是“+”或“-”，使用基于数组的输出形成一个形如

```
SingleCharFuncName(x) = x**n ± x**m ±…
```

的字符串，如果指数为 1，则省略“**1”，将各个串都写入到文件“funcs.val”中，例如，以下行

```
F x x x + x x - x
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
G x x - x x x + x x x x
```

产生文件“funcs.val”，它包含

```
F(x) = x**3 + x**2 - x
G(x) = x**2 - x**3 + x**4
```

2.14 编写程序，输入一个 N x N 的整数矩阵并打印出矩阵的 trace 值，即对角元素的和。

```
Trace(A) = A[0, 0] + A[1, 1] +…+ A[N - 1, N - 1]
```

2.15 本题使用书面作业 2.27 中的结果，编写函数 f(z)对复数多项式函数求值：

$$z^3 - 3z^2 + 4z - 2$$

对以下 z 值求上述多项式的值：

```
z = 2 + 3i, - 1 + i, 1 + i, 1 - i, 1 + 0i
```

注意后 3 个值是 f 的根。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第3章 抽象数据类型和类

3.1 用户类型类
3.2 类的举例
3.3 对象和信息传递
3.4 对象数组
3.5 多构造函数
3.6 应用举例：三角矩阵
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 1 章简单介绍了抽象数据类型及其用 C++ 类的表示，并描述了类的数据封装及信息隐藏的性质。本章将深入学习类的基本概念，包括类构造函数的设计和使用、类方法的实现、类和其它数据结构的关系。为使读者更好地理解类，本章提供了许多类的实例及其在完整程序中的应用。精心选择的 ADT 用来说明抽象结构与其类定义之间的关系。

3.1 用户类型类

类是用户定义的包含数据项及函数(方法)的数据类型。我们把数据项和函数称为类的成员，对象是类类型的变量。类通过将数据成员定义为私有、保护和公共部分来建立对其成员的不同访问级别。对象的私有部分仅供类中的函数使用，公共部分可供外部程序单元使用，但该对象应在外部程序单元的作用范围内(如图 3.1 所示)，保护成员供派生类使用，我们将在第 12 章讨论继承时介绍。

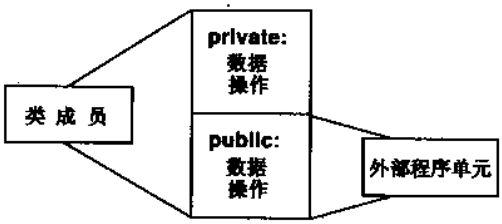


图 3.1 对象成员的访问

类的声明

类声明从类头开始，它包含保留字 class，其后是类的名称。其成员在类体中给出。类体由花括号({ })括起，以分号结束。保留字 public 和 private 分开类的成员，它们后面跟冒号(:)。数据成员按 C++ 变量格式说明，方法按 C++ 函数格式说明。下面是类定义的一般形式：

```
class <类名>
{
    private:
    //<私有数据>
    //<私有方法定义>
    //…
    public:
    //<公有数据>
    //<公有方法定义>
    //…
};
```

一般情况下，我们建议将类的数据成员放在私有段，这样，数据只能由成员函数修改，可防止应用程序对数据的非法修改。

例 3.1

矩形类

对几何算法来说，矩形由其长和宽决定，有了它们我们就可计算矩形的周长和面

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

积。数据参数长和宽及其操作组合在一起构成了矩形图形的抽象数据类型。我们把描述该 ADT 作为练习留给读者，而直接设计出实现该 ADT 的 C++ 类 Rectangle。该类包括构造函数和一系列方法——GetLength，PutLength，GetWidth，PutWidth——来访问私有的数据成员。以下是类 Rectangle 的定义：

```
class Rectangle
{
private:
  // 矩形对象的长度和宽度
     float length,width;
public:
   // 构造函数
   Rectangle(float l = 0, float w = 0);
   // 存取和修改私有数据的方法
   float GetLength(void) const;
   void PutLength(float l);
   float GetWidth(void) const;
   void PutWidth(float w);

   // 计算并返回矩形的测量值
   float Perimeter(void) const;
   float Area(void) const;
}
```

注意方法 GetLength，GetWidth，Perimeter 及 Area 的参数表后都有关键字 const。这定义了这些函数为常量函数，它们不改变类的任何数据。换句话说，执行定义为常量的方法不会改变对象 Rectangle 的状态。

类 Rectangle 的私有和公共部分明显分开。如果在开始时没有指定访问方式，则类定义中这些成员的缺省方式为私有，直到第一次遇到 public 或 protected 关键字为止。C++ 允许程序员交错使用私有、保护和公共部分，虽然并不提倡这种风格。

构造函数

与类同名的函数称为类的**构造函数**。与其它 C++ 函数一样，构造函数可以有用来初始化一个或多个数据成员的参数。在类 Rectangle 中，构造函数的名字为 Rectangle，并接受分别用来初始化对象的长和宽的参数 l 和 w，当没有显式给出参数时，其缺省值为 0。由于其方法仅给出了函数说明，例 3.1 只是类的定义，类的实现还需用 C++ 代码来实现每个函数。

对象声明

类声明描述了新的数据类型。类的**实例**通过声明该类型的对象产生。对象定义不但使类类型的对象实体化，还自动调用构造函数来初始化某些或全部数据成员。对象参数在对象名后的括号内传递给构造函数。由于只在对象创建时被调用，构造函数没有返回类型。

```
类名        对象(<参数>);        // 参数表可空
```

例如，下述定义创建了两个类型为 Rectangle 的对象：

```
Rectangle    room(12,10);
Rectangle    t;              // 缺省参数为(0,0)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

每个对象都拥有类中定义的所有数据成员和方法，其公共成员可通过对象名与成员名之间加点(“.”)来访问。例如：

```
x = room.Area();          // 将面积 = 12 × 10 = 120 赋给 x
t.PutLength(20);          // 将矩形对象的长度赋值为 20，由于使用缺省参数，
                          // 其当前值为 0
cout < < t.GetWidth();    // 输出当前宽度，即缺省指定的 0
```

在对象 Room 的说明中，构造函数初始化长度为 12，宽度为 10，用户可通过方法 PutLength 和 PutWidth 来改变这些值。

```
room.PutLength(15);    // 将长度和宽度改为 15 和 12
room.PutWidth(12);
```

类的声明并不要求一定有构造函数。本书中并不提供使用这种用法，因为它使对象在说明时带有未经初始化的数据。例如，类 Rectangle 去掉构造函数而由用户通过公共函数来对长和宽进行赋值。有了构造函数后，可保证重要数据被正确地初始化，它可使对象初始化自己本身的数据。

类 Rectangle 包含类型为 float 的数据成员。一般来说，类成员可使用任何合法的 C + + 类型，甚至其它类。当然，类不允许将本身类型的对象作为成员。

类的实现

对类声明中的每个方法都需提供函数实现。其代码可在类体中以内部代码形式给出，也可在类体外给出。在类体外给出时，应将类名和两个冒号放在函数名之前。符号“::”称为**范围限定符**，它表明函数属于该类范围之内，它允许函数中所有语句访问类的私有成员。在类 Rectangle 中，标识符“Rectangle::”放在函数名之前。

下面是函数 GetLength()的定义，它被写在类 Rectangle 的说明之外：

```
float Rectangle::GetLength(void) const
{
    return length;    // 访问私有成员 length
}
```

注意 const 后缀在函数定义中也必须使用。

类的成员函数也可写在类体之内，此时，代码在类说明之中。因此，不需要类限定符。下面是函数 GetLength 的类内定义：

```
class Rectangle
{
   private:
     float  length;
     float  width;
     public:
     . . .
     float  GetLength(void) const    // 代码在行内给出
     {
         return(length);
     }
     . . .
};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

为强调类说明和实现之间的区别，本书通常在类体之外定义成员函数，只在少数情况下使用类内定义。

构造函数的实现

构造函数也可在类体的内部或外部实现。例如，可用以下代码定义 Rectangle 的构造函数：

```
Rectangle::Rectangle(float l,float w)
{
    length = l;
    width = w;
}
```

C++提供特殊的语法来初始化数据成员。初始化成员表是用逗号隔开的类数据成员的名称表。每个名称后面跟一个用括号括起的初始值，初始值通常是构造函数的参数，完成将参数值赋给表中相应数据成员的动作。初始化成员表置于函数头之后，并用冒号(:)与参数表隔开：

```
类名::类名(参数表):数据 1(参数 1),…,数据 n(参数 n)
```

例如，构造函数参数 l 和 w 可用下述方法赋给数据成员 length 和 width：

```
Rectangle::Rectangle(float l,float w):length(l),width(w)
{}
```

建立对象

可在对象声明中初始化另一个对象。例如，下面语句是符合语法的：

```
Rectangle square(10,10),yard = square,S;
```

对象 square 的长和宽都是 10，第二个对象 yard 通过从对象 square 拷贝初始化数据建立，对象 S 的长宽均为缺省值 0。

同一类型对象之间可自由赋值，除非用户创建自己的赋值操作。对象赋值均通过数据成员的位拷贝进行。例如，下述赋值：

```
S = yard;
```

将所有数据从对象 yard 拷贝到对象 S，此时，对象 yard 的长和宽被分别拷贝到对象 S 的长和宽。

也可通过引用构造函数创建对象。例如，声明 Rectangle(10,5)创建了一个 length = 10，width = 5 的临时对象。下述语句的赋值操作将数据从临时对象拷贝到矩形 S。

```
S = Rectangle(10,5)
```

例 3.2

1、语句

```
S = Rectangle(10,5)
cout << S.Area() << endl;
```

输出 50

2、语句

```
cout << Rectangle(10,5). GetWidth() << endl;
```

打印 5

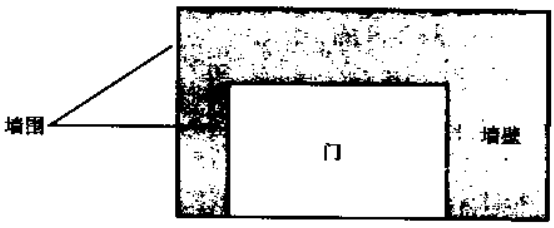
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

程序 3.1　类 Rectangle 的使用

本程序计算建造车库大门的相对费用。用户提供车库正面的尺寸，程序提供不同类型门的大小和费用。用户注意到若选择大一些的门，则建造墙围和墙壁的费用就小，这样，若木材价格固定，则建造大一些的门可能花费较少。

假设墙围环绕车库正面四周和门的四周，程序首先提示用户输入车库正面大小，然后进入用户选择不同门大小的循环，循环在用户选择"Quit"时结束。对选择的每种门，程序计算完建造车库正面的费用并输出。程序假设木墙的价格为每平方英尺 2 美元，而墙围的价格为每英尺 0.5 美元。



墙围的长度是门的周长和车库正面周长之和，而木墙的费用应是其价格乘以车库正面面积和门的面积之差。

```
#include <iostream.h>
class Rectangle
{
    private:
        // 矩形对象的长度和宽度
        float length, width;
    public:
        // 构造函数
        Rectangle(float l = 0, float w = 0);
        // 存取和修改私有数据的方法
        float GetLength(void) const;
        void PutLength(float l);
        float GetWidth(void) const;
        void PutWidth(float w);
        // 计算并返回矩形的测量值
        float Perimeter(void) const;
        float Area(void) const;
};
// 构造函数。将 l 赋值给 length,w 赋值给 width
Rectangle::Rectangle(float l, float w) : length(l), width(w)
{}
// 返回矩形的长度
float Rectangle::GetLength(void) const
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    return length;
}
// 给矩形长度赋一新值
void Rectangle::PutLength(float l)
{
    length = l ;
}
// 返回矩形的宽度
float Rectangle::GetWidth (void) const
{
    return width;
}
// 给矩表的宽度赋一新值
void Rectangle::PutWidth(float w)
{
    width = w;
}
// 计算并返回矩形的周长
float Rectangle::Perimeter(void) const
{
    return 2.0 * (length + width);    // 返回矩形周长
}
// 计算并返回矩形的面积
float Rectangle::Area(void) const    // 返回矩形面积
{
    return length * width;
}
void main(void)
{
    // 墙壁和墙围的造价固定
    const float sidingCost = 2.00, moldingCost = 0.50;
    // 循环完成标志
    int completedSelections = 0;
    // 供用户选择的门类型
    char doorOption;
    // 长/宽和门的造价
    float glength, gwidth, doorCost;
    // 包括门、墙壁、墙围的总造价
    float totalCost;
    cout < < "Enter the length nd width of the garage:";
    cin > > glength > > gwidth;
    // 根据用户提供的尺寸产生门及墙壁两个对象
    Rectangle garage(glength,gwidth);
    Rectangle door;
    while(! completedSelections)
    {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        cout << "Enter 1-4 or 'q' to quit" << endl << endl;
        cout << "Door 1 (12 by 8; $ 380)    "
             << "Door 2 (12 by 10; $ 420)" << endl;
        cout << "Door 3 (16 by 8; $ 450)    "
             << "Door 4 (16 by 10; $ 480)" << endl;
        cout << endl;
        cin >> doorOption;
        if (doorOption == 'q')
            completedSelections = 1;    // 结束循环
        else
        {
            switch(doorOption)
            {
                case '1': door.PutLength(12);    // 12x8 ( $ 380)
                          door.PutWidth(8);
                          doorCost = 380;
                          break;
            case '2': door.PutLength(12);    // 12x10 ( $ 420)
                      door.PutWidth(10);
                      doorCost = 420;
                      break;
            case '3': door.PutLength(16);    // 16x8 ( $ 450)
                      door.PutWidth(8);
                      doorCost = 450;
                      break;
            case '4': door.PutLength(16);    // 16x10 ( $ 480)
                      door.PutWidth(10);
                      doorCost = 480;
                      break;
            }
            totalCost = doorCost +
                moldingCost * (garage.Perimeter() + door.Perimeter())
                + sidingCost * (garage.Area()-door.Area());
            cout << "Total cost of door, siding, and molding: $"
             << totalCost << endl << endl;
        }
    }
}
/*
<程序 3.1 运行结果>
Enter the length and width of the garage: 20 12
Select the door by number or 'q' to quit
Door 1 (12 by 8; $ 380)    Door 2 (12 by 10; $ 420)
Door 3 (16 by 8; $ 450)    Door 4 (16 by 10; $ 480)
1
Total cost of door, siding, and molding is $ 720
Select the door by number or 'q' to quit
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Door 1 (12 by 8; $380)     Door 2 (12 by 10; $420)
Door 3 (16 by 8; $450)     Door 4 (16 by 10; $480)
q
*/
```

3.2 类的举例

我们举两个类的例子来说明 C++类的构成。类 Temperature 保持高低两个温度值的记录。在应用中,可用对象来记录水的最高温度(沸点)和最低温度(凝固点)。ADT RandomNumber 定义了可产生一系列整型或浮点型随机数的数据类型。在 C++实现过程中,构造函数可让用户输入随机数序列的种子或用程序根据系统的时间函数自动产生种子。

类 Temperature

类 Temperature 维持高低两个温度值,构造函数对两个浮点类型的私有数据成员 highTemp 和 lowTemp 赋初值。函数 UpdateTemp 带来新值并判断是否修改对象中的某个值,即如果该值是一个新的最低值,则修改 lowTemp。类似地,新的最高值将修改 highTemp。该类有两个访问数据的函数:GetHighTemp 返回高温值,而 GetLowTemp 返回低温值。

类 Temperature 的说明

声明

```
class Tmperature
{
private:
    float highTemp, lowTemp;        // 私有数据成员
public:
    Temperature (float h, float l);
    void UpdateTemp (float temp);
    float GetHighTemp (void) const;
    float GetLowTemp (void) const;
};
```

讨论

Constructor 必须给对象传递初始的高低温值。这些值可被函数 UpdateTemp 修改。函数 GetLowTemp 和 GetHighTemp 是常量函数,因为它们并不改变类中的任何数据成员。类 Temperature 的说明在文件"temp.h"中。

例子

```
// 华氏温度下水的沸点/凝点
Temperature fwater(212,32);
// 摄氏温度下水的沸点/凝点
Temperature cwater(100,0);
cout << "Water freezes at " << cwater.GetLowTemp << "C"
     << endl;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
cout << "Water boils at " << fwater.GetHighTemp << "F" << endl;
```

输出：Water freezes at 0 C
Water boils at 212 F

类 Temperature 的实现

类中的每个函数均用类范围符在类体之外实现。Constructor 将读入的初始高低温度传入并赋值给域 highTemp 和 lowTemp。这些值只可在有一新的高或低温度作为参数传入时，由函数 UpdateTemp 改变。数据访问函数 GetHighTemp 和 GetLowTemp 返回读入的高低温度。

```
// 构造函数。将 h 赋值给 highTemp,l 赋值给 lowTemp
Temperature::Temperature(float h,float l): highTemp(h)
                lowTemp (l)
{}
// 若 temp 低于低温或高于高温,则修改高温成低温值
void Temperature::UpdateTemp(float temp)
{
    if (temp > highTemp)
        highTemp = temp;
    else if (temp < lowTemp)
        lowTemp = temp;
}
// 返回最高温度
float Temperature::GetHighTemp(void) const
{
    return highTemp;
}
// 返回最低温度
float Temperature::GetLowTemp(void) const
{
    return lowTemp;
}
```

程序 3.2　Temperature 的使用

```
#include <iostream.h>
#include "temp.h"  // "temp.h" 中包含有类 Temperature
void main(void)
{
    // 初始化 temperature 的对象 today,使其 high = 70,low = 50
    Temperature today (70,50);
    float temp;
    cout << "Enter the noon temperature: ";
    cin >> temp;
    // 修改对象使其为中午时的温度
    today.UpdateTemp(temp);
    cout << "At noon: High " << today.GetHighTemp();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    cout << "  Low" << today.GetLowTemp() << endl;
    cout << "Enter the evening temperature: ";
    cin >> temp;
    // 修改对象使其为傍晚时的温度
    today.UpdateTemp(temp);
    cout << "Today's High" << today.GetHighTemp();
    cout << "  Low" << today.GetLowTemp() << endl;
}
/*
<程序 3.2 运行结果>
Enter the noon temperature: 80
At noon: High 80 Low 50
Enter the evening temperature: 40
Today's High 80 Low 40
*/
```

随机数类

许多应用需要可表示事件发生机会的随机数。如用来测验飞行员对飞机意外变故的反应的飞行模拟器、扑克牌中的洗牌和市场研究时假设客流的波动，都是计算机应用中依赖随机数的例子。计算机用**随机数发生器**来产生均匀分布在一定范围之内的随机数，该发生器用确定的算法从称为**种子**的初始数值开始产生随机数。算法对种子进行加工，产生一个数的序列。我们称它为确定的，因为它使用初始值执行固定的指令集。其输出由指令和数据唯一决定。因此，计算机产生的并不是真正的随机数，而是一个**伪随机数序列**，即数据均匀地分布在范围内。由于依赖种子，它对相同的种子产生相同的序列。这种产生相同随机序列的能力可用来模拟研究不同的策略对相同的随机数的反应情况。例如，在飞行模拟中可评估两个飞行员处理同一飞行异常的能力。当然，若选用不同种子，得到的结果也不同。这种情况的典型应用是游戏，它要求每次都提供不同的序列供人玩。

多数编译器提供实现伪随机数发生器的库函数。不幸的是，不同编译器之间区别很大，为提供一个可适用于多个系统的随机数发生器，我们创建了类 RandomNumber。这个类包括一必须由用户初始化的种子。相应初始种子值，它产生伪随机数序列。它也可自动选择种子，当没有种子传给构造函数时，它可允许用户产生独立的伪随机数序列。

随机数类的定义

声明

```
#include <time.h>
// 用当前种子产生随机数
const unsigned long maxshort = 65536L;
const unsigned long multiplier = 1194211693L;
const unsigned long adder = 12345L;
class RandomNumber
{
    private:
      // 存放当前种子的私有成员
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        unsigned long randSeed;
      public:
        // 构造函数。缺存值 0 表示系统自动给出种子
        RandomNumber(unsigned long s = 0);
        // 产生 0 < = value < = n - 1 的随机整数
        unsigned short Random(unsigned long n);
        // 产生 0 < = value < = 1.0 的随机实数
        double fRndom(void);
    };
```

说明

开始时种子为一无符号长整数。方法 Random 接受一个类型为 unsigned long 的参数 n≤65536并返回一个 16 位的短整数，范围为 0..n - 1。值得注意的是，若 Random 的返回值赋给了一个有符号的整型变量，则返回值有可能为负数，除非 $n < 2^{15} = 32768$。函数 fRandom 返回范围在 0≤fRandom() < 1.0 的浮点数。

例

```
RandomNumber rnd;          // 用系统种子产生随机数序列
RandomNumber R(1);         // 用 1 为种子产生随机数序列
Cout < < R.fRandom();      // 输出在 0 到 1 范围的实数
// 输出 5 个范围在 0 到 99 的随机整数
    for (int i = 0; i < 5; i + + )
        cout < < R.Random(100) < < " ";      // < 输出 > 93  21  45  5  3
```

例 3.3

生成随机数据

1. 骰子的点数为 1 到 6 之间(6 种可能)。我们可用函数 die.Random(6)来模拟掷骰子。将它返回的值(范围为 0..5)加 1 后转换成正确的点数。

```
RandomNumber Die;        // 用缺少种子
dicevalue = die.Random(6) + 1;
```

2. 对象 FNum 使用缺省种子产生随机数序列：

```
RandomNumber FNum;
```

为得到范围为 50≤x < 75 的浮点数，可用 f Random 生成的随机数乘以 25，将随机结果的范围从 1 个单位(0≤x < 1)扩展到 25 个单位(0≤x < 25)。再加上 50，使其范围为 50≤x < 75：

```
Value = FNum.fRandom() * 25 + 50;     // 乘 25；再加 50
```

随机数类的实现

我们用线性全等算法来产生伪随机数。它用一个大奇常数 multiplier 和 adder 以及种子一起迭代产生随机数并更新种子的值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
const unsigned long maxshort = 65536;
const unsigned long multiplier = 1194211693;
const unsigned long adder - 12345;
```

随机数序列以长整数 randSead 为初始值开始，该初始值由构造函数设置。

构造函数要让用户传递种子或用系统的时间函数来产生种子。time 函数定义在文件 <time.h> 中。用参数 0 调用时，它返回一个无符号长整数(32 位)，表示从基准时间至今已过去的秒数。目前用的基准时间包括 1970 年 1 月 1 日午夜和 1904 年 1 月 1 日午夜。任何情况下，都是一个巨大的无符号长整数：

```
// 种子生成器
RandomNumber::RandomNumber(unsigned long s)
{
    if (s = = 0)
        randSeed = time(0);     // 用系统时间作种子
    else
        randSeed = s;           // 用户提供种子
}
```

每次迭代，我们用常量来产生新的无符号长种子：

```
randSeed = multiplier * randSeed + adder;
```

这个 32 位值 randSead 的高 16 位是由乘法和加法随机("均匀地混合")产生的。算法通过将 randSead 右移 16 位产生范围为 0 至 65535 的随机数。我们可通过除法来把这个数映射到范围 0…n - 1。这就是 Random(n)的结果。

```
// 返回随机整数 0 < = value < = n - 1 < 65536
unsigned short RandomNumber::Random (unsigned long n)
{
    randSeed = multiplier * randSeed + adder;
    return (unsigned short)((randSeed > > 16) % n);
}
```

为产生浮点随机数，我们首先调用整数 Random(maxshort)，它可返回范围在 0 至 maxshort-1 的下一个随机数，将其用 double(maxshort)除后，我们得到在 0 < = fRamdom() < 1.0 的实数随机数。

```
// 返回随机实数
double RandomNumber::fRandom (void)
{
    retrun Random(maxshort)/double(maxshort);
}
```

RandomNumber 的说明及实现在文件"random.h"中。

应用：硬币投掷正面朝上的频度 我们用类 RandomNumber 来模拟重复投掷 10 个硬币。投掷时，一些硬币正面朝上，其余的背面朝上。每次投掷 10 个硬币正面朝上的个数范围为 0 到 10。仅从书本知识，你也会推测 0 个朝上和 10 个朝上的可能性相对较小。进一步来说，朝上的个数应该在一半左右，也就是 4 到 6 个。我们可通过大量(50000)次数的重复投掷来验证这个书本知识。数组 head 用来统计有 i 枚硬币正面朝上的次数，其下

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

标为 0,1,…,10,即 head[i](0 < = i < = 10)。head[i]是 50000 次重复投掷中正好有 i 枚硬币朝上的次数。

程序 3.3　频度图

投掷 10 枚硬币构成一个事件。设随机函数 Random(2)返回 0 表示正面朝下,1 表示正面朝上。函数 TossCoins 声明了一个使用自动播种的 RandomNumber 的静态对象 CoinToss。由于是静态对象,每次调用 TossCoins 都用独立的随机数序列的下一个值。通过积累 10 个 CoinToss.Random(2)产生的值来实现投掷 10 枚硬币。返回的结果使数组 head 的相应统计值加 1。

程序的输出是硬币朝上个数的频度图。x 坐标表示硬币朝上的次数,y 坐标为硬币枚数。对每个下标 i,i 枚朝上的概率为:

head[i]/float(NTOSSES)

我们用这个值在列 1 至列 72 之间的相对位置处放一"*"号,这样的结果图是对坐标系的一个近似。

```
# include < iostream.h >
$ include < iomanip.h >
# include "random.h"    // 引入随机数发生器
// 投掷 numberCoins 并返回正面朝上的次数
int TossCoins(int numberCoins)
{
    // 用于随机投掷硬币
    static RndomNumber coinToss;
    int i, tosses = 0;
for (i = 0; i < numberCoins; i + + )
        // Random(2) = 1 表示正面朝上
        tosses + = coinToss. Random(2);
    return tosses;
}
void main(void)
{
    // 每次投掷的硬币数及投掷次数
    const int NCOINS = 10;
    const long NTOSSES = 50000;
    // heads[0]为 0 个硬币正面朝上,heads[1]为 1 个硬币朝上,余此类推
    long i, heads[NCOINS + 1];
    int j, position;
    // 初始化投掷数组
    for (j = 0;j < = NCOINS + 1;j + + )
        heads[j] = 0;
    // 投掷 50,000 次并将结果记录在数组 heds 中
    for (i = 0;i < NTOSSES;i + + )
        heads[TossCoins(NCOINS)] + + ;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 输出频度图
    for (i = 0;i < NCOINS + 1;i + +)
    {
        position = int(float(heads[i])/flot(NTOSSES) * 72);
        cout < < setw(6) < < i < < "  ";
        for (j = 0;j < position - 1;j + +)
            cout < < " ";
        // '*'位置表示有'1'枚硬币朝上的次数
        cout < < '*' < < endl;
    }
}
/*
{程序 3.3 运行结果}
     0  *
     1  *
     2    *
     3        *
     4              *
     5                *
     6              *
     7        *
     8    *
     9  *
    10  *
*/
```

3.3 对象和信息传递

对象是数据类型的实例，因此，它可作为函数参数传递或作为函数的返回值，和C++其它类型一样，对象参数可以值或地址传递。本节用类 Temperature 作例子来说明这些概念。任何类类型都可作为函数的返回值类型。例如，函数 SetDailyTemp 以读入的温度数组为参数，它从中取得最高和最低温度，并返回一个带有这两个极端温度的 Temperature 对象。

```
Temperature SetDailyTemp(float reading[], int n)
{
    // 第一次读入的 t 值同时作为最高值和最低值
    Temperature t(reading[0], reading[0]);
    // 必要时修改高温或低温值
    for (int i = 1;i < n;i + +)
        t.UpdateTemp(reading[i]);
    // 返回 t 中当天的最高温度和最低温度
    return t;
}
```

数组 reading 包含 6 个温度值，为判定最高和最低温度，可调用 SetDailyTemp 并将结果赋值给对象 today，可用函数 GetHighTemp 和 GetLowTemp 打印这两个温度。

```
float reading[6]  {40,90,80,60,20,50};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Temperature today = SetDailyTemp(reading,6);
  cout < < "Today's high and low temperatures are "
       < < today.GetHighTemp() < < " and "
       < < today.GetLowTemp() < < endl;
```

对象作函数参数

对象可以值或地址作为参数传递给函数。下面用类 Temperature 来说明其语法。

函数 TemperatureRange 通过值调用 Temperature 类型参数 T,并返回高温与低温之差。函数执行时,调用部件拷贝一个 Temperature 对象(实际参数)到 T。

```
float TemperatureRange(Temperature T)
{
    return T.GetHighTemp() - T.GetLowTemp();
}
```

函数 Celsius 以引用方式传递 Temperature 参数 T,其功能为将 T 中的原来的华氏温度转变为摄氏温度。

```
void Celsius(Temperature& T)
{
    float hi, low;
    // c = 5/9 * (f - 32)
    hi = float(5)/9 * (T.GetHighTemp() - 32);
    low = float(5)/9 * (T.GetLowTemp() - 32);
    T = Temperature(hi,low);
}
```

例如,Water 对象以水的沸点(212℉)和凝固点(32℉)作为其高低温度值。TemperatureRange 表明,水的温度范围是 180℉。用函数 Celsius 可将水温转换为摄氏温度,再调用 TemperatureRange 可得出水的温度范围是 100℃。

```
Temperature Water(212,32);  // 沸点为 212F; 凝固点为 32F
cout < < "The temperature range for water in Fahrenheit is"
     < < TemperatureRange(Water) < < endl;
Celsius(Water);             // 将华氏温度转换为摄氏温度
cout < < "The temperture range for water in Celsius is"
     < < TemperatureRange(Water) < < endl;
```

3.4 对象数组

数组元素的类型不但可以是原始类型如整型或字符型,也可以是用户定义的类类型。对象的数组可用来作链表等。需要引起注意的是对象的用法。数组的定义对表中的每个对象调用构造函数。比较单个对象 Rectangle 的简单定义和 100 个 Rectangle 对象的数组。在每次定义中,都调用构造函数来创建对象,对其长度和宽度赋初值。在数组中,对 100 个对象的每个都调用构造函数。

```
Rectangle pool(150,100);    //创建一个 150 × 100 的池
Rectangle room[100];        // 对 room[0]至 room[99]调用构造函数
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

对象 pool 的定义将初始值传给了构造函数。对象 room 也有初值，因为 Rectangle 的构造函数将缺省值 0 赋给每个对象的长度和宽度。

```
Rectangle(float l = 0,float w = 0);    // 缺省参数
```

数组声明后，每个对象 room 的长度和宽度均为 0。

```
cout < < room[25]. GetLength()    // 输出 0;
cout < < room[25]. GetWidth()     // 输出 0;
room [25].PutLength(10)           // 置 room[25]的长度为 10
room [25].PutWidth(10)            // 置 room[25]的宽度为 5
```

Rectangle 对象的声明给数组和类带来了一个重要问题，如果 Rectangle 类的构造函数没有缺省参数，则 room 数组的声明将出错，这是因为每个构造函数必须带参数。该声明需要一个初始化表以处理数组中的所有元素。例如，若要声明一个含 100 个元素的 room 数组并将 length 和 width 参数置为 0，那么就需要一个包含 100 个 Rectangle 对象的初始化表。实际上，这很不实用。

```
Rectangle room[100] = {Rectangle(0, 0),…, Rectangle(0, 0)};
```

若要声明一个对象数组，我们提供一个带缺省值的构造函数或不带参数的构造函数。

缺省构造函数

"缺省构造函数"就是不需要参数的构造函数。即构造函数不带任何参数或每个参数都有缺省值。在本章中，Rectangle 类包含一个缺省的构造函数，而在声明对象时 Temperature 类需要参数。

类 Rectangle

构造函数

```
Rectangle(float l = 0, float w = 0);
```

构造函数包含缺省值都为 0 的参数 l 和 w。当建立 Rectangle 数组时，每个对象都被赋以缺省值。

```
Rectangle R[25];    // 每个元素值均为 Rectangle(0,0)
```

类 Temperature

构造函数

```
Temperature(float h, float l);
```

类 Temperature 不带缺省构造函数。但对象必须被赋以高温和低温的初始值。以下的 today 和 week 对象的声明无效！

```
Temperature today;        // 无效;缺参数
Temperature week[7];      // Temperature 没有缺省的构造函数
```

3.5 多构造函数

前面的类中，我们介绍了使用缺省值的构造函数和有初值的构造函数。由于我们所举的例子都是单个的构造函数，也许这些讨论会让读者误以为它们是互相排斥的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

C++考虑到用户多种方法初始化对象的需要，允许用户在同一个类中定义多个构造函数。创建对象时，编译器用函数重载来选择构造函数的正确形式（函数重载的概念及规则将在第6章讨论）。多构造函数极大地增加了类的灵活性。使用最多的是包含动态数据成员的复制构造函数，我们将在第8章介绍。

类Date表明了多构造函数的使用，该类共有三个数据成员，分别表示某一时期的年、月、日。

月	日	年
$1 \le m \le 12$	$1 \le d \le 31$	$1990 \le y \le 1999$

我们设计一对应其三个数据成员有三个参数的构造函数，其功能是初始化变量。第二个构造函数允许用户以"月/日/年"（mmddyy）格式定义日期，它读入该串并将字符对"mm"转换为月，"dd"转换为日，"yy"转换为年。对每个构造函数，假定指定年的参数是一范围为00－99的两位数字，实际存放的年份为该值加上1900。

```
year = 1900 + yy
```

类Date有一输出函数，它将日期按月名，日期，年份的格式打印输出。例如，20世纪的第一天为：

```
January 1,1900
```

类Date定义

声明

```
#include <string.h>
#include <strstream.h>
class Date
{
    private:
        // 指定日期的私有成员
        int month, day, year;
    public:
        // 构造函数。缺省日期为1990年1月1日
        Date(int m=1, int d=1, int y=0);
        Data(char *dstr);
        // 以"月日年"格式输出日期
        void PrintDate(void);
};
```

说明

两个构造函数均创建一个Date对象，但所用参数不一样。编译器在创建Date对象时选定构造函数。例如，下述Date对象中，

例

```
Date day1(6,6,44);          // 1994年6月6日
Date day2;                  // 缺省为1990年1月1日
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Date day3("12/31/99");          // 1999 年 12 月 31 日
```

类 Date 实现

类 Date 的关键为两个构造函数，它们分别通过给定年、月、日的值或"mm/dd/yy"格式串定义日期。

第一个构造函数有对应 1900 年 1 月 1 日的三个缺省参数，可作为缺省构造函数

```
// 构造函数。月、日、年分别以整数 m,d,y 给出
Date::Date (int m, int d, int y) : month(m), day (d)
{
    year = 1900 + y;        // y 为 20 世纪某年
}
```

另一个构造函数以一个串作为参数。该串格式为"mm/dd/yy"。为转换相应的数据，我们用一个数组，将"mm"对应转换成整数表示的月，"dd"转换成整数的日，"yy"转换为整数的年。将参数串拷贝到 inputBuffer 数组中，然后按下述格式读入字符：

```
month - ch - day - ch - year
```

读入两个 ch，从输入串中去掉了分隔符"/"。

```
// 构造函数。月，日，年以书"mm/dd/yy"格式给出
Date::Date(char * dstr)
{
    char inputBuffer[16];
    char ch;
    // 将串拷贝至 inputBuffer 中，定义一个基于数组的流 input
    strcpy(inputBuffer,dstr);
    intrstream input(inputBuffer, sizeof(inputBuffer));
    // 从流 input 中读入年，月，日，用 ch 过滤掉字符'/'
    input > > month > > ch > > day > > ch > > year;
    year + = 1900;
}
```

对于输出，Print 方法给出了完整的日期串，包括月的名称，日和年份。数组 months 中存放有一个空串(下标 0)和 12 个日历月的名称。月的整数值作为下标从数组中取出月名并与输出。

```
// 用完整的月名打印日期
void Date::PrintDate(void)
{
    // 建立存放月名的静态数组
    static char * Months[] = {"","January","February",
                              "March","April","May",
                              "June","July","August",
                              "September","October",
                              "November","December"};
cout < < Months[month] < < "" < < day < < ", "< < year;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

程序 3.4　20 世纪日期

下述测试程序用构造函数建立起示例中的对象，并打印出结果日期。类 Date 的定义在文件“date.h”中。

```
# include < iostream.h >
# include "date.h"          // 引入类 Date
void main (void)
{
    // 分别用整数,缺省值和串给出的 Date 对象
    Date day1(6,6,44);      // 1944 年 6 月 6 日
    Date day2;              // 1900 年 1 月 1 日
    Date day3("12/31/99");  // 1999 年 12 月 31 日

    cout < < "D-Day in World War II -";
    day1.PrintDate();
    cout < < endl;
    cout < < "The first day in the 20th century - ";
    day2.PrintDate();
    cout < < endl;
    cout < < "The last day in the 20th century - ";
    day3.PrintDate();
    cout < < endl;
}
/*
< 程序 3.4 运行结果 >
D-Day in World War II - June 6, 1944
The first day in the 20th century - January 1, 1900
The last day in the 20th century - December 31, 1999
*/
```

3.6　应用举例：三角矩阵

二维数组，通常也叫矩阵，是数学上的一种重要的数据结构。本节中我们讨论行数和列数相等的方阵，开发类 TriMat，它定义的是对角线以下的元素值全为 0 的上三角矩阵。

$$A = \begin{bmatrix} A_{00} & A_{01} & \cdots & \cdots & A_{0n-1} \\ 0 & A_{11} & A_{12} & \cdots & A_{1n-1} \\ 0 & 0 & A_{22} & \cdots & A_{2n-1} \\ 0 & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & A_{n-2n-2} & A_{n-2n-1} \\ 0 & 0 & \cdots & 0 & A_{n-1n-1} \end{bmatrix}$$

上述方阵可用数学形式表示为方阵 A，其元素 $A_{ij}=0, j<i$。上三角矩阵由元素 A_{ij} 定

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

义，其中 $j \geq i$。它具有可用于方程求解的重要代数性质。在类 TriMat 中实现上三角矩阵操作表明，上三角矩阵可有效地用一维数组来表示。

上三角矩阵性质

若上三角矩阵中有 n^2 个元素，则其中大约半数元素为 0，并不需要显式地存放。如果我们从 n^2 个元素中去掉对角线的 n 个元素，则剩下元素中一半为 0。例如，若 $n=25$，则有 300 个元素值为 0。

$$(n*n-n)/2=(25*25-25)/2=(625-25)/2=300$$

下面是三角矩阵的运算集。我们定义了加、减、乘和行列式运算，这些都在方程求解中有着重要的应用。

两个三角矩阵 A 和 B 的和与差为矩阵中对应元素的和与差，其结果矩阵还是三角矩阵。

加法　　C = A + B

C 也为三角矩阵，其元素值 $C_{i,j}=A_{i,j}+B_{i,j}$。

减法　　C = A − B

C 也为三角矩阵，其元素值 $C_{i,j}=A_{i,j}-B_{i,j}$。

$$C=\begin{bmatrix}1&3&2\\0&4&7\\0&0&5\end{bmatrix}+\begin{bmatrix}2&1&9\\0&8&2\\0&0&1\end{bmatrix}=\begin{bmatrix}3&4&11\\0&12&9\\0&0&6\end{bmatrix}$$

乘法　　C = A * B

乘积矩阵 C 是一个三角矩阵，$C_{i,j}$的值是根据 A 的第 i 行和 B 的第 j 列计算出来的：

$$C_{i,j}=(A_{i,0}*B_{0,j})+(A_{i,1}*B_{1,j})+(A_{i,2}*B_{2,j})+\cdots+(A_{i,n-1}*B_{n-1,j})$$

例如，如果

$$A=\begin{bmatrix}1&1&0\\0&2&1\\0&0&2\end{bmatrix},\quad B=\begin{bmatrix}3&-2&4\\0&4&1\\0&0&3\end{bmatrix}$$

$C_{0,2}$是 A 的第 0 行和 B 的第 2 列的乘积的和

$$\begin{bmatrix}1&1&0\\&&\\&&\end{bmatrix}*\begin{bmatrix}&&4\\&&1\\&&3\end{bmatrix}=\begin{bmatrix}&&5\\&&\\&&\end{bmatrix}$$

$$1*4+1*1+0*3=5$$

A 和 B 的乘积为

$$\underset{A}{\begin{bmatrix}1&1&0\\0&2&1\\0&0&2\end{bmatrix}}*\underset{B}{\begin{bmatrix}3&-2&4\\0&4&1\\0&0&3\end{bmatrix}}=\underset{C}{\begin{bmatrix}3&2&5\\0&8&5\\0&0&6\end{bmatrix}}$$

对于一般的正方矩阵，其行列式的值计算起来很复杂；然而，计算三角矩阵的行列

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

式的值则很简单，只要计算出对角线上元素的乘积即可。

$$\det(A)=\begin{vmatrix} A_{00} & A_{01} & \cdots & \cdots & A_{0n-1} \\ 0 & A_{11} & A_{12} & \cdots & A_{1n-1} \\ 0 & 0 & A_{22} & \cdots & A_{2n-1} \\ 0 & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & A_{n-2n-2} & A_{n-2n-1} \\ 0 & 0 & \cdots & 0 & A_{n-1n-1} \end{vmatrix}=A_{00}A_{11}\cdots A_{n-1n-1}$$

三角矩阵的存储 标准的数组定义需要全部 n^2 个内存位置，尽管我们可以算出对角线以下存储多少个零，为了节省这一部分空间，我们将三角矩阵中的各项存储到一维数组 M 中，主对角线以下的各项不再被存储。表 3.1 中示意了各行所存储的项数。

表 3.1 按行存储三角矩阵

行	存储数据个数	存储的数据
0	n	$(A_{00}\cdots A_{0,n-1})$
1	n－1	$(A_{11}\cdots A_{1,n-1})$
2	n－2	$(A_{22}\cdots A_{2,n-1})$
…	…	……
n－2	2	$(A_{n-2,n-2}\cdots A_{n-2,n-1})$
n－1	1	$(A_{n-1,n-1})$

存储算法需要一个存取函数，它必须可以确定存储了元素项 $A_{i,j}$ 的数组 M 中的位置，若 j < i，元素 $A_{i,j}$ 为 0，它未存放在 M 中。若 j > = i，存取函数利用一直到 i 行的各行所存储的元素个数的信息，对每一行 i 都可以计算这一数据，将其存放到一个数组(rowTable)中，以供存取函数使用。

```
行    RowTable                        说明
0     rowTable[0] = 0                 第 0 行前的 0 项
1     rowTable[1] = n                 第 1 行(从第 0 行开始)前的 n 项
2     rowTable[2] = n + n-1           第 2 行前的 n + n-1 项
3     rowTable[3] = n + n-1 + n-2     第 3 行前的所有项
.......
n-1   rowTable[n-1] = n + n-1 +… + 2
```

例 3.4

现有 3×3 矩阵 X

$$X=\begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

```
行    RowTable                        说明
0     rowTable[0] = 0                 第 0 行前的 0 项
1     rowTable[1] = 3                 从第 0 行开始的 3 项
2     rowTable[2] = 5                 第 0 和 1 行的 5 项
```

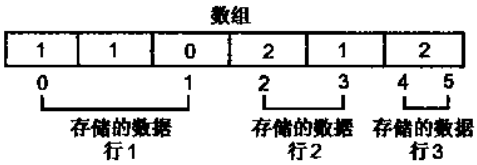

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Row Table

0	3	5
0	1	2

三角矩阵中的各项被按行存储在数组 M 中。



假设三角矩阵中的所有元素按行存储于数组M中，则存取$A_{i,j}$的函数用到以下参数：

索引 i 和 j

数组 rowTable

存取 $A_{i,j}$的算法如下：

1. 若 j < i，则 $A_{i,j}$ = 0，不存储该项。
2. 若 j > = i，则取 rowTable[i]的值，即数组 M 中存储的一直到第 i 行的所有元素项的个数；在第 i 行，前 i 项是零，不在 M 中存放，元素项 $A_{i,j}$ 的位置是 M[rowTable[i] + (j - i)]。

例 3.5

考察例 3.4 中的三角矩阵 X[3][3]：

1. $X_{0,2}$ = M[rowTable[0] + (2 - 0)]
   = M[0 + 2]
   = M[2] = 0
2. $X_{1,0}$不作存储
3. $X_{1,2}$ = M[rowTable[1] + (2 - 1)]
   = M[3 + 1]
   = M[4] = 1

类 TriMat 类 TriMat 实现了多种三角矩阵的操作，三角矩阵的减法和乘法被留作本章末尾的习题，由于只能使用静态数组的限制，类中限制行和列的长度都不超过 25，总共有 300 = $(25^2 - 25)/2$ 个 0 元素，所以数组 M 中共有 325 个元素。

类 TriMat 的说明

声明

```
#include <iostream.h>
#include <stdlib.h>
// 上三角矩阵存储元素个数及行数的最大值
const int ELEMENTLIMIT = 325;
const int ROWLIMIT = 25;
class TriMat
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    private:
      // 私有数据成员
        int rowTable[ROWLIMIT];  // M 中各行的起始下标
        int n;                   // 行/列数
        double M[ELEMENTLIMIT];  // 存放上三角中的元素
    public:
      // 构造函数,无缺省参数
        TriMat(int matsize);
        // 访问矩阵元素方法
        void PutElement(double item, int i, int j);
        double GetElement(int i, int j) const;
        // 矩阵的算术运算
        TriMat AddMat(const TriMat& A) const;
        double DetMat(void) const;
        // 矩阵的 I/O 操作
        void ReadMat(void);
        void WriteMt(void) const;
        // 取矩阵维数
        int GetDimension(void) const;
};
```

说明

类的构造函数所需要的参数是矩阵的行和列的尺寸,方法 PutElement 和 GetElement 存储和检索上三角矩阵的元素,GetElement 对所有下三角元素均返回 0 值,AddMat 返回矩阵 A 和当前对象的和,它不改变当前矩阵的值,I/O 操作 ReadMat 和 WriteMat 要用到全部 n×n 个矩阵元素,对于 ReadMat,仅仅上三角元素被保存。

例

```
#include "trimat.h"                  // 引入类 TriMat
TriMat A(10), B(10), C(10);          // 10×10 上三角矩阵
A.ReadMat();                         // 输入矩阵 A 和矩阵 B
B.ReadMat();
C = A.AddMat(B);                     // 计算 C = A + B
C.WriteMat();                        // 输出 C
```

TriMat 类的实现

构造函数用参数 matsize 初始化数据成员 n,这就确定了矩阵的行和列的尺寸,用同样的函数可以初始化用于存取矩阵各元素项的 rowTable,如果 matsize 超出 ROWLIMIT,则产生出错消息,程序终止。

```
// 初始化 n 及 rowTable
TriMat::TriMat(int matsize)
{
    int storedElements = 0;  // 记录存放元素的个数
    // 若 matsize 超出 ROWLIMIT,程序退出
    if (matsize > ROWLIMIT)
    {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        cerr << "The matrix cannot exceed size" << ROWLIMIT
                "x" << ROWLIMIT << endl;
        exit(1);
    }
    n = matsize;
    // 建立数组 rowTable
    for (int i = 0; i < n; i++)
    {
        rowTable[i] = storedElements;
        storedElements += n - i;
    }
}
```

矩阵存取方法 三角矩阵的关键问题是我们能否将非零元素高效地保存在线性数组中，为取得这种高效率并能使用通常的二维索引 i 和 j 来存取元素，我们需要函数 PutElement 和 GetElement 来存放和检索数组元素，实用函数 GetDimension 供客户程序访问矩阵的维数，这一数据可以用来保证存取方法所带的参数对应于有效的行和列：

```
// 返回矩阵维数
int TriMat::GetDimension(void) const
{
    return n;
}
```

方法 PutElement 检查下标 i 和 j，如果 j >= i，我们使用三角矩阵的矩阵存取函数将数据值存放在 M 中。如果 i 或 j 不在 0..(n-1)的范围内，程序终止：

```
// 在数组 M 中存放矩阵元素[i,j]
void TriMat::PutElement(double item, int i, int j)
{
    // 若元素下标越界，退出程序
      if ((i<0||i>=n)||(j<0||j>=n))
    {
        cerr << "PutElement: index out of range 0 to"
             << n-1 << endl;
        exit (1);
    }
    // 忽略对角线下所有元素
    if (j>=i)
            M[rowTable[i]+j-i]=item;
}
```

若要检索一个数据项，方法 GetElement 检查下标 i 和 j，如果 i 或 j 不在 0.(n-1)的范围内，则程序终止，如果 j < i，则该项在矩阵的值为 0 的下三角区内，这时，GetElement 返回未保存的值 0，反之，当 j >= i 时，GetElement 就能从数组 M 中检索出该数据项：

```
// 从数组 M 中取出矩阵元素[i,j]
double TriMat::GetElement(int i, int j) const
{
    // 若元素下标越界，退出程序
    if ((i<0||i>=n)||(j<0||j>=n))
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    {
        cerr << "PutElement: index out of range 0 to"
             << n-1 << endl;
        exit (1);
    }
    if (j >= i)
      // 若在对角线之上,则返回该元素值
      return M[rowTable[i]+j-i];
    else
      // 若在对角线之下,返回0
      return 0;
}
```

TriMat 对象输入/输出 按常规方法，在输入矩阵时，一次输入一行数据，行和列数据都要输入完整，对于 TriMat 对象，下三角区的值是 0，它们不保存在数组中 但是，为了与通常的矩阵输入一致，用户仍然需要输入这些 0 值。

```
// 按行读入矩阵元素。用户必须输入所有 n×n 个元素
void TriMat::ReadMat(void)
{
    double item;
    int i, j;
    for (i=0;i<n;i++)                // 扫描行
        for (j=0;j<n;j++)            // 对每行扫描列
        {
            cin >> item;             // 读入矩阵元素[i,j]
            PutElement(item,i,j);    // 存储该元素
        }
}
// 逐行输出矩阵元素
void TriMat::WriteMat(void) const
{
    int i,j;
    // 定点输出浮点数,保留3位小数,不足补0
    cout.setf(ios::fixed);
    cout.precision(3);
    cout.setf(ios::showpoint);
    for (i=0;i<n;i++)
    {
        for (j=0;j<n;j++)
          cout << setw(7) << GetElement(i,j);
        cout << endl;
    }
}
```

矩阵运算 Trimat 类可以计算两个矩阵的和以及矩阵的行列式的值，AddMat 方法只要带一个参数，它就是求和式中的右操作数，而当前对象与左操作数相对应，例如，三角矩阵 X 和 Y 的和使用对象 X 中的 AddMat 方法，假设和保存在对象 Z 中，若要计算

Z = X + Y

可使用下列语句：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Z = X.AddMat(Y);
```

将两个 TriMat 对象相加的算法返回新矩阵 B，其元素 $B_{i,j}$ = $CurrentObject_{i,j}$ + $A_{i,j}$：

```
// 返回 A 和当前对象之和。当前对象不变
TriMat TriMat::AddMat(const TriMat& A) const
{
    int i,j;
    double itemCurrent, itemA;
    TriMt B(A,n);     // 和存放于 B 中

    for (i=0;i<n;i++)       // 逐行计算
    {
        for (j=i;j<n;j++) // 忽略对角线之下的元素
        {
            itemCurrent=GetElement(i,j);
            itemA=A.GetElement(i,j);
            B.PutElement(itemCurrent+itemA,i,j);
        }
    }
    return B;
}
```

方法 DetMat 返回当前对象的行列式的值，返回值是对角元素的乘积，它是一个实数，关于具体实现，请参考补充的软件资料。

程序 3.5　TriMat 操作

下面的测试程序示意了 TriMat 类的 I/O 操作以及矩阵的加法和求行列式值的操作，程序的每一部分均用字符串的形式提供描述性的语句：

```
#include <iostream.h>
#include <iomanip.h>
#include "trimat.h"       // 引入类 TriMat
void main(void)
{
    int n;
    // 给定矩阵大小
    cout << "What is the matrix size?";
    cin >> n;
    // 定义三个 n×n 矩阵
    TriMat A(n), B(n), C(n);
    // 读入矩阵 A 和 B
    cout << "Enter a" << n << "x" << n
         << " triangular matrix" << endl;
    A.ReadMat();
    cout << endl;
    cout << "Enter a" << n << "x" << n
        " triangular matrix" << endl;
    B.ReadMat();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    cout << endl;
    // 执行运算并打印结果
    cout << "The sum A + B" << endl;
    C = A.AddMat(B);
    C.WriteMat();
    cout << endl;
    cout << "The determinant of A + B is " << C.DetMat() << endl;

/*
<程序 3.5 运行结果>
What is the matrix size? 4
Enter a 4 by 4 triangular matrix:
1 2 -4 5
0 2 4  1
0 0 3  7
0 0 0  5
Enter a 4 by 4 triangular matrix:
1 4 6  7
0 2 6  12
0 0 3  1
0 0 0  2
The sum  A + B
  2.000  6.000  2.000  12.000
  0.000  4.000 10.000  13.000
  0.000  0.000  6.000   8.000
  0.000  0.000  0.000   7.000
The determinant of A + B is 336.000
*/
```

书面作业

3.1 为 n 个硬币的集合定义 ADT Coins，其数据包括硬币个数，最后一掷时反面朝上的硬币个数，以及最后一次投掷时正面朝上的硬币清单，操作包括初始化，投掷硬币，返回正面的个数以及打印最后一掷时正面朝上的硬币。

3.2 (a) 定义箱子(box)的 ADT，包括初始化，返回各边长度以及计算面积和体积的的操作。

(b) 编写类 Box，实现上述 ADT。

(c) 箱子的围长(girth)是指由两条边所形成的矩形的周长，一个箱子有 3 种可能的围长值，邮寄长度取决于围长加上第三条边的长度，如果一个包裹的任一邮寄长度小于 100 则准予邮寄，编写一段代码决定对象 B 是否可以邮寄。

3.3 找出以下类声明中的所有句法错误：

(a)
```
class X
{
    private    int t;
    private    int q;
    public
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            int X (int a, int b);
            {     t = a; q = b;
            }
            void printX(void);
    }
(b) class Y
    {
        private:
            int p;
            int q;
        public
            Y (int n, int m) : n(p) q(m);
            {
            }
    };
```

3.4 (a) 给出类 X 的完整声明，它包括：

私有成员：整数变量 a，b 和 c

公共成员：将值赋给 a，b，c 的构造函数，缺省值是 1，返回 a，b，c 的最大值的函数 F

(b) 写出(a)中类 X 的构造函数。

(c) 编写公共函数 F 的代码，将其定义在类外。

3.5 假设有以下声明：

```
class Student
{
    private:
        int studentid;
        int gradepts, units;
        flot gpa;
        float ComputeGPA(void);
    public:
        Student(int studid; int studgradepts, int studunits);
        void ReadGradeInfo(void);
        void PrintGradeInfo(void);
        void UpdateGradeInfo(int newunits, int newgradepts);
};
```

此类维护的是学生的成绩记录，变量 gradepts 和 units 用于 ComputeGPA 以将学生的等级平均分赋值给变量 gpa，用以下公式

```
gpa = gradepts/units
```

写出成员函数的代码，构造函数和 ComputeGPA 必须是内联(in-line)的。

3.6 ADT Calendar 包含了数据项 year 和逻辑数据值 leapyr，其操作如下：

构造函数	初始化 year 和 leapyr。
NumDays(mm, dd)	返回从该年第一天至给定月(mm)，日(dd)的天数。
Leapyear(void)	指示某年是否是闰年。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

PrintDate(ndays)　　按格式 mm/dd/yy 将日期 ndays 打印到 year 中。

(a) 写出此 ADT 的正规定义。

(b) 实现作为类的 ADT Calendar。

3.7 写出具有下列数据成员的类的声明，并为该类型的对象声明适当的操作。

(a) 学生的姓名，专业，预期毕业年份，等级平均分(gpa)

(b) 州，首府，人口，面积，州长

(c) 一个圆柱体，允许修改半径和高度，并可计算表面积和体积。

3.8 下面是表示一摞牌的类的声明。

```
class CardDeck
{
    private:
        // 牌用从 0 至 51 的整数数组实现
        int cards[52];
        int currentCrd;
    public:
        // 构造函数。洗牌
        CardDeck(void);
        // 洗牌
        void Shuffle(void);
    // 返回下一张牌。提示：先建立起牌的位置关系
    int GetCard(void);
        // 梅花在 0 - 12,方块在 13 - 25,红心在 26 - 38,黑桃在 39 - 51。每个花色中,第一张
           牌为"A",最后三张牌为"J"、"Q"、"K"。将牌 C 输出为"花色 牌值"
    void PrintCard(int c);
};
```

(a) 实现类方法，提示：洗牌时，用一个循环扫描 0..51 范围内的牌，对于第 i 张牌，在 i 到 51 范围内选择一个随机数并将具有此随机数下标的牌与下标 i 对应的牌相交换。

(b) 编写函数

```
void DealHand(CardDeck& d, int n);
```

从 d 中发 n 张牌，对它们进行排序并打印出其面值。

3.9 利用 3.2 节中的 Temperature 类，编写函数

```
Temperature Average(Temperature a[], int n);
```

它返回包含 n 个读数的平均低温和平均高温的 Temperature 对象。

3.10 使用 3.5 节中的 Date 类。

(a) 修改 Date 类，使其包括方法 IncrementDate，它所带参数是 0 到 365 之间的正的天数值，将此值加到当前日期上，返回具有新日期的对象。

(b) 允许 IncrementDate 的参数为负值。

3.11 示意随机数类的使用，使其模拟以下情况：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(a) 一个州有 1/5 的汽车没有满足尾气排放标准，用 fRandom 判断一辆随机抽出的汽车是否满足标准。

(b) 在一个群体中，个人的重量在 140 到 230 磅之间，用 Random 在此群体中选择某个人的重量。

3.12 考察类 Event，事件的上限时间和下限时间作为初始值被传递给它，边界缺省值为 0 和 1，操作如下：

构造函数　初始化数据边界，如果下界超过上界，则打印出错信息并退出程序。

GetEvent　在下界到上界的范围内取得一个随机事件。

(a) 用内联代码实现此类。

(b) 在类声明外定义成员函数，实现此类。

(c) 一个应用程序需要一个含 5 个 Event 对象的的数组，每个对象的事件都在 10 到 20 范围内发生，如何初始化数组？如果每个对象的范围都在 0 到 1 之间，问题是否简单一些？

3.13 编写函数 DateInterval，它接受书面作业 3.6 中的两个 Calendar 类对象，返回两个日期之间的天数。

3.14* 我们可以用矩阵来解含 n 个未知数的方程组，我们示意 3 个未知数情况下的算法，元素项 $A_{i,j}$ 是方程组中未知数 X_0，X_1 和 X_2 的系数，方程的右边是元素项 C_i。

$$A_{0,0}X_0 + A_{0,1} * X_1 + A_{0,2}X_2 = C_0$$
$$A_{1,0}X_0 + A_{1,1} * X_1 + A_{1,2}X_2 = C_1$$
$$A_{2,0}X_0 + A_{2,1} * X_1 + A_{2,2}X_2 = C_2$$

上面的方程式可以用矩阵方程

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} C_0 \\ C_1 \\ C_2 \end{bmatrix}$$

表示，以上矩阵被称为系数矩阵，例如，方程组

$$\begin{aligned} 1X_0 + 1X_1 + 0X_2 &= 4 \\ -3X_0 - 1X_1 + 1X_2 &= -11 \\ 2X_0 + 2X_1 + 2X_2 &= 14 \end{aligned}$$

对应于矩阵方程

$$\begin{bmatrix} 1 & 1 & 0 \\ -3 & -1 & 1 \\ 2 & 2 & 2 \end{bmatrix} * \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 11 \\ 14 \end{bmatrix}$$

数学理论告诉我们，方程组可以被简化成一系列其系数矩阵为三角矩阵的等价方程，在上述例子中：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

1. 消去项 $A_{10} = -3$。

用常数 3 去乘第 0 行中的各项，然后将第 0 行中的项加到第 1 行中。

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 2 & 2 & 2 \end{bmatrix} * \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 14 \end{bmatrix}$$

2. 消去项 $A_{20} = 2$。

用常数 -2 去乘第 0 行中的各项，然后将第 0 行中的各项加到第 2 行中，这一过程同时也消去了项 A_{21}。

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix} * \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 6 \end{bmatrix}$$

新方程组的矩阵方程的系数矩阵是三角矩阵。

(a) 对下列方程进行化简，使其包含三角矩阵。

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & -1 & 1 \\ 2 & 4 & 2 \end{bmatrix} * \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 10 \end{bmatrix}$$

(b) 求系数矩阵的行列式的值。

上机题

3.1 许多应用程序都要用到需要不断进行更新的累加器，作为抽象数据类型的一个简单例子，假设累加器是由加法操作进行更新，由打印操作进行输出的数据类型。

```
ADT Accumulator is
    Data
        一个作为总和的实数值
    Operations
    Initialize
        Input:                  实数值 N。
        Preconditions:          无
        Process:                将总和值赋值为 N。
        Postconditions:         总和被初始化。
        Output:                 无
    Add
        Input:                  实数 N。
        Preconditions:          无
        Process:                将 N 加到总和上。
        Postconditions:         总和被更新。
        Output:                 无
    Print
        Input:                  无
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        Preconditions:          无
        Process:                读取总和。
        Postconditions:         无
        Output:                 打印总和。
end ADT Accumulator
```

某银行交易应用程序读取初始余额以及一系列交易，负交易表示借方，正交易表示贷方，程序用到 3 个 Accumulator 类型的对象，由初始余额定义的 Balance 对象作为构造函数的参数，对象 Debits 和 Credits 的初始值为 0，用于维护借方和贷方交易的动态的总和，Add 运算符更新对象中的和。

读入一系列交易，在交易值为 0.00 时终止，打印余额、借方和贷方的最终值。

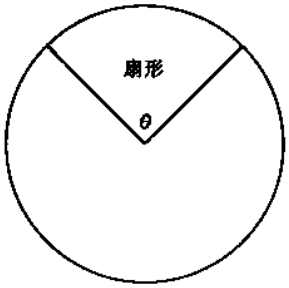
3.2 用书面作业 3.5 中实现的类编写一段主程序，数据值如下：

学号	等级分	学分单位
1047	120	40
3050	75	20
0020	100	75

(a) 打印每个学生的信息。

(b) 最后一个学生(学号 0020)另有夏季学期的成绩，用以下新数据更新其记录：等级分 40，10 个学分单位，打印该学生的新数据。

3.3 扩展 1.3 节的 Circle 类的定义，扇形的面积由公式$(\theta/360) * \pi r^2$决定，用此类解决以下问题：



一个圆形游乐场被定义为半径为 100 英尺的对象，程序确定给游乐场围上围墙的成本，围墙造价是每英尺 $2.40，游乐场的表面多为草，但有一 30°角的扇形区域不是草坪，程序确定以每 2×8 幅面(16 平方英尺)铺草坪的成本。

```
围墙(Fencing)
    Cost = Circumference * 2.40
草坪(Lawn)
    Lawn_Area = Area - Sector_Area
    Number_Rolls = Lawn_Area / 16
    Cost = Number_Rolls * 4.00.
```

3.4 编写一个类，它包括性别指示(M 或 F)，年龄和 0 到 1000 范围内的身份证号，操作包括 Read，Print 以及返回对象中存储的身份证号，年龄和性别的函数 GetId/

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

GetAge/GetGender。

编写一个定义了对象 YoungWomen 和 OldMen 的应用程序，程序用 Read 操作输入若干人的信息并将数据赋值给 YoungWomen 中的最年轻的女子以及 OldMen 中最年长的男子，输入在身份证号为 0 时终止，用 Print 从这些对象中输出数据。

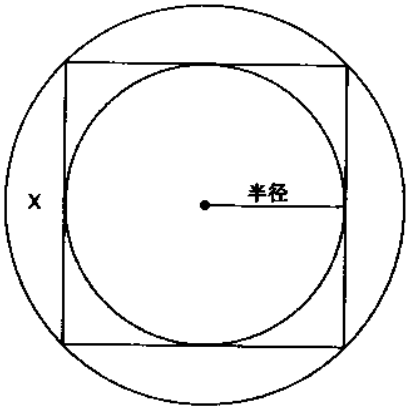
3.5 实现类 Geometry，其私有数据包含两个类型为 double 的数据成员 V1 和 V2 以及类型为 Figure 的变量 figuretype。

类中应包括两个构造函数，它们应分别带 1 个和 2 个参数，类中还包括返回对象周长的方法 Border，返回面积的方法 Area 以及计算对角线的方法 Diagonal。

```
Enum Figure {Circle, Rectangle}
class Geometry
{
    private:
        double V1, V2;
        Figure figuretype;
    public:
        Geometry(double radius);          // 圆
        Geometry(double l, double w);     // 矩形
        double Border(void) const;
        double Area(void) const;
        double Diagonal(void) const;
}
```

(a) 用外部函数实现类 Geometry，圆的构造函数将有一个参数，因而应将 Circle 赋值给 figuretype，另外一个构造函数应将 Rectangle 赋值给 figuretype，度量方法将不得不先检查一下 figuretype 再计算返回值。

(b) 用户输入将用来生成小圆的内径，用此数据声明一个内接矩形和内切圆，打印出每个对象的周长，面积和对角线长。



计算两个圆之间的环(在大圆之内，小圆之外)的面积。

计算标以 X 的小块区域的周长。

3.6 类 Ref 统计“提交”给它的正数和负数的个数，构造函数不带参数，它把数据项 posi-

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

tiveCount 和 negativeCount 初始化为 0。

```
Class Ref
{
    private:
        int positiveCount;
        int negativeCount;
    public:
        Ref(void);
        void Count(int x);
        void Write(void) const;
};
```

Ref 型的对象被传递给一个函数，该函数用它来记录 5 个整数的输入序列中正数和负数的个数，我们用两种版本的函数来示意用值和引用传递对象的区别，第 1 个版本用值传递对象：

```
void PassByValue(Ref V)
{
    int num;
    for (int i = 0; i < 5; i + +)
    {
        cin > > num;
        if (num ! = 0)
            v.Count(num);
    }
}
```

函数的第 2 个版本通过引用传递参数。

```
void PassByReference (Ref &V)
{
    int num;
    ...
}
```

实现 Ref 并编写主程序调用每个函数，用方法 Write 打印出结果，每个实例的数据均为 1，2，3，－1 和－7，详细解释为什么 PassByValue 不能正确工作，而 PassByReference 却取得成功。

3.7 有类声明如下：

```
class Grade
{
    private:
        char name[30];
        float score;
    public:
    Grade(char student[], float score);
        Grade(void);
        int Compare(char s[]);
        void Red(void);
        void Write(void);
};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

按(a)到(e)的各步骤编写一个主程序。

(a) 写出成员函数的实现代码，用字符串函数 strcpy 为第 1 个构造函数中的 name 赋值，注意第 2 个构造函数是缺省构造函数，它将 name 设为 NULL 串，score 置为 0.0，Compare 在 s 与 name 相等时返回 1，否则返回 0，用 strcmp。

(b) 在主程序中，声明一个含 5 个对象的数组 Students，其初始值为

```
{Grade("John", 78.3), Grade("Sally", 86.5),
  Grade("Bob", 58.9), Grade("Donna", 98.3)};
```

(c) 第 5 个对象，Students[4]，将由成员函数 Read()进行输入。

(d) 编写函数

```
int Search(Grade Arr[], int n, char keyname[]);
```

搜索 n 个元素的数组 Arr 并返回其 name 值与键值 keyname 匹配的对象的索引，如果在 Arr 中未找到 keyname 则返回 -1。

(e) 用不同的名字调用 Search 三次以证实它能正确工作。

3.8 用书面作业 3.8 中设计的类 CardDeck 玩下面的被称为 Hi - Low 的游戏，发 5 张牌，对其中每张牌都询问玩家：从牌堆中剩下的牌中随机抽取一张比这一张牌大还是小？每种花色最大者为爱司(Ace)，花色的顺序是从梅花到黑桃，打印猜测成功的次数。

3.9 本题使用书面作业 3.6 中所设计的 Calendar 类，编写函数

```
int DayInterval(Calendar C, int mm1, int dd1, int mm2, int dd2);
```

返回两个日期之间的天数。写一个主程序，完成以下任务：

(1) 打印出当前年份是否闰年的信息。

(2) 用 NumDays 确定从一年的开始直到圣诞节的天数。

(3) 将(2)的结果传递给 PrintDate 并验证是否打印出了正确日期。

(4) 计算从今天开始直到圣诞节的天数。

(5) 计算 2 月 1 日到 3 月 1 日之间的天数。

3.10 将第 1 章中的 Dice 类扩展为掷 n 个骰子，n < = 20，如果骰子的数目未提供给构造函数，则其默认值为 2，用 Dice 类解以下问题：

声明一个含 30 个骰子对象的数组，将每个元素初始化为掷 5 次骰子，你将需要使用声明中的缺省构造函数，并用循环将每个元素初始化为掷 5 次骰子。

对每个元素执行 Toss。

扫描表并求出 5 或 12 被掷出多少次。

求下次投掷时某总和值被重复几次，例如，8 8 8 被计作重复两次。

扫描表，找出最大的总和值，显示骰子的正面。

按骰子的计数值对表排序并打印出总和。

3.11 声明枚举类型

```
enum unit {metric, English};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类 Height 中包含以下私有数据成员：

```
char name[20];
unit measureType;
float h;    // 英制或公制高度
```

操作包括

```
// 构造函数：参数名，高度，计量单位
Height(char nm[], float ht, unit m);
PrintHeight(void);    // 按给定单位输出高度
// 读入名称，计量单位及高度值
ReadHeight(void);
float GetHeight(void);    // 返回高度值
void Convert(unit m);     // 将 h 转换为 m 制
```

(a) 实现上述类，1 英寸等于 2.54 厘米。

(b) 编写一个函数，扫描表，将所有表项转换为由参数给出的度量单位，假设对象是用其他度量单位给出的。

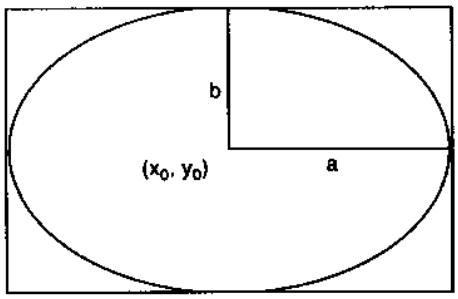
(c) 编写一个对 Height 对象的数组进行排序的函数，假定每个对象都使用相同的度量单位。

(d) 编写一个函数，扫描数组，返回表示身材最高者的对象，假定所有对象都使用相同的计量单位。

(e) 写一个程序，建立一个含 5 个 Height 类型元素的表，测试上述类，在声明中初始化前 3 个并读入后 2 个，可利用(b)，(c)，(d)中设计的函数。

3.12 某银行只有一个柜员，它注意到：客户交易时间一般在 5 到 10 分钟的范围内，在开门时已经有 10 个客户排起了队，利用书面作业 3.12 中设计的类 Event 计算柜员为 10 个客户服务的时间。

3.13 椭圆形或卵形是由边长分别为 2a 和 2b 的外切矩形所定义的。



常数 a 和 b 被称为椭圆的半轴，两个半轴相等的椭圆即为圆，椭圆的数学方程为：

$$(x - x_0)^2/a^2 + (y - y_0)^2/b^2 = 1$$

其面积为 πab，设计类 Ellipse，其成员函数由构造函数和 area 方法组成，用 Ellipse 和类 Rectangle 解决以下问题：

在一个 80ft × 60ft 的矩形区域内要建一个椭圆形的游泳池，其长度分别为 30

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

和 40，池子的成本为 $ 25 000，池子外面要铺设水泥，其成本为 $ .50/ft^2，试计算总造价。

3.14 棒球运动员的数据包括运动员的编号、击球次数、击中次数以及平均命中率(命中次数/击球次数)。这些资料作为类 BaseBall 的私有数据成员保存。构造函数的所有参数都有缺省值。统一编号的缺省值为 -1，而击球次数和击中次数的缺省值为 0。当接受缺省的统一编号时，我们就认为将用 ReadPlayer 读入运动员的统一编号、击球次数、命中次数。当知道统一编号以后，ReadPlayer 输入击球次数和命中次数。方法 GetBatAve 返回平均命中率。私有方法 ComputeBatAve()被构造函数和 ReadPlayer 作为实用函数使用以设置包含平均命中率的成员变量的值。方法 WritePlayer 按以下格式输出运动员的所有资料：

```
Player <UniformNo> Average <BattingAvg>
```

平均命中率按 3 位整数输出。例如，如果击中 30 次而击球次数为 100，则 WritePlayer 输出的平均命中率为 300。

类 Baseball 的声明

```
class Baseball
{
    private:
        int playerno;
        int atbats;
        int hits;
        float batave;
        // ComputeBatAve 按内部代码给出
        float ComputeBatAve(void) const // 私有方法
        {
            if (playerno = = -1 || tbats = = 0)
               return(0);
            else
               return(float(hits)/atbats);
        }
    public:
        Baseball(int n = -1,int ab = 0, int h = 0);
        void ReadPlayer(void);
        void WritePlayer(void) const;
        float GetBatAve(void) const;
};
```

实现类 Baseball，并将它用于下面的主程序：

(1) 声明 4 个对象：

Catcher	统一编号 10，击球 100 次，中 30 次
Shortstop	仅仅知道统一编号 44
Centerfielder	不知道任何资料
Maxobject	不知道任何资料

(2) 读取 shortstop 和 centerfielder 对象的必要资料。

(3) 写出 catcher，shortstop 和 centerfielder 的所有所有资料。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(4) 用操作 GetBatAve 和对象赋值方法，将具有最高平均命中率的运动员赋值给 maxobject 对象并打印其资料。

3.15 在类 TriMat 中增加三角矩阵的减法和乘法并用与程序 3.5 类似的程序测试它们。

3.16 在求解一般的 n×n 的代数方程组的时候，可以用一系列操作将问题简化为求解一个三角矩阵方程。只要其系数矩阵的行列式的值不为零，三角矩阵方程就有唯一解。通过将系数矩阵中的每一行与未知数的数组列相乘就可以得到代数方程组。按 n－1 到 0 的次序逐个解方程，就可以得到变量 $X_{n-1}, X_{n-2}, \cdots, X_1, X_0$ 的唯一解。例如，书面作业 3.14 中讨论的三角系数矩阵方程就是用此方法求解的。

方程 0　　$1X_0 + 1X_1 + 0X_2 = 4$
方程 1：　　$2X_1 + 1X_2 = 1$
方程 2：　　$2X_2 = 6$
求解 X_2：　在方程 2 中，$X_2 = 6/2 = 3$
求解 X_1：　在方程 1 中，用 3 替换 X_2；解关于未知数 X1 的方程。

$$2X_1 + 3 = 1$$
$$X_1 = -1$$

求解 X_0：　在方程 0 中，用 －1 替换 X_1，用 3 替换 X_2；解关于未知数 X_0 的方程。

$$X_0 - 1 = 4$$
$$X_0 = 5$$

最后结果为 $X_0 = 5, X_1 = -1, X_2 = 3$
综合这些思路，设计函数

```
void SolveEqn(const TriMat& A, double X[], double C[]);
```

它求出一般三角矩阵的唯一解，如果有的话。

$$\begin{bmatrix} A_{00} & A_{01} & \cdot & \cdots & A_{0n-1} \\ 0 & A_{11} & A_{12} & \cdots & A_{1n-1} \\ 0 & 0 & A_{22} & \cdots & A_{2n-1} \\ 0 & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & A_{n-2n-2} & A_{n-2n-1} \\ 0 & 0 & \cdots & 0 & A_{n-1n-1} \end{bmatrix} * \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{n-2} \\ X_{n-1} \end{bmatrix} = \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ \vdots \\ C_{n-2} \\ C_{n-1} \end{bmatrix}$$

(a) 在程序中使用 SolveEqn 解以下方程组。

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix} * \begin{bmatrix} X_0 \\ x_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 6 \end{bmatrix}$$

(b) 求解书面作业 3.14(a)中的方程组。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第4章　群体类

4.1　线性群体
4.2　非线性群体
4.3　算法分析
4.4　顺序查找与折半查找
4.5　基本的顺序表类
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在第 1 章,我们介绍了直接由程序设计语言支持的基本数据类型,包括数字、字符数据及数组、串和记录类型等。这些结构化的数据类型就是群体的例子,它存放数据并提供对数据元素的增、删、改操作。对群体类型的研究是本书的重点之一。

群体可分为两个大类:线性和非线性。图 4.1 根据访问数据的不同方法进一步细化了群体的分类,并列出了本书讨论的数据结构。本章将对图中的每种群体做一简单的总结,包括它们的数据描述,操作及其在实践中的运用。

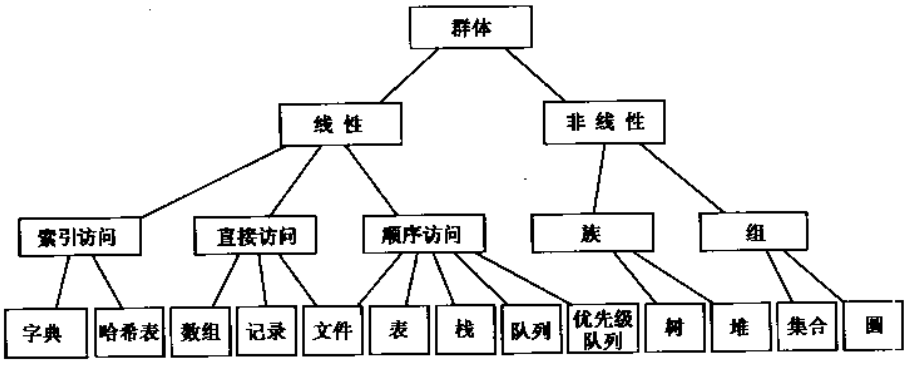


图 4.1　群体层次及路线图

线性群体中的元素按位置排列有序(如图 4.2),可区分为第一个元素,第二个元素,等等。用下标映射元素顺序的数组是线形群体的典型例子。

图 4.2　一个线性群体

非线性群体不用位置顺序来标识元素,例如,企业中职员的上下级关系链及三角球柜里的球就都是非线性群体(图 4.3)。

本章还介绍了算法性能分析,指出决定算法性能的因素并引入 O 方法来测量它。本书自始至终采用该方法来比较及对比不同的算法。

第 1 章中介绍的类 SeqList 是群体类型的基础。本章我们给出该类的基于数组的实现法。在第 9 章,我们还将用线性表来实现它。而在第 12 章,SeqList 和继承一起产生有序链表。C++用类实现群体时,编译器要求函数参数为指定的数据类型并通过类型检查来保持类型一致。为实现可生成的群体类型,我们在第 7 章介绍 C++的类 template。类 template 用一生成名如 T 来给出由群体处理的数据类型。到定义对象时,再由参数给出 T 的实际类型。Template 是 C++用来生成类定义的一个有力工具。例如,假定某群体类有一 10 元素的数组,下述说明中第一种说明定义了一个整数数组,而其 template 版并没有给定一固定的类型,而是用生成名 T 定义了数组元素的类型。其实际类型在对象定义时给出。

说明 1

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

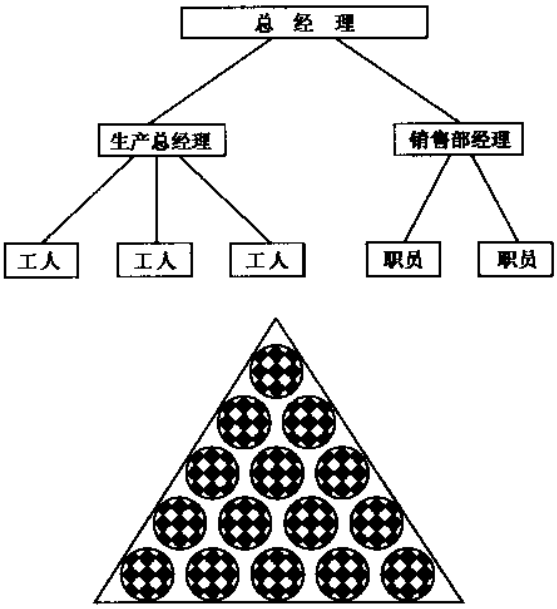


图 4.3 非线性群体

```
class Collection
{
    ...
    int A[10];          // 数据成员为整数数组
}
Collection object;      // A 为整数数组
```

说明 2

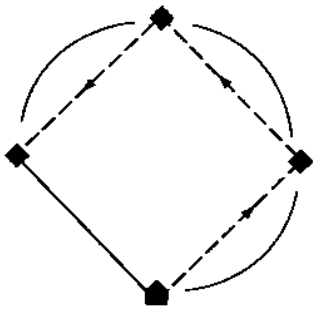
```
template < class T >
    class Collection
    {
        ...
        T A[10];                        // 数组的可生成说明
    }                                   // 在定义对象对给出 T
    Collection < int > object;          // A 为整数数组
    Collection < char > object;         // A 为字符数组
```

4.1 线性群体

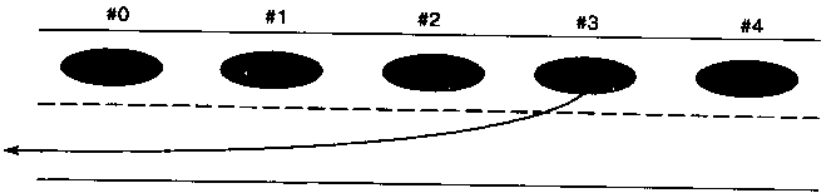
如图 4.1 所示，可通过访问元素的不同方法来区分线性群体。对可直接访问的线性群体，我们可直接选中一个元素，而不必先去访问表中该元素的前一个元素。目中的字符可以被直接访问。例如，单词 LIMEAR 的第三个字母拼写有误，其前两个字母都一点没错。我们可直接修改第三个字母，而不必先去访问前两个字母。但另一些线性类，如顺序链表，就不允许直接访问，只能从表头开始顺着链表找到指定元素。如棒球比赛中，击球手必须在触一垒和二垒后才能安全抵达三垒。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

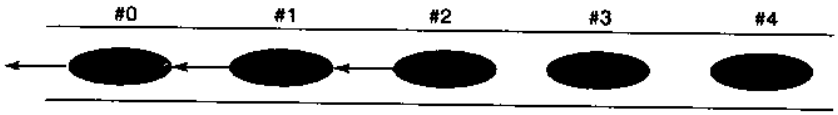


我们可以停车场为例来说明可直接访问的链表和顺序链表的区别。下图所示停车场在场地旁有行车道，车主可利用行车道直接将第三辆车开出。



直接开出第三辆车

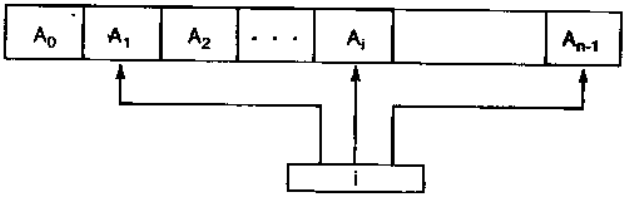
下图所示的停车场中，所有汽车单行停放，车主只能按顺序取车。若要开出第三辆车，必须按顺序移动第 0 到第 2 辆车。



顺序开出第三辆车

直接访问群体

数组是由数据类型相同的元素组成的群体，可通过整数下标直接访问。



数组群体

Data

具有相同类型的对象的群体

Operations

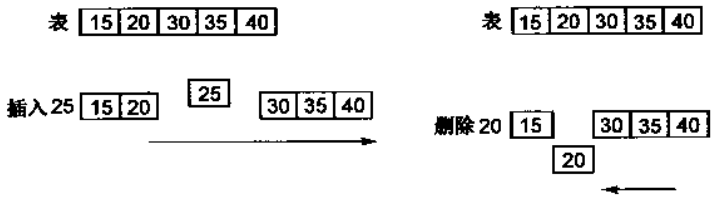
数组中任何位置的数据均可通过整数下标直接访问

静态数组中含有固定个数的元素，在编译时给定空间大小。**动态数组**可通过动态内存管理创建并可调整大小。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

数组可用来表示表。在顺序表情况下，数组允许在表尾方便地增加元素。但删除元素时并不方便，因为此时我们必须频繁地平移各元素。当表按元素大小排列时，若要插入新的元素，也必须移动元素位置。



第 8 章将介绍类 **Array**，它对数组概念做了扩充。它增加了一个检查下标是否越界的下标操作。每次存取数据之前都可用这个操作来防止下标越界。这样，用户就可用这种"安全数组"在运行时动态创建数组。

字符串，是由字符构成的数组，并包括求串长度，拼接两个串，删除子串等操作。第 8 章将介绍扩充了其它操作的更有普遍意义的类 **String**。

串群体

Data

长度已知的字符群体

Operations

包括求串长度，拷贝或拼接一个串到另一个串，比较两个串，字符串匹配及串的输入/输出操作

记录是一用来存放不同类型数据的最基本的群体结构。对许多应用来说，单一对象可由不同的数据元素共同描述。例如，飞机票包括航班号，座位号，旅客姓名，票价，承运人等数据。每一张飞机票对象都是不同类型域的集合。记录群体将这些域集中在一起，并提供对每个单独域的数据的直接访问。

记录群体

Data

具有多个类型可能不同的域的元素

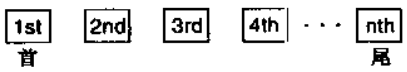
Operations

圆点(.)操作允许对域数据的直接访问

顺序访问群体

另一类用途广泛的线性群体是元素按顺序存放的表。这种线性表的结构存放着任意个数的元素。表的大小通过增加或删除表中的元素来改变。且表中元素通过其位置来访问。第一个元素位于表头，最后一个元素是表尾，除最后一个元素外，每一个元素都有唯一的后继。

表群体



Data

个数不定的同一类型的对象的集合

Operations

必须通过遍历表来访问单独的元素。从表头顺序处理每个元素直到找到该元素，插入或删除元素将改变表的大小

线性表群体可有任意个数的元素，且可随新元素的加入而扩充或随原有元素的被删

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

除而减小。线性表的局限性在于其不能随意访问某一元素，因为它没有直接访问的操作，访问表中元素需要从表头遍历表。根据线性表的实现技术，我们可有两种遍历表的方法：从左至右或双向遍历。本章我们用数组来实现顺序表。这种表的大小受数组大小的限制。我们将在第9章使用动态结构来实现没有大小限制的表。

购物单提供了一个顺序表的实例。顾客最初通过写下需购商品的名称，创建了一个表，购物时，当采购到某一商品或不再想买某商品时，顾客将其名称从表中划去。

有序线性表是一按数据值大小排列的线性表。例如，表

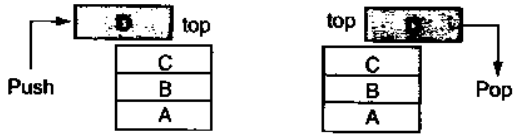
3，5，6，12，18，33

按大小排列，但表

1，6，2，5，8

不是，本章要讨论的折半查找是一用到有序表的算法。

栈和队列是两种特殊的线性表，它们访问数据元素的方法受到限制。在栈中，元素只允许在表的一端加入或删除。这端称为栈顶。餐具架上放置的盘子类似于栈。从栈中移去一个元素的操作称为弹出栈；而加入一个元素到栈中的操作称为压入栈。



增加元素时，所有当前栈中其它元素都被“压低”以给新元素腾出栈顶的位置。而当从栈中删除元素（“弹出”）时，则正好相反。最后一个被压入堆栈中的元素第一个离开堆栈。元素的这种存取方式称为后进先出方式（LIFO）。

栈群体

Data

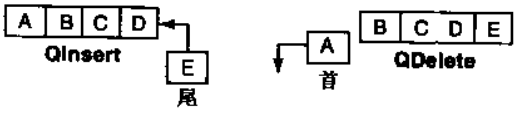
只能在表的顶端访问元素的表

Operations

该表支持压入和弹出操作，压入在表顶端增加一个新元素，弹出从表顶端删除一个元素

我们将在许多应用中用到栈，如表达式求值、递归和树的遍历等。在这些应用中，我们用 LIFO 顺序扫描和访问元素。编译器也是用栈传递参数给函数并用栈存放局部变量。

队列是只在队首或队尾访问元素的表，即只在队尾插入元素，在队首删除元素。这样，元素离开队列的顺序与其进入队列的顺序是一致的。这种方式我们称为先进先出方式（FIFO）。



队列群体

Data

只在队首或队尾访问的表

Operations

在队尾增加一个元素和在队首删除一个元素

队列可用来模拟等待，如银行和超级市场的顾客队列。在计算机应用中，操作系统用队列作并发研究和打印作业的调度。

在一些应用中，我们通过对元素授优先级来改变队列结构。当要从队列中删除元素

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

时，总是选择优先级最高的进行删除。这种群体，我们称之为优先级队列，有自己的插入和删除操作。数据插入优先级队列的位置是随意的，重要的是选择最高优先级的元素进行删除。医院中的急诊使用优先级队列，病人一般按先来后到的次序就诊，除非病人有生命危险，此时这个病人的优先级最高，并可最先得到医疗服务。

优先级队列群体

Data

每一元素都有优先级的表

Operations

往表中增加元素，删除优先级最高的元素

操作系统中使用优先级队列进行作业调度。优先级最高的作业必须首先运行。优先级队列也可用在事件驱动的模拟处理中。例如，第5章中程序实现银行中顾客流的进出。到达和离开的每种事件都插入到优先级队列中，银行首先处理时间最早的事件。

计算机系统中，**文件**是与数据结构"流"相对应的外部群体。我们把文件和它对应的"流"等同起来，并主要讨论数据流。对磁盘文件可直接存取，但磁带文件只能顺序存取。读操作从输入流中删除数据，写操作在输出流的最后增加数据。文件常用来存放大量的数据。例如，在程序编译过程中，产生的大的表经常存放到临时文件中。

文件群体

Data

和外部设备相连的字节序列，通过"流"，数据流可从外部设备中读入或写出到外部设备中

Operations

打开文件，从文件中读取数据，往文件中写数据，在文件中找指定的点(直接存取)，关闭文件

索引访问

数组是一典型群体，它可通过整数索引(下标)来直接访问每个元素。对许多其它应用，我们也常将一个可用来访问数据记录的键和数据记录相连。如向银行或保险公司查询信息时，我们给出的帐号就是用来查询此帐号记录的主键。**哈希表**就是一个将存储的数据和主键联系起来的群体，它的主键被转换成一个整数索引用来检索数据。有一种常用的哈希方法就是将整数值作为群体数组的下标。将主键转换为下标值后，就可查询到相关的数据。主键并不一定非得是整数，例如，某数据记录由姓名，工作类别，在公司工作的年限，工资等组成。这时，作为姓名的串即是该记录的主键。

"Wilson, Sndra R."
3
15
42500

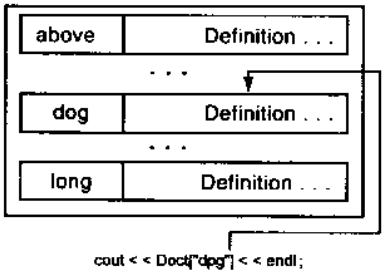
现实生活中的字典是由单词及其含义组成的群体。人们用该单词作为主键来查字典。在数据结构中，**字典**群体由一系列称为**关联**的主键－值组成的对构成。

例如，要查找的单词是主键，而说明其含义的字符串可看作为值。关联中的值可用主

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

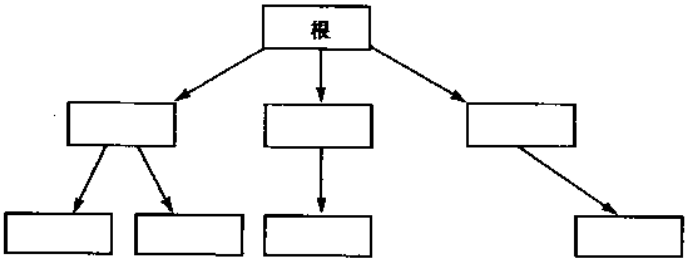
www.Linuxidc.com

键做索引直接查到。这样，字典就和数组类似，除了其下标并不一定是整数外。即，若 Dict 是字典群体，则通过 Dict["Dog"]即可查到"dog"的含义。字典经常被称为关联的数组，因为它们可将索引和数据值联系到一起。



4.2 非线性群体

如图 4.1 所示，非线性群体可分为族和组两大类。**族群体**是按层次区分的元素的集合，位于指定层次的元素可以在下层有多个后继。我们介绍树这种特殊的族群体，它的所有数据元素都从唯一的称为根的元素发源而来。树中的元素称为结点，它所指向的下层结点称为孩子。除根之外的每个结点都有唯一的父亲。树的路径为从根开始并从父亲到孩子。



树是描述具有目录和子目录的文件系统的理想数据结构。用来定义从老板(总裁)到副总裁，部门经理等上下级关系的商业组织图也是树的一种模型。

本书中，我们详细讨论树的一种特例——**二叉树**。它的每个结点最多有两个孩子。这种结构在算术表达式计算及编译理论中有着重要的应用。如果结点有序，则成为**二叉查找树**，它可有效存放大量数据。二叉查找树通过将元素置于从根结点开始的最短路径上，提供了一种快速访问元素的机制。图 4.4 所示为一有 16 个结点的树，从根到结点的最长路径为 4。如果树的枝叶繁茂程度相对不变的话，随着树的规模增大，结点数与最长路径之比将显著增大。举例来说，若某二叉查找树有 $2^{20}-1=1048575$ 个结点，并且这些结点的安放使树的层次最少，则树中任一元素均可在访问不超过 20 个结点后找到。二叉查找树中的 AVL 树，可保证结点的均匀分布和高效查找。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

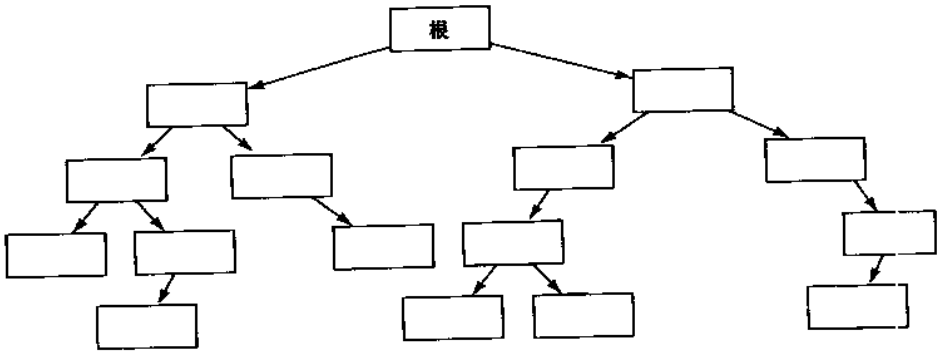


图 4.4　16 结点树

树群体

Data

结点发源于根的族群体，每一结点指向多个其本身是子树根的孩子结点

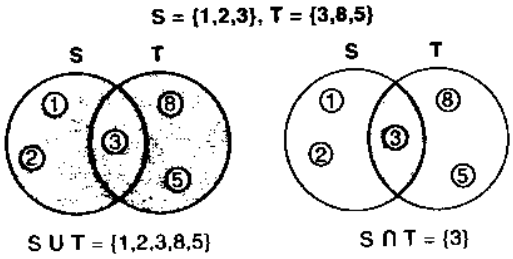
Operations

树结构允许增加或删除结点，虽然树本身是非线性结构，树遍历算法允许我们访问单个结点并查找键值。

堆是树的特殊形式，它的根结点总是由值最小的元素占据。删除操作去掉根结点，不管是插入或删除操作，都将引起树的重组，使树根是最小的元素。堆重组算法十分高效，只需扫描从根到叶子结点的最短路径。这样，它可用来对元素排列。区别于低效排序算法，我们只需建堆后重复从堆中删除根结点就可得到排序结果。我们称其为快速堆排序。同样，优先级队列也通常由堆实现。

组群体

组用来表示元素间没有任何顺序关系的非线性群体，不同元素的集合是组的例子。集合群体的操作包括并集和交集。还有判断元素是否存在子集关系等。在第 8 章，我们将介绍类 Set 及操作符重载实现的集合运算。



集合群体

Data

没有顺序关系的不同元素构成的群体

Operations

二元运算判断是否为成员，并集，交集及差集，还有测试子集关系的其它操作

集合可用于数据无序且元素各不相同的应用中。例如，民意调查机构建立电话号码

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

库进行调查，每次通过电话调查完一人后，即将该电话号码放入“已调查”集合中，并从原集合中删除。显然，尚未调查的电话也是一个集合。调查可持续到未调查集合为空或已调查集合元素个数达到一定数量为止。

图是描述点集合及联结这些点的边的集合的数据结构。它在作业调度，运输等方面有着应用。例如，建房时必须将建筑工作分解成各个阶段，这就要求施工计划的制定应保证新阶段开始时所有前序工作都已完成。比如，屋顶必须在结构完工之后才能施工，结构施工又必须在地基完工之后才能进行。

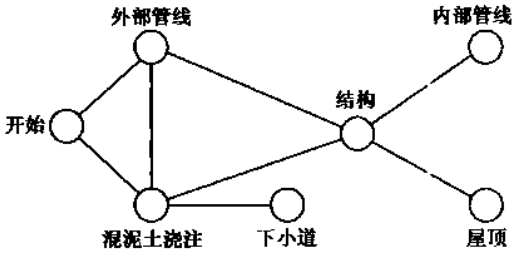
图群体

Data

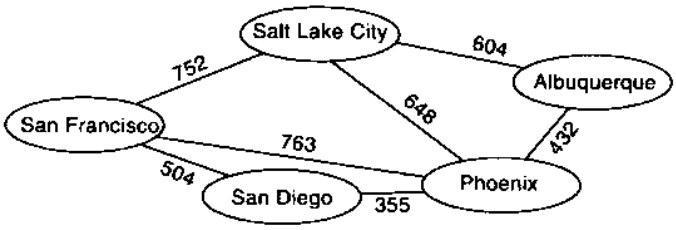
点及联结点的边的集合

Operations

作为点和边的集合，图有增加及删除这些元素的操作。图的扫描算法从指定点开始寻找所有从该点能够到达的其它点，包括深度优先和广度优先算法。



加权图是一种给每条边赋予权值的特殊图。权值即为该边在遍历图时的费用。例如，下面加权图中，点为城市名，每边的权值为两城市间的距离。



4.3 算法分析

本书中，我们用类来实现群体，其中的算法常用传统算法来实现。除详细描述算法的设计和实现外，我们常要分析这些算法的性能。

用户对程序的评价不外乎其正确性，可用性和性能几个方面。程序的可用性和正确性仅依赖于设计和测试过程。影响性能的因素却是多方面的，如计算机的硬件性能，可用内存的多少，及算法复杂度等。本书中我们忽略其它因素，将注意力集中到算法的复杂度上。我们给出一种可根据群体规模来评价算法性能的 O 方法评价系统。它独立于具体的计算机系统，只与算法本身的特性有关，并可给出算法性能的定量数据。

性能评价

算法最终都以特定指令集形式运行于计算机主机和外部设备上。对某一确定的系统，算法可利用系统硬件上的优势来获得高性能。我们称这种评价方法为系统评价法，它在同一机器上比较完成相同任务的两个或多个算法的运行速度。这样，在相同机器和相同数据环境下运行算法，可得出每种算法运行的时间，这些时间可作为每种算法的系统性能评价。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

有时，对某些算法来说，内存空间是制约其性能的重要因素。这些算法为读入初始数据可能要求大量的临时空间或被迫进行费时的磁盘交换。空间性能用来测定算法需要的内存空间大小，它可用来说明算法适合在何种计算机上运行及算法的总体系统性能。由于计算机内存容量的不断扩大，空间性能分析的重要性不断降低。

和上面两种性能评价方法不同，第三种方法主要针对算法的内部结构上，它通过分析算法的设计，包括算法中使用的比较、赋值等各种语句的次数来评价算法性能。这种方法独立于特定的计算机系统，它用算法处理的数据元素的个数 n 来衡量算法的计算复杂度。我们称这种方法为算法的计算性能，并用 O 方法给出其计量函数 $f(n)$。

O 方法

一般来说，算法的计算性能与它所处理的数据总量有关。计算性能分析主要分析算法的关键运算。这些运算与数据群体的类型、数据总数和数据的原始顺序有关。

在数组中查找最小元素是一简单算法，它的主要运算为数据元素的比较。对一 n 元数组，算法需做 $(n-1)$ 次比较，其性能指数为 n。别的算法就复杂一些，对第 2 章的交换排序算法来说，每次循环都要对所处理的数据进行多次比较。若 A 是一 n 元数组，则交换排序算法要执行 $n-1$ 次循环。图 4.5 给出了算法的运行状态图。

循环 1: 用 A[0]与元素 A[1]...A[$n-1$]各比较一次，共 $n-1$ 次，若需要的话，将元素交换使 A[0]保持最小值。

循环 2: 用 A[1]与元素 A[2]...A[$n-1$]各比较一次，共 $n-2$ 次。

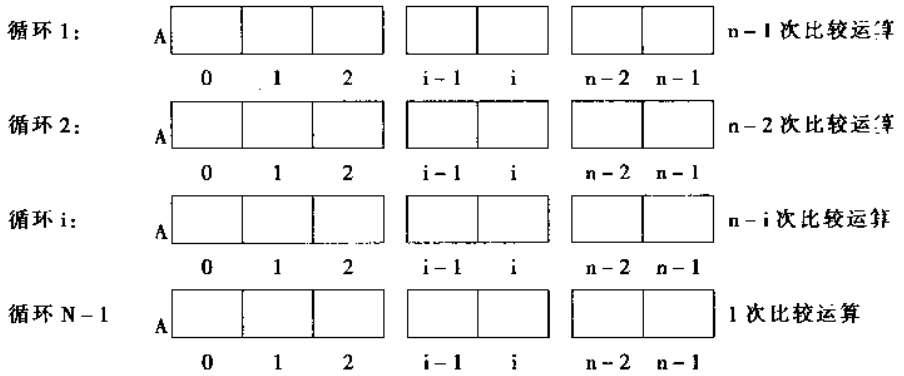


图 4.5 交换排序中的循环

循环 i: 用 A[i-1]与元素 A[i]...A[$n-1$]各比较一次，共 $n-i$ 次。

综上所述，交换排序的比较总数为上述比较次数之和，即

$$f(n)=(n-1)+(n-2)+\cdots+3+2+1=n(n-1)/2$$

比较次数依赖于 n^2。

对其它群体类，如顺序表和树，我们也用比较作为算法效率的衡量。

算法的计算性能也与其处理的数据是否有序有关。例如，若已知数组中的数据为有序的，则查找数组中最小值的算法可极大简化。对升序排列的数组，最小值为第一个元素。若降序排列，则是最后一个元素。此时，计算复杂度降低为访问一个唯一的元素，可立即完成。对排序来说，若表已经有序，则没有必要进行交换。这是最理想的情况，也是

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

算法效率最高的时候。然而，若表中元素完全是逆序排列，则每一次比较都将都导致一次元素交换，这是排序算法遇到的最糟糕的情况。一般情况下，交换次数应处于这两种情况之间，和表中数据的顺序有关。对群体类的查找和排序算法，我们可将比较次数作为算法的主要动作来衡量算法的计算性能。这些分析应区分算法的最理想情况，最糟糕情况和一般情况。我们可以认为，若算法对随机数据执行多次，一般情况下的性能应是预期的性能。

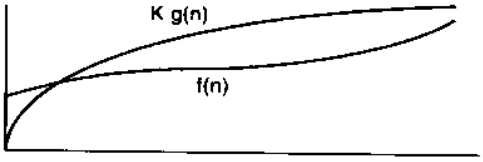
在计算算法的计算性能时，我们常给出函数 $f(n)$ 来表示比较的次数。事实上，很难给出函数 $f(n)$ 的精确形式，因此，我们用近似方法来给出该函数的上限。

我们定义一个简单函数 $g(n)$ 和一个常量 K，使 $K*g(n)$ 在 n 相当大时近似于 $f(h)$，也就是说，对于一个大 n，$f(n)$ 的性质与一个常量 K 乘以 $g(n)$ 类似。这样，我们用数学上的"O 方法"给出算法计算性能的衡量。

定义：若存在一常量 K 和起点 n_0，使 $n \geqslant n_0$ 时，

$$f(n) \leqslant K*g(n)，则 f(n)=O(g(n))。$$

这意味着函数 g 最终可以控制函数 f 的值。此时，算法的**计算复杂度**为 O(g(n))。



一般来说，数据结构算法的 O 值都可由多项式函数、对数函数和指数函数给出。对传统的数据结构，这些函数给出了算法复杂度的"最好"上限。

在交换排序的例子中，我们寻找定界函数 $f(n)$ 的 g 函数，表 4.1 给出了 $g(n)=\frac{1}{2}n^2$ 和 $f(n)$ 的值。从表中可得到，当 $g(n)=n^2$ 时，函数 $f(n)$ 的值最终在 $\frac{1}{2}g(n)$ 的范围之内。此时，我们马上可以得到条件 $n_0=1$ 和 $K=\frac{1}{2}$，即：

$$f(n) \leqslant \frac{1}{2}n^2, \quad n \geqslant 1$$

因此，$f(n)$ 即是 $O(g(n))=O(n^2)$，即交换排序的计算复杂度为 $O(n^2)$。最好、最差情况分析也可得出同样结果，因为交换排序总是需要 $\frac{1}{2}n(n-1)$ 次比较，不管数据的原始顺序如何，算法均需计算 $O(n^2)$ 次。

在以后我们将发现，某些排序算法的计算复杂度为 $O(n\log_2 n)$，即：

比较次数 $\leqslant K n\log_2 n$，当 $n \geqslant n_0$ 时

表 4.1 同时列出了 n^2 和 $n\log_2 n$ 的值，可以看到 $O(n\log_2 n)$ 算法比交换排序算法要有效的多。例如，对一有 10 000 元素的表，交换排序比较次数为 100 000 000 次，而更有效算法的比较次数不超过 132 000 次，提高效率将近 750 倍。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

表 4.1

n	$(1/2)n^2$	$S(n)=n^2/2-n/2$
10	50	45
100	5 000	4 950
1 000	500 000	499 500
5 000	12 500 000	12 497 500
10 000	50 000 000	49 995 000
n	n^2	$n\log_2 n$
5	25	11.6
10	100	33.2
100	10 000	664.3
1 000	1 000 000	9 965.7
10 000	100 000 000	132 877.1

为计算函数 $f(n)$ 的近似 O 函数，我们可用主要因子决定算法计算复杂度。这可在数学上用不等式简单证明。例如，对于函数

$f(n)=n+2$

其主要因子为 n，可用函数 $g(n)=n$ 通过下列不等式来证明 $f(n)$ 的 O 函数为 $O(n)$。

$f(n)=n+2\leqslant n+n=2*n$，　当 $n\geqslant 2$ 时

同理，f 也可以是 $O(n^2)$ 或 $O(n^3)$，因为 $g(n)=n^2$ 和 $g(n)=n^3$ 都大于 $f(n)$。由于 $O(n)$ 是 $f(n)$ 的最好的近似，所以我们选择它。

例 4.1

1. $f(n)=n^2+n+16$ 主要因子为 n^2，所以 f 是 $O(n^2)$，因为

   $f(n)=n^2+n+1\leqslant n^2+n^2+n^2=3n^2$，当 $n\geqslant 1$ 时

2. $f(n)=\sqrt{(n+3)}$　主要因子为 $\sqrt{n}$，所以 f 是 $O(\sqrt{n})$，因为

   $f(n)=\sqrt{n+3}\leqslant\sqrt{n+n}=\sqrt{2n}=\sqrt{2}\sqrt{n}$，当 $n\geqslant 3$ 时

3. $f(n)=2^n+n+2$　主要因子为 2^n，所以 f 是 $O(2^n)$，因为

   $f(n)=2^n+n+2\leqslant 2^n+2^n+2^n=3\cdot 2^n$，当 $n\geqslant 1$ 时

算法复杂度　上述方法给出了对算法运行时间的衡量。一般来说，算法对最好、最差情况的计算性能不同。因此我们对每种情况都给出一个值。4.4 节给出了顺序查找和折半查找的运行时间，两种算法的最好情况及最差情况不同，复杂度也不同。算法最好情况通常并不重要，因为它对于决定是否选择该算法没有用处。但最差情况是有用的，因为这将严重限制应用程序的性能，用户无法忍受这种最差情况，希望选择性能波动范围较窄的算法。一般情况下，要判断算法的平均性能更困难一些。我们将用一种简单的衡量办法，而将其数学细节留给算法的复杂性理论课程。

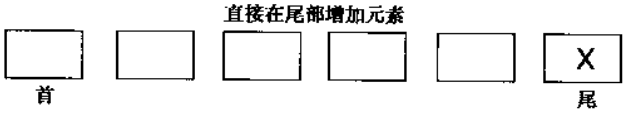
算法复杂度的数量级

大多数数据结构算法的复杂度属于很少几个数量级。下面我们给出这些数量级及相应的算法。若算法为 $O(1)$，其复杂度与群体中的数据个数无关。这种算法的运行时间为

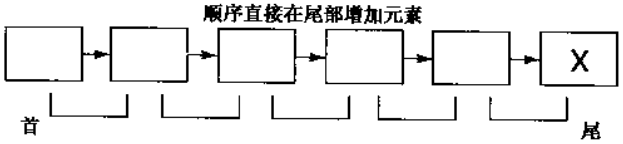
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

常量。例如，给数组的最后一个元素赋值的算法复杂度为 $O(1)$，它只需你给出数组最后一个元素的下标，用一个简单的赋值语句就可实现。



$O(n)$算法被称为线性算法，算法的复杂度与表的大小成正比。例如，如果我们不能直接访问链表尾，在一有 n 个元素的链表尾插入一个元素就是线性的，此时，我们必须一个一个扫描元素，经过 n 次以后才能确定链表尾。在一 n 元数组中查找最大值也是 $O(n)$，因为必须检查每个元素。



许多算法的复杂度数量级为对数函数 $\log_2 n$，这主要在于这些算法循环一次则其处理的数据减半。对数级经常和二叉树有关。4.4 节讨论的折半查找的平均复杂度及最差复杂度都是 $O(\log_2 n)$，第 13 章和第 14 章讨论的树排序算法和快速排序算法数量级为 $O(n \log_2 n)$。

表 4.2　算法复杂度的不同数量级

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4 096	65 536
32	5	160	1 024	32 768	4 294 957 296
128	7	896	16 384	2 097 152	3.4×10^{38}
1 024	10	10 240	1 048 576	10 737 411 824	1.8×10^{308}
65 536	16	1 048 576	4 294 967 296	2.8×10^{14}	忘了它吧！

大多数简单排序算法如交换排序的复杂度数量级为 $O(n^2)$。这种算法只在 n 相对来说比较小时才有实际价值。当 n 增加一倍，算法的运行时间将乘以 4。如果复杂度数量级为 $O(n^3)$，算法就很慢了，每当 n 增加一倍时，算法运行时间要乘 8。处理图的 Warshall 算法是一 $O(n^3)$算法。

复杂度为 $O(2^n)$的算法具有指数复杂度。这些算法只有在 n 很小时才有用，常常是那些和决策树有关的算法。

表 4.2 给出了线性、平方、立方、指数和对数级算法对指定值的复杂度。显然，除非 n 非常小，我们应避免立方和指数算法。

4.4　顺序查找与折半查找

本节介绍顺序查找，它可确定指定元素值在表中的位置。事实上，该算法可适用于所

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

有定义了“==”运算符的数组类型，而不仅仅是整型，所做的改动只是定义一个通用的数据类型 DataType，以它作为实际类型的别名。我们可以用关键字 typedef 创建这一别名，如下所示：

```
typedef int DataType;          // DataType 为 int
```

或

```
typedef double DataType;       // DataType 为 double
```

若程序员已定义 DataType，则通用的顺序查找算法为：

```
// 用顺序查找在 n 元数组 list 中查找与 key 等值的元素，返回该数组元素的下标；
// 若未找到，则返回 -1。
int SeqSearch(DataType List[ ],int n, DataType key)
{
    for (int i=0; i<n; i++)
      if(List[i]==key)
        return  i;          // 返回等值数组元素的下标
    return  -1;             // 未找到，则返回 -1
}
```

顺序查找的复杂度在最好情况和最差情况下差别很大。最好情况是表中第一个元素即是要找的元素，其运算时间为 O(1)。最差情况是直到找到表中最后一个元素或要找的元素根本不存在，这时需查找所有 n 个元素，其复杂度为 O(n)。一般情况下查找的次数要少一些。对于随机表，等值元素可能在表的任何位置发现。在执行次数足够多时，找到等值元素的平均查找长度为 n/2，即 n/2 是预期的查找次数。因此，我们说顺序查找算法的平均性能为 O(n)。

折半查找

顺序查找可适用于任意表。若被查找的表有序，则可用折半查找算法来查找，它是一种改进的查找算法。人们在电话号码簿中查找电话号码的方法就用的是折半查找算法，他们一般根据给定姓名的首字符来查找号码中的前、中、后部分，也许幸运地就找到了相应的页码。若没找到，也可根据姓名的相对位置往前或往后查找号码簿。例如，若要查找的人的姓名以“R”开头，而此时你翻开的页为“T”，则应向前翻。重复这个操作直到找到相应的电话号码或者确定号码簿中没有这个姓名。这也是我们用来查找有序表的办法。首先找到表中的中间元素，将其值与我们要找的值进行比较，若不等，则根据该值与要找值的大小来决定查表的前半部分或后半部分。通常情况下，如果表有序的话，可用这个算法来缩短查找时间。

假定表以数组形式存放。数组中有 n 个元素，其下标范围为 low = 0 和 high = n - 1，则折半查找算法可描述如下：

1. 计算数组中间点的下标：

```
mid = (low + high)/2
```

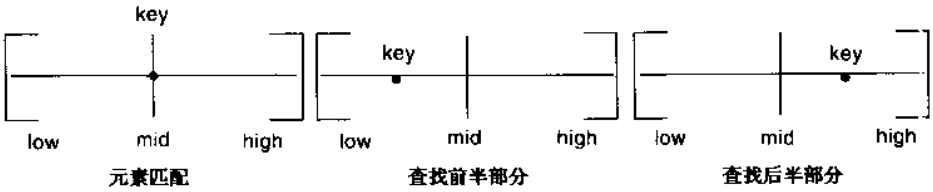
2. 将中间点元素的值与要查找元素的值进行比较：

若相等，则返回存放要查找元素位置的下标 mid；

```
if (A[mid] == key)
```

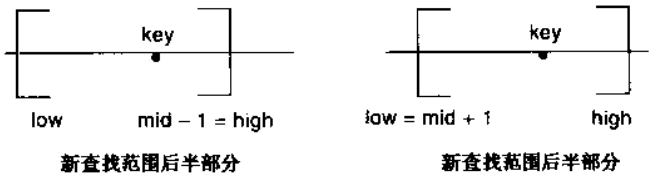

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
return (mid);
```

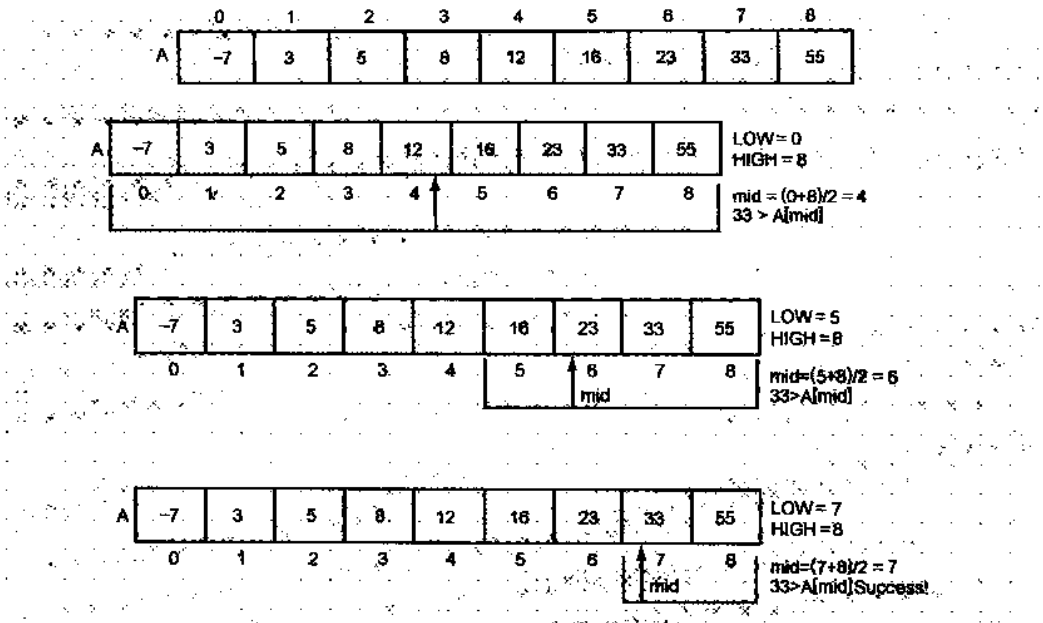
若 A[mid] < key，由于表有序，则所找元素必在下标 mid + 1 与 high 之间，即原始表的右半部分，此时新的查找范围为 low = mid + 1 和 high；若 key < A[mid]，则所找元素必在下标 low 到 mid − 1 之间，即原始表的左半部分，此时新的查找范围为 low 和 high = mid − 1；



该算法通过确定要查找元素存在的半个表，并在这个规模更小的子表上执行同样查找算法，使算法得到了优化。若最终表中不存在要查找的元素，则 low 值会超过 high 值，此时算法返回查找失败标志 −1(即没有找到)。

例 4.2

考虑整数数组 A，本例给出了查找元素 33 的算法的“快照”。



注意：算法只要求做 3 次比较，若该表无序而用顺序查找算法，则需 8 次比较。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

折半查找算法的实现

函数用支持等于('==')和小于('<')运算的通用数据类型实现。最初 low 为 0，high 为 $n-1$(n 是数组中元素的个数)，函数返回找到元素的下标值。若元素不存在，则返回 -1(此时 low > high)。

```
// 用折半查找算法在一有序数组中查找 key 值。若找到,返回其下标值,否则返回 -
int BinSearch(DataType list[], int low, int high, DataType key)
{
    int mid;
    DataType midvalue;

    while(low <= high)
    {
        mid = (low + high)/2;          // 子表中中间元素的下标
        midvalue = list[mid];          // 中间元素的值
        if (key == midvalue)
           return mid;                 // 找到 key 值,返回其下标值
        else if (key < midvalue)
            high = mid - 1;            // 继续查找子表的左半部分
        else
            low = mid + 1;             // 继续查找子表的右半部分
    }
    return -1;                         // 元素没有找到
}
```

顺序查找与折半查找算法实现的原型存放于文件"dsearch.h"中。因为函数运行时与 DataType 有关，在引用这些文件之前必须首先定义 DataType。

程序 4.1　顺序查找与折半查找比较

下述程序比较顺序查找和折半查找的计算时间。数组 A 中存放有 1 000 个已排序的随机整数，范围为 0—1 999。另有一数组 B 存放有大小也为 0—1 999 的 500 个随机整数，用来做被查找的元素。计时函数 Tickcount 在文件"ticks.h"中定义，它返回自程序启动开始过了多少个 1/60 秒。我们以此来测算两个算法各自完成 500 个查找的时间。程序的输出包括程序运行的秒数和找到元素的次数。

```
#include <iostream.h>
typedef int DataType;          // 整型数据
#include "dsearch.h"
#include "random.h"
#include "ticks.h"
// 将 n 元整数数组按升序排序
void ExchangeSort(int a[], int n)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    int i, j, temp;

    // 循环 n-1 次,排好 a[0], a[1], …, a[n-2]
    for (i=0; i<n-1; i++)
        // 将 a[i]…a[n-1]中的最小值置于 a[i]中
        for (j=i+1; j<n; j++)
            // 若 a[j]<a[i],则互换两元素
            if (a[j]<a[i])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
}
void main(void)
{
    // A 为被查数组,B 为 key 值数组
    int A[1000], B[500];
    int i, matchCount;
    // 用于计时
    long tcount;
    RandomNumber rnd;
    // 用随机数生成法生成 1000 个 0...1999 之间的整数放于 A 中
    for (i=0; i<1000; i++)
      A[i]=rnd.Random(2000);
    ExchangeSort(A,1000);
    // 在同样范围内生成 500 个 key 值置于 B 中
    for (i=0; i<500; i++)
      B[i]=rnd.Random(2000);
    cout << "Timing the Sequential Search" << endl;
    tcount = TickCount();        // 开始计时
    matchCount = 0;
    for (i=0; i<500; i++)
        if (SeqSearch(A,1000,B[i]) != -1)
          matchCount++;
    tcount = TickCount()-tcount;     // 时间单位为 1/60 秒
    cout << "Sequential Search takes " << tcount/60.0
         << " seconds for " << matchCount << " matches."<< endl;
    cout << "Timing the Binary Search" << endl;
    tcount = TickCount();        // 开始计时
    matchCount = 0;
    for (i=1;i<500;i++)
        if (BinSearch(A,0,999,B[i]) != -1)
          matchCount++;
    tcount = TickCount() - tcount;     // 时间单位为 1/60 秒
    cout << "Binary Search takes " << tcount/60.0
         << " seconds for " << matchCount << " matches."<< endl;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
/*
<程序 4.1 运行结果>
Timing the Sequential Search
Sequential Search takes 0.816667 seconds for 181 matches.
Timing the Binary Search
Binary Search takes 0.016667 seconds for 181 matches.
*/
```

折半查找算法复杂度的一般分析

算法的最好情况是第一次比较就找到元素，此时算法复杂度为 $O(1)$，因为只进行了一次比较。最差情况的复杂度为 $O(\log_2 n)$，这种情况为表中根本不存在该元素或该元素在最后一次比较时才找到。我们大致分析一下，最差情况时我们必须一直找到子串长度为 1 的状态，每匹配一次可将子串的长度减半，则子串的大小为如下序列：

$n, n/2, n/4, n/8, \ldots, 1$

即应将子串分解 m 次（$m \approx \log_2 n$，详见下面的分析）。最差情况下，我们的比较次数为最初的一次和每次分解子串进行的一次，即：

总比较次数 $\approx 1 + \log_2 n$

结果，折半查找在最差情况下复杂度为 $O(\log_2 n)$，该结果也可通过程序 4.1 的结果得到验证。顺序查找和折半查找算法所用时间比率为 49.0，而其预期的理论比率近似为 $500/(\log_2(1000)) = 50.2$。

折半查找算法复杂度的详细分析

算法循环第一次分解处理整个表。其后的每次循环使子表的大小减半。因此，算法处理的表的大小为：

$n, n/2, n/2^2, n/2^3, n/2^4, \ldots, n/2^m$

最终，存在一个整数 m 使

$n/2^m < 2$ 即 $n < 2^{(m+1)}$

由于 m 是使 $n/2^m < 2$ 的第一个整数，因此，它也必须满足

$n/2^{(m-1)} \geqslant 2$，　即 $2^m \leqslant n$

由此，可得：

$2^m \leqslant n < 2^{(m+1)}$

对上述不等式的每项取对数，并令 $\log_2 n = x$，则

$m \leqslant log_2 n = x < m + 1$

即 m 为小于或等于 x 的最大整数值，例如，若 $n = 50$，$\log_2 50 = 5.644$，则

$m = \mathrm{int}(5.644) = 5$

故算法的平均复杂度为 $O(\log_2 n)$。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

4.5 基本的顺序表类

购物清单、公共汽车调度计划、电话号码录、纳税表还有进货记录等都是表的实例。每种情况下，这些对象都包括一个元素序列。日常生活中的许多事情都可看作是对表的维护。例如，商业企业进货时要修改库存和订货单；公司财务部门产生所有雇员的工资信息；计算机编译器要用到的关键字表等等。

第 1 章介绍过基本的顺序表的 ADT，最基本的表操作包括在表尾插入一个新元素，删除元素，按位置访问表中的元素及将表清空。我们还介绍了检测表是否为空及表中是否存在某元素这两个操作。可以日常生活中的购物单作为这种表的实例(如图 4.6)。当在商店中穿行时，也许会在脑海中增加一两件欲购商品，一般放在购物单的最后。当找到一种商品后，则将它从购物单中去掉。只具有这些简单操作的表可以用来解决大问题。例如，可用于音像出租店中维护可租影片表和顾客表。影片出租后，将其从可租影片队列中删除并将其加入到顾客表中。归还时则进行相反的操作。

图 4.6 采购清单

ADT List 描述了每个元素类型均为 DataType 的类型同类型表。在 ADT 的定义中，并没有提到元素的存储形式。实际应用中，可用数组或动态申请内存的线性表来实现，而其 Insert，Delete 和 Find 操作则须根据元素的存储形式来具体实现。

第 1 章只提供了类实现的设想。本节我们给出元素以数组形式存放的类 SeqList 的具体实现。在第 9 章中，给出其用线性表的实现，第 12 章给出其从抽象基类 List 派生出的 SeqList，并分别于第 11，13 和 14 章给出结构相似的类二叉查找树，哈希表，字典。

类 SeqList 定义

声明

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
const int MaxListSize = 50;
class SeqList
{
    private:
        // 存放表的数组及当前表中元素个数
        DataType listitem[MaxListSize];
        int size;
    public:
        // 构造函数
        SeqList(void);
        // 访问表元素的方法
        int ListSize(void) const;
        int ListEmpty(void) const;
        int Find (DataType& item) const;
        DataType GetData(int pos) const;
        // 改变表元素的方法
        void Insert(const DataType& item);
        void Delete(const DataType& item);
        DataType DeleteFront(void);
        void ClearList(void);
};
```

说明

该类的定义和实现存放于文件"aseqlist.h"中。DataType 用来代表某种数据类型。在引用这个类之前，必须用 typedef 命令将 DataType 和指定类型联系起来。变量 size 存放表的当前长度，其初值为 0。因为表是由静态数组实现，常量 MaxListSize 是表大小的上限。试图向表中插入超过 MaxListSize 个元素将引发错误信息并导致程序终止。

例

```
typedef int DataType;     // 类 SeqList 中存放整型数据
#include "aseqlist.h"
void PrintList(const SeqList & L)       // 输出当前表中各元素
{
    int i, length = L.ListSize();
    for (i=0; i<length; i++)
        cout << L.GetData(i) << " "; // 输出第 i 个元素
    cout << endl;
}
SeqList L;                              // 表 L
L.Insert(10);                           // 在表尾放入元素 10
L.Delete(25);                           // 从表中删除值为 25 的元素
PrintList(L);                           // 输出修改后的表
if (L.Find(5))                          // 判定 5 是否在表中
    cout << "Have 5 in list" << endl;
while (! L.ListEmpty())
    cout << L.DeleteFront() << endl;// 输出并清空表
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类 SeqList 的实现

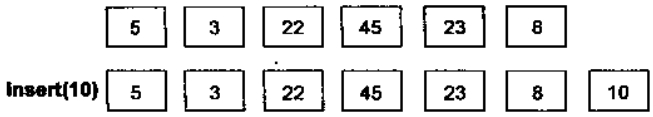
在下述类 SeqList 的实现中，我们用数组 listitem 来存放数据，准备了可存放 MaxListSize 个类型为 DataType 的空间。表中元素个数存放在变量 size 中，文件"iostream.h"和"stdlib.h"用来在插入操作 Insert 导致 size 超过 MaxListSize 时给出出错信息并终止程序运行。

私有数据成员 size 为 Insert 和 Delete 操作保存表的长度。构造函数和 ListSize，ListEmpty及 ClearList 都与 size 有关。本书只给出了将 size 赋值为 0 的构造函数。

```
// 构造函数将 size 赋值为 0
SeqList::SeqList(void):size(0)
{}
```

改变表的方法

Insert 操作在表尾部增加一个新元素并使其大小加 1。若该元素的插入使数组 listitem 越界，则打印出错信息并终止程序运行。对表大小的限制在第 9 章用线性表实现时可以去除。



参数 item 以形参常量传递。若 DataType 类型占用空间较大，这种参数传递法可避免以值参传递时需进行的数据拷贝工作。关键字 Const 保证了实际传递的参数不被改变。Delete 操作也使用同样的参数传递方法。

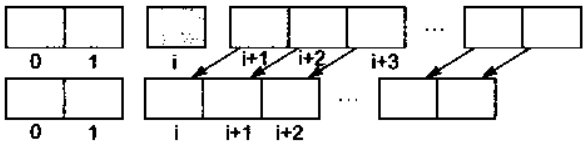
```
Insert
    // 在表尾插入元素。若表的大小超出 MaxListSize,则终止程序
    void SeqList::Insert(const DataType& item)
    {
        // 插入本元素会使表越界吗?
        if (size + 1 > MaxListSize)
        {
            cerr << "Maximum list size exceeded" << endl;
            exit(1);
        }

        // 表尾的下标为当前表的元素个数 size
        listitem[size] = item;
        size++;             // 元素个数 size 加 1
}
```

Delete 操作删除表中第一个与指定值相等的元素。这个函数要求 DataType 支持相等运算符("==")。在某些情况下，这也许需用户对 DataType 重定义操作符"=="。这将在第 6 章中详细介绍。若表中不存在指定元素，则该操作不对表做任何改动；若第 i 个元素与指定值相等，则将表中从下标 i+1 的元素开始到表尾所有元素左移一个位置。

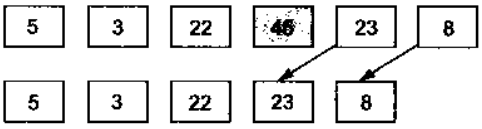
例如，从下表中删除元素 45 将使其后的元素 23 和 8 左移，表的大小从 6 变为 5，删除

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



元素 30 则该表不做任何改变。




```
Delete
// 在表中查找欲删元素,找到后将其删除
void SeqList::Delete(const DataType& item)
{
    int i = 0;

    // 查找欲删元素
    while (i < size && ! (item = = listitem[i]))
        i + + ;

    if (i < size)          // 若 i < size,则找到该元素
    {
      // 将表的后面部分左标一个位置
      while (i < size - 1)
      {
          listitem[i] = listitem[i + 1];
          i + + ;
      }
        size - - ;          // 元素个数 size 减 1
    }
}
```

访问表的方法 GetData 操作返回位于表中 pos 位置的数据值。若 pos 不属于范围 0 至 size - 1,则程序打印出错信息并终止运行。

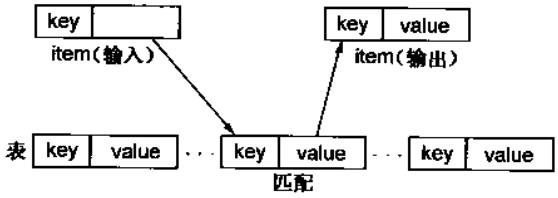
```
// 返回表中位于 pos 位置的数据值。若 pos 非法,则终止程序并给出出错信息
DataType SeqList::GetData(int pos) const
{
  // 若 pos 越界,则退出程序
  if (pos < 0 || pos ≥ size)
  {
      cerr < < "pos is out of range!" < < endl;
      exit(1);
  }
  return listitem[pos];
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Find 操作通过参数传入要查找的数值并顺序扫描整个表。若表为空或没找到元素，Find 返回 0(出错)。若在下标 i 找到要找的元素，则将记录 listitem[i]赋值给参数 item 并返回 1(正确返回)。

在找到相应元素时，将表元素的数据值通过参数传出。例如，假定 DataType 是一有 key 域和 value 域的结构，查找只在 key 域上判相等。输入时，参数 item 可只有 key 域；输出时，item 将包括 key 域和 value 域。



Find

```
// 用 item 作为 key 值查找表。若 item 在表中则返回 True,否则返回 False,如果找到,还需将
// 表中找到的元素由形参 item 返回
int SeqList::Find(DataType& item) const
{
    int i = 0;
    if (ListEmpty())
        return 0;               // 若表为空,则返回 False
    while (i < size && ! (item = = listitem[i]))
        i + + ;
    if (i < size)
    {
        item = listitem[i];     // 将表中元素赋给 item
        return 1;               // 返回 True
    }
    else
        return 0;               // 返回 False
}
```

类 SeqList 并未提供直接改变表中元素的方法。为达到这一目的，我们必须先找到该元素，检索到数据记录，然后删除该元素，修改记录值，并将新数据重新插入表中。当然，由于新元素存放于表尾，这种方法改变了元素在表中的位置。

例 4.3

记录 InventoryItem 中存放了零件号及零件在库中的个数。可用相等运算符来比较两个 InventoryItem 的零件号域。在 SeqList 对象 L 中寻找零件号为 5 的记录。若找到，则将其个数加 1。

```
struct InventoryItem
{
    int partNumber;
    int count;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
int operator = = (InventoryItem x, InventoryItem y)
{
    return x.partNumber = = y.partNumber;
}
typedef InventoryItem DataType;
# include "aseqlist.h" \        ...
SeqList L;
InventoryItem item;
    ...
item.partNumber = 5;
if (L.Find(item))
{
    L.Delete(item);
    item.count + + ;
    L.Insert(item);
}
```

由于元素总是在表尾插入，Insert 的复杂度(运行时间)与 n 无关，为 $O(1)$。Find 由顺序查找实现，所以其平均运行时间为 $O(n)$。忽略一些细微差别后，Delete 平均查找 $n/2$ 个表元素并移动 $n/2$ 个表元素。也就是说，Delete 的平均运行时间为 $O(n)$。Find 和 Delete 最差情况下的复杂度均为 $O(n)$。

应用举例

我们用对象 SeqList 来维护音像出租店中的可租影片表和已租影片表。表中每个元素都是包括影片名和在已租影片表中用到的顾客名的记录。

```
// 存放影片和顾客数据的记录结构
Struct FilmData
{
    char filmName[32];
    char customerName[32];
}
```

由于类 SeqList 中的 Find 操作要求决定于相等操作符“= =”，我们为 FilmData 结构重载“= =”运算符。该操作符用 C + + 的串函数 strcmp 来比较文件名。

```
// 用比较电影名来重载“= =”
int operator = = (const FilmData &A, const FilmData &B)
{
    return strcmp(A.filmName, B.filmName);
}
```

为在类中 SeqList 使用 FilmData，引用下列定义：

```
typedef FilmData DataType;
```

FilmData 的定义及“= =”操作的重载和 DataType 的定义放在文件“video.h”中。

音像店保存着电影库存表。为简单起见，假定每部电影在店中只有一个拷贝，新来的电影用 Insert 加入到库存表中。为出租某部影片，可用 Find 操作来查询库存表。若该片

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

存在，则将其从库存表中删除，并将其插入到已租表中。

程序 4.2　音像店

主程序模拟音像店的业务，最初库存表的电影从文件"films"中读入并存入 inventoryList 表中。程序给出了四次出租业务，每次均输入顾客名及其要租的片名，并查找该片是否在库存表中。若是，则从库存表中将其删去并插入到已租表中。若没有，则给顾客提示信息。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include "video.h"          // 有关数据说明
#include "aseqlist.h"       // 引入类 SeqList
// 从磁盘中读入可供影片
void SetupInventoryList(SeqList &inventoryList)
{
    ifstream filmFile;
    FilmData fd;
    // 打开文件，若出错，则终止程序
    filmFile.open("Films",ios::in|ios::nocreate);
    if (! filmfile)
    {
        cerr << "File 'films' not found!" << endl;
        exit(1);

    // 逐行读入文件内容，直到文件结束，将片名插入表 inventoryList 中
    while (filmFile.getline(fd.filmName,32,'\n'))
        inventoryList.Insert(fd);
}
// 遍历 inventoryList 表并输出所有片名
viod PrintInventoryList(const SeqList &inventoryList)
{
    int i;
    FilmData fd;
    for (i=0;i<inventoryList.ListSize();i++)
    {
        fd=inventoryList.GetData(i);        // 取到电影的记录
        cout << fd.filmName << endl;        // 输出片名
    }
}
// 遍历 customerlist 表，并输出顾客名及片名
void PrintCustomerList(const SeqList &customerList)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    int i;
    FilmData fd;
    for (i = 0;i < customerList.ListSize();i + + )
    {
        fd = customerList.GetData(i);        取到顾客记录
        cout < < fd.customerName < < "(" < < fd.filmName
        < < ")" < < endl;| = }
}
void main(void)
{
    // 可供影片表和顾客表
    SeqList inventoryList, customerList
    int i;
    // 可供影片的文件
    FilmData fdata;
    char customer[20]
    SetupInventoryList(inventoryList);        // 读入可供影片文件

    // 对 4 个顾客,询问他们的名字及欲租片名,若该片还有,则将该片从可供影片表中删除,
    // 并将片名和顾客名插入顾客表中;否则,告诉顾客该片已租出。
    for (i = 0;i < 4;i + + )
    {
        // 读入顾客名及欲租影片
        cout < < "Customer Name:";
        cin.getline(customer,32,'\n');
        cout < < "Film Request: ";
        cin.getline(fdata.filmName,32,'\n');
        // 检查该片是否还有。若有,则产生一个顾客记录
        if (inventoryList.Find(fdata))
        {
            strcpy(fdata.customerName,customer);
            // 插入顾客表中
            customerList.Insert(fdata)
            // 从可供影片表中删除该片
            inventoryList.Delete(fdata);
        }
        else
            cout < < "Sorry!" < < fdata.filmName
                < < " is not available." < < endl;
    }
    cout < < endl;

    // 输出最后的顾客表和可供影片表
    cout < < "Customers Renting Films " < < endl;
    PrintCustomerList(customerList);
    cout < < endl;
    cout < < "Films Remaining in Inventory:" < < endl;
    PrintInventoryList(inventoryList);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
/*
<输入文件"Films">
War of the Worlds
Casablanca
Dirty harry
Animal House
The Ten Commandments
Beauty and the Beast
Schindler's List
Sound of Music
La Strata
Star Wars
<程序 4.2 运行结果>
Customer Name: Don Baker
Film Request: Animal House
Customer Name: Teri Molton
Film Request: Beauty and the Beast
Customer Name: Derrick Lopez
Film Request: La Strata
Customer Name: Hillary Dean
Film Request: Animal House
Sorry! Animal House is not available.
Customers Renting Films
Don Baker (Animal House)
Teri Molton (Beauty and the Beast)
Derrick Lopez (La Strata)
Films Remaining in Inventory:
War of the Worlds
Casablanca
Dirty Harry
The Ten commandments
Schindler's List
Sound of Music
Star Wars
*/
```

书面作业

4.1 说明线性和非线性结构的区别，并各举一例。

4.2 说明下列情况下应使用哪种数据结构：

(a) 在一整数表中第 5 个元素存放数的绝对值；

(b) 按学生姓名的字母顺序逐个输出其成绩；

(c) 遇到一算术运算符时，从群体中删除此前的两个数；

(d) 模拟队列时，事件被逐个加入群体中并按其插入顺序删除；

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(e) 国际象棋中“皇后”移到某位置时，就将该位置加入到群体中；

(f) 数据结构中一个域为整型，另外一个域是实型，最后一个域是串型；

(g) 为以后使用将程序永久保留在外部设备上；

(h) 若要找的元素比表中当前元素小，则检查当前值的后继；

(i) 最小元素总是处于表的顶部；

(j) 用串作为关键值来定位散布在群体中的数据记录；

(k) 用词作索引定义其在群体中的定义；

(l) 假期，电话网负载过重，从可选路径中选择到达被叫方的最好路径。

4.3 下面是解决同一问题的三种算法在最差情况下的复杂度：

算法 1　$O(n^2)$

算法 2　$O(n\log_2 n)$

算法 3　$O(2^n)$

哪种算法最可取？为什么？

4.4 对下述函数进行 O 方法分析：

(a) $n+5$

(b) n^2+6n+7

(c) $\sqrt{(n+3)}$

(d) $\dfrac{n^3+n^2-1}{n+1}$

4.5 (a) $n>1$ 时，n 取何值时能使 $2^n>n^3$？

(b) 证明 2^n+n^3 是 $O(2^n)$

(c) 给出 $4(n^2+5)/(n+3)+6\log_2 n$ 的 O 值估计。

4.6 用数组存放某整数表。打印数组的第一个和最后一个元素的复杂度是多少？

4.7 说明为什么算法复杂度为 $O(\log_2 n)$ 时也可以说其复杂度为 $O(n)$。

4.8 对算法来说，每个循环都是其主要部分。用 n 的函数给出下述循环在最差情况下的计算时间。

(a)
```
for (dotprd = 0.0, i = 0; i < n; i++)
    dotprd += a[i] * b[i];
```

(b)
```
for (i = 0; i < n; i++)
  if (a[i] == k)
    return 1;
```

(c)
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    b[i][j] *= c;
```

(d)
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
  {
    entry = 0.0; \;    for (k = 0; k < n; k++)
      entry += a[i][k] * b[k][j];
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        c[i,j] = entry;
    }
```

4.9 用下述 n 元群体来存放数据。在下述结构中求最小值的算法复杂度是多少？

(a) 栈

(b) 有权队列

(c) 二叉查找树

(d) 升序排列的有序表

(e) 降序排列的可直接访问的表

4.10 Fibonacci 数列为：

1, 1,2,3,5,8,13,21,34,55,89,…

数列最前面两个元素为 1，后面的元素分别为前两个元素之和。可用表达式表示如下：

$$f_1 = 1, f_2 = 1, f_n = f_{n-2} + f_{n-1}, \qquad n \geq 3$$

下面函数用来计算第 n 个 Fibonacci 数，求其复杂度。

```
long Fibonacci(int n )
{
    long fnm2 = 1,fnm1 = 1,fn;
    int i;

    if (n< =2)
        return 1;
    for (i = 3;i< =n;i+ +)
    {
        fn = fnm2 + fnm1;
        fnm2 = fnm1;
        fnm1 = fn
    }
    return fn;
}
```

第 10 章介绍递归函数求第 n 个 Fibonacci 数，它的复杂度为指数级。显然，递归算法是无法接受的。

4.11 (a) 用顺序查找算法查找一个具有 50000 元素的表。

- 该算法可能的最少比较次数是多少？
- 该算法可能的最大比较次数是多少？
- 期望比较次数是多少？

(b) 用折半查找算法查找一个具有 50000 元素的表。

- 该算法可能的最少比较次数是多少？
- 该算法可能的最大比较次数是多少？

4.12 假设一个 SeqList 的对象 L 由下述元素组成：

34 11 22 16 40

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(a) 给出经过下面每条指令后表中的元素：

```
n = L.DeleteFront();
L.Insert(n);
if (L.Find(L.GetData(0) * 2)
    L.Delete(16);
```

(b) 对于对象，给出下面程序段的输出：

```
for (int i = 0;i < 5.i + + )
{
    L.Insert(L.DeleteFront());
    cout < < L.GetData(i) < < "  "
}
```

4.13 编写函数实现下述任务：

(a) 将 SeqList 的对象 L 连到对象 K 的尾部：

```
void Concatenate (SeqList& K, SeqList& L);
```

(b) 将 SeqList 的对象 L 的元素顺序反转：

```
void Reverse(SeqList& L);
```

4.14 函数 Ques 的输入参数为所有元素都是正整数的 SeqList 的对象 L。若 L 为：

{1,3,7,2,15,0,12}

其输出是什么？为什么 L 必须以形参传入？

```
typedef int Datatype;
# include "aseqlist.h"
int M(const SeqList &L)
{
    int i, mval, length = L.ListSize();
    if (length = = 0)
    {
        cerr < < "The list is empty" < < endl;
    }
mval = L.GetData(0);
    for (i = 1;i < length;i + + )
        if (L.GetData(i) > mval)
            mval = L.GetData(i);
    return mval;
}
void Ques(SeqList &L)
{
    int mval = M(L)

    L.Delete(mval);
}
```

4.15 说明在以数组实现的类 SeqList 中实现 Delete 操作时，为什么必须移动数据？

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

上机题

4.1 用 DataType 为 int 的类 SeqList 实现下述函数 InsertMax:

```
void  InsertMax(SeqList &L,  int  elt);
```

InsertMax 只在 elt 大于表中所有元素的情况下才将 elt 插入表 L 中。主程序读入 10 个整数,对每个整数调用 InsertMax。最后打印出表 L。

4.2 有一记录定义如下:

```
struct Person
{
    char name[20];
    int age;
    char gender;
};
```

它构成的类 SeqList 如下:

```
#include <string.h>
// 类 SeqList 的 Find 操作需下述定义:
int operator == (Person X, Person Y)
{
    return strcmp(x.name, y.name) == 0;
}
typedef Person DataType;
#include "aseqlist.h"
```

(a) 编写函数

```
void PrintByGender( const SeqList &L, char sex);
```

功能为遍历表 L 并打印出指定性别的所有记录。

(b) 编写函数

```
int Inlist(const SeqList &L, char *nm, Person &p);
```

来判定表 L 中是否含有 name 为 nm 的记录。其算法为创建一个 name 域的值为 nm 的对象 Person,然后用 Find 操作来实现。创立 Person 时不必初始化记录的 age 和 gender 域。因为对象间的比较仅仅针对 name 域。若比较相等,则将记录值赋给参数 p 并返回 1;否则返回 0。参数 p 值只在找到记录时才改变。

(c) 编写一主程序来测试这个函数。

4.3 编写程序提示用户输入整数 n,然后创建一个取值范围为 0~999 的随机整数组成的 n 元整数数组,并用交换排序算法对其排序。用"ticks.h"中定义的 TickCount 函数对上述算法进行计时。分别在 n=50,500,1000 和 10000 时运行该程序,并验证交换排序的复杂度为 $O(n^2)$。

注意: 由于系统可能无法分配 1000 或 10000 个元素的局部数组,可用下述方法申请

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

动态数组来存放元素。用户同样可用标准的访问数组方法“a[i]”来访问动态数组。

```
int   *a;          // 定义指针
...
a = new int[n];    // n 为 50,500,1000 或 10000
```

4.4 扩展程序 4.2 的功能，使其能处理归还影片的情况。首先问顾客是来租片还是还片。如果是还片，将其从已租片表中删除并将该片插入到可租片中。

4.5 考试也可看作是 SeqList 结构的例子。学生们将试卷顺序交给老师(在表尾增加记录)，总有性急的学生想早点知道成绩，并检查一下试卷。老师必须从头一张起查找试卷直到找到该学生的卷子，借给学生查看(将试卷从表中删除)。学生看完后，老师再将试卷放到表尾。

用类 SeqList 写一程序来实现上述模型。用下述记录将学生与其试卷联系起来：

```
Struct Test
{
    char name[30];
    int testNumber;
};
```

主程序根据下述菜单循环：

```
1:交卷                    2:学生查卷
3:交回借出的试卷          4:退出
```

每次选择分别完成下列功能：

1：提示输入姓名和试卷号；将其插入到 Submitted Tests 表中；

2：提示输入姓名，从 SubmittedTests 表中删除该生的记录，并将其插入到 borrowedTests 表中；

3：提示输入姓名，从 borrowedTests 表中删除记录，并将其插入到 SubmittedTests 表中；

4：收回所有借出的试卷，从 borrowedTests 表中删除所有记录，并将它们插入 SubmittedTests 表中，打印最后的 SubmittedTests 表。为判断两个 Test 记录相等，可用下面的函数重载操作符“==”：

```
#include <string.h>
int operator==(const Test& t1,Test& t2)
{
    return strcmp(t1.name, t2.name)==0;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第5章 栈和队列

5.1 栈
5.2 类 Stack
5.3 表达式求值
5.4 队列
5.5 类 Queue
5.6 优先级队列
5.7 实例研究:事件驱动模拟
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

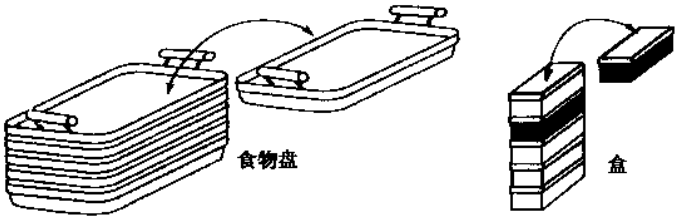
www.Linuxidc.com

本章我们详细讨论前面介绍的两种从表中限定存取元素的数据结构——栈和队列。同时，我们还讨论优先级队列，它是队列的一个特例，它每次删除表中优先级最高的元素。栈、队列和优先级队列都用 C++ 类来实现。我们举两个程序实例来说明这些基本概念，即 RPN 计算器的操作符栈和模拟银行等待队列中的顾客和职员行为的事件驱动处理。后面的应用也用到了优先级队列，并介绍了一个重要的商业管理工具。

5.1 栈

栈是常用的重要数据结构，其应用也十分广泛。例如，编译器对表达式的语法分析就用到了栈。另外，函数的参数传递、函数的调用和返回也是用栈来实现的。

栈是一种只在表的一端访问元素的表，其元素只能从栈顶端增加或删除。餐厅中的食物盘或堆在一起的盒子都是栈的很好的例子。



栈被设计用来存放那些本身就只能从一端访问的元素。比如烤肉用的烤肉叉。图 5.1 所示烤肉叉上串了一些准备用来烧烤的蔬菜，在第 1 根叉上的蔬菜顺序为洋葱、蘑菇、青椒、洋葱。在将其放去烧烤之前，客人声称他不吃蘑菇，希望将其拿掉。这就需先拿下洋葱(第 2 根叉)，拿下蘑菇(第 3 根叉)，再将拿下的洋葱叉上(第 4 根叉)。不喜欢青椒会给厨师带来更大的麻烦。

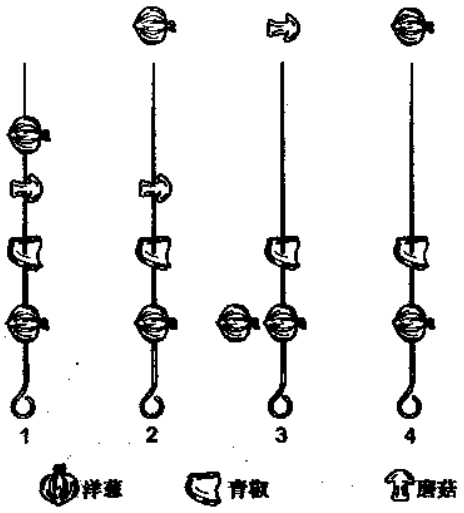


图 5.1 烤肉叉

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

栈结构就给出了其增加和删除元素的操作。**压入**(push)操作往栈顶增加一个元素。从栈顶删除一个元素的操作称为**弹出**(Pop)。图 5.2 给出了一系列的压入和弹出操作。最后压入栈中的元素是第一个被弹出的。因此,出入栈的顺序为**后进先出**(LIFO,last-in/first-out)。

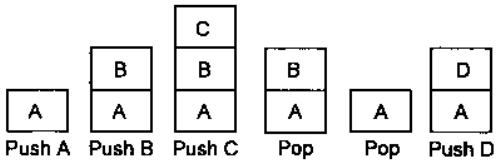


图 5.2 栈的压入和弹出

理论上讲,栈可以是一个无穷大的表,就如从逻辑上说,食物盘可以堆得和天一样高。实际上,食物盘堆在架子上,烤肉叉上也叉不上多少蔬菜。当架子或肉叉上满了时,你就无法再往栈中增加(压入)元素了。此时栈已达到它所能处理的元素个数的最大值。这种情况称为"栈满"。另一种极端情况是不可能从空架上取出食物盘来。这种情况称为"栈空",意味着无法从栈中删除(弹出)元素。ADT Stack 的描述只处理了栈空的情况。栈满的情况与应用程序设置的表的上限有关。

ADT Stack is

Data

含栈顶位置信息的数据项列表

Operations

Constructor

```
        Initial values:        无
        Process:               初始化栈顶
```

StackEmpty

```
    Input:                     无
    Preconditions:             无
    Process:                   检查堆栈为空
    Output:                    若堆栈为空,则返回 True,否则返回 False
    Psotconditions:            无
```

Pop

```
    Input:                     无
    Preconditions:             堆栈非空
    Process:                   从栈顶删除数据项
    Output:                    返回栈顶元素
    Postconditions:            栈顶元素被删除
```

Push

```
    Input:                     准备压入堆栈的一个数据项
    Preconditions:             无
    Process:                   将数据项压入栈顶
    Output:                    无
    Postconditions:            栈顶元素为新元素
```

Peek

```
    Input:                     无
    Preconditions:             堆栈非空
    Process:                   检索栈顶数据项的值
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

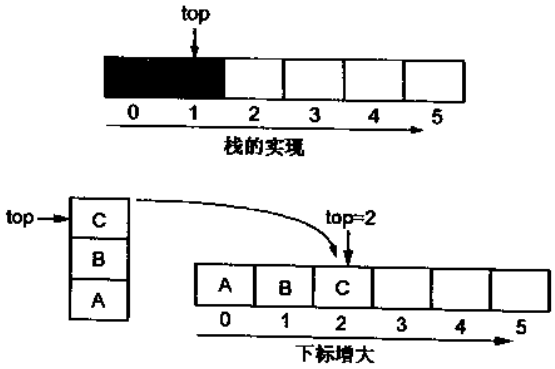
```
        Output:                 返回栈顶数据项的值
        Postconditions:         堆栈不变
    ClearStack
        Input:                  无
        Preconditions:          无
        Process:                删除堆栈中的所有数据项并重新置栈顶
        Output:                 无
        Postconditions:         堆栈被重置为初始状态
end ADT Stack
```

5.2 类 Stack

栈由表、指向栈顶的下标或指针和栈的操作集组成。我们用数组来存放栈元素。因此，栈的大小不能超过数组的元素个数，"栈满"状态也与此有关。我们将在第 9 章用链表实现类 Stack，这就可以取消这个限制了。

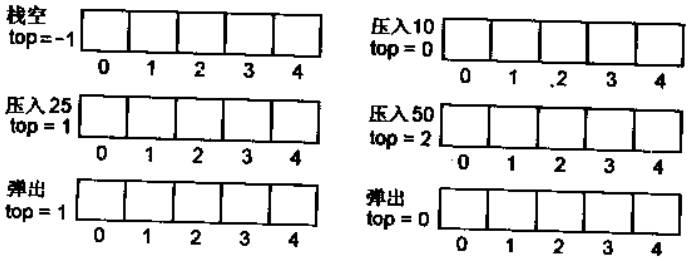
栈的对象的说明包括栈大小，它定义了表中最多能存放的元素个数。其缺省值为 MaxStackSize = 50。表 Stacklist，栈的大小 Size 和下标(top)都是类 Stack 的私有成员，其所有操作都是公共的。

初始时栈为空，此时 top = －1。元素以下标递增的顺序(top = 0,1,2)进入(压入)数组，以下标递减的顺序(top = 2,1,0)离开(弹出)。例如，下面是一字符栈(DataType = char)。经过 n 次压入/弹出操作后，下标 top = 2，位于栈顶的元素是 stacklist[top] = 'C'。



例 5.1

下图是可含 5 个元素的整数栈，我们给出了下列操作的示意图。压入 10；压入 25；压入 50；弹出；弹出。下标 top 在每压入一个元素时加 1，每弹出一个元素时减 1。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类 Stack 定义

声明

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

const int MaxStackSize = 50;

class Stack
{
    private:
        // 私有数据成员。栈数组及顶指针 top
        DataType stacklist[MaxStackSize];
        int top;
    public:
        // 构造函数,初始化 top
        Stack(void);

        // 改变栈的操作
        void Push(const DataType& item);
        DataType Pop(void);
        void ClearStack(void);

        // 访问栈顶元素
        DataType Peek(void) const;

        // 检测栈状态的方法
        int StackEmpty(void) const;
        int StackFull(void) const;        // 用数组实现
}
```

说明

栈中的数据类型为 DataType,使用前由 typedef 定义。

用户负责在压入元素时检查栈是否满和弹出元素时栈是否空。若压入或弹出的前提有一个不满足,则打印出错信息且程序终止执行。

若栈为空,StackEmpty 返回 1(True),否则返回 0(False)。可用 StackEmpty 来判定是否可进行 Pop 操作。

若栈为满,StackFull 返回 1(True),否则返回 0(False)。可用 StackFull 来判定是否可进行 Push 操作。

ClearStack 通过将 top 赋值为 -1 来清空栈。这样,栈可移作它用。

例

栈的说明和实现均放在文件"astack.h"中。

```
typedef int DataType;
# include "astack.h"          // 引入类 Stack
Stack S;                      // 定义 S 为 Stack 的对象
S.Push(10);                   // 在栈 S 中插入 10
cout << S.Peek() << endl;     // 输出 10
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 从栈中弹出 10 并让其为空
if (! S.StackEmpty())
    temp = S.Pop();
cout < < temp < < endl;
S.ClearStack();                    // 清空栈
```

