新世纪计算机专业系列教材

数据结构——C++实现

缪淮扣　顾训穰　沈　俊　编著

科学出版社

2002

内 容 简 介

数据结构是计算机专业教学计划中的一门核心课程,也是信息管理、通信电子、自动控制等与计算机技术关系密切的专业的一门基础课程。要从事和计算机科学与技术相关的工作,尤其是计算机应用领域的开发和研制工作,必须具备坚实的数据结构的基础。本书对 C++ 语言作了简单介绍,叙述了抽象数据类型和面向对象的概念,介绍了线性表、栈、队列、数组、广义表、树和图等数据结构,并且介绍了查找和排序的方法。全书用 C++ 语言描述并实现了所有数据结构的类和程序,并附有习题,便于教学。

本书是为高等院校开设"数据结构"课程编写的教材,可作为计算机专业本科生教材使用,也可供从事计算机软件开发和应用的工程技术人员阅读、参考。

图书在版编目(CIP)数据

数据结构:C++ 实现/缪淮扣,顾训穰,沈俊编著.—北京:科学出版社,2002
(新世纪计算机专业系列教材)
ISBN 7-03-010457-9

Ⅰ.数… Ⅱ.①缪… ②顾… ③沈… Ⅲ.①数据结构—高等学校—教材 ②C 语言—程序设计—高等学校—教材 Ⅳ.TP311.12

中国版本图书馆 CIP 数据核字(2002)第 033707 号

科学出版社 出版
北京东黄城根北街 16 号
邮政编码:100717

印刷

科学出版社发行 各地新华书店经销

*

2002 年 7 月第 一 版 开本:787× 1092 1/16
2002 年 7 月第一次印刷 印张:21 1/4
印数:1- 5 000 字数:483 000

定价:29.00 元

(如有印装质量问题,我社负责调换〈科印〉)

前　言

作为计算机程序组成部分的数据结构和算法的研究,一直受到计算机领域工作者的高度重视。数据结构是计算机专业教学计划中的一门核心课程,也是信息管理、通信电子、自动控制等与计算机技术关系密切的专业的一门基础课程。

要从事与计算机科学与技术相关的工作,尤其是计算机应用领域的开发和研制工作,必须具备坚实的数据结构的基础。

数据结构课程的教学目的是使学生学会分析研究计算机所要加工处理的数据的特征,掌握组织数据、存储数据和处理数据的基本方法,并加强在实际应用中选择合适的数据结构和相应算法的训练。

面向对象技术是软件工程领域中的重要技术,它不仅是一种程序设计方法,更重要的是一种对真实世界的抽象思维方式。目前,面向对象的软件分析和设计技术已发展成为软件开发的主流方法。为了适应软件开发方法与技术的发展以及应用领域的要求,就有必要改进和充实数据结构的教学内容。因此,用面向对象的观点来描述数据结构就成为一种既顺理成章又紧迫的选择。

用面向对象的观点来描述数据结构,要涉及到面向对象程序设计语言的选用问题。目前被广泛采用作为程序设计语言教学的是 C 语言,C++ 是以 C 语言为基础、使用比较普遍的面向对象的程序设计语言。因此本书采用 C++ 作为数据结构的描述语言。

数据结构课程内容丰富,学习量大;隐藏在各部分内容中的方法和技术多,贯穿于全书的动态链表存储结构和递归技术令不少初学者望而生畏。本书的编写者长期以来从事数据结构课程的教学,对该课程的教学特点和难点有比较深切的体会。作者在认真总结二十多年讲授“数据结构”课程的基础上,参考了美国 ACM/IEEE-CS 公布的“计算 2001 教程”,吸收了国内外各种数据结构教材的优点,对多年来形成的数据结构课程的教学内容进行了合理的剪裁,既强调了数据结构的原理和方法,又注重了其实践性,使之适应于现代大学生的学习特点和要求。

本书的一个重要特点就是将程序设计的基础与数据结构的方法尽可能地结合起来。第 1、2 章介绍 C++ 语言时尽可能给出比较完整的程序,使学生能对 C++ 语言有比较全面和深入的了解,也便于上机实习,从而为数据结构课程的实验建立良好的基础。

全书共分 9 章,第 1、2 章介绍了数据结构、算法及其复杂度的基本概念,对 C++ 作了简单介绍,并叙述了抽象数据类型和面向对象的概念。第 3~ 5 章介绍了线性结构——线性表、栈、队列、数组、广义表;第 6 章和第 7 章介绍了非线性结构——树和图;第 8 章和第 9 章分别介绍了查找和排序的方法。

本书第 1~ 5 章由缪淮扣编写,第 6~ 9 章由沈俊和顾训穰共同编写。占学德和刘玲调

试了部分算法。全书由缪淮扣和顾训穰统稿。

在本书的写作过程中,本丛书的编委会、科学出版社、上海大学教务处和计算机学院给予了很大支持。在此致以诚挚感谢。

由于时间仓促和作者水平有限,本书难免存在疏漏和缺点,敬请广大读者批评指正。

缪淮扣

2002 年 3 月于上海

目　　录

1　绪论 …… 1
1.1　(算法+ 数据结构)= 程序 …… 1
1.2　数据结构的基本概念 …… 2
1.2.1　两个简单的数据结构实例 …… 2
1.2.2　什么是数据结构 …… 3
1.3　C+ + 语言基础 …… 4
1.3.1　程序结构 …… 5
1.3.2　数据声明和作用域 …… 6
1.3.3　输入/输出 …… 7
1.3.4　函数 …… 9
1.3.5　参数传递 …… 10
1.3.6　函数名重载 …… 11
1.3.7　动态内存分配 …… 11
1.3.8　结构与联合 …… 12
1.4　算法性能与复杂度 …… 16
1.4.1　算法的定义 …… 16
1.4.2　算法的性能标准 …… 17
1.4.3　算法的复杂度 …… 17
习题 1 …… 21
2　抽象数据类型和 C+ + 类 …… 23
2.1　抽象数据类型 …… 23
2.1.1　从数据类型到抽象数据类型 …… 23
2.1.2　封装和信息隐藏 …… 24
2.1.3　抽象数据类型描述 …… 25
2.2　类与对象的基本概念 …… 26
2.2.1　类与对象 …… 26
2.2.2　消息与合作 …… 28
2.2.3　多态性 …… 28
2.3　面向对象的程序设计方法 …… 28
2.4　C+ + 类与对象 …… 29
2.5　构造函数和析构函数 …… 31

2.6 工具函数 …… 35
2.7 继承 …… 38
2.8 this 指针的使用 …… 41
2.9 虚函数、多态性以及动态联编 …… 42
2.9.1 虚函数和多态性 …… 42
2.9.2 动态联编 …… 50
2.10 模板类 …… 52
习题 2 …… 54
3 线性表 …… 56
3.1 线性表的定义 …… 56
3.2 线性表的顺序表示 …… 57
3.2.1 顺序表的类定义 …… 57
3.2.2 顺序表插入、删除算法的复杂度分析 …… 60
3.2.3 顺序表的应用 …… 61
3.3 线性表的链表表示 …… 62
3.3.1 单链表 …… 62
3.3.2 单循环链表 …… 73
3.3.3 双向循环链表 …… 73
3.3.4 静态链表 …… 79
3.4 多项式抽象数据类型 …… 81
3.4.1 多项式表示 …… 81
3.4.2 多项式相加 …… 81
习题 3 …… 83
4 栈、队列和递归 …… 85
4.1 栈 …… 85
4.1.1 顺序栈 …… 86
4.1.2 链式栈 …… 88
4.1.3 表达式的计算 …… 90
4.2 队列 …… 94
4.2.1 循环队列 …… 95
4.2.2 链队列 …… 98
4.3 递归 …… 100
4.3.1 递归的概念 …… 100
4.3.2 递归过程与递归工作栈 …… 101
4.3.3 消除递归 …… 103
4.3.4 迷宫问题 …… 107
习题 4 …… 109