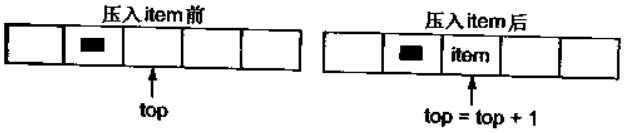
类 Stack 的实现

Stack 的构造函数将 top 赋初值为 -1,即栈为空时的值。

```
// 初始化栈下标 top
Stack::Stack(void):top(-1)
{}
```

栈操作 栈的两个最基本的操作为将元素插入(Push)栈中和从栈中删除(Pop)元素。类还提供了 Peek 操作,它允许用户检索栈顶的元素而不将其从栈中删除。

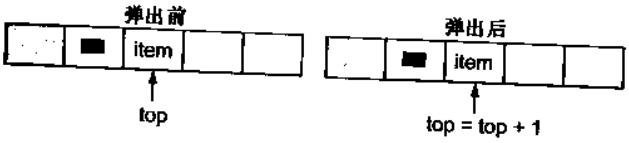
将元素压入(Push)栈时,可将下标 top 加 1 并将新值赋给数组 stacklist。若此时栈已满,则打印出错信息并终止程序运行。



Push

```
// 往栈中压入 item
void Stack::Push(const Datatype& item)
{
    // 若栈已满,则终止程序
    if (top = = MaxStackSize - 1)
    {
        cerr < < "Stack overflow!" < < endl;
        exit(1);
    // 将 top 加 1 并将 item 拷贝到 Stacklist 中
    top + + ;
    stacklist[top] = item;
}
```

Pop 操作先将栈顶元素的值拷贝到一局部变量 temp,然后将 top 减 1,这就将栈顶元素从栈中删除了。temp 即为该操作的返回值。若此时栈已空,则打印出错信息并终止程序运行。



Pop

```
// 将顶元素弹出堆栈并返回其值
DataType Stack::Pop(void)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    DataType temp;

    // 若栈为空,程序退出
    if (top = = -1)
    {
        cerr < < "Attempt to pop an empty stack!" < < endl;
        exit(1)
    // 将顶元素赋值给 temp
    temp = stacklist[top];
// top 减 1 并返回 temp 值
    top- -
    return temp;
}
```

Peek 操作除一重要不同点外,本质上是 Pop 的复制。这不同点就是不将 top 减 1,保持栈的状态不变。

```
Peek
// 返回栈顶元素的值
Datatype Stack::Peek(void) const
{
    // 若栈为空,则程序终止
    if (top = = -1)
    {
        cerr < < "Attempt to peek t an empty stack!" < < endl;
        exit(1);
    }
    return stacklist[top];
}
```

检测栈状态 执行过程中,当以不正确方式访问栈时,程序将非正常退出。如在栈为空时执行 Peek 操作,将导致这种情况。为保持栈的完整性,类 Stack 还提供了检测栈状态的操作。

函数 StackEmpty 检查 top 是否为 -1。若是,则栈为空,返回 1(True);否则,返回 0(False)。

```
// 检查栈是否为空
int Stack::StackEmpty(void) const
{
    // 返回 top = = -1 的逻辑值
      return top = = -1;
}
```

函数 StackFull 检查 top 是否为 MaxStackSize - 1。若是,则栈已满,返回 1(True);否则,返回 0(False)。

```
// 检查栈是否为满
int Stack::StackFull(void) const
{
    // 检测 top 的位置
    return top = = MaxStackSize - 1;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
```

函数 ClearStack 将 top 重置为 -1，这使栈回到了构造函数定义的初始状态。

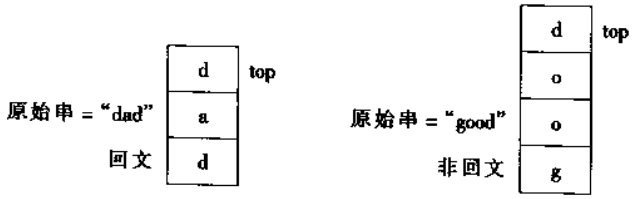
```
// 从栈中清除所有元素
void Stack::ClearStack(void)
{
    top = -1;
}
```

栈的 Push 和 Pop 操作直接访问栈顶元素，与表中元素个数无关。因此，它们的计算时间均为 O(1)。

应用：回文 当 DataType 为 char 时，栈为字符栈，它可用于回文游戏中。所谓回文，指的是顺读与逆读字符串一样（不包括空格）。例如，"dad"，"sees"和"madam im adam"为回文，"good"则不是。程序 5.1 用类来判定回文。

程序 5.1 回文

本程序调用 cin.getline( )读入一行文本并用函数 Deblank 将文本中的空格滤掉。Deblank 还将非空格字符拷贝到第二个串中。程序通过扫描两次去掉了空格的串来检查其是否为回文。第一次扫描，将每个字符压入栈，产生一个存放串的逆序的表。第二遍扫描，将原串中的每个字符与从栈中弹出的元素比较。若字符不等，则程序终止，该串不是回文。若直到栈空时还在比较，则该文本是回文。




```
#include <iostream.h>
typedef char DataType;        // 栈元素为字符类型
#include "astack.h"
// 产生去掉所有空格后的新串
void Deblank(char *s, char *t)
{
    // 扫描整个串，直到串结束符 NULL
    while(*s != NULL)
    {
      // 若字符非空格，则拷贝至新串中
      if (*s != '')
          *t++ = *s;
      s++;                    // 取下一字符
    }
    *t = NULL;                // 在新串后拼接 NULL
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
void main()
{
    const int True = 1, False = 0;
    // 栈 S 中存放串的逆序
    Stack S;
    char palstring[80],deblankstring[80],c;
    int i = 0;
    int ispalindrome = True;    // 首先假定串为回文
    // 读入一行
    cin.getline(palstring,80,'n');
    // 去掉空格并将结果置于 deblankstring 串中
    Deblank(palstring, deblankstring);
    // 将无空格的串压入栈中
    i = 0;
    while(deblankstring[i] ! = 0)
    {
        S.Push(deblankstring[i]);
        i + +;
    }
    // 将字符出栈并与原始串进行比较
    i = 0;
    while(! S.StackEmpty())
    {
    c = S.Pop();    // 从栈中取下一字符
    // 若字符不相等,则退出循环
    if (c ! = deblankstring[i])
    {
        ispalindrome = False;    // 不是回文
        break;
    }
    i + +;
}
if (ispalindrome)
    cout < < '\"' < < palstring < < '\"'
         < < " is a palindrome < < endl;
else
    cout < < '\"' < < palstring < < '\"'
         < < " is not a palindrome < < endl;
}
/*
<程序 5.1 运行结果之一>
madam im adam
"madam im adam" is a palindrome
<程序 5.1 运行结果之二>
a man a plan a canal panama
"a man a plan a canal panama" is a palindrome
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
<程序 5.1 运行结果之三>
palindrome
"palindrome" is not a palindrome
*/
```

应用：多进制输出 大多数程序语言的输出语句以十进制形式作为数据的缺省输出形式。栈可用于以其它进制输出数据。我们在第 2 章中讨论了二进制数，希望读者能够将其原理扩展到其它进制。

例 5.2

下述例子将十进制数转换为其它进制：

1. (八进制)$28_{10} = 3 \cdot 8 + 4 = 34_8$
2. (四进制)$72_{10} = 1 \cdot 64 + 0 \cdot 16 + 2 \cdot 4 + 0 = 1020_4$
3. (二进制)$53_{10} = 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 = 110101_2$

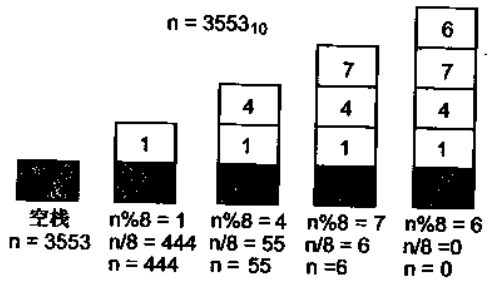


图 5.3 用栈以八进制输出数据

我们用堆栈解决以非十进制形式输出数据的问题。将十进制整数 n 以 B 进制形式输出的算法可描述如下：

1. n 的最右边位上的数字为 $n\%B$，将它压入栈 S 中；
2. n 的剩余部分为 n/B，用 n/B 代替 n；
3. 重复上述 1 和 2 直到 $n=0$，此时 n 的所有位数都出现过了；
4. 从栈中可得到 B 进制表示的数 N。从 S 中弹出并打印所有字符直到栈为空。

图 5.3 给出了 $n=3553(10)$ 转化为八进制数的过程，描述了产生四个八进制位的栈的增长过程。算法最后从栈中弹出每个字符并打印输出结果为 6741。

程序 5.2 多进制输出

下述程序给出了一个输出函数，它读入一个非负长整数 num 和一范围为 2～9 的基 B。在屏幕上输出 num 的 B 进制数。主程序提示用户输入三个非负整数及基，然后输出

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

结果。

```
#include <iostream.h>
typedef int DataType;
#include "astack.h"
// 按 B 进制输出整数 num
void MultibaseOutput(long num, int B)
{
    // 自左向右存放 B 进制的各位数
    Stack S;
// 自右向左取得 B 进制的各数并压入栈 S 中
    do
    {
        S.Push(num % B);        // 压入栈中
        num /= B;               // 去掉最高位
    } while(num != 0);          // 继续至所有数字计算完毕
    while (! S.StackEmpty())    // 倒空栈
        cout << S.Pop();
}
void main(void)
{
    long num;                   // 十进制数
    int B;                      // 基数
    // 分别读入 3 个正整数及要转换成的基数
    for (int i=0;i<3;i++)
    {
        cout << " Enter non-negative decimal number and base"
             << "(2 <= B <= 9): ";
        cin >> num >> B;
        cout << num << " base " << B << " is ";
        MultibaseOutput(num,B);
        cout << endl;
    }
}
/*
<程序 5.2 运行结果>
Enter non-negative decimal number and base (2 <= B <= 9): 72 4
72 base 4 is 1020
Enter non-negative decimal number and base (2 <= B <= 9): 53 2
53 base 2 is 110101
Enter non-negative decimal number and base (2 <= B <= 9): 3553 8
3553 base 8 is 6741
*/
```

5.3 表达式求值

电子计算器中给出了一个栈的重要用途。用户输入一个由数据(操作数)和运算符组

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

成的算术表达式，计算器使用一个堆栈来计算其运算结果。运算器规定了表达式的输入数据格式。如下述表达式：

-8 + (4 * 12 + 5^2)/3

中包含**二目运算符**(+,*,/,^)、**操作数**(8,4,12,5,2,3)和创建了一个表达式的括号。表达式的最前面是运算符负号，它是一**单目运算符**(如：-8)。其它的运算符都需两个操作数，故称为双目运算符。运算符^(乘方)代表表达式 5*5=25。

若表达式的每个二目运算符都放在两个操作数之间，且每个单目运算符都放在其操作数的前面，则表达式为**中缀**形式。例如：

-2+3*5

就是一个中缀表达式，中缀表达式是最常见的书写表达式的格式。大多数程序语言和计算器使用的也是中缀表达式。中缀表达式的求值算法用到两个数据类型不同的栈，一个存放操作数，另一个存放运算符。由于类 Stack 要求使用一个"DataType"的定义，我们无法在本节中实现中缀表达式的求值。在第七章我们引入模板类后，将继续讨论这个问题。那时，我们可以定义具有两种或更多种不同数据类型的栈对象。

除中缀表达式外，还有操作数在运算符前面的**后缀表达式**，也称之为 **RPN(逆波兰表示法**，Reverse Polish Notation)。例如，中缀表达式"a+b"的后缀形式为"ab+"。用后缀表达式，我们对变量和数字按其出现顺序输入，而在运算符的两个操作数都已输入后，再输入运算符。例如，在下述表达式中，在'*'的两个操作数 b 和 c 都输入后马上输入"*"。当操作数 a 和(b*c)都已输入完毕后输入"+"。在后缀形式下，运算符的优先级是显然的。运算符'*'在'+'的前面。

中缀：　a+b*c=a+(b*c)　　后缀：　abc*+

中缀表达式	后缀表达式
a*b+c	ab*c+
a*b*c*d*e*f	ab*c*d*e*f*
a+(b*c+d)/e	abc*d+e/+
(b*b-4*a*c)/(2*a)	bb*4a*c*-2a*/

后缀表达式求值

后缀表达式求值算法，用一个栈从左到右扫描表达式就可实现。为简单起见，假定所有运算符均为二目运算符。包括单目运算符的情况，我们以习题形式请读者自己完成。

后缀表达式中只有操作数和运算符。我们读入每个元素，根据其类型，进行下述动作。若元素是一操作数，将其压入栈中；若是操作符 <op>，用两个弹出动作取得操作数 X 和 Y。然后，计算表达式 X<op>Y 的值并将结果压回栈中。读完表达式的所有元素后，栈顶元素就是表达式的结果。

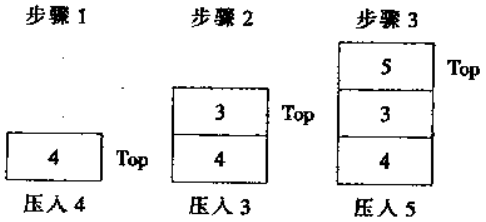
例　5.3

中缀表达式 4+3*5 的后缀表达式为 435*+。它的求值可分为下述五个步骤：

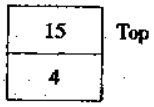
步骤 1～3：读入操作数 435 并将每个数压入栈中；

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



步骤 4：读入操作符 * 并通过弹出两个操作数 5 和 3 来计算 3 * 5，将结果 15 压回栈中。



步骤 4

步骤 5：读入操作符 + 并通过弹出两个操作数 15 和 4 来计算 4 + 15 = 19，将结果 19 压回栈中，既是表达式的最后结果。

步骤 5

应用：后缀计算器

我们通过模拟一个有 +，-，*，/和^(乘方)五种运算符的 RPN 计算器来说明后缀表达式的求值，其操作数为浮点型。运算器数据和运算符都在类 Calculator 中，用一个简单的主程序来进行调用。

类 Calculator 中有一些公共的成员函数来输入表达式和对计算器清零，用来对表达式求值的一系列函数定义为私有成员函数。

类 Calculator 的定义

声明

```
enum Boolean (False,True);
typedef double DataType;          // 计算器可接受实数
#include "astack.h"                 引入类 Stack

class Calculator
{
    private:
        // 私有成员:计算器栈及操作数
        Stack S;                    // 存放操作数

        void Enter(double num);
        Boolean GetTwoOperands(double& opnd1,double& opnd2);
        void Compute(char op);

    public:
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 构造函数
    Calculator(void);
    // 计算表达式值及清空计算器
    void Run(void);
    void Clear(void);
};
```

说明

缺省构造函数建立了一个空计算器栈。由于计算器一直在运行，用户必须调用 Clear 来清除计算器栈；然后才能继续运行，计算新的表达式的值。

函数 Run 输入一个 RPN 格式的表达式，输入“=”表明表达式结束，只显示表达式的最终结果。当某运算符的操作数不够两个时，打印出错信息“Missing operands”，除以 0 也将产生错误信息，两种情况下，计算器都清空栈并准备读入新的表达式。

例

```
Calculator CALC;     // 创建计算器 CALC
CALC.Run( );
<运行实例>
    4 3 * =
    12               //显示表达式 4 * 3 的结果
```

类 Calculator 的实现

计算器函数由下述函数实现：输入表达式，完成计算，在屏幕上打印结果。类定义放在文件“calc.h”中。

函数 Enter 读入一个浮点型参数并将它压入栈中。

```
// 往栈中存放数据值
void Calculator::Enter (double num)
{
    S. Push(num);
}
```

函数 GetTwoOperands 被 Compute 方法用来从计算器栈中取得操作数并将其赋给输出参数 operand1 和 operand2。该函数有错误检测并用返回值表示是否两个操作数都存在。

```
// 从栈中取得操作数并赋值给形参。若操作数不够，则打印出错误信息，并返回 False
Boolean Calculator::GetTwoOperands(double& opnd1,double& opnd2)
{
    if (S.StackEmpty())          // 检查操作数是否存在
    {
        cerr << "Missing operand!" << endl;
        return False;
    }
    opnd1 = S.Pop();             // 取右操作数
    if (S.StackEmpty())
    {
        cerr << "Missing operand!" << endl;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        return False;
    }
    opnd2 = S.Pop();                // 取左操作数
    return True;
}
```

所有的内部运算均在函数 Compute 的控制下进行。Compute 首先调用 GetTwoOperands 来得到栈顶的两个值。若 GetTwoOperands 返回 False，则操作数非法，Compute 清除计算器栈。否则，Compute 函数执行字符参数 op('+','-','*','/','^')指定的运算并将结果压入栈中。若试图除以 0，则打印出错信息，并清除计算器栈。对乘方运算，我们用函数

```
double pow(double x, double y);
```

来计算 x^y，它在 C++库 <math.h> 中定义。

```
// 运算求值
void Calculator::Compute(char op)
{
    Boolean result;
    double operand1,operand2;
    // 取两个操作数，并判断是否成功取到
    result = GetTwoOperands(operand1,operand2);
    // 若成功取得，则计算本次运算值，否则将栈清空。注意被 0 除
    if (result = = True)
        switch(op)
        {
            case '+':   S.Push(operand2 + operand1);
                        break;
            case '-':   S.Push(operand2 - operand1);
                        break;
            case '*':   S.Push(operand2 * operand1);
                        break;
            case '/':   if (operand1 = 0.0)
                        {
                            cerr < < "Divide by 0!" < < endl
                            S.ClearStack();
                        }
                        else
                            S.Push(operand2/operand1);
                        break;
            case '^':   S.Push(pow(operand2,operand1));
                        break;
        }
    else
      S.ClearStack();               // 出错!,清空计算器
}
```

计算器的首要动作由公共函数 Run 实现，它进行后缀表达式计算。Run 中主循环从 input 流中读入字符，直到读入"="时终止输入。空格符被滤掉。若字符是运算符('+','-','*','/','^')，调用函数 Compute 来进行计算。若字符不是运算符，由于流中只有运算

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

符和操作数，Run 认为这是操作数的第一个字符。Run 将其放回 input 流，因此它可被作为浮点操作数的一部分被顺序读入。

```
// 读入字符串，同时对后缀表达式求值，直到读入"="时停止
void Calculator::Run(void)
{
    char c;
    double newoperand;
    while (cin > > c,c! ='=')        // 读入字符，至'='时退出
    {
    switch(c)
    {
        case '+':                    // 检查是否为可能的运算符
        case '-':
        case '*':
        case '/':
        case '^':
            Compute(c);              // 读到运算符；求值
            break;
        default:
            // 非运算符；则必为操作数，将字符送回
            cin.putback(c);
            // 读入操作并将其存入栈中
            cin > > newoperand;
            Enter(newoperand);
            break;
    }
    }
    // 答案已在栈顶，用 Peek 输出之
    if (! S.StackEmpty())
        cout < < S.Peek() < < endl;
}
// 清空操作数栈
void Calculator::Clear(void)
{
    S.ClearStack()
}
```

程序 5.3　后缀计算器

对象 CALC 就是一个计算器，我们给出一个计算直角三角形斜边长度的实例。三边长度分别为 6,8,10。另外两种情况为出错情况。

```
    ( √(6² + 8²))    (RPN 格式：6 6 * 8 8 * + .5^ = )
# include "calc.h"
void main(void)
{
    Calculator CALC;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    CALC.Run();
}

/*
<程序 5.3 运行结果之一>
8 8*6 6* + .5^=
10
<程序 5.3 运行结果之二>
3 4+ *
Missing operand!
3 4+8* =
56
<程序 5.3 运行结果之三>
1 0 /=
divide by 0!
*/
```

5.4 队列

队列是以表形式存放元素，但只允许在表的两端访问元素的数据结构，如图 5.4 所示，队列中的元素在表尾插入，在表头删除，应用中使用队列按出现顺序存放元素。

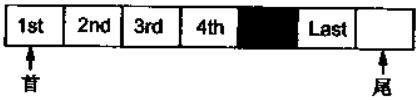


图 5.4 队列

队列删除元素的顺序和元素到达队列的顺序相同，可称为先进先出(FIFO, first-in/first-out)或先到先服务(FCFS, first-come/first-served)顺序。商店中顾客的服务顺序和打印池中打印作业缓冲区是队列的典型例子。

队列由表及指定的对表头和表尾位置元素的访问操作组成，如图 5.5 所示，这两个位置分别用来插入和删除队列中的元素。与栈一样，队列也以形式类型 DataType 来存放元素。理论上也不限制队列中元素个数，但是，如果用数组来实现表，有可能出现“队列满”的情况。

ADT Queue is

Data

```
    数据项列表
    front: 表示队列中第一个数据项的位置
    rear:  表示队列中最后一个数据项的位置
    count: 任一时刻队列中元素项的个数
```

Operations

Constructor

```
    Initial values:          无
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

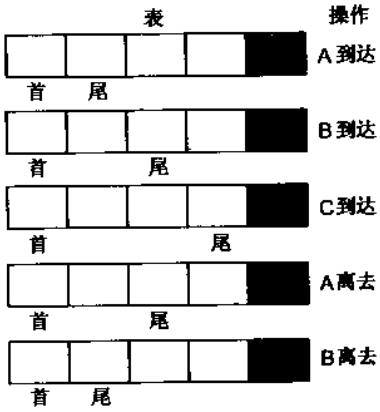


图 5.5 队列操作

Process: 初始化队头和队尾

Qlength

Input: 无

Preconditions: 无

Process: 确定队列中元素项的个数

Output: 返回队列中元素项的个数

Postconditions: 无

QEmpty

Input: 无

Preconditions: 无

Process: 检查队列是否为空

Output: 若队列为空则返回 1(True),否则返回 0(False)。注意此条件与检测 QLength 是否为 0 是等价的

Postconditions: 无

QDelete

Input: 无

Preconditions: 队列非空

Process: 从队列头部删除数据项

Output: 返回被删除数据项

Postconditions: 队列中删除一个数据项

QInsert

Input: 要存到队列中的一个数据项

Preconditions: 无

Process: 将数据项放入队尾

Output: 无

Postconditions: 一个新的数据项被加入到队列中

QFront

Input: 无

Preconditions: 队列非空

Process: 取出队列头部数据项的值

Output: 返回队列头部数据项的值

Postconditions: 无

ClearQueue

Input: 无

Preconditions: 无

Process: 删除队列中的所有数据项并重置初始状态

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    Output:                    无
    Postconditions:            队列为空
end ADT Queue
```

例 5.4

图 5.6 表示在一系列操作过程中一个四元素队列的变化情况。每种情况下，我们都给出一个队列是否为空的标志 QEmpty。

队列在计算机模型中有广泛应用。如银行中的职员流水线。多用户操作系统也有等待执行的程序队列和等待打印的作业队列。

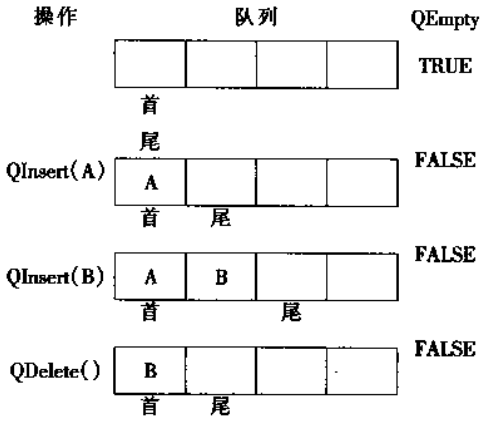


图 5.6 四元队列各操作对队列的改变

5.5 类 Queue

类 Queue 用数组存放表中元素，并定义变量来指定表头和表尾位置，以实现 ADT Queue。正是由于用数组实现的表，该类还定义了函数 QFull 来检测数组是否已满。第 9 章我们用链表实现队列时就不需要这个函数了。

类 Queue 的定义

声明

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

const int MaxQSzie = 50;
class Queue
{
   .private:
       // 队列数组及其参数
       int front, rear, count;
       DataType qlist[MaxQSize];
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    public:
        // 构造函数
        Queue(void);    // 初始化数据成员
        // 改变队列的操作
        void QInsert(const DataType& item);
        DataType QDelete(void);
        void ClearQueue(void);
        // 访问队列
        DataType QFront(void) const;
        // 检测队列状态
        int QLength(void) const;
        int QEmpty(void) const;
        int QFull(void) const;
};
```

说明

形式类型 DataType 使队列可处理不同的数据类型。类 Queue 中包括一个表(q ist),其最大大小由常量 MaxQSize 决定。

数组成员 Count 记录队列中元素的个数,该值也就给定了队列是满还是空。

QInsert 读入一个 DataType 类型的元素并将其插入队尾,而 QDelete 从队首中删除元素并返回其值。QFront 返回位于队首的元素值。这可使我们"预知"下个被删除的元素。

程序员删除元素之前应该用 QEmpty 检测队列是否为空,插入元素之前应该用 QFull 检测队列是否已满。如果 QInsert 或 QDelete 的前提不满足,程序打印出错信息后终止运行。

队列的说明与实现均存放在文件"aqueue.h"中。

例

```
typedef int DataType;

# include "aqueue.h"

Queue Q;                          // 定义队列

Q.QInsert(30);                    // 往队列中插入 30
Q.Qinsert(70);                    // 往队列中插入 70
cout < < Q.QLength() < < endl;    // 打印队列中元素个数 2
cout < < Q.QFront() < < endl;     // 打印首元素值 30
if (! Q.QEmpty())
    cout < < Q.QDelete();         // 输出值为 30
cout < < Q.QFront() < < endl;     // 输出值为 70
Q.ClearQueue();                   清空队列
```

类 QUEUE 的实现

我们以现实中的顾客排队来说明队列。队首定义为队列中的第一个顾客,队尾则定义在最后一个顾客的后一位置。当队排满时,顾客就得上另一个队列了。图 5.7 描述了

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

队列的变化，也指出影响我们实现的一个问题。我们假定队中只能排 4 个顾客。在情形 2 中，顾客 A 接待完后，顾客 B 和顾客 C 均向前移动。在情形 3 中，顾客 B 也接待完了，顾客 C 往前移。在情形 4 中，顾客 D,E,F 排入队中，将队排满，顾客 G 只能排在其它队中。

这些情形反映了顾客在队列中的行为。接待完某位顾客后，队列中的其它顾客前移。从队列的角度看，每当有元素离开队列，其它元素前移一个位置。这种模型不利于提高计算机实现的效率。如果队列中有 1000 个元素，当从队首中删除一个元素后，就得将 999 个元素左移一位。

我们引入环状模型来实现队列。与前述模型删除元素需将元素左移相比，这种模型将队列在逻辑上置于一圆环上，用变量 front 来指定队首位置，删除元素后将其顺时针移动一格；变量 rear 指向下一元素要插入的位置，插入完后将其顺时针移动一格；变量 count 存放队列中元素的个数。若 count 等于 MaxQSize，则队列已满。图 5.8 所示就是环状模型。

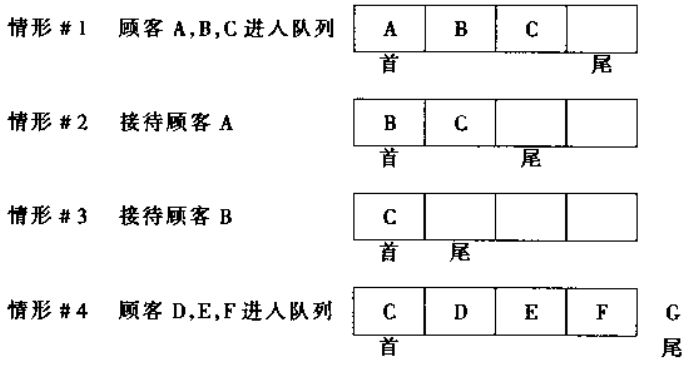


图 5.7 四位顾客的队列

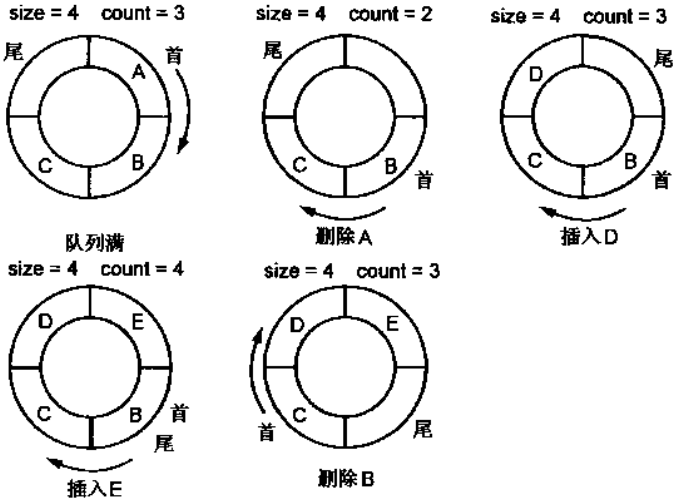


图 5.8 队列的环状模型

环状队列实现要用到求余运算：

rear 指针前移： rear = (rear + 1) % MaxQSize;

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

front 指针前移：　front = (front + 1) % MaxQSize;

例 5.5

用含 4 个元素的整数数组 qlist(size=4)来实现环状队列。初始时，count = 0 且下标 front 和 rear 均为 0。图 5.9 描述了元素插入和删除的情况。

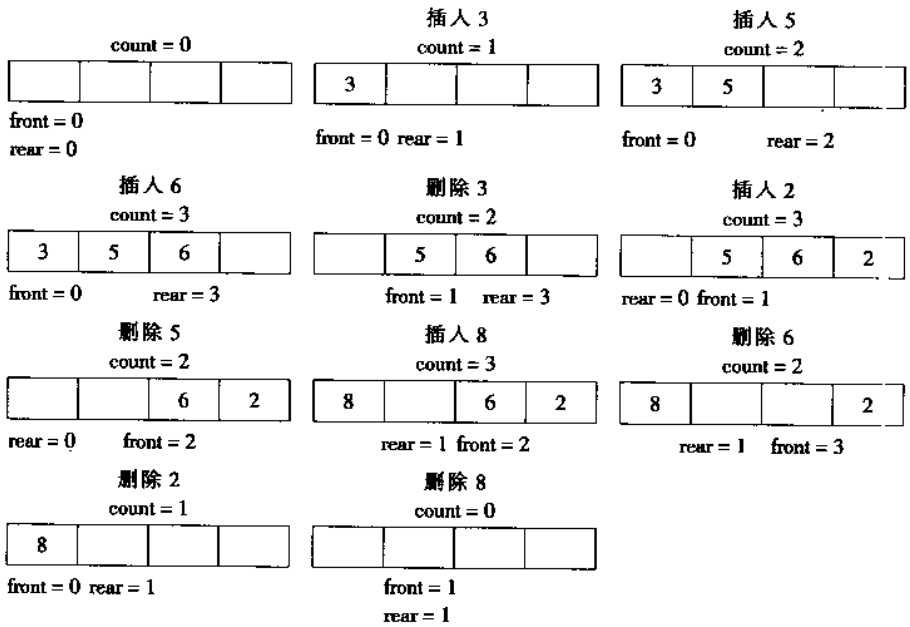


图 5.9　环状队列的插入和删除

Queue 的构造函数　Queue 构造函数初始化 front，rear 和 count 为 0，这就建立了一个空队列。

```
// 初始化队列的 front,rear 和 count
Queue::Queue(void):front(0),rear(0),count(0)
{}
```

队列操作　队列只允许有限的几个操作，如增加一个元素(QInsert)和删除元素(QDelete)等。该类还提供 QFront 来检索队列中的第一个元素。对一些应用来说，这种"预知"操作可使我们决定是否从表中删除这个元素。

本节中，我们只介绍改变队列的函数，即增加元素和删除元素。其它函数可从类 Stack 中找到模型，也可在文件"aqueue.h"中找到。

插入操作开始前，rear 指针指向表中的下一个可插入元素的位置，新元素将放在这个位置，队列的计数器 count 加 1。

```
qlist[rear] = item;
count + + ;
```

将元素在表中放置好以后，rear 指针必须改为指向下一位置(如图 5.10(A)所示)。由于我们用的是环状队列，插入位置可能是在数组的尾部(qlist[size - 1])。rear 可能会

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

重新指向队首(如图 5.10(B)所示)。

上述指针计算用求余运算符“%”实现。

```
rear = (rear + 1) % MaxQSize;
```

QInsert

```
// 往队列中插入元素
void Queue::QInsert(const DataType& item)
{
    // 若队列已满,则出错退出
    if (count = = MaxQSize)
    {
        cerr < < "Queue overflow!" < < endl;
        exit(1);
    }
    // Count 加 1,将元素加入 qlist 并修改 rear 值
    count + + ;
    qlist[rear] = item;
    rear = (rear + 1) % MaxQSize;
}
```

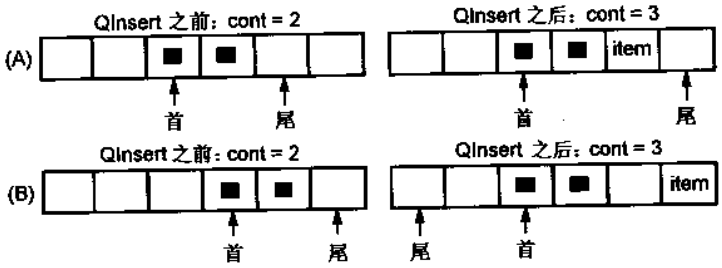


图 5.10 QInsert

QDelete 操作从队首位置删除元素。该位置由指针 front 给出。我们通过将元素值拷贝到一临时变量并将队列计数器 count 减 1 来实现这个操作。

```
item = qlist[front];
count - - ;
```

在环状模型中,front 必须重新指向表中的下一元素,这也用求余运算符“%”实现(如图 5.11 所示)。

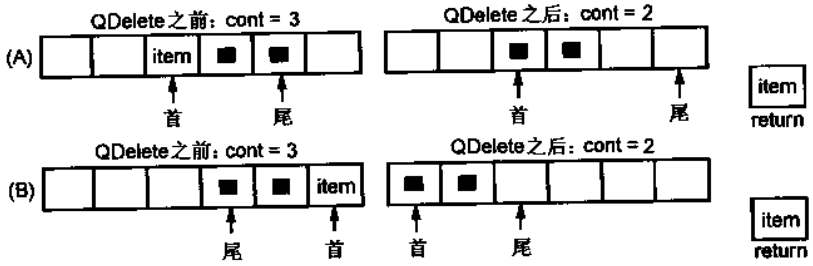


图 5.11 QDelete

```
front = (front + 1) % MaxQSize;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

临时变量的值可作为函数的返回值。

QDelete

```
// 删除队首元素并返回其值
DataType Queue::QDelete(void)
{
    DataType temp;

    // 若队列 qlist 为空,则退出程序
    if (count = = 0)
    {
        cerr < < "Deleting from an empty queue!" < < endl;
        exit(1);
    }

    // 记录队首元素的值
    temp = qlist[front];

    // 元素个数减 1,前移首指针,并返回原队首值
    count - - ;
    front = (front + 1) % MaxQsize;
    return temp;
}
```

函数 QInsert,QDelete 和 QFront 的复杂度均为 O(1)。因为它们都从表头或表尾直接访问元素。

程序 5.4 舞伴

舞会在星期五晚举行。参加舞会的男士和女士各自排队进入舞厅。舞会开始,从两队中按顺序组成舞伴开始跳舞。如果男士和女士人数不等,则多出的人只能等到下一舞曲才能开始。程序从文本文件"dance.dat"读入男士和女士的信息,文件格式如下:

性别　姓名

其中性别是一个字符"F"或"M"。当文件中记录读完后,两个队列都已形成。可通过同时删除两个队列中的元素来组成舞伴,直到某一队列为空。若有人在等待,程序给出有多少人在等待,并打印出等待队列中第一个人的名字。

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

// 跳舞者记录
struct Person
{
    char name[20];
    char sex;          // 'F'(女性)  'M'(男性)
};

// 存放 Person 对象的队列
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
typedef Person DataType;
# include "aqueue.h"
void main(void)
{
    // 分开男伴和女伴的两个队列
    Queue maleDancers(10), femaleDancers(10);
    Person p;
    char blankseparator;
    int i;
    // 存放舞伴情况的输入文件
    ifstream fin;
    // 打开文件并确认该文件存在
    tin.open("dance.dat");
    if (! fin)
    {
      cerr < < "Unable to open file" < < endl;
      exit (1);
    }
    // 读入包含性别,姓名的输入行
    while(fin.get(p.sex))              // 文件结束时终止
    {
        fin.get(blankseparator);       // 滤空格
        fin.getline(p.name, 20, '\n'); // 读入姓名
        // 插入到相应队列
        if (p.sex = = 'F')
            femaleDancers.QInsert(p);
        else
            maleDancers.QInsert(p);
    }
// 从两队列中取舞伴配对,直到其中一个队列为空
cout < < "The dancing partners are:" < < endl < < endl;
while (! femaleDancers.QEmpty() && ! maleDancers.QEmpty())
{
    p = femaleDancers.QDelete();
    cout < < p.name < < " ";       // 输出女伴姓名
    p = maleDancers.QDelete();
    cout < < p.name < < endl;      // 输出男伴姓名
}
cout < < endl;
// 若任一队列中还有没有选上的舞伴,输出剩下舞伴人数及队中第一个人的姓名
if (! femaleDancers.QEmpty())
{
    cout < < "There are " < < femaleDancers.QLength()
         < < " women waiting for the next round." < < endl;
    cout < < femaleDancers.QFront().name
         < < " will be the first to get a partner." < < endl;
}
    else if (! maleDancers.QEmpty())
    {
        cout < < "There are " < < maleDancers.QLength()
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

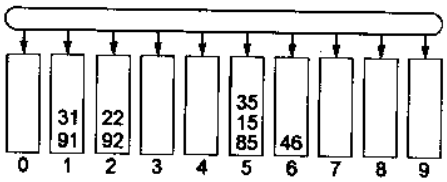
www.Linuxidc.com

```
            << "men waiting for the next round." << endl;
        cout << maleDancers.QFront().name
            << "will be the first to get a partner." << endl;
    }
}
/*
<文件"dance.dat">
M George Thompson
F Jane Andrews
F Sandra Williams
M Bill Brooks
M Bob Carlson
F Shirley Granley
F Louise Sanderson
M Dave Evans
M Harold Brown
F Roberta Edwards
M Dan Gromley
M John Gaston
<程序 5.4 运行结果>
The dancing   partners are:
Jane Andrews   George Thompson
Sandra Williams   Bill Brooks
Shirley Granley   Bob Carlson
Louise Sanderson   Dave Evans
Roberta Edwards   Harold Brown
There are 2 men waiting for the next round.
Dan Gromley will be the first to get a partner.
*/
```

应用:用队列将数据排序 在早期计算中,常用排序机来对一组穿孔卡片排序。下述程序模拟排序机的动作。为解释这个过程,我们假定卡片上的数为两位整数,范围为位00~99。排序机有十个箱子,编号为0~9。排序机处理卡片两遍,第1遍处理个位数,第2遍处理十位数。每个卡片通过排序机时掉入相应的箱子中。这种方法称为基数排序,可扩展到任何位数。

初始表:91 46 85 15 92 35 31 22

第1遍,将卡片按个位数分散到10个箱子。



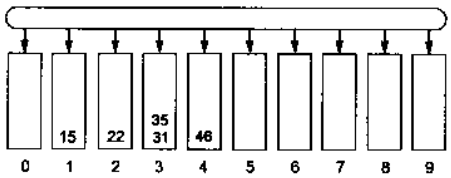
按从0~9的顺序从10个箱子中回收卡片

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 1 遍后的表为:91　31　92　22　85　15　35　46

第 2 遍,将卡片按十位数分散到 10 个箱子。



按从 0~9 的顺序从 10 个箱子中回收卡片

第 2 遍后的表为:15　22　31　35　46　85　91　92

两遍过后,得到排序后的表。实际上,第 1 遍保证所有个位数小的卡片在个位数大的卡片的前面。例如,所有以 1 结尾的数在以 2 结尾的数的前面。第 2 遍,如果两卡片的值分别为 3s 和 3t,且 s < t,它们进入箱子 3 的顺序为 3s 后跟 3t(注意:经过第一遍后,以 s 结尾的卡片在以 t 结尾的卡片的前面)。由于箱子有序,卡片从箱子 3 中以 FIFO 的顺序取出。经过两遍后,卡片排序完毕。

程序 5.5　基数排序

本程序完成两位数的基数排序。数组 L 中存放一个 50 个随机数的队列。

```
int L[50];              // 存放 50 个随机整数
```

一个队列数组模拟 10 个有序的箱子:

```
Queue digitQueue[10]; // 10 个整数队列
```

函数 Distribute 的参数为整数数组,digitQueue 队列数组和用户定义的描述符 Ones 或 Tens 来表示本次排序是对个位数(第 1 遍)还是十位数(第 2 遍)进行排序。

第 1 遍时,用 L[i]%10 来取得个位数并用该值来将此数放入正确的队列中。

```
digitQueue[L[i]%10].QInsert(L[i])
```

第 2 遍时,用 L[i]/10 来取得十位数并用该值来将此数放入正确的队列中。

```
digitQueue[L[i]/10].QInsert(L[i])
```

函数 Collect 按 digit = 0 到 9 的顺序扫描队列 digitQueue 数组并从每个队列中取出所有元素放入表中:

```
while(! digitQueue[digit].QEmpty( ))
    L[i++]=digitQueue[digit].QDelete( )
```

函数 Print 输出表中的数,以一行 10 个数,每个数占 5 个打印位置的格式打印。

```
#include <iostream.h>
#include "random.h"        // 随机数发生器
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
typedef int DataType;     // 数据类型为整型
#include "aqueue.h"
// 自定义类型区分一个数的个位数和十位数
enum DigitKind {ones, tens};
// 将数组中的数放入10个下标从0~9的队列之一。用户定义的类型DigitKind表明了这次分
发根据的是个位数还是十位数
void Distribute(int L[], Queue digitQueue[],int n, DigitKind kind)
{
    int i;
        // 循环处理n个元素的数组
        for (i = 0; i<n; i++)
            if (kind == ones)
            // 按个位数插入到相应队列中
                digitQueue[L[i] % 10].QInsert(L[i]);
            else
            // 按十位数插入到相应队列中
                digitQueue[L[i] / 10].QInsert(L[i]);
}
// 从队列中取到元素并送回数组中
void Collect(Queue digitQueue[], int L[])
{
    int i = 0, digit = 0;

    // 按下标从0到9扫描队列数组
    for (digit = 0; digit < 10; digit++)
         // 收集队列中元素并送回数组中
         while (! digitQueue[digit].QEmpty())
           L[i++] = digitQueue[digit].QDelete();
}
// 扫描n元素数组并输出,每行输出10个数
void PrintArray(int L[], int n)
{
    int i = 0;
while(i < n)
    {
        cout.width(5);          // 输出5个空格
        cout << L[i];           // 输出相应元素
        if (++i % 10 == 0) // 每10个数后换行
            cout << endl;
    }
    cout << endl;
}

void main(void)
{
    // 用来暂存数据的10个队列
    Queue digitQueue[10];
    // 50个整数的数组
    int L[50];
    int i = 0;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    int item;
    RandomNumber rnd;          // 提供随机数

    // 用 50 个在范围 0~99 中的随机数初始化数组
    for (i = 0; i < 50; i++)
        L[i] = rnd.Random(100);

    // 将它们按个位数分到 10 个队列中,收集回来后并输出
    Distribute(L, digitQueue, 50, ones);
    Collect(digitQueue,L);
    PrintArray(L,50);

    // 将它们按十位数分到 10 个队列中,收集回来后打印排好序的数组
    Distribute(L, digitQueue, 50, tens);
    Collect(digitQueue, L);
    PrintArray(L,50);
}

/*
<程序 5.5 运行结果>

 40  70  20  51  11  81  21  12  52  92
 62  72  82  82  62  72  52  83  63  23
  3  73  33  54  24  84  55  15  65  85
 25  16  46  86  36  67  17  27   7  97
 88  98  68  69  79  89  29  69  99  59

  3   7  11  12  15  16  17  20  21  23
 24  25  27  29  33  36  40  46  51  52
 52  54  55  59  62  62  63  65  67  68
 69  69  70  72  72  73  79  81  82  82
 83  84  85  86  88  89  92  97  98  99

*/
```

每遍的复杂度为 O(n),均由分解、插入队列、删除等操作组成,由于排序两遍,二位数基数排序的复杂度为 O(2n),因此是线性算法。算法也可扩展为对 m 位的 n 个数进行排序,此时,算法复杂度为 O(mn)。这是因为排序 m 遍,每遍的复杂度均为 O(n)。本算法显然优于我们将在第 13 和第 14 章讨论的复杂度为 $O(n\log_2 n)$ 的排序算法 heapsort 和 quicksort。然而,基数排序在空间性能上无法与这些原地排序算法相比。原地排序算法在原始数组中对数据进行排序,不需使用临时空间。而基数排序要用到 10 个队列,每个队列有自己的空间来存放 front,rear 指针,队列计数器 count 和数组。另外,若数据很多,基数排序的性能随 mn 增大而降低。可以说,使基数排序优于 $O(n\log_2 n)$ 排序算法的 n 值很大,但在实际可以遇到的情况下,复杂度为 $O(n\log_2 n)$ 的算法,如 quicksort 可能优于基数排序。

5.6 优先级队列

正如我们前面讨论的一样,队列是以 FIFO 顺序访问元素的数据结构,它从中删除"最老"的元素。实际应用中常需要另外一种队列,它删除优先级最高的元素。这种结构,我们称为优先级队列,具有操作 PQInsert 和 PQDelete。PQInsert 只是简单地将元素插入到队

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

列中，而 PQDelete 根据某些外部的评价因素来判断元素的重要程度(优先级)，并删除优先级最高的元素。例如，假设公司中设有一个公共的秘书处来处理职员的各种事务，公司规定董事长的事务优先级最高，其次为经理的工作事务，然后为总管等，公司员工的职务成为其事务重要程度的标准。替代平常的先来先服务的事务处理机制(队列)，秘书处根据事务的重要程度来处理它们(优先级队列)。

操作系统中，优先级队列可用来记录进程表，然后按其优先级调度执行。例如，大多数操作系统将打印进程的优先级放在其它进程之下。优先级 0 通常用来定义"最高优先级"，而一般优先级是一比较大的数，如 20 等。下面是一任务及其优先级的表：

任务 #1	任务 #2	任务 #3	任务 #4	任务 #5
20	0	40	30	10

存放顺序

任务按顺序 2,5,1,4,3 执行：

任务 #2	任务 #5	任务 #1	任务 #4	任务 #3

执行顺序

大多数应用中，优先级队列是一权——值对，其中权表示优先级，如操作系统中，每个任务都有任务描述符及充当权值的优先级。

优先级	任务描述符

从优先级队列中删除元素时，队列中也许会有多个相同优先级的元素，此时，我们可以将这些元素按队列处理，即将同一优先级的元素按其到达先后进行服务。在下述 ADT 中，我们假设没有优先级相同的元素。

优先级队列包含一系列增加或删除表中元素的操作，另外还有一些检查表长度及判断表是否为空的操作。

ADT Priority Queue is

Data

元素组成的表。

Operations

```
Constructor
    Initial Values:     无
    Process:            初始化表元素个数为 0

PQLength
    Input:              无
    Preconditions:      无
    Process:            确定表中元素的个数
    Output:             返回表中元素的个数
    Postconditions:     无
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

PQEmpty

Input:	无
Preconditions:	无
Process:	检查表中元素个数是否为 0
Output:	若表中无元素,则返回 1(True);否则,返回 0(False)
Postconditions:	无

PQInsert

Input:	将要插入表中的元素
Preconditions:	无
Process:	将元素插入表中,并将表的元素个数加 1
Output:	无
Postconditions:	表中增加了元素且长度增加

PQDelete

Input:	无
Preconditions:	优先级队列非空
Process:	从表中删除优先级最高的元素,并将表长度减 1
Output:	返回被删除的元素
Postconditions:	表中被删除一个元素且表长度减 1

ClearPQ

Input:	无
Preconditions:	无
Process:	从优先级队列中删除所有元素,并恢复到初始情况
Output:	无
Postconditions:	优先级队列为空

end ADT Priority Queue

优先级队列类

本书中,我们给出优先级队列的多种实现方法。每种情况下,均用表来存放元素。用 count 参数及访问表的操作来插入和删除元素。本章中,我们用元素类型为形式类型 DataType 的数组来存放元素。由于元素在数组中存放,这就要求类提供 PQFull 操作。在后续章节,我们用有序表和堆来存放优先级队列的元素。

PQueue 类定义

声明

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

// 优先级队列数组的元素个数
const int MaxPQSize = 50;

class PQuene
{
    private:
      // 存放优先级队列的数组及队列中元素个数的计算器
      int count;
      DataType pqlist[MaxPQSize];
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
public:
    // 构造函数
    PQueue(void)
    // 修改优先级队列的操作
    void PQInsert(const DataType& item);
    DataType PQDelete(void);
    void ClearPQ(void);
    // 检测优先级队列状态的操作
    int PQEmpty(void) const;
    int PQFull(void) const;
    int PQLength(void) const;
};
```

说明

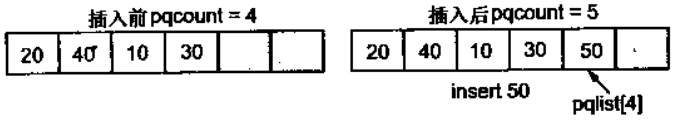
常量 MaxPQSize 定义了数组的大小；PQInsert 函数简单地往表中插入元素，它没有严格说明元素该存放在表中什么位置；PQDelete 函数从表中删除优先级最高的元素，我们假定优先级最高的元素是值最小的元素，最小值可用定义的小于比较符'<'来判定。

例

```
typedef int DataType;                    // 整型可用"<"进行比较
PQueue PQ;
PQ.PQInsert(20);
PQ.PQInsert(10);
cout << PQ.PQLength() << endl;           // 输出 2
N = PQ.PQDelete();                       // 取得 N = 10
```

类 PQueue 的实现

优先级队列的操作 与队列类似，优先级队列也有插入元素的操作。ADT 中没有规定元素插入表中的位置，而是将这个实现细节留给了 PQInsert 函数。在下面的实现中，我们先检测表是否已满。若是，则终止程序的执行；否则，将新元素插入由 pqlist[count]指定的表尾。



PQInsert

```
// 将元素插入优先级队列
void PQueue::PQInsert(const DataType& item)
{
    // 若队列已满，则退出程序
    if (count == MaxPQSize)
    {
        cerr << "Priority queue overflow!" << endl;
        exit(1);
    }
```

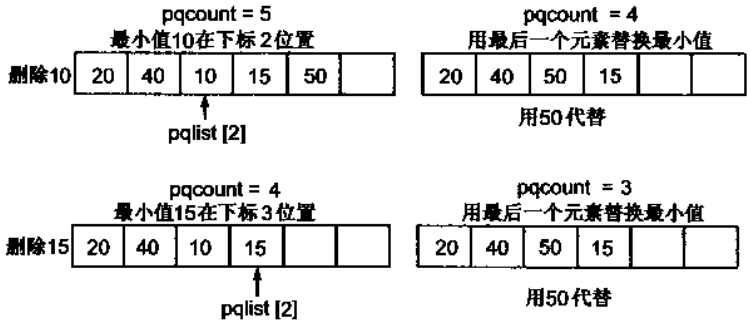

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 将元素置于队尾并使 count 加 1
    pqlist[count] = item;
    count + + ;
}
```

PQDelete 函数要求我们必须从表中删除优先级最高的元素，它并没有指定在有两个或多个元素可删除时应删除哪个，也没有指定在删除过程中应保持元素间的什么顺序。这些都是实现细节。在下面的实现中，我们首先判断表是否为空，若是，则退出程序；否则，我们扫描最小权值，用表中最后一个元素代替该元素，将表的长度减 1。这样，我们删除了优先级最高的元素。最后一个元素的下标就是新的 count 值。

在下述例表中，最小值是在下标 2 所指位置的 10。用下面的办法删除元素：表的长度减 1，并用表中最后一个元素（用 pqlist[count]得到）代替选中的元素，下一个该删除的元素是 15，即表中最后一个元素。



PQDelete

```
// 从优先级队列中删除元素并返回其值
DataType PQueue::PQDelete(void)
{
    DataType min;
    int i, minindex = 0;

    if (count > 0)
    {
      // 在 pqlist 中找到最小值及其下标
      min = pqlist[0];    // 假定 pqlist(0)为最小值

      // 顺序访问其它元素,修改最小值及下标
      for (i = 1; i < count; i + + )
          if (pqlist[i] < min)
          {
          // 新的最小值为 pqlist[i], 相应下标为 i
              min = pqlist[i];
              minindex = i;
          }
      // 将尾元素移入最小元素处并将 Count 减 1
      pqlist[minindex] = pqlist[count - 1];
      count - - ;
    }
    // 若 qlist 为空,则退出程序
    else
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    {
        cerr << "Deleting from an empty pqueue!" << endl;
        exit(1);
    }
    // 返回最小值
    return min;
}
```

PQInsert 函数的复杂度为 O(1),因为它直接访问表尾元素。但是,PQDelete 开始要扫描整个表来确定最小值及其下标,函数的复杂度为 O(n),n 为优先级队列的当前长度。

数据成员 count 存放表中元素的个数。该值用于 PQLength、PQEmpty 和 PQFull 的实现。这些函数的代码可在文件"apqueue.h"中找到。

应用:公司事务服务 公司员工可分为部门经理、主管及工人。我们建立一个具有不同级别的枚举类型,其顺序给出了当他们需要事务服务时的优先级。

```
// 员工的优先级别 (manager=0,…)
enum staff {Manager, Supervisor, Worker}, //manager=0,…
```

公司的事务服务由公共的秘书处提供。每个员工都可填写请求单要求服务,请求单上包括下述信息:请求服务员工的职务,服务的 ID 号,及该任务需要花费的时间。这些信息存放在记录 JobRequest 中。服务请求将输入到一个由员工级别来决定的优先级队列中,这个顺序用来对 JobRequest 的对象定义运算符'<'。

```
// 定义请求单的记录
struct JobRequest
{
    Staff staffPerson;
    int jobid;
    int jobTime;
};
// 重载运算符"<"来比较两个 JobRequest
int operator< (const JobRequest& a, const JobRequest& b)
{
    return a.staffPerson < b.staffPerson;
}
```

文件"job.dat"中存放着要装入优先级队列的服务请求表。应用程序假定服务请求已装入队列,正等待处理。程序从优先级队列中读出服务请求并执行。用数组 jobServicesUse 存放每类不同员工所花服务时间的总数。

```
// 对花费在不同类别员工上的服务计时
int jobServicesUse[3]={0,0,0};
```

打印函数 PrintJobInfo 和 PrintSupportSummary 给出每个服务及给公司中各个类别员工提供服务的时间两方面的信息。

```
// 打印一个 JobRequest 记录
void PrintJobInfo(JobRequest PR)
{
    switch(PR.staffPerson)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    {
        case Manager:        cout << "Manager        ";
                             break;
        case Supervisor:     cout << "Supervisor     ";
                             break;
        case Worker:         cout << "Worker         ";
                             break;
    }
    cout << PR.jobid << "      " << PR.jobTime << endl;
}

#include <iomanip.h>     // 用到 setw
// 输出每类员工的总服务时间
void PrintJobSummary(int jobServicesUse[])
{
    cout << "\nTotal Support Usage\n";
    cout << "  Manager    " << setw(3) <<
         << jobServicesUse[0] << endl;
    cout << "  Supervisor  " << setw(3) <<
         << jobServicesUse[1] << endl;
    cout << "  Worker     " << setw(3) <<
         << jobServicesUse[2] << endl;
}
```

程序 5.6　服务请求

每个服务请求均以记录形式存放于文件"job.dat"中。记录给出了员工类别('M','S','W'),服务 ID 号和服务所需时间。程序读入整个文件并将每条记录都插入到优先级队列 jobPool 中。输出时,从优先级队列中取出每个服务,用 PrintJobInfo 打印其信息。程序最后通过调用 PrintJobSummary 来打印服务的汇总信息。JobRequest 记录及有关函数放在文件"job.h"中。

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

#include "job.h"        // 定义 JobRequest

// 优先级队列中元素类型为
typedef JobRequest DataType;

#include "apqueue.h"      // 引入类 PQueue

void main()
{
    // 处理不超过 50 个服务请求单
    PQueue jobPool;

    // 从 fin 中读入服务请求单
    ifstream fin;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 为每类员工服务的时间
    int jobServicesUse[3] = {0, 0, 0};

    JobRequest PR;
    char ch;

    // 打开输入文件"job.dat"。若失败则退出程序
    fin.open("job.dat",ios::in | ios::nocreate);
    if (! fin)
    {
        cerr < < "Cannot open file 'job.dat'" < < endl;
        exit(1);
    }

    // 从文件中读入每个请求并插入到优先级队列 jobPool 中。
    // 每行的开始为表明员工类别的字符
    while(! fin. > > ch)
    {
        // 给 staffPerson 域赋值
        switch(ch)
        {
            case 'M':  PR.staffPerson = Manager;
                       break;
            case 'S':  PR.staffPerson = Supervisor;
                       break;
            case 'W':  PR.staffPerson = Worker;
                       break;
            default:    break;
        }
        // 读入 PR 的 jobid 和 jobTime
        fin > > PR.jobid;
        fin > > PR.jobTime;
        // 将 PR 插入优先级队列中
        jobPool.PQInsert(PR);
    }
        // 从优先级队列中删除各服务并输出该服务的信息
        cout < < "Category    Job ID    Job Time < < endl";
        while (! jobPool.PQEmpty())
        {
            PR = jobPool.PQDelete();
            PrintJobInfo(PR);
            // 对每类员工累计服务时间
            jobServicesUse[int(PR.staffPerson)] + = PR.jobTime;
        }
        PrintJobSummary(jobServicesUse);
}
/*
<输入文件"job.dat">
M 300 20
W 301 30
M 302 40
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
S 303 10
S 304 40
M 305 70
W 306 20
W 307 20
M 308 60
S 309 30
<程序 5.6 运行结果>
Category        Job ID        Job Time
Manager          300           20
Manager          302           40
Manager          308           60
Manager          305           70
Supervisor       304           40
Worker           306           20
Worker           307           20
Worker           301           30
Total Job Usage
    Manager       190
    Supervisor     80
    Worker         70
*/
```

5.7 实例研究:事件驱动模拟

对于现实世界的一些问题,我们可通过模拟创立其模型,以使我们对问题有更深的理解。模拟中可引入各种情况,并观察其影响。例如,飞行模拟器模拟恶劣条件来测试飞行员的反应,检查反应速度及反应是否适当。市场研究也可通过模拟来评价市场情况。本节中,我们通过程序模拟来研究银行顾客在一家有 $n \geq 2$ 个窗口的银行的到达和离开的情况。通过计算每位顾客的平均等待时间及每一窗口处于"繁忙"状态的百分比,我们可得出银行的服务效率。模拟中,我们还可分析不同的服务模式,并评价它们的"性能价格比"。实现中,我们定义代表银行活动的对象,用事件驱动来模拟这些活动,并用一些可能的值来描述预期的顾客到达率和职员为一个顾客服务所需的时间。我们可用随机数发生器来反映一个工作日中顾客到达和离开的情况。

同时,我们还可在模拟中改变参数值来测试如果改变顾客和职员的行为会对服务产生什么影响。例如,市场部估计送些小礼品给顾客可提高 20% 的客流量。我们就可研究是否会提高这么多并研究其对职员行为的影响。在某些时候,银行为维持合理的服务时间需增加开销来应付顾客的增加。这将减少送礼品带来的效果。模拟还可供银行经理来评价对顾客服务的水平,若顾客的平均等待时间太长,经理可增加新的窗口。这些模拟都可通过简单改变条件来重复多次。

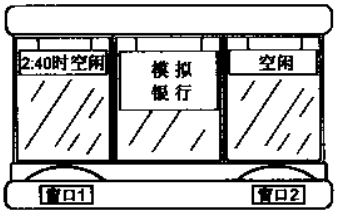
模拟设计

模拟程序运行过程中,需跟踪单个顾客的到达和离开。例如,如果第 50 位顾客在 2:30到达银行,他需要柜员 12 分钟的服务。模拟程序假定每个柜员都能准确地完成工作计划,这样,新来的顾客能知道该上哪个窗口,何时能得到服务。

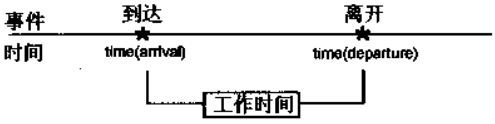
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在上述情况下，假定窗口 2 空闲，顾客将立即得到服务，并在 2:42 时离开。该顾客的等待时间为 0，窗口 2 的工作时间增加 12 分钟。

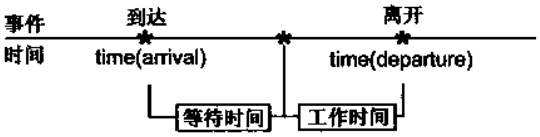



```
time(departure) = time(arrival) + servicetime
time(departure) = 2:30 + 0:12 = 2:42
```

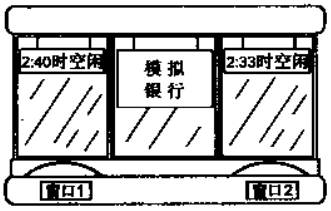


我们为第 50 位顾客设计一些其它情况。假定两个窗口都有客户，且顾客应排队等候服务。若窗口 1 在 2:40 时空出，而窗口 2 在 2:33 时就空出，那么这位顾客选择柜员 2，等待 3 分钟后用 12 分钟完成交易。顾客在 2:45 分离开，窗口 2 在这之后空出。

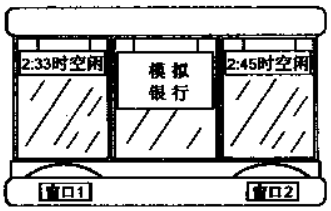
```
time(departure) = time(arrival) + waittime + servicetime
time(departure) = 2:30 + 0:03 + 0:12 = 2:45
```



在这一过程中，顾客选择完窗口后，就改变了该窗口的“何时空闲”标志，将对后来的顾客产生影响。



到达之前



到达之后

银行模拟的关键部分为客户事件，包括其到达和离开。执行过程中，我们可观察到客户的到达、被服务和离开事件的实际时间。这些事件定义为 C++ 的类，类中有表示顾客和职员的私有数据成员，如事件发生的时间 time，事件类型 etype(到达或离开)，对顾客的服务所需的时间 servicetime 和顾客被迫等待服务的时间 waittime。

顾客进入银行时产生到达事件，如第 50 位顾客在 2:30 到达，则相应的记录为：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

到达事件数据

2:30	Arrival	50	—	—	—
time	etype	CustomerID	tellerID	waittime	servicetime

到达事件不用 tellerID,waittime 和 servicetime 域。到达以后,这些域值才确定下来并产生离开事件。

顾客选定窗口后,我们才能给出确定顾客何时离开的离开事件。这个事件可用来重现顾客在银行中的行动。例如,下面是一在 2:30 到达,在窗口 2 等待 3 分钟,并在 12 分钟服务后离开的顾客记录。

离开事件数据

2:45	Departure	50	2	3	12
time	etype	CustomerID	tellerID	waittime	servicetime

对象的这些数据域描述了顾客在银行中的所有信息。类提供函数访问这些数据域,它们可用来得到整个银行服务的综合评价。

事件类的定义

声明

```
#include <iostream.h>
#include "random.h"    // 引入随机数发生器
// 区分两种类型的事件
enum EventType {arrival, departure};
class Event
{
    private:
        // 顾客号,窗口号,事件类别,事件发生的时间,服务时间及需等待时间
        int time;
        EventType etype;
        int customerID;  // 顾客编号为 1,2,3,…
        int tellerID;    // 窗口编号为 1,2,3,…
        int waittime;
        int servicetime;
    public:
      // 构造函数
      Event(void);
      Event (int t, EventType et, int cn ,int tn,
            int wt, int st);

      // 存取私有数据的方法
      int GetTime(void) const;
      EventType GetEventType(void) const;
      int GetCustomerID(void) const;
      int GetTellerID(void) const;
      int GetWaitTime(void) const;
      int GetServiceTime(void) const;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

说明

缺省的构造函数使得对象可在声明之后通过赋值来初始化，第二个构造函数使其可在对象定义时即被初始化。其它的函数返回数据成员的值。

例

```
Event e; // 声明一个使用缺省构造函数的事件

// 顾客 3 在 120 分钟时离开。他一共等候了 10 分钟，在 1 窗口用 5 分钟完成交易

e = Event(120, departure, 3,1,10,5);
cout < < e.GetService();      // 输出服务时间为 5
```

例中时间从模拟运行开始以分为单位计算。

模拟数据

在模拟运行时，我们可以收集到每个窗口服务顾客的总人数，顾客等待时间及窗口每天提供的服务时间等数据。我们在记录 TellerStats 中存放这些数据，该记录中还有一个域 finishService，表示该窗口何时可对新顾客提供服务。

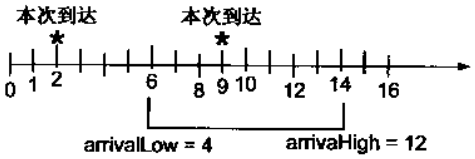
TellerStats 记录

finishService	totalCustomerCount	totalCustomerWait	totalService

```
// 有关窗口信息的结构
struct TellerStats
{
    int finishService;        // 该窗口何时空闲
    int totalCustomerCount;   // 服务顾客数
    int totalCustomerWait;    // 顾客等候时间总计
    int totalService;         // 服务时间总计
};
```

模拟产生了每个顾客的到达和离开事件。所有事件均被加上时间戳，然后放到优先级队列中，队列中优先级最高的事件就是时间戳最早的事件。这种表结构使我们可以按顾客的到达和离开的时间顺序处理这些事件。

银行模拟中用随机数发生器来决定下个顾客何时到达及当前顾客所需的服务时间，它可保证事件在某一范围内等概率发生。如果当前到达事件发生在 T 时刻，下一个到达事件将发生在范围 T + arrivalLow 和 T + arrivalHigh 之间。而每个顾客的服务时间在范围 ServiceLow 到 ServiceHigh 之间。例如，假定顾客在第 2 分钟时到达，下一到达事件的范围为 arriveLow = 4 到 arriveHigh = 12 分钟后，那么下一个顾客可能在第 6,7,...,14 分钟到达，共有 9 种可能性。



上图中第一个顾客在第 2 分钟到达，下个顾客到达事件在第 9 分钟，我们可创建这些

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

事件，并将其放入优先级队列中供模拟程序处理。

实现上述模拟的数据和函数都封装在类 Simulation 中，其数据成员包括模拟的时间（单位：分钟），窗口号，下个顾客的编号。包含每个柜员信息的 TellerStats 记录数组放有事件表的优先级队列。类中还存放着下一到达事件发生的时间范围和当前顾客的服务时间范围。

类 Simulation 定义

声明

```
class Simulation
{
    private:
      // 模拟中用到的数据
      int simulationLength;            // 模拟长度
      int numTellers;                  // 窗口个数
      int nextCustormer;               // 下一顾客号
      int arrivalLow, arrivalHigh;     // 下次到达时间段
      int serviceLow, serviceHigh;     // 服务时间段
      TellerStats tstat[11];           // 最多 10 个窗口
      PQueue pq;                       // 优先级队列
      RandomNumber rnd;                // 随机数用于到达和服务时间

      // RunSimulation 用到的私有函数
      int NextArrivalTime(void);
      int GetServiceTime(void);
      int NextAvailableTeller(void);
    public:
      // 构造函数
      Simulation(void);

      void RunSimulation(void)            // 执行模拟
      void PrintSimulationResults(void); // 输出结果
};
```

说明

构造函数初始化数组 TellerStats 并将 nextCustomer 赋值为 1。数组 tstat 的下标 0 不用。这样，窗口号可直接用作下标来访问数组 tstat。然后，构造函数提示用户输入模拟参数，包括模拟多长时间，窗口个数及到达事件和服务时间的范围。类提供的窗口个数上限为 10 个。

对每个到达事件，我们可调用函数 NextArrivalTime 来判定下一顾客何时到达银行。同时，也可调用 GetServiceTime 来得到当前顾客要占用窗口多长时间，用 NextAvailableTeller 来判定下一顾客该选择哪个窗口。

函数 RunSimulation 执行模拟过程，PrintSimulationResults 输出最后的统计结果。

建立模拟

模拟的建立包括用构造函数初始化模拟数据并读入模拟参数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Simulation:Simulation(void)
{
    int i;
    Event firstevent;
    // 初始化窗口信息参数
    for (i = 1;i < = 10; i + + )
    {
        tstat[i].finishService = 0;
        tstat[i].totalService = 0;
        tstat[i].totalCustomerWait = 0;
        tstat[i].totalCustomerCount = 0;
    }
    nextCustomer = 1;
    // 读入用户输入的模拟条件
    cout < < "Enter the simulation time in minutes: ";
    cin > > simulationLength;
    cout < < "Enter the number of bank tellers: ";
    cin > > numTellers;
    cout < < "Enter the range of arrival times in minutes: ";
    cin > > arrivalLow > > arrivalHigh;
    cout < < "Enter the range of service times in minutes: ";
    cin > > serviceLow > > serviceHigh;

    // 产生第一个到达事件,它不需要窗口号等待时间和服务时间
    pq.PQInsert(Event(0,arrival,1,0,0,0));
}
```

运行模拟

模拟时间(simulationLength)用来判定是否应该生成下一顾客的到达事件。若银行在某一顾客到达时间之前关门(arrival time > simulationLength),则银行将不接待新顾客,只对还在银行内的顾客提供服务。数据成员 nextCustomer 记录整个模拟过程的顾客总数,由于模拟中没有顾客名,只对顾客进行编号,则 nextCustomer 也可作为下一到达事件的顾客号。函数 NextArrivalTime 产生下一顾客到达与当前顾客到达的时间差,其内部是用随机数发生器及参数 arrivalLow 和 arrivalHigh 来得到返回值。一个类似的函数还有使用参数 serviceLow 和 serviceHigh 的 GetServiceTime,它返回顾客所需服务的时间。

```
// 决定下一到达事件的时间
int Simulation::NextArrivalTime(void)
{
return arrivalLow + rnd.Random(arrivalHigh - arrivalLow + 1);
}
// 决定客户服务时间
int Simulation::GetServiceTime(void)
{
return serviceLow + rnd.Random(serviceHigh - serviceLow + 1);
}
```

在模拟过程中,刚到的顾客总是先看每个窗口上标出的何时能提供服务的时间,顾客据此来选择窗口。每个窗口记录中的 finishService 值就是窗口上标出的值。函数 Nex-

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

tAvailableTeller 扫描每个窗口并返回 finishService 的最小值。如果所有窗口都要在关门之后才空闲，则随机地给顾客安排一个窗口。

```
// 返回第一个空闲窗口
int Simulation:NextAvailableTeller(void)
{
    // 开始时假定所有窗口在关门时都结束服务
    int minfinish = simulationLength;

    // 给在关门之前到达但服务时间超出关门时间的顾客随机分配一个窗口
    int minfinishindex = rnd.Random(numTellers) + 1;

    // 找到最先空闲的窗口
    for (int i = 1; i <= numTellers; i++)
        if (tstat[i]. finishService < minfinish)
        {

            minfinish = tstat[i].finishService;
            minfinishindex = i;
        }
    return minfinishindex;
}
```

模拟程序的主要函数为 RunSimulation，它处理优先级队列中的事件。初始时，队列中只有一个单独的到达事件，这是在银行开门时发生的。从此，整个处理过程的时间由函数本身调控。函数 RunSimulation 是一个从优先级队列中读入事件的循环。这种循环我们称之为事件循环，它在队列为空时结束。银行的窗口在关门后(time > SimulationLength)继续为在关门前到达的顾客服务。程序的最后处理在关门后离开的顾客，我们将 SimulationLength 改为最后一个离开事件的时间。这个时间可用下面语句计算得到。若 e 是存放当天最后一个事件的 Event 对象，e.GetTime( )返回该事件的时间，语句为：

```
// 调整模拟时间，加上窗口的超时
SimulationLength = ( e.GetTime( ) <= SimulationLength)? SimulationLength:e.Get-
Time( );
```

下面分别介绍对两种事件的处理：

到达

1. 刚到达的顾客负责给出下一个到达事件，然后将其放入优先级队列中等待后续处理。若下一到达事件发生在模拟结束之后，则取消。

```
// 计算下次到达的时间
nexttime = e.GetTime() + NextArrivalTime();

if (nexttime > simulationLength)
   // 处理已有事件，但不再产生新事件
   continue;
else
{
    // 为下一顾客产生到达事件并放入队列中 nextCustomer++;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    newevent = Event(nexttime, arrival,
                     nextCustomer, 0, 0, 0);
    pq.PQInsert(newevent);
}
```

2. 创建下一到达事件后，修改 TellerStats 的信息，然后创建离开事件。我们先初始化 servicetime，确定好当前顾客所需的服务时间，然后再决定可提供服务的第一个窗口。

```
// 顾客所需服务时间
servicetime = GetServiceTime( );
// 提供此次服务的窗口
TellerID = NextAvailableTeller( );
```

下面再看提供此次服务的窗口的 finishService 域。若 finishService 不为 0，则它为该窗口可提供此次服务的时间；若 finishService 为 0，则该窗口空闲。此时，将其 finishSerice 置为当前时间表示该窗口为顾客服务。顾客等待时间(waittime)为 finishService 减去当前时间，若窗口空闲，将其 finishService 域置为当前时间。

```
// 若窗口空闲，置其 finishService 域为当前时间
if (tstat[tellerID].finishService = = 0)
tstat[tellerID].finishService = e.GetTime( );

// 计算顾客等待时间
waittime = tstat[tellerID].finishService - e.GetTime( );
```

再用以上数值，修改该窗口的 TellerStates 信息。finishService 域值增加了当前顾客所需服务的时间。

```
// 修改窗口的统计数据
tstat[tellerID].totalCustomerWait + = waittime;
tstat[tellerID].totalCustomerCount + + ;
tstat[tellerID].totalService + = Servicetime;
tstat[tellerID].finishService + = Servicetime;
```

3. 最后，定义离开事件并将其放到优先级队列中。建立时间所必需的所有参数都已具备。

事件数据项	传递的参数
time	tstat[tellerID].finishService
etype	departure
customer	e.GetCustomerID()
tellerID	tellerID
waittime	waittime
servicetime	servicetime

```
newevent = Event(tstat[tellerID].finishService,
                 departure, e.GetCustomerID(), tellerID,
                 waittime, servicetime);
pq.PQInsert(newevent);
```

离开

离开事件使我们可以访问客户在银行期间的历史活动情况。在模拟研究中，此信息可以打印出来。根据离开事件，如果柜员没有其他客户，我们就必须更新 finishService

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

域。若 finishService 的当前值等于离开时间则发生这种情况。此时，置 finishService 的值为 0。

```
tellerID = e.GetTellerID();
// 若该窗口无人等待,则将其置为空闲
if (e.GetTime() = = tstat[tellerID].finishService)
    tstat[tellerID].finishService = 0;
```

模拟过程汇总 结束模拟时，可以调用 PrintSimulationResults。它打印出客户和各柜员的数据汇总。数据由 TellerStats 记录进行汇总。该记录中包含每个柜员所服务的客户人数以及客户的累计等待时间。

```
for (i = 1; i < = numTellers; i + + )
{
    cumCustomers + = tstat[i].totalCustomerCount;
    cumWait + = tstat[i].totalCustomerWait;
}
```

最后汇总可求出客户的平均等待时间和每个柜员处于忙碌状态的时间百分比。

```
cout < < endl;
cout < < "* * * * * * * * Simulation Summary * * * * * * *" < < endl;
cout < < "Simulation of " < < simulationLength
     < < "minutes" < < endl;
cout < < "  No.  of Customers:  " < < cumCustomers < < endl;
cout < < "  Average Customer Wait: ";

avgCustWait = float(cumWait)/cumCustomers + 0.5;
cout < < avgCustWait < < "minutes" < < endl;
for(i = 1;i < = numTellers; i + + )
{
    cout < < "    Teller #" < < i < < " % Working: ";
    // 显示最接近的百分数
    tellerWork = float(tstat[i].totalService)/simulationLength;
    tellerWorkPercent = tellerWork * 100.0 + 0.5;
    cout < < tellerWorkPercent < < endl;
}
```

模拟过程示范 主程序中定义了一个 Simulation 对象 S，然后执行由 RunSimulation 实现的事件循环。在事件循环结束后，再调用 PrintSimulationResults。

例 5.6

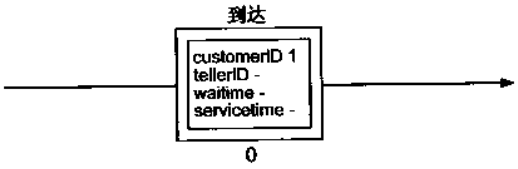
我们先定义如下的模拟过程参数再跟踪模拟过程：

simulationLength = 30(分钟)	numTellers = 2
arrivalLow = 6	arrivalHigh = 10
serviceLow = 18	serviceHigh = 20

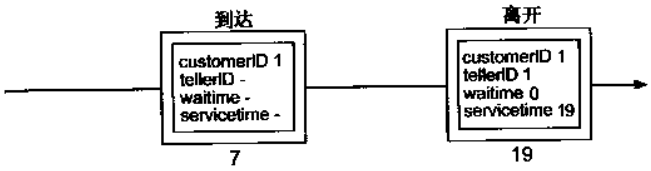
客户 1 在 0 分钟这一时刻到达。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

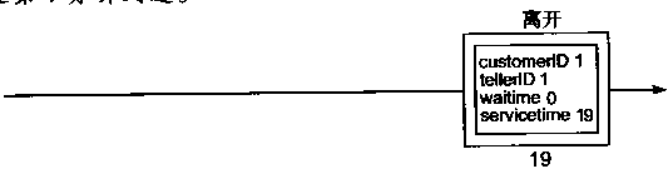
www.Linuxidc.com



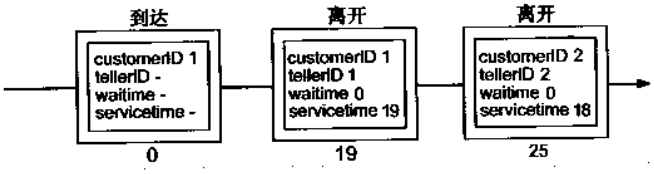
客户 1 在第 7 分钟时为客户 2 生成一个到达事件，第 19 分钟时生成她自己的离开事件。



客户 2 在第 7 分钟到达。

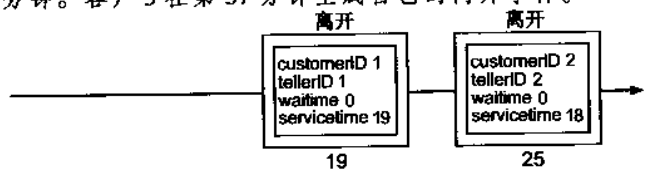


客户 2 在第 16 分钟时为客户 3 生成一个到达事件，第 25 分钟时生成他自己的离开事件。



客户 3 在第 16 分钟到达。

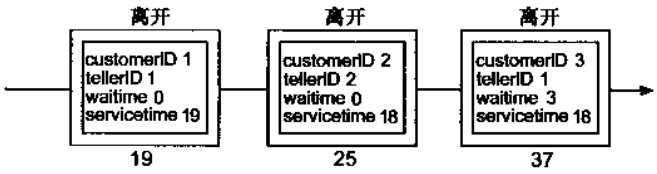
客户 3 生成一个在银行关门以后的到达事件以终止到达事件的生成。柜员 1 使客户 3 等待 3 分钟。客户 3 在第 37 分钟生成自己的离开事件。



按如下顺序从优先级队列中删除离开事件：客户 1 于第 19 分钟，客户 2 于第 25 分钟，客户 3 于第 37 分钟。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



银行在营业 37 分钟后关门。

程序 5.7　代码及执行情况

本程序运行两次。第 1 次运行对应于例 5.6 中的数据。第 2 次运行用一个 480 分钟的模拟过程。输出方法 PrintSimulationResults 给出了光顾次数、每个客户的平均等待时间、以及每个柜员的忙碌时间。Event，TellerStats 以及 Simulation 类的实现包含于文件"sim.h"中。

```
#include "sim.h"
void main(void)
{
    // 定义模拟对象 S
    Simulation S;
    // 执行模拟过程
    S.RunSimulation();
    // 输出模拟结果
    S.PrintSimulationResults();
}
/*
<程序 5.7 运行结果之一>
Enter the simulation time in minutes: 30
Enter the number of bank tellers: 2
Enter the range of arrival times in minutes: 6 10
Enter the range of service times in minutes: 18 20
Time:  0   arrival of customer 1
Time:  7   arrival of customer 2
Time: 16   arrival of customer 3
Time: 19   departure of customer 1
        Teller 1 Wait 0 Service 19
Time: 25   departure of customer 2
        Teller 2 Wait 0 Service 18
Time : 37   departure of customer 3
        Teller 1 Wait 3 Service 18

* * * * * * * * Simulation Summary * * * * * * *
Simulation of 37 minutes
    No. of Customers:  3
    Average Customer Wait: 1 minutes
    Teller #1 % Working: 100
    Teller #2 % Working: 49
<程序 5.7 运行结果之二>
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Enter the simulation time in minutes 480
Enter the number of bank tellers 4
Enter the range of arrival times in minutes 2 5
Enter the range of service times in minutes 6 20
< arrival and departure of 137 customers >
* * * * * * * * Simulation Summary * * * * * * * *
Simulation of 521 minutes
    No. of Customers: 137
    Average Customer Wait: 2 minutes
    Teller #1 % Working: 89
    Teller #2 % Working: 86
    Teller #3 % Working: 83
    Teller #4 % Working: 86
*/
```

书面作业

5.1 找出所有合适的答案。“栈”结构实现的是

(a) 先进/后出　(b) 后进/先出　(c)先来/先服务

(d) 先进/先出　(e) 后进/后出

5.2 为栈编写两个应用程序。

5.3 以下栈操作的输出是什么？(DataType = int)；

```
Stack S;
int x = 5, y = 3;

S.Push(8);
S.Push(9);
S.Push(y);
x = S.Pop();
S.Push(18);
x = S.Pop();
S.Push(22);
while (! S.StackEmpty())
{
    y = S.Pop();
    cout < < y < < endl;
}
cout < < x < < endl;
```

5.4 编写函数

```
void StackClear(Stack& S);
```

清空栈 S。为什么用引用传递 S 这一点很关键？

5.5 下列函数完成什么操作？(DataType = int)；

```
void Ques5(Stack& S)
{
    int arr[64], n = 0, i;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    int elt;

    while(! S.StackEmpty())
      a[n++] = S.Pop();
    for (i=0; i < n; i++)
      S.Push(a[i]);
}
```

5.6 以下代码段执行什么操作？

```
Stack S1, S2, tmp;
Datatype x;
    ...
while (! S1.StackEmpty())
{
    x = S1.Pop();
    tmp.Push(x);
}
while (! tmp.StackEmpty())
{
    x = tmp.Pop();
    S1.Push(x);
    S2.Push(x);
}
```

5.7 编写函数

```
int StackSize(Stack S);
```

使用栈操作返回栈 S 中的元素个数。

5.8 以下的函数 Ques8 执行什么操作？(DataType = int)；

```
void Ques8(Stack& S, int n)
{
    Stack Q;
    int i;
    while(! S.StackEmpty{})
    {
        i = S.Pop();
        if (i ! = n)
           Q.Push(i);
    }
    while)! Q.StackEmpty()(
    {
        i = Q.Pop();
        S.Push(i);
    }
}
```

5.9 如果 DataType 为 int，编写函数

```
void SelectItem(Stack& S, int n);
```

使用栈操作查找数据项 n 在栈中的第 1 次出现之处并将其移至栈顶。其他元素保

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

持原来的次序。

5.10 将中缀表达式转换为后缀式：

(a) a + b * c

(b) (a + b)/(d - e)

(c) (b2 - 4 * a * c)/(2 * a)

5.11 以中缀式改写以下表达式：

(a) ab + c *

(b) abc + *

(c) abcde + + * * ef - *

5.12 选出所有合适的答案。队列结构实现的是

(a) 先进/后出　(b)后进/先出　(c)先来/先服务

(d) 先进/先出　(e)后进/后出

5.13 选出所有合适的答案。队列结构可用于

(a) 表达式求值

(b) 操作系统作业调度

(c) 排队过程的模拟

(d) 按逆序打印表

5.14 以下队列操作的输出是什么？(DataType = int)；

```
Queue Q;
DataType x = 5, y = 3;

Q.QInsert(8);
Q.QInsert(9);
Q.QInsert(y);
x = Q.QDelete();
Q.QInsert(18);
x = Q.QDelete();
Q.QInsert(22);
while(! Q.QEmpty())
{
    y = Q.QDelete();
    cout < < y < < endl;
}
cout < < x < < endl;
```

5.15 以下函数执行什么操作？(DataType = int)；

```
void Ques15(Queue& Q, int n = 50)
{
    Stack S;
    int elt;
while (! Q.QEmpty())
    {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        elt = Q.QDelete();
        S.Push(elt);
    }
    while(! S.StackEmpty())
    {
        elt = S.Pop();
        Q.QInsert(elt);
    }
}
```

为什么 Q 由引用传递这一点很关键？

5.16 以下代码段执行什么操作？

```
(DataType = int)

Queue Q1, Q2;
int n = 0, x;
    . . .
while (! Q1.QEmpty())
{
    x = Q1.QDelete();
    Q2.QInsert(x);
    n++;
}
for (int i=0; i < n; i++)
{
    x = Q2.QDelete();
    Q1.QInsert(x);
    Q2.QInsert(x);
}
```

5.17 假设优先级队列中包含整数值，用小于运算符“<”定义优先级次序。当前表中包含下列元素

45	15	50	25	65	30

通过跟踪 PQueue 类中的 PQDelete 和 PQInsert 方法，描述下列各条指令执行以后表的内容。

```
(a) Item = pq.PQDelete()      Item =    List:
(b) pq.PQInsert(20)                     List:
(c) Item = pq.PQDelete()      Item =    List:
(d) Item = pq.PQDelete()      Item =    List:
```

5.18 改写 PQDelete 例程以保证具有同一优先级的数据项 FIFO(先进先出)次序。

上机题

5.1 编写方法

```
void Qprint(void);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

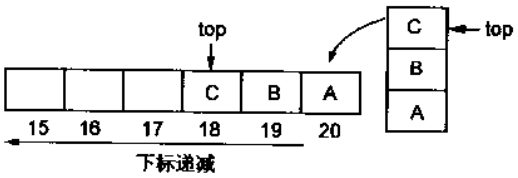
www.Linuxidc.com

打印队列内容，每行 8 个元素。编写一段程序，从文件“pq.dat”中将 20 个双精度数值输入到队列中。打印元素值。

5.2 往数组中读入 10 个整数，并逐个压入栈。打印原始表并通过弹出元素的方法打印栈内容。显然，第 2 次打印时元素按逆序列出。

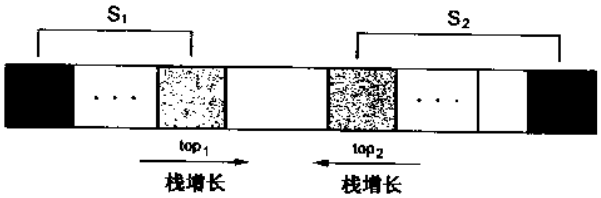
5.3 栈可以用来识别某些模式。考察模式 STRING1 # STRING2，其中任一串都不含“#”且 STRING2 必须是 STRING1 的逆串。例如，串 123&^a # a^&321 与该模式匹配，而串 a2qd # dq3a 则不匹配。编写程序读入 5 个串并指出各个串是否与该模式匹配。

5.4 在另一种模型的栈中，栈是按数组索引减小的方向增长的。初始时栈为空且 Top = 21。往栈中压入 3 个字符后，索引 Top = 18，栈顶的元素为 LIST[TOP] = C。



在此模型中索引 Top 在每次 Push 索引操作后减小，而在 Pop 操作后则增大。用此模型实现 Stack 类。运行程序 5.1 进行测试。

5.5 可以用一个数组存放两个栈，一个从左端开始增长，一个从右端增长。



(a) S_1 为空的条件是什么？S_2 何时为空？

(b) S_1 为满的条件是什么？S_2 何时为满？

(c) 实现 DualStack，其声明如下。

```
const int MaxDualStackSize = 100
class DualStack
{
    private:
        int top1, top2;
        DataType stackStorage[MaxDualStackSize];
    public:
        DualStack(void);
        // 往栈 n 中压入 elt
        void Push(DataType elt, int n);
        // 从栈 n 中弹出
        DataType Pop(int n);
        // 取栈 n 的顶元素
        DataType Peek(int n);
        // 栈 n 为空否？
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        int StackEmpty(int n);
        // 栈 n 已满否?
        int StackFull(int n);
        // 清空栈 n
        void ClearStack(int n);
};
```

(d) 编写一个主程序,读取 20 个整数,将所有偶数压入一个栈,将奇数压入另一个栈。打印每一个栈的内容。

5.6 读取一行文本,将所有非空格字符放到队列和栈中。检测文本是否回文。

5.7 扩展后缀计算器,增加由符号@表示的单目减。例如,输入表达式

```
7
@
12
+
    <Display> 5
```

5.8 本题将后缀计算器扩展到包括对含一个变量的表达式求值。新的声明为

```
class Claculator
{
    private:
        Stack S;
        struct component              // 新
        {
            short type;
            float value;
        };
        int expsize:                  // 新
        component expComponent[50];   // 新
        int GetTwoOperands (double& operand1,
                            double& operand2);
        void Enter(double num);
        void Compute(char op);
    public:
        Calculator(void);
        void Run(void);
        void Clear(void);
        void Variable(void);          // 新
        double Eval(double x);        // 新
};
```

Variable 操作允许用户输入一个包含变量 x 以及数值和运算符的表达式。例如,若要求中缀表达式 x^2+3x+5 的值则输入表达式

```
x x * 3 x * + 5 +
```

每读入一个部件时,在数组 expComponent 中生成一个表项。每一项的 type 域设置如下:1 = 数值,2 = 变量,3 = +,4 = -,5 = *,6 = /。如果 type = 1,数值被存到 value 域。每生成一个新表项,expsize 增 1。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

成员函数 Eval 遍历数组 expComponent 的 expsize 个成员并用栈对表达式求值。每当元素的 type 域为 2 时，将参数 x 压入栈。

写一个主程序，对以下表达式求值

$$(x^2 + 1) / (x^4 + 8x^2 + 5x + 3)$$

x 的取值分别为 $x = 0, 10^1, 10^2, 10^3, \cdots, 10^8$。

5.9 修改程序 5.7 中的模拟研究方案，使其包含以下内容：

(a) 输出客户在银行平均花费时间。客户时间是按从到达到离开计算的。

(b) 现时，所有柜员将一直呆在银行，直到最后一个客户离开为止。当在关门时间后让柜员接待一名晚到的客户时，这就显得不太经济。如果银行已关门且没有客户等待服务时，允许一名柜员离开。提示：在 TellerStat 记录中增加一个域记录柜员在银行所呆的时间。用此变量计算柜员处于忙碌时间的百分比。

5.10 假设银行对每个柜员都单独设立客户队列。当客户到达时，他或她总是选择最短的队列而不是考虑柜员的工作负荷。修改模拟程序 5.7，求出客户的平均等待时间以及每个柜员所做的工作量。将此结果与单队列模型的结果相比较。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第6章 抽象操作

6.1 运算符重载
6.2 有理数
6.3 有理数类
6.4 作为成员函数的有理数运算
6.5 作为友元函数的有理数流运算符
6.6 有理数的转换
6.7 有理数的使用
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

抽象数据类型定义了初始化和管理数据的函数集，C++语言以类这一强有力的工具实现了抽象数据类型(ADT)。本章中，我们将语言中定义的运算符(如+，[]等)扩展到抽象类型。我们把这个过程称为运算符重载，它将标准的运算符重定义为实现抽象类型运算的运算符。运算符重载是面向对象程序设计语言的重要特性。它使我们可以将标准的运算符用于具有相关特性的类函数。例如，类 Matrix 中用 AddMat 和 MultMat 定义了矩阵加法和矩阵乘法，通过重载，我们可用熟悉的中缀运算符"+"和"*"来代表它们。若 P、Q、R、S 是 Matrix 的对象：

标准的类函数	重载运算符
R = P.AddMat(Q);	R = P + Q;
S = R.MultMat(Q.AddMat(P));	S = R * (Q + P);

关系表达式定义了元素间的顺序，如整数 3 是正数，10 比 15 小。

$3 \geq 0$　　$10 < 15$

顺序的概念并不只在数值中存在。在第 3 章中的类 Date 中，两个日期对象可以通过年、月、日进行比较：

```
Date(6,6,44) < Date(12,25,80)      // D 日在 1980 年圣诞之前
Date(4,1,99) = = Date("4/1/99")    // 愚人节的两种表示法
Date(7,1,94) < Date("8/1/94")      // 7 月在 8 月之前
```

C++允许对其本身的大多数运算符重载，包括流运算符。程序员无法创造一个新运算符，必须通过对现有操作符的重载来定义新的运算。

类 Date 中的函数 PrintDate 读入对象 D 并以标准形式输出年、月、日各个域。我们可通过重写"<<"操作，并将其用于流 cout 中来实现相同的函数。例如，Date 对象 D(12,31,99)定义了 20 世纪的最后一天，为产生下列输出：

```
The last day of the 20th century is December 31, 1999
```

我们可用第 3 章提供的打印函数，也可用操作符重载。

```
PrintDate 函数：
    cout < <"The Last day of the 20th century is";
    D.Printdate( );

重载"< <"运算符：
    cout < <"The last day of the 20th century is"< < D;
```

运算符重载在本书多章中介绍。本章我们主要介绍算术运算符，有理数运算符及流的 I/O 的重载。我们以有理数类为例来介绍这些基本概念，因为，"分数"大家都很熟悉，在学校中也有过多年的计算练习。类 Rational 在介绍算术及有理数运算、整型和实型的类型转换和简单流的 I/O 的重载都是很好的例子。

6.1 运算符重载

运算符重载可使语言定义的运算符如+、/、-、[ ]和=等经重载定义后用于类类型。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

这样，我们可以通过类用我们熟悉的运算符来实现抽象数据类型的运算。C++提供了多种实现运算符重载的途径，包括用户定义外部函数、类成员及友元函数等。C++要求重载后的运算符至少有一个参数是对象或对象地址。

用户定义外部函数

在类 Seqlist 中，Find 和 Delete 函数要求 DataType 上定义了关系运算符"==”。若该类型本身不具备这个运算符，用户则需显式定义一个。定义中用标识符"operator"和符号"==”，由于该运算的结果是 True 或 False，故返回值的类型为 int：

```
int operator ==(const DataType& a, const DataType& b);
```

例如，如果表中存放的是包括雇员 ID，姓名，地址等域的记录 Employee。

```
struct Employee
{
    int ID
        ...
}
```

ID	Name	Address	...

可定义比较 ID 域的关系运算符等于：

```
int operator ==(const Employee& a, const Employee& b)
{
    return a.ID==b.ID;     //比较 ID 域
}
```

重载后的运算符必须有一个类型为类的参数。下面将"==”重载到串中的努力是徒劳的，因为参数中不存在对象或对象的地址。

```
int  operator==(char *s, char *t)     //无效重载
{
    return strcmp(s,t)==0;
}
```

关系运算符等于定义好后，用户就可使用类 Seqlist 的 Find 和 Delete 函数。

```
typedef Employee Datatype;        //数据类型为 Employee
#inclde "aseqlist.h"
      ...
Seqlist L;
Employee emp;                     // 定义一个 Seqlist 对象
      ...
emp.ID=1000;                      // 找寻 ID 为 1000 的员工
if (L.Find(emp))
    L.Delete(mp)                  // 找到后，删除该员工
```

综上所述，为使用类 Seqlist，用户必须将运算符"==”定义为一个外部函数。这是一个额外但又必须的要求，因为"==”运算符定义在数据类型上，而不是定义在类中。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类成员

一般的数据类型已定义了算术和关系运算符。程序员可以中缀表达式的格式用它们连接操作数。类也可能以相似的方式来连接对象。多数情况下，这些函数可用标准的C++运算符重载为类的成员函数。当左操作数为对象时，该运算符以定义在对象上的函数形式执行。我们以二维向量运算中的加法，求反和点积运算为例。

向量加法

两个向量 $u=(u_1,u_2)$ 和 $v=(v_1,v_2)$ 相加，结果也为一个向量。(图 6.1(a))

$u+v=(u_1+v_1,u_2+v_2)$

向量求反

对向量 $u=(u_1,u_2)$ 求反是对其每个分量求反。新向量与原向量长度相同，但方向相反(图 6.1(b))。

$-(u_1,u_2)=(-u_1,-u_2)$

向量乘法(点积)

向量 $u=(u_1,u_2)$ 和向量 $v=(v_1,v_2)$ 相乘，结果为对应分量相乘之和(图 6.1(c))，也就是向量长度之积再乘以向量间夹角的余弦。

即：$u\cdot v=u_1v_1+u_2v_2=\text{magnitude}(u)*\text{magnitude}(v)*\cos(\theta)$

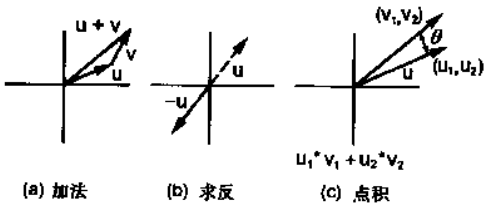


图 6.1 向量运算

我们定义向量类 Vec2d，其坐标 x 和 y 为私有数据成员。该类的成员函数包括一个构造函数，两个二目运算符(加，点积)和一个单目运算符(求反)，还有两个定义向量数量和输出流的友元函数。友元函数的用法在后面讨论。

```
class Vec2d
{
    private:
        double x1,x2;                    // 坐标值

    public:
        // 带缺省值的构造函数
        Vec2d(double h=0.0,double v=0.0);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 成员函数
        Vec2d operator-(void);              // 求反
        Vec2d operator+ (const Vec2d& V);   // 加法
        double operator* (const Vec2d& V);  // 点积

        // 友元函数
        friend Vec2d operator* (const double c, const Vec2d& V);
        friend ostream& operator<< (ostream& os const Vec2d& U);
};
```

上述成员函数加、求反和点积以表达式中的左操作数为当前对象，并假定 U 和 V 均为向量。

加法： 表达式 U+V 用二目运算符"+"与对象 U 结合，以参数 V 调用函数

```
operator+U.operator+(V)        //返回值为 Vec2d(x+V.x,y+V.y)
```

求反： 表达式 -U 使用单目运算符"-"，对对象 U 执行函数

```
operator-U.operator-()         //返回值为 Vec2d(-x,-y)
```

点积： 与加法类似，表达式 U*V 用双目运算符"*"与对象 U 结合，用函数对当前对象 U 和参数 V 求积。

```
U.operator*(V)                 // 返回值为 x*V.x+y*V.y
```

例如，有两个对象

```
Vec2d  U(1,2), V(2,3)
U+V=(1,2)+(2,3)=(3,5)
-U=-(1,2)=(-1,-2)
U*V=(1,2)*(2,3)=8
```

运算符重载满足下列条件：

1. 重载的运算符必须遵循重载前运算符遵循的运算优先级、结合律及操作数个数。如"*"是二目运算符，重载后也必须是两个操作数；
2. C++语言中除下述运算符外，所有运算符均可重载：

   ,(逗号运算符) sizeof ::(类范围运算符) ?:(条件表达式)

3. 重载后的运算符不允许使用缺省参数。换句话说，运算符的所有操作数都必须给出。例如，下面的运算符定义是非法的：

   ```
   double Vec2d::operator*(Vec2d V=Vec2d(1,1));
   ```

4. 运算符重载为成员函数时，与运算符结合的总是左操作数；
5. 作为成员函数时，单目运算符没有参数，双目运算符有一个参数。

友元函数

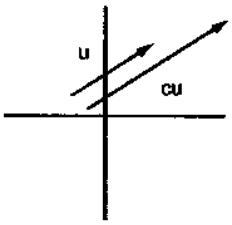
典型的面向对象程序设计规定只有成员函数才能访问类中的私有数据成员，这才能保证数据封装性和信息隐藏性。这条原则应尽可能地适用于运算符重载。然而，有些情况下，用成员函数来实现运算符重载是不可能的，或者虽然可能，也特别不方便。我们需

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

要用定义在类外部但可访问私有数据成员的友元函数。例如，标量乘是另一种形式的乘法，它由一个数乘以一个向量，其结果为向量的每个分量乘以该数后组成的向量。即：数 c 乘以向量 $U=(u_1,u_2)$ 的结果为将 c 乘以向量 U 的每个分量：

$$c*U=(cu_1,cu_2)$$



这个运算用重载号“*”来实现是很自然的。

然而，使用成员函数实现的运算符重载不允许将 c 作为左操作数，左操作数必须是向量 U。我们可将“*”定义为类外部运算符，它有两个参数 c 和 U，可以访问参数 U 的私有成员 x 和 y。这可通过在类的说明内部将操作符定义为 friend 来实现。友元函数定义时，关键字 friend 置于函数说明之前。

```
friend Vec2d operator *(const double c,const Vec2d& U);
```

由于友元函数的实现在类外部，不在类的范围之内，所以可以作为标准函数实现，注意此时不要在函数名前加关键字 friend。

```
Vec2d operator *(double c, Vec2d U)
{
    return Vec2d(c*U.x, c*U.y);
}
```

使用标量乘时，两个参数都传递给“*”，且标量必须是第一个参数。

```
Vec2d  Y(8,5),Z;
Z=3.0*Y;    //结果 Z 为(24,15)
```

友元函数的使用及它们的访问权遵守下列规则：

1. 友元函数应在类中通过前置关键字 friend 说明，而其实际实现放在类外部。友元函数的实现与普通的 C++函数相同，也不是类的成员。
2. 关键字 friend 只在类内部的函数说明中使用，而在函数实现时不用。
3. 友元函数可访问类的私有成员。
4. 友元函数只能通过参数传递的对象访问类的成员，它用操作符'.'来访问数据成员。
5. 为将单目运算符以友元函数重载，只需将操作数作为一个参数传递；当双目运算符以友元函数重载时，将两个操作数都作为参数。

   例如：

```
friend Vec2d operator -(Vec2d X);              // 单目求反运算
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
friend Vec2d operator + (Vec2d X, Vec2d Y);      // 二目加法
```

6.2 有理数

有理数是分数 P/Q 的集合，其中 P、Q 均为整数且 Q < >0，P 称为分子，Q 称为分母。

2/3　-6/7　8/2　10/1　0/5　5/0(非法)

有理数的表示

有理数表示的是分子占分母的比例，因此可用许多等值的数来表示一个有理数，例如：

```
2/3 = 10/15 = 50/75            // 等值有理数
```

这些等值的数中有一个为最简分数，其分子和分母没有公因子。它是这些等值数的最有代表性的形式。例如，2/3 就是 2/3，10/15，50/75 等的最简分数。可用将分子和分母同时除以其最大公因子(GCD)来得到最简分数。例如：

```
10/15 = 2/3
  (GCD(10,15) = 5;  10/5 = 2  15/5 = 3)

24/21 = 8/7
  (GCD(24,21) = 3;  24/3 = 8  21/3 = 7)

5/9 = 5/9
  (GCD(5,9) = 1;    // 有理数已化为简单形式)
```

有理数中，分子和分母均可为负整数。我们用分母为正数的标准形式来存放这两个数。

```
2/-3 = -2/3            // -2/3 为标准形式
-2/-3 = 2/3            // 2/3 为标准形式
```

有理数的算术运算

读者已经熟悉了有理数的加，减，比较运算和乘，除的规则，我们给出几个例子来说明有理数类中实现这些运算的算法。

两个有理数的加(减)可求它们的等值的具有相同分母的有理数，然后，加(减)其分子。

$$(+)\ \frac{1}{3}+\frac{5}{8}=\frac{8*1}{3*8}+\frac{3*5}{3*8}=\frac{8*1+3*5}{3*8}=\frac{23}{24}$$

$$(-)\ \frac{1}{3}-\frac{5}{8}=\frac{1*8}{3*8}-\frac{3*5}{3*8}=\frac{1*8-3*5}{3*8}=\frac{-7}{24}$$

两个有理数相乘等于其分子和分母分别相乘，其除法为将被除数分子和分母互换后相乘

$$(*)\ \frac{1}{3}*\frac{5}{8}=\frac{1*5}{3*8}=\frac{5}{24}$$

$$(/)\ \frac{1}{3}/\frac{5}{8}=\frac{1}{3}*\frac{8}{5}=\frac{8}{15}$$

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

所有的关系运算符都用同一原则:求分母相同的等值数然后比较分子。

(<) $\frac{1}{3} < \frac{5}{8}$ 等同于关系表达式 $\frac{8*1}{3*8} < \frac{5*3}{8*3}$,该表达式为 TRUE,因为 $1*8<5*3$ $(8<15)$

例 6.1

本例说明有理数运算,假设 $U=4/5, V=-3/8, W=20/25$

1. $U+V=\frac{4}{5}+(-\frac{3}{8})=\frac{32}{40}+(-\frac{15}{40})=\frac{17}{40}$

   $U*V=\frac{4}{5}*(-\frac{3}{8})=\frac{-12}{40}$

   $U>V$ 因为 $\frac{32}{40}>-\frac{15}{40}$ $(32>-15)$

2. $U==W$ 因为 $\frac{4}{5}=\frac{100}{125}$, $\frac{20}{25}=\frac{100}{125}$

有理数的转换

有理数集合中包含整数集为其子集。因为整数可用分母 1 表示成有理数。

(整数)25 = 25/1(有理数)

复杂一些的转换为实数和有理数之间的转换。例如,实数 4.25 为 4 + 1/4,对应于有理数 17/4。将实数转换为有理数的算法在第 6.6 节中给出。反过来,将有理数转换为实数只需做一个简单的除法(分子除以分母)。例如:

3/4 = 0.75　　// 用 3.0 除以 4.0

6.3 有理数类

6.2 节提出的想法可以用有理数类来实现,该类以数据成员分子和分母来描述有理数,并将其算术和条件运算符作为重载的成员函数,而用友元函数来实现其输入和输出操作,并创造性地用构造函数和 C++ 的转换运算实现整数或实数与有理数之间的相互转换。

类 Rational 定义

声明

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

class Rational
{
    private:
        // 以分子/分母形式定义一个有理数
        long num,den;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 供算术运算符使用的构造函数,可处理大分子或分母
        Rational (long num,long denom);
        // 有理数例程
        void Standardize(void);
        long gcd(long m, long n) const;
    public:
        // 用于整数→有理数  实数→有理数转换的构造函数
        Rational (int num = 0, long denom = 1);
        Rational (double x);
        // 有理数的输入/输出
        friend istream& operator > >(istream& istr,Rational &x);
        friend ostream& operator > >(ostream& ostr,const Rational &x);
        // 双目运算符:加、减、乘、除
        Rational operator + (Rational v) const;
        Rational operator - (Rational v) const;
        Rational operator * (Rational v) const;
        Rational operator/ (Rational v) const;
        // 单目运算求反
        Rational operator - (void) const;
        // 关系运算符
        int operator < (Rational v) const;
        int operator < = (Rational v) const;
        int operator = = (Rational v) const;
        int operator! = (Rational v) const;
        int operator > (Rational v) const;
        int operator > = (Rational v) const;
        // 转换运算符:有理数→实数
        operator double(void) const;
        // 例程
        int GetNumerator(void) const;
        int GetDenominator(void) const;
        void Reduce(void);
}
```

说明

私有函数 Standardize 将有理数转换为分母为正整数的"标准形式"。构造函数用 Standardize 来保证生成的数为标准形式。在读入有理数或两数相除时,也要用该函数,因为这些操作可能产生负分母。由于加法和减法的两个分母均已是正数,它们就不用调用 Standardize。私有函数 gcd 返回两个整数 m 和 n 的最大公约数。

公有函数 Reduce 调用 gcd 将有理数对象化简成最简形式。

类的两个构造函数充当从整型(long int)和实型(double)到有理数的转换函数。它们将在下面有关类型转换运算的章节中介绍。

实际应用中,我们可用函数 GetNumerator 和 GetDenominator 来存取有理数的分子和分

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

母。输入运算符"<<"是以 P/Q 形式读入有理数的友元函数。有理数的输出形式也是 P/Q。若读入的有理数分母为零,则程序出错退出。

例

```
Rational A(-4,5),B, C;          // A 为 -4/5, B 和 C 均为 0/1
cin >> c;                       // 将输入 -12/-18 标准化后存储为 12/18
cout << C.GetNumerator();       // 输出分子 12
C.Reduce();                     // 将 C 简化为 2/3
B = C + A;                      // B 为 2/3 + (-4/5) = -2/15
cout << -B;                     // 调用求反方法,输出为 2/15
cout << float(A);               // 转换为实数
                                // 输出 -.8

cout << .5 + Rational(3)        // 将 0.5 转换为有理数 1/2
                                // 输出两数之和 1/2+3/1=7/2
```

6.4 作为成员函数的有理数运算

类 Rational 用成员函数重载的办法定义了有理数的算术和关系运算。每个双目运算符都以其右操作数作为参数。

假设加法表达式中,U,V 和 W 均为有理数类型的对象:

Rational　　u, v, w

$w = u + v$

重载后的加法运算符'+'是对象 u(左操作数)的成员,它以 v 为参数,其返回值为 Rational 类型,并赋给 w,实际上,$w = u + v$ 是通过 $w = u.+(v)$来求值的。

对于表达式:

v = -u

C++执行对象 u(单操作数)的求反运算"-"。并将返回的 Rational 对象赋给 v。实际上,$v = -u$ 是通过 $v = u.-(\ )$来求值的。

有理数运算的实现

这里我们以有理数的加法、除法、等于和求反运算来说明运算符重载。假设有理数 u,v 定义如下:

Rational　　u(a,b), v(c,d);

两个有理数相加("+"),先对操作数求一公共的分母,然后再用分子相加。

$$(+)u+v=\frac{a}{b}+\frac{c}{d}=\frac{a*d}{b*d}+\frac{b*c}{b*d}=\frac{a*d+b*c}{b*d}$$

实现时,a 和 b 分别是左操作数的 num 和 den,是运算符("+")所处的对象。c 和 d 是右操作数的数据成员 v.num 和 v.den。

```
// 有理数加法
Rational Rational::operator+(Rational v) const
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{
    return Rational(num * v.den + den * v.num, den * v.den);
}
```

除法("/")可通过将右操作数的分子与分母颠倒后相乘,a 和 b 对应于左操作数的数据成员,c 和 d 是对象 v(右操作数)的成员。由于结果的分母可能为负数,所以需对商进行标准化。

$$(/)u/v = \frac{a}{b}/\frac{c}{d} = \frac{a}{b} * \frac{d}{c} = \frac{a * d}{b * c}$$

```
// 有理数除法
Rational Rational::operator/ (Rational v) const
{
    Rational temp = Rational(num * v.den, den * v.num);
    // 保证分母为正数
    temp.Standardize();
    return temp;
}
```

为比较有理数的大小,只需将其转化为具有相同分母的分数后比较其分子。例如相等运算符"= =",在分子相等时返回 True。下面为等于的逻辑推导:

// 关系运算符"= ="

u = = v　　< = = >

$$u == v \Leftrightarrow \frac{a}{b} == \frac{c}{d}$$

$$\Leftrightarrow \frac{a * d}{b * d} == \frac{b * c}{b * d}$$

$$\Leftrightarrow a * d == b * c$$

只需检查条件 a * d = = b * c;

```
// 有理数"相等"
int Rational::operator = = (Rational v) const
{
    return num * v.den = = den * v.num;
}
```

单目运算符求反"-"只对定义该操作符的对象起作用,即对其分子求反。

```
// 有理数求反
Rational Rational::operator- (void) const
{
    return Rational(-num,den);
}
```

6.5 作为友元函数的有理数流运算符

文件 < iostream.h > 中包含有两个名为 ostream 和 istream 的类,分别提供流输出和流输入操作。它们定义的 I/O 流操作符"> >"和"< <"可对原始类型 char, int, short, long,

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

float,double 和 char * 进行输入输出。例如：

输入	输出
istream & operator > >(short v);	ostream & operator < <(short v)
istream & operator > >(double v);	ostream & operator < <(double v)

用户可重载流操作符来实现对用户自定义类型的 I/O。例如，对于类 Date，运算符"< <"和"> >"可用与输出原始类型相同的方式提供对 Date 类型的 I/O 操作，即：

```
Date  D;
cin > > D;          // < 输入 > 10/5/75
cout < < D;          // < 输出 > October 5,1975
```

如果对类 Date 的这些运算符要以成员函数形式重载，那么它们就应在 < iostream.h > 中显式地说明，类 ostream 必须有一个可接受 Date 类型参数的"< <"重载版本。这显然是不现实的，所以我们只能用友元函数的形式重载。这样，可在类的外部定义可访问类的数据成员的两个运算符。

我们用形式类型 CL 类给出重载的流操作的结构化形式：

```
class CL
{
        ...
    public:
        ...
        friend istream& operator > >(istream& istr,CL& Variable);
        friend ostream& operator > >(ostream& ostr,const CL& Value);
}
```

参数 istr 代表 cin 之类的输入流，ostr 代表 cout 之类的输出流。由于 I/O 操作要改变流的状态，参数必须以地址形式传递。

对于输入来说，由于要从流向数据元素赋值，故 Variable 以地址传递。函数返回一个指向 istream 的指针，使操作符可以下述链状形式使用：

```
cin > > m > > n;
```

在输出情况下，Value 被拷贝到输出流中。由于数据不变，Value 是以指针常量形式给出。这可避免可能发生拷贝一个大的对象这种操作。函数返回一个指向 ostream（输出）的指针，故该运算符可用下述形式使用：

```
count < < m < < n;
```

有理数流运算符的实现

有理数的输入和输出通过重载流运算符来实现。对于输入，我们以 P/Q(Q≠0)形式读入有理数。若分母为 0，则程序终止。

```
// 重载流输入运算符，按格式 P/Q 读入有理数
istream& operator > > (istream& istr, Ratioanl& x)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    char c;          // 读入分隔符'/'
    // 作为友元函数,">>"可访问 x 的分子和分母
    istr >> x.num >> c >> x.den;
    // 若分母为 0,则程序退出
    if (x.den == 0)
    {
        cerr << "A Zero denominaltor is invalid \ n";
        exit (1);
    }
    // 将 x 转化为标准形式
    x.Standardize();
    return istr;
}
```

重载后的流输出运算符以 P/Q 格式输出有理数。

```
// 重载流输出运算符,以 P/Q 格式输出有理数
ostream& operaotr << (ostream& ostr, const Rational& x)
{
    // 作为友元函数,"<<"可访问 x 的分子和分母
    ostr << x.num << '/' x.den;
    return ostr;
}
```

6.6 有理数的转换

类 Rational 给出了类型转换的运算符。程序员也可以类似方式实现自己定义的转换运算符。这里,我们把注意力集中到类 Rational 和 C++的相关数据类型之间的转换。

转换到有理数对象类型

类的构造函数可用来创建对象。此时,构造函数读入输入参数,然后将其转换为对象。Rational 类有两个构造函数,可用来作类型转换,其中第一个将整数转换为有理数,第二个将浮点数转换为有理数。

当用单独的一个整型参数 num 调用构造函数时,它将该整数转换为等值的有理数 num/1。例如:

```
Rational P(7), Q(3,5), R(0), S;     // 显式地将 7 转换为 7/1
R = Rational(2) ;                   // 显式地将 2 转换为 2/1
S = 5 ;                             // 创建 Rational(5),并将新对象赋值给 S
```

声明部分分别创建对象 Q = 3/5 和 R = 0/1,赋值语句 R = Rational(2),显式地将 R 的值改为 2/1。赋值语句 S = 5 导致类型间转换,编译器将 S = 5 当作 S = Rational 处理。

```
// 构造函数,形成有理数 num/den,当使用缺省值 den = 1 时,即将整数转换为有理数
Rational::Rational(long p,long q):num(p),den(q)
{
    if (den == 0)
    {
        cerr << "A Zero denominator is invalid" << endl;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        exit (1);
    }
}
```

第二个构造函数将实数转换成有理数。例如，下述语句创建有理数 A = 3/2 和 B = 16/5。

```
Rational A = Rational(1.5), B;     // 显式转换
B = 3.2;                           //将 3.2 转换为 Rational(16,5)
```

这要求有一个算法将实数转换为近似的有理数。该算法通过小数点移位来求近似值。得到一个有理数后，再调用化简函数 Reduce，将有理数化为最简形式。

```
// 构造函数。将浮点数 x 转换为近似有理数
Rational::Rational(double x)
{
    double val1,val2;
    // 将浮点数 x 的小数点右移 8 位后赋值给 val1
    val1 = 100000000L * x;
    // 将浮点数 x 的小数点右移 7 位后赋值给 val2
    val2 = 10000000L * x;
    // 此时，val1 - val2 = 90,000,000 * x。将 val1 - val2 值取整，得 x 的近似值
    num = long(val1 - val2);
    den = 90000000L;
    // 最简化
    Reduce()
}
```

从有理数类型转换

C++语言中定义了原始类型之间的显式转换。如，为打印字符 C 的 ASCII 码值，我们可用语句：

```
cout << int(0) << endl;          // 将 char 型转换为 int 型
```

类型也可进行隐式转换。例如，若 I 为长整型而 Y 为双精度型，语句

```
Y = I;          // Y = double(I)
```

将 I 转换为双精度型并将结果赋给 Y。一个类中可有一个或多个函数来将对象转换为其它数据类型的值。以类 CL 为例，假设我们希望将对象转换为名为 NewType 的类型，运算符 NewType 读入一个 CL 对象，并返回一个 NewType 类型的值。目的类型 NewType 通常是诸如 int 或 float 的标准类型。由于这是一个单目运算符，它不含参数。同样，转换运算也没有返回值类型，因为它暗含为类型 NewType。定义形式如下：

```
Class CL
{
    ...
    operator NewType(void);
};
```

转换运算可如下使用：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
NewType  a;
CL obj;
a = NewType(obj);        // 显式转换
a = obj;                 // 隐式转换
```

类 Rational 包含将有理数转换为双精度数的运算符。它可将 Rational 数据赋值给浮点型变量。其输入为有理数 p/q,用分子除以分母,将结果返回给浮点型变量。如:

3/4 为 0.75,　　4/2 为 2.0

```
// 转换有理数至双精度数
Rational::operator double(void) const
{
    return double(num)/den;
}
```

例 6.2

假设有如下定义:

```
Rational    R(1,2), S(3,5);
double      Y,Z;
```

1. 语句 Y = double(R)使用显式转换,结果为 Y = 0.5;
2. 语句 Z = S 使用隐式转换,结果为 Z = 0.6。

6.7 有理数的使用

在编写有理数的应用程序之前,我们先描述以下分式化简的算法。该算法需要找分子和分母的最大公因子(gcd)。

为得到有理数的化简式,Reduce 使用了私有成员函数 gcd,它带两个正整数参数并返回其最大公因子。函数 gcd 在补充程序中实现。对于非零的有理数,Reduce 将分子和分母都除以它们的 gcd:

```
// 将有理数最简化
void Rational::Reduce(void)
{
    int bigdivisor, tempnumerator;

    // tempnumerator 为分子的绝对值
    tempnumerator = (num < 0)? -num:num;

    if (num = 0)
        den = 1;        // 标准化为 0/1
    else
    {
        // 求两个正整数 tempnumerator 和 den 的最大公约数
        bigdivisor = gcd(tempnumerator, den);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 若 bigdivisor = = 1,有理数已为最简有理数。
        // 否则将其分子和分母同时除以最大公约数
        if (bigdivisor > 1)      // 最大公约数为 1 时,不需做除法
        {
            num /= bigdivisor;
            den /= bigdivisor;
        }
    }
}
```

程序 6.1　有理数类的使用

本程序演示 Rational 类的主要特性。我们还要示意几种不同类型数之间的转换方法,包括整数到有理数、有理数到实数之间的转换。我们还将演示有理数的加、减、乘、除。程序最后将浮点数隐式转换为有理数并再转换回浮点数。Rational 类的实现包含于文件"rational.h"中。

```
#include <iostream.h>

#include "rational.h"      // 引入有理数类

// 对每次运算后输出结果
void main(void)
{
    Ratioanl r1(5),r2,r3;
    float f;

    cout << "1. Rational value for integer 5 is " << r1 << endl;

    cout << "2. Enter a rational number: ";
    cin >> r1;
    f = float(r1);
    cout << "     Floating point equivalent is " << f << endl;

    cout << "3. Enter two rational number: ";
    cin >> r1 >> r2;
    cout << "     Results: " << r1 + r2 << "(+)"
         << r1 - r2 << "(-)" << r1 * r2 << "(*)"
         << r1/r2 << "(/)" << endl;

    if (r1 < r2)
        cout << "     Relation (less than): " << r1 << "<"
        else if (r1 = = r2)
            cout << "     Realton (equal to): " << r1 << "= ="
                 << r2 << endl;
          else
            cout << "     Relation (greater than): "<< r1 << ">"
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
                < < r2 < < endl;

        cout < < "4. Input a floating point number: ";
        cin > > f;
        r1 = f;
        cout < < "    Convert to Rational " < < r1 < < endl;
        f = r1;
        cout < < "    Reconvert to float " < < f < < endl;
}
/*
<程序 6.1 运行结果>
1. Rational value for integer 5 is 5/1
2. Enter a rational number: - 4/5
   Floating piont equivalent is - 0.8
3. Enter two rational number :1/2 - 2/3
   Results: - 1/6 ( + )  7/6 ( - )  - 2/6 ( * )  - 3.4 (/)
   Relation (greater than): 1/2 < - 2/3
4. Input a floating piont number:4.25
   Couvert to Rational 17/4
   Reconvert to float 4.25
*/
```

应用：有理数实用函数 读者们很早就从学校里学习过分数了。回想以下老师的训导："以带分数的形式给出答案。"为说明 Rational 类的应用，我们先讲述一系列进行分数计算的函数。

函数 PrintMixedNumber 将分数写为带分数的形式：

分数	带分数
10/4	$2\frac{1}{2}$
- 10/4	$-2\frac{1}{2}$
200/4	50

在代数中，一类基本问题是求解一般的分数方程

2/3X + 2 = 4/5

求解过程是：先将 2 移到方程的右边，以便隔离开含 X 的项：

2/3X = - 6/5

将方程两边同时除以 X 的系数 2/3 即可得到方程的解。我们用函数 SolveEquation 实现此过程。

X = - 6/5 * 3/2 = - 18/10 = - 9/5 (约简后)

程序 6.2 有理数实用程序

程序中用一系列有理数实用函数进行计算。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

PrintMixedNumber　　// 以带分数形式输出有理数

SolveEquation　　// 解一般方程 $\frac{a}{b}x + \frac{c}{d} = \frac{e}{f}$

输出语句直接描述了每个操作的动作。

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

#unclude "rational.h"          // 引入有理数类

// 以带分数形式(+/-)N p/q 输出有理数
void PrintMixedNumber (Rational x)
{
    // 有理数 x 的整数部分
    int wholepart = int(x.GetNumerator() / x.GetDenominator());

    // 存贮带分数的小数部分
    Rational fractionpart = x - Rational(wholepart);

    // 若没有小数部分,输出整数
    if (fractionpart == Rational(0))
        cout << wholepart << " ";
    else
    {
        // 最简化小数部分
        fractionpart.Reduce();
        // 输出带符号的整数部分
        if (wholepart < 0)
            fractionpart = -fractionpart;
        if (wholepart! =0)
            cout << wholepart << "" << fractionpart << " ";
        else
            cout << fractionpart << " ";
    }
}

// 解方程 ax+b=c,其中 a,b,c 为有理数
Rational SolceEquation(Rational a, Rational b, Rational c)
{
    // 检查 a 是否为 0
    if (a == Rational (0))
    {
        cout << "Equation has no solution." << endl;
        // 若无解,则返回有理数 0
        return Rational (0);
    }
    else
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        return (c - b)/a;
}

void main(void)
{
    Rational r1, r2, f3, ans;
    cout << "Enter coefficients for"
         << "'a/b X + c/d = e/f': ";
    cin >> r1 >> r2 >> r3;

    cout << "Simplified equation: " << r1 << "X = "
         << (r3 - r2) << ans << endl;
    ans = SolveEquation(r1,r2,r3);
    ans.Reduce();
    cout << "X = "<< ans << endl;

    cout << "Solution as a mixed number: ";
    PrintMixedNumber(ans);
    cout << endl;
}

/*
<程序 6.2 运行结果之一>
Enter coefficient for 'a/b X + c/d = e/f': 2/3 2/1 4/5
Simplified equation is: 2/3 X = -6/5
X = -9/5
Solution as a mixed number: -1 4/5
<程序 6.2 运行结果之二>
Enter coefficients for 'a/b X + c/d = e/f': 2/3 -7/8 -3/8
Simplified equation is: 2/3 X = 32/64
X = 3/4
Solution as a mixed number: 3/4
*/
```

书面作业

6.1 C++允许一个程序中有两个或两个以上的函数具有相同的名字，只要其变元表不尽相同以使编译器能分辨这些函数调用。编译器求出调用块中的参数值并选择正确的函数。这一过程被称为"函数重载(function overloading)"。例如，C++数学库中定义两个版本的 sqr 函数，它们都返回其变元的平方。

```
整数版本:     int sqr (long);       // 选择整数版
浮点数版本:   double sqr(double);   // 选择实数版
```

有效的函数重载由以下一系列规则组成：

规则 1：函数需要独立于返回类型和缺省值的可区分的参数。

规则 2：枚举类型是用于重载目的的可以区分的类型。关键字 typedef 不影响重载。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

规则 3：如果一个参数不精确地匹配一系列重载函数中的正规参数，则用匹配算法确定“最佳匹配”函数。如有需要，则进行转换。例如，当传递一个短整数给一个参数类型为整数的重载函数时，编译器将变量转换为 int 型以建立匹配。如果某种选择导致二义性则编译器拒绝转换参数。

说明以下各例中可以应用的规则。如果规则被违反，指出错误所在。

(a) 假设以下代码段被用来重载函数 f。

```
    <函数 1>                    <函数 2>
int f(int x, int y)         double f(int x, int y)
{                           {
    return x * y;           return x * y;
}                           }

<函数 3>
int f(int x = 1, int y = 7)
{
    return x + y + x * y;
}
```

(b) 函数 max 用 4 个不同的定义进行重载。

```
    <函数 1>                    <函数 2>
int max(int x, int y)       double max(double x, double y)
{                           {
    return x > y ? x : y;       return x > y ? x : y;
}                           }

    <函数 3>                    <函数 4>
int max(int x, int y, int z)  int max(void)
{                           {
    int lmax = x;               int a, b
    if (y > lamx)               cin > > a > > b;
        lmax = y;               return abs(a) > abs(b)
                                a : b;
if (z > lmax)               }
        lmax = z;
    return lmax;
}
```

(c) 为区分输入的整数类型和枚举类型而编写了 3 个版本的函数。

```
    <函数 1>                    <函数 2>
void read(int& x)           void read(Boolean& x)
{                           {
    cin > > x;                  char c;
    }                           cin > > c;
                                x = (c = = 'T') ? TRUE : FALSE;
                            }

    <函数 3>
typedef int Logical;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
const int TRUE = 1,FALSE = 0;
void read(Logical& x)
{
    char c;
    cin > > c;
    x = (c = = 'T') ? TRUE : FALSE;
}
```

6.2 本题使用书面作业 6.1(b)中的函数,假设用户输入值为 m = - 29,n = 8。说明哪个函数被调用,返回值是什么。

(a) cin > > m > > n;　　(b) max();

(c) max(m, -40, 30);　　(d) max(m,n);

6.3 对于书面作业 6.1(b)中的 max 函数,以下各语句的输出是什么?

```
int a = 5, b = 99, c = 153
int m, n;
double h1 = .01, h2 = .05;
long t = 3000, u = 70000, v = -100000;

cout < < "Maximum of a and b is " < < max(a,b);
cout < < "Maximum of a, b and c is " < < max(a,b,c);
cout < < "1.0 + max(h1,h2) = " < < 1.0 + max(h1,h2);
cout < < "Maximum of t, u and v is " < < max(t,u,v);
```

6.4 以下函数在重载时是否可以区分? 为什么?

```
(a) enum E1{one, two};        (b) type def double scientific;
    enum E2{three, four};

    int f(int x, E1 y);           double f(double x);
    int f(int x, E2 y);           scientific f(scientific x);
```

6.5 编写函数 Swap 的重载版本,使其可以带两个 int, float, 和 string(char *)型的参数。

```
void Swap(int& a, int& b);
void Swap(float& x, float& y);
void Swap(char *s, char *t);
```

6.6 解释用成员函数进行运算符重载和使用友元函数的区别。

6.7 类 ModClass 中仅含一个数据成员 dataval,其范围在 0 到 6 之间。构造函数以任意正整数 v 为参数,将其除以 7 的余数赋值给 dataval。例如:

```
    ModClass a(10);      // a 中的 datavl 值为 3;
    ModClass b(6);       // b 中 bataval 值为 6;
    ModClass c;          // c = a + b 的值为(3 + 6) % 7 = 2

class ModClass
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    private:
        int dataval;
    public:
        ModClass(int v = 0);
        ModClass operator + (const ModClass& x);
        int GetValue(void) const;
};
```

(a) 实现上述类的方法。

(b) 将运算符"*"声明为 ModClass 的友元函数,并实现之。该运算符将两个 ModClass 对象中的数值域相乘,并求出除以 7 后的余数。

(c) 编写函数

```
ModClass Inverse(ModValue& x);
```

以具有非零值的对象 x 为参数,返回值 y,使得 x * y = 1(y 被称为 x 的倒数)。(提示:用数据值为 1 到 6 之间的对象重复乘以 x。其中一个对象为倒数。)

(d) 重载 ModClass 的流输出,并将方法加到类中。

(e) 重载转换运算符 int () 以替换 GetValue。该运算符返回 dataval 值,从而将 ModClass 对象转换为整数:

```
operator int (void);
```

(f) 编写函数

```
void Solve(ModClass a, ModClass& x, ModClass b);
```

通过调用方法 Inverse,解关于 x 的方程 ax = b。

6.8 将整套关系运算符加到第 3 章的 Date 类中。按年份的时间顺序对两个日期进行比较。例如:

```
Date(5, 5, 77) > Date(10, 24, 73)
Date("12/25/44") <= Date(9, 30, 82)
Date(3, 5, 99) != Date(3, 7, 99)
```

6.9 复数形如 $x + iy$,其中 $i^2 = -1$。复数在数学、物理和工程领域都有广泛的应用。复数的算术运算规则如下:

令 $u = a + ib$, $v = c + id$

u 的绝对值 Magnitude(u) = sqrt($a^2 + b^2$)

实数 f 所对应的复数为 $f + i0$

u 的实部 = a

u 的虚部 = b

$u + v = (a + c) + i(b + d)$

$u - v = (a - c) + i(b - d)$

$u * v = (ac - bd) + i(ad + bc)$

$$u / v = \frac{ac + bd}{c^2 + d^2} + i\left(\frac{bc - ad}{c^2 + d^2}\right)$$

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

$$-u = -a + i(-b)$$

实现类 Complex，其声明如下：

```
class Complex
    {
        private:
            double real;
            double imag;
        public:
            Complex(double x = 0.0, double y = 0.0);

            // 二目运算符
            Complex operator + (Complex x) const;
            Complex operator - (Complex x) const;
            Complex operator * (Complex x) const;
            Complex operator/ (Complex x) const;
            // 求反
            Complex operator - (void) const;
            // 流输入/输出操作
            // 输出格式为(实数,虚数)
            friend ostream& operator < < (ostream& ostr, const
                                          Complex& x);
    };
```

6.10 在书面作业 6.9 的类 Complex 中增加方法 GetReal 和 GetImage，它们分别返回复数的实部和虚部。用这些方法编写函数 Distance，计算两个复数之间的距离。

```
double Distance (const Complex &a, const Complex &b);
```

6.11 (a) 在 Rational 类中增加一个返回 ModClass 对象的转换程序。

(b) 在 ModClass 类中增加一个返回 Rational 对象的转换程序。

6.12 (a) 实现 6.1 节中的类 Vec2d。

(b) 在 Vec2d 类中增加一个提供标量乘法的成员函数。其中，标量操作数在右侧。用成员函数和友元函数两种方式实现此运算符。

```
Vec2d v(3, 5);
cout < < v*2 < < "" < < 2*v < < endl;
<输出>
(6, 10) (6, 10)
```

6.13 集合(set)由从被称为全集(universal set)的一组对象中选出的若干对象组成。我们将集合写成表的形式，并以逗号为间隔，用大括号将两侧括起来。

$$X = \{I_1, I_2, I_3, \cdots, I_m\}$$

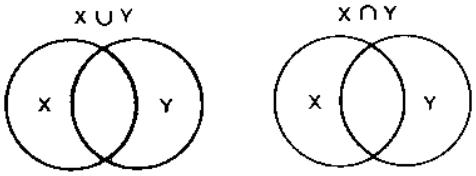
在本题中，集合成员从 0 到 499 范围内的整数中选出。类支持集合 X 和 Y 之间的一系列二元运算。

· 并集(∪) X∪Y 是不重复地包含 X 中所有元素和 Y 中所有元素的集合。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

· 交集(∩)　X∩Y 是所有既在 X 中又在 Y 中的元素的集合。



X = {0,3,20,55}, Y = {4,20,45,55}
X ∪ Y = {0,3,4,20,45,55}　　X ∩ Y = {20,55}

· 属于集合(∈)　如果元素 n 是集合 X 的成员,则 n∈X 为真;否则为假。

X = {0, 3, 20, 55}　　// 20∈X 为真,35∈X 为假

Set　类定义

在以下声明中,集合是从 0…SETSIZE - 1 范围内的整数中选出的元素表,其中 SETSIZE = 500。集合的操作有输入/输出、并集、交集、属于运算等。

声明

```
const int SETSIZE = 500;
const int False = 0, True = 1;

class Set
{
    private:
        // 集合的元素
        int member[SETSIZE];

public:
        // 构造函数,创建一个空集
        Set(void);
        // 构造函数,创建一个包含初始元素 a[0],…,a[n-1]的集合
        Set(int a[],int n);

        // 并集运算符(+),交集运算符(*),属于运算符(^)
        Set operator + (Set x) const;              // 并集
        Set operator * (Set x) const;              // 交集
        friend int operator^(int elt, Set x);      // elt 属于 x 吗?
        // 插入及删除元素
        void Insert(int n)     // 往集合中加入元素 n
        void Delete(int n);    // 从集合中删除元素 n
        // 集合的流输出运算
        friend ostream& operator << (ostream& istr, const Set& x);
};
```

说明

第 1 个构造函数只是建立一个空表。用 Insert 将元素加入到集合中。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

如果元素 n 对应的数组元素值为真(true),则 n 在集合中。

```
n∈x 当且仅当 member[n]为真
```

例如,集合 X = {1, 4, 5, 7}对应的数组中,其元素 member[1], member[4], member[5], member[7]为真,而其余项为假。

false	true	false	false	true	true	false	true	false	false	…	false
0	1	2	3	4	5	6	7	8	9		499

流输出运算符打印出集合元素,元素之间用逗号隔开,且都括在大括号中。例如:

```
int setvals[] = {4,7,12};
Set S, T(setvals,3);

s.Insert(5);
cout << S << endl;
cout << T << endl;

<输出>
{5}
{4,7,12}
```

(a) 实现 Set 的构造函数。

(b) 实现集合的成员函数 in('^')。若 member[n]为真它也返回真,否则返回假。因为'^'运算符的优先级相对较低,为安全起见,包含'^'的表达式应用括弧括起来。例如:

```
Set A;
...
if ((0 ^ A) == 0)      // 检查 0 是否在集合 A 中
```

(c) S 等于集合{1, 4, 17, 25},T 等于集合{1, 8, 25, 33, 53, 63}。试进行以下运算:

(i) S + T　　(ii) S * T　　(iii) 5 ^ S

(iv) 4 ^ (S + T)　　(V) 25 ^ (S * T)

(d) 说明以下语句序列的动作:

```
int a[] = {1,2,3,5};
int b[] = {2,3,7,9,25};
int n;
Set A(a,4), B(b,5), C;

C = A + B; cout << C;
C = A * B; cout << C;

cin >> n;    // 输入 55
A.Insert(n);
if (n^A)
cout << "Success" << endl;
```

(e) 实现其余的 Set 方法。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

上机题

6.1 本题用到书面作业 6.5 中的结果。写一个程序，输入两个整数和两个浮点数，打印出它们的值，然后调用相应的 Swap 函数将它们的值互相交换。该程序还输入两个字符串，用 Swap 交换其值，并打印出结果串。

6.2 本题测试书面作业 6.7 中设计的类 ModClass。编写程序验证 ModClass 对象的分配律。

```
a * (b + c) = a*b + a*c
```

定义构造函数参数分别为 10，20 和 30 的 3 个对象 a，b，和 c。程序应该输出等式两边表达式的值。

6.3 用 ModClass 和书面作业 6.7 中设计的函数 Solve 构造数组

```
ModClass a[] = {ModClass(4), ModClass(10), ModClass(50)};
```

程序应该执行循环

```
for (int i = 0; i < 3; i ++)
   cout << a[i] << " " << int(a[i]) << endl
```

用函数 Solve 打印以下方程的解：

```
ModClass(4) * x = ModClass(3)
```

6.4 本题使用书面作业 6.8 中的 Date 类运算符。编写函数

```
Date Min(const Date& x, const Date& y);
```

返回两个日期中较早的一个。定义满足以下要求的对象 D1 到 D4：

D1 是 6/6/44　　D2 是 1999 年元旦
D3 是 1976 年圣诞节　　D4 是 1976 年 7 月 4 日

比较对象 D1 和 D2、D3 和 D4，对以上函数进行测试。

6.5 考虑二维向量的以下特性：

(a) u * (v + w) = u * v + u * w (分配律)
(b) 如果两个向量的点积为零，则两个向量互相垂直。
(c) c * v = v * c，其中 c 是实数。

用书面作业 6.12 中设计的类 Vec2d，编写程序，用例子直接说明(a)到(c)。程序中还应该读入两个实数 x 和 y，验证向量(x, y)和(y, −x)是垂直的。

6.6 本题使用书面作业 6.9 中设计的复数类 Complex。程序必须完成以下动作：

(a) 验证 $i^2 = -1$。
(b) 编写函数 f(z)，求复数多项式函数的值：

$$z^3 - 3z^2 + 4z - 2$$

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

对下列 z 值,求多项式的值:

```
z = 2+3i, -1+i, 1+i, 1-i, 1+0i
```

注意最后 3 个值是 f 的根。

6.7 用有理数类 Rational 及其转换运算符对实数和有理数进行下述比较。声明实数 pi = 3.14159265 以及年轻学生经常使用的有理数近似值 Rational(22, 7)。编写程序完成下面的两次计算并打印出结果。

(a) 计算将这两个数作为有理数时它们的差。Rational(pi) - Rational(22, 7)

(b) 计算将这两个数作为实数时它们的差。pi - float(Rational(22, 7))

6.8 第 8 章利用指针和动态内存设计了广义串操作类。本题设计一个用数组存储数据的简单串类。现有以下声明

```
class String
{
    private:
        char str[256];
    public:
        String(char s[] = "");
        int Length(void) const;            // 串长度
        void CString(char s[]) const;      // 将串拷贝到 C++的数组 s 中
        // * * * * * 流输入/输出 * * * * *
        friend ostream& operator<<(ostream& ostr,
                                   const String& s);
        // 读入空格隔开的串
        friend istream& operator>>(istream& istr,
                                   String& s);
        // * * * * * 关系运算符串==串 * * * * *
        int operator==(String& s) const;
        // * * * * * 串拼接 * * * * *
        String operator+(String& s) const;
};
```

实现此类,并运行以下程序,对类进行测试。

```
void main(void)
{
    String S1("It is a"), S2("beautiful day!"), S3;
        char s[30];
    if (S1 == String("It is a"))
       cout << "Equality test O.K." << endl;
    else
       cout << "Equality test failed." << endl;
    cout << "The length of S1 = " << S1.Length() << endl;
    cout << "Enter a string S3: ";
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        cin > > S3;
        S3 = S1 + S2;
        cout < < "The concatenation of S1 and S2 is "
             < < S3 < < endl;
        S3.CString(s);
        cout < < "The C + + string stored in S3 is '"
             < < s < < "'" < < endl;
    }
```

6.9 本题测试书面作业 6.13 中的 Set 类。考虑以下集合

S = {1, 5, 7, 12, 24, 36, 45, 103, 355, 499}

T = {2, 3, 5, 7, 8, 9, 12, 15, 36, 45, 56, 103, 255, 355, 498}

U = {1, 2, 3,4, 5, ..., 50}

创建集合 S,T,和 U。用 Insert 初始化集合 U。完成以下计算。

(1) 计算并打印 S + T。

(2) 计算并打印 S * T。

(3) 计算并打印 S * U。

(4) 从 T 中删除 8,36,103 和 498。

(5) 生成 1 到 9 之间的一个随机数,将其打印出来,并验证它是否在集合 T 中。

6.10 本题用 Set 类模拟在 0 到 4 范围内连续随机抽取 5 次得到 5 个不同数的概率。数学概率为:

1·4/5·3/5·2/5·1/5 = .0384

编写函数

int fillSet(void);

完成抽取 5 个数并将它们插入到集合 S 中的实验。循环 5 次并用"^"运算符进行测试,看 0~4 是否在集合中。如果所有整数都 在集合中,则返回 1;否则返回 0。

写一个主程序(main),调用 fillSet 100 000 次并记录所有 5 个整数都被选中的实例个数,最后再将这一数据除以 100 000 以确定模拟概率。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第7章　形式数据类型

7.1　模板函数
7.2　模板类
7.3　表的模板类
7.4　中缀表达式求值
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类 Seqlist、Stack 和 Queue 都是为形式数据类型 DataType 设计的，在使用这些类之前，用户必须用 typedef 将 DataType 和指定的类等同起来，这样，就将用户限定在只能对一个类使用一种数据类型上，在同一程序中不能同时使用整数栈和记录栈，为打破使用形式类型 DataType 的局限性，应将数据类型和对象而不是程序绑在一起，例如，

```
SeqList < int >        A;    // 整数表
Stack < float >        B;    // 实数栈
Queue < CL >           C;    // CL 对象的队列
```

C++用模板定向提供了这种能力，允许为函数和类设置通用类型的参数。对群体类使用模板可以让我们定义存放不同数据类型的对象，对函数使用模板可以让我们定义函数的形式参数并在运行时以不同类型的参数对之进行两次或更多次的调用，模板给数据结构提供了强大的概括功能。本章我们先介绍模板函数，然后将概念扩展到模板类，介绍中，以顺序查找为模板函数的例子，并用模板类重写了类 Stack，在最后的中缀表达式求值的研究中，我们使用多栈作为主要的数据结构。

7.1 模板函数

很多情况下，我们设计的算法可以处理多种数据类型，例如，顺序查找的算法读入 key 值，然后在元素表中寻找匹配，它只要求所用的数据类型定义关系运算符"==",同样的算法可用来查找整数表、实数表或对象表。可是，读者可能已经发现，到目前为止，本书已经有了很多顺序查找的应用，而对每种应用，我们却针对表中的元素类型给出了函数 SeqSearch 的一个特定版本，即对同一基础算法给出了多个实现版本，我们当然希望用到能运用不同类型表的通用代码，C++用模板提供了此项功能，下面讨论模板函数的语法。

模板函数的声明以下面模板参数表的形式开始：

```
template < class T1,class T2,…,class Tn >
```

关键字 template 后面紧跟着在尖括号内的非空的类型参数表，每个类型前面是关键字 class，标识符 T_i 是一特定的 C++数据类型，在调用模板函数时作为参数传入 class 只用来标识 T_i 代表一种类型，读者完全可以把它当作"类型"看等，调用模板函数时 T_i 可以是标准类型如 int，或是用户定义类型，比如类。下述模板参数表中，T 和 U 均为数据类型：

```
template < class T>            // T 为类型
template < calss T, class U>   // T 和 U 均为类型
```

定义完模板参数表后，函数体与通常函数相同，并可访问参数表中所有类型的表，例如，用模板函数定义的 SeqSearch 为：

```
// 查询 n 元数组表中与 key 等值的元素并返回其下标;若找不到,则返回 - 1
template < class T>
int SeqSearch(T list[], int n ,T key)
{
    for (int i = 0, i < n; i + + )
        if (list[i] = = key)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        return i;      // 返回元素下标值
    return -1;         // 设找不到,返回 -1
}
```

程序调用模板函数时,编译器取得实际参数的数据类型,并将这些类型与模板参数表中的各项相关联,例如,在调用 SeqSearch 函数时,编译器得到整型和实型参数:

```
int A[10],Aindex;
float M[100],fkey=4.5,Mindex;
Aindex=SeqSearch(A,10,25);        //在 A 中找整数 25
Mindex=SeqSearch(M,100,fkey);     //在 M 中找 fkey 4.5
```

编译器对每种不同的运行参数表创建独立的实例。在第一种情况下,模板类型 T 为 int,则 SeqSearch 用整数比较运算符"==”扫描整个表。在第二种情况下,类型参数 T 为 float,则用浮点型的比较运算符"==”。

当用指定类型调用基于模板的函数时,该类型必须定义了函数中用到的所有运算符。若函数中用到了不是该类型固有的运算符时,程序员必须自己定义该运算或使用函数的非模板版本。例如,C++没有定义结构或类的比较运算符"==”,除非用户自己定义该运算符(重载),他无法使用 SeqSearch 函数的通用版本比较类的对象。

举例来说,记录类型 Student 包括整型和浮点型域,我们重载"==”运算符使它能用于 SeqSearch 函数。

```
// student 记录中包含有学生的学号(studentID)及等级平均分(GPA)
struct Student
{
    int       studID;
    float     gpa;
};
// 用比较学号来重载"==”
int operator==(Student a, Student b)
{
    return a.studID==b.studID;
}
```

Student 记录的声明和运算符"==”的重载存放于文件"student.h"中。

C++的串类型 char* 给用户提出了一实现难题,"==”比较的是指针值,并不是一个字符一个字符地比较。实际上的串,由于 char* 不是结构,也不是类,我们也无法给定重载一个"==”运算符。因此对 C++的串类型,用户必须使用非模板函数的 SeqSearch 版本。

```
// 搜寻字符串数组中与串 key 匹配的串
int SeqSearch(char *list[], int n, char *key)
{
    for (int i=0;i<n;i++)
        // 用 C++串的库函数来比较两串
        if (strcmp(list[i], key)==0)
            return i;      // 找到时返回下标值
    return -1;             // 找不到,则返回 -1
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

本节主要给出了一个通用的查询程序，模板函数 SeqSearch 的编码及其用于 C++ 串类型的版都在文件“utils.h”中。

程序 7.1　通用查询

本程序给出了对三种不同类型数据的顺序查找。

- 用 10 个整数值初始化“list”，在数组中查找元素 8 的下标。
- 对 student 记录类型进行运算符重载，将记录{1555,0}作为 key 来查找表中具有该符号的具体情况，得到其 GPA 值为 2.6。
- 对串类型的数组，编译器用数据类型为“char *”的模板 SeqSearch 函数查找串“TWO”，返回值为下标 2。

```
#include <iostream.h>
// 引入基于顺序查找的模板及针对 C++ 串的顺序查找函数
#include "utils.h"
// 定义 Student 结构及为 Student 结构重载"= ="
#include "student.h"
void main()
{
    // 三种不同的数组类型
    int       list[10] = {5, 9, 1, 3, 4, 8, 2, 0, 7, 6};
    Student   studlist[3] = {{1000, 3.4}, {1555, 2.6},
                             {1625, 3.8}}
    char      *strlist[5] = {"zero","one","two","three",
                             "four"};

    int       i, index;
    // 下面记录是用来查找 Studentlist 数组的 key 值
Student studentKey = {1555,0};

    if ((i = SeqSearch(list,10,8)) >= 0)
        cout << "Item 8 is found at index " << i << endl;
    else
        cout << "Item 8 is not found" << endl;
    index = SeqSearch(studlist, 3, studentKey);
    cout << "Student 1555 has gpa " << studlist[index].gpa
         << endl;

    cout << "String 'two' is at index"
         << SeqSearch(strlist, 5, "two") << endl;
}
/*
<程序 7.1 运行结果>
Item 8 is found at index 5
Student 1555 has gpa 2.6
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
String 'two' is at index 2
*/
```

基于模板的排序

交换排序提供了用比较运算符“<”来对表中元素排序的算法，一个具有简单的模板参数 T 的函数 ExchangeSort 可用来实现该算法。相应于 T 的数据类型必须定义运算符“<”，或由用户提供一重载的运算符。

```
// 用交换排序算法对 n 个元素排序
template <class T>
void ExchangeSort(T a[], int n)
{
    T temp;
    int i, j;

// 循环 n-1 遍
    for (i=0; i<n-1; i++)
        // 将 a[i+1],...,a[n-1]中的最小值置于 a[i]中
        for (j=i+1; j<n; j++)
            if (a[j]<a[i])
            {
                // 交换 a[i]和 a[j]的值
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
```

为方便起见，函数 ExchangeSort 也存放在文件“utils.h”中。

7.2 模板类

本节我们以基于模板的类 Store 为例，给出模板类的基本概念。

模板类的定义

为定义模板类，应在类的声明之前加上一个模板参数表，里面的形式类型名用来说明数据元素和成员函数的类型，下面是模板类 Store 的声明：

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

template <class T>
class Store
{
private:
  T item;              // item 用来存放数据的值
  int haveValue;       // item 是否已初始化的标记

public:
  // (缺省的)构造函数
  Store(void);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 取得及有效数据的函数
    T GetElement(void);
    void PutElement(const T& x);
};
```

声明模板类对象

定义模板类对象时应给模板类传递类型，这样，就给出了模板类的一个实例。例如，下述声明建立了 Store 类型的对象：

```
// x 中的数据成员的类型为 int
Store < int >     X;
// 创建一个数据类型为 char 的有 10 个 Store 对象的数组
Store < char > S[10];
```

定义模板类的方法

模板类函数可用类内部的代码实现，也可在类体之外实现，对于类体之外的实现，该函数必须作为模板函数，其函数定义中应包括模板参数表。所有对类的引用都必须包含置于尖括号之内的模板类型，即：

```
ClassName < T > ::
```

例如，下述代码定义了类 Store 的函数 GetItem。

```
// 取已初始化的 item
template < class T >
T Store < T > ::GetElement(void)
{
    // 若数据未初始化，则退出程序
    if (haveValue = = 0)
    {
        cerr < < "No item present!" < < endl;
        exit(1);

    }
    return item;          // 返回 item 值
}
```

函数 PutItem 假定赋值运算符“ = ”是 T 类型元素的合法运算符：

```
// 往 store 中加入元素。
template < class T >
void Store < T > ::PutElement(const T & x)
{
    haveValue + + ;       // hareValue 置为 TRUE
    item = x;             // 存入 x
}
```

外部定义的构造函数用类名和类范围运算符作为其名字，由于是类型类，应使用模板参数，例如，类类型为 Store < T > ，而构造函数的名字只是简单的 Store。

```
// 定义的数据未初始化
template < Class T >
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Store<T>::Store(void):haveValue(0)
{}
```

Store 的声明和实现在文件“store.h”中。

程序 7.2　模板类 Store 的作用

程序对整型对象、Student 记录类型对象和 double 类型对象使用模板类 Store，前两种情况，用 PutElement 对数据赋值，然后用 GetElement 打印输出，数据类型为 double 时，由于试图去检索未经过初始化的数据值，程序中止运行：

```
#include <iostream.h>
#include "store.h"
#include "student.h"
void main(void)
{
    Student graduate={1000, 3.5}
    Store<int>A, B;
    Store<Student>   S;
    Store<double>    D;
    A.PutElement(3);
    B.PutElement(-7);
    cout << A.GetElement() << " " << B.GetElement() << endl;
    S.PutElement(graduate);
    cout << "The student id is " << S.GetElement().studID
         << endl;
    // D 尚未初始化
    cout << "Retrieving object D" << D.GetElement() << endl;
}
/*
<程序 7.2 运行结果>
3  -7
The student id is 1000
Retrieving object D No item present!
*/
```

7.3　表的模板类

本书中我们用模板类增加了集合的灵活性，本节我们用模板重新定义类 Stack，并在 7.4 节中用它来对中缀表达式求值，它的定义在文件“tstack.h”中。

类 Stack 的重定义直接使用模板机制，在类声明前加上模板参数表并用 T 代替 DataType。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

模板类 Stack 的定义

声明

```
#include <iostream.h>
#include <stdlib.h>
const int MaxStackSize = 50;
template <class T>
class Stack
{
    private:
      // 私有数据成员：堆栈数组及栈顶指针
      T stacklist[MaxStackSize];
      int top;
public:
    // 构造函数
    Stack(void);        // 初始化栈顶指针 top
    // 改变栈的函数
    void Push(const T& item);
    T Pop(void);
    void ClearStack(void);
    // 访问栈顶元素
    T Peek(void) const;
    // 检测栈的方法
    int StackEmpty(void) const;
    int StackFull(void) const;
};
```

基于模板的 stack 的实现

实现中，我们将类的函数都定义成外部模板函数，这需在每个函数前放置模板参数表，并用 stack(t)替换类的类型 Stack，在函数体内，也必须用模板类型 T 替换形式类型 DataType，下述程序例为 Push 和 Pop 函数的新的定义。

Push

```
// 将 item 压入栈中
template <class
void Stack<T>::Push(const T& item)
{
    // 若栈已满，则程序退出
    if (top = = MaxStacksize - 1)
    {
        cerr << "Stack overflow!" << endl;
        exit(1);
    }
    // 顶指针加 1 并将 item 拷入 stacklist 中
    top++;
    stacklist[top] = item;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    }
Pop
    // 将顶元素弹出堆栈并返回其值
    template <class T>
    T Stack<T>::Pop(void)
    {
        T temp;
    // 若栈为空,则退出程序
    if (top == -1)
    {
        cerr << "Attempt to pop an empty stack!" << endl;
        exit(1);
    }
    // 记录下顶元素值
    temp = stacklist[top];
    // 将 top 指针值减 1,并返回前顶元素值
      top--;
      return temp;
    }
```

7.4 中缀表达式求值

第 5 章给出了栈在后缀表达式(逆波兰表达式 RPN)求值中的应用,由于中缀表达式求值要用到两个栈,一个用来存放操作数,一个用来存放运算符,两个栈用到的数据类型不同,所以在第 5 章中,我们将它跳过去了。在介绍完模板类后,本节我们可以用模板类 Stack 来设计和实现中缀表达式求值算法。

读者已经熟悉了格式算术运算的表达式。如下列都是由单目运算符'-',双目运算符'+','-','*','/',括号和浮点数组成的表达式:

8.5+2*3　　-7*(4/3-6.25)+9

这些表达式用的是将双目运算符放在两个操作数之间的中缀形式,括号内是单独计算的子表达式。在高级语言中,运算符之间有优先顺序及结合性,具有最高优先级的运算符最先执行,如果运算符优先级相同,则在左结合时(+,-,*,/)先执行最左边的运算,右结合(单目'+',单目'-')时,先执行最右边的运算符:

优先级顺序(低到高)	运算符
1	+,-
2	*,/
3	单目'+','-'

例 7.1

```
1. 8.5+2*3=14.5          // 先做乘法运算,再做加法运算
2. (8.5+2)*3=31.5        // 括号内为子表达式
3. 9--6=15               // 单目'-'的优先级最高
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

表达式等级 中缀表达求值算法使用等级概念，给表达式中每项元素赋一个等级值 -1,0 或 1。

浮点操作数等级为 1

单目运算符'-','+'的等级为 0

双目运算符'+','-','*','/'的等级为 -1

左括号的等级为 0。

我们读入表达式的元素时，可用等级来识别由于操作数或运算符位置不对而可能导致的非法表达式。识别方法为对表达式的每个元素，我们都求一个从表达式第一个符号到当前元素的等级和。我们称它为累计等级，它可用来保证每个双目运算符两边都各有一个操作数，且中缀运算之后不会紧跟一个运算符，例如，一个简单表达式

2+3

的每个累计等级为

读入 2：累计等级 = 1

读入 +：累计等级 = 1 + -1 = 0

读入 3：累计等级 0 + 1 = 1。

规则 对表达式中的每个元素，表达式的累计等级必须为 0 或 1，整个表达式的累计等级为 1。

例 7.2

下面为可用等级法识别的非法表达式：

表达式	非法等级	原因
1. 2.5 4 +3	读入 4 时等级 = 2	两个连续的操作数
2. 2.5 + * +3	读入 * 时等级 = -1	两个连续的运算符
3. 2.5 +3 -	最后等级 = 0	缺操作数

中缀表达式求值算法 中缀表达式求值算法用操作数栈(浮点数栈)来存放操作数和中间运算结果，另一个运算符栈存放运算符和可使我们实现优先级的左括号，读入表达式时，将表达式元素压入各自的栈中。即遇到操作数时，将其压入操作数栈，而当运算需要时，将其从栈中弹出，而仅当当前栈中优先级大于或等于当前的运算符已经计算完后，才将运算符压入栈中，当执行完后才将运算符从栈中弹出，即当后续输入的运算符优先级小于或等于它的优先级或表达式结束时弹出。以下面表达式为例：

2+4-3*6

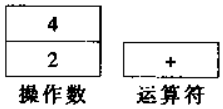
读入 2：将 2 压入操作数栈

读入 +：将 + 压入运算符栈

读入 4：将 4 压入操作数栈

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

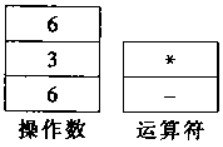


读入 - ：运算符'-'和栈中的'+'运算符优先级相同。先弹出'+'，并弹出两个操作数执行加法运算，将 2+4 的结果 6 压回操作数栈。将'-'压入运算符栈。

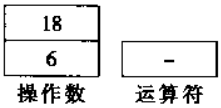
读入 3：将 3 压入操作数栈

读入 * ：运算符'*'的优先级高于栈中的运算符'-'，将'*'压入栈。

读入 6：将 6 压入操作数栈



结束：清空运算符栈。将'*'从中弹出，然后从操作数栈中弹出 6 和 3 执行乘法运算，将结果 18 压入操作数栈。



对从操作数栈中弹出的操作数 6 和 18 执行减法运算 6 - 18 = - 12 即为表达式的结果。

运算符优先级分两次给定，第一次为读入时，第二次为压入栈后，最初的优先级，我们称为**读入优先级**，用来和栈中的运算符相比较。运算符压入栈后，就被给定一个新的优先级，即**栈优先级**，读入优先级和栈优先级之间的差由括号和右结合运算符使用，这时，栈优先级比读入优先级小。出现相同优先级和右结合运算符时，栈优先级会超出在栈顶的运算符的栈优先级，前面的运算符将不被弹出，而是将新运算符压入栈中，这样，表达式的求值顺序为从右至左。

表 7.1 给出了用于中缀表达式求值的运算符的读入和栈优先级及等级。这些运算符包括括号和双目运算符 +，-，*，/。双目运算符为左结合的，具有相同的读入优先级和栈优先级，我们把右结合的乘方运算符留做练习。

有括号时，表达式求值算法要复杂一些，在遇到左括号时，它表示子表达式的开始，因此应立即压入栈中，这可通过给"("赋一最高的读入优先级来实现，一旦左括号已在栈中，只有当与之匹配右括号出现并在子表达式计算完之后才能将其弹出，因此，在栈中左括号的栈优先级应最低，以保证它在整个子表达式计算完后才被弹出。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

算法 整个算法用两个栈实现，操作数栈为浮点数栈，而运算符栈的元素为类类型 MathOperator 的对象。

表 7.1 运算符的读入优先级、栈优先级和等级

运算符	读入优先级	栈优先级	等级
+，-（双目）	1	1	-1
*，/	2	2	-1
(	3	-1	0
)	0	0	0

```
// 用于处理运算符栈中运算符的类
class MathOperator
{
    private:
      // 运算符及其优先级
      char op;
      int inputprecedence;
      int stackprecedence;
public:
      // 初始化对象的构造函数及允许未初始化数据的缺省构造函数
      MathOperator(void);
      MathOperator(char ch);
  // 处理栈中运算符的成员函数
      int operator > = (MathOperator a) const;
      void Evaluate(Stak < float > &operandStack);
      char GetOp(void);
};
```

对象 MathOperator 中存放运算符及其相应的优先级。非缺省的构造函数设置运算符的读入优先级 inputpredence 和栈优先级 stachperecedence。

```
// 构造函数读入运算符并给出其优先级
MathOperator::MathOperator(char ch)
{
    op = ch;    // 读入运算符
    switch(op)
    {
        // '+'和'-'的读入优先级和栈优先级均为 1
        case '+':
        case '-':    inputprecedence = 1;
                     stackprecedence = 1;
                     break;

        // '*'和'/'的读入优先级和栈优先级均为 2
        case '*':
        case '/':     inputprecedence = 2;
                      stackprecedence = 2;
                      break;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // '('的读入优先级为 3,栈优先级为 -1
        case '(':     inputprecedence = 3;
                      stackprecedence = -1;
                      break;

        // ')'的读入优先级和栈优点级均为 0
        case '(':     inputprecedence = 0;
                      stackprecedence = 0;
                      break;
    }
}
```

类 MathOperator 重载 C++的“>=”运算符用来比较优先级的大小。

```
// 通过比较当前对象的栈优先级和 a 的输入优先级来重载“>=”运算符,用于在读入运算符时来
// 判断在新运算符被压入栈之前是否应先求已有表达式的值
int MathOperator::operator>=(MathOperator a) const
{
    return stackprecedence >= a.inputprecedence;
}
```

该类中还包含成员函数 Evaluate,负责执行运算符,该函数弹出两个操作数,计算完后将结果压回操作数栈。

```
// 为当前运算求值。先从操作数栈中弹出两个操作数,然后执行该运算并将结果压回操作数栈
void MathOperator::Evaluate (Stack< float > &operandStack)
{
    float operand1 = operandStack.Pop();  // 取右操作数
    float operand2 = operandStack.Pop();  // 取左操作数
    // 执行运算并将结果压回栈中
    switch(op)                            // 执行相应运算
    {
        case '+': operandStack.Push(operand2 + operand1);
                  break;
        case '+': operandStack.Push(operand2 - operand1);
                  break;
        case '+': operandStack.Push(operand2 * operand1);
                  break;
        case '+': operandStack.Push(operand2 / operand1);
                  break;
    }
}
```

中缀表达式求值算法读入表达式的各项,将其压入相应的栈中,并修改累计等级,在表达式结束或累计等级越界时,结束输入,读入时应用下列规则:

读入操作数: 将操作数压入操作数栈。

读入运算符: 从栈中弹出所有栈优先级大于或等于当前运算符的读入优先级的运算符,可用类 MathOperator 的“>=”运算来进行比较,在运算符出栈时,用函数 Evaluate 执行该运算。

读入右括号“)”: 弹出并执行所有栈优先级大于 0 的运算符,由于左括号“(”的优先

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

级为 -1,则这一过程在遇到"("时停止,其结果为计算括号内的所有运算符,如果没有"(",则表达式非法(丢失左括号)。

表达式结束,清空运算符栈: 此时累计等级必须为 1,如果小于,则少一个操作数。若还有一个"(",则表达式非法(丢失右括号)。由于运算符均已出栈,函数 Evaluate 执行完了所有运算。表达式的结果可从操作数栈中弹出得到。

程序 7.3 中缀表达式求值

下述程序用来对中缀表达式求值,读入表达式的各项,滤去空格,直到读入"=",处理过程中,若发生错误,则输出相应错误信息,读入结束后,计算表达式中剩余各项,最后打印表达式的结果。

```
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>          // 用于函数'isdigit'中
#include "tstack.h"         // 引入基于模板的类 stack
#include "mathop.h"         // 定义类 MathOperator

// 检查读入字符是否为运算符或括号
int isoperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '(')
        return 1;
    else
        return 0;
}

// 检查读入字符是否为空格
int iswhitespace(char ch)
{
    if (ch == ' ' || ch == '\t' || ch == '\n')
        return 1;
    else
        return 0;
}

// 出错处理函数
void Error(int n)
{
    // 出错信息表
    static char *errormsgs[] = {
                    "Operator expected",
                    "Operand expected",
                    "Missing left parenthesis",
                    "Missing right parenthesis",
                    "Invalid input"
            };
    // 参数 n 为出错信息下标;输出该条信息后退出程序
    cerr << errormsgs[n] << endl;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    exit(1);
}
void main(void)
{
    // 定义对象为 MathOperator 的运算符栈
    Stack < MathOperator > OperatorStack;
// 定义操作数栈
    Stack < float > OperandStack;
MathOperator opr1, opr2;
    int rank = 0;
    float number;
    char ch;
// 处理表达式直到读入"="
    while(cin.get(ch) && ch ! = '=')
    {
        // * * * * * * * * 处理浮点操作数 * * * * * * * *
        if (isdigit(ch) || ch = = '.')
        {
            // 留下数字及小数点
            cin.putback(ch);
        cin > > number;
        // 操作数级别为 1,累计级别也必为 1
        rank + + ;
        if (rank > 1)
            Error(OperatorExpected);
        // 将操作数压入操作数栈
        OperandStack.Push(number);
    }
    // * * * * * * * * 处理运算符 * * * * * * * *
    else if (isoperator(ch))
    {
        // 余外所有运算符的级别为 - 1,累计级别应为 0
        if (ch ! = '(')        // '('的级别为 0
            rank - - ;
        if (rank < 0)
            Error(OperandExpected);
        // 建立一个存放当前运算符的 MathOperator 对象,其栈顶运算符的优先级 > = 当
        // 前运算符优先级,则弹出栈顶运算符并执行其运算。将当前运算符压入栈中
        opr1 = MathOperator(ch);
        while(! OperatorStack.StackEmpty() &&
            (opr2 = OperatorStack.Peek()) > = opr1)
        {
            opr2 = OperatorStack.Pop();
            opr2.Evaluate(OperandStack);
        }
        OperatorStack.Push(opr1);
    }
    // * * * * * * * * 处理右括号 * * * * * * * *
    else if (ch = = rightparenthesis)
    {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 建立一个存放")"的 MathOperator 对象,弹出运算符栈并求值直到遇到'('或栈为
        空。
        // 若栈为空,则遗失"(";否则,删除')'
        opr1 = MathOperator(ch);
        while(! OperatorStack.StackEmpty() &&
              (opr2 = OperatorStack.Peek() > = opr1))
        {
            opr2 = OperatorStack.Pop();
            opr2.Evaluate(OperandStack);
        }
        if(OperatorStack.StackEmpty())
            Error(MissingLeftParenthesis).
            opr2 = OperatorStack.Pop();      // 删除'('
        }
        // * * * * * * 非法输入 * * * * * *
        else if (! iswhitespace(ch))
            Error(InvalidInput);
    }
    // 完整表达式的级别应为 1
    if (rank ! = 1)
        Error(OperandExpected);
    // 请用运算符栈并完成表达式运算。若发现左括号,则遗失右括号
    while(! OperatorStack.StackEmpty())
    {
        opr1 = OperatorStack.Pop();
        if (opr1.GetOp() = = leftparenthesis)
            Error(MissingRightParenthesis);
        opr1.Evaluate(OperandStack);
    }
    // 表达式结果在操作数栈的栈顶
    cout < < "The value is " < < OperandStack.Pop() < < endl;
}
/*
<程序 7.3 运行结果之一>
2.5 + 6/3 * 4 - 3 =
The value is 7.5
<程序 7.3 运行结果之二>
(2 + 3.25) * 4 =
The value is 21
<程序 7.3 运行结果之三>
(4 + 3) - 7) =
missing left parenthesis
*/
```

书面作业

7.1 (a) 编写通用函数 Max,返回两个值的最大值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(b) 为 C++ 串类型写一个 Max 的重载版本，用参数传入指向字符串的指针并返回最大串的指针。

7.2 编写具有下列函数的模板类：

int Insert(T elt);	往一可存放五个类型为 T 的私有数组 dataElements 中插入 elt。dataElements 中可插入元素位置的索引由数据成员 loc 给出，它也是 dataElements 的数据成员值。若 dataElements 中已满，则返回 0。
int Find(T elt);	在 dataElements 中查找元素 elt，若找到，则返回其下标，否则返回 -1。
int Numelts(void);	返回存放于 dataElements 中的元素个数。
T & GetData(int n);	返回 dataElements 中位于位置 n 的元素。若 n<0，或 n>4，打印出错误信息并退出。

7.3 编写函数

```
template <class T>
int Max(T arr[ ], int n);
```

返回数组中最大值的下标。

7.4 实现函数

```
template <class T>
int BinSearch (T A[ ], T key, int low, int high);
```

在数组 A 中用折半查找法查找 key。

7.5 编写函数

```
template <class T>
void InsertOrder (T A[ ], int n, T elem);
```

往数组 A 中插入 elem 并保持 A 为升序顺序，注意往某一位置插入 elem 时，必须先右移所有其它的元素。

7.6 (a) 给出类 SeqList 的基于模板的声明。

(b) 对于该模板类，实现其构造函数及 DeleteFront 和 Insert 函数。

(c) 声明一个 SeqList<T> 的对象之前，必须对类型 T 定义什么操作？

(d) 声明 S 为 Stack 的对象，栈中的元素为 SeqList 对象。

上机题

7.1 用主程序调用方式测试书面作业 7.1 中的 Max 函数，主程序中应包括两个C++串。

7.2 实现下面声明的模板函数 Copy。

```
template <class T>
void Copy (T A[], T B[], int n);
```

它从数组 B 中拷贝 n 个元素到数组 A，再写一主程序来测试 Copy，主程序至少包括下列数组

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
(a) int AInt[6], BInt[6] = {1,3,5,7,9,11};
(b) struct Student
    {
        int field1;
        double field2;
    };
    student AStudent[3];
    student BStudent[3] = {{1,3,5},{3,0},{5,5,5}};
```

7.3 本题使用书面作业 7.2 中的模板类 DataStore，写一重载运算符“<<”用函数 GetData 输出对象 DataStore 中的数据，记录 Person 结构如下：

```
Struct Person
{
    char name[50];
    int age;
    int height;
};
```

对 Person 重载运算符“==”来比较域 name。写一程序对 Person 插入元素直到 DataStore 对象已满。在输入中应包括数据 Person P {"70hn", 25, 72}。

查找 P 并打印查找的结果，用“CC”输出该对象。

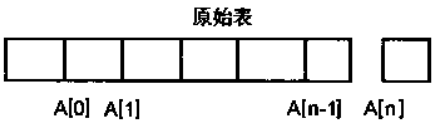
7.4 扩充程序 7.3 使其可处理由字符“^”表示的乘方运算，乘方为右结合的。如：

```
2^3 = 8            // 2^3 = 8
2^2^3 = 2^(2^3) = 256
```

计算 a^b 可用数学库 <math.h> 中的函数 pow，即

```
a^b = pow(a,b)
```

7.5 顺序查找要在表中查找与 key 匹配的元素，对表中的每个元素，我们都要完成两个任务：检查是否匹配和是否到表尾。算法的一个改进可在表尾增加一个 key 值，由于肯定可以找到匹配的元素，就不再需要判别是否到表尾了，这个算法称为快速顺序算法。



(a) 用代码实现上述算法

```
template<class T>
int FastSeqSearch(T A[ ] int n T key);
```

(b) 用 FastSeaSearch 重写程序 7.1。

7.6 为统计任意数据类型表中各值的出现频率，定义类 DataInfo，它包括两个数据域，一个为表中的元素值，另一个为该值出现次数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
template <class T>
class DataInfo
{
    private:
        T data;                 // 元素值
        int frequency;          // 该值出现次数

    public:
        // 增加出现次数
        void Increment(void);
        // 关系运算符"==”及"<"必须为类型 T 的合法运算

        // 比较元素值
        int operator==(const DataInfo<T>& w);
        // 比较出现次数
        int operator<(const DataInfo<T>& w);
        // 流运算符"<<"和">>"必须为类型 T 的合法运算
        friend istream& operator>> (istream & is,
                                    DataInfo<T> &w);
        friend ostream& operator<< (ostream& os, const
                                    DataInfo<T>& w);
};
```

类 DataInfo 中函数 Increment 给计数器加 1，该输出函数按格式"value:count"形式输出对象，执行重载的流输入符"<<"，从输入流中读入数据值时，创建一个具有初始值和计数值为 1 的记录，为方便从 DataInfo 对象中寻找某一特定值及按出现次数排序表，在类上定义了关系运算符。

(a) 实现上述类 DataInfo。

(b) 写一主程序提示用户输入一个整数表的元素个数，然后创建 DataInfo <int> 数组 dataList 并读入各整型值，用函数 SeqSearch 来判定输入的值是否已在数组 dataList 中，若返回 -1，表明读入了一个新值，并将其加入到 dataList 数组中；否则 dataLish 数组中相应值的计数域上加 1，最后，调用 ExchangeSort 对 dataList 排序并打印各值及其出现次数。

7.7 修改上机题 7.6，使 DataInfo 保持升序，用折半查找来判断某值是否还存在，可用书面作业 7.4 和 7.5 提供的函数实现这些功能，当然，最后就不必进行排序了。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 8 章　类和动态存储

8.1　指针与动态数据结构
8.2　动态申请对象
8.3　赋值与初始化
8.4　安全数组
8.5　串类
8.6　模式匹配
8.7　整型集合
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

到现在为止，我们在书中还只用到**静态数据结构**来实现集合类，包括用静态数组实现的类 Seqlist、Stack 及 Queue。由于静态数组的大小在统计时就已确定，所以使用静态数组无法在运行时重新调整数组大小。为满足不同用户的需要，每个类都将准备一个合理大小的数组。这样，对许多应用来说，浪费了一些内存；但在有些情况下，又会使数组空间不够，用户必须修改源代码来增加数组的大小，并重新编译程序。

本章我们引入**动态数据结构**，运行时从系统中申请内存。从程序的模块结构中，我们已经得到一些申请内存的知识，编译器创立全局数据，并在进入模块时申请自动变量的内存，在退出模块时将申请到的内存释放，例如，C++编译器需为下列代码段申请存放全局数据、参数和局部变量的内存空间，编译时已确定每个变量的类型和大小。

```
int global = 8;          // 全局变量
// 为参数 x 和 y 从系统栈中申请内存
void subtask (int x, long *y)
{
    int z;               // 局部变量 z
    …                    // 退出时，释放所有变量和参数的空间
}
```

编译器也给用户提供创建动态数据的能力，运算符 new 可在程序执行时从系统内存中申请内存来使用，对应的运算符 delete 将内存送回系统以供下次申请。动态数据结构对于那些仅在运行时才知道内存要求的应用十分重要，它的运用是研究集合的基础，并有效地解除了我们用静态结构时的大小限制。例如，类 Stack 的最大空间由缺省值 MaxStackSize 限制，并要求用户增加元素时检查栈满(StackFull)状态。动态内存增强了类 Stack 的可用性，它可使类申请到足够的空间来适应具体应用的需要。数据库应用中，经常要用临时空间存放数据库表和用户查询产生的数据，内存的大小根据不同的查询申请，用完以后释放。

动态内存的使用也有一定的限制和危险，在用到动态申请的可变大小的数据的应用中，随着程序的运行，申请不断增加，可能导致可用内存的枯竭，用户最后得到“内存用完”的信息，此时，程序将被中止运行，这种情况可能在运行图形用户界面的应用中出现，这些程序用许多窗口来显示数据，建立菜单等。即使程序结构非常好，用户也可能打开太多的活动图形窗口，最终用完所有可用的内存。所以在使用动态内存时，程序员必须记住内存是一种资源，并有效地进行管理，当不再需要时，动态申请的内存必须立即释放掉。编译器为参数和局部变量申请空间也遵循这条原则，程序可用 delete 运算符来遵守这条原则。

C++为处理动态数据提供了许多函数，删除对象时，可用析构函数来释放由该对象申请的动态内存，另外，类还可以有复制构造函数和重载的赋值运算符，用来进行对象间的拷贝和赋值。本章将集中讨论这些函数，我们以 DynamicClass 为例来介绍这些函数。

动态数组允许我们在运行时申请一块内存空间，对大多数应用来说，我们能在建立数组时确定其需要的大小，但在一些特殊情况下，我们可能需要扩充或改变数组大小。为提供这种能力，我们给出类 Array，它可创建任意大小的表并实现数组的越界检查和大小调整，同时，给出了析构函数、复制构造函数及重载的赋值运算符的用法。由于希望 Array 的对象看起来像标准的 C++数组，我们重载了下标运算符“[]”。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

串是一种基本的数据结构，一些语言中定义了固定的串类型，本章中，我们给出了一个完整的类 String，用它来解决模式匹配的问题，这个类将一直用到本书结束。

集合是许多数学理论的基础。在计算机应用中，集合也是一种有力的数据结构，在许多领域和图形、网络的实现中，有着重要的应用。

我们用 C++ 的位操作实现了存放整型数据的类 Set，这不仅节省了内存空间，也提高了运行效率。类 Set 可用来实现著名的求质数的算法——"筛子"算法。

8.1 指针与动态数据结构

第 2 章中已介绍过指针这种数据结构，本节中，我们将指针变量和 C++ 的 new 及 delete 操作结合起来，用来申请和释放动态内存。

动态申请内存操作符 new

C++ 用操作符 new 在程序执行期间为数据申请内存。得到数据的大小后，可用该运算符请求系统分配足够的内存来存放数据。并返回指向新开辟内存的指针，若申请不到足够的内存，则返回 0(NULL)。

操作符 new 以数据类型 T 为参数，并为类型为 T 的变量申请内存，返回得到内存的地址，如：

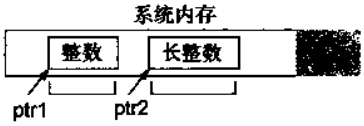
```
T * p;          // 定义 p 为指向 T 的指针
p = new t;      // p 为类型 T 数据的内存地址
```

再如，变量 ptr1 和 ptr2 分别为指向 int 和 long 类型数据的指针：

```
int * ptr1;       // int 的大小为 2
long * ptr2;      // long 的大小为 4
```

运算符 new 可将一个 int 类型的指针赋给 ptr1，long 类型的指针赋给 ptr2。

```
ptr1 = new int;     // ptr1 指向内存中的一个整数
ptr2 = new long;    // ptr2 指向内存中的一个长整数
```



这时，ptr1 中存放的是一个 2 字节长的整数的内存地址，与此类似，ptr2 中存放的是 4 字节长的长整数的内存地址。缺省情况下，内存中的内容没有初值，如果需要初值的话，必须以操作符 new 的参数形式提供：

```
P = new T (value);
```

如下述操作：

```
ptr2 = new long(100000);
```

为长整数中请内存并对其赋初值 100000。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

申请动态数组

动态内存申请的作用在申请整个数组时尤其明显。在具体应用中，我们假定运行时才知道数组大小，可用 new 运算符为带方括号"[ ]"的数组申请内存，若 P 指向类型 T 的数据，语句

```
P = new T[n];    // 为类型为 T 的 n 元数组申请内存
```

将 P 指向数组的第一个元素，用这种方式申请的数组不能指定初始值。

例 8.1

下述语句为一个有 50 个长整数元素的数组申请内存，若 p 为 NULL，即内存申请失败，程序中断执行。

```
long *p;
p = new long[50];    // 为一有 50 个长整数的数据申请内存
if (p = = NULL)
{
    cerr < < "Memory allocatim error !" < < endl;
    exit(1);          // 终止程序运行
}
```

释放内存操作符 delete

管理内存是程序员的职责。C + + 提供操作符 delete 来释放原来由 new 申请的内存，delete 语法十分简单，只与 C + + 运行时每次调用 new 保留的信息有关。假定 p 和 q 指向动态申请的内存：

```
T *p, *q;          // p 和 q 均为指向类型 T 的指针
p = new T;         // 指向单个元素
q = new T[n]       // 指向一个 n 元数组
```

函数 delete 使用这些指针来释放为它们申请的内存。释放数组时，delete 与"[ ]"运算符一起使用，即：

```
delete p;       // 释放由 p 指向的变量空间
delete [ ] q;   // 释放由 q 指向的数组空间
```

例 8.2

用 delete 操作符释放动态申请的数组 p：

```
long *p;
p = new long[50]      // 为一有 50 个长整数的数组申请空间
delete [ ] p;         // 释放该数组空间
```

8.2 动态申请对象

与其它类型变量一样，类类型的对象可被定义成静态变量，也可由 new 动态申请。一般来说，它们都调用构造函数来初始化变量并为一个或多个数据成员动态申请内存，语法

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

规则与简单类型的数组相同，操作符 new 为对象申请内存，并调用类的构造函数对其初始化，若构造函数需要参数，也由 new 提供。

我们以基于样板的类 DynamicClass 为例来介绍对象的动态申请，它有一个静态数据成员和一个动态数据成员，下面是类的定义，其成员函数在本节和 8.3 节中介绍，该类也可在附录程序的文件"dynamic.h"中找到。

```
#include <iostream.h>
template <class T>
class DynamicClass
{
    private:
        // 类型 T 的变量及一个指向类型 T 数据的指针
        T  member1;
        T  *member2;
    public:
        // 构造函数
        DynamicClass(const T& m1, const T& m2);
        DynamicClass(const DynamicClass<T>& obj);
    // 析构函数
        ~DynamicClass(void);
        // 赋值运算符
        DynamicClass<T>& operator=(const DynamicClass<T>& rhs);
};
```

这个只有两个数据成员的简单的类给出了动态申请的对象中成员函数处理动态申请的实质功能，它只为说明这些功能而设计，没有实际意义。

类的构造函数用参数 m1 初始化静态数据 member1，对 member2，则需先为变量申请内存，然后再赋初值 m2：

```
// 用带参数的构造函数来初始化成员数据
template <class T>
DynamicClass<T>::DynamicClass(const T& m1, const T& m2)
{
    // 参数 m1 初始化静态成员
    member1 = m1;
    // 申请动态内存并用 m2 初始化
    member2 = new T(m2);
    cout << "Constructor:" << member1 << '/'
         << *member2 << endl;
}
```

例 8.3

下面的语句定义了静态变量 staticObj 和指针变量 dynamicObj。对象 staticObj 用参数 1 和 100 初始化其数据成员。

```
// DynamicClass 的对象
DynamicClass<int> staticObj(1,100);
```

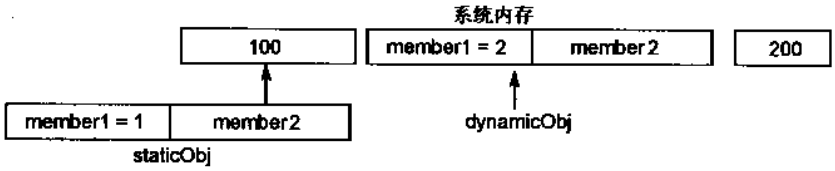
dynamicObj 指向的对象由 new 操作符产生，并将参数 2 和 200 传递给构造函数，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在创建对象 * dynamicObj 时，类的构造函数将其数据成员初始化为 2 和 200。

```
//指针变量
DynamicClass < int > *dynamicObj;
// 申请一个对象的空间
dynamicObj = new DynamicClass < int > (2,200);
```



释放对象数据：析构函数

下述函数 DestroyDemo 创建了一个数据类型为整型的 DynamicClass 对象。

```
void DestroyDemo (int m1, int m2)
{
    DynamicClass < int > obj (m1,m2);
}
```

从 DestroyDemo 返回后，obj 被编译器删除了，但并没有释放该对象申请的动态内存。如图 8.1 所示。

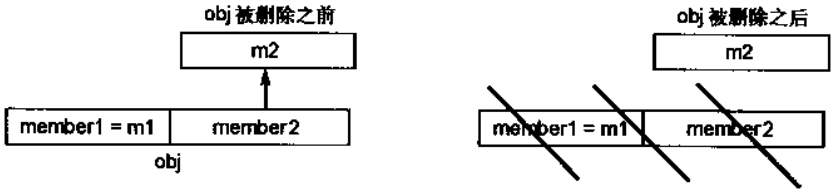


图 8.1 需要释放内存

为有效管理内存，我们应在删除对象的同时释放由对象申请的内存空间。即完成申请空间的构造函数的反动作。C + + 语言提供了称为析构函数的成员函数，供编译器在删除对象时调用，对于 DynamicClass，其析构函数的定义为：

```
~DynamicClass(void);
```

字符“~”表示“求反”，所以 ~DynamicClass 是构造函数的反操作。析构函数没有参数和返回类型，对于例子中的类，析构函数负责释放 member2 的动态内存：

```
// 析构函数，用于释放由构造函数申请的内存
template < class T >
DynamicClass < T > :: ~DynamicClass(void)
{
    cout < < "Destructor: " < < "member1 < < '/'
        < < *member2 < < endl;
    delete member2;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

析构函数在删除对象时调用，程序结束时，将删除所有全局对象及主程序中定义的对象，对在程序段中创建的局部对象，则在程序退出该程序段时删除。

程序 8.1　析构函数

本程序给出析构函数的定义及使用，程序中有三个对象。obj1 是主程序中定义的变量，obj2 是指向动态对象的指针，上面讨论过的函数 DestroyDemo 也在其中，并定义了一个局部对象 obj，图 8.2 给出各对象的构造函数和析构函数出现的不同位置。

```
void DestroyDemo(int m1,int m2)
{
  DynamicClass<int> Obj(m1,m2);  <---- Obj(3,300) 的构造函数
}  <---------------------------- Obj 的析构函数
void main(void)
{
  DynamicClass<int>Obj_1(1,100),*Obj_2;  <---- Obj_1(1,100) 的构造函数
  Obj_2 = new DynamicClass<int>(2,200);  <---- *Obj_2(2,200) 的析构函数
  DestroyDemo(3,300);
  delete Obj_2;  <---------------------- Obj_2 的构造函数
}  <---------------------------------- Obj_1 的析构函数
```

图 8.2　三个对象的构造函数和析构函数

```
#include <iostream.h>
#include "dynamic.h"
void DestroyDemo(int m1,int m2)
{
    DynamicClass(int) obj(m1,m2);
}
void main(void)
{
    // 创建自动变量对象 obj_1,使其 member1 = 1, * member2 = 100
    DynamicClass < int > Obj_1(1,100);
    // 定义一指象对象的指针
    DynamicClass < int >  * Obj_2;
    // 为 Obj_2 申请动态内存,并使其 member1 = 2, * member2 = 200
    Obj_2 = new DynamicClass < int > (2,200);
    // 用参数 3/300 调用函数 DestroyObject
    DestroyDemo(3,300);
    // 显式释放 Obj_2
    delete Obj_2;
    cout < < "Ready to exit program." < < endl;
}
/*
<程序 8.1 运行结果>
Constructor: 1/100
Constructor: 2/200
Constructor: 3/300
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Destructor: 3/300
Destructor: 2/200
Ready to exit program.
Destructor: 1/100
*/
```

8.3 赋值与初始化

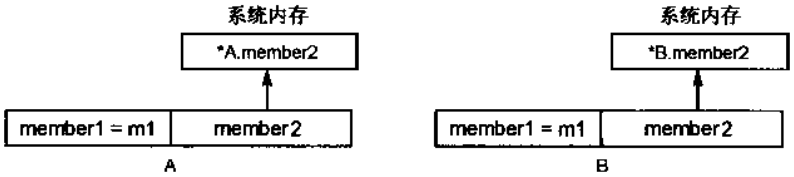
赋值和初始化是可用于所有对象的最基本的操作，赋值语句 Y = X 将对象 X 中的数据按位拷贝给对象 Y。初始化则创建一个新的对象，它是另一个对象的拷贝，下面以对象 C 和 Y 为例说明这个操作：

```
// 创建 DynamicClass 的对象 X 和 Y,并用 X 的数据初始化 Y
DynamicClass X(20,50), Y = X;
Y = X;          // Y 中的数据被 X 的数据覆盖
```

对于动态内存，我们必须作些特殊的考虑来保证不出现意外的错误，用一些新的函数来处理对象赋值和初始化，本节中，我们先讨论问题的实质，然后给出新的类函数。

对象的赋值

DynamicClass 的构造函数初始化 member1，并申请由 member2 指向的内存，例如，在上面的对象 A 和 B 的定义中，我们创建了两个对象及与其相关联的用 new 操作符申请的两块内存。



赋值语句 B = A 将 A 中的数据拷贝到 B。

```
B 的 member1 = A 的 member1     // 从 A 拷贝静态数据到 B
B 的 member2 = A 的 member2     // 从 A 拷贝到指针 B
```

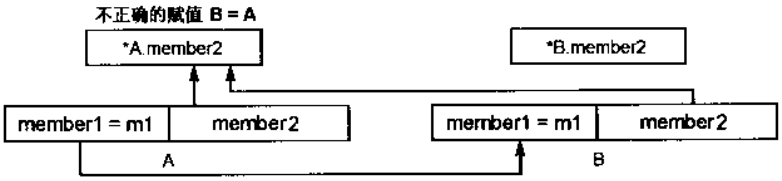
这样，将对象 B 的指针值 member2 赋值为对象 A 的指针值 member2，使两个指针指向内存中的同一位置，而原由 B 指向的动态内存没有指针指向。假设函数 F 中包含上述赋值语句。

```
void F(void)
{
    DynamicClass < int > A(2,3) ,B(7,9);
        ...
    B = A;     // 将对象 A 赋值给对象 B
        ...
}
```

当从上述函数中返回时，将调用类的析构函数删除该程序块中创建的所有对象，即释

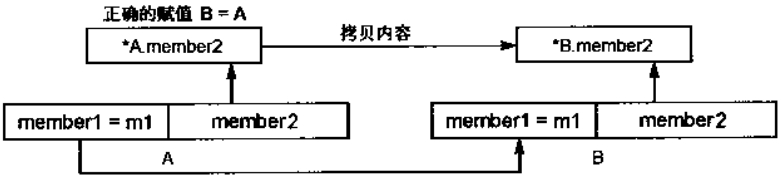
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



放由 member2 指向的动态内存。我们假定先删除对象 B,析构函数释放变量 * B.member2 指向的内存,也就是 A 中 member2 指向的内存,再删除对象 A 时,本来应释放变量 * A.member2 指向的内存,但它在删除 B 时已被释放掉了,这将导致 A 的析构函数发生错误,多数情况下,这都是致命错误。

问题的根源在于赋值语句 B = A。它将 A 中的 member2 指针值拷贝到 B 的 member2 指针,我们实际上希望的是将 A 中的 member2 指针指向的内容拷贝到 B 的 member2 指向的内存。



重载赋值运算符

为正确处理带动态数据的对象间的赋值,C++ 允许重载赋值运算符"="为一成员函数,在 DynamicClass 中重载赋值运算符的语法为:

```
DynamicClass<T>& operator= (const DynamicClass<T> & rhs);
```

它将参数 rhs 作为等式右边的操作数实现了"="这个运算符,如:

```
B = A;          // 作为 B.=(A)实现
```

对于每个包含有类型为 DynamicClass 的对象的赋值语句都将执行重载后的"="运算符。它用数据成员,包括由这些成员指向的数据之间的显式的赋值语句代替了简单的从对象 A 到对象 B 的数据的位拷贝。我们用常量地址来传递参数 rhs 过程,不仅避免了可能拷贝一个大对象到参数中,且不允许对对象作任何修改。另外,请注意不管什么时候用到样板类的名称,都必须将类型"<T>"拼在类名的后面。

对于 DynamicClass,赋值运算符必须从对象 rhs 中将数据成员 member1 的值拷贝到当前对象的 member1 中,同时将 rhs 中 member2 指向的内容拷贝到当前对象的 member2 指向的内容:

```
// 重载赋值运算符。返回指向当前对象的指针
template <class T>
DynamicClass<T>& DynamicClass<T>::operator=
                    (const DynamicClass<T>& rhs)
{
    // 将静态数据从 rhs 中拷贝到当前对象
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    member1 = rhs.member1;
    // 动态内存中的数据也必须同 rhs 保持一致
    * member2 = * rhs.member2;
    cout < < "Assignment Operator:" < < member1 < < '/'
         < <  * member2 < < endl;
    return * this;
}
```

保留字 this 用来返回当前对象的地址，我们将在后面讨论。重载后的赋值运算符将对象 rhs 的数据传送到当前对象，保证了赋值语句

```
B = A
```

的正确执行。

由于运算符“=”返回的是当前对象的地址，因此，我们可将两个或多个赋值语句连起来，如

```
C = A = B;          // 将结果(B = A)赋值给 C
```

this 指针

每个 C++对象都有一个名为 this 的指针，它在对象建立时自动定义，是一个可用于类中成员函数的保留字。this 为指向当前对象的指针，“* this”即为对象本身。如在类型 DynamicClass 的对象 A 中。

```
* this 为对象 A
this→member1 为 A 中的数值成员 member1 的值
this→member2 为 A 中的指针 member2
```

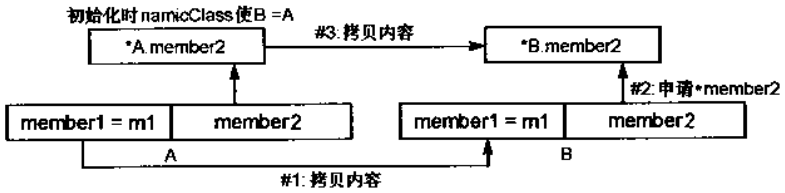
对于赋值运算符，返回值为参数的引用。表达式“return * this”返回对当前对象的引用。

对象的初始化

对象的初始化创建一个新的对象，并将另一对象拷贝到新对象。与赋值相同，当对象中有动态数据时，它需要一个特定的成员函数，我们称为复制构造函数来完成拷贝功能，我们用下面的例子来说明复制构造函数的功能：

```
DynamicClass < int > A(3,5), B = A;    // 用 A 初始化对象 B
```

上述声明创建了对象 A，其初值为 member1 = 3 及 * member2 = 5，同时创建对象 B，并将 A 中的数据值存放到 B 中，这个初始化过程应包括从 A 中拷贝数据值 3 到 B 的 member1，为 B 中 member2 指向的数据申请内存，然后将 * A.member2 的值 5 拷贝到 B 动态申请的内存中。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

除了在声明对象时将其初始化外，函数调用时以对象为值参及从函数中返回对象值时也要 DynamicClass < int > 的值参 X。

```
DynamicClass < int > F(DynamicClass < int > X) // 值参
{
    DynamicClass < int > obj;
    ...
    return obj;
}
```

当调用程序以对象 A 为实际参数调用 F 时，将通过拷贝 A 来创建局部对象 X。

```
DynamicClass < int > A (3,5), B(0,0);     // 声明对象
B = F(A);                                 // 将 A 拷贝到 X 来调用 F
```

从 F 中返回时，先创建一个 obj 的副本，然后调用局部对象 X 和 obj 的析构函数，并将 obj 的副本作为函数值返回。

创建复制构造函数

为正确处理申请动态内存的类，C + + 提供复制构造函数来为新的对象申请动态内存并初始化其数据值，我们通过类 DynamicClass 的复制构造函数来说明这一概念。

复制构造函数是一函数名与类名相同且有一个参数的成员函数，由于它是一个构造函数，它没有返回值：

```
DynamicClass(const DynamicClass < T > & X);     // 复制构造函数
```

DynamicClass 的复制构造函数从 X 的 member1 中拷贝数据到当前对象，并申请由 member2 指向的内存，然后用值 * X.member2 对其进行初始化。

```
// 复制构造函数。初始化新的对象，并使其数据与 obj 相同
template < class T >
DynamicClass < T > ::DynamicClass(const DynamicClass < T > & obj)
{
    // 从 obj 中拷贝静态数据到当前对象
    member1 = obj.member1;
    // 申请动态内存并用 * obj.member2 的值对其初始化
    member2 = new T( * obj.member2);
    cout < < "Copy Constructor: " < < member1
         < < '/' < < * member2 < < endl;
}
```

如果类中定义了复制构造函数，编译器在需要初始化时调用它，复制构造函数只在创建对象时用到。

尽管有些相似之外，对象的赋值和初始化显然是有区别的。当操作数据已存在时，我们才能进行赋值运算；而初始化将创建一个新的对象，并从一个已存在的对象中拷贝数据给新的对象，而且在初始化过程中，在拷贝动态数据之前，必须用操作符 new 为它申请内存。

复制构造函数的参数必须以地址传递。否则，如果编译器也没有识别这个错误的话，

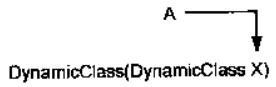
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

则会引起灾难性的后果，例如，定义一个用值参的复制构造函数。

```
DynamicClass (DynamicClass <T> X);
```

它要求以值参的形式来调用复制构造函数，假设对象 A 的值作为参数传递给 X：



由于我们用值将 A 传递给 X，拷贝 A 到 X 时就必须调用复制构造函数；这样，我们将产生一个调用复制构造函数的死循环。幸运的是，编译器可以捕捉这个本质错误，它要求参数以地址传递，另外，参数的地址也应定义为 const，因为我们在拷贝时不需要修改该对象。

程序 8.2　DynamicClass 的使用

下面程序用整型数据给出了 DynamicClass 成员函数的功能，其中的注释可方便对程序的理解。

```
# include <iostream.h>
# include "dynamic.h"
template <class T>
DynamicClass<int> Demo (DynamicClass<T> one,
                        DynamicClass<T>& two, T m)
{
    // 调用构造函数
    DynamicClass<T> obj(m,m);
    // 复制 obj 并将其作为函数的返回值
    return obj;
    // 从 Demo 返回时，临时对象 T 和 one 被删除
}
void main()
{
        /*    A(3,5) 调用构造函数(member1 = 3, *member2 = 5)
              B = A 调用复制构造函数用对象 A 初始化对象 B (member1 = 3, *member2 = 5)
              对象 C 调用构造函数(member1 = 0, *member2 = 0) */
    DynamicClass<int> A(3,5), B = A, C(0,0)
        /*    调用函数 Demo，复制构造函数通过拷贝对象 A 创建了参数 one(member1 = 3, *
              member2 = 5)。由于参数 two 用形参传送，所以不调用复制构造函数。到返回
              时，创建一个局部变量 obj 的副本，然后将其赋值给对象 C
    C = Demo(A,B,5);
    // 程序退出时删除所有未删除的对象。
}
/*
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
<程序 8.2 运行结果>
Constructor: 3/5
Copy Constructor: 3/5
Constructor: 0/0
Copy Constructor: 3/5
Constructor: 5/5
Copy Constructor: 5/5
Destructor: 5/5564
Destructor: 3/5556
Assignment Operator: 5/5
Destructor: 5/5
Destructor: 5/5
Destructor: 3/5
Destructor: 3/5
*/
```

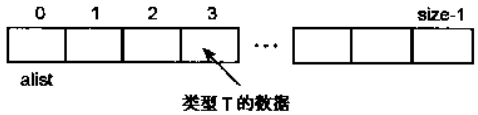
8.4 安全数组

静态数组是具有固定元素个数的群体。它通过下标访问其中的元素，是实现表的基础数据结构。尽管静态数组是十分重要的数据结构，但也存在缺憾，其大小在编译时就已确定，在运行时无法修改。

为了弥补静态数组的这些不足，我们创建一个基于模板的类 Array，它由一系列位置连续的任意类型的元素组成，而其元素的个数可在程序运行时改变；并有实现下标和指针转换的函数，这是通过重载 C++下标运算符"[ ]"来实现的。它保证每个下标对应于表中的一个元素，如果下标越界，则进行转换时将产生出错信息。这样得到的对象，我们可以称之为"安全数组"。因为它能捕捉非法的数组下标，因此，数组对象可用于以一般的数组为参数的函数。我们定义一个通用的指针转换运算符 T* 将 Array 对象和普通的元素类型为 T 的数组联系起来。

类 Array

下面的基于样板的类 Array 为任意类型的一系列元素申请内存。



类 Array 定义

声明

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

enum ErrorType {invalidArraySize, memoryAllocationError,
                indexOutOfRange};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
char * errorMsg[] =
{
"Invalid array size", "Memory allocation error",
"invalid index: "
};

template < class T >
class Array
{
    private:
        // 一个动态申请的包含 size 个元素的表
        T* alist;
        int size;

        // 出错处理函数
        void Error(ErrorType errorCommitted, int badIndex = 0)const;

    public:
        // 构造函数和析构函数
        Array(int sz = 50);
        Array(const Array < T > & X);
        ~Array(void);

        // 赋值,下标和指针转换操作
        Array < T > & operator = (const Array < T > & rhs);
        T& operator[] (int i);
        operator T* (void) const;

        // 有关表大小的函数
        int ListSize(void) const;     // 取表的大小
        void Resize(int sz);          // 修改表的大小
};
```

说明

重载的下标运算符"[ ]"及转换运算符 T* 的使用使 Array 对象可以起到普通语言定义的数组的作用。可将一个 Array 对象赋值为另一个 Array 对象的赋值运算符,增加了类 Array 的灵活性,因为对语言中定义的数组来说,赋值是非法的操作。

Resize 函数可使我们改变表的大小。当参数 sz 大于当前数组的大小 size 时,保留旧表中的数据并增加表的大小到 sz;而当 sz 小于表的当前大小时,我们则只保留表中前 sz 个元素并将其它元素删除。

例:

```
Array < int > A(20);              // 20 个整数的数组
cout < < A.Size();                // 输出当前大小为 20

for (int i = 0; i < 20; i + + )   // 用'[]'访问数据元素
    A[i] = i;
A[25] = 50;                       // 非法下标
A.Resize(30);                     // 将数据大小增加到 30

A[25] = 50;                       // 这时才合法
ExchangeSort(A,30);               // 转换允许使用数组参数
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

为类 Array 申请内存

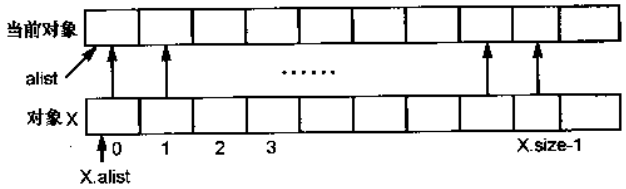
下面我们给出函数 Array 的构造函数，析构函数和复制构造函数，这些函数中都有必要的动态数组，初始数组的大小由参数 sz 决定，其缺省值为 50：

```
// 构造函数
template <class T>
Array<T>::Array(int sz)
{
    // 检查数组大小参数是否合法
    if (sz <= 0)
       Error(invalidArraySize);
    // 给 size 赋值并动态申请内存
    size = sz;
    alist = new T[size];
    // 确保系统分配了所需内存
    if (alist == NULL)
       Error(memeoryAllocationError);
}
```

析构函数删除为数组 alist 申请的内存：

```
// 析构函数
template <class T>
Array <T>::~Array(void)
{
    delete [ ] alist;
}
```

复制构造函数实现了语言本身定义的数组没有实现的功能，它使我们可用另一个数组 X 来初始化一个数组，也可将一个 Array 对象以值传递给函数，其处理过程为先取得对象 X 的大小，申请足够大小的动态内存，然后拷贝 X 的元素当前对象。




```
// 复制构造函数
template <class T>
Array<T>::Array(const Array<T>& X)
{
    // 取得对象 X 的大小并将其赋值给当前对象
    int n = X.size;
    size = n;
    // 为对象申请新内存并进行出错检查
    alist = new T[n]            // 申请动态内存
    if (alist == NULL)
       Error(memoryAllocationError);
    // 从 X 中拷贝数组元素到当前对象
    T* srcptr = X.alist;        // X.alist 的首地址
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    T* destptr = alist;          // alist 的首地址
    while(n - - )                // 拷贝表
        * destptr + + = * srcptr + + ;
}
```

Array 越界检查及重载下标运算符[ ]

数组下标用[ ]引用,在表达式中,用下述形式表示:

```
P[n]
```

其中 P 为类型 T 的指针,n 为整数表达式,这实际上是 C + + 中的数组下标运算符,它有两个操作数 P 和 n,返回位于位置 P + n 的数据。

该运算符只能以成员函数方式重载并用于对 Array 对象的数据进行下标访问。

下面我们为类 Array 重载下标运算符,设 A 为整型 Array 对象,希望用 A[n]的形式来访问其元素,例如,语句:

```
A[0] = 5;
```

将数值赋给数组中第一个元素(alist[0] = 5)。成员函数[ ]的定义为如下形式:

```
T& operator [ ] (int n);
```

其中 T 为对象中存放数据的类型,n 为下标;由于重载的运算符[ ]返回值为地址,因此下标运算符也可在赋值语句左边出现。

```
Value = A[n];      // 将 A[n]赋值给 value
A[n] = Value;      // 将 value 赋值给 A[n]
```

对于类 Array,重载的下标运算符通过检查下标 n 是否在数组的下标范围 0 至 size - 1 之间来提供"安全"数组,若 n 不在范围之内,则打印出现错信息,程序终止运行,否则,它返回 alist[n]。

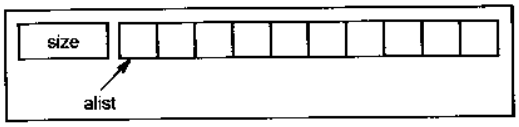
```
// 重载下标运算符
template < cCass T >
T& Array < T > ::Operator[ ] (int n)
{
// 数据越界检查
if (n < 0 || n > size - 1)
    Error(index out of Range, n);
// 从私有数组中返回元素值
return alist [n];
}
```

将 Array 对象转换为指针

通过指针转换,可让用户在任何以普通数组为参数的函数中用 Array 对象作运行参数。为实现这个目的,必须重载类型转换运算符 T * (),让它将对象转换为指针。具体到 Array 对象,即给出数组 alist 的起始地址。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



对象A

例如，函数 ExchangeSort 和 SeqSearch 需要数组参数。对于元素类型为整数的对象 A，下面函数调用是正确的。

```
// 对具有 A.size( )个元素的数组 A 排序
ExchangeSort(A,A.size());
// 在数组 A 中查找 key
index = SeqSearch(A,A.Size(), int Key);
```

函数调用时，将对象 A 传递给参数 T * Arr。

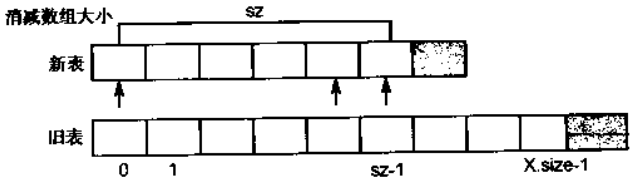
定义： ExchangeSort(T * arr, int n)

调用： ExchangeSort(A,A.size())

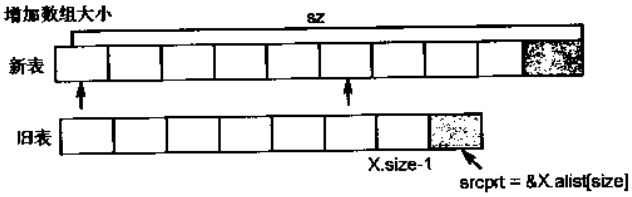
这就需要用类型转换运算符将 alist 的指针赋值给 arr，使变量 arr 指向对象 A 中的数组。

```
// 指针转换运算符
template <class T>
Array <T>::operator T* (void) const
{
    // 返回当前对象中私有数组的首地址
    return alist;
}
```

调整数组大小 类 Array 提供了 ListSize 函数来取得数组中当前元素的个数。与动态数据关系更密切的是 Resize 函数，它可改变对象中数组元素的个数：如果调整后的数组大小 sz 与 size 相同，则函数只需简单返回；否则，则申请一块大小为 sz 个元素的空间。若调整后数组元素少了(sz < size)，则拷贝前 sz 个元素到新数组中。



如果数组调大了(sz > size)，则将所有旧元素拷贝到新表中，且新表中还有可用的空间，最后，将旧数组的空间释放掉。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 调整数组大小运算符
template <class T>
void Array<T>::Resize(int sz)
{
    // 检查新的大小参数;若其小于等于0,则退出程序
    if (sz <= 0)
       Error(invalidArraySize);
    // 若大小不变,则简单返回
    if (sz == size)
       return;
    // 需申请新的内存;确认系统已分配所需内存
    T* newlist = new T[sz];
    if (newlist == NULL)
       Error(memoryAllocationError);
    // n为需拷贝元素的个数
    int n = (sz <= size) ? sz : size;
    // 从旧表中拷贝n个数组元素到新表
    T* srcptr = alist;          // alist 首地址
    T* destptr = newlist;       // newlist 首地址
    while (n--)                 // 拷贝表元素
      *destptr++ = *srcptr++;
    // 删除旧表
    delete[] alist;
    // 将 alist 指针指向 newlist 并改变大小值
    alist = newlist;
    size = sz;
}
```

类 Array 的使用

类 Array 的实现提供了许多新的思路,用户可用类 Array 来代替语言本身的数组,充分利用其安全特性及因大小调整带来的灵活性。

程序 8.3　调整数组大小

设 Array 对象 A 定义了一个 10 个整数元素的表,用来存放质数。

定义:质数是一大于等于 2 的整数,它只能被其本身和 1 整除。

本程序用来求在范围 2~N 中的质数,其中 N 为用户给出的上限由于我们不能预先知道需多大的数组来存放数据,程序用当前质数个数(primecount)和数组大小进行比较来检查"表满"状态,当表满时,我们调整数组大小,给它增加 10 个元素。程序最后以 10 个质数 1 行的格式输出这些质数。

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
#include "array.h"
void main(void)
{
    // 用来存放质数的数组,开始有 10 个元素
    Array < int >  A(10);
    // 用户给出决定质数范围的上限
    int     upperlimit, primecount = 0, i, j;
    cout << "Enter value >= 2 as upper limit for prime numbers:";
    cin >> upperlimit;
    A[primecount++] = 2;    // 2 为质数
    for (i = 3; i < upperlimit; i++)
    {
        // 若质数表满,则再申请 10 个元素
        if (primecount == A.ListSize())
          A.Resize(primecount + 10);
        // 大于 2 的偶数均非质数,直接跳到下次循环
        if (i % 2 == 0)
          continue;
        // 检查参数因子 3, 5, 7, …, i/2
        j = 3;
        while(j <= i/2 && i % j != 0)
          j += 2;
        // 若上述参数不为 i 的因子,则 i 为质数
        if (j > i/2)
          A[primecount++] = i;
    }
    for (i = ;i < primecount; i++)
    {
        cout << setw(5) << A[i];
        // 每 10 个质数回车一次
        if ((i+1) % 10 == 0)
          cout << endl;
    }
    cout << endl;
}
/*
<程序 8.3 运行结果>
Enter a value >= 2 as the upper limit for prime numbers: 100
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97
*/
```

8.5 串类

字符串是许多非数值计算算法处理的主要对象,在模式匹配、程序编译及数据处理等

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

领域中均有应用。鉴于此,设计一个串数据类型将各种串运算封装进去,并留有扩展的余地是十分必要的。C++就用以 null 结束的字符数组实现了串类型,而且,C++所有系统都在 <string.h> 中提供了库函数来支持对串的处理。在有经验的程序员手中,这些函数是实现高效的串算法的有力工具,尽管如此,对许多应用来说,这些函数的技术性太强,使用起来不太方便。

一些语言,如 BASIC,定义了处理串的运算符。BASIC 的串变量是以 $ 结束,并支持"="运算符表示的赋值运算和"<"运算符表示的比较运算等,它的串类型就是语言本身定义的一部分:

```
NAME$  = "JOE"                          // 赋值
IF NAME$  < STUDENTSPRES$  THEN …       // 比较
```

对这些语言,编译器提供了许多增强串处理能力的扩展,许多 C++的程序员也希望用这些扩展来方便对串的访问。

本节我们给出一个类 String,它定义了串类型并提供许多功能强大的串函数,类 String 的对象用动态内存存放长度可变的串,用运算符重载来产生串表达式。它不但给用户定义串类型提供了新的选择,同时也提供与 C++定义的 C++串的充分交互。类 String 将在后续章节中用到,读者可充分了解其简洁性,例如比较两个串 S 和 T,然后将较小的串赋值给变量 firststring。下面是分别用 C++的库函数和类 String 实现的语句。

```
用 C++库函数实现                用类 String 实现

if (strcmp (S,T) < 0)           firststring = (S < T) ? S:T;
   strcpy (firststring,S);
else
   strcpy (firststring,T);
```

本节我们给出类 String 的完整的声明,并实现其中的一些函数,最后还提供测试程序对其进行测试。用户可在附录程序的文件"trclass.h"中找到类 String 的完整实现。8.6 节用类 String 实现了模式匹配算法。

类 String 的定义

声明

```
#include <string.h>
#include <stdlib.h>

const int outOfMemory = 0, indexError = 1;

class String
{
    private:
        // 指向动态申请的串的指针
        // 串长度包括 NULL 字符
        char *str;
        int size;

        // 错误报告函数
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    void Error(int errorType, int badIndex = 0) const;
public:
    // 构造函数
    String(char * s = "");
    String(const String& s);
    // 析构函数
    ~String(void);
    // 赋值运算符
    // String = String, String = C++String
    String& operator = (const String& s);
    String& operator = (char * s);
    // 关系运算符
    // String = = String, String = = C++String, C++String = = String
    int operator = = (const String& s) const;
    int operator = = (char * s) const;
    friend int operator = = (char * str, const String& s);
    // String! = String, String! = C++String, C++String! = String
    int operator! = (const String& s)const;
    int operator! = (char * s) const;
    friend int operator! = (char * str, const String& s);
    // String < String, String < C++String, C++String < String
    int operator < (const String& s) const;
    int operator < (char * s) const;
    friend int operator < (char * str, const String& s);
    // String < = String, String < = C++String, C++ String < = String
    int operator < = (const String& s) const;
    int operator < = (char * s) const;
    friend int operator < = (char * str, const String& s);
    // String > String, String > C++String, C++String > String
    int operator > (const String& s) const;
    int operator > (char * s) const;
    friend int operator > (char * str, const String& s);
    // String > = String, String > = C++String, C++String > = String
    int operator > = (const String& s) const;
    int operator > = (char * s) const;
    friend int operator > = (char * str, const String& s);
    // 串拼接运算符
    // String + String, String + C++String, C++String + String
    // String + = String, String + = C++String
    String operator + (const String& s) const;
    String operator + (char * s) const;
    friend String operator + (char * str, const String& s);
    void operator + = (const String& s);
    void operator + = (char * s);
    // 有关串函数
    // 从 Start 位置开始找字符 c
    int Find(char c, int start) const;
    // 找字符 c 最后一次出现的位置
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        int FindLast(char c) const;
        // 取子串
        String Substr(int index, int count) const;
        // 往 String 中插入另一个 String
        void Insert(const String& s, int index);
        // 插入一个 C++ 串
        void Insert(char *s, int index);
        // 删除子串
        void Remove(int index, int count);

        // String 的下标运算
        char& operator[](int n);

        // 将 String 转换为 C++ 串
        operator char* (void) const;

        // String 的输入/输出
        friend ostream& operator<<(ostream& ostr,
                                   const String& s);
        friend istream& operator>>(istream& istr,
                                   String& s);
        // 读入字符直到结束符
        int ReadString(istream& is=cin,
                       char delimiter='\n');

        // 其它函数
        Length(void) const;
        int IsEmpty(void) const;
        void Clear(void);
};
```

说明

string 的对象可和 C++ 串(char *)共同使用,操作符的两个参数类型可为 String 对象和 C++ 串,比如定义中用"+"运算符定义了三个各不相同的元素来实现串之间的连接。

```
// String + String
String operator + (const String& s);
// String + C++ String
String operator + (char *s);
// C++ String + String
friend string operator + (char *str, const string& s);
```

类 String 中有构造函数,复制构造函数及两个重载的赋值运算符,可供用户将 String 对象或 C++ 串赋值给新的 String 对象:

```
String S("Hello"), T = S, R;      // T = "Hello", R 为 NULL
R = "World!";
```

该类实现了多种串连接的运算,包括将串连接到当前串之后的运算符"+="。

```
R = T + "world!"      // R = "Hello world!"
R += "    ";
R += S;               // R = "Hello world! Hello"
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

用 ASCII 顺序定义了许多关系运算符来比较两个串：

```
String U("Smith"), V("Smithsonian"), W("Thomas");

if (U > = v)...            // FALSE
if (W = = "Thomas")...     // TRUE
if ("Tom" ! = W)...        // TRUE
```

提供了多个十分有用的串操作，包括寻找特定字符、抽取子串、将一个串插入另一个当中，及删除一个子串，并对串中的每个字符提供下标访问机制，就像一个字符数组一样。

```
int sindex;
String V ("Smithsonian");
// 从下标 0 开始找's'
sindex = V.Find('s',0);
R = V.Substr(sindex,3);        // R = "son"
V.Remove(sindex,6);            // V = "Smith"
R[0] = 'S';                    // R = "Son"
R.Insert("ilvert",1);          // R = "Silverton"
```

输入运算符“> >”用空格隔开输入的串：

```
cin > > S > > T > > R;
< 输入 > Seperate by blanks
S = "Seperate"     T = "by"     R = "blanks"
```

函数 ReadString 读入字符串，直到结束符，并用 NULL 代替结束符：

```
R.ReadString(cin);
< 输入 > The fox Leaped over the big brown dog < newline >
    R = "The fox Leaped over the big brown dog"
```

程序 8.4　类 String 的使用

本程序给出类 String 中某些算法的用法。每一运算之后都用输出语句来描述功能。

```
# include < iostream.h >
# include "strclass.h"
# define TF(b)     ((b) ? "TRUE" : "FALSE")
void main(void)
{
    String s1("STRING "), s2("CLASS");
    String s3;
    int loc, i;
    char c, cstr[30];
    s3 = s1 + s2;
    cout < < s1 < < "concatenated with " < < s2 < < " = "
         < < s3 < < endl;
    cout < < "Length of " < < s2 < < " = " < < s2.Length() < < endl;
    cout < < "The first occurrence of 'S' in " < < s2 < < " =  "
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        << s2.Find('S',0) << endl;
    cout << "The last occurrence of 'S' in " << s2 << " is "
        s2.FindLast('S') << endl;
    cout << "Insert "OBJECT 'into s3 at position 7."<< endl;
    << s3.Insert("OBJECT ",7); \ lcout
cout << s3 << endl;
    s1 = "FILE1.S";
    for (i = 0; i < s1.Length(); i++)
    {
        c = s1[i];
        if (c >= 'A; && c <= 'Z')
        {
            c += 32;    // 将 c 转换为小写字符
            s1[i] = c;
        }
    }
    cout << "The string 'FILE1.S' converted to lower case is ";
    cout << s1 << endl;
    cout << "Test relational operators with strings ";
    cout << "s1 = 'ABCDE' s2 = 'BCF'" << endl;
    s1 = "ABCDE";
    s2 = "BCF";
    cout << "s1 < s2 is " << TF(s1 < s2) << endl;
    cout << "s1 == s2 is " << TF(s1 == s2) << endl;
    cout << "Use 'operator char* ()' to extract s1"
        << " to a C++ string: ";
    strcpy(cstr, s1);
    cout << cstr << endl;
}
/*
<程序 8.4 运行结果>
STRING concatenated with CLASS = STRING CLASS
Length of CLASS = 5
The first occurrence of 'S; in CLASS = 3
The last occurrence of 'S' in CLASS is 4
Insert 'OBJECT ' into s3 at position7.
STRING OBJECT CLASS
The string 'FILE1.S' converted to lower case is file1.s
Test relational operators with strings s1 = 'ABCDE' s2 = 'BCF'
s1 < s2 is TRUE
s1 == s2 is FALSE
Use 'operator char* ()' to extract s1 to a C++ string: ABCDE
*/
```

类 String 的实现

下面我们给出类 String 的实现，其数据成员为 str 和 size，其中指针变量 str 存放以 NULL 结束的字符串的首地址，size 的值为字符串长度加 1，多出来的一个字节用来存放 NULL 字符，因此，size 就是用于存放字符串的实际字节数，若某一操作改变了字符串大小，则申请一块新的内存来存放修改后的字符串，并释放旧的内存块。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类 String 中的 str 域就是 C++串的首地址，附加的 size 域用于 String 变量(对象)提供的丰富的成员函数，正是这些函数带来了它与 C++串变量的区别。

构造函数和析构函数 构造函数创建一个存放由参数传递来的 C++串的 String 对象，在初始化期间，它应给 size 赋值，申请动态内存并将 C++串拷贝到由数据成员 str 指向的动态内存中，若没有参数，则拷贝一个空串。复制构造函数完成相同的功能，只是从另一个 String 对象中拷贝串，而不是拷贝 C++的串。析构函数释放存放串的动态内存。

```
// 构造函数。申请内存并拷入一个 C++串
String::String(char *s)
{
// 长度中包括 NULL 字符
size = strlen(s) + 1;
// 为串及 NULL 字符申请空间并将 s 拷入
str = new char [size];
// 若申请失败，则退出程序
if (str == NULL)
   Error(outOfMemory);
strcpy(str,s);
}
```

重载赋值运算符 赋值运算符可将 String 对象或 C++串赋值给 String 对象。下面的语句都是合法的：

```
String S("I am a String variable"), T;
// 将 String 赋给 String
T = S;
// 将 C++串赋给 String
T = "I am a C++ String";
```

要将一个 String 对象 s 赋值给当前对象，先比较两个串的长度，若长度不同，则释放当前对象的动态内存，并重新申请 s.size 个字节。然后再将 s.str 拷贝到新内存中。

```
// 赋值运算符。从 String 到 String
String& String::operator= (const String& s)
{
    // 若大小不符，则删除当前串，并重新申请内存
    if (s.size != size)
    {
        delete [] str;
        str = new char [s.size];
        if (str == NULL)
            Error(outOfMemory);
        // 将其 size 值赋为 s 的 size 值
        size = s.size;
    }
    // 拷贝 s.str 到新串中，并返回其指针
    strcpy(str,s.str);
    return *this;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

关系运算符 类 String 提供完整的关系运算集，按 ASCII 顺序进行串比较。这些运算符可比较两个 String 对象或 String 对象和 C++串。

```
String S("Cat"), T("Dog");
    // 串比较
if (S == T) . . .          // 条件为 FALSE
if (T < "Tiger") . . .     // 条件为 TRUE
if ("Aardvark" >= T) . . // 条件为 FALSE
```

关系运算符"=="可用来测试 String 对象和 C++串 s 是否相等，注意 C++串变量在"=="左边的函数必须作为友元函数重载。

```
// C++String == String. 友元函数
// 因为 C++串在左边
int operator== (char *str, const String& s)
{
    return strcmp(str, s.str) == 0;
}
```

串运算符 类 String 中定义了用来连接串的函数集，它是通过重载"+"和"+="运算符来实现的，前一种情况，连接函数返回一个新串；后一种情况将串连接到当前串之后，例如，串"cool water"可用下面三种连接运算符产生：

```
string S("cool"), T("water"), U, V;
U = S + T;          // 连接两个 String
V = S + "water"     // 连接一个 String 和一个 C++串
S += T;             // 此时 S 为"Cool Water"
```

下面的代码可实现连接运算符的"String 对象 + String 对象"版本，该函数返回一个 String 对象为当前 String 对象和右边 String 对象的连接。算法中，我们先创建了一个 String 对象 temp，它用来存放包括 NULL 字符在内的 size + s.size - 1 个字符。注意当声明 temp 时，我们就释放了由构造函数产生的 NULL 串。该函数从当前对象拷贝字符串到新串中，并将 s 中的字符串连接结在新串之后，这时，temp 对象即为返回值。

```
// 连接: String + String
String String::operator+ (const String& s) const
{
    // 在 temp 中建立一个长度为 len 的新串
    String temp;
    int len;
    // 删除定义 temp 时产生的 NULL 串
    delete [] temp.str;
    // 计算拼接后的串长度并为之申请内存
    len = size + s.size - 1;    // 只有一个 NULL 结尾
    temp.str = new char [len];
    if (temp.str == NULL)
       Error(outOfMemory);
    // 建立新串
    temp.size = len;
    strcpy(temp.str,str);          // 拷贝 str 到 temp
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    strcat(temp.str, s.str);        // 连接 s.str
    return temp;                    // 返回
}
```

String 函数 函数 Substr 返回一个当前串的子串，该串从 index 位置开始，共有 count 个字符：

```
String Substr (int index, int count);
```

Substr 在模式匹配算法中经常用到。例如，

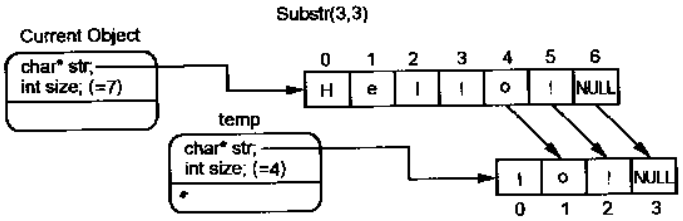
```
String S("cool water"), u; u = S.Substr(1,2)     // 从 Cool 中取出'oo'。
```

若 index 超出了串的最后一个字符，则函数返回一个带空串的对象。串中从 index 到串尾的字符个数为(size - index - 1)，若 count 超过这个值，则用 size - index - 1 代替 count，即将串尾部分作为子串。实现这个函数时，先为对象 object 申请 count + 1 个字符的内存空间，从 string 对象的串的 index 位置开始拷贝 count 个字符到 temp 的新申请的空间，再用 NULL 结束整个字符串，将 temp 的 size 赋值为 count + 1，然后将 temp 作为函数值返回。

```
// 返回从 index 开始共 count 个字符的子串
String String::Substr(int index, int count) const
{
    // 从 index 到串尾的字符个数
    int charsLeft = size - index - 1,i;

    // 建立子串 temp
    String temp;
    char *p, *q;

    // 若 index 越界，返回空串
    if (index >= size - 1)
       return temp;

    // 若 count > 剩下的字符，则只用剩下的字符
    if (count > charsLeft)
       count = charsLeft;

    // 删除定义 temp 时产生的空串
    delete [] temp.str;

    // 为子串申请动态内存
    temp.str = new char [count + 1];
    if (temp.str == NULL)
       Error(outOfMemory);

    // 从 str 中拷贝 count 个字符到 temp.str
    for (i = 0, p = temp.str, q = &str[index];i < count; i++)
         *p++ = *q++;
    // 用 NULL 结束该串
    *p = 0;

    temp.size = count + 1;
    return temp;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



成员函数 Find 和 FindLast 检查字符是否在串中出现，若没有发现，它们均返回 -1。函数

```
int Find(char c, int start) const;
```

从 start 位置开始查找字符 c 第一次出现时的位置，若找到该字符，Find 返回其在串中的位置，函数

```
int FindLast (char c) const;
```

查找 c 最后出现的位置，若找到，则返回该值。

```
// 返回字符 c 在串中的最后位置
int String::FindLast(char c) const
{
    int ret;
    char *p;
    // 用 C++ 库函数 strrchr.返回该字符在串中最后位置的指针
    p = strrchr(str,c);
    if (p ! = NULL)
       ret = int(p-str);     // 计算下标值
    else
       ret = -1;          // 失败时返回 -1
    return ret;
}
```

String 的输入/输出 我们用 C++ 串的流输入和输出操作实现了 String 对象的运算符“>>”和“<<”，其中“>>”读入以空格隔开的单词。

函数 ReadString 从文本文件中读入一行文本(最多 255 个字符或指定的结束符)，将其整个赋给一个 String 对象。若没有指定文件参数，则缺省文件为 cin，整个输入在遇到结束符时中止，但结束符并不读到串中，结束符的缺省值为换行('\n')，ReadString 返回读入字符的个数，若到文件尾，则返回 -1，例如：

```
String S, T;

cin >> S;            // 跳过行首的空格
T.ReadString();      // 读入整行
cout << "The components are:" << S << " and" << T << endl;

<输入>
Super ! Grade A      // ! 号后面的空格为 T 的一部分
<输出>
The components are : Super! and Grade A
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

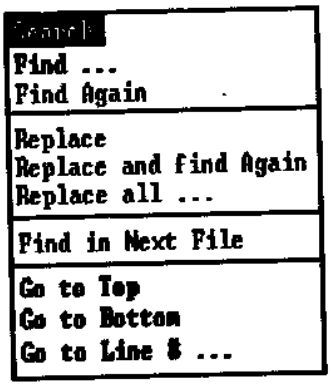
www.Linuxidc.com

用 getline 从输入流 istr 中读入字符到字符数组中，直到结束符或超过 255 个字符。若 getline 遇到文件结束符，则返回 -1；否则，释放现有的动态数组 str 并重新申请新一块内存，大小 size = strlen(temp) + 1，将 temp 拷贝到新数组中，并返回读入的字符个数(size - 1)

```
// 从流 istr 中读入一行文本
int String::ReadString (istream& istr, char delimiter)
{
    // 读入一行到 tmp 中
    char tmp[256];
    // 文件未结束，读入最多 255 个字符的一行
    if (istr.getline(tmp, 256, delimiter))
    {
        // 删除旧串并申请一个新串
        delete [] str;
        size = strlen(tmp) + 1;
        str = new char [size];
        if (str = = NULL)
           Error(outOfMemory);
        // 拷贝 tmp,返回读入字符个数
        strcpy(str, tmp)
        return size - 1;
    }
    else
        return -1;          // 文件结束时返回 -1
}
```

8.6 模式匹配

一个最普通的模式匹配问题就是在文本文件中查找串。大多数编辑程序都有“搜索”(Search)菜单项，菜单项下有诸如“查找(Find)”、“替换(Replace)”及“全部替换”(Replace All)等匹配内容，如下图所示：



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Find 从文件的当前位置开始向前或向后查找模式的下一次出现，Replace 将 Find 匹配上的模式换成另一个串。Replace All 搜索整个文件并用新串替换所有匹配上的模式。

查找(Find)过程

Find 问题可简单描述如下：给定两个字符串变量 S 和 P，从 S 中的给定位置开始搜索模式 P，若找到，返回该模式第一个字符在 S 中的下标，若 S 中没有 P，则返回 -1. 例如：

1. 给定串 S = "aaacabc" 及 P = "abc"，模式 P 从 S 的第 4 个位置开始。
2. 若 S = "Blue Bar ranch lies outside the city of the animals"，P = "the"，则 P 分别在 S 的 28 和 40 两个位置出现两次。
3. 模式 P = "aca" 没有在串 "acbaccacbcbcac" 中出现。

程序 8.5 给出了一个用类 String 实现的模式匹配算法。

模式匹配算法

该算法由函数 FindPat 实现，它从 S 的 startindex 位置开始查找模式 P 的第一次出现，我们先给出函数的代码，因为算法的分析涉及到函数中的特定变量。

```
int FindPat(String S, String P, int startindex)
{
    // 模式的首字符和尾字符及模式长度
    char    patStartChar, patEndChar;
    int     patLength;
    // 模式尾字符的下标
    int     patInc;
    // searchIndex 为开始查找模式的首字符的起始位置。matchStart 为首字符匹配上的
    // 下标,matchEnd 为此时模式尾字符下标
    int     searchIndex, matchStart, matchEnd;
    // S 中尾字符下标。matchEnd 小于等于此值
    int     lastStrIndex;
    // insidePattern 中存放去掉首、尾字符后的模式
    String insidePattern;
    patStartChar = P[0]          // 模式的首字符
    praLength = P.Length();      // 模式长度
    patInc = patLength - 1;      // 模式尾字符下标
    patEndChar = P[patInc];      // 模式的尾字符
    // 若模式长度 > 2,从中取到去掉首尾字符后的模式
    if (patLength > 2)
        insidePattern = P.Substr(1, patLength - 2);
    lastStrIndex = S.Length() - 1;    // S 中最后一个字符的下标
    // 从 Start index 开始找第一个匹配字符
    searchIndex = startindex;
    // 开始匹配第一个字符
    matchStart = S.Find(patStartChar, searchIndex);
    // 可能匹配的尾字符的下标
    matchEnd = matchStart + patInc;
    // 重复此过程,匹配上首字符后,判尾字符是否相同
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    while(matchStart ! = -1 && matchEnd < = lastStrIndex)
    {
        // 首尾字符均匹配吗?
            if (S[matchEnd] = = patEndChar)
            {
                // 若模式字符不超过两个,则找到匹配串
                if (patLength < = 2)
                   return matchStart;
                // 比较其它字符
                if (S. Substr (matchStart + 1, patLength - 2) = =
                            insidePattern)
                return matchStart;
            }
                // 没有找到模式,从下一字符继续匹配
                searchIndex = matchStart + 1;
                matchStart = S.Find(patStartChar, searchIndex);
                matchEnd = matchStart + patInc;
    }
    return -1;          // 没有发现模式
}
```

整个算法可归纳为下述步骤,我们以串

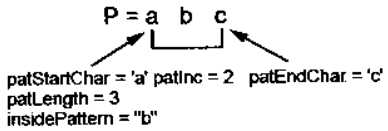
S=b a d c a b c a b d a b c

和模式

P=a b c

为例来说明。

1. 模式可用下述文本块描述:起始字符 patStartChar = P[0],长度 patLength = P.Length( ),增量 patInc = patLength - 1,它给出了模式的最后一个字符的下标,即模式的结束字符为 PatEndChar = P[patInc]。如我们将在步骤 3 和步骤 4 中看到的,算法取出 S 中长度为 PatLength 的文本块的第一个字符和最后一个字符,分别与 P 的第一个字符和最后一个字符比较,若 P 的长度超过 2,我们从 P 中抽取不含第一个和最后一个字符的子串并将此串赋给变量 insidePattern,即 insidePattern = P.Substr(1,PatLength - 2)。



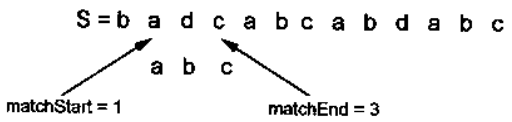
2. 为标记串 S 的结束位置,将其最后一个字符的下标赋给 lastStrIndex,即 lastStrIndex = (S.Length( ) - 1),搜索的起始位置(startIndex)可以为 0(从 S 的首字符开始),若希望从 S 串的中间位置开始,则 startIndex 为某个正整数,将变量 SearchIndex 的值初始化为 startIndex,它可用作匹配模式中第一个字符定位点。
3. 从 searchIndex 开始,用类 String 的函数 Find 查找 patStartChar 在串中第一次出现的位置,将匹配上的下标赋给变量 matchStart,变量 matchEnd = (matchStart + patInc),

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

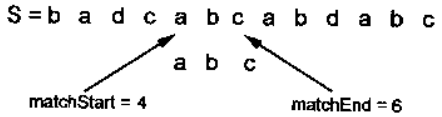
www.Linuxidc.com

应该是 patEndChar 在字符串 S 中的下标，若 Find 没有找到 patStartChar 或找到 patStartChar 后得到的 matchInd 超出 lastStrIndex，则程序以失败结束。

4. 用 patEndChar 和 S 中文本块的最后一个字符比较(S[matchEnd] = = patEndChar)，若这两个字符不等，则转步骤 5 进行下一次比较。用最后一个字符进行比较是对算法的优化，它使我们减少一些不必要的尝试。若模式的长度为 1 或 2(pathLength < = 2)，则匹配成功，返回下标 matchStart，否则，将文本块中除首尾字符外的字符串(S.Substr(matchStart + 1, patLength - 2))与串 insidePattern 比较。若它们匹配，返回 matchStart，在例中，matchStart = 1 且 matchEnd = 3，其最后一个字符也匹配上了，但串 insidePattern = "b"而 S.Substr(2,1) = "d"，匹配失败。



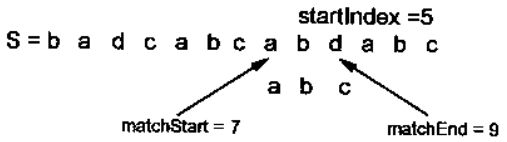
5. 从下标 searchIndex = matchStart + 1 开始重复步骤 3 和步骤 4，在例中，首字符的下一个匹配位置的下标为 4。



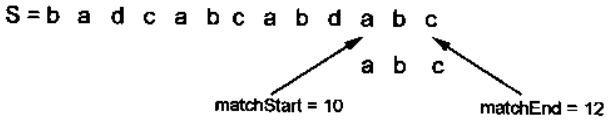
此时，P 的尾字符和文本块的最后一个字符匹配，且 S.Substr(5,1)和 insidePattern 也相同，将返回下标 4(matchStart)。

6. 为查找模式在串中多次出现的情况，可用起始位置为 FindPat 返回的下标加 1 再次调用该函数，例如，在 S 中继续查找"abc"将产生如下结果：

·当下标为 7 时，可能匹配，但最后失败。



·当下标为 10 时，可能匹配，最后成功匹配。



程序 8.5　模式匹配

函数 FindPat 实现了上述模式匹配算法，它存放在文件"FindPat.h"中，本程序读入一个 String 对象作为模式，然后读入 String 对象 linestr 直到文件尾，对每一行都调用 FindPat

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

函数来搜索模式出现的次数，然后，将出现的次数和行号一起打印输出。

```
#include <iostream.h>
#include "strclass.h"
#include "findpat.h"
void main()
{
    // 定义模式串及被查找的串
    String pattern, lineStr;
    // 设定查找参数
    int     lineno = 0, lenLineStr, startSearch, patIndex;
    // 在此串中匹配次数
    int numberOfMatches;
    cout << "Enter the pattern to search for: ";
    pattern.ReadString();
    cout << "Enter a line or EOF: ";
    while(lineStr.ReadString() ! = -1)
    {
        // 记录行号，并设定查找参数
        lineno++;
        lenLineStr = lineStr.Length();
        startSearch = 0;
        numberOfMatches = 0;
        // 在本行内查找模式
        while(startSearch <= lenLineStr-1 &&
        patIndex = FindPat(lineStr, pattern, startSearch)) ! = -1)
        {
            numberOfMatches++;
            // 继续查找下一次匹配的情况
            startSearch = patIndex+1;
        }
        cout << numberOfMatches << " matches on line "
             << lineno << endl;
        cout << "Enter a line or EOF: ";
    }
}
/*
<程序 8.5 运行结果>>
Enter the pattern to search for: iss
Enter a line or EOF:
Alfred the snake hissed because he missed his Missy.
3 matches on line 1
Enter a line or EOF:
Mississippi
2 matches on line 2
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

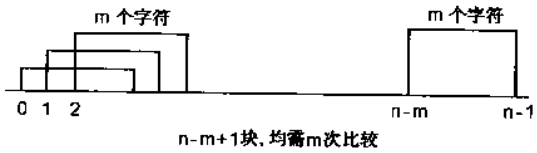
```
Enter a line or EOF:
He blissfully walked down the sunny lane.
1 matches on line 3
Enter a line or EOF:
It is so.
0 matches on line 4
Enter a line or EOF:
*/
```

模式匹配算法分析

设模式有 m 个字符，而串 S 有 n 个字符。若 S 的前 m 个字符与 P 匹配，我们可在 m 次比较后找到匹配结果，因此，最好情况下算法的复杂度为 O(m)。算法的最差情况可估计如下：假设我们不进行比较最后一个字符之类的优化，且第一个字符总是能匹配上，但却永远没有匹配上的模式，如下面这种情况：

p = "abc"且 S = "aaaaaaaa"(m = 3, n = 8)

此时，模式"abc"的 m(m = 3)个字符必须和 S 中的文本块一共比较 n − m + 1 = 6 次，一般情况下，我们必须比较 m 个字符共(n − m + 1)次，即一共进行 m(n − m + 1)次比较。



由于

$$m(n-m+1) \leqslant m(n-m+m) = mn,$$

所以算法在最差情况下的复杂度估计值为 O(mn)。

模式匹配在计算机科学中是一重要的方向，在文学研究中也十分有用。Knuth-Morris-Pratt 模式匹配算法(Knuth, 1977)的复杂度为 O(m + n)，优于我们上面介绍的简单算法。

8.7 整型集合

集合是从一个称为全集的群体中挑选出的一组对象。集合可用大括号内的用逗号隔开的表来表示：

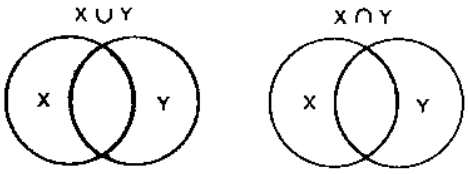
$$X = \{I_1, I_2, I_3, \cdots, I_m\}$$

- 并集(∪)　X∪Y 是一集合，它包括所有集合 X 和集合 Y 的元素(重复只计一次)。
- 交集(∩)　X∩Y 是一集合，它包括所有同时在集合 X 和集合 Y 中出现元素。
- 属于(∈)　当且仅当 n 为集合 X 中的一个元素，元素 n 属于集合 X(n∈X)。

X = {0,3,20,55}　　// 20∈X 为 TRUE，35∈X 为 FALSE。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



X = {0,3,20,55}, Y = {4,20,45,55}
X ∪ Y = {0,3,4,20,45,55}　　X ∩ Y = {20,55}

整型集合

本节我们定义整型为所有用整数数值表示的数据类型，包括字符类型，各种大小的整数类型及枚举类型等，例如，整个 ASCII 字符集对应于范围从 0 到 127 的单字节整数，当程序用'A'，'B'，…表示字符时，其内部存储的是整数 65，66 等，程序员可灵活使用这两种表示方式：

```
char ch1 = 'A',ch2 = 97,ch3;
ch3 = ch1 + 4          //CH3 = 'E'
cout < < ch2 < < " " < < int ('A') ;    // 输出:a  65
```

本节我们定义元素类型为整型的集合，其元素为范围从 0 到 setrange－1 的所有无符号整数，其中 setrange 为集合中元素的个数，如下述集合：

数字集合 = {0,1,2,3,4,5,6,7,8,9}
　对应范围为 0～9。

ASCII 字符集 = {…,'A','B',…,'Z',…}
　对应范围为 0～127。

enum Color(read,white,blue)
Color 集合 = {0,1,2}
　对应范围为 0～2

我们可通过维护一个二值(0 和 1)数值来实现集合类型，若元素 i 在集合中，则数组中第 i 个位置的值为 1(TRUE)，否则其值为 0(FALSE)，我们在书面作业 6.13 中讨论过用静态数组实现的整型集合。这样，我们对集合中每个可能的元素都要保留一个整数值。集合也可用动态数组实现如下：

```
// 定义元素范围为 0～setrange－1 的集合
class Set
{
    private:
        int *member;        // 指向集合数组的指针
        int setrange;
        . . .
    public:
        // 申请集合数组的构造函数
        Set(int n): setrange (n)
        {
            member = new int [setrange];
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        ...
        }
};
    ...
Set S(20);          // 集合{0,...,19}
n ∈ S ==> S.member[n] == 1
```

如果我们用 C++ 的位运算符来实现集合，则可大大地减少对内存的需求，而且通过模板还可使集合适用于所有整型类型。

C++ 位运算符

在逻辑表达式中，运算符或（"||"），与（"&&"）及非（"!"）用来连接整型操作数并返回结果 TRUE(1)或 FALSE(0)，同样，对于整型操作数的每一单独的位，我们也可定义类似的算术运算符来求值，位运算符包括或（"|"），与（"&"），非（"~"）和异或（"^"），它们对整型操作数的各位进行计算，并返回结果 0 或 1。表 8.1 给出了这些运算符的运算规则。也许大家不太熟悉异或运算符，它在两个操作数不等时返回值 1。

表 8.1　位运算符

x	y	~x	x|y	x&y	x^y
0	0	1	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	1	0	1	1	0

对于 n 位的整数，位运算符对每个对应位进行计算，设

$a = a_{n-1}a_{n-2}\cdots a_2a_1a_0 \qquad b = b_{n-1}b_{n-2}\cdots b_2b_1b_0$

c = a op b 结果如下：

$c = c_{n-1}c_{n-2}\cdots c_2c_1c_0 = a_{n-1}a_{n-2}\cdots a_2a_1a_0 \text{ op } b_{n-1}b_{n-2}\cdots b_2b_1b_0$

其中 $c_i = a_i \text{ op } b_i$，$0 \leqslant i \leqslant n-1$，op = '|'，'&'，'^'。

单目运算符'~'将操作数的各位求反。

例 8.4

设 8 位数 X = 11100011 及 Y = 01110110，分别计算

```
a. x   11100011     b. x   11100011     c. x   11100011
   | y 01110110        & y 01110110        ^ y 01110110
     --------           --------            --------
     11110111           01100010            10010101
```

d. ~x = 00011100

另外，C++ 也提供移位运算符，将整数的各位左移（"<<"）或右移（">>"）。实际上，表达式 a<<n 即为将 a 乘以 2^n，而 a>>n 则为将 n 将除以 2^n。一般来说，对于乘（或除）2 的幂来说，用移位运算要快得多。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

位操作一般只用于无符号整数，本书中如不特别说明，则位操作的操作数均为无符号整数。

例 8.5

变量 x,y 和 z 定义如下：

```
// 每个变量均为 16 位
unsigned short x = 10, y = 13, z;
```

下述位运算表达式及其结果分别为：

```
a. z = x|y;              // z = 15
b. z = x & y;            // z = 8
c. z = ~0 < < 2;         // z = 65532
d. z = ~0 & (y > > 2);   // z = 1
```

类 Set 的定义

类 Set 的对象包含一个从范围为 0 到 setrange - 1 中选择出来的整型元素组成的表，整型元素的具体数据类型由模板类型 G 给出。我们假定类型 T 中已定义了到 int 的转换，可以显式地将整数转换为类型 T 的值。例如，设 val 为类型 T 的元素，I 为整数变量。

```
T val;
int I;
```

若 I 与元素 val 等值，则有

```
I = int(val) 且 val = T(I)
```

例 8.6

a. char 为整型类型

```
char C = 'A';
int i;

i = int(c);      // i = 65
C = char(i);     // C = 'A'
```

b. 枚举类型为整型类型。

```
enum Days {Sun,Mon,Tues,Wed,Thurs,Fri,Sat};

Days day = Thurs;
int d;

d = int (day);     // d = 4
day = days(d);     // day = Thurs
```

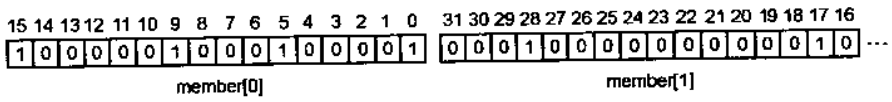
集合内元素的表示

用上 C + + 的位运算符后，可将类 Set 的对象的每一位都有效地利用起来。集合中的元素范围为 0 到 setrange - 1，可存放于一个由 16 位的无符号数组成的动态数组 number

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

中，这个数组将范围从 0 到 setrange－1 的整数串成一个位链。用它的每一位表示一个元素，且用该位的值为 1 来表示该元素在数组中。如集合元素 0 用第一个数组元素的最低位来表示，15 用第一个数组元素的最高位来表示，再用第二个数组元素的最低位表示 16，直到 setrange－1，可用下图来表示集合在内存中的映象



上图表示整数 0，5，9，15，17 和 28 在集合中。

声明

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

enum ErrorType
{
    InvalidMember, ExpectRightBrace, MissingValue,
    MissingComma, InvalidChar, MissingLeftBrace,
    InvalidInputData, EndOfFile, OutOfMemory,
    InvalidMemberRef, SetsDifferentSize
};

template <class T>
class Set
{
    private:
        // 集合中元素个数的最大值
        int setrange;

        // 位数组的字节数及数组指针
        int arraysize;
        unsigned short *member;

        // 出错处理函数
        void Error(ErrorType n) const;

        // 实现集合元素在 16 位短整数中的映象
        int ArrayIndex(const T& elt) const;
        unsigned short BitMask(const T& elt)const;

    public:
        // 构造函数，创建空集
        Set(int setrange);

            // 复制构造函数
        Set(const Set<T>& x);

        // 析构函数
        ~Set(void);

        // 赋值运算符
        Set<T>& operator= (const Set<T>& rhs);

        // 属于运算符
        int IsMember(const T& elt);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 集合相等
        int operator = = (const Set < T > & x) const;
        // 并集
        Set operator + (const Set < T > & x) const;
        // 交集
        Set operator * (const Set < T > & x) const;
        // 加入/删除元素
        void Insert(const T& elt);
        void Delete(const T& elt);
        // 集合输入/输出
        friend istream& operator > > (istream& istr,
                                      Set < T > & x);
        friend ostream& operator < < (ostream& ostr,
                                      const Set < T > & x);
};
```

说明

模板类 Set 实现了整型值的集合，类型 T 可以是任何可用 i = int(V)和 V = V(i)进行转换的数据类型，其中 V 和 i 的定义为：

```
T V;
int i;
```

构造函数创建一个空集，并用算术运算符来定义集合的并（“+”）、交（“*”）和等于（“= =”）运算，Insert 和 Delete 函数和赋值运算符（“=”）一起用来改变集合。

输入输出运算符用于读入和输出集合，格式为大括号内用逗号隔开的整数。

例：

```
// 范围为 0..24 的整数集合
Set < int > S(25);
// ASCII 字符集
Set < char > T(128), U(128);
cin > > S;                  // 输入{4,7,12}
S.Insert(5);
cout < < S < < endl;        // 输出{4,5,7,12}
cin > > T;                  // 输入{a,e,i,o,u,y}
U = T;                      // U = {a,e,i,o,u,y}
T.Delete('y');
cout < < T < < endl;        // 输出{a,e,i,o,u}
if (T * U = = T)
   cout < < T < < " is a subset of " < < U < < endl;
```

类 Set 的实现可在文件“set.h”中找到。

类 Set 可供用户来创建自定义的枚举类型的集合和常规的整型类型的集合，如 int 和 char 等。如果希望集合的输入输出也能用于枚举类型则还需重载枚举类型的输入输出运算符。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

例 8.7

有枚举类型定义如下：

```
enum Days {Sun, Mon, Tues, Wed, Thurs,Fri,Sat};
```

附录程序中给出了为该类型重载的运算符“<<”，它与主程序一起放在文件“enumset.cpp”中。

下面举例说明它们的用法。

```
// 定义 3 个对象表示不同日期集合
Set<Days> weekdays(7), weekend(7), week(7);
// 数组 wd 和 we 定义了一周中的日期，并用它们来初始化对象 weekdays 及 weekend
Days wd[] = {Mon, Tues, Wed, Thurs, Fri}, we[] = {Sat, Sun};
// 将数组值插入到集合中
for (int i=0; i<5; i++)
    weekdays.Insert(wd[i]);
for (i=0; i<2; i++)
    weekend.Insert(we[i]);
// 在中括号内输出集合
cout << weekdays << endl;
cout << weekend << endl;
// 将两集合求并集后输出
week = weekdays + weekend;
cout << week << endl;
<运行>
{Mon, Tues, Wed, Thurs, Fri}
{Sun, Sat}
{Sun, Mon, Tues, Wed, Thurs, Fri, Sat}
```

亚里士多德筛子

生活在公元前 3 世纪的古希腊数学家和哲学家亚里士多德，提出了著名的用集合来求小于或等于整数 n 的质数的算法。该算法首先初始化一个包括 2～n 的所有整数的集合，通过对集合中元素的多次筛选，我们将其中的合数从中筛去，最后留在集合中的元素就是质数了。筛选时，从集合中的最小元素 m=2 开始，扫描整个集合并将集合中所有 m 的倍数(因子>1)，如 2*m,3*m,…,k*m 删去，由于这些数均可被 m 整除，它们不可能是质数。筛子中剩下的更大一些的数为 m=3，它是质数，由于在 m=2 时，我们已经删除了 2 的所有位数，如 6,12,18 等，所以这遍扫描删除了 9,15,21,…继续这一过程，集合中的下一个数为 m=5。因为 4 已经作为 2 的倍数被删除了，此时，我们可以删除 5 的倍数(25,35,55,…)，继续这一过程，直到扫描完整个集合。此时依然留在筛子中的数即是从 2 到 n 的所有质数。

算法的原理很简单，就是从全集中去掉所有的合数，算法可用反证法加以简单证明，设 m 为合数，且最后留在筛子中。由于 m 为合数，它可以分解成：

$$m = p * k,\ p > 1$$

其中 p 为 2 到 m－1 之间的一个质数，而在筛子算法中，p 的所有倍数都将被删除，所

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

以 m 不可能留在筛子中。

例 8.8

下图给出用亚里士多德筛子求所有 2 到 25 之间的质数的过程。

亚里士多德筛子：n=25

删除 2 的倍数		2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~	21	~~22~~	23	~~24~~	25
删除 3 的倍数		2	3		5		7		~~9~~		11		13		~~15~~		17		19		~~21~~		23		25
删除 5 的倍数		2	3		5		7				11		13				17		19				23		~~25~~

集合中剩下的 7,11,13,17,19 及 23 没有它们的倍数。

质数为{2,3,5,7,11,13,17,19,23}

程序 8.6 亚里士多德筛子

我们用函数 PrintPrimes 来实现上述筛子算法。算法进行了一点优化，将 m 的范围限制在 $2 \leq m \leq \sqrt{n}$ 之间，这同样可以将集合中所有合数删除掉，因为若合数 t = p * q 留在筛子内，只有当 p 和 q 均大于 $\sqrt{n}$ 时才有可能，而此时

$$t = p * q > \sqrt{n} * \sqrt{n} = n$$

则 t 已经超出集合范围之外了。因此，必有一因子，设为 p，有 $p \leq \sqrt{n}$，这时，t 作为 p 的倍数，算法也会将其从集合中删除。实际实现过程中，我们用测试 m * m 是否小于等于 n 来代替求 n 的平方根。

```
#include <iostream.h>
#include <iomanip.h>
#include "set.h"    // 使用集合类
// 用亚里士多德筛子计算并输出所有 <= n 的质数
void PrintPrimes(int n)
{
    // 筛子中的数值为 2..n
    Set<int> S(n+1);
    int m, k, count;
    // 将 2..n 所有的数插入集合中
    for (WB)(m=2;m <= n; m++)
        S. Insert(m);
    // 检查从 2 到 sqrt(n)中的所有整数
    for (m=2; m*m <= n; m++)
        // 若 m 在集合中，删除其在集合中的所有倍数
        if (S.IsMember(m))
            for (S.IsMember(k))
                S.Delete(k);
    // S 中留下的所有元素均为质数，按每行输出 10 个质数打印输出
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    count = 1;
    for (m=2;m <= n; m++)
        if (S.IsMember(m))
        {
            cout << setw(3) << m << " ";
            if (count++ % 10 == 0)
               cout << endl;
        }
    cout << endl;
}
void main(void)
{
    int n;
    cout << "Enter n: ";
    cin >> n;
    cout << endl;
    PrintPrimes(n);
}
/*
<程序 8.6 运行结果>
Enter n: 100
   2   3   5   7  11  13  17  19  23  29
  31  37  41  43  47  53  59  61  67  71
  73  79  83  89  97
*/
```

类 Set 的实现

类 Set 的私有函数 ArrayIndex 和 BitMask 实现了集合的存储模式，ArrayIndex 判定值 elt 属于哪个数组元素。这只需将该值除以 16，我们用其右移 4 位来实现。

```
// 判定表示值 elt 的位属于哪个数组元素
template <class T>
int Set<T>::ArrayIndex(const T& elt) const
{
    // 将 elt 转换为整数并右移 4 位
    return int(elt) >> 4;
}
```

得到正确的数组下标后，BitMask 可返回一个 unsigned short 值，它代表 elt 的此位的值为 1，该函数可用来设置或清除该位。

```
// 产生一个无符号短整数，其 elt 位的值为 1
template <class T>
unsigned short Set<T>::BitMask(const T& elt) const
{
    // 用 & 来术除以 16 的余数
    return 1 << (int()elt & 15);
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

出错处理 类 Set 通过调用私有成员函数 Error 来响应各种出错情况。用枚举类型 ErrorType 来将可能出错的代码和它的错误名称对应起来，函数的输入参数为 ErrorType，用来标识错误原因，然后终止程序运行。Error 的实现在补充程序中给出。

类 Set 的构造函数 类 Set 共有两个构造函数可创建集合对象，一个产生空集，另一个为复制构造函数，空集建立过程如下：先通过全集大小 sz 求得需要的数组来表示集合，申请数组空间，将数组的所有元素值赋 0。

```
// 构造函数。产生空集
template <class T>
Set<T>::Set(int sz): setrange(sz)
{
    // 存放集合元素所需无符号短整数的个数
    arraysize = (setrange+15) >> 4;
    // 申请数组空间
    member = new unsigned short [arraysize];
    if (member == NULL)
       Error(OutOfMemory);
    // 将所有位置 0 创建空集
    for (int i = 0; i < arraysize; i++)
       member[i] = 0;
}
```

Set 的运算 通过重载 C++ 的运算符"+"和"*"来分别实现集合的二目运算求并集和求交集，对于集合运算符"+"(并)，先创建一个集合对象 tmp 存放范围 0 至 setrange－1 的元素，然后将其元素赋值为当前集合和集合 X 的相应数组元素按位或后的值，集合 tmp 即为运算符"+"的返回值，注意，当两集合的全集不同时，函数返回一个出错信息。

```
// 求当前集合对象和对象 X 的并集，然后返回该集合
template <class T>
Set<T> Set<T>::operator+ (const Set<T>& x) const
{
    // 两集合全集应相同
    if (setrange != x.setrange)
       Error(SetsDifferentSize);
    // tmp 中存放它们的并集
    Set<T> tmp(setrange);
    // tmp 的每个数组元素为当前对象和 X 的按位或
    for (int i = 0; i < arraysize; i++)
        tmp.member[i] = member[i] | x.member[i];
    // 返回并集
    return tmp;
}
```

与并集类似，交集运算符("*")也创建集合对象 tmp，然后将其赋值为当前集合与集合 X 的对应数组元素按位与。然后返回集合 tmp，即为两个集合的交集。

函数 IsMember 判断元素与集合的从属关系，若元素在集合中，返回值为 TRUE；否则返回 FALSE。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 判断 elt 是否在集合中
template <class T>
int Set<T>::IsMember(const T& elt)
{
    // elt 在范围 0 至 Setrange-1 当中吗?
    if (int(elt) < 0 || int (elt) >= setrange)
        Error(InvalidMemberRef);
    // 返回对立于 elt 的位
    return member[ArrayIndex(elt)] & BitMask(elt);
}
```

Set 元素的插入与删除 元素的插入通过将参数 elt 对应的位置 1 来实现。

```
// 往集合中插入 elt
template <class T>
void Set<T>::Insert(const T& elt)
{
    // elt 可以是集合的元素吗?
    if (int(elt) < 0 || int (elt) >= setrange)
        Error(InvalidMemberRef);
    // 置对立于 elt 的位为 1
    member[ArrayIndex(elt)] |= BitMask(elt);
}
```

删除将 elt 对应的位置 0,这可用"与"运算符和一个只是"elt"位为 0 的屏蔽字来实现,屏蔽字由求反运算符"~"产生。

```
// 从集合中删除 elt
template <class T>
void Set<T>::Delete(const T& elt)
{
    // elt 在范围 0 至 setrange-1 吗
    if (int(elt) < 0 || int(elt) >= setrange)
        Error(InvalidMemberRef);
    // 清除对立于 elt 的位,注意 ~BitMask(elt)只在我们感兴趣的位为 1,其余所有位均为 0
    member[ArrayIndex(elt)]&= ~BitMask(elt);
}
```

Set 的输入输出 我们将流运算符">>"和"<<"重载以实现集合的输入输出,输入运算符">>",用$\{i_0,i_1,\cdots,i_m\}$的格式读入集合 X,集合元素均在大括号内,并用逗号隔开,每个整型数值 i_n 代表一个集合元素,输出运算符"<<"用$\{i_0,i_1,\cdots,i_m\}$的形式输出集合 X,其中 $i_0<i_1<\cdots<i_m$,均为集合中的元素。

输入是最难实现的函数,它先用输入函数 get 滤掉所有空格,然后检查当前字符是否为"{",若不是,调用函数 Error,打印出错信息并退出,找到集合的首字符"{"后,函数读入逗号隔开的各整型数值,并将每个元素加入集合中,运算过程中,还要检查逗号位置是否正确,元素是否在 0 到 setrange-1 之间。但各元素间的空格可以为任意个数。

书面作业

8.1 声明一个含 10 个整数的数组以及一个指向整数的指针:

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
int a[10], *p;
```

假设有以下语句：

```
for(i = 0;i < 2;i++)
{
    p = new int [5];
    for (j = 0; j < 5;j++)
        a[5*i + j]= *p++ = i + j;
}
```

(a) 求下列语句的输出结果：

```
for(i = 0; i < 10; i++)
    cout << a[i] << "  ";
cout << endl;
```

(b) 讨论语句

```
p = p - 10;
```

是否将 p 重置到原始动态表的开始。

(c) 假设 q 是指向原始动态表的指针。下列代码是否产生与(a)同样的输出？

```
for(i = 0; i < 10; i++)
    cout << *(q+1) << "  ";
cout << endl;
```

8.2 对以下各个声明，用运算符 new 动态分配指定的内存。

(a) `int *px;`

建立指向值为 5 的整数的指针。

(b)
```
long *a;
int n;
cin >> n;
```

建立含 n 个长整数的动态数组，其头指针为 a。

(c)
```
struct DemoC
{
    int one;
    long two;
    double three;
} DemoC *p;
```

建立一个由 p 所指向的结点。然后将各个域赋值为{1, 500000,3.14}。

(d)
```
struct DemoD
{
    int one;
    long two;
    double three;
    char name[30];
};
DemoD *p;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

动态建立一个由 p 所指向的结点，然后将其各域赋值为{3, 35, 1.78, "Bob C++"}。

(e) 写出释放(a)~(d)中所用内存的语句。

8.3 在类 DynamicInt 中，构造函数用 new 运算符动态为一个整数分配空间并将其地址赋给数据值 pn。公有方法 GetVal 从动态内存中取出数据，而 SetVal 则存入数据。

```
class DynamicInt
{
    private:
        int *pn;
    public:
        // 构造函数及复制构造函数
        DynamicInt(int n = 0);
        DynamicInt(const DynamicInt& x);
        // 析构函数
        ~DynamicInt(void);
        // 赋值运算符
        DynamicInt& operator = (const DynamicInt& x);
        // 数据处理方法
        int GetVal(void);       // 取整数值
        void SetVal(int n);     // 置整数值
        // 转换运算
        operator int(void);     // 返回整数值
        // 流输入及输出函数
        friend ostream& operator << (ostream& ostr, const Dy-
                                     namicInt& x);
        friend istream& operator >> (istream& istr,
                                     DynamicInt& x);
};
```

(a) 编写实现构造函数和析构函数的方法。

(b) 编写重载赋值运算符"="和实现复制构造函数的方法。

(c) 实现 GetVal，运算符 int 以及 SetVal。

(d) 实现能读写 *pn 的值的流 I/O 函数。

8.4 用书面作业 8.3 题中的 DynamicInt 的声明完成以下各小题。

(a) `DynamicInt *p;`

若要为一个初始值为 50 的对象分配空间，试写出其声明。

(b) `DynamicInt *p;`

为一个含 3 个元素的数组 p 分配空间。数组中每个对象的数据值是多少？

(c) `DynamicInt a[10];`

如何声明一个内含 10 个 DynamicInt 对象的数组并将其每个元素值初始化为 100？

(d) 用 delete 语句释放(a)~(c)中所使用的动态内存。

8.5 将书面作业 8.3 中的 DynamicInt 类改写为模板类 DynamicType。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
template <class T>
class DynamicType
{
    private:
        T *pt;

    public:
        // 构造函数及复制构造函数
        DynamicType(T value);
        DynamicType(const DynamicType<T>& x);
        // 析构函数
        ~DynamicType(void);
        // 赋值运算符
        DynamicType<T>& operator= (const DynamicType<T>& x);
        // 数据处理函数
        T GetVal(void);          // 取值
        void SetVal(T value);    // 置值
        // 转换函数
        operator T(void);        // 返回值
        // 流输入/输出
        friend ostream& operator<< (ostream& ostr, const
                                    DynamicType<T>& x);
        friend istream& operator>> (istream& istr,
                                    DynamicType<T>& x);
};
```

8.6 本题使用书面作业 8.3 和 8.5 中所编写的的类 DynamicInt 和 DynamicType 声明以下对象：

```
DynamicType<DynamicInt> D(DynamicInt(5));
```

求下列语句的输出：

```
cout << D << endl;
cout << D.GetVal().GetVal() << endl;
cout << int(D.GetVal()) << endl;
cout << DynamicInt(D) << endl;
cout << int(DynamicInt(D)) << endl;
```

8.7 类 ReadFile 声明如下：

```
class ReadFile
{
    private:
        // 从 fin 中读入字符流并申请可存放 bufferSize 个字符的数组
        ifstream fin;
        char *buffer;
        int bufferSize;
    public:
        // 构造函数,传递文件名及缓冲区的大小
        ReadFile(char *name, int size);
        // 打印出错信息并返回
        ReadFile(const ReadFile& f);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 删除缓冲区并关闭 fin
        ~ReadFile(void);
        // 赋值运算符
        void operator = (const ReadFile x);
        // 从文件中读下一行。若文件结束,返回 0
        int Read(void);
        // 拷贝当前行到 buf
        void CopyBuffer(char *buf);
        // 输出当前行到 cont
        void Printbuffer(void);
};
```

(a) 实现此类。

(b) 编写函数

```
void LineNum(ReadFile& f);
```

读入 f 并列出带行号的文件内容。

8.8 书面作业 8.5 中设计了类 Dynamictype。假设有如下声明,试回答下面各个问题。

```
DynamicType <int> *p, Q;
DynamicType <char> *c;
```

(a) 写一条语句,为指针 p 要指向的值为 5 的 DynamicType 对象分配空间。

(b) 用 3 种不同的方法打印 *p 中所分配的值 5。

(c) 下列各条语句是否合法? 若是,其动作是什么?

```
c = new DynamicType <char> [65];
c = new DynamicType <char> (65);
```

(d) 若输入为 35,则输出是什么?

```
cin >> *p;
Q = *p;
cout << Q << endl;
```

(e) 若使用 d 所得到的 Q 值,输出是什么?

```
DynamicType <int> R(Q);
cout << Q << endl;
```

(f) 输出是什么?

```
Q = DynamicType<int> (68);
c = DynamicType<char> (char(int (Q)));
cout << c << char(c) << int(c) << endl;
```

(g) 写出删除对象 *p 和 *c 的语句。若执行下列语句会发生什么?

```
delete Q;
```

8.9 (a) 若 CL 是类,说明为什么不能将复制构造函数声明为:

```
CL (const CL x);
```

(b) 说明为什么一般不用以下方式声明 CL 的赋值运算符:

```
void operator = (const CL& x);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

8.10 (a) 关键字 this 的含义是什么？说明为什么它仅在成员函数内部有效。

(b) 说出 this 的主要用途。

8.11 第 6 章中的类 Rational 实施有理数的算术运算。在类中加入"+ ="运算符。说明为什么以下实现是正确的。

```
Rational& Rational::operator+ = (const Rational& r)
{
    *this = *this + r;
    return *this;
}
```

8.12 类 ArrCL 用重载的数组下标运算符和指针转换实现数组边界检查。类中包含一个 50 个元素的数组 arr 和一个长度域(length)。当建立一个 ArrCL 型的对象时，客户程序可以约定一个最大表长度，将此值作为参数传递给构造函数。

```
const int ARRAYSIZE = 50;
template < class T >
class ArrCL
{
    private:
        T arr[ARRAYSIZE];
        int length;
    public:
        // 构造函数
        ArrCL(int n = ARRAYSIZE);
        // 返回表的大小
        int ListSize(void) const;
        // 下标运算符，应实现一个"安全数组"
        T& operator[] (int n);
        // 指针转换。返回地址 addr
        operator T* (void) const;
};
```

内嵌表 arr 的大小被限定在 ARRAYSIZE = 50 以内。若客户程序试图开辟更大的数组，构造函数会将其大小(size)置为 ARRAYSIZE 并打印警告信息。

(a) 编写开辟 20 个整数的数组 A、50 个元素的字符数组 B 和 25 个元素的浮点数组 C 的声明。

(b) 以下循环是否有问题？

```
ArrCL < long > A(30);
for(int i = 0; i < = 30; i + + )
    A[i] = 0;
```

(c) 试解释为什么输出是"420 420"？

```
int Suml(const ArrCL < int > & A)
{
    int s = 0;
    for (int i = 0; i < A.ListSize();i + + )
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            s + = A[i];
        return s;
    }
int Sum2(int * A,int n)
{
    int s = 0;
    for (int i = 0; i < n; i + + )
        s + = * A + + ;
    return s;
}
. . .
ArrCL < int > arr(20);
for (int i = 0; i < 20; i + + )
    arr[i] = 2 * (i + 1);
cout < < Sum1(arr) < < " " < < Sum2(arr, 20) < < endl;
```

(d) 实现类 ArrCL。

8.13 现有声明如下：

```
String A("Have a"), B("nice day!"), C(A), D = B;
```

(a) C 的值是多少？

(b) D 的值是多少？

(c) D = A + B 的值是多少？

(d) C + = B 的值是多少？

8.14 假设有如下串：

```
String S("abc12xya52cba"), T;
```

(a) S.FindLast('c')的值是多少？

(b) S[6]的值是多少？

(c) S[3]的值是多少？

(d) S[24]的值是多少？

(e) T = S.Substr(5,6)的值是多少？

(f) 执行完以下代码后 T 的值是多少？

```
T = S;
T.Insert("ABC",5);
```

8.15 现有以下声明：

```
# define TF(b) ((b) ? "TRUE" : "FALSE")
String s1("STRING "), s2("CLASS");
String s3;
int loc, i;
char c, cstr[30]
```

确定以下各个语句的输出。

(a) s3 = s1 + s2;

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    cout << s1 << "concatenated with " << s2
         << " = " << s3 << endl;
(b) cout << "Length of " << s2 << " = "
         << s2.Length() << endl;
(c) cout << "The first occurrence of 'S' in " << s2
         << " = " << s2.Find('S', 0) << endl;
    cout << "The last occurrence of 'S' in " << s2
         << " = " << s2.FindLast('S') << endl;
(d) cout << "Insert 'OBJECT' into s3 at position 7."
         << endl;
    s3 = s1 + s2;
    s3.Insert("OBJECT", 7);
    cout << s3 << endl;
(e) s1 = "FILE1.S";
    for (i = 0; i < s1.Length();i++)
    {
        c = s1[i];
        if (c >= 'A' && c <= 'Z')
        {
            c += 32;
            s1[i] = c;
        }
    }
    cout << s1 << endl;
(f) s1 = "ABCDE";
    s2 = "BCF";
    cout << "s1 < s2 = " << TF(s1 < s2) << endl;
    cout << "s1 == s2 = " << TF(s1 == s2) << endl;
(g) s1 = "Testing pointer conversion operator.";
    strcpy(cstr,s1);
    cout << cstr << endl;
```

8.16 假设变量 x, y 和 z 的定义如下：

```
unsigned short x = 14, y = 11, z;
```

执行完下列语句后，z 的值分别是多少？

(a) z = x | y;
(b) z = x & y;
(c) z = ~0 << 4;
(d) z = ~x & (y >> 1);
(e) z = (1 << 3) & x;

8.17 本题给出了实施位处理操作的 4 个函数。对每个函数，在以下描述性语句中找出与之匹配的一句。

(a) 对一个特定的机器，确定其整数类型(int)的位数。
(b) 返回从位元(bit)位置 p 开始的 n 个位元的数值。
(c) 对从位元(bit)位置 p 开始的 n 个位元取反。位元位置 0 对应整数值最左边的

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

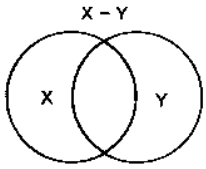
www.Linuxidc.com

一个位元。

(d) 顺时针转动一个整数的位元。

```
unsigned int one(unsigned int n, int b)
{
    int rightbit;
    int lshift = three() - 1;
    int mask = (unsigned int) ~0 > > 1;
    while (b- -)
    {
        rightbit = n & 1;
        n = (n > > 1) & mask;
        rightbit = rightbit < < lshift;
        n = n | rightbit;
    }
    return n;
}
unsigned int two(unsigned int x, int p, int n)
{
    unsigned int mask = (unsigned int) ~(~0 < < n);

    return(x > > (p-n+1)) & mask;
}
int three(void)
{
    int i;
    unsigned int u = (unsigned int) ~0;
    for(i=1;u = u > > 1;i+ +);
    return i;
}
unsigned int four(unsigned int x, int p, int n)
{
    unsigned int mask;
    mask = ~0
    return x^(~(mask > > n) > > p);
}
```

8.18 在 Set 类中增加差运算符("-")。此运算符需要两个集合 X 和 Y 作为其参数,返回集合由属于 X 但不属于 Y 的元素组成。



X = {0, 3, 7, 9}, Y = {4, 7, 8, 10, 15}
X - Y = {0, 3, 9}

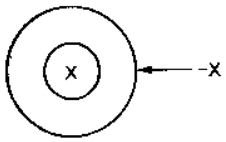
(提示:用以下算法: diff.member[i] = member[i]& ~ x.member[i])

8.19 在 Set 类中增加求补运算符("~")。此单目运算符的集合由全集(Universal Set)中

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

所有不属于 X 的值所组成。




```
Set < int > X(10), Y(10);
for(int i = 0; i < 10; i + = 2)
    X.Insert(i);
Y = -X; // X = {1, 3, 5, 7, 9}
```

8.20 (a) 用求补运算符以及以 Set 对象方式建立全集。

(b) 用求补和求交运算符实现书面作业 8.18 中的差运算。

上机题

8.1 本题使用书面作业 8.3 中所编写的 DynamicInt 类。

(a) 将运算符"<"重载为 DynamicInt 的外部函数。它可以不是类的友元。

(b) 编写函数

```
DynamicInt * Initialize(int n);
```

返回指向含 n 个动态分配对象的数组的指针。将对象数组初始化为 1 ~ 1000 范围内的随机整数。

(c) 主程序应读取一个整数 n 并用 Initialize 建立一个 DynamicInt 对象的数组。用第 7 章中基于模板的交换排序法对表进行排序。打印结果表。

8.2 本程序使用书面作业 8.7 中的 ReadFile 类。用函数 LineNum 读出一个文件并以带行号的方式将其打印到屏幕上。

8.3 本程序使用书面作业 8.7 中的 ReadFile 类。

(a) 编写函数

```
void CapLine(ReadFile& f, char * capline);
```

读出文件 f 的下一行内容,将所有小写字母转换为大写,将结果放在 capline 中返回。

(b) 用以上函数读一个文件并以大写方式将其打印出来。

8.4 本题使用书面作业 8.12 中的 ArrCL 类。建立一个 10 个整数的安全数组并要求用户输入 10 个数据项。在调用完文件"arrsort.h"中的基于模板的交换排序法以后,将排好序的表打印出来。若企图访问 A[10],则会产生一个出错消息且程序终止。

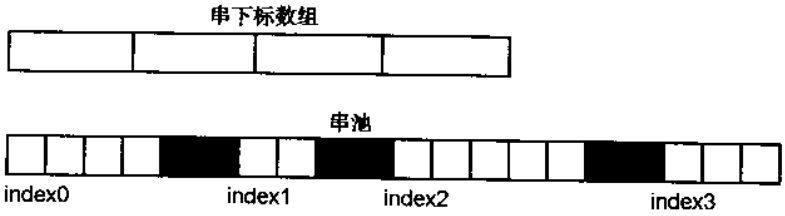
8.5 本题对第 7 章中设计的基于模板的 Stack 类(在"tstack.h"中)进行修改。用最初包含 MaxStackSize 个元素的 Array 对象代替静态数据成员。从构造函数中删除堆栈尺寸参数并重写 Push 方法使得堆栈尺寸可以随需要增长。用以下方法对修改结果进行

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

测试:将整数值 1 ~ 100 压入堆栈,然后再删除它们,每隔 10 个打印其中的一个值。

8.6 用流方法 getline 读入一个文件并将串存入一个叫作字符串池的 Array 对象[1]。往池中插入一行时要先用 Resize 操作给池子增加足够的空间,过后还要将该行的起始下标放入串下标数组中,如有必要应调整其大小。此数组将标明池中相继出现的每个字符串。输入一个整数 N,打印出文件的最后 N 行。若文件少于 N 行则打印出整个文件。



8.7 修改程序 8.3,将已计算出的所有素数放入一个表中,令程序维护此表。对于序列中的下一个整数 n,仅用表中素数而不用 3 到 n/2 之间的所有因子。若表中没有一个素数可以除尽 n,则 n 是一个新素数,可以将它加入到表中。这种做法的数学根据是任何数都可以分解为其素因子的乘积。

8.8 (a) 编写函数

```
void Replace(String& source, int pos,
             const String& repstr);
```

替换 source 中从下标 pos 开始的 repstr.Length()个字符。若 source 尾部的字符数不是 repstr.Length()个,则插入 repstr 中的所有字符。

(b) 编写函数

```
void Center(String& S);
```

调用 Replace 打印 S,令其在含 80 个字符的一行"%"处于居中位置。

写一个主程序,对(a)和(b)进行测试。

8.9 读取一个代表文档中文本行数的整数 n。为 n 个指向 String 对象的指针动态分配空间。用 ReadString 读取 n 个 String 对象。

文本行中可能包括特殊符号"#"和"&",例如

```
Dear #

  Your lucky gift is avalilable at &. By going to & and identify-
  ing your name, the attendant will give you your prize. Thank
  you # for your interest in our contest.

Sincerely,
The String Man
```

输入一个字符串 poundstr,用它替换文档中的所有"#"。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

输入一个字符串 ampstr，用它替换文档中所有"&"。

遍历前述串指针数组并实施替换。打印出最后的文档。

8.10 写一个程序，初始化两个 10 元素数组 int A 和 int B，并用它们建立集合 A 和 B。集合求差运算符"-"在书面作业 8.18 中已定义过。本程序必须计算 A - B 和 B - A 并打印出结果。程序还应验证

A + B = (A - B) + (B - A) + (A * B)

8.11 函数

```
template <class>T
Set <T> ExclusiveUnion(const Set<T>& X,const Set<T>& Y);
```

返回所有属于 X 或 Y 但不同时属于 X 和 Y 的元素的集合。

(a) 用图形说明可以用下面两个公式之一计算 ExclusiveUnion(异或)：

1. (X - Y) + (Y - X)
2. X * ~Y + ~X * Y

集合的差("-")和补("~")运算在书面作业 8.18 和 8.19 中已作过定义。

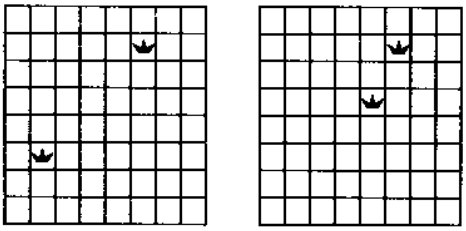
(b) 用第 2 个公式实现 ExclusiveUnion。用以下集合对程序进行测试：

X = {1, 3, 4, 5, 7, 8, 12, 20}, Y = {3, 5, 9, 10, 12, 15, 20, 25}

对以上数据，应有

```
ExclusiveUnion(X,Y) = {1, 4, 7, 8. 10, 15, 25}
```

8.12 在国际象棋棋盘上的两个皇后，若其中任意一个都可以移动到另一个的位置，则我们称它们为"对峙的(attacking)"，即它们位于同一行、列或对角线上。附图中给出了对峙的和非对峙的皇后的例子。



对峙皇后　　非对峙皇后

如果给出每个皇后的位置，我们讨论下面两个问题：一是求每个皇后的所有可能的走法；二是判断皇后是否处于对峙状态。

棋盘上的每个位置都可以用一个行号和一个列号来表示，其取值范围都是 0 到 7。可以为棋盘上的每个小方块赋一个范围在 0 ~ 63 之间的唯一值，方法是将每个行/列对(i, j)与整数 i * 8 + j 相关联。这样我们可以将棋盘看作 64 个范围在 0 到 63 之间的整数的集合。

编写函数

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
void ComputeQueenPositions(Set < int > & Board, int rowq, int colq);
```

它以集合 Board 和皇后位置(rowq, colq)为参数，将皇后能够走到的所有小方块加入到集合中。

编写函数

```
void PrintQueenPositions(Set < int > & Queen,int Qrow,int QCol,int QOther-
                         Row, int QOtherCol);
```

有一个皇后在位置(QRow, QCol)处，Queen 是该皇后能走到的位置的集合。另有一皇后在位置(QOtherRow, QOtherCol)处。函数打印出棋盘，在位于(QRow, QCol)的皇后所能走到的所有小方块上置一字符"X"，在其余地方则置" "。若小方块上有一个皇后则打印"Q"。

编写函数

```
int AttackingQueens(int Q1Row,int Q1Col,int Q2Row,int Q2Col);
```

它以两个皇后的位置为参数并判断它们是否对峙。为此，只要判断它们是否处于同一行、列或对角线即可。

写一个主程序，读入两个皇后的位置并打印每个皇后能走到的位置的集合。然后再显示皇后是否对峙的消息。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第9章 链表

9.1 结点类
9.2 构造链表
9.3 设计链表类
9.4 类 LinkedList
9.5 LinkedList 类的实现
9.6 用链表实现集合
9.7 实例研究：打印缓冲池
9.8 循环表
9.9 双向链表
9.10 实例研究：窗口管理
书面作业
上机题

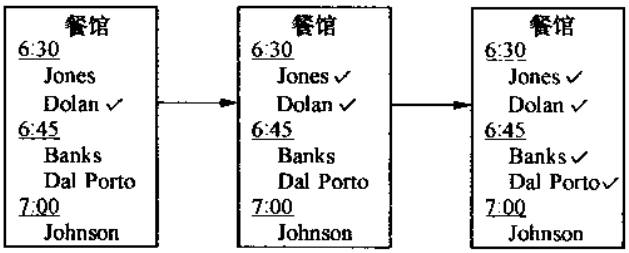
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

我们已经定义过用数组实现的集合类。这些集合包括堆栈、队列以及更一般地维护顺序表项的类 Seqlist。这一章将研究提供更灵活地删除或增加表项的方法——链表类。

数组可以用于简单的表存储应用程序。例如，许多飞行航班出售非预定机票，旅客可以用它们领取登机牌。起飞前出售机票时，航空公司将旅客的姓名加到一表中。登机前，旅客在空乘人员那里登记，后者把旅客的名字从机票的表中去掉，并将其加到登记旅客的表中。有了这些表，航空公司就可以知道机上的旅客数以及仍未登机的旅客数。

基于数组的表对于需要更灵活的表处理方法的应用程序来说是不够的。例如，考虑一家可以预订的餐馆的情况。餐馆经理需要将名字输入到一个以时间排序的表中，并使用一些判别因素，以从表中删除一个名字。如果顾客打电话取消预定或者顾客入店则立即进行删除。餐馆经理必须周期性地扫描这张表，将那些在所要求的时间过后 15 分钟之内没有露面的顾客的名字删掉。



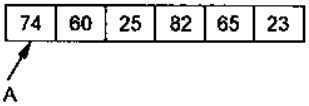
时间 6:15
(Dolan取消预定)

时间 6:40
(Jones到达
Dal Porto到达)

时间 7:05
(删除Banks)

基于数组的表无法有效地处理餐馆定座系统。名字必须被插入到表中不同的位置以代表不同的到达时间，这就要求将名字右移；而删除则要将名字往左移。餐馆不可能预计到定座数，因而必须比较宽松地分配一块较大的内存以应付极端的情况。一旦开始服务，定座表将处理非常活动的状态，名字将快速地出入表中。数组因其存储的连续性将不能很好地响应这些动态变化。在下面的讨论中，我们将看看用数组会存在的一些问题，并引进动态链表以找到解决办法。

数组是一种定长结构。即使是动态数组在变长操作后，也有一固定长度。例如，动态地建立 6 个元素的整数数组 A：

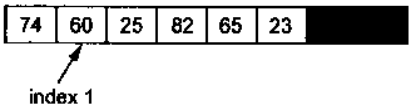


一旦数组满了，则只有将表加长才能添加新项，这一过程需要将每个元素拷贝到新存储区。经常对大的表进行变长操作可能会严重影响系统性能。

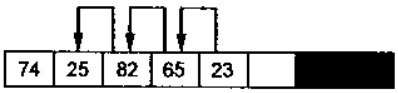
数组在内存中连续存放表项，这样可以直接存取表项，但不能有效地处理添加和删除表项，除非这些操作发生在数组的末尾。例如，假定你要从下表的第 2 个位置删除 60 这一项：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

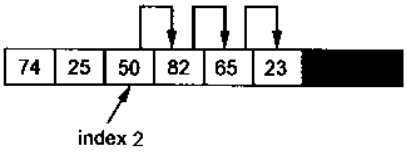
www.Linuxidc.com



为了保持元素在表中的全部邻接次序，必须将 4 个表项左移。



假设我们要在下标 2 处增加新的元素 50。因数组连续存放元素，所以我们必须将表中的 3 个元素右移一个位置以便腾出一个位置。

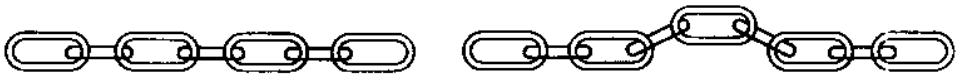


对于一个含 N 个元素的表，在表尾插入和删除元素所需的计算机时间为 O(1)。但在一般情况下，移位次数的期望值是 N/2，所需计算时间为 O(N)。

描述一个链表

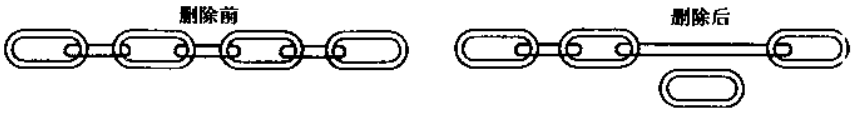
有效的数据管理对于任何表的实现都是至关重要的。我们需要开发一种新的结构以便从连续存储模式下解脱出来。我们可以使用链条作为模型。

通过增加新链，表的长度可以无限增大。更进一步，新的表项插入到表中时只需拆除连接、增加一个新链，然后恢复连接即可。



插入前　　插入后

删除表项时，只需拆除两个连接、删去一个链，然后重新接上表链即可。



删除前　　删除后

本章我们将研究这种被称为“链表”的结构，以实现一个顺序表。链表将提供充足的表处理手段，并能克服以上所提及的有关数组的许多局限性。

本章概述

链表中的独立表项被称为“结点”。我们将生成一种定义结点对象并提供链表操作方

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

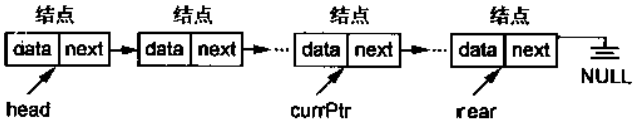
www.Linuxidc.com

法的结点类，特别是要实现在当前结点后插入和删除结点的方法。我们还要研究使用单个结点、建立链表、扫描结点以及更新其值的方法。

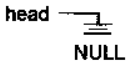
我们将设计一种将主要结点算法封装在类结构中的链表类，可以用该类实现队列(Queue)以及顺序表(SeqList)类，因而可以克服基于数组的表的局限性，用来解决许多有趣的问题，包括从表中删除所有重复值。LinkedList 类用于两种情况的研究开发。在打印缓冲池研究中，用它来激发操作系统的打印管理器。第二个方面是用它讨论动态窗口的构建。表设计为一种循环结构，在概念和编码两方面都有其优越性。双向链表可以用于需要在两个方向查找元素的场合。我们定义 CNode 和 DNode 类以分别实现循环链表和双向链表结构。

9.1 结点类

一个结点由数据域和指向链表中下一项的指针组成。指针是将表中单个结点维系在一起的纽带。



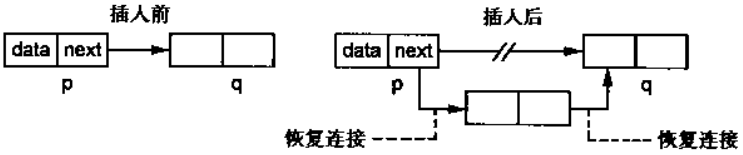
链表由系列结点组成，其第 1 个元素，或称“头(front)”，是一个由指针 head 指向的结点。表链将结点从表头至表尾(rear)串在一起。识别表尾结点的方法是其指针域的值为 NULL=0。我们可以将链表当作一个滑道，其头部为入口，在 NULL 处为出口。表应用程序通过按指针访问下一个结点的方法遍历结点。在扫描过程中的任一点，当前位置可以由指针 CurrPtr 访问。在表中不含结点的情况下，头指针为 NULL。



定义一个结点类

带有数据和指针域的结点是链表的基本构件。结点结构中含有初始化数据成员和访问下一个结点的指针方法。每个结点都提供在它自己后面插入新的结点以及删除后继结点的方法。

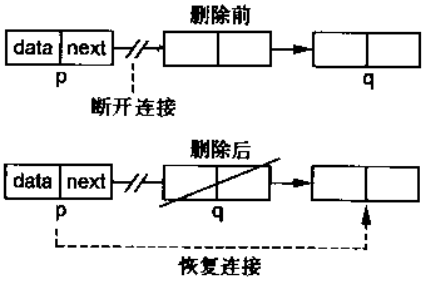
下面的图中演示了基本的结点操作方法。在任何给定的结点 p 上，我们可以实现操作 InsertAfter，它将一个新结点附加到当前结点后。具体过程是断开与后继结点 q 的连接，然后插入新结点(NewNode)，最后恢复连接。



用相似的过程可以描述操作 DeleteAfter，它删除紧随在当前结点之后的结点。我们将 p 与其后继 q 拆分开，然后将它与 q 的后继连接上。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



具有插入和删除操作的结点结构描述了一种抽象数据类型。对于每个结点，操作与其后继结点直接相关。

ADT *Node* **is**

Data

data 域用来保存信息，其值仅仅用于初始化过程

next 域是指向下一个结点的指针.如果 next 为 NULL，则无下一个结点

Operations

Constructor

Initial values:	数据值和指向下一个结点的指针
Process:	初始化这两个域

NextNode:

Input:	无
Preconditions:	无
Process:	取 next 域的值
Output:	返回 next 的值
Postconditions:	无

InsertAfter

Input:	指向新结点的指针
Preconditions:	无
Process:	将 next 值设为指向新结点，并将新结点的 next 值指向当前结点的后继
Output:	无
Postconditions:	结点现在指向新结点

DeleteAfter

Input:	无
Preconditions:	无
Process:	断开与下一个结点的连接，将 next 值指向下一个结点的后继
Output:	指向已删除结点的指针
Postconditions:	结点的 next 值被更新

end ADT *Node*

Node ADT 由 C++ 的基于模板的类描述。

结点类说明

声明

```
template <class T>
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
class Node
{
    private:
        // next 为指向下一结点的指针
        Node < T >  * next;
    public:
        // data 为公有成员
        T data;
        // 构造函数
        Node (const T& item, Node < T > * ptrnext = NULL);
        // 修改表的方法
        void InsertAfter(Node < T >  * p);
        Node < T >  * DeleteAfter(void);
        // 保持下一结点的指针
        Node < T >  * NextNode(void) const;
};
```

说明

next 域的值是指向一个 Node 对象的指针。Node 类是一种自引用的结构，其指针指向要引用其自身类型的对象。

Node 对象可用于许多集合中，如词典和哈希表。我们可定义一公共数据域以提供便利的数据访问。这种方法较之使用一对诸如 GetData/SetData 之类的成员函数可以使客户程序更少一些累赘。如果必须提供数据成员的引用时，这一优点更明显。当实现象词典这样的更高级的类时，这是必须的。next 域仍为私有的，且由成员函数 NextNode 访问。它仅可由 InsertAfter 和 DeleteAfter 修改。

Node 类的构造函数初始化公共数据域以及私有指针域。缺省情况下，next 值为 NULL。

例

```
Node < int > t(10),            // 创建结点 t,其数据值 = 10 且 next 指针为空
```

```
Node < int >  * u;
u = new Node < int > (20);     // 为结点 u 申请空间,并使其数据值 = 20,next 指针为空
Node < char >  * p, * q, * r;
q = new Node < char > ('B');   // q 的数据值为'B'
p = new Node < char > ('A',q); // p 的数据值为'A'且 next 指针指向 q
r = new Node < char > ('C');   // 结点 r 的数据值为'C'
q - > InsertAfter(r);          // 将 r 插入表尾
cout < < p - > data;           // 输出字符'A'
p = p - > NextNode();          // 移向下一结点
cout < < p - > data;           // 输出字符'B'
cout < < endl;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
r = q->DeleteAfter();        // 删除表尾;将其赋值给 r
```

结点类的实现

Node 类既包含公有数据成员又包含私有数据成员。对于公有的 data 域,客户程序和集合类可以直接访问其值。next 域是私有的,因为对此指针域的访问是由成员函数来完成的。我们仅仅允许 InsertAfter 和 DeleteAfter 方法来改变新域的值。若使此域公有,则用户可能会拆断连接、毁坏链表。Node 类包含成员函数 NextNode,它使得客户程序可以遍历链表。

构造函数 构造函数初始化 Node 值和指针 next。类要求每个新的结点对象都要被初始化。对应于指针域 next 的一个参数可以传递给构造函数。在本例中,对象被初始置为指向已知地址的结点,如果没有传递参数,指针域的缺省值为 NULL。

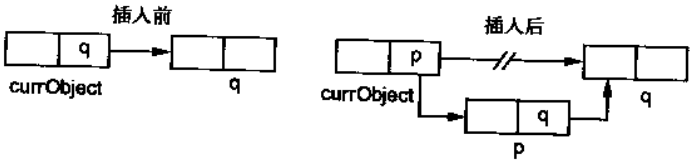
```
// 构造函数. 初始化数据及指针成员
template <class T>
Node<T>::Node(const T& item, Node<T> * ptrnext):
                data(item), next(ptrnext)
{}
```

表操作 NextNode 方法使得客户程序可以访问指针域 next。该方法返回的是 next 的值,因而可用来遍历表。

```
// 返回私有成员 next 的值
template <class T>
Node<T> *Node<T>::NextNode(void) const
{
    return next;
}
```

函数 InsertAfter 和 DeleteAfter 是两种主要的表生成操作。这二者都只涉及到指针更新。

InsertAfter 以结点 p 为参数,并将其作为下一个结点而加入到表中。开始时,当前对象(currObject)指向地址为 q 的结点,其中 q 的值是由 next 域而得到的。该算法修改两个指针。p 的指针域被置为 q,而当前对象的指针域被赋值为 p。



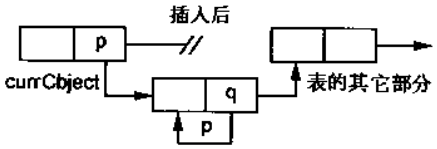

```
// 在当前结点之后插入结点 p
template <class T>
void Node<T>::InsertAfter(Node<T> *p)
{
    // p 指向当前结点的后继结点,然后将当前结点指向 p
    p->next = next;
    next = p;
}
```

指针赋值的顺序很关键,假设赋值语句顺序被颠倒了:

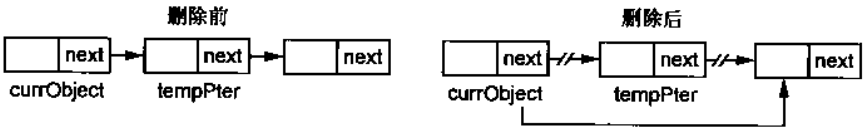
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
next = p;        // 找不到当前对象后的结点
p->next = next
```



DeleteAfter 删除紧随在当前对象之后的结点并将当前对象的指针域连接到表中下一个结点上。如果当前对象之后没有结构点(next = = NULL),函数返回 NULL;否则函数返回已删除的结点的地址;程序员可用它来释放内存。DeleteAfter 算法将下一个结点的地址保存在 tempPtr 中。tempPtr 的 next 域则表示当前对象目前必须指向的表结点。返回值为 tempPtr。此过程只进行一次指针赋值。




```
// 删除当前结点的后继结点并返回其指针
template <class T>
Node<T> *Node<T>::DeleteAfter(void)
{
    // 保存指向被删除结点的指针
    Node<T> *tempPtr = next;
    // 若没有后继结点,返回 NULL
    if (next = = NULL)
        return NULL;
    // 使当前结点指向 tempPtr 的后继结点
    next = tempPtr->next;
    // 返回被删除结点的指针
    return tempPtr;
}
```

9.2 构造链表

我们将用 Node 类构造链表,在此过程中将介绍大多数应用程序中所用到的基本链表算法。本节内容对于理解链表是很重要的。你可以借此学会如何建立表以及如何访问结点。我们将以独立的函数实现链表算法。掌握这些内容对你用上述技术建立通用链表类是很有帮助的。为方便起见,讨论中均假定表结点中保存的是整型数据。

链表以一个指向表头的结点指针开始。我们称之为头指针(head),因为它指向表头。初始情况下,头指针值为 NULL,表示它指向空表。

可以以不同的方法建立链表。首先我们将介绍将每个新结点都放在表头的方法,然后再考虑将结点加入到表尾或中间位置的情况。

生成结点

我们用基于模板的函数 GetNode 实现结点的生成,该函数需要一个初始数据值和一

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

个指针值，以动态地分配一个新的结点。

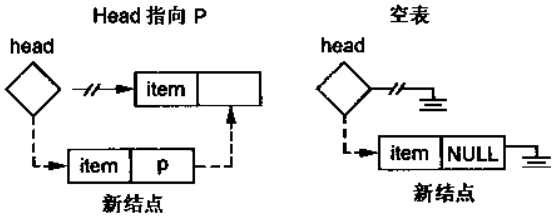
```
// 创建一个结点，数据值为 item，指针为 nextPtr
template <class T>
Node<T> *GetNode(const T& item, Node<T> *nextPtr = NULL)
{
    Node<T> *newNode;
    // 为新结点申请内存，并将参数传入。若失败，则退出程序
    newNode = new Node<T>(item, nextPtr);
    if (newNode == NULL)
    {
        cerr << "Memory allocation failure!" << endl;
        exit(1);
    }
    return newNode;
}
```

插入结点：INSERTFRONT

将结点插入到表前面的操作需要对头指针重新赋值，因为有了新的表头。如何维护表头是表管理中的一个基本问题。倘若失去表头则会丢失整个表！

在插入开始以前，头指针指向表头；插入以后，新结点将占据表头，而旧表头将占据第二个位置。因此，新结点的指针域被赋予当前头指针的值，而头指针则被赋予新结点的地址。此赋值是通过使用 GetNode 生成新结点的方法来完成的。

```
head = GetNode(item,head);
```



函数 InsertFront 需要以下参数：表的当前头指针，它定义了表的指针；以及新的数据值。函数将数据结点插入到表的前面。因为头指针在操作中要被修改，所以它是作为引用参数被传递的。

```
// 往表头插入结点
template <class T>
void InsertFront(Node<T> * & head, T item)
{
    // 申请新结点，并使其指向原表头，再修改原表头
    head = GetNode(item, head);
}
```

此函数以及 GetNode 函数均包含在 Node 库文件“nodelib.h”中。

链表的遍历

任何遍历算法的起始点都是头指针，因为它标明了表的开始。当在表中移动时，我们

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

使用指针 CurrPtr 来指向当前的位置。初始情况下，CurrPtr 被设为指向表头：

```
currPtr = head;
```

在扫描过程中，我们想访问当前位置的数据域(data)。因为 data 是公有成员，所以我们可以取得其值或给其赋以新值。

```
currentDataValue = currPtr->data;
currPtr->data = newData;
```

例如，遍历过程可以包括一条简单的 cout 语句以打印每个结点的值。

```
cout << currPtr->data;    // 取数据值并输出
```

扫描中，我们连续将 CurrPtr 移到下一个结点直到到达表尾。我们用函数 NextNode 确定表中的下一个结点。

```
currPtr = currPtr->NextNode();
```

表的遍历在 CurrPtr 值为 NULL 时终止。例如，函数 PrintList 打印每个结点的数据值(data)。头指针被作为参数传递，因为它定义了表。第 2 个参数是用户定义类型的 AppendNewLine，它表示输出是否跟以两个空格或回车符。该函数包含于文件"nodelib.h"中。

```
enum AppendNewline {noNewline, addNewline};
// 输出链表
template <class T>
void PrintList(Node<T> *head, AppendNewline addnl)
{
    // 用 currPtr 指针从表头开始遍历表
    Node<T> *currPtr = head;
    // 输出当前结点的数据，直到表结束
    while(currPtr != NULL)
    {
        // 当 addl == addNewline 时输出换行符
        if (addnl == addNewline)
            cout << currPtr->data << endl;
        else
            cout << currPtr->data << " ";
        // 指向下一结点
        currPtr = currPtr->NextNode();
    }
}
```

程序 9.1　匹配键值

程序在 1 到 10 的范围内产生 10 个随机数并用 InsertFront 将这些值作为结点插入到链表的表头。我们用 PrintList 显示该表。

程序的一部分代码用于计算表中键值出现的次数。用户首先被提示输入键值。然后遍历程序将键值与每个表结点中的数值(data)域相比较。出现的总次数将被打印出来。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
#include <iostream.h>
#include "node.h"
#include "nodelib.h"
#include "random.h"
void main(void)
{
    // 将表的头指针置为 NULL
    Node<int> *head = NULL, * currPtr;
        int i, key, count = 0;
        RandomNumber rnd;
    // 往表头中插入 10 个随机整数结点
    for (i=0; i < 10; i++)
        InsertFront(head, int(1+rnd.Random(10)));
    // 输出原始表
    cout << "List:";
    PrintList(head, noNewline);
    cout << endl;
    // 提示用户输入键值
    cout << "Enter a key:";
    cin >> key;
    // 遍历表
    currPtr = head;
    while (currPtr != NULL)
    {
        // 若结点数据与键值相等,则计数器加 1
        if (currPtr->data == key)
            count++;
        // 移向表中下一结点
        currPtr = currPtr->NextNode();
    }
    cout << "The data value" << key << "occurs" << count
         << "times in the list" << endl;
}
/*
<程序 9.1 运行结果>
List: 3  6  5  7  5  2  4  5  9  10
Enter a key: 5
The data value 5 occurs 3 times in the list
*/
```

插入结点：INSERTREAR

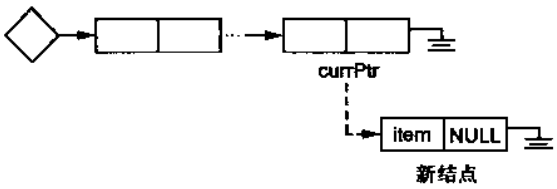
将结点放到表尾需要先测试一下表是否为空。若是，则生成一新结点，其指针域为NULL，并将其地址赋给头指针；整个操作由 InsertFront 来实现；对于非空表，则必须扫描表结点以定位表尾结点。当前对象的 next 域为 NULL 时，我们即可标注此位置。

```
currPtr->NextNode() == NULL
```

插入操作是通过首先生成一个新结点(GetNode)，然后将其插入到当前 Node 对象后(In-

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



sertAfter)而实施的。

因为插入可能改变头指针的值,所以头指针被作为引用参数传递。

```
// 寻找表尾并往其后加入值为 item 的结点
template < class T >
void InsertRear(Node < T > * & head, const T& item)
{
    Node < T >  * newNode, * currPtr = head;
// 若表为空,则在表头插入 item
if (currPtr = = NULL)
    InsertFront(head, item);
else
{
    // 找到指针指向 NULL 的结点
    while(currPtr - > NextNode() ! = NULL)
        currPtr = currPtr - > NextNode();
        // 创建新结点并插入到表尾
        newNode = GetNode(item);
        currPtr - > InsertAfter(newNode);
    }
}
```

InsertRear 包含于库文件“nodelib.h”中。

程序 9.2 乱字

此程序随机地混合单词中的字以产生一个“乱字谜”。处理过程为扫描串中的每个字符并随机地将其放到表的头部或尾部。对每个字符调用 random(2)。如果返回值为 0,则调用 InsertFront;否则调用 InsertRear。例如,对于输入 jumble 以及随机数序列 011001,结果表为“lbjume”。程序读入并混合 4 个单词。

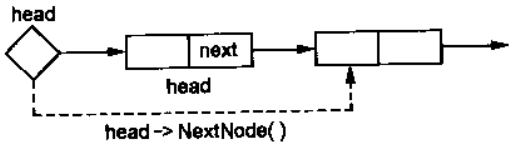
```
# include < iostream.h >
# include "random.h"
# include "strclass.h"
# include "nodelib.h"
void main(void)
{
    // 存放乱字的结点表
    Node < char >  * jumbleword = NULL;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 输入串,随机数对象,计数器
    String s;
    RandomNumber rnd;
    int i, j;
    // 读入 4 个串
    for (i = 0; i < 4; i++)
    {
        cin >> s;
        // 用 Random(2)来决定将字符放入表头(value = 0)还是表尾(value = 1)
        for (j = 0; j < s.Length(); j++)
            if (rnd.Random(2))
               InsertRear(jumbleword, s[j]);
            else
               InsertFront(jumbleword, s[j]);
    // 输出读入的串及得到的乱字
    cout << "String/Jumble: " << s << "  ";
    PrintList(jumbleword);
    cout << endl << endl;
    }
}
/*
<程序 9.2 运行结果>
pepper
String/Jumble: pepper  r  p  p  e  p  e
hawaii
String/Jumble: hawaii  i  i  h  a  w  a

jumble
String/Jumble: Jumble  e  b  m  j  u  l
C++
String/Jumble: C++  +  C  +
*/
```

删除结点 这节中,我们已经讨论了扫描表并插入新结点的算法。第 3 种表操作,从表中删除结点,引入了一套新的机制。我们经常想删除表中第 1 个结点。这一操作要求我们更新表的头指针,将其指向头结点的后继。



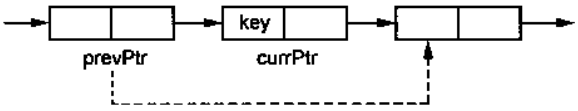
设计函数 DeleteFront,表头指针作为引用参数被传递给它。函数将第 1 个结点从链上断开并释放其内存。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 删除表中第一个结点
template <class T>
void DeleteFront(Node<T> * & head)
{
    // 保存指向被删除结点的指针
    Node<T> *p = head;
    // 确认该表非空
    if (head ! = NULL)
    {
        // 将头指针 head 指向原表的第二个结点
        head = head->NextNode();
        delete p;
    }
}
```

通用的删除函数将在表中搜索并删掉第 1 个 data 值与键值匹配的结点。将 prevPtr 初始化为 NULL,且 CurrPtr 初始化为头指针(head)。将 CurrPtr 沿表移动以查找匹配值并维护 prevPtr 以使它指向 CurrPtr 的前一个位置。



指针 prevPtr 和 CurrPtr 前后相随沿表移动,直到 CurrPtr 变成 NULL 或 CurrPtr 标明了一次匹配(CurrPtr->data==key)。

```
while(currPtr ! = NULL && currPtr->data ! = key)
{
    // 将 prevPtr 前移至 currPtr
    prevPtr = currPtr;
    // 将 currPtr 前移 1 个结点
    currPtr = currPtr->NextNode();
}
```

如果退出"while"语句时 CurrPtr! = NULL,则发生了一次匹配。这时当前位置即为要删除的结点。有两种可能:若 prevPtr 为 NULL,则删除表的第 1 个结点;否则,通过对结点 prevPtr 执行 DeleteAfter 操作对结点进行删除。

```
if (prevPtr = = NULL)
    head = head->NextNode();
else
    prevPtr->DeleteAfter();
```

如果没有找到匹配键值,则 Delete 方法什么也不做就返回。因为删除表头需要更新头指针,所以以引用传递参数。

```
// 删除表中第一个与键值相等的结点
template<class T>
void Delete (Node<T> * & head, T key)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{
    // currPtr 遍历表,prevPtr 紧随其后
    Node<T> *currPtr = head, *prevPtr = NULL;
    // 若表为空,则返回
    if (currPtr == NULL)
        return;
    // 遍历表,直到找到上键值相等的结点或已到表尾
    while(currPtr != NULL && currPtr->data != key)
    {
        // currPtr 指针前移,并用 prevPtr 跟随
        prevPtr = currPtr
        currPtr = currPtr->NextNode();
    }
    // 若 currPtr != NULL,则 currPtr 处找到键值
    if (currPtr != NULL)
    {
        // prevPtr == NULL 表示在头结点找到键值
        if (prevPtr == NULL)
           head = head->NextNode();
        else
            // 在第二个或三后的结点
            prevPtr->DeleteAfter();
            // 删除结点
            delete currPtr;
    }
}
```

应用程序：毕业生名单

记录 StudentRecord 给出即将大学毕业的学生的名字和 GPA(平均分数)。我们的目的是建立一个参加毕业典礼的学生表。候选毕业的学生表从文件"studrecs"读入且被插入到表的前面。因为大学校规不允许 GPA 低于 2.0 的学生毕业,所以我们扫描此表,将那些不满足 GPA 最低要求的候选学生删掉。第 2 个表从文件"noattend"读入,代表那些选择不参加典礼的学生。这些名字是按每行一个给出的,它们用来删除毕业生表中多余的名字。剩余项则代表那些获准毕业且准备参加毕业典礼的学生表。