5 串、数组和广义表 …… 112
5.1 字符串 …… 112
5.1.1 字符串的定义、存储结构和操作 …… 112
5.1.2 串的操作 …… 113
5.1.3 常用的 C++ 字符串函数 …… 114
5.1.4 串类及其实现 …… 115
5.1.5 模式匹配算法 …… 121
5.2 数组 …… 125
5.2.1 C++ 中数组的定义 …… 126
5.2.2 数组的抽象数据类型表示 …… 126
5.2.3 数组的顺序存储结构 …… 128
5.3 稀疏矩阵 …… 130
5.3.1 三元组顺序表 …… 131
5.3.2 十字链表 …… 133
5.4 广义表 …… 135
5.4.1 广义表的定义 …… 135
5.4.2 广义表的存储结构 …… 136
5.4.3 n 元多项式的表示 …… 139
5.4.4 广义表的递归算法 …… 141
习题 5 …… 144
6 树和森林 …… 147
6.1 树的概念 …… 147
6.1.1 树的定义 …… 148
6.1.2 树的术语 …… 148
6.1.3 树的表示形式 …… 149
6.1.4 树的基本操作和抽象数据类型 …… 149
6.2 二叉树 …… 153
6.2.1 二叉树的定义 …… 153
6.2.2 二叉树的性质 …… 153
6.2.3 二叉树的基本操作和抽象数据类型 …… 155
6.3 二叉树的存储结构 …… 158
6.3.1 数组表示法 …… 159
6.3.2 链表表示法 …… 160
6.3.3 二叉树的二叉链表类声明 …… 160
6.4 遍历二叉树 …… 164
6.4.1 前序遍历 …… 164
6.4.2 中序遍历 …… 165

6.4.3 后序遍历 …… 165
6.4.4 层序遍历 …… 166
6.5 线索二叉树 …… 168
6.5.1 线索二叉树的定义 …… 168
6.5.2 线索二叉树的类定义 …… 170
6.5.3 中序线索二叉树 …… 173
6.6 二叉树的应用 …… 177
6.6.1 堆 …… 177
6.6.2 哈夫曼树 …… 183
6.7 树和森林 …… 187
6.7.1 树的存储结构 …… 187
6.7.2 树、森林和二叉树的转换 …… 190
6.7.3 树的遍历 …… 192
6.7.4 森林的遍历 …… 193
6.8 等价类及其表示 …… 194
6.8.1 等价关系与等价类 …… 194
6.8.2 并查集 …… 195
习题6 …… 199
7 图 …… 203
7.1 图的基本概念 …… 203
7.1.1 图的定义 …… 203
7.1.2 图的术语 …… 204
7.1.3 图的基本操作和抽象数据类型 …… 207
7.2 图的存储结构 …… 209
7.2.1 邻接矩阵 …… 209
7.2.2 邻接表 …… 212
7.2.3 邻接多重表 …… 217
7.2.4 十字链表 …… 219
7.3 图的遍历与连通性 …… 220
7.3.1 深度优先遍历 …… 220
7.3.2 广度优先遍历 …… 221
7.3.3 连通分量 …… 223
7.4 最小生成树 …… 224
7.4.1 克鲁斯卡尔算法 …… 225
7.4.2 普里姆算法 …… 228
7.5 最短路径 …… 230
7.5.1 弧上权值为非负情形的单源点最短路径问题 …… 231

7.5.2 弧上权值为任意值的单源点最短路径问题 …… 234
7.5.3 所有顶点之间的最短路径 …… 236
7.6 活动网络 …… 238
7.6.1 用顶点表示活动的网络 …… 238
7.6.2 用边表示活动的网络(AOE 网络) …… 242
习题 7 …… 246
8 查找 …… 250
8.1 基本概念 …… 250
8.2 顺序表 …… 251
8.2.1 顺序表的查找 …… 251
8.2.2 有序表的折半查找 …… 252
8.3 索引顺序表 …… 256
8.3.1 索引顺序表 …… 256
8.3.2 倒排表 …… 258
8.4 二叉排序树 …… 260
8.4.1 二叉排序树定义 …… 260
8.4.2 二叉排序树上的查找 …… 262
8.4.3 二叉排序树的插入 …… 263
8.4.4 二叉排序树的删除 …… 265
8.4.5 二叉排序树查找的性能分析 …… 266
8.5 平衡二叉树 …… 266
8.5.1 平衡二叉树的定义 …… 267
8.5.2 平衡旋转 …… 267
8.5.3 平衡二叉树的插入和删除 …… 269
8.6 B- 树 …… 273
8.6.1 动态的 m 路查找树 …… 273
8.6.2 B- 树 …… 274
8.6.3 B- 树的插入 …… 274
8.6.4 B- 树的删除 …… 276
8.6.5 B+ 树 …… 278
8.7 散列表查找 …… 279
8.7.1 散列表的基本概念 …… 279
8.7.2 散列函数 …… 281
8.7.3 处理溢出的闭散列方法 …… 282
8.7.4 处理溢出的开散列方法——链地址法 …… 286
8.7.5 散列表分析 …… 287
习题 8 …… 288

9 **排序** …… 292
9.1 基础知识 …… 292
9.1.1 基本概念 …… 292
9.1.2 排序表的抽象数据类型描述和类定义 …… 293
9.2 交换排序 …… 299
9.2.1 冒泡排序 …… 299
9.2.2 快速排序 …… 300
9.3 插入排序 …… 302
9.3.1 直接插入排序 …… 302
9.3.2 折半插入排序 …… 306
9.3.3 希尔排序 …… 306
9.4 选择排序 …… 308
9.4.1 直接选择排序 …… 308
9.4.2 锦标赛排序 …… 310
9.4.3 堆排序 …… 312
9.5 归并排序 …… 314
9.5.1 归并 …… 314
9.5.2 两路归并排序 …… 315
9.5.3 递归的归并排序 …… 317
9.6 基数排序 …… 319
9.6.1 多关键字排序 …… 319
9.6.2 链式基数排序 …… 320
9.7 各种排序方法的选择和使用 …… 322
习题 9 …… 323
主要参考文献 …… 326

绪　论

1.1　(算法+ 数据结构)= 程序

计算机能快速计算,能证明“四色定理”,能帮助工程师设计图纸,能帮助医生为病人诊断配药……它之所以如此神通广大、聪明能干,是因为人们教给了它本领。人是用“程序”来“教”计算机的。要想让计算机具备解题的能力,就要为计算机编写程序。随着计算机科学与技术的不断发展,计算机的应用领域也在不断扩大。“程序”越来越庞大、越来越复杂,因而解题的过程就不仅仅是编写程序了,而是一个包括编程序在内的软件开发过程,该过程包括对用户的需求进行分析,对要开发的系统进行软件设计,然后编程序,再进行测试和改错,最后再运行等一系列的工作。“软件”不单纯是指程序,而是指程序以及开发程序的过程中所产生的各种文档。软件开发的目的是产生能让计算机有效工作的程序,因此程序是软件的核心。

程序到底是什么呢?计算机科学家,图灵奖获得者 N.Wirth 给出过一个著名的公式:算法+ 数据结构= 程序。从这个公式可以看到,数据结构和算法是构成程序的两个同样重要的组成部分。这个公式在软件开发的进程中产生了深远的影响。但是它并没有强调数据结构与解题的算法是一个整体。该公式现在已经受到了挑战。

我们知道,非面向对象的过程语言,如 FORTRAN、C 和 PASCAL,其数据结构是问题解的中心。一个软件系统的结构是围绕一个或几个关键数据结构为核心而组成的。“算法+ 数据结构= 程序”完全体现了这种思想。但随着软件系统的规模越来越大、复杂性不断增长,人们不得不对“关键数据结构”重新评价。在这种系统中,许多过程和函数(子程序)的实现严格依赖于关键数据结构,如果这些关键数据结构的一个或几个数据有所改变,则涉及到整个软件系统,许多过程和函数必须重写,甚至因为几个关键数据结构的改变,导致软件系统整个结构彻底崩溃。

20 世纪 90 年代,面向对象的方法得到了很大的重视,并得到了比较广泛的推广和应用。在面向对象程序设计中,密切相关的数据与过程被定义为一个整体(即对象),而且一旦作为一个整体定义了之后,就可以使用它,而无需了解其内部的实现细节,从而提高软件开发的效率。

封装和数据隐藏是面向对象问题解和面向对象程序设计的基本要素。它可以使软件的许多维护问题简单化。一旦数据(类型)结构修改了,只要对实现模型的局部代码作适当修改,软件系统的其余部分无需修改。