```
struct StudentRecord
{
    String name;
    float gpa;
}
```

我们在类中包含了重载的输入和输出运算符以读写学生记录。重载的"!="运算符允许对学生数据使用 Delete 函数。

程序 9.3　毕业生名单

读入文件"studrecs",用 Node 库中的函数 InsertFront 将每条记录插入到链表 gradu-

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

ateList 的前面。指针 prevPtr 和 currPtr 用来扫描名单，并将那些 GPA 低于 2.0 的学生删掉。删除所有不合格学生后，从文件“noattend”读入名字，并从名单中将这些学生删掉。最后，将那些参加毕业典礼的毕业生名单打印出来。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <iomanip.h>

#include "node.h"
#include "nodelib.h"
#include "studinfo.h"

void main(void)
{
    Node<StudentRecord> *graduateList = NULL,
                        *currPtr, *prevPtr,
                        *deleteNodePtr;
    StudentRecord srec;
    ifstream fin;

    fin.open("studrecs",ios::in | ios::nocreate);
    if (! fin)
    {
        cerr << "Cannot open file studrecs." << endl;
        exit(1);
    }

    // 用一位小数形式输出 GPA 值
    cout.setf(ios::fixed);
    cout.precision(1);
    cout.setf(ios::showpoint);

    while(fin >> srec)
        // 将 srec 插入表头
        InsertFront(graduateList,srec);

    // 删除 GPA < 2.0 的学生
    prevPtr = NULL;                          // prevPtr 紧随 currPtr
    currPtr = graduateList;                  // currPtr 从表头开始
    while (currPtr != NULL)                  // 遍历到表尾
    {
        if (currPtr->data.gpa < 2.0)         // 该学生能毕业吗?
        {
            if (prevPtr == NULL)             该学生在表头吗?
            {
                graduateList = currPtr->NextNode();
                deletedNodePtr = currPtr;
                currPtr = graduateList;
            }
            else    // 删除表中的学生
            {
                currPtr = currPtr->NextNode();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
                deletedNodePtr = prevPtr->DeleteAfter();
            }
            delete deletedNodePtr;  // 清除被删除的结点
        }
        else
        {
            // 不删除,指针后移
            prevPtr = currPtr;
            currPtr = currPtr->NextNode();
        }
    }
    fin.close();
    fin.open("noattend",ios::in | ios::nocreate);
    if (! fin)
    {
        cerr << "Cannot open file noattend." << endl;
        exit(1);
    }
    // 从文件中读出将不参加毕业典礼的学生并从队列中删除
    while(srec.name.ReadString(fin) ! = -1)
        Delete(graduateList,srec);
    // 打印所有参加毕业典礼的学生名单
    cout << "Students attending graduation:" << endl;
    PrintList(graduateList,addNewline);
}
/*
<文件"studrecs">
Julie Bailey
1.5
Harold Nelson
2.9
Thomas Frazer
3.5
Bailey Harnes
1.7
Sara Miller
3.9
Nancy Barnes
2.5
Rebecca Neeson
4.0
Shannon Johnson
3.8
<文件"noattend">
Thomas Frazer
Sara Miller
<程序 9.3 运行结果>
Students attending graduation:
Shannon Johnson 3.8
Rebecca Neeson 4.0
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

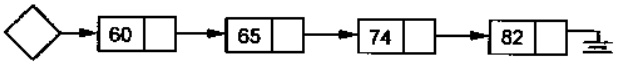
```
Nancy Barnes 2.5
Harold Nelson 2.9
*/
```

建立有序表

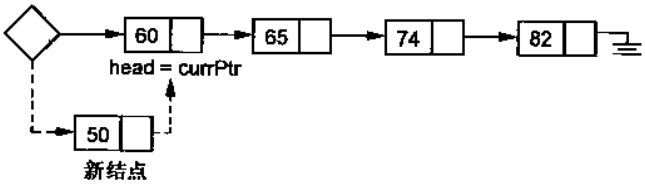
许多应用程序中，我们希望维持一个有序的数据表，其结点按升序或降序排列。插入算法首先必须扫描表以找到加入新结点的正确位置。下面的讨论演示了以升序建立表的过程。

要想加入数值 X，我们首先对表进行扫描并将 currPtr 定位在第 1 个其 data 值比 X 大的结点上。带有值 X 的新结点应该被插入到 currPtr 左边。在扫描过程中。prevPtr 随 currPtr 而移动，但它总是指向前一个位置的记录。

下面的例子对算法进行了演示。假设开始时表 L 中包含有整数 60，65，74 和 82。

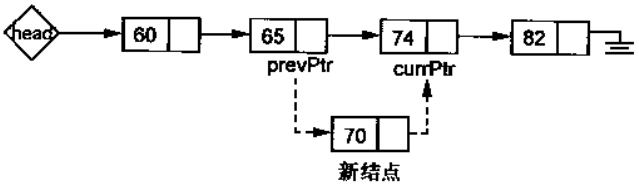


在表中插入 50：因为 60 是表中第 1 个比 50 大的结点，所以将 50 插入到表头位置。




```
InsertFront (head, 50);
```

在表中插入 70：74 是表中第 1 个比 70 大的结点，指针 prevPtr 和 currPtr 分别指向结点 65 和 74。



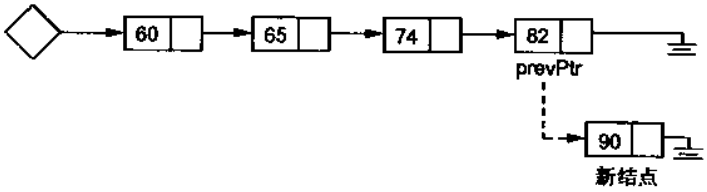

```
newNode = GetNode(70);
prevPtr->InsertAfter(newNode);
```

在表中插入 90：扫描完整个表但没有发现大于 90 的结点（currPtr==NULL）。新数值大于等于表中所有值，因此新结点必须放在表尾。扫描终止后，将新结点插入到 prevPtr 后面。

下述函数实现通用的有序插入算法。如果表中有 n 个元素，最糟糕的情况是新元素被插入到表尾。这种情况下，必须进行 n 次比较；因此最坏情况复杂度是 O(n)。平均计算，预计搜索到表长一半时即会找到插入点。结果，平均复杂度为 O(n)。当然最好的情况为 O(1)。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
            newNode = GetNode(90);
            prevPtr->InsertAfter(newNode);
// 往有序表中增加结点
template <class T>
void InsertOrder(Node<T> * & head, T item)
{
    // currPtr 遍历表,prevPtr 随它移动
    Node<T> *currPtr, *prevPtr, *newNode;
    // prevPtr == NULL 表示在表头匹配
    prevPtr = NULL;
    currPtr = head;
    // 遍历表并找到插入点
    while (currPtr != NULL)
    {
        // 若 item < 当前结点数据,则找到插入点
        if (item < currPtr->data)
            break;
        // currPtr 前移一个结点,prevPtr 紧随着移动
        prevPtr = currPtr;
        currPtr = currPtr->NextNode();
    }
    // 插入
    if (prevPtr == NULL)
        // 若 prevPtr == NULL, 则在表头插入
        InsertFront(head, item);
    else
    {
        // 在前一结点之后插入新结点
        newNode = GetNode(item);
        prevPtr->InsertAfter(newNode);
    }
}
```

应用程序：用链表排序

InsertOrder 可以用来对一组数据项进行排序，前提是为数据类型 T 定义比较运算符。函数 LinkSort 以一含 n 个元素的数组 A 为参数，将这些元素插入到一个有序的链表中。遍历新表，将排好序的元素拷贝回数组中。调用函数 ClearList 将释放表中每个结点所占内存。函数对表进行遍历，记录下每个结点的地址并移到下一个结点，然后再删除掉原来的结点。ClearList 是库“nodelib.h”的成员。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 清除链表中所有结点
template <class T>
void ClearList(Node<T> * &head)
{
    Node<T> *currPtr, *nextPtr;
    // 遍历链表并删除所有结点
    currPtr = head;
    while(currPtr != NULL)
    {
        // 记录下一结点的指针,删除前结点
        nextPtr = currPtr->NextNode();
        delete currPtr;
        // 当前结点前移
        currPtr = nextPtr;
    }
    // 置表为零
    head = NULL;
}
```

程序 9.4　表插入排序

该程序用模板函数 LinkSort 对给定的 10 元素整数数组 A 进行排序。最后得到的有序数组用 PrintArray 打印出来。

```
#include <iostream.h>
#include "node.h"
#include "nodelib.h"
template <class T>
void LinkSort(T a[], int n)
{
    // 建立有序链表 ordlist 来存放数组元素
    Node<T> *ordlist = NULL, *currPtr;
    int i;
    // 按序将元素从数组插入表中
    for (i=0; i < n; i++)
        InsertOrder(ordlist, a[i]);
    // 扫描表并将数据拷回数组
    currPtr = ordlist;
    i = 0;
    while(currPtr != NULL)
    {
        a[i++] = currPtr->data;
        currPtr = currPtr->NextNode();
    }
    // 删除所有为有序表创建的结点
    ClearList(ordlist);
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 扫描数组并输出其所有元素
void PrintArray(int a[], int n)
{
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
}
void main(void)
{
    // 对 10 个整数值排序
    int A[10] = {82, 65, 74, 95, 60, 28, 5, 3, 33, 55};
    LinkSort(A,10);                          // 将数组排序
    cout << "Sorted array:    ";
    PrintArray(A,10);                        // 输出数组
    cout << endl;
}
/*
<程序 9.4 运行结果>
Sorted array:    3  5  28  33  55  60  65  74  82  95
*/
```

对 LinkSort 算法的运行效率的分析必须考虑到数组元素的初始顺序。最糟糕的情况是表已经以升序排列，这时每个元素都将被插入到表尾。第 1 次插入不需要比较；第 2 次插入进行 1 次比较；第 3 次进行 2 次比较…，以此类推。总的比较次数为：

$$0+1+2+\cdots+(n-1)=\frac{(n-1)*n}{2}$$

其复杂度为 $O(n^2)$。另一个极端是已经按降序排列的表仅需进行 n－1 次比较，因为每个数组元素都被插入到表的前面。因此，最佳情况为 O(n)，而最差情况是 $O(n^2)$。直观看，平均情况下，在 n 次插入中第 i 次预计要进行 i/2 次比较。总比较次数是 $O(n^2)$。

与那些按位置排序的算法如 ExchangeSort 所不同的是，LinkSort 需要额外的存储空间以存放链表的 n 个数据元素以及指针。另外将元素拷贝到或拷贝出链表时还要花费时间。

9.3 设计链表类

程序员可以使用"nodelib.h"中的 Node 类以及实用函数以处理链表应用程序。该方法迫使程序员生成每个结点并且直接进行低级表操作。一种更结构化的方法则是定义一种链表类，其中基本的表操作是作为成员函数定义的。在这一节中，我们用设计结点类的算法时所获取的知识来讨论链表类中应包括的数据成员和操作的种类。我们还期望链表类可被用来实现其他表集合，包括链栈、队列以及 SeqList 类。下一节我们将定义 LinkedList 类及其成员函数。

链表的数据成员

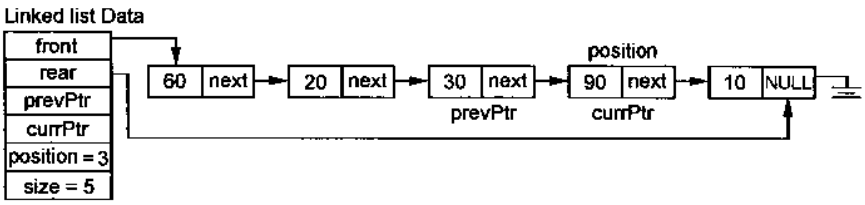
链表是由一组 Node 对象从表头到表尾串在一起而组成的。表的起始结点称为表头，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

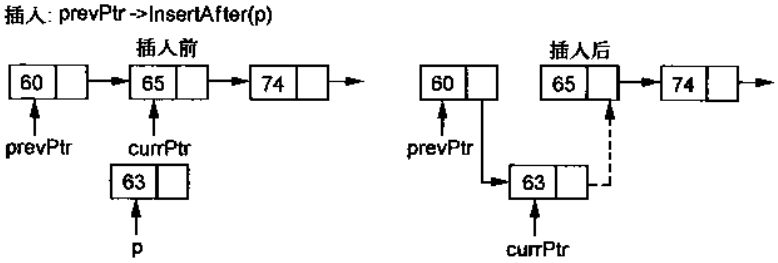
www.Linuxidc.com

它表示表的第 1 个结点。表的最后一个结点的指针域的值为 NULL,我们用表尾指针指向新结点。我们的目的是让链表类维护指向表头和表尾的指针,因为这对于许多应用程序是很有用的,并且对于实现链表队列也是很关键的。

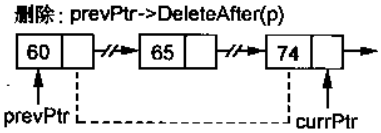
链表可以提供对数据项的顺序访问并且用指针指示当前遍历位置。我们的链表中含有指向当前位置的指针 currPtr,以及伴随指针 prevPtr,它指向前一个位置。另外我们还维持一个叫 position 的变量,它根据其在表中的位置描述当前位置。表头的 position 为 0,下一个为 1,……依次类推。链表中的元素个数由变量 size 来维持。这样我们就可以标明一个空表或返回表中元素的计数。下表中,当前位置结点 90,其 position 为 3:



地址 prevPtr 用于在当前位置插入和删除一个结点,这只要用 Node 方法 InsertAfter 和 DeleteAfter 即可。例如,下面的链表中,我们演示了使用 prevPtr 指向的 Node 对象的一次插入过程。



从链表中删除一项时,也要用到 prevPtr 所指向的结点:



链表操作

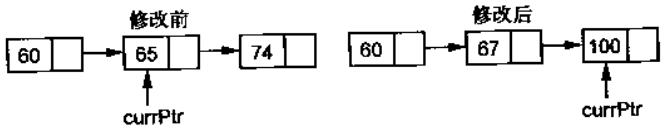
用户必须能够在表的元素间移动。一个名为 Next 的简单方法能把当前位置向下一个结点推进,这就使我们能够检索和修改结点的数据域而不需要知道数据是如何存储在 Node 类中的。为了说明这些操作,假设 L 是一个整数链表,其当前位置在 data 值为 65 的结点处。下面的语句将当前结点的值改为 67,而下一个结点的值改为 100:

```
LinkedList < int > L;
      ...
if (L.Data() < 70)     // 将当前结点的值与 70 作比较
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

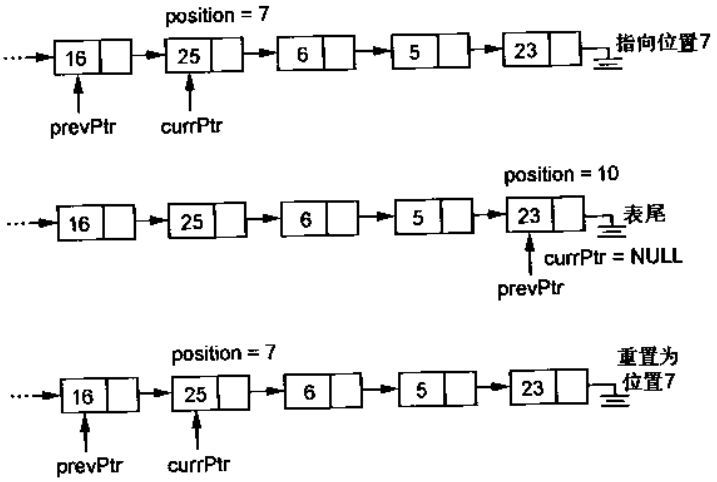
```
    L.Data() = 67;      // 若小于,则赋以新值 67
L.Next();               // 前进到下一结点
L.Data() = 100;         // 将该结点值改为 100
```



应用程序中,有时需要将当前位置设在表中特定位置处。Reset 方法就可以做到这一点。它以 pos 为参数,将当前表位置移到 pos 处。pos 的缺省值为 0,这时将当前位置设为表头。应用程序可以从此位置开始用 Next 扫描所有结点。当条件 EndOfList 为真时扫描终止。例如,用一简单的循环打印表项。在扫描表之前,先测试一下 ListEmpty 条件:

```
L.Reset();                          // 使 currPtr 指向表头
if (L.ListEmpty())                  // 检查是否空表
    cout < < "Empty list \ n";
else
    while(! L.EndOfList())          // 扫描全表
    {
        cout < < L.Data() < < endl;  // 输出数据值
        L.Next();                   // 指向下一结点
    }
```

客户程序可以用 CurrentPosition 方法访问当前的表位置。如果在表中移动了很远,则可以用 Reset 方法恢复原始的表位置。此项功能可用于诸如查找表中的最大值以及在表中定位一个结点以便进行插入或删除这样一些问题中。




```
// 保存当前位置
int currPos = L.CurrentPosition();
< 扫描表的右部的指令 >
// 重置当前指针指向上次的 currPos
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
L.Reset(currPos);
```

增加和删除结点是链表的基本操作。操作可以发生在表头、表尾或当前位置。

插入操作 插入操作产生一个带有新的数据域的新结点。结点将被放在表的当前位置或紧随当前位置之后。

InsertAfter 操作将新结点放在当前位置之后且将 currPtr 指向新的结点。该操作与 Node 类中的 InsertAfter 作用相同。

InsertAt 方法将新结点放在当前位置。新结点将放在当前结点之前且与之相邻。当前位置被置为新的结点。该操作用于建立一个有序表。

我们的链表类中还提供了 InsertFront 和 InsertRear 操作以在表头和表尾增加新结点。这些操作都将当前位置设到新结点处。

删除操作 删除操作把结点从表中删除掉。DeleteAt 将当前位置处结点删掉，而 DeleteFront 则删掉表中第 1 个结点。

```
例:     Linked List < int > L;
        L.InsertFront(100);        // 表为 100
        L.InsertAfter(200);        // 表为 100  200
        L.InsertAt(300);           // 表为 100  300  200
        L.InsertRear(50);          // 表为 100  300  200  50
        L.Reset(1);                // 将 currPtr 指向
        L.DeleteAt();              // 表为 100  200  50
        L.DeleteAt();              // 表为 100  50
        L.DeleteAt();              // 表为 100;
```

其他方法 链表类要建立动态数据，所以它必须具有复制构造函数、析构函数以及重载的赋值运算符。用户可以用 ClearList 操作显式地清空一个表。

9.4 类 LinkedList

本节引入类 LinkedList，将其作为用于动态表处理的简单而实用的工具箱。侧重点在于类描述以及示范其使用的样本程序。类定义和实现包含于文件“link.h”中。

LinkedList 类的说明

声明

```
# include < iostream.h >
# include < stdlib.h >
# include "node.h"
template < class T >
class LinkedList
{
    private:
        // 指向表头和表尾的指针
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    Node<T> *front, *rear;
    // 用于数据访问、插入和删除的指针
    Node<T> *prevPtr, *currPtr;
    // 表中元素的个数
    int size;
    // 表中位置值,用于 Reset
    int position;
    // 申请及释放结点的私有函数
    Node<T> *GetNode(const T& item, Node<T> * ptrNext = NULL);
    void FreeNode(Node<T> *p);
    // 将表 L 拷贝到当前表
    void CopyList(const LinkedList<T>& L);
public:
    // 构造函数
    LinkedList(void);
    LinkedList(const LinkedList<T>& L);
    // 析构函数
    ~LinkedList(void);
    // 赋值运算符
    LinkedList<T>& operator= (const LinkedList<T>& L);
    // 检查表状态的函数
    int ListSize(void) const;
    int ListEmpty(void) const;
    // 遍历表的函数
    void Reset(int pos = 0);
    void Next (void);
    int EndOfList(void) const;
    int CurrentPosition(void) const;
    // 插入函数
    void InsertFront(const T& item);
    void InsertRear(const T& item);
    void InsertAt(const T& item);
    void InsertAfter(const T& item);
    // 删除函数
    T DeleteFront(void);
    void DeleteAt(void);
    // 访问/修改数据
    T& Data(void);
    // 清空表的函数
    void ClearList(void);
};
```

讨论

本类使用了动态内存,所以需要复制构造函数、析构函数以及重载的赋值运算符。私

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

有方法 GetNode 和 FreeNode 完成类中的所有内存分配工作。如果内存分配失败，GetNode 终止程序。

本类中保存有表长，它可由 ListSize 和 ListEmpty 方法进行访问。

私有数据成员 currPtr 和 prePtr 记录了表的当前遍历位置的信息。而插入和删除操作则负责在操作完成后更新这两个值。Reset 方法则显示设置 currPtr 和 prePtr 的值。

本类在遍历方法上具有灵活性。Reset 以一个位置为参数并将当前位置设在该处。Reset 的缺省值为 0，因此当它不带变元时，当前位置将被设在表头。Next 方法推进到表中下一个结点，而 EndOfList 则指示是否已到表的末端。对于表 L，用一个 for 循环可以扫描整个表：

```
for (L.Reset(); ! L.EndOfList(); L.Next())
    <访问当前结点>
```

函数 CurrentPosition 在遍历过程中返回表的当前位置。以后若要访问当前结点，则可以保存返回值并将其作为参数传递给 Reset。

用 InsertFront 和 InsertRear 可以在表的两端插入。InsertAt 在表的当前位置处插入新结点，而 InsertAfter 则在当前位置后插入结点。如果当前位置在表的末尾（EndOfList = = True），那么 InsertAt 和 InsertAfter 都将新结点放到表尾。

DeleteFront 删除表中第 1 个元素，而 DeleteAt 则删除当前位置处的结点。无论哪种方法，如果试图从空表中删除结点都会导至程序终止。

方法 Data 用来读取或修改表的当前位置处的数据。因为 Data 返回的是对结点中数据的引用，所以它可用在赋值语句的左、右两边。

```
// 取当前结点的数据值并将其加 5
L.Data() = L.Data() + 5;
```

ClearList 删除表中所有结点且将表标为空。

例

```
LinkedList < int > L, K;          // 定义整型表 L,K
// 往表中加入 25 个整数
for(i = 0; i < 25; i + + )
{
    cin > > num;
    L.InsertRear(num);            // 按输入顺序加入结点
    K.InsertFront(num);           // 逆输入逆序加入结点
}
// 扫描表 L 并将每个结点值修改为其绝对值
for(L.Reset();! L.EndOfList();L.Next())
    // 若数据为负则改为其正数
    if (L.Data() < 0)
      L.Data() = - L.Data();

K.InsertFront(100);               // 经 K 的表头加入结点 100
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
K.InsertAfter(200);          // 在结点 100 后加入结点 200
K.InsertAt(150);             // 在结点 100 和结点 200 间加入结点 150

// 输出表 L
void PrintList(LinkedList < int > & L)
{
    // 从表头开始遍历表 L 并打印各元素的值
    for (L.Reset();! L.EndOfList();L.Next())
        cout < < L.Data() < < " ";
}
```

在实现 LinkedList 类之前，我们将用一系列例子演示其主要特性。首先介绍连接函数，将一个表附加到另一个表之后从而实现两个表的连接。第 2 个例子则用链表实现选择排序。经典的算法是对数组按位置排序。我们用链表的插入和删除操作来实现它。本节最后给出的是删除表中重复结点的算法。

连接两个表

函数 ConcatLists 按顺序将第 2 个表中的结点插入到第 1 个表的表尾而实现两个表的连接。此函数被用于补充程序中的"concat.cpp"中。

函数 ConcatLists 扫描第 2 个表并用 Data 方法从每个结点上取出数据值。然后再使用方法 InsertRear 将含有该数值的新结点附加到第 1 个表的表尾。

```
template < class T >
// 将 L2 连到 L1 的表尾
void ConcatLists(LinkedList < T > & L1, LinkedList < T > & L2)
{
    // 重置两个表指针到表头
    L1.Reset();
    L2.Reset();
    // 遍历 L2,并每个数据值插入到 L1 的表尾
    while (! L2.EndOfList())
    {
        L1.InsertRear(L2.Data());
        L2.Next();
    }
}
```

表排序

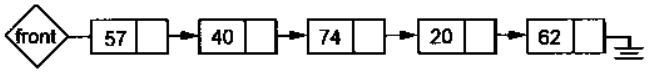
用两个独立的链表就可以用链表法实现选择排序。第 1 个表 L 中含有一组未排序的数据；第 2 个表 K 是作为表 L 的副本而建立的，但它的数据是排好序的。(排序)算法按从大到小的顺序从表 L 中删除各元素并将它们插入到表 K 的前面，最后表 K 就变成了有序表。选择排序需要对表进行重复扫描。我们用函数 FindMax 来扫描表并将当前位置设到最大元素所在处。从该位置取出数据值后，我们用 InsertFront 将具有该值的新结点插入到表 K 的前面，并用 DeleteAt 从表 L 中删除这个最大值结点。

请看下面的例子：

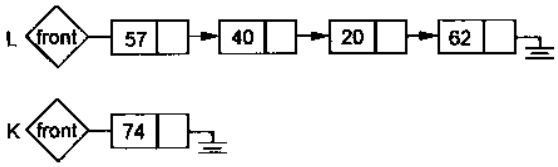
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

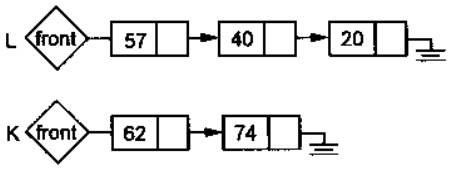
表 L 中含有元素:57,40,74,20,62



步骤 1: 找到最大值 74,从表 L 中删除它,并将其加入到 K 中。



步骤 2: 找到最大值 62,将其从表 L 中删除掉,并加入到 K 中。



程序 9.5 表选择排序

程序建立一个内含 10 个随机数的表 L,随机数的范围在 0 到 99 之间。我们先用函数 PrintList 打印出初始表,然后再用选择排序算法将表 L 中的元素转到 K 中。最后再次调用 PrintList 以输出 K 的值,而 K 中的元素已按升序排列。

```
#include <iostream.h>
#include "link.h"                    // 引入链表类
#include "random.h"
// 定位表 L 中最大元素
template <class T>
void FindMax(LinkedList<T> &L)
{
    if (L.ListEmpty())
    {
        cerr << "FindMax: list empty!" << endl;
        return;
    }
    // 重置指针到表头
    L.Reset();
    // 将位置 0 的值置为当前最大值
    T max = L.Data();
    int maxLoc = 0;
    // 扫描全表
    for (L.Next(); ! L.EndOfList(); L.Next())
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

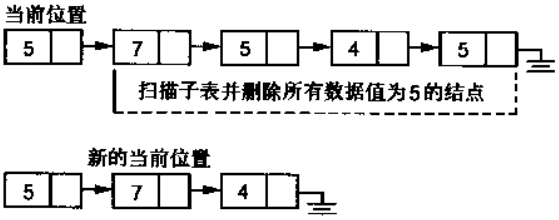
```
        if (L.Data() > max)
        {
            // 新的最大值,记录其值及在表中的位置
            max = L.Data();
            maxLoc = L.CurrentPosition();
        }
    // 将指针指向最大值
    L.Reset(maxLoc);
}
// 输出表 L
template < class T >
void PrintList (LinkedList < T > & L)
{
        // 从表头开始遍历表并输出各结点值
        for (L.Reset(); L.EndOfList(); L.Next())
            cout < < L.Data() < < "  ";
}
void main(void)
{
        // 将表 L 中各结点在表 K 中升序排序
        LinkedList < int > L, K;
        RandomNumber rnd;
        int i;
        // L 为从 0 ~ 99 中选出的 10 个随机整数组成的表
        for (i = 0; i < 10; i + + )
            L.InsertRear(rnd.Random(100));
        cout < < "Original list: ";
        PrintList(L);
        cout < < endl;
        // 从 L 中删除结点,直到其为空,将这些结点插入 K
        while (! L.ListEmpty())
        {
            // 在剩余元素中求最大值
            FindMax(L);
            // 将最大值结点插入表 K 后从表 L 中删除
            K.InsertFront(L.Data());
            L.DeleteAt();
        }
    cout < < "Sorted list:    ";
    PrintList(K);
    cout < < endl;
}
/*
< 程序 9.5 运行结果 >
Original list: 82  72  62  3  85  33  58  50  91  26
Sorted list:    3  26  33  50  58  62  72  82  85  91
*/
```

删除重复值 LinkedList 类的一个有趣应用是从表中删除重复值的算法。我们先建

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

立一个表 L,然后开始扫描其结点。对每个结点,我们都记录下它的位置以及其数据值。这样我们就有了一个可以在表的其余部分寻找其重复值的键值以及在删除重复值以后可以返回的位置。从当前位置开始,一直扫描到表尾,删除所有其数据值与键值匹配的结点。然后再将遍历位置重新设回到原始值的位置,向前推进一个结点以继续整个操作过程。



程序 9.6　删除重复值

本程序使用上述算法删除重复值。初始表中含有 15 个其值在 1 ~ 7 之间的随机数。程序先打印出表来,然后调用函数 RemoveDuplicates,以删除表中重复值。最后再次调用函数 PrintList 以打印出结果表。函数 PrintList 包含在文件"link.h"中。

```
# include <iostream.h>

# include "link.h"                    // 引入类 LinkedList
# include "random.h"

void RemoveDuplicates(LinkedList < int > & L)
{
    // 当前表位置及数据值
    int currPos, currValue;

    // 指针指向表头
    L.Reset();

    // 遍历表
    while(! L.EndOfList())
    {
        // 记录当前结点的数据值及其位置
        currValue = L.Data();
        currPos = L.CurrentPosition();

        // 移到下一结点
        L.Next();

        // 移到表尾,删除所有具有 currValue 的结点
        while(! L.EndOfList())
            // 若该结点被删除,当前位置为下一结点
            if (L.Data() = = currValue)
                L.DeleteAt();
            else
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            L.Next();              // 移到下一结点
        // 移动具有 currValue 值的第一个结点后,再移到下一结点
        L.Reset(currPos);
        L.Next();
    }
}
void main(void)
{
    LinkedList < int > L;
    int i;
    RandomNumber rnd;
    // 从 1~7 中随机生成 15 个整数插入表中,并输出该表
    for (i = 0; i < 15; i + +)
        L.InsertRear(1 + rnd.Random(7));
    cout < < "Original list: ";
    PrintList(L);
    cout < < endl;
    // 删除所有重复值并输出新表
    RemoveDuplicates(L);
    cout < < "Final list:    ";
    PrintList(L);
    cout < < endl;
}
/*
 < 程序 9.6 运行结果 >
Original list: 1  7  7  1  5  1  2  7  2  1  6  6  3  6  4
Final list:    1  7  5  2  6  3  4
 */
```

9.5 LinkedList 类的实现

LinkedList 类的描述中引用了 Node 类。LinkedList 的实现中用到 9.1 节中所讨论的 Node 类的许多技术。例如,在链表前端插入和删除结点的算法也适用于 LinkedList 类。"nodelib.h"中的函数所用的算法是我们研究 LinkedList 类的基础。当然我们还应意识到复杂程度增大了,因为要维持可以访问表头和表尾的头指针和尾指针,还有记录当前遍历位置有关信息的指针 currPtr 和 prePtr,以及整数位置值和表长等。LinkedList 的方法在表的状态发生变化时负责更新这些值。

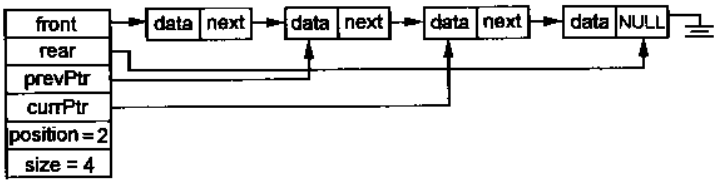
私有数据成员 类限制对这类数据的访问,因为它们仅供成员函数使用。链表是由一组 Node 对象从表头到表尾串在一起的。我们将头指针作为数据成员。为方便在表尾插入,类中设有一个表尾指针指向表尾结点。这样就省去了扫描整个表以找出表尾的时间。变量 size 保存的是表中的结点个数。它可用于确定表是否为空,并返回表中数据的个数。变量 position 可使 Reset 方法中对当前遍历位置的重新定位变得方便一些。

LinkedList 对象中有两个指针用来标明表中的当前位置(currPtr)以及前一个位置

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(prePtr)。指针 currPtr 指向表中的当前结点。它可用于 Data 方法以及插入方法 InsertAfter。指针 prePtr 用于方法 DeleteAt 以及 InsertAt,它们都对当前位置进行操作。当插入和删除都完成时,类更新表对象的 front,rear,position 和 size 域。



内存分配方法 类为客户提供所有插入和删除服务。复制构造函数和插入方法生成结点,而 ClearList 和删除方法则注销结点。LinkedList 类能在这些方法中直接使用运算符 new 和 delete。当然,函数 GetNode 和 FreeNode 所提供的是更结构化的内存管理访问。

GetNode 方法试图以给定的数据值和指针域动态地分配一个结点。如果分配成功,函数返回指向新结点的指针;否则函数打印出错信息并终止程序。FreeNode 方法仅需释放由结点占用的内存。

构造函数和析构函数 构造函数建立一个空表,所有指针值均设为空。在此初始状态下,size 被置为 0 而 position 的值被置为 -1。

```
// 创建空表,使其指针指向 NULL,size 置为 0,position 置为 -1
template <class T>
LinkedList<T>::LinkedList(void): front (NULL), rear(NULL),
      prevPtr(NULL),currPtr(NULL),size(0),position (-1)
{}
```

复制构造函数以及赋值运算符复制 LinkedList 对象 L。为此,类中实现了方法 CopyList,它遍历表 L 并将每个数据值插入到当前表的表尾。仅当当前表为空时才调用私有函数。它将遍历参数 prevPtr、currPtr 和 position 的配置设置为与表 L 中的相同。这样,经过赋值和初始化后,这两个表具有相同的遍历状态。

```
// 将 L 拷贝到一初始为空的当前表中
template <class T>
void LinkedList<T>::CopyList(const LinkedList<T>& L)
{
    Node<T> *p = L.front;        // 用指针 p 遍历表 L
    // 往当前表的表尾插入 L 的每个元素
    while (p != NULL)
    {
        InsertRear(p->data);
        p = p->NextNode();
    }
    // 若表为空则返回
    if (position == -1)
        return;
    // 在新表中重置 prevPtr 及 currPtr
    prevPtr = NULL;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        currPtr = front;
        for (int pos = 0; pos ! = position; pos + + )
        {
            prevPtr = currPtr;
            currPtr = currPtr - > NextNode();
        }
    }
```

ClearList 使用 9.1 节中研究的算法遍历链表并注销所有结点。析构函数只需调用 ClearList 即可实现。

```
template < class T >
void LinkedList < T > ::ClearList(void)
{
    Node < T >  * currPosition, * nextPosition;
    currPosition = front;
    while(currPosition ! = NULL)
        {
            // 取下一结点指针并删除当前结点
            nextPosition = currPosition - > NextNode();
            FreeNode(currPosition);
            currPosition = nextPosition;    // 移到下一结点
        }
    front = rear = NULL;
    prevPtr = currPtr = NULL;
    size = 0;
    position = -1;
}
```

表遍历方法 Reset 将当前遍历位置设到由参数 pos 所指定的位置。同时它还将更新 currPtr 和 prevPtr 的值。如果 pos 不在 0 至 size - 1 范围内，则打印出错消息，且程序终止。为设置 currPtr 和 prevPtr，函数区分 pos 是表头位置以及在表中间的情况。

pos = = 0：重置当前位置到表的前端，具体做法是将 prevPtr 设为 NULL，currPtr 指向表头而 position 为 0。

pos! = 0：既然 pos = = 0 的情况已经考虑过，那么可以假设 pos 值比 0 大且表遍历必须到表的中间位置。为重新定位 currPtr，从表的第 2 个结点开始，移向 pos 位置处。

```
    // 将表位置置到 pos
    template < class T >
    void LinkedList < T > ::Reset(int pos)
    {
        int startPos;
        // 若表为空，则返回
        if (front = = NULL)
            return;
        // 若位置非法，退出程序
        if (pos < 0 || pos > size - 1)
        {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        cerr << "Reset: Invalid list position: " << pos
            << endl;
        return;
    }
    // 遍历表置 pos 结点
    if (pos == 0)
    {
        // 重新指向表头
        prevPtr = NULL;
        currPtr = front;
        position = 0;
    }
    else
    // 重置 currPtr,prevPtr 及 position
    {
        currPtr = front->NextNode();
        prevPtr = front;
        startPos = 1;
        // 右移直到 position == pos
        for(position=startPos; position != pos; position++)
        {
            // 将两个指针右移
            prevPtr = currPtr;
            currPtr = currPtr->NextNode();
        }
    }
}
```

表的顺序扫描过程是通过执行方法 Next 从表中一个元素移到另一个元素的。函数将 prevPtr 移到当前结点而将 currPtr 再向下移动一个结点。如果已遍历完表中所有结点，则变量 position 的值为 size 而 currPtr 的值则为 NULL。

```
// 将 prevPtr 和 currPtr 指针右移一个结点
template <class T>
void LinkedList<T>::Next(void)
{
    // 若已到表尾或表为空,返回
    if (currPtr != NULL)
    {
        // 将两个指针右移一个结点
        prevPtr = currPtr;
        currPtr = currPtr->NextNode();
        position++;
    }
}
```

数据存取 用 Data 方法可以存取表结点中的数据。如果表为空或遍历已经到达表的末端，则打印出错信息且程序终止。否则，Data 返回 currPtr->data。

```
// 返回当前结点的数据值
template <class T>
T& LinkedList<T>::Data(void)
{
```

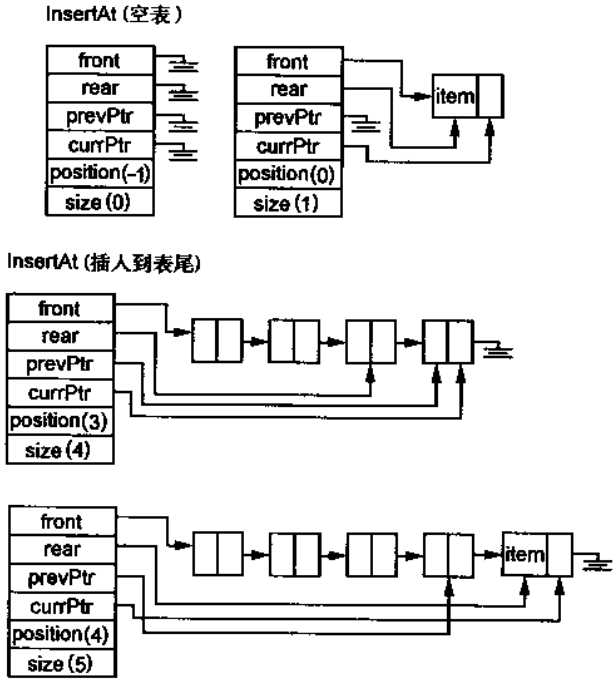

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 若表为空或已到表尾,则出错
    if (size = = 0 || currPtr = = NULL)
    {
        cerr < < "Data: invalid reference!" < < endl;
        exit(1);
    }
    return currPtr - > data;
}
```

表插入方法 LinkedList 类中提供了一系列在表头或表尾增加结点(InsertFront, InsertRear)或者在以当前位置为参考的相对位置处插入结点(InsertAt 和 InsertAfter)的方法。插入方法需要一个参数值以初始化新结点的 data 域。

InsertAt 将具有数据值的结点插入到表的当前位置。该方法用 GetNode 分配一个结点,它拥有一个数据值,地址为 NewNode。该算法必须处理两种情况。如果插入发生在表的前端(prevPtr = = NULL),则将表头指针更新为指向新结点。如果插入发生在表的中间,要往空表中或在非空表的末尾插入表项则必须考虑对指针 rear 的特殊处理。




```
// 往表的当前位置插入结点
template < class T >
void LinkedList < T > ::InsertAt(const T& item)
{
    Node < T >  * newNode;
    // 两种情况: 往表头插入或往表中插入
    if (prevPtr = = NULL)
    {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 往表头插入,包括往空表中插入
        newNode = GetNode(item, front);
        front = newNode;
    }
    else
    {
        // 往表中插入,在 prevPtr 后插入结点
        newNode = GetNode(item);
        prevPtr->InsertAfter(newNode);
    }
    // 若 prevPtr == rear, 表示往空表中插入或为空表;应修改 rear 及 position 值
    if (prevPtr == rear)
    {
        rear = newNode;
        position = size;
    }
    // 改变 currPtr 及增加表的大小
    currPtr = newNode;
    size++;              // 增加表大小
}
```

表删除方法 两个删除操作分别从表头(DeleteFront)和当前位置(DeleteAt)删除结点。

DeleteAt 删除地址 currPtr 处的结点。如果 currPtr 为 NULL,则表为空或客户程序已经遍历完整个表。在此情况下,函数输出一条出错消息并终止程序。如果不是这样,算法要处理两种情况。如果要删除的是表中的第 1 个结点(prevPtr == NULL),则更新指针 front;如果删除表中最后一个结点,则 front 变为 NULL。第二种情况是当 prevPtr≠NULL 且要删除的结点在表的中间时,则使用 Node 类的方法 DeleteAfter 将 prevPtr 后的结点从表中拆开。

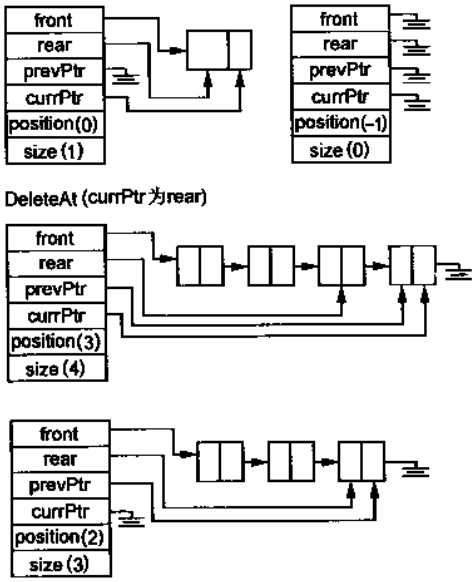
与 InsertAt 方法一样,我们要特别留心指针 rear。如果我们删除的结点在尾部(currPtr == rear),则新的 rear 值变为 prevPtr,position 值减小,而 currPtr 则为 NULL。在其他所有情况下,position 保持不变。如果删除表中最后一个结点,则 rear 变为 NULL,而 position 的值从 0 变为 -1。调用 FreeNode 可以从内存中释放结点并减小表长。

```
// 删除表中当前结点
template <class T>
void LinkedList<T>::DeleteAt(void)
{
    Node<T> *p;
    // 若表为空或已到表尾,则出错退出
    if (currPtr == NULL)
    {
        cerr << "Invalid deletion!" << endl;
        exit(1);
    }
    // 被删除的必是头结点或表中结点
    if (prevPtr == NULL)
    {
        // 保存表头指针并删除它
        p = front;
        front = front->NextNode();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    }
    else
        // 删除 prevPtr 之后的中间结点,并保存其地址
        p = prevPtr->DeleteAfter();
    // 若表尾被删除,则 prevPtr 为新表尾且 position 减 1;否则,position 不变。若 p 是是
       后结点,则 rear = NULL 且 position = -1
    if (p == rear)
    {
        rear = prevPtr;
        position--;
    }
    // 将 currPtr 指向下一结点。若 p 为表中最后结点,则 currPtr 为 NULL
    currPtr = p->NextNode();
    // 释放结点空间并将表大小减 1
    FreeNode(p);
    size--;
}
```



9.6 用链表实现集合

到现在为止,在本书中,我们已经使用了基于数组的类 Stack,Queue 以及 SeqList。在上述情况下,表项都存放在作为类的私有数据成员定义的数组中。本章中我们研究了可提供功能强大的动态存储结构以及增删表项或更新数据值的许多方法。因为可以将表项存储在链表对象而不是数组中,所以我们就有了可增强基本表类功能的各种方法。通过使用 LinkedList 类中的各种方法,我们就拥有了以直接方式实现栈、队列以及表操作的工具。

本节中,我们研究用链表实现 Queue 和 SeqList 类。对于类 SeqList,我们将比较动态链

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

表存储和和数组存储的效能。

研究打印缓冲池模拟时将用到类 Queue。Stack 类的链表实现将被留作作业。

链式队列

LinkedList 对象在存储一组表项时具有灵活的存储结构。类 Queue 通过 LinkedList 对象的复合方法直接实现了队列。上述对象通过执行 LinkedList 操作而完成与之等价的 Queue 操作。例如，LinkedList 对象允许在表尾插入表项(InsertRear)以及从表头删除表项(DeleteFront)。通过将当前指针重新定位到表头(Reset)，我们可以定义从表头直接获取数据值的操作(QFront)。其他队列方法判断表的状态，而这些工作可由表操作 ListEmpty 和 ListSize 来完成。要清空队列只需调用表方法 ClearList 即可。

Queue 类的说明(用 LinkedList 对象)

声明

```
#include "link.h"

template <class T>
class Queue
{
    private:
        // 存放队列元素的链表对象
        LinkedList<T> queueList;
    public:
        // 构造函数
        Queue(void);

        // 改变队列的方法
        void QInsert(const T& elt);
        T QDelete(void);

        // 存取队列元素
        T QFront(void) const;

        // 检测队列状态
        int QLength(void) const;
        int QEmpty(void) const;
        void QClear(void);
};
```

说明

LinkedList 对象 queueList 中存有各队列项，该对象提供了一套完整的链表操作以用于实现公有 Queue 方法。

Queue 类中没有自己的析构函数，复制构造函数以及赋值运算符，因为它们已经由 queueList 对象实现了。编译器通过执行对象 queueList 的赋值运算符或复制构造函数来实现赋值运算和初始化操作。queueList 的析构函数在 Queue 对象被注销时，将被自动调用。

因为表项存储在链表中，所以队列的长度不必受限于某一个诸如 MaxQueueSize 之类

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

的常数。

例

```
Queue < int > Q1,Q2;          // 定义两个值为整数的队列
Q1.QInsert(10);               // 往 Q1 中加入 10,然后加入 50
Q1.QInsert(50);
cout < < Q1.QFront();         // 输出表头中的元素 10

Q2 = Q1;                      // 使用 queueList 的赋值运算符 =
Q1.QClear();                  // 清除队列并释放空间
```

用链表实现的类 Queue 包含于文件“queue.h”中。

Queue 方法的实现

为演示 Queue 方法的实现,我们定义队列修改方法 QInsert 和 QDelete 以及存取方法 QFront,这些方法都直接调用了等价的 LinkedList 方法。

QInsert 操作使用 LinkedList 操作 InsertRear 将新表项加到队列的尾部。

```
// LinkedList 方法 InsertRear 往队尾中插入元素
template < class T >
void Queue < T > ::QInsert(const T& elt)
{
    queueList.InsertRear(elt);
}
```

QDelete 首先检查队列的状态,若是空表则终止;否则调用表操作 DeleteFrcnt 将表中第 1 项拆下,释放其内存并返回数据值。

```
// LinkedList 方法 DeleteFront 从队首中删除元素
template < class T >
T Queue < T > ::QDelete(void)
{
  // 若队列为空,则退出程序
  if (queueList.ListEmpty())
  {
      cerr < < "Calling QDelete for an empty queue!" < < endl;
      exit(1);
  }
  return queueList.DeleteFront();
}
```

QFront 操作从 queueList 第 1 个元素中取出数据值,这需要将当前指针定位到表头并读出其数值。试图对空表调用此函数时将会产生出错消息并使程序终止。

```
// 返回队列中首元素的数据值
template < class T >
T Queue < T > ::QFront(void)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{
    // 若队列为空,退出程序
    if (queueList.ListEmpty())
    {
        cerr << "Calling QFront for an empty queue!" << endl;
        exit(1);
    }
    // 指针置回队首并返回队首元素值
    queueList.Reset();
    return queueList.Data();
}
```

链式 SeqList 类

SeqList 类定义了一种有严格限制的存储结构,它仅允许将表项插入到表尾,删除表中第 1 项或与键值匹配的项。客户程序访问表中数据的方法是用 Find 方法或 position 索引读出结点中的数值。与链表队列相似,在实现 SeqList 类时也可以使用 LinkedList 对象保存数据。同时,该对象还可以提供实现类方法的强有力的操作工具箱。

SeqList 类定义

声明

```
#include "link.h"

template <class T>
class SeqList
{
    private:
        // 链表对象
        LinkedList<T> llist;

    public:
        // 构造函数
        void SeqList(void);

        // 访问表的方法
        int ListSize(void) const;
        int ListEmpty(void) const;
        int Find (T& item);
        T GetData(int pos);

        // 改变表的方法
        void Insert (const T& item);
        void Delete(const T& item);
        T DeleteFront(void);
        void ClearList(void);
};
```

说明

类方法与“aseqlist.h”中的那些基于数组的版本中所定义的相同。不需要定义析构函

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

数、复制构造函数，以及赋值运算符。编译器通过使用 LinkedList 类中的相应操作生成它们。此类在文件"seqlist.h"中。

例

```
SeqList < int > chList;        // 申请一个整型动态表
chList.Insert(40);             // 往表尾加入 40
cout < < chList.DeleteFront () < < endl;      // 输出 40
```

SeqList 数据存取方法实现

SeqList 类允许用户用 Find 方法加上一个键值访问数据或按表中位置访问数据。第一种情况下，我们用 LinkedList 类的遍历机制来扫描整个表并搜索键值。

```
// 将 item 作为键值搜索表.若表中存在该 item 值则返回 True,否则返回 False
template < class T >
int SeqList < T > ::Find (T& item)
{
    int result = 0;
    // 在表中搜索 item.若找到,则将 result 置为 True
    for (llist.Reset();! llist.EndOfList();llist.Next())
        if (item = = llist.Data())
        {
            result + + ;
            break;
        }
        return result;
}
```

GetData 按数据元素在表中的位置来访问它。用 LinkedList 方法 Reset 来在表中所需位置建立遍历机制，并执行 Data 方法以取出数据值。

```
// 返回位于 pos 位置的数据值
template < class T >
T SeqList < T > ::GetData(int pos)
{
    // 检查 pos 是否合法
    if (pos < 0 || pos > = llist.ListSize())
    {
        cerr < < "post is out of range!" < < endl;
        exit(1);
    }
    // 将当前链表的 position 置为 pos 并返回数据
    llist.Reset(pos);
    return llist.Data();
}
```

应用程序：比较 SeqList 的实现方法

基于数组的 SeqList 类在删除表项时需要较多的开销，因为表尾部的所有元素必须左移。而如果用链表实现，则只需断开指针即可完成相同的操作。为演示使用链表存储结构的好处，我们将基于数组的 SeqList 类与基于链表的 SeqList 类作一比较。程序先建立一

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

个含 500 个成员的初始表，然后重复从表头删除表项并插入到表尾。这一过程重复50 000次且代表了基于数组的 SeqList 对象的最糟糕的情况。我们在同一计算机系统上运行两个等价的程序并统计出执行 50 000 次插入/删除操作所需的秒数。

程序 9.7a （表类——数组实现）

本测试用例使用了文件"aseqlist.h"中的基于数组的类 SeqList，为了本程序之用，特地将常数 ARRAYSIZE 改为 500 次以允许更多表项。整个过程耗时 55 秒。

```
#include <iostream.h>
// 用 DataType = int 往表中存放整型值
typedef int DataType;
// 引入基于数组的类 SeqList
#include "aseqlist.h"
void main(void)
{
    // 可存放 500 个整数的表
    SeqList L;
    long i;
    // 用 0..499 初始化表
    for (i = 0; i < 500; i++)
        L.Insert(i);
    // 执行删除/插入操作 50000 次
    cout << "Program begin!" << endl;
    for (i = 1; i <= 50000; i++)
    {
        L.DeleteFront();
        L.Insert(0);
    }
    cout << "Program done!" << endl;
}
<程序 9.7a 运行结果>
Program begin!
Program done!        // 55 秒
```

程序 9.7b （表类——链表实现）

本程序测试"seqlist1.h"中的基于链表的类 SeqList。整个过程耗时 4 秒！

```
#include <iostream.h>
// 引入用 SeqList 实现的链表
#include "seqlist1.h"
void main(void)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{
    // 定义整型表
    SeqList < int > L;
    long i;
    // 用 0..499 初始化表
    for (i = 0; i < 500; i + + )
        L.Insert(i);
    // 执行删除/插入操作 50000 次
    cout < < "Program begin!" < < endl;
    for (i = 1; i < = 50000; i + + )
    {
        L.Delete Front();
        L.Insert(0);
    }
    cout < < "Program done!" < < endl;
}
```

< 程序 9.7b 运行结果 >

```
Program begin!
Program done!        // 4 秒
```

9.7 实例研究：打印缓冲池

操作系统中可以用队列实现打印缓冲池。打印缓冲池接受打印请求并将待打印文件插入到队列中。当打印机可用时，缓冲池从队例中删除作业并将文件打印出来。缓冲池的工作使得打印可以在后台进行而用户仍可执行前台进程。

问题分析

本例中编写了一个缓冲池类，其操作模拟用户往打印队列中加入新作业并检查已经在队列中的作业的状态的过程。打印作业是一个结构，它包含一个整数表编号、文件名以及页计数。

```
struct PrintJob
{
    int number;
    char filename[20];
    int pagesize;
};
```

模拟过程以每分种 8 页的速度连续打印。

下面的时间表示意用户往打印缓冲池发送了 3 个作业。

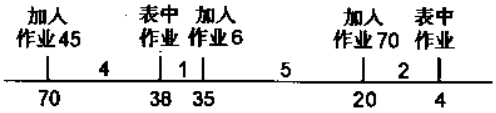
作业	名称	页数
45	论文	70
6	信件	5
70	通知	20

在 12 分钟时间内，用户将 3 个作业加到队列中并两次请求列出队列中的作业。4，1，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

5,2 各单元代表用户操作之间相隔的时间。数值 70,38,35,20 和 4 则指示用户操作的各个时刻未打印完的页数。



未打印页数

图 9.1 中跟踪了打印缓冲池从 0 到 12 分钟之间的操作。在每次事件发生处,列出缓冲池中的页数,已打印完的总页数以及打印队列。

程序设计

打印缓冲池是一存储 PrintJob 记录的表。因为作业是依据先来先服务的原则而处理的,所以我们将表作为队列看待,即作业请求被插到表尾,而表头的作业请求则被删去,并打印出来。在此实例中,我们要完成正规队列集合所不提供的操作。我们将扫描表中的作业,请求并打印出它们的状态。在更新操作中,我们将修改当前作业的长度,但不从队列中删除它。

本实例中用事件来导引整个模拟过程。事件可以是将打印作业加到缓冲池中、列出缓冲池中的作业,以及检查某特定作业是否还未打印等。事件以随机选择的时间片隔开,时间片的长度在 1 到 5 分钟之间。为模拟后台连续打印作业的过程,实例中用事件的发生来更新打印队列。

事件之间的时间间隔长度可以用来计算已经打印出多少页。假设自上一次事件以来的时间间隔是 deltaTime,则已打印出的页数是:

```
pagesPrinted = deltaTime * 8
```

时间(分钟)	事件	作业页数	缓冲池中作业页数	已打印页数
0	加入作业	70	70	0
	打印队列	45 论文 70		
4	列出作业		38	32
	打印队列	45 论文 38		
5	加入作业 6	5	35	40
	打印队列	45 论文 30	6 信件 5	
10	加入作业 70	20	20	75
	打印队列	70 通知 20		

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

12	列出作业	4	91

打印队列	70	通知	4

图 9.1 跟踪缓冲池操作

打印作业的存储结构以及缓冲池存取函数是由以下的 Spooler 类定义的。

Spooler 类描述

声明

```
#include "random.h"              // 模拟打印时间
#include "link.h"                // 引入 LinkedList 类

const int PRINTSPEED = 8;        // 每分钟打印页数

// 打印缓冲池类
class Spooler
{
    private:
        // 存放打印作业及状态的队列
        LinkedList < PrintJob > jobList;

        // deltaTime 存放范围 1 至 5 的随机数来模拟时间片
        int deltaTime;
        // 用 UpdateSpooler 方法修改作业信息
        void UpdateSpooler(int time);
        RandomNumber rnd;

    public:
        // 构造函数
        Spooler(void);

        // 将作业加入缓冲池
        void AddJob(PrintJob J);

        // 输出缓冲池状态的方法
        void ListJobs(void);
        int CheckJob(int jobno);
        int NumberOfJobs(void);
};
```

说明

私有数据成员 deltaTime 模拟自上次缓冲池事件发生以来打印过程所进行的分钟数。每次事件开始时都要以 deltaTime 为参数调用操作 UpdateSpooler，该函数更新 jobList 以表明打印已经在后台进行了 deltaTime 分钟。公有方法负责给组 deltaTime 赋新值，该值由 RandomNumber 生成器以 1 到 5 为范围生成，它表示在下一次更新事件之前所经历的分钟数。

打印作业由方法 AddJob 加入到缓冲池中。ListJob 和 CheckJob 这两个操作提供有关

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

缓冲池的状态的信息。在任何时刻都可调用 ListJobs 打印出缓冲池中作业清单。CheckJob 方法接受一个作业号并返回其在缓冲池中的状态信息。该函数返回未打印完的页码数，若已打完则返回 0。

NumberOfJobs 返回未打印的作业的计数。PrintJob，PRINTSPEED 以及缓冲池类的定义包含在文件“spooler.h”中。

Spooler 更新方法的实现

更新过程删除累计页数小于已打印页数的那些作业。如果要打印的总页数小于或等于已打印总页数，那么所有作业都已完成且打印队列被清空；否则从队列中删除一个或多个作业且当前打印作业的部分页已被打印出来。更新留下了当前作业中未被打印的部分。

```
// 更新操作。假定在打印当中进行，该方法删除已打完的作业并修改当前作业的剩余页数
void Spooler::UpdateSpooler(int time)
{
    PrintJob J;
    // 可在给定时间内打印的页数
    int printedpages = time * PRINTSPEED;
    // 根据 printedpages 值及队列中的作业数为作业更新打印队列
    jobList.Reset();
    while (! jobList.ListEmpty() && printedpages > 0)
    {
        // 重找第一个作业
        j = jobList.Data();
        // 若已打印页数大于作业的页数，更新已打印页数记数并删除该作业
        if (printedpages >= J.pagesize)
        {
            printedpages -= J.pagesize;
            jobList.DeleteFront();
        }
        // 部分作业完成；更新剩余页数
        else
        {
            J.pagesize -= printedpages;
            printedpages = 0;
            jobList.Data() = J;              // 更新结点中信息
        }
    }
}
```

缓冲池状态判断方法

缓冲池状态判断方法的作用是响应客户请求给出有关等待打印的作业以及特定作业的状态的信息。ListJobs 和 CheckJob 方法对缓冲池表进行顺序扫描。我们将演示 ListJobs 算法并给读者介绍文件“spooler.h”中的其他方法。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

ListJobs 从表头开始(Reset),一个结点一个结点往下走(Next),一直到表尾(EndOfList)。(最后)输出每一个 PrintJob 的信息。

```
// 更新缓冲池并列出池中当前所有作业
void Spooler::ListJobs(void)
{
    PrintJob J;
    // 更新打印队列
    UpdateSpooler(deltaTime);
    // 产生下一事件发生的时间
    deltaTime = 1 + rnd.Random(5);
    // 扫描队列之前检查是否为空池
    if (jobList.ListSize() = = 0)
        cout < < "Print queue is empty \ n";
    else
    {
        // 从表头开始扫描作业队列直到表尾,打印每个作业的有关信息
        for (jobList.Reset(); ! jobList.EndOfList();
                        jobList.Next())
        {
            J = jobList.Data();
            cout < < "Job" < < J.number < < ": " < < J.filename;
            cout < < "    " < < J.pagesize < < " pages remaining"
                 < < endl;
        }
    }
}
```

程序 9.8　打印缓冲池

main 程序中定义了一个 Spooler 对象 spool,并建立一个与用户交互动作的对话框。在每一轮操作中,可供用户选择的是一含 4 个选项的菜单。选项'A'(AddJob),'L'(ListJob)和'C'(CheckJob)更新打印队列并执行缓冲池操作。选项'Q'终止程序。若打印队列为空则不列出选项'L'和'C'。

```
# include < iostream.h >
# include < ctype.h >
# include "spooler.h"
void main(void)
{
// 打印池对象
Spooler spool;
int jnum, jobno = 0, rempages;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
char response = 'C';
PrintJob J;
for (;;)
{
    // 用户选项,只有当作业存在时,才有'C'选项
    if (spool.NumberOfJobs() ! = 0)
        cout < < "Add(A) List(L) Check(C) Quit(Q) = = > ";
    else
        cout < < "Add(A) Quit(Q) = = > ";
    cin > > response;
    // 将用户回答转换为大写字母
    respond = toupper(response);
    // 每种回答对应的动作
    switch(response)
    {
        // 将一新作业加入队列中,并使作业号加 1;读入作业的文件名及页数
        case 'A':
            J.number = jobno;
            jobno + + ;
            cout < < "File name: ";
            cin > > J.filename;
            cout < < "Number of pages: ";
            cin > > J.pagesize;
            spool.AddJob(J);
            break;
        // 输出留在队列中作业的信息
        case 'L':
            spool.ListJobs();
            break;
        // 输入作业号: 用它作键值来检索整个队列,输出该作业是否完成或尚未打印的页数
        case 'C':
            cout < < "Enter job number: ";
            cin > > jnum;
            rempages = spool.CheckJob(jnum);
            if (rempages > 0)
                cout < < "Job is in the queue. " < < rempages
                     < < "pages remain to print" < < endl;
            else
                cout < < "Job has completed" < < endl;
            break;
        // 若输入为'Q',退出 switch 及 for 循环
        case 'Q':
            break;
        // 输入错误,则输出出错信息,并重现菜单
        default:
            cout < < "Invalid spooler command. \ n";
            break;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

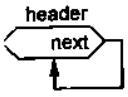
```
    }
    if (response = = 'Q')
        break;
    cout < < endl;
  }
}

/*
<程序 9.8 运行结果>
Add (A)    Quit (Q) = = > a
File name: notes
Number of pages: 75
Add (A)    List (L)    Check (C)    Quit (Q) = = > a
File name: paper
Number of pages: 25
Add (A)    List (L)    Check (C)    Quit (Q) = = > 1
Job 0: notes    19 pages remaining
Job 1: notes    25 pages remaining
Add (A)    List (L)    Check (C)    Quit (Q) = = > c
Enter job number: 1
Job is in the queue. 20 pages remain to be printed
Add (A)    List (L)    Check (C)    Quit (Q) = = > 1
Print queue is empty
Add (A)    Quit (Q) = = > q
*/
```

9.8 循环表

一个以 NULL 终止的链表是一个以头结点开始、以 NULL 指针域结尾的结点序列。在 9.1 节中，我们编写了一组用来扫描这样一个表并插入和删除结点的一组函数。每种算法用于检测的代码的实际复杂度都要增加，因为必须确定表是否为空或要增加更新表头指针的代码。在本节中，我们研究表的另一种模型——循环链表，它可以简化顺序表算法的设计和编码。许多专业程序员都使用循环模型实现链表。

一个空链表中包含一个结点，它有一个未经初始化的数据域。该结点叫 header，初始时它指向自己。header 的作用是指向表中第 1 个"真正的"结点，因此 header 常被称为"哨位"结点。在循环型的链表中，空表实际上包含一个结点，NULL 指针是用不上的。我们在结点的侧边用折线示意 header。

注意对于标准链表和循环链表，检测表是否为空的方法是不同的。

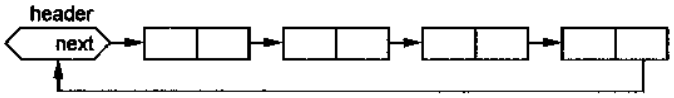


标准链表：　head = = NULL
循环链表：　header - > next = = header

结点被加入到表中以后，最后一个结点指向 header 结点。我们可以将循环链表看成一个手镯，其 header 结点作为搭扣。header 结点将表中的真结点拴在一起。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



在 9.1 节中，我们描述了 Node 类并用其方法建立链表。本节中，我们定义 CNode 类，它用于建立循环表。该类提供缺省的构造函数，可以对数据域不作初始化。此构造函数用来生成 header。

循环节点类定义

声明

```
template < class T >
class CNode
{
    private:
        // 指向下一结点的循环指针
        CNode < T >  * next;

    public:
        // data 为公共成员
        T data;

        // 构造函数
        CNode(void);
        CNode(const T& item);

        // 更新表的方法
        void InsertAfter(CNode < T >  * p);
        CNode < T >  * DeleteAfter(void);

        // 保存下一结点的指针
        CNode < T >  * NextNode(void) const;
};
```

说明

此类与 9.1 节中的 Node 类相似。实际上，所有数据成员具有相同的名称和功能。公有成员的细节将在下一小节类实现时给出。循环节点类包含于文件“Conde.h”中。

循环结点类的实现

构造函数初始化结点时将该结点指向它自己，因此每个结点都可以作为一个空表的表头(header)。“自己”就是指针 this。因此赋值语句就变得简单了：

```
next = this;        // 下一结点为结点本身
```

缺省的构造函数对数据域不作初始化。第 2 个构造函数带一个参数并用它初始化数据域。

两个构造函数都不需要用来指定 next 域初始值的参数。对 next 域所要作的任何改变都要用到 InsertAfter 或 DeleteAfter 方法。

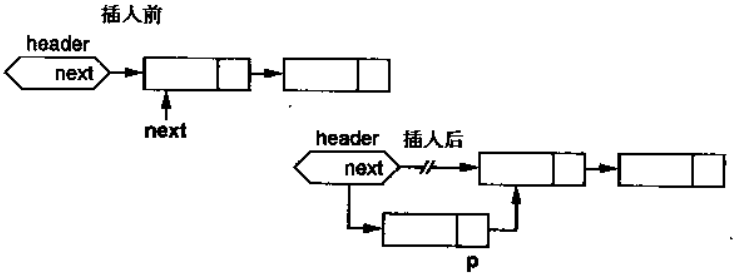
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 产生空表并初始在数据的构造函数
template < class T >
CNode < T > ::CNode(const T& item)
{
    // 将结点指向自身并初始化数据
    next = this;
    data = item;
}
```

循环结点操作 循环结点类提供用于表遍历的 NextNode 方法。与 Node 类方法相似，NextNode 返回指针值 next。

InsertAfter 将结点 p 加入到紧挨在当前对象的位置上。不需要用一个特殊的算法将结点装入到表头，因为仅需执行 InsertAfter(header)。哨卫结点或头结点的存在使得技术上困扰表处理的特殊情况不复存在。




```
// 在当前结点之后插入结点 p
template < class T >
void CNode < T > ::InsertAfter(CNode < T >  * p)
{
    // p 指向当前结点的后继结点，当前结点指向 p
    p - > next = next;
    next = p;
}
```

从表中删除结点的操作则是由 DeleteAfter 方法完成的。DeleteAfter 方法将紧接在当前结点之后的结点删除并返回指向已删除结点的指针。如果 next = = this，则表中没有其他结点了，结点不能删除自己。在这种情况下，操作返回 NULL 值。

```
// 删除当前结点之后的结点，并返回指向其的指针
template < class T >
CNode < T >  * CNode < T > ::DeleteAfter(void)
{
    // 保存指向被删除结点的指针
    CNode < T >  * tempPtr = next;

    // 若 next 指针为 this，表明没有其他结点，返回 NULL
    if (next = = this)
        return NULL;

    // 当前结点指向 tempPtr 的后继结点
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    next = tempPtr->next;
    // 返回指向被删除结点的指针
    return tempPtr;
}
```

应用程序:解决 Josephus 问题

循环链表的一个有趣的应用是用它可以干脆利落地解决 Josephus 问题。以下是该问题的一种版本:

一旅行社选择 n 个客人参加一次竞赛,胜者可免费周游世界。旅行社让客人围成一个圆圈并从帽子中随机抽取一个数 m。游戏方法是沿圆圈顺时针方向数客人,每数到 m 时便停下,让这里的竞赛者出局,游戏接着进行,如此这番继续下去,直到只剩一个人。这个幸存者就获得了周游世界的资格。

例如,如果 n=8 且 m=3,则参赛者的出局顺序是 3,6,1,5,2,8,4,第 7 个人会赢得比赛。

程序 9.9 Josephus 问题

本程序模拟游世界竞选,函数 CreateList 用 CNode 方法 InsertAfter 建立循环表 1,2,…,n。

选择过程由函数 Josephus 完成,它要求的参数之一是循环表的头结点,另一个是随机数 m。函数重复进行 n-1 次操作,每次连续数 m 个表项并删除第 m 项。在表中转圈时,打印出每个出局者的号码。循环结束时只剩下一项。主程序输入参赛者的数目,并用 CreateList 建立循环表。程序生成一个 1~n 范围内的随机数 m,调用 Josephus 以决定参赛者出局的顺序以及游世界的获胜者。

```
#include <iostream.h>
#include "cnode.h"
#include "random.h"
// 用给定的头结点产生整型循环链表
void CreateList(CNode<int> *header,int n)
{
    // 开始往头结点后插入
    CNode<int> *currPtr = header, *newNodePtr;
    int i;
    // 建立起 n 元循环链表
    for (i=1; i <= n; i++)
    {
        // 用数据值 i 申请结点空间
        newNodePtr = new CNode<int>(i);
        // 往表尾插入结点并将 currPtr 指针前移至表尾
        currPtr->InsertAfter(newNodePtr);
        currPtr = newNodePtr;
    }
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}

// 对给定的 n 元循环链表,通过每次使第 m 个客人出局直到最后一个来解决 Josephus 问题
void Josephus(CNode<int> *list, int n, int m)
{
    // prevPtr 紧随 currPtr 循环扫描链表
    CNode<int> *prevPtr = list, *currPtr = list->NextNode();
    CNode<int> *deletedNodePtr;
    // 删除表中结点直到留下一个
    for(int i=0; i < n-1; i++)
    {
        // 从 currPtr 指向的当前客人开始数 m 个客人,即指针前移 m-1 次
        for (int j=0; j < m-1; j++)
        {
            // 前移指针
            prevPtr = currPtr;
            currPtr = currPtr->NextNode();
            // 若 currPtr 指向表头,再移一次指针
            if (currPtr == list)
            {
                prevPtr = list;
                currPtr = currPtr->NextNode();
            }
        }

        cout << "Delete person" << currPtr->data << endl;
        // 记录被删除的结点,并前移 currPtr 指针
        deletedNodePtr = currPtr;
        currPtr = currPtr->NextNode();
        // 从表中删除该结点
        prevPtr->DeleteAfter();
        delete deletedNodePtr;
        // 若 currPtr 为头指针,则再移动一次
        if (currPtr == list)
        {
            prevPtr = list;
            currPtr = currPtr->NextNode();
        }
    }
    cout << endl << "Person" << currPtr->data
         << "wins the cruise." << endl;
    // 删除表中最后一个结点
    deletedNodePtr = list->DeleteAfter();
    delete deletedNodePtr;
}

void main(void)
{
```

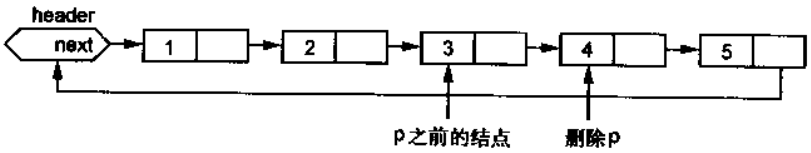

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 人员表
    CNode < int > list;
    // n 为表中人数,m 为循环选择因子
    int n, m;
    // 产生随机数 1 < = m < = n
    RandomNumber rnd;
    cout < < "Enter the number of contestants? ";
    cin > > n;
    // 产生客人 1,客人 2,...客人 n 组成的循环表
    CreateList(&list,n);
    m = 1 + rnd.Random(n);
    cout < < "Generated the random number " < < m < < endl;
    // 解决 Josephus 问题并输出最终结果
    Josephus(&list,n,m);
}
/*
< 程序 9.9 运行结果 >
Enter the number of contestants? 10
Generated the random number 5
Delete person 5
Delete person 10
Delete person 6 = Delete person 2
Delete person 9
Delete person 8
Delete person 1
Delete person 4
Delete person 7
Person 3 wins the cruise.
*/
```

9.9 双向链表

对以 NULL 终结的链表或循环链表的扫描都是从左到右进行的。但循环链表更灵活一些,扫描可以从任何位置开始,最后折返到起始位置。这些表的局限性是它们不允许用户回溯以逆向扫描表。它们在执行删除结点 p 这样一种简单的操作上往往很费周折。因为必须遍历表以查找指向 p 之前的结点的指针。



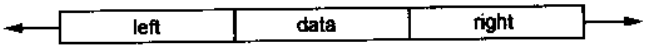
有些应用程序中,用户想以相反的顺序访问表。例如,一个棒球运动组织者有一张按击球平均得分数从低到高排列的运动员名单。为衡量运动员作为击球手的击中率,必须

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

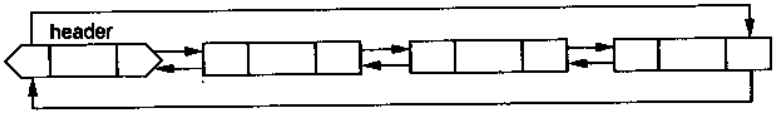
www.Linuxidc.com

以逆向遍历表。这可以用堆栈做到，但不方便。

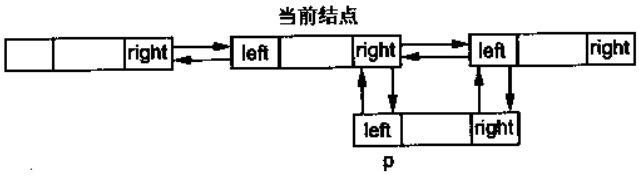
当我们需要双向访问结点时，双向链表是很有用的。双向链表中的结点含有两个指针域，一个数据域。



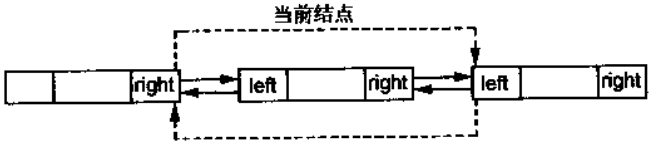
双链结点对表进行扩展，建立了一种有效而灵活的表处理结构。



两个方向都有插入和删除操作。下图示意了将结点 p 插入到当前结点右边的过程。4 个新链必须被赋值。



在双向链表中，结点可以通过改变两个指针而将自身从表中删除。



DNode 类是一种处理双循环链表的结点类。类定义和成员函数包含于文件“dnode.h”中。

DNODE 类定义

声明

```
template <class T>
class DNode
{
    private:
        // 指向左、右结点的指针
        DNode<T> *left;
        DNode<T> *right;
    public:
        // data 为公有成员
        T data;
        // 构造函数
        DNode(void);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        DNode(const T& item);

    // 改变表的方法
    void InsertRight(DNode < T >  * p);
    void InsertLeft(DNode < T >  * p);
    DNode < T >  * DeleteNode(void);

    // 取得指向左、右方向结点的指针
    DNode < T >  * NextNodeRight(void) const;
    CNode < T >  * NextNodeLeft(void) const;
};
```

说明

除了使用两个"next"指针以外，数据成员类似于单链的 CNode 类。有两种插入操作，每个方向一种。删除操作将当前结点从表中删除掉。用函数 NextNodeRight 和 NextNodeLeft 可以返回一个私有指针的值。

例

```
// 双向链表其头指针为 dlist
DNode < int > dlist;

// 遍历全表并打印每个结点的数据值，直到表头
DNode < int >  * p = &dlist;          // 初始化指针
p = p - > NextNodeRight();            // 将 p 指向表中第一个结点

while (p ! = &dlist)
{
    cout < < p - > data < <" ";       // 输出数据值
    p = p - > NextNodeRight();        // 将 p 指向表中下一结点
}

DNode < int >  * newNode1(10);        // 申请新结点，并使其值分别为 10 和 20
DNode < int >  * newNode2(20);
DNode < int >  * p = &dlist;          // p 指向头结点
p - > InsertRight(newNode1);          // 往表头插入
p - > InsertLeft(newNode2);           // 往表尾插入
```

应用：双向链表排序

程序 9.4 中用函数 InsertOrder 建立一个有序表。算法从头结点开始对表进行扫描以寻找插入点。对于双向链表，我们可以维持一个用来标识放入表中的最后一个结点的指针 CurrPtr 从而优化这一过程。

若要插入新的一项，我们先将它的值与当前位置值进行比较。如果新值较小，则用左指针往表头扫描；如果新值较大，则使用右指针往表尾扫描。例如，假设我们刚刚将 40 存入表"dlist"中：

dlist:10　25　30　40　50　55　60　75　90

若加入结点 70，则向后扫描表并将 70 插入到 60 的右边。若加入结点 35，则向前扫描表并将 35 插入到 40 的左边。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

程序 9.10　用双向链表排序

DlinkSort 通过建立有序表并将元素拷贝回数组中实现了用双向链表对 n 个元素的数组排序。函数 InsertHigher 将一个新结点加入到当前表位置的右边。与之对称的函数 InsertLower 将新结点加到当前位置的左边。

插入 item1
　　以头结点为参数调用 InsertRight，保存 a[0]

插入 item 2 ~ 10
　　如果 item < currPtr - > data，调用 InsertLower
　　如果 item > currPtr - > data，调用 InsertHigher

本程序中，用 DlinkSort 对 10 个整数的表进行排序。已排好序的表用 PrintArray 进行输出。

```
# include < iostream.h >
# include "dnode.h"
template < class T >
void InsertLower(DNode < T >  * dheader,DNode < T > * &currPtr, T item)
{
    DNode < T >  * newNode = new DNode < T > (item), * p;
    // 寻找插入点
    p = currPtr;
    while (p ! = dheader && item < p - > data)
        p = p - > NextNodeLeft();
    // 插入元素
    p - > InsertRight(newNode);
    // 使 currPtr 指向新结点
    currPtr = newNode;
}
template < class T >
void InsertHigher(DNode < T > * dheader, DNode < T > * & currPtr,
                  T item)
{
    DNode < T >  * newNode = new DNode < T > (item), * p;
    // 寻找插入点
    p = currPtr;
    while (p ! = dheader && p - > data < item)
        p = p - > NextNodeRight();
    // 插入元素
    p - > InsertLeft(newNode);
    // 使 currPtr 指向新结点
    currPtr = newNode;
}
template < class T >
void DLinkSort(T a[], int n)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 建立双向链表存放数组元素
    DNode<T> dheader, *currPtr;
    int i;
    // 往双向链表中插入数组的第一个元素
    DNode<T> *newNode = new DNode<T>(a[0]);
    dheader.InsertRight(newNode);
    currPtr = newNode;
    // 往双向链表中插入数组的其它元素
    for (i=1; i < n;i++)
        if (a[i] < currPtr->data)
            InsertLower(&dheader, currPtr, a[i]);
        else
            InsertHigher(&dheader, currPtr, a[i]);
    // 遍历全表并将数据值拷贝回数组中
    currPtr = dheader.NextNodeRight();
    i = 0;
    while(currPtr != &dheader)
    {
        a[i++] = currPtr->data;
        currPtr = currPtr->NextNodeRight();
    }
    // 删除表中所有结点
    while(dheader.NextNodeRight() != &dheader)
    {
        currPtr = (dheader.NextNodeRight())->DeleteNode();
        delete currPtr;
    }
}
// 扫描数组并输出其元素
void PrintArray(int a[], int n)
{
    for (int i=0; i < n; i++)
        cout << a[i] << "  ";
}
void main(void)
{
    // 用 10 个整数值初始化数组
    int A[10] = {82, 65, 74, 95, 60, 28, 5, 3, 33, 55};
    DLinkSort(A,10);                        // 对数组排序
    cout << "Sorted array:    ";
    PrintArray(A,10);                       // 输出数组
    cout << endl;
}
/*
<程序 9.10 运行结果>
Sorted array:    3  5  28  33  55  60  65  74  82  95
*/
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

DNode 类的实现

构造函数通过将结点地址 this 赋给 left 和 right 而生成一个空表。如果给构造函数传递一个参数，结点的 data 域将被初始化为该参数。

```
// 构造函数,创建一个空表,并初始化其 data 域
template < class T >
DNode < T > ::DNode(const T& item)
{
    // 建立一个指向自身的结点并初始化 data 域
    left = right = this;
    data = item;
}
```

表操作 若要将结点 p 插入到当前结点的右边，则必须对 4 个指针赋值。图 9.2 中示意了 C++ 语句和新链的对应关系。注意赋值顺序并不是随意的。例如，如果先完成(4)则接在当前结点之后的结点的链就会丢失。读者应该验证一下算法在往空表中插入的情况下的正确性。

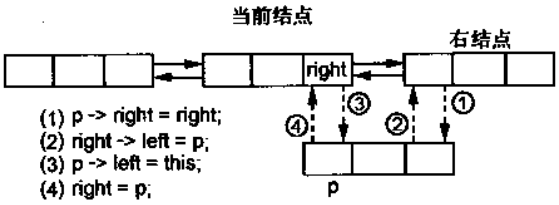


图 9.2 在双向循环链表中将结点插入到当前结点的右侧

```
// 将结点 p 插入到双向链表中当前结点右边
template < class T >
void DNode < T > ::InsertRight(DNode < T > *p)
{
    // 将 p 和当前结点的右后继结点相连
    p->right = right;
    right->left = p;
    // 将 p 的左边和当前结点相连
    p->left = this;
    right = p;
}
```

InsertLeft 方法则将 InsertRight 算法中的 right 与 left 互换即可。

```
// 将结点 p 插入到当前结点左边
template < class T >
void DNode < T > ::InsertLeft(DNode < T > *p)
{
    // 将 p 和当前结点的左后继结点相连
    p->left = left;
    left->right = p;
    // 将 p 的右边与当前结点相连
    p->right = this;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    left = p;
}
```

为删除当前结点，必须修改两个指针，如图 9.3 所示。读者应该验证一下算法在删除表中最后一个结点的情况下的正确性。该方法返回指向已删除结点的指针。

```
// 从链表中删除当前结点并返回其地址
template <class T>
```

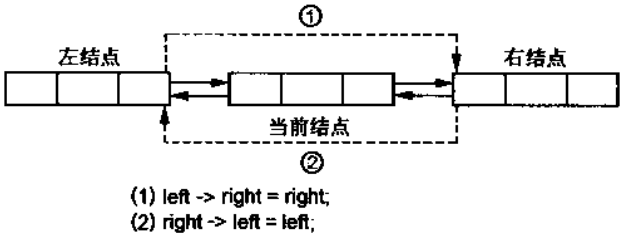


图 9.3　从双循环链表中删除一个结点

```
DNode<T> *DNode<T>::DeleteNode(void)
{
    // 将在结点的右指针指向右结点
    left->right = right;
    // 将右结点的左指针指向左结点
    right->left = left;
    // 返回当前结点的指针
    return this;
}
```

9.10　实例研究：窗口管理

图形用户界面(GUI)维护屏幕上的多个窗口。这些窗口按层次组织，最前面的窗口作为活动窗口(active window)。有些应用程序维护一个当前打开窗口表。从菜单中可以访问此表。用户可以选择一个窗口标题以使之成为最前面的或活动的窗口。当一个底层窗口的视线被挡时，这就显得特别有用。从菜单的表中选择“Winddow_1”可以激活该窗口并使之成为当前窗口。

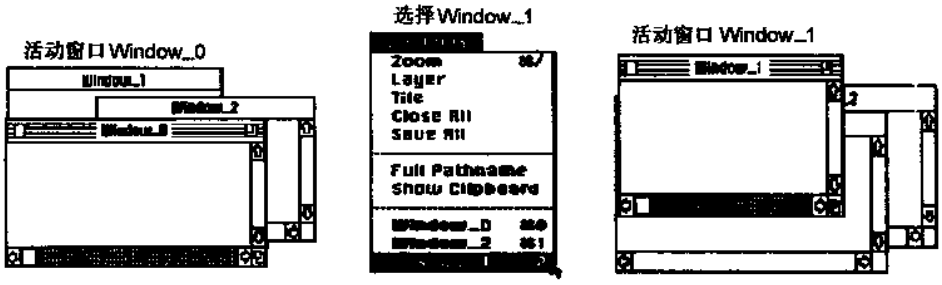


图 9.4　窗口表

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

屏幕上的每个窗口都与一个文件相关联。当打开相应的文件时窗口被创建,文件关闭时窗口被注销。我们用链表存放窗口表,并将每一个文件操作都与窗口表中的相应操作相关联。文件操作 New 创建一个前排窗口。该窗口将被加入到表的头部。诸如 Close 和 Save As 一类的文件操作都作用于表头的活动窗口。一般操作,如 Close All 可以这样实现:从表中删除每个窗口并关闭窗口和相应的文件。

本实例中维护 GUI 应用的一个窗口表。应用程序支持以下的文件和表操作:

New: 插入名为"Untitled"的窗口。

Close: 删除前排窗口。

Close All: 清空表以关闭所有窗口。

Save As: 将窗口的内容保存到不同的文件名中并更新窗口的标题。

Quit: 终止应用程序。

Menu Display: 窗口菜单按层次显示每个窗口的号码和名字。用户可以输入号码并激活窗口,将它移到窗口表的最前面。

窗口表

任一时刻所允许打开的窗口最大数目为 10。每个打开的窗口都对应有一个 0 到 9 范围内的号码。关闭一个窗口时,其号码可以用于创建新的打开窗口的 New 操作。处理各个窗口并修改当前活动窗口由 Close, Close All, Save As 和 Activate 方法负责。每个窗口都由一个窗口对象表示,它描述了窗口名及其在可用窗口表中的号码(索引)。

窗口对象由 Window 类描述,它包含 windowTitle 和 windowNumber 两个域。类中还有改变窗口标题、获取窗口号码并以格式标题[窗口号]输出信息的成员函数。重载的关系等于运算符("= =")根据窗口号比较两个 Window 记录。

Window 类定义

声明

```
// 存放单个窗口信息的类
class Window
{
    private:
      // 描述窗口的数据,包括其标题及其在可用窗口表中的索引
      String windowTitle;
      int windowNumber;

    public:
      // 构造函数
      Window(void);
      Window(const String& title, int wnum);

      // 数据访问函数
      void ChangeTitle(const String& title);
      int GetWindowNumber(void);

      // 重载运算符
      int operator = = (const Window& w);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        friend ostream& operator < < (ostream& ostr,
                                     const Window& w);
};
```

Window 类的完整实现在文件“windlist.h”中给出。窗口表以及创建、保存、关闭以及激活窗口对象的操作都由 WindowList 类定义。

WindowList 类定义

声明

```
#include "link.h"        // 引入类 LinkedList
// 存放窗口对象的链表结构
class WindowList
{
    private:
        // 活动窗口表
        LinkedList < Window > windList;
        // 可用窗口表及打开窗口的个数
        int windAvail[10];
        int windCount;
        // 得到及释放窗口的函数
        int GetWindowNumber(void);
        int FindAndDelete(Window& wind);
        // 输出已打开窗口表
        void PrintWindowList(void);
    public:
        // 构造函数
        WindowList(void);
        // 窗口菜单函数
        void New(void);
        void Close(void);                    // 关闭最前面窗口
        void CloseAll(void);                 // 关闭所有窗口
        void SaveAs(const String& name);     // 窗口改名
        void Activate(int windowum);         // 使窗口 windownum 成为活动窗口
        // 模拟窗口管理
        void SelectItem(int& item,String& name);
};
```

说明

构造函数为初始窗口数据赋值。它将窗口计数设为零，并将 0 到 9 范围内的每个窗口标记为可用。New 从可用窗口表中取得一个窗口并将索引设为窗口号。窗口被赋以一个一般性的标题“Untitled”。新窗口被插入到打开窗口表的表头位置。

WindowList 类另外还支持 Close，Close All，Save As 和 Activate 操作。这些操作可实现对应的菜单选项的功能。

SelectItem 操作可实现菜单。用户可以键入字母 N(New)，C(Close)，A(Close All)，S

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(Save As), Q(Quit)或数字 0,1,…。该方法返回输入选择并内部产生一个指示操作的项目号。下表列出了项目号与选择的对应关系。

项目号	名称	选择的字符
1	New	n(N)
2	Close	c(C)
3	Close All	a(A)
4	Save As	s(S)
5	Quit	q(Q)
6	WindowName[i_0]	i_0
7	WindowName[i_1]	i_1
	…	
15	WindowName[i_9]	i_9

菜单项 2~4 只有在至少有一个窗口打开的情况下才会打印出来。窗口名和号码的打印是由私有方法 PrintWindowList 控制的。从第 6 项开始的项目都对应于当前打开窗口。第 6 项是排头窗口,其窗口号为 i_0;第 7 项是第 2 个窗口,其号码为 i_1,……依此类推。只要键入窗口号就可以将窗口放到表头的位置。新窗口被赋以 0~9 范围内最小的可用窗口号。例如,如果窗口 0,1,3 和 5 正在使用中,且 0 被关闭,那么下一个新建立的窗口号将是 0。函数 GetWindowNumber 执行分配窗口号的算法。

例

下表给出一组窗口命令。在完成一次操作以后,窗口表中的第 1 项即为活动窗口。对于 SaveAs 命令来说,紧接其后的输入被括在括弧中。

被选择的字符	项目号	动作	窗口表
N	1	开新窗口	Untitled[0]
N	1	开新窗口	Untitled[1]Untitled[0]
S(One)	4	存为"One"	One[1]Untitled[0]
0	7	窗口 0 成为活动窗口	Untitled[0]One[1]
N	1	开新窗口	Untitled[2]Untitled[0]One[1]
C	2	关闭窗口	Untitled[0]One[1]
C	2	关闭窗口	One[1]
N	1	开新窗口	Untitled[0]One[1]
A	3	关闭所有窗口	< empty >
Q	5	退出	terminate application

WindowList 类的实现

WindowList 方法的完整清单包含于文件"windlist.h"中。我们说明一些函数的代码以演示链表用于维护打开窗口表以及执行菜单选项的情况。函数 GetWindowNu nber 遍历数组 WindAvail 以搜索可用窗口号。它返回找到的第 1 个,因而所有的新窗口都具有最小可

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

能的窗口号。

```
// 从可用窗口表中取得第一个可用窗口号
int WindowList::GetWinodwNumber(void)
{
    for (int i = 0; i < 10; i + + )
      // 若该号可用,则选中该号并将其置为不可用

        if (windAvail[i])
        {
            windAvail[i] = 0;
            break;
        }
    return i;            // 返回该窗口号
}
```

私有函数 PrintWindowList 扫描窗口表并打印每个窗口的标题和号码。该方法从第 1 个窗口开始(Reset)并一个窗口一个窗口地移动(Next)直到表尾(EndOfList)从而实现了对表的简单的顺序扫描。Window 类中的"< <"运算符以"格式标题[号码]"的方式打印出窗口数据。

```
// 输出所有打开窗口的信息
void WindowList::PrintWindowList(void)
{
    for (windList.Reset(); ! windList.EndOfList();
        windList.Next())
            cout < < windList.Data();
}
```

WindowList 操作　若要创建一个窗口,New 方法为 Window 对象 win 赋以标题"Untitled"。通过调用 GetWindowNumber,函数将一个窗口号赋值给该对象,将其插入到窗口表中并使窗口计数增加。

```
// 创建新窗口并将其名字置为'untitled'
void WindowList::New(void)
{
    // 检查该窗口是否可用,若不可用,则退出
    if (windCount = = 10)
    {
        cerr < < "No more windows available until one is"
             < < "closed" < < endl;
        return;
    }
    // 调用 GetWindowNumber 得到一个新的名为'Untitled'的窗口
    Window win("Untitled",GetWindowNumber());
    // 将其插入表头使其成为活动窗口
    windList.InsertFront(win);
    windCount + + ;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

若要激活一个位于其它窗口后面的窗口，必须先以该窗口的窗口号为键值找到该窗口并将其从表中删掉。将窗口重新插入到表头位置后，它将变为活动的。我们调用私有方法 FindAndDelete，它将扫描表以寻找匹配的窗口号。当找到一个窗口时，该方法将其从表中拆卸下来并返回窗口数据。然后可以用该信息建立一个新的窗口。它将被插入到表头的位置。

```
// 在 windList 中根据窗口号找到相应窗口，并将其删除，然后用 wind 返回其值
int WindowList::FindAndDelete(Window& wind)
{
    int retval;
    // 遍历该表寻找 wind
    for (windList.Reset();! windList.EndOfList();windList.Next())
        // window 的"= ="运算符比较窗口号，若相等，则中止循环
        if (wind = = windList.Data())
            break;
    // 有相匹配的窗口号吗？
    if(! windList.EndOfList())
    {
        // 赋值给 wind，删除该窗口并返回 1(成功)
        wind = windList.Data();
        windList.DeleteAt();
        retval = 1;
    }
    else
        retval = 0;        // 返回 0(失败)
    return retval;
}
// 将 windownum 置为活动窗口
void WindowList::Activate(int windownum)
{
    Window win("Dummy Name",windownum);
    if (FindAndDelete(win))
        windList.InsertFront(win);
    else
        cerr < < "Incorrect window number."endl;
}
```

程序 9.11　窗口表的维护

此程序定义了一个 WindowList 对象 windops，其中含有打开窗口的表。事件循环调用函数 SelectItem 以响应其返回的项目号。此循环一直持续到用户选择"Q"(Quit)。

```
# include < iostream.h >
// 引入类 Window 和 WindowList
# include "windlist.h"
// 清除输入缓冲区到一行结束
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
void ClearEOL(void)
{
    char c;
    do
        cin.get(c);
    while(c ! = '\n');
}
void main(void)
{
    // 程序中用到的窗口
    WindowList windops;
    // 窗口标题
    String wtitle, itemText;
    int done = 0, item;
    // 运行模拟过程直到用户输入'q'
    while(! done)
    {
        // 输出菜单选项读入用户输入
        windops.SelectItem(item,itemText);
        // 若用户选择一个数字,则激活相应窗口
        if (item > = 6)
           windops.Activate(itemText[0] - '0');
        // 若用户选择选项 0 - 5,调用相应函数处理用户请求
        else
            switch(item)
            {
                case 0:    break;
                case 1:    windops.New();
                           break;
                case 2:    windops.Close();
                           break;
                case 3:    windops.CloseAll();
                           break;
                case 4:    cout < < "New Window Title: ";
                           ClearEOL();
                           wtitle.ReadString();
                           windops.SaveAs(wtitle);
                           break;
                case 5:    done = 1;
                           break;
            }
    }
}
/*
< 程序 9.11 运行结果 >
New  Quit: n
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
New  Close  Close All  Save As  Quit  Untitled[0]: n
New  Close  Close All  Save As  Quit  Untitled[1]  Untitled[0]: s
New Window Title: one
New  Close  Close All  Save As  Quit  one[1]  Untitled[0]: 0
New  Close  Close All  Save As  Quit  Untitled[0]  one[1]: s
New Window Title: two
New  Close  Close All  Save As  Quit  two[0]  one[1]: n
New  Close  Close All  Save As  Quit  Untitled[2]  two[0] one[1]: s
New Window Title: three
New  Close  Close All  Save As  Quit  three[2]  two[0]  one[1]: 0
New  Close  Close All  Save As  Quit  two[0]  three[2]  one[1]: c
New  Close  Close All  Save As  Quit  three[2]  one[1]: a
New  Quit: q
*/
```

书面作业

9.1 假定以下语句被执行：

```
Node<int> *p1, *p2;
p1 = new Node<int(2);
p2 = new Node<int(3);
```

则下面的程序段分别打印出什么内容？

(a)
```
cout << p1->data << " " << p2->data << endl;
```

(b)
```
p1->data = 5;
p1->InsertAfter(p2);
cout << p1->data << " " << p1->NextNode()->data << endl;
```

(c)
```
p1->data = 7;
p2->data = 9;
p2 = p1;
cout << p1->data << " " << p2->data << endl;
```

(d)
```
p1->data = 8;
p1->data = 15;
p2->InsertAfter(p1);
cout << p1->data << " " << p2->NextNode()->data << endl;
```

(e)
```
p1->data = 77;
p1->data = 17;
p1->InsertAfter(p2);
p2->InsertAfter(p1);
cout << p1->data << " " << p2->NextNode()->data << end ;
```

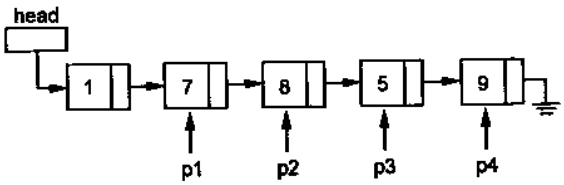
9.2 给出 Node 对象的下述链表以及指针 P1,P2,P3 和 P4。对于每个代码段,画出一个类似的表示表如何改变的图。

(a)
```
P2 = P1->NextNode();
```

(b)
```
head = P1->NextNode();
```

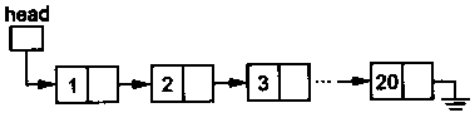

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
(c) p3->data = p1->data;
(d) p3->data = p1->NextNode()->data;
(e) p2 = p1->DeleteAfter();
    delete p2;
(f) p2->InsertAfter(new Node<int>(3));
(g) p1->NextNode()->NextNode()->NextNode()->data = p1->data;
(h) Node<int> *p = p1;
    while(p != NULL)
    {
        p->data *= 3;
        p = p->NextNode();
    }
(i) Node<int> *p = p1;
    while(p->NextNode() != NULL)
    {
        p->data *= 3;
        p = p->NextNode();
    }
```

9.3 编写一段代码,建立数值为 1,2,…,20 的链表。



9.4 在下面每条语句之后列出链表内容。对于"cout"语句。给出输出结果。

```
Node<char> *head, *p, *q;
head = new Node<char>('B');
head = new Node<char>('A',head);
q = new Node<char>('C');
p = head;
p = p->NextNode();
p->InsertAfter(q);
cout << p->data << "" << p->NextNode()->data << endl;
q = p->DeleteAfter();
delete q;
q = head;
head = head->NextNode();
delete q;
```

9.5 编写函数

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
template < class T >
Node < T >  * Copy(Node < T >  * p);
```

复制以结点 p 开始的链表。

9.6 假设要编写一个 Node 类的方法以便能改变 next 数据成员。说明以下语句的作用。因为代码在方法中，所以可以访问指针“next”和“this”。

```
Node < T >  * p;
```

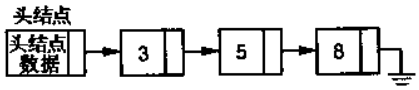
(a)
```
p = next;
p - > next = next;
```

(b)
```
p = this;
next - > next = p;
```

(c)
```
next = next - > next;
```

(d)
```
p = this;
next - > next = p - > next;
```

9.7 我们可以用维护一个头结点的方法代替指向 Node 对象链表头结点的指针。其 data 值不作为表的一部分，而它的 next 数据成员指向链表中的第一个数据结点。头结点被称为哨位结点，它可以避免出现空表的情况。



假设一个整数链表由以下头结点维护：

```
Node < int > header(0);
```

(a) 编写一段代码将结点 p 插入到表的前端。

(b) 编写一段代码将结点 p 从表的前端删除。

9.8 本题假定链表是由 Node 类创建和维护的。两个有序表 L_1 和 L_2 可以合并到第 3 个表 L_3 中，方法是遍历两个有序表，将结点插入到第 3 个表中。在任一点，比较来自 L_1 和 L_2 中的值，将较小的值插入到 L_3 的尾部。插入一个值后，原所在表指针下移一个位置。例如，考虑以下插入序列。其中 L_1 和 L_2 中当前比较的元素都加了圆圈。

编写函数

```
void MergeLists(Node < T >  * L1, Node < T > *  L2,Node < T >  * &l3);
```

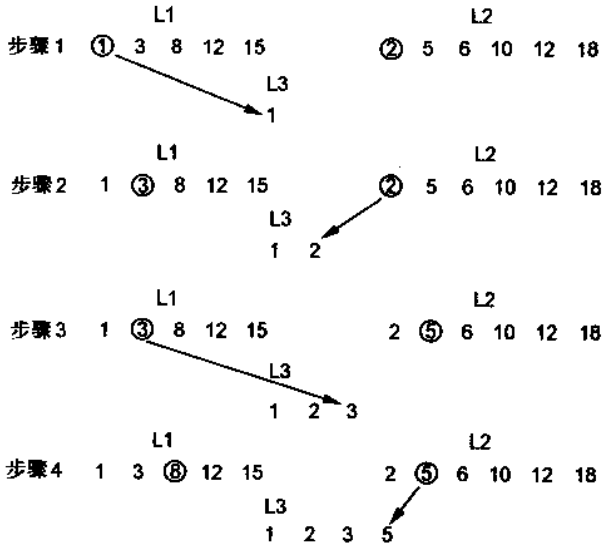
将 L_1，L_2 合并为 L_3。

9.9 以下代码段会起什么作用？

```
while(currPtr ! = NULL)
{
    currPtr - > data + = 7;
    currPtr = currPtr - > NextNode();
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



9.10 说明函数 F 的作用。

```
template <class T>
void F(Node<T> * &head)
{
    Node<T> *p, *q;
    if (head != NULL&& head->NextNode() != NULL)
    {
        q = head;
        head = p = head->NextNode();
        while(p->NextNode() != NULL)
            p = p->NextNode();
        p->InsertAfter(q);
    }
}
```

9.11 编写函数

```
template<class T>
void CountKey(cons Node<T> *head, T key);
```

统计键值(key)在表中出现的次数。

9.12 编写函数

```
template<class T>
void DeleteKey(Node<T> * &head, T key);
```

删掉表中所有出现键值(key)的结点。

9.13 修改 9.2 节中的函数 InsertOrder 以便可以在复制代码段的开始加入数据值。

9.14 给出由 a~b 各指令序列所得到的结果链表。

```
LinkedList<int> L;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
int i, val;
```

(a)
```
for(i=1;i <= 5;i++)
    L.InsertFront (2*i);
```

(b)
```
for(i=1;i <= 5;i++)
    L.InsertAfter (2*i);
```

(c)
```
for(i=1;i <= 5;i++)
    L.InsertAt (2*i);
```

(d)
```
for(i=1;i <= 5;i++)
{
    L.InsertAt(i);
    L.next();
    L.insertAt(2*i)
    val = L.DeleteFront();
}
```

(e) 用 InsertFront 和 For 语句建立链表 50,40,30,20,10。

(f) 用 InsertRear,InsertAt 和 InsertAfter 重复插入 e。

9.15 假设有如下说明:

```
LinkedList<int> L;
int i, val;
```

假设链表 L 中含有值 10,20,30,…,100,执行完以下程序段以后链表中各结点的数据值分别是什么?

(a)
```
L.Reset();
for(i=1;i <= 5;i++)
    L.DeleteAt();
```

(b)
```
L.Resert();
for(i=1;i <= 5;i++)
{
    L.DeleteAt();
    L.Next();
}
```

(d)
```
L.Reset();
for(i=1;i <= 3;i++)
{
    val = L.DeleteFront();
    L.next();
    L.InsertAt(val);
}
```

9.16 编写函数

```
template<class T>
void Split(const LinkedList<T>& L, LinkedList<T>& L1,
           LinkedList<T>& L2);
```

以表 L 为参数,生成新表 L_1 和 L_2。L_1 中包含第 1,第 3,第 5 以及后续的奇数结点。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

L_2 中包含偶数结点。

9.17 假设 L 是一个整数表。编写函数

```
void OddEven(const LinkedList < int > & L, LinkedList < int > & L1,
             LinkedList < int > & L2);
```

以表 L 为参数，生成新表 L_1 和 L_2。L_1 中包含 L 中数据值为奇数的结点。L_2 中包含数据值为偶数的结点。用循环语句。

9.18 编写一个 List 类方法

```
int operator + (const List < T > & L);
```

将表 L 连接到当前表的末尾。若成功则返回 1，若不能为新结点分配内存则返回 0。禁止一个表试图与它自身连接，在此情况下亦返回 0。

9.19 编写函数

```
template < class T >
void DeleteRear(LinkedList < T > & l);
```

删除表 L 的尾部元素。

9.20 在操作序列

```
Push(1),  Push(2)  Pop,  Push(5),  Push(7),  Pop,  Pop.
```

之后，画出链栈

```
Stack < int > L;
```

的图。

9.21 用复合法包含 LinkedList 对象，实现 Stack 类。模仿 9.6 节的 Queue 类写出方案模型。

9.22 通过维护一个 Node 对象的链表实现 Stack 类。

9.23 以下函数执行什么操作？

```
template < class T >
void ActionS(LinkedList < T > & L)
{
    Stack < T > S;
    for(L.Reset(); ! L.EndOfList();L.Next())
        S.Push(L.Data());
    L.Reset();
    while(! S.StackEmpty())
    {
        L.Data() = S.Pop();
        L.Next();
    }
}
```

9.24 在操作序列

```
QInsert(1),  QInsert(2),  QDelete,  QInsert(5),  QInsert(7),
QDelete, QInsert(9)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

之后，画出链队列

```
Queue < int > Q;
```

的图。图中一定要包含队尾指针。

9.25 下列函数执行什么操作：

```
template < class T >
void ActionQ(LinkedList < T > & L, Queue < T > & Q)
{
    Q.QClear();
    for(L.Reset(); ! L.EndOfList(); L.Next())
        Q.Q.Insert(L.Data());
}
```

9.26 修改 9.6 节中的 Queue 类，使其包含成员函数

```
T PeekFront(void);
T PeakRear(void);
```

它们分别返回队列的头和尾的数值。

9.27 Queue 类中没有显式赋值运算符。试解释为什么 Queue 对象 Obj1 和 Obj2 会出现在下列语句中。

```
Obj2 = Obj1;
```

9.28 实现函数 Replace，在循环链表中查找指定数值。

```
template < class T >
CNode < T > * Replace(CNode < T > * header, CNode < T > * start, T elem,
                      T newelem);
```

从 start 所指向的结点开始扫描链表以查找 elem。如果在表中找到 elem，用新的数据值 newelem 替换它并返回指向匹配结点的指针；否则，返回 NULL。注：扫描时可忽略 header。

9.29 实现函数

```
template < class T >
void InsertOrder(CNode < T > * header, CNode < T > * elem);
```

将结点 elem 插入到一个循环表中且使数据有序排列。

9.30 考察下面的结构

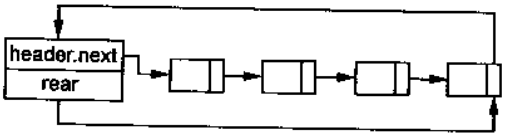
```
template < class T >
struct CList
{
    CNode < T > header;
    CNode < T > * rear;
};
```

它定义一个带有尾指针的循环链表。

编写以下函数：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
// 将结点 p 插入表 L 的头部
template <class T>
void InsertFront(CList<T>& L,CNode<T> *p);
// 将结点 p 插入表 L 的尾部
template <class T>
void InsertRear(CList<T>& L, CNode<T> *p);
// 删除表 L 中第一个结点
template <class T>
CNode<T> *DeleteFront(CList<T>& L);
```

要注意正确地维护好表尾指针以便任何时候它总是指向表中最后一个结点。

9.31 (a) 编写函数

```
template <class T>
void Concat(CNode<T>& s, CNode<T>& t);
```

将头结点为 t 的循环表连接到头结点为 s 的循环链表中。

(b) 编写函数

```
template <class T>
int Length(CNode<T>& s);
```

确定以 s 为头结点的循环表中的元素个数。

(c) 编写函数

```
template <class T>
CNode<T> *Index(CNode<T>& s, T elem);
```

返回循环表中第 1 个数据值为 elem 的结点的指针。如果未找到则返回头指针。

(d) 编写函数

```
template <class T>
void Remove(CNode<T>& s, T elem);
```

删除表 s 中所有包含数值 elem 的结点。

9.32 编写双链结点类成员函数：

```
DNode<T> *DeleteNodeRight(void);
DNode<T> *DeleteNodeLeft(void);
```

DeleteNodeRight 删除当前结点右边的结点，DeleteNodeLeft 删除左边的结点。

上机题

9.1 本题扩展文件“nodelib.h”的结点实用函数。编写具有如下声明的函数：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 删除表中尾结点并返回指向该结点的指针
template < class T >
Node < T > * DeleteRear(Node < T > * & head);
// 删除表中所有值为 key 的结点
template < class T >
void DeleteKey(Node < T > * & head, const T& key);
```

编写一个程序测试以上函数。

- 输入 10 个整数并用 InsertFront 将它们存在表中。用 PrintList 输出表。
- 输入 1 个作为键值的整数。用 DeleteKey 将表中所有出现键值的结点删去。输出最后得到的表。
- 用 DeleteRear 清除表中所有结点。每进行一次删除都输出数据值。

9.2 本题中使用 Node 类。其数据域是结构 IntEntry。

```
struct IntEntry
{
    int value;    // 整数
    int count;    // value 出现的次数
};
```

输入 10 个整数并建立一个 IntEntry 结点的有序表，其中结点的 count 域用来指明表中重复元素的个数。输出最后得到的结点信息，其中包括互不相同的整数以及每个整数在表中的出现次数。

输入一个键值并删除数据值比键值大的所有结点。输出最后得到的表。

9.3 本题中使用 Node 类。其数据域包含一个雇员的姓名、标识号以及每小时薪水数。

```
struct Employee
{
    char   name[20];
    int    idnumber;
    float  hourlypay;
};
```

重载"<<"和">>"以便对雇员信息进行 I/O 操作：用 PrintList 输出表。输入下数据记录并将它们存到一个链表中。

```
40 9.50 Dennis Williams
10 6.00 Harold Barry
25 8.75 Steve Walker
```

实现以下对表进行修改的更新操作：

(a) 读取一个标识号并在表中查找雇员记录。如果找到，则将雇员的每小时薪水数增加 3 美元。输出最后得到的表。(必须重载"==")。

(b) 对表进行扫描，将每小时收入超过 9 美元的雇员结点删掉。输出最后得到的表。

9.4 有两个表：$L=\{L_0,L_1,\cdots,L_i\}$，$M=\{M_0,M_1,\cdots,M_j\}$，它们可以捉对合并产生表 $\{L_0,M_0,L_1,M_1,\cdots,L_i,M_i,\cdots,M_j\}$，$j\geq i$。用 Node 类编写一个程序，生成含 i 个($1\leq i\leq 10$)

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

个数的表 L,含 j(1≤j≤20)个数的表 M。表 L 中的数是 0 到 99 之间的随机数;表 M 中的数是 100 到 199 之间的随机数。程序打印出初始表 L 和 M,将它们合并到新表 N 中,然后打印得到的表。

9.5 使用两个 LinkedList 对象 L 和 M,重做一遍上机题 9.4 题。

9.6 用 Node 类编写一个程序,用下述算法往链表中输入 5 个整数:

对于每个输入数 N,将 N 插入到表的前部。扫描表的其余部分,将所有值比 N 小的结点删掉。

使用以下输入数据运行该程序 3 遍。打印出最后得到的表。

```
1,2,3,4,5
5,4,3,2,1
3,5,1,2,4
```

9.7 LinkedList 类中同时包含了一个复制构造函数和一个重载的赋值运算符。编写一个测试程序以评价这两种方法的正确性。

9.8 用第 8 章中的 String 类读取以空格隔开的文本标识串。输入一行,其中可能包括以'—'开头的字符串。例如:

```
// t 和 includelist 为选项
run -t -includelist linkemo
```

本例中,字符串包括选项(以'—'开头的标记串)以及非选项标记串。用 LinkedList 类将非选项标记串存放到链表 tokenlist 中,将选项标记串存放到链表 optionlist 中。每个标记串都插入到其表尾位置。打印出每个表中的标记串。

9.9 一个正整数 n(n>1)可以被唯一地表示为若干素数的乘积。这被称为数的分解质因子。例如

```
12 = 2 * 2 * 3        18 = 2 * 3 * 3        11 = 11
```

函数 LoadPrimes 使用 IntEntry 记录上机题 9.2 中的结构。建立一个链表,其中存放不同的素数以及在分解质因子后每个素数出现的次数。例如,对于 18 来说,素数 2 出现 1 次、素数 3 出现 2 次。

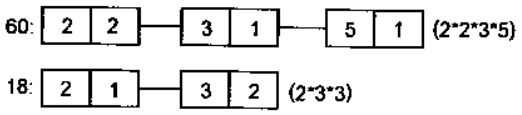
```
void LoadPrimes(LinkedList < IntEntry > &L, int n)
{
    int i = 2;
    int nc = 0;    // 当前质数出现的次数
    do
    {
        if (n & i = = 0)
        {
            nc + + ;
            n = n/i;
        }
        else
        {
            if (nc > 0)
                < load i and nc as a node at rear of the list >
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            nc = 0;
            i++;
        }
    }
    while(n > 1);
}
```

编写一段程序，输入两个整数 M 和 N 并用函数 LoadPrimes 建立一个素数链表。扫描表，打印出每个数分解质因子的结果。



建立一个由 M 和 N 中公共素数所组成的新表。存放每一个这样的素数时，取两个结点中较小的 count 值，并将其作为新表中结点的 count 值。例如，2 是 60 的素因子且也是 18 的素因子。

```
18 = 2 * 3 * 3      // 素因子 2 出现一次
60 = 2 * 2 * 3 * 5 // 素因子 2 出现两次
```

在新表中，2 的 count 值是为 1(取 1,2 中的最小值)。得到的结果是 M 和 N 的最大公因子，即 GCD(M,N)。

9.10 一个 n 次多项式形如

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a2_x^2 + a_1x^1 + a_0x^0$$

其中 a_i 被称为系数。在本题中使用 LinkedList 类，其数据记录 Term 中包含每一项的系数项及 x 的幂次。

```
struct Term
{
    double coeff;
    int power;
};
```

在程序中以(系数，幂次)对的序列输入一个多项式。在输入的幂次为 0 时终止程序，将每对系数/幂次存储到按幂次排序的链表中。

(a) 按以下形式输出结果多项式中的每一项

```
ai * x^i
```

(b) 输入 X 的 3 个值并调用函数

```
double poly(LinkedList<Term>& f);
```

求出多项式的值并输出结果。

9.11 本题使用书面作业 9.2 和 9.13 中编写的函数。读取 10 个整数值到一个有序表中并打印出该表。提示用户输入一个值，并用 CountKey 记录该值在表中的出现次数。用 DeleteKey 删除表中所有出现该值的结点。打印出新表。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

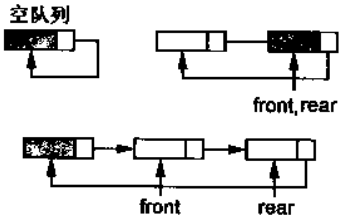
www.Linuxidc.com

9.12 用整数值测试书面作业 9.8 中的函数 MergeLists。在测试中包含表 $L_1 = \{1,3,4,6,7,10,12,15\}$ 以及 $L_2 = \{3,5,6,8,11,12,14,18,22,33,55\}$。

9.13 用 CNode 类实现 Queue 类。在类的私有部分声明一个头结点并以如下方式实现插入和删除：

QInsert：对当前结点进行 InsertAfter 操作并将指针移向新结点。

QDelete：对头结点进行 DeleteAfter 操作。要注意不要试图从空表中删除。



将整数 1,2,…,10 插入到一个队列中以测试你的程序。将队列清空并将数值打印出来。

9.14 我们可以通过将具有唯一性的数据元素存到链表中的方法实现元素类型为 T 的 Set(集合)类。试实现以下两个函数：

```
template <class T>
LinkeDlist <T> Union(LinkedList <T> & x, LinkedList <T> & y);
template <class T>
Linkedlist <T> Intersection(LinkedList <T> & x, LinkedList <T> & y);
```

并集(Union)返回的是表示至少在 x 或 y 两者之一中出现的所有元素的集合的链表。交集(Intersection)则返回表示在 x 和 y 中同时出现的所有元素的集合的链表。令 A 为整数集合 $\{1,2,5,8,10,12,33,55\}$，B 为集合 $\{2,8,10,12,33,99,99\}$。用两个函数计算并打印

$$A \cup B = \{1,2,5,8,10,12,33,55,88,99\}$$

以及

$$A \cap B = \{2,8,12,33\}$$

9.15 给定一个空的双循环整数链表。输入 10 个整数，将正数插入到紧靠头结点右边的位置而负数则插入到左边。打印出该表。将表拆分成分别包含正数和负数的两个单链表。分别将它们打印出来。

9.16 本题修改 9.10 节中的 Window 类。增加方法 SaveAll。必须从前到后遍历窗口表并打印每个窗口的标题以及消息“Saving < window title >”。如果窗口现在还没有标题，则提示输入一个标题。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第十章　递　　归

10.1　递归的概念
10.2　设计递归函数
10.3　递归代码和运行时堆栈
10.4　用递归进行问题求解
10.5　递归评估
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

递归是计算机科学和数学中的一个极其重要的问题求解工具。在程序设计语言中可以用它来定义语言的语法，在数据结构中可以用它来编制表和树结构的查找和排序算法。数学家们则将递归应用于组合数学领域，其处理对象是大量的计数和可能性问题。无论在理论还是在实际应用方面，递归都是算法研究、运算研究模型、博弈论和图论的重要课题。

本章中，我们对递归进行一般性的介绍并通过大量应用程序演示其用法。在后续各章，我们将用递归研究树和排序。

10.1 递归的概念

大多数人不会自然而然地想到递归。例如，如果要求定义幂函数 x^n，其中 x 为实数，而 n 为非负整数，一种典型的做法是用 x 的重复乘积：

$$X^n = \underbrace{X * X * X * \cdots * X * X}_{n\text{ 个 }X\text{ 相乘}}$$

例如，以下是 2 的各次幂的值：

```
2^0 = 1                    // 特殊定义
2^1 = 2
2^2 = 2 * 2 = 4
2^3 = 2 * 2 * 2 = 8
2^4 = 2 * 2 * 2 * 2 = 16
```

函数 $S(n)$ 计算前 n 个正整数的和，此问题可以用重复加的方法解决。

$$S(n) = \sum_{1}^{n} i = 1 + 2 + 3 + \cdots + n - 1 + n$$

例如；对于 $S(10)$，我们可以算出前 10 个整数的和为 55。

$$S(10) = \sum_{1}^{10} i = 1 + 2 + 3 + \cdots + 9 + 10 = 55$$

如果我们想用同样的算法计算 $S(11)$，则以上加法会被重复一遍。一种更实际一些的方法是使用前面的结果 $S(10)$ 并加上 11 就得到了 $S(11) = 66$：

$$S(11) = \sum_{1}^{11} i = S(10) + 11 = 55 + 11 = 66$$

该方法利用了前一次用较小值计算出的结果，从而求出了答案。我们将其称为递归过程。

我们仍回到幂函数并将其作为递归过程来编写。当求 2 的后续幂值时，我们注意到前次幂值可以被利用来计算下次幂值。例如：

```
2^3 = 2 * 2^2 = 2 * 4 = 8
2^4 = 2 * 2^3 = 2 * 8 = 16
```

一旦我们有了 2 的初始幂值（$2^0 = 1$），其后续各幂次的值只需将前一幂次值加倍即可。用小的幂次值计算出另一值的过程便导致了幂函数的递归定义。对于实数 x，x^n 的

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

值由下式得出：

$$\begin{cases} x^n = 1 & n = 0 \\ x * x^{(n-1)} & n > 0 \end{cases}$$

可以用类似的递归定义描述函数 S(n)，其功能是求前 n 个整数的和。最简单的情况是 S(1)，其值为 1。一般情况下，我们可以用前 n－1 个整数的和 S(n－1)求 S(n)。

$$\begin{cases} S(n) = 1, & n = 1 \\ n + S(n-1), & n > 1 \end{cases}$$

如果解决问题的方法是把一个问题分解成小的子问题，并且这些小的子问题可以用同样的算法解决，那么就可以用递归。当分解到可以解决的比较简单的子问题时分解过程即终止。我们将这些子问题称为终止条件。递归运用的是“分而治之”的策略。

如果一种算法的定义组成如下，则它就是递归的。

1. 对应于某些参数可以求值的一个或多个终止条件。

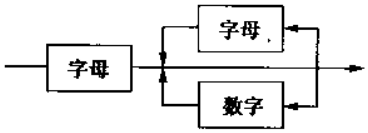
2. 一个递归步骤，它根据先前某次值求当前值。递归步骤最终必须导致终止条件。

例如，幂函数的递归定义只有一个终止条件，那就是 n＝0 的情况($x^0 = 1$)。递归步骤中描述了一般情况：

$$x^n = x * x^{(n-1)}, \quad n > 0$$

递归定义

程序设计语言在定义语法时大量使用递归方法。大多数人都熟悉“铁路”图，下图描述了“标识符”，它由字母开头，后面的任选项是字母数字串。例如，“class”，“float”和“var4”都是可以在 C＋＋程序中出现的有效标识符，而 x＋＋和 2team 则都不是标识符。



从该图的左边进入，沿铁轨移动，直到从右边出来。箭头指示沿铁轨的有效移动方向。在图中移动期间，将方框内的所有符号都包括进去。例如，图中最简单的通路是从左边进入、经过一个字母符号并一直走到出口处。这对应于单字母标识符，如 A n，t 等等。对于以下每个标识符，我们描述其在图中对应的路径。

标识符	路径
A	进入“字母(A)，”出图
XY3	进入“字母(X)”，往回折，经过“字母(Y)”，再饶回，经过“数字(3)”，出图
7A	无效标识符；没有路径

虽然“铁路”可以有效地描述某些语言，但实际上几乎所有语言的语法都是由递归术语来描述的，这种术语被称为巴科斯—诺尔范式(BNF，Bakus-Naur Form)。该范式是为了定义 Algol 60 而制定的，它由一系列产生规则构成，这些产生规则定义了非终结符如何才

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

能被其他非终结符或终结符串所替代。产生规则用"|"符号分隔备选的替换串。非终结符由尖括号"< >"括起,表示语言构成,如标识符和表达式。终结符定义语言中的实际标记串。例如:BNF 对于标识符的定义是由如下产生规则构成的。标识符是位于"::="符号左边的非终结符。

```
<标识符>::=<字母>|<标识符><字母>|<标识符><数字>
<字母>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<数字>::=0|1|2|3|4|5|6|7|8|9
```

产生规则规定了标识符要么是一个字母,要么就是由字母后跟以字母/数字串构成。注意标识符是根据自身进行定义的,其定义是递归的。

解释递归定义是对递归函数的自然应用。例如,有些编译器使用语言语法的 BNF 定义以及递归下降的语法分析算法,将程序翻译成机器代码。不同的产生规则,被编写成不同的函数,这些函数调用自身或其他产生规则结束。这一过程可能包含间接递归,例如规则 P 调用规则 Q,而规则 Q 以调用规则 P 结束。

例 10.1

简化的算术表达式仅仅允许二元运算符 +,-,* 和/,其 BNF 定义如下:

```
<表达式>::=<项>+<项>|<项>-<项>|<项>
<项>::=<因式>*<因式>|<因式>/<因式>|<因式>
<因式>::=(<表达式>)|<字母>|<数字>
```

例如:下面的式子都是表达式。

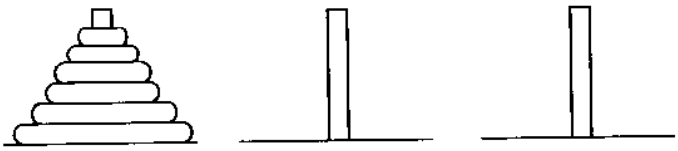
A+5,B*C+D,2*(3+4+5),(A+B*C)/D

上述规则中包含了间接递归。例如,表达式可能是项,也可能是因子,也有可能是带括弧的表达式。因此,表达式是间接地根据自身进行定义的。

递归问题

递归的强大功能使得许多问题有非常简单而优雅的解决办法。我们将概括性地介绍一些问题以及利用递归进行问题求解的技术。

汉诺(Hanoi)塔 喜欢玩智力游戏的人很久以来一直着迷于汉诺塔问题。



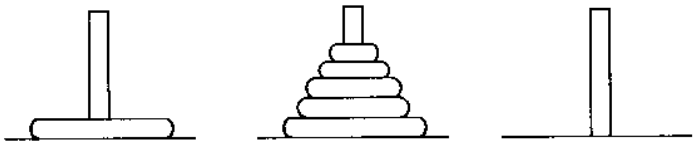
传说布拉马圣殿(Temple of Brahma)的教士们有一黄铜浇铸的平台,上立三根金刚石柱子。A 柱上堆放了 64 个金盘子,每个盘子都比其下面的盘子略小一些。当教士们将盘子全部从 A 柱移到 C 柱以后,世界就到了末日。当然,这个问题还有一些特定的条件,那就是在柱子之间一次只能移动一个盘子并且任何时候大盘子都不能放到小盘子上。教士

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

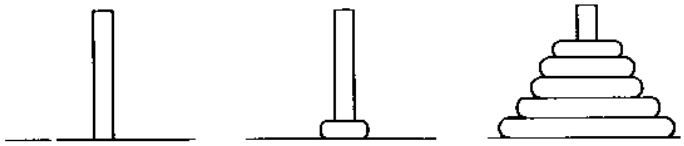
www.Linuxidc.com

们当然还在忙碌着，因为这需要 $2^{64}-1$ 次移动。如果一次移动需要 1 秒钟，那么全部操作需要 5000 亿年以上时间。

智力游戏爱好者会被汉诺塔的问题难住，而计算机科学家则可以用递归迅速解出这道题。我们用带 6 个盘子的柱子演示这个问题。我们将先把 5 个盘子移到 B 柱上，然后再把最大的盘子移到 C 柱上。



这样，问题就简化成仅仅将 5 个盘子从 B 柱移到 C 柱。用同样的算法，我们只需将上面 4 个盘子先移走，然后将大盘子从 B 移到 C 上。这样，我们要移动的只剩下 4 个盘子。这一过程反复进行下去直到只剩下一个盘子时将其移到 C 柱上。



很显然这是一种递归解法。它将问题分解成若干同样类型的小问题，移动一个盘子的简单操作就是终止条件。

迷宫 人们都熟悉走迷宫的游戏，走迷宫时我们可能面临无数的引我们走向死胡同并最终跌入陷阱的选择。心理学研究人员使用迷宫以及一块作诱饵的奶酪来研究老鼠的认知模式。我们假想动物不能得到奶酪是由于它不能进行递归思维。考察图 10.1 以及一条保证到达“终点”的策略。我们将引入一种叫“回溯”的问题求解工具。

迷宫是一些互相连通的交叉路口的集合。每个交叉路口都有 3 个关联值，分别代表左、前、右。如果值为 1，则表示相应的方向上可移动一步，值为 0 则表示在相应的方向上移动受阻。

3 个 0 值代表死胡同。如果能到达“终点”则表示我们已成功地穿越了迷宫。这个过程中包括了一系列递归步骤。我们研究在每一个交叉路口可能采取的动作。如果可能我们将首先尝试往左。如果往左是死胡同，再往回走，然后往右。如果从交叉路口出发的所有可能方向上都是死胡同，那就往回走到下一个交叉路口并进行一次新的选择。这种令人厌倦的保守策略当然谈不上什么效率，但可以保证最终找到一条走出迷宫的道路。

我们来看看上面的迷宫中前 10 步走法：

交叉路口	选择步骤	到达路口
1	往前	2
2	往左	3

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

3	往左	7
7	死胡同;退回到 3	3
3	往前走	4
4	往前走	6
6	死胡同;退回到 4	4
4	往右走	5
5	死胡同;退回到 4,3,2	2
2	往前走	8

……

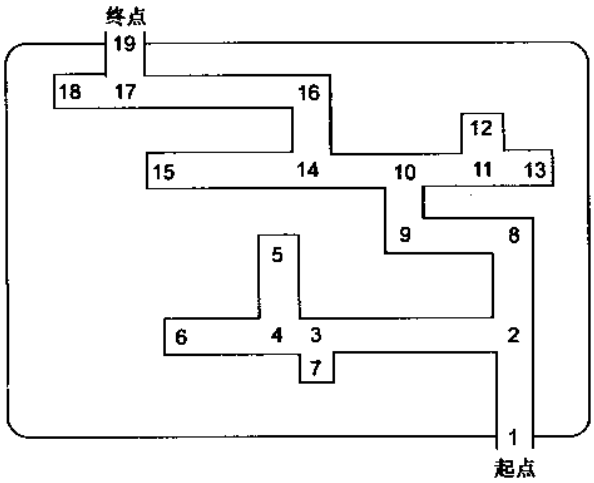


图 10.1　穿越迷宫

组合数学　递归在组合数学领域有很广的应用。组合数学研究的是对象的计数问题。例如,假设我们掷 3 个骰子并记录其总和,组合数学上的问题之一是求得到的总和为 8 时有多少种可能的掷法。递归方法试图将这个问题逐步缩小为更简单的问题并用得到的结果解决更复杂的问题。在我们这个例子中,我们将 3 个骰子视为复杂的问题,先集中解决掷 2 个骰子的简单问题。我们假定在掷 2 个骰子时得到的和 N 可以是 2 至 12 之间的任一数并可确定在掷 2 个骰子时得到的和为 N 的所有不同的方法。对于 3 个骰子和为 8 的情况,我们先掷第 1 个骰子,将其值记录在表中,其余两个骰子掷得的数之和 N 也记录在表中,例如,若第 1 个骰子为 3,那么后两个骰子之和应为 5。用我们的"两个骰子"方法,可以得出掷两个骰子其和为 5 的可能情况共有 4 种。将这些结果与第 1 次掷得的 3 合在一块,我们会发现至少有 4 种方法掷 3 个骰子得到的和为 8。

下表列出了用 3 个骰子得到 8 的 15 种方法。因为掷的次数少,所以我们可以不用递归而列出所有选项。我们还要看一个类似的例子,这个例子用表求解就显得不切实际。

第 1 个骰子	N	掷 2 个骰子的不同结果	数目
1	7	(3,4)(2,5)(1,6)(4,3)(5,2)(6,1)	6
2	6	(3,3)(4,2)(2,4)(5,1)(1,5)	5

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

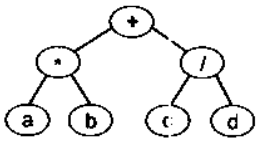
3	5	(4,1)(3,2)(1,4)(2,3)	4
4	4	(3,1)(2,2)(1,3)	3
5	3	(2,1)(1,2)	2
6	2	(1,1)	$\frac{1}{21}$

若要计算掷3个骰子得到8的几率，我们需要用21除以掷3个骰子可能出现的所有不同的结果总数，即 $6^3=216$，所以掷出8的几率为21/216即约10%。

表达式树 在堆栈研究中我们引入了中缀和后缀(RPN)格式来书写算术表达式。通过将操作符和操作数存放到二叉树中可以对这些格式进行比较。操作数被放在树梢作为叶子结点。操作符的优先级决定了它在树中所在的层次。在树中深度较大的操作符必须在深度较小的操作符之前进行运算。例如，表达式

```
a * b + c/d
```

所对应的7结点的表达式树为：



和处理迷宫一样，我们可以给出在树中每一个结点进行方案选择的递归扫描方法。例如，假设我们按以下规则行动：

如果可能，到左结点
输出结点值
如果可能，到右结点

因为是在扫描指令之间输出结点值，所以我们称之为"中序扫描"。对表达式树的一次扫描结果是按以下顺序访问结点序列。当往走或往右都无法走时终止条件即得到满足。

动作	到达结点	输出
从根结点开始	+	
往左	*	
往左	a	
(a没有左分支)		
输出值		a
(a没有右分支)		
(回到*，左边走完)		
输出值		a *
往右	b	
(b没有左分支)		
输出值		a * b
(b没有右分支)		
(回到结点+；左边走完)		
输出值		a * b +
往右	/	
往左	c	

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(c 没有左分支)

输出值　　a * b + c

(c 没有右分支)

(回到结点/,左边走完)

输出值　　a * b + c/

往右　　d

(d 没有左分支)

输出值　　a * b + c/d

(d 没有右分支)

所有右分支都访问完后,扫描终止,我们就得到了中序格式的表达式,

另一种递归扫描的次序由以下规则定义。因为输出值是在两个扫描指令之后,所以我们称之为"后序扫描"。

如果可能,到左结点

如果可能,到右结点

输出结点值

略去细节,结点按以下顺序输出:

```
ab * cd/ +
```

这是表达式的后缀式(RPN 式)。

递归是对树进行定义和扫描的强有力工具。我们将在第 11 章中大量使用递归算法。第 13 章中为了设计树的循环算法我们将写出与这些算法等价的循环算法。

10.2 设计递归函数

递归函数的结构可以用计算非负整数的阶乘问题进行说明。为了加强说明效果,我们同时写出该函数的递归和循环定义。

非负整数的阶乘 Factorial(N)被定义为所有小于或等于 N 的正整数的积。若用 N! 表示该值,则

$$N! = N * (N-1) * (N-2) * \cdots * 2 * 1$$

例如

```
Factorial(4) = 4! = 4 * 3 * 2 * 1 = 24
Factorial(6) = 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720
Factorial(1) = 1! = 1
Factorial(0) = 0! = 1     // 特殊情况
```

用循环实现该函数的方法如下:如果 n 为 0 则返回 1,否则用循环将连续各项进行相乘。

```
// 阶乘的循环算法
long Factorial(long n)
{
    int prod = 1, i;
    // 若 n = = 0 返回 prod = 1; 否则计算 prod = 1 * 2 * ... * n 的值
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    if (n > 0)
       for (i = 1; i < = n; i + +)
           prod * = i;
    return prod;
}
```

考察一下阶乘的不同例子中的各项就可以得到阶乘的递归定义。对于 4!,第 1 项为 4 而剩余各项(3 * 2 * 1)为 3!。对于 6!,结果也类似,它可由 6 与 5! 相乘而得到。

任何非负整数 n 的递归定义包括一个终止条件和递归步骤:

$$n! = \begin{cases} 1, & n = 0 \quad // \text{终止条件} \\ n * (n-1)!, & n > 0 \quad // \text{递归步骤} \end{cases}$$

我们可以把 Factorial(n)看作是计算 n! 的 n-处理机,它只需计算乘积 n * (n - 1)! 即可得出结果。为了使机器工作,它必须与一系列其他能前后传递信息的机器连网。0-处理机是个例外,因为它能独立工作并产生结果 1 而无需其他机器的帮助。我们叙述一下用 4-处理机计算 4! 时所需要的连网以及机器之间的交互动作。

4-处理机(4 * 3!)必须启动 3-处理机

3-处理机(3 * 2!)必须启动 2-处理机

2-处理机(2 * 1!)必须启动 1-处理机

1-处理机(1 * 0!)必须启动 0-处理机

图 10.2 中描述了各机器的动作。一旦启动 0-处理机,我们就立即能得到结果 1,而它将被回传给 1-处理机。接着,1-处理机就可以用它完成乘法并将结果返回给 2-处理机。

1 * 0! = 1 * 1 = 1

从 1-处理机逐级传递,一直到 4-处理机,这样就能得到所有需要用的返回值。

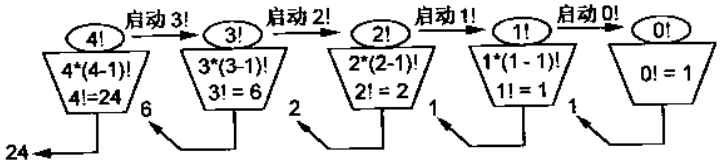


图 10.2 阶乘处理机

1-处理机使用来自 0-处理机的数值 1 并计算 1 * 0! = 1

2-处理机使用来自 1-处理机的数值 1 并计算 2 * 1! = 2

3-处理机使用来自 2-处理机的数值 2 并计算 3 * 2! = 6

4-处理机使用来自 3-处理机的数值 3 并计算 4 * 3! = 24

对 N!,求值时必须注意区别 0! 和 N > 0 时的情况,前者代表终止条件,而后者代表递归步骤。这种区分是设计递归算法的基础。程序员可以用 IF…ELSE 语句实现这种区分。IF(语句)块处理终止条件而 ELSE(语句)块处理递归步骤。对于阶乘,IF 语句块求出终止条件 N = 0 情况下的值并返回结果 1;而 ELSE 语句块处理递归步骤,它求出表达式 N * (N - 1)! 的值并返回结果。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 阶乘的递归算法
long Factorial(long n)
{
    // 终止条件为 n = = 0
    if (n = = 0)
       return 1;
    else
       // 递归计算
       return n * Factorial(n - 1);
}
```

图 10.3 中给出了用来计算 Factorial(4)的函数调用序列。假设对函数的起始调用是发生在主程序中。在函数体内,ELSE 语句执行时所带参数是 3,2,1,0。最后一次函数调用在 n=0 的条件下执行 IF 语句。一旦满足终止条件,函数调用的递归链就被打断,从而依次发生一系列计算:1 * 1,2 * 1,3 * 2 和 4 * 6。最后结果 24 被返回给主程序。

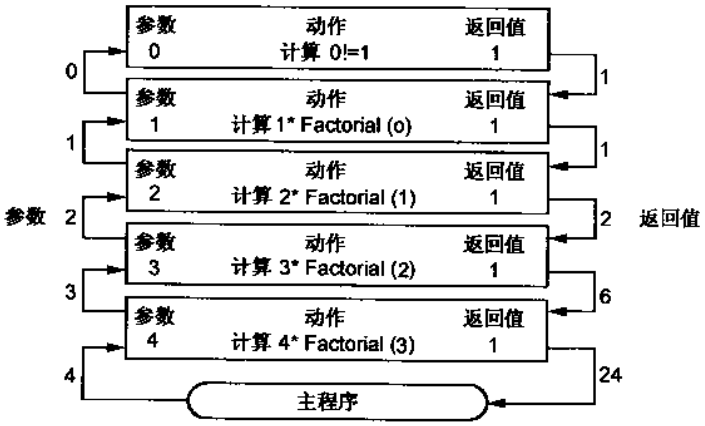


图 10.3 阶乘堆栈

例 10.2

在 10.1 节的幂函数和求和函数中由 IF…ELSE 条件区分终止条件和递归步骤。在幂函数中,power(0,0) = 0^0 未作定义,这时产生出错消息。

1. 幂函数(递归形式)

```
// 用递归计算 x^n
float power(float x, int n)
{
    // 终止条件为 n = = 0
    if (n = = 0)
      // 0^0 未作定义
      if (x = = 0)
      {
          cerr < < "power(0,0) is undefined" < < endl;
          exit(I);
      }
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        else
            // x^0 为 1
            return 1;
    else
        // 递归计算
        // power(x,n) = x * power(x,n-1)
        return x * power(x,n-1)
}
```

2. 求和函数(递归形式)

```
// 递归计算 1+2+…+n
int S(int n)
{
    // 终止条件为 n == 1
    if (n == 1)
        return 1;
    else
        // 递归计算: S(n) = n+ S(n-1)
        return n + S(n-1)
}
```

程序 10.1　阶乘的使用

本程序示范阶乘函数的递归形式。用户输入 4 个整数并打印出它们的阶乘。

```
#include <iostream.h>
// 递归计算 n!
long Factorial(long n)
{
    // 若 n == 0, 则 0! = 1;否则,n! = n*(n-1)!
    if (n == 0)
        return 1;
    else
        return n * Factorial(n-1);
}
void main(void)
{
    int i, n;
    // 输入 4 个正整数然后分别计算其阶乘值
    cout << "Enter 4 positive integers: ";
    for (i = 0; i < 4; i++)
    {
        cin >> n;
        cout << n << "! = " << Factorial(n) << endl;
    }
}
/*
<程序 10.1 运行结果>
Enter 4 positive integers: 0 7 1 4
0! = 1
7! = 5040
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
1! = 1
4! = 24
*/
```

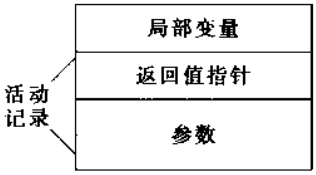
10.3 递归代码和运行时堆栈

函数是为了响应函数调用而执行的指令序列。执行过程以调用模块设置活动记录开始,记录中包括参数表以及返回调用模块以后要执行的下一条指令。

参数	位置
<实际参数>	<下一条指令>

活动记录

当函数被调用时,来自活动记录的数据被压入系统提供的堆栈(运行时栈)。这些数据与局部变量一起定义了函数可以使用的活动帧。



活动帧

程序在退出函数时读取下一条指令的位置(图 10.4)。然后删除堆栈中与活动记录相对应的数据。递归函数每次用不同的参数表重复调用自身。这一过程将一系列活动记录压入堆栈,直到满足终止条件为止。这些记录过后再相继弹出,这使我们有了递归解决的方案。阶乘函数可示意活动记录的使用。

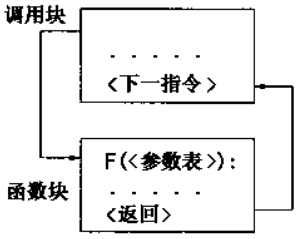


图 10.4 函数调用及返回

运行时堆栈

以 4 的阶乘 factorial(4)为例,我们示意在递归函数运行期间运行时堆栈和活动记录的使用。对 factorial 的初始调用是在主程序中。执行完函数后,程序控制返回到 N 被赋值为 24(4!)处,即位置 RetLoc1。

```
void main (void)
{
    int N;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            // 压入调用 FACTORIAL(4)的记录。
            // RetLoc1 为赋值语句 N = FACTORIAL(4)处。
            N = FACTORIAL(4);
RetLoc1
        }
```

函数 FACTORIAL 中的递归调用结束后返回到位置 RetLoc2 处，其对应指令是计算 n *(n-1)!。计算结果被存储到变量 temp 中以帮助跟踪代码并便于演示运行时堆栈的情况。

```
long FACTORIAL(long n)
{
    int temp;
    if (n = = 0)
        return 1;       // 弹出活动记录
    else
    {
        // 压入调用 FACTORIAL(n-1)的活动记录
        temp = n * FACTORIAL(n-1);
RetLoc2
        return temp;       // 弹出活动记录
    }
}
```

调用块		
FACTORIAL(1)	参数 0	返回：RetLoc2
FACTORIAL(2)	参数 1	返回：RetLoc2
FACTORIAL(3)	参数 2	返回：RetLoc2
FACTORIAL(4)	参数 3	返回：RetLoc2
main	参数 4	返回：RetLoc1

图 10.5　活动堆栈

对于函数 FACTORIAL，活动记录有两个域。

参数	位置
long n	<下一条指令>

活动记录

执行 FACTORIAL(4)可引发 5 次函数调用。图 10.5 描述了每次函数调用对应的活动记录。这些记录从底向上进入堆栈，最先进入的是栈底的主程序。

调用函数 FACTORIAL(0)时即满足了终止条件，这时一系列终止条件开始执行，堆栈顶部的活动记录被弹出，程序控制被交给返回位置处。以下操作描述了从堆栈中清除活动记录的情况。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

参数返回位置		返回处指令

0	RetLoc2	RetLoc2 temp = 1 * 1; // 从 FACTORIAL(0)可得结果为 1
		return temp; // temp = 1;

1	RetLoc2	RetLoc2 temp = 2 * 1; // 从 FACTORIAL(1)可得结果为 1
		return temp; // temp = 2;

2	RetLoc2	RetLoc2 temp = 3 * 2; // 从 FACTORIAL(2)可得结果为 2
		return temp; // temp = 6;

3	RetLoc2	RetLoc2 temp = 4 * 6; // 从 FACTORIAL(3)可得结果为 6
		return temp; // temp = 24;

4	RetLoc1	RetLoc2 N = FACTORIAL(4); // 返回主程序

10.4 用递归进行问题求解

许多计算机问题在转化成递归代码后都有一种简单而优美的表示形式。在 10.1 节中，我们已经给出了一系列的例子，包括汉诺塔、迷宫和组合数学。在这一节中，我们将扩大例子的范围，使之包括用于求解计数问题和排列的折半查找算法的递归定义，以演示用组合数学求解汉诺问塔题以及处理一般迷宫问题的迷宫类的设计。

折半查找

折半查找以指定的键值为参数扫描含 N 个元素的有序数组以查找与键值匹配的值。函数返回匹配值的下标，如果没有匹配值则返回 -1。折半查找算法可以用递归法描述。

假设有序表 A 的下界下标为 low，而上界下标为 high。给定一个键值 key，我们从表的中部(下标为 mid)开始寻找匹配值。

mid = (low + high)/2　　比较 A[mid]和键值 key

如果匹配，就满足了终止条件，这时可以停止搜索并返回下标 mid。如果不匹配，我们可以利用表是有序表这一条件集中搜索 mid 左边的“下子表”或 mid 右边的“上子表”。

如果 key < A[mid]，匹配只有可能在表的左部，下标值为 low 到 mid - 1 之间发生。

搜索 low 到 mid - 1 之间的下子表

搜索 mid + 1 到 high 之间的上子表

如果key > A[mid]，匹配只有可能在表的右半部分，下标值为mid + 1到high之间发生。

递归步骤使得折半查找继续在子表中进行。随着递归过程的进行，子表将越来越小。最后，如果子表为空则搜索失败。这种情况在表的上界比下界小时发生。条件 low > high 是第二个终止条件，这时算法返回索引值为 -1。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

折半查找(递归表) 折半查找的模板程序使用元素类型为T的数组、键值、上下界下标作为参数。IF语句处理两个终止条件:(1)发生匹配;(2)表中找到键值。IF语句的ELSE部分用递归令搜索继续在下子表(key < A[mid])或上子表(key > A[mid])中进行。将同样的"分而治之"方法用于后续的更小的区间直到成功(匹配)或失败。

```
// 在有序数组A中折半查找法的递归实现
template <class T>
int BinSearch(T A[], int low, int high, T key)
{
    int mid;
    T midvalue;
    // 终止条件: key未找到
    if (low > high)
        return (-1);
    // 与表的中点值进行比较.若与key不等,则将表分为两半,并对相应子表进行折半查找
    else
    {
        mid = (low+high)/2;
        midvalue = A[mid];
        // 终止条件: 找到key
        if (key == midvalue)
            return mid;        // key的下标为mid
        // 若key<midvalue,则查"下子表";否则,查"上子表"
        else if (key < midvalue)
            // 递归
            return BinSearch(A, low, mid-1, key);
        else
            // 递归
            return BinSearch(A, mid+1, high, key);
    }
}
```

程序 10.2 折半查找测试

程序从文件"vocab.dat"中读出一串单词,放到数组WordList中,单词表按ASCII顺序排序。用户被提示输入一个键值以引发对表的搜索。如果单词被找到,它在表中的位置就被打印出来,否则就有消息告知单词不在表中。搜索函数包含在文件"search.h"中。

```
#include <iostream.h>
#include <fstream.h>
#include "strclass.h"
#include "search.h"                    // 引入折半查找
void main(void)
{
    // 查找从fin流中读入的有序串组成的数组
    String wordlist[50];
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    ifstream fin;
    String word;
    int pos, i;
    // 打开存放已排好序的单词文件"vocab.dat"
    fin.open("vocab.dat");
    // 读入单词到 wordlist 直到文件结束
    i = 0;
    while(fin > > wordlist[i])
        i + + ;
    // 提示用户输入查找的单词
    cout < < "Enter a word:  ";
    cin > > word;
    // 用折半查找法查找该单词
    if ((pos = BinSearch(wordlist, 0, i, word)) ! = -1)
        cout < < word < < " is in the list at index "
             < < pos < < endl;
    else
        cout < < word < < " is not in the list." < < endl;
}
/*
< 输入文件"vocab.dat" >
array
class
file
struct
template
vector
< 程序 10.2 运行结果之一 >
Enter a word: template
template is in the list at index 4
< 程序 10.2 运行结果之二 >
Enter a word: mark
mark is not in the list.
*/
```

组合数学：委员会问题

组合数学提供计算一个事件可能以多少种方式发生的算法。

以经典的委员会问题为例，给出非负整数 N 和 K，我们来求从 N 个人中选出 K 个成员组成一个委员会的方法总数，即 C(N,K)。

我们从 N = 5、K = 2 的特定例子入手来对一般问题求解。由于这个例子比较简单，所以一开始我们可能只要稍微组织一下便能迅速穷举出共有 10 种不同的选择方案。假设成员名分别为 A、B、C、D 和 E。我们将委员会的所有可能情况列在成员组的周围。

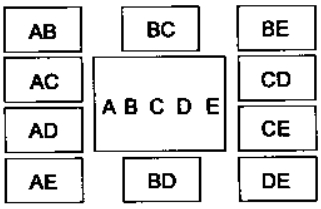
但这种方法对于较大的数就不适用了，这时需要用“分而治之”的策略将问题分解为较简单的子问题。这一过程中，我们编写出委员会问题求解方法。

简化问题的第一步是将 A 从小组中撤开，这样只剩下 B、C、D 和 E。

子问题 1：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



列出小组中剩下的4个人组成2人委员会的所有可能的情况。共有6种不同的子委员会。

表1：(B,C),(B,D),(B,E),(C,D),(C,E),(D,E)

注意每个新的委员会并不包括缺席者A。

子问题2：

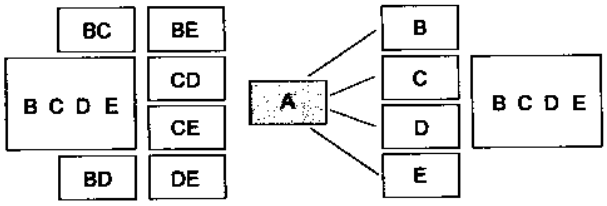
列出4个成员的小组选出1人委员会的所有可能情况。

(B),(C),(D),(E)

每个委员会要组成2人小组还缺1人。将成员A加入其中，它们即可变成两人委员会。这些两人委员会为：

表2：(A,B),(A,C),(A,D),(A,E)

可以断定子问题1和子问题2所得到的两人委员会包含了从原始问题得到的所有可能的委员会。下图描述了这两种情况：



子问题1
2个成员的委员会

子问题2
1个成员的委员会

表1中的6个小组代表不含成员A的所有委员会。表2中的4个小组代表含A的所有委员会。既然一个委员会要么含A，要么不含A，那么这两个表合并起来得到的10个委员会就构成了所有可能的两人委员会。

算法设计 我们用分而治之(递归)的分析方法处理从N个人中选出K人委员会的一般问题。先撇开其中一人(A)，考虑其余N－1个人的情况。委员会的总数为从N－1个人中选K个成员(不包括A)的情况数加上从N－1个人中选K－1个人(包括A)的可能的情况数。前一种情况的数目为C(N－1,K)，后一种情况下的数目为C(N－1,K－1)。后一种情况下，K－1个成员的委员会只要加入成员A就可以变成K个成员的委员会。

```
C(N,K) = C(N-1,K-1) + C(N-1,K)        // 递归步骤
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

终止条件包括几种极端情况，这可以分析出来。

如果 K > N，那么就没有足够的人可以组成任何委员会。从 N 个人中选出 K 人委员会的情况数为 0。

如果 K = 0，则委员会没有成员，这只有 1 种情况。

如果 K = N，则每个成员都必须在委员会中，只有选中所有人的委员会这一种情况。

```
C(N,N) = 1    C(N,0) = 1    C(N,K) = 0    K > N
```

将终止条件和递归步骤结合在一起以后，我们就可以实现递归函数 comm(n,k) = C(n,k)。该函数包含于文件"comm.h"中。

```
// 从 n 个人中选出 k 个人组成委员会的组合数
int comm(int n, int k)
{
    // 终止条件：人数太少
    if (k > n)
        return 0;
    // 终止条件：委员会为所有人或没有成员
    else if (n = = k || k = = 0)
        return 1;
    else
        // 递归步骤：A 不参加委员会的组合数加上 A 一定参加委员会的组合数
        return comm(n - 1,k) + comm(n - 1,k - 1);
}
```

程序 10.3　产生委员会

用户输入候选人数 n 和委员会成员数 k。输出值 C(n,k)即为委员会的可能数。

```
# include < iostream.h >
# include "comm.h"          // 引入函数 comm
void main (void)
{
    int n, k;
    cout < < "Enter # of candidates and # on a committee: ";
    cin > > n > > k;
    cout < < "The # of possible committees is "
         < < comm(n,k) < < endl;
}
/*
< 程序 10.3 运行结果之一 >
Enter # of candidates and # on a committee: 10 4
The # of possible committees is 210
< 程序 10.3 运行结果之二 >
Enter # of candidates and # on a committee: 9 0
The # of possible committees is 1
*/
```

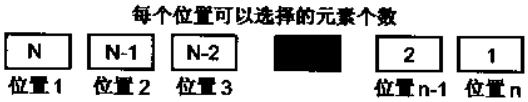

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

组合数学：排列

许多有趣的递归算法都涉及到数组。本节我们将考察产生 N 项的全排列的问题。算法涉及到用值传递数组，但因 C++用地址传递数组，所以必须进行数组复制。

N 个元素(1,2,…,N)的排列(permutation)是这些元素的有序放置。对于 N=3,(1 3 2)的排列顺序与(3 2 1)、(1 2 3)不同，依此类推。经典的组合数学问题得出排列的总数目是 N!，只要从直观上考虑一下排列的每个位置即可很清楚地得出这一结论。



排列的总数是每个位置上所有可能的选择数的乘积。

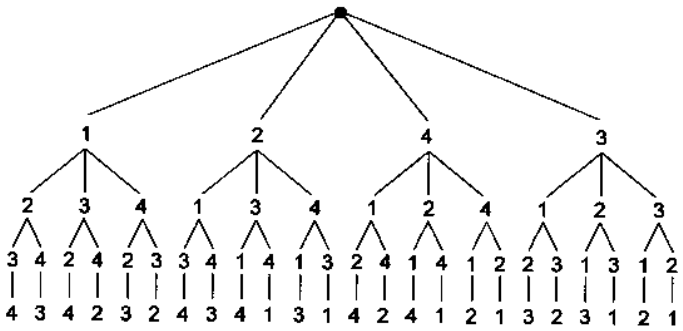
Permutation(N) = N * (N − 1) * (N − 2) * …… * 2 * 1 = N!

一种更能引起我们兴趣的递归算法可以列出 N≥1 时，N 个元素的所有排列，为了便于说明，我们列出 4 个元素的 24(4!)种排列，这些排列分成 4 列，每列的第一个元素都相同。每列又分成 3 对，表示每对的第 2 个元素相同。

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

一棵层次树中所含的有序路径与排列相对应并可示意计算排列数的算法。开始时，有 4 种选择——1、2、3、4——对应于 4 个分支。沿着树向下移动，以下各层分别分解为 3、2、1 个分支。总的路径(排列)数是：

4 * 3 * 2 * 1 * = 4!



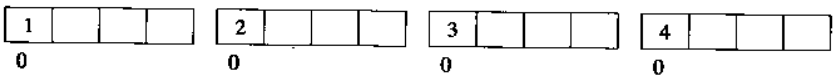
列出所有排列的算法就相当于走遍这棵树的所有路径。从一层推进到下一层，也就是指定排列的下一个位置。这种过程构成了递归步骤。

索引 0：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

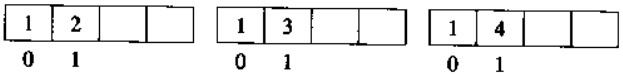
www.Linuxidc.com

用迭代过程求出排列中索引 0 对应的第 1 个位置的所有可能选择。本例中，第 1 项共有 N = 4 种可能。



索引 1：

在树的下一层，每个结点分解为除第 1 项外的 N—1 项所组成的 N - 1 个结点。例如，结点 1 对应于以索引 0 的值为 1 开始的排列。再往下一层，包含结点 2 的分支路径对应于头两个位置为 1、2 的排列，依此类推。通过扫描第 2、3、4 项，我们可以用循环过程求出排列的第 2 项。

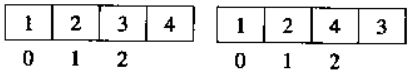


索引 2：

在树下一层，每个结点又分出两个分支，它们代表两项的排列。例如，在结点 2，这两项就是[3,4]，其有序排列是：

[3,4]的排列

最后得到的 4 个项的排列是：



索引 3：

因为排列不允许重复，所以表中第 3 项一旦被确定以后，最后一项也就定了。最后一项也就是终止条件，每个结点都完成 1 个不同的排列。

算法设计 实现算法的 C + + 代码以 N 个元素的数组的形式存储每个排列。每次调用之前，递归函数建立数组的一个副本，这样从递归步返回以后各个值仍保持在数组的原来的位置上。记住我们实现的 N = 4 的算法最后必须得到 24 个不同数组。开始时，我们建立第 1 个位置分别为 1,2,3,4 的 4 个数组。往下走一层，这 4 个数组的每 1 个都建立 3 个数组。它们保留基数组的第 1 项，依此类推。

```
void copy(int x[], int[], int n)
{
    for (int i = 0; i < n; i + + )
        x[i] = y[i];
}
```

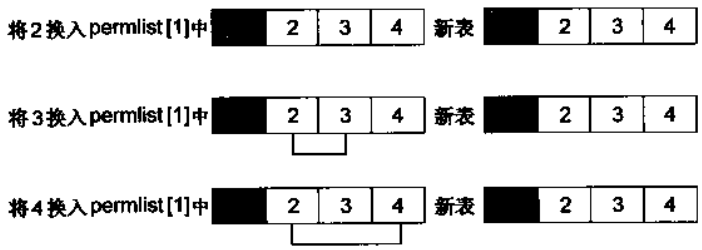
递归算法按索引 0,1,2,…,N - 1 的次序依次将各项放在数组 Permlist 中。

1. 终止条件发生在索引 N - 1 处，这时 N 元素的排列完成，数组各项被打印出来。
2. 递归步骤：递归步骤从索引 0 开始向前推进，一直到 N - 2。

   在索引为 k(0≤k < N - 1)时，前 k 项已经设置在数组 Permlist 中。我们可重复扫描其他项并将它们放在位置 Permlist[k]中。具体做法是将表的其余部分的每一项与 Permlist[k]的值相交换。例如，假设 N = 4，k = 1，且 Permlist[0] = 1。迭代步骤将 2,3,4 放到索引 1 处。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



每次交换后得到的表都被复制到1个临时表中并被传递给递归函数 Permute，同时传递的还有下一个索引和参数 N。图 10.6 中示意了以 Permlist[1] = =2 时调用 Permute 的情况。

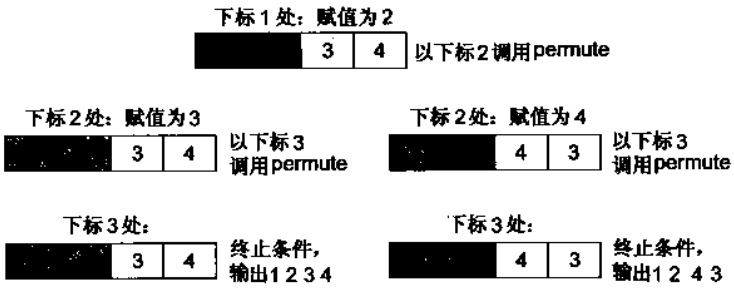


图 10.6　排列 12xx

按(1234)和(1243)的顺序建立起排列。图 10.7 中示意了将 3 换到 Permlist[1]时调用 Permute 的情形。然后再将 4 换到 Permlist[1]，递归过程从数组 1432 开始继续下去。

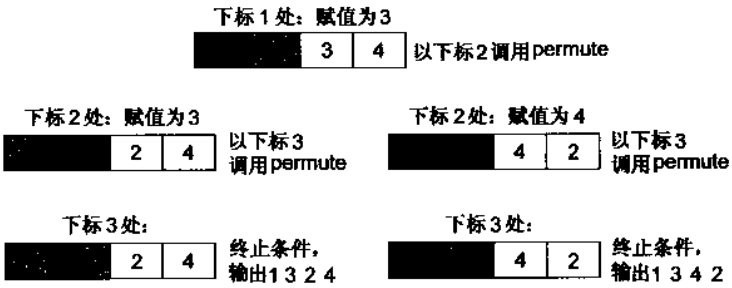


图 10.7　排列 13xx

递归函数 Permute

```
// UpperLimit 为排列元素的最大数
coust int UpperLimit = 5;
// 从数组 y 中拷贝 n 个元素到数组 x
void copy(int x[], int y[], int n)
{
    for (int i = 0; i < n; i++)
        x[i] = y[i];
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
// permlist 为一 n 元整型数组。本函数产生一个对下标 i(start <= i <= n-1)的元素的
// 排列。排出一种情况后,输出整个数组。若从 start=0 开始,则是对所有 n 个元素排列。
void permute(int permlist[], int start, int n)
{
    int tmparr[UpperLimit];
    int temp, i;
    // 终止条件:已到数组的最后一个元素
    if (start == n-1)
    {
        // 输出本次排列结果
        for (i=0; i < n; i++)
            cout << permlist[i] << " ";
        cout << endl;
    }
    else
        // 递归步骤:交换元素 permlist[start]和 permlist[i]的值,将数组拷贝到
        // temparr,然后从 start+1 开始到数组结束对 temparr 进行全排列
for (i=start;i < n; i++)
{
    // 交换 permlist[i]和 permlist[start]的值
    temp = permlist[i];
    permlist[i] = permlist[start];
    permlist[start] = temp;
    // 产生一个新表后调用 permute
    copy(tmparr, permlist, n);
    permute(tmparr, start+1, n);
    }
}
```

程序 10.4　排列

本题处理 N=3 时的排列情况。函数 Permute 包含在文件"permute.h"中。

```
#include <iostream.h>
#include "permute.h"        // 引入函数 Permute
void main(void)
{
    // permlist 存放我们要排列的 n 个数
    int permlist[UpperLimit];
    int n, i;
    cout << "Enter a number 'n' between 1 and "
         << UpperLimit << ":";
    cin << n;
    // 初始化 permlist 为{1,2,3,…,n}
    for (i=0; i < n; i++)
        permlist[i] = i+1;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

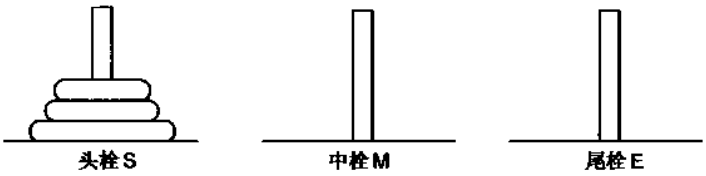
www.Linuxidc.com

```
    cout << endl;
    // 从下标 0 到 n-1 输出 permlist 数组中的排列结果
    permute(permlist, 0, n);
}
/*
<程序 10.4 运行结果>
Enter a number 'n' between 1 and 5: 3
1  2  3
1  3  2
2  1  3
2  3  1
3  1  2
3  2  1
*/
```

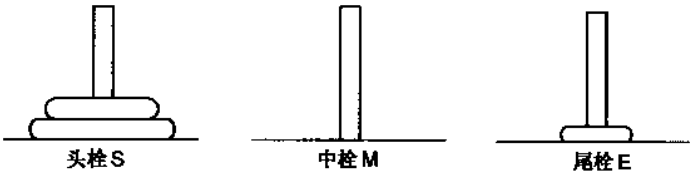
汉诺塔 10.1 节中讨论的汉诺塔游戏是递归算法的一个例子，用这种算法可以很简单地解决问题并可隐去细节。本节编写重放盘子的递归步骤和终止条件。

游戏表述：游戏台上有 3 个木栓，它们分别是头栓、中栓和尾栓。头栓上串 N 个盘子，这些盘子按从大到小的顺序堆放，最大的盘子在最底层。游戏的目标是将 N 个盘子从头栓移到尾栓。盘子一次只能移动一个并且大盘子不能放到小盘子上。

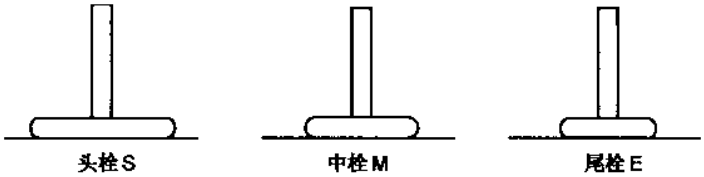
图 10.8 中描述了 N = 3 的情况下木栓之间盘子的移动情况。木栓的标记分别为头栓(S)、尾栓(E)、中栓(M)。我们用这个相对简单的例子来说明递归过程，而递归算法是根据 N 个盘子的情况给出的。



步骤1 ：将小盘移到E (S->E).



步骤2：将中盘移到M(S->M).



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

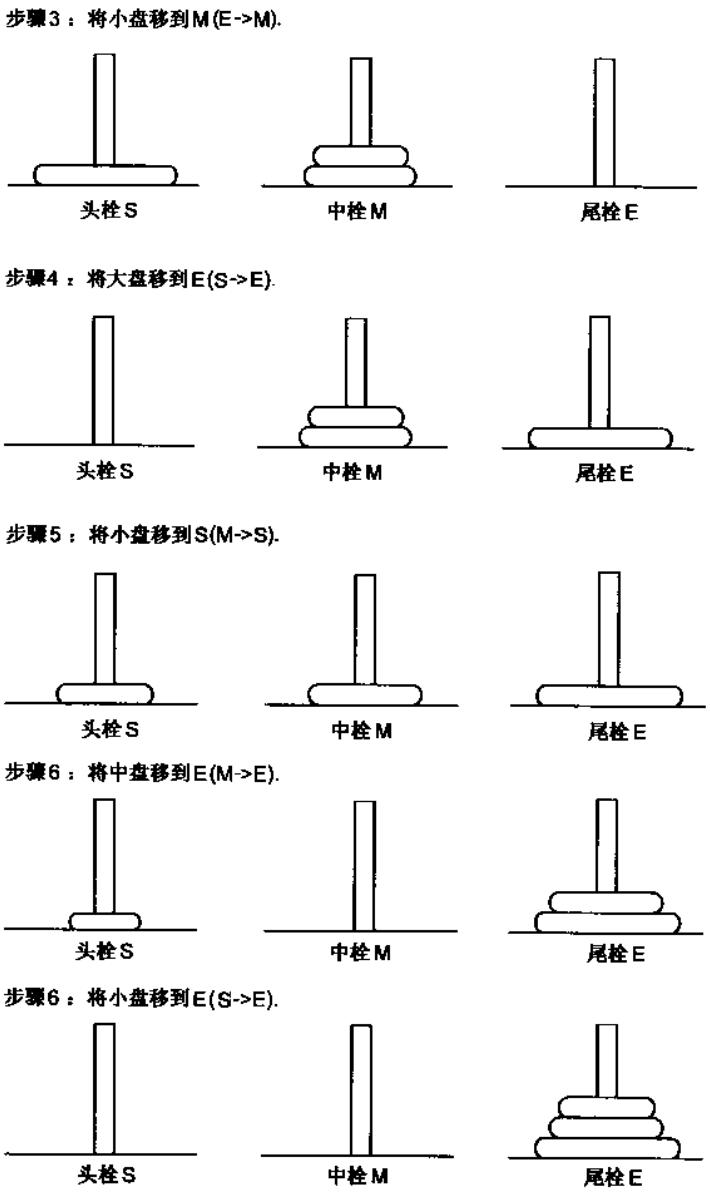


图 10.8　汉诺塔($N=3$)

汉诺塔算法　3 个盘子的例子可以被推广到 3 步递归算法(图 10.9)。函数 Hanoi 将木栓声明为 String 对象。在参数表中,变量的次序是:

startpeg-middlepeg-endpeg

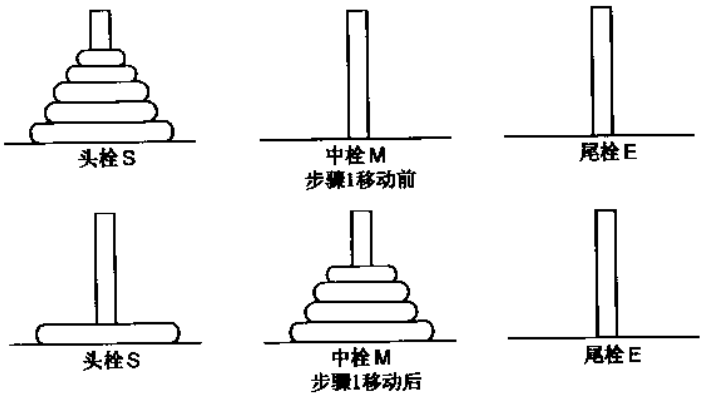
并且约定我们将盘子从头栓移到尾栓,用中栓暂时存放盘子。如果 N = 1 则碰到特殊的终止条件,这时将唯一的 1 个盘子从头栓移到尾栓即可。

```
cout << "move" << startpeg << "to" << endpeg << endl;
```

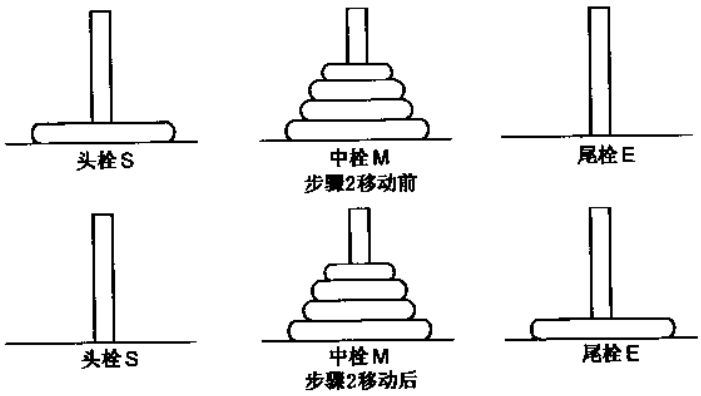

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

步骤 1：将 N－1 个盘子从 S 移到 M。



步骤 2：将 1 个盘子从 S 移到 E。



步骤 3：将 N－1 个盘子从 M 移到 E。

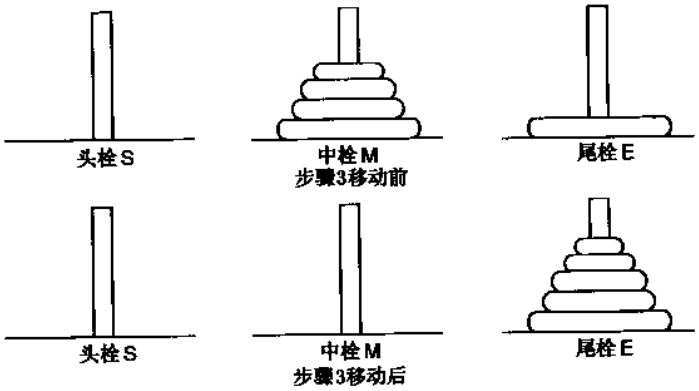


图 10.9　汉诺塔中盘子的移动

如果 N 不等于 1，我们就用 3 步过程将 N 个盘子从头栓移到尾栓。第 1 步是利用尾栓将 N－1 个盘子从头栓移到中栓，递归函数调用的参数次序是：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 用 endpeg 作为临时栓
Hanoi(n-1,startepeg,endpeg,middlepeg);
```

第 2 步只需将最大的盘子从头栓移到尾栓：

```
cout << "move" << startpeg << " to " >> emdpeg >> emdl;
```

第 3 步将 N-1 个盘子从中栓移到尾栓，用头栓作临时存放之用，递归函数调用的参数次序是

```
// 用 startpeg 作临时栓
Hanoi(n-1,middlepeg,startpeg,endpeg);
```

程序 10.5　汉诺塔

作为参数传递给函数的栓名为“start”，“middle”和“end”。程序一开始先提示用户输入盘子数 N，然后调用递归函数 Hanoi 以求出将 N 个盘子从“start”栓移到“end”栓的移动步骤。

整个算法需要 2^N-1 次移动。对于 10 个盘子的情况，此游戏需要 1023 次移动。在我们的测试例子中，N=3，移动步数是 $2^3-1=7$。

```
#include <iostream.h>
#include "strclass.h"
// 将 n 个盘子从 startpeg 栓移至 endpeg 栓，用 middlepeg 作为临时栓
void hanoi (int n, String startpeg, String middlepeg,
            String endpeg)
{
    // 终止条件：移动一个盘子
    if (n == 1)
       cout << "move  " << startpeg << " to " << endpeg << endl;
    // 将 n-1 个盘子移到 middlepeg 栓，将底盘移到 endpeg 栓，
    // 然后再将 n-1 个盘子从 middlepeg 栓移至 endpeg 栓
    else
    {
        hanoi(n-1, startpeg, endpeg, middlepeg);
        cout << "move  " << startpeg << " to "<< endpeg << endl;
        hanoi(n-1, middlepeg, startpeg, endpeg);
    }
}
void main()
{
    // 盘子个数及各个栓名
    int n;
    String startpeg = "start ",
           middlepeg = "middle",
           endpeg = "end  ";
    // 提示用户输入 n 并输出 n 个盘子的移动办法
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    cout << "Enter the number of disks: ";
    cin >> n;
    cout << "The solution for n = " << n << endl;
    hanoi(n, startpeg, middlepeg, endpeg);
}

/*
<程序 10.5 运行结果>

Enter the number of disks: 3
The solution for n = 3
move start      to end
move start      to middle
move end        to middle
move start      to end
move middle     to start
move middle     to end
move start      to end
*/
```

走迷宫 许多递归算法都使用回溯原理，该原理适用于当我们面对若干步骤和抉择时的情况。为了获得最终解法，我们一步一步地建立符合最终答案要求的分步解法。如果走出的一步或作出的一个决定与最终答案不符，则回溯一步或几步至上一次与最终答案相符之处。这正如古谚语所云："进两步，退一步。"

回溯有时可能需要退 n 步(n 为一较大的值)才能进 1 步。这一节中，我们将在读者熟知的迷宫环境中讨论回溯。在分析中，假定迷宫中没有可能使得我们在其中兜圈子的回路。这条限制并非必须，因为只要我们维持一张指示途经结点是否被再次访问的地图，那么回溯同样运用于带回路的迷宫。图 10.10 中给出了一个带回路的迷宫，回路由交叉点 2,3,4,5 组成。

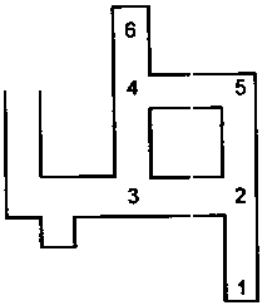


图 10.10 带回路迷宫

算法分析 迷宫是交叉点的集合。旅行者从一个方向进入并沿以下三条路径之一出发：向左、向前或向右。路径由下一个路口的号码表示。如果沿路径走没有出口，我们用 0 值表示在该方向的移动受阻。

没有出口的交叉点是死胡同。

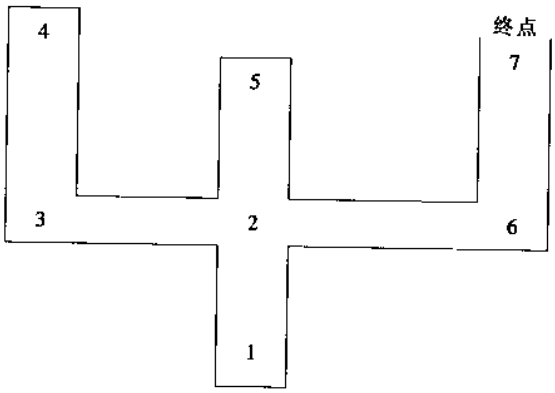
本着冒险精神和愿意回溯的决心，旅行者进入迷宫，到起始路口并开始大胆地搜索目标——终止路口、通向自由的出口。路径上的每个路口代表一个分解步骤。遗憾的是，冒险可能导致走向死胡同并需要回溯到路径的前一路口。

为了进行选择，我们对每个交叉点运用递归策略。我们首先尝试向左出发(如果有路的话)并建立一条通往终点的路径。仅当这种抉择引导我们走向死胡同时，它才与最终目标不一致，在这种情况下，我们后退到交叉点并尝试向前方出发并向终点前进。同样，如果碰到死胡同便不合要求，这时选择右边的路径。如果这一选择仍不成功，那么当前交叉点就是死胡同，我们退回到前一个有效的("一致的")交叉点。以上策略描述起来较简单，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

将其编写为递归函数也比较容易。从图 10.11 的 7 个路口的迷宫的部分走法中可以明显看出这是一个需要编写递归函数的问题。走出迷宫的最终道路是沿路口 1—2—6—7 走出。



路口	动作	结果
1	向前	进入 2
2	向左	进入 3
3	向右	进入 4
4(死胡同)	回溯	进入 3
3(死胡同)	回溯	进入 2
2	向前	进入 5
5(死胡同)	回溯	进入 2
2	向右	进入 6
6	向左	进入 7 (终点)

图 10.11 小迷宫

走迷宫的策略保证一旦旅行者走出路口,则只有沿这条路的所有可能选择都已被尝试过且都导致死胡同的情况下才会折返回来,更进一步说,只有确信从一个路口出发能走出迷宫时,我们才将其正式放到路径上。一旦旅行者到达了迷宫的终点,我们可以回溯行走的过程并标出路径上的每个路口。

Maze 类 迷宫结构由数据(路口)和使得我们可以建立迷宫并用遍历策略遍历迷宫的方法组成。我们假定每个路口是一个记录,它的域代表向左、向前和向右走的结果。整数域值代表该方向路径上下一个路口。0 值表示该方向不通。路口记录是以结构 Intersection 实现的。

```
// 给出从当前路口向左、向前、向右能到达的路口的记录
struct Intersection
{
    int left;
    int forward;
    int right;
};
```

Maze 由以下各项组成:一个表示迷宫大小的整数值、迷宫出口、用动态数组分配的迷宫路口列表。所有数据访问由构造函数和另外一种方法提供,前者建立迷宫,后者遍历迷

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

宫以寻找出路。迷宫数据从文件中读入，这些数据包括路口的数目、每个路口的 3 个 exit 值，以及出口“EXIT”的号码。例如，图 10.11 中的小迷宫的数据如下：

```
6        // 交叉点个数
0 2 0    // 1: 向前进入 2
3 5 6    // 2: 向左到 3;向前到 5,向右到 6
0 0 4    // 3: 向右到 4
0 0 0    // 4: 4 为“死胡同”
0 0 0    // 5: 5 为“死胡同”
7 0 0    // 6: 向左到终点
7        // 终点的号码
```

Maze 类定义

声明

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

class Maze
{
    private:
        // 迷宫中路口个数及出口号
        int mazesize;
        int EXIT;
        // 迷宫路口的数组
        Intersection * intsec;
    public:
        // 构造函数;从文件 < filename > 中读数据
        Maze(char * filename);
        // 遍历并求解迷宫问题
        int TraverseMaze(int intsecvalue);
};
```

说明

包含迷宫数据的文件名被传递给构造函数。此外我们还标明路口的数目以便为动态数组 intsec 分配内存。

TraverseMaze 是一个递归函数，用于寻找迷宫的解法。参数 intsecValue 初始化为 1，表示旅行者从路口 1 进入迷宫。在递归过程中，该变量维护要考虑的当前路口的数目。迷宫类的定义和实现包含在文件“maze.h”中。

Maze 类的实现

构造函数负责迷宫的设置，这包含打开输入文件、读取迷宫的尺寸、初始化路口数组并标明出口。

```
// 从文件 filename 中读入路口及出口号建立迷宫
Maze::Maze(char * filename)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    ifstream fin;
    int i;
    // 打开 filename.若文件不存在则退出
    fin.open(filename, ios::in | ios::nocreate);
    if (! fin)
    {
        cerr << "The maze data file" << filename
             << " cannot be opened!" << endl;
        exit(1);
    }
    // 文件中第一个数为迷宫的路口数
    fin >> mazesize;
    // 申请数组存放迷宫的路口。由于不用下标 0,故需申请 mazesize+1 个记录。
    // 若下一路口号为 0,则是"死胡同"
    intsec = new Intersection[mazesize+1];
    // 从文件中读入各路口情况
    for (i = 1; i <= mazesize; i++)
        fin >> intsec[i].left >> intsec[i].forward
            >> intsec[i].right
    // 读入出口号并关闭文件
    fin >> EXIT;
    fin.close();
}
```

递归策略是由 TraverseMaze 方法管理的,它以当前路口号(intsecValue)作为其参数。

在前一个路口调用函数,如果能找到以当前路口到出口的路径则返回值 1(TRUE)。如果 intsecValue 值为 0,则表示"碰壁"并立即返回 0(FALSE)。

方法的核心是一棵决策树,它使得旅行者能沿着最终走向出口的路径前进。

情况 1:如果 intsecValue == EXIT,则成功到达目的地。这时我们打印出路口的值并将 TRUE 值回送给正在等待我们是否成功的结果的前一个路口。

情况 2:如果不在 EXIT 处,则从路口向左并等待表示通往左边的路径是否成功的 TRUE 或 FALSE 消息。如果得到的消息为 TRUE,我们打印出当前路口并将 TRUE 值回送给前一个路口。

情况 3:与 2 类似。只是若此时不在出口且向左已经失败时,我们尝试向前。若成功,则打印出当前路口并将 TRUE 值回送给前一路口。

情况 4:与 2 和 3 类似的,处理向右的情况。

如果上述情况下都没有返回消息 TRUE,则当前路口是一个死胡同。这时返回消息 FALSE 以表明事实。将信息回送给前一个路口(TraverseMaze 的前一个实例)的能力来自于代码的递归结构。最终,TraverseMaze(1)返回 TRUE 或 FALSE 给主程序以表示迷宫是否有一自由通路。

```
// 用回溯法解迷宫问题
int Maze::TraverseMaze(int intsecvalue)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 若 intsecvalue 为 0,我们处于"死胡同";否则,我们继续尝试发现一条路径
    if (intsecvalue > 0)
    {
        // 终止条件: 我们已到达出口
        if (intsecvalue = = EXIT)
        {
            // 输出路口号并返回 TRUE
            cout < < intsecvalue < < " ";
            return 1;
        }
        // 尝试向左
        else if (TraverseMaze(intsec[intsecvalue].left))
        {
            // 输出路口号并返回 TRUE
            cout < < intsecvalue < < " ";
            return 1;
        }
        // 向左为死胡同.试试向前
        else if (TraverseMaze(intsec[intsecvalue],forward))
        {
            // 输出路口号并返回 TRUE
            cout < < intsecvalue < < " ";
            return 1;
        }
        // 向左和向前均为死胡同.试试往右
        else if (TraverseMaze(intsec[intsecvalue].right))
        {
            // 输出路口号并返回 TRUE
            cout < < intsecvalue < < " ";
            return 1;
        }
    }
    // 死胡同,返回 FALSE
    return 0;
}
```

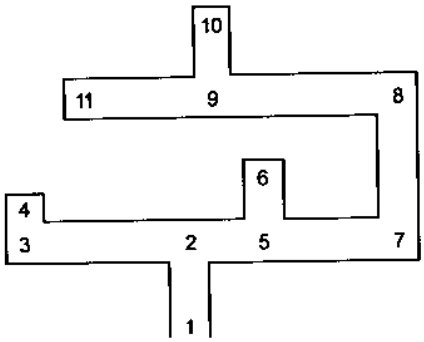
程序 10.6　解迷宫

我们先测试图 10.11 中的小迷宫(输入文件"maze1.dat"),然后测试下面的迷宫(输入文件"maze2.dat"):此迷宫无解。最后再遍历图 10.1 中的大迷宫(输入文件"bigmaze.dat")。上述每种情况下的路径都以逆序打印出来。

```
#include <iostream.h>
#include "maze.h"            // 引入 maze 类
void main(void)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
{
    // 存放迷宫的文件
    char filename[32];
    cout << "Enter the data file name: ";
    cin >> filename;
    // 读入文件,建立迷宫
    Maze M(filename);
    // 遍历迷宫并输出结果
    if (M.TraverseMaze(1))
       cout << endl << "You are free!" << endl;
    else
       cout << "No path out of the maze" << endl;
}
/*
<程序 10.6 运行结果之一>
Enter the data file name: maze1.dat
7  6  2  1
You are free!
<程序 10.6 运行结果之二>
Enter the data file name: maze2.dat
No path out of the maze
<程序 10.6 运行结果之三>
Enter the data file name: bigmaze.dat
19  17  16  14  10  9  8  2  1
You are free!
*/
```

10.5 递归评估

递归通常并非解决问题的高效方法。考察阶乘问题,迭代算法用 for 循环而不用递归算法中的重复函数调用。具有讽刺意味的是递归简化了算法设计和编码,但却降低了运行效率。这种冲突可以用 Fibonacci 数列示意:

1,1,2,3,5,8,13,21,34

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

数列 F(n)是递归定义的，n≥1。头两项直接定义为 1，以后各项定义为前两项的和。

$$F(n)=\begin{cases}1 & 若\ n=1\ 或\ 2\\ F(n-1)+F(n-2) & 若\ n>2\end{cases}$$

此定义可以直接转化为递归函数。假设 F(n)是 Fibonacci 数列的第 n 项，n≥1。

```
终止条件：F(1) = 1          F(2) = 1
递归步骤：For N ≥ 3, F(n) = F(n-1) + F(n-2);
```

C++函数 Fib 实现递归函数 F。它以整数 n 为参数并返回一个长整数作为结果。

```
// 用递归法求第 n 个 Fibonacci 数
long Fib(int n)
{
    // 终止条件：f1 = f2 = 1
    if (n = = 1 || n = = 2)
       return 1;
    // 递归步骤：Fib(n) = Fib(n-2) + Fib(n-1)
    else
        return Fib(n-1) + Fib(n-2)
}
```

你马上可以看到函数 Fib 用相同的参数对自身进行了多次调用。例如，考虑 N=6 的情况。表达式 Fib(n-1)+Fib(n-2)建立一棵 N=5,4,3,2,1 时的 Fib 调用的层次树。(图 10.12)

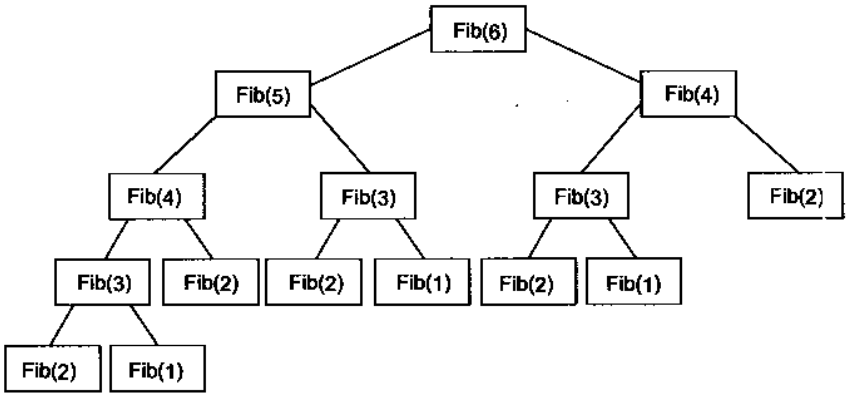


图 10.12　求 fib(6)的递归调用数

注意 Fib(3)被计算 3 次，而 Fib(2)被计算 5 次。树上的 15 个结点代表计算 Fib(6)=8 时需要进行的递归调用次数。

总调用次数 15 与值 Fib(6)直接关联。假设 NumCall(k)是计算 Fib(k)的递归调用次数，则

```
NumCall(k) = 2 * Fib(k) - 1
```

例如：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
NumCall(6) = 2 * Fib(6) - 1 = 2 * 8 - 1 = 15
NumCall(35) = 2 * Fib(35) - 1 = 2 * 9277465 - 1 = 18 554 929
```

算法复杂度为 $O(2^n)$。递归算法的运行时间是指数量级的。

Fibonacci 数：迭代式 第 n 个 Fibonacci 数用一简单的循环进行迭代计算。函数的算法复杂度为 O(n)。

```
// 迭代法计算第 n 个 Fibonacci 数
long FibIter(int n)
{
    long twoback = 1, oneback = 1, current;
    int i;
    // FibIter(1) = FibIter(2) = 1
    if (n = = 1 || n = = 2)
       return 1;
    // current = twoback + oneback, n > = 3
    else
        for (i = 3; i < = n; i + +)
        {
            current = twoback + oneback;
            twoback = oneback;
            oneback = current;
        }
    return current;
}
```

对于第 k 个 Fibonacci 数($k \geq 3$)，迭代式需要进行 k－2 次加法和 1 次函数调用。对于 k = 35 的情况迭代式需进行 33 次加法，而递归函数需要进行 185 万次函数调用！

用公式计算 Fibonacci 数 计算 Fibonacci 数的最有效的方法是直接使用由递归方程求出的公式。公式的推导过程超出了本书范围，故从略。

$$F(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

因为平方根和幂函数属于 C + + 的 < math.h > 库，所以第 n 个 Fibonacci 数可以直接用 O(1)时间复杂度计算出来。

```
# include < math.h >
const double sqrt5 = sqrt(5.0);
// 用代数公式法计算第 n 个 Fibonacci 数
double FibFormula(int n)
{
    double p1, p2;
    // C + + 数学库函数 double pow(double x, double y)计算 x 的 y 次法
    p1 = pow((1 + sqrt(5))/2.0,n);
    p2 = pow((1 - sqrt(5))/2.0,n);
    return (p1 - p2)/sqrt(5);
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

程序 10.7　递归评估(以 Fibonacci 为例)

程序分别统计用公式法、迭代法以及文件"fib.h"中的递归函数计算第 35 个 Fibonacci 数所用时间。非递归函数在几分之一秒内可以执行完,而递归函数则需要超过 82 秒的时间。

```
#include <iostream.h>
#include "fib.h"
void main(void)
{
    int i;
    // 将结果 FibFormula 用一无小数的定点数输出
    cout.setf(ios::fixed);
    cout.precision(0);
    // 用三种方法计算第 35 个 Fibonacci 数
    cout << FibFormula(35) << endl;
    cout << FibIter(35) << endl;
    cout << Fib(35) << endl;
}
/*
<程序 10.7 运行结果>
9227465        <公式法用不到 1 秒的时间>
9227465        <迭代法用不到 1 秒的时间>
9227465        <递归函数用了 82 秒>
*/
```

递归评价　Fibonacci 数的例子对于使用递归的潜在问题是个很好的警示。由于函数调用的额外开销,一个简单的递归函数也有可能严重地损害运行时性能。更严重的是,一次递归的调用可能产生一层接一层的递归调用,其堆砌会超出程序员的控制并且对堆栈的需求也会超出可用栈空间的范围。Fibonacci 的例子是一个极端的情况。实际上迭代式是很容易设计和实现的。

尽管有上述警示,递归仍然是一个重要的设计和编程工具。许多算法用递归更便于叙述和设计。它们很自然地就可以用终止条件和递归步骤实现递归。例如,在迷宫问题中可以用递归加快回溯过程。

虽然递归不是面向对象的概念,它却具有面向对象程序设计所具有的好处。它允许程序员管理算法中的一些关键逻辑部件而隐藏其复杂的实现细节。关于何时使用递归没有一条硬性规定。必须权衡一下设计和运行时复杂度。当强调算法设计且在运行时有合理的空间和时间复杂度时可使用递归。

尾部递归　如果函数的最后一个动作是进行一次递归调用,我们则说该函数使用尾部递归。这种递归调用需要要额外开销去建立活动记录并将其存到堆栈中。当递归过程遇到终止条件后,我们必须执行一系列返回步骤以将活动记录从堆栈中弹出,这些记录被放到堆栈中又取出来,却没有用它们进行什么有用的计算。

消除尾部递归可以显著地改善递归函数的效率。一个简单的例子可以示意这个问题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

并能得到一个一般求解方法。考察递归函数 refunc，它从下标 n 到 0 打印出数组中的各项。这个例子并不实际，它只是被用来对尾部递归提供简单的示意。

```
void recfunc(int A[], int n)
{
    if (n > = 0)                // 若 n 在范围内则继续
    {
        cout < < A[n] < < " ";
        n - - ;                 // 将下标 n 减 1
        recfunc(A,n);
    }
}
```

假设数组 A[] = {10,20,30}。以 n = 2 开始对函数 recfunc(A,n)进行初始调用，其输出为 30　20　10。

函数 recfunc 示意了尾部递归的典型情况。我们可以用流控制图示意这个问题，其中 n≥0 被作为需要进一步递归的条件。

```
 ┌→ if <递归条件>   ←- - n > = 0
 │     扫行任务     ←- - 输出 A[n]
 │     修改条件     ←- - 将 n 减 1(n - - )
 │
 └──────调用 recfunc
```

此流控制图等价于一个测试条件为 n≥0 的 WHILE 循环。在 recfunc 中，用低效的递归操作将控制转到条件上。

```
 ┌→ while <条件>   ←- - n > = 0
 │     扫行任务     ←- - 输出 A[n]
 │     修改条件     ←- - 将 n 减 1(n - - )
 │
 └──────
```

本例中，递归函数可以用逻辑上等价于 WHILE 语句的 interfunc 函数替代。消除尾部递归的问题只需一些小的技巧。比较保险的方法是建立递归函数的流控制图，然后用 WHILE 建立同样的迭代图。

```
// 迭代法消除尾部递归
void iterfunc(int A[], int n)
{
    while (n > = 0)
    {
        cout < < "While value    " < < A[n] < < endl;
        n - - ;
    }
}
```

书面作业

10.1　解释为什么下列函数执行后可能给出错误值。

```
long factorial(long n)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    if (n = = 0 || n = =1)
        return 1;
    else
        retrun n * factorial( - - n);
}
```

结果依赖于编译器对操作数进行求值的顺序。如果 n=3,并且先对右操作数求值,则结果为 3*2! =6。如果先对右操作数求值,结果为 2*2! =4

10.2 以下代码中的递归函数 f 所产生的数值序列是什么?

```
long f(int n)
{
    if (n = = 0 || n = =1)
        return 1;
    else
        retrun 3 * f(n-2) + 2 * f(n-1);
}
```

10.3 以下代码中的递归函数 f 所产生的数值序列是什么?

```
int f(int n)
{
    if (n = = 0)
        return 1;
    else if(n = = 1)
        return 2;
    else
        return 2 * f(n-2) + f(n-1);
}
```

10.4 如果以下程序的输入为 5 和 3,则输出是什么?

```
# include <iostream.h>
long f (int b, int n)
{
    if (n = = 0)
        return 1;
    else
        return b * f(b,n-1);
}
void main(void)
{
    int b,e;
    cin > > b > > e;
    cout < < f(b,e) < < endl;
}
```

10.5 如果以下程序的输入为"This is interesting!",则输出是什么?

```
# include <iostream.h>
void Q(void)
{
    char c;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    cin.get(c);
    if (c ! = '\n')
      Q();
    cout < < c;
}
void main(void)
{
    cout < < "Enter a line of text:" < < endl;
    Q();
    cout < < endl;
}
```

10.6 n 个元素的数组的最大元素可以用递归计算出来。定义函数

```
int max(int x, int y);
```

它返回 x 和 y 两个整数中的较大值。定义函数

```
int arraymax(int a[], int n);
```

它使用递归返回数组 a 的最大元素值

终止条件：n = = 1

递归步骤：arraymax = max(max(a[0],…,a[n-2]),a[n-1])

10.7 编写递归函数

```
float avg(float a[], int n);
```

它返回含有 n 个浮点型元素的数组中各元素的平均值。

终止条件：n = = 1

递归步骤：avg = ((n-1)/n) * (n-1 个元素的平均值) + (第 n 个元素)/n

10.8 编写递归函数

```
int rstrlen(char s[]);
```

以计算字符串的长度。

终止条件：s[0] = = 0(空串)

递归步骤：length(s) = 1 + length(从第 2 个字符开始的子串)

10.9 编写一个递归函数测试一个字符串是否回文。回文即是指顺读和倒读都是一样的不含空格的串。例如，下面几个单词都是回文：

dad level did madamimadam

用以下声明

```
int pal(char A[],int s,int e);
```

pal 判断 A 中从下标 s 开始一直到下标 e 的字符是否构成回文。

终止条件：

s > = e(成功)

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

A[s]! = A[e](失败)

递归步骤：A 中从下标 s+1 开始到 e-1 之间的字符是否回文？

10.10 二项式系数是指$(x+1)^n$的展开式中的系数$C_{n,k}$：

$$(x+1)^n = C_{n,n}x^n + C_{n,n-1}x^{(n-1)} + C_{n,n-2}x^{(n-2)} + \cdots + C_{n,2}x^2 + C_{n,1}x^1 + C_{n,0}x^0$$

注意 $C_{n,n}$和 $C_{n,0}$对于任何 n 其值都为 1。二项式系数的递归关系可由下式给出：

$$C(n,0) = 1$$
$$C(n,n) = 1$$
$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

注意每个系数 $C(n,k)(0 \leqslant k \leqslant n)$都是书中提到并求解的委员会问题的解。二项式系数 $C(n,k)$确定了从 n 个项中选出 k 项来的方法数。

这些系数来自于著名的 Pascal 三角形。在此三角形中，第 0 列都是 1，对角线(行号 = 列号)上的值也是 1。其余每个元素都是同一列中上一行的值与其左边的值的和。

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
```

编写函数，对于给定的整数 n 建立 Pascal 三角形。上例中 $n=4$。

10.11 编写递归函数

```
int gcd(int a, int b);
```

计算正整数 a 和 b 的最大公约数。见第 6 章函数递归形式的软件补充。

10.12 下面的数据表示迷宫的输入文件。画出迷宫并按递归算法一步一步求解。

```
11              // 路口数
0     2     0   // 路口 1:(左,前,右)
4     3     6
0     0     0
0     0     5
0     0     0
7     0     0
8     11    9
0     0     0
0     0     10
0     0     0
12    0     0
12              // 出口点
```

上机题

10.1 用书面作业 10.6 题中的函数 arraymax 完成以下动作：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

1. 在 1—20 000 范围内随机产生 10 个整数并将它们存到数组中。
2. 打印出数组。
3. 调用 arraymax 并打印结果。验证其正确性。

10.2 前 n 个正整数的和可由以下公式得出：

$$1+2+3+\cdots+n=n(n+1)/2$$

初始化数组 A 以包含前 50 个整数，这时可以求出这些数组元素的平均值为 51/2 = 25.5。检查书面作业 10.7 题中你的答案，方法是对数组 A 调用 avg。

10.3 测试书面作业 10.8 题中递归函数 rstrlen，方法是用 cin.getline 从键盘读取 5 个串并用 rstrlen 和 C++ 库函数 strlen 打印串长度。

10.4 读字符串一直到文件结束，用流运算符"<<"读取用空格隔开的"单词"。对于每个单词，用书面作业 10.9 中的递归函数 pal 来判断其是否回文。若是，将其赋值给串数组中的一个元素。到文件结尾时，打印找到的回文，每行一个。

10.5 在一个特定的数制系统中打印整数值是一种经典的算术问题：

$N=45$　进制基数 = 2　输出：1011101　$[32+8+4+1]$
$N=90$　进制基数 = 8　输出：132　$[1(64)+3(8)+2(1)]$
$N=75$　进制基数 = 5　输出：300　$[3(5^2)+0(5)+0(1)]$

一般的算法是用基数 B 进行重复除以实现到指定数制之间的转换，如果

$$N=d_{n-1}d_{n-2}d_{n-3}\cdots d_1d_0$$

则余数序列按 d_0 到 d_{n-1} 的次序给出 N 中的数字。

定义递归函数

```
void intout(long N, int B);
```

在 B 进制下打印 N，假定 B≤10。在主程序中测试该函数并读入 5 对 N，B 值，在 B 进制下打印出 N 的值。

10.6 读取正整数 n，$n<10$。参考书面作业 10.10 题打印出二项式 $(x+1)^n$ 的展开式。幂 x^i 以 $x**i$ 的形式打印。

10.7 编制一个递归函数，统计 n 位二进制数中不含连续两个 1 的数的个数。（提示：二进制数以 0 或 1 开始，如果以 0 开始，则可能的数目由其余 $n-1$ 位数字决定；如果以 1 开始，则下一位数字必须是什么？）

10.8 数学、工程和其他学科中常见的问题之一是求解实方程的根，如果 f(s)是函数，f 的根 r 是指满足 f(r) = 0 的实数。有些情况下，根可以用代数公式求出。例如，二次方程 $f(x)=ax^2+bx+c$ 的所有根可由以下公式求出：

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

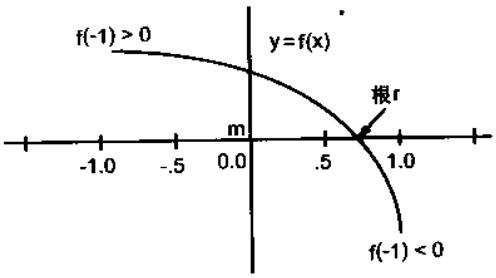
一般情况下没有公式，只能用数值方法求根。

如果 f(a)和 f(b)符号相反（f(a) * f(b) < 0），并且 f 是单调函数，那么 f 在 a 和 b 之间有一根 r。

二分方法定义如下：

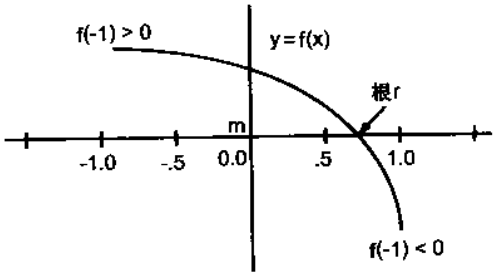
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



令 m = (a + b)/2.0 是区间 a≤x≤b 的中点。如果 f(m)等于 0.0,则 r = m 是根,否则,要么 f(a)和 f(m)符号相反(f(a) * f(m) < 0),要么 f(m)和 f(b)符号相反(f(m) * f(b < 0)。

如果 f(m) * f(b) < 0,则根 r 在区间 m≤x≤b 内;否则在区间 a≤x≤m 内,下面取原区间长度一半作为新区间,重复上述操作。如此反复下去直到区间足够小或找到精确的根为止。



f(0) * f(− 1) > 0, f(0) * f(1) < 0.
根在区间
0 < r < 1.0 之间

编写递归函数。

```
double Bisect(double f (double x),
              double a, double b, double precision);
```

计算作为参数传递的函数 f(x)的近似根。如果 f(m)等于 0.0 或子区间小于指定精度则终止二分法。

(a) 求 $f(x) = x^3 - 2x - 3$ 在 1 和 2 之间的根;

(b) 有函数如下:

```
double Balance(double principal, double interest,
               int nmonths, double payment);
```

它对给定的本金 principal 按月息 interest 支付 n 个月(nmonths)单利后计算并返回余额 balance。用二分法计算一笔年息 10%,金额 150 000 的贷款 25 年应付款额(payment)。

10.9 用书面作业 10.12 中的数据运行程序 10.6。验证你的答案。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第11章 树

11.1 二叉树结构
11.2 设计 TreeNode 函数
11.3 树扫描算法的使用
11.4 二叉搜索树
11.5 二叉搜索树的使用
11.6 BinSTree 的实现
11.7 实例研究：索引(concordance)
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

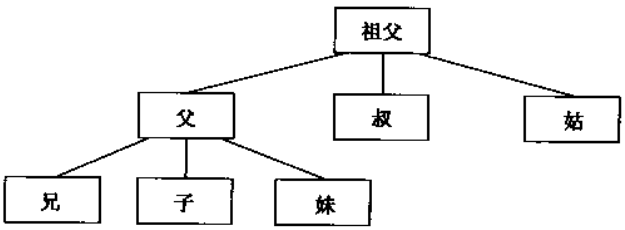


图 11.1　家族树

数组和链表定义的是顺序访问的对象的集合。它们的数据结构被称作线性表，这是因为它们具有唯一的头元素和尾元素，中间各项都有唯一后继。线性表是数组、堆栈、队列和链表等结构的统称。

在许多应用中，对象呈现一种非线性的次序，其成员可能有多个后继。例如，描述一棵家族树时，双亲可能有多个后代(孩子)。图 11.1 中示意了从孩子角度看一个家庭的三代情况。用类似的排序方法可以描述一个公司内的上下级关系：从首席执行官(CEO)开始往下排，其下一级分为部门首长和经理(图 11.2)。这种排序被称作等级制度，其来源是宗教中的权力划分：从主教到牧师、到主祭，如此下去。

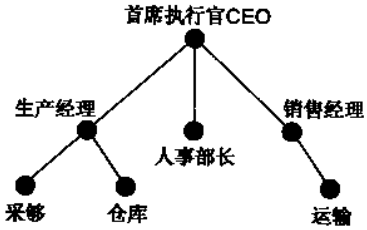


图 11.2　层次结构

在本章中，我们将研究一种非线性结构，我们称之为“树”，它由结点和叶子组成。其组织方式是从根向外层结点(被称作树的叶子)扩展。第 13 章中，我们将研究一种描述非线性结构的图，在这种结构中，两个或更多的结点可能会通向同一个对象。和图 11.3 中的通信网格一样，这些结构需要单独进行算法设计，可应用于特殊领域。

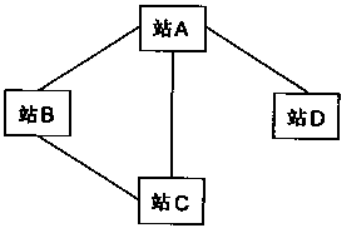


图 11.3　电话交换站

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

树术语

树结构的特点是它是由唯一起始点“**根(root)**”开始的“**结点**”集合。图 11.4 中，结点 A 是根。如果用家族树的概念，一个结点可被看作“**双亲**”，它指向 0 个、1 个或更多的被称作“**孩子**”的结点。例如，结点 B 是孩子 E 和 F 的双亲。结点的孩子和这些孩子的孩子被称为“**后代**”，而结点的双亲和祖辈被称为“**祖先**”。例如，结点 E、F、I 和 J 是 B 的子孙。每个非根结点都有一个唯一的双亲，而每个双亲可能没有也可能有多个子结点。没有孩子的结点，如 E、G、H、I、J，被称为“**叶子**”结点。

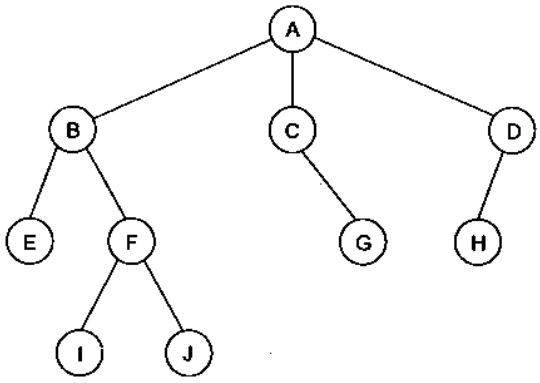
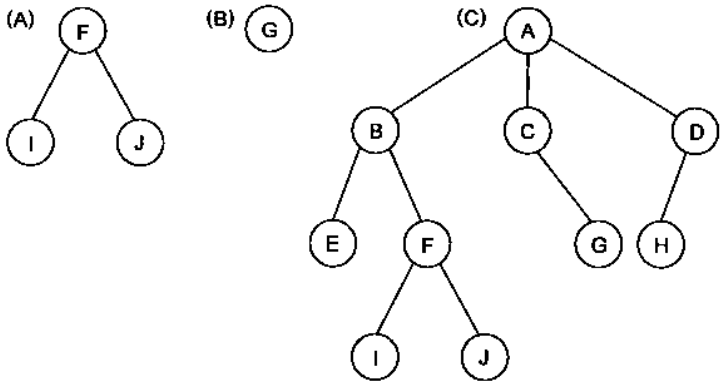


图 11.4　树

树中的每个结点都是一棵“**子树**”的根。这个子树是由结点和结点的后代定义的。下图示意了图 11.4 中的树的 3 棵子树。结点 F 是包含结点 F、I 和 J 的子树的根。结点 G 是没有后代的子树的根。根据定义，我们也可以说结点 A 是以这棵树本身作为子树的根。



从父结点移动到孩子和其他后代所经路线叫“**路径**”。例如，在图 11.5 中，从根 A 到结点 F 的路径是从 A 到 C，从 C 到 F。由于每个非根结点只有一个双亲，这就保证从任一结点到其后代之间只有唯一一条路径。从根到结点之间的路径可以提供一种被称作结点的“**层次**”这样的度量。结点的层次等于从根到结点之间路径的长度。根的层次为 0，根的每个孩子的层次为 1，下一层结点的层次为 2，以此类推。例如，图 11.5 中，F 是路径长度为 2、层次为 2 的结点。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

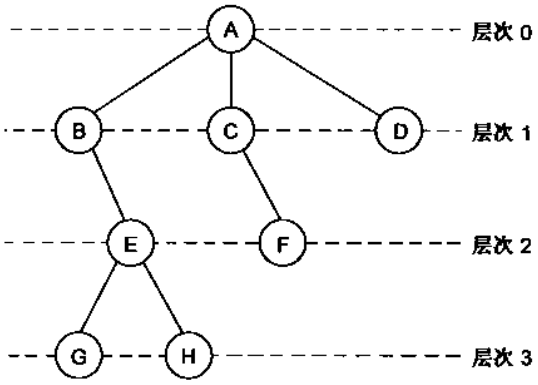


图 11.5　树的结点层次和路径长度

树的"**深度**"是树中所有结点层次中的最大值。深度的概念可以用路径来描述，树的深度是从根到结点之间最长路径的长度。图 11.5 中，树的深度为 3。

二叉树

虽然普通的树也可以有极其重要的应用，我们还是重点讨论一种特定的树，在这类树中，每个双亲的孩子数不超过两个(图 11.6)。这些"**二叉树**"具有统一的结构、有多种扫描算法可供使用并可提供对元素的高效访问。

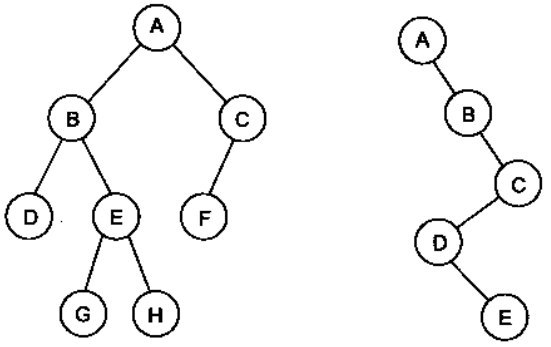


图 11.6　二叉树

在二叉树中，每个结点有 0 个、1 个或 2 个孩子。我们将左边的结点称作"**左孩子**"，右边的结点称为"**右孩子**"。标记"左"或"右"是我们对树形象化的表示。二叉树是一种递归结构。每个结点都是其子树的根并有孩子，这些孩子分别是被称为结点的左子树和右子树的根。树访问例程很自然地就是一种递归过程。以下是二叉树的递归定义。

二叉树是满足下列条件的结点 B 的集合。

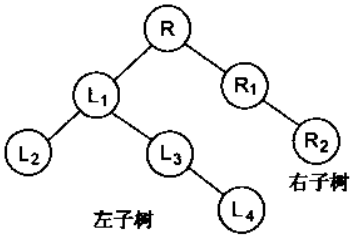
(a)　如果结点集为空，B 是一棵树。(空树亦是树。)

(b)　B 可划分为 3 个独立的子集。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

$\{R\}$	根结点
$\{L_1, L_2, \cdots, L_m\}$	R 的左子树
$\{R_1, R_2, \cdots, R_n\}$	R 的右子树



在任一层次 n,二叉树可能包含 1 到 2^n 个结点。每一层的结点数决定了树的密度。直观上说,密度是相对于树的深度而言对树的大小(结点数)的一种度量。图 11.6 中,树 A 包含了深度为 3 的 8 个结点。而树 B 包含了深度为 4 的 5 个结点。后一种情况是特殊形式,其树被称为"退化"树,其中只有一个叶子结点(E),每个非叶子结点只有 1 个孩子。一棵退化树等价于一个链表。

密度较大的树在数据结构上很重要,因为它们在接近树的顶部离根较近的地方集中了较大比例的元素数目。一棵致密的树使得我们可以存储大量数据并对这些项进行有效访问。快速访问是用树来存放数据的关键。

退化树是密度度量中的极端情况。另一种极端情况是深度为 N 的"**完全二叉树**",其中从 0 到 N-1 各层的结点都是满的,而第 N 层的所有叶子结点占据树的最左边的位置。第 N 层包含 2^N 个结点的完全二叉树是一棵"**满二叉树**",满二叉树是其每个非叶子结点都有两个孩子的二叉树。图 11.7 示意了完全二叉树和满二叉树。

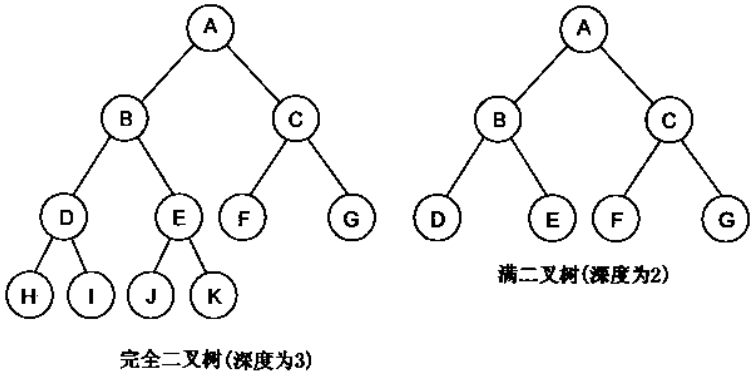


图 11.7　二叉树分类

深度为 k 的完全二叉树和满二叉树具有一些有趣的数学特性。这两种树在第 0 层(根)都只有 1(2^0)个结点;在第 1 层有 2(2^1)个结点;在第 2 层有 4(2^2)个结点,依此类推。一直到 $k-1$ 层,总共有 2^k-1 个结点。

$$1+2+4+\cdots+2^{k-1}=2^k-1$$

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

在第 k 层，结点数在最小值 1 到最大值 2^k（满）之间。对于满树。结点数为：

$1+2+4+\cdots+2^{k-1}+2^k=2^{k+1}-1$

完全二叉树中的结点数 N 满足不等式：

$2^k \leqslant N \leqslant 2^{k+1}-1<2^{k+1}$

对 k 求解，我们得到：

$k \leqslant \log_2(N)<k+1$

例如，深度为 3 的满二叉树有 $2^4-1=15$ 个结点。

例 11.1

1. 具有 5 个结点的树的最大深度为 4[图 11.6(B)]。而具有 5 个结点的树的最小深度为

$k \leqslant \log_2(5)<k+1$

$\log_2(5)=2.32,\ k=2$

2. 树的深度是从根到结点之间的路径的最大长度。对于具有 N 个结点的退化树，其最长路径图长度是 $N-1$。

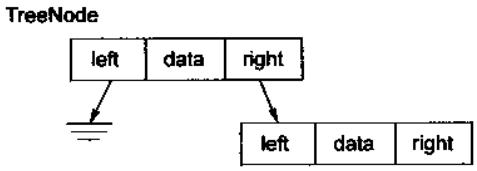
对于具有 N 个结点的完全二叉树，树的深度是 $\log_2 N$ 取整后的值。这也是从根到结点之间路径的最大长度值。假设完全二叉树有 $N=10\,000$ 个元素，最长路径是：

$\mathrm{int}(\log_2 10000)=\mathrm{int}(13.28)=13$

11.1 二叉树结构

二叉树结构是由结点生成的。像链表一样，这些结点包含数据域和指向集合中其他结点的指针。在这一节中，我们将定义树结点并提供能用来建立和扫描二叉树的操作。与第 9 章中的 Node 类的表示类似，我们定义类 TreeNode，然后再设计一系列用来建立二叉树和扫描各个结点的函数。

一个树结点(TreeNode)包含一个数据域和两个指针域。指针域被称为"**左指针(left)**"和"**右指针(right)**"，因为它们分别指向结点的左、右子树。NULL 值表示一棵空树。



根结点定义进入二叉树的入口点，而指针域指示树中下一层次的结点。叶子结点的左右指针均为 NULL。

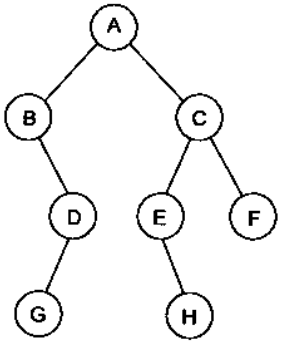
设计类 TreeNode

这一节我们设计用来定义二叉树中结点对象的类 TreeNode。结点中的数据域是作为

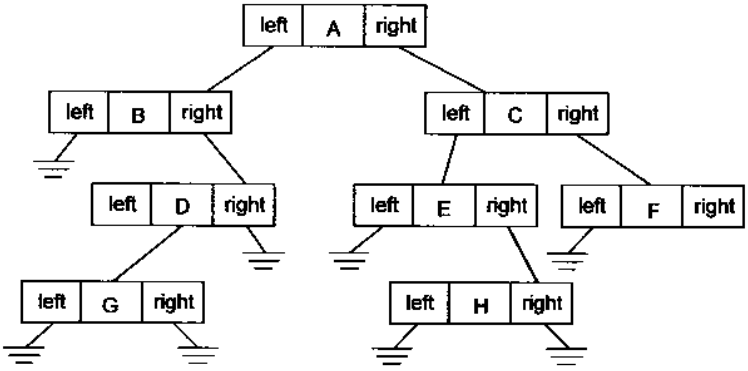
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

公有成员给出的，这样用户可以直接访问其值。这样客户程序在扫描树的时候就可以读取和更新数据并使得数据值的引用可以被返回。这一特性被用于诸如辞典一类的高级数据结构中。两个指针域是私有成员，它们可以由公有成员函数 Left 和 Right 访问。类 TreeNode 的声明和定义包含于文件“treenode.h”中。



树



TreeNode 结构

图 11.8　二叉树结点

类 TreeNode 定义

声明

```
// BinSTree 依赖于 TreeNode
template < class T >
class BinSTree;

// 为二叉树定义树的结点对象
template < class T >
class TreeNode
{
    private:
      // 指向结点左右孩子的指针
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

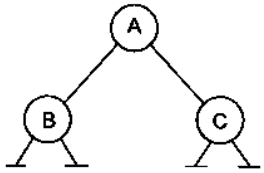
```
    TreeNode<T> *left;
    TreeNode<T> *right;
  public:
    // 公有成员,允许外部修改的数据值
    T data;
    // 构造函数
    TreeNode(const T& item, TreeNode<T> *lptr = NULL,
                            TreeNode<T> *rptr = NULL);
    // 访问指针域的函数
    TreeNode<T> * Left(void) const;
    TreeNode<T> * Right(void) const;
    // 由于 BinSTree 需要访问结点的左右指针,我们将它设计为友元函数
    friend class BinSTree<T>;
};
```

说明

构造函数初始化数据和指针域。用缺省指针 NULL 将结点初始化为叶子结点。构造函数以 TreeNode 指针 P 为参数,将 P 作为新结点的左孩子或右孩子。

例

```
// 指向整数树结点的指针
TreeNode<int> *root, *lchild, *rchild;
TreeNode<int> *p;
// 申请叶子结点,使其数据值分别为 20 和 30
lchild = new TreeNode<int> (20);
rchild = new TreeNode<int> (30);
// 创建数据值为 10 的根结点,其左右孩子结点分别为 lchild 和 rchild
root = new TreeNode<int> (10,lchild, rchild);
```




```
root->data = 50;        // 将 50 赋给根结点
```

TreeNode 类的实现 TreeNode 类初始化对象的域。构造函数以 item 为参数初始化数据域。指针将左右孩子(子树)赋值给结点。当结点没有左孩子或右孩子时便用缺省值 NULL。

```
// 构造函数;初始化结点的数据和指针域,对于空子树,将其指针域赋值为 NULL
template <class T>
TreeNode <T>::TreeNode(const T& item, TreeNode<T> *lptr,
        TreeNode<T> *rptr):data(item), left(lptr), right(rptr)
{}
```

方法 Left 和 Right 返回左指针域和右指针域,这使得客户程序可以访问结点的左右孩子。

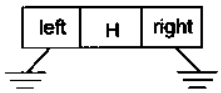
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

建立二叉树

一棵二叉树由通过指针域相连接的 TreeNode 对象的集合组成。TreeNode 对象是通过调用函数 new 动态建立的。

```
TreeNode < int >  * p;              // 定义指向整数树结点的指针
p = new TreeNode(item);             // 左右指针域均为 NULL
```



当调用函数 new 时，必须带一个数据值。如果同时传递一个 TreeNode 指针，则新分配的结点用指针连接孩子。我们定义函数 GetTreeNode，它用给定数据值以及 0 个或多个 TreeNode 指针来分配和初始化二叉树的结点。如果内存不足，程序给出错误信息并终止。

```
// 创建左右指针域分别为 lptr 和 rptr 的对象 TreeNode，指针的缺省值为 NULL
template < class T >
TreeNode < T >  * GetTreeNode(T item, TreeNode < T >  * lptr = NULL,
                              TreeNode < T >  * rptr = NULL)
{
    TreeNode < T >  * p;

    // 调用函数 new 创建新的结点，将参数 lptr 和 rptr 传递给函数
    p = new TreeNode < T > (item, lptr, rptr);

    // 若内存不够，输出错误信息后退出程序
    if (p = = NULL)
    {
      cerr < < "Memory allocation failure! \n";
      exit(1);
    }

    // 返回指针
    return p;
}
```

函数 FreeTreeNode 以 TreeNode 指针为参数，通过调用 C + + 函数 delele 释放相应结点的内存。

```
// 释放与结点相连的动态内存
template < class T >
void FreeTreeNode(TreeNode < T >  * P)
{
    delete p;
}
```

以上两个函数都包含于文件“treelib.h”中。第 11.2 节中所介绍的一系列二叉树函数也在此文件中。

定义样本树　函数 GetTreeNode 可以用来明确定义树上的每个结点因而也能定义整个树。这种方法可以用分别含数据值 10，20 和 30 的 3 结点树示意。对于规模较大的例子，这一过程可能会显得有些令人厌倦，因为必须包括所有数据和指针值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

为本章使用之便，我们编写函数 MakeCharTree 建立 3 棵树，其结点所含数据都是字符型。下一节我们将用这些树来讲解函数 TreeNode。MakeCharTree 的参数包括树根的引用和一个指代树的参数 $n(0 \leqslant n \leqslant 2)$。以下声明建立一个被称为根的 TreeNode 指针并将其作为 Tree_2 的根。

```
TreeNode<char> *root;        // 定义 root 指针
MakeCharTree(root,2);        // 建立基于 root 的 Tree_2
```

图 11.9 中画出了用上述方法建立的 3 棵字符树。函数 MakeCharTree 的完整形式在文件"treelib.h"中。函数将例 11.2 中的方法扩大到 5 个和 9 个结点的情况。

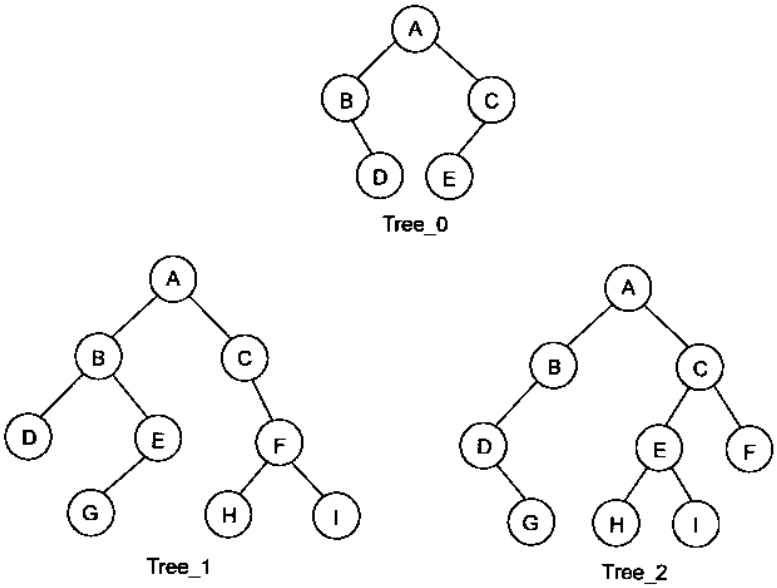


图 11.9　用 MakeCharTree 建立的树

11.2　设计 TreeNode 函数

链表是一种线性结构，它允许我们用指针 next 去顺序遍历结点。因为树是一种非线性结构，所以不能用类似的遍历算法。我们只能从各种遍历算法中进行选择，常用的有前序、中序以及后序遍历。这些方法的依据都是二叉树的递归结构。

遍历算法是有效使用树的基础。我们先设计递归遍历算法，然后用它们编制打印树、复制和删除树以及确定树的深度的算法。这一节中，我们还将设计用队列存储结点的深度优先算法。这种遍历方法从根开始逐层对树进行扫描，先到孩子的第一代、然后是第二代，如此进行下去。这种方法在机构树中有重要应用。在这种树中，领导关系从顶层向各层次的机构推进。

实现遍历方法时要用到一个叫 visit 的函数参数，它用来访问结点数据，我们可以用函数参数指定遍历过程中每个结点处要发生的动作。对 C++ 中用函数作为参数的语法下面将有一简单论述。

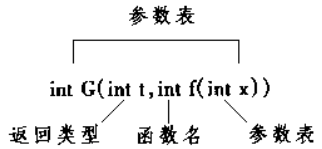
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
template <class T>
void <Traversal_Method> (TreeNode<T> *t, void visit(T& item));
```

每次调用遍历方法时，客户程序必须传递对结点中的数据实施动作的函数名。随着遍历过程从一个结点到另一个结点，函数被调用，动作被实施。

注：函数参数的概念虽然较简单但有必要作些澄清。一般来说，通过指定函数名，其参数表和返回值就可以用函数作变元。例如，函数 G 带一个函数参数 f。参数指明了函数名(f)、参数表(int x)以及返回类型(int)。




```
int G(int t, int f(int x))          // 函数参数 f
{
    // 用函数 f 和参数 t 计算函数 f(t),返回该值与 t 的积
    return t * f(t);
}
```

客户程序若要调用函数 G 必须用相同的结构传递函数 f。下面的例子中，客户程序定义计算 x^2 的函数 XSquared。

```
// X 一参数和返回值均为整数的函数
int xsquared(int x)
{
    return x * x;
}
```

客户程序用整数参数 t 以及函数 XSquared 调用函数 G。语句

```
Y = G(3, XSquared)
```

调用函数 G，G 则用参数 3 调用函数 XSquared。cout 语句打印输出结果 27。

```
cout << G(3.0, xsquared) << endl;
```

递归树的遍历

二叉树的递归定义规定二叉树的结构是由根和左、右子树组成，其中左右子树分别由根的左右指针域所定义。递归的威力在遍历方法上可以清楚地体现出来。每种树遍历算法都对一个结点实施 3 个动作：访问结点(N)、递归遍历左子树(L)以及右子树(R)。转到子树以后，算法确定其结点并实施同样的 3 个动作。遇到空树(指针 = = NULL)时遍历终止。各种递归算法的不同之处在于它们对结点实施动作的次序。在我们编制的中序和后序方法中都是先走左子树再走右子树。其他方法则留作练习用。

中序遍历 中序扫描中对一个结点实施的第 1 个动作是转到左子树以便扫描该子树中的结点。在递归遍历完左子树后，中序遍历对结点实施第 2 个动作，对结点的数据值进行处理。遍历对结点完成的最后一个动作是递归扫描右子树。在递归扫描过程中，对每

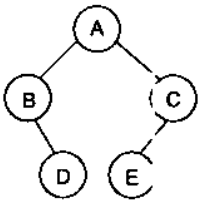
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

个新结点重复同样的动作。

中序遍历的操作次序如下：

1. 遍历左子树
2. 访问结点
3. 遍历右子树



Tree_0

这种遍历方法被称作 LNR（左、结点、右）。对于函数 MakeCharTree 中的树 Tree_0，假设"visit"输出结点中的数据域。

中序遍历 Tree_0 时要执行的操作如下：

动作	打印	观察
从 A 到 B，转到左子树：		B 的左孩子为 NULL
访问 B；	B	
从 B 到 D，转到右子树：		D 是叶子结点
访问 D；	D	A 的左子树访问完毕
访问根 A	A	
从 A 到 C，转到右子树：		E 是叶子结点
从 C 到 E，转到左子树：		
访问 E；	E	
访问 C；	C	完毕！

结点的遍历次序是 B D A E C。递归函数先递归访问结点的左子树[d→left]，然后访问结点，最后再递归访问右子树[t→Right()]。

```
// 中序递归遍历树中各结点
template <class T>
void Inorder (TreeNode<T> *t, void visit(T& item))
{
    // 当子树为空时，终止遍历
    if (t ! = NULL)
    {
        Inorder(t->Left(), visit);        // 遍历左子树
        visit(t->data);                   // 访问结点
        Inorder(t->Right(),visit);        // 遍历右子树
    }
}
```

后序扫描 后序扫描将对结点的访问推迟到递归访问左子树和右子树以后。这种操作次序被称为 LRN 扫描（左、右、结点）。

1. 遍历左子树
2. 遍历右子树
3. 访问结点

后序扫描树 Tree_0 时，结点的访问次序是 D B E C A。

动作	打印	观察
从 A 到 B，转到左子树：		B 的左孩子为 NULL

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

从 B 到 D,转到右子树:		D 是叶子结点
访问 D;	D	B 的孩子访问完毕
访问 B;	B	A 的左子树访问完毕
从 A 到 C,转到右子树:		
从 C 到 E,转到左子树:		E 是叶子结点
访问 E;	E	C 的左孩子访问完毕
访问 C;	C	A 的右孩子访问完毕
访问根 A;	A	完毕!

函数自底向上对树进行扫描,先递归访问结点的左子树[t→Left()],然后访问右子树,最后才访问结点。

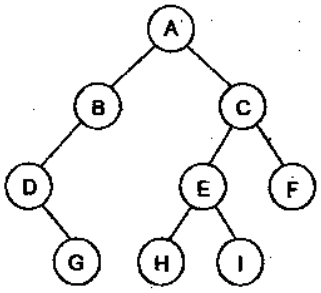
```
// 后序递归遍历树中各结点
template <class T>
void Postorder (TreeNode<T> *t, void visit(T& item))
{
    // 子树为空时,终止遍历
    if (t != NULL)
    {
        Postorder(t->Left(), visit);        // 遍历左子树
        Postorder(t->Right(), visit);        // 遍历右子树
        visit(t->data);                      // 访问结点
    }
}
```

前序扫描则规定先访问结点,然后再扫描左右分支(NLR)。

很显然,前缀词"前"、"中"、"后"分别表示对结点的访问发生在何时。但在上述 3 种情况下对左子树的访问总是先于右子树。实际上在这 3 种算法之外也有先访问右子树再访问左子树的算法。我们可以用 RNL 扫描打印一棵树。树遍历算法允许我们访问树中所有结点。它们提供与顺序扫描数组或链表等价的算法。前序、中序和后序扫描函数包含于文件"treescan.h"中。

例 11.2

1. 对于字符树 Tree_2,下表描述了我们访问结点的次序。



Tree_2

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

前序:A B D G C E H I F

中序:D G B A H E I C F

后序:G D B H Z E F C A

2. 中序扫描 Tree_2 的输出结果由以下语句得到:

```
// visit 函数输出结点数据值
void PrintChar(char& elem)
{
    cout << elem << " ";
}
TreeNode<char> *root;
MakeCharTree(root, 2);        // root 指向 Tree_2
// 输出标题,然后遍历该树,用函数 PrintChar 来访问每个结点
cout << "Inorder: ";
Inorder (root, PrintChar);
```

11.3 树扫描算法的使用

树遍历递归算法是许多树应用程序的基础。它们提供了对结点及其数据值的有序访问。这一节我们将演示用遍历算法统计叶子结点数,计算树的深度以及打印一棵树。在上述情况下,我们都必须使用扫描策略以访问每个结点。

应用、访问树结点

许多应用程序仅仅想扫描二叉树的结点而不关心遍历的次序。在这些情况下,客户程序自然选择任意一种扫描算法,在本例中,函数 CountLeaf 遍历树以统计叶子结点的数目。每识别出一个叶子结点,引用参数 LeafCount 都增 1。

```
// 本函数用后序遍历法遍历数。访问函数判断结点是否为叶子结点
template <class T>
void CountLeaf(TreeNode<T> *t, int& count)
{
    // 后序遍历
    if (t != NULL)
    {
        CountLeaf(t->Left(), count);        // 遍历左子数
        CountLeaf(t->Right(), count);        // 遍历右子数

        // 检查结点 t 是否为叶子结点(王后代)。若是,则定量 Count 加!
        if (t->Left() == NULL && t->Right() == NULL)
           count++;
    }
}
```

函数 Depth 用后序扫描计算二叉树的深度。它对每个结点都计算左右子树的深度。结点的最终深度是其子树深度的最大值加 1。

```
// 本函数用后序遍历法计算结点左、右子树的深度并返回该树的深度为 1 + max(depthLeft +
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// depthRight);空树的深度为 -1
template < class T >
void Depth(TreeNode < T >  * t)
{
    int depthLeft, depthRight, depthval;
    if (t = = NULL)
        depthval = -1;
    else
    {
      depthLeft = Depth(t - > Left());
      depthRight = Depth(t - > Right());
      depthval = 1 + (depthLeft > depthRight? depthLeft:depthRight);
    }
    return depthval;
}
```

程序 11.1　叶子计数和深度

本程序用函数 CountLeaf 和 Depth 去扫描字符树 Tree_2。leafCount 和 treeDepth 的值将被打印出来。

```
# include < iostream.h >
// 引入类 TreeNode 及库函数
# include "treelib.h"
void main(void)
{
    TreeNode < char >  * root;
    // 创建字符树 Tree_2
    MakeCharTree(root, 2);
    // 叶子结点的计数,该值由函数 CountLeaf 改变
    int leafCount = 0;
    // 调用函数 CountLeaf,计算叶子结点的点数
    CountLeaf(root, leafCount);
    cout < < "Number of leaf nodes is " < < leafCount < < endl;
    // 调用函数 Depth 并输出树的深度
    cout < < "The depth of the tree is "
         < < Depth(root) < < endl;
}
/*
 < 程序 11.1 运行结果 >
Number of leaf nodes is 4
The depth of the tree is 3
 */
```

应用：树打印

树打印函数建立逆时针旋转 90°以后树的图。图 11.10 画出了原始树 Tree_2 和打印

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

出的树。因为打印机是逐行输出信息的，所以算法用 RNL 扫描先输出右子树中的结点再输出左子树中的结点。对于 Tree_2，结点的打印次序是 F C I E H A B G D。

在函数 PrintTree 中，对结点的打印既涉及到它的数据值又涉及到它所在的层次。调用程序将根作为 0 层。每次递归调用 PrintTree 时我们都必须缩进到结点所在的层次。我们所用的格式是用 indentBlock * level 计算缩进的空格数，其中 indentBlock 是常数 6，表示每打印一个结点层次所需空出的空格数。若要打印一个结点则先缩进与其层次相对应的空格数，然后输出数据值。因为函数 PrintTree 用标准的 cout 流，操作符"<<"必须面向类型 T 定义。图 11.11 中给出了打印 Tree_2 时每个结点的层次和前导空格数。

PrintTree 的代码在文件"treeprint.h"中。

```
// 层间空格数
const int indentBlock = 6;
// 经当前行插入 num 个空格
void IndentBlanks(int num)
{
    for(int i = 0; i < num; i++)
      cout << " ";
}
// 用右子树，结点，左子树顺序遍历并输出树
template <class T>
```

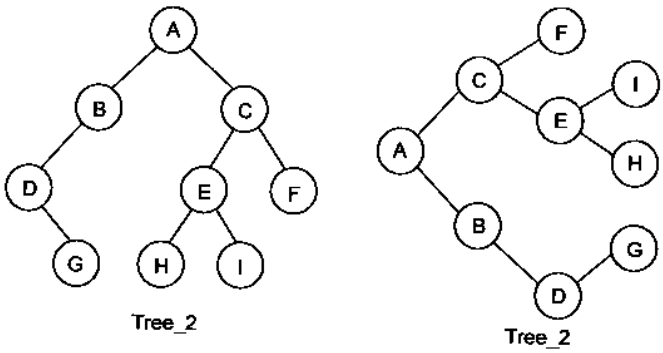


图 11.10　Printed Tree_2

	层次			
	1	2	3	
	12		F	
	6	C		
	18			I
缩进	12		E	
空格数	18			H
	0	A		
	6	B		
	18			G
	12		D	

图 11.11　输出 Tree_2

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

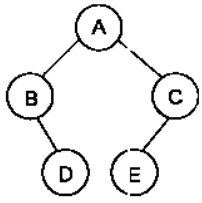
```
void PrintTree (TreeNode < T >  * t, int level)
{
    // 当 t ! = NULL 时,输出以 t 为根的树
    if (t ! = NULL)
    {
        // 输出树 t 的右半部分
        PrintTree(t - > Right(),level + 1);
        // 缩进到当前层,输出结点的数据
        IndentBlanks(indentUnit * level);
        cout < < t - > data < < endl;
        // 输出树 t 的左半部分
        PrintTree(t - > Left(),level + 1);
    }
}
```

应用：树的复制与删除

复制和删除整个树的实用函数将引入新的概念，并为我们开发一个需要析构函数和复制构造函数的树类作好准备。

函数 CopyTree 以原始树为参数建立其复制品。DeleteTree 例程则删除树中每个结点，包括根结点并释放结点所占内存。本节中，函数是针对一般二叉树编写的，它们包含于文件“treelib.h”中。

树的复制 函数 CopyTree 用后序扫描法访问树的结点。遍历次序可以确保我们能到达树的最大深度。然后进行访问操作，此操作用来产生新树的结点。CopyTree 函数自底向上建立一棵新树。函数先生成孩子，然后当生成双亲时再把它们连结到双亲。这种方法与函数 MakeCharTree 一起使用。例如，对于 Tree_0，操作次序是



Tree_0

```
d = GetTreeNode('D');
d = GetTreeNode('E');
d = GetTreeNode('B', NULL, d);
d = GetTreeNode('C', e, NULL);
d = GetTreeNode('A', b, c);
root = a;
```

先生成孩子 D，当它的双亲 B 产生后再将其连结到 B 上。类似地，先生成 E，当其双亲 C 产生时再将 E 连结到 C。最后生成根结点并将其连结到孩子 B 和 C。

树的复制算法从根开始先建立结点的左子树，然后再建立右子树。最后我们才能开始生成新结点。对树中每个结点执行的是相同的递归过程。对于原始树中的结点 t，生成左右指针分别为 newlptr 和 newrptr 的新结点。

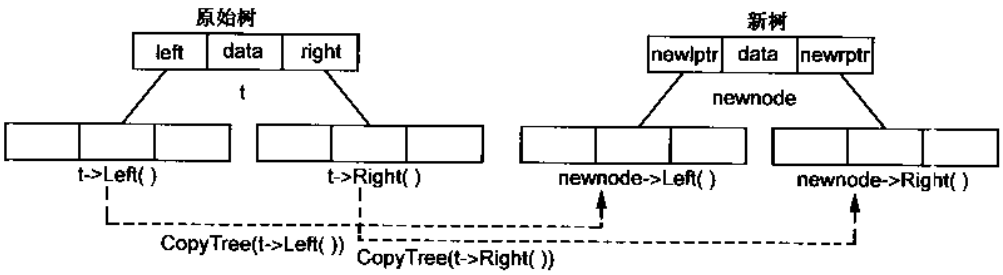
在后序扫描中，孩子先于其双亲被访问。结果，我们建立了新树中与 t→Left()和 t→Right()相应的子树。当双亲生成时孩子被连结到双亲上。

```
newlptr = = CopyTree(t - > Left());
newlptr = = CopyTree(t - > Right());
// 生成双亲结点并与其孩子连上
newnode = GetTreeNode(t - > data,newlptr, newptr);
```

生成复制树中新结点的过程也就构成了对原始树中结点 t 的访问。

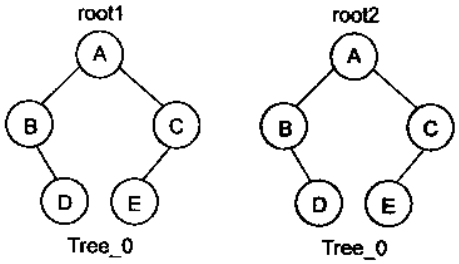
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



字符树 Tree_0 可以为我们演示递归函数 CopyTree 提供一个范例。假设主程序定义了根 root1 和 root2 并生成了树 Tree_0。函数 CopyTree 建立基于根 root2 的一棵新树。我们将对算法进行跟踪并示意建立复制树中 5 个结点的事件。

```
TreeNode < char >  * root1, * root2;        // 定义两个根
MakeCharTree(root1, 0);                     // root1 指向 Tree_0

root2 = CopyTree(root1);                    // 创建 Tree 0 的复制品
```



1. 扫描结点 A 的后代:先到结点 B 所在的左子树,然后到结点 D 所在的 B 的左子树。用数据 D 以及值为 NULL 的左右指针建立新结点。
2. 结点 B 的孩子已经被扫描。用数值 B 建立新结点,其左孩子为 NULL,右孩子为第 1 步中所得到的结点 D[图 11.12(B)]。这就完成了对结点 B 的操作。

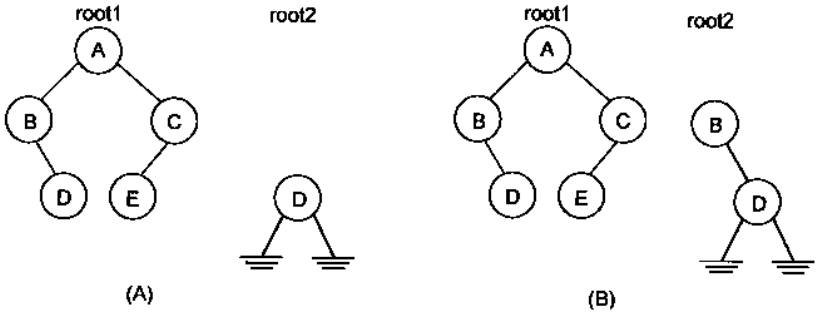


图 11.12　复制 A 的左子树

3. 一旦扫描完 A 的左子树,我们就开始扫描 A 的右子树并终止于结点 E。生成数据值为 E 的新结点。其左右指针都为 NULL。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

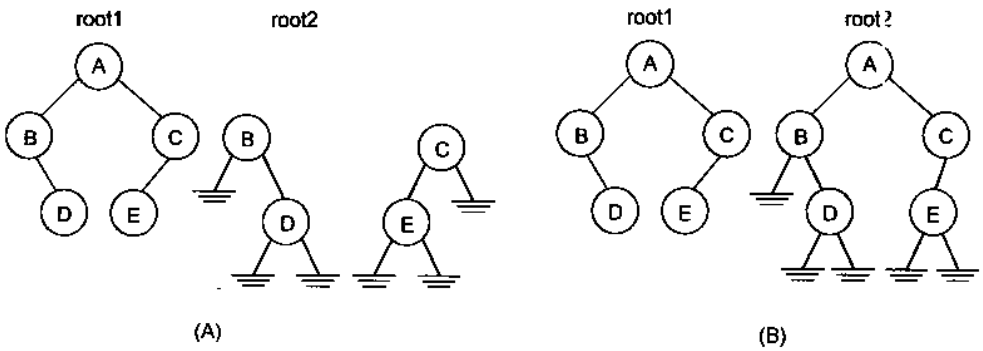


图 11.13　复制 A 的右子树

4. 处理完 E 后再转到其双亲，生成一个新结点，其数据为 C、右孩子为 NULL、左孩子为结点 E[图 11.13(A)]。

5. 最后一步在结点 A 处发生。或一个数据为 A、孩子分别为结点 B(左)和结点 C(右)[图 11.13(B)]的新结点。至此，树复制完毕。

CopyTree 函数返回的是指向新生成的结点的指针。双亲要用此返回值复制自身结点并与其孩子连结。

```
// 复制树 t 并返回新树的根
template < class T >
TreeNode < T >  * CopyTree(TreeNode < T >  * t)
{
    // 变量 newnode 指向每个由调用 GetTreeNode 产生的新结点，然后将这个结点连入
    // 新树 p. newlptr 和 newtptr 指向 newnode 的孩子结点，作为参数传给 GetTreeNode
    TreeNode < T >  * newlptr, * newrptr, * newnode;
    // 当树为空时终止递归
    if (t = = NULL)
       return NULL;

    // CopyTree 通过扫描 t 的结点建立一颗新树，对 t 中每个结点，CopyTree 先检查其左孩子，
    // 若存在，则将其复制，否则返回 NULL. 对右孩子也作类似处理. 然后，用 GetTreeNode
    // 拷贝结点，并将左右孩子的复制点连上该结点.

    if (t - > Left() ! = NULL)
       newlptr = CopyTree(t - > Left());
    else
       newlptr = NULL;

    if (t - > Right() ! = NULL)
       newrptr = CopyTree(t - > Right());
    else
       newrptr = NULL;

    // 先创建孩子结点，然后再创建双亲结点，自底向上地建立一颗新树
    newnode = GetTreeNode(t - > data, newlptr, newrptr);

    // 返回指向新创建结点的指针
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    return newnode;
}
```

树的删除 当应用程序使用诸如树这样的动态结构的时候，由程序员负责回收树所占用的内存。对于一般的二叉树，我们编写了对结点进行后序扫描的函数 DeleteTree。这种操作次序可以确保我们在删除结点（双亲）之前先访问结点的孩子。访问过程调用 FreeTreeNode 删除结点。

```
newlptr = = CopyTree(t - > Left());
newrptr = = CopyTree(t - > Right());

void DeleteTree(TreeNode < T > * t)
{
    if (t ! = NULL)
    {
        DeleteTree(t - > Left());
        DeleteTree(t - > Right());
        FreeTreeNode(t);
    }
}
```

一个更具一般性的树清除例程删除结点并重置根。函数 ClearTree 调用 DeleteTree 释放结点然后将根结点赋值为 NULL。

```
// 调用函数 DeleteTree 释放所有结点，然后将根指针置为 NULL
template < class T >
void ClearTree(TreeNode < T > * &t)
{
    DeleteTree(t);
    t = NULL;        // 此时根为 NULL
}
```

程序 11.2 测试：CopyTree 和 DeleleTree

本程序以 Tree_0 为样本生成一棵以 root2 为根的复制树。我们用后序遍历扫描新生成的树并将每个数据转化为小写。函数 PrintTree 打印树 root2 的最终数据。

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

# include "treescan.h"
# include "treelib.h"
# include "treeprnt.h"

// 用于在后序遍历中将字符数据值转化为小写字符
void LowerCase(char &ch)
{
    ch = tolower(ch);
}

void main(void)
{
```

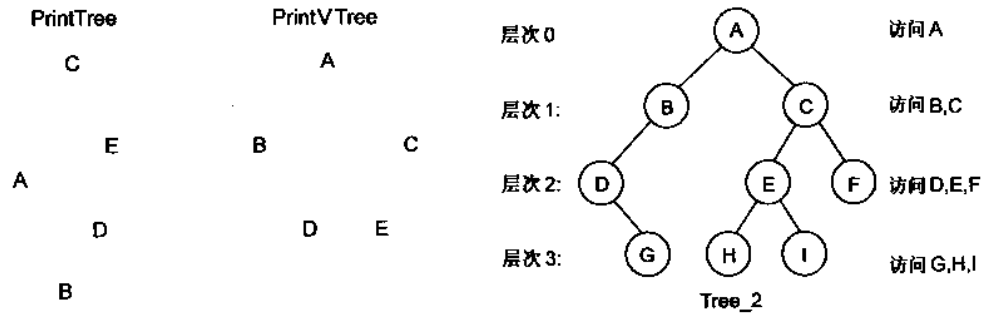

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 指向原始树及复制树的指针
    TreeNode < char >  * root1, * root2;
    // 创建 Tree_0 并输出
    MakeCharTree(root1,0);
    PrintTree(root1,0);
    // 拷贝到新树,其根为 root2
    cout < < endl < < "Copy:" < < endl;
    root2 = CopyTree(root1);
    // 后序遍历并输出树
    Postorder(root2, LowerCase);
    PrintTree(root2, 0);
}
/*
< 程序 11.2 运行结果 >

        C
                E
A
                D
        B
Copy:
        c
                e
a
                d
        b
*/
```

应用：垂直树的打印

函数 PrintTree 生成一棵树的侧视图。在每一行我们都画出位于该层次的结点。虽然画出的树难看一些，但用这种方法可以画出较大的树。对于 80 列一页面长度不限的页面，如果树的各层之间的分隔空格数是 indentBlock = 5，则一棵树可以容纳 $2^{16} - 1 = 65\ 535$ 个结点。垂直打印树则要受到更多的限制，因为我们需要足够的页宽以容纳数据和分隔各层。对于相对较小的树，图就更现实和更有吸引力些。本应用程序中，我们将开发实现 PrintVTree(垂直打印)函数的工具，介绍算法的设计。其实现在文件"treeprint.h"中。



函数 PrintVTree 需要一种从第 0 层的根开始对树结点逐层扫描的新的遍历算法。这种方法被称为广度优先扫描或层次扫描，它不再递归扫描子树，而是必须先访问同一层的所有结点

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(兄弟姐妹)然后才转到下一层。这次我们不用递归扫描算法而是开发出用队列存放数据项的循环迭代算法。对于每个结点,我们将往队列中插入任一非空的左孩子和右孩子,这就保证可以按树的下一层的次序访问兄弟姐妹们。字符树 Tree_2 演示了算法。

逐层扫描算法

初始化步骤:

将根结点插入到队列中。

递归步骤:

队列为空时过程终止。

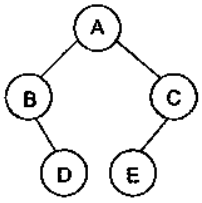
将头结点 p 从队列中删除并打印其数据值。

用该结点去标识位于树的下一层的孩子。

```
if (p->Left() != NULL)      // 检查左孩子
    Q.QInsert(p->Left());
if (p->Right() != NULL)     // 检查右孩子
    Q.QInsert(p->Right());
```

例 11.3

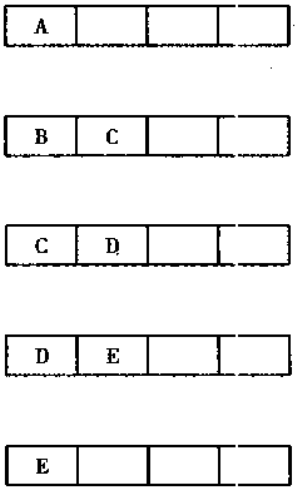
以下步骤示意了树为 Tree_0 时的层次扫描算法。



Tree_0

初始化:将结点 A 插入到队列中。

1. 从队列中删除结点 A。
   打印 A。
   将 A 的孩子插入到队列中。
   左孩子 = B
   右孩子 = C
2. 从队列中删除结点 B。
   打印 B。
   将 B 的孩子插入到队列中。
   右孩子 = D
3. 从队列中删除结点 C。
   打印 C。
   将 C 的孩子插入到队列中。
   左孩子 = E
4. 从队列中删除结点 D。
   打印 D。
   D 没有孩子。
5. 从队列中删除结点 E。
   算法终止。队列为空。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 广度优先法遍历树并访问每个结点
template <class T>
void LevelScan(TreeNode<T> *t, void visit(T& item))
{
  // 将每个结点的左、右孩子存入队列中,使得它们在访问树的下一层次时可被顺序访问
  Queue<TreeNode<T> *> Q;
  TreeNode<T> *p;
  // 通过插入根来初始化队列
  Q.QInsert(t);
  // 继续进行以下处理,直到队列为空
  while(! Q.QEmpty())
  {
      // 删除首结点并执行访问函数
      p = Q.QDelete();
      visit(p->data);
      // 若左孩子存在,将其插入队列中
      if(p->Left() ! = NULL)
          Q.QInsert(p->Left());
      // 若右孩子存在,则将其插到左孩子后边
      if(p->Right() ! = NULL)
          Q.QInsert(p->Right());
  }
}
```

PrintVTree 算法 需要传递给垂直打印函数的参数有树的根、所有数据值的最大宽度以及屏幕的宽度。

```
void PrintVTree (TreeNode<T> *t, int dataWidth, int screenWidth)
```

宽度参数使我们能对屏幕输出进行安排。为了说明这一点,假设 dataWidth 是 2,而 screenWidth 为 $64 = 2^6$。宽度为 2 的幂使得我们可以逐层对数据的组织进行描述。因为我们不能确定树的结构,所以假定空间可以容纳完全二叉树。假设结点是在坐标(level,indentedSpaces)处画出:

第 0 层:根在(0,32)处画出。

第 1 层:因为根结点缩进(偏移)了 32 个空格,所以下一层的偏移量(offset)为 $32/2 = 16 = screenWidth/2^2$。第 1 层的两个结点的位置分别为(1,32 - offset)和(1,32 + offset),即(1,16)和(1,48)。

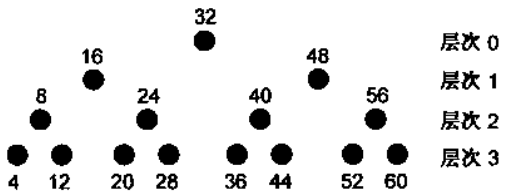
第 2 层:第 2 层的偏移量(offset)为 $S = screenWidth/2^3$。第二层的 4 个结点的位置分别为(2,16 - offset),(2,16 + offset),(2,48 - offset)和(2,48 + offset),即(2,8),(2,24),(2,42)和(2,56)。

第 i 层:第 i 层的偏移量(offset)为 $screenWidth/2^{i+1}$。第 i 层的每个结点的位置是访问第 i - 1 层其双亲结点时确定的。假设其双亲位置是(i - 1,parentPos)。若其第 i 层的结点是左孩子,那么它的位置是(i,parentPos - offset);若是右孩子则位置为(i,parentPos + offset)。

PrintVTree 用两个队列和广度优先扫描遍历树中的结点。队列 Q 中存放结点,而队列

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



QI 中则以记录类型 Info 存放结点的层次和打印位置。当结点被加入到 Q 中时，相应的打印信息也被存储到 QI 中。在结点访问期间这些项被相继删除。

```
// 存放 PrintVTree 中结点坐标(x,y)的记录
struct Info
{
    int xIndent, yLevel;
};
// 存放结点及结点打印信息的队列
Queue < TreeNode < T > * > Q;
Queue < Info > QI;
```

程序 11.3　垂直打印

程序分别在宽度为 30 个字符的页面和 60 个字符的页面上打印字符树 Tree_2。输出的 dataWidth 值为 1。

```
#include <iostream.h>
// 从树的打印库中引入 PrintVTree
#include "treelib.h"
#include "treeprnt.h"
void main(void)
{
    // 定义一个字符树
    TreeNode < char > * root;
    // 将 Tree_2 赋给 root
    MakeCharTree(root,2);
    cout << "Print tree on a 30 character screen" << endl;
    PrintVTree(root, 1, 30);
    cout << endl << endl;
    cout << "Print tree on a 60 character screen" << endl;
    PrintVTree(root, 1, 60);
}
/*
<程序 11.3 运行结果>
Print tree on a 30 character screen
              A
        B            C
      D          E        F
       G      H      I
Print tree on a 60 character screen
```

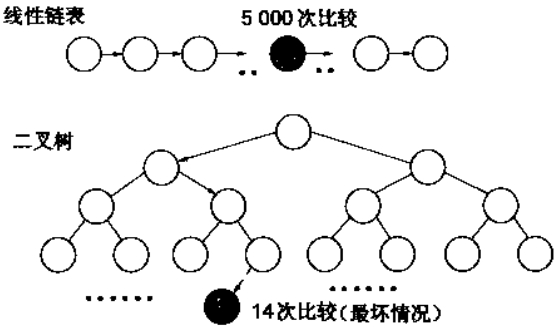

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
                    A
          B                   C
    D                 E               F
        G           H       I
*/
```

11.4 二叉搜索树

一棵普通的二叉树中可以存放大量的数据，且在我们需要增加、删除或查找数据项时能提供快速访问。建立集合类是树的最重要的应用之一。我们都熟知用 SeqList 类(顺序表类)及其数组或链表实现来构造一般集合类时所碰到的问题，SeqList 类中包含方法 Find，它用来进行顺序查找。对于线性结构，其算法复杂度为 O(N)，这对于大的集合来说缺乏效率。一般来说，树结构可以显著地改善搜索的性能，因为到达一个数据的路径最长不超过树的深度。搜索性能最优的是完全二叉树，其算法复杂度为 $O(\log_2 N)$。例如，对于一个含 10 000 个元素的表，用顺序搜索法查找一个元素的预期比较次数是 5 000，而在一棵完全二叉树上进行同样的搜索，需要进行的比较不会超过 14 次。以二叉树实现表结构可以给我们带来极大的好处。



为了将元素存储在树中以供有效访问，必须设计一种可以标识到达元素的路径的搜索结构。这种结构被称作二叉搜索树，它用关系运算符“<”对元素进行排序。为了比较树中的结点，我们可以将数据域的整体或部分指定为键值，每当树中添加一项，“<”运算符都要将其与键值比较。二叉搜索树由下列规则构成：

对每个结点，其左子树中的数据值都小于结点本身的值，而右子树中的数据值都大于或等于结点值。

图 11.14 给出了二叉搜索树的一个例子。这棵树之所以称作“搜索树”是因为我们可以沿着一条特定路径去查找一个元素(key)。从根开始，如果键值小于当前结点值则扫描左子树；否则扫描右子树。树的生成方法决定了我们可以沿从根开始的最短的路径搜索一个元素。例如，搜索 37，由根开始需要 4 次比较。

当前结点	动作
根 = 50	比较键值 37 和 50，因为 37 < 50，故扫描左子树
结点 = 30	比较 37 和 30，因为 37 > = 30，故扫描右子树

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

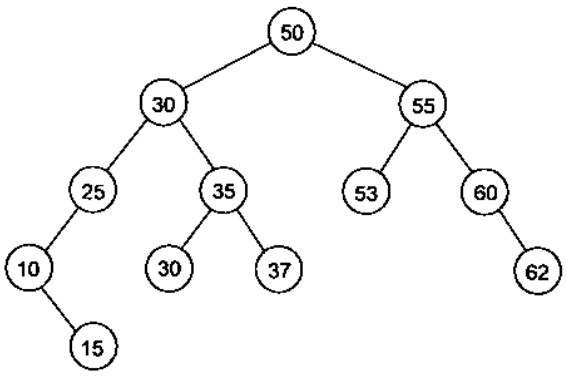


图 11.14　二叉搜索树

结点 = 35　　比较键值 37 和 35，因为 37 > = 35，故扫描右子树

结点 = 37　　比较键值 37 和 37。找到数据项。

图 11.15 中示意了几种二叉搜索树。

二叉搜索树结点中的键值

数据域中的键值起识别结点的标志作用。在许多应用场合，数据是具有若干独立域的记录。在这种情况下，键值是其中一个域值。例如，社会保险号(ssn)就是用来标识大学生的键值。

社会保险号 (9 字符串)	学生姓名 (字符串)	平均等级分 GPA (浮点数)

键域

```
struct Student
{
    String  ssn;
    String  name;
    float   gpa;
}
```

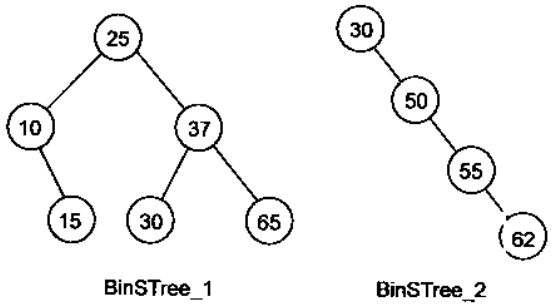


图 11.15　二叉搜索树的例子

键值可以是数据的整体或部分。图 11.15 中，树结点中所包含的是单纯的整数值，这个值就是键值。在这种情况下，结点 25 的键值是 25，要比较两点只需比较它们的整数值。整数关系运算符“<”和“= =”实施比较操作。对于大学生，键值是 ssn，我们要比较的是两个串值，这可以由重载运算符完成。例如，以下代码对两个 Student 对象进行“<”运算：

```
int operator< (const Student& s, const Student& t)
{
    return s.ssn < t.ssn; // 比较 ssn 的值
}
```

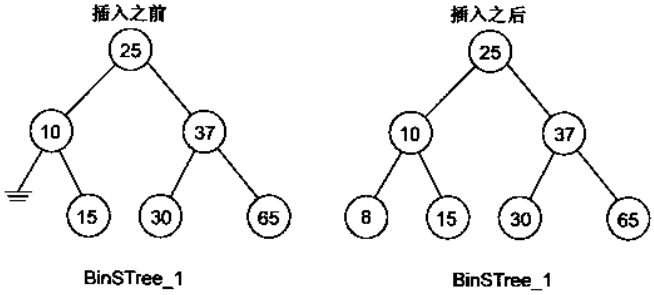

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

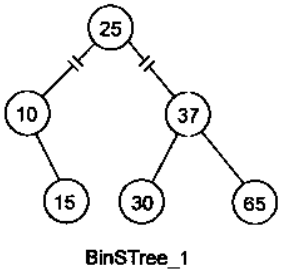
在应用过程中，键值和数据有多种对应关系。而在例子中我们只使用键值与数据值相同的简单格式。

二叉搜索树的操作

二叉搜索树是用来存放数据表的一种非线性结构。与任何表结构一样，树也必须使我们能够插入、删除和查找元素。搜索树要求插入操作可以正确地对新元素进行定位。例如，考察将结点 8 增加到 BinSTree_1 中的情况。从根结点 25 开始，8 必须在 25 的左子树上(8 < 25)。在结点 10 处，8 必须位于 10 的左子树上，而此时左子树正为空。结点 8 作为结点 10 的左孩子被加到树中。

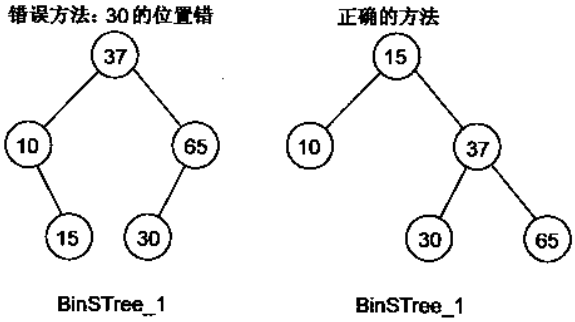


每个结点都沿特定的路径加入到树中。同样的路径也可以用来搜索元素。查找算法以键值为参数沿路径搜索每个结点的左右子树。例如，在图 11.15 所示的树 BinSTree_1 中，若要查找元素 30 则从根结点 25 进入到右子树(30 > = 25)，然后再进入左子树(30 < 37)。搜索在第 3 次比较时终止，因为这时键值等于结点值 30。



在链表操作中，删除操作将结点从链中断开并将其前驱与下一个结点连结上。在二叉树中进行类似操作要远比此复杂，这是因为删除一个结点可能会破坏树中元素的排序。考虑从 BinSTree_1 中删除根 25 的问题。首先是它产生了两个互不相连的子树，这些子树需要一个新根。

乍一看，你可能会选择 25 的一个孩子，即 37 来代替双亲。但若是这样，则结点与根的位置关系就会发生错位，因而这种简单的解决办法行不通。因为这个例子相对较小，所



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

以可以找出 15 或 30 有效地替代根结点。

二叉搜索树 ADT

表的抽象数据类型是在 SeqList 类之后建立的。二叉搜索树用非线性表存放元素在具体实现其方法时可用到。你也许会注意到这个 ADT 几乎就是 SeqList ADT 的一个翻版，只是增加了 Update 和 GetRoot 方法，前者供我们修改数据值用，而后者使我们可以访问根结点。

A 二叉搜索树 is

Data

二叉树中存储的(数据项)表以及数据值 size，它用来指示表中现有的数据项数。树中包含了指向树的根的指针以及树中访问的上一个位置的引用，我们将后者作为当前位置。

Operations

Constructor	〈与 SeqList ADT 相同〉
ListSize	〈与 SeqList ADT 相同〉
ListEmpty	〈与 SeqList ADT 相同〉
ClearList	〈与 SeqList ADT 相同〉
Find	
Input	数据值的引用。
Preconditions	无
Process	通过比较结点中的数据值对树进行搜索。 如果匹配，则从结点中取出数据值。
Output	如果发生匹配则返回 1(True)并将匹配结点的数据或值给引用参数。否则赋值 0(false)。
Postconditions	刷新当前位置，将匹配结点值作为新的当前位置值。
Insert	
Input	一个数据项
Preconditions	无
Process	用所给数据项的值对树进行搜索以便对插入点进行定位。加入新的数据项。
Output	无
Postconditions	将新结点的值作为当前位置值。
Delete	
Input	一个数据项。
Preconditions	无
Process	搜索整个树，找出数据项第 1 次出现的位置。 删除该结点并将所有子树重新连结以维持二叉搜索树的结构。
Output	无
Postconditions	将替换结点的位置作为当前位置。
Update	
Input	一个数据项
Preconditions	无
Process	如果当前位置处的键值与所给数据项的键值匹配， 则将数据项赋值给结点；否则，将数据项插入到树中。
Output	无

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Postconditions　　表中可能产生新的数据值。

GetRoot

Input　　无

Preconditions　　无

Process　　取得指向根的指针。

Output　　返回根指针。

Postconditions　　无

end ADT 二叉搜索树

定义 BinSTree 类 我们用带动态表结构的类实现二叉搜索树。因此类中应有标准的析构函数、复制构造函数以及重载的赋值运算符，后者使我们可以对对象进行初始化并执行赋值语句。析构函数负责在对象的作用域关闭后清除整个表。类似的任务由函数 ClearList 完成。析构函数、赋值运算符以及 ClearList 方法要调用私有方法 DeleteTree。还有一个私有方法，即 CopyTree，它可用于复制构造函数和重载的赋值运算符。

BinSTree 定义

声明

```
#include <iostream.h>
#include <stdlib.h>
#include "treenode.h"
template <class T>
class BinSTree
{
    protected:              // 第 12 章中继承所需
        // 指向树根及当前结点的指针
        TreeNode<T> *root;
        TreeNode<T> *current;
        // 树中数据项个数
        int size;
        // 申请/释放内存
        TreeNode<T> *GetTreeNode(const T& item,
                                 TreeNode<T> *lptr, TreeNode<T> *rptr);
        void FreeTreeNode(TreeNode<T> *p);
        // 用于复制构造函数及赋值运算符
        TreeNode<T> *CopyTree(TreeNode<T> *t);
        // 用于析构函数，赋值运算符及 ClearList 方法
        void DeleteTree(TreeNode<T> *t);
        // 在函数 Find 和 Delete 中用来定位结点及其双亲在树中的位置
        TreeNode<T> *FindNode(const T& item,
                              TreeNode<T> * & parent) const;
    public:
        // 构造函数，析构函数
        BinSTree(void);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    BinSTree(const BinSTree<T>& tree);
    ~BinSTree(void);
    // 赋值运算符
    BinSTree<T>& operator= (const BinSTree<T>& rhs);
    // 标准的表处理方法
    int Find(T& item);
    void Insert(const T& item);
    void Delete(const T& item);
    void ClearList(void);
    int ListEmpty(void) const;
    int ListSize(void) const;
    // 树的特殊方法
    void Update(const T& item);
    TreeNode<T> *GetRoot(void) const;

}
```

说明

类中含有受保护的数据成员。这就引入了第 12 章中要讨论的基于继承的结构。对类的受保护的访问在功能上与私有访问是等价的。变量 root 指向树的根结点。另一指针叫 current，它指向最近一次表更新所发生的位置。例如，current 指向 Insert 操作后新加入项所在位置；Find 方法使得 current 指向与数据项匹配的结点值。

标准的表处理操作使用的名字和参数与 SeqList 类中所定义的相同。

BinSTree 类中含有两个树所特有的操作。Update 将新数据项赋值给树中的当前位置，如果它与当前位置键值不匹配，则将新数据项添加到树中。GetRoot 方法提供对树根的访问。用户可以用 root 指针访问库文件"treelib.h"，"treescan.h"和"treeprint.h"。这就将类的功能扩展至包括许多树算法，连 PrintTree 亦包含在内。

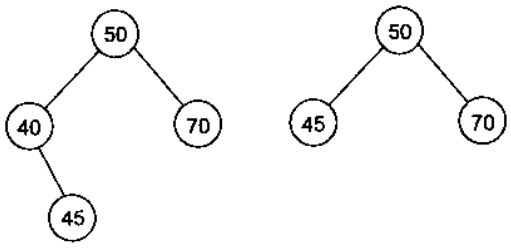
例

```
BinSTree<int> T;        // 数据类型为整型的树
T.Insert(50);           // 创建一个四结点树(A)
T.Insert(40);
T.Insert(70);
T.Insert(45);
T.Delete(40);           // 删除数据值为 40 的结点(B)
T.ClearList();          // 清除树中所有结点
// 树 univInfo 中存放以 ssn 为键值的学生信息
BinSTree<Student> univInfo;
Student stud;
// 将键值赋为"986543789"并搜索该树
stud.ssn = "986543789";
if (univInfo.Find(stud))
{
    // stud 已在树中。对该结点赋新的 GPA 值
    stud.gpa = 3.86;
    univInfo.Update(stud);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
else
    cout << "Student is not in the data base." << endl;
```



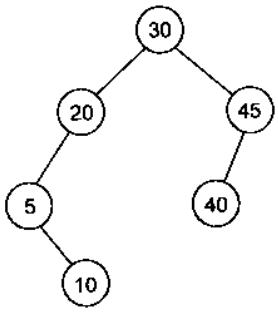
(A) (B)

11.5 二叉搜索树的使用

类 BinSTree 是用于动态表处理的一种有效数据结构。本章的实例研究是建立单词索引，这个程序示范了搜索树的典型应用。其中用到了第 14 章的带辞典的结构。这一节我们将看几个体现搜索树应用的简单程序。

定义样本搜索树 在 11.1 节中，我们用函数 MakeCharTree 生成带字符数据的二叉树。类似的函数 MakeSearchTree 用 Insert 方法建立带整型数据的二叉搜索树。例如，树 SearchTree_0 用 BinSTree 类型的对象 T 以及预定义数组 arr0 中的 6 个元素项建立一棵树。

```
int arr0[6] = {30, 20, 45, 5, 10, 40},
for (i = 0; i < 6; i++)
    T.Insert(arr0[i]);                    // 往树中插入一个元素
```



SearchTree_0

图 11.16 中 MakeSearchTree 分别建立了一棵 8 个元素以及含有 10 个其值在 10 到 99 之间的随机数的树。函数参数包括一个 BinSTree 对象以及用来标识树的参数 type。MakeSearchTree 的代码位于文件“makesrch.h”中。

中序遍历 中序遍历二叉树的算法 LNR 先访问结点的左子树，然后访问结点，最后访问右子树。如果将这种方法应用于二叉搜索树，则对结点的访问按已排好的次序进行。若是注意一下当前结点的左子树中的结点的相对值，就能很清楚地看出这一点。当前结

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

点的左子树中的所有结点的数据值都小于当前结点的值，而右子树中的所有结点的值都大于或等于当前结点的值。对二叉树的中序遍历可以确保对每个结点先访问位于左子树中的值较小的结点然后再访问位于右子树中的具有较大值的结点。最终结果是我们按从小到大的次序对结点进行了遍历。

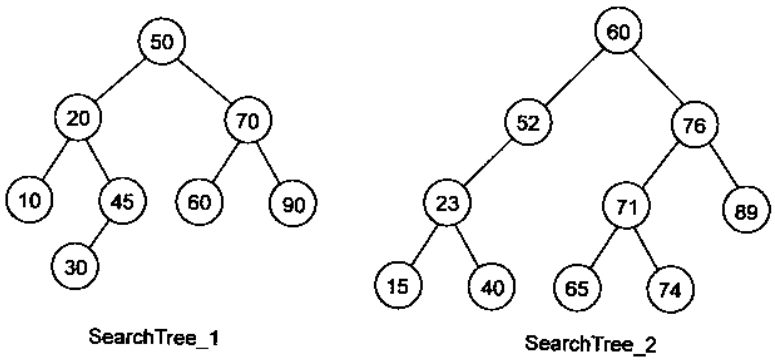


图 11.16　MakeSearchTree 树

程序 11.4　二叉树的使用

本程序用函数 MakeSearchTree 建立含以下值的二叉搜索树 SearchTree_1：

50,20,45,70,10,60,90,30

通过方法 GetRoot，我们可以获得对根的访问，进而可以调用 PrintVTree。GetRoot 还使得我们能够以函数 PrintInt 为参数调用 Inorder 按升序打印各数据项。程序最后删除数据项 50 和 70 并重新打印出树。

```
#include "makesrch.h"          // 引入函数
#include "treescan.h"
#include "treeprnt.h"          // 引入 PrintVTree
#include "bstree.h"            // Inorder 函数使用类 BinSTree
// 输出整数值。供函数 Inorder 使用
void PrintInt(int& item)
{
    cout << item << "  ";
}

void main(void)
{
    // 定义一个整型数据的树
    BinSTree<int> Tree;
    // 建立搜索树 #1,并输出,树的宽度为 40 个字符
    MakeSearchTree(Tree,1);
    PrintVTree(Tree.GetRoot(), 2, 40);
    // 中序遍历树,按升序访问数据值
    cout << endl << endl << "Sorted List:  ";
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

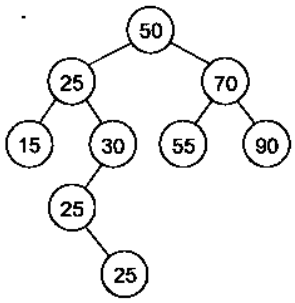
www.Linuxidc.com

```
    Inorder(Tree.GetRoot(), PrintInt);
    cout << endl;
    cout << endl << "Deleting data values 70 and 50." << endl;
    Tree.Delete(70);
    Tree.Delete(50);
    PrintVTree(Tree.GetRoot(), 2, 40);
    cout << endl;
}
/*
<程序 11.4 运行结果>
                    50
          20                    70
    10          45          60          90
              30
Sorted List: 10  20  30  45  50  60  70  90
Deleting data values 70 and 50.
                    45
          20                    60
    10          30                    90
*/
```

重复的结点

二叉搜索树中可能有重复的结点。在 Insert 操作中，当新数据项与当前结点值相匹配时我们还应继续扫描右子树。最终，重复的结点应位于匹配结点的右子树上。例如，以下树是由表 50 70 25 90 30 55 25 15 25 生成的。

表：50,70,25,90,30,55,25,15,25



许多应用中并不允许重复结点存在，而是在数据结构中设一域对数据项的出现次数进行计数。这就是记录单词所在行的行号的索引(concordance)的原理。我们处理重复出现的单词的方法不是多次将单词加入到树中，而是在行号表中放置多个项。程序 11.5 中示意了一种直接将重复计数值作为单独数据项的方法。

程序 11.5　重复次数计数

记录 IntegerCount 中包含一个整数变量 number 以及用来存储表中整数出现频率的 count。number 域作为键值，重载运算符“<”和“==”使得我们可以对两个 IntegerCount 记

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

录进行比较。Find 和 Insert 函数中将用到这两个运算符。

程序在 0～9 范围内产生 100 000 个随机数并将每个值都与一个 IntegerCount 记录相关联。Find 方法先确定一个数是否已经在树中。如果是则 count 域增 1 并更新记录；否则往树中插入一个新记录。程序最后对树进行中序扫描，将各个数值及其计数打印出来。在随机访问的情况下，0 到 9 之间的每一项出现的可能性是相同的。因此，每项应该会出现 10 000 次左右。IntegerCount 记录及其运算符在文件"intcount.h"中。

```
#include <iostream.h>
#include "random.h"          // 生成随机整数
#include "bstree.h"          // 引入类 BinSTree
#include "treescan.h"        // 引入函数 Inorder
#include "intcount.h"        // 定义 IntegerCount 记录
// 供 Inorder 调用输出一个 IntegerCount 记录
void PrintNumber(IntegerCount& N)
{
    cout << N.number << ':' << N.count << endl;
}
void main(void)
{
    // 定义一个存放 IntegerCount 记录值的树
    BinSTree<IntegerCount> Tree;
    long n;
    IntegerCount N;
    RandomNumber rnd;
    // 生成 100000 个范围为 0..9 的随机整数
    for(n=0; n < 100000L; n++)
    {
        // 对某一随机键值生成 IntegerCount 记录
        N.number = rnd.Random(10);
        // 在树中搜索键值
        if (Tree.Find(N))
        {
            // 找到键值，使 count 值加 1 并修改该结点
            N.count++;
            Tree.Update(N);
        }
        else
        {
            // 当该键值第一次出现时，将其 count 值赋值为 1 后插入树中
            N.count = 1;
            Tree.Insert(N);
        }
    }
    // 按键值域中序遍历输出记录
    Inorder(Tree.GetRoot(), PrintNumber);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
/*
<程序 11.5 运行结果>
0:10116
1:9835
2:9826
3:10028
4:10015
5:9975
6:9983
7:10112
8:10082
9:10028
*/
```

11.6 BinSTree 的实现

类 BinSTree 定义的是具有基本的插入、删除和查找数据项操作的非线性表。除了表处理方法之外，内存管理操作在类实现中也占重要位置。私有方法 CopyTree 和 DeleTree 被构造函数、析构函数和赋值运算符用来分配和回收表结点。

类 BinSTree 的数据成员 二叉搜索树是由用来启动 Insert，Find 和 Delete 操作的指针 root 所定义的，类 BinSTree 中包含数据成员 root，它是一个树结点指针，其初值为 NULL 并指向树的根结点。客户程序可以调用 GetRoot 得到 root 的值，接下来就可以调用遍历和打印函数。第 2 个指针 current 记录的是树中进行更新的位置。Find 操作令 current 指向匹配结点，而 Update 则用该指针修改表中数据。Insert 和 Delete 方法令 current 指向新结点或替代结点处。BinSTree 对象作为一个表，其大小不断被 Delete 和 Insert 方法改变。表中现有的数据项个数记录在私有数据成员 size 中。

```
// 指向树根及当前结点的指针
TreeNode<T> *root;
TreeNode<T> *current;
// 树中元素个数
int size;
```

内存管理 在 Insert、Delete 方法以及实用函数 CopyTree 和 DeleteTree 中，对结点的分配和回收是由 GetTreeNode 和 FreeTreeNode 实现的。GetTreeNode 是根据文件“treelib.h”中的函数而建立的，它用来分配内存、初始化结点的数据和指针域。FreeTreeNode 直接调用 delete 运算符释放内存。

构造函数、析构函数以及赋值运算 类中有一构造函数用来初始化数据成员。复制构造函数和重载的赋值运算符用私有方法 CopyTree 生成与当前对象相同的新二叉树。在 11.3 节中我们已经编制好了 TreeNode 类的 CopyTree 算法。这一节我们将编制从树中删除结点的算法，在 BinSTree 类中它由 DeleteTree 实现，在析构函数和 ClearList 方法中都要用到它。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

重载的赋值运算符将右边这一侧的对象复制到当前对象。在证实对象不是给自己赋值以后，函数清除当前树并用 CopyTree 建立运算符右侧(rhs)对象的副本。指针 current 被赋值给 root 指针，表的大小(size)被复制，返回值为当前对象的引用。

```
    // 赋值运算符
    template <class T>
    BinSTree<T> & BinSTree<T>::operator= (const BinSTree<T> & rhs)
    {
    // 不能将树复制到自身
    if (this == &rhs)
       return * this;
    // 清除当前树，将新树复制到当前对象
    ClearList();
    root = CopyTree(tree.root);
    // 将 current 指针指向 root 并设置树的 size 值
    current = root;
    size = tree.size;
    // 返回当前对象的指针
    return * this;
}
```

表操作

Find 和 Insert 方法从根开始在树中走过唯一的路径。根据二叉搜索树的定义，当键值或新数据项大于或等于当前结点值时，算法遍历右子树；否则算法遍历左子树。

Find 操作 Find 操作用的是私有成员函数 FindNode，它需要一个键值作参数以便在树中进行遍历查找。查找操作返回的是指向匹配点的指针以及指向双亲的指针。如果匹配发生在根结点处，则双亲指针为 NULL。

```
// 在树中搜索数据项，若找到，则返回结点地址及一个指向其双亲的指针；否则，返回 NULL
template <class T>
TreeNode<T> * BinSTree<T>::FindNode(const T& item,
                                    TreeNode<T> * & parent) const
{
    // 用指针 t 从根开始遍历树
    TreeNode<T> *t = root;
    // 根的双亲为 NULL
    parent = NULL;
    // 若子树为空，则循环结束
    while(t != NULL)
    {
        // 若找到键值，则退出
        if (item == t->data)
           break;
        else
        {
            // 修改双亲指针，并移到左子树或右子树
            parent = t;
            if (item < t->data)
               t = t->left;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            else
                t = t->right;
        }
    }
    // 返回指向结点的指针;若没找到,则返回 NULL
    return t;
}
```

有关双亲的信息是供 Delete 操作使用的。对于 Find,我们仅关心将 current 指向匹配的结点位置并将结点的数据值赋值给引用参数 item。Find 方法用返回值 True(1)或 False(0)表示搜索是否成功。Find 需要用到关系运算符"==""和"<"来比较结点中的数据。如果数据类型不在其定义范围内则运算符必须重载。

```
// 在树中搜索 item.若找到,则将结点数据赋给 item
template <class T>
int BinSTree<T>::Find(T& item)
{
    // 使用 FindNode,它需要 parent 参数
    TreeNode<T> *parent;
    // 在树中搜索 item.将匹配的结点赋给 current
    current = FindNode(item, parent);
    // 若找到,则将数据赋给 item 并返回 True
    if (current != NULL)
    {
        item = current->data;
        return 1;
    }
    else
        // 在树中没找到 item,返回 False
        return 0;
}
```

Insert 操作 Insert 方法以新数据项为参数对树进行搜索以找到合适的位置将其添加到树中。函数反复扫描左右子树中的路径直到找到插入点的位置。对于路径中的每一步,算法都记录下当前结点(称作 t)以及当前结点的双亲(称作 parent)。整个过程在我们识别出空子树(t = NULL)后终止,它表示我们已经找到插入新数据项的位置。此时将新结点作为双亲的孩子插入到该位置。例如,下面的步骤将 32 插入到图 11.17 所示的树中。

1. 整个操作从根结点开始,先比较数据项 32 和根的值 25[图 11.17(A)]。因 32 >= 25,故遍历右子树,视点转到结点 35 上。

   t 是结点 35,而 parent 是结点 25。

2. 35 是其子树的根。将 32 和 35 比较后选择遍历 35 的左子树[图 11.17(B)]。

   t 是 NULL,parent 是结点 35。

3. 用 GetTreeNode 可以生成含数据值 32 的叶子结点。新结点作为结点 35 的左孩子被插入到树中[图 117(c)]:

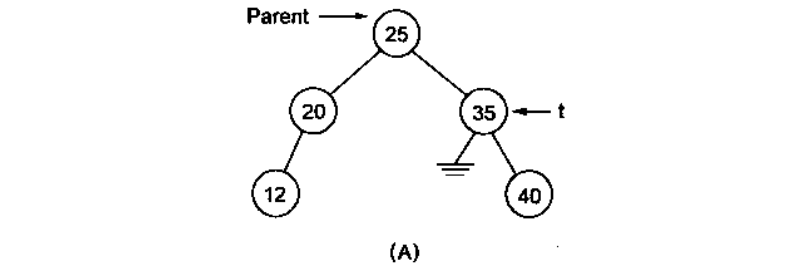
```
// 由于 BinSTree 是 TreeNode 的友类,赋值是可以的
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    newNode = GetTreeNode(item,NULL,NULL);
    parent->left = newNode;
```

指针 parent 和 t 是局部变量，在沿路径扫描以寻找插入点的过程中它们的值也随着变化。



步骤 1：比较 32 和 25，遍历右子树

步骤 2：比较 32 和 35，遍历左子树　步骤 3：将 32 作为 Parent 的左孩子插入树中

图 11.17　往二叉搜索树中插入结点

```
// 往查找树中插入数据项
template <class T>
void BinSTree<T>::Insert(const T& item)
{
    // t 为遍历过程中的当前结点，parent 为前一结点
    TreeNode<T> *t = root, *parent = NULL, *newNode;
    // 若子树为空，则退出循环
    while(t != NULL)
    {
        // 修改 parent 指针，然后往左或往右
        parent = t;
        if (item < t->data)
           t = t->left;
        else
           t = t->right;
    }
    // 创建新的叶子结点
    newNode = GetTreeNode(item, NULL, NULL);
    // 若 parent 为 NULL，则将其作为根结点插入
    if (parent == NULL)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        root = newNode;
    // 若 item < parent - > data,则将其作为左孩子插入
    else if (item < parent - > data)
        parent - > left = newNode;
    else
        // 若 item > = parent - > data 作为右孩子插入
        parent - > right = newNode;
    // current 赋值为新结点的地址并将 size 加 1
    size + + ;
}
```

Delete 操作 Delete 操作根据给定键值从树中删除结点。删除过程的第 1 步是调用实用方法 FindNode,它可以找到结点在树中的位置以及指向其双亲的指针。如果表中未找到该项则 Delete 操作什么也不做而返回。

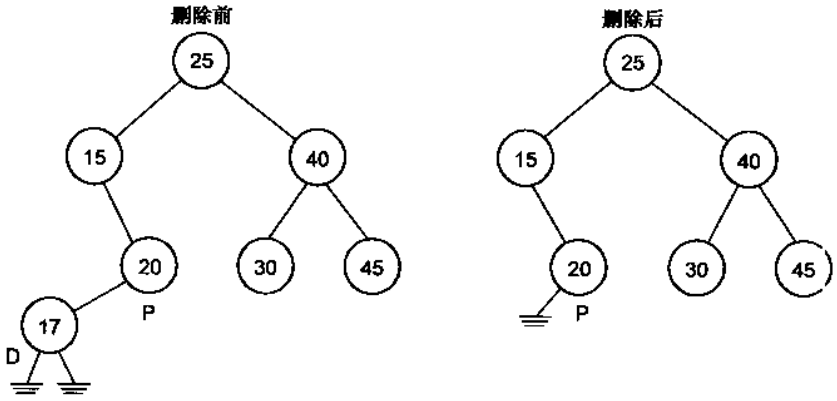
从树中删除结点的操作需要用一系列测试,以确定如何将结点的孩子重新连结到树上。重新连结子树以后必须仍能维持二叉树的结构。

调用函数 FindNode 所得到的返回值是指针 DNodePtr,它指向要删除的结点 D。另一指针 PNodePtr 指向已删除的结点双亲 P。Delete 方法要做的是寻找替换结点。R 连结双亲结点以代替被删除的结点。变量 RNodePtr 用来标识替换结点 R。

寻找替代结点的算法必须考虑 4 种情况,这 4 种情况是由连接到该结点的孩子数所决定的。注意若双亲为 NULL,则删除的是根。本例中恰好出现了这种情况,随之而来的问题是根必须更新。既然 BinSTree 是树结点类的友类,那么我们可以访问私有成员 left 和 right。

情况 A:结点 D 没有孩子。它是叶子结点。

更新双亲结点,令其子树为空。



删除叶子结点 17:
PNodePtr - > left 为 DNodePtr

不需要替代结点
PNodePtr - > left 为 NULL

令 RNodePtr 等于 NULL 就可以完成更新操作。若我们连结一个 NULL 替换结点,则双亲指向 NULL。

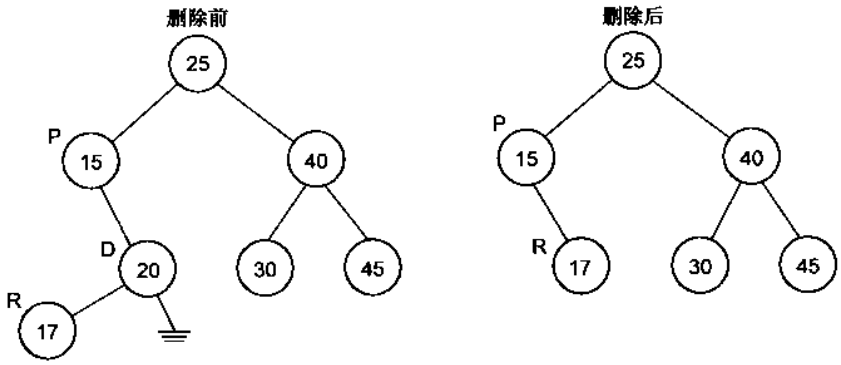
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
RNodePtr = NULL;
        ...
PNodePtr->left = RNodePtr;
```

情况 B：结点 D 有左孩子但没有右孩子。

将 D 的左子树连结到其双亲上。



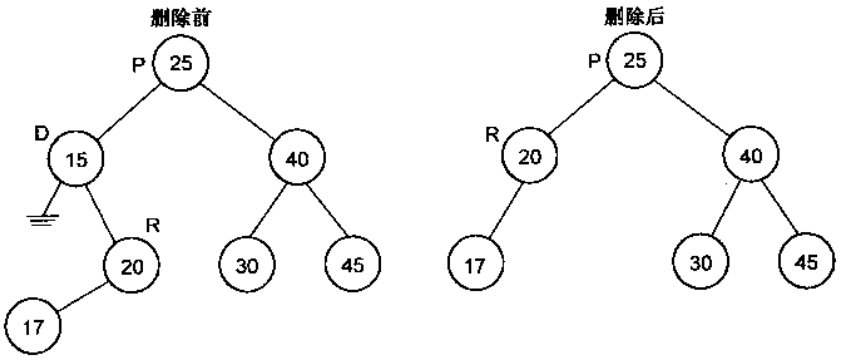
删除只有一个左孩子的结点 20：结点 R 为左孩子　　将结点 R 连到 parent 上

令 RNodePtr 等于 D 的左孩子且将结点 R 连结到双亲即可完成更新操作。

```
RNodePtr = DNodePtr->left;
        ...
PNodePtr->right = RNodeptr;
```

情况 C：结点 D 有右孩子但没有左孩子。

将 D 的右子树连结到双亲。



删除只有右孩子的结点 15：结点 R 为右孩子　　将结点 R 连到 parent

与 B 类似，令 RNodePtr 等于 D 的右孩子并将结点 R 连结到双亲。

```
RNodePtr = DNodePtr->right;
```

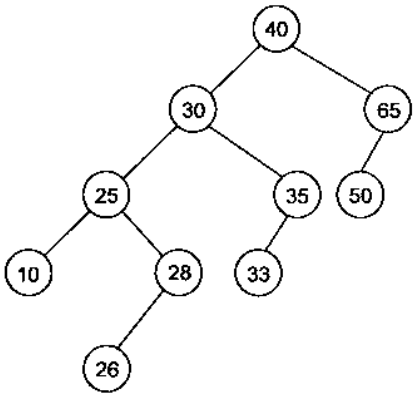

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

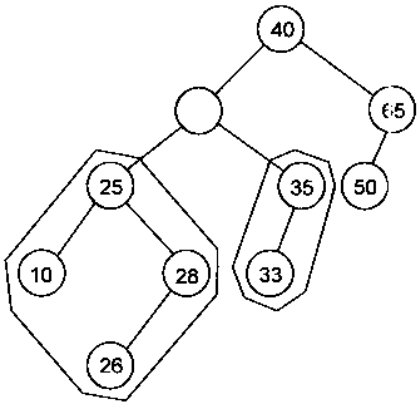
```
...
PNodePtr->left = RNodePtr;
```

情况 D：删除有两个孩子的结点。

有两个孩子的结点具有小于以及大于或等于其键值的子树。算法必须选择维持各项正确排序的替换结点。考察下面的例子。



例子树　删除 20



孤立的子树

删除结点 30 以后，我们得到了两个孤立的子树，它们必须被重新连结到树上。我们需要用某种策略从剩余的结点中选择一个替代结点。最后得到的树必须满足二叉搜索树的条件。我们使用最大—最小原则。

选择左子树中最右边的结点作为替代结点 R。这是数据值小于要删除的结点值中的具有最大值的结点。拆开结点 R 在树中的连结，将其左子树与其双亲相连，然后在被删除的结点处将结点 R 连上。在示范树中，结点 28 是替代结点。我们将其左孩子（结点 26）与其双亲（结点 25）相连。最后，用替代结点 28 替换已删除结点 30。

用一个简单的算法查找左子树中最右边的结点。

步骤 1：因为替换结点 R 小于被删除结点 D，所以转 D 的左子树。到结点 25。

步骤 2：因为 R 是左子树中具有最大值的结点，所以在（D 的）右子树中寻找其值。在扫描过程中，记录下指向各前继结点（替换结点的双亲）的指针 PofRNodePtr 的值。本例中，转到结点 28，故 PofRNodePtr 指向结点 25。

沿右子树的路径下行又分两种情况。

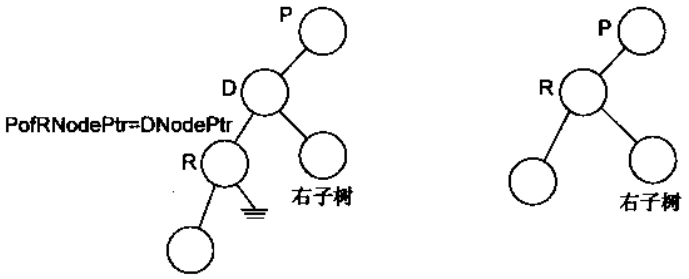
如果右子树为空，则当前位置就是替换结点 R，PofRNodePtr 指向被删除结点 D。更新过程如下：将 D 的右子树作为 R 的右子树连结到树上并将被删除的结点 P 的双亲连结到 R 上。

```
RNodePtr->right = DNodePtr->right;
PNodePtr->left = RNodePtr;
```

如果右子树非空，则扫描在遇到叶子结点或仅含左子树的结点时终止。不管是哪种情况，都将结点 R 在树中的连结拆开，将 R 的孩子重新连结到双亲结点 PofRNodePtr，且要用如下语句重置结点 PofRNodePtr 的右孩子：

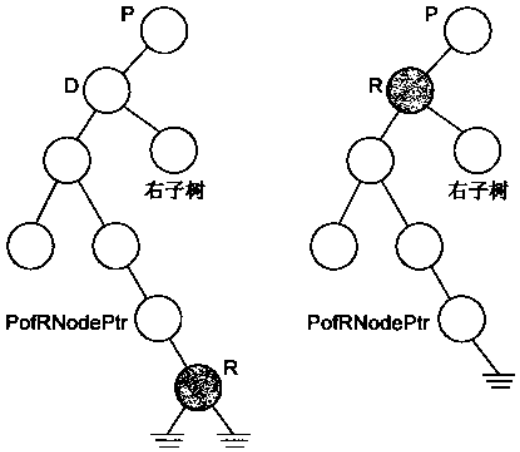
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

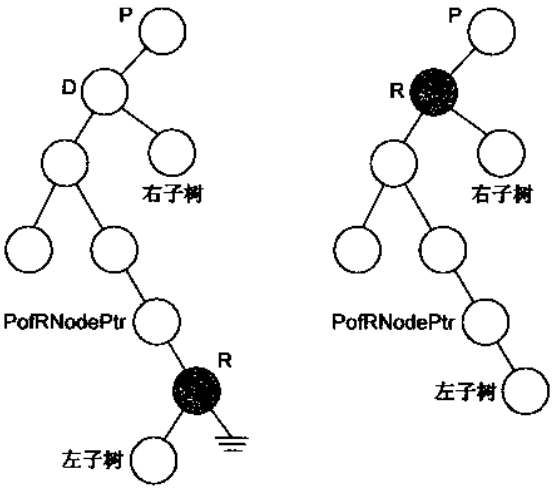



```
(**)      pofRNodePtr->right = RNodePtr->left;
```

1. R 是叶子结点。拆开它在树中的连结。因 RNodePtr→left 等于 NULL,故语句(**)将 PofRNodePtr 的右孩子置为 NULL。



2. R 有一棵左子树。语句(**)将此子树作为 PofRNodePtr 的右孩子连结到树中。



最后,算法用结点 R 替代被删除的结点。首先,将 D 的孩子作为 R 的孩子连结到树

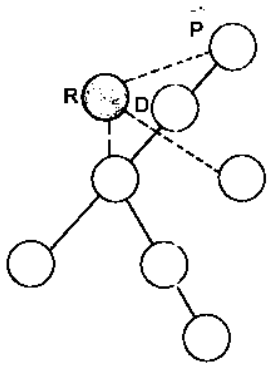
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

中。R 代替 D 作为 D 子树的根。

```
RNodePtr->left = DNodePtr->left;
RNodePtr->right = DNodePtr->right;
```

完成到双亲结点 P 的连结。




```
// 删除根结点,并赋一个新值给根
if (PNodePtr == NULL)
   root = RNodePtr;
// 将 R 连到 P 的正确的枝上
else if (DNodePtr->data < PNodePtr->data)
    PNodePtr->left = RNodePtr;
else
    PNodePtr->right = RNodePtr;
```

也可以不用以 R 代替 D 连结到树中的方法,而代之以 D 不动,将 R 中的数据复制到 D 中的方法。但如果数据对内存的消耗量较大,这种操作就不太合算。现有的方法只需更改两个指针。

Delete 方法

```
// 若 item 在树中,将其删除
template <class T>
void BinSTree<T>::Delete(const T& item)
{
    // DNodePtr = 指向被删除结点 D 的指针
    // PNodePtr = 指向结点 D 的双亲结点 P 的指针
    // RNodePtr = 指向替换 D 的结点 R 的指针
    TreeNode<T> *DNodePtr, *PNodePtr, *RNodePtr;

    // 搜索数据值为 item 的结点,并保存该结点的双亲结点的指针
    if ((DNodePtr = FindNode (item, PNodePtr)) == NULL)
       return;

    // 若 D 有一个指针为 NULL,则替换结点为其另一枝的某一结点
    if (DNodePtr->right == NULL)
       RNodePtr = DNodePtr->left;
    else if (DNodePtr->left == NULL)
       RNodePtr = DNodePtr->right;

    // DNodePtr 的两个指针均不为 NULL
    else
    {
        // 寻找并卸下 D 的替换结点。从结点 D 的左子树开始,找数据值小于 D 的数据值的
        // 最大值,将该结点从树中断开

            // PofRNodePtr = 指向替换结点对亲的指针
            TreeNode<T> *PofRNodePtr = DNodePtr;

            // 第一种可能的替换为 D 的左孩子
            RNodePtr = DNodePtr->left;

            // 从 D 的左孩子的右子树继续往下搜索最大值,并记录当前结点及其双亲结点的
            // 指针.最后,我们将找到替换结点.
            while (RNodePtr->right != NULL)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        {
            PofRNodePtr = RNodePtr;
            RNodePtr = RNodePtr->right;
        }
        if (PofRNodePtr == DNodePtr)
            // 被删除结点的左孩子为替换结点,将 D 的右子树赋给 R
            RNodePtr->right = DNodePtr->right;
        else
        {
            // 至少往右子树移动了一个结点,从树中删除替换结点,将其左子树赋给其双亲
            PofRNodePtr->right = RNodePtr->left;
            // 用替换结点代替 DNodePtr.
            RNodePtr->left = DNodePtr->left;
            RNodePtr->right = DNodePtr->right;
        }
    }
    // 完成到双亲结点的连结。删除根结点,并给新根赋值
    if (PNodePtr == NULL)
        root = RNodePtr;
    // 将 R 连到 P 的正确一枝上
    else if (DNodePtr->data < PNodePtr->data)
        PNodePtr->left = RNodePtr;
    else
        PNodePtr->right = RNodePtr;
    // 释放被删结点内存并将树的大小减 1
    FirstTreeNode(DNodePtr);
    size--;
```

树更新方法 在进行 Find 操作后,用户可能希望更新当前结点的数据域。为此,我们提供了需要带一个数据值参数的方法 Update。如果当前结点有定义(非空),Update 将当前结点值与所给数据值比较,若相等则对结点进行更新操作。如果当前结点未定义或数据项不匹配,则将新数据值插入到树中。

```
// 若当前结点已定义且数据值与给定数据值相等,则将结点值赋给 item;否则,将 item
// 插入到树中
template <class T>
void BinSTree<T>::Update(const T& item)
{
    if (current != NULL && current->data == item)
            current->data = item;
    else
            Insert(item);
}
```

11.7 实例研究:索引(Concordance)

文本分析中经常碰到的一个问题是确定单词在文档中出现的频率和位置。这种信息通常存放到索引(Concordance)中。索引将不同的单词按字母顺序排列并记录用到单词的

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

各文本行。例如,有这么一段话:

Peter Piper pick a peck of pickled peppers. A peck of pickled peppers peter Piper picked. If Peter Piper picked a peck of pickled peppers, where is the peck that Peter Piper picked?

单词"piper"在文本中出现了4次,位置分别在1、2、3行。单词"pickled"出现了3次,位置分别在1、2行。

本实例研究用以下方案建立文本文件的索引。

输入:按文本方式打开文档,逐词输入文本,跟踪记录当前行(第1行、第2行,等等)。

动作:定义一个记录,它由单词、其频率计数以及存放出现该单词的各行行号的表组成。对文本的每一行进行逐词处理。对于在文本中第1次出现的单词,生成一个记录并将它插入到树中。如果单词已经在树中,则更新频率和行号表。

输出:读完文件后,打印出按字母排序的单词表、频率计数以及出现单词的各行行号的有序表。

对于大的文档,二叉搜索树是存储单词的有效结构。最后得到的树通常比较均衡、易于更新。

数据结构

树结点中的数据是一个包含串、频率计数以及行号表的Word对象。另外,对象中还包括上次出现该单词的文本行的行号。这就确保我们可以处理单词在一行中多次出现的情况,此时我们只将行号放到表中一次。

wordText	count	LinkedList < int > LineNumbers	LastLineNo

Word类的成员函数重载关系运算符"==""<"以及标准输入/输出流运算符。

```
class Word
{
    private:
        // wordText 为单词的文本串,count 为它出现的频率
        String wordText;
        int count;
        // 行计数器可在整个 word 对象中共享
        static int lineno;
        // 单词上次出现的行号,用于是否将行号插入到 lineNumbers 中
        int lastLineNo;
        LinkedList < int > lineNumbers;
    public:
        // 构造函数
        Word(void);
        // 公有的类操作
        void CountWord(void);
```


Linux公社(www.LinuxIDC.com)是专业的Linux系统门户网站,实时发布最新Linux资讯。

www.Linuxidc.com

```
        String& Key(void);
        // 供类 BinSTree 使用的比较运算符
        int operator = = (const Word& w) const;
        int operator < (const Word& w) const;
        // Word 的输入/输出运算符
        friend istream& operator > > (istream& istr, Word& w);
        friend ostream& operator < < (ostream& ostr, Word& w);
};
```

实现 Word 类

对于每个单词,构造函数都将 count(频率)初始化为 0,将 lastLineNo(上次行号)初始化为 -1。重载的关系运算符"<"和"==”对于树类操作 Insert 和 Find 是必须的,它们是通过比较两个对象的文本串实现的。这些函数的代码在文件"word.h"中。

类中定义了静态数据成员 lineno。该变量是私有的,仅供类成员和友元函数访问。但其实质是以"Word::lineno"名义定义的外部于类的量。因此,它可供所有 Word 对象共享。这是正确的,因为所有 Word 类对象都要访问输入文件的当前行。使用静态数据成员允许带访问控制的数据共享,这要优于用全局变量。

输入运算符">>" 输入运算符从流中读取数据,一次一个单词。一个词必须以字母开头,后跟以字母或数字序列。输入一个单词时要忽略前面的非字母字符,这样可以确保跳过单词之间的空格和标点符号。输入过程在文件结尾处终止。如果到了行尾,则全局变量 lineno 增 1。

```
// 跳过单词前面的非字母字符
while (istr.get(c) && ! isalpha(c))
    // 若到行结束符,则行号计数器加 1
    if (c = = '\n')
        n.lineno + +;
```

单词的开头被识别向由">>"运算符接收字符,它读取字母或数字直到发现非字母数字字符为止。单词中的字符被转换为小写并存储在 C++局部串变量 wd 中。这使得我们所作的索引不区分大小写。如果在单词后遇到行尾符,则将它回放到流中,这样从文档中读取下一个单词时会再碰到它。函数最后将 wd 赋值给 wordText,将 count 设为 0,lastLineNo 置为值 lineno。

```
// 若非文件结束,则读入单词
if (! istr.eof())
{
    // 将单词的第一个字符换成小写字符,并赋给 wd
    c = tolower(c);
    wd[i + +] = c;
    // 继续读入后续的字母或数字字符,并转换成小写
    while (istr.get(c) && (isalpha(c) || isdigit(c)))
        wd[i + +] = tolower(c);
    // 给 wd 加上串结束符 null
    wd[i] = '\0';
    // 若当前单词后有新行符,留给下一个单词处理
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    if (c = = '\n')
        istr.putback(c);
    // 将 wd 赋给 wordText, count 置为 0 且 lastLineNo 置为 lineno
    w.wordText = wd;
    w.count = 0;
    w.lastLineNo = w.lineno;
}
```

函数 CountWord 从文本中读取一个单词后，我们调用函数 CountWord 对 count 值和行号表进行更新。count 首先增 1。如果 count 值为 1 则该单词是加入到树中的数据项，其第 1 次出现之处的行号被加入到表中。如果单词已经在树中，则要检查自上次碰到该单词以来行号是否已发生变化。如果已经改变，则当前的行号被加入到表中且用该值更新 lastLineNo。

```
// 记录单词出现的频率
void Word::CountWord(void)
{
    // 将单词出现频率 count 加 1
    count + +;
    // 若该单词第一次出现或在新行中第一次出现，则将行号加入到行号表中，
    // 并将 lastLineNo 值改为当前行
    if (count = = 1 || lastLineNo ! = lineno)
    {
        lineNumbers.InsertRear(lineno);
        lastLineNo = lineno;
    }
}
```

输出运算符"< <" 流输出运算符打印单词和计数值，后面跟以单词出现之处的行号的有序表。

$\langle text \rangle \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots < count > : l_1\ l_2 \cdots\ l_n$

具体做法是打印文本后以右对齐方式打印 count 值，填充字符为"."。行号则通过遍历链表打印出来。

```
// 输出 Word 对象
ostream& operator < < (ostream& ostr, Word& w)
{

    // 输出单词
    ostr < < w.wordText;
    // 以右对齐方式输出 count 值，填充符为'.'
    ostr.fill('.');
    ostr < < setw(25 - w.wordText.Length()) < < w.count < < ":";
    ostr.fill(' ');          // 将填充符重置为空格
    // 遍历链表输出行号
    for(w.lineNumbers.Reset(); ! w.lineNumbers.EndOfList();
                    w.lineNumbers.Next())
        ostr < < w.lineNumbers.Data() < < "  ";
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    ostr << endl;
    return ostr;
}
```

程序 11.6　文本索引

本程序定义了存储 Word 对象的二叉搜索树 concordTree。打开文本文件“concord.txt”以后，流输入运算符不断读取单词，直到文件尾为止。每个单词要么被插入到树中，要么用来更新信息(如果该单词以前出现过)。处理完所有单词后，执行一遍中序遍历，按字母顺序打印出单词。Word 类包含于文“word.h”中。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "word.h"                    // 引入类 Word
#include "bstree.h"                  // 引入类 BinSTree
#include "treescan.h"                // 用于中序遍历
// 用于函数 Inorder
void PrintWord(Word& w)
{
    cout << w;
}
void main(void)
{
    // 定义 Word 对象构成的树及输入流 fin
    BinSTree<Word> concordTree;
    ifstream fin;
    Word w;
    // 打开文件"concord.txt"
    fin.open("concord.txt", ios::in | ios::nocreate);
    if (! fin)
    {
        cerr << "Cannot open 'concord.txt'" << endl;
        exit(1);
    }
    // 从 fin 中读入 Word 对象，直到文件结束
    while(fin >> w)
    {
        // 在树中搜索 w
        if (concordTree.Find(w) == 0)
        {
            // w 不在树中。修改该单词的计数器并往树中插入该单词
            w.CountWord();
            concordTree.Insert(w);
        }
        else
        {
            // w 在树中，修改该单词计数器并修改其在树中的信息
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            w.CountWord();
            concordTree.Update(w);
        }
    }
    // 按字母序输出该树
    Inorder(concordTree.GetRoot(), PrintWord);
}
/*
<输入文件"concord.txt">
Peter Piper picked a peck of pickled peppers. A peck of pickled
peppers Peter Piper picked. If Peter Piper picked a peck of
pickled peppers, where is the peck that Peter Piper picked?
<程序 11.6 运行结果>
a................3: 1  2
if...............1: 2
is...............1: 3
of...............3: 1  2
peck.............4: 1  2  3
peppers..........3: 1  2  3
peter............4: 1  2  3
picked...........4: 1  2  3
pickled..........3: 1  3
piper............4: 1  2  3
that.............1: 3
the..............1: 3
where............1: 3
*/
```

书面作业

11.1 试解释为什么树是一种非线性结构。

11.2 包含下列数目的结点的二叉树的最小深度值是多少？

(a) 15

(b) 5

(c) 91

(d) 800

11.3 (a) 画出一棵含 10 个结点、深度为 5 的二叉树。

(b) 画出一棵含 14 个结点、深度为 5 的二叉树。

11.4 一棵二叉树中含数据值 1 3 7 2 12。

(a) 画出两棵包含以上数据的深度最大的树。

(b) 画出双亲值比任一孩子值大的两棵完全二叉树。

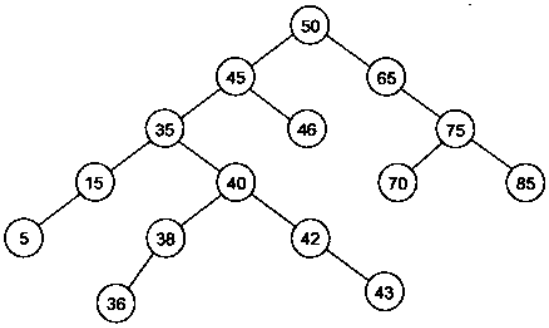
11.5 画出含 3 个结点的所有可能的二叉树。

11.6 具有 n 个结点的二叉树必有 n－1 条边(非空指针)吗？

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

11.7 考察以下二叉树：



(a) 如果值 30 被插入到树中，则其双亲是哪个结点？

(b) 如果值 41 被插入到树中，则其双亲是哪个结点？

(c) 用前序、中序及后序扫描遍历树。

11.8 描述函数 F 中发生的动作。假设 F 是类 BinSTree 的成员函数。

```
template <class T>
void F(TreeNode<T> * & t, T item)
{
    if (t == NULL)
       t = GetTreeNode(item);
    else if (item < t->data)
       F(t->left, item);
    else
       F(t->right, item);
}
```

为什么 t 作为引用参数传递是很关键的？

11.9 画出以下各个字符序列所对应的二叉搜索树，并分别用中序、前序和后序扫描遍历各棵树。

(a) M,T,V,F,U,N

(b) F,L,O,R,I,D,A

(c) R,O,T,A,R,Y,C,L,U,B

11.10 分别用 RLN,RNL,NRL 和层次扫描法遍历书面作业 11.9 中的各棵树。

11.11 画出以下各个整数序列所对应的二叉搜索树并分别用中序、前序和后序扫描遍历各棵树。

(a) 30,20,10,6,5,35,56,1,32,40,48

(b) 60,25,70,99,15,3,10,30,38,59,62,34

(c) 30,20,25,22,24,23

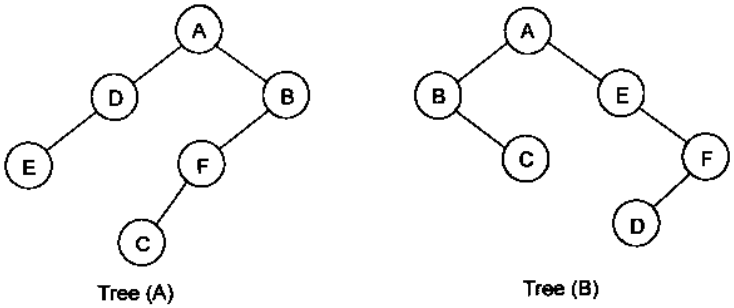
(c)所对应的树对于搜索来说是不是一种好的结构？说明你的理由。

11.12 分别用 RLN,RNL 和 NRL 扫描遍历书面作业 11.11 中的各棵树。

11.13 修改 MakeCharTree，使其可以生成以下树并将它们列为第 3 和第 4 种情况。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

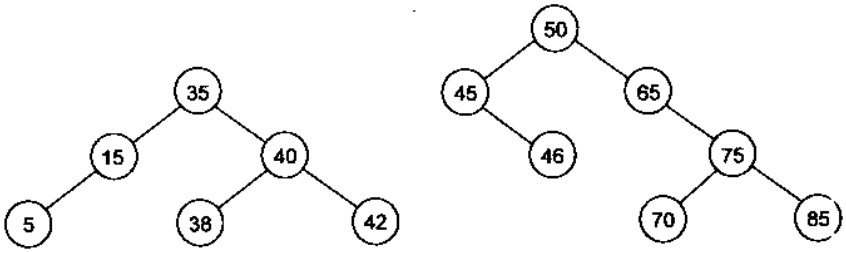
www.Linuxidc.com



11.14 (a) 如果修改逐层算法，将结点放到堆栈中而不是队列中，则遍历次序是什么？以 11.1 节中所提供的 Tree_2 为例进行说明。

(b) 假设结点被插入到一个优先级队列中，其次序由数据域决定。用此算法给出 Tree_2 的遍历次序。

11.15 参照 MakeCharTree 定义函数 MakeIntTree 以建立以下二叉树。显式生成所有结点。



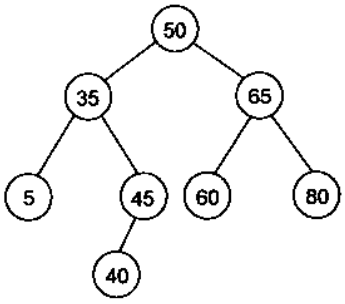
11.16 (a) 以下整数序列是通过前序遍历二叉搜索树获得的。构造一棵具有这种序列的树。

50 45 35 15 40 46 65 75 70

(b) 构造一棵二叉搜索树，对其进行中序遍历可得到如下元素序列：

40 45 46 50 65 70 75

11.17 以下二叉搜索树用于(a)到(e)各步。每一步都使用原始树。



(a) 插入值 1,48,75,100 后画出树。

(b) 删除结点 5,35。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(c) 删除 45。

(d) 删除 50。

(e) 删除 65,插入 65。

11.18 修改函数 CopyTree,使其成为带有参数"target"的新函数 TCopyTree。在复制过程中,函数仅复制那些数据值大于 target 的结点。如果小于的话,则将结点复制为不含左右子树的叶子结点。注意在复制过程中生成叶子结点时必须删除左右子树中的所有结点。

```
TreeNode<T> *TCopyTree (TreeNode<T> *t, T target);
```

11.19 编写函数

```
TreeNode<T> *ReverseCopy (TreeNode<T> *tree);
```

将所有左右指针进行交换再对树进行复制。

11.20 编写函数

```
void PostOrder_Right(TreeNode<T> *t, void visit(T& item));
```

用 RLN 扫描遍历树。

11.21 编写函数

```
void *InsertOne(BinSTree<T>&t, T, item);
```

若 item 不在二叉搜索树 t 中,则将 item 插入到 t 中,否则函数不作插入而返回。

11.22 编写函数

```
TreeNode *Max(TreeNode *t);
```

返回指向二叉搜索树的最大结点的指针。用循环迭代法。

11.23 编写函数

```
TreeNode<T> *Min(TreeNode<T> *t);
```

返回指向二叉搜索树的最小结点的指针。用递归法。

11.24 用 1 到 9 的整数建立一棵不含重复数据值的 9 结点二叉树。

(a) 若树的深度为 4,给出根结点的可能值。

(b) 在深度为 5,6,7,8 的情况下再解答(a)题。

11.25 对以下各字母表,画出按所给顺序插入字母时所生成的二叉搜索树:

(a) D,A,E,F,B,K

(b) G,J,L,M,P,A

(c) D,H,P,Q,Z,L,M

(d) S,J,K,L,X,F,E,Z

11.26 编写循环迭代函数。

```
template <class T>
int NodeLevel(const BinSTree<T>& T, const T& elem);
```

确定 elem 在树中的层次,若不在树中则返回 -1。

11.27 (a) 假设树结点中的数据值为串,两个串按 ASCII 顺序进行比较。按所给次序插

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

入如下 C++ 关键字，生成一个二叉搜索树。

for, case, while, class, protected, virtual, public
private, do, template, const, if, int

(b) 用前序、后序和中序扫描遍历树。

11.28 (a) 二叉树结点有时被修改为包含指向其双亲结点的指针。将类 TreeNode 修改为 PTreeNode，使其包含这一指针。

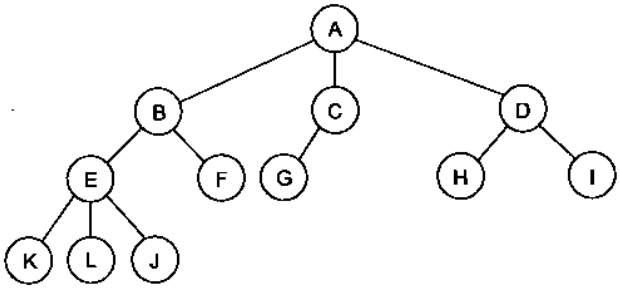
(b) 编写函数

```
template <class T>
void PrintAncestors(PTreeNode<T> *t);
```

打印从结点 t 到根结点的结点链中的数据。

(c) 使用 MakeCharTree 中的技术，用 PtreeNode 对象建立树 Tree_2。

11.29 有些问题，例如在计算机上实现游戏，会涉及到一般树。在一般树中，结点可以有两个以上的孩子。例如，以下是孩子数最多为 3 的一棵树(三叉树)。



(a) 在一般树中，中序遍历是否具有明确定义？

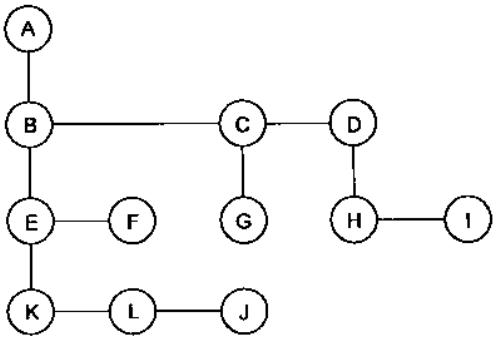
(b) 对三叉树进行前序和后序遍历。

(c) 一棵一般树可以用以下算法转换成二叉树：

(1) 二叉树中每个结点的左指针都指向一般树中相应结点的最左孩子。

(2) 二叉树中每个结点的右指针都指向一般树中结点的一个兄弟(具有同一双亲的结点)。

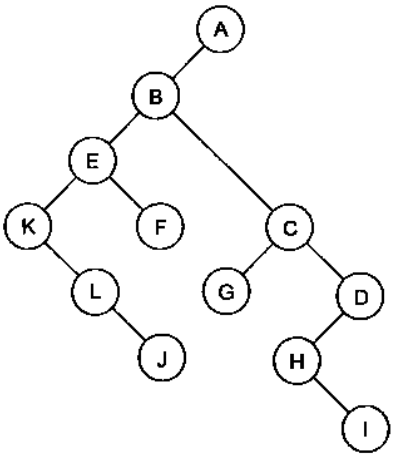
画二叉树时，将孩子直接置于结点下而将兄弟放到右侧。按结点列安排树。例如，下图是与样本树相对应的二叉树：



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

若将树顺时针旋转 45°，就会得到一棵更为我们所熟悉的二叉树：



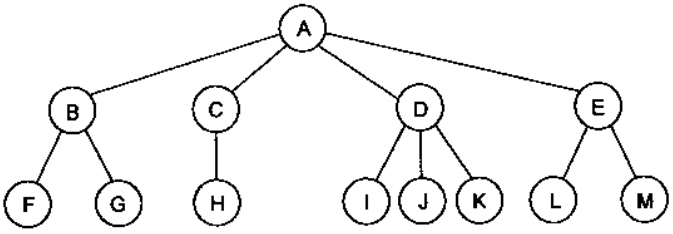
用前序、中序和后序扫描遍历此二叉树。这些扫描和对一般树的扫描有何相似之处？

(d) 对下图中的一般树进行如下操作：

(1) 用前序和后序扫描遍历树。

(2) 画出相应的二叉树。

(3) 用前序、中序和后序扫描遍历二叉树。



11.30 因为含 n 个结点的二叉树中有 n + 1 个 NULL 指针，所以分配给指针的存储空间有一半被浪费了。有一个巧妙的算法可以利用这类被浪费的空间。如果左指针为 NULL，就使其指向中序遍历时结点的前驱结点。若右指针为 NULL，则使其指向中序遍历时结点的后继。这种结构被称作“线索树”，这些指针被称为“线索”。对类 TreeNode 进行简单的扩展即可表示线索树的结点。具体做法是增加私有逻辑型变量 leftThread 和 rigntThread 以及方法 LeftThread 和 RightThread，前者表示相应的指针是否线索，后二者则返回前二者的值。假设类名为 ThreadTreeNode。编写一个循环迭代函数。

```
template < class T >
void ThreadedInorder(ThreadedTree < T >  * t);
```

用中序扫描遍历 t 并打印结点值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

上机题

11.1 编写函数

```
int CountEdges(TreeNode<T> *tree);
```

计算二叉树中边(非空指针)的条数。用"treelib.h"中的 Tree_1 对函数进行测试。

11.2 编写函数

```
void RNL(TreeNode<T> *tree, void visit(T& item));
```

用 RNL 遍历法访问树。读取 10 个整数并利用 BinSTree 类将它们放到二叉搜索树中。用 RNL 遍历树。遍历所产生的数据序列是什么?