有人主张将 Wirth 的公式“算法+ 数据结构= 程序”修改为“(算法+ 数据结构)= 程序”,以体现面向对象的方法。

本书以面向对象的观点来介绍各种数据结构以及与这些数据结构有关的算法的知

识。

本章将介绍数据结构以及算法的基本概念,并介绍用来描述数据结构和算法的语言C++。

1.2 数据结构的基本概念

计算机科学是一门研究信息表示和处理的科学,人们是用程序来处理信息的。信息的表示和组织直接关系到处理信息的程序的效率。由于许多系统软件和应用软件的程序规模很大,结构又相当复杂,因而必须对程序设计方法进行系统的研究。这不仅涉及到研究程序结构和算法,同时也涉及到研究程序加工的对象。

用计算机解题,首先应从具体问题抽象出一个适当的数学模型,然后才能设计算法和编制程序。而构建数学模型的过程就是分析和概要设计的过程,要从对问题的分析中提取操作的对象,并找出这些操作对象之间的关系,然后用数学的语言加以描述。例如,许多工程中的数值计算问题采用的数学模型是线性方程组或微分方程。但更多的非数值计算问题是难以用数学方程来描述的。

1.2.1 两个简单的数据结构实例

为了获得对数据结构的感性认识,我们先来看两个简单的例子。

【例 1-1】 人事登记表。

在任何一个单位,人事登记表是人事部门关于职工信息的必不可少的表格。例如,某个单位人事部门的工作人员想要查找当年退休的人员,或要查找基本工资在800元以下的人员,如果该单位有几千名职工,那么由人工来查找显然是很费时的;但如果在计算机中有一个关于职工信息的登记表,由有关的检索系统来查找,那是很方便的。表1-1所示就是一张简单的人事登记表。

表 1-1 人事登记表

编号	姓名	性别	出生日期	婚否	基本工资
0001	王 军	男	1960/5/30	已	650
0002	李 平	女	1953/6/2	已	710
0003	周丽娟	女	1948/7/8	已	980
0004	赵忠良	男	1950/12/2	已	950
0005	张国庆	男	1978/10/1	未	500
⋮	⋮	⋮	⋮	⋮	⋮

表1-1中,每个职工的信息放在一行中,我们称之为一个数据元素,所有职工的信息按照某一种顺序依次存放在表中。一般称这种由记录组成的线性表为文件。

类似这样表格在不同的计算机软件系统中是很多的,如学校的学生信息管理系统中的学生学籍登记表、图书馆管理系统中的图书目录表等。这类表格有一个共同的特性,就是各数据元素之间的关系是线性的关系,即一个元素的前面只有一个元素,后面也只有一个元素,第一个数据元素前面和最后一个数据元素后面没有元素。这样的一类表格就是一

种数据结构,称为线性数据结构。

【例 1-2】 一个典型的学校行政机构如图 1-1 所示,这是一个层次结构。顶层结点“学校”代表整个系统,它的下一层结点代表这个系统的各个子系统,即部处与学院。再下一层结点代表更小的机构,如教务科、计算机系等,直到最底层一个小组或一个教研室等。在该图中,每一个结点代表一个数据元素,每一个结点的下面可能有多个结点。这样的一种结构称为层次型数据结构。

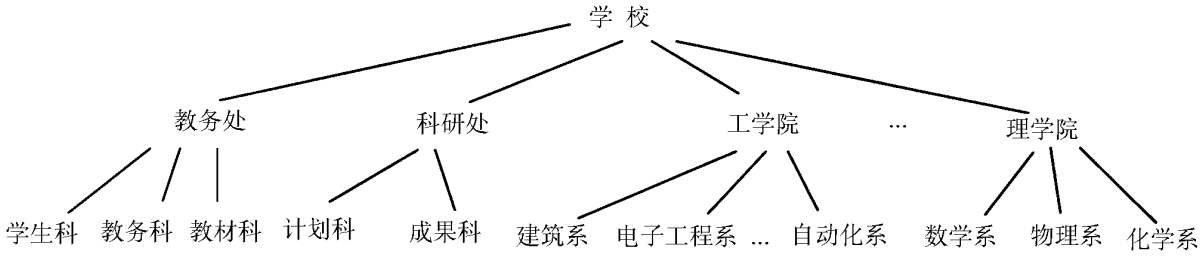

图 1-1 学校机构的“树结构”

在各种应用程序中会涉及到各种各样的数据,为了组织它们、存储它们,并且能对它们进行操作,就要考虑它们的归类以及它们之间的关系,从而建立相应的数据结构,并依此实现所要求的功能。

1.2.2 什么是数据结构

一个水平再高的厨师,如果不给他原料,他也无法做出色、香、味俱全的菜。这也就是人们常说的“巧妇难为无米之炊”。对一个程序来讲,数据就是“原料”。一个程序所要进行的计算或处理总是以某些数据为对象的。

大千世界中有各种各样的信息,如马路上的交通灯、进出地铁站的交通卡、股市投资者与证券商之间的交易、人们用语言交流的思想等。这些信息必须转换成数据才能在计算机中进行处理。让我们先来定义什么是数据以及与之相关的概念,然后再来回答“什么是数据结构”这个问题。

数据(data)是信息的载体,是描述客观事物的数、字符、图形、图像、声音以及所有能输入到计算机中并被计算机程序识别和处理的符号的集合。

例如,一个解代数方程的程序处理的对象只是整数和实数,一个 C++ 语言的编译程序处理的对象是由 ASCII 码字符组成的字符串等。

数据的基本单位是数据元素(data element)。一个数据元素可由若干个数据项(data item)组成,数据项是数据的最小单位。

为了解题的需要,具有相同特性的数据元素经常归类,被称之为数据对象的集合。数据对象是数据的子集。例如,所有的“数”构成了“数据”集合,而自然数集合 N = {0, 1, 2, …}是“数”的数据对象;所有的字符是数据,字母集合 AS= {A,B,…,Z,a,b,…,z}是该数据的数据对象。

在大多数情况下,一个数据元素不只是含有一个数据项,而是由多个数据项组成的。如在例 1-1 中,一个职工的记录就是一个数据元素,它包括编号、姓名、性别、出生日期、婚否和基本工资六个数据项。

数据结构(data structure)的概念与数据的概念不同。若想描述一个数据结构,不仅要描述数据,而且还要描述各数据元素之间的相互关系。

从以上两个例子可以看到,在实际应用中的各种数据元素都不是孤立存在的,它们之间存在着某种关系。这种数据元素之间的关系就是结构。根据数据元素之间关系的不同,数据结构分为两大类:线性结构和非线性结构。线性结构中各个数据元素依次排列成一个线性序列;而非线性结构中各个数据元素不再保持成为一个线性序列,每个数据元素可能与其他多个数据元素发生关系。这两类结构通常又可分为下列四类基本结构:①集合,结构中的数据元素之间就是"同属于一个集合",此外,没有其他关系;②线性结构,结构中的数据元素之间存在的是一种线性关系,即一对一的关系,前面的例 1-1 给出的就是线性结构;③树形结构,结构中的元素存在着一对多的关系,前面的例 1-2 给出的就是树形结构;④图形结构或网状结构,结构中的元素之间存在着多对多的关系。图 1-2 描述了这四种不同结构的关系图。其中(b)描述的是线性结构,而(a)、(c)和(d)属于非线性结构。

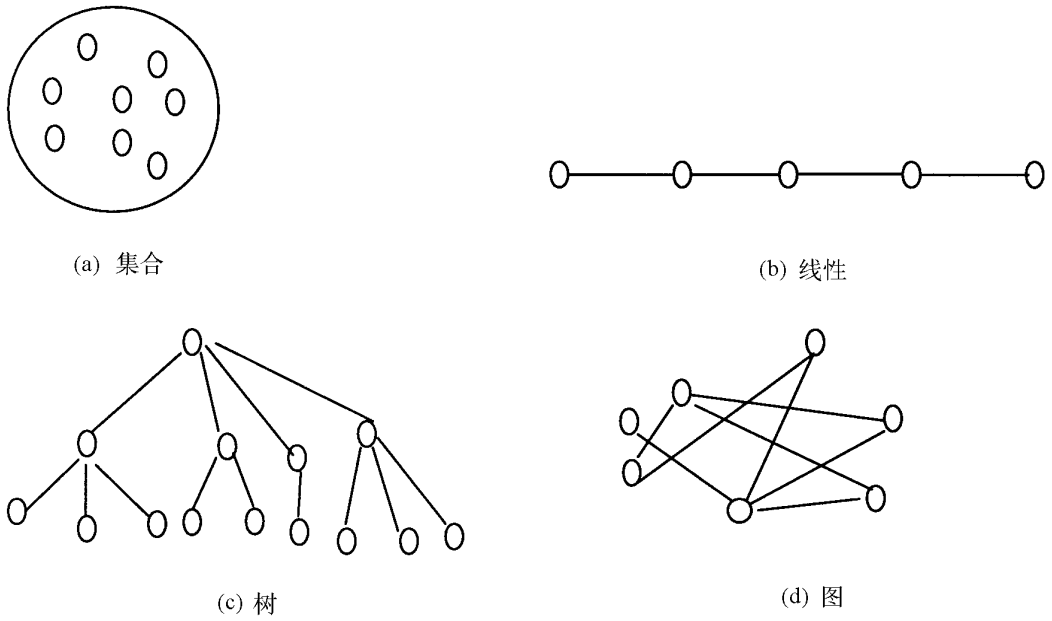

图 1-2　四种基本结构的关系图

对数据结构,又可按照其视点的不同分为数据的逻辑结构和物理结构。

数据的逻辑结构属于用户视图,是用户所看到的数据结构,是面向问题的。它描述的是数据元素之间的逻辑关系。数据的物理结构,又称为存储结构,是数据的逻辑结构在计算机中的物理存储方式,它属于具体实现的视图,是面向计算机的。

数据的逻辑结构和物理结构是密切相关的两个方面。一般来说,算法设计是基于数据的逻辑结构,而算法实现则基于数据的物理结构。

本书在讨论各种数据结构时,既讨论其逻辑结构,又讨论其物理结构。

1.3　C++ 语言基础

本书以面向对象的观点来介绍数据结构,因而所涉及的程序设计的方法自然是面向对象的程序设计方法。用面向对象的观点来描述数据结构所采用的语言应该是面向对象

的程序设计语言。现有的面向对象的程序设计语言有 Smalltalk、Effel、C++ 和 Java 等。出于实用和易学的考虑,本书选择了目前比较流行的 C++ 语言来描述各种数据结构以及相应的算法。

C++ 与 C 具有许多相同的功能,但除了数据抽象和继承以外,C++ 对 C 有很多扩充的功能。我们认为本书读者已经熟悉 C 语言。

在本节中先对 C++ 语言作简单的介绍,有关抽象数据类型和面向对象的程序设计的一些概念将在第 2 章中介绍。

1.3.1 程序结构

一个 C++ 程序可由若干个文件组成。C++ 的文件分为头文件和源文件两类。

头文件以.h 为后缀,用于存放函数声明,它给出了函数的参数类型、个数以及函数的返回类型,称为原型。有一些头文件是系统定义的,如〈iostream.h〉,而另一些头文件是用户定义的;而源文件是用来存放 C++ 的源代码的。用于源文件的后缀为.cpp。可通过预处理指令# include 将头文件包含在适当的文件中。

现在通过一个典型的 C++ 程序来看看 C++ 的程序结构。该程序输出字符串“Hello, Shanghai”,它由如下三个文件组成:

```
/ 头文件 hello.h /
#  ifndefFILENAME H
#  defineFILENAME H
char  hello( char );
#  endif
/ 源代码文件 hello.cpp /
#  include< stdio.h > //含有 sprint( )的原型
#  include< string.h>  //含有求字符串长度函数 strlen( )的原型
#  include< hello.h>  //含有 hello( )的原型
char   hello(char  world){
char   result= new char[9+ strlen(world)];
/  Return the string "Hello, world". /
sprintf(result, "Hello,%s. ", world);
returnresult;
}

/ 源代码文件 main.cpp /
#  include< iostream.h >
#  include"hello.h"
main( ){
   cout< <  hello("Hello, Shanghai");
}
```

在本例子中,头文件 hello.h 是 hello 函数的原型。源文件 hello.cpp 定义了 hello 函数,该函数有一个形式参数,其类型为 string,返回函数的类型为 string。main.cpp 是打印“Hello, Shanghai”的主程序,它构造并打印了一个欢迎词的字符串。程序中的 sprintf( )

是系统内定义的打印函数。main.cpp 中调用的 hello 函数,其参数的类型、个数以及函数的返回类型必须与预处理指令“include”定向的头文件“hello.h”所给出的原型中的函数的参数类型、个数和函数的返回类型相匹配。关于函数,下面将专门讨论。

将函数和数据定义在互相独立的源文件中,可支持模块化的程序设计。在各个源文件使用后缀为“.h”的头文件,定义了对其他各个模块的调用接口。

在上面的三个文件中,每个文件的开头是注释行,hello.cpp 文件的三个 include 指令的后面给出的也是注释行。

在 C++ 中有两种注释方法:

(1) 多行注释:包含在定界符“/*”和“*/”之间的所有文本内容都是注释,这与 C 中的注释格式相同。例如:

```
/  This book is designed to present
the fundamentals of data structures
from an object-oriented perspective.  /
```

(2) 单行注释:这是 C++ 所独有的注释格式。在符号“//”之后至本行末的所有文本内容均为注释。

例如,C 注释

```
/* This is a C++ program.  /
```

可写为 C++ 的单行注释

```
//This is a C++ program.
```

因为 C++ 是 C 的超集,所以上述两种注释方法都可用在 C++ 程序中。

1.3.2 数据声明和作用域

数据声明的作用是将数据类型与数据名联系在一起。C++ 的基本数据类型与 C 一样,有 char、int、float 和 double。这些数据类型中的某些又可以用 short、long、signed 和 unsigned 进行修饰。修饰词 short 和 long 用于说明分配给它们修饰的基本类型的存储位数。修饰词 signed 和 unsigned 说明了一个整型数的二进制表示法中的最高有效位是否为符号位。

在数据声明时,主要形式如下:

(1) 常数值:内容不变的文字,如 57、3.1416 和“Good morning”。

(2) 变量:数据类型的实例,其内容在程序中可被修改。

(3) 常量:在其生命期中不可被赋值的变量。其内容在声明时给定,一旦给定,则在声明它的程序运行时不可被修改。说明时,在变量类型名前加上 const,如 const int pi=3.1415926。

(4) 枚举:声明一个整型常数序列的方式。它是用关键字 enum 声明的,它可以用来建立新的数据类型。

例如声明:

```
enum month= {Jan= 1, Feb, Mar, Apr, May, Jun, July, Aug, Sep, Oct, Nov, Dec}
```

将一年中的月份定义为一个枚举类型 month。该整数序列可以从任意值开始,只需将该值赋给第一个元素,后面的每一个值依次加 1。本例中,一月(Jan)的值为 1,1~12 月份对应

的顺序为 1,2,…,12 。枚举值在默认情况下是从 0 开始的。

(5)引用:这是 C+ + 的一个新功能。引用类型用于为一个对象提供一个可替换的名字。对于某一个类型的对象的引用,所采用的声明方式就是在该类型后面添上一个符号“&”。例如:

```
intx= 9;
int&y= x;
x= 13
printf("x= %d,y= %d",x,y,);
```

此时,y 是一个引用类型,它代表 x 的一个替换名。当 x 的值改变时,y 的值也随之被改变。本例中,当 printf 语句执行后,打印出的 x 和 y 的值都是 13。

还有几个类型,会在后面的章节中介绍。

在 C 中,程序块的所有声明都必须出现在所有可执行语句之前;在 C+ + 中,声明可放在使用所声明的内容之前的任何地方。例如:

```
printf("Enter two integers:");
intx,y;
printf("x= %d, y= %d", x, y,)
```

声明了变量 x 和 y,声明语句出现在可执行的语句的第一个 printf 语句之后。变量也可以在 for 结构的初始化部分予以声明,其作用域仍然是在定义 for 结构的程序块内。例如:

```
for( inti= 0; i< = 5; i+ + )
printf("i= %d", i)
```