11.3 利用书面作业 11.15(a)和 11.15(b)中所编写的函数写一个打印两棵树的主程序。分别用 PrintTree 和 PrintVTree 打印树。

11.4 用书面作业 11.21 中的函数 InsertOne,在一个测试程序中建立一棵含 8 个结点的树。输入数据中必须有重复。用 PrintTree 示意最终得到的树。

11.5 在一个主程序中,用 BinSTree 类建立一棵含 500 个值在 1 到 10 000 之间的随机数的树。分别用书面作业 11.22 和 11.23 中的函数 Max 和 Min 求最大和最小值。

11.6 修改索引(concordance)问题(程序 11.6),在输出中用如下格式表示每个单词在各行的出现次数。

```
行号(#出现次数)
```

例如

```
<Input>   one two one two three
<output> one.................................2: 1(2)
         three...............................1: 1(1)
         two.................................2: 1(2)
```

11.7 (a) 编写函数

```
void LinkedSort(Array<int>& A);
```

这样对 A 排序:将其元素插入到一个有序链表中再将排好序的数据值拷回到 A 中。

(b) 编写函数

```
void TreeSort(Array<int>& A);
```

这样对 A 排序:将其元素插入到一个二叉搜索树中,中序遍历树,将排好序的数据值拷回到 A 中。(提示:写一个递归函数实现中序遍历和对数组元素赋值。)

```
void InorderAssign(TreeNode<int> *t,Array<int>& A, int i);
```

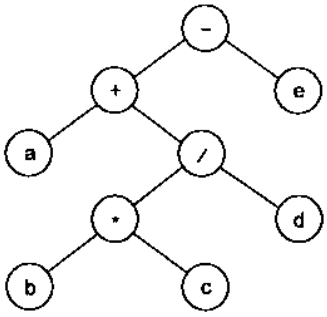
(c) 编写一个主程序,建立一个含 10 000 个随机整数的数组并用系统定时器确定(a)和(b)中各个算法所费时间,各个函数必须对同样的数据排序。

11.8 使用二元运算符加(+)、减(-)、乘(*)和除(/)的算术表达式可以用"表达式二叉

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

树"表示。在表达式二叉树中，每个运算符都有两个孩子，它们要么是操作数，要么是子表达式。叶子结点中所含的是操作数，非叶子结点中所含的是一个二元运算符。运算符的左右子树所表示的是要对其求值并作为操作数之一而使用的子表达式。例如，表达式 a+b*c/d 与以下表达式二叉树相对应：



(a) 对表达式二叉树进行前序、中序和后序遍历。这些扫描与表达式的前缀、中缀和后缀表示式(RPN)之间有什么关系？

(b) 对以下各算术表达式，分别建立相应的表达式树。通过扫描树的方法给出表达式的前缀、中缀和后缀式。

(1) a+b=c*d+e　　(2) /a-b*c d

(3) a b c d/-*　　(4) *-/+a b c d e

(c) 可以编写一个以前缀式读取表达式并建立表达式的递归程序。

- 如果当前项为操作数，则用其数据值生成一个叶子结点，其左右指针均为 NULL。
- 如果当前项为运算符，则将其作为结点的数据值并生成其左右孩子。

编写函数

```
void BuildExpTree(TreeNode<char> *t, char * & exp);
```

根据以 NULL 终结的串 exp 中所含的前缀表达式建立一棵表达式树。假设操作数都是 a 到 z 之间的单字母标识符，运算符选自'+'、'-'、'*'、'/'。

(d) 编写一个主程序，读取一个表达式，然后建立表达式树。用 PrintVTree 垂直打印出树并打印出表达式的中缀和后缀式。

11.9 用书面作业 11.18 中的函数 TCopyTree。在一个测试程序中，建立一棵含 10 个结点的二叉搜索树，结点中的数据为整型。输入一个 target 值并用 TCopyTree 复制结点。用 PrintVTree 打印原始和复制树。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 12 章　继承和抽象类

12.1　继承概述
12.2　C++中的继承
12.3　多态性和虚函数
12.4　抽象基类
12.5　迭代算子
12.6　有序表
12.7　异构表
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

继承是面向对象的程序设计中的一个基本概念。本章将深入研究第1章中曾简单介绍过的继承的关键特性。我们以基类 Shape 及其派生出的一族几何图形类为例重点介绍继承的概念及其在 C++ 中的实现。

多态性和虚函数在12.3节作简单论述，它们被应用于显示几何对象的属性问题。

12.4节中引入抽象基类的概念。除了提供功能部件外，抽象基类在其派生类中强制实现其纯虚函数。这一节将设计一个一般线性表或非线性表所对应的抽象类。

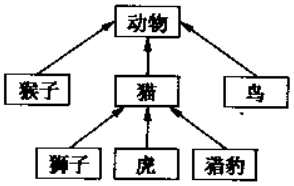
迭代算子(iterator)是用来遍历诸如数组、链表或树一类的数据结构的对象。因此，它是一种"控制抽象"。12.5节中设计迭代算子的过程是定义抽象基类再用它派生面向 SeqList 和 Array 类的迭代算子。Array(数组)迭代算子可用于归并有序的归并段(run)，第14章中还将用到这一技术。

12.6节中运用继承从第9章中所设计的 SeqList 类的链表版中派生出一个有序表类。该类可以用作生成有序归并段的过滤器，而归并段可用于对外部文件进行归并排序。

12.7节是选读内容，它介绍如何用继承和多态性来设计含不同类型对象的数组和链表。后续各章中，在有益且合适的情况下，还将用继承来设计数据结构。

12.1 继承概述

从动物学的观点来看，继承所讨论的是各种动物之间的共性与个性。例如，令动物类代表包括猴子、猫、鸟等在内的动物。虽然它们都有作为动物的共性，但还是有一些具有特殊个性的不同的动物科。每一种动物类型又进一步划分到种属。例如猫科又可分为狮子、虎、猎豹等等。



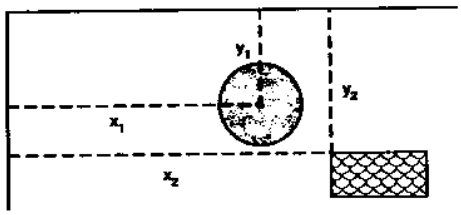
现存所有动物都可以用一棵等级树来抽述，从整个动物王国到各种属都包括在树中。"猫是动物"，"虎是猫科"，等等。较深层次所表现出的属性是由较浅层次项派生出来的，相隔多层的关系仍然是有效的，如"虎是动物"。

上下继承关系在程序设计中也存在。建议你复习一下1.4节，其中设计了 Point，Line 和 Rectangle 对象并用继承将它们联系起来，那些类中包括 Draw 方法，它从基点出发在屏幕上画图。这一节我们得为选定的几何图形，如圆、矩形等等设计与上述相似的类并将所有类都共用的方法分离出来。

几何对象都具有一些共同的属性。它们都是可以在屏幕上画出的图形并且每个图都有一个固定其位置的基点。例如，我们在中心点周围画圆、用左上角位置固定矩形。此外，每个图都以一定的填充模式画出，该填充模式由一个整数值所标识。在许多图形库中，0值代表在图形内的空填充模式。例如，下图中以(x_1, y_1)为圆心，以实心填充模式(solid fill pattern)画出一个圆。而矩形则是从角(x_2, y_2)开始，以砖块填充方式(brick fill pattern)画出的。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



我们可以将各种几何类的共同属性基点和填充模式分离出来，定义包含这些成员的 Shape 类。类中包括返回基点坐标的方法，重新定位基点的方法以及取得或改变填充模式的方法。依赖于系统的 Draw 方法初始化图形系统。绘图操作对不同的几何形状采用特定的填充模式。Draw 是一个“虚函数”，也就是说它在虚类中一定要被重新定义。12.3 节中将论述虚函数。

在第 1 章中，我们设计了具有度量面积和周长操作的 Circle 类。这些操作可以应用于任何封闭图形，因此我们将它们包括到 Shape 类中。但这些方法不在 Shape 类中定义，而是作为其在派生类中定义的模板(template)。它们被称作“纯虚函数”，Shape 则被称为“抽象类”。虽然许多概念将在 12.3 节和 12.4 节作介绍，但我们这里还是给出 Shape 类描述的一个轮廓。类描述中所包含的不同元素预示着本章的各个论题。

```
class Shape
{
    protected:
        float x, y:      // 基点的水平位置及垂直位置
        int fillpat;
    public:
        // 缺省的构造函数
        Shape(float h=0, float v=0, int fill=0);
          . . .
        // 供派生类中 Draw 方法调用的虚函数,初始化填充方式
        virtual void Draw(void)const;
        // 派生类必须定义 Area 和 Circumference
        virtual float Area(void) const = 0;
        virtual float Perimeter(void) const = 0;
};
```

我们将 Shape 类用于继承的层次系统中，在任何情况下，派生类都要使用 Shape 类中的方法建立自己的特定方法以改写抽象类中的一般方法。例如，Circle(圆)对象是带半径的 Shape(基点在圆心)。它包含在作图介质表面(以一定的填充模式)画圆的 Draw 方法。类中还有特定的 Area 和 Perimeter 方法，它们要用到 radius(半径)和常数 π。在继承链中，Circle 是由 Shape 派生得到的。

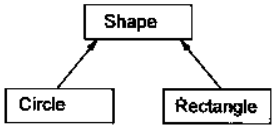
类似地，Rectangle(矩形)对象是具有两个边长(长和宽)的 Shape(基点在其左上角)。Draw 方法便用 length(长)和 width(宽)画出一个矩形框并按一定的填充模式填充内部。公式

area = length * width
Perimeter = 2 * (length + width)

是 Area 和 Perimeter 方法的基础。在继承链中，Rectangle 是由 Shape 派生得到的。

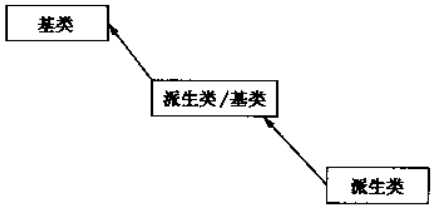
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



类继承术语

在 C++中,继承是面向类而定义的。这就暗示我们,"派生类"从"基类"继承数据和操作。派生类本身又可以是另一继承层次的基类。使用继承的类系统形成了一个"类层次系统(class hierarchy)"。



派生类常被称作"子类(subclass)",而相应的基类则被称作"超类(superclass)"。

12.2 C++中的继承

启动继承链的基类用普通类声明方法进行声明,而派生类的声明则有些变化,它要引用其与基类的关系。

```
基类
    // 定义普通的 C++ 类
    class BaseCL
    {
        <data and methods>
    }
派生类
    // 定义派生类,它包含有对基类的引用
    class DerivedCL: public BaseCL
    {
        <data and methods>
    }
```

BaseCL 是 Derived CL 所继承的基类名。关键字"Public"说明使用的是"公有继承(public inheriance)"。

C++中允许将派生类声明为公有继承,私有继承或保护继承。软件设计中多使用公有继承。保护继承很少使用。私有继承将在习题中讨论。

例 12.1

1. Shape 类是派生类 Circle 的基类

```
class Shape
{<members>}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
class Circle: public Shape   // Circle 继承类 Shape
{ < members > }
```

2. 对于动物(Animal)、猫科(Cat)和虎(Tiger)继承链,相应的类声明是

```
class Animal
{ < members > }
    class Cat: public Animal
{ < members > }
    class Tiger: public Cat
{ < members > }
```

对于公有继承,基类中的私有成员保持私有性质,仅供基类的成员函数访问。公有成员则可供派生类的所有成员函数以及派生类的任何客户程序访问。除了私有和公有成员以外,C++中还定义了在基类中具有特殊意义的"保护(protected)成员"。当基类被继承后,其保护成员可以被派生类中所有方法访问,但不能被类的任何客户程序访问。

```
class BaseCL
{
    private:
        { < Members > }          // 只允许 BaseCL 成员访问
    protected:
        { < Members > }          // 允许 DerivedCL 和 BaseCL 的成员访问
    public:
        { < Members > }          // 允许所有客户程序访问
}
```

如果有一个具有多次派生关系的层次链,则每个派生类都保留对链中高层基类的保护和公有成员的访问。图 12.1 中示意了分别含 1 个和 2 个派生类的类层次系统。左边的箭头指明派生类成员访问不同属性的类成员。每个图右边示意客户程序只能访问基类和派生类的公有成员。

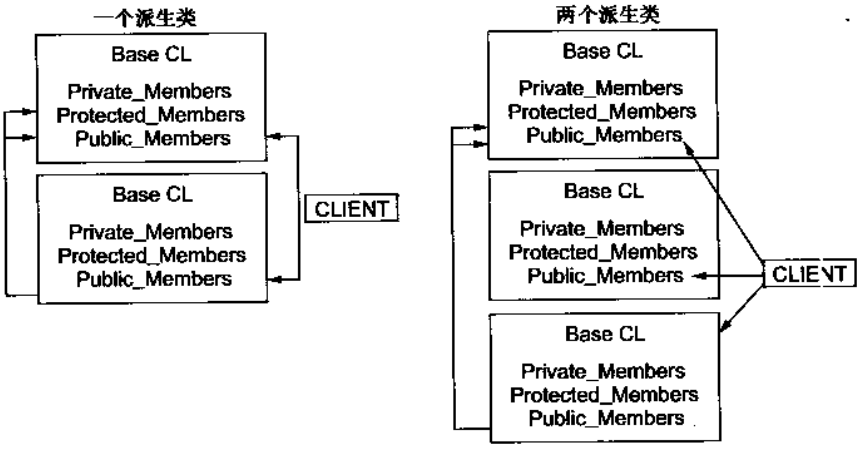


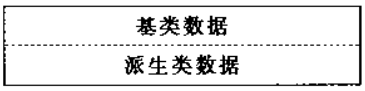
图 12.1 公有继承中对基类的访问

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

构造函数和派生类

在继承链中，派生对象从基类中继承数据和方法，我们称基类是派生类的“子类型(subtype)”。派生对象的资源扩充至包括基对象的资源。



派生对象

生成派生对象时要调用其构造函数以对其数据进行初始化。同时，该对象还从由基类构造函数所初始化的基类继承数据。因为在定义对象时要调用构造函数，所以在基类和派生类的构造函数之间必有些交互动作。当定义一个派生对象时，先执行基类的构造函数，然后再执行派生类的构造函数。从直观上看，必须先建立继承链的头部，因为派生类经常要用到基类数据。如果链中有两个以上的类，初始化过程先初始化基类，再沿派生类的链向前推进：

```
DerivedCL    obj      // 供基类 BaseCL 构造函数调用
                      // 然后是派生类 DerivedCL 的构造函数
```

若基类构造函数需要参数，则派生类构造函数必须显式调用此基类构造函数并传递所需参数。其做法是将基类构造函数的名字和参数放到派生类构造函数的参数初始化表中。如果基类具有默认的构造函数且设定了默认值则派生类无需显式执行默认的构造函数。当然，最好是进行显式调用。

例 12.2

设有基类构造函数说明如下：

```
BaseCL(int n, char ch);          // 构造函数有两个参数
```

一般地，派生类构造函数的参数表是基类构造函数参数表的超集(superset)。

```
// 派生类构造函数参数表是基类构造函数参数表的超集，它至少得有基类构造函数要求
// 的两个参数
DerivedCL(int n, char ch, int sz);
```

派生类构造函数必须在初始化表中显式调用基类构造函数。以下是派生类构造函数的实现示例：

```
// 在初始化表中调用基类构造函数 BaseCL(n,ch)，表达式 data(sz)为将值 sz 赋给
// 数据成员的标准赋值操作
DerivedCL::DerivedCL(int n, char ch, int sz):
            BaseCL(n,ch), data(sz)
{}
```

在一个继承链中，析构函数被调用的顺序正好与构造函数相反，首先调用派生类的析构函数，然后是成员对象的析构函数，最后调用基类的析构函数，这正好与它们的出现次序相反。从直观上看，派生对象是在基对象之后生成的，所以应在基对象之前被撤销。若派生类没有析构函数但基类有，则自动生成派生类的析构函数。此析构函数撤销派生类的成员并执行基类析构函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

继承中命名冲突的解决 在继承链中的类可以包含相同名字的成员。派生类中的成员与基类中相同(名字)成员的作用域是不一样的。派生类的定义中隐含了基类的定义但并没重载它。必须用类作用域运算符"::"引用基类中具有同一名字的方法。例如,考察下面的链:

```
class BaseCL
{
    public:
        . . .
        void F(void);
        void G(int x);
        . . .
};
```

```
class DerivedCL: public BaseCL
{
    public:
        . . .
        void F(void);
        void G(int x);
        . . .
};
```

假定在派生类中函数 G 的一个具体实现要调用基类中的函数 G,则派生类必须前置作用域运算符 BaseCL::以获准访问基类的方法 G。

```
void DerivedCL::G(float x)
{
    . . .
    BaseCL::G(x);        // 在成员函数中用域运算符
    . . .
};
```

对于含有派生对象的客户程序,对 F 的调用是由 DerivedCL 类中的方法 F 处理的,对基类中 F 的调用是通过前置作用域运算符实现的:

```
derived OBJ;
```

客户程序调用:

```
OBJ.F();              // 指派生类中的函数 F
OBJ.base::F();        // 指基类中的函数 F
```

应用:继承 Shape 类 在 12.1 节中我们引入了 Shape 类中作为抽象基类的一个例子。其数据和方法可以用于几何 Circle 和 Rectangle 类。Shape 类可以作为描述几何图形的派生类的基类。为说明继承技术细节,我们先将 Shape 类声明为基类,然后再派生 Circle 类。Rectangle 类的派生在专门论述虚函数的 12.3 节中给出。

类 Shape 的说明

声明

```
// 定义点,填充方式及访问和改变这些参数的基类。该类由其它几何图形类继承,实现了各自
// 的画图函数、求面积及周长的函数
class Shape
{
    protected:
        // 基点在屏幕中的水平及垂直位置,用于派生类的各个方法中
        float x, y;
        // 为画图函数提供的填充方式
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        int fillpat;

    public:
        // 构造函数
        Shape(float h=0, float v=0, int fill=0);
        // 访问基点的 x 和 y 坐标的方法
        float GetX(void) const;                    // 返回 x 坐标值
        float GetY(void) const;                    // 返回 y 坐标值
        void SetPoint(float h, float v);           // 改变基点
            // 访问填充方式的方法
        int GetFill(void) const;                   // 返回填充方式
        void SetFill(int fill);                    // 改变填充方式
            // 纯虚函数,派生类必须定义各自的求面积和周长的方法
        virtual float Area(void) const = 0;
        virtual float Perimeter(void) const = 0;
        // 供派生类的 Draw 在初始化填充方式时调用的纯虚函数
        virtual void Draw(void) const;
};
```

说明

构造函数带默认值,它们定义了位于窗口左上角的基点(0,0)。默认的填充模式 0 通常表示不填充。

GetX 和 GetY 返回基点的 x 和 y 坐标值,而 SetPoint 则允许客户程序改变基点。与此类似,Getfill 和 SetFill 方法提供对填充模式的访问。

Draw 方法初始化作图系统,使得图形以 fillpat 填充模式画出。我们假定客户程序负责打开图形窗口并关闭作图界面。方法 Area 和 Perimeter 是纯虚函数,它们在 Shape 类中声明并起类似于模板的作用,其定义由各个继承了 Shape 类的派生类给出。

Shape 类的实现

文件"geometry.h"中给出了 Shape 类的完整声明。这一节我们只详细描述一下构造函数和 Draw 函数。

构造函数需要用到基点的坐标和填充模式。客户程序可以用 SetPoint 和 SetFill 改变它们的值。

```
// 构造函数,初始化坐标值及填充方式
Shape::Shape(float h, float v, int fill):
      x(h), y(v), fillpat(fill)
```

Draw 调用初级图形系统函数 SetFillStyle。当派生类画出实际的图形后,就要用到这种填充风格。

```
void Shape::Draw(void) const
{
    SetFillstyle(fillpat); // 调用图形系统函数
}
```

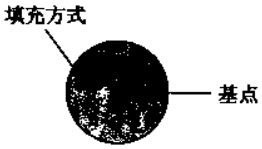

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

派生类 Circle

第 1 章中我们定义了 Circle 类，类中包括半径(radius)值以及计算面积(Area)和周长(Perimeter)的方法。创建对象时 radius 被传递给构造函数，但 radius 不能被(客户程序)访问，这一节中我们将类扩展到包括作图功能和访问半径的方法。Draw 方法从 Shape 类中继承基点和填充方式。

声明




```
// 用于计算面积和周长的常数
const float PI = 3.14159;
// 在基类 Shape 基础上定义类 Circle
class Circle: public Shape
{
    protected:
        // 若 Circle 类成为基类，其派生类可访问 radius
        float radius;
    public:
        // 带有圆心，半径及填充方式参数的构造函数
        Circle(float h=0, float v=0, float r=0, int fill = 0);
        // 访问半径的方法
        float GetRadius(void) const;
        void SetRadius(float r);
        // 画圆的方法，调用了类 Shape 的 Draw
        virtual void Draw(void) const;
        // 计量方法
        virtual float Area(void) const;
        virtual float Perimeter(void) const;
};
```

说明

Draw 以基点(x,y)为圆心、r 为半径画出一个圆。

类的声明及其实现在文件“geometry.h”中给出。

Circle 类的实现

实现 Circle 类时假定存在一个包含了低级作图操作的文件“graphlib.h”。Circle 的构造函数要用一些参数来初始化基类 Shape 以及它自身的数据成员 radius。

```
// 构造函数，参数 h 和 v 初始化类 Shape 中的基点。点(h,v)表示圆心。参数 fill 初始化类
// Shape 的填充方式。只有 r 是专门为类 Circle 用到的参数。类 Shape 中的基对象由初始
// 化表中的构造函数 Shape(h,v,fill)初始化
Circle::Circle(float h, float v, float r, int fill):
{}
```

Draw 操作 从基类中调用 Draw 方法设置填充方式。既然基类中的数据提供受保护的访问，Circle 中的 Draw 方法就可以访问基类中的数据。但是对象的客户程序是不能直

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

接访问基点坐标的。

```
// 以圆心(x,y)及给定半径画圆
void Circle::Draw(void) const
{
    Shape::Draw(); // 设置填充方式
    DrawCircle(x,y, radius)
}
```

程序 12.1　画圆

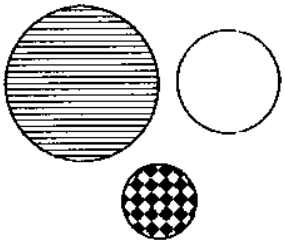
本程序示范 Circle 和 Shape 类的使用。在声明了两个 Circle 对象后，执行一系列 Shape 类和 Circle 类中的方法。

```
#include <iostream.h>
#include "graphlib.h"
#include "geometry.h"
void main(void)
{
    // 定义对象 C,D 填充方式分别为 7 及不填充
    Circle C(1.0, 1.0, 0.5, 7), D(2.0, 1.0, 0.33);
    char eol;            // 用于画图前等待用户输入
    cout << "Coordinates of C are " << C.GetX() << " and "
         << C.GetY() << endl;

    cout << "Circumference of C is " << C.Perimeter() << endl;
    cout << "Area of C is " << C.Area() << endl;

    cout << "Type <return> to view figures: ";
    cin.get(eol);       // 等待用户输入回车
    // 初始化图形界面,该函数与系统有关。
    InitGraphics();

    // 以填充方式 7 及半径 0.5 画圆 C
    C.Draw();
    // 以缺省的填充方式 0 及半径 0.33 画圆 D
    D.Draw();

    // 生成圆 D,圆心为 11.5,1.8,半径为 0.25.填充方式为 11
    D.SetPoint(1.5, 1.8);
    D.SetRadius(.25);
    D.SetFill(11);
    D.Draw();

    // 等待用户输入后关闭图形系统
    ViewPause();
    ShutdownGraphics();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
/*
<程序 12.1 运行结果>
    Coordinates of C are 1 and 1
Circumference of C is 3.14159
Area of C is 0.785398
Type <return> to view figures:
*/
```



什么不能被继承

虽然派生类可以继承基类对保护数据成员的访问，但基类的某些成员和属性是不能被继承的。构造函数不被继承，因此不能被声明为虚方法。如果基类构造函数需要参数，则派生类必须有自己的构造函数以调用基类构造函数。友元关系不被继承。若函数 F 是类 A 的友元，B 派生自 A，那么 F 不会被自动看作是 B 的友元。

12.3 多态性和虚函数

1.3 节中曾经从非技术性的观点出发对多态性作了论述，现在将它重新读一遍会对我们有所帮助。这一节我们将对多态性的表达作一扩充并举几个例子。

面向对象的程序设计提供了一种叫作“多态性”的属性。这个词是从古希腊语中派生出来的，其含义是“多种形态”。在程序设计中，多态性的含义是指同一种方法可以针对不同类类型的对象而定义。方法的特定动作因类而异。C++ 中用“动态联编(dynamic binding)”和“虚成员函数”的技术支持多态性。动态联编允许系统中的不同对象对同一消息作出特定于其类型的响应。消息的接受者是在运行时动态确定的。

要使用多态性，必须在基类中声明虚成员函数，这只需在函数声明前加上关键字“virtual”即可。例如，在类 BaseCL 中，F 和 G 被声明为虚函数：

```
class BaseCL
{
    private:
      ...
    public:
      ...
    virtual void F(int n);
    virtual int G(long m);
      ...
};
```

声明派生类时，必须包括具有严格相同的参数表和返回类型的成员函数 F 和 G。在派生类中，关键字 virtual 不是必须的，因为虚函数属性可从基类中继承得到。虽然如此，但最好还是在派生类中使用 virtual，这样可以省却用户查看基类定义以确定其是否虚函数的麻烦。

```
class BaseCL
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    private:
        ...
    public:
        ...
        virtual void F(int n);
        virtual int G(long m);
        ...
};
```

一旦定义了继承链和虚函数，我们就可以讨论新的访问类成员的条件了。假设 DObj 是一个 DerivedCL 对象：

```
DerivedCL DObj;
```

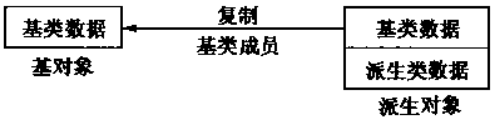
派生类中的成员函数 F 是用对象名进行访问的。对基类中的函数 F 的访问要用到对象名和基类作用域运算符。

```
DObj.F(n);              // 派生类成员函数
DObj.BasedCL::F(n);     // 基类成员函数
```

这些调用都是属于静态联编(static binding)的范围。编译器能知道调用的是基类的还是派生类的 F 特定版本。但当用到指针或引用时多态性就起作用了。考察如下声明

```
BaseCL *P, *Q
BaseCL BObj;
DerivedCL DObj;
```

既然派生类是基类的一个子类型，派生对象就可以赋值给基类对象。在赋值过程中，在基类对象中也有的那一部分派生对象数据被复制。



另一方面，从基对象到派生对象的赋值是非法的，因为派生类中的有些数据项可能未定义。例如，有赋值语句如下：

```
Bobj = DObj;      // 拷贝基类数据到 Bobj;
Dobj = BObj;      // 非法拷贝。派生类数据未被初始化
```

在指针及地址操作中，基类指针可指向派生类对象，所以下列赋值语句是合法的：

```
Q = &BObj;     // 将基类对象的地址赋给基类指针
P = &DObj;     // 将派生类对象地址赋给基类指针
```

以下语句

```
Q->F(n);   // 调用基类中的 F 函数
```

调用基类中的函数 F。与之相类似的语句 P 说明了多态性的本质特征，因为尽管是 BaseCL 类的指针，它还是调用 DerivedCL 类的方法 F。

```
P->F(n);   // 调用派生类中的 F 函数
```

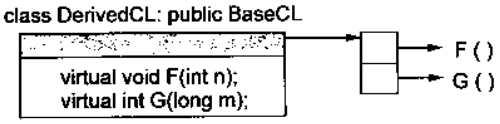
当通过指针或引用访问时，C++根据指针或引用所实际指向的对象决定调用哪个

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

版本的函数。这一过程被称作“动态联编(dynamic binding)”。

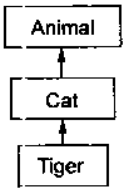
一个对象若至少包含一个虚函数,则它必有一个指向运行时“虚函数表(virtual function table)”的指针。该表中包含类中声明的所有虚函数的起始地址。当通过指针或引用调用虚函数时,运行系统使用对象的地址访问指向虚函数表的指针,按指针找到表,从中查找函数地址,然后再调用函数。



在上述例子中,P 指向一个 DerivedCL 类型的对象,所以类 DerivedCL 中的 F 版本被调用。这样我们就可以建立各种对象,它们都由基类指针进行引用。当虚函数被执行时,与实际对象类型相对应的函数版本被调用。多态性允许那些带基类指针或引用变元的函数在派生类的虚函数新版本中被复用。

展示多态性

本章开始时我们曾举了一个有关动物分类的层次结构的简单的继承例子。从基类 Animal(动物)开始,细化到 Cat(猫科)类,然后再分为各种特定类型的猫科,如 Tiger(虎),Tiger 属于 Cat,而 Cat 属于 Animal。



下面设计类 Animal,Cat 和 Tiger 以模拟动物的层次结构。每一个类中都包含一个由构造函数进行初始化用来提供有关对象的特定信息的串,另外还有一个打印信息的 Identify 方法。

Animal 类声明

```
class Animal
{
    private:
        char animalName[20];
    public:
        Animal(char nma[])
        {
            strcpy(animalName, nma);
        }
        virtual void Identify(void)
        {
            cout << "I am a " << animalName << " animal" << endl;
        }
};
```

Cat 类声明

```
class Cat: public Animal
{
    private:
        char catName[20];
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    public:
        Cat(char nmc[], char nma[]): Animal(nma)
        {
            strcpy(catName, nmc);
        }
        virtual void Identify(void)
        {
            Animal::Identify();
            cout << "I am a " << catName << " cat" << endl;
        }
};
```

Tiger 类声明

```
class Tiger: public Cat
{
    private:
        char tigerName[20];
    public:
        Tiger(char nmt[], char nmc[], char nma[]): Cat(nmc, nma)
        {
            strcpy(tigerName, nmt);
        }
        virtual void Identify(void)
        {
            Cat::Identify();
            cout << "I am a " << tigerName << " tiger" << endl;
        }
};
```

程序 12.2　动物的多态性

本程序用两个函数 Announce1 和 Announce2 说明静态和动态联编，这两个函数调用作为参数传递给对象的 Identify 方法，并使用了两种不同的参数传递技术。

Announce1-用值参传递 Animal 对象

```
void Announce1(Animal a)
{
    // 静态联编实例;编译器决定哪个 Animal 对象的 Identify 方法被执行
    cout << "In static Announce1, calling Identify:" << endl;
    a.Identify();
    cout << endl;
}
```

Announce2-传递一个指向 Animal 对象的指针

```
// Announce2-传递指向 Animal 对象的指针
void Announce2(Animal *pa)
{
    // 采用动态联编.pa 指向的对象的 Identify 方法被调用
    cout << "In dynamic Announce2, calling Identify:" << endl;
    pa->Identify();
    cout << endl;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

主程序中声明了一个 Animal 对象 A、一个 Cat 对象 C 以及一个 Tiger 对象 T。我们用"Announce"函数说明不同的参数传递方式所产生的不同效果。静态联编的演示是通过将 Tiger 对象 T 传递给 Announce1 而实现的。用分别指向对象 A,C 和 T 的指针对 Announce2 进行 3 次独立调用,从而强调说明多态性。作为多态性的另一个例子,用被初始化为指向一个 Cat 对象的 Animal 指针执行方法 Identify。结果 Cat 中的 Identify 方法被调用。最后一段代码展示了派生对象到基对象的赋值。派生对象中的基类数据部分被复制到左边。

Animal 类以及 Announce 函数在文件"animal.h"中给出。

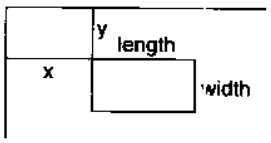
```
#include <iostream.h>
#include <string.h>
#include "animal.h"
void main(void)
{
    Animal A("reptile"), *p;
    Cat C("domestic", "warm blooded");
    Tiger T("bengal", "wild", "meat eating");

    // 静态联编。Announce11 中为值参,由于 T 为 Tiger,所以函数调用 Animal
    // 的 Identify.
    Announce1(T);                    // 静态联编;调用 Animal 的方法
    // 多态调用的例子。由于参数为指针,Announce2 用动态联编执行实际对象的 Identify
    // 方法
    Announce2(&A);                   // 静态联编;调 Animal 方法
    Announce2(&C);                   // 静态联编;调 Cat 方法
    Announce2(&T);                   // 静态联编;调 Tiger 方法

    // 直接调用类 Animal 的 Identify 方法
    A.Identify();                    // 静态联编
    cout << endl;

    // 动态联编;调用 Cat 的方法
    p = &C;
    p->Identify();
    cout << endl;

    // 将 Tiger 对象赋值给 Animal 对象仅复制从 Animal 中继承的数据
    A = T;
    A.Identify();                    // 输出从 Tiger T 中拷贝来的 animal 数据
    cout << endl;
}
/*
<程序 12.2 运行结果>
    In static Announce1, calling Identify:
I am a meat eating animal
In dynamic Announce2, calling Identify:
I am a reptile animal
In dynamic announce2, calling Identify:
I am a warm blooded animal
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
I am a domestic cat
In dynamic Announce2, calling Identify:
I am a meat eating animal
I am a wild cat
I am a bengal tiger
I am a reptile animal
I am a warm blooded animal
I am a domestic cat
I am a meat eating animal
*/
```

应用：几何图形以及虚方法



Shape 类可以作为包括 Circle 和 Rectangle 类在内的一系列几何派生类的基础。本应用中，我们将给出 Rectangle 类的说明并在同一个程序中使用它和 Circle 以说明虚函数的使用。

在 Rectangle（矩形）类中，基点位于对象的左上角。与 Circle 类相似，Rectangle 类要用它自己的虚方法 Draw 重写基类中的 Draw 方法以显示矩形。类中还定义了面积 Area(length * width)以及周长 Perimeter (2 * (length + width))以及获取和改变 length(长)和 width(宽)值的方法。我们给出类的声明并请读者参考补充程序中的文件"geometry.h"。

```
// 派生类 Rectangle;继承了类 Shape
class Rectangle: public Shape
{
    protected:
        // 用保护数据成员描述矩形
          float length, width;

    public:
        // 构造函数.参数为基点,长度,宽度及填充方式
        Rectangle(float h=0, float v=0, float l=0,
                  float w=0, int fill = 0);
        // 矩形数据的访问函数
        float GetLength(void) const;
        void SetLength(float l);
        float GetWidth(void) const;
        void SetWidth(float w);
        // 重写基类中的虚函数
        virtual void Draw(void) const;  // 显示矩阵
        virtual float Area(void) const;
        virtual float Perimeter(void)const;
};
```

程序 12.3　几何类和虚函数

本程序演示与 Shape(基)类、Circle(派生)类和 Rectangle(派生)类相关的动态联编技术以及多态性。Circle 对象 C 是一个静态对象，变量 one，two 和 three 是指向 Shape 对象的

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

指针。

先使用静态联编技术，将圆 C 的面积和周长分别用方法 C.Area()和 C.Circumference()计算出来。接着定义用于指向动态建立的 Circle 和 Rectangle 对象的 Shape 指针变量。然后使用动态联编技术，用派生类中相应的 Area/Perimeter 方法计算对象的面积和周长。指针 three 被赋予 Circle 对象 C 的地址，因此，当要求 C 的面积和周长时可以动态地联编相应的 Circle 方法。

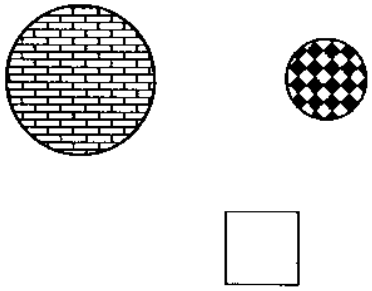
动态联编允许我们用 3 个基类指针执行 3 个图形各自的 draw 方法。

```
#include <iostream.h>
#include "graphlib.h"
#include "geometry.h"
void main(void)
{
    // 圆 C 的圆心为(3,1),半径为 0.25;变量 three 为一指向它的 Shape 指针
    Circle C(3,1,.25,11);
    Shape *one, *two, *three = &C;
    char eol;

    // 圆 *one 圆心为(1,1),半径为 0.5.矩形 *two 的基点为(2,2),长宽均为 0.5.
    one = new Circle(1,1,.5,4);
    two = new Rectangle(2,2,.5,.5,6);

    cout << "Area/perimeter of C and figures one-three:" << endl;
    cout << "C:" << C.Area() << "  " << C.Perimeter() << endl;
    cout << "one:" << one->Area() << "  "
         << one->Perimeter() << endl;
    cout << "two:" << two->Area() << "  "
         << two->Perimeter() << endl;
    cout << "three:"<< three->Area() << "  "
         << three->Perimeter() << endl;

    cout << "Type <return> to view figures." << endl;
    cin.get(eol);

    // 初始化图形系统
    InitGraphics();

    one->Draw();          // 画圆
    two->Draw();          // 画矩形
    three->Draw();        // 画圆
    ViewPause();
    // 关闭图形系统
    ShutdownGraphics();
}
/*
<程序 12.3 运行结果>
    Area/perimeter of C and figures one-three:
C: 0.196349  1.570795
one: 0.785398  3.14159
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
two: 0.25  2
three: 0.196349  1.570795
Type <return> to view figures one and two and three.
*/
```



虚方法和析构函数

当一个类需要分配动态内存时就必须定义类析构函数。如果类被用作基类，则析构函数必须是虚函数。这是我们要通过基类指针维护对象表时必须要考虑的细微但重要的一点，如果基类的析构函数不是虚函数，则用基类指针指向的派生对象的析构函数将得不到执行。下面的例子可以说明这一问题。假设 BaseCL 类的构造函数动态地分配含 7 个整数的数组。其析构函数必须释放这些数据的空间。

```
class BaseCL
{
    ...
    public:
        BaseCL(...);        // 申请 7 个元素的数组
        ~BaseCL (void);  // 释放空间(非虚函数)
}
```

类 DerivedCL 继承 BaseCL 类并执行同样的操作：

```
class DerivedCL : public BaseCL
{
    ...
    public:
        DerivedCL(...);          // 申请 7 个元素的数组
        ~DerivedCL(void);        // 释放空间(非虚函数)
}
```

假设 P 是一个 BaseCL 类的指针，但被指向了一个动态 DerivedCL 对象，我们调用 delete 函数：

```
Base *p = new DerivedCL();   // 建立一个 DerivedCL 对象
delete p;                    // 调用基类的析构函数
```

由派生类所生成的动态数据将不会被释放。如果基类的析构函数被声明为虚函数，那么派生类的析构函数将被调用。这种情况下也要调用基类的析构函数，但要等到派生类析构函数被调用之后。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

一般地，如果类被用作继承层次结构中的基类，则一定要使其具有虚析构函数，即使建立一个什么也不做的析构函数，如：

```
virtural BaseCL::~BaseCL(void)
{}
```

这就确保了若派生类有析构函数则它可以被调用。

12.4 抽象基类

介绍完继承以后，我们就知道可以通过调用派生类中具有相同名字的方法使用基类中的虚方法。因为基类方法为虚，所以可以用动态联编，这样可以确保被调用的是合适的版本。例如，Shape 类中定义了虚方法 Draw，令其完成设置填充模式的初级任务。我们所派生出的每个几何类都有自己的重载 Draw 函数以描绘出特定的形状。在 Shape 类中，我们还定义了虚方法 Area 和 Perimeter。这些操作对于仅含基点和填充模式的 Shape 对象来说是没有意义的。我们假定它们的定义对于派生几何类才有意义。在基类中操作声明为虚方法就可以保证动态联编过程将调用特定几何对象的合适的函数版本。我们可以定义返回 0 值的函数。

```
// 基类中仅放置函数的定义
float Shape::Area(void) const
{
    return 0.0;              // 点的面积
}
float Shape::Perimeter(void) const
{
    return 0.0;              // 点的周长
}
```

为了省却程序员不得不建立这类空壳子的麻烦，C++中允许使用在定义中附加“=0”的“纯虚函数”。例如

```
virtual float Area(void) const = 0;
virtual float Perimeter(void) const = 0;
```

在基类中使用纯虚函数意味着不在基类中进行具体实现，而是强制在各派生类中进行具体实现。例如，每个派生几何类都必须定义一个 Area 和 Perimeter 方法。若 Shape 类中包括了纯虚函数，则我们可以肯定不可能创建任何单独的 Shape 对象。这个类只能被用作另一类的基类。具有一个或多个纯虚函数的类被称作“抽象类(abstract class)”。从抽象类中派生出的任何类必须为每个纯虚函数提供具体实现，否则它也只是一个抽象基类，也不能生成对象。

例 12.3

抽象类 BaseCL 中包含两个纯虚函数，因此它是抽象基类。

```
class BaseCL
{
    ...
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    public:
        virtual void F(void) = 0;   // 纯虚函数
        virtual void G(void) = 0;   // 纯虚函数
};
```

派生类 DerivedCL 中定义了 F,但没有定义 G,因而 DerivedCL 仍是抽象类。

```
class DerivedCL: public BaseCL
{
    public:
        // 由于未定义 G,无法定义 DerivedCL 对象.该类仍然为抽象基类,可供其它定义
        // 了函数 G 的派生类继承.
        virtual void F(void);
};
```

以下声明会导致编译错误:

```
DerivedCL D;
```

错误:

```
connot create instance of abstract class 'DerivedCL'
```

抽象基类——表

抽象基类相当于派生类的模板,其中的数据和方法可能被所有派生类所共享。纯虚函数实际上是声明了必须由派生类实现的公有方法。作为例子,我们设计了抽象类 List 作为表集合的模板。类中的数据值 size 用来定义方法 ListSize 和 ListEmpty。每个派生类都可以调用这些函数,但要求类能在增加或删除表项或清空表时正确维护 size 的值。虽然基类中定义了方法 ListSize 和 ListEmpty,但派生类中可以重写它们或将其作为缺省值接受。基类中的其它方法被声明为纯虚方法,在派生类中它们必须被重写。诸如 Insert 一类的函数则与特定的集合类有关。在某种派生情况下,Insert 可以将数据插入到线性表中。对于二叉搜索树或词典集合的情况,则需要不同的插入算法。

List 类说明

```
template <class T>
class List
{
    protected:
        // 表中元素个数,由派生类修改
        int size;

    public:
        // 构造函数
        List(void);
        // 访问表的函数
        virtual int ListSize(void) const;
        virtual int ListEmpty(void) const;
        virtual int Find(T& item) = 0;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 表修改方法
        virtual void Insert(const T& item) = 0;
        virtual void Delete(const T& item) = 0;
        virtual void ClearList(void) = 0;
};
```

List 方法的实现

任何派生类中的表修改方法都必须维护基类数据成员 size。size 的值由 List 构造函数初始化为 0。

```
// 构造函数置 size 值为 0
template <class T>
int List<T>::List(void): size(0)
{}
```

List 类方法 ListSize 和 ListEmpty 仅仅取决于 size 的值。它们由基类实现并可用于任何派生类。

```
// 返回表的大小
template <class T>
int List<T>::ListSize(void) const
{
    return size;
}
// 检验空表
template <class T>
int List<T>::ListEmpty(void) const
{
    return size == 0;
}
```

由抽象基类 List 派生出 SeqList 类

我们在第 1 章中先引入了 SeqList 类，然后在后续各章中实现了数组和链表。现在我们将 SeqList 类作为抽象类 List 的派生类进行研究。抽象类中没有 DeleteFront 和 GetData 方法，因为它们仅适用于顺序表。

SeqList 类描述

声明

```
template <class T>
class SeqList: public List<T>
{
    protected:
        // 链表对象，派生类可以访问
        LinkedList<T> llist;

    public:
        // 构造函数
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        SeqList(void);
        // 访问表的方法
        virtual int Find (T& item);
        T GetData(int pos);
        // 表修改方法
        virtual void Insert(const T& item);
        virtual void Delete(const T& item);
        T DeleteFront(void);
        virtual void ClearList(void);
        // SeqListIterator 需访问 llist
        friend class SeqListIterator < T > ;
};
```

说明

既然继承了抽象类 List，SeqList 类就必须遵循 List 中所规定的操作。因为 SeqList 类实现的是顺序表，所以带位置参数的 GetData 以及从表中删除头一个结点的方法 DeleteFront 都加入到派生类的公有方法中。

Insert，Delete 和 ClearList 方法维护的是被保护的类数据成员 size。因而，方法 ListSize 和 ListEmpty 不需要进行重写。

可以用一种叫作"迭代算子(iterator)"的工具对 SeqList 对象进行遍历。这种工具被声明为 SeqListIterator 对象，必须能够访问表。访问权是通过将 SeqListIterator 声明为友元函数获得的。下一节中将专门论述迭代算子。SeqList 的派生版本在文件"seqlist2.h"中给出。

SeqList 的实现(派生类版本)

大部分实现工作已经在第 9 章完成。我们必须做的是定义函数 Insert，Delete，ClearList 和 Find。可以照搬它们在链表中的定义但要加上维护来自 List 类的数据值 size 的操作。例如，Insert 方法是：

```
// 用方法 InsertRear 在表尾中加入元素
template < class T >
void SeqList < T > ::Insert(const T& item)
{
    list.InsertRear(item);
    size+ + ;              // 修改 List 中的 size 值
}
```

在派生类中，构造函数 SeqList 调用 List 构造函数，将 size 置为 0。

```
// 缺省构造函数，初始化基类
template < class T >
SeqList < T > ::SeqList(void): List < T > ()
{}
```

12.5 迭代算子

许多表处理算法都假定我们可以对表项进行扫描并进行一些操作。由 List 派生出的

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类可以提供增加和删除数据项的方法，但一般来说它并不提供专门用来对表进行扫描的方法。实际上它是假定有一些外部过程可以遍历表并维护表当前位置的记录。

对于数组或 SeqList 对象 L，可以用循环和位置索引遍历表。对于 SeqList 对象 L，方法 GetData 用来访问数据值。

```
for (pos = 0; pos < List Size(); pos + +)
    cout < < L.GetData(pos) < < " ";
```

对于二叉树、哈希表以及词典，表遍历更复杂一些。例如，树遍历是递归的，可以用递归的中序、前序或后序扫描完成。这些扫描方法可以加到二叉树的维护类中。然而，递归函数并不允许客户程序中断遍历过程去执行其它任务，然后再继续递归过程。正如我们在第 13 章中将要见到的，可以通过在堆栈中维护树结点指针完成迭代遍历过程：对每一种遍历次序，树类中都必须包含其迭代实现，即使客户程序可能并不进行树遍历或者可能始终用一种次序进行遍历。更好的方法是将数据抽象从控制抽象中分离出来：表遍历问题的解决方法是建立一个迭代算子(iterator)类，其任务是对数据结构，如链表或树中的元素进行遍历。迭代算子被初始化时，(当前指针)指向表头位置(头、根等)。迭代算子提供了允许我们在表中移动的方法 Next()和 EndOfList()。迭代算子对象还维护一个在两次 Next 调用之间迭代算子的有关状态记录。

有了迭代算子，客户程序就可以中断扫描过程去检查数据项的内容并执行其他一些任务。客户程序因此就有了一种可以遍历表而无需理会层次索引或指针的工具。令迭代算子作为类中的友元函数，我们可以建立起遍历对象和类的关联，使迭代算子能访问表数据项。迭代算子中方法的具体实现要用到表的底层结构。

这一节我们对迭代算子进行一般性讨论。我们用虚函数技术声明一个抽象基类以供构造所有迭代算子之用。抽象类的所有迭代算子操作提供共同的接口，虽然派生的迭代类在具体实现上各异。

迭代算子抽象基类

我们定义抽象类 Iterator 作为通用表迭代算子的模板。本文其余部分所设计出的所有迭代算子都是从此类所派生出的。文件“iterator.h”中包含该类。

Iterator 类说明

声明

```
template < class T >
class Iterator
{
    protected:
        // 是否已到表尾标志，由派生类维护
        int iterationComplete;

    public:
        // 构造函数
        Iterator(void);

        // 迭代算子需要的方法
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        virtual void Next(void) = 0;
        virtual void Reset(void) = 0;
        // 数据检索/修改方法
        virtual T& Data(void) = 0;
        // 是否已到表尾
        virtual int EndOfList(void) const;
};
```

讨论

迭代算子是一种表遍历工具。其基本方法是 Reset(置为第一个数据元素),Next(位置移到下一项),以及 EndOfList(识别表尾)。函数 Data 访问当前表元素的数据值。

Iterator 类的实现

抽象类中只有一个数据值 iterationComplete,它必须由各个派生类中的 Reset 和 Next 进行维护。只有构造函数和 EndOfList 方法是在抽象类中实现的:

```
// 构造函数,置 iterationComplete 为 0(False)
template < class T >
Iterator < T > ::Iterator(void): iterationComplete(0)
{}
```

EndOfList 方法只是返回 iterationComplete 的值。若表为空则派生方法 Reset 将数据值置为 1(True)。若派生类方法 Next 将要越过表尾,则它将 iterationComplete 置为 True。

表迭代算子的派生

SeqList 在本书中已得到广泛使用并且它还是设计抽象类 List 的基础。因其重要性,我们从派生 SeqListIterator 入手。迭代算子中维护一个指向当前被扫描的 SeqList 对象的指针 listPtr。既然 SeqListIterator 是派生类 SeqList 的友元函数,它就被允许访问 SeqList 类的私有成员。

SeqListIterator 类说明

声明

```
// 从抽象类 Iterator 派生出的 SeqListIterator 类
template < class T >
class SeqListIterator: public Iterator < T >
{
    private:
        // 指向供遍历的 SeqList 的指针
        SeqList < T > * listPtr;
        // 指向表中上一位置及当前位置的指针
        Node < T > * prevPtr, * currPt

    public:
        // 构造函数
        SeqListIterator(SeqList < T > & lst);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 必须定义的遍历方法
        virtual void Next(void);
        virtual void Reset(void);
        // 必须定义的数据检索/修改方法
        virtual T& Data(void);
        // 重置迭代算子以遍历新表
        void SetList(SeqList < T > & lst);
};
```

讨论

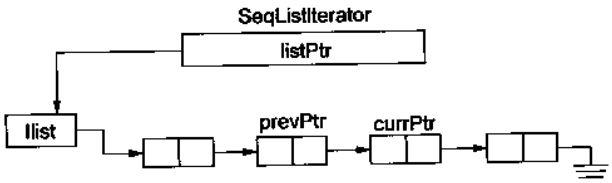
迭代算子中实现了在基类 Iterator 中被声明为纯虚函数的 Next，Reset 和 Data。方法 SeqList 是 SeqListIterator 类所特有的，它允许客户程序在运行时为迭代算子分配另一SeqList 对象。迭代算子和 SeqList 类都在文件"seqlist.h"中。

例

```
SeqList < int > L;                    // 创建表
    SeqListIterator < int > iter(L);  // 创建迭代算子；拼在表 L 的尾部
cout < < iter.Data();                 // 输出当前数据值
iter.Next();                          // 移往表中下一位置
    // 循环扫描表并输出数据值
for (iter.Reset(); ! iter.EndOfList(); iter.Next())
    cout < < iter.Data() < < " ";
```

建立 SeqList 迭代算子

当迭代算子由构造函数生成后，它就被联编到特定的 SeqList 中，其所有操作都适用于该表。迭代算子维护一个指向 SeqList 对象的指针。



将迭代算子附着到表上以后，对 iterationComplete 进行初始化并将当前位置置为表头。

```
// 构造函数。初始化基类并设置 SeqList 指针
template < class T >
SeqListIterator < T > ::SeqListIterator(SeqList < T > & lst):
      Iterator < T > (), listPtr(&lst)
{
    // 置 iterationComplete 值
    iterationComplete = listPtr- > llist.ListEmpty();
    // 将迭代算子指向表头
Reset();
}
```

Reset 通过初始化 iterationComplete 并将指针 prevPtr 和 currPtr 设为当前位置在表前端时各自的指向位置。SeqListIterator 类也是 LinkedList 类的友元函数，因而可以访问数据成

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

员 front。

```
// 移到表头
template <class T>
void SeqListIterator<T>::Reset(void)
{
    // 重置迭代算子状态
    iterationComplete = listPtr->llist.ListEmpty();
    // 若表为空,则返回
    if (listPtr->llist.front == NULL)
        return;
    // 移到表的头结点
    prevPtr = NULL;
    currPtr = listPtr->llist.front;
}
```

SetList 方法相当于运行时的构造函数。新的 SeqList 对象 lst 被作为参数传递,迭代算子对 lst 进行遍历。重新对 listPtr 赋值并调用 Reset。

```
// 用 iterator 遍历 lst,重置 listPtr 并调用 Reset
template <class T>
void SeqListIterator<T>::SetList(SeqList<T>& lst)
{
    listPtr = &lst;

    // 将遍历指向新表中 lst 数据
    Reset();
}
```

迭代算子可以通过方法 Data()对当前表元素的数据值进行访问。该函数所返回的数据项值是通过 currPtr 访问 LinkedList 结点的数据值得到的。

```
// 返回当前表元素的数据值
template <class T>
T& SeqListIterator<T>::Data(void)
{
    // 若表为空或遍历结束,则出错退出
    if (listPtr->llist.ListEmpty() || currPtr == NULL)
    {
        cerr << "Data: invalid reference!" << endl;
        exit(1);
    }
    return currPtr->data;
}
```

从一个数据项到另一个数据项是通过方法 Next 实现的。扫描过程一直持续到当前位置到达表尾为止。这个条件的标志是由 Next 所维护的整数值 iterationComplete。

```
// 走向下一表元素
template <class T>
void SeqListIterator<T>::Next(void)
{
    // 若 currPtr 为 NULL,则已到表尾
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    if (currPtr = = NULL)
        return;
    // 将 prevPtr 和 currPtr 均前移一个结点
    prevPtr = currPtr;
    currPtr = currPtr - > NextNode();
    // 若已到达表尾,置 iterationComplete 值为 1
    if (currPtr = = NULL)
        iterationComplete = 1;
}
```

程序 12.4　SeqList Iterator 类的使用

某公司每个月都要生成一个包括推销员的识别码和所推销出的商品数量在内的记录(SalesPerson)。表 SalesList 中包含商店的累计 SalesPerson 信息。另一个表 idList 中所记录的是雇员的识别码。我们从文件"sales.dat"中读取若干个月的销售信息并将每个记录添加到 SalesList 中。因为记录中可能涉及若干个月份,所以同一个推销员可能对应若干个记录。但是一个雇员在 idList 中只能出现一项。

一旦输入了数据,我们就将迭代算子 idIter 和 salesIter 分配给相应的表。在扫描 id List 的过程中,用识别码(id)识别每个雇员并用它作参数调用函数 PrintTotalSales。此函数扫描 salesList 并统计识别码为 id 的雇员所销售的商品总数。函数最后打印出雇员的识别码以及总销售量。

```
# include < iostream.h >
# include < fstream.h >
# include "seqlist2.h"
// 使用从 List 及 SeqListIterator 继承来的 SeqList
// 存放推销员 id 号及销售量的记录
struct SalesPerson
{
    int idno;
    int units;
};
// 重载运算符 = = ,比较雇员的 id 号
int operator = = (const SalesPerson &a, const SalesPerson &b)
{
    return a.idno = = b.idno;
}
// 用 id 作为键值扫描表,并将 id 号 = = id 的雇员的销售量求和,输出最后结果
void PrintTotalSales(SeqList < SalesPerson > & L, int id)
{
    // 定义 SalesPerson 变量并初始化其值
    SalesPerson salesP = {id, 0};
    // 定义顺序表迭代算子并用它扫描表
    SeqListIterator < SalesPerson > iter(L);
    for (iter.Reset(); ! iter.EndOfList(); iter.Next())
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 若雇员号相等,则增加其销售量
        if (iter.Data() = = salesP)
            salesP.units + = (iter.Data()).units;

    // 输出推销员 id 及其销售总量
    cout < < "Sales person " < < salesP.idno
         < < "    Total Units Sold " < < salesP.units < < endl;

void main(void)
{
    // 存放推销员记录及雇员号的表
    SeqList < SalesPerson > salesList;
    SeqList < int > idList;
    ifstream salesFile;          // 存放输入数据的文件
    SalesPerson salesP;          // 存放输入的变量
    int i;

    // 打开输入文件
    salesFile.open("sales.dat", ios::in | ios::nocreate);
    if (! salesFile)
    {
        cerr < < "File 'sale.dat' not found!";
        exit(1);
    }
    // 以'idno units'格式读取数据直到文件结束
    while (! salesFile.eof())
    {
        // 读入数据并将其插入到链表 salesList 中
        salesFile > > salesP.idno > > salesP.units;
        salesList.Insert(salesP);
        // 若 id 未在 idlist 中,将其加入
        if (! idList.Find(salesP.idno))
            idList.Insert(salesP.idno);
    }
    // 为两个表建立迭代算子
    SeqListIterator < int > idIter(idList);
    SeqListIterator < SalesPerson > salesIter(salesList);
    // 扫描 id 表并用每个 id 作为参数调用函数 PrintTotalSales 来求每个推销员的销售
    // 总量
    for (idIter.Reset();! idIter.EndOfList();idIter.Next())
        PrintTotalSales(salesList, idIter.Data());

/*
< 文件'Sales.dat' >
300     40
100     45
200     20
200     60
100     50
300     10
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
400      40
200      30
300      10
<程序 12.4 运行结果>
    Sales person 300     Total Units Sold 70
Sales person 100     Total Units Sold 95
Sales person 200     Total Units Sold 110
Sales person 400     Total Units Sold 40
*/
```

数组迭代算子

当我们试图将迭代算子联编到表类上时，可能会因为数组索引操作比较容易而忽略Array(数组)类。事实上，Array 迭代算子也是一种有用的抽象。通过将迭代算子初始化为从特定元素开始到另一特定元素结束，应用程序中就可以不再用索引了。更进一步地，多个迭代算子甚至可以同时遍历同一数组。下面我们以归并同一数组中的两个有序归并段(runs)为例说明多个迭代算子的情况。

ArrayIterator 类说明

声明

```
# include "iterator.h"
template <class T>
class ArrayIterator: public Iterator<T>
{
    private:
        // 当前位置;起始点和终止点
        int currentIndex;
        int startIndex;
        int finishIndex;
        // 需遍历的数组对象的地址
        Array<T> *arr;
    public:
        // 构造函数
        ArrayIterator(Array<T>& A, int start=0, int finish=-1);
        // 基类要求的标准迭代算子操作
        virtual void Next(void);
        virtual void Reset(void);
        virtual T& Data(void);
};
```

讨论

构造函数将 Array 对象联编到迭代算子上并初始化数组的起始和终止索引。缺省的起始值为 0，它将迭代算子置于数组头一个元素的位置。终止索引的缺省值为 -1，表示

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

客户程序将数组中上一项的索引作为其上界。在迭代过程中的任一时刻，currentIndex 为当前数组元素的索引。索引的初始值被置为 StartIndex。ArrayIterator 类在文件“arriter.h”中给出。

在 Array 迭代算子中的公有成员函数中，重写基类中的纯虚函数的个数是最少的。

例

```
// 50 个浮点数组成的数组，下标范围为 0 至 49
Array < double > A(50);
// 用数组迭代算子扫描范围为 3 到 10 的数组元素
ArrayIterator < double > arriter(Arr, 3, 10);
// 输出下标为 3 到 10 的数组元素
for (arriter.Reset();! arriter.EndOfList();arriter.Next())
    cout < < arriter.Data() < < "";
```

应用：归并有序归并段

第 14 章将正式研究包括外部归并排序在内的排序算法，所谓外部归并排序，是对磁盘上的数据进行排序。该算法将所需数据放在一个元素表中，并将它们分隔为被称作“归并段(runs)”的有序子表中。

定义

在表 $X_0, X_1, \cdots, X_{n-1}$ 中，若满足以下条件则子表 $X_a, X_{a+1}, \cdots, X_b$ 是一个归并段：

$X_i \leqslant X_{i+1}$；对 $a \leqslant i < b$

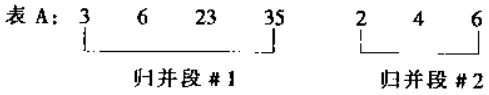
$X_{a-1} > X_a$；若 $a > 0$

$X_{b+1} < X_b$；若 $b < n-1$

例：在数组 X 中，$X_2 \cdots X_5$ 是一个归并段：

X：20　35　　15　25　30　65　　50　70　10

归并过程将归并段叠到一起并生成有可能更大的有序子表，直到产生有序表为止。



本应用中只涉及到全部算法中的一小部分。假定数据按两个归并段放在 N 个元素的数组中。第 1 个归并段占据 0 到 $R-1$ 的位置；第 2 个归并段占据 R 到 $N-1$ 的位置。如上含 7 个元素的数组 A 中，归并段以下标 $R=4$ 为分界线。

对归并段进行逐项归并以后的结果是产生了有序表。我们在每个归并段的开始设置当前遍历位置。比较当前位置值并将较小者复制到一个数组中。若某个归并段的值被取用，则进到该归并段的下一个值的位置并继续进行比较。既然子表本来就是有序的，那么将元素复制到输出数组中后也是有序的。一个归并段被用完后，将另一个归并段中的剩余各项复制到输出数组中。

本算法用 3 个迭代算子 left，right 和 output 实现，编码非常简洁。迭代算子 left 遍历第

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

1 个归并段；right 遍历第 2 个归并段；而 output 则将数据值赋值到输出数组中。图 12.2 中示意了算法用于样本数据中选定项的情况。

程序 12.5　归并有序归并段

函数 Merge 所用数据是 Array 对象 A 中所存储的两个归并段，它把这两个归并段归并到局部 Array 对象 Out 中。这一过程使用由参数 lowIndex，endOfRunIndex 和 highIndex 所初始化的迭代算子 left 和 right。迭代算子 output 负责将有序数据发送到 Out 中。比较过程在某一归并段结束时终止。函数 Copy 将另一归并段中的剩余数据添加到数组 Out 中，在重置完 output 数组的迭代算子后，我们将排好序的表拷回到 A 中。

本程序从文件"rundata"中读取 20 个整数值。在输入期间，我们将数据存储到数组 A 中并标出归并段的终止下标，因为 Merge 函数要用到它。调用 Merge 对数组排序，将数组打印出来。

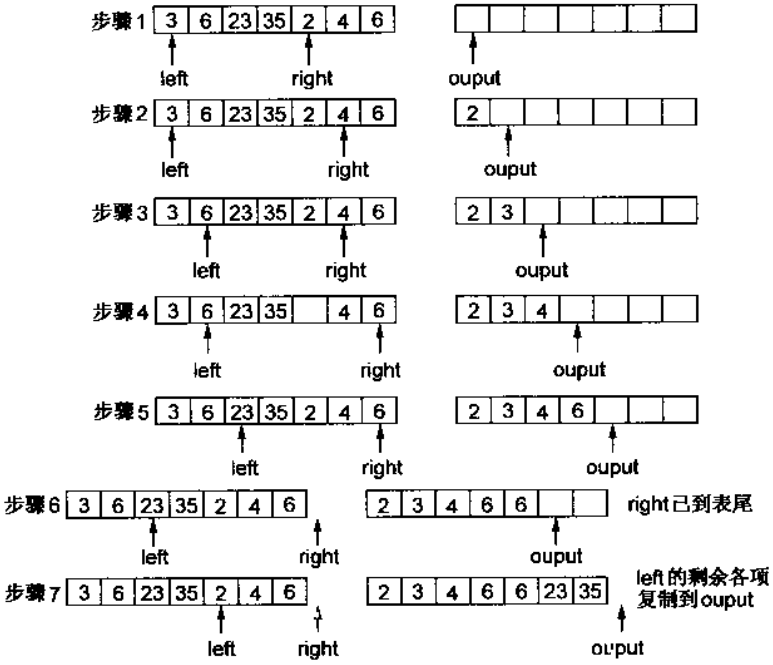


图 12.2　归并有序归并段

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

#include "array.h"
#include "arriter.h"

// 用迭代算子将一个数组中的值拷贝到另一数组中
void Copy(ArrayIterator<int>& Source, ArrayIterator<int>& Dest)
{
    while (! Source.EndOfList())
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    {
        Dest.Data() = Source.Data();
        Source.Next();
        Dest.Next();
    }
}

// 将两个归并段归并到数组 A 中。第一个归并段下标范围为 lowIndex 至 endOfRun - 1.
// 第二个归并段为 endOfRun 至 highIndex
void Merge(Array < int > & A, int lowIndex, int endOfRunIndex,
           int highIndex)
{
    // 归并段要归并到的数组,大小和数组 In 相同
    Array < int > Out(A.ListSize());

    // left 遍历第一个归并段,right 遍历第二个
    ArrayIterator < int > left(A, lowIndex, endOfRunIndex - 1);
    ArrayIterator < int > right(A, endOfRunIndex, highIndex);
    // output 将排序后数据送入 Out
    ArrayIterator < int > output(Out);
    // 拷贝数据,直到遇到一个或两个归并段的结尾
    while (! left.EndOfList() && ! right.EndOfList())
    {
        // 若 left 的数据值小于或等于 right 数据值,将其拷贝到 Out 并将 left 指针前移
        // 一个元素
        if (left.Data() < = right.Data())
        {
            output.Data() = left.Data();
            left.Next();
        }
        // 拷贝 right 的数据并前移 right 指针
        else
        {
            output.Data() = right.Data();
            right.Next();
        }
        output.Next();            // 前移 output
    }
    // 若某归并段已结束,将另一个拷贝到 Out
    if (! left.EndOfList())
        Copy(left,output);
    else if (! right.EndOfList())
        Copy(right, output);

    // 重置 Out 的迭代算子并将 Out 数据拷回 A 中
    output.Reset();

    ArrayIterator < int > final(A);  // 用于拷回 A 中.
    Copy(output, final);
}

void main(void)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{
    // 存放从流 fin 中读入的数据的数组
    Array < int > A(20);
    ifstream fin;
    int i;
    int endOfRun = 0;

    // 打开数据文件"rundata"
    fin.open("rundata", ios::in | ios::nocreate);
    if (! fin)
    {
        cerr < < "Cannot open the file 'rundata'" < < endl;
        exit(1);
    }

    // 将 20 个数据值读入到两个归并段中
    fin > > A[0];

    for (i = 1; i < 20; i + + )
    {
        fin > > A[i];
        if (A[i] < A[i - 1])
            endOfRun = i;
    }

    // 归并两个归并段 A[0]..A[endOfRun - 1]和 A[endOfRun]..A[19]
    Merge(A, 0, endOfRun, 19);

    // 输出归并后数据,每行 10 个数据
    for (i = 0; i < 20; i + + )
    {
        cout < < A[i] < < "  ";
        if (i = = 9)
            cout < < endl;
    }
    cout < < endl;
}
/*
< 文件"rundata" >
1 3 6 9 12 23 33 45 55 68 88 95
2 8 12 25 33 48 55 75
< 程序 12.5 运行结果 >
1 2 3 6 8 9 12 12 23 25
33 33 45 48 55 55 68 75 88 95
*/
```

ArrayIterator 的实现

构造函数设置迭代算子的初始状态,它将迭代算子联编到数组上并初始化 3 个下标。如果 startIndex 和 finishIndex 为缺省值(0 和 - 1),则迭代算子作用于整个数组。

```
// 构造函数。初始化基类及数据成员
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
template < class T >
ArrayIterator < T > ::ArrayIterator(Array < T > & A, int start,
                                    int finish): arr(&A)
{
    // 数组下标的最大值
    int ilast = A.ListSize() - 1;
    // 初始化下标值。若 finish = = -1,遍历整个数组
    currentIndex = startIndex = start;
    finishIndex = finish ! = -1 ? finish : ilast;
    // 下标必须在数组范围内
    if (! ((startIndex > = 0 && startIndex < = ilast) &&
          (finishIndex > = 0 && finishIndex < = ilast) &&
          (startIndex < = finishIndex)))
    {
        cerr < < "ArrayIterator: Index parameter incorrect!"
             < < endl;
        exit(1);
    }
}
```

Reset 将当前下标重新定位到起始点并将 iterationComplete 初始化为 0,以表示开始新的遍历。

```
// 重置到数组的起始点
template < class T >
void ArrayIterator < T > ::Reset(void)
{
    // 置当前下标为开始遍历的下标
    currentIndex = startIndex;

    // 迭代尚未结束
    iterationComplete = 0;
}
```

Data 方法用 CurrentIndex 访问数据项。如果当前遍历位置越过了表尾,则该方法产生出错信息并终止程序。

```
// 返回当前数组元素的值
template < class T >
T& ArrayIterator < T > ::Data(void)
{
    // 若已遍历完数组,则出错退出
    if(iterationComplete)
    {
        cerr < < "Iterator has passed the end of the list!"
             < < endl;
        exit(1);
    }

    return ( * arr) [currentIndex];
}
```

如果迭代结束,Next 无需做什么即可返回;否则 Next 需要将 CurrentIndex 的值增 1 并

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

更改逻辑型基类变量 iterationComplete 的值。

```
// 前移至下一数组元素
template <class T>
void ArrayIterator<T>::Next (void)
{
    // 若迭代尚未结束,将 currentIndex 加 1,若已超过 finishIndex,则置迭代结束
    if (! iterationComple' e)
    {
        currentIndex + + ;
        if(currentIndex > finishIndex)
            iterationComplete = 1;
    }
}
```

12.6 有序表

对于 SeqList 类所建立的表,表项从表尾加入到表中,最终得到的表是没有任何特定次序的。在许多应用中,客户程序在使用表结构时都要求数据项有序存放。这样,应用程序可以有效地判断表项是否在表中并可以按序输出表项。

为建立有序表,我们用 SeqList 作为基类设计出派生类 OrderedList,它用"<"运算符按升序插入表项。这是继承操作中的一个强有力的例子。我们仅重新定义 Insert 方法,因为所有其他表操作都不影响排序且可从基类中继承。

OrderedList 类定义

声明

```
# include "seqlist2.h"

template <class T>
class OrderedList: public SeqList<T>
{
    public:
        // 构造函数
        OrderedList(void);

        // 重写 Insert 来得到有序表
        virtual void Insert (const T& item);
};
```

说明

除 Insert 之外的所有表操作都直接取自 SeqList,因为它们不影响排序,只有 Insert 必须定义,因为它要重写 SeqList 中的 Insert 方法。Insert 方法对表进行扫描并将一个表项插入到适当的位置以维持表的次序。

OrderedList 类在文件"ordlist.h"中给出。

OrderedList 类的实现

OrderedList 类中定义了一个构造函数,它仅调用 SeqList 构造函数。这就初始化了基

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

类,进而也就初始化了 List 基类。我们有了 3 个类的层次结构链例子。

```
// 构造函数。初始化基类
template <class T>
OrderedList<T>::OrderedList(void): SeqList<T>()
{}
```

类中定义了将数据项放在表中适当位置的新的 Insert 函数。新的 Insert 方法使用 LinkedList 类中的内置遍历机制搜寻比新数据项值大的第 1 个数据值。InsertAt 用来将新值插入到链表中的当前位置。如果新值比现有值都大,则数据值被添加到表尾。Insert 方法负责增加基类 List 中的 size 变量的值。

```
// 按升序将表项插入表中
template <class T>
void OrderedList<T>::Insert(const T& item)
{
    // 用链表的遍历机制确定插入位置
    for (llist.Reset();!llist.EndOfList();llist.Next())
        if (item < llist.Data())
            break;

    // 往当前位置插入表项
    llist.InsertAt(item);
    size++;
}
```

应用:长归并段 程序 12.5 中给出了将两个有序归并段合并到一个有序归并段中的归并排序算法。程序假定输入数据经预处理后被放到两个归并段中。在本应用中我们将讨论过滤(预处理)数据以生成长归并段的技术。

假设有一大块数据以随机顺序存储在数组或磁盘中。在这种形式下,数据可能由一系列短归并段组成。例如下面的 15 字符数据集由 8 个归并段组成。

```
CharArray:[a  k]  [g]  [c  m  t]  [e  n]  [l]  [c  r  s]  [c]  [b  f]
```

当我们试图用归并法对数据排序时,那么多要合并的归并段也许会使我们望而生畏。本例中,我们将短归并段合并为 4 个归并段。

```
[a  g  k]  [c  e  m  n  t]  [c  l  r  s]  [b  c  f]
```

在下一趟归并排序时,我们须将 4 个并段合并为 2 个,将这 2 个并段归并以后我们会最终得到一个有序表。如果数据被初始化为比较合适的长归并段,算法的性能会好一些。这可以通过扫描表项并将它们收集到有序子表中实现。外部排序算法必须克服磁盘访问相对较慢这种不利因素的影响,因此常包含一个用来预处理数据的过滤器。我们必须注意在过滤数据上花费一些时间确实可以提高算法的总体性能。

一个有序表可以提供过滤器的简单示范。假设一个初始数组或文件中包含 n 个元素。我们将每一 k 元组插入到一个有序表中并将表复制回数组中。过滤器保证归并段中至少有 k 个元素。例如,假设 k 等于 5,要处理的是 CharArray 中的字符。所产生的归并段是:

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

[a c g k m] [c e l n t] [b c f r s]

第 13 章中将用堆设计本过滤器的一个改进版本。

程序 12.6 长归并段

程序用有序表将范围在 100 到 999 内的 100 个随机整数所构成的数组过滤到若干归并段中，每个归并段至少含有 25 个元素。每个新随机数都被插入到 OrderedList 对象 L 中。对于每一组(25 个)元素，Copy 函数将它们从 L 中删除并将它们插回到数组 A 中。程序最后打印所得到的数组 A。

```
#include <iostream.h>
#include "ordlist.h"
#include "array.h"
#include "arriter.h"
#include "random.h"
// 遍历整数数组并按每行 10 个整数输出所有表项
void PrintList(Array<int>& A)
{
    // 用数组迭代算子
    ArrayIterator<int> iter(A);
    int count;

    // 遍历表并输出之
    count = 1;
    for(iter.Reset();!iter.EndOfList();iter.Next(), count++)
    {
        cout << iter.Data() << " ";
        // 输出表,每行 10 个数据
        if (count % 10 == 0)
            cout << endl;
    }
}
// 从有序表 L 中删除表项并插入到数组 A 中。修改指 A 中下一下标的 loadIndex 值
void Copy(OrderedList<int> &L, Array<int> &A, int &loadIndex)
{
    while (! L.ListEmpty())
        A[loadIndex++] = L.DeleteFront();
}
void main(void)
{
    // 用有序表 L 创建 A 中归并段
    Array<int> A(100);
    OrderedList<int> L;
    // 随机数发生器
    RandomNumber rnd;
    int i, loadIndex = 0;
    // 生成 100 个范围在 100 到 999 的随机数。用一 25 元素的有序表进行过滤。表满后,
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 用 Copy 将元素拷贝到数组 A 中
    for (i = 1; i <= 100; i++)
    {
        L.Insert(rnd.Random(900) + 100);
        if (i % 25 == 0)
            Copy(L,A,loadIndex);
    }
    // 输出数组 A 中的最后结果
    PrintList(A);
}

/*
<程序 12.6 运行结果>

110  116  149  152  162  240  345  370  422  492
500  532  578  601  715  730  732  754  815  833
850  903  929  947  958  105  132  139  139  190
205  216  221  243  287  348  350  445  466  507
513  524  604  634  641  730  784  940  969  982
296  375  412  437  457  466  507  550  594  652
725  728  771  799  803  815  859  879  909  915
940  990  991  992  994  101  118  123  155  310
343  368  372  434  443  489  515  529  557  574
641  739  774  784  829  875  883  922  967  972
*/
```

12.7 异构表

存储具有同一数据类型的对象的集合被称为“同构的(homogeneous)”。到现在为止，本书中所出现的集合都是同构的。反之，具有不同类型的对象的集合被称为“异构的(heterogeneous)”。因为 C++ 数据结构的类型是在编译时确定的，所以我们必须引入新的程序设计技术才能实现异构集合。这一节我们将实现异构数组和链表，假定表中每个对象都派生自一个公共基类。

异构数组

对异构数组的一般情况进行彻底讨论已经超出了本书的范围。我们仅把讨论限定于指向不同的对象的指针数组。在此，我们要回顾一下第 1 章中说明多态性的一个例子。为方便起见，我们重述一遍例子：

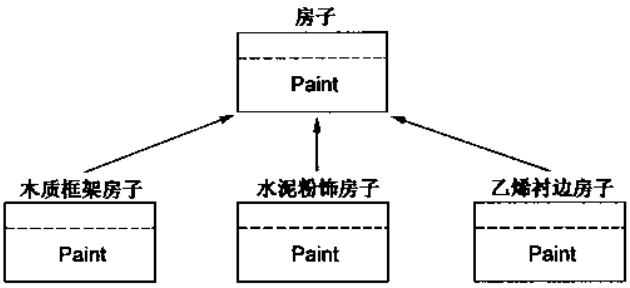
在油漆房子时必须进行一系列基本操作。此外，每种类型的房子还需要一些特定的专门技术。例如，对于木质框架的房子可能需要用砂擦折叠板；对于乙烯衬边的房子则需要清洗折叠板。用面向对象的程序设计术语来说，这些房子代表由包含通用油漆操作的基类 House 派生出的各种不同类。名为 Paint 的方法与每个类都有关联。

基类 House 包含标识串“House”以及打印标识串的虚方法 Paint。每个派生类都重写此 Paint 方法并指示这种类型的房子已被油漆过。

```
// 用于房子油漆的基类
class House
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
{
    private:
        String id;          // 房子的 id 号
    public:
        // 构造函数,将 id 初始化为"House"
        House(void)
        {
            id = "House";
        }
        // 虚方法.输出串"House"
        virtual void Paint(void)
        {
            cout < < id;
        }
};
```

每个派生类中都包含一个标识房子类型的串。虚方法 Paint 打印串并调用基类的 Paint 方法。WoodFrameHouse 的声明可以作为样板。房屋类的完全描述在文件"houses.h"中给出。

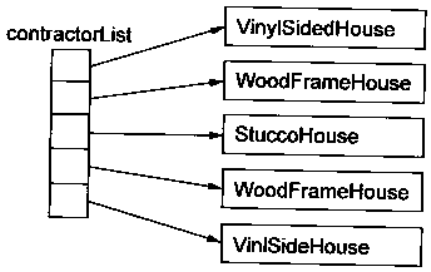
```
class WoodFrameHouse: public House
{
    private:
        // 木质框架房子的 id
        String id;
    public:
        // 构造函数,将 id 赋值为"Wood Frame"
        WoodFrameHouse(void): House()
        {
            id = "Wood Frame";
        }
        // 虚方法.输出 id 并调用基类中的 Paint
        virtual void Paint(void)
        {
            cout < < "Painting a" < < id < < "";
            House::Paint();
        }
};
```

为了说明异构数组的概念,我们定义一个包含 5 个基类指针的数组 contractorList。数

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

组的初始化是这样实现的：对于数组中的每个指针，从派生房屋类 WoodFrameHouse，StuccoHouse 和 VinylHouse 中随机选择一个，将类的对象分配给数组中的指针。例如，以下是一个含 5 个房屋的表 contractorList。



我们可以认为这个指针数组是含有 5 个待油漆的住宅地址的表。承包商要派员工去工作。在本例中，承包商给每个员工分派一个住宅地址并假定一旦他们见到房屋就可以正确地进行油漆操作。

程序 12.7　异构数组

程序遍历指针数组 contractorList 并调用每个对象的 Paint 方法。因为对象由指针引用，所以动态联编可以确保执行正确的(房屋)类中的 Paint 方法。这相当于承包商派员工到表中各住宅去并粉刷它们。

```
# include < iostream.h >
# include "random.h"        // 引入随机数发生器
# include "houses.h"        // 引入油漆屋类
    void main(void)
{
    // 房屋地址动态表
    House * contractorList[5];
    RandomNumber rnd;
    // 构造需油漆的 5 个房屋表
    for (int i = 0, i < 5; i + + )
        // 随机选择房屋类型 0,1,2,创建对象,并将其地址赋给 contractorList.
        switch(rnd.Random(3))
        {
            case 0: contractorList[i] = new WoodFrameHouse;
                    break;
            case 1: contractorList[i] = new StuccoHouse;
                    break;
            case 2: contractorList[i] = new VinylSidedHouse;
                    break;
        }
    // 用方法 paint 油漆房子,由于它为虚函数,用动态联偏技术来调用正确的方法
    for (i = 0; i < 5; i + + )
        contractorList[i] - > Paint();
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
/*
<程序 12.7 运行结果>
Painting a Wood Frame House
Painting a Stucco House
Painting a Vinyl Sided House
Painting a Stucco House
Painting a Wood Frame House
*/
```

异构链表

与异构数组类似,异构表结构中的每个对象都派生自一个通用的基类。每个类对象的基本成分中都包含一个指象表中下一个对象的指针。使用多态性技术,指针就可以用于执行派生对象中的方法,而不管其类型如何。

我们设计一个几何对象的链表来说明这些概念。这些几何对象从 Shape 类的某种结点型变异派生得到。

NodeShape 类定义

声明

```
#include "graphlib.h"
// 基类 Shape
class NodeShape
{
    protected:
        // 基点坐标,填充方式及指向下一结点的指针
        float x, y;
        int fillpat;
        NodeShape *next;
    public:
        // 构造函数
        NodeShape(float h = 0, float v = 0, int fill = 0);
        // 虚方法 Draw
        virtual void Draw(void) const;
        // 表处理方法
        void InsertAfter(NodeShape *p);
        NodeShape *DeleteAfter(void);
        NodeShape *Next(void);
};
```

说明

坐标(x,y)给出的是将以填充模式 fillpat 画出的派生对象的基点。Shape 中包含初始化图形系统填充模式的 Draw 方法以及指向链表中后继 Shape 对象的指针域 next。方法 InsertAfter 在当前结点后插入结点,DeleteAfter 删除当前结点的后一个结点,它们维持链表为循环表。方法 Next 返回指向下一个结点的指针。

NodeShape 类在文件"shapelst.h"中给出。

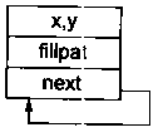
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

NodeShape 类的实现

NodeShape 类是以第 9 章中的 CNode 类为样板实现的。因为我们假定链表为循环表，所以构造函数必须初始化一个指向自身的结点。

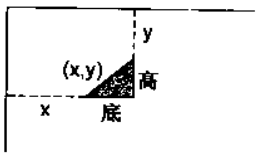
```
// 构造函数。初始化基点,填充方式,并将其置为指向自身
NodeShape::NodeShape(float h, float v, int fill):
                    x(h), y(v), fillpat(fill)
{
    next = this;
}
```



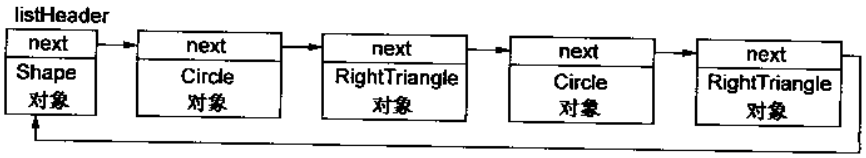
派生链式几何类 几何类 CircleFigure 和 Rectangle 可以由 NodeShape 类派生得到。除了基类中的方法以外,这些类中还提供了 Draw 方法以重写基类中的虚方法 Draw。Area 和 Perimeter 方法则未包括在内。我们以 CircleFigure 类为例进行说明。

```
// CircleFigure 派生自基类 NodeShape
class CircleFigure: public NodeShape
{
    protected:
        // 圆的半径
        float radius;
    public:
        // 构造函数
        CircleFigure(float h, float v, float r, int fill);
        // Draw 方法,画一个圆
        virtual void Draw(void) const;
};
// 构造函数。初始化基类及半径
    CircleFigure ::CircleFigure(float h, float v, float r, int fill):
                NodeShape(h, v, fill), radius(r)
{}
// 调用基类 Draw 方法设置填充模式,然后画圆
void CircleFigure::Draw(void) const
{
    NodeShape::Draw();
    DrawCircle(x,y,radius);
}
```

文件"shapelist.h"中还包括一个新的几何类 RightTriangle,它用斜边最左点、底和高描述一个直角三角形。



为了形成链表,先声明一个名叫 listHeader、类型为 NodeShape 的头结点。从头结点出发,动态生成每个结点,并用 InsertAfter 将它们添加到表尾。例如,下面的迭代过程建立一个 4 结点表,其中圆(Circle)和直角三角形(RightTriangle)对象相间排列。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 表头及遍历新表的指针
NodeShape listHeader, *p;
float x, y, radius, base, height;
// 将 p 指向表头结点
p = &listHeader;
// 往表中装入 4 个结点
for (int i=0; i < 4; i++)
{
    // 基点坐标
    cout << "Enter x and y: ";
    cin >> x >> y;
    if (i % 2 == 0)        // 若 i 为偶数,则插入圆
    {
        cout << "Enter the radius for a circle: ";
        cin >> radius;
        // 申请填充模式为 i 的对象,加入表中
        p->InsertAfter(new Circlue(x,y,radius, i));
    }
    else          // i 为参数,则加入直角三角形
    {
        cout << "Enter base and height for a right triangle: ";
        cin >> base >> height;
        p-InsertAfter(new RightTriangle(x,y,base, height, i));
    }
    // 将 p 指向刚产生的结点
    p = p->Next();
}
```

在遍历表以及描画对象时动态联编是很关键的。在以下代码段中的每个结点处,p 实际上指向一个 Circle 或 RightTriangle 对象,而 Draw 是虚函数。派生类中的 Draw 方法被执行。

```
p = listHeader.Next();
while (p != &listHeader)
{
    p->Draw();
    p = p->Next();
};
```

现在我们就可以完整地讨论有关异构表建立和管理的问题了。

程序 12.8　异构表

程序对上述例子中的原理进行了推广,生成了一个包括 Circle,Rectangle 和 Right Triangle 类型对象在内的链表。文件"figures"中指定了各表项,其每一行的格式如下:

```
<图形>    <基点 x,y>    <图形参数>
```

图形用字符 c(circle,圆),r(rectangle,矩形)或 t(right triangle 直角三角形)表示。基点是一对浮点数;参数包括半径或边长。例如,以下是各图形的输入参数示范:

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
c  0.5  0.5  0.25          // 圆心为(1/2, 1/2),半径为 1/4 的圆
r  1.0  0.25  .5  .5       // 基点为(1,1/4),两边分别为 1/2,1/2 的矩形
t  2.0  0.75  .25  .5      // 基点在(2,3/4,底和高分别为 1/4,1/2 的直角三角形)
```

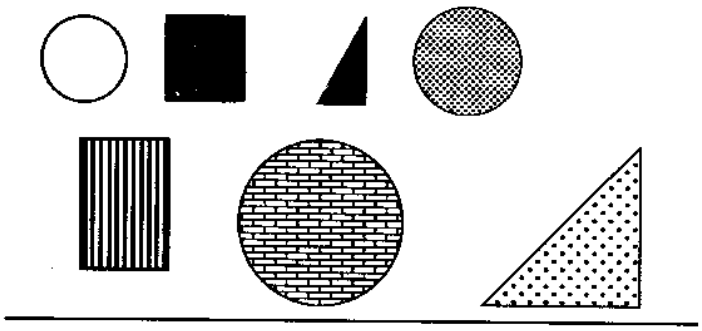
程序读(数据)文件并建立几何对象的链表。在表遍历期间画出图形。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "graphlib.h"
#include "shapelst.h"
void main(void)
{
    // listHeader 为图形循环表的表头
    NodeShape listHeader, *p, *nFig;
    // 图形: 'c'(圆),'r'(矩形),'t'(直角三角形)
    char figType;
    // 开始时填充方式为不填充
    int pat = 0;
    float x, y, radius, length, width, tb, th;
    // 打开存放图形数据的文件"figures"
    fin.open("figures", ios::in | ios::nocreate);
    if (! fin)
    {
        cerr << "Cannot open 'figures'" << endl;
        exit(1);
    }
    // 将 p 指向表头
    p = &listHeader;
    // 读文件数据直到文件结束,建立图形链表
    while(! fin.eof())
    {
        // 读入图形类型及基点
        fin >> figType;
        if(fin.eof())
            break;
        fin >> x >> y;
        // 建立相立图形
        switch(figType)
        {
        case 'c':
            // 读入半径,并往表中插入一个圆
            fin >> radius;
            nFig = new CircleFigure(x,y,radius, pat);
            p->InsertAfter(nFig);
            break;
        case 'r':
            // 读入长、宽,并往表中插入矩形
            fin >> length >> width;
            nFig = new RectangleFigure(x, y, length, width, pat);
            p->InsertAfter(nFig);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            break;
        case 't':
            // 读入底和高并往表中插入直角三角形
            fin >> tb > th;
            nFig = new RightTriangleFigure(x,y,tb,th,pat);
            p->InsertAfter(nFig);
            break;
        }
        // 填充方式值加 1,前移指针到表尾
        pat = (pat+1) % 12;
        p = p->Next();
    }

    // 初始化图形系统
    InitGraphics();

    // 从第一个图形开始,遍历表并画出每个图形
    p = listHeader.Next();
    while (p != &listHeader)
    {
        p->Draw();
        p = p->Next();
    }

    // 暂停下来,观察图形后关闭图形系统
    ViewPause();
    ShutdownGraphics();
}
/*
<程序 12.8 运行结果>
    <图形在此>
    */
```



书面作业

12.1 (a) 以 12.1 节中的动物层次结构为样板,为下列事物设计层次树:
运载工具、汽车、柴油机、汽油、飞机、电动式、推进器、喷气式

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(b) 找出“电动式”和“喷气式”的基类。

(c) 列出所有既是基类又是派生类的类。

(d) “运载工具”的派生类有哪些？

12.2 有声明如下：

```
class BASE
{
    private:
        Base_Priv;
    protected:
        Base_Prot;
    public:
        Base_Pub;
};
class DERIVED: public BASE
{
    private:
        Derived_Priv;
    protected:
        Derived_Prot;
    public:
        Derived_Pub;
};
```

(a) 基类和各派生类中都包含私有、保护和公有成员。填写下表以标明客户程序或对象对成员的访问权限。在表中“×”表示对象可以访问成员。例如，表中派生对象可以访问基类的保护成员，客户程序可以访问基类的公有成员。

	基类私有成员	基类保护成员	基类公有成员	派生类私有成员	派生类保护成员	派生类公有成员
基类						
派生对象		×				
客户程序			×			

(b) 类可以用“私有”继承进行派生。在此情况下，基类的所有公有和保护成员可供派生类使用。但对于派生类的客户程序来说，基类的的公有成员会被当作私有，因而是不可访问的。当基类仅被用作实现派生类的工具时有时会用到这类继承。对这类继承，完成下表。

	基类私有成员	基类保护成员	基类公有成员	派生类私有成员	派生类保护成员	派生类公有成员
基类						
派生对象						
客户程序						

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

12.3 下面给出类 Base 的轮廓，试说明各派生类声明中有何错误。

```
class Base
{
    ...
  public:
    Base(int a, int b);
    ...
};
```

(a)

```
class DerivedCL1: public Base
{
  private:
    int q;
  public:
    DerivedCL1(int z): q(z)
    {}
    ...
};
```

(b)

```
class DerivedCL2: public Base
{
  private:
    ...
  public:
    // DerivedCL2 无构造函数
    ...
};
```

12.4 考察以下的派生类和基类轮廓：

```
class BaseCL
{
  protected:
    int data1;
    int data2;
  public:
    BaseCL(int a, int b = 0): data1(a), data2(b)
    {}

    BaseCL(void): data1(0), data2(0)
    {}
      ...
};
class DerivedCL
{
  private:
    int data3;
  public:
    // 构造函数一
    DerivedCL(int a, int b, int c = 0);
    // 构造函数二
    DerivedCL(int a);
      ...
};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(a) 编写构造函数一，使得 a 被分给派生类，而 b 和 c 则被分给基类。

(b) 编写构造函数二，使得 a 被分给派生类，而对基类用缺省构造函数。

(c) 假定 DerivedCL 类的构造函数定义如上，试给出下列对象中 data1，data2 和 data3 的值：

```
Derired CL    obj1(1,2),    obj2(3,4,5),    obj3(8);
```

12.5 下面的程序演示在继承链中的构造函数和析构函数的执行次序。Base1，Base2 和 Derived 这 3 个类都各有一个构造函数和析构函数。给出程序的输出。

```
# include <iostream.h>
class Base1
{
    public:
        Base1(void)
        {
            cout << "Base1 constructor called." << endl;
        }

        ~Base1(void)
        {
            cout << "Base1 destructor called." << endl;
        }
};
class Base2
{
    public:
        Base2(void)
        {
            cout << "Base2 constructor called." << endl;
        }

        ~Base2(void)
        {
            cout << "Base2 destructor called." << endl;
        }
};
class Derived: public Base1, public Base2
{
    public:
        Derived(void): Base1(), Base2()
        {
            cout << "Derived class constructor called." <<
            endl;
        }
        ~Derived(void)
        {
            cout << "Derived class destructor called." <<
            endl;
        }
};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
void main(void)
{
    Derived objD;
    {
        Base1 objB1;
        {
            Base2 objB2;
        }
    }
}
```

12.6 考察下面的继承链：

```
class Base
{
    ...
    public:
        void F(void);
        void G(int x);
    ...
};
class Derived: public Base
{
    ...
    public:
        void F(void);
        void G(float x);
    ...
};
void Derived::G(float x)
{
    ...
    Base::G(10);        // 成员函数中使用域操作符
    ...
};
```

有声明如下

```
Derived OBJ;
```

(a) 客户程序如何引用基类中的函数 F?

(b) 客户程序如何引用派生类中的函数 F?

(c) 编译器如何对语句 OBJ.G(20)作出反应?

注：正如我们在 12.5 节中所提到的，建立类似于这种在派生类中重写非虚函数的类结构并不是一种好的做法。

12.7 12.2 节中建立了由抽象基类 Shape 和 Circle 类所构成的继承链。以下程序使用了这些类。读程序并回答随后的问题。

```
#include <iostream.h>
#include "graphlib.h"
#include "geometry.h"
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
void main(void)
{
    Circle C;

    C.SetPoint(1,2);
    C.SetRadius(0.5);
    cout << C.GetX() << " " << C.GetY() << endl;

    C.SetPoint(C.GetX(), 3);
    cout << C.GetX() << " " << C.GetY() << endl;
    C.SetFill(11);

    InitGraphics();
    C.Draw();
    ViewPause();
    ShutdownGraphics();
}
```

(a) 试解释为什么 GetX 可以由派生类调用。

(b) 试解释为什么 x 可以在方法 Draw 的定义内被引用。

(c) 程序的输出是什么?

(d) 试解释为什么语句

```
C.SetPoint(3,5);
```

是合法的,但语句

```
c.x = 3;
c.y = 5;
```

是非法的。

12.8 以下程序的输出是什么?

```
#include <iostream.h>
#include <string.h>
class Base
{
    private:
        char msg[30];
    protected:
        int n;
    public:
        Base(char s[], int m = 0): n(m)
        {
            strcpy(msg,s);
        }
        void output(void)
        {
            cout << n << endl << msg << endl;
        }
};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
class Derived1: public Base
{
    private:
        int n;
    public:
        Derived1(int m = 1): Base("Base", m-1), n(m)
        {}
        void output(void)
        {
            cout << n << endl;
            Base::output();
        }
};
class Derived2: public Derived1
{
    private:
        int n;
    public:
        Derived2(int m = 2): Derived1(m-1), n(m)
        {}
        void output(void)
        {
            cout << n << endl;
            Derived1::output();
        }
};
void main(void)
{
    Base B("Base Class",1);
    Derived2 D;

    B.output();
    D.output();
}
```

12.9 试解释为什么 Shape 类中的 Area 和 Circumference 方法是纯虚函数。

12.10 考察以下类声明：

```
class Base
{
    private:
        int x,y;
    ...
};
class Derived: public Base
{
    private:
        int z;
    ...
};
```

现有声明：

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Base B;
Derived D;
```

(a) 赋值语句

```
B = D;
```

是否合法？为什么？图示你的解答。

(b) 赋值语句

```
D = B;
```

是否合法？为什么？图示你的解答。

12.11 考察下面的类

```
class BaseCL
{
    protected:
        int one;
    public:
        BaseCL(int a): one(a)
        {}
        virtual void Identify(void)
        {
            cout << one << endl;
        }
};
class DerivedCL: public BaseCL
{
    protected:
        int two;
    public:
        DerivedCL(int a, int b): BaseCL(a), two(b)
        {}
        virtual void Identify(void)
        {
            cout << one << " " << two << endl;
        }
};
```

以及函数：

```
void Announce1(BaseCL x)
{
    x.Identify();
}
void Announce2(BaseCL& x)
{
    x.Identify();
}
void Announce3(BaseCL * x)
{
    x->Identify();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
```

给出以下代码段的输出：

```
BaseCL A(7), *p, *arr[];
DerivedCL B(3,5), C(2,4);
Announce1(A);
Announce1(C);
Announce2(B);
Announce3(&C);
p = &C;
p->Identify();
for(int i=0; i < 3; i++)
    if (i == 1)
        arr[i] = new BaseCL(7);
    else
        arr[i] = new DerivedCL(i, i+1);
for(i=0; i < 3; i++)
    arr[i]->Identify();
```

12.12 试解释为什么析构函数在任何可能被用作基类的类中应被声明为 virtual。

12.13 设计抽象基类 StackBase，其中声明堆栈操作 Push，Pop，Peek 和 StackEmpty。基类中必须包含保护的整数变量 numElements 并定义方法 StackEmpty 以返回此变量的值。派生类 Stack 在每次 Push 操作以后应增加 numElements 的值；在 Pop 操作后应减小它的值。用数组和链表这两种不同的方法实现派生类 Stack。

12.14 用书面作业 12.13 题中提供的样板设计描述队列的抽象类 QueueBase。类中应至少包含一个非纯虚方法。

12.15 什么叫迭代算子？为什么迭代算子通常必须是它要对其数据成员进行遍历的类的友元函数？称迭代算子是控制抽象有什么含义？

12.16 派生书面作业 12.14 题中的抽象类 QueueBase。使用复合法所包含的 SeqList 对象设计类 Queue。令 QueueIterator 为友类，由 SeqListIterator 派生出类 QueueIterator。所缺的仅是一个构造函数。

12.17 实现返回队尾元素的函数

```
template <class T>
T GetRear(Queue<T>& q);
```

如果队列为空，则打印出错消息并终止程序，请使用书面作业 12.16 题中所设计的 QueueIterator。

12.18 假设有一个 String 对象的数组 Array，请使用 ArrayIterator 遍历表并用 4 个空格替换所有 tab 字符。

12.19 编写删除数组中所有重复数据值并相应地修改对象大小的函数

```
void RemoveDuplicates(Array<int>& A);
```

例如，若 A 初始时是一个 20 元素的数组

```
A = {1,3,5,3,2,3,1,4,6,3,5,4,2,6,7,8,1,3,9,7},
```

调用 RemoveDuplicates 之后，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
A = {1  3  5  2  4  6  7  8  9}    (A.ListSize() = 9)
```

至少要用两个 ArrayIterator 对象，一个指定唯一数据值的位置，另一个扫描数组的尾部。

12.20 实现以下函数

```
template <class T>
T Max(Iterator<T>& collIter);
```

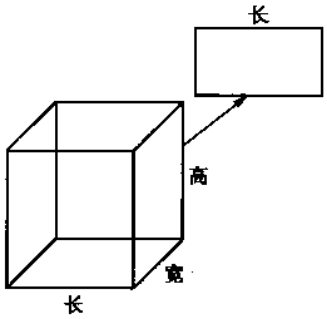
搜寻以 collIter 为迭代算子的集合中的数据的最大值。假设类型 T 定义了运算符“>”。注意函数中利用了迭代算子方法为虚这样一个事实。

12.21 请示意如何利用虚函数建立指向 Circle 和 Rectangle 对象的指针数组(异构数组)并遍历数组、打印图形的面积和周长。

上机题

12.1 实现书面作业 12.1 的继承层次结构。每个类构造函数中都必须包含打印基类和自身信息的 Identify 方法。编写一个声明各种类对象的测试程序。

12.2 从长方形可以派生出长方体。



编写实现继承链的类 Rectangle(矩形)和 Box(长方体)。Rectangle 类中包含一个面积函数以及一个返回 0 值的体积函数；Box 类中有面积和体积函数。在一个要用户给出图形类型和相关尺寸的主程序中测试你的类。为每个类定义对象并打印图形的面积和体积。

12.3 用 SeqLis 类及以下声明派生类 MidList。

```
template <class T>
class MidList: public SeqList<T>
{
    public:
        <constructor>
        virtual void Insert(const T& elt);
        virtual void Delete(const T& elt);
};
```

方法 Insert 将 elt 插入到表的中部。Delete 则从表的中部删除一项。Delete 操作假定用户检查表大小以确保元素存在。实现 MidList 并将它用于下面的测试程

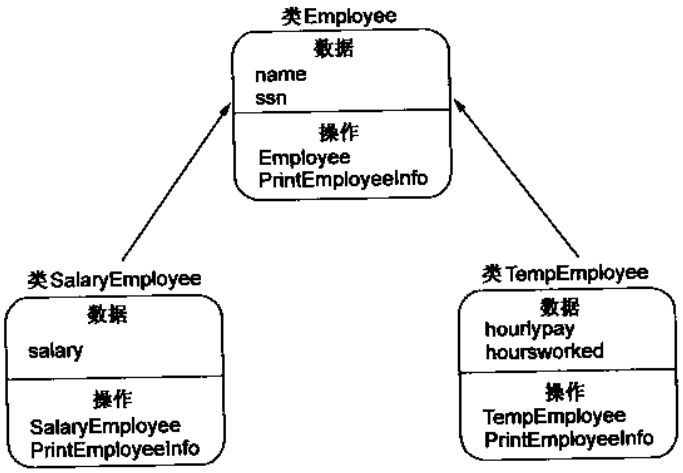
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

序中：

读取 5 个整数并用 Insert 将它们插入到表中。打印表，从表中删掉两个整数然后再打印表以及表的大小。

12.4 本题为一个数据处理问题设计继承结构。类 Employee 中包含数据项 name 和 ssn，还包含构造函数以及打印 name 和 ssn 域的 PrintEmployeeInfo。与有薪水的雇员相关的数据除了基类数据外还包括一份固定月薪。派生类 SalaryEmployee 中包含 Salary 以及打印基类和派生类中雇员数据的 PrintEmployeeInfo 操作。临时雇员数据中除了来自基类 Employee 中的信息外，还包括小时工资以及一个指定的月份中的工作小时数。信息被保存到 TempEmployee 类中，它由数据项 hourlypay 和 hoursworked 以及操作 PrintEmployeeInfo 组成。



实现以上层次结构并将它放到文件“employee.h”中。编写一个主程序，声明一个有薪水的雇员对象和一个临时雇员对象并对每个对象调用 PrintEmployeeInfo。

12.5 重新实现类 Array，将它作为抽象类 List 的派生类。必须处理一种重要的结构问题。底层的 List 类定义了一系列纯虚方法，它们必须在派生类 Array 中被重写。有些方法在派生类中可能没有意义或不起作用。下表列出了出现上述问题的方法以帮助你重新定义。

```
ListSize
    返回 Array 对象中元素个数
ListEmpty
    由于假定数组永不为空，则返回 False
ClearList
    错误！该操作对 Array 对象毫无意义
Find
    对数据元素进行顺序查找
Insert
    错误！数组为直接存取结构，插入对于 Array 对象来说是含混操作
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Delete
    错误! 删除 Array 对象中的一个元素也是含混操作.
```

运行程序 12.5 检查你的实现。

12.6 用书面作业 12.13 题中所设计的 Stack 类读取字符串以确定其是否回文。

12.7 本题中使用书面作业 12.14 和 12.16 中所设计的类的两种实现方法 Queue 和 QueueIterator。在测试程序中,读取一串整数值,直到遇到 0 为止,将正数插入到一个队列中,负数插入到另一个队列中。用 QueueIterator 对象遍历并打印每个队列中的元素。

12.8 编写程序测试书面作业 12.17 中所设计的函数 GetRear。

12.9 在主程序中,从文件中读取若干行并将每一行插入到 String 对象数组的一个元素中。用 ArrayIterator 遍历表并将所有 tab 字符替换为 4 个空格。打印修改后的串。注意本题使用书面作业 12.18 的结果。

12.10 运行下列主程序以测试书面作业 12.19 中实现的函数 RemoveDuplicates。

```
void main(void)
{
    Array < int > A(20);
    int data[] = {1,3,5,3,2,1,4,6,3,5,4,2,6,7,8,1,3,9,7};
    for (int i = 0; i < 20; i + + )
        A[i] = data[i];

    RemoveDuplicates(A);

    for(i = 0; i < A.ListSize(); i + + )
        cout < < A[i] < < " ";
    cout < < endl;
}
/*
< Run of Program >
1 3 5 2 4 6 7 8 9
*/
```

12.11 定义一个包含整数 1 至 10 的 Array < int > 对象以及数据值为'a'…'e'的 SeqList < char > 对象。用书面作业 12.20 中的函数 Max 打印各个表中的最大值。

12.12 为 12.7 节中的类 NodeShape, CircleFigure, RectangleFigure 和 RightTriangleFigure 增加方法 Area 和 Perimeter。将基类 NodeShape 中的方法定义为返回 0 值。设计一个与程序 12.8 类似的程序以建立派生对象的异构表。程序应循环遍历表并打印出每个图形的面积和周长。第二次遍历表时打印出图形。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第13章 高级非线性结构

13.1 基于数组的二叉树
13.2 堆
13.3 Heap 类的实现
13.4 优先级队列
13.5 AVL 树
13.6 AVL 树类
13.7 树迭代算子
13.8 图
13.9 Graph 类
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

本章继续研究二叉树并介绍另外几种非线性结构。第 11 章中，树是用动态生成的结点实现的，而本章讲述基于数组的树，它以数组形式建立完全二叉树。在设计堆结构和有趣的竞赛排序时它们可以发挥极大的用处。我们将对堆进行广泛研究并用这一概念实现堆排序和优先级队列。

二叉搜索树实现表并提供平均搜索时间为 $O(\log_2 n)$的搜索结构。但是当树不很平衡时效率将会降低。我们要设计一种名为作 AVL 树的新型树，它是一种深度平衡树并保持了二叉搜索树的优点。

第 12 章中曾引入了迭代算子并将其用于实现 SeqlistIterator 和 ArrayIterator 类，本章中，迭代算子的概念将延伸到针对树和图。这一功能强大的扫描工具使得我们可以用通常是线性表中才可以用的简单方法遍历非线性结构。我们将设计一个中序树迭代算子，增加树的功能，并用它来设计树排序。

图(graph)是一种一般化的层次结构，它由结点和边组成，其结点被称作顶点，而边则是两个顶点间的连接。图是有限数学中的重要课题，它的一系列经典算法可用于运算研究领域。这一章最后我们研究图的基本理论并设计一个能处理各种应用的类。

13.1 基于数组的二叉树

第 11 章中，我们曾用树结点建立一棵二叉树。树中每一项都有一个数据域以及指向左右子树结点的指针域。空树由 NULL 指针表示。插入和删除是通过动态分配结点和对指针域赋值实现的。这种表示方式可以处理包括从退化树一直到完全树在内的树。这一节我们要介绍树的顺序(数组)表示方式，即用数组项存储数据以及标识结点的索引。我们在数组和完全二叉树之间建立一种极其有用的关系，这种关系可以用于堆和优先级队列。

回顾第 11 章，深度为 n 的完全二叉树在其 n－1 层拥有所有可能的结点，其第 n 层的结点则从左至右不间断排列。

数组 A 是一个顺序表，其表项可以代表以 A[0]为根的完全二叉树；第 1 层孩子 A[1]和 A[2]；第 2 层孩子 A[3]，A[4]，A[5]，A[6]，依此类推。根结点的索引为 0，其余结点则按层次顺序被赋予索引值。图 13.1 中示意了带 10 个元素的数组 A 所对应的完全二叉树。

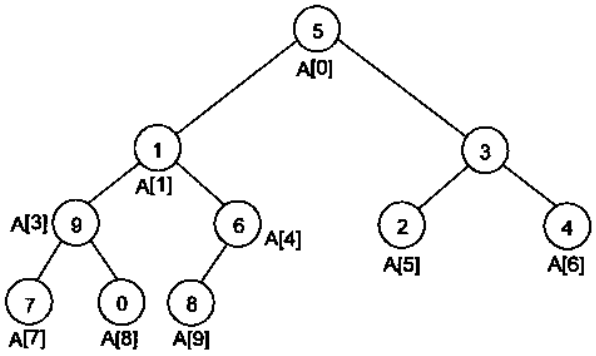


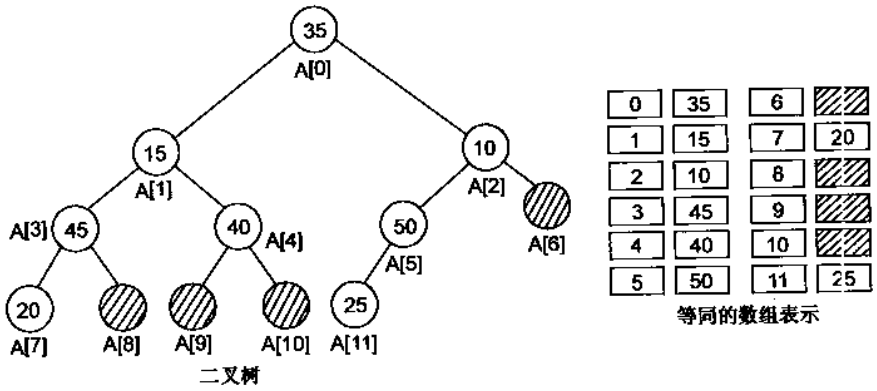
图 13.1 10 元素数组 A 构成的完全二叉树

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
int A[10] = {5, 1, 3, 9, 6, 2, 4, 7, 0, 8};
```

虽然可以很自然地找到用树表示数组的方法，但却没有直接将一棵普通二叉树表示为数组的方法。问题出在那些与未使用的数组元素相对应的空缺结点上。在以下例子中，数组含 4 个未使用项，空缺率为 1/3。仅含左子树的退化树效率会更低。



当需要直接访问结点数据时，基于数组的树的威力即可显示出来。用简单的计算方法标注结点的孩子和双亲。表 13.1 中使用了图 13.1 中所示的树。它为树的每一层的结点以及结点的孩子和双亲标注索引。

对于 N 个元素的数组中的每个结点 A[i]，孩子结点的索引由以下公式计算出来：

表项 A[i]　左孩子索引为　$2*i+1$

若 $2*i+1 \geqslant N$ 则未定义

表项 A[i]　右孩子索引为　$2*i+2$

若 $2*i+2 \geqslant N$ 则未定义

表 13.1

层次	双亲	值	左孩子	右孩子
0	0	A[0] = 5	1	2
1	1	A[1] = 1	3	4
	2	A[2] = 3	5	6
2	3	A[3] = 9	7	8
	4	A[4] = 6	9	10 = NULL
	5	A[5] = 2	11 = NULL	12 = NULL
	6	A[6] = 4	13 = NULL	14 = NULL
3	7	A[7] = 7	–	–
	8	A[8] = 0	–	–
	9	A[9] = 8	–	–

从孩子转到双亲，我们注意到结点 A[3]和 A[4]的双亲是 A[1]，结点 A[5]和 A[6]的双亲是 A[2]，依此类推。一般地，计算结点 A[i]的双亲的公式为：

表项 A[i]　双亲索引为　$(i-1)/2$

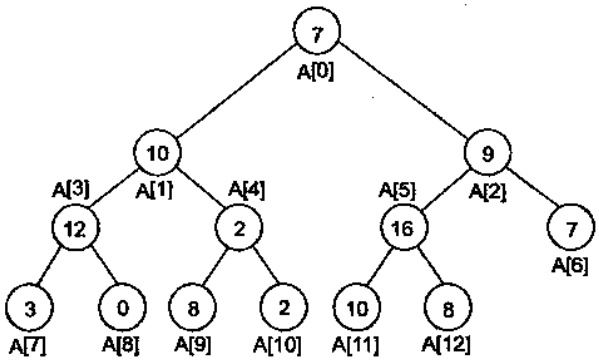
若 $i=0$ 则未定义

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

例 13.1

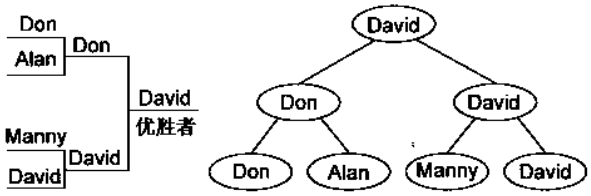
用基于数组的树遍历方法，我们可以向下走到孩子、向上走到双亲。本例遍历以下数组：



1. 从根出发，选择到较小的孩子的路径：
   路径：A[0]=7，A[2]=9，A[6]=7
2. 从根出发，选择到左孩子的路径：
   路径：A[0]=7，A[1]=10，A[3]=12，A[7]=3
3. 从结点 A[10]出发，选择到双亲的路径：
   路径：A[10]=2，A[4]=2，A[1]=10，A[0]=7

应用：竞赛排序(Tournament Sort)

二叉树的重要应用之一是作为抉择树(decision tree)，树中每个结点都代表一个具有两个可能去向的分支。这类应用的一个例子是用树存储参加一种单淘汰比赛的运动员的记录。每一个非叶子结点对应于两个运动员之间比赛的胜者。叶子结点代表了初始运动员以及对手分配情况。例如，在下面这个网球比赛的例子中，David 是优胜者，他在决赛中击败了 Don。而这两个参赛者都是在赢得了预赛后进入决赛的。Don 击败了 Alan，David 击败了 Manny。比赛的各对对手安排以及结果情况用树记录下来。



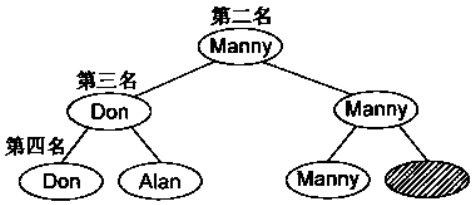
在单淘汰制比赛中，可以很快得出优胜者。例如，对于 4 个运动员的情况，比赛共有三次，而 $2^4=16$ 个运动员时则需要 $2^4-1=15$ 次比赛。

比赛虽然得出了优胜者，但是我们还是不清楚第 2 名应是谁。虽然 Don 是在决赛中负于优胜者，但他也可能不是仅次于最好的运动员。我们应该给 Manny 一个机会，因为他在第 1 轮碰上的对手也许是唯一能击败他的对手。为得出第 2 名，我们必须删除 David，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

重画竞赛树及安排 Don 和 Manny 之间的比赛。



一旦比赛的优胜者确定了，我们就可以得到运动员的正确排名。

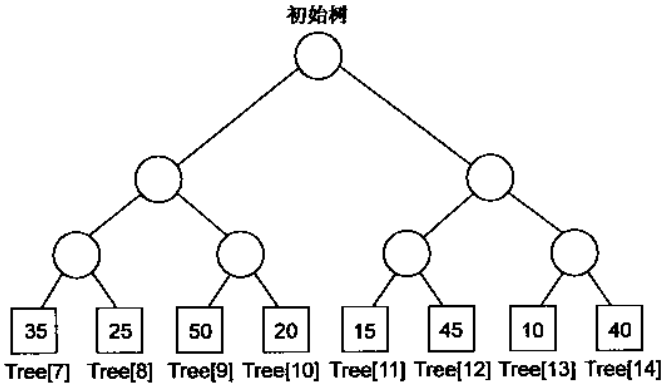
Manny 赢： 名次 David Manny Don Alan

Don 赢： 名次 David Don Manny Alan

（译注：在 Don 赢的情况下，还有一种可能的名次排列是 David，Don，Alan，Manny，但这里略去未予考虑。）

竞赛树可以用来对 N 个项目的表排序。为此我们编制一套充分利用基于数组的树的程序。建立一棵基于数组的树，其 N 个表项作为叶子结点位于树的底层。这些元素位于第 k 层，其中 $2^k \geq N$。假设我们以升序对表进行排序，我们将对各元素进行比较并将较小者（获胜者）存到双亲结点中。这一过程持续到根结点中具有最小的元素（优胜者）为止。例如，下面这棵树给出了元素个数 N = 8 的整数数组所对应的初始设置。元素被存储在第 3 层，而 $2^3 = 8$。

```
int A[8] = {35, 25, 50, 20, 15, 45, 10, 40};
```

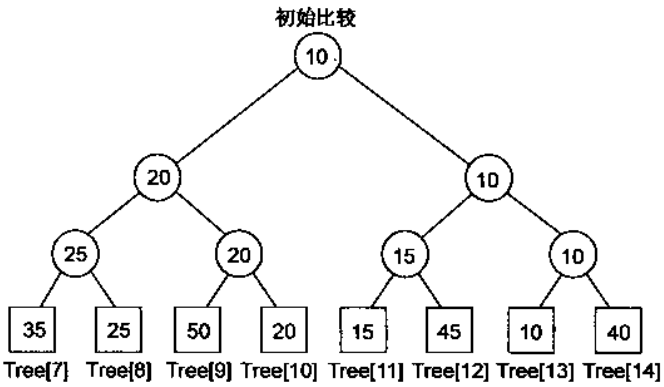


我们从第 2 层开始“比赛”，将各对中的较小者放到双亲结点中。例如 Tree[7]和 Tree[8]比较，其较小者 25 被存放到 Tree[3]中。将第 2 层和第 1 层所有各对元素进行比较。最后一次比较后，将最小的元素存到第 0 层的根结点中。

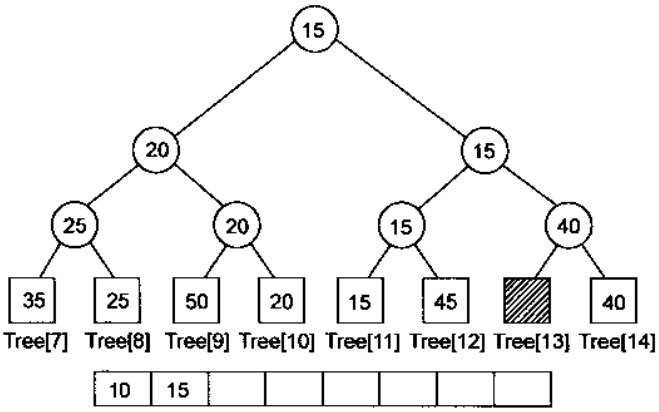
一旦最小的元素放到根结点上，就将其删除并重新赋值给数组。首先，10 被赋值给 A[0]，然后再更新树以寻找次最小元素。在竞赛模型中，有些比赛必须重新进行。因为 10 一开始是处于 A[13]的位置，这使得第 1 轮的失败者 A[14] = 40 能重新进入比赛。将 A[14]赋值给其双亲 A[6]，并在索引 6 和索引 2 处重新比赛（结果，索引 6 处：15 击败 40，索引 2 处：15 击败 20）。结果是将 15 放到根结点上，并将它标为表中次最小的元素。根

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

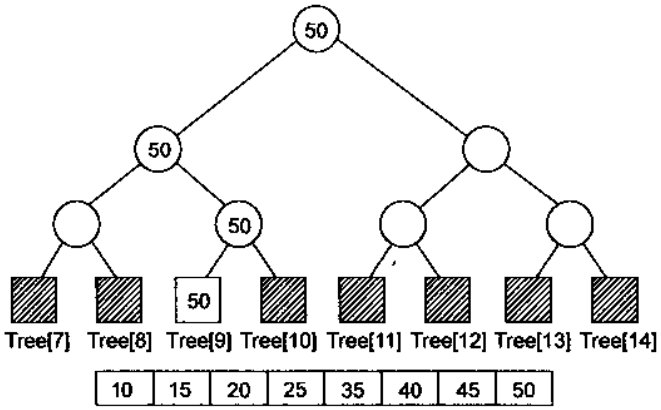
www.Linuxidc.com



然后就可以被赋值给 A[1],整个过程继续进行。



上述过程一直进行到所有叶子结点都被删除为止。在本例中,最后一个(最大的)结点进行一系列的比赛并默认为全赢。将 50 插入到 A[7]位置后,我们就得到了有序表。



算法复杂度 竞赛排序的复杂度为 $O(n \log_2 n)$。对于具有 $n = 2^k$ 个元素的数组,需要进行 $n-1$ 次比较(比赛)以找出最小元素。这可以证明如下:当往根移动时每一层竞

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

赛者都有 1/2 被淘汰掉，因此总的比赛次数是：

$$2^{k-1}+2^{k-2}+\cdots+2^1+1=2^k-1=n-1$$

随后，树被更新，其余 n－1 个元素沿通往双亲的路径各进行 k－1 次比较。总的比较次数为：

$$\begin{aligned}(n-1)+(k-1)*(n-1)&=(n-1)+(n-1)*(\log_2 n-1)\\&=(n-1)(\log_2 n)\end{aligned}$$

虽然竞赛排序中的比较次数为 $O(n\ \log_2 n)$，但空间复杂度却较高。整棵树需要 $2*n-1$ 个结点以供 $k-1$ 个轮次的比赛。

竞赛排序算法 为实现竞赛排序，我们定义一个 DataNode 类并建立一棵基于数组的对象树。类的数据成员中存储了数据项、数据项在树的底层的位置以及指示该数据项在比赛中是否仍然要用。运算符“<=”被重载以进行结点比较。

```
template <class T>
class DataNode
{
    public:
      // 数据值，其在树中的下标及是否还要用的标志
      T data;
      int index;
      int active;
      friend int operator <= (const DataNode<t> &x,
                              const DataNode<T> &y);
}
```

用函数 TournamentSort 以及实用工具 UpdateTree 实现排序，后者在通往双亲的路径上重新实施比较。实现竞赛排序的函数和变量的全部代码在文件“toursort.h”中给出。

```
// 建立基于数组的树，将数组元素拷贝到新树中；对元素进行排序后再将值拷回数组中
template <class T>
void TournamentSort (T a[], int n)
{
    // tree 为基于数组的树的根
    DataNode<T> *tree;
    DataNode<T> item;
    // 树最后一行结点个数
    int bottomRowSize;
    // 最后一行有 bottomRowSize 个结点的完全树中的结点个数
    int treesize;
    // 最后一层的起始下标
    int loadindex;
    int i, j;
    // 调用 PowerOfTwo 来决定树的最后一行所需结点个数
    bottomRowSize = PowerOfTwo(n);
    // 计算树的大小并动态申请其结点
    treesize = 2*bottomRowSize-1
    loadindex = bottomRowSize-1;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    tree = new DataNode<T>[treesize];
    // 拷贝数组到 DataNode 对象的树中
    j = 0;
    for (i = loadindex; i < treesize; i++)
    {
        item.index = i;
        if (j < n)
        {
            item.active = 1;
            item.data = a[j++];
        }
        else
            item.active = 0;
        tree[i] = item;
    }
    // 第一次比较,找到最小元素
    i = loadindex;
    while(i > 0)
    {
        j = i;
        while(j < 2*i)        // 处理参加比较的选手
        {
            // 找到对手,比较两个对手 tree[j]和 tree[j+1]的值,将胜者赋值给双亲结点
            if (! tree[j+1].active || tree[j] < tree[j+1])
                tree[(j-1)/2] = tree[j];
            else
                tree[(j-1)/2] = tree[j+1];
            j += 2;           // 取下一对选手
        }
        // 移到上一层处理本次比赛的优胜者
        i = (i-1)/2;
    }
    // 处理其余的 n-1 个元素。将优胜者从根拷到数组中,使其为不活跃态.修改树,允许优胜
    // 者的对手再次进入比赛.
    for (i = 0; i < n-1; i++)
    {
        a[i] = tree[0].data;
        tree[tree[0].index].active = 0;
        UpdateTree(tree,tree[0].index);
    }
    // 将最大值拷回到数组中
    a[n-1] = tree[0].data;
}
```

索引 i 被传递给函数 Update 作参数,它表示树的底层的当前最小元素的原始位置。这是一个要被删除(不再用到)的结点。在开始的一轮中输给最终获胜者(最小值)的值可以重新进入比赛。

```
// 参数 i 为当前表中元素的最小值的起始下标
template <class T>
void UpdateTree(DataNode<T> *tree, int i)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
    int j;
// 标明优胜者 i 的对手。通过将其赋值给双亲结点允许它重新参加比赛
    if ( I % 2 = = 0)
        tree[(i-1)/2] = tree[i-1];          // 对手为左结点
    else
        tree[(i-1)/2] = tree[i+1];          // 对手为右结点
    // 在将优胜者出局(使其不活跃)状态下,重新进行比赛
    i = (i-1)/2;
    while (i > 0)
    {
        // 位置 i 的对手为左结点或右结点?
        if (i % 2 = = 0)
            j = i-1;
        else
            j = i + 1;
        // 是否对手为不活跃?
        if (! tree[i].active || ! tree[j].active)
            if (tree[i].active)
                tree[(i-1)/2] = tree[i];
            else
                tree[(i-1)/2] = tree[j];
        // 将比赛的胜者赋给双亲
        else
            if (tree[i] < tree[j])
                tree[(i-1)/2] = tree[i];
            else
                tree[(i-1)/2] = tree[j];
        // 移动上一层进行比赛
        i = (i-1)/2;
    }
    // 本轮比赛结束。根中存放下一次最小值
}
```

13.2 堆

基于数组的树的重要应用场合之一就是堆,堆是结点间具有层次次序关系的完全二叉树。其中,双亲值大于或等于其孩子值的,叫"最大堆(maximum heap)";双亲值小于或等于其孩子值的,叫"最小堆(minimum heap)"。图 13.2 中示意了上述情况。在最大堆中,根中的元素最大;在最小堆中,根中的元素最小。本书研究最小堆。

作为表的堆

堆是按某种顺序将一系列数据以完全二叉树形式存放的一种表。这种顺序叫作"堆顺序(heap order)",它要求堆中每个结点的值都小于或等于其孩子结点的值。按这种顺序,根具有最小的数据值。作为抽象表结构,堆允许增加和删除表项。插入过程不用假定新表项占有一个特定的位置而只需维持堆顺序。但是删除操作总是删去表中的最小项(根)。堆可以用于那些客户程序想直接访问最小元素的应用场合。作为表,堆并不提供 Find(查代)操作,而对表元素的直接访问是只读的。所有的堆处理算法都有责任更新树和维护堆顺序。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

堆是一种非常有效的表管理结构，它可以充分利用其作为完全二叉树的结构优势。对于每次插入和删除操作，堆仅需扫描从根到树的末端的短路径即可恢复其顺序（“堆化”）。堆在实现优先级队列以及对一组元素排序方面可以发挥重要作用。我们可以摒弃那些较慢的排序算法，而将元素插入到堆中并通过反复删除根结点而对它们进行排序。这就产生了堆排序这种快速算法。

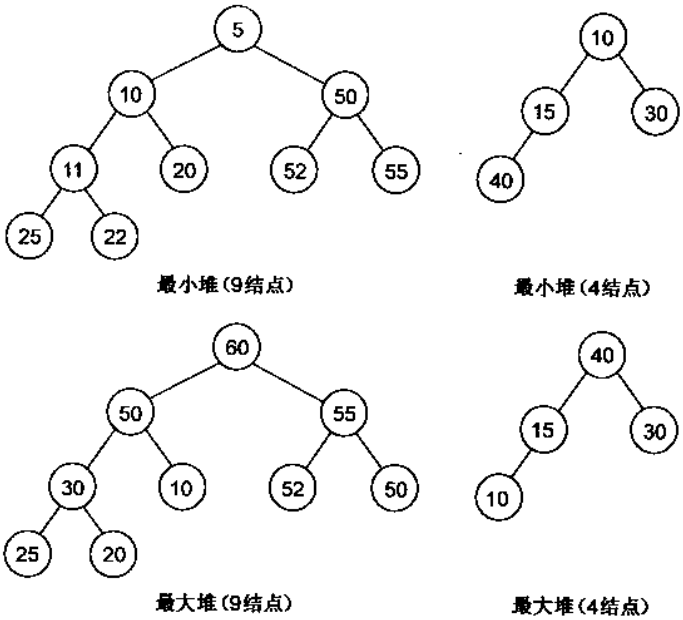


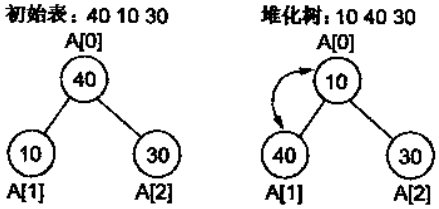
图 13.2　最大堆和最小堆

我们在设计堆类的过程中讨论堆的内部组织。增加和删除表项的算法由 Insert 和 Delete 方法实现。例 13.2 给出了一个堆并示意其部分算法。

例　13.2

1. 建立堆：数组具有对应的树表示形式。一般情况下，树并不满足堆的条件，通过重新排列元素，可以建立一棵“堆化”的树。

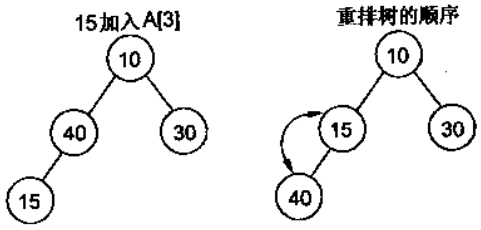
初始表：40　10　30　　堆化树：10　40　30



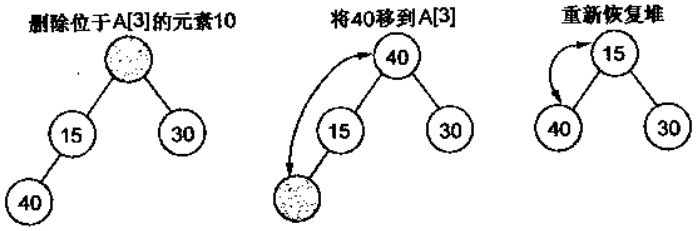
2. 插入一个元素：新元素被加入到表层，随后树被更新以恢复堆次序。例如，下面的步骤将 15 加入到表中。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



3. 删除一个元素：删除总是发生在根 A[0]处。表中最后一个元素被用来填补空缺位置，结果树被更新以恢复堆条件。例如，以下步骤删除旧。



Heap 类

与任何线性或非线性表一样，堆具有插入和删除数据项的操作，还有返回诸如表长之类的状态信息的操作。这些方法以及存储基于数组的二叉树的数组都被封装到类 Heap 中了。

Heap 类说明

声明

```
#include <isostream.h>
#include <stdlib.h>
template <class T>
class Heap
{
    private:
        // hlist 指向一个数组，该数组由构造函数申请(inArray = = 0)或作为参数传递
        // (inArray = 1)
        T *hlist;
        int inArray;
        // 堆中可存放的元素的最大个数及当前堆大小
        int maxheapsize;
        int heapsize;                    // 指明表尾
        // 出错处理函数，给出错误信息.
        void error(char errmsg[]);
        // 插入/删除后重排堆需要用到实用方法
        void FilterDown(int i);
        void FilterUp(int i);
    public:
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 构造函数/析构函数
        Heap(int maxsize);                  // 创建空堆
        Heap(T arr[], int n);               // 将 arr 建堆
        Heap(const Heap<T>& H);             // 复制构造函数
        ~Heap(void);                        // 析构函数
        // 重载运算符:"=","[]","T*"
        Heap<T>& operator= (const Heap<T>& rhs);
        const T& operator[] (int i);
        // 有关的表函数
        int ListSize(void) const;
        int ListEmpty(void) const;
        int ListFull(void) const;
        void Insert(const T& item);
        T Delete(void);
        void ClearList(void);
};
```

说明

第 1 个构造函数带一个尺寸参数,用它为数组动态分配内存。所得到的堆初始为空,新元素由 Insert 方法加入。析构函数、复制构造函数以及赋值运算符支持动态内存的使用。第 2 个构造函数以数组为参数并用数组与堆联编。构造函数按堆顺序排列数组。通过这种方法,客户程序可以在已有的数组上实现堆结构并利用堆的特性。

重载的下标运算符"[ ]"允许客户程序以数组方式访问堆对象。因为运算符返回的是常数引用,所以访问是只读的。

方法 listEmpty,ListSize 和 ListFull 返回的是有关堆的当前大小的信息。

Delete 方法总是删除堆中第 1 个(最小的)表项。Insert 将元素加入到表中并维护堆顺序。

例

```
Heap<int> H(4);                     // 4 元素整数堆
int A[] = {15, 10, 40, 30};         // 4 元素数组
Heap<int> K(A, 4);                  // 将 A 编入堆 K 中
H.Insert(85);                       // 将 85 加入堆 H
H.Insert(40);                       // 将 40 加入堆 H
cout << H.Delete();                 // 输出 H 中最小元素 40
// 输出"堆化"后数组 A
for (int i = 0; i < 4; i++)
    cout << K[i] << " ";            // 输出 10  15  40  30
K[0] = 99;                          // 非法语句
```

程序 13.1　Heap 类示范

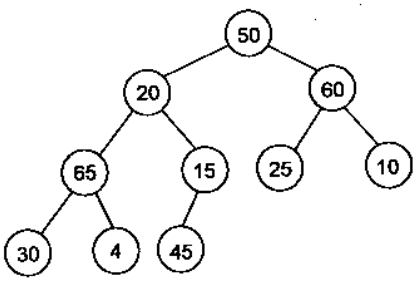
程序从一个已被初始化好的数组 A 入手,在其元素上实现堆结构。

A:50,20,60,65,15,25,10,30,4,45从堆中重复删除元素并将它们打印出来,直到堆空为止。因为堆在每次删除后要重新组织,所以元素按升序打印出来。

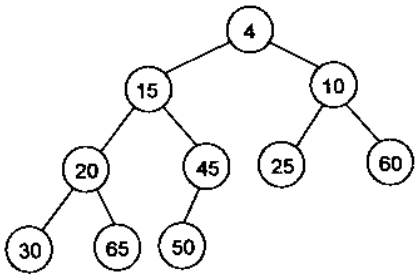
```
#include <iostream.h>
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



数组A



堆化后的表A

```
# include "heap.h"
// 输出 n 元素数组
template < class T >
void PrintList (T A[], int n)
{
    for (int i = 0; i < n; i + + )
        cout < < A[i] < < "  ";
    cout < < endl;
}
void main(void)
{
    // 初始化 10 元素数组
    int A[10] = {50, 20, 60, 65, 15, 25, 10, 30, 4, 45};
    cout < < "Initial array:" < < endl;
    PrintList(A,10);
    // 将数组 A 转化为堆
    Heap < int > H(A, 10);
    // 输出"堆化"后的数组 A
    cout "Heapified array:" < < endl;
    PrintList(A, 10);

    cout "Deleting elements from the heap:" < < endl;
    // 重复抽取最小值
    while(! H.ListEmpty())
        cout < < H.Delete() < < "  ";
    cout < < endl;
}
/*
< 程序 13.1 运行结果 >
Initial array:
50  20  60  65  15  25  10  30  4  45
Heapified array:
4  15  10  20  45  25  60  30  65  50
Deleting elements from the heap:
4  10  15  20  25  30  45  50  60  65
*/
```

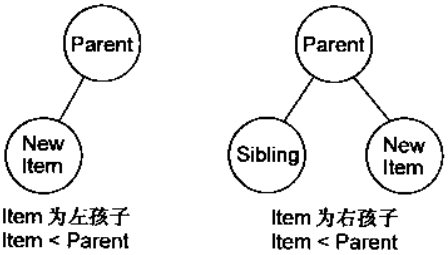

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

13.3 Heap 类的实现

下面我们对堆的插入和删除操作以及实用方法 FilterUp 和 FilterDown 进行彻底的讨论。实用方法在堆被建立和修改时负责更新堆。

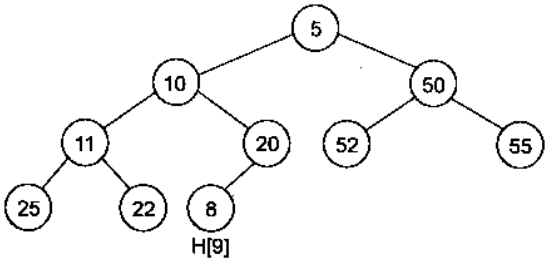
堆插入 起始时总是把新表项加入到堆尾。这样把元素放到表中可能会违反堆的条件，如果新表项的值比其双亲小，则将它们的值互相交换。下图示意了可能的情况。



交换以后，在双亲处，堆条件重新得到满足。但在树的更上层又有可能违反条件。现在必须将新的双亲结点作为孩子结点并在其双亲处检查堆顺序。如果新表项较小一些，则必须将它挪到沿双亲方向的树的更上层。考虑下面的例子，假定有一个 9 元素堆 H：

```
H  Insert(8);    // 往堆中插入 item = 8
```

将 8 插入 A[9]：将新表项插入到堆尾。该位置由索引 heapsize 所确定，它记录堆中当前表项数。



将 8 放到双亲路径上：比较 8 与双亲 20。因为孩子比双亲小，故交换它们的值(A)。沿双亲路径继续前进。现在 8 比其双亲 H[1] = 10 小，必须交换其值(B)。整个过程终止于此，因为下一个双亲结点处满足了堆条件。

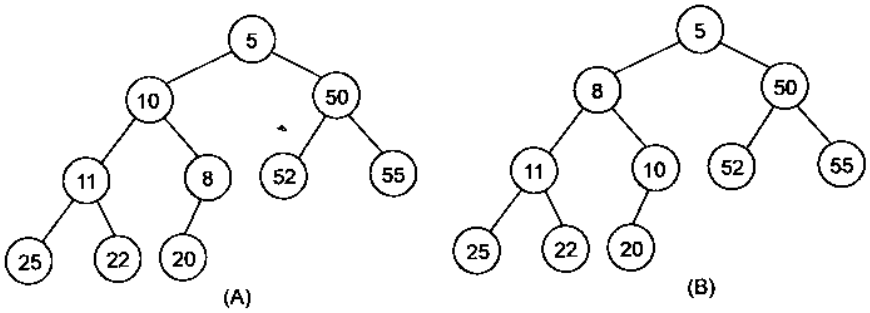
插入过程扫描双亲路径并在发现"小"双亲(双亲≤表项)或到达根结点时终止。后一种情况下，既然根结点没有双亲，新值就放到根结点处。

Insert 操作使用一种过滤方法将结点放到双亲路径上。方法 FilterUp 将新表项放到从下标 i 到根之间的双亲路径上的正确位置处。

```
// 用于重排堆的实用方法。从下标 i 开始，沿双亲路线移动，若孩子的值小于双亲，则互换两值
template <class T>
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
void Heap<T>::FilterUp (int i)
{
    int currentpos, parentpos;
    T target;
    // currentpos 为遍历双亲路径上的下标。target 为 hlist[i]的值,并被重新安置在路
    // 径中
    currentpos = i;
    parentpos = (i-1)/2;
    target = hlist[i];
    // 沿双亲路径到根
    while (currentpos ! = 0)
    {
        // 若 parent <= target, 满足堆条件,退出循环
        if (hlist[parentpos] <= target)
            break;
        else
        // 将孩子结点值移到双亲结点并修改下标访问下一个双亲
        {
        // 将两值交换后,与下一个双亲结点比较
          hlist[currentpos] = hlist[parentpos];
          currentpos = parentpos;
          parentpos = (currentpos-1)/2;
        }
    }
    // 找到恰当位置,将 target 赋值给该位置
    hlist[currentpos] = target;
}
```

公有方法 Insert 先检查是否满堆,然后才开始插入操作。将表项放到堆尾以后,该方法用 FilterUp 调整堆。

```
// 将新元素插入堆中并修改结构
template <class T>
vid Heap<T>::Insert(const T& item)
{
    // 检查堆是否已满,若是则退出
    if (heapsize == maxheapsize)
            error("Heap full");
    // 将元素存入堆尾并使堆大小加 1,调用 FilterUp 来重排堆使其满足堆条件.
    hlist[heapsize] = item;
```

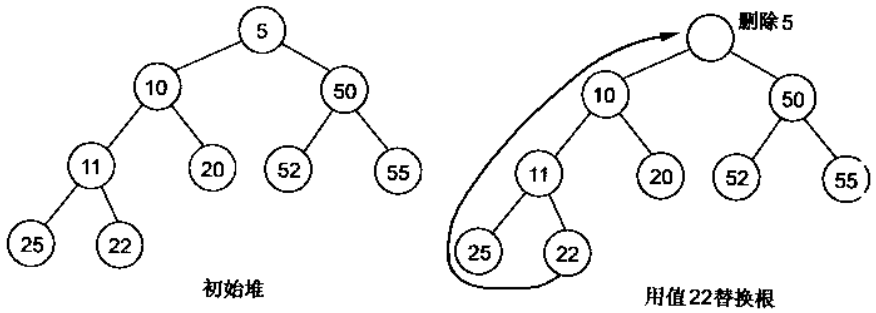

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

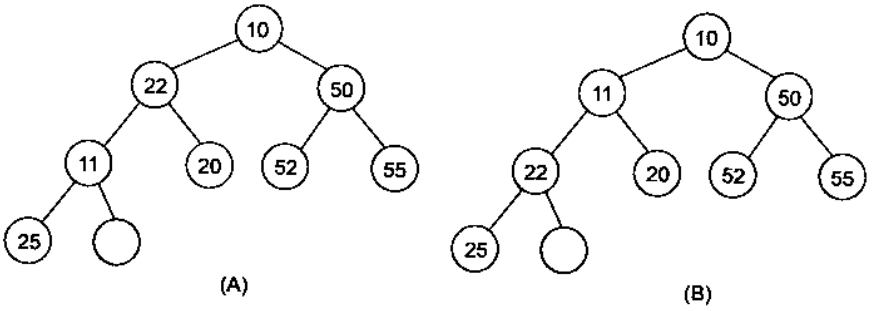
```
    FilterUp(heapsize);
    heapsize++;
}
```

堆删除 数据总是从树根被删除。删去数据后，根为空，我们先用堆中最后一个元素填充它。然而，这种替换有可能违反堆条件。我们必须沿较小的孩子的方向在堆中定位该值。如果替换值大于其孩子之一，则不满足堆条件，必须将它与较小的孩子交换位置。扫描一直持续到正确地找到双亲结点或遇到表尾为止。在后一种情况下，我们将值放到叶子结点中。例如，在以下堆中，Delete 从堆中删除结点 5。

删除结点 5 并用最后一个结点 22 替换它：堆的最后一项被复制到根结点。新的根可能不满足堆条件，我们必须扫描到孩子的路径以正确定位该值。



扫描到最小孩子的路径并定位值 22：比较根 22 与其孩子。位于 H[1]处的最小的孩子小于 22，双亲与孩子应交换位置(A)。在第 1 层，H[1]处的新双亲与其在 H[3]和 H[4]的孩子相比较。最小的孩子值为 11，它必须与其双亲交换位置(B)。所得到的树满足堆条件。



Delete 方法 Delete 操作用 FilterDown 方法将结点放到通往最小孩子的路径上。函数以初始索引 i 为参数，将它作为扫描的起始点。对于删除操作，FilterDown 被调用时所带参数为 0，因为替换值从堆尾被复制到根部。FilterDown 方法还被构造函数用来建立堆。

```
// 重排堆的实用函数;从下标 i 开始,互换双亲与孩子的值使从 i 开始的子树为堆
template <class T>
void Heap<T>::FilterDown (int i)
{
    int currentpos, childpos;
    T target;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 从 i 开始,并将其值赋给 target
    currentpos = i;
    target = hlist[i];
    // 计算左孩子的下标,开始向下扫描孩子,直到表结束
    childpos = 2 * i + 1;
    while (childpos < heapsize)          // 检查表是否结束
    {
        // 右孩子下标为 childpos + 1.置 childpos 为较小孩子的下标
        if ((childpos + 1 < heapsize) &&
                (hist[childpos + 1] < = hlist[childpos]))
            childpos = childpos + 1;
        // 双亲小于孩子,已经是堆,则退出
        if (target < = hlist[childpos])
            break;
        else
        {
         // 将较小孩子值移到双亲,此时该位置为空
         hlist[currentpos] = hlist[childpos];
         // 修改下标继续扫描
         currentpos = childpos;
         childpos = 2 * currentpos + 1;
        }
    }
    // 将 target 赋值给最新的空位置
    hlist[currentpos] = target;
}
```

公有方法 Delete 将根结点的值复制到临时变量中,并用堆中最后一个值替换根。heapsize 的值减小以后,FilterDown 对堆进行更新。然后,临时变量中的值被返回给客户程序。

```
// 返回堆中第一个元素值并修改堆.若对空堆进行删除,则产生出错信息并退出程序.
template < class T >
T Heap < T > ::Delete(void)
{
    T tempitem;
    // 检查堆是否为空
    if (heapsize = = 0)
        error("Heap empty");
    // 将根赋值给 tempitem;然后将根值换为堆中最后元素的值并将堆大小减 1
    tempitem = hlist[0];
    hlist[0] = hlist[heapsize - 1];
    heapsize - - ;
    // 调用 FilterDown 来生成堆的新根
    FilterDown(0);
    // 返回原根的值
    return tempitem;
}
```

堆化数组(Heapifying an array) Heap 类提供的构造函数以现有数组作为表并将其转化为堆。这一过程被称为"堆化"数组。该算法成功地将 FilterDown 方法应用于所有非

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

叶子结点。最后一个堆元素的索引为 n－1。其双亲位于索引

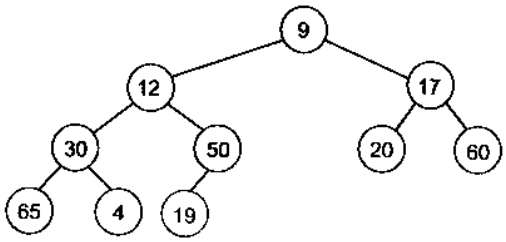
$$\text{currentpos} = \frac{(n-1)-1}{2} = \frac{n-2}{2}$$

该处定义了堆中最后一个非叶子结点。该双亲结点给出了堆化数组的起始索引，如果以 currentpos 到 0 范围内的索引值执行 FilterDown，就可以保证每个双亲结点都满足堆条件。作为一个例子，考虑下面的整数数组：

int A[0] = {9,12,17,30,50,20,60,65,4,49};

叶子结点索引:5,6,…,9

双亲结点索引:4,3,…,0

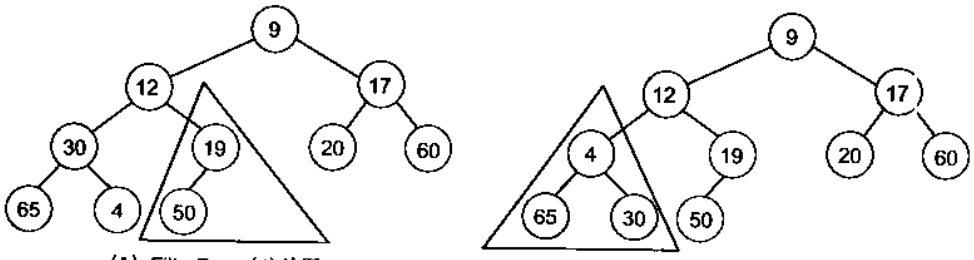


初始表

以下的系列图演示了“堆化”过程。对于所有 FilterDown 调用，受影响的子树被强调表示。

FilterDown(4)：值 H[4] = 50，大于其孩子 H[9] = 19，因此必须与其孩子交换(A)。

FilterDown(3)：值 H[3] = 30，大于其孩子 H[8] = 19，因此必须与其孩子交换(B)。



(A) FilterDown(4)放置50　　(B) FilterDown(3)放置30

在第 2 层，双亲 H[2] = 17 已经满足了堆条件。函数 FilterDown(2)不作任何更改：

FilterDown(1)：值 H[1] = 12 比其孩子 H[3] = 4 大，因此必须与其孩子交换(C)。

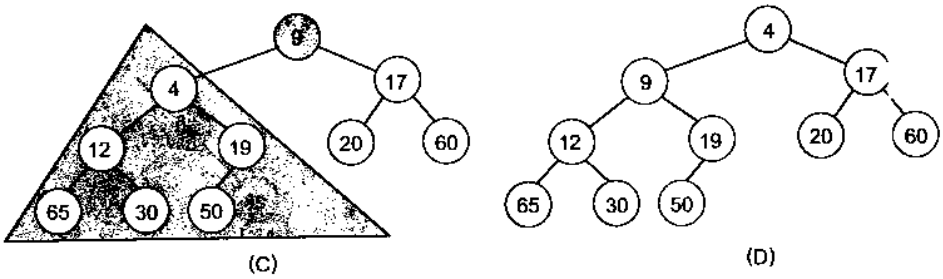
FilterDown(0)：整个过程在根结点处终止。值 H[0] = 9 必须与其孩子 H[1] 交换位置。最后得到的树是一个堆(D)。