在 for 结构中,把变量 i 声明为一个整数并把它初始化为 0。

在 C+ + 中,每一个变量都有一个作用域,函数中声明的变量只能在函数内部使用;在类中定义的变量,只能在该类内部使用。这些变量都称为局部变量。

C+ + 的局部变量的作用域从其声明开始到结束程序块的右花括号终止。因此,变量声明之前的语句即使在同一个程序块内也不能引用该变量。变量声明不能放在 while、do/while、for 和 if 结构的条件中。

把变量声明放在靠近首次引用的位置,即用到时再声明后写上使用,可提高程序的可读性。

在整个程序中都能引用的变量叫全局变量。如果一个全局变量在文件 1 中声明,而在文件 2 中使用,则在文件 2 中必须使用关键字 extern 对该变量进行声明。

在构成一个程序的两个文件中,如果分别声明了具有相同名字的一个全局变量,它们分别代表不同的实体,此时就要在两个文件中分别使用关键字 static 对变量进行声明,以保证不发生混淆。

如果一个程序块中某个局部变量与某个全局变量同名,但又要在该程序块中引用该全局变量,则可以使用域操作符“: : ”来引用全局变量。

1.3.3 输入/输出

在 C+ + 中要执行输入输出操作,必须用 # include 预处理指令包括一个 < iostream.h> 头文件。

关键字 cin 用于 C+ + 中的输入,操作符“≫”用于分开输入的变量。空白(即 Tab 键、

回车或空格键)用于在标准输入设备上将不同变量的值分开。

关键字 cout 用于输出到一个标准输出设备。cout 和将被输出的每一个内容之间用操作符“≪”分开。要输出的内容从左向右被输出到标准输出设备上。

此外,定向到错误文件的命令由 cerr 定义。

【例 1-3】 程序:C++ 中的输入输出。

```
#  include< iostream.h>
void main( ) {
   int x,y;
   cin≫x ≫y;
   cout≪"x= "≪ x ≪ endl;
   cout≪"y= "≪ y ≪ endl;
}
```

在输出语句的最后出现的“endl”也是 C++ 的输入输出操作符,其作用是输出一个换行符并清空流。与此类似的操作符还有 ends(输出空)、flush(清空流)和 WS(跳过前导空白)等。

执行上述程序时,按照输入格式

5　　10　　〈回车〉

或

5　　〈回车〉

10　　〈回车〉

均使变量 x 和 y 分别得到输入值 5 和 10,并输出如下结果:

x= 5

y= 10

C++ 中的输入输出,在功能上的一个优点是“自由格式”。程序员不必使用格式化符号来指明输入输出项的类型和顺序。此外,与 C++ 中其他的操作符一样,输入输出操作符能够被重载。

在 C++ 的程序中如有对文件的输入输出,则必须在程序中包含头文件 fstream.h,它定义了类 ifstream、ofstream 和 fstream。关于类,我们将在第 2 章详细讨论。要创建一个输入流,必须声明它为类 ifstream;要创建一个输出流,必须声明它为类 ofstream;而执行输入输出的流必须声明为类 fstream。

【例 1-4】 含有文件输入输出的程序。

```
#  include< iostream.h>
#  include< fstream.h>
void main ( ) {
  ofstream outFile("my.out", ios: : out);
   if(! outFile){
     cerr≪ "can not open my .out"≪ end;
        //standard error device return;
}
  int n= 70; floatf= 30.2;
```

```
    outFile ≪ "n: "≪n≪ endl;
    outFile ≪ "f: "≪ f ≪ endl;
}
```

在本例中,定义了一个输出文件流变量 outFile,它由两个参数来初始化,第一个参数是字符串(此处为 my.out),它代表文件名;第二个参数用于指明该文件被使用的模式,此处 ios: : out 表明该文件用作输出。如果文件未被打开,则 outFile= 0;如果文件被成功地打开,则它将替代 cout,用于将输出引导到文件 my.out 中。

1.3.4 函数

C++ 中有两种函数,即常规函数和成员函数。成员函数用于类方法的定义,第 2 章中将作详细介绍。常规函数用于完成一个特定的功能。无论是常规函数还是成员函数,其定义都包括四个部分:函数名、形式参数表、返回类型和函数体。函数的使用者通过函数名来调用函数,调用过程把实际参数传送给相应的形式参数作为数据的输入;然后通过执行函数体中的语句实现该函数的功能;最终得到的返回值由函数名带回给函数的调用者。

函数如果有返回值,则该值的类型就是该函数的返回类型。函数的返回值是通过函数体中的 return 语句返回的。return 语句的作用是返回一个其类型与返回类型相同的值,并终止函数的执行。

【例 1-5】 一个函数。

```
intmin ( inta, intb){
   ifa < b returna;
   else returnb;
}
```

在该例中,min 是函数名,int a 和 int b 是形式参数表,函数名 min 前面的 int 是返回类型,由花括号括起来的部分是函数体。

对于无返回值的函数,其返回类型要声明为 void。

在 C++ 中,指定空参数列表的方法是在圆括号中写入 void,或什么也不写。声明语句

```
void emp( );
```

指定函数不接收任何参数,也没有返回值。下面的程序例子演示了声明和使用不带参数的函数的方法。

【例 1-6】 使用不带参数的函数。

```
#  include< iostream.h>
voidf1( );
voidf2 (void);
main( ) {
   f1( );
   f2( );
    return0;
}
voidf1 ( ) {
   cout≪"Function f1 takes no arguments \n";
```

```
}
void f2(void){
  cout<<"Function f2 also takes no arguments\n";
}
```

输出结果为：

Function f1 takes no arguments

Function f2 also takes no arguments

1.3.5 参数传递

函数调用时传送给形参表的实参与形参在类型、个数以及顺序上要保持一致。C++中函数的参数传递有四种方式：值参数传递、引用参数传递、常值参数传递和常值引用参数传递。常值参数传递方式因为没有必要，一般不用。

值参数传递是缺省的参数传递方式。若采用这种传递方式，程序在运行时，对应的实际参数的值传送给形式参数所对应的局部工作区中的单元。当函数的执行终止时，函数修改的是形式参数所对应的工作单元的值，而该值不传回给实际参数，因此值参数的传递方式不会改变对应形式参数的实际参数的值。

使用引用参数传递方式时，需要在函数的形参表中将形参声明为引用类型，即在参数名前加上一个“&”。当一个实参与一个引用类型形参结合时，被传递的不是实参的值，而是实参的地址，函数通过地址存取被引用的实参。当函数执行时，任何对形式参数的改变也就是对实际参数的改变。

当要求一个函数调用返回多于一个参数时，也应采用引用参数传递方式。此时，将参数中的一个由 return 语句返回，而其他参数由引用返回。

常值引用参数传递方式的格式为

```
constT &a
```

其中 T 是参数的类型。在函数体中，不能对常值参数修改，否则将导致编译出错。

【例 1-7】 参数传递的方式。

```
#  include< iostream.h>
intExampleByValue(inta, intb, intc){
  intx,y,z;
  x= a; y= b; z= c;
  a= 3 a ;b= 3 b ;c= 3 c;
  return(x+ y+ z)/3;
}
intExampleByRefer(int& a, int& b, int& c){
代码同上
}
intExampleByConsRefer(const int& a, const int& b, const int& c){
  return(a+ b+ c)/3;
}
main(){
  ints= 0, p= 2, q= 3, r= 4; s= ExampleByValue(p,q,r);
```

```
    cout≪"p,q,r,and s: "≪ p≪q≪r≪s≪"\n";
    s= ExampleByRefer(p,q,r);
    cout≪"p,q,r and S: "≪p≪q≪r≪s≪"\n";
    s= ExampleByConsRefer(p,q,r);
    cout≪"p,q,r, and s: "≪p≪q≪r≪s≪"\n";
}
```

输出结果：

```
p,q,r and s:    2 3 4  3
p,q,r and s:    6 9 12 3
p,q,r and s:    6 9 12 9
```

因为函数 ExampleByValue 是以值参数传递方式调用的，在函数体的执行中，实际参数的值不修改，所以第一次输出的 p,q,r 的值为 2、3、4。而 ExampleByRefer 是以引用参数传递方式调用的，p,q,r 的值在函数体的执行中，修改为 6、9、12，所以输出的结果 p,q,r,s 为 6,9,12,3。调用 ExampleByConsRefer 时，实际参数 p,g,r 不会改变，保留字 const 保证只能对其值初始化一次，因而输出结果为 6,9,12,9。

1.3.6 函数名重载

在 C 中，不允许在同一个程序中声明两个同名的函数；而在 C++ 中，允许在同一个程序中用同一个名字定义多个函数，但它们的形参表不同。这种能力称为函数名重载。在调用一个重载的函数时，编译程序通过检查参数的个数、类型和顺序自动选择一个合适的函数。

【例 1-8】 用重载函数 sum 来计算两个 int 类型值的和及三个 float 类型值的和。

```
# include< stdio.h>
intsum( inta, intb) {
   returna+ b;
}
floatsum( floata, floatb, floatc){
   returna+ b+ c;
}

voidmain( ) {
  printf(" sum(2,3)= %d",sum(2,3));
  printf("sum(1.1,2.2,3.3)= %f", sum(1.1,2.2,3.3));
}
```

在例 1-8 中，两个 sum 函数的返回类型不同，参数个数也不同。

函数重载通常用来建立在不同数据类型的基础上完成类似任务的多个同名函数，这可使程序易于阅读和理解。

1.3.7 动态内存分配

在 ANSI C 中，动态内存分配通常是用标准库函数 malloc()和 free()来实现的。C++ 兼容了 C 语言中的这两个函数，并提供了两个新的操作符 new 和 delete，使程序实现动态

内存分配。考虑如下的声明语句:

```
ptrtype *ptr
```

语句中的 ptrtype 可以是任何数据类型(如 int、char、float 等),则语句

```
ptr= new ptrtype
```

从程序的空闲内存区中为 ptrtype 类型的对象分配内存。new 运算符以类型为参数,自动建立一个具有合适大小的对象,并返回指向该类型对象的指针,此处类型为 ptrtype,返回的指针为 ptr。如果分配内存不成功,则返回一个空指针,在 C++ 中,以 0 而不是 null 来表示空指针。

在 C++ 中,用如下语句来释放该对象所占用的空间:

```
deleteptr;
```

运算符 delete 只能释放用运算符 new 分配的内存。把 delete 用于空指针对程序执行没有任何影响,但把 delete 用于已经释放的指针是不允许的。

C++ 允许初始化新分配的对象,例如,语句

```
int* thisptr= new int(57);
```

建立了一个指向 int 类型对象的指针 thisptr,把新分配的 int 类型的对象初始化为 57。该语句把指针 thisptr 的声明与动态内存分配以及初始化放在一条语句中。

虽然 new 和 delete 所完成的功能与 C 语言中的 malloc()和 free()类似,但更方便。首先,new 按指定类型自动分配足够的空间,不要求调用者提供所需存储空间的数量并使用 sizeof 运算符进行计算;其次,new 自动返回指定类型的指针,不必像 malloc()那样在分配时需要显式地使用强制类型;此外,new 和 delete 可以重载。

1.3.8 结构与联合

到目前为止,我们所见到的数据类型都只包含一种类型信息,即使是由多个数据元素组成的数组,亦是如此。如果想要把多个不同数据类型的数据项组合成一个数据元素,则可以使用结构这样的数据类型。

1. 结构

C++ 用数组存储许多相同类型的相关信息,但是有些数据信息是由若干不同数据类型的数据所组成。例如,一个职工工资记录包括姓名、工号和工资等,这些数据信息的类型是不一样的,不能用数组直接把它们组织起来。用结构变量就可以有组织地把这些不同数据类型的数据信息存放在一起。

结构是用户自定义数据类型,它可与 int、float 等基本数据类型一样被使用。声明结构类型时,先指定关键字 struct 和结构名,然后用一对花括号将若干个结构成员及其数据类型的说明括起来。

通常,结构声明在所有函数之外,位于 main 函数之前。这就使所声明的数据类型在程序的任何地方都可以被使用。

声明一个结构并不分配内存,内存分配发生在定义这个新数据类型的变量时。

结构中包含的数据变量称为该结构的成员。如例 1-9 中的 id、salary 是结构 Person 的成员。

一旦通过定义相应结构变量,分配了空间,就可以使用点操作符“.”(或称结构成员操作符)来访问结构中的成员。左操作元为结构类型变量,右操作元为结构中的成员。

在数组中,我们称数组分量为元素,在结构中,我们称结构分量为成员。数组的“[]”运算符与结构的点运算符具有相同的运算优先级,它们是所有运算符中优先级最高的。

因为我们用结构来实现的目的,就是要区分一个数据聚集中的每个分量的类型和意义。每个分量数据类型可以相异。更重要的是,为了区分数据内容的意义,结构的成员有自己单独的名字。

结构可以被赋值。

【例 1-9】 声明一个关于职工工资记录的结构,并使用它。

```
//  Person-salary. cpp
#   include< iostream.h>
structperson
{
char   name[20];    // 姓名
unsigned longid;    //工号
int   salary;       //工资
};

voidmain( )
{
person pr1= {"Frank Voltaire",12345678, 2456};
person pr2;
pr2= pr1;
cout≪ pr2.name ≪" "
    ≪pr2.id ≪""
    ≪pr2.salary≪ end;
}
```

运行结果为:

Frank Voltaire 12345678 2456

程序中定义了一个全局 Person 结构变量 prl,它使用与初始化数组相似的方法进行初始化。在 main()函数中,定义了一个结构变量 pr2,然后使用赋值运算符将 pr1 的内容赋值给 pr2。

在结构 Person 中,成员 name 是一个字符数组,通过结构变量的赋值,该数组作为成员也被赋值了。

两个不同结构名的变量是不允许相互赋值的,即使两者包含有同样的成员。

根据结构类型可以定义一个变量,是变量就有地址。结构不像数组,结构变量不是指针。通过取地址“&”操作,可以得到结构变量的地址,这个地址就是结构的第一个成员地址。

可以将结构变量的地址赋给结构指针,结构指针通过箭头操作符“->”(也是一种结构成员操作符)来访问结构成员。

【例 1-10】 定义结构指针,通过结构指针来访问结构成员。

```
//Structure-pointer.cpp
#  include< iostream.h >
#  include< string.h >
structperson
{
     charname [20];
     unsigned long  id;
     intsalary;
};
voidmain( )
{
    person pr1;
    person  prPtr;
    prPtr= & pr1;     //定义了一个结构类型的指针
    strcpy(prPtr-> name,"David  Marat");//字符串拷贝赋值,函数 strcpy 在第 5 章中介绍
    prPtr-> id= 987654321;
    prPtr-> salary= 2567;
     cout≪prPtr-> name≪"     "
         ≪prPtr-> id ≪ "     "
         ≪prPtr-> salary ≪ endl;
}
```

运行结果为:

```
Davit  Marat     987654321  2567
```

使用箭头操作符就是对结构成员进行操作。但必须清楚,当用点操作符时,它的左边应是一个结构变量,当用箭头操作符时,它的左边应是一个结构指针。

例 1-10 中把 pr1 的地址赋给 Person 结构指针 prPtr,然后通过这个指向 prl 的指针对 prl 进行赋初值和输出。这一步很重要,如果不把 prl 的地址赋给 prPtr,那么 prPtr 是个随机地址,在这个地址上赋值是危险的。

指针是有类型的,引用一个整型指针得到一个整数,引用一个结构指针得到一个结构。即*prPtr 的值就是结构 person 的变量 prl 的值,而不会是其他类型的值。

结构是一个数据类型,所以也可以拥有结构数组。要定义结构数组,必须先声明一个结构,然后定义这个结构类型的数组。

【例 1-11】 定义一个由 100 个元素组成的 person 结构类型数组。

```
struct  person
{
   charname [20];
   unsigned longid;
   intsalary;
};
person  allone [100];     // 定义一个 person 类型的数组
```

结构数组中,每个元素都是结构变量,访问结构数组元素中的成员,方法与前面介绍的方法类似。

2. 联合

联合(union)是一种变量,它可以在不同时间内维持不同类型和不同长度的对象。它提供了在单个存储区域中操作不同类型数据的方法,而无需在程序中存放与机器有关的信息。联合的语法是以结构为基础的。