构造函数

```
// 构造函数将数组转化为堆.数组及其大小作为参数传递
template <class T>
Heap<T>::Heap(T arr[], int n)
{
    int j, currentpos;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
    // 若 n < = 0,则数组大小非法;退出程序
    if (n < = 0)
       error("Bad list size.");
    // 将堆大小置为 n,则最大堆大小边为 n;并将数组 arr 赋给堆表
    maxheapsize = n;
    heapsize = n;
    hlist = arr;
    // 置 crrentpos 为最大的双亲结点下标;在范围 currentpos 到 0 之间循环调用
    // FilterDown
    currentpos = (heapsize - 2)/2
    while(currentpos > = 0)
    {
        // 以 hlist[currentpos]为根使其子树满足堆条件
        FilterDown(currentpos);
        currentpos - - ;
    }
    // 置 inArray 为真使堆不释放数组空间
    inArray = 1;
}
```

应用：堆排序

堆排序是一种非常有效的排序方法[算法复杂度为 $O(n \log_2 n)$]。该算法用到了以下事实，最小的元素位于根结点(索引 0)处，而 Delete 删除此值。为将堆排序应用于数组 A，以数组 A 为参数声明一个 Heap 对象。构造函数将数组 A 转换为堆。排序是通过重复删除 A[0]并将其插入到表尾位置 A[N-1]，A[N-2]，…，A[1]处而实施的。试回想一下，当表项被从堆中删除以后，前一个表尾元素变成了根结点处的替代值，它的位置不再是堆的一部分。我们尽可以将被删除项复制到该位置。堆顺序删除表中最小元素并将其存放到数组尾部。然后删除次最小元素并将其存放到次最后位置，等等，依此类推。数组 A 以降序排序。习题中将要求读者建立一个最大堆类，用它可以将数组按升序排序。

下列各步实现对 5 个元素的数组 A 的排序：

int A[ ] = {50,20,75,35,25}

因为唯一一个剩下的元素在根中，所以数组已排好序：A = 75 50 35 25 20。

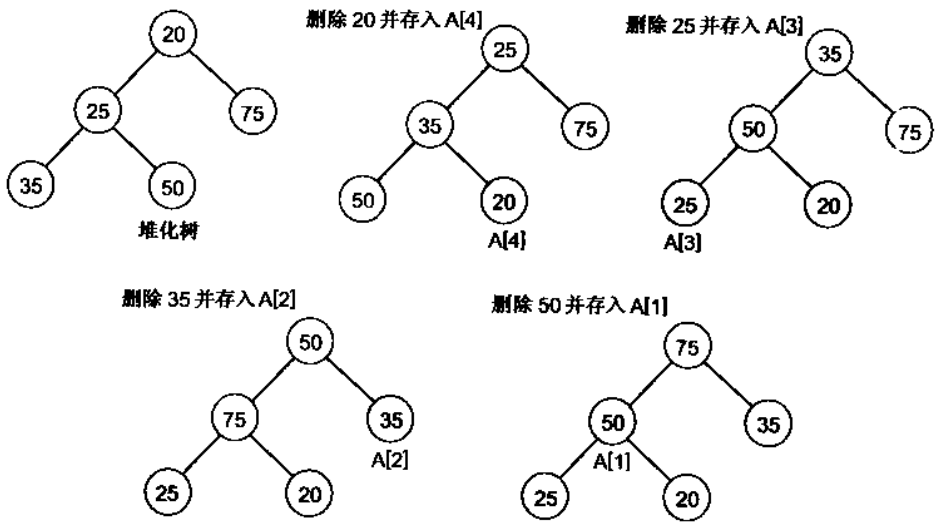
以下是堆排序算法的一种具体实现。函数 HeapSort 在文件"heapsort.h"中给出。

HeapSort

```
// 引入类 Heap
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
# include "heap.h"
// 将数组 A 按降序排列
template < class T >
void HeapSort (T A[], int n)
{
    // 构造函数将 A 转换为堆
    Heap < T > H(A,n);
    T elt;
    // 迭代装入元素 A[n - 1],…,A[i]
    for(int i = n - 1; i > = 1; i - - )
    {
     // 从堆中删除最小的元素并存入 A[i]
     elt = H.HDelete();
     A[i] = elt;
    }
}
```

堆排序的算法复杂度 一个 n 元素的数组对应于一棵深度为 $k = \log_2 n$ 的完全二叉树。堆化数组的初始阶段总是需要 $n/2$ 次 FilterDown 操作。每一个这种操作都需要不超过 k 次的比较。在排序的第 2 阶段，FilterDown 操作要执行 $n-1$ 次。在最坏的情况下，操作需要 k 次比较。两个阶段合并起来，最坏情况下堆排序的(时间)复杂度为：

$$k * \frac{n}{2} + *(n-1) = k * (\frac{3n}{2} - 1)$$

$$= \log_2 n * (\frac{3n}{2} - 1)$$

其量级为 $O(n\log_2 n)$。

堆排序并不需要额外存储空间，因为排序是在原地进行的。竞赛排序的算法复杂度虽然也是 $O(n\log_2 n)$，但它需要为具有 $2^{k+1}-1$ 个结点的基于数组的树分配空间，其中 k 是满足 $n \leqslant 2^k$ 的最小整数。有些复杂度为 $O(n\log_2 n)$ 的排序在最坏的情况下性能为

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

$O(n^2)$。13.7 节中的树排序就是一个例子。作为对照，堆排序是 $O(n log_2 n)$量级，而不管其数据的初始分布如何。

程序 13.2　几种排序比较

用 HeapSort 函数对 2000 个随机整数的数组 A 排序。为了比较，数组 B 和 C 被赋予同样的元素值。对它们分别调用 TournamentSort 和 ExchangeSort 函数。ExchangeSort 函数在文件"arrsort.h"中给出。用函数 TickCount 对排序进行计时，它以 1/60 秒为时间单位，返回自系统启动以后所经过的时间。交换排序的复杂度为 $O(n^2)$，这使得我们在较慢的算法与具有 $O(n\log_2 n)$量级的竞赛和堆排序之间可以作一生动的比较。函数代码未包括在程序清单中，而是在补充程序文件"prg13_2.cpp"中给出。

```
# include < iostream.h >
# include "random.h"
# include "arrsort.h"
# include "toursort.h"
# include "heapsort.h"
# include "ticks.h"                // 用于 TickCount
enum SortType {heap, tournament, exchange};
void TimeSort(int * A, int n, char * sortName, SortType sort)
{
    long tcount;
    // TickCount 为一依赖系统函数.它返回从起动开始迄今为止的 1/60 秒的个数
    cout < < "Sorting with" < < sortName < < ':' < < endl;
    // 开始计时,然后对数组 A 排序.以 1/60 秒为单位计算排序时间并赋给 tcount
    tcount = TickCount ();
    switch(sort)
    {
        case heap:              HeapSort(A,n);
                                break;
        case tournament:        TournamentSort(A,n);
                                break;
        case exchange:          ExchangeSort(A,n);
                                break;
    }
    tcount = TickCount() - tcount;
    // 输出排好序后的数组的前 5 个及后 5 个元素
    for (int i = 0; i < 5; i + + )
        cout < < A[i]< < "  ";
    cout < < "...";
    for (i = n - 5; i < n; i + + )
        cout < < A[i] < < "  ";
    cout < < endl;
    cout < < sortName < < "time is" < < tcount < < "\n\n";
}
void main(void)
{
    // 指向数组 A,B,C 的指针
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    int *A, *B, *C;
    RandomNumber rnd;
    // 动态申请数组内存
    A = new int [2000];
    B = new int [2000];
    C = new int [2000];
    // 分别往 3 个数组中装入 2000 个相同的随机数
    for(int i = 0; i < 2000; i++)
        A[i] = B[i] = C[i] = rnd.Random(10000);
    TimeSort(A,2000,"Heap sort", heap);
    delete [] A;
    // 用竞赛排序重排数组
    TimeSort(B, 2000, "Tournament sort", tournament);
    delete [] B;
    // 用交换排序重排数组
    TimeSort(C, 2000, "Exchange sort", exchange);
    delete [] C;
}
/*
<程序 13.2 运行结果>
Sorting with Heap sort:
9999  9996  9996  9995  9990  ...  11  10  9  6  3
Heap sort time is 16
Sorting with Tournament sort:
3  6  9  10  11  ...  9990  9995  9996  9996  9999
Tournament sort time is 36
Sorting with Exchange sort:
3  6  9  10  11  ...  9990  9995  9996  9996  9999
Exchange sort time is 818
*/
```

13.4 优先级队列

第 5 章介绍了优先级队列并将其用于事件驱动模拟研究。客户程序可以访问插入和删除运算 PQueue 对象的支撑表(underlying list)。

这一节我们将用堆实现优先级队列。因为我们用的是最小堆,所以假定表项按优先级从小到大的顺序排列。从堆中执行一次删除操作返回的是优先级队列中的最小(最高优先级)元素。堆实现法极大地提高了 PQDelete 方法的效率,因为它仅需进行 $O(\log_2 n)$ 次比较。而与之相比,数组实现的比较次数为 $O(n)$。

本节最后我们将设计一个过滤程序,将数据项数组转换为长归并段。使用优先级队列的过滤程序在对文件中的大块数据排序时可显著提高归并排序的效率。第 14 章将对此进行专门讨论。

PQueue 类说明(堆版本)

声明

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
# include "heap.h"
template < class T >
class PQueue
{
    private:
      // 存放队列的堆
      Heap < T >  * ptrHeap;
    public:
        // 构造函数
           PQueue (int sz);
           // 修改优先级队列的函数
           void PQInsert(const T& item);
           T PQDelete(void);
           void ClearPQ(void);
           // 检测优先级队列的函数
           int PQEmpty(void) cont;
           int PQFull(void) const;
           int PQLength(void) const;
};
```

说明

构造函数带一个尺寸参数，用它来动态分配堆结构 ptrHeap。要实现此方法，只需调用堆类中的相应方法即可。例如，PQDelete 使用堆删除方法。PQueue 的实现在文件“pqueue.h”中。

```
// 通过删除对应堆中的根来删除优先级队列的第一个元素，返回被删除的值
template < class T >
T PQueue < T > ::PQDelete(void)
{
    return ptrHeap- > Delete();
}
```

应用：长归并段

归并排序是对大型外部文件进行排序时的主要算法。如果数据被过滤或预处理到长归并段中，算法的效率会有所提高。第 12 章中我们已见过这类过滤程序，它一次读取 k 个数据元素并对其进行排序。这样所得到的数据是具有最小长度 k 的归并段。本应用中，我们使用 k 个元素的优先级队列并生成长度通常显著大于 k 的归并段。算法从原始表 A 中读取表项并将它们传递到优先级队列过滤程序。然后元素以归并段形式返回到表中。

我们用一个例子来说明该算法，例子中假定数组 A 有 12 个整数，优先级队列 PQ1 是 k = 4 个元素的过滤器。优先级队列中存放的是最终将出现在当前归并段的元素。第 2 个优先级队列 PQ2 中存放的是下一个归并段的元素。我们用两个索引扫描数组。变量 loadIndex 标识当前正被读取的表项。变量 currIndex 标识从 PQ1 中释放并返回到数组的最后一个表项。例子中，数组 A 初始时被分为 6 个归并段，最长的一个含 3 个元素。

A = [13]　[6　61　96]　[26]　[1　72　91]　[37]　[25　97]　[21]

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

过滤完元素后，归并段会变为3个，最长的一个有7个元素。

首先我们将表中元素A[0]到A[3]装入PQ1。既然优先级队列按从小到大的顺序释放元素，我们也就有了至少对4个元素排序使之成为一个归并段的工具。我们甚至可以做得更好一些。从PQ1中删除第1项(最小值)并将其赋值给A[currIndex]=A[0]=6。这样PQ1中就有了一个空位，第1个归并段开始形成。因A中头4个表项已被复制到PQ1中，故我们从loadIndex=4处继续处理A的元素。在此过程中的每一步，都比较A[loadIndex]与A[currIndex]。如果loadIndex对应的表项更大，它将最终出现在当前归并段中，因而被存储到PQ1中。否则它将出现在下一个归并段，因而也就存储在PQ2中。我们将描述本例中每一项上所发生的动作。处理完一项以后，我们用以下格式列出A中重新装入的项、有待读入的项以及两个优先级队列的内容。

A:　〈装入归并段中的项〉　A[loadIndex]…〈剩余项〉
PQ1:　〈存放当前归并段〉　PQ2:〈存放下一个归并段〉

一步一步执行：

表项A[4]=26>A[currIndex]=6。将26存到PQ1中，将13从PQ1中释放到表中A[1]处。

A:　6　13　　A[5]到A[11]:1　72　91　37　25　97　21
PQ1:　61　96　26　　PQ2:〈空〉

表项A[5]=1<A[currIndex]=13。因而，值1是下一个归并段的一部分。将1存到PQ2中，将26从PQ1中释放到表中A[2]处。

A:　6　13　26　　A[6]到A[11]:72　91　37　25　97　21
PQ1:　61　96　　PQ2:1

表项A[6]=72大于当前归并段中的表项26。将它存到PQ1并释放61。类似地，下一项91进入PQ1并将72释放到当前归并段的currIndex=4处。

A:　6　13　26　61　72　　A[8]到A[11]:37　25　97　21
PQ1:　91　96　　PQ2:1

表项A[8]=37与A[9]=25都小于72，因而属于下一个归并段。它们被存到PQ2，同时从PQ1中释放两项到数组中，队列随之变空。

A:　6　13　26　61　72　91　96　　A[10]到A[11]:97　21
PQ1:　〈空〉　　PQ2:1　37　25

我们已经完成当前归并段，可以开始下一个归并段了。将PQ2中的元素复制到PQ1中并从新填满的队列PQ1中释放最小的元素。本例中，从PQ1中释放1并开始下一个归并段。

A:　6　13　26　61　72　91　96　1　　A[10]到A[11]:97　21
PQ1:　25　37　　PQ2:〈空〉

表项A[10]=97>1且已存到PQ1中。然后我们从PQ1中释放最小值25。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

A:　6　13　26　61　72　91　96　1　25　A[11]:21

PQ1:　37　97　　PQ2:〈空〉

表项 A[11]=21<25,必须等待下一个归并段。将它存到 PQ2 中并释放 37。

A:　6　13　26　61　72　91　96　1　25　37　〈表扫描完成〉

PQ1:　97　　PQ2:21

对原始表的扫描已经完成。将 PQ1 中的所有元素释放到当前归并段,PQ2 的所有元素释放到下一个归并段。

归并段 1:　6　13　26　61　72　91　96

归并段 2:　1　25　37　97

归并段 3:　21

归并段算法　我们用 LongRunFilter 类实现长归并段算法。其私有成员包括数组以及存放当前和下一个归并段的两个优先级队列。构造函数将数组联编到类对象中并建立与其关联的优先级队列。算法主要由私有方法 LoadPQ 和 CopyPQ 实现,前者将元素从数组插入到 PQ1 中,后者将元素从 PQ2 中复制到 PQ1 中。

声明

```
template <class T>
class LongRunFilter
{
    private:
        // 过滤器中用到的关键指针。A 为表,PQ1 和 PQ2 为两个优先级队列
        T *A;
        PQueue<T>     *PQ1, *PQ2;
        int loadIndex;
        // 给定数组和优先级队列的大小
        int arraySize;
        int filterSize;
        // 从优先级队列 PQ2 拷贝元素到 PQ1
        void CopyPQ(void);
        // 从数组 A 中装入元素到优先级队列 PQ1
        void LoadPQ (void);
    public:
        // 构造函数/析构函数
        LongRunFilter(T arr[], int n, int sz);
        ~LongRunFilter(void);
        // 产生长归并段
        void LoadRuns(void);
        // 评估归并段
        void PrintRuns(void) const;
        int CountRuns(void) const;
};
```

讨论

构造函数初始化数据成员并将数据成员从数组中装入 PQ1 中,这样设置了第 1 个归

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

并段的数据项。方法 LongRuns 是将元素从数组过滤到归并段中的主要算法。

方法 PrintRuns 和 CountRuns 可以说明算法。我们用它们比较调用 LongRuns 前后数组中的归并段。

```
// 扫描数组 A,用过滤器归并元素并产生长归并段
template < class T >
void LongRunFilter < T > ::LoadRuns(void)
{
    T value;
    int currIndex; = 0;
    if (filterSize = = 0)
        return;
    // 从将最小元素从 PQ1 装入到 A 开始
    A[currIndex] = PQ1 - > PQDelete();
    // 将 A 中元素装入 PQ1 后,看 A 中剩余的元素
    while (loadIndex < arraySize)
    {
        // 表中下一元素
        value = A[loadIndex + + ];
        // 若该元素 > = A[currIndex],它属于当前归前段,将这放入 PQ1 中,否则,它将到下
        // 一归并段 PQ2 中
        if (A[currIndex] < = value)
           PQ1 - > PQInsert(value);
        else
           PQ2 - > PQInsert(value);
        // 一旦 PQ1 为空,当前归并段完成,将 PQ2 元素拷入 PQ1 中开始下一归并段
        if (PQ1 - > PQEmpty())
            CopyPQ();
        // 从 PQ1 中抽取元素放入归并段
        if (! PQ1 - > PQEmpty())
            A[ + + currIndex] = PQ1 - > PQDelete();
    }
    // 清除当前归并段元素和下一归并段元素
    while (! PQ1 - > PQEmpty())
        A[ + + currIndex] = PQ1 - > PQDelete();
    while (! PQ2 - > PQEmpty())
        A[ + + currIndex] = PQ2 - > PQDelete();
}
```

程序 13.3　长归并段

本程序演示了长归并过滤程序的使用。第 1 个例子用了一个 15 个元素的数组,过滤器用的是 4 个元素的优先级队列。输出包括在调用过滤器前后的归并表。另一个更现实一点的例子用的是 10 000 个元素的数组,过滤器则分别用 5、50 和 500 个元素的优先级队列。我们打印出每一种情况所得到的归并段的个数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
#include <iostream.h>
// 引入随机数发生器和过滤器类
#include "random.h"
#include "longrun.h"
// 将数组 A 拷贝到数组 B
void CopyArray(int A[], int B[], int n)
{
    for (int i = 0; i < n; i++)
        B[i] = A[i];
}
void main()
{
    // 15 个元素构成的整数数组来演示过滤器
    int demoArray[15];
    // 10000 个元素的数组来统计归并段
    int *A = new int [10000], *B = new int[10000];
    RandomNumber rnd;

    // 过滤器大小将分别为 5,50 和 500
    int i, filterSize = 5;
    // 产生 15 个随机整数,建立过滤器
    for (i = 0; i < 15; i++)
        demoArray[i] = rnd.Random(100);
    LongRunFilter<int> F(demoArray, 15, 4);
    // 输出用 4 元素过滤器产生长归并段前后的表
    cout << "Raw List" << endl;
    F.PrintRuns();
    cout << endl;
    F.LoadRuns();
    cout << "Filtered List" << endl;
    F.PrintRuns();
    cout << endl

    // 用 10000 个随机整数初始化数组
    for (i = 0; i < 10000; i++)
        A[i] = rnd.Random(25000);
    cout << "Runs with 3 filters" << endl;
    // 建立原始表并统计归并段数目
    LongRunFilter<int> LR(A, 10000, 0);
    cout << "Number of runs in initial 10000 element array is"
         << LR.CountRuns() << endl;
    // 分别用大小为 5,50,500 的过滤器归并并统计归并段
    for (i = 0; i < 3; i++)
    {
        CopyArray(A,B,10000);
        LongRunFilter<int> LR(B, 10000, filterSize);
        // 产生长归并段
        LR.LoadRuns();
        cout << " Runs with a filter of " << filterSize
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            << " is " << LR.CountRuns() << endl;
        // 改变过滤器大小
        filterSize * = 10;
    }
}
/*
<程序 13.5 运行结果>
Raw List
36
22  79
26  84
44  88
44  66  81
19  86
40
2   47
Filtered List
22  26  36  44  44  66  79  81  84  86  88
2   19  40  47
Runs with 3 filters
Number of runs in initial 100000 element array is 5077
  Runs with a filter of 5 is 991
  Runs with a filter of 50 is 101
  Runs with a filter of 500 is 11
*/
```

13.5 AVL 树

二叉树是为快速访问数据而设计的。理想情况下，树是平衡分布的，其高度为 $O(\log_2 n)$。但对有些数据来说，二叉搜索树也可能是退化的。那时树高将是 $O(n)$，访问数据的速度将明显变慢。本节我们将设计一个修正的树类，它具有二叉搜索树的威力但又可避免最坏的情况。我们要设计的是各结点具有平衡高度的 AVL 树。也就是说，在 AVL 树中任一结点的两个子树的高度差最多为 1。

在 AVL 树类中，新版本的插入和删除方法可以保证结点在高度上保持平衡。图 13.3 中示意了同一数组对应的 AVL 树及二叉搜索树两种等价表示法。(A)中的一对表示的是其元素按从小到大顺序排列的只有 5 个元素的简单数组。第 2 个例子中画出了两棵其元素取自数组 B 的树。(A)中的二叉搜索树的高度为 5，而 AVL 树的高度为 2。一般来说，AVL 树的高度绝不会超过 $O(\log_2 n)$，因此当需要快速访问元素时，AVL 树是一种极具威力的存储方式。

A[5] = {1,2,3,4,5}　　A[8] = {20,30,80,40,10,60,50,70}

这一节我们使用第 11 章中由树结点建立搜索树的方法。我们先定义 AVLTreeNode 类然后再使用这些对象设计一个 AVLTree 类。我们所关注的焦点在 AVL 树方法 Insert 和 Delete 上。这些方法的算法需要仔细设计以确保新树中的每个结点保持高度上的平衡。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

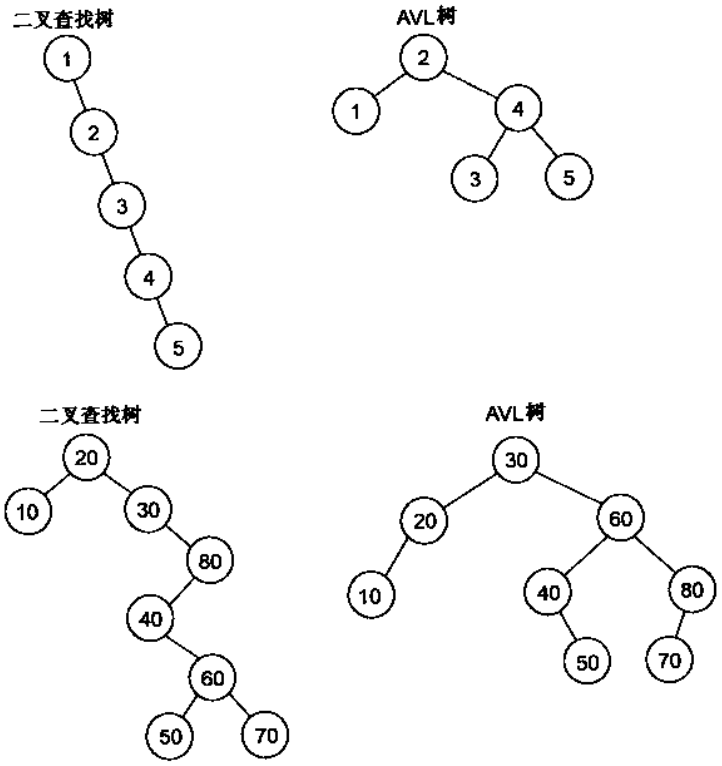


图 13.3　同一数组的 AVL 和二叉查找树表示

AVL 树结点

AVL 树的表示与二叉搜索树类似，其操作基本相同，但 Insert 和 Delete 方法除外，因为它们必须不断监控结点的左右子树的相对高度。为此我们扩展 TreeNode 对象的定义，使之包括一个 balanceFactor 域。

left	data	balanceFactor	right

AVLTreeNode

这个域的值为左右子树的高度差：

```
balanceFactor = height(right subtree) - height(left subtree)
```

若 balanceFactor 为负，则结点"左侧偏重"，因为左子树的高度大于右子树的高度。若 balanceFactor 为正值，则结点"右侧偏重"。具有平衡高度的结点的 balanceFactor 值为 0 值。在 AVL 树中，balanceFactor 的值必须在 -1 到 1 之间。

图 13.4 中画出了 3 棵 AVL 树，每棵树的结点上都有标记 -1，0 或 +1 以表示左右子树的相对大小。

-1：　左子树比右子树高度大 1。

0：　左右子树高度相等。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

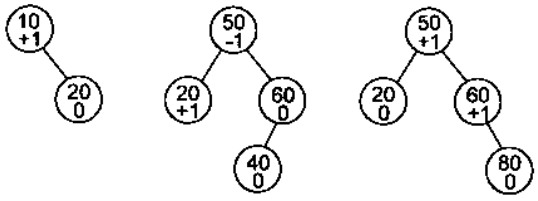


图 13.4　AVL 树

1：　右子树比左子树高度大 1。

使用继承工具使我们可以从 TreeNode 基类中派生 AVLTreeNode 类。AVLTreeNode 对象继承 TreeNode 的域并增加一个 balanceFactor 域。TreeNode 中的两个数据成员 left 和 right 是保护的，因此 AVLTreeNode 或其他派生类可以直接访问它们。AVLTreeNode 类在补充程序文件"avltree.h"中给出。

AVLTreeNode 类说明

声明

```
// 从类 TreeNode 中继承
template < class T >
class AVLTreeNode: public TreeNode < T >
{
    private:
      // AVLTreeNode 所需的附加数据成员
      int balanceFactor;
      // 用于类 AVLTree 的方法，允许不做类型转换将 AVLTree 指针赋值给树结点指针
      AVLTreeNode < T > * & Left(void);
      AVLTreeNode < T > * & Right(void);

    public:
      // 构造函数
      AVLTreeNode (const T& item, AVLTreeNode < T > * lptr = NULL,
                   AVTreeNode < T > * rptr = NULL, int balfac = 0);
      // 以 AVLTreeNode 指针形式返回左/右 TreeNode 指针
      AVLTreeNode < T > * Left(void) const;
      AVLTreeNode < T > * Right(void) const;
      // 访问新数据域的方法
      int GetBalanceFactor(void);
      // 用于访问左、右子树的 AVLTree 方法
      friend class AVLTree < T >;
};
```

讨论

数据成员 balanceFactor 是私有的，因为只有 AVL 的插入和删除操作才更新此值。

构造函数中，参数包括 TreeNode 支撑结构的数据以及缺省参数 balfac = 0。

客户程序可以用 Left 和 Right 访问指针域。这些方法需要重新定义，因为返回值是指

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

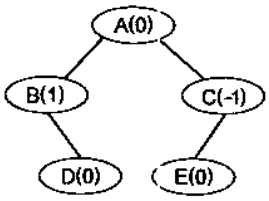
www.Linuxidc.com

向较大的 AVLTreeNode 结构的指针。

声明虚析构函数的一般性的理由在 12.3 节中已讨论过。在我们的设计中，我们是由 BinSTree 派生 AVLTree 类，这样也就复用了基类析构函数和 ClearList。这些基类方法通过执行 delete 运算符注销结点。在每一种情况下，指针都是指向 AVLTreeNode 对象，而不是 TreeNode 对象。如果结点基类 TreeNode 中的析构函数为虚，调用 delete 时会用到动态联编从而删除 AVLTreeNode 对象。

例

```
    AVLTreeNode < char >  * root;        // AVL 树的根
    // 函数创建如下 AVLTree 每一结点被赋了 balanceFactor 值
void MakeAVLCharTree(AVLTreeNode < char > * &root)
{
    AVLTreeNode < char >  * a, * b, * c, * d, * e;
    e = new AVLTreeNode < char > ('E', NULL, NULL, 0);
    d = new AVLTreeNode < char > ('D', NULL, NULL, 0);
    c = new AVLTreeNode < char > ('C', e, NULL, - 1);
    b = new AVLTreeNode < char > ('B', NULL, d, 1);
    a = new AVLTreeNode < char > ('A', b, c, 0);
    root = a;
}
```



AVLTreeNode 类的实现　AVLTreeNode 类的构造函数调用基类构造函数并初始化 balanceFactor。

```
// 构造函数;初始化 balanceFactor 及基类.用缺省指针值 NULL 将结点初始化为叶子结点
template < class T >
        AVLTreeNode < T > ::AVLTreeNode (const T& item,
    AVLTreeNode < T >  * lptr, AVLTtreeNode < T >  * rptr, int balfac):
TreeNode < T > (item,lptr,rptr), balanceFactor(balfac)
{}
```

AVLTreeNode 类的方法 Left 和 Right 可以简化客户程序对域的访问。试图用基类方法 Left 访问左孩子时会得到一个指向 TreeNode 的返回指针。这就需要进行类型转换以返回一个指向大的结点结构的指针。下列语句示意了这一问题。

```
AVLTreeNode < T >  * p, * q;
q = p - > Left();                    // 非法操作
q = (AVLTreeNode < T >  * )p - > Left(); // 需类型转换
```

我们不用强制进行重复的指针类型转换，而是定义 AVLTreeNode 类的 Left 和 Right 方法并返回 AVLTreeNode 指针值。

```
// 将左子树转换为 AVLTreeNode 指针后返回
template < class T >
AVLTreeNode < T > * AVLTreeNode < T > ::Left(void)
{
    return (AVLTreeNode < T >  * )left;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

13.6 AVL 树类

AVL 树提供的表结构与二叉搜索树类似，所不同的是它增加了树在插入和删除操作以后保持高度上的平衡这一条件。既然 AVL 树是扩展的二叉搜索树，我们就用继承技术从 BinSTree 类派生 AVLTree 类。

Insert 和 Delete 方法必须被重写以满足 AVL 条件。此外，我们在派生类中还定义了复制构造函数和重载的赋值运算符，因为我们要建立具有大结点结构的树。

AVLTree 类说明

声明

```
// 表明结点平衡因子的常量
const int leftheavy = -1;
const int balanced = 0;
const int rightheavy = 1;
// 从搜索树中派生
template <class T>
class AVLTree: public BinSTree<T>
{
    private:
        // 申请内存
        AVLTreeNode<T> *GetAVLTreeNode(const T& item,
                AVLTreeNode<T> *lptr, AVLTreeNode<T> *rptr);
        // 用于复制构造函数和赋值运算
        AVLTreeNode<T> *CopyTree(AVLTreeNode<T> *t);
        // 供 Insert 和 Delete 方法在结点加入子树或从子树中删除时重建 AVL 树
        void SingleRotateLeft (AVLTreeNode<T> * &p);
        void SingleRotateRight (AVLTreeNode<T> * &p);
        void DoubleRotateLeft (AVLTreeNode<T> * &p);
        void DoubleRotateRight (AVLTreeNode<T> * &p);
        void UpdateLeftTree (AVLTreeNode<T> * &tree,
                int &reviseBalanceFactor);
        void UpdateRightTree (AVLTreeNode<T> * &tree,
                int &reviseBalanceFactor);
        // AVL 树的 Insert 和 Delete 方法
        void AVLInsert(AVLTreeNode<T> * &tree,
                AVLTreeNode<T> * newNode, int &reviseBalanceFactor);
        void AVLDelete(AVLTreeNode<T> * &tree,
                AVLTreeNode<T> * newNode, int &reviseBalanceFactor);
    public:
        // 构造函数，析构函数
        AVLTree(void);
        AVLTree(const AVLTree<T>& tree);
        // 赋值运算符
        AVLTree(T)& operator= (const AVLTree<T>& tree);
        // 标准的表处理函数
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        virtual void Insert(const T& item);
        virtual void Delete(const T& item);
};
```

说明

常数 leftheavy,balanced 和 rightheavy 供插入和删除算法使用以描述结点的平衡状态。

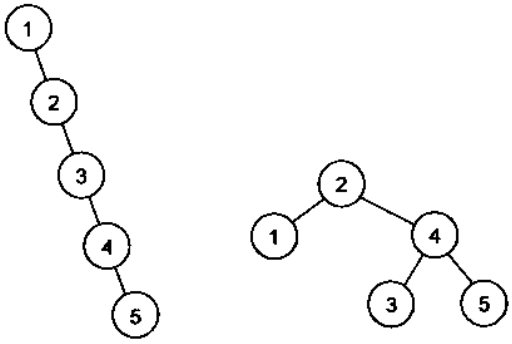
方法 GetAVLTreeNode 负责为类分配结点。缺省情况下,新结点的 balanceFactor 值被设置为 0。

类中定义了新的 CopyTree 函数以用于复制构造函数和重载的赋值运算符。虽然算法与 BinSTree 中的 CopyTree 相同,但函数在建立新树时生成的是较大的 AVLTreeNode 结点。

函数 AVLInsert 和 AVLDelete 分别实现插入和删除方法。诸如 SingleRotateLeft 一类的私有方法被用于实现 AVLInsert 和 AVLDelete。我们将公有方法 Insert 和 Delete 声明为虚函数以重写基类中的那些方法。除了这些树更新方法以外,我们从 BinSTree 中继承其他所有搜索树操作。

例

```
AVLTree < int > vltree;          // 整型 AVLTree 表
BinSTree < int > bintree;        // 整型 BinSTree 表
for (int i = 1; i < = 5; i + + )
{
    bintree.Insert(i);           // 创建树(A)及树(B)
    avltree.Insert(i);
}
```



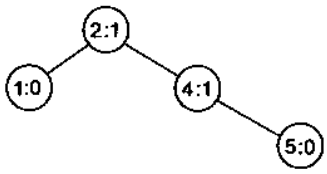

```
avltree.Delete(3);               // 从 AVL 树中删除了
// 函数 AVLPrintVTree 和第 11 章中树的垂直打印函数相似,将扩展为同时输出数据值和
// 平衡因子。树(C)为 AVLTree 树(B)在结点上被删除后的情况.AVLPrintVTree 在文件
// "avltree.h"中
AVLPrintTree((AVLTreeNode < int > * )avltree.GetRoot(), 0);
```

AVLTree 对象的内存分配

AVLTree 类由 BinSTree 派生得到并继承了它的大部分操作。当需要建立较大的 AVLTreeNode 对象时,我们就设计单独的内存分配和复制方法,这正如 GetAVLTreeNode 所示。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



(C)

```
// 申请 AVLTreeNode 结点,若出错则退出程序
template < class T >
AVLTreeNode < T >  * AVLTree < T > ::GetAVLTreeNode(const T& item,
                AVLTreeNode < T >  * lptr, AVLTreeNode < T >  * rptr)
{
    AVLTreeNode < T >  * p;
    p = new AVLTreeNode < T > (item, lptr, rptr);
    if (p = = NULL)
    {
        cerr < < "Memory allocation failure!" < < endl;
        exit(1);
    }
    return p;
}
```

用基类方法足以删除较大的 AVLTreeNode 对象。BinSTree 类中的 DeleteTree 方法利用了 TreeNode 类中的虚析构函数。

AVLTree 类的 Insert 方法　AVL 树的威力在于其能够建立和维护具有平衡高度的树。这种威力是靠 AVL 插入和删除算法实现的。我们在 AVLTree 类中给出了重写其基类 BinSTree 类中相同操作的 Insert 方法。插入的实际实现用的是递归方法 AVLInsert 来存放新元素。我们先给出 Insert 的 C + + 代码,然后再把注意力集中到实现算法的递归方法 AVLInsert 上,它们是由 AVL 树的发明人 Adelson-Velskii 和 Landis 提出的。

```
template < class T >
void AVLTree < T > ::Insert(const T& item)
{
    // 定义指向 AVL 树结点的指针;用基类方法 GetRoot 得到结点指针后转换成 AVLTree 指针
    // 并赋值给根
    AVLTreeNode < T >  * treeRoot = (VLTreeNode < T >  * )GetRoot(),
                       * newNode;
    // 供 AVLInsert 重新平衡结点的标志
    int reviseBalanceFactor = 0;
    // 得到一个指针域为空的新的 AVL 树结点
    newNode = GetAVLTreeNode(item, NULL, NULL);
    // 调用递归函数实际插入元素
    AVLInsert(treeRoot, newNode, revise BalanceFactor);
    // 赋新值给基类中的数据成员
    root = treeRoot;
    current = newNode;
    size + + ;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

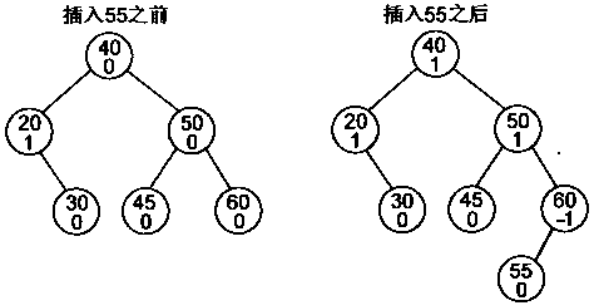
www.Linuxidc.com

插入算法的核心是递归方法 AVLInsert。与 BinSTree 中的同类方法类似，若表项 < 结点值则它遍历左子树；若表项 ≥ 结点值则遍历右子树。此私有函数的参数包括用来记录当前扫描结点的 tree、要插入到树中的新结点以及名为 reviseBalanceFactor 的标志。当我们扫描结点的左或右子树时，标志会提醒我们子树中的 balanceFactor 是否有变化。若有，必须检查是否保持了 AVL 高度上的平衡。如果新插入的结点破坏了树的平衡，扰乱了平衡因子，必须重建 AVL 平衡。我们用一系列的例子说明该算法。

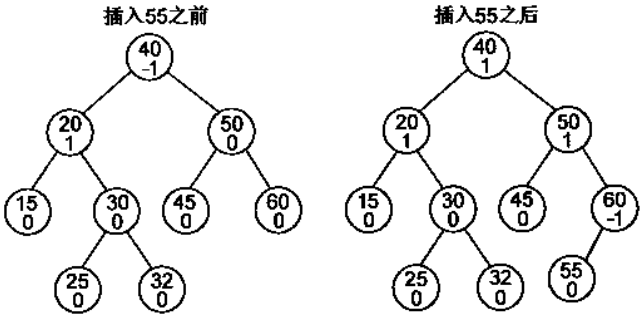
AVL 的插入算法 插入过程与二叉搜索树中所用的类似。我们沿左右孩子方向进行递归扫描直到识别出空子树为止，然后暂时将新结点插入到那个位置。在此过程中，我们访问从根到新表项的搜索路径上的每一个结点。

既然整个过程是递归的，我们就可以以相反的顺序访问结点并在知道新表项加到它的一棵子树中所产生的影响以后更新双亲结点中的平衡因子。对于搜索路径上的每一个结点，都要确定更新是否必须。我们面临 3 种可能的情况。在前两种情况中，结点保持 AVL 平衡，无需重构子树，仅需更新结点的 balanceFactor。第 3 种情况导致树的不平衡。这就需要对结点进行 1 次或 2 次旋转以重新平衡树。

情况 1：搜索路径上的结点初始时是平衡的(balanceFactor = 0)。将新表项加入到子树中以后，结点变为偏重于左侧或右侧，这取决于它的哪棵子树中存放了新表项。如果表项被存到左子树中则 balanceFactor 改为 -1；若存到右子树中则其值改为 1。例如，路径 40—50—60 上的每一个结点是初始平衡的。在插入 55 以后，balanceFactor 就发生了变化。



情况 2：路径上有某结点偏重于左或右子树，新表项被存放到其他(更轻的)子树中。结点随后得到平衡。例如，比较插入 55 前后树的状态。



情况 3：路径上的结点偏重于左或右子树，新表项被放置到同一(较重的)子树上。所

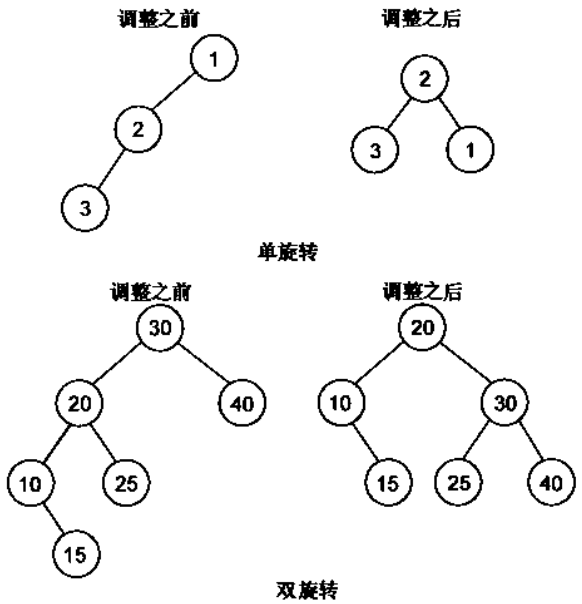
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

得到的结点违反 AVL 条件，因为 balanceFactor 超出了 -1 至 1 的范围。算法指引我们旋转结点以恢复高度上的平衡。

我们用图和例子来说明这些情况。假设树偏重于左侧，我们调用向右旋转函数进行调整。树偏重于右侧时的情况则正好与之相反。

下图示意了动作。在设计旋转算法时还要考虑更多的细节。



AVLInsert 方法 当沿搜索路径遍历以插入结点时，递归方法识别 3 种结点更新情况。当情况 3 发生时，AVL 条件被违反，我们不得不重新平衡结点。具体操作由函数 UpdateLeftTree 和 UpdateRightTree 实现。

```
template <class T>
void AVLTree<T>::AVLInsert(AVLTreeNode<T> * &tree,
                AVLTreeNode<T> * newNode, int& reviseBalance Factor)
{
    // 是否需修改结点的 balanceFactor 值的标志
    int rebalanceCurrNode;
    // 扫描到空子树;此时应插入新结点
    if (tree = = NULL)
    {
        // 修改双亲结点使其指向新结点
        tree = newNode;
        // 将新结点的 balanceFactor 赋值为 0
        tree->balanceFactor = balanced;
        // 广播信息; balanceFactor 值被改变
        reviseBalanceFactor = 1;
    }
    // 若新结点的数据值<当前数据值,则递归遍历左子树
    else if (newNode->data < tree->data)
    {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        AVLInsert(tree->Left(), newNode, rebalanceCurrNode);
        // 检查是否应修改 balanceFactor 值
        if (rebalanceCurrNode)
        {
            // 从左偏重的子树往左,将违背 AVL 条件,进行旋转(情况 3)
            if (tree->balanceFactor == leftheavy)
                UpdateLeftTree(tree, reviseBalanceFactor);
            // 从平衡结点往左,往左子树增加结点,满足 AVL 条件(情况 1)
            else if (tree->balanceFactor == balanced)
            {
                tree->balanceFactor = leftheavy;
                reviseBalanceFactor = 1;
            }
            // 从右偏重子树往左,将产生平衡子树,满足 AVL 条件(情况 2)
            else
            {
                tree->balanceFactor = balanced;
                reviseBalanceFactor = 0;
            }
        }
        else
            // 不需平衡此结点,也不用平衡上结点
        reviseBalanceFactor = 0;
    }
    // 否则,递归遍历右子树
    else
    {
        AVLInsert(tree->Right(), newNode, rebalanceCurrNode);
        // 检查是否应该修改 balanceFactor 值
        if (rebalanceCurrNode)
        {
            // 从左偏重子树往右,将平衡结点,满足 AVL 条件(情况 2)
            if (tree->balanceFactor == leftheavy)
            {
                // 扫描右子树.结点左偏重,则将成为平衡结点
                tree->balanceFactor = balanced;
                reviseBalanceFactor = 0;
            }
            // 从平衡子树往右,将产生右偏重结点,满足 AVL 条件(情况 1)
            else if (tree->balanceFactor == balanced)
            {
                // 结点原为平衡;将成为右偏重
                tree->balanceFactor = rightheavy;
                reviseBalanceFactor = 1;
            }
            // 从右偏重结点往右,将违背 AVL 条件,应进行旋转(情况 3)
            else
              UpdateRightTree(tree, reviseBalanceFactor);
        }
        else
            reviseBalanceFactor = 0;
    }
}
```

AVLInsert 识别出导致结点违反 AVL 条件的情况 3。插入过程使用方法 UpdateLeftTree

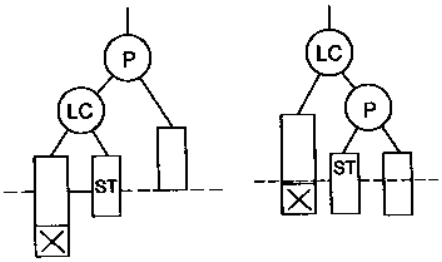
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

和 UpdateRightTree 实施重新平衡的工作。这些私有函数适当地选择一次或两次旋转以平衡结点，然后将标志 reviseBalanceFactor 置为 0(FALSE)以通知双亲子树已平衡。我们在说明旋转的细节之前先给出代码。

```
template < class T >
void AVLTree < T > ::UpdateLeftTree (AVLTreeNode < T > * &p,
                    int &reviseBalanceFactor)
{
    AVLTreeNode < T >  * lc;
    lc = p - > left();                    // 左子树边偏重
    if (lc - > balanceFactor = = leftheavy)
    {
        SingleRotateRight(p);             // 需单旋转
        reviseBalanceFactor = 0;
    }
    // 右子树偏重吗?
    else if (lc - > balanceFactor = = rightheavy)
    {
        // 做一次双旋转
        DoubleRotateRight(p);
        // 此时,根结点平衡了
        revisedBalanceFactor = 0;
    }
}
```

旋转 当双亲结点 P 变得不平衡以后就必须进行旋转。若在位置 X 处插入结点后双亲结点(P)和左孩子(LC)都偏重于左侧则发生“单右旋(single right rotation)”。我们这样旋转结点：让 LC 代替双亲，而双亲变为右孩子。同时，我们取下 LC 的右子树(ST)上的结点，将它挂到 P 的左子树上。这样就保持了顺序，因为 ST 中的结点大于或等于 LC 但小于 P。旋转使双亲与左孩子都得到了平衡。



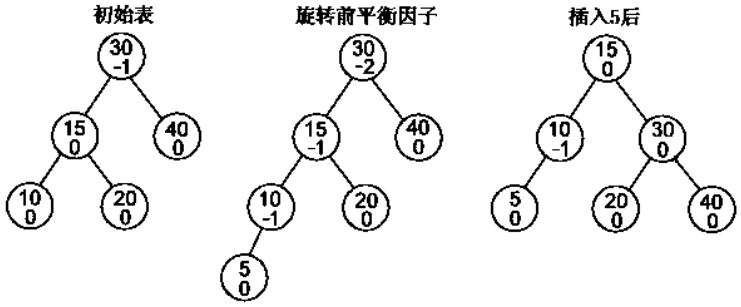

```
// 绕结点 p 顺时针旋转;使 lc 成为新轴
template < class T >
void AVLTree < T > ::SingleRotateRight (AVLTreeNode < T > * &p)
{
    // p 的左子树“超重”
    AVLTreeNode < T >  * lc;
    // 将 p 的左子树给 lc
    lc = p - > Left();
    // 修改双亲结点及左孩子的平衡因子
    p - > balanceFactor = balanced;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

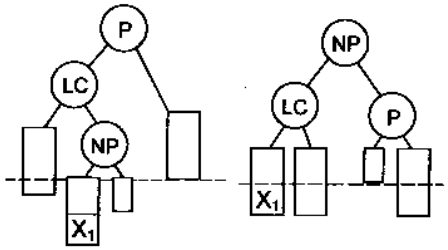
www.Linuxidc.com

```
    lc->balanceFactor = balanced;
    // lc 的右子树 st 应继续为 lc 右子树的一部分,将它改为 p 的左子树
    p->Left() = lc->Right();
    // 旋转 p 使其为 lc 的右子树,lc 成为新轴
    lc->Right() = p;
    p = lc;
}
```

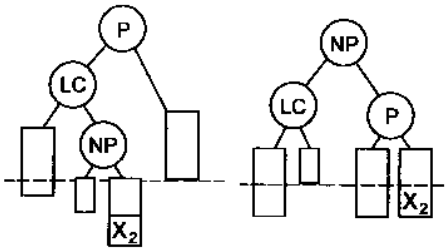
在下面的 AVL 树中,插入 5 以后导致结点 30 违反 AVL 条件。同时,结点 15 的左子树(LC)变得重一些,我们调用 SingleRotateRight 例程对结点重排序。最后,双亲结点(结点 30)得到平衡,结点 10 偏重于左侧。



若双亲结点(P)偏重于左侧而左孩子(LC)偏重于右侧,则发生"双右旋(double right rotation)"。设 NP 是 LC 的较重右子树的根。我们旋转结点以使 NP 代替双亲结点。在下图中,我们说明新结点作为 NP 的孩子被插入的两种情况。不管哪一种情况,NP 都变为双亲结点而原来的双亲 P 则旋转为 NP 的右子树。



在第 1 幅图中,我们看到了结点 X_1 在被加入到 NP 的左子树中以后的移动情况。下面的第 2 幅图则示意了将 X_2 插入到 NP 的右子树以后重新定位的情况。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 绕结点 p 双右旋
template <class T>
void AVLTree<T>::DoubleRotateRight (AVLTreeNode<T> * &p)
{
    // 被旋转的两个子树
    AVLTreeNode<T> *lc, *np;
    // 在树中,结点 lc <= 结点(np)<结点(p)
    lc = p->Left();          // lc 为双亲的左孩子
    np = lc->Right();        // np 为 lc 的右孩子
      // 修改 p, lc 及 np 的平衡因子
      if (np->balanceFactor == rightheavy)
      {
          p->balanceFactor = balanced;
          lc->balanceFactor = rightheavy;
      }
    else if (np->balanceFactor == balanced)
    {
        p->balanceFactor = balanced;
        lc->balanceFactor = balanced;
    }
    else
    {
        p->balanceFactor = rightheavy;
        lc->balanceFactor = balanced;
    }
    np->balanceFactor = balanced;
    // 在 np 代替 p 之前,注意卸掉其老子树,连上新子树
    lc->Right() = np->Left();
    np->Left() = lc;
    p->Left() = np->Right();
    np->Right() = p;
    p = np;
}
```

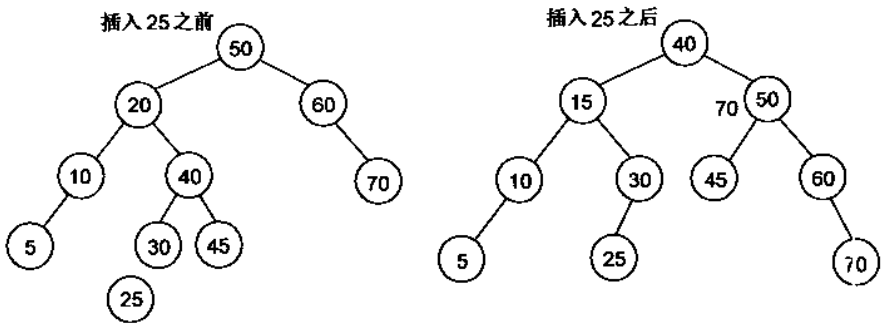
下面的树示意了双旋转。插入 25 以后导致根结点 50 不平衡。在此情况下,结点 20 的右子树重,因此需要双旋转。结点 40 变成新的双亲结点。原来的双亲旋转到右子树并挂上结点 45。此结点也是从树的左侧旋转过来的。

评价 AVL 树

AVL 树的价值取决于应用,因为当插入和删除结点时维护高度上的平衡需要额外开销。如果树重复增加或删除数据项,操作可能需要大量的时间。另一方面,如果你的数据使二叉搜索树变成了退化树,则树在搜索元素方面的优势会丧失,这时可有效地使用 AVL 树,AVL 树没有最坏的情况,因为这种结构接近完全二叉树。Find 操作的执行时间为 O($\log_2 n$)。经验表明大约 50% 的插入和删除需要进行旋转。平衡操作的复杂度表明 AVL 树仅仅应当被用在数据搜索占主要操作的场合。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



程序 13.4　评价 AVL 树

本程序在 N 个随机项的情况下比较 AVL 和二叉搜索树。数据项存储于一个数组中但要被插入到两棵树中。我们对数组进行扫描，在两棵树中搜寻其每一项，累计各次搜索的长度。这就是结点距根的深度。我们计算出二叉搜索树和 AVL 树中元素的平均深度并输出它们的值。

程序分别在 1 000 和 10 000 个元素的情况下运行。对于随机数据，应注意到，AVL 树的深度在某种程度上要小一些。在最坏的情况下，具有 1 000 个元素的退化搜索树平均深度是 500 个数据项，而 AVL 树保持稳定的平均深度为 9。

```
#include <iostream.h>
#include "bstree.h"
#include "avltree.h"
#include "random.h"
// 从数组中将相同的 n 个随机数(范围为 0 至 999)分别装入到二叉搜索树及 AVL 树中
void SetupLists(BinSTree<int> &Tree1, AVLTree<int> &Tree2,
                int A[], int n)
{
    int i;
    RandomNumber rnd;

    // 在数组 A 中存放随机数并将这些随机数插入到二叉搜索树及 AVL 树中
    for (i = 0; i < n; i++)
    {
        A[i] = rnd.Random(1000);
        Tree1.Insert(A[i]);
        Tree2.Insert(A[i]);
    }
}
// 搜索树 t 中的表项,将从根到值的路径长度累积到总长度中
template <class T>
void PathLength(TreeNode<T> *t, long & totallength, int item)
{
    // 找到表项或表项不在表中时,返回已经确定的路径长度
    if (t == NULL || t->data == item)
        return;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    else
    {
        // 移到下一层;增加总长度
        totallength++;
        if (item < t->data)
            PathLength(t->Left(), totallength, item);
        else
            PathLength(t->Right(), totallength, item);
    }
}
void main(void)
{
// 用于树及数组的变量
BinSTree<int>     binTree;
AVLTree<int>      avlTree;
int    *A;
// 从 BinSTree 和 AVLTree 中搜寻所有表项用的总长度
long totalLengthBintree = 0, totalLengthAVLTree = 0;
int n, i;
cout << "How many nodes do you want in the trees?";
cin >> n;
// 建立数组和树
    A = new int[n];
    SetupLists(binTree, avlTree, A, n);
    for (i = 0; i < n; i++)
    {
        PathLength(binTree.GetRoot(), totalLengthBintree, A[i]);
        PathLength((TreeNode<int> *)avlTree.GetRoot(),
                   totalLengthAVLTree, A[i]);
    }
    cout << "Average search length for BinSTree is "
         << float(totalLengthBintree)/n << endl;
    cout << "Average search length for AVLTree is "
         << float(totalLengthAVLTree)/n << endl;
}
/*
<程序 13.4 运行结果之一>
How many nodes to test? 1000
Average search length for BinSTree is 10.256
Average search length for AVLTree is 7.901
<程序 13.4 运行结果之二>
How many nodes to test? 10000
Average search length for BinSTree is 12.2822
Average search length for AVLTree is 8.5632
*/
```

13.7 树迭代算子

我们已经见识过迭代算子在遍历包括数组和顺序表在内的线性结构时的威力。对树

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

中结点的扫描更困难一些，因为树是非线性结构，不止一种遍历顺序。第11章中的TreeNode实用类中提供了前序、中序和后序算法，它允许递归扫描树并在每个结点处调用函数。这些遍历方法的一个问题是在递归过程完成之前不能退出来。我们不能中断扫描以检查结点的内容、对数据实施各种操作然后继续扫描树的另一结点。通过使用迭代算子，客户程序就拥有了扫描树结点的工具，即在扫描时将这些结点看作是线性表成员，从而免去了了解底层扫描算法细节的麻烦。

中序迭代算子

第12章中，我们设计了抽象类Iterator以建立一套基本的表遍历方法。Iterator类中给出了不依赖于派生类的实现细节的通用格式的遍历方法。这一节我们用此基类派生出一个中序二叉树迭代算子。若将中序扫描用于二叉搜索树，则它按从小到大的顺序访问数据。这是一个有用的工具。前序、层次顺序和后序迭代算子的构造留作练习。

声明

```
// 二叉树的中序迭代算子，使用基类 Iterator
template < class T >
class InorderIterator: public Iterator < T >
{
    private:
        // 维护 TreeNode 地址栈
        Stack < TreeNode < T >  * S;
        // 树根及当前结点
        TreeNode < T >  * root, * current;
        // 遍历左路径，用于 Next 函数
        TreeNode < T >  * GoFarLeft(TreeNode < T >  * t);
    public:
        // 构造函数
        InorderIterator(TreeNode < T >  * tree);
        // 基本遍历函数的实现
        virtual void Next(void);
        virtual void Reset(void);
        virtual T& Data(void);
        // 将新树赋值给迭代算子
        void SetTree(TreeNoe < T >  * tree);
};
```

说明

InorderIterator以通用迭代算子为模板。方法EndOfList在基类Iterator中实现。构造函数初始化基类并用GoFarLeft定位第1个中序结点。类在文件“treeiter.h”中给出。

例

```
TreeNode < int >  * root;                // 二叉树
InorderIterator treeiter(root);          // 连上迭代算子
// 输出扫描中的第一个结点。对中序遍历来说，即树中最左边的结点
cout < < treeiter.Data();
// 扫描各结点并输出其值
```

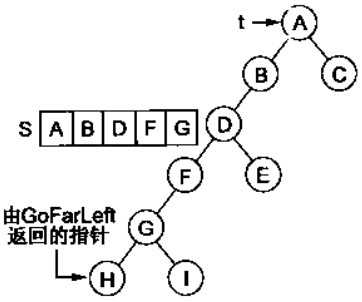

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
for (treeiter.Reset(); ! treeiter.EndOfList(); treeiter.Next())
    cout < < treeiter.Data() < < " ";
```

InorderIterator 类的实现

迭代中序遍历通过使用堆栈存放结点地址的方法仿真递归扫描。从根结点开始遍历左子树链，将其上每个结点的指针放到堆栈中。这一过程在左指针为 NULL 的结点处停止。这就是以中序扫描法访问的第 1 个结点。



方法 GoFarLeft 从结点地址 t 出发并将路径上的所有结点地址压入堆栈中直到遇上 NULL 指针为止。令 t = root(根)调用 GoFarLeft 以定位要访问的另一个结点。

```
// 返回从 t 开始的老子树的最后一个结点的指针,将经过结点的地址压入栈中
template < class T >
TreeNode < T >  * InorderIterator < T > ::GoFarLeft(TreeNode < T >  * t)
{
    // 若 t 为 NULL,则返回 NULL
    if (t = = NULL)
        return NULL;

    // 遍历到树 t 的最左边,往栈中压入每个结点的地址直到遇到左指针为 NULL 的结点返回指
    // 向该结点的指针
    while (t - > Left() ! = NULL)
    {
        S.Push(t);
        t = t - > Left();
    }
    return t;
}
```

初始化基类以后，构造函数将数据成员 root 设为二叉搜索树的根。中序遍历的第 1 个结点是通过将 root 作为参数调用 GoFarLeft 而获得的。返回值被赋予 TreeNode 指针 current。

```
// 初始化 iterationComplete.基类将其置为 0,但树可能为空,遍历的第一个结点为最左边
// 的结点
template < class T >
InorderIterator < T > ::InorderIterator(TreeNode < T >  * tree):
        Iterator < T > (), root(tree)
{
    iterationComplete = (root = = NULL);
    current = GoFarLeft(root);
}
```

Reset 方法与构造函数基本相同，不同的是它要清空堆栈。

在第 1 次调用 Next 之前，current 已经指向中序扫描的第 1 个结点。Next 实现从结点到结点的中序遍历。它完成下列算法的第 1 步和第 2 步。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

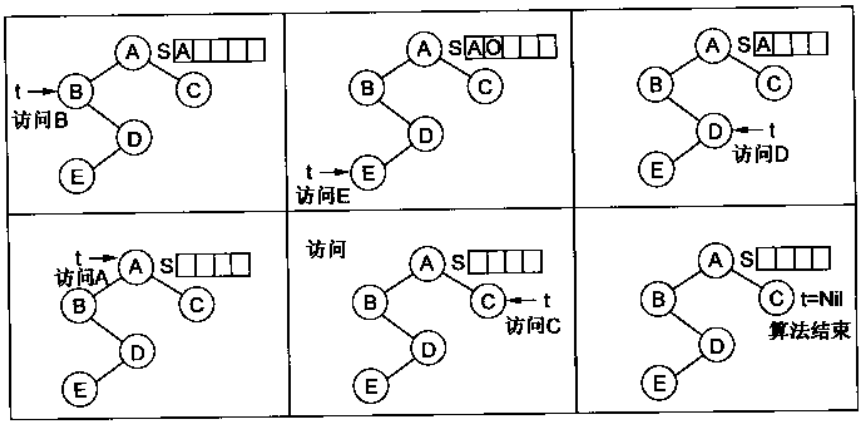
www.Linuxidc.com

1. 如果结点的右分支非空，则往右走并遍历左分支链，将每个结点的指针压入堆栈，直到发现具有 NULL 左指针的结点为止。
2. 如果结点的右分支为空，则我们已完成对结点的左分支、结点自身及其右分支的扫描。

下一个要访问的结点地址在堆栈中。如果堆栈非空，弹出栈顶以确定要扫描的下一个结点。

如果堆栈为空，所有结点都已被访问过，扫描完成。

下图示意了对 5 个结点的树进行扫描的情况。




```
template <class T>
void InorderIterator<T>::Next(void)
{
    // 若已访问所有结点，则出错退出
    if (iterationComplete)
    {
        cerr << "Next: iterator has passed the end of the list!"
             << endl;
        exit(1);
    }

    // 访问完当前结点后，若有右子树，则往右走后遍历左分支链，将每个结点的指针压入堆栈
    if (current->Right() != NULL)
        current = GoFarLeft(current->Right());

    // 无右子树，栈不为空时，则处理栈中其他结点，将栈顶结点弹出作为当前结点
    else if (! S.StackEmpty())              // 往上移动一层
        current = S.Pop();

    // 当前结点无右子树且栈中无结点，遍历结束
    else
        iterationComplete = 1;
}
```

应用：树排序（TreeSort）

用 InorderIterator 对象遍历搜索树时，最终得到的表项是按所排顺序遍历的，我们可以

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

设计出另一种叫做 TreeSort 的排序算法。该算法假定表项初始时以 n 个元素的数组形式存储。搜索树起过滤器作用,元素则是用搜索树插入算法从数组中被复制到树中。通过中序遍历树并将元素插回数组中,所得到的表就排好了序。图 13.5 中示意了对 8 个元素的整数数组排序的情况。树排序算法由函数 TreeSort 实现,此函数在文件"treesort.h"中给出。

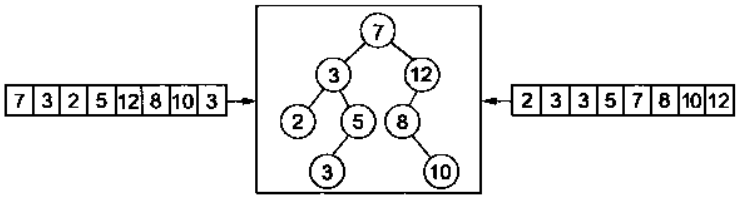


图 13.5 TreeSort

```
#include "bstree.h"
#include "treeiter.h"
// 用二叉搜索树对数组排序
template <class T>
void TreeSort(T arr[], int n)
{
    // 数组数据存放的二叉搜索树
    BinSTree<T> sortTree;
    int i;

    // 往搜索树中插入数组元素
    for (i=0; i < n; i++)
        sortTree.Insert(arr[i]);

    // 为 sortTree 定义中序迭代算子
    InorderIterator<T> treeSortIter(sortTree.GetRoot());

    // 中序遍历树,将每个元素拷回数组 arr 中
    i = 0;
    while(! treeSortIter.EndOfList())
    {
        arr[i++] = treeSortIter.Data();
        treeSortIter.Next();
    }
}
```

树排序的复杂度 在二叉搜索树中定位一个数据值所需的预期比较次数为 $O(\log_2 n)$。因为树排序要将 n 个值放到树中,所以直观上看其平均计算复杂度为 $O(\log_2 n)$。但最坏的情况是 $O(n^2)$,当表已经排好序或已逆序排列时会发生这种情况。其对应的搜索树是链表形式的退化树。对最坏的情况作一透彻的分析会发现 $O(n^2)$ 次比较是必须的。第 1 次插入需要 0 次比较;第 2 次插入需要 2 次比较(1 次是与根结点,1 次是确定在哪个子树中插入数值);第 3 次插入需要 3 次比较;第 4 次插入需要 4 次比较;……第 n 次插入需要 n 次比较。总比较次数是:

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

$$
\begin{aligned}
&0+2+3+4\cdots+n \\
&=(1+2+3+4+\cdots+n)-1 \\
&=n(n+1)/2-1=O(n^2)
\end{aligned}
$$

树的各个结点还需要分配存储空间，故最坏的情况不比交换排序好。

当 n 个随机数据值被重复插入到一棵二叉搜索树中时，我们期望这棵树达到相对平衡。最好的情况是树为完全树。我们以深度为 d 的满树为例估算一下最好的情况下的上界。在任一 i 层，$1\leqslant i\leqslant d$，结点数为 2^i 个。因为将 1 个结点放到第 i 层需要 $i-1$ 次比较，所以满树的树排序需要 $(i+1)*2^i$ 次比较才能定位第 i 层的所有元素。我们还记得 $n=2^{d+1}-1$，则以下不等式可推导出以大 O 度量的算法复杂度。

$$
\begin{aligned}
\sum_{i=1}^{d}(i+1)2^i \leqslant (d+1)\sum_{i=1}^{d} &= (d+1)(2^{d+1}-2) \\
&= (d+1)(2^{d+1}-1-1)=(d+1)(n-1) \\
&= (n-1)\log_2(n+1) \\
&= O(n\log_2 n)
\end{aligned}
$$

上述计算表明树排序的最好情况是 $O(n\log_2 n)$。

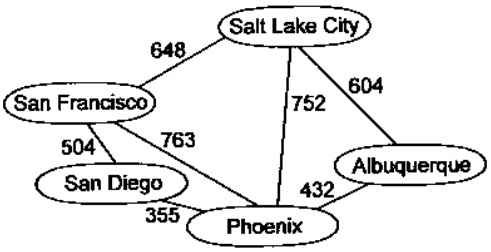
13.8 图

树是一种由从根发散出的结点所组成的层次结构。结点由指针连结起来，即双亲由指针链接到孩子。这一节我们介绍图。图是一种概念化的层次结构。图由一组叫作“顶点(vertex)”的数据项和一组连接各对顶点的“边(edge)”所组成。边 $E=(V_i,V_j)$ 将顶点 V_i 连接到 V_j：

顶点 $=\{V_0,V_1,V_2,V_3,\cdots,V_{m-1}\}$

边 $=\{E_0,E_1,E_2,E_3,\cdots,E_{n-1}\}$

如果顶点表示城市，边表示公路系统，城市间可以进行双向运动。这时边不具有方向性，G 被称作“**无向图**(undirected graph)”。



如果边表示的是通信系统，有可能信息从一个结点流向另一个结点但反过来则不行。在这种情况下，G 就成了“有向图(directed graph, digrahp)”。图 13.6 中示意了图 G 作为有向图和无向图的情况。我们的研究重点是无向图。

有向图的边由顶点对 (V_i,V_j) 表示，其中 V_i 是起始顶点，V_j 是终止顶点。**路径** $P(V_S,V_E)$ 是顶点序列 $V_S=V_R,V_{R+1},\cdots,V_{R+T}=V_E$，其中 V_S 是起始顶点，V_E 是终止顶点，相邻各

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

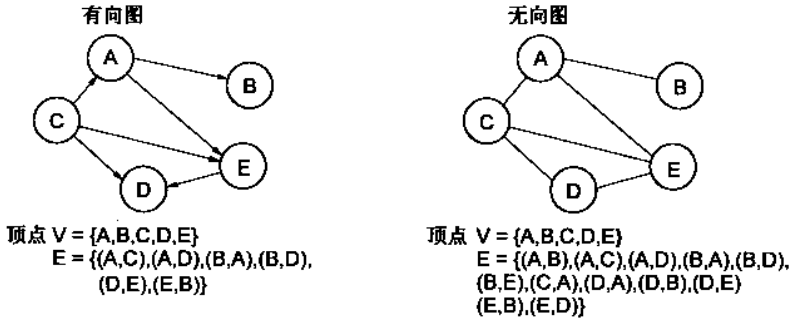


图 13.6　有向图和无向图

对顶点是边。在有向图中，从顶点 V_E 到 V_C 具有有向路径时，不一定存在从 V_C 到 V_E 的路径。例如，对于图 13.6 中的有向图 G，

Path(A,B) = {A,D,E,B}　　Path(E,C) = {E,B,A,C}

Path(B,A) = {B,A}　　Path(C,E) = { }　　//没有路径连通。

连通分量

路径的概念也定义了图中的连接关系。如图中存在从顶点 V_i 到 V_j 的路径则称 V_i 和 V_j 是"**连通的**(connected)"，如果有向图中任意两个结点之间都有一条有向路径则称该有向图是"**强连通的**(strong connected)"。如果对于每一对顶点 V_i 和 V_j，都有一条有向路径 $P(V_i, V_j)$ 或 $P(V_j, V_i)$，则有向图为"**弱连通的**(weak connected)"。图 13.7 中示意了有向图的连通性。

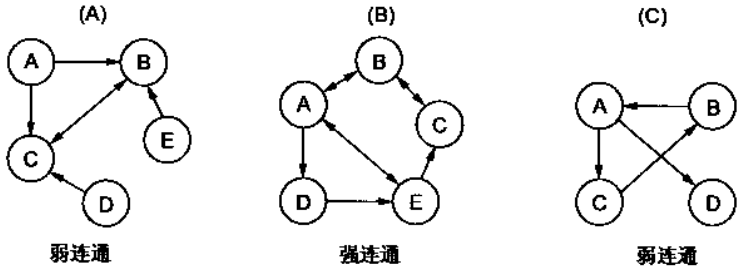


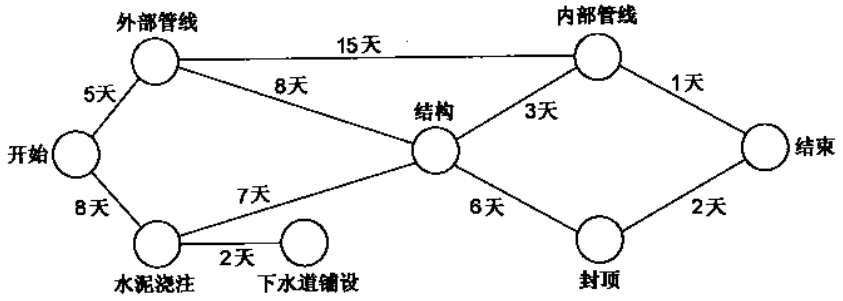
图 13.7　有向图中的强连通和弱连通分量

我们将强连通顶点的概念扩展为"强连通分量(strong connected component)"。所谓强连通分量是指满足以下条件的最大顶点集合 $\{V_i\}$：对于集合中每一对顶点 V_i 和 V_j，同时存在从 V_i 到 V_j 以及从 V_j 到 V_i 的路径。如果含 3 个或 3 个以上顶点的路径从一个顶点出发不经过重复边又回到其自身，则此路径被称为"回路(cycle)"。在图 13.7 的有限图(C)中，顶点 A, B, C 可形成回路：$A \rightarrow C \rightarrow B \rightarrow A$。没有回路的图被称为"无回路(acyclic)"图。

若有向图的每一条边都有一个关联值或权，则称该有向图为"赋权有向图(weighted digraph)"。在一个运输有向图中，权值可以代表城市间的距离。在一个工作调度图中，边值可以定义完成任务的时间长度。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



13.9 Graph 类

本节介绍赋权图的数据结构。我们从图的数学定义入手，因为它是 Graph ADT 的基础。全部顶点以数据元素表形式给出，而所有边则定义为连接顶点的有序对的表。

声明 Graph ADT

赋权图由顶点和赋权边组成。ADT 是围绕增加或删除这些数据项的操作而设计的。对于每个顶点 V_i，ADT 用边 $E(V_i,V_j)$ 标明所有邻接顶点 V_j。

ADT Graph **is**

Data

顶点集$\{V_i\}$及边集$\{E_i\}$。边即顶点对(V_i,V_j)，表示顶点 V_i 与 V_j 之间有一连接。每条边都对应一个权值，它定义了沿边(V_i,V_j)旅行时的开销。

Operations

Constructor

Input: 无
Process: 生成由一组顶点和边组成的图。

InsertVertex

Input: 一个新顶点。
Preconditions: 无
Process: 将顶点插入到顶点集合中。
Output: 无
Postconditions: 顶点表扩大。

InsertEdge

Input: 一对顶点 V_i 和 V_j，以及权 W。
Preconditions: V_i 和 V_j 必须在顶点集合中；边(V_i,V_j)必须在边集合中。
Process: 将具有权值 W 的边(V_i,V_j)插入到边集合中。
Output: 无
Postconditions: 边集扩大。

DeleteVertex

Input: 顶点 V_D 的引用。
Preconditions: 输入值必须在顶点集合中。
Process: 从顶点表中删除顶点并删除所有形如(V,V_D)和(V_D,V)的与顶点 V_D 相连的边。
Output: 无
Postconditions: 顶点集合和边集合被修改。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

DeleteEdge

Input:	顶点对 V_i 和 V_j
Preconditions:	输入值必须在顶点集合中。
Process:	如果存在边(V_i,V_j),则从边表中删掉它。
Output:	无
Postconditions:	边集被更改。

GetNeighbors

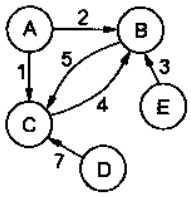
Input:	顶点 V。
Preconditions:	无
Process:	找出所有顶点 V_E,只要(V,V_E)是一条边。
Output:	返回所有这样的顶点表。
Postconditions:	无

GetWeight

Input:	顶点对 V_i 和 V_j。
Preconditions:	输入值必须属于顶点集合。
Process:	如果存在边(V_i,V_j),则取其权值。
Output:	返回边的权值,若边不存在则返回 0。
Postconditions:	无

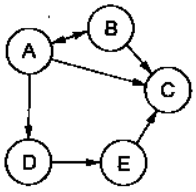
end ADT Graph

图的表示 可以用多种方法表示有向图的顶点 V 和边 E。一种简单的方法是以顺序表 $V_0,V_1,\cdots,V_{m-1}$的形式存储所有顶点。图的边则以 $m\times m$ 矩阵表示,这种矩阵叫作"邻接矩阵(adjacency matrix)",其中第 i 行、第 j 列分别对应顶点 V_i 和 V_j。矩阵中一项(i,j)给出边 $E_{i,j}=(V_i,V_j)$的权值,若这条边不存在则为 0 值。对于非赋权有向图,邻接矩阵中的各项具有布尔值 0 和 1,表示这对顶点是否对应图中一条边。下面的例子中给出了两个有向图及其各自的邻接矩阵。



02100
00500
04000
00700
03000

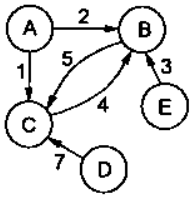
(A)



01110
10100
10000
00001
00100

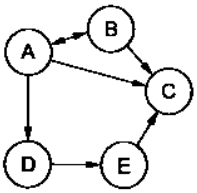
(B)

图的另一种表示法中,每个顶点都对应有一个与其邻接的顶点所组成的链表。表标明顶点的邻居。这种动态模型存储图的各条边的有关信息。对于赋权有向图,链表中每个结点都有一个权值(weight)域。下表给出了有向图(A)和(B)的表示。



顶点	邻接点表
A	B 2 C 1
B	C 5
C	B 4
D	C 7
E	B 3

(A)



顶点	邻接点表
A	B C D
B	A C
C	A
D	E
E	C

(B)

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

这一节我们设计一个 Graph 类，假定用邻接矩阵表示边。我们使用图的静态模型，即假定顶点个数有一上界。使用矩阵可简化类实现，使得我们把精力集中到各种图算法上。习题中给出了用链表表示的实现方法。Graph 类的主要内容包括 Graph ADT 的表示、输入方法 ReadGraph 以及对顶点进行深度优先和广度优先遍历的一系列搜索算法。类中还包括供应用程序使用的顶点表迭代算子。

Graph 类说明

声明

```
const int MaxGraphSize = 25;
template <class T>
class Graph
{
    private:
        // 主要数据.包括顶点表,邻接矩阵和图的大小(顶点个数)
        SeqList < T > vertexList;
        int edge [MaxGraphSize][MaxGraphSize];
        int graphsize;
        // 搜索顶点及对其在表中定位的方法
        int FindVertex(SeqList < T > &L, const T& vertex);
        int GetVertexPos(const T& vertex);

    public:
        // 构造函数
        Graph(void);
        // 检测图的函数
        int GraphEmpty(void) const;
        int GraphFull(void) const;
        // 数据访问方法
        int NumberOfVertices(void) const;
        int NumberOfEdges(void) const;
        int GetWeight(const T& vertex1, const T& vertex2);
        SeqList < T > & GetNeighbors(const T& vertex);
        // 修改图的方法
        void InsertVertex(const T& vertex);
        void InsertEdge(const T& vertex1, const T& vertex2,
                        int weight);
        void DeleteVertex(const T& vertex);
        void DeleteEdge(const T& vertex1, const T& vertex2);
        // 例程
        void ReadGraph(char * filename);
        int MinimumPath(const T& sVertex, const T& eVertex);
        SeqList < T > & DepthFirstSearch(const T& beginVertex);
        SeqList < T > & BreadthFirstSearch(const T& beginVertex);
        // 用于扫描顶点的迭代算子
        friend class VertexIterator < T > ;
};
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

说明

类数据成员包括存储于顺序表中的顶点、由 2 维整数矩阵所表示的边，以及用来统计顶点个数的数据值 grahsize。grahsize 的值由方法 NumberOfVertices 返回。

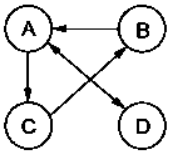
实用方法 FindVertex 检查顶点是否已经在表 L 中。它由搜索方法调用。方法 GetVertexPos 求顶点在 vertexList 中的位置。这个位置相当于邻接矩阵中的行或列索引。

ReadGraph 方法要求一个文件名参数，用于输入图的顶点和边。

VertexIterator 派生自 SeqListIterator 类，它使用户能够对顶点进行扫描。迭代算子可简化应用程序。

例

```
Graph < char > G;                    // 顶点为字符的图
G.ReadGraph("graph.dat");            // 从"graph.dat"
        // 图的输入格式
    < Number of Vertices >     4
    Vertex0                    A
    Vertex1                    B
    Vertex2                    C
    Vertex3                    D
    < Number of Edges >        5
    Edge0     Weight0          A  C  1
    Edge1     Weight1          A  D  1
    Edge2     Weight2          B  A  1
    Edge3     Weight3          C  B  1
    Edge4     Weight4          D  A  1
```




```
VertexIterator < char > Viter(G);      // 顶点的迭代算子
SeqList < char > L;
for (viter.Reset(); ! viter.EndOfList(); viter.Next())
{
    cout < < "For vertex " < < viter.Data() < < ": Neighbors are ";
    L = G.GetNeighbors(viter.Data());

    // 输出邻居
    SeqListIterator < char > liter(L);    // 邻居表
    for (liter.Reset(); ! liter.EndOfList(); liter.Next())
        cout < < liter.Data() < < "  ";
}
```

Graph 类的实现

Graph 类的构造函数负责初始化 MaxGraphSize × MaxGraphSize 的邻接矩阵，并将图形大小设置为 0。构造函数将矩阵中的每一项设为 0 以表示没有一条边。

```
// 构造函数.将邻接矩阵的所有项设为 0 并将图的大小设为 0
template < class T >
Graph < T > ::Graph(void)
{
    for (int i = 0; i < MaxGraphSize; i + + )
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        for (int j = 0; j < MaxGraphSize; j++)
            edge[i][j] = 0;
    graphsize = 0;
}
```

图分量计数 私有数据成员 graphsize 维护顶点表的大小。其值可以由方法 NumberOfVertices 访问。类运算符 GraphEmpty 指示 graphsize 值是否为 0。

图分量的访问 图分量包含于顶点表和邻接矩阵中。对于顶点，我们可以用顶点迭代算子扫描顶点表中的表项。迭代算子是 Graph 类的友元，在此它可以访问 vertexList。图迭代算子由 SeqListIterator 类继承得来。

```
template <class T>
class VertexIterator: public SeqListIterator<T>
{
    public:
    VerexIterator(Graph<T>& G);
};
```

构造函数只是初始化基类以遍历私有数据成员 vertexList。以下代码给出了此构造函数的具体实现。

```
template <class T>
VertexIterator<T>::VertexIterator(Graph<T>& G):
    SeqListIterator<T> (G.vertexList)
{}
```

迭代算子扫描 vertexList 表项，它可以用于实现函数 GetVertexPos。GetVertexPos 扫描顶点表并返回顶点在表中的位置。下面一段迭代算子编码搜寻顶点：

```
template <class T>
int Graph<T>::GetVertexPos(const T& vertex)
{    SeqListIterator<T> liter(vertexList);
    int pos = 0;
    while(! liter.EndOfList() && liter.Data() ! = vertex)
    {
        pos++;
        liter.Next();
    }
    return pos;
}
```

方法 GetWeight 返回连接顶点 vertex1 和 vertex2 的边的权值。该方法用 GetVertexPos 取得两个顶点在表中的位置，进而确定在邻接表中行和列的位置。如果任一顶点不在表中，则方法返回 -1。

方法 GetNeighbors 以顶点为参数建立所有邻接顶点表。该方法扫描邻接矩阵，找出满足(V,V_E)为边这样一个条件的结点 V_E，形成一个表。表以参数形式返回，可以用 SeqList 迭代算子对其进行扫描。如果顶点没有邻居，该方法返回空表。

```
// 返回所有邻接顶点
template <class T>
```

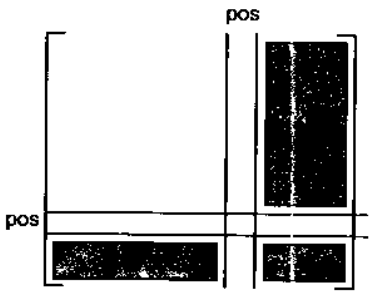

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
SeqList < T > & Graph < T > ::GetNeighbors(const T& vertex)
{
    SeqList < T >  * L;
    SeqListIterator < T > viter(vertexList);
    // 申请顺序表的空间
    L = new SeqList < T > ;

    // 在表中查找 pos 来确定使用邻接矩阵中的哪一行
    int pos = GetVertexPos(vertex);
    // 若顶点不在表中,则退出
    if (pos = = -1)
    {
        cerr < < "GetNeighbors: vertex not in graph." < < endl;
        return * L;    // 返回空表
    }
    // 扫描邻接矩阵中该行并往表中插入所有具有非 0 权边的顶点
    for (int i = 0; i < graphsize; i + + )
    {
        if (edge[pos][i] > 0)
            L - > Insert(viter.Data());
        viter.Next();
    }
    return * L;
}
```

顶点和边的更新 若要插入一条边,我们先用 GetVertexPos 检查 vertex1 和 vertex2 是否都在顶点表中,若其中任一项不在表中,则打印出错消息,方法返回。一旦确定了顶点的位置 pos1 和 pos2,InsertEdge 将在邻接矩阵中设置边(pos1,pos2)的权值。本操作的时间复杂度为 O(n),因为 GetVertexPos 的执行时间为 O(n)。

Graph 类中还提供了从图中删除顶点的方法 DeleteVertex。如果顶点不在表中,则打印出错消息,方法返回。但如果在表中则需要删除所有与被删除结点连接的边。邻接矩阵中有 3 个区域需要调整,故整个操作的时间复杂度为 $O(n^2)$,因为每个区域都是 $n \times n$ 邻接矩阵的一部分。



区域Ⅰ:将列索引左移。

区域Ⅱ:将行索引上移,列索引左移。

区域Ⅲ:将行索引上移。

```
// 从顶点表中删除一个顶点并修改邻接矩阵删除所有与该顶点相连的边.
template < class T >
void Graph < T > ::DeleteVertex(const T& vertex)
{
    // 取顶点在顶点表中的位置
    int pos = GetVertexPos(vertex);
    int row, col;
    // 若该顶点不存在,则出错退出
    if (pos = = -1)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

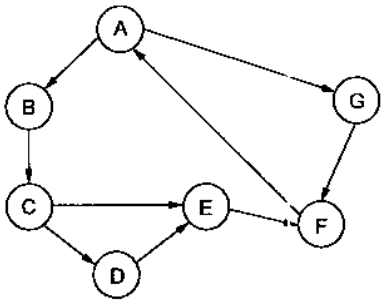
```
    {
        cerr << "DeleteVertex: vertex is not in the graph."
             << endl;
        return;
    }
    // 删除该顶点并将 graphsize 减 1
    vertexList.Delete(vertex);
    graphsize--;
    // 邻接矩阵分成三个区域
    for (row = 0; row < pos; row++)                          // 区域Ⅰ
        for (col = pos + 1; col < graphsize; col++)
            edge[row][col-1] = edge[row][col]
    for (row = pos+1; row < graphsize; row++)                // 区域Ⅱ
        for (col = pos + 1; col < graphsize; col++)
            edge[row-1][col-1] = edge[row][col];
    for (row = pos+1;row < graphsize; row++)                 // 区域Ⅲ
        for (col = 0; col < pos; col++)
            edge[row-1][col] = edge[row][col];
}
```

若要删除一条边，我们只需删除两个顶点之间的连接。在证实顶点在 VertexList 中之后，方法 deleteEdge 将权值 0 赋给这条边，而其他边不变。若边不在图中，则程序打印出错消息并返回。

图的遍历

扫描非线性结构时，我们必须设计一种结点访问策略，并且一旦访问过它们便做上标记。二叉树中定义了一系列搜索方法，图中也有相应的方法。前序二叉树扫描策略是先访问结点再访问子树。对于图来说，"深度优先(depth-first)搜索"是一种广义的前序遍历。起始顶点被作为参数并且成为被访问的第 1 个结点。在沿路径移动一直到"死胡同"的过程中，我们将邻接顶点放到堆栈中，这样如果还有未被访问的结点便返回并搜索其他路径。被访问过的顶点组成了从起始顶点出发可以到达的所有顶点的集合。

树有一种逐层扫描算法，即从根开始按层次由浅入深地访问树中结点。对于图，"广度优先(breadth-first)搜索"用的是类似的策略，即从起始顶点开始访问它的每个邻接顶点，然后继续扫描下一层邻接顶点；如此进行下去，直到到达路径终点为止。算法在逐层扫描图时用队列存储邻接顶点。我们以下图为例说明两种搜索算法。两种情况下的初始顶点都是 A。



深度优先搜索 搜索算法用表 L 维护被访问顶点的记录，用堆栈 S 存储邻接顶点。在将初始顶点放入堆栈以后，我们开始迭代过程，从堆中弹出一个顶点并访问它。当堆栈为空时过程终止并返回被访问过的顶点表。对每一步操作，用以下策略：

从堆栈中弹出一个顶点 V，并检查表 L 看 V 是否已经被访问过。如果没有，则我们正

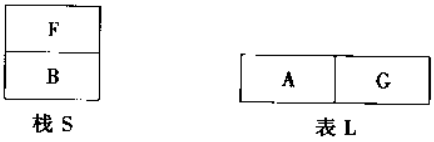
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

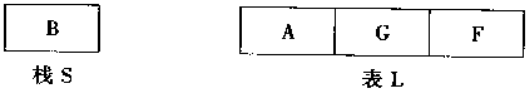
在访问的是一个新顶点，这时求出其邻接顶点表。我们将 V 插入到表 L 中以免它再次被访问。最后，我们把 V 的邻接顶点中未进入 L 的顶点放入堆栈中。

对于例图，假设 A 为起始顶点。搜索开始时将 A 弹出堆栈并处理该顶点。将 A 插入到被访问结点的表中并将邻接顶点 B 和 G 压入堆栈。处理完 A 后，S 和 L 存储了以下数据：

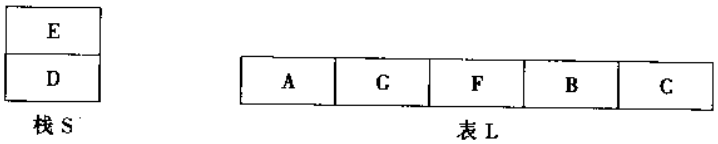
迭代继续进行：从堆栈中弹出 G。因为该顶点不在 L 中，所以将它添加到 L 中并将其唯一的邻接顶点 F 放入堆栈中：



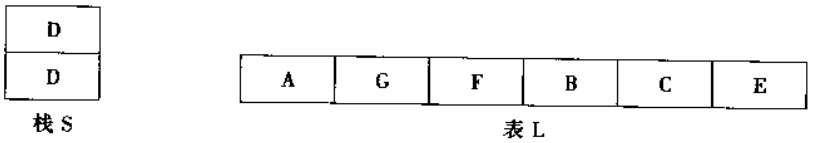
从堆栈中弹出 F 并将该顶点放入 L 中以后，我们遇到了“死胡同”，因为 F 的邻接点 A 已经在表 L 中。现在堆栈中还剩下顶点 B，它在第 1 阶段搜索中被标识为 A 的邻接点。



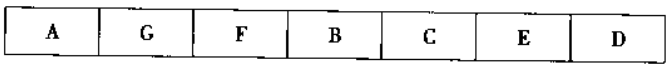
搜索继续进行：按顺序访问结点 B 和 C。堆栈中所包含的是 C 的邻接点结点 D 和 E。



D 和 F 都与 E 邻接，因而都有资格进入堆栈，但是我们已沿路径 A—G—F 访问过 F，故不再用它。而结点 D 则再次被压入堆栈，这是因为我们的算法并不能意识到 D 也可由 C 得到。



搜索过程在访问结点 D 后结束。第 2 次出现的 D 被舍弃，因为该顶点已经在表中：




```
// 从起始顶点开始深度优先法遍历图，返回遍历顺序的顶点表
template <class T>
SeqList<T> & Graph<T>::DepthFirstSearch(const T& beginVertex)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{
    // 临时存放等待扫描顶点的栈
    Stack<T> S;
    // L 为扫描结点表;adjL 存放当前顶点的邻接点。L 由动态内存产生,可返回指向其的指针
    SeqList<T> *L, adjL;
    // iteradjL 用于遍历邻接顶点表
    SeqListIterator<T> iteradjL(adjL);
    T vertex;
    // 初始化表 L;将起始顶点压入栈中
    L = new SeqList<T>;
    S.Push(beginVertex);
    // 继续扫描,直到栈为空
    while (! S.StackEmpty())
    {
        // 弹出下一顶点
        vertex = S.Pop();
        // 检查该顶点是否已在表 L 中
        if (! FindVertex(*L,vertex))
        {
            // 若不在,将它加入 L 中并取得其所有邻接顶点
            (*L).Insert(vertex);
            adjL = GetNeighbors(vertex);
            // 置迭代因子为当前 adjL
            iteradjL.SetList(adjL);
            // 扫描邻接顶点表;若不在 L 中则压入栈
            for (iteradjL.Reset();! iteradjL.EndOfList();iteradjL.Next())
                if (! FindVertex(*L, iteradjL.Data()))
                    S.Push(iteradjL.Data());
        }
    }
    // 返回深度优先扫描得到的顶点表
    return *L;    // 返回表 L
}
```

广度优先搜索 与二叉树的逐层扫描一样,广度优先搜索使用队列。除此之外,该算法使用与深度优先搜索相同的技术。即顶点是放到队列中而非堆栈中。迭代过程从队列中删除顶点直到队列空为止。

从队列中删除顶点 V 并检查 V 是否在已访问的结点表 L 中。如果 V 不在 L 中,则它是新顶点,必须将它添加到 L 中。同时应找出 V 的所有邻接点,将其中不在已访问结点表中的插入到队列中。

如果对上面的用来说明深度优先搜索的图执行该算法,则顶点的访问次序是:

A B C F D E

复杂度分析 在搜索算法中,我们要访问每一个顶点,这需要的计算时间为 O(n)。一旦顶点被加到已访问结点表中,我们就检查邻接矩阵中的相应行以标出所有邻接结点。每行所费时间为 O(n),因此每次搜索的总计算时间为 $n*O(n)=O(n^2)$。图用邻接矩阵

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

表示时所需要的比较次数与图有多少条边无关。即使图的边数很少(“稀疏图”),我们仍然需要对图中每个顶点进行 n 次比较。如果图是用表表示,则搜索算法的计算时间取决于图中边的密度。最好的情况是没有边,每个邻接表的长度都是 1。每次搜索的计算时间将是 $O(n+n)=O(n)$。最坏的情况是每个顶点都与其他所有顶点相连,每个邻接表的长度为 n。这时搜索时间量变为 $O(n^2)$。

应用

我们回顾一下,在有向图中,若从任一顶点到其他任一顶点都有有向路径,则称其为强连通的。“强连通分量”是彼此强连通的顶点的子集。一个强连通图只有一个强连通分量,而任何图都可被分割为若干强连通分量。例如图 13.8 中,图被分割为 3 个强连通分量。

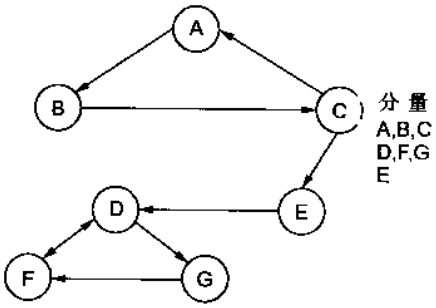


图 13.8 强连通分量

在图论中,确定强连通分量用的是一些经典算法。在以下应用程序中,我们用深度优先搜索设计一个非优化的用于查找图的强连通分量的方法。函数 PathConnect 用深度优先搜索检测从顶点 V 到 W 是否有直达路径,函数返回布尔值 TRUE 或 FALSE。

```
template <class T>
int PathConnect (Graph<T> *G, T V, T W)
{
    SeqList<T> L;
    // 搜索与 V 相连的顶点
    L = G.DepthFirstSearch(V);
    // 若 W 在表中,返回 TRUE
    if (L.Find(W))
       return 1;
    else
       return 0;
}
```

函数 ConnectedComponent 开始时,顶点表 markedList 为空。在算法中,这个表所包含的是已经在强连通分量中的顶点的集合。用迭代算子循环遍历图中每个顶点。对于每个顶点 V,检查它是否在 markedList 中。如果不在,必须构建包含 V 的新的强连通分量。将包含新的强连通分量的表 scList 清空,并用深度优先搜索找出从 V 出发可以到达的所有顶点,令其组成表 L。对于 L 中的每个元素,用 PathConnect 确定是否有返回 V 的路径。只要这种路径存在,就将顶点放到 scList 和 markedList 中。注意 V 被同时加入到 scList 和 markedList 中。既然从 V 到 scList 中的所有顶点都有路可达且从 sc 中每个顶点有路可返回 V,则 scList 任意两个顶点之间都有路径。这些顶点构成了下一个强连通分量。因为 scList 中的每个顶点也在 markedList 中,所以不用再考虑它们。

```
template <class T>
void ConnectedComponent (Graph<T> &G)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
}
    VertexIterator<T> viter(G);
    SeqList<T> markedList, scList, L, K;
    for (viter.Reset(); ! viter.EndOfList(); viter.Next())
    {
        // 循环检查每个顶点 viter.Data()
        if (! markedList.Find(viter.Data()))
        // 若未标记,放入强连通分量
        {
            scList.ClearList();
            // 取与 viter.Data 相连的顶点
            L = G.DepthFirstSearch(viter.Data());
            // 扫描表,看该结点是否又连回了 viter.Data()
            SeqListIterator<T> liter(L);
            for (liter.Reset();! liter.EndOfList();liter.Next())
                if (PathConnect(G, liter.Data(), viter.Data()))
                {
                    // 将该顶点插入到当前强连通分量和 markedList 中
                    scList.Insert(liter.Data());
                    markedList.Insert(liter.Data());
                }
            PrintList(scList);    // 输出强连通分量
            cout << endl;
        }
    }
```

程序 13.5　强连通分量

本程序求图 13.8 中图的强连通分量。图是用 ReadGraph 由文件"sctest.dat"中读入的。函数 PathConnect，ConnectedComponent 以及 PrintList 在文件"conncomp.h"中给出。

```
#include <iostream.h>
#include <fstream.h>
#include "graph.h"
#include "conncomp.h"
void main(void)
{
    Graph<char> G;

    G.ReadGraph("sctest.dat");

    cout << "Strong Components:" << endl;
    ConnectedComponent(G);
}
/*
<程序 13.5 运行结果>
Strong Components:
A  B  C
D  G  F
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
E
*/
```

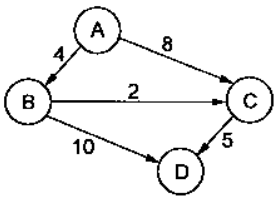
最短路径 深度优先和广度优先搜索只是找出从初始结点开始的路径上的所有顶点，并不谋求优化顶点到顶点间的移动。而我们现在要讨论的正是这一问题。许多应用要求选择“成本”最少的路径，这种“成本”可以用顶点间路径的累计权值来度量。PathInfo对象被存储在一个优先级队列中。此队列可以供我们访问队列中具有最小成本的对象。

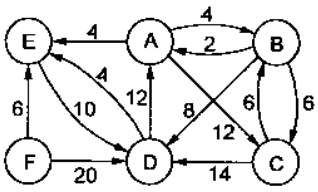
```
template <class T>
struct PathInfo
{
    T startV, endV;
    int cost;
};
template <class T>
int operator <= (const PathInfo<T>& a, const PathInfo<T>& b)
{
    return a.cost <= b.cost;
}
```

因为图中两个顶点之间可能存在不同的路径，所以两个 PathInfo 对象可能连接相同的顶点但却有不同的成本。例如，从顶点 A 到顶点 D 有 3 条路径，各条路径成本不同。

PathInfo类通过比较成本值定义“<=”运算符。算法检查优先级队列中存储的PathInfo对象并从中选择一个具有最短路径的。下图示意用来找出连接起始顶点 sVertex 与终止顶点 eVertex 的最短路径的算法。如果顶点之间没有路径相通，则算法终止并打印消息。例子中，起始顶点为 A，终止顶点为 D。

路径	成本
A-C-D	13
A-B-D	14
A-B-C-D	11





算法从生成第 1 个 PathInfo 对象开始，该对象将 sVertex 与自身连接，置初始成本为 0。将对象插入到优先级队列中。

A 至 A	0

优先级队列

为寻找最短路径，我们采用从优先级队列中反复删除对象的迭代过程。如果对象的终止顶点为 eVertex，则我们已得到最短路径，其成本为数据值 cost。否则，我们要考察当前顶点 endV 的所有邻居以寻找沿另一条边扩展从 sVertex 开始的路径。

在我们的例子中，我们希望找出从 A 到 D 的最短路径。我们删掉一个 PathInfo 对象，

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

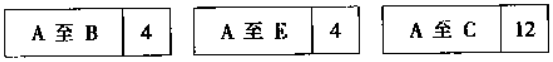
www.Linuxidc.com

其顶点 A 就是 endV。若 A 是所指定的 eVertex，则过程结束，最小成本为 0。若 A 不是终点，将其存于表 L 中。表 L 中包含所有那些已求出从 A 出发到达它们的最短路径的顶点。我们找出 A 的邻接点 B，C 和 E。建立与每个顶点相对应的 PathInfo 对象并将它们放到优先级队列中。从 A 到每个结点的成本为：

A 到邻接点的成本 = cost(A,endV) + weight(endV, < 邻接点 >)

PathInfo 对象	startV	endV	cost
$O_{A,B}$	A	B	4
$O_{A,C}$	A	C	12
$O_{A,E}$	A	E	4

对象以下列次序进入优先级队列：



优先级队列

下一步，从优先级队列中删除对象 $O_{A,B}$。顶点 B 是具有最小成本 4 的 endV。因为 B 不在 L 中，所以将它放到表中。显然，此后 A 到 B 之间不可能再有成本小于 4 的路径。因为若是有的话，我们会有一条子路径

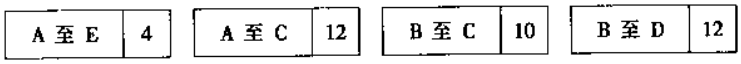
A—X—…—B

其中 X 是与 A 距离小于 4 的邻接点，这样 X 将在 B 之前从优先级队列中删掉。

我们找出 B 的邻点 A，C，D。既然 A 已经在 L 中，我们只关注 C 和 E，建立相应的 PathInfo 对象以进入优先级队列。

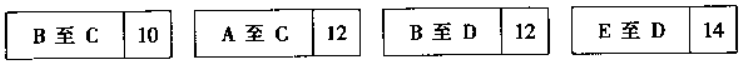
PathInfo 对象	startV	endV	cost
$O_{B,C}$	B	C	10 = 4 + 6
$O_{B,D}$	B	D	12 = 4 + 8

这样，优先级队列中就有了 4 个成员。注意有两个不同的对象以顶点 C 为终点。最小值 10 刚进入优先级队列，它代表路径 A—B—C。直接路径 A—C 在步骤 1 中已标注过，其成本是 12。



优先级队列

删除对象 $O_{A,E}$ 并求出 A 到 E 的最小成本为 4 后，我们建立成本为 14 的对象 $O_{E,D}$。



优先级队列

接下来从优先级队列中删除的对象 $O_{B,C}$ 是最小值，我们可以将 C 加入到表 L 中，因为 10 是从 A 到 C 的最小成本。

假设 endVertex 是 D，则我们有待删除以 D 为终止顶点的对象。处理完结点 C 后，我

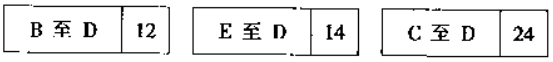
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

们找出其邻接点 B 和 D。因 B 已经处理过，所以只有对象 $O_{C,D}$进入优先级队列。

对象	startV	endV	距离
$O_{C,D}$	C	D	24 = 10 + 14

$O_{A,C}$从优先级队列中被删除后就被舍弃掉，因为 C 已经在表中。优先级队列现在有 3 个元素。



优先级队列

从优先级队列中删除 $O_{B,D}$时，我们就得到了 A 到 D 的最小成本为 12。

```
template <class T>
int Graph<T>::MinimumPath(const T& sVertex, const T& eVertex)
{
    // 存放从 sVertex 开始的路径，成本信息的优先级队列
    PQueue<PathInfo<T> > PQ(MaxGraphSize);
    // 供往优先级队列中插入或删除 PathInfo 使用
    PathInfo<T> pathData;
    // L 中存放所有可从 sVertex 到达的顶点及路径成本。adjL 为正在访问的顶点的邻接顶
    // 点表.adjLiter 用来遍历 adjL
    SeqList<T> L, adjL;
    SeqListIterator<T> adjLiter(adjL);
    T sv, ev;
    int mincost;

    // 初始化优先级队列
    pathData.startV = sVertex;
    pathData.endV = sVertex;
    // 从 sVertex 到 sVertex 的成本为 0
    pathData.cost = 0;
    PQ.PQInsert(pathData);

    // 处理顶点，直到找到至 eVertex 的最小路径或优先级队列为空
    while (! PQ.PQEmpty())
    {
    // 删除优先级队列中的一个顶点并记录其终止顶点和从 sVertex 起始的成本
    pathData = PQ.PQDelete();
    ev = pathData.endV;
    mincost = pathData.cost;

    // 若为已到 eVertex，则已找到从 sVertex 到 eVertex 的最小路径
    if (ev == eVertex)
        break;

    // 若终止顶点已在 L 中，则不需再考虑它
    if (! FindVertex(L,ev))
    {
```

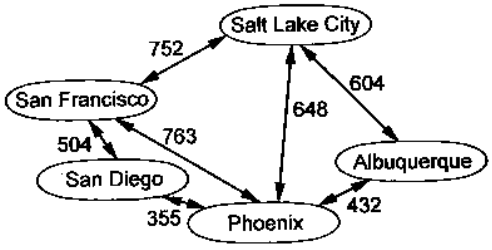

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            // 将 ev 插入到 L
            L.Insert(ev);
            // 搜索当前顶点 ev 的所有邻接顶点,对每个不在 L 中的顶点,将其插入到起始顶点
            // 为 ev 的优先级队列中,遍历新表 adjL
            sv = ev;
            adjL = GetNeighbors(sv);
            // adjLiter 遍历新表 adjL
            adjLiter.SetList(adjL);
            for(adjLiter.Reset();! adjLiter.EndOfList();adjLiter.Next())
            {
                ev = adjLiter.Data();
                if (! FindVertex(L, ev))
                {
                    // 为优先级队列创建新数据
                    pathData.startV = sv;
                    pathData.endV = ev;
                    // 新成本为当前最小成本加上从 sv 到 ev 的成本
                    pathData.cost = mincost + GetWeight(sv,ev);
                    PQ.PQInsert(pathData);
                }
            }
        }
    }
    // 返回成本或失败
    if (ev = = eVertex)
        return mincost;

        else
            return -1;
    }
```

程序 13.6　航空运输系统

航空运输系统列出航线所经过的城市。用户输入始发城市,用最短路径可确定从始发地到所有可能的目的地之间的最短旅行距离。此处的航线到达西部主要城市。




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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
#include <fstream.h>
#include "strclass.h"
// 引入带 MinimumPath 方法的类 graph
#include "graph.h"
void main(void)
{
    // 顶点为字符串(航线经过的城市名称)
    Graph<String>   G;
    String  S;

    // 为运输图输入各顶点
    G.ReadGraph("airline.dat");

    // 提示输入离港城市
    cout << "Give the minimum distance when departing from ";
    cin >> S;
    // 用图迭代算子,扫描城市表并得到从离港城市出发的最短距离
    VertexIterator<String> viter(G);
    for (viter.Reset(); ! viter.EndOfList(); viter.Next())
        cout << "Minimum distance from " << S << " to "
             << viter.Data() << " is "
             << G.MinimumPath(S, viter.Data()) << endl;

/*
<程序 13.6 运行结果之一>
Give the minimum distance when departing from SaltLakeCity
Minimum distance from SaltLakeCity to SaltLakeCity is 0
Minimum distance from SaltLakeCity to Albuquerque is 604
Minimum distancefrom SaltLakeCity to Phoenix is 648
Minimum distance from SaltLakeCity to SanFrancisco is 752
Minimum distance from SaltLakeCity to SanDiego is 1003
<程序 13.6 运行结果之二>
Give the minimum distance when departing from SanFrancisco
Minimum distance from SanFrancisco to SaltLakeCity is 752
Minimum distance from SanFrancisco to Albuquerque is 1195
MiNimum distance from SanFrancisco to Phoenix is 763
Minimum distance from SanFrancisco to SanFrancisco is 0
Minimum distance from SanFrancisco to SanDiego is 504
*/
```

可及性(reachability)和 Warshall 算法

对于图中任一对顶点 V_i 和 V_j,当且仅当从 V_i 到 V_j 有一条有向路径时,我们称 V_j 自 V_i"可及(reachable)"。这样我们就定义了"可及关系(reachable relation)R"。对顶点 V_i,用深度优先搜索得到自 V_i 可及的所有顶点的表。如果我们对图中每个顶点都使用深度优先搜索,我们可以得到标志关系 R 的一系列可及表(reachability list):

V_1:〈V_1 可及表〉

V_2:〈V_2 可及表〉

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

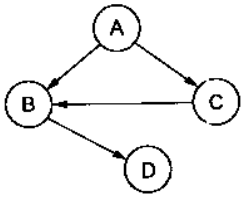
www.Linuxidc.com

…

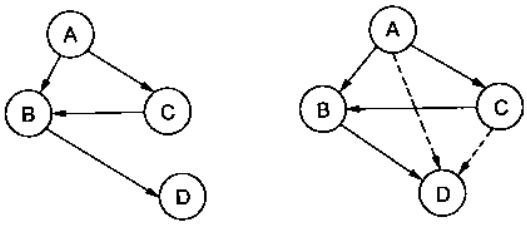
V_n:〈V_n 可及表〉

也可以用 n × n"可及矩阵(reachability matrix)"来描述上述关系,若存在关系 V_i R V_j,则矩阵中位置(i,j)处的值为 1。下面给出一个图,我们定义其可及表和可及矩阵:

可及表	可及矩阵
A:A B C D	1 1 1 1
B:B D	0 1 0 1
C:C B	0 1 1 0
D:	0 0 0 1

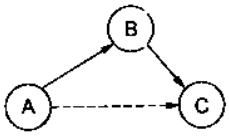


可及矩阵可以用来确定两个顶点之间是否有路径存在。若(i,j)处的元素为 1,则 V_i 和 V_j 之间有一条最短路径。我们可以用可及表中的顶点扩充原始图的边。如果从顶点 v 到 w 之间有路径(w 自 v 可及),我们在图中添加一条连接两个顶点的边 E(v,w)。扩展图 G'由图 G 中的顶点 V 以及连接那些由有向路径连通的顶点的边组成。扩展图被称为"转移闭包(transitive closure)"。例如 G 的转移闭包如下所示:



用深度优先搜索求可及表的问题留作习题。一种更简洁的方法是使用 Stephen Warshall 提出的著名算法。他注意到,图的可及矩阵可以通过以下过程建立起来:对连接到共同顶点的每一对顶点,在矩阵中为其赋值 1。假设我们正在建立可及矩阵 R 且顶点 a,b,c 对应索引 i,k,j。如果 R[i][k] = 1 且 R[k][j] = 1,则 R[i][j] = 1。

Warshall 算法以索引 i,j,k 建立三重嵌套循环,对所有可能的三元组进行检查。对每一对(i,j),若存在顶点 V_k,使得 E(V_i,V_k)和 E(V_k,V_j)都在扩展图中,则增加一条边 E(V_i,V_j)。重复这一过程,加入所有连接各对可及顶点的边。这就得到了可及矩阵。



可以用一个直观的例子说明 Warshall 算法。假设顶点 V 和 W 是可及的,有一条 5 个顶点组成的有向路径将 V 连接到 W。组成路径的顶点序列是

$V = X_1, X_2, X_3, X_4, X_5 = W$

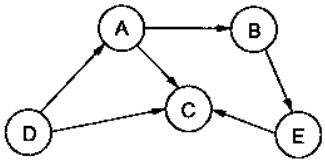
在可及矩阵中,若从 V 到 W 有路径,则我们必须说明用 Warshall 算法最终也能找出同样的路径。我们用三重嵌套循环检查所有可能的顶点三元组。假设顶点 X_1 到 X_5 按以上次序出现。在考察各三元组时,我们发现 X_2 是连接 X_1 和 X_3 的共同结点。因此,我们用 Warshall 算法引入一条新边 E(X_1,X_3)。对于顶点 X_1 和 X_4,X_3 是共同(连接)顶点,因为在前一阶

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

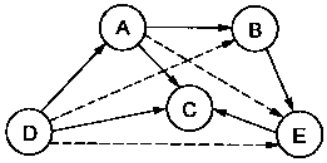
www.Linuxidc.com

段的迭代中已经发现连接 X_1 和 X_3 的路径。这样我们建立边 $E(X_1,X_4)$。继续分析下去，X_4 是连接 X_1 和 X_5 的共同顶点，因而我们增加边 $E(X_1,X_5)$并赋值 $R[1][5]=1$。

我们以下图为例说明 Warshall 算法。图中增加虚线画出的边以形成转移闭包。



初始图



转移闭包

可及矩阵

1	1	1	0	1
0	1	1	0	1
0	0	1	0	0
1	1	1	1	1
0	0	1	0	1

Warshall 算法的时间复杂度为 $O(n^3)$。在扫描邻接矩阵时，我们用的是三重嵌套循环。即使用表表示图也会产生 $O(n^3)$的复杂度。

程序 13.7　Warshall 算法的使用

本程序用 Warshall 算法建立和打印可及矩阵。

```
#include <iostream.h>
#include <fstream.h>
#include "graph.h"
template <class T>
void Warshal(Graph<T>& G)
{
    VertexIterator<T> vi(G), vj(G);

    int i, j, k;
    int WSM[MaxGraphSize][MaxGraphSize];  // Warshall 矩阵
    int n = G.NumberOfVertices();

    // 产生初始矩阵
    for (vi.Reset(), i=0;! vi.EndOfList(); vi.Next(), i++)
        for(vj.Reset(), j=0; ! vj.EndOfList(); vj.Next(), j++)
            if (i == j)
                WSM[i][i] = 1;
            else
                WSM[i][j] = G.GetWeight(vi.Data(), vj.Data());

    // 考察所有三元组，当有一条边从 vi 到 vj 或存在一个三元组 Vi-Vk-Vj 连接 Vi 和 Vj 时，
    // 将 WSM 赋值为 1
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            for (k = 0; k < n; k++)
                WSM[i][j] |= WSM[i][k] & WSM[k][j];

    // 输出每个顶点及其对应的可及矩阵行
    for (vi.Reset(), i = 0;! vi.EndOfList();vi.Next(), i++)
    {
        cout << vi.Data() << ":  ";
        for (j = 0; j < n; j++)
            cout << WSM[i][j] << " ";
        cout << endl;
    }
}
void main(void)
{
    Graph<char> G;

    G.ReadGraph("warshall.dat");

    cout << "The reachability matrix is:" << endl;
    Warshall(G);
/*
<程序 13.7 运行结果>
The reachability matrix is:
A:  1  1  1  0  1
B:  0  1  1  0  1
C:  0  0  1  0  0
D:  1  1  1  1  1
E:  0  0  1  0  1
*/
```

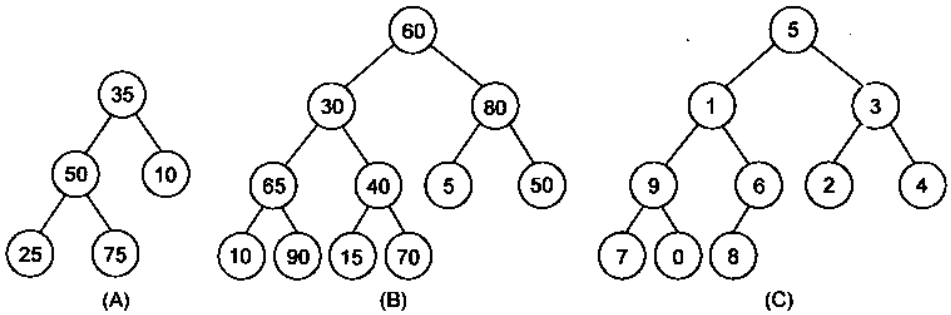
书面作业

13.1 画出下列各数组所对应的完全树：

(a) int A[8] = {5,9,3,6,2,1,4,7};

(b) char * B = "array – based tree";

13.2 求以下各树对应的数组：



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

13.3 为基于数组的树编写(a)前序和(b)中序递归扫描函数。参照为链式树结点所编写的代码。

13.4 假设 A 是有 70 个成员的基于数组的树。

(a) A[4]是叶子结点吗?

(b) 第 1 个叶子结点的下标是多少?

(c) A[50]的双亲是谁?

(d) A[10]的孩子是谁?

(e) 有没有仅有 1 个孩子的表项?

(f) 树的深度是多少?

(g) 树有多少叶子结点?

13.5 (a) 编写一个函数,以类型为 T 的 N 个元素的数组为参数,对相应的基于该数组的树进行逐层扫描。所编例程必须用队列存储元素。输出所有元素,每换一层时亦进行换行。

(b) 你能利用树用数组表示这一事实设计出更简单的逐层扫描算法吗?

13.6 解释为什么:(1) 完全二叉树中叶子结点数大于或等于非叶子结点数;(2) 满二叉树中叶子结点数多于非叶子结点数。

13.7 完全二叉树 B 中含 50 个结点,代表一个数组。

(a) 树的层数是多少?

(b) 有多少个叶子结点? 非叶子结点?

(c) B[35]的双亲的索引是什么?

(d) 结点 B[20]的孩子的索引是什么?

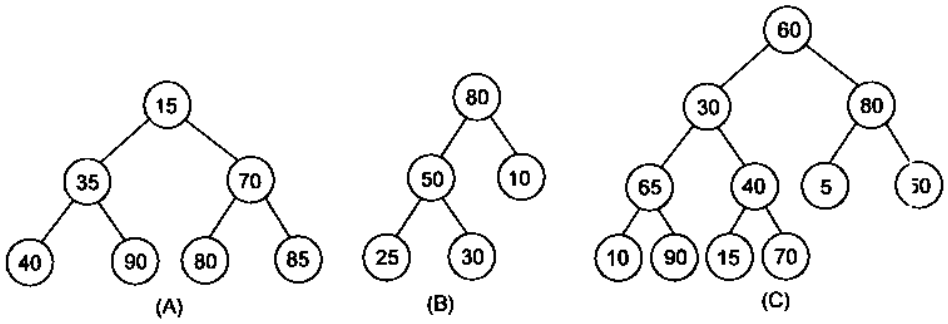
(e) 第 1 个没有孩子的结点的索引是多少? 1 个孩子的呢?

(f) 树中第 4 层各结点的索引是多少?

13.8 写出用竞赛排序法对数组 A = {7,10,3,9,4,12,15,5,1,8}进行排序的各步骤。

13.9 修改竞赛排序法,使得叶子结点和非叶子结点都包含数组中的实际数据。将修改后的代码放到函数 ModTournamentSort 中。

13.10 说明下列各树是否为堆(最大或最小)。



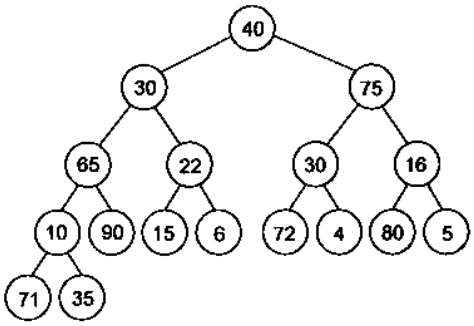
13.11 为书面作业 13.10 中的每一棵非堆树同时建立最大堆树和最小堆树,而为每一最

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

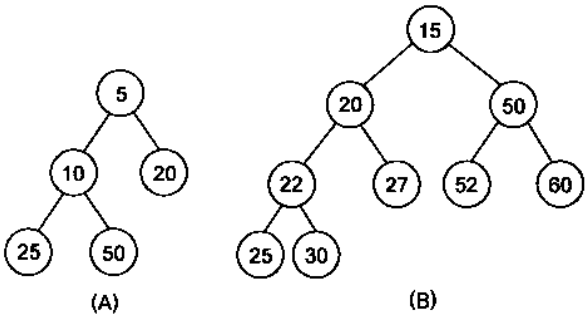
www.Linuxidc.com

小(最大)堆树建立相应的最大(最小)堆树。

13.12 用 FilterDown 和构造函数算法"堆化"以下树并建立最小堆:



13.13 本题插入和删除堆(A)和(B)中的元素。每个堆都有一系列要顺序执行的相应操作。执行每步操作时都要用到前次操作的结果。



堆(A)	堆(B)
(a) 插入 15	(a) 删除 22
(b) 插入 35	(b) 插入 35
(c) 删除 10	(c) 插入 65
(d) 插入 40	(d) 删除 15
(e) 插入 10	(e) 删除 27
	(f) 插入 5

13.14 (a) 如果一棵树既是最小堆树又是二叉搜索树,那么它可能拥有的最多结点数是多少(不允许重复值)?

(b) 如果一棵树既是最大堆树又是二叉搜索树,那么它可能拥有的最多结点数是多少(不允许重复值)?

13.15 对以下堆,列出指定路径上的结点值:

(a) 从结点 47 开始,往双亲方向的路径

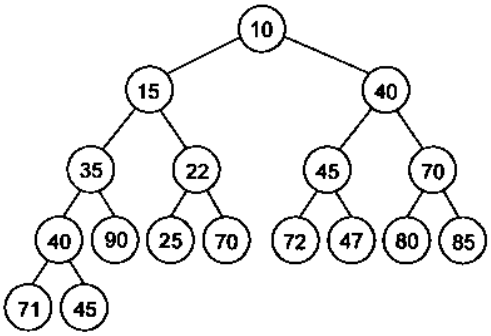
(b) 从结点 71 开始,往双亲方向的路径

(c) 从结点 35 开始,往最小孩子方向的路径

(d) 从结点 10 开始,往最小孩子方向的路径

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



(e) 从结点 40(第 1 层)开始,往最小孩子方向的路径

13.16 对以下数组,建立相应的最小堆。

(a) int A[10] = {40, 20, 70, 30, 90, 10, 50, 100, 60, 80};

(b) int A[8] = {3, 90, 45, 6, 16, 45, 33, 88};

(c) char * B = "heapify"

(d) char * B = "minimal heap";

13.17 用 BinSTree 类实现优先级队列类 PQueue。(提示:修改 PQdelete 方法以搜寻最小元素并将其从树中删除。)

13.18 对以下数据,构造由插入表项得到的 AVL 树。

(a) < int > 30,50,25,70,60,40,55,45,20

(b) < int > 44,22,66,11,0,33,77,55,88

(c) < int > 1,2,3,4,5,6,7,8,9,10

(d) < String > tree, AVL, insert, delete, find, root, search

(e) < String > class, object, public, private, derived, base, inherit, method, constructor, abstract

13.19 编写一个迭代的前序扫描函数 Preorder_I。它应仿真以下的递归函数 Preorder。为帮助你设计算法,我们给出一些提示。

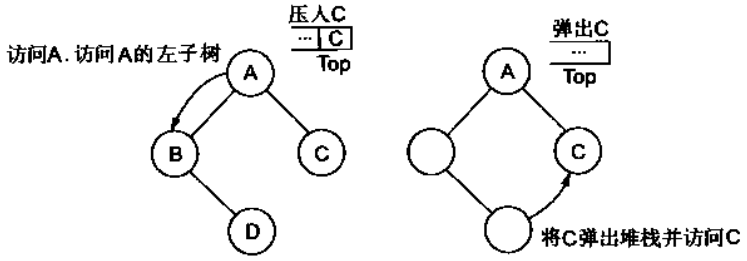
```
template < class T >
void Preorder (TreeNode < T > * t, void visit(T&,item))
{
    while (t ! = NUL)
    {
        visit(t - > data);
        Preorder(t - > Left(), visit);
        t = t - > Right();
    }
}
```

对每一个结点,它自身被访问过后,接下来被访问的是其左子树,然后是右子树。迭代的前序扫描要仿真递归扫描时,必须用堆栈存放结点地址。假设我们现位于二叉树的结点 A 处并已对它进行了访问。然后我们必须要访问 A 的左子树,

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

此后再返回并访问 A 的右子树。为了记住还必须访问右子树，将 A 的右指针存入(压入)堆栈。访问完 A 的左子树中的所有结点后，堆栈进行弹出操作，我们回过头来扫描右子树。这两种情况如下图所示。

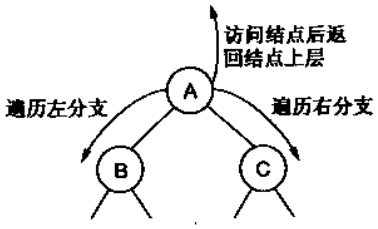


迭代扫描算法按以下步骤执行：从根结点开始，用一个循环前序遍历树。

1. 访问结点。
2. 将结点的右孩子的地址存放到结点地址堆栈中。
3. 转到左孩子。

一旦循环中遇到 NULL 结点，则堆栈进行弹出操作以处理新的右孩子。当遇到 NULL 指针而堆栈中再也没有右孩子时循环终止。

13.20 函数 Postorder_I 以后序方式遍历树。这个问题要比中序或前序遍历难，因为我们必须分转入右分支(状态 0)和返回双亲结点(状态 1)。在树中往上走时，有两种可能的动作——访问结点的右分支或访问结点。此时的状态用整型变量 state 记录。若 state = = 0，则动作是往下走。若 state = = 1，动作是往上。当往上走时，当前结点的双亲位于栈顶。为了确定我们是否从左子树过来，将结点指针与双亲的左指针进行比较。如果它们一致且双亲有右子树，则转到该子树；否则访问结点并继续往上走。



画出一系列类似图 13.5 和图 13.6 中的图以示意下列程序中的算法是如何工作的。

```
#include <iostream.h>
#include "treenode.h"
#include "treelib.h"
#include "stack.h"
void printchar(char& item)
{
    cout << item << " ";
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
template < class T >
void Postorder_I(TreeNode < T >  * t, void visit(T& item))
{

    Stack < TreeNode < T >  * S;
    TreeNode < T >  * child;
    int state = 0, scanOver = 0;

    while (! scanOver)
    {
      if (state = = 0)
      {
          if (t ! = NULL)
          {
            S.Push(t);
            t = t - > Left();
          }
          else
            state = 1;
      }
      else
      {
          if (S.StackEmpty())
            scanOver = 1;
          else
          {
            child = t;
            t = S.Peek();
            if (child = = t - > Left() && t - > Right() ! = NULl)
            {
              t = t - > Right();
              state = 0;
            }
            else
            {
            visit(t - > data);
            S.Pop();
            }
        }
      }
    }
}
void main(void)
{
    TreeNode < char >  * root;

    MakeCharTree(root, 0);
    PrintTree(root,0);
    cout < < endl;
    Postorder_I(root, printchar);
    cout < < endl;
}
```

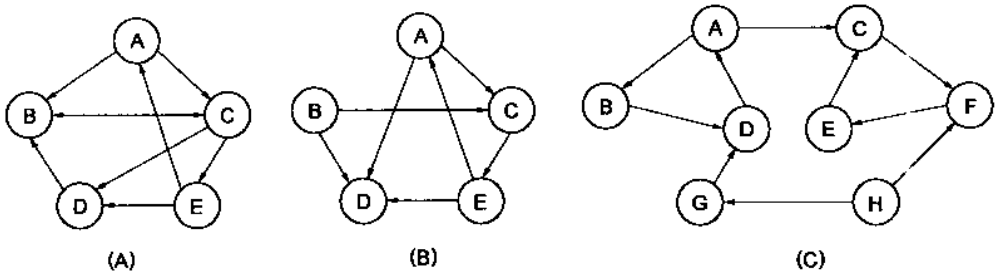
< 运行 >

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        C
            E
A
            D
        B
D   B   E   C   A
```

13.21 求以下各有向图的邻接矩阵和邻接表：



13.22 根据书面作业 13.21 中的图，回答下列有关路径的问题。如果路径不存在，则回答无路径。

(a) 在(A)中，找出从 E 到 B 的有向路径。
(b) 在(B)中，找出从 A 到 E 的有向路径。
(c) 在(C)中，找出从 B 到 E 以及从 E 到 B 的有向路径。
(d) 在(B)中，找出连接到 A 以及连接到 E 的所有结点。
(e) 在(C)中，找出满足以下条件的所有结点 X：存在路径 P(A,X)和 P(X,A)。
(f) 列出每个图中的连通分量。
(g) 哪个图是强连通的？哪个是弱连通的？

13.23 根据书面作业 13.21 中的图回答下列问题。按深度优先搜索和广度优先搜索的遍历次序列出结点。

(a) 在(A)中，从顶点 A 出发。
(b) 在(B)中，从顶点 A 出发。
(c) 在(C)中，从顶点 A 出发。
(d) 在(B)中，从顶点 B 出发。

13.24 部分实现用邻接表表示的 Graph 类。定义类 VertexInfo，其中包含顶点名和存储其邻接结点信息的链表。邻接结点的信息可以存放在一个结构中，该结构包括终止顶点名以及连接起始顶点和终止顶点的边的权值。必须将重载的比较运算符“==”加入到类 VertexInfo 实现构造函数。

(a) 实现构造函数。
(b) 实现方法 GraphEmpty，NumberOfVertices，GetNeighbors，InsertVertex，InsertEdge 和 DepthFirstSearch。

13.25 (a) 对函数 MinimumPath 进行修改，编写函数 MinimumLength，它找出从起始顶点

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

到终止顶点的路径上的最少结点数。若路径不存在，则返回 -1。

(b) 编写函数 VertexLength，它以图 G 和起始顶点 V 为参数，打印出 G 的顶点以及它们距 V 的长度。用(a)中的函数 MinimumLength。

13.26 描述以下函数的动作：

```
template <class T>
SeqList<T> RV(Graph<T> &G)
{
    SeqList<T> L;
    VertexIterator<T> liter(L);

    for (viter.Reset(); ! viter.EndOfList(); viter.Next())
    {
    cout << viter.Data() << ":      ";
    L = G.BreadthFirstSearch(viter.Data());
    liter.SetList(L);
    PrintList(L);
    cout << endl;
    }
}
```

13.27 画出以下各邻接矩阵所对应的图。假设顶点是字符 A,B,…依此类推。

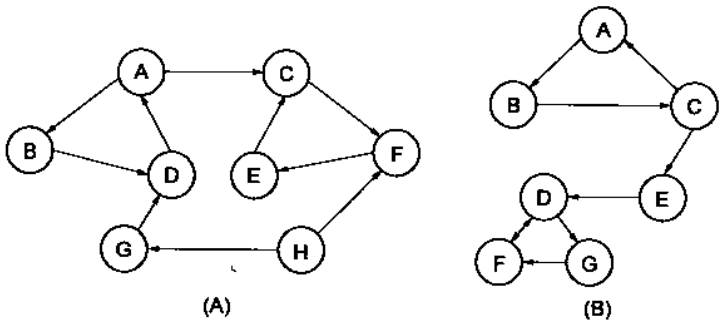
(a)

0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

(b)

0	1	1	0	0
0	0	0	1	0
1	0	0	0	1
1	0	0	1	0
0	1	0	1	0

13.28 求以下各图的可及矩阵。



上机题

13.1 考虑对书面作业 13.3 中的基于数组的树进行前序扫描。用函数求叶子结点的数目、只有 1 个孩子的结点数目以及有 2 个孩子的结点数目。在测试和打印全局数

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

据值的程序中用全局变量 nochild,onechild 以及 twochild 存储相应的信息：

```
int A[50];     // 存放数据 0, 1, 2, …, 49
int P[100];     // 存放数据 0, 1, 2, …, 99
```

13.2 (a) 在一个测试程序中,用竞赛排序法对以下串进行排序。建立 String 对象的数组并将它们传递给 TonrnamentSort 函数。

class, object, public, private, derived, base,
inherit, method, constructor, abstract

(b) 书面作业 13.9 中曾要求编写修改后的比赛排序函数 ModTournamentSort。用此排序法建立范围在 0 到 999 之间的 100 个随机整数组成的数组并按从小到大的次序对它们进行排序。打印结果数组。

13.3 修改 Heap 类以实现最大堆。调用新类 MaxHeap,并用对数组元素进行排序的程序 13.3 的变异版本测试其操作。

13.4 用递归法实现 Heap 类的 FilterUp 和 FilterDown 例程并用以下方法测试:用 0 到 99 范围内的 15 个随机整数的数组初始化堆。用 HDelete 例程从堆中删除 N 个元素并打印出它们的值。

13.5 计算机系统通过为每个进程分配一个优先级的方法运行程序(进程)。优先级 0 代表最高优先级而 39 代表最低优先级。当要执行一个进程时,往优先级队列中插入一个进程请求记录。当 CPU 有空时,删除进程记录,运行进程。进程请求记录的声明如下：

```
struct ProcessRequestRecord
{
    int priority;
    String name;
};
```

name(名字)域对进程标识。随机生成 10 个进程记录,其名字分别为“进程 A”,“进程 B”,…,“进程 J”,而 priority(优先级)的值在 0 到 39 之间。打印各个记录,并将其插入到优先级队列中。然后从队列中删除各记录并打印出来。

13.6 定义一个包含数据和优先级的结构。

```
template <class T>
struct PriorityData
{
    T data;
    int priority
}
```

用该结构和优先级队列实现 Queue 类。(提示:将队列声明为 PriorityData 记录的表。这些表项被存储在根据各记录优先级排序的优先级队列中。定义整数 PL,当有一表项插入到表中时,该整数值会增 1。PL 被指定为记录的优先级。

13.7 参照上机题 13.6 并用上机题 13.3 中的 MaxHeap 类以优先级队列法实现堆栈。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

为测试新堆栈类，先读取5个整数。用Push将数据值存入堆栈。用Pop删除表项并打印其值，直到栈为空。

13.8 设计迭代算子ArrPreorderIterator，使之前序遍历基于数组的树。其声明如下：

```
template < class T >
class ArrPreorderIterator: public Iterator < T >
{
    privte:
        Stack < int > S;
        T * A;
        int arraySize;
        int current;
    public:
        ArrPreorderIterator(T * Arr, int n);
        virtual void Next(void);
        virtual void Reset(void);
        virtual T& Data(void);
};
```

为测试此类，试遍历由数组

```
int A[15] = {1,4,6,2,8,9,12,25,23,55,18,78,58,14,45};
```

所确定的树并以层次顺序和前序将其打印出来。

13.9 一元运算符"++"可以作为成员函数被重载。用此运算符可以很容易地实现树迭代算子成员函数Next。下面是此运算符的声明和使用举例。

```
void operator++ (void);
```

遍历一棵二叉搜索树。

```
BinSTree < Type > * tree;
      ...
InorderIterator < T > inorderIter(tree.GetRoot());
while(! inorderIter.EndOfList())
{
      ...
      inorderIter++;
}
```

用"++"实现InorderIterator类中的Next方法并用程序对其进行测试，程序运行如下：往二叉搜索树中放入25个随机整数并用树排序函数对其进行排序。

13.10 设计迭代算子类LevelOrderIterator，使之以层次顺序遍历二叉搜索树。类声明如下：

```
template < class T > : public Iterator < T >
class LevelorderIterator
{
    private:
        Queue < TreeNode < T > * > S;
        TreeNode < T > * root, * current;
    public:
```

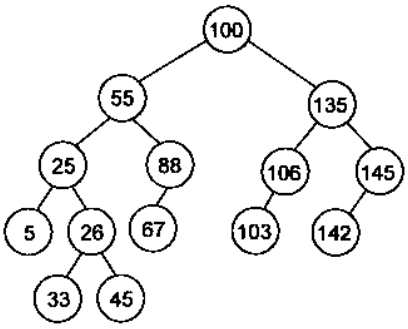

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        LevelorderIterator(TreeNode<T> * lst);
        virtual void Next(void);
        virtual void Reset(void);
    virtual T& Data(void);
};
```

在主程序中,用以下数据建立二叉搜索树

```
int data[] =
{100, 55, 135, 145, 25, 106, 88, 90, 5, 26, 67, 45, 99, 33};
```



用 PrintTree 打印树,然后用 LevelOrderIterator 遍历树。

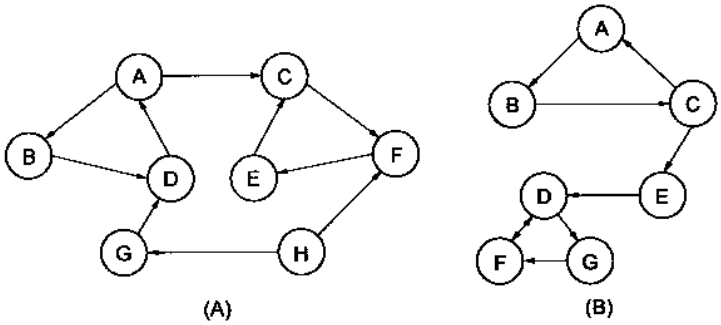
13.11 按以下算法设计类 PostOrderIterator:

初始化 NRL 迭代算子时,将所有结点的地址压入堆栈。在遍历期间,当前结点总是在栈顶,下一个结点由堆栈弹出操作获得。

测试迭代算子,方法是遍历并打印上机题 13.10 中所用的树中的数据。

13.12 采用书面作业 13.20 中所提出的后序遍历算法设计类 PostOrderIterator。扫描上机题 13.10 中所给树,对迭代算子进行测试。

13.13 用 ReadGraph 为图 A 和 B 建立一个数据文件。写一个主程序,输入数据并使用书面作业 13.25 中的函数 VertexLenght 打印 A 到各顶点的长度。对每个图都运行程序进行测试。



13.14 用上机题 13.13 中的图和 Warshall 算法打印各个图的可及矩阵。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第14章 群体数据的组织

14.1 数组排序的基本算法
14.2 快速排序(QuickSort)
14.3 哈希法(Hashing)
14.4 哈希表类
14.5 搜索方法的性能
14.6 二进制文件和外部数据操作
14.7 辞典
书面作业
上机题

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

这一章我们讨论一般性的数据组织问题，以此结束全书内容。在此过程中，我们要设计一系列经典算法对数据进行排序。在前面的章节中，我们在说明一般表结构时引入了排序，例如，作为队列的应用的基数排序(radix sort)。现在我们要集中讨论数组排序，介绍经典的选择排序、冒泡排序和插入排序，它们都要进行 $O(n^2)$ 次比较。虽然这些算法对于实际的数据量较大的情况不大实用，但我们可以从中了解到数组排序的主要方法。最后将介绍著名的快速排序(QuickSort)。

第4章介绍了代表基本表搜索算法的顺序和二分搜索算法。本章我们将研究另一种搜索法——哈希法(hashing)，这种算法用一个键值提供对元素的快速访问，其复杂度为 $O(1)$。

本书前面各章节重点针对的是内存中的数据。对于较大的数据群体，数据可能存于磁盘上并需要外部访问方法。为此我们设计 BinFile 类以处理二进制文件并用其方法说明带索引的外部顺序搜索和外部归并排序算法。

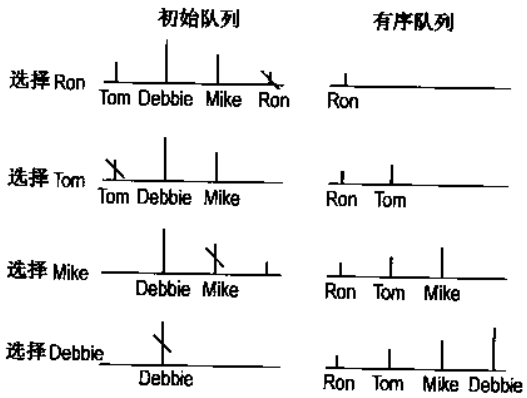
本章将有一节讨论关联数组(associative arrays)或辞典。这一节将数组索引的概念进行了推广。这个概念允许我们用非整数索引组织数据。我们用关联数组建立一个小的单词辞典。

14.1 数组排序的基本算法

我们先介绍3个算法，它们涵盖了以升序进行原地排序的主要技术，对每一种情况，我们求出其计算效率。

选择排序

选择排序所遵循的过程来自于我们的经验。幼儿园老师通常用选择法将孩子们按身高排队。对以随机顺序排列的一组学生，老师反复从组中挑选出最矮的学生并将他或她移到正在形成的按高矮个排列的队列中。这一过程一直持续到所有学生都被移动到有序队列中，如以下一系列图所示。



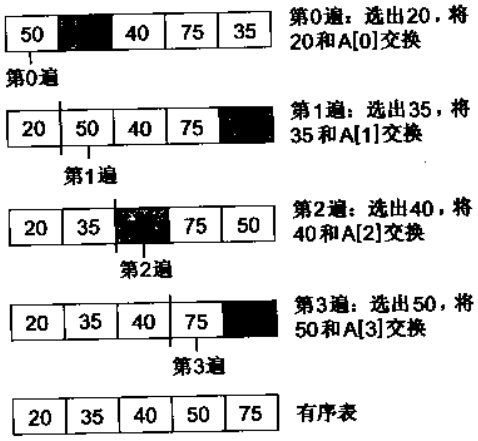
若写成计算机算法，我们假定几个数据项被存于数组 A 中且对表进行 n－1 遍操作，在第0遍，我们选出表中最小元素并将其与表中第1个元素 A[0]交换。第0遍结束后，表头(A[0])已排好序，而表尾(A[1]到 A[n－1])仍未排好序。第1遍检查未排好序的表

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

尾并选出其最小元素，该元素将被存到 A[1]中。第 2 遍则找出了表 A[2]到 A[n-1]中的最小元素并将它与 A[2]交换。上述过程持续进行到第 n-1 遍，那时表尾将缩减为单个元素(表中最大者)且整个数组已排好序。

考察数组 A 中含 5 个整数值 50,20,40,75 和 35 时的选择排序算法：



在第 I 遍，选择过程扫描子表 A[I]到 A[n-1]并将 SmallIndex 设为最小元素的索引。扫描完成后，元素 A[I]和 A[SmallIndex]交换位置。SelectionSort 函数及支撑函数 Swap 在文件“arrsort.n”中给出。

```
// 用选择排序算法对类型为 T 的 n 元数组进行排序
template <class T>
void SelectionSort(T A[], int n)
{
    // 每遍循环中最小元素的下标
    int smallIndex;
    int i, j;
    // 对 A[0]..A[n-2]进行排序后,A[n-1]也将有序.
    for (i = 0; i < n-1; i++)
    {
        // 从下标 i 开始扫描,将 smallIndex 置为 i
        smallIndex = i;
        // 用 j 扫描子表 A[i+1]..A[n-1]
        for (j = i+1; j < n; j++)
          // 若找到更小元素,则修改 smallIndex 的值
          if (A[j] < A[smallIndex])
              smallIndex = j;
        // 结束后,将最小元素放入 A[i]
        swap(A[i], A[smallIndex]);
    }
}
```

选择排序的计算复杂度 选择排序需要进行的比较次数是固定的，它仅仅取决于数组的大小，而不取决于数据的初始分布。在第 i 遍，子表 $A[i+1]$到 $A[n-1]$所需比较次数为

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

$(n-1)-(i-1)+1=n-i-1$

总的比较次数是

$$\sum_{0}^{n-2}(n-1)-i=(n-1)^2-\sum_{0}^{n-2}i$$
$$=(n-1)^2-(n-1)(n-2)/2$$
$$=\frac{1}{2}n(n-1)$$

算法复杂度按比较次数衡量是 $O(n^2)$，交换次数为 $O(n)$。既没有最好的情况也没有最坏的情况，因为算法进行固定遍数的循环，且每一遍扫描的元素个数也是确定的，复杂度为 $O(n\log_2 n)$的堆排序是选择排序的推广。

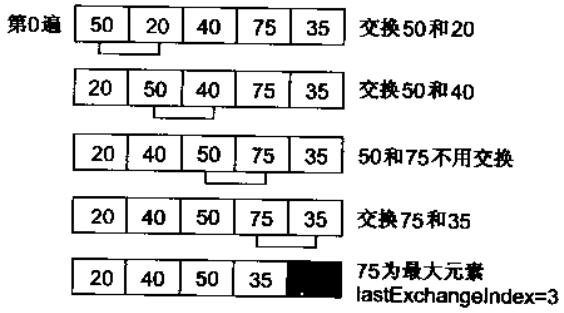
冒泡排序

第 2 章中我们介绍了交换排序。这种排序法进行 n－1 遍循环，且每一遍需要的比较次数是固定的。这一节我们讨论冒泡排序，它也是在每一遍进行一系列内部交换。

对于 n 个元素的数组 A，冒泡排序至多需要 $n-1$ 遍循环。每一遍循环中，我们将各相邻元素进行比较，若前一个元素大于后一个元素时则交换其值。最后一遍结束时，最大的元素将“往上冒泡”到当前子表的顶部。例如，第 0 遍结束后，表尾（$A[n-1]$）已排好序，而表头仍未排好序。

我们来看看各遍循环的详细情况。在整个过程中，我们维护交换中涉及到的最后一个索引的记录。变量 lastExchangeIndex 就是用于此目的，并在每一遍开始时被置为 0。第 0 遍比较各对相邻元素（A[0]，A[1]），（A[1]，A[2]），…，（A[n－2]，A[n－1]）。对每一对元素（A[i]，A[i＋1]），若 A[i＋1]＜A[i]则交换其值并将 lastExchangeIndex 更新为 i。这一遍结束时，最大的元素为 A[n－1]，lastExchangeIndex 的值表示从 A[lastExchangeIndex＋1]到 A[n－1]之间的所有表尾元素已排好序。对后面的各遍循环，我们只需比较子表 A[0]到 A[lastExchangeIndex]的相邻元素。当 lastExchangeIndex＝0 时过程终止。算法最多进行 n－1 遍循环。

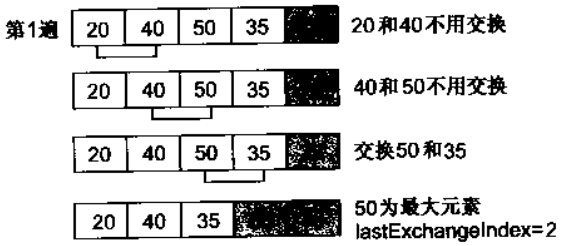
我们用 5 个元素的数组 A＝50，20，40，75，35 说明冒泡排序算法。



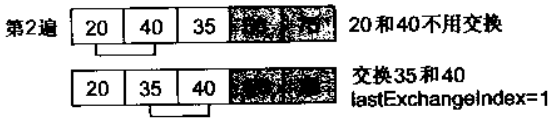
因为 lastExchangeIndex 不为 0，过程继续。在第 1 遍循环中，我们扫描 A[0]到 A[3]＝A[lastExchangeIndex]的元素子表。lastExchangeIndex 的新值为 2。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

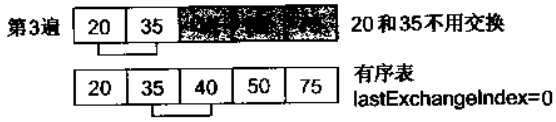
www.Linuxidc.com



第 2 遍循环中，我们扫描 A[0]到 A[2]之间的子表并交换 40 和 35。lastExchangeIndex 的值变为 1。



第 3 遍，只需对 20 和 35 进行一次比较。没有交换，lastExchangeIndex 为 0，过程终止。




```
// BubbleSort 的参数为数组 A 和表元素个数 n.它通过多遍交换直到 lastExchangeIndex = 0
// 来将数据排序.
template <class T>
void BubbliSort (T A[], int n)
{
    int i, j;
    // 最后一次交换时的下标值
    int lastExchangeIndex;

    // i 为子表中最后一个元素的下标
    i = n-1;

    // 继续处理直到不再需要交换
    while (i > 0)
    {
        // 从 lastExchangeIndex = 0 开始
        lastExchangeIndex = 0;
        // 扫描子表 A[0]到 A[i]
        for (j = 0; j < i; j++)
            // 交换相邻数据并修改 lastExchangeIndex
            if (A[j+1] < A[j])
            {
                swap(A[j], A[j+1]);
                lastExchangeIndex = j;
            }
        // 置 i 值为最后一次交换发生的下标,继续对 A[0]到 A[i]排序
        i = lastExchangeIndex;
    }
}
```

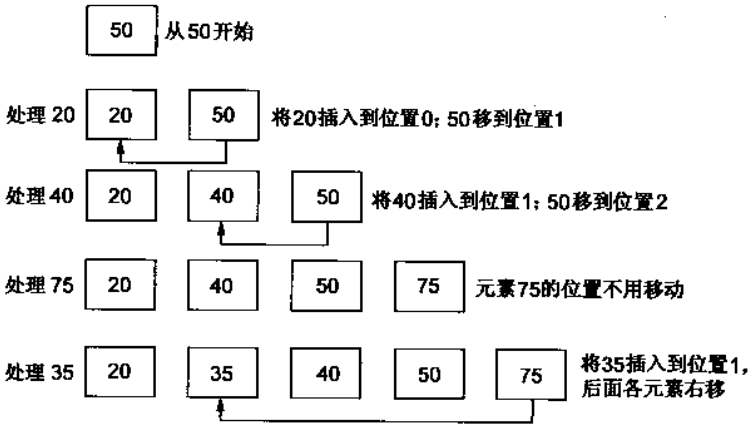

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

冒泡排序的计算复杂度 因为冒泡排序维护最后一次交换的记录，所以无需进行冗余的比较。这使得特定情况下算法效率显著提高。最明显的是，对于已经按升序排列的元素表，冒泡排序只需一遍扫描。因此，最好的情况是 $O(n)$。最坏的情况是表以降序排列，这时冒泡排序的效率最低。整个过程需要全部 $n-1$ 遍扫描。在第 i 遍扫描中，共发生 $(n-i-1)$ 次比较，以及 $(n-i-1)$ 次交换。整个排序需要 $n(n-1)/2$ 次比较以及与之相同次数的交换。这种最坏情况的复杂度是 $O(n^2)$ 次比较，$O(n^2)$ 次交换。对一般情形的分析则要复杂一些，因为扫描的遍数可能要少一些。可以证明，第 k 遍的平均比较次数是 $O(n)$，因而总的比较次数 $O(n^2)$。即使冒泡排序可能用不了 $(n-1)$ 遍扫描就结束了，也因其通常需要比选择排序更多的交换次数而使其平均性能较低。选择排序一般要优于冒泡排序，因为它需要的交换次数较少。

插入排序

插入排序类似于我们所熟悉的要对一串名单进行排序的“洗牌”过程。管理注册者将每个人名写在一个 3×5 的卡片上，然后按字母顺序重新排列卡片，方法是将卡片往前移直到找到合适的位置。在操作进行过程中，卡片堆的前部是排好序的，而尾部有待处理。我们用含 5 个整数值的表描述这一过程：$A=50,20,40,75,35$。



函数 InsertionSort 需要的参数是数组 A 和表的大小 n。我们看一下第 $i(1\leqslant i\leqslant n-1)$ 遍的情况。子表 $A[0]$ 到 $A[i-1]$ 已按升序排列。这一遍将 $A[i]$ 分配到表中。令 $A[i]$ 为目标值，往表的前部移动，将目标值与数据项 $A[i-1]$，$A[i-2]$ 等进行比较。在小于或等于目标值的头一个元素 $A[j]$ 或表头 $(j=0)$ 处停止扫描。我们在表中往前移动时要将所遇到的每个元素往右挪一个位置 $(A[j]=A[j-1])$。当找到 $A[i]$ 的正确位置时，将其插入到位置 j。

```
// 用插入排序法对 A[0]...A[i],1 < = i < = n-1 排序,对每一个 i,将 A[i]插入到正确位置
// A[j]中
template < class T >
void InsertionSort(T A[], int n)
{
    int i, j;
    T temp;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // i 标明的子表 A[0]到 A[i]
    for (i = 1; i < n; i++)
    {
        // 用下标 j 从 A[i]往前扫描子表,为 A[i]找到适当位置后,将 A[i]赋给 A[j]
        j = i;
        temp = A[i];
        // 若 temp < A[j-1]且还未到表头,继续往前扫描
        while ( j > 0 && temp < A[j-1])
        {
            // 右移表中元素
            A[j] = A[j-1];
            j--;
        }
        // 找到位置;将 temp 插入
        A[j] = temp;
    }
}
```

插入排序的计算复杂度 插入排序需要固定遍数的扫描。将元素 $A[1]$到 $A[n-1]$插入共需 $n-1$ 遍扫描。一般地,在第 i 遍,插入发生于子表 $A[0]$到 $A[i]$,平均需要 $i/2$ 次比较。总的比较次数为

$$1/2+2/2+3/2+\cdots+(N-2)/2+(N-1)/2=N(N-1)/4$$

与其他排序不同,插入排序不需要交换。按比较次数衡量,算法的复杂度为 $O(n^2)$。最好的情况是原始表已经排好序。在第 i 遍,插入在 $A[i]$处发生,总的比较次数为 $n-1$,复杂度为 $O(n)$。最坏的情况是原始表已按降序排序。每次插入都发生于 $A[0]$处且需 i 次比较。总的比较次数为 $n(n-1)/2$,复杂度为 $O(n^2)$。

14.2 快速排序(QuickSort)

本章到目前为止,我们已经设计了一系列对数组元素进行原地排序的算法,其计算效率均为 $O(n^2)$。与它们相比,基于树的算法(竞赛、树)具有 $O(n\log_2 n)$量级的性能,明显要高一筹。尽管它们使用数组时需要往树中来回复制元素,但排序效率的提高足以抵消这些额外开销。堆排序是被广泛使用的复杂度为 $O(n\log_2 n)$的原地数组排序方法。但是由 C.J.A.Hoare 所发明的快速排序(QuickSort)法在大多数排序应用中的性能都超过了堆排序。这是已知最快的排序方法。

快速排序的原理

与多数排序算法一样,快速排序也是由我们所熟知的经验得来的。若我们要将一摞作业纸按姓名顺序堆放,我们可以用某个中心字符对表进行分割,将作业分为两堆。所有名字小于或等于 K 的放在一堆,其余的放在另一堆。然后我们再将第一堆又分作两部分。例如,图 14.1 中,分割点为'F'和'R'。如此继续下去,我们可以将各堆分得越来越小。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

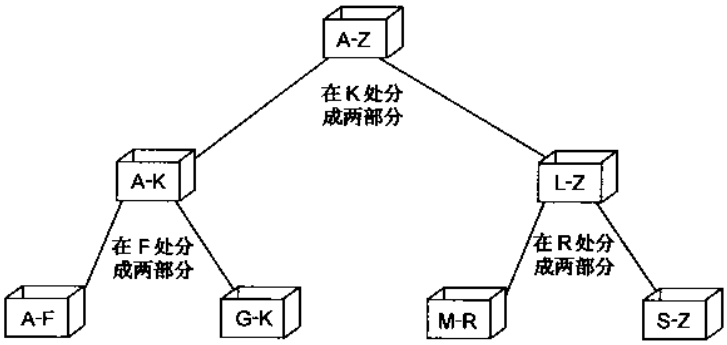
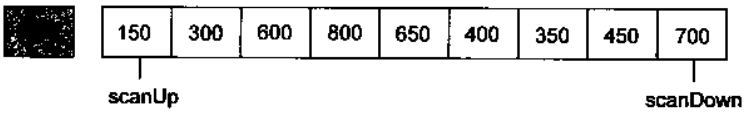


图 14.1　快速排序

快速排序使用分割法对表进行排序。算法定出一个中心值，用它将表分为两部分。与作业纸的排列不同，我们不能奢望有能铺放各堆东西的地板。快速排序在数组为将元素分为若干部分。我们用一个例子来介绍该算法然后再加入技术细节。假设数组 A 中含 10 个整数值：

$$A = 800,150,300,600,550,650,400,350,450,700$$

扫描阶段　我们要扫描 A[0]到 A[9]范围内的全部元素。索引范围是从 low = 0 到 high = 9，中间索引为 mid = 4。第一个中间值为 A[mid] = 550，算法将 A 中元素划分到两个子表 S_l 和 S_h 中。子表 S_l 是低端子表，它所包含的元素值小于或等于中心值。既然我们知道 S_l 将以中心值结尾，我们就暂时将它放在低端，即将其值与 A[0](A[low])交换。这就使得我们可以用两个索引 scanUp 和 scanDown 扫描子表 A[1]到 A[9]。变量 scanUp 初始时被置为索引 1(low + 1)，它负责定位子表 S_l 中的元素。变量 scanDown 被设置为索引 9(high)，它负责定位子表 S_h 的元素。这一遍扫描的目的是确定各个子表的元素。



快速排序的独创之处在于两个索引在扫描表期间的交互作用。scanUp 往表的高端移动，而 scanDown 则往表的低端移动。我们将 scanUp 往右移，以寻找比中心值大的元素 A[scanUp]。一旦找到，扫描中止，准备将元素重新放置到高端子表中。在重新定位以前，我们将索引 scanDown 往表的低端移动，等它找出一个小于等于中心值的元素。这样，两个子表中各有一个错位的元素，因而可以将其进行交换。

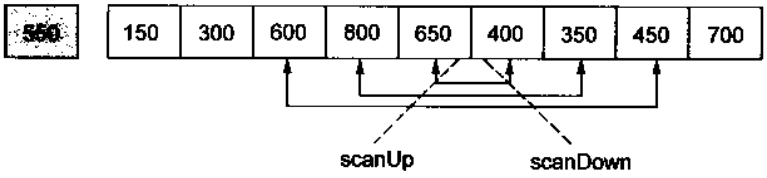
```
Swap(a[scanUp],A[scanDown]);        // 交换错位元素
```

这一过程一直持续到 scanUp 和 scanDown“擦肩而过”，即 scanDown = 5，scanUp = 6。此时，scanDown 首次进入元素值小于或等于中心值的低端表。我们找到了两个表之间的分隔点并求出了中心值的最后位置。在例子中，在 600 和 450，800 和 350，650 和 400 之间发生了交换。