【例 1-12】 定义一个联合。

```
unionutag {
    intival;
    floatfval;
    char  pval;
} uval;
```

联合元素的引用,语法上也类似于结构成分的引用:

union-name.member

或

union-pointer-> member

如果变量 utype 是用来记录存储在 uval 中的最近类型的,那么可使用下列程序段存取联合中的元素。

```
if(utype= = INT)
  printf("%d\n", uval.ival);
else if(utype= = FLOAT)
  printf("%d\n", uval.fval);
else if(utype= = STRING)
  printf("%d\n", uval.pval);
else
  printf("bad type %d in utype\n", utype);
```

联合可以和结构、数组组合使用。存取结构中的联合或联合中的结构的记号与存取嵌套结构是一样的。

【例 1-13】 结构、数组和联合的组合。

```
struct{
        char      name;
        int       flags;
        int       utype;
        union{
                intival;
                floatfval;
                char  pval;
        } uval;
} symtab[NSYM];
```

上述 symtab 的定义是一个结构数组,可以用 symtab[i].uval.ival 来引用结构数组中第 i 个结构的成分——联合中的元素(最近存储的是整数类型),如果联合中最近存储的是字符串,则可用 symtab[i].uval.pval 来引用 pval 中的第一个字符。

事实上,联合是一种形式特殊的结构变量。和结构一样,对联合施加的操作只能是存

取成员和取其地址。不能把联合作为参数传递给函数,也不能由函数返回联合。

联合只是一种变量,为了弄清在联合中存储的是哪一种类型,通常是在联合外设置一个变量以作表征。正如在 systab 中,每一个结构,都含有整型变量 utype 以指出在该结构的联合 uval 中存储的是什么类型的变量。结构中往往含有几个不同类型的变量,如 systab 中就有四个变量。

1.4 算法性能与复杂度

公元 825 年,一位名叫 al-Khowarizmi 的波斯数学家写了一本教科书,书中概括了进行数字四则算术运算的法则,所有的数字都是用今天的印度十进制形式来表示的(按个、十、百位等排列,并有表示小数部位的小数点)。现代名词"算法"(algorithm)就来源于这位数学家的名字。美国 Webster's 字典中将算法定义为:解某种问题的任何专门的方法。在计算机科学里,算法这个词有一个专门的解释:算法——用计算机解题的精确描述。

本书在讨论各类数据结构的同时,介绍了对数据结构进行操作的算法,因此将对算法的概念作比较充分的讨论。

1.4.1 算法的定义

通常,人们将算法定义为一个用于实现某个特定任务的有穷指令集,这些指令规定了一个运算序列。一个算法应当具有以下特性:

(1) 输入性:一个算法必须具有零个或多个输入量。

(2) 输出性:一个算法应有一个或多个输出量,输出量是算法计算的结果。

(3) 确定性:算法中的每一条指令应含义明确、无歧义,即对每一种情况、需要执行的动作都应严格地、清晰地规定。

(4) 有穷性:算法中的指令执行序列是有穷的,即算法无论在什么情况下都应在执行有穷步后终止。

(5) 有效性:每条指令必须是足够基本的。也就是说,它们原则上都能精确到可用笔和纸来模拟实现其指令过程。对于每一步操作,仅有(3)中的确定性是不够的,它还必须是可行的。

需要指出的是,一个程序与一个算法对于上述(4)是有重大区别的。一个程序可以不满足特性(4),例如,以用户名和口令来进行登录的系统就是一个处于"键入用户名、键入口令,检查是否已在系统中注册过并且匹配,如不满足,则重新输入"的循环之中,直至键入的用户名和口令匹配,这样的一个程序可能会不终止。但本书中所给出的程序均是可终止的,所以在本书中对"算法"和"程序"这两个术语不作严格的区分。

算法设计者在构思和设计了一个算法之后,必须准确清楚地将所设计的解题步骤记录下来,或提供交流,或编写成程序供计算机执行。

记录算法中的解题步骤又叫描述算法。常用的描述算法的方式有自然语言、流程图和程序设计语言等。

用汉语或英语这样的自然语言来描述算法,最大的优点是使用者不必对描述工具本身花精力去学习,对写出来的算法的理解是直接的。但其缺点是:容易出现二义性;语句一

般太长,这就使所建立的算法也显得冗长;算法中的分支及循环等结构表示不能清晰地显示出来。

使用规定式样的图形、指向线和文字说明组合起来的流程图方式来描述算法,其优点是直观、清晰、易懂,便于检查、修改和交流,其缺陷是严密性不如程序设计语言,灵活性不及自然语言;此外,对于大型的算法描述有困难。

用计算机程序设计语言描述算法显得清晰、明了,写出的算法一步到位,能由计算机处理。事实上,用程序设计语言来描述算法就是对算法的实现。其缺点是抽象性差一些,可能会使写算法的人拘泥于计算步骤描述的细节,而忽略算法的实质。此外,必须熟练掌握程序设计语言及其编程技巧。

考虑到本书的使用者已经熟悉了像 C 这样的程序设计语言,并且掌握了程序设计的基本方法和技术,并因本书的内容是以面向对象的方法来讨论数据结构的,因而采用 C++ 来描述算法。它的优点是类型丰富、语句精练,具有面向过程和面向对象的双重特点。此外,C++ 也支持抽象数据类型。为了使写出的算法可读性和可理解性更强,本书有时还采用了 C++ 语句与自然语言结合的方式来描述算法,有时对算法加以合适的注释。

1.4.2 算法的性能标准

在面向对象的程序中,通过类来实现数据结构,并且还要实现类中各个服务的算法。一旦确定了类的数据结构,就必须描述这些算法的设计和实现。

算法的设计主要有以下几个标准:

(1) 正确性:算法应确切地满足所要求解的问题的需求,这是最重要也是最基本的标准。

(2) 可用性:算法应能很方便地使用。为了便于用户使用,要求算法具有良好的界面和完备的用户文档。

(3) 可读性:算法应当是可读的,即易于理解的。一个算法不仅要被设计者自己读,而且也可能被他人读,不仅是编程、调试和测试时被读,而且在维护时也被读,因此,可读性是很重要的标准。

(4) 效率:算法的效率主要是指算法执行时存储单元的开销和运行时间的耗费,前者称为算法的空间代价,后者称为算法的时间代价。

(5) 健壮性:当输入非法数据时,算法应能作出适当的处理,而不应当产生不可预料的结果。这就要求算法中有对输入参数、打开文件、读文件记录以及子程序调用状态等操作的检错、报错和纠错等功能。

在设计一个算法时,上述的几条标准有时会有矛盾,如可用性强、可读性强会降低算法的效率。而对效率这个标准,算法的低时间代价和低空间代价也会产生矛盾。例如,有些问题若采用较多的内存空间可使时间代价降低,若采用较少的内存空间,则使时间代价提高。在计算机硬件价格快速下降的趋势下,算法的时间效率应首先予以考虑。

1.4.3 算法的复杂度

算法效率的度量一般采用事前估计和后期测试两种方法。例如对于程序运行的时间耗费,后期测试主要通过在算法中的某些部位插装计时函数来测定算法完成某一规定功

能所需的时间。但这种方法与算法的运行环境有关。同样的算法在速度不同的计算机上运行,执行速度相差却非常大;此外,一个算法用不同的编译系统编译出的目标代码的长度不一样,质量也不一样,完成同样的功能所需时间也不同;还有,对于一个存储需求很大的算法,如果可用的存储空间不够,在运行时不得不频繁地进行内外存交换,需要的运行时间就很多。而如果可用的存储空间足够大,运行时间就可以大大减少。因此,算法的实际运行时间依赖于所用的计算机系统。在不同的机型、不同的编译系统版本、不同的软硬件配置情况下,想通过后期测试的方法来测定算法的复杂度是比较困难的。人们常常采用事前估计即对算法进行分析的方法来测定算法的复杂度。因为算法的复杂度与具体的运行环境和编译系统无关,所以可以通过复杂度的分析来对算法进行比较和评估。

1. 算法的时间复杂度

算法的时间复杂度与具体的机器以及运行环境无关,它与所求解的问题的规模有关,可以说,它是问题规模的函数。

一般来说,问题的规模可以从问题的描述中找到。例如,在一个具有 n 个教职工记录的文件中查找某个名叫李华的教师,则该问题的规模为 n。又如,对一个具有 n 个整数组成的数组进行排序,则问题的规模也是 n。

一个算法是由控制结构(顺序、分支和循环)和基本操作构成的,则算法的时间复杂度与这两者有关。为了比较解同一问题的不同算法,通常的做法是,从算法中选取一种对于所研究的问题来说是基本运算的操作,即基本操作,以该基本操作重复执行的次数作为算法的时间量度。有时也需要同时考虑几种基本操作,甚至可以对不同的操作赋以不同权值,以反映执行不同操作所需的相对时间,这种做法便于综合比较解决同一问题的那些完全不同的算法。

算法的时间效率分析通常采用 O(f(n))表示法,读作“大 O 的 f(n)”。其定义可叙述为 T(n)= O(f(n))当且仅当存在正常数 c 和 n_0,使得对所有的 n,当 $n \geqslant n_0$ 时,都满足 $T(n) \leqslant cf(n)$。换句话说,O(f(n))给出了函数 T(n)的上界。

T(n)= O(f(n))表示:随问题规模 n 的增大,算法执行时间的增长率和 f(n)的增长率相同,称作算法的渐近时间复杂度(asymptotic time complexity),简称时间复杂度。或者说,两者具有相同的数量级。

【例 1-14】 设 a 和 b 是两个已经赋了值的数组,对如下求解了两个 N× N 矩阵相乘的算法求其时间复杂度。

```
for(i= 0; i< n; i+ + )
  for(j= 0; j< n; j+ + )
  { C[i,j]= 0;      //基本操作语句 1
for(k= 0; k< n; k+ + )
  C[i][j]= C[i][j]+ a[i][k] b[k][j];     // 基本操作语句 2
}
```

设基本操作语句的总执行次数为 T(n),因为(基本)操作语句“c[i][j]= 0”的执行次数为 n^2,“c[i][j]= C[i][j]+ a[i][k]*b[k][j]”的执行次数为 n^3,因此 $T(n)= n^2+ n^3$。而 $T(n)= n^2+ n^3 \leqslant cn^3$,选择 $c \geqslant 2$,则 $T(n) \leqslant cn^3 = O(n^3)$。所以该算法的时间复杂度为 $O(n^3)$。

当算法的时间复杂度T(n)和问题的规模n无关时,$T(n) \leqslant c^* 1$,此时算法的时间复杂度 $T(n)= O(1)$,称为常量级;当算法的时间复杂度T(n)与问题规模n为线性关系时,$T(n) \leqslant c^* n$,此时算法的时间复杂度 $T(n)= O(n)$,称为线性级;当算法的时间复杂度T(n)和问题的规模n为平方关系时,$T(n) \leqslant c^* n^2$,此时算法的时间复杂度 $T(n)= O(n^2)$,称为平方级。

【例 1-15】 求出下面三个程序段的时间复杂度。

(1)
```
x= x+ 1
```

(2)
```
for(i= 1; i< = n; i+ + )
    x= x+ 1
```

(3)
```
for(i= 1; i< = n; i+ + )
    for (j= 1; i< = n;j+ + )
        x= x+ 1
```

这三个程序段中均含基本语句x= x+ 1,各自所含的次数为1、n和n^2,所以这三个程序段所表示的算法的时间复杂度分别为O(1)、O(n)和$O(n^2)$。

依此类推,还有O(logn)和$O(2^n)$等时间复杂度。

【例 1-16】 设n为如下算法处理的数据个数,求出其时间复杂度。

```
for(i= 1; i< = n; i= 2  i)
printf("i= %d \n" , i)      // 基本语句
```

设基本语句的执行次数为T(n),有$2^{T(n)} \leqslant n$,即有$T(n) \leqslant \log n$,因$T(n) \leqslant \log n \leqslant c^* \log n= O(\log n)$,其中c为常数,所以该算法的时间复杂度为O(logn)。

由于算法的时间复杂度考虑的只是对于问题规模的增长率,则在难以精确计算基本操作语句执行次数的情况下,只需求出它关于n的增长率或数量级即可。

在许多情况下,算法的时间复杂度会随着算法中数据元素的取值情况的不同而不同。

例如,下面的算法是用冒泡排序法对数组a中的n个整数类型的数据元素(a[0]~a[n- 1])从小到大进行排序。

【例 1-17】 冒泡排序算法。

```
void BubbleSort( inta[ ], intn) {
  inti, j, flag= 1;
  inttemp;
  for(i= 1; i< n && flag= = 1; i+ + ) {
    flag= 0;
    for(j= 0; j< n- i;j+ + ){
      if(a[j]> a[j+ 1]){
        flag= 1,
        temp= a[j],
        a[j]= a[j+ 1];
        a[j+ 1]= temp;
      }
    }
  }
}
```

这个算法的时间复杂度随待排序数据的不同而不同。当某次排序过程中没有任何两个数组元素交换位置,则表明数组元素已排序完毕,此时算法将因标记 flag= 0 不满足循环条件而结束。

“交换两个相邻的整数”为基本操作,当 a 中的初始序列为“正序”,即自小至大有序时,基本操作的执行次数为 0,这是最好的情况;当初始序列为“逆序”,即自大至小有序时,基本操作的执行次数为 n(n-1)/2,这是最坏的情况。再从“两个相邻元素比较”这一基本操作来看,当初始序列为“正序”时,则 i 循环体执行一次,在 j 循环中进行了 n- 1 次关键字之间的比较。反之,若初始序列为“逆序”时,则 i 循环体执行 n- 1 次,需要进行的关键字之间的比较为 $\sum_{i=1}^{n-1}(i-1)=n(n-1)$ 次。对这类算法的分析,一种解决的方法是计算它的平均值,即考虑它对所有可能的输入数据集的期望值,此时相应的时间复杂度为算法的平均时间复杂度。如假设 a 中初始输入数据可能出现 n! 种排列情况的概率相等,则冒泡排序算法的平均时间复杂度为 $Ta(n)=O(n^2)$。然而,在许多情况下,各种输入数据集出现的概率难以确定,算法的平均时间复杂度也就难以确定。因此,另一种更可行也更常用的办法是讨论算法在最坏情况下的时间复杂度,即分析在最坏情况下,估算出算法执行时间的一个上界。例如,上述冒泡排序的最坏情况为 a 中初始序列为自大至小有序,则冒泡排序算法在最坏情况下的时间复杂度为 $O(n^2)$。

算法的时间复杂度是衡量一个算法优劣的重要标准。一般来说,具有多项式时间复杂度的算法是可接受、可实际使用的算法。而具有指数时间复杂度的算法,只有当 n 足够小时才是可使用的算法。表 1-2 给出了多项式增长和指数增长的比较。从表中可看出,当 n= 50时,多项式函数 $n^3=125000$,而指数函数 $2^n=1.0\times10^{15}$, $n!=3.0\times10^{64}$, $n^n=8.9\times10^{84}$。我们解题时,应尽可能选用多项式级 $O(n^k)$的算法,而尽量避免使用指数级的算法。

表 1-2　多项式增长和指数增长的比较

大　小	多项式			指　数		
n	n	n	n	2	n!	n
1	1	1	1	2	1	1
2	2	4	8	4	2	4
3	3	9	27	8	6	27
4	4	16	64	16	24	256
5	5	25	125	32	120	3 125
6	6	36	216	64	720	46 656
7	7	49	343	128	5 040	823 543
8	8	64	512	256	40 320	16 777 216
9	9	81	729	512	362 800	3.9E8
10	10	100	1 000	1 024	3 628 800	1.0E10
…	…	…	…	…	…	…
20	20	400	8 000	1 048 376	2.4E18	1.0E25
30	30	900	27 000	1.0E9	2.7E32	2.1E44
40	40	1 600	64 000	1.0E12	8.2E47	1.2E64
50	50	2 500	125 000	1.0E15	3.0E64	8.9E84
…	…	…	…	…	…	…
100	100	10 000	1.0E6	1.3E30	9.3E157	1.0E200