然后我们交换中心值 A[0]和 A[scanDown]。

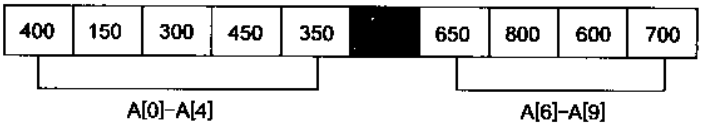
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com




```
Swap(a[0],A[scanDown]);
```

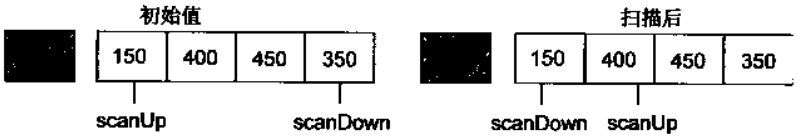
结果我们得到子表 A[0]—A[4],其元素值比子表 A[6]—A[9]中的要小。位于 A[5]的中心值(550)分割出了两个大小约为原始表一半的子表。在我们所称的递归阶段,这两个子表将用相同的算法进行处理。



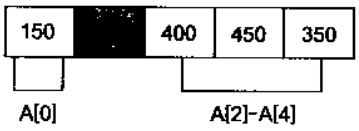
递归阶段 用相同的方法处理两个子表 S_l(A[0]—A[4]和 S_h(A[6]—A[9])。

子表 S_l: 子表的索引范围为 0 到 4,即 low = 0,high = 4,mid = 2,中心值为 A[mid] = 300。将中心值与 A[low]交换,为 scanUp 和 scanDown 赋初值:

```
scanUp = 1 = low + 1
scanDown = 4 = high
scanUp 中止于索引 2(A[2] > 中心值)
scanDown 中止于索引 1(A[1] < 中心值)
```



因为 scanDown < scanUp,过程中止,scanDown 成为更小的子表 A[0]和 A[2] - A[4]的分隔点。将 A[scanDown] = 150 与 A[low] = 300 进行交换以完成这部分操作。注意中心位置为我们分割出的是一个单元素和一个 3 元素的子表。递归过程终止于空子表和单元素子表。



子表 S_h: 索引范围为 6 到 9,即 low = 6,high = 9,mid = 7,中心值为 A[mid] = 800。将中心值与 A[low]进行交换,并为 scanUp 和 scanDown 赋初值:

```
scanUp = 7 = low + 1
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

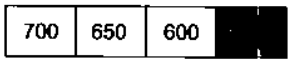
scanDown = 9 = high

scanUp 在逾越表尾时中止

scanDown 维持其初始位置

因为 scanDown < scanUp，过程中止，scanDown 确定了中心值的插入位置。将 A[scanDown] = 700与 A[low] = 800 交换以完成这部分操作。注意中心位置为我们分割出的是一个 3 元素的子表和一个空表。递归过程终止于空子表或单元素子表。

完成排序



处理子表 400,450,350(A[2]—A[4])

中心值 = 450

扫描过程将元素按 350、400、450 的次序排列。对两元素子表 350、400 还需再进行一次递归调用。

处理子表 700,650,600(A[6]—A[8])

中心值 = 650

扫描结束后，元素按 600、650、700 的次序排列。值 600 和 700 构成两个单元素子表。快速排序至此已经完成，所得到的表已排好序。

150	300	350	400	450	550	600	650	700	800

QuickSort 算法

递归算法从表中选择一个中心位置来分割表 A[low]到 A[high]：

```
pivot = A[mid];    // mid = (hight + low)/2
```

将中心值与 A[low]交换以后，令索引 scanUp = low + 1，scanDown = high。使它们分别指向表头和表尾。算法管理这两个索引的值。先将 scanUp 往表尾移动，只要不超过 scanDown 且所指向的元素小于或等于中心值。

```
// 索引 scnUp 扫描小于或等于 pivot 的元素
while (scanUp < = scanDown && A[scanUp] < = pivot)
scanUp + + ; // 取下一元素
```

scanUp 位置定了以后，将 scanDown 往表头方向移动，只有它所指向的元素比中心值大。

```
// scanDown 往表头移动，直到它所指向的元素小于或等于 pivot
while (pivot < A[scanDown])
scanDown - - ;
```

在此循环结束处，若 scanUp < scanDown，则这两个索引对应两个子表中的错位元素。将相应的值进行交换。

```
// 交换错位元素
Swap (A[scanUp],A[scanDown]);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

当 scanDown 小于 scanUp 时终止进行元素交换。此时，scanDown 表示的是其元素值小于或等于中心值的左子表的高端。索引 scanDown 就是表的中心值的位置。由 A[low]取出中心值：

```
Swap(A[low],A[scanDown]);
```

QuickSort(快速排序)使用递归处理子表。在确定分割表的中心值后，我们用参数 low 到 mid - 1 以及 mid + 1 到 high 递归调用 QuickSort。终止条件是子表元素个数小于 2，因为单元素表或空表必然有序。QuickSort 在文件"arrsort.h"中给出。

```
// QuickSort 的参数为一个数组及两个范围值
template < class T >
void QuickSort(T A[], int low, int high)
{
    // 存放中心索引及其值的局部变量及用于扫描的索引
    T pivot;
    int scanUp, scanDown;
    int mid;
    // 若表元素个数小于 2 个，则返回
    if (high - low < = 0)
        return;
    else
        // 若子表有两个元素，对其进行比较，并在必要时进行交换
        if (high - low = = 1)
        {
        if (A[high] < A[low])
            Swap(A[low], A[high]);
        return;
        }

    // 取中心索引并将其值赋给 pivot
    mid = (low + high)/2;
    pivot = A[mid];
    // 交换 pivot 及低端元素的值并初始化扫描索引 scanUp 和 scanDown.
    Swap(A[mid], A[low]);
    scanUp = low + 1;
    scanDown = high;
    // 定位错位元素；当 scanDown < scanUp 时结束
    do
    {
        // 从低端子表往上扫描，当 scanUp 进入到高端子表或遇到大于 pivot 的元素时结束
        while (scanUp < = scanDown && A[scanUp] < = pivot)
            scanUp + + ;
        // 从高端子表往下扫描，当 scanDown 指向的元素小于或等于 pivot 时停止；
        while (pivot < A[scanDown])
            scanDown - - ;
        // 若两个索引还在各自的子表中，则表示两个元素错位，将两个元素换位.
        if (scanUp < scanDown)
        {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

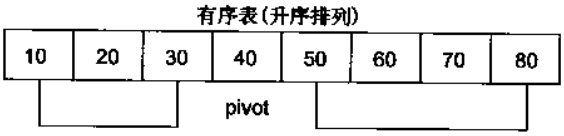
```
            Swap(A[scanUp], A[scanDown]);
        }
    }
    while (scanUp < scanDown);
    // 将 pivot 拷贝到 scanDown 位置,分开两个子表
    A[low] = A[scanDown];
    A[scanDown] = pivot;
    // 若低端子表(low 至 scanDown - 1)有 2 个或更多个元素,则进行递归调用
    if (low < scanDown - 1)
        QuickSort(A, low, scanDown - 1);
    // 若高端子表(scanDown + 1 至 high)有 2 个或更多个元素,则进行递归调用
    if (scanDown + 1 < high)
        QuickSort(A, scanDown + 1, high);
    }
}
```

QuickSort 的计算复杂度 对 QuickSort 效率进行一般性分析比较困难。我们可以只考虑非常理想的情况,这样可以较好地说明其复杂度并求出比较次数。假设 n 是 2 的幂,$n = 2^k (k = \log_2 n)$。另外,假设中心位置在表的中间,这样 QuickSort 可以将子表分割成两个大小基本相同的子表。

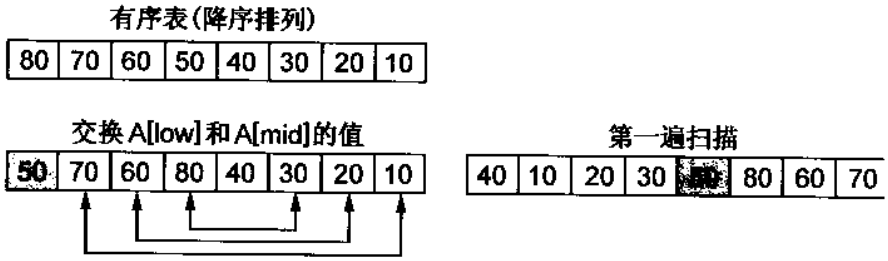
在第 1 遍扫描中,共进行 $n - 1$ 次比较。这一遍的结果是生成了两个大小约为 $n/2$ 的子表。下一阶段,处理每个子表需要大约 $n/2$ 次比较。这一阶段需要的总的比较次数为 $2(n/2) = n$。再下一阶段要处理 4 个子表,总共需要 $4(n/4)$ 次比较,依此类推。最后,分割过程在 k 遍扫描后终止,所得到的子表大小都为 1。总的比较次数约为:

$$
\begin{aligned}
n + 2(n/2) + 4(n/4) + \cdots + n(n/n) &= n + n + \cdots + n \\
&= n * k = n * \log_2 n
\end{aligned}
$$

对于一般的表,QuickSort 的计算复杂度为 $O(n \log_2 n)$。我们所讨论过的理想情况在数组已经按升序排列时确实能够实现。这种情况下,中心值恰好位于每个子表的中间。



如果数组是按降序排列,则第 1 遍在表的中间位置找到中心值并将低端和高端子表中的每个元素进行交换。所得到的表几乎已排好序,算法的量级为 $O(n \log_2 n)$。



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

QuickSort 最坏的情况是中心值总是落在一个单元素子表中，而其余元素都在另一个子表中。当中心值总是子表中的最小元素时这种情况就会发生。例如，数据 3,8,1,5,9 就是呈现出这种特点。初次扫描时，共进行 n 次比较，大的子表中包含 $n-1$ 个元素。下一次扫描时，大子表需 $n-1$ 次比较并生成 $n-2$ 个元素的子表，依此类推。总的比较次数为

$$n+n-1+n-2+\cdots+2=n(n+1)/2-1$$

复杂度为 $O(n^2)$，这种情况并不比选择或插入排序更好。但这终究是一种“病态”的情况，在现实中不太可能发生。一般来说，QuickSort 的平均性能在我们已讨论过的排序算法中是最好的。

QuickSort 是多数排序工具程序所选择的算法。如果你不能忍受它在最坏情况下的性能，那么可以选择堆排序，它是一种更健壮的 $O(n\log_2 n)$ 量级的算法，其复杂度仅取决于表的大小。

数组排序算法的比较

我们分别以 4 000、8 000、10 000、15 000 和 20 000 个整数作为运行数据，对几种排序算法进行比较。各种排序的比较都是基于同一组随机数据，以 tick(1/60 秒)为单位度量每种算法的运行时间。在 $O(n^2)$ 量级的排序算法中，插入排序具有较好的时间性能是由于它在第 i 遍只需进行 $i/2$ 次比较，明显优于其他 $O(n^2)$ 量级的排序算法。注意冒泡排序所表现的总体性能最差。结果和图表如图 14.2 所示。

	交换排序	选择排序	冒泡排序	插入排序
n = 4000	12.23	17.30	15.78	5.67
n = 8000	49.95	29.43	64.03	23.15
n = 10000	77.47	46.02	99.10	35.43
n = 15000	173.97	103.00	223.28	80.23
n = 20000	313.33	185.05	399.47	143.67

用 20 000 个以升序及降序排列的整数分别进行特殊运行以示意极端情况下的排序效率。冒泡和插入排序对以升序排列的数据只需 1 遍操作，但选择排序只取决于表大小，因而需 19 999 遍操作。当数据以降序排列时，交换、冒泡以及插入排序所表现的性能均属各自最差的情况，而选择排序则一如既往。

$O(n^2)$ 量级排序算法对有序表排序

	交换排序	选择排序	冒泡排序	插入排序
n = 20000 （升序排列）	185.27	185.78	.03	.05
n = 20000 （降序排列）	526.17	199.00	584.67	286.92

一般来说，QuickSort 是最快的排序算法。其 $O(n\log_2 n)$ 效率显然优于 $O(n^2)$ 算法。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

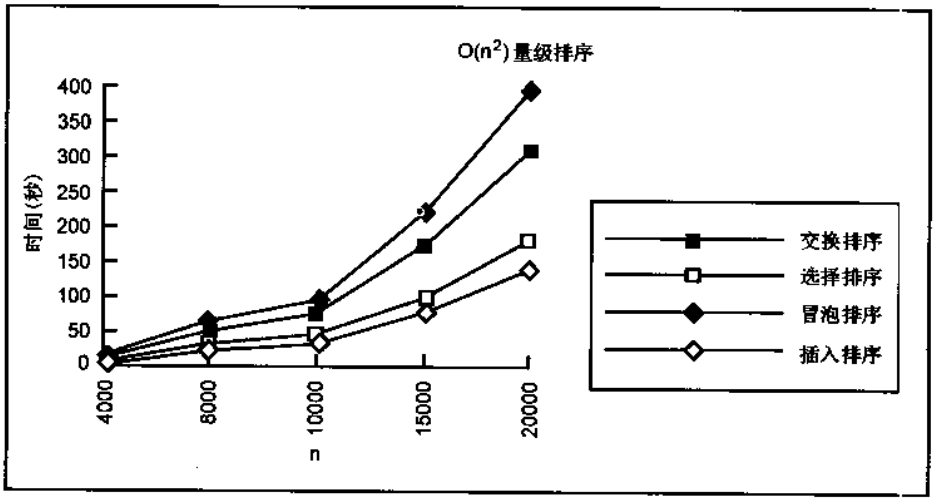


图 14.2　$O(n^2)$量级排序算法时间比较

事实上，在图 14.3 中我们可以注意到它比第 13 章所设计的其他任何 $O(n \log_2 n)$算法都要快。注意 QuickSort 处理极端情况的时间是 $O(n \log_2 n)$。而树排序在两种情况下均变成$O(n^2)$，因为所形成的树是退化的。

$O(n \log_2 n)$量级排序算法

	竞赛排序	树排序	堆排序	快速排序
n = 4000	0.28	0.32	0.13	0.07
n = 8000	0.63	0.68	0.28	0.17
n = 10000	0.90	0.92	0.35	0.22
n = 15000	1.30	1.40	0.58	0.33
n = 20000	1.95	1.88	0.77	0.47
n = 20000(升序排列)	1.77	262.27	0.75	0.23
n = 20000(降序排列)	1.65	275.70	0.80	0.23

图 14.3　$O(n\log_2 n)$量级排序算法比较

程序 14.1　排序法比较

本程序对排序算法进行比较并提供图 14.2 和 14.3 中的数据。这里只给出程序的基本结构，完整的清单在补充程序文件“prg14_1.cpp”中给出。计时功能由函数 TickCount 提供，它以 1/60 秒为单位，返回自程序开始以后流逝的时间。该函数包含于文件“ticks.h”中。

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


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 引入各种排序算法
#include "arrsort.h"
.....
// 描述数组数据初始状态的枚举类型
enum Ordering {randomorder, ascending, descending};
// 标明排序算法的枚举类型
enum SortType {exchange, selection, bubble, insertion,
               tournament, tree, heap, quick};
// 从数组 y 中拷贝 n 个元素到数组 x
void Copy(int *x, int *y, int n)
{
    for (int i=0; i<n; i++)
        *x++ = *y++;
}
// 对给定数组进行各种排序的函数框架
void Sort(int a[], int n, SortType type, Ordering order)
{
    long time;

    cout << "Sorting" << n;
    // 标明数据是否有序
    switch(order)
    {
        case random:      cout << "items.";
                          break;
        case ascending:   cout << "items in ascending order.";
                          break;
        case descending:  cout << "items in descending order.";
                          break;
    }
    // 开始排序的时间
    time = TickCount();
    // 进行各种排序
    switch(type)
    {
        case exchange:    ExchangeSort(a,n);
                          cout << "Exchange sort:";
                          break;
        case selection:   SelectionSort(a,n);
                          cout << "Selection sort:";
                          break;
        case bubble:      ....
        case insertion:   ....
        case tournament:  ....
        case tree:        ....
        case heap:        ....
    case quick:           ....
    }

    // 根据当前时间及起始时间计算算法用去的时间,并转换为秒
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

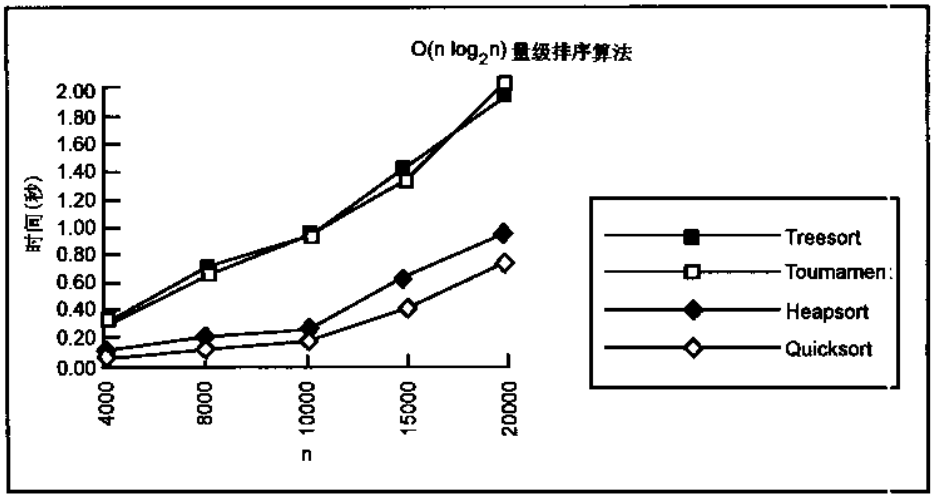
www.Linuxidc.com

```
    time = TickCount () - time;
    cout << time/60.0 << endl;
}
// 对给定顺序的 n 个数据进行排序
void RunTest(int n, Ordering order)
{
    int i;
    int *a, *b;
    SortType stype;
    RandomNumber rnd;
    // 为两个 n 元数组 a 和 b 申请空间
    a = new int [n];
    b = new int [n];
    // 判断使用何种顺序的数据
    if (order == randomorder)
      // 用随机整数初始化数组 b
      for (i = 0; i < n; i++)
      {
          b[i] = rnd.Random(n);
      }
    else if (order == ascending)
      // 排序数据为 0,1,2,3,…,n-1
      for (i = 0; i < n; i++)
      {
          b[i] = i;
      }
    else
      // 排序数据为 n-1, n-2, …,1,0
      for (i = 0; i < n; i++)
      {
          b[i] = n-1-i;
      }
    // 将数据拷贝到 a,对各类顺序的数据执行每种排序算法
    for (stype=exchange;stype <= quick;stype = SortType(stype+1))
    {
      Copy(a,b,n);
      Sort(a, n, stype, order);
    }

    // 释放两个动态数组
    delete [] a;
    delete [] b;
}
// 分别对 4000,8000,10000,15000 和 20000 个随机数进行排序,然后再对 20000 个数据的
// 无序和有序表进行排序
void main(void)
{
    int nelts[5] = {4000,8000,10000,15000,20000},i;
    cout.precision(3);
    cout.setf(ios::fixed|ios::showpoint);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

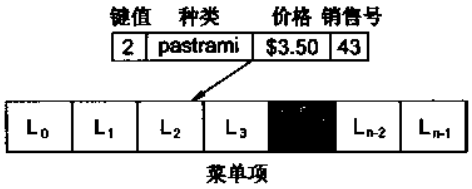
www.Linuxidc.com

```
    for (i=0; i < 5; i++)
        RunTest(nelts[i],randomorder);
    RunTest(20000,ascending);
    RunTest(20000,descending);
}
```



14.3 哈希法(Hashing)

在本书中,我们已经派生了一系列允许客户程序搜索和取出数据的表结构。每种结构都有一个 Find(查找)方法,它以键值为参数,遍历表以找到匹配的数据项。这一过程的效率取决于表结构。对于顺序表,Find 对表项进行 O(n)次扫描,而二叉搜索树和二分搜索则可提供效率更高的 O($\log_2 n$)次搜索。理想情况下,我们可在 O(1)时间内取出数据。这时所需的比较次数与数据元素的个数无关。当键值是作为数组的索引时,元素可以在 O(1)时间内取出。例如,三明治店用号码标注其菜单以简化簿记。要一份美味的"黑麦五香烟熏牛肉(pastrami on rye)"在数据库中仅用 2 号表示。对店主来说,只需将键值 2 关联到表中记录。



其他例子也是我们熟悉的。影像店的客户记录中含有一个 7 位数的电话号码。此电话号码就是用来获得客户记录的键值。

电话号码	客户名称,租借影片,等。

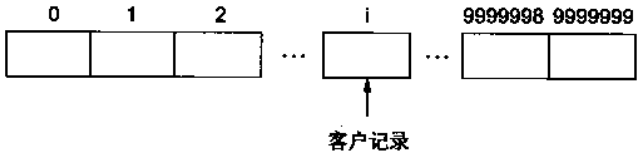
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

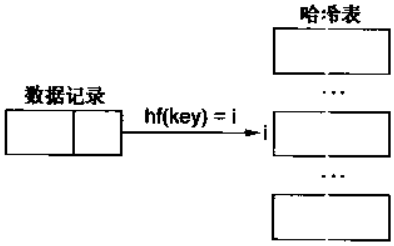
键值不一定是整数。例如,编译器生成的一种表叫“符号表(symbol table)”,其中包含了程序中用到的所有标识符以及与各个标识符有关的固定信息。符号表中每个记录的键值是表示标识符的串。

键和哈希函数

一般情况下,键值很少有像三明治店中使用的那么简单的。虽然它们提供对数据的访问,但大多数并不是作为数组记录的索引。例如,一个电话号码可以标识一个客户,但影像店并不能维护一个 1000 万个元素的数组。



在多数应用中,键用来提供对数据的间接引用。我们用一个函数将键映射到一定的整数范围内,然后用整数值来访问数据,这种函数被称作“哈希函数(hash function)”。下面我们说明这一思路。



假设我们有一组键值为整数的数据记录。哈希函数 HF 将键值映射为 0 到 n - 1 索引范围内的整数值。与哈希函数相关联的是一个表,其索引范围同样是 0 到 n - 1。这个表被称为“哈希表(hash table)”,它存放数据或数据的引用。

例 14.1

设 key 是正整数,HF(key)是 key 的个位值。索引范围为 0—9。例如,若 key = 49,则 HF(key) = HF(49) = 9。哈希函数用模 10 运算求返回值。

```
// 哈希函数返回键值的个位数
int HF(int key)
{
    return key % 10;
}
```

哈希函数经常是“多对一”,这就导致了“冲突(collisions)”。对于例 14.1 中的哈希函数,HF(49)和 HF(29)的值都 9。更一般地,此例中,个位相同的所有数的哈希函数值均相同。当冲突发生时,两个或更多的数据值被关联到哈希表的同一表项。因为两个数据项不可能占据表中同一位置,所以我们必须设计一种策略以处理冲突。我们将在介绍完几种哈希函数后讨论冲突的解决方案。

哈希函数

哈希函数必须将键值映射为 0 到 n - 1 范围内的整数值。其设计必须考虑减少冲突并保证执行效率。有几种方法可以满足这些条件。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

"除留余数法(division method)"是最常用的哈希技术,它分为两个步骤。首先必须将键转换成整数值,然后再用求余数运算符将这一值缩到 0～n－1 的范围内。在实际的哈希应用中,除法用得最多。

例 14.2

1. 键是 5 位数,哈希函数取其低两位。例如,若数值是 56389,则 HF(56389) = 89。最低两位数是除以 100 以后所得的余数。

```
int HF(int key)
{
    return key % 100; // 模 100
}
```

哈希函数的性能取决于它产生的值是否均匀分布于 0…n－1 范围内。如果后两位数对应的是出生年份,则用此哈希函数来标识参加某社区的"小小社团"活动的小孩时会产生过多的冲突。

2. 键是 C++ 串。哈希函数将串映射为整数,方法是将首尾字符(ASCII 值)相加,然后再除以哈希表的长度 101。

```
// 键值为串的哈希函数;返回值范围为 0 至 100
int HF(char * key)
{
    int len = strlen(key), hashf = 0;

    // 若串长为 0 或 1,返回首字符的 ASCII 值;否则,返回首尾字符的 ASCII 值
    // 之和
    if (len <= 1)
        hasf = key[0];
    else
        hashf = key[0] + key[len-1];
    return hasf & 101;    // 模 101
}
```

当两个串的首字符和尾字符都一样时会产生冲突。例如,串"start"和"slant"都映射到索引 29。如果求整个串的字符和也会发生类似的情况。

```
int HF(char * key)
{
    int hashf = 0;
    // 整串字符 ASCII 值之和对 101 取模
    while(*key)
        hashf += *key++;
    return hashf % 101;
}
```

串"bad"和"dab"被哈希函数映射到相同的索引。如果在计算 hashf 的值时将字符串的位元提取出来重新处理可以得到更好的哈希函数。程序 14.2 就附带一个较好的串哈希函数。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

一般地，n 越大，哈希索引的分布范围就越大。更进一步地，数学理论告诉我们，若 n 为素数则分布更均匀一些。

其他哈希算法

"平方取中法(midsquare technique)"将键转换为整数，求其平方值，然后从平方值的中间位置取连续若干位。假设键是整数且用 32 位表示整数，以下哈希函数取键的平方值的中间 10 位。

```
// 返回 key * key 的中间 10 位
int HF(int key)
{
    key * = key;          // key 的平方
    key > > = 11;         // 去掉低 11 位
    return key % 1024;    // 返回低 10 位
}
```

"随机乘法(multiplicative method)"使用一个随机实数 f，$0 \leq f < 1$。乘积 f * key 的分数部分的值在 0 到 1 之间，用这个值与 n(哈希表的长度)相乘，由乘积的整数部分给出 0 到 n - 1 范围内的哈希值。

```
// 用随机乘法的哈希函数;返回范围为 0…700 的哈希值
int HF(int key)
{
    static RandomNumber rnd;
    float f;
    // 将 key 与一个随机范围在 0 至 1 的实数相乘
    f = key * rnd.fRandom();
    // 取 f 的小数部分
    f = f - int(f);
    // 返回范围为 0 至 700 的整数
    return 701 * f;
}
```

冲突的解决

两个或更多的数据项可能有相同的哈希值，但它们不能占据哈希表中同一位置。我们面临的选择是要么将(引起冲突的)新项放到表中另外的位置，要么为每个哈希值单独建立一个表，表中包含具有相同哈希值的所有数据值。这些选择代表了解决冲突的两种经典策略，即"线性探测开放寻址法(linear probe open addressing)"和"独立表链地址法(chaining with separate lists)"。我们将举例说明开放探测寻址法，但我们的讲解重点还是独立表法，因为它用得最多。

线性探测开放寻址法 这种方法假定哈希表中每一项都被标记为空。这样，在我们试图加入新数据项之前就可以判断表项(entry)是否正被占用。如果位置已经被填，算法会绕表进行循环"探测"以找到表中第 1 个空位，其名称即由此而得。如果表的大小远超过要存储的数据项个数，这种方法效果就比较好，因为好的哈希函数将在允许的范围内均匀分布索引值，碰撞发生次数也将达到最少。当表长与数据记录个数的比较接近于 1 时，这种方法明显缺乏效率。我们以 7 个数据记录为例说明线性探测寻址法。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

例 14.3

假定数据项的类型为 DataRecord，它们被存储在一个含 n 个元素的哈希表中。

```
Struct DataRecord
{
    int key;
    int data;
}
```

哈希函数 HF 用对 11 求模的除法运算得到一个 0～10 之间的值。

```
HF(item) = item.key % 11
```

以下数据存储在哈希表中。每一项中都加注了将元素放入表中所需的探测次数。

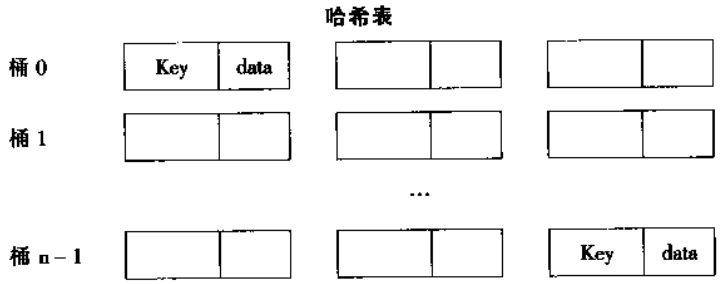
表：{54,1}，{77,3}，{94,5}，{89,7}，{14,8}，{45,2}，{76,9}

0	1	2	3	4	5	6	7	8	9	10
77(1) 3	89(1) 7	45(2) 2	14(1) 8	76(6) 9		94(1) 5				54(1) 1

前 5 项被哈希映射到不同的值，因此可以直接存到表中。例如，HF({54,1}) = 10，数据项被放入索引为 10 的表项中。头一次冲突在 89 和 45 之间发生，因为它们都哈希映射到 1。数据项{89,7}先在表中出现，它占据了哈希表中索引为 1 的位置。当我们试图装入{45,2}时发现位置已经被占。于是用开放探测寻址法开始对哈希表进行顺序扫描(探测)以找到空位。这里 HashTable[2]为空，数据项被添加到表中。由键 76 可看出此算法的低效率。76 映射到 10，而这一位置已经被占。在找到第一个空位 HashTable[4]之前，线性探测必须顺序扫描另外 5 个位置。将数据项存入表中所需总探测次数是 13，平均每项 1.9 次。

实现开放探测寻址的算法在习题中给出。

独立表链地址法 第 2 种哈希法将哈希表定义为诸如链表或树一类的集合的数组。每个集合被称为“桶(bucket)”，它存放哈希映射到相同表位置的一组数据记录。这种新的冲突解决策略被称为“独立表链地址法”。



如果表是链表数组，则只需将数据项作为表中另一结点即可将其插入。若要放入一个数据项，先用哈希函数确定链表，然后再进行顺序搜索。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

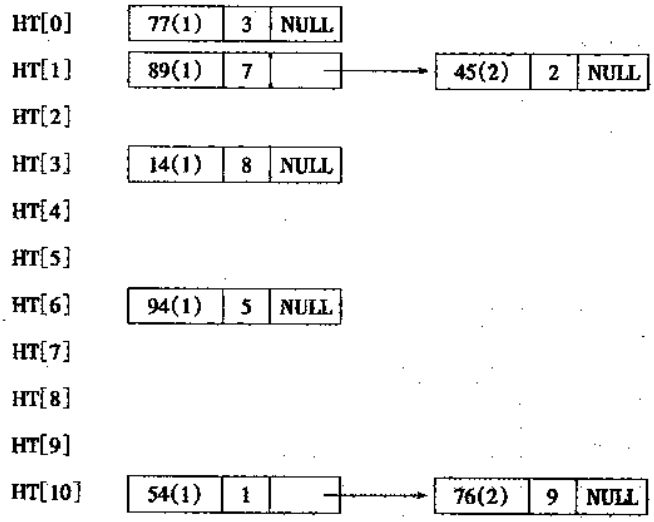
www.Linuxidc.com

例 14.4

用例 14.3 中 7 个 DataRecord 数据组成的表以及哈希函数 HF 来说明独立表链地址法：

```
表:{54,1},{77,3},{94,5},{89,7},{14,8},{45,2},{76,9}
HF = item.key % 11
```

每一个新数据项都被插入到其对应的链表的尾部。在下表中，每个数据值后都附注了扞其存入表中所需的探测次数。



注意如果我们将结点插入也计入探测次数，则插入 7 个数据项所需要的总的探测次数为 9，平均每项的探测次数是 1.3。

独立表链地址法一般要快于开放探测寻址法，因为前者要搜索的仅仅是那些被哈希映射到相同表位置的键。再进一步说，开放探测寻址假设表长一定，而独立表链地址法中哈希表的表项是动态分配的，其表长仅受内存容量限制。链表法的主要缺点是需要为结点指针域分配额外空间。但总的来说，独立表链地址法的动态结构使它成为哈希的首选方法。

14.4 哈希表类

本节定义一个通用的 HashTable 类，用独立表链地址法实现哈希。这个类由抽象类 List 派生得来，它提供具有高效访问方法的存储机制。类中允许任何类型的数据，但要求必须为此数据类型定义比较运算符“==”。客户程序必须重载“==”以比较两个数据项的键值域。

我们还编写一个 HashTableIterator(哈希表迭代算子)以方便从哈希表中收集数据。当我们需要对数据群体进行排序和打印时，HashTableIterator 对象可以发挥重要作用。

这些类的声明和实现都在文件“hash.h”中给出。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

HashTable 类说明

声明

```
#include "array.h"
#include "list.h"
#include "link.h"
#include "iterator.h"
template <class T>
class HashTableIterator;

template <class T>
class HashTable: public List<T>
{
    protected:
        // "桶"的个数,表示哈希表的大小
        int numBuckets;
        // 哈希表为链表构成的数组
        Array<LinkedList<T> > buckets;
        // 哈希函数及指向当前数据项的指针
        unsigned long (*hf)(T key);
        T *current;

    public:
        // 参数为哈希表大小及哈希函数的构造函数
        HashTable(int nbuckets, unsigned long hashf(T key));
        // 处理表的方法
        virtual void Insert(const T& key);
        virtual int Find(T& key);
        virtual void Delete(const T& key);
        virtual void ClearList(void);
        void Update(const T& key);
        // 相关的迭代算子可以访问数据成员
        friend class HashTableIterator<T>;
};
```

说明

HashTable 对象就是类型为 T 的元素表。它实现了抽象基类 List 中需要的所有方法。客户程序必须提供哈希表的大小以及将类型为 T 的元素转换为无符号长整数的哈希函数。这一返回类型使哈希函数具有较大的数据范围。除以哈希表大小的工作是在内部完成的。

Insert,Find,Delete 和 ClearList 提供了基本的表处理方法。另外还单独提供了一个 Update 方法对已经在哈希表中的元素进行更新。

ListSize 和 ListEmpty 方法由基类提供。数据成员 current 总是指向最近一次访问的数据值。它用于 Update 方法和派生类,因为它们必须返回数据引用。14.7 节中将讨论该类的一个例子。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

例

假设 NameRecord 是一个包含名字(name)域和计数(count)域的记录。

```
struct NameRecord
{
    String name;
    int count;
}
// 数据类型为 NameRecord 的一个 101 个元素的哈希表,其哈希函数为 hash
HashTable < NameRecord > HF(101, hash);
// 往表中插入记录{"Betsy", 1}
NameRecord rec;             // 类型为 NameRecord 的变量
rec.name = "Betsy";         // 赋值,name = "Betsy", count = 1
rec.count = 1;
HF.Insert(rec);             // 插入记录
cout < < HF.ListSize();     // 输出哈希表大小 1
// 检索对应于键值"Betsy"的记录,将其 count 域加 1 后存回该记录
rec.name = "Betsy";
if (HF.Find(rec))           // 查找"Betsy"
{
    rec.count + = 1;        // 个性数据域
    HF.Update(rec);         // 存回哈希表中
}
else
    cerr < < "Error: \"Betsy should be in the table.\"\n";
```

HashTableIterator 类由抽象类 Iterator 派生得来,它包含遍历表中数据值的方法。

HashTableIterator 类的描述

声明

```
template < class T >
class HashTableIterator: public Iterator < T >
{
    private:
        // 指向需遍历的哈希表的指针
        HashTable < T > * hashTable;
        // 已被遍历的"桶"的下标及指向其链表的指针
        int currentBucket;
        LinkedList < T > * currBucketPtr;

        // 取下一个结点的例程
        void SearchNextNode(int cb);

    public:
        // 构造函数
        HashTableIterator(HashTable < T > & ht);
        // 基本的迭代算子方法
        virtual void Next(void);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        virtual void Reset(void);
        virtual T& Data(void);
        // 用迭代算子遍历另外一个表
        void SetList(HashTable < T > & lst);
};
```

说明

Next 通过在链表(桶)间移动并遍历每个链表中的结点的方法对哈希表进行遍历。迭代算子所得到的数据值是无序的。该方法使用了函数 SearchNextNode,它可以定位下一个必须遍历的表。

例

```
// 为哈希表对象 HF 定义一个迭代算子
HashTableIterator < NameRecord > hiter(HF);

// 扫描数据库中所有元素
for(hiter.Reset(); ! hiter.EndOfList();hiter.Next())
{
    rec = hiter.Data();
    cout < < rec.name < < ": " < < rec.count < < endl;
}
```

应用:字符串频度

我们可以用 HashTable 类存储一组字符串并确定其在文件中的出现频度。每个串都存储于一个 NameRecord 对象中,对象中包含串名及其频度计数。

```
Struct NameRecord
{
    String name;
    int count;
};
```

哈希函数将串中各字符的位元进行混合处理,方法是将当前哈希值左移 3 位(乘以 8)再与下一个字符相加。对于 n 个字符的串 $C_0C_1\cdots C_{n-2}C_{n-1}$,

$$\text{hash}(s) = \sum_{i=0}^{n-1} C_i 8^{n-i-1}$$

这种计算方法可以防止出现例 14.2 中所讨论的串哈希函数所存在的问题。

```
// 用于类 Hash 的函数
unsigned long hash(NameRecord elem)
{
    unsigned long hashval = 0;

    // 将字符左移三位后加上下一字符
    for (int i = 0; i < elem.Length(); i + + )
        hashval = (hashval < < 3) + elem.name[i]
    return hashval;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

程序 14.2　计算字符串的频度

程序从文件"strings.dat"中读取字符串并将它们存放到一个含 101 个元素的哈希表中。读入文件中的每一个串,若未遇见过则将其存入表中。对于重复串,则从哈希表中检索出其记录并将计数值(count)增 1。程序最后定义一个用来遍历哈希表并打印其表项的迭代算子。NameRecord 的定义、哈希函数以及为 NameRecord 数据定义的"==="运算符都包含于文件"strfreq.n"中。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "hash.h"
#include "strclass.h"
#include "strfreq.h"
void main(void)
{
    // 从流 fin 中读入串
    ifstream fin;
    NameRecord rec;
    String token;
    HashTable<NameRecord> HF(101,hash);

    fin.open("strings.dat", ios::in | ios::nocreate);
    if (! fin)
    {
        cerr << "Could not open \"strings.dat\"!" << endl;
        exit(1);
    }
    while(fin >> rec.name)
    {
        // 从哈希表中检索串;若存在,修改其 count 值
        if (HF.Find(rec))
        {
            rec.count += 1;
            HF.Update(rec);
        }
        else
        {
            rec.count = 1;
            HF.Insert(rec);
        }
    }
    // 输出所有串及其频度
    HashTableIterator<NameRecord> hiter(HF);
    for(hiter.Reset();! hiter.EndOfList();hiter.Next())
    {
        rec = hiter.Data();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        cout << rec.name << ": " << rec.count << endl;
    }
}
/*
<文件"strings.dat">
Columbus Washington Napoleon Washington Lee Grant
Washington Lincoln Grant Columbus Washington
<程序 14.2 运行结果>
Lee: 1
Washington: 4
Lincoln: 1
Napoleon: 1
Grant: 2
Columbus: 2
*/
```

HashTable 类的实现

HashTable 类由提供 ListSize 和 ListEmpty 方法的抽象类 List 派生得到。下面我们讨论 HashTable 类的数据成员以及实现纯虚函数 Insert,Find,Delete 和 ClearList 的操作。

类中的关键数据成员是 Array 对象 buckets,它定义了构成哈希表的链表对象数组。函数指针 hf 指向哈希函数,numBuckets 是哈希表长度。指针 current 指向哈希表方法最近一次访问的数据项,其值由 Find 和 Insert 方法设置。用 Update 方法可改变表中数据项的值。

表处理方法 Insert 计算出哈希值(桶索引)并对 LinkedList 对象进行搜索以判断数据项是否已经在表中。如果找到匹配值,Insert 将替换数据项,将 current 改为指向该数据值,然后返回。如果没有匹配值,Insert 将把数据项加入到表尾,令 current 指向新数据值,并将表长增 1。

```
template <class T>
void HashTable<T>::Insert(const T& key)
{
    // hashval 为哈希值(桶索引)
    int hashval = int(hf(key) & numBuckets);
    // lst 为 buckets[hashval]的别名,可避免下标寻址
    LinkedList<T> & lst = buckets[hashval];
    for(lst.Reset();! lst.EndOfList();lst.Next())
        // 若找到匹配值,修改其数据后返回
        if (lst.Data() = = key)
        {
            lst.Data() = key;
            current = &lst.Data();
            return;
        }
    // 若没找到,则将数据项加入表中
    lst.InsertRear(key);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    current = &lst.Data();
    size++;
}
```

Find 遍历由哈希函数所得到的链表以寻找匹配值。如果找到，则将数据复制到 key 中，将 current 赋值为匹配结点的地址，然后返回 True。否则，该方法返回 False。

```
template <class T>
int HashTable<T>::Find(T& key)
{
    // 计算键值的哈希值并将 lst 指向它对应的 LinkedList
    int hashval = int(hf(key) % numBuckets);
    LinkedList<T>& lst = buckets[hashval];

    // 在链表中扫描结点并寻找与 key 匹配的记录
    for(lst.Reset();! lst.EndOfList();lst.Next())
        // 若找到匹配值，则取其数据值，将 current 指向该记录
        if (lst.Data() == key)
        {
            key = lst.Data();
            current = &lst.Data();
            return 1;          // 返回 True
        }
    return 0;                  // 否则，返回 False
}
```

方法 Delete 遍历目标链表，若发生匹配则删除结点。Delete，ClearList 和 Update 方法都在文件“hash.h”中给出。

HashTableIterator 类的实现

HashTableIterator 类必须能够遍历哈希表中的数据。这使得它比 HashTable 类更令人感兴趣但也更难实现。先在链表数组中找出一个非空桶，以此开始对表中数据项的遍历。当找到非空桶时，就遍历表中所有结点，然后继续寻找下一个非空桶。当遍历完一个内含元素的桶且没有可供搜索的桶时，迭代算子就到达了表尾。

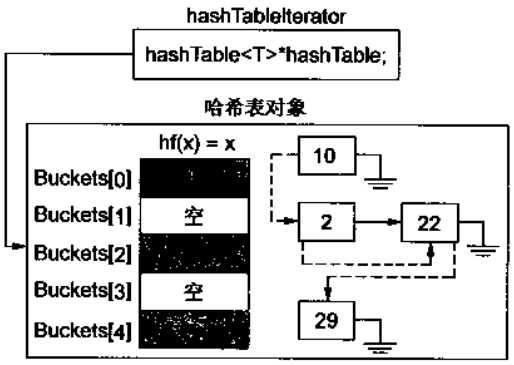


图 14.4 HashTableIterator

迭代算子必须联编到表上。这里，成员变量 HashTable 被赋予哈希表的地址。因为

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

HashTableIterator是 HashTable 的友元，故可以访问它的所有私有变量，包括 Array 对象 buckets 和它的大小 numBuckets。变量 currentBucket 是当前正在遍历的链表的索引，currBucketPtr 则是指向链表的指针。用 LinkedList 类中内置的迭代算子对每个桶进行遍历。图 14.4 中示意了迭代算子遍历 4 个数据项的哈希表的情况。

方法 SearchNextNode 用来确定下一个必须遍历的表的位置。它搜索从 cb 开始的所有桶，直到发现一个非空表为止。将此表的索引赋值给 currentBucket，其地址赋值给 currBucketPtr。若没有非空表，令 currentBucket = -1 并返回。

```
// 从下标 cb 开始寻找下一个非空表进行扫描
template <class T>
void HashTableIterator<T>::SearchNextNode(int cb)
{
    currentBucket = -1;

    // 若下标 cb 大于哈希表的大小，终止检索，扫描结束
    if (cb > hashTable->numBuckets)
        return;

    // 否则，继续扫描非空表并修改私有数据成员
    for (int i=cb; i < hashTable->numBuckets;i++)
        if (! hashTable->buckets[i].ListEmpty())
        {
            // 返回前，置 currentBucket 为 i 并将 currentBucket 指向新的非空表
            currBucketPtr = &hashTble->buckets[i];
            currBucketPtr->Reset();
            currentBucket = i;
            return;
        }
}
```

构造函数初始化 Iterator 基类并使私有指针变量 hashTable 指向哈希表的地址。以 0 为参数调用 SearchNextNode，找出一个非空表。

```
// 构造函数;初始化基类及哈希表，用 SearchNextNode 找到表中第一个非空桶.
template <class T>
HashTableIterator<T>::HashTableIterator(HashTable<T>& hf)
    Iterator<T>(hf), hashTable(&hf)
{
    SearchNextNode(0);
}
```

Next 在当前表中往前推进一个元素。如果到达表尾，则函数 SearchNextNode 对迭代算子进行刷新，使其可以遍历下一个非空桶。

```
// 指向表中下一个数据结点
template <class T>
void HashTableIterator<T>::Next(void)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 将当前表移向下一结点或表尾
      currBucketPtr->Next();
    // 若已到表尾,调用 SearchNextNode 找哈希表下一非空桶
    if (currBucketPtr->EndOfList())
        SearchNextNode(++currentBucket);
    // 置 iterationComplete 标志来表示 currentBucket 是否已在表尾
    iterationComplete = currentBucket == -1;
}
```

14.5 搜索方法的性能

本书中我们已经介绍了4种搜索算法,即顺序搜索法、二分搜索法、二叉树搜索法以及哈希法。搜索算法的运行时间通常取决于在表中定位一个数据项所需要的平均比较次数。我们已经知道,顺序搜索为 $O(n)$,而二分搜索和二叉树搜索为 $O(\log_2 n)$。

对哈希法进行性能分析则比较复杂。其性能取决于哈希函数的品质以及哈希表的大小。好的哈希函数能使哈希值均匀分布。如果配以相对较大的表,则可以减少冲突次数。由表的大小推导出哈希表的装载因子(load factor)。若哈希表共有 m 个表项,其中 n 项正在使用,则表的装载因子 λ 由下式定义

$$\lambda = n/m$$

当表为空时,$\lambda=0$。当越来越多的数据项被加入到表中时,λ 的值增大,而冲突机会也随之增加。对于开放探测法,当表满($m=n$)时,λ 保持最大值1。若使用独立表链地址法,各个链表可能随需求而扩大,因而 λ 可能大于1。

我们可以对使用链地址的哈希法的复杂度进行直观上的讨论。最坏的情况是所有数据项都被哈希映射到表中同一位置。如果链表中含 n 个数据元素,则表的搜索时间为 $O(n)$,所以链地址法在最坏情况下的性能为 $O(n)$。

一般情况下,哈希值的分布相对比较均匀,我们预计每个链表中有 $\lambda = n/m$ 个元素。这样每个链表的搜索时间为 $O(\lambda)=O(n/m)$。假设放入表中的元素个数被限制在某个量,如 $R*m$ 的范围内,则每个表的搜索时间为 $O(R*m/m)=O(R)=O(1)$,链地址法就是一种 $O(1)$ 方法。对实行独立表链地址法的哈希表的数据访问可在常量时间内完成,与数据项个数无关。

对哈希表进行正规的数学分析已超出本书的范围。表14.1中列出了当哈希表的表项数目 m 较大时每种哈希算法在搜索成功和不成功的情况下所需探测次数的近似值。每个公式都是装载因子 λ 的函数。请参阅 Knuth 在1973年对此及相关结果进行的讨论。当 $\lambda=1$ 时,成功的搜索平均需要 $m/2$ 次探测,而不成功的搜索则需要 m 次探测。

表14.1 计算哈希算法复杂度的公式

	成功搜索探测次数	不成功搜索探测次数
开放探测法	$\frac{1}{2(1-\lambda)}+\frac{1}{2},\ \lambda\neq 1$	$\frac{1}{2(1-\lambda)^2}+\frac{1}{2},\ \lambda\neq 1$
链地址法	$1+\frac{\lambda}{2}$	$e^{-\lambda}+\lambda$

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

上表说明，只要装载因子 λ 保持较小值，开放探测法不失为一种好的方法。但一般说来，链地址法更好一些。例如，当 $m=n(\lambda=1)$时，对于成功的搜索，链地址法只需 1.5 次探测，而开放探测搜索全表，它平均需要 $m/2$ 次探测。当表为半充满($\lambda=1\backslash 2$)时，对于成功的搜索，链地址法需要 1.25 次探测，而开放探测则需要 1.5 次。

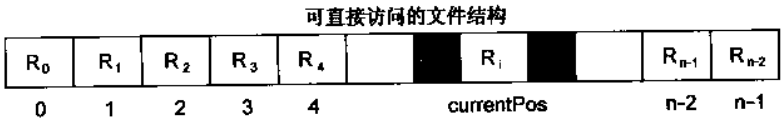
很显然，哈希法是一种极快的搜索方法。但话又说回来，4 种搜索方法又各有其用途。当元素个数较少，且数据无需排序时顺序搜索[$O(n)$]比较有效。二分搜索[$O(\log_2 n)$]很快但需要将数据在数组中排好序。二分搜索不适用于那些数据值在运行时才能确定的场合(如编译器符号表)，因为有序数组对于表的插入和删除操作是一种低效工具。对以上问题，二叉树搜索[$O(\log_2 n)$]和哈希法[$O(1)$]都能很好地解决。二叉树搜索虽不及后者快但它的好处在于进行中序遍历时能"顺便"将数据排好序。当需要访问无序数据时哈希法是最好的方法。

14.6 二进制文件和外部数据操作

许多应用程序需要访问磁盘文件中的数据。这一节我们将概述用文件"fstream.h"中所包含的类 fstream 进行二进制文件的 I/O 操作。在此过程中我们设计类 BinFile，它包含用来打开和关闭二进制文件、访问文件中的单个记录以及对文件中的数据进行块读和块写操作的方法。大的数据群体可能包含几百万个记录，它们无法都驻留到内存中。当数据驻留在外部时，我们需要对它们进行管理并将搜索和排序算法扩展到文件。我们只对此作一简单介绍，因为对文件以及外部搜索和排序的详细讨论已超出本书的范围。

二进制文件

文本文件中包含的是 ASCII 字符，其各行用一系列换行符隔开。二进制文件则由数据记录构成，这些数据记录可以是单个字符(字节)，也可以是更复杂的结构，如整数、浮点数以及数组。在硬件一级，文件的数据记录以定长数据块的形式存储在磁盘上。磁盘数据块通常是不相邻的。但从逻辑上看，我们将文件数据当作记录序列。文件系统允许直接访问单独的数据记录，这使得我们可以将文件看作外部记录的数组。在数据被读入或写出的过程中，系统维护一个指向文件当前位置的指针。



文件也是一种顺序结构，它维护一个指向当前数据位置的文件指针。读和写操作在此当前位置访问数据，然后将当前位置移到下一个数据记录处。

C++的 fstream 类描述的是既可用来输入又可用来输出的文件对象。当建立对象

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

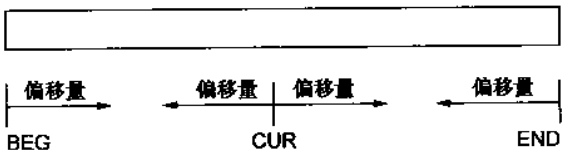
时，我们用 open 方法获取其文件名和访问方式。各种可能的方式在基类 ios 中定义，

方式	动作
in	打开文件供读出
out	打开文件供写入
trunc	在读出或写入前删除文件记录
nocreate	若文件不存在，无需建立空文件。返回流错误条件。
binary	以二进制方式打开文件(非文本文件)

例 14.5

```
1. #include <fstream.h>
   fstream f;            // 定义文件指针
   // 以只读方式打开文件"Phone"。若文件不存在，指出错误
   f.open("Phone", ios::in|ios::nocreate);
2. fstream f;            // 定义文件指针
   // 以可读写方式打开二进制文件
   f.open("DataBase", ios::in|ios::out|ios::binary);
```

每个文件对象都有一个标识当前输入或输出记录的文件指针与其关联。对于输入文件，函数 tellg()将当前指针位置作为距离文件头开始的偏移量，返回其字节数。对于输出文件，函数 tellp()返回当前文件指针位置所对应的字节数。函数 seekg()和 seekp()允许用户重新定位当前文件指针。定位函数带一个偏移量参数，它表示距离文件头(beg)、尾(end)或当前位置(cur)指针的字节数。如果文件同时用于输入和输出，则用定位函数 tellg 和 seekg。



例如，下列程序段示意了函数 seekg 和 tellg 的操作：

```
// 整数数值构成的二进制文件
fstream f;
f.open("datafile", ios::in | ios::nocreate | ios::binary);
// 重置当前指针到文件头
f.seekg(0, ios::beg);
// 置当前指针到文件中最后一个数据值
f.seekg(-sizeof(int), ios::end);
        . . .
// 将当前指针指向下一个整数
f.seekg(sizeof(int), ios::cur);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    . . .
// 指向文件尾
f.seekg(0, ios::end);
// 输出文件中字节数
cout < < f.tellg() < < endl;
// 输出文件中整数个数
cout < < f.tellg()/sizeof(int);
```

fstream 类中具有对字节流进行 I/O 操作的低级读写方法。每种方法的参数包括缓冲区地址和表示要传送的字节数的计数值。缓冲区是用来存储送往或取自磁盘的数据的字符数组。非字符型数据的操作需要用(char *)进行强制类型转换。例如,以下操作将一个整数数据块送往文件或从文件中取出。

```
fstream f;                          // 定义 fstream 对象
int data = 30, A[20];               // 初始化数据值及数组
// 从字符串格式将整数 30 写入到 sizeof(int)个字节块中
f.write((char * ) &data, sizeof(int));
// 从流 f 中读出 20 个整数到数组 A
f.read((char * )A, 20 * sizeof(int));
```

BinFile 类

许多应用中都有一个文件用于处理数据的输入和输出。这一节我们将文件从应用中抽象出来并定义一个为二进制文件提供一般文件处理操作的类。作为特定的例子,这种类对用户悄悄隐藏了低级系统的细节,因为它是一个基于模板的类,所以客户程序可以用文件存放各种不同类型的数据。

BinFile 类说明

声明

```
// 有关文件及处理文件方法的系统文件
# include < iostream.h >
# inclue < fstream.h >
# include < stdlib.h >
# include "strclass.h"
// 访问文件的几种方式
enum Access {IN, OUT, INOUT};
// 移动文件指针的几种方式
enum SeekType {BEG, CUR, END};
template < class T >
class BinFile
{
    private:
        // C + + 文件流对象及其文件名和访问它的方式
        fstream f;
        Access accessType;          // 访问文件方式
        String fname;               // 文件名
        int fileOpen;               // 该文件已被打开否?
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 可直接访问文件的几个参数
        int Tsize;                  // 数据记录的大小
        int fileSize;               // 文件中记录个数
        // 输出错误信息并退出程序
        void Error(char *msg);

    pulic:
        // 构造函数和析构函数
        BinFile(const String& fileName, Access atype = OUT);
        ~BinFile(void);
        // 复制构造函数.对象必须以形参传递
        BinFile(BinFile<T>& bf);
        // 有关文件的例程
        void Clear(void);           // 删除文件中的记录
        void Delete(void);          // 关闭并删除文件
        void Close(void);           // 关闭文件
        int EndFile();              // 是否已到文件尾
        long Size();                // 返回文件中记录个数
        void Reset(void);           // 重置文件指针到第一个记录
        void Seek(long pos, SeekType mode);
        // 读入n个数据值的数据块到地址A
        int Read(T *A, int n);
        // 从地址A写入n个数据值的数据块到文件
        void Write(T *A, int n);
        // 当前位置的记录值
        T Peek(void);
        // 拷贝数据到文件中第pos个记录
        void Write(const T& data, long pos);
        // 读入文件中第pos个记录
        T Read(long pos);
        // 往文件尾追加一个记录
        void Append(T item);
};
```

讨论

构造函数负责打开文件以及初始化类参数。建立 BinFile 对象时,客户程序必须指明文件访问方式(IN,OUT 或 INOUT)。具有方式 OUT 的文件在打开时其内容被删掉。创建 IN 方式的文件必须成功,否则程序打印出错消息然后终止。如果文件被声明为 INOUT,则文件记录可读写。打开文件以后,构造函数将数据成员 fileOpen 设为 1(True),表示诸如 Read 一类的文件操作是合法的。

类中包含一个复制构造函数,当它被调用时会产生出错消息。文件对象依附于物理文件而建立。允许文件被复制意味着新对象必须打开同一文件。在某些系统中这是不可能的,若允许这样做则会导致危险的后果。BinFile 对象可以通过引用进行传递。

文件可以被当作直接访问的数组来处理。方法 Write 和 Read 以记录索引 pos 为参数,在它所指定的文件位置读或写数据项。针对块的 Read 和 Write 方法用了多个记录的 I/O 操作。块方法 Read 返回所读取的记录数,若遇到文件尾则返回 0。文件中剩余记录数可能少于 n,因此返回值也可能小于 n。数据的地址和记录个数均被作为参数传递。传

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

送从当前文件位置开始。方法 Peek 允许客户程序读取文件的当前位置中的值而无需推进指针。

方法 EndFile 返回一个逻辑值，表示是否已经到达文件尾部。此方法仅用于 IN 方式的文件。对其他类型的文件，则用一个 for 循环，当循环变量超过文件最后一个记录的索引时循环中止。

方法 Close 关闭应先文件流但并不删除物理文件。如果文件将被另一对象打开，并且其方式也不尽相同，则使用 Close。

方法 Clear 删掉文件内容，使其大小变为 0 且使文件处于打开状态。Delete 关闭文件并从文件系统中删除它。Delete 方法置 fileOpen 为 0(False)，此后再企图访问文件将导致程序中止。Seek 允许对文件指针进行重定位。参数 mode 指明是以文件头、当前位置还是以文件尾为基准将文件指针移动 pos 个记录。

例

```
// 可读写整数文件，文件名为"demofile"
BinFile < int > BF("demofile", INOUT);
BinFile < int > BG("outfile", OUT);// 只写整数文件
int i, m = 5, n = 10, A[10];        // 整型变量
// 将把下列数据写入"demofile"
int vals[] = {30,40,50,60,80,90,100};
for (i = 0; i < 5; i + +)
    BF.Write(&i,1);
BF.Append(m);                       // 往文件中加入 5
BF.Reset();                         // 指向文件头
BF.Write(n, 0);                     // 在文件头写入 10
cout < < BF.Size() < < endl;        // 输出文件大小 6
cout < < BF.Read(3) < < endl;       // 输出 3(第三个整数值)
BF.Read(A,2);                       // 读出两个整数到数组 A
cout < < A[0] < < " " < < A[1];     // 输出它们的值
BF.Reset();                         // 重置到文件头
cout < < BF.Peek() < < endl;        // 输出 10
BF.Read(A,4);                       // 读出 4 个整数到 A 中
BG.Write(A,4);                      // 将 A[0] - A[3]写入 BG
A[0] * = 2;                         // 将 A[0]加倍
BG.Write(A[0],0);                   // 将新值写回 BG 的第 1 个记录
BF.Seek(2,beg);                     // 移到 BF 的第 2 个记录
BF.Write(vals,8);                   // 从第 2 个记录开始，将 30..100 写入 BF.
BF.Reset();                         // 重置到"demofile"文件头
// 用 Size 参数读出并输出文件"demofile"
for(i = 0; i < BF.Size(); i + +)
{
    BF.Read(&m,1);
    cout < < m < < " ";
}
cout < < endl;
BF.Delete();                        // 删除文件 BF
BG.Close();                         // 关闭"outfile"
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
Binfile<int> BH("outfile",IN);  // 重新打开"outfile"(只读方式)
while(! BH.EndFile())           // 用 EndFile 标志读出并输出文件"outfile"
{
    BH.Read(&m, 1);
    cout << m << ".";
}
cout << endl;
BH.Close();
<输出结果>
6
3
4  5
10
10  1  30  40  50  60  70  80  90  100
20  1  2  3
```

BinFile 的具体实现

BinFile 类的完整实现在文件"binfile.h"中给出。这一节我们要讨论的是构造函数、直接访问的读方法、n 个记录的块写以及实用方法 Clear。

构造函数负责打开文件以及初始化类参数。建立对象时，我们要为构造函数提供文件名和访问类型。

```
// 构造函数;用指定的文件名及访问方式打开文件
template <class T>
BinFile<T>::BinFile(const String& fileName, Access atype)
{

    // 根据访问方式打开文件。对于方式 IN,若给定文件不存在也不创建;对于方式 OUT,
    // 则清空字文件;对于 INOUT,该文件可读写
    if (atype == IN)
        f.open(fileName, ios::in | ios::nocreate | ios::binary);
    else if (atype == OUT)
        f.open(fileName, ios::out | ios::trunc | ios::binary);
    else
        f.open(fileName, ios::in | ios::out | ios::binary);
    if(! f)
        Error("BinFile constructor: file cannot be opened");
    else
        fileOpen = 1;
    accessType = atype;
    // 计算文件中记录个数

    // Tsize 数据类型 T 的字节数
    Tsize = sizeof(T);
    if (accessType == In || accessType == INOUT)
    {
        // 计算文件中记录个数,将指针置回文件头
        f.seekg(0, ios::end);
        fileSize = f.tellg()/Tsize;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        f.seekg(0, ios::beg);
    }
    else
        fileSize = 0;          // OUT 文件的大小为 0
    // 将文件名保存在 fname 中
    fname = fileName;
}
```

文件访问 用 Seekg 方法，我们就可以把文件当作是直接访问的数组。Read 需要位置参数 pos。Seekg 将数据大小和位置参数相结合，将当前文件指针定位于指定记录处并读取数据值。

```
// 返回文件中记录 pos 的数据值
template <class T>
T BinFile<T>::Read (long pos)
{
    // 存放读出数据的变量
    T data;

    if (! fileOpen)
        Error("BinFile Read(int pos): file closed");

    // Read 方法对只写文件来说是非法操作
    if (accessType = = OUT)
        Error("Invalid file access operation");
    // 检查 pos 值是否合法
    else if (pos < 0 || pos > = fileSize)
        Error("Invalid file access operation");

    // 置文件指针并读出文件中数据
    f.seekg(pos * Tsize, ios::beg);
    f.read((char *)&data, Tsize);

    // 若打开方式为 IN 且已到文件尾，则置文件结束标志
    if (accessType = = IN)
        if (f.tellg()/Tsize > = fileSize)
            f.clear(ios::eofbit);
    return data;
}
```

当把文件作为顺序访问设备使用时，就可以定义将多个记录复制到文件中的 Write 方法。数据的地址和记录个数被作为参数传递。fstream 的写方法将字节流复制到输出文件。因为写操作可能从文件中部开始，所以要小心维护 fileSize 的值以保持其正确性。

```
// 将 n 元数组 A 写入到文件中
template <class T>
void Binfile<T>::Write(T *A, int n)
{
    long previousRecords;
    // 对只读文件，Write 为非法操作
    if (accessType = = IN)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        Error("Invalid file access operation");
    if (! fileOpen)
        Error("BinFile Write(T * A, int n); file closed");
    // 计算写入后文件的大小,以决定是否修改 fileSize 值
    previousRecords = f.tellp()/Tsize;
    if (previousRecords + n > fileSize)
        fileSize + = previousRecords + n - fileSize;
    // 写入的字节数为 n * Tsize;
    f.write((char * )A, Tsize * n);
}
```

实用方法 类中包含一系列用来管理文件的实用方法。Clear 方法删除文件中的现有记录,具体过程是先关闭文件,然后再以清空方式重新打开。第 2 次打开时使用的也是原始文件参数,这些参数是以私有数据成员的形式保存的。

```
// Clear 删除文件中所有的记录
template < class T >
void BinFile < T > ::Clear(void)
{
    // 只读文件不能被清空
    if (accessType = = IN)
        Error("Invalid file access operation");
    // 关闭文件后再重新打开
    f.close();
    if (accessType = = OUT)
        f.open(fname, ios::out|ios::trunc|ios::binary);
    else
        f.open(fname, ios::in|ios::out|ios::trunc|ios::binary);
    if (! f)
        Error("BinFile Clear: cannot reopen the file");
    fileSize = 0;
}
```

外部文件搜索

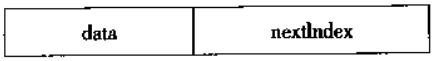
我们已经设计了一系列用来存储数据的内部表结构。对文件中的数据也可以定义一组类似的结构。外部搜索和排序技术的效率取决于我们组织文件记录的方法。我们将哈希法的概念进行推广,使其包括文件结构,并用 BinFile 方法访问数据。

哈希技术提供了可以扩展到外部结构的有效算法。哈希函数将每个数据记录与 0 到 n-1 范围内的一个整数相关联。这个整数值可以被用作访问记录数组的索引。在此数组中,数据用开放探测寻址法存放。而在更有效的独立表链地址法中,上述值可以被用作访问链表数组的索引。这两种存储方法都可用于文件。本节中我们用独立表法,文件以链表形式存放。我们在内存中建立哈希表并用它反复访问慢速的磁盘设备。

文件结点中包含了数据和文件索引。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

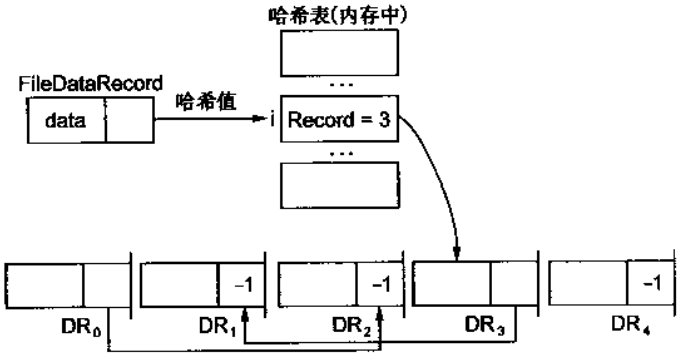
www.Linuxidc.com



FileDataRecord

所有记录以链表形式存于磁盘上，每个记录的 nextIndex 域都指向文件中下一个记录的位置。为建立链表，我们在内存中创建哈希表，其内容指向文件中的各链表。哈希函数将每个数据记录与一个表索引相关联。

```
int hashtable[n];        // 文件索引值的数组
```



FileDataRecord记录(磁盘上)

哈希表是用 n 个元素的数组实现的。初始时表为空(每一项都置为 -1)，表示还没有记录存储到文件中。当我们从数据库中输入记录，由哈希函数确定一个表索引。若该表项为空，则将记录存盘并将文件位置记录在表中。这样，表项中所给出的是哈希映射到该值的第 1 个记录在磁盘上的位置。一旦表项中填了数据，我们就可以用其值访问相应的链表并插入新记录。插入过程将记录写入磁盘并更新 nextIndex 域的指针。

我们用一个简单的例子来说明这一过程的主要特点。

```
// 文件中存放数据记录的结点
struct FileDataRecord
{
    // 本例中数据为整型;一般来说数据为复杂的记录
    int data;
    int nextIndex;        // 磁盘上下一记录的位置
};
```

哈希函数将数据值(整数)映射为其个位值。

```
h(data) = data % 10;          // h(456) = 6    h(891) = 1    h(26) = 6
```

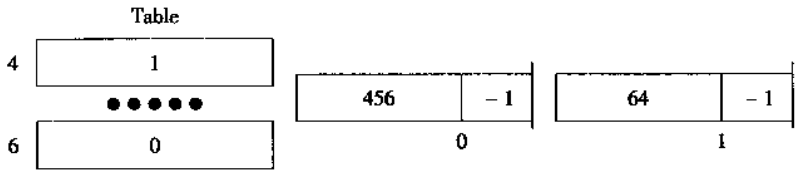
本例子将以下数据记录存入文件：

```
456    64    84    101    144
```

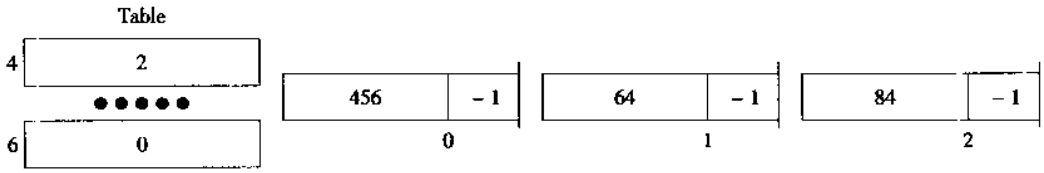
前两项哈希映射到空表项，因此可以直接作为结点插入到文件中。第 1 个结点被存于磁盘位置 0 处，第 2 个结点在位置 1。添加完结点后，在相应的表项中填入其位置值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

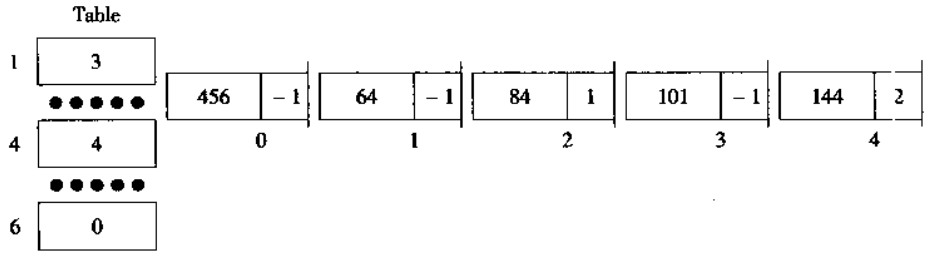
www.Linuxidc.com



对数据值 84,相应表项中已填入标识链表的值 1。新记录被插入到表前端。



装入 101 和 144 后,文件中包含 5 个 FileDataRecord 结点。在逻辑上,这些结点存储于 3 个链表中。表项中包含的是表的起始结点位置。



这种存储法有效地利用了文件的直接访问结构。我们可以通过将记录添加到文件中然后再更新表项的方法为数据建立存储空间。哈希表通常保存在一个单独的文件中,当需要访问数据库时再将其装入内存。

程序 14.3 外部哈希法

本程序示意这一节所讨论的外部哈希算法。函数 LoadRecord 往文件中增加一个新记录,函数 PrintList 以 1 个哈希值为参数,打印出对应链表中的记录。LoadRecord 将每个记录插入到链表的前端。这里不考虑如何避免重复值。主程序插入 50 个取值范围在 0 到 999 之间的随机整数。为示范搜索,要求用户输入一个哈希值,然后将相应的链表中的数据项打印出来。

```
# include <iostream.h>
# include "random.h"
# include "binfile.h"
const long Empty = -1;
// 文件中数据记录的结构
struct FileDataRecord
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 本例中数据为整数
    int data;
    long nextIndex;    // 磁盘上下一记录的位置
};
// startindex 为哈希表的一个下标,以形参形式传递
void LoadRecord(BinFile < FileDataRecord > &bf, long &startindex,
                FileDataRecord & dr)
{
    // 若哈希表非空,startindex 指向表头;否则基值为 -1,bf.Size()给出了文件中下一结
    // 点的位置
    dr.nextIndex = startindex;
    startindex = bf.Size();
    // 往文件中追加新记录
    bf.Append(dr);
}
// 扫描文件的结点表并输出各结点数据值
void PrintList(BinFile < FileDataRecord > &bf, long startindex)
{
    // index 为表中第一个位置
    long index = startindex;
    FileDataRecord rec;
    // 将 index 移到表尾(index = -1)
    while (index ! = Empty)
    {
        // 取记录,输出其值后指向下一记录
        rec = bf.Read(index);
        cout < < rec.data < < " ";
        index = rec.nextIndex;
    }
    cout < < endl;
}
void main(void)
{
    // 文件结点的哈希表,哈希范围为 0 至 9
    long hashTable[10];

    // 随机数生成器及数据记录
    RandomNumber rnd;
    FileDataRecord dr;
    int i, item, request;
    // 以可读写方式打开文件"DRfile"
    BinFile < FileDataRecord > dataFile("DRfile", INOUT);
    // 初始化哈希表
    for(i = 0; i < 10; i + + )
        hashTable[i] = Empty;
    // 在范围 0 到 999 间生成 50 个随机整数
    for (i = 0; i < 50; i + + )
    {
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        item = rnd.Random(1000);
        // 初始化数据记录后将其装入文件及哈希表中
        dr.data = item;
        LoadRecord(dataFile, hashTable[item % 10], dr);
    }

    // 提示用户输入哈希表下标并输出表
    cout << "Enter a hash table index: ";
    cin >> request;
    cout << "Printing entries that hash to " << request << endl;
    PrintList(dataFile, hashTable[request]);
    // 删去数据文件
    dataFile.Delete();
}

/*
<程序 14.3 运行结果>
Enter a hash table index: 5
Printing entries that hash to 5
835  385  205  185  455  5
*/
```

外部文件排序

文件数据排序所提出的特殊问题是如何处理由于文件太大而无法装入内存的情况。既然不可能将所有数据放到一个数组中，我们就用临时文件存储数据项。这一节我们要设计的归并排序中用到 3 个文件。我们讨论直接归并算法，也讨论使用归并段的自然归并算法。这些算法可以扩展到使用 n 个文件的 n 路归并($n>3$)。

第 12 章中我们介绍了将两个有序表合并到一个表中的简单归并法。"直接归并排序(straight merge sort)"正是用了这种方法，它归并的是定长子表。假设初始时元素被存在文件 fC 中，而文件 fA 和 fB 是对数据进行分割得到的临时外部文件。算法按以下步骤进行：

1. 将 fC 中的元素交替写入 fA 和 fB，将 fC 分割成两半。这样在每个新文件中就会产生一系列的单元素子表。
2. 将子表进行配对，从 fA 中选出一个元素，再从 fB 中选出一个元素。将它们归并到 fC 中的一个 2 元素有序子表中。继续这一过程，一直到两个文件中的所有元素都被拷回到 fC 中。
3. 重复步骤 1，将 fC 中的 2 元素有序子表交替写入 fA 和 fB。
4. 将 fA 和 fB 中的 2 元素子表结对归并到 fC 中的 4 元素子表中。继续这一过程，一直到所有文件都被拷回到 fC 中。
5. 重复以上步骤，依次用 4 个、8 个…元素子表将文件 fC 分割为 fA 和 fB。然后将各对子表归并为 fC 中的 8 元素、16 元素等等的有序子表。当文件 fA 和 fB 中只含一

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

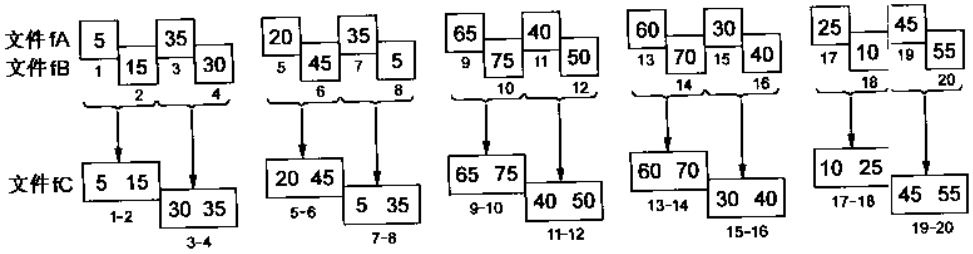
www.Linuxidc.com

个有序子表时，它们经过归并就形成完全有序的文件 fC。

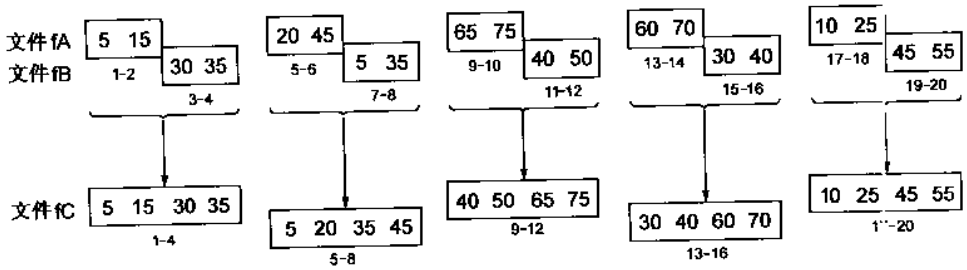
我们以下列 20 个整数为例说明文件 fC 的直接归并排序。

5 15 35 30 20 45 35 5 65 75 40 50 60 70 30 40 25 10 45 55

步骤 1 中，fC 分裂为分别包含 10 个单元素子表的两个临时文件。步骤 2 的操作生成 2 元素有序子表。



在步骤 3，fA 和 fB 中所包含的是 2 元素子表，它们被归并为 fC 中的 4 元素子表。



将临时文件中的子表进行划分然后再将它们归并回 fC 的过程还需要 3 遍操作才能完成，这 3 遍操作分别在 fC 中生成 8 元素、16 元素、最终 20 个元素的子表。最后一遍完成后，fC 就是一个有序文件。在生成 8 元素和 16 元素子表时，fA 尾部有一个“零头”子表，它被直接拷回到 fC 中。在最后一遍，对 fA 中的 16 元素子表和 fB 中的 4 元素子表进行归并，过程终止。

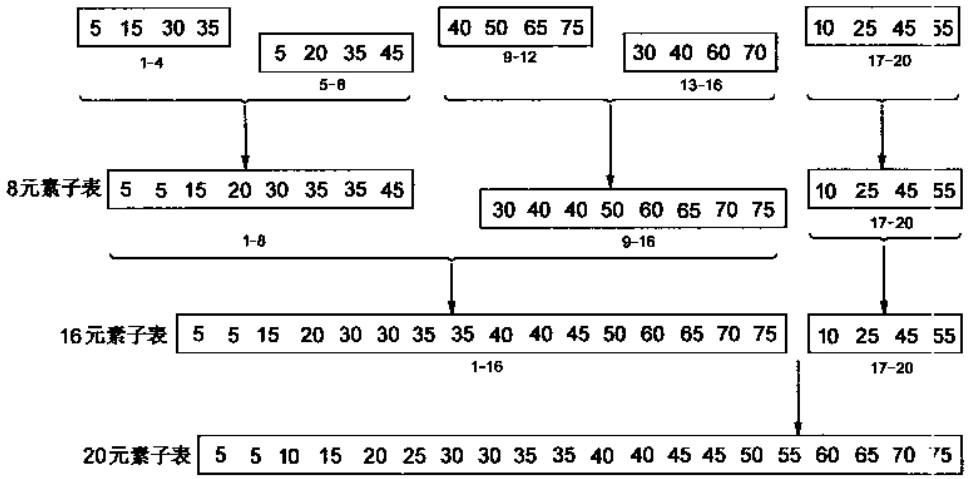
直接归并排序分析 直接归并排序以单元素子表开始，由若干遍操作所组成。每一遍都将子表长度加倍，直到其长度 $s >= n$。这需要独立进行 $\log_2 n$ 遍的如下操作。将 n 个元素复制到临时文件中，然后再将它们复制回文件 fC。直接归并排序总共需要 $2 * n * \log_2 n$ 次数据访问，其数量级为 $O(n \log_2 n)$。

长归并段归并排序

直接归并中使用的子表长度从 1 开始，逐步扩大到 2，4，8，…依此类推。最后有序子表构成了整个文件，外部排序就结束了。对于小的子表，归并算法将它们分割到两个临时文件中再归并回原始文件中所消耗的时间太多而显得不合算。若用较长的子表则算法的效率可以大大提高，因为归并段少了，文件操作也不是那么频繁了。这一节我们将修改直接归并排序，使得我们能够从较长的子表开始，更高效地完成排序。算法根据文件 fC 和一个内存缓冲区建立有序子表。我们将原始文件中的数据以块形式读入到缓冲区中。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



然后用一种快速内部排序法(QuickSort)将块排好序。这些块被交替复制到文件 fA 和 fB 中。归并从具有较大初始长度的子表开始。

为了示意用长子表的效果,表 14.2 中比较了用单元素块以及大小为 10、100 和 1000 的块对 30 000 个随机整数进行排序所花费的时间。

表 14.2　使用长归并段对外部排序的影响

块大小	对 30 000 个整数排序所有的时间(秒)
1	205
10	75
100	49
1000	31

文件归并的实现　函数 MergeSort 建立两个临时文件,然后执行若干遍以下操作:将各相邻归并段分割到临时文件 fA 和 fB 中,然后再把它们归并回原始文件 fC 中。以上过程重复进行,直到 fC 只含一个归并段。

```
// 用块大小不同的归并段对文件 fC 进行排序。先读入一块数据,用 QuickSort 排序后作为归并
// 段交替写入临时文件 fA 和 fB 中
template <class T>
void MergeSort(BinFile<T>& fC, int blockSize)
{
    // 分离从 fC 读出的归并段的临时文件
    BinFile<T> fA("fileA", INOUT);
    BinFile<T> fB("fileB", INOUT);
    // 文件大小及块大小
    int size = int (fC.Size()), n = blockSize;
    int k = 1, useA = 1, readCount;
    T *A;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
// 重置文件 fC 的指针指向文件头
fC.Reset();

// 对小文件,从 fC 读入数据,排序后写回文件中
    if (size < = blockSize)
    {
        // 申请存放数据的缓冲区并读入一块数据
        A = new T[size];
        if (A = = NULL)
        {
            cerr < < "MergeSort: memory allocation failure." < < endl;
            exit(1);
        }
        fC.Read(A, size);
        // 用快速的内部排序法对其排序
        QuickSort(A, 0, (int)size - 1);
        // 清空文件并将排序后数据写回文件中
        fC.Clear();
        fC.Write(A,size);
        // 释放缓冲区并返回
        delete [] A;
        return;
}
else
{
    // 申请缓冲区并读入数据块,直到文件结束
    A = new T[blockSize];
    if(A = = NULL)
    {
        cerr < < "MergeSort: memory allocation failure." < < endl;
        exit(1);
    }
    while (! fC.EndFile())
    {
        readCount = fC.Read(A,blockSize);
        if (readCount = = 0)
            break;
        // 对数据块排序并交替写入到文件 fA 和 fB 中
        QuickSort(A,0,readCount - 1);
        if (useA)
            fA.Write(A,readCount);
        else
            fB.Write(A,readCount);
        useA = ! useA;
    }
    delete [] A;
}
// 将两个文件 fA 和 fB 中块大小为 blockSize 的归并段写回到 fC 中
Merge(fA, fB, fC, blockSize);

// 将当前归并段大小加倍
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    k * = 2;
    n = k * blockSize;
    // 当 n≥文件大小时,fC 仅剩一个已排好序的归并段
    while (n < size)
    {
        // 每次循环,分离归并段,排序后再重新写回到 fC 中
        Split(fA, fB, fC, k, blockSize);
        Merge(fA, fB, fC, n);
        k * = 2;
        n = k * blockSize;
    }

    // 删除临时文件
    fA.Delete();
    fB.Delete();
}
```

每一遍操作中,函数 Split 都要扫描文件 fC 中的归并段并将它们交替复制到文件 fA 和 fB 中。每次调用此函数,子表的大小都被加倍到 k * blockSize。因为块大小代表所使用的缓冲区大小,子表将以 k 个块大小的形式被复制到文件中。当文件 fC 中的所有归并段已经被复制到临时文件中后,操作过程终止。

```
// 扫描文件 fC 中的归并段并将它们交替分送到文件 fA 和 fB 中.本次归并段的大小
// 为 k * blockSize
template < class T >
void Split(BinFile < T > &fA, BinFile < T > &fB, BinFile < T > &fC,
           int k, int blockSize)
{
    int useA = 1;
    int i = 0;
    int readCount;

    // 用大小为 blockSize 的块读写文件
    T * A = new T[blockSize];
    if(A = = NULL)
    {
        cerr < < "MergeSort: memory allocation failure." < < endl;
        exit(1);
    }

    // 开始前初始化文件
    fA.Clear();
    fB.Clear();
    fC.Reset();

    // 从文件 fC 中将归并段分送到 fA 和 fB 中,直到文件结束
    while (! fC.EndFile())
    {
        // 读入一数据块到动态数组中 readCount 为读入元素个数
        readCount = fC.Read(A,blockSize);
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        // 若 readCount 为 0,文件结束
        if (readCount = = 0)
            break;
        // 写入到文件 fA(当 useA 为 True)或文件 fB
        if (useA)
            fA.Write(A,readCount);
        else
            fB.Write(A,readCount);
        // 写完 k 块后,将文件交换
        if ( + + i = = k)
        {
            i = 0;
            useA = ! useA;
        }
    }

    // 释放动态内存
    delete [] A;
}
```

一旦归并段已经被复制到临时文件 fA 和 fB 中,把数据归并回原始文件的过程就可以开始了。这一过程由函数 Merge 处理,它将两个临时文件中的归并段结对合并,生成一个有序归并段。若文件 fA 中多余一个归并段,则调用 CopyTail,将它复制到文件 fC 中。

```
// 将文件 fA 和 fB 中长度为 n 的归并段合并回 fC 中
template < class T >
void Merge (BinFile < T > &fA, BinFile < T > & fB,
            BinFile < T > &fC, int n)
{
    // currA 和 currB 表示归并段在每个文件中所处的位置
    int currA = 1, currB = 1;

    // 分别从 fA 和 fB 中读出的数据项,用 haveA/haveB 为 0/1 来表示是否成功读出
    T dataA, dataB;
    int haveA, haveB;

    // 合并开始前初始化文件
    fA.Reset();
    fB.Reset();
    fC.Clear();

    // 从每个文件中读一个数据段
    fA.Read(&dataA, 1);
    fB.Read(&dataB, 1);

    for (;;)
    {
        // 若(dataA < = dataB),则将 dataA 拷贝到 fC 并修改当前归并段在文件 fA 中的
        // 位置
        if (dataA < = dataB)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        {
            fC.Write(&dataA,1);
            // 从 fA 中取下一归并段,若不存在,则已到文件尾,应将 fB 的后续归并段拷入到
            // fC;若当前位置 > n,则已将所有 fA 的归并段拷完,应拷贝 fB 的后续归并段
            if ((haveA = fA.Read(&dataA, 1)) = = 0 || + + currA > n)
            {
                // 从 fB 中拷贝,并修改当前归并段在 fB 的位置
                fC.Write(&dataB,1);
                currB + + ;
                CopyTail(fB,fC,currB,n);
                // fA 的大小 > = fB 的大小;若在 fA 的文件尾,则结束
                if (! haveA)
                    break;
                // 否则,应在新的归并段中;重置当前位置
                currA = 1;
                // 取 fB 中的下一项。若不存在,则只有 fA 中剩余的部分要拷贝到 fC,
                // 退出循环前将当前归并段写入 fC.
                if ((haveB = fB.Read(&dataB, 1)) = = 0)
                {
                    fC.Write(&dataA,1);
                    currA = 2;
                    break;
                }
                // 否则,置 fB 中当前归并段
                currB = 1;
            }
        }
        else
        {
            // 将 dataB 拷贝到 fC 并修改 fB 中的当前位置
            fC.Write(&dataB,1);
            // 检查是否已是归并段尾或文件尾
            if ((haveB = fB.Read(&dataB,1)) = = 0 || + + currB > n)
            {
                // 若是,将已从 fA 读入的数据项写入 fC,修改其位置后将后续归并段写入
                fC.Write(&dataA,1);
                currA + + ;
                CopyTail(fA,fC,currA,n);
                // 若 fB 中设有更多项,则置 fA 的当前位置,准备拷贝 fA 中的最后的归并段
                if (! haveB)
                {
                    currA = 1;
                    break;
                }
                // 否则,置 fB 的当前位置并从 fA 中读入数据
                currB = 1;
                if ((haveA = fA.Read(&dataA,1)) = = 0)
                    break;
                currA = 1;
            }
        }
    }
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 将 fA 中可能存在的后续归并段写入 fC
    if (haveA && ! haveB)
        CopyTail(fA,fC,currA,n);
}
```

对两个归并段进行合并时，我们会先遇到一个归并段的结束。函数 CopyTai. 将另一归并段的尾部复制到输出文件。

```
// n 为当前归并段大小;将文件 fX 的后续归并段拷入到 fY.变量 currRunPos 为当前归并段
// 的索引
template < class T >
void CopyTail (BinFile < T > &fX,BinFile < T > &fY,
               int &currRunPos, int n)
{
    T data;
    // 从当前位置到归并段结束,拷贝每个数据
    while (currRunPos < = n)
    {
        // 若没有更多的数据项,则文件结束且归并段结束
        if (fX.Read(&data, 1) = = 0)
            return;
        currRunPos + + ;
        // 修改当前位置并将数据项写入 fY
        fY.Write(&data,1);
    }
}
```

程序 14.4　归并排序的比较

本程序用自然归并排序对含 1 000 个随机整数的文件进行排序，归并段长度为 100。用函数 LoadFile 创建初始文件。用 PrintFile 打印出原始文件中的头 45 个元素以及排好序的文件。

```
# include < iostream.h >
# include < iosanip.h >
# include "binfile.h"
# include "merge.h"
# include "random.h"
// 从二进制文件 f 中按 9 个数据一行输出 n 个元素
void PrintFile (BinFile < int > &f, long n)
{
    // 初始化 n 值
    int data;
    long i;
    n = (f.Size() < n) ? f.Size() : n;
    // 置文件指针指向文件头
    f.Reset();
    // 顺序扫描文件,读入各元素并输出其值,每行 9 个元素
    for (i = 0; i < n; i + + )
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    {
        if (i % 9 = = 0)
            cout < < endl;
        f.Read(&data, 1);
        cout < < setw(5) < < data < < " ";
    }
    cout < < endl;
}
// 创建包含 n 个范围在 0 到 32767 的随机整数的文件
void LoadFile(BinFile < int > &f, int n)
{
    int i, item;
    RandomNumber rnd;
    // 重置文件指针
    f.Reset();

    // 将 n 个随机整数写入文件
    for (i = 0; i < n; i + + )
    {
        item = rnd.Random(32768L);
        f.Write(&item,1);
    }
}
void main(void)
{
    // 将随机数装入 fC 并对其排序
    BinFile < int > fC("fileC", INOUT);
    // 往 fC 中装入 1000 个随机整数
    LoadFile(fC,1000);
    // 输出排序前的文件中头 45 个元素
    cout < < "First 45 elements of the data file:" < < endl;
    PrintFile(fC,45);
    cout < < endl;
    // 执行归并排序
    MergeSort(fC, 100);
    // 输出排序后文件中头 45 个元素
    cout < < "First 45 elements of the sorted data file:"
        < < endl;
    PrintFile(fC, 45);
    // 删除文件
    fC.Delete();
}
/*
< 程序 14.4 运行结果 >
First 45 elements of the data file:
  14879  26060  28442  20710  19366  10959  17112   7880  22963
  16103  22910   6789   4976  19024   1470  25654  31721  28709
    997  23379  14186  14986  21650   7351  25237  28059   5942
   9593  20294  27928   8267   9837  17191   8398  18261  21620
   5139    964  10393  16777  15915  18986  22175   2697  20409
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
First 45 elements of the sorted data file:
      19      76      94      98     106     119     188     192     236
     259     308     344     346     371     383     424     463     558
     570     605     614     714     741     756     794     861     864
     891     910     923     964     979     997    1000    1007    1029
    1051    1079    1112    1223    1232    1347    1470    1515    1558
*/
```

14.7 辞典

数组中存放的元素是通过描述其在数组中位置的索引来访问的。例如，若 A 是数组，则 A[n]在数组中位置 n 处。索引不作为数据的一部分存放。“辞典(表，关联数组)”是一种类似于数组的带索引的数据结构。但是其索引值和数据值的类型可以是任意的。例如，若 CommonWords 是一个辞典，则 CommonWords[“decide”]就可以是单词“decide”的定义。与数组不同，辞典索引与数据是有关系的，并非只是指明其存储位置。而且，辞典可以容纳的元素个数是没有限制的。

辞典被称作“关联结构(association structure)”，因此它维护的是键值及其关联数据的表。例如，语言辞典是单词(键)及其定义(值)的表。辞典不同于数组，因为其元素的实际位置是隐含的。不可能通过直接指定表中位置进行访问，而只能通过使用键作为索引完成访问。

辞典使用与其数据值相关的键作为索引访问数据。底层数据在内存中被存储为一系列的键-值对(key-value pairs)，也叫关联组(associations)。这些键-值对可以存储在链表、树或哈希表中。若数据是按键值顺序存放的，则我们说表是有序的。图 14.5 中示意了辞典的概念。

为实现一个实用辞典，我们先设计出存储键-值对的方法，而这又是通过类 KeyValue 实现的。每个键-值对都是一个键值为常量的 KeyValue 对象，且任意两个对象都可以用“==”和“<”进行比较。比较是根据键值进行的。这是我们举的第一个具有两个模板参数的模板类的例子。其中 K 是键值类型，而 T 是与键关联的数据值的类型。

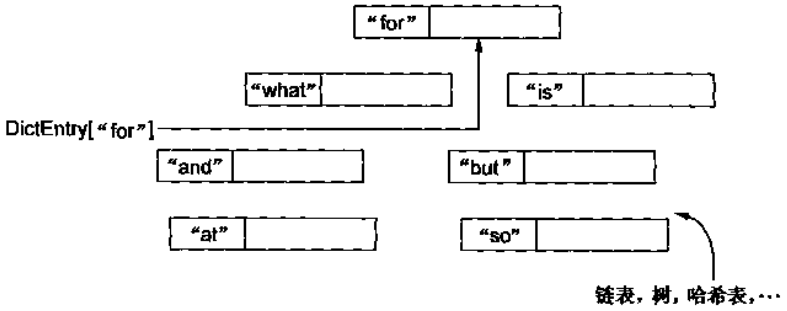


图 14.5 辞典(关联数组)

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

KeyValue 类说明

声明

```
template <class K, class T>
class KeyValue
{
    protected:
        // 一旦 key 被初始化后,不能再改变
        const K key;

    public:
        // 可公共访问的数据
        T value;
        KeyValue(K KeyValue, T datavalue);
        // 赋值操作符,不改变 key 值
        KeyValue<K,T> & operator = (const KeyValue<K,T> & rhs);
        // 比较操作符,基于两个不同的 key 值
        int operator = = (const KeyValue<K,T> & value) const;
        int operator = = (const K& keyval) const;
        int operator < (const KeyValue<K,T> & value) const;
        int operator < (const K& keyval) const;
        // 供外部函数取约 key 值的方法
        K Key(void) const;
}
```

说明

构造函数生成一个键-值对。没有缺省构造函数,且一旦对象建立了,键(key)就不能改变。赋值运算符仅改变数值(value)成员,关系运算符对键(key)值进行比较。方法 Key 可用于读取键(key)值。

例

下面声明的键-值对中,键是社会保险号码(String),值是数据记录(Data)。

```
struct Data
{
    char name[30];
    int yearsThisCompany;
    int jobclass;
    float salary;
};

Data empData = {"George Williams", 10, 5, 45000.00};
KeyValue<String, Data> Employee("345789553",empData);
```

KeyValue 类的实现非常简单,在文件"keyval.h"中可找到它。

要实现非常复杂的辞典概念实际上也比较容易。选择一个集合类以存储 KeyValue 对象。可以用类 OrderedList, BinSTree 和 AVLTree 存储有序的键-值对。若要存储无序的键-值对,可以用集合类 SeqList 或 HashTable。这些类中的每一个都具有方法 Insert,

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Delete，Find 等。为创建辞典，我们必须做的是对这样的集合进行功能扩展，使其具有下标运算符“[ ]”。此运算符在一个用作下标的键与集合中所存储的相应的 KeyValue 对象的 value 域之间建立关联。本书中我们曾经用继承表达“是(is a)”关系。继承的另一个应用是扩展基类的功能。我们将用继承扩展键-值对的集合，使之具有“[ ]”运算符以及其他与辞典有关的操作。为说明上述思想，图 14.6 示意了如何由几种基类派生出 Dictionary 类。

我们将设计由 BinSTree 类派生出的类 Dictionary，这样它可以维持一个有序辞典。其他的有序和无序辞典的实现在习题中给出。

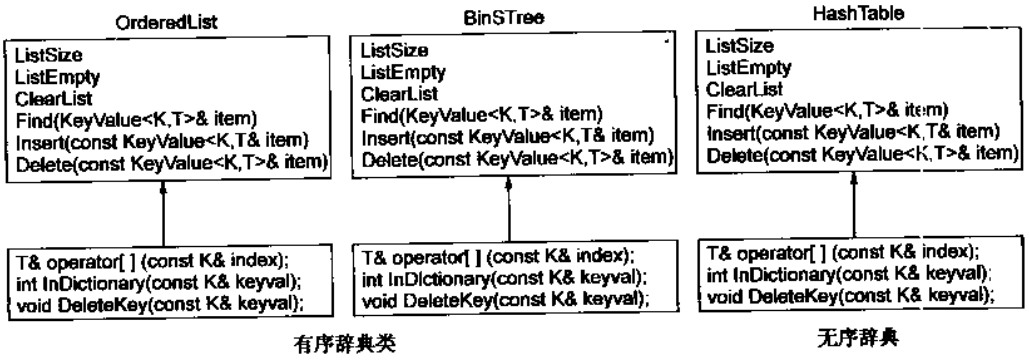


图 14.6　将 KeyValue 对象集扩充为辞典

Dictionary 类的描述

声明

```
#include "keyval.h"
#include "bstree.h"
#include "treeiter.h"
template <class K, class T>
class Dictionary: public BinSTree<KeyValue<K,T>>
{
    // 创建辞典的缺省值，用于下标运算符，InDictionary 和 DeleteKey
    private:
        T defaultValue;
    public:
        // 构造函数
        Dictionary(const T& defval);
        // 下标运算符
        T& operator[] (const K& index);
        // 其他辞典方法
        int InDictionary(const K& keyval);
        void DeleteKey(const K& keyval);
};
```

说明

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

下标运算符完成大部分工作。当它被调用时，则在辞典中检查键值是否存在。如果存在具有该键值的表项，则返回数据元素的引用；若不存在，则生成一个新词条并返回指向新值的引用。这样，所有词条的建立和数据更新都可以用索引运算符完成。KeyValue类没有缺省构造函数，因此必须指定键值和数据值。这样，当"[]"建立对象时，对象必须有缺省值。此缺省值是以参数形式提供给构造函数的。缺省值必须仔细选择以使新词条在表达式中易于操作。

例如，辞典可能由String类型的描述单词的键以及String类型的定义单词的值所组成。我们可以将Dictionary对象BasicDict声明为

```
// defaultValue 为空串
Dictionary < String,String > BasicDict("");
```

假设在接下来的一条语句中，BasicDict["sextet"]被第一次引用。运算符"[]"为"sextet"建立一个值为NULL的词条。String类的"+ = "运算符将"A group of six"连接到NULL串以建立单词的初始定义。

```
BasicDict["sextet"] + = "A group of six";
```

方法InDictionary检查集合中是否包含key域为keyval值的键-值对，而DeleteKey则删除key域为keyval值的词条。方法ListEmpty，ListSize和ClearList可由基类得到。也可以用基类方法Insert，Find和Delete直接处理KeyValue对象，但在辞典操作中一般不用它们。

多数应用中需要一个辞典迭代算子以收集输出数据。既然Dictionary对象由BinSTree派生得到，那么也可以用类InorderIterator声明一个简单的DictionaryIterator类，如下所示：

```
template < class K, class T >
class DictionaryIterator :
                        public InorderIterator < KeyValue < K,T > >
{
    public:
        // 构造函数
        DictionaryIterator(Dictionary < K,T > & dict);
        // 辞典迭代算子
        void SetList(Dictionary < K,T > & dict);
};
// 构造函数.dict"扩展"BinSTree 对象.用公共方法 GetRoot 初始化 InorderIterator 基类
template < class K, class T >
DictionaryIterator < K,T > ::DictionaryIterator
                        (Dictionary < K,T > & dict):
            InorderIterator < KeyValue < K,T > > (dict.GetRoot())
{}
// 用基类方法 SetTree
template < class K, class T >
void DictionaryIterator < K,T > ::SetList(Dictionary < K,T > & dict)
{
    SetTree(dict.GetRoot());
}
```

Dictionary和DictionaryIterator的实现可在文件"dict.h"中找到。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

程序 14.5　编制一个单词辞典

本程序声明一个 Dictionary 对象 WordDictionary，它包含一个 String 键值(key)和一个 String 数据(data)。辞典数据项的缺省值为 NULL 串。文件"defs.dat"中包含单词列表及其定义。单词被放在每一行的开始，其后是空格，剩下的部分即是单词的定义。用一个循环读出每个单词并以它为键值将其定义添加到辞典中。

程序中建立了一个辞典迭代算子对象 dictIter 并用它来遍历辞典。得到各个键-值对以后，调用函数 PrintEntry 将词条打印出来。函数先打印出单词，其后跟以连字符，然后按每行 65 个字符打印其定义。单词不能跨行。

```
#include <fstream.h>
#include <stdlib.h>
#include "keyval.h"          // 迭代算子返回的 KeyValue 对象
#include "dict.h"            // 引入辞典类
#include "strclass.h"        // 键和数值域均为串
// 取出一个包含有单词及其意义的 KeyValue 对象，并输出
void PrintEntry(const KeyValue<String, String>& word)
{
    KeyValue<String,String> w = word;
    // 单词后需跟"-"，故其意义在单词长度加 3 个字符的位置开始输出
    int linepos = w.Key().Length() + 3;
    int i;

    // 输出后跟"-"的单词
    cout << w.Key() << "-";
    // 按每行 65 个字符输出单词的意义
    while(! w.value.IsEmpty())
    {
        // 判断该行还能打多少个字符，决定最后一个字符的下标
        if(w.value.Length() > 65-linepos)
        {
            // 不能在一行打完，退到上一个空格符，以使不将单词分开在两行输出
            i = 64-linepos;
            while(w.value[i] ! = ' ')
                i--;
        }
        else
            // 可在一行内输出
            i = w.value.Length()-1;
        // 输出可在该行输出的子串
        cout << w.value.Substr(0,i+1) << endl;
        // 删除已输出子串，准备输出下一行
        w.value.Remove(0,i+1);
        linepos = 0;
    }
}
void main(void)
{
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    // 供程序读入数据的流
    ifstream fin;
    String word, definition;
    // 辞典
    Dictionary < String, String > wordDictionary("");
    // 打开存有单词及意义的文件"defs.dat"
    fin.open("defs.dat",ios::in | ios::nocreate);
    if (! fin)
    {
        cerr < < "The file 'defs.dat' is not found." < < endl;
        exit(1);
    }
    // 读入单词及其意义。通过下标运算符,插入该单词及其意义或将新意义拼在原有意义之后
    while(fin > > word)
    {
        if (fin.eof())
            break;
        // 读入单词后的空格
        definition.ReadString(fin);
        wordDictionary[word] + = definition;
    }
    // 定义迭代算子来顺序遍历辞典
    DictionaryIterator < String,String > dictIter(wordDictionary);
    // 遍历辞典,输出每个单词及其意义
    cout < < "The dictionary is:" < < endl < < endl;
    for(dictIter.Reset();! dictIter.EndOfList();dictIter.Next())
    {
        PrintEntry(dictIter.Data());
        cout < < endl;
    }
    wordDictionary.ClearList();
}
/*
< 文件"defs.dat" >
Program A list of the acts, speeches, pieces.
finish To bring to an end.
cause Anything producing an effect or result.
sextet A group of six performers.
program A sequence of operations executed by a computer.
velocity Quickness of motion; swiftness.
cook To prepare by boiling, baking, frying, etc.
muff A cylindrical covering of fur to keep the hands warm.
banner A headline running across a newspaper page.
sextet A composition for six instruments.
< 程序 14.5 运行结果 >
The dictionary is:
banner - A headline running across a newspaper page.
cause - Anything producing an effect or result.
cook - To prepare by boiling, baking, frying, etc.
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
finish - To bring to an end.
muff - A cylindrical covering of fur to keep the hands warm.
program - A list of acts, speeches, pieces. A sequence of
operations executed by a computer.
sextet - A group of six performers. A composition for six
instruments.
velocity - Quickness of motion; swiftness.
*/
```

Dictionary 类的实现

构造函数对基类和缺省值进行初始化：

```
// 构造函数,初始化基类及缺省值
template <class K, class T>
Dictionary<K,T>::Dictionary(const T& defaultval):
        BinSTree<KeyValue<K,T>>(), defaultValue(defaultval)
{}
```

"[]"运算符建立一个具有指定的键值以及缺省数据值的 KeyValue 对象 targetKey，并在树中搜索该键值。如果没有找到，则将 targetKey 插入到树中。基类成员 current 指向刚刚找到或插入的结点。返回指向结点中数据值的索引。

```
// 下标运算符
template <class K, class T>
T& Dictionary<K,T>::operator[](const K& index)
{
    // 定义一个带有缺省值的目标对象 KeyValue
    KeyValue<K,T> targetKey(index, defaultValue);
    // 搜索键值.若未找到,则插入 targetKey
    if (! Find(targetKey))
        Insert(targetKey);
    // 返回指向找到或插入对象的数据值
  return current->data.value;
}
```

为实现 InDictionary，建立一个具有给定的键值以及缺省数据值的 KeyValue 对象 tmp。在树中搜索该键值并返回结果。

```
// 判断给定键值的 KeyValue 对象是否存在
template <class K, class T>
int Dictionary<K,T>::InDictionary(const K& keyval)
{
    // 定义一个带有缺省值的目标对象
    KeyValue<K,T> tmp(keyval, defaultValue);
    int retval = 1
    // 搜索 tmp 并返回结果
    if (! Find(tmp))
        retval = 0;
    return retval;
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

DeleteKey 建立一个具有给定的键值以及缺省数据值的 KeyValue 对象 tmp，并将辞条从树中删除。

```
// 从辞典中删除给定键值的 KeyValue 对象
template <class K, class T>
void Dictionary<K,T>::DeleteKey(const K& keyval)
{
    KeyValue<K,T> tmp(keyval, defaultValue);
    Delete(tmp);
}
```

书面作业

14.1 (a) 用选择排序对以下序列排序：8,4,1,9,2,1,7,4。必须说明每一遍结束以后表的情况。

(b) 对以下字符表重复(a)中步骤：

V,B,L,A,Z,Y,C,H,S,B,H

14.2 用插入排序法重复书面作业 14.1 中的操作。

14.3 用插入排序对字符序列 C,A,M,T,B,B,A,L 进行排序。跟踪排序的每个步骤。

14.4 (a) 当选择排序对含 n 个相同元素的数组进行排序时，其运行效率如何？

(b) 针对插入排序和冒泡排序，回答上述问题。

14.5 用冒泡排序对数组 A 进行排序。列出每一遍结束以后表中内容并指明仍需要排序的子表。

A = 85,40,10,95,20,15,70,45,40,90,80,10

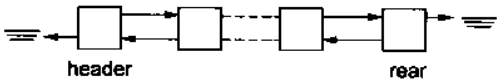
14.6 用快速排序算法对数组 A 进行排序。将中心点(pivot)选为表的中点(midpoint)。在每一遍操作过程中，列出所有在高端子表和低端子表的元素对之间发生的交换。并列出每一遍操作以后的元素次序。

A:790 175 284 581 374 799 852 685 486 347

14.7 另有一种版本的快速排序法，将中心点选为 A[low]而不是 A[mid]。算法复杂度仍为 $O(n \log_2 n)$，但最坏情况下的性能会有所变化，如何变化？

14.8 通过将其元素插入到双向链表中的方法对数组 A 进行排序。算法对链表的当前操作位置进行维护，当需要插入表项时，若新元素比当前位置值要大，则往前移动；若小则往后移动。编写函数 DoubleSort 实现上述排序算法。

```
template <class T>
void DoubliSort (T a[], int n);
```



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

14.9 若用书面作业 14.8 中的方法建立有序表，求其算法复杂度。讨论应包括最好、最坏和一般情况。

14.10 哪一种基本排序算法（选择、插入、冒泡）可以最有效地处理已排序的表？若原始表是以逆序排列则答案相同吗？

14.11 本书中我们已经介绍过以下排序方法：

二叉树、冒泡、交换、堆、插入、基数、选择、竞赛

对每一种排序法，说明其复杂度和空间需求，并对其效率进行评价。评价内容包括当表已经排好序时是否能及早退出，最坏的情况发生的可能性有多大，算法所实施的交换次数以及 O 比较常数的大小。

14.12 如果两个具有相同值的数据项在排序过程中其相对位置没有发生方向性改变则我们称该排序算法是"稳定的(stable)"。例如，对于 5 个元素的数组

5_1 55 12 5_2 33

稳定的排序算法可以保证最后次序是

5_1 5_2 12 33 55

分析书面作业 14.11 中每种算法的稳定性。

14.13 说明若 m 为偶数则哈希函数

```
hashf(x) = s % m
```

无法使用。若 m 为奇数情况有变化吗？（提示：考察一下随机偶数和随机奇数的分布。）

14.14 假设哈希函数具有以下特性：

键值 257 和 567 哈希映射为 3。
键值 987 和 313 哈希映射为 6。
键值 734,189 和 575 哈希映射为 5。
键值 122,391 哈希映射为 8。

假设插入顺序为 257,987,122,575,189,734,567,313,391。

(a) 若使用开放探测寻址法解决冲突，试标明数据的位置。

HT

0	1	2	3	4	5	6	7	8	9	10

(b) 若使用独立表链地址法解决冲突，试标明数据的位置。

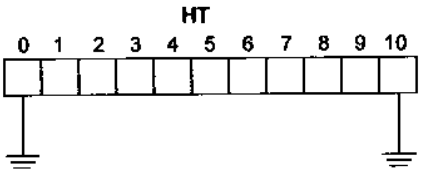
14.15 若插入次序完全颠倒，重复做一遍书面作业 14.14。

14.16 用哈希函数 hashf(x) = x % 11 将整数值映射为哈希表的索引。数据 1,13,12,34,38,33,27,22 将被插入到哈希表中。

(a) 用开放探测寻址法建立哈希表。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com



(b) 用独立表链地址法建立哈希表。

(c) 求以上两种方法的装载因子、在表中定位一个值所需的平均探测次数以及确知值不在表中所需的平均探测次数。

14.17 试说明将串中字符相加从而将串映射为整数的函数不是好的哈希函数。讨论如何移位才能改善这一情况。

14.18 有时我们用折叠(folding)技术编写哈希函数:将键分割为若干部分,再综合各部分,产生一个较小的整数值,这个值将被用作哈希值,也可以用除法进一步缩小它。假定程序必须将社会保险号作为键值处理。我们将键值划分为3个3位数,然后将它们相加。这样产生的整数范围是0~2997。例如,社会保险号码523456795产生的哈希索引为523+456+795=1774。编写哈希函数

```
int hashf(char * ssn);
```

实现以上方法。(提示:要提取子串并将每个子串转换为整数。)

14.19 已知哈希函数

```
unsigned short hashf(unsigned short key)
{
    return (key>>4) % 256;
}
```

(a) 哈希表的长度是多少?

(b) hashf(16)和hashf(257)的值分别是多少?

(c) 概括地说,此哈希函数执行什么操作?

14.20 已知哈希函数

```
unsigned long hashf(unsigned long key)
{
    return (key * key>>8) % 65536;
}
```

(a) 哈希表的长度是多少?

(b) hashf(16)和hashf(10000)的值分别是多少?

(c) 概括地说,此哈希函数执行什么操作?

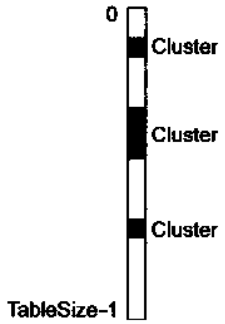
14.21 开放探测法所带来的一个问题是它会产生表集束(clustering)现象。当发生冲突时,表项在表中趋于"束在一起(bunch together)"。

假设有n个表项。若哈希函数良好,则哈希映射到位置p的概率有多大?一旦数

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

据占据了表中位置 p，则位置 p+1 可以被映射到位置 p 或 p+1 的数据所占据。位置 p+1 被填充的概率是多少？填充位置 p+2 的概率又是多少？针对一般情况，解释为什么会产生集束现象。



14.22 若数据不在首次哈希索引处，则开放探测法执行以下函数：

```
index = (index + 1) % m;     // 取下一索引
```

这种函数叫做"再哈希函数(rehash function)"，开放探测正是使用"再哈希法(rehushing)"解决冲突。上述方法助长了集束现象。用不同的哈希函数可以较好地将表项分散开来。如果两个整数没有异于 1 的公因子则我们称它们互素。例如，3 和 10、18 和 35 都是互素的。对于开放探测法，用以下形式的再哈希函数：

```
index = (index + d) % m;
```

其中，d 和 m 互素。若成功地运用这种关系，索引相继覆盖 0 ~ m - 1 的范围。传统的开放探测法使用 d = 1。

(a) 若 d 和 m 不互素，则有不少表项被闲置不用。试说明，若 d = 3，m = 93，再哈希函数 index = (index + 3) % 93
仅仅覆盖三分之一的表项。

(b) 说明若 m 是素数且 d < m，则整个表可以被再哈希函数覆盖。

(c) 用下列再哈希函数重做一遍书面作业 14.16(a)：

```
index = (index + 5) % 11
```

14.23 哈希表适合于主要操作为检索的应用场合，即数据记录被插入又多次被查询。开放探测哈希法不适用于那些需要从哈希表中删除数据元素的应用程序。
考察下面的哈希表，它有 101 个表项，哈希函数为 hashf(key) = key%101：

0	202
1	304
2	508
3	707
100	

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(a) 在表位置 1 处删除 304，放入 -1。对 707 进行查找时会发生什么？针对一般情况，解释为什么将表项置为空不是解决删除问题的正确方法。

(b) 要解决以上问题，可以将一个特定键值 DeletedData 放在删除位置。当查找键值时，标有 DeletedData 的表位置将被跳过。在表中可以用键值 -2 表示在特定表位置发生了删除。说明用这种方法删除 304 就可以使 707 的查找正确进行。必须修改开放探测算法中的插入和检索操作以适应删除操作。

(c) 描述删除表元素的算法。

(d) 描述在表中查找一个元素的算法。

(e) 描述往表中插入一个元素的算法。

14.24 有时我们用到另一种冲突解决方法，那就是"联合表链地址(chaining with coalescing lists)"。这种方法类似于开放探测法，但它要用链表将那些以循环推进方式在表中放置的冲突数据串在一起。每个链中可能包括键值哈希到不同表位置的数据。这些表是联合在一起的。例如，若 hashf(x) = x%7，整数 12，3，5，20 和 7 被输入到表中，我们得到以下图：

-1 代表空项及 NULL 指针

0	20	1
1	7	-1
2	-1	-1
3	3	-1
4	-1	-1
5	12	6
6	5	0

(a) 用联合表链地址法重做一遍书面作业 14.16(a)。

(b) 你认为本方法的运行效率能与开放探测法和独立表链地址法相比吗？就总体速度对它们进行排名。

(c) 在本方法中实施删除操作是否比开放探测法中的简单？解释为什么。

14.25 给定一组键值 $k_0, k_1, \cdots, k_{n-1}$，不产生任何冲突的哈希函数 H 被称为"完美哈希函数(perfect hashing function)"。除非键的集合是静态的，否则寻找完美哈希函数是不切实际的。需要完美哈希函数的一种情形是编译器所要查询的保留关键字(如 c++ 中的"while"，"template"，"class"等)表。当读入一个标识符时，只需一次探测即可判定它是不是保留关键字。

为一组特定键值找一个完美哈希函数是非常困难的，对这个课题进行一般性讨论已经超出了本书的范围。再者，如往集合中加入一组新的键值，则哈希函数通常不再"完美"了。

(a) 现有一组整数键值 81，129，301，38，434，216，412，487，234 以及哈希函数 H

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(x) = (x + 18)/63。

问:H 是否完美哈希函数?

(b) 已知由下列字符串组成的键值集合:

Bret,Jane,Shirley,Bryce,Michelle,Heather

为一个包含 7 个元素的表设计一个完美哈希函数。

14.26 以下声明建立一个文本文件类。其操作以 Pascal 程序设计语言为原型。用C++的 fstream 操作实现此类。

```
enum Access {IN, OUT};                    // 定义文件的数据流
class PascalTextFile
{
    private:
        fstream f;                        // C++文件流
        char fname[64];                   // 文件名
        Access accesstype;                // 数据流为 IN 或 OUT
        int isOpen;                       // 供 Reset 使用
        void Error(char *msg);            // 用于输出错误信息
    public:
        PascalTextFile(void);             // 构造函数
        void Assign(char *filename);      // 置文件名
        void Reset(void);                 // 打开输入文件
        void Rewrite(void);               // 打开输出文件
        int EndFile(void);                // 读到文件尾标志
        void Close(void);                 // 关闭文件
        int PRead(T A[], int n);          // 读入 n 个字符到 A 中
        void PWrite(T A[], int n);        // 将 A 中的 n 个字符写入文件
};
```

上机题

14.1 编写一个程序,用书面作业 14.8 中的双向链表插入算法建立取值范围在 0~1 000 之间的 N 个随机整数的有序表。表建好以后,打印其元素。

14.2 实现以下排序算法:

将 n 个元素的表一分为二。用选择排序法对每一半进行排序;然后对这两半进行归并。

(a) 对此排序法进行复杂度分析。

(b) 用此算法对 20 000 个随机整数的数组进行排序,计量程序的运行时间。

(c) 用普通的选择排序法运行同样的程序。哪个版本更快一些?

14.3 在 14.6 节中我们曾讨论了文件的直接归并排序。试实现此算法的"内排序版本",即对 n 个元素的数组排序。用编出的程序对 1 000 个随机双精度数进行排序。打印出前 20 项和后 20 项。

14.4 有记录定义如下:

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
struct TwoKey
{
    int primary;
    int secondary;
};
```

建立一个含 100 个记录的数组。主键(primary)域放 0～9 范围内的随机整数；辅键(secondary)域放 0～100 范围内的随机整数。调整插入排序算法，使 primary 保持有序，primary 键值相同时再用 secondary 键排序。用上述算法对数组排序，用以下格式打印数组中的每条记录：

```
primary(secondary)
```

14.5 希尔排序以其发明者 Donald Shell 而命名。这是一种简单而极其有效的排序算法。这种排序法先将 n 个元素的表划分为 k 个子表，其成员如下：

a[0],a[k+0],a[2k+0],…
a[1],a[k+1],a[2k+1],…
…
a[k-1],a[k+(k-1)],a[2k+(k-1)],…

每个子表的头一个元素 a[i]在 a[0]～a[k-1]中取值，子表中包括了后续的所有索引与之相距为 k 的整数倍的元素。例如，k=4 时，下列数组被划分为 4 个子表：

7 5 8 6 2 4 9 1 3 0

子表 0	7	2	3
子表 1	5	4	0
子表 2	8	9	
子表 3	6	1	

用插入排序算法对每个子表排序。在上述例子中，我们将得到如下子表：

子表 0	2	3	7
子表 1	0	4	5
子表 2	8	9	
子表 3	1	6	

其对应的部分排好序的数组如下：

2 0 8 1 3 4 9 6 7 5

令 k=k/3，重复以上过程，一直到 k=1 时，表就排好序了。如何选择起始值 k 是算法理论的课题之一。这种算法之所以成功，是因为数据交换发生在数组的非毗邻段，与普通插入排序中的交换相邻项相比，它将元素挪向离最终位置更近的地方。

在主程序中建立一个含 100 个范围在 0 到 999 之间的随机整数的数组。令初始值 k=40，用希尔排序对数组进行排序。按每行 10 个整数打印出原始表和最终表。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

14.6 本题设计一个简单的拼写检查器。补充程序文件“words”中包含500个常用词，它们均由空格隔开。读文件并将单词插入到哈希表中。读一个文档并将其划分为一系列单词。单词由下列简单函数所确定。

```
// 摘取一个字母打头,后跟字母/数字的单词
int GetWord (ifstream& fin, char w[])
{
    char c;
    int i = 0;
    // 跳过非可见字符的输入
    while (fin.get(c) && ! isalpha(c));
    // 若文件结束,返回 0
    if (fin.eof())
        return 0;

    // 记录单词的第一个字母
    w[i++] = c;

    // 取后面的字母和数字,最后用 NULL 结束该单词
    while (fin.get(c) && (isalpha(c) || isdigit(c)))
        w[i++] = c;
    w[i] = '\0';
    return 1;
}
```

用哈希技术，打印出看来是拼错的单词。

14.7 设计出用开放探测寻址法维护哈希表的类 OpenProbe 和 OpenProbeIterator。以下是 OpenProbe 类的说明。用书面作业 14.23 中的方法实现该类。

```
// 取放控测表中的记录
template <class T>
struct TableRecord
{
    // 是否存在该项(True 或 False)
    int available;
    T data;
};
template <class T>
class OpenProbeIterator;
template <class T>
class OpenProbe: public List<T>
{
    protected:
        // 动态创建开放探测表及其大小
        TableRecord<T> *table;
        int tableSize;
        // 哈希函数
        unsigned long (*hf) (T key);
        // 可访问的最后表元素的下标
        int lastIndex;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    public:
        // 构造函数,析构函数
        OpenProbe(int tabsize, unsigned long hashf(T key));
        ~OPenProbe(void);
        // 标准的表处理函数
        virtual void Insert(const T& key);
        virtual void Delete(const T& key);
        virtual int Find(T& key);
        virtual void ClearList(void);
        // 修改可访问的最后表元素
        void Update(const T& key);
        friend class OpenProbeIterator<T>;
};
```

用所编写的类运行程序 14.2。

14.8 将书面作业 14.27 中的类 PascalTextFile 的声明放入文件"ptf.h"中。编写一段程序,用此类读取文件"ptf.h",将每个小写字母转换为大写,然后将其写入到文件"ptf.uc"中。用操作系统的输出命令列出文件"ptf.uc"的内容。

14.9 在下列各程序段中用 BinFile 类。某一段程序的输出可以用作下一个应用的输入。

(a) Person 记录定义的是数据库中的一系列字段。

```
Struct Person
{
    char first[20];     // 名
    char last[20];      // 姓
    char id[4];         // 4 个数字的标识号
};
```

定义函数 DelimRec,其参数为 Person 记录和 buffer(缓冲区)。

```
void DelimRec (const Person & p, char * buffer);
```

函数将记录的每一个域都转换为由定界符"|"结尾的变长字符串。这 3 个域在缓冲区中连在一起。例如:

Person	First	Tom
	Last	Davis
	ID	6192
Buffer	Tom|Davis|6192|	

编写一段程序,输入 5 个 Person 记录,每行 1 个域,并建立字符文件"rec1.out"。对每个记录,建立一个长度为 n 的致密缓冲区,然后将缓冲区大小作为一个 2 字节短整数写入文件中,后面再写入缓冲区中的 n 个字符。若系统有十六进制码的实用程序,则用它列出"rec1.out"的内容。

(b) 编写一段程序,从文件"rec1.out"中连续读取记录,将由定界符隔开的各个域还原到 Person 记录的定长域中。根据需要,在各个域的右边补足空格。这样每个 Person 记录就是一个 44 字节的结构。将记录写入文件"rec2.out"中。

(c) 编写一段程序,输入一个 4 位数的 ID,然后在文件"rec2.out"中寻找匹配值。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

如果找到，打印此人的名(first name)和姓(last name)。

14.10 假设有记录类型如下：

```
struct charRec
{
    char key;
    int count;
};
```

用 BinFile 类建立文件“letcount”，文件由 26 个上述记录组成，记录中所包含的键(key)值分别是‘A’，…，‘Z’，计数值(count)都为 0。读一个文本文件，将所有字母都转换为大写。更新二进制文件“letcount”中相应记录的 count 域。打印出每个字符的频度计数。用 14.6 节中所讨论的自然归并排序法，令块大小为 4，以 count 域为键，对“letcount”进行排序。打印结果文件。

14.11 从 OrderedList 类派生出一个有序辞典类。用此类及与其关联的迭代算子运行程序 14.5。

14.12 由 Hash 类派生一个辞典类。用此类及其关联迭代算子运行程序 14.5。按迭代算子的访问次序打印出辞典对象。对遍历结果进行排序并按字母顺序打印出辞典对象。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

附录　部分书面作业答案

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 1 章

1.2 (a) **ADT** Cylinder is

Data

圆柱的半径和高度,其值为大于 0 的浮点数。

Operations

Constructor

Initial Values: 圆柱的半径和高度。

Process: 用初始值指定 ADT 数据。

Area

Input: 无

Preconditions: 无

Process: 用半径和高度计算面积。

Output: 返回面积。

Postconditions: 无

Volume

Input: 无

Preconditions: 无

Process: 用半径和高度计算体积。

Output: 返回体积。

Postconditions: 无

end ADT Cylinder

1.3 令 Cyl 和 Hole 是半经分别为 R 和 Rh 的圆柱体,令 C 是半经为 Rh 的圆。

(a) 产生的圆筒的体积为 Cyl.Volume - Hole.Volume( )。

(b) 圆筒的表面积为 Cyl.Area( ) + Hole.Area( ) - 4 * C.Area()。

1.5
```
const float PI = 3.14159;
class Cylinder
{
    private:
        float radius, height;
    public:
        cylinder(float r, float h): radius(r),height(h) { }
        float Area(void)
        {return 2.0 * PI * radius(radius + height);}
        float volume(void)
        {return PI * radius * radius * height;}
};
```

1.11 (a) 类继承层次结构中两个或两以上的对象具有同名的但执行不同任务的方法。这一特性使得不同类对象可以对同一消息作出响应。消息的接收者在运行时动态确定的。

第 2 章

2.1 (a) 5 (b) 14 (c) 55 (d) 127

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

2.3 (a) 26 (b) 1055 (c) 4332 (d) 255 (e) 65536
(f) 17 (g) 57 (h) 73 (i) FF

2.4 (a) C (b) A6 (c) F2 (d) BDE3 (e) 11000010000
(f) 1010111100100000

2.5 (a) 32 50 32 (b) 32 32 40

2.8 (a) 'N' (b) 'K' (c) '*': 42_{10}, 101010_2 'q': 113_{10}, 111001_2 <cr>: 13_{10}, 1101_2

2.9 V 113 8

2.11 (a) 6.75

(d) $.111...111... = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} + \cdots = 1 - \left(\frac{1}{2}\right)^n$

当 $n \to \infty$ 时,分数部分趋近于 1。因而等价的十进制算式为 3 + 1 = 4。

2.12 (b) 1.001

2.14 (a) 40f00000 (d) 29.125

2.17 X = 55, Y = 10, A = {5.3 6.8 8.9 1 5.5 3.3}

2.18 (a) (1) 为 A 分配 10 个字节。
(2) &A[3] = 6000 + 2 * 3 = 6006, &A[1] = 6000 + 2 * 1 = 6002
(b) (1) 33
(2) A = {60,50000, - 10000,10,33}
(3) &A[3] = 2050 + 4 * 3 = 2062

2.20 (a) 30 * 2 = 60 字节
(b) &A[3][2] = 1000 + 3 * 12 + 2 * 2 = 1040,
&A[1][4] = 1000 + 1 * 12 + 4 * 2 = 1020
(c) &A[1][4] = 1020, &A[2][5] = 1034

2.22 (a) 't','t',NULL (b) Stockton,CA,March 5, 1994 (c) 1 (d) 1

2.23
```
void strinsert (char *s, char *t, int i)
{    char tmp[128];            // 存放 s 的尾部
     if (i > strlen(s))        // 若下标 i 超出了串 s 中的 NULL 字符,则
                               // 退出
          return;
    strcpy(tmp, &s[i]);        // 将 s 的尾部拷入到 tmp 中
    strcpy(&s[i],t);           // 用 t 替换 s 的尾部
    strcat(s,tmp);             // 再拼接上 tmp 中的原尾部
}
```

2.25 (b)
```
void PtoCStr(char *s)
{    int n = *s++;             // 字符个数
     while(n--)                // 将每个字符左移一位
          *(s-1) = *S++;
     *S = 0;                   // 以 NULL 结束新串
}
```

2.27
```
Complex cadd(Complex& x, Complex& y)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{
    Complex sum = {x.real + y.real,x.imag + y.imag};
    return sum;
}
Complex cmul(Complex& x, Complex& y)
{    Complex product = {x.real * y.real - x.imag * y.imag,
                        x.real * y.imag + x.imag * y.real};
    return product;
}
```

第 3 章

3.2 (b)

```
class Box
  {
    private:
        float length, width,height;
    public:
        box(float l, float w, float h);
        float GetLength(void) const;
        float GetWidth(void) const;
        float GetHeight(void) const;
        float Area(void) const;
        float Volume(void) const;
  };
Box::Box(float l, float w, float h): length(l), width(w),
height(h) {}
float Box::Area(void)
{return 2.0 * (l * w + l * h + w * h);}
```

(c) 编写函数 intQuality(Box B),若箱子合格则返回 1;否则返回 0。例如:

```
if ((2 * (B.GetLength() + B.GetWidth()) + B.GetHeight()) < 100)
return 1;     // 等等
```

3.3 (a) private 和 public 后必须跟以“:”。

最后一个“}”后必须跟以“;”。

(b) Y(int n, int m): p(n), q(m) {}

3.4 (a)

```
class X
{
    private:
        int a, b, c;
    public:
        X(int x = 1, int y = 1, int z = 1);
        int f(void);
};
```

(b) X::X(int x, int y, int z): a(x), b(y), c(z) {}

3.5

```
Class Student
{   ...
    public:
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
        Student(int id, int studgradepts, int studunits):
            studentid(id), gradepts(studgradepts),
            units(studunits)
        {ComputeGPA();}
        ...
};
void Student::UpdateGradeInfo(int newunits, int newgradetps)
{
    units + = newunits;
    gradepts + = newgradepts;
    ComputeGPA();
}
```

3.8 (a)

```
CardDeck::CardDeck (void)
{
    for(int i = 0;i < 52; i + + )
        cards[i] = i;
    currentCard = 0;
}
void CardDeck::Stuffle(void)
{
    static RandomNumber rnd;
    int randIndex, tmp;

    for(int i = 0;i < 52; i + + )
    {
        randIndex = i + rnd.Random(52 - i);
        tmp = cards[i];
        cards[i] = cards[randIndex];
        crads[randIndex] = tmp;
    }
    currentCard = 0;
}
```

(b) 以下给出算法的描述。在 DealHand 中声明一个含 52 个整数的局部数组。用 GetCard 将几张牌的值赋给数组。用一种排序算法，如第 2 章中所讨论的交换排序法，对 n 个整数进行排序。在数组中循环，用方法 PrintCard 打印出牌的面值。

3.9

```
Temperature Average(Temperature& a[],int n)
{
    flat avgLow = 0.0, avgHigh = 0.0;

    for (int i = 0;i < n;i + + )
    {
        avgLow + = a[i].GetLowTemp();
        avgHigh + = a[i].GetHighTemp();
    }
    avgLow / = n;
    avgHigh / = n;
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
    return Temperature(avgLow, avgHigh);
}
```

3.11 RandomNumber rnd;

(a) if (rnd.fRandom() < = 0.2) …

(b) int weight;

weight = 140 + rnd.Random(91);

3.12 (a) 注意:静态类成员在类的外部定义,但只有类成员函数才能访问它。对于静态数据成员,类的所有对象共享其值。

```
#include "random.h"
class Event
{
    private:
        int lowTime, highTime;
        static RandomNumber rnd;
    public:
        Event(int low = 0, int high = 1): lowTime
            (low),highTime(high)
        { if (lowTime > highTime)
          { cerr < < "Lower bound exceeds upper bound."
                < < endl;
            exit(1);
          }
        }
        int GetEvent(void)
        {return lowTime + rnd.Random (highTime - lowTime + 1);}
};
// rnd 为类 Event 中的静态数据成员
RnadomNumber Event::rnd;
```

(c) Event A[5] = {Event(10,20),Event(10,20),Event(10,20),Event(10,20),Event(10,20)};

// 对每个数组元素使用缺省构造函数

Event B[5]; // lowTime = 0; highTime = 1

3.14 (a) $\begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -8 \\ 0 & 0 & -4 \end{bmatrix} * \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} 5 \\ -14 \\ 0 \end{bmatrix}$

(b) Determinate = (1) (-7) (-4) = 28

第 4 章

4.2 (a) 数组 (c) 堆栈 (e) 集合 (g) 文件 (i) 堆 (k) 辞典

4.4 (b) $n^2 + 6n + 7 \leqslant n^2 + n^2 + n^2 = 3n^2, n \geqslant 6$

(d) $\frac{n^3 + n^2 - 1}{n + 1} \leqslant \frac{n^3 + n^2 - 1}{n} = n^2 + n - \frac{1}{n} \leqslant 2n^2, n \geqslant 1$

4.5 (a) n = 10

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(b) $2^n + n^3 \leqslant 2^n + 2^n = 2(2^n), n \geqslant 10$

4.7 $K \log_2 n < Kn, n \geqslant 1$，因此算法也是 O(n)量级的。

4.8 (b) O(n) (c) $O(n^2)$

4.11 (a) (3) n/2 或 O(n) (b) (1) 1

4.14 删除表中具有最大值的元素。L 必须由引用传递，才能使运行表被更新。

第 5 章

5.3 <第 1 行>22 <第 2 行>9 <第 3 行>8 <第 4 行>18

5.4
```
void StackClear(Stack& S)
{    while(! S.StackEmpty())
         S.Pop();
}
```

5.6 用中间栈 tmp，将堆栈 S1 复制到 S2 中。

5.7
```
int StackSize(Stack S)
{    int size = 0;
     // S 以值参传递，运行栈不被改变
     while(! S.StackEmpty())
     {    size++;
          S.Pop();
     }
     return size;
}
```

5.9
```
void SelectItem(Stack& S, int n)
{    stack Q;
     int i, foundn = 0;
     while( ! S.StackEmpty())
     {
          i = S.Pop();
          if (i == n)
          {    foundn++;
               break;
          }
          Q.Push(i);
     }
     while(! Q.StackEmpty())
         S.Push(Q.Pop());
     if (foundn)
            S.Push(n);
}
```

5.10 (b) ab + de -/

5.11 (b) a*(b+c)

5.14 <第 1 行>3<第 2 行>18<第 3 行>22<第 4 行>9

5.15 将队列逆置。Q 必须由引用传递以便修改运行参数。

5.18 DataType PQueue::PQDelete(void)

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{   DataType min;
    int i, minindex = 0;
    if (count > 0)
    {   min = pqlist[0]; // 假定 pqlist[0]为最小值
        // 访问剩余元素,修改最小值及其下标
        for (i = 1; i < count ; i++)
            if (pqlist[i] < min)
            // 新的最小值为 pqlist[i],其下标为 i
            {   min = pqlist[i];
                minindex = i;
            }
        // 将 pqlist[minindex+1]..pqlist[count-1]左移
        i = minindex;
        while (i < count-1)
        {   pqlist [i] = pqlist[i+1];
            i++
        }
        count--;                    // count 减 1
    }
    // qlist 为空,退出程序
    else
    {
        cerr << "Deleting from an empty priority queue!"
             << endl;
        exit(1);
    }
    return min;                     // 返回最小值
}
```

第 6 章

6.1 (a) 1 号规则。参数没有不同。函数 1 和 2 有相同的参数表,而函数 3 的缺省参数并不被看作是用于重载的。

(b) 1 号规则。参数表不同。重载正确。

(c) 2 号规则。函数 1 和 2 是可以的,因为枚举类型被看作是不同于其他类型的。但函数 3 中的类型定义都无效。编译器识别的参数表是“int& x”,与函数 1 相同。

6.3 a 和 b 最大值是 99。a,b 和 c 的最大值是 153。

1.0 + max(h1,h2) = 1.05　t,u 和 v 的最大值是 70000。

6.5 仅对 C++ 串进行交换。

```
void Swap (char *s, char *t)
// 假定 s,t 均为不超过 79 个字符的字符串
{   char tmp[80];
    strcpy(tmp,s);
    strcpy(s,t);
    strcpy(t,tmp);
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

6.7 (a)
```
ModClass::ModClass(int v): dataval(v % 7) { }
ModClass ModClass::operator + (const ModClass& x)
{   return ModClass(dataval + x.dataval);
}
```

(b)
```
ModClass operator * (const ModClass& x, const ModClass& y)
{   return ModClass(x.dataval * y.dataval);
}
```

(c)
```
ModClass Inverse(const ModClass& x)
{   ModClass prod, value;
    for(int i = 0;i < 7;i + + )
    {   value = modClass(i);
        prod = x * ModClass(i);
        if(prod.GetValue() = = 1)
            break;
    }
    return value;
}
```

6.9
```
Complex::Complex(double x, double y):real (x), imag(y) {}
Complex Complex::operator + (Complex x) const
{   return Complex(real + x.real, imag + x.imag);
}
Complex Complex::operator/ (Complex x) const
{   double denom = x.real * x.real + x.imag * x.imag;
    return Complex((real * x.real + imag * x.imag)/denom,
                   (imag * x.real - real * x.imag)/denom);
}
// 以(实数,虚数)形式输出
ostream& operator < < (ostream& ostr,const Complex& x)
{
    ostr < < '(' < < x.real < < ',' < < x.imag < < ')';
    return ostr;
}
```

6.11 (a) `return ModClass(num/den);` (b) `return Rational(dataval);`

6.13 (a)
```
Set::Set(int a[], int n)
{   for (int i = 0; i < SETSIZE; i + + )
        member[i] = FALSE;
    for(i = 0;i < n;i + + )
        member[a[i]] = TRUE;
}
```

(b)
```
int operator ^ (int n, Set x)
{   if (n < 0|| n > = SETSIZE)
    {   cerr < < "^ operator: Invalid member reference
                 " < < n < < endl;
        exit(1);
    }
    return (x.member[n]);
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(c) (i) {1,4,8,17,25,33,53,63}

(ii) {1,25}　(iii) 0　(iv) 1　(v) 1

(d) <输出> <第 1 行> {1,2,3,5,7,9,25} <第 2 行> {2,3} <第 3 行> 55 <第 4 行> 55 在 A 中

(e)
```
Set Set::operator + (Set x) const
{   int i;
    Set tmp;
    for (i = 0; i < SETSIZE; i + + )
        tmp.member[i] = member[i]|| x.member[i];
    return tmp;
}
void Set::Insert(int n)
{   if (n < 0||n > = SETSIZE)
    {   cerr < < "Insert: Invalid parameter"
            < < n < < endl;
        exit(1);
    }
    member[n] = TRUE;
}
```

第 7 章

7.1 (a)
```
template < class T >
T Max(const T &x, const T&y)
{return (x < y)? y:x;}
```

(b)
```
char * Max(char * x, char * y)
{return (strcmp(x,y) < 0) ? y:x;}
```

7.3
```
template < class T >
int Max(T Arr[], int n)
{   T currMax = Arr[0]; // 假定 n > 0
    int currMaxIndex = 0;
    for (int i = 1; i < n; i + + )
        if (currMax < Arr[i])
        {   currMax < Arr[i];
            CurrMaxIndex = i;
        }
    return currMaxIndex;
}
```

7.5
```
template < class T >
void InsertOrder (T A[], int n, T elem)
{   int i = 0; // 假定 n > 0
    while (i < n && a[i] < elem)      // 定位插入点
        i + + ;
    if (i < n)
        for (int j = n; j > i; j - - )      // 后续元素右移
            A[j] = A[j - 1];
    A[i] = elem;      // 插入该元素
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

第 8 章

8.1 (a) 0 1 2 3 4 1 2 3 4 5

(b) 否。两次调用 new 函数时，p 被赋予不同的地址。若要使 p - 10 指向第 1 个含 10 个整数的表，则 new 函数必须分配连续的内存块。

8.2 (a)
```
int *px = new int(5);
```
(b)
```
a = new long[n];
```
(c)
```
p = new DemoG;
p->one = 1; p->two = 500000; p->three = 3.14;
```
(d)
```
p = new DemoD;
p->one = 3; p->two = 35; p->three = 1.78;
strcpy(p->name,"Bob C++");
```
(e)
```
delete px; delete [] a; delete p; delete p;
```

8.3 (a)
```
DynamicInt::DynamicInt(int n)
{pn = new int (n);}
```
(b)
```
DynamicInt::DynamicInt(const DynamicInt &x)
{pn = new int(*x.pn);}
```
(c)
```
DynamicInt::operator int(void) // 返回整型值
{return *pn;}
```
(d)
```
istream& operator>> (istream& istr,DynamicInt& x)
{    istr >> * (x.pn);
     return istr;
}
```

8.4 (a)
```
p = new DynamicInt(50);
```
(b)
```
r = new DynamicInt[3];    // 每个元素值均为 0
```
(c)
```
for(int i=0; i < 10; i++)
     a[i].SetVal(100);    // 或 a[i] = DynamicInt(100);
```
(d)
```
delete p; delete [] q;
```

8.8 (a)
```
DynamicType<int> *p = new DynamicType<int>(5);
```
(b)
```
cout << * p;                // 用重载<<运算符
cout << p->GetValue();      // 使用成员函数
cout << int(p);             // 整型转换
```
(c) 开辟一个含 65 个类型为 DynamicType < char > 的对象的数组。每个数组元素的字符值为 0(NULL 字符)。

建立一个类型为 DynamicType < char > 的对象。字符值为'A'。

(d) 35 (e) 35 (f) D D 68

(g)
```
delete p; delete c;
delete Q; // 错误,Q 不是动态申请的
```

8.9 (a) 参数 x 由值传递，因此将会调用复制构造函数。复制构造函数反复调用复制构造函数，程序将进入死循环。

(b) 这样就不能将赋值运算符串在一起，如 C = B = A。

8.11 运算符'+ ='将其右侧 r 与当前 Rational 对象相加，并将结果赋给同一对象。当前对象由 * this 得到。值" * this + r"被赋给当前对象(* this)并返回。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

8.12 (a) ArrCL < int > A(20); ArrCL < char > B; ArrCL < float > C(25);

(b) ArrCL 类进行边界检查。A[30]越界。

(c) 20 个元素的数组 arr 的值为 2,4,6,8,...,40,其和为(20 * 42)/2 = 10 * 42 = 420。两个函数计算的是同一值。

8.13 (a) "Have a" (b) "nice day!" (c) "Have a nice day!"

(d) "Have a nice day!"

8.14 (a) 10 (b) y (c) 1 (d) 下标 24 越界

(e) xya52c (f) abc12ABCxya52cba

8.16 (a) 15 (b) 10 (c) 65520 (d) 1 (e) 8

8.17 (1) 匹配函数 3 (4) 匹配函数 1

8.19
```
template <class T> Set<T> Set<T>::operator~(void) const
{  Set<T> tmp(setrange);
   for (int i = 0; i < arraysize; i++)        // 形成全集
        tmp.member[i] = ~tmp.member[i];       // 对 tmp 的每个元素求反
                                              // 111... 111
return tmp - *this;    // 返回全集和当前集的差集
}
```

8.20 (a)
```
Set<T> UniversalSet(n);
UniversalSet = ~UniversalSet;
```

(b)
```
template<class T>
Set<T> Difference(const Set<T>& S,const Set<T>& T)
{return S* ~T;}
```

第 9 章

9.1 (a) 2 3 (b) 5 3 (c) 7 7 (d) 15 15 (e) 17 17

9.3
```
Node<int> *head = NULL, *p;
for (int i = 20; i > 0; i--)
{    p = new Node<int>(i,head);
     head = p;
}
p = head;
while (p != NULL)
{    cout << p->data << " ";
     p = p->NextNode();
}
```

9.6 (b) 下一个结点将指向 p。(c) 下一个结点指向自身。

9.7 (a)
```
template <class T> void InsertFront (Node<T> header, T item)
{    Node<T> *P = new Node<T>(item);
     header.InsertAfter(p);
}
```

9.9 从当前结点出发扫描表尾并将每个结点的值加 7。

9.10 删除表中头一个结点并将其放到表尾。

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

9.11
```
template <class T>
int CountKey(Node<T> *head, T key)
{    Node<T> *p = head;
     int count = 0;
     while (p != NULL)
     {    if (p->data == key)
                count++;
          p = p->NextNode();
     }
     return count;
}
```

9.14 (a) 10 8 6 4 2 (b) 2 4 6 8 10 (c) 10 8 6 4 2
(d) 10 8 6 4 2

9.15 (a) 60 70 80 90 100 (b) 20 40 60 80 100
(c) 20 10 30 40 50 60 70 80 90 100

9.17
```
void OddEven (linkedList <int> & L,LinkedList<int> & L1,
LinkedList<int> & L2)
{    L.Reset();
     while (! L.EndOfList())
     {    if (L.Data() % 2 == 1)
               L1. InsertAfter(L.Data());
          else
               L2.InsertAfter(L.Data());
          L.Next();
     }
}
```

9.19
```
template <class T> void DeleteRear(LinkedList<T> & L)
{    L.Reset();
     for (int i = 0; i < L.ListSize() - 1; i++)
          L.Next();
     L.DeleteAt();
}
```

9.23 逆置链表中的各项，方法是先将它们复制到一个中间栈表中，然后再将它们复制回链表中。

9.27 每个队列中都有一个由复合得到的 LinkedList 对象。当队列对象彼此赋值时，调用 LinkedList 类中的重载的赋值运算符，将两个表互相复制到对方。

9.29
```
template <class T> void InsertOrder(CNode<T> *header,
                                    CNode<T> *newNode)
{    CNode<T> *curr = header->NextNode(), *prev = header;
     while (curr != header && curr->data < newNode->data)
     {    prev = curr;
          curr = curr->NextNode();
     }
     prev->InsertAfter(newNode);
}
```

9.32
```
template <classT> DNode<T> *DNode<T>::DeleteNodeRight(void)
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
{    DNode<T> *tempPtr = next; // 保存结点地址
     if (next == this)
         return NULL;        // 指向自身,退出
     // 当前结点指向 temp.Ptr 的后继
     right = tempPtr->right;
     // tempPtr 的后继指向当前结点
     tempPtr->right->left = this;
}
```

第 10 章

10.1 结果取决于编译器对操作数求值的顺序。若 n = 3 且先求左操作数的值,则结果为 3 * 2! = 6。如果先求右操作数的值则结果为 2 * 2! = 4。

10.2 1 1 5 13 41 121 365 1093 3281 9841…

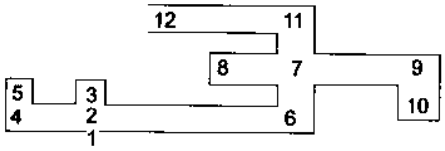
10.4 125

10.5 ! gnitseretni si sihT

10.7
```
float avg(float a[], int n)
{    if (n == 1)
         return a[0];
     else
         return float(n-1)/n*avg(a,n-1) + a[n-1]/n;
}
```

10.8
```
int rstrlen(char *s)
{    if (*s == 0)
         return 0;
     else
         return 1+rstrlen(s+1);
     }
```

10.12 走法:1 2 6 7 11 12



第 11 章

11.2 (a) 3 (b) 2

11.6 Yes

11.7 (a) 15 (b) 42

(c) 前序:50 45 35 15 5 40 38 36 42 43 46 65 75 70 85

11.8 将结点插入到一棵二叉树中。指针由引用传递使得根和 TreeNode 指针域可以被更改。

11.9 (a) 前序:M F T N V U

(b) 后序:A D I R O L F

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

11.10 (a) RNL: V U T N M F (b) RLN: R O I L A D F

(c) 逐层: R O T A R Y C U B L

11.14 (a) A C F E I H B D G

11.20
```
template <class T>
void PostOrder_Right (TreeNode<T> *t,void visit(T& item))
{    if (t ! =NULL)
     {
     // 递归扫描过程在遇到空子树时结束
     postOrder_Right(t->Right(),visit); // 扫描右子树
     postOrder_Right(t->Left(), visit); // 扫描左子树
         visit(t->data);                // 访问结点
     }
}
```

11.22
```
template <class T> TreeNode<T> *Max(TreeNode<T> *t)
{    while (t->Right() ! = NULL)
         t = t->right();
     return t;
}
```

11.26
```
template <class T> int NodeLevel (TreeNode<T> *t, const T&elem)
{    int level = -1;
     while (t ! = NULL)
     {
          level++;
          if (t->data == elem)
               break;
          else if (elem < t->data)
               t = t->Left();
          else
               t = t->Right();
     }
     if (t == NULL)
          level = -1;
     return level;
}
```

第 12 章

12.2

	Base_Priv	Base_Prot	Base_Pub	Derived_Priv	Derived_Prot	Derived_Pub
BASE	×	×	×			
DERIVED		×	×	×	×	×
CLIENT						×

12.3 (a) 派生类构造函数不调用基类构造函数。

12.4 (a) DerivedCL::DerivedCL(int a, int b, int c): data3(a),BaseCL(b,c){}

(b) DerivedCL::DerivedCL(int a): data3(a), BaseCL(){}

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

(c)

	data1	data2	data3
obj1	2	0	1
obj2	4	5	3
obj3	0	0	8

12.5 <第 1 行> Base1 constructor called. <第 2 行> Base2 constructor called.
<第 3 行> Derived class constructor called. <第 4 行> Base1 constructor called.
<第 5 行> Base2 constructor called. <第 6 行> Base2 destructor called.
<第 7 行> Base1 destructor called. <第 8 行> Derived class destructor called.
<第 9 行> Base2 destructor called. <第 10 行> Base1 destructor called.

12.7 (a) GetX 是 Shape 类的公有方法。
(b) x 在基类 Shape 中是保护的,因此可以由派生类引用。但它不能被客户程序引用。

12.8 <第 1 行> 1 <第 2 行> Base Class <第 3 行> 2 <第 4 行> 1
<第 5 行> 0 <第 6 行> Base

12.11 <第 1 行> 7 <第 2 行> 2 <第 3 行> 3 5 <第 4 行> 2 4 <第 5 行> 2 4
<第 6 行> 0 1 <第 7 行> 7 <第 8 行> 2 3

12.13
```
template <class T> class StackBase
{  protected:
       int numElements;
   public:
       StackBase(void): numElements(0) {}
       virtual void Push(const T& item) = 0;
       virtual T Pop(void) = 0;
       virtual T Peek(void) = 0;
       virtual int StackEmpty(void){return numElements = = 0}
};
```
用数组(第 5 章)或链表(第 9 章)实现派生类 Stack。

12.19
```
int LookForMatch(Array < int > & A, int end, int elem)
{  ArrayIterator < int > aiter(A,0,end - 1);
   while(! aiter.EndOfList())
   {    if (aiter.Data() = = elem)  return 1;
        aiter.Next();
   }
   return 0;
}
void RemoveDuplicates(Array < int > & A)
{    ArrayIterator < int > assign(A), march(A);
     int assignIndex;
     if (A.ListSize() < = 1) return;
     assign.Next();
     march.Next();
     assignIndex = 1;
     while(! march.EndOfList())
     {  if (! LookForMatch(A,assignIndex,march.Data()))
        {    assign.Data() = march.Data();
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
            assign.Next();
            assignIndex++;
        }
        march.Next();
    }
    A.Resize(assignIndex);
```

12.20
```
template <class T> int Max(Iterator<T>& collIter, T& maxval)
// 移到迭代算子的第1个元素
{ collIter.Reset();
  // 若已到表尾,则表为空
  if (collIter.EndOfList())
      return 0;
  // 记录第1个表元素值,并开始比较
  maxval = vollIter.Data();
  for(collIter.Next();! collIter.EndOfList();collIter.Next())
      if (collIter.Data() > maxval)
          maxval = collIter.Data();
  return 1;            // 成功返回
  }
```

第 13 章

13.2 树(B):60 30 80 65 40 5 50 10 90 15 70

13.3
```
template <class T> void Preorder (T A[], int currindex, int n, void visit(T&
                                    item))
{ if (currindex < n)
  {    visit(A[currindex]);                    // 访问结点
       Preorder(A,2*currindex+1, n, visit);  // 访问左子树
       Preorder(A,2*currindex+2, n, visit);  // 访问右子树
  }
}
```

13.4 (a) 是 (c) A[24] (e) 是 A[34]

13.6 最后一层有 2^n 个元素。非叶子结点数为 $1+2+4+\ldots+2^{n-1}=2^{n-1}$

13.7 (a) 5 (d) 41 和 42 (f) 15－30

13.10 (b) 是满树、完全树、最小堆。(d) 是完全树、最大堆。

13.13

原始堆数组 (A)：	5	10	20	25	50			
插入 15：	5	10	15	25	50	20		
插入 35：	10	25	15	35	50	20	35	
删除 5：	10	25	15	35	50	20		
插入 40：	10	25	15	35	50	20	40	
插入 10：	10	10	15	25	50	20	40	35

13.15 (a) 47 45 40 10 (c) 35 40 45

13.16 (b) 3,6,33,88,16,45,45,90 (c) "aehpify"

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

13.21 对于图(B);

矩阵

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	1	0
C	0	0	0	0	0
D	0	0	0	0	0
E	1	0	1	1	0

邻接表

A: C D
B: C D
C:
D:
E: A C D

13.22 (b) 无路径 (d) 从 A 出发,有路径到 C 和 D。

13.23 (b) 深度优先: A E D C 广度优先: A C D E

13.26 用广度优先搜索法打印图中每个顶点。

第 14 章

14.1 (a) Pass 0: 1 4 8 9 2 1 7 4 Pass 1: 1 1 8 9 2 4 7 4
Pass 2: 1 1 2 9 8 4 7 4 Pass 3: 1 1 2 4 8 9 7 4
Pass 4: 1 1 2 4 4 9 7 8 Pass 5: 1 1 2 4 4 7 9 8
Pass 6: 1 1 2 4 4 7 8 9

14.2 Pass 0: 4 8 1 9 2 1 7 4 Pass 1: 1 4 8 9 2 1 7 4
Pass 2: 1 4 8 9 2 1 7 4 Pass 3: 1 2 4 8 9 1 7 4
Pass 4: 1 1 2 4 8 9 7 4 Pass 5: 1 1 2 4 7 8 9 4
Pass 6: 1 1 2 4 4 7 8 9

14.7 当表已经按升序或降序排列时,算法复杂度为 $O(n^2)$。

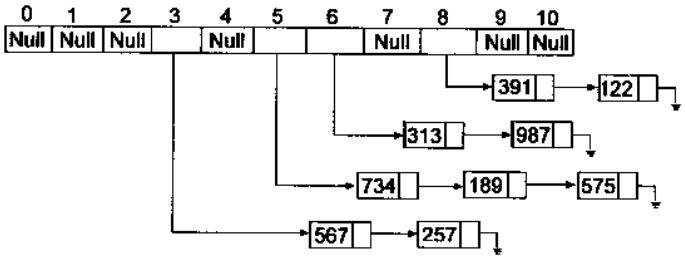
14.13 若 $r = x\%m$,q 是 x/m 的商,则必有 $x = mq + r$,因此 $r = x - mq$。既然 m 为偶数,则只要 x 为偶数,哈希函数的值即为偶数。所有的偶数键值都哈希到偶数表索引处。哈希值在表中的扩散不够充分。

14.14

(a)

	0	1	2	3	4	5	6	7	8	9	10
HT	391	Empty	Empty	257	567	575	987	189	122	734	313

(b)



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

14.19 (b) hashf(16) = 1 \ hashf(257) = 16

14.21 若哈希函数是良好的,则哈希到位置 i 的概率为 1/N。一旦位置 i 已有数据。则哈希到位置 i + 1 的概率为 2/N;填充到 i + 2 的概率为 3/N,由于填充到一组连续位置的某一位置的概率超过填充到表中其它位置的概率,则发生"集束"。

14.22 (a) 若哈希到下标 i,我们可知再哈希函数最后还将回到 i。为此,对某个 k,有 i = (i + 3k)%93,即 i + 3k = 93% + i,也即 3k = 99%。此方程的最小解为 k = 31,q = 1,经过 31 次迭代后,再哈希函数回到 i,即只覆盖了 1/3 个表。

(b) 若 d 小于 m,则 d 和 m 互素且再哈希函数覆盖表。

14.24 (a)

14.25 (a) 是

(b)
```
int H(char * s)
{return (s[2] - 'a') % 7;}
```

14.26
```
void PascalTextFile::Assign(char * filename)
{strcpy(fname, filename);}

void PascalTextFile::Reset(void)
{  if (isOpen)
         f.seekg(0,ios::beg);
   else
   {      accesstype = IN;
          f.open(fname, ios::in | ios::nocreate);
          if (f! = NULL)
                isOpen + + ;
   else
                Error("Pascal file cannot be open");
   }
}
void PascalTextFile::PWrite(char A[], int n)
{  if (accesstype = = IN)
         Error("Invalid file access operation");
   if(! isOpen)
         Error("file closed");
   f.write(A,n);
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